Universidad de Granada

Proyecto de Fin de Carrera
Ingeniería en Informática

GNU Psychosynth

A framework for modular, interactive and collaborative sound synthesis and live music performance

Autor
Juan Pedro Bolívar Puente
Director
Joaquín Fernández-Valdivia

Departamento de Ciencias de la Computación e Inteligencia Artificial

Granada, Junio de 2011

PSYCHONTH

GNU Psychosynth

A framework for modular, interactive and collaborative sound synthesis and live music performance

Autor
Juan Pedro Bolívar Puente

Director
Joaquín Fernández-Valdivia

GNU Psychosynth: A framework for modular, interactive and collaborative sound synthesis and live music performance

Juan Pedro Bolívar Puente

Palabras clave:

Resumen

Escribir el abstract en español.

GNU Psychosynth: A framework for modular, interactive and collaborative sound synthesis and live music performance

Juan Pedro Bolívar Puente

Keywords:

Abstract

Escribir el abstract en inglés.

Yo, **Juan Pedro Bolívar Puente**, alumno de la titulación Ingeniería en Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 48941569F, autorizo la ubicación de la siguiente copia de mi Proyecto Fin de Carrera en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Juan Pedro Bolívar Puente

Granada a 1 de Junio de 2011.

D. **Joaquín Fernández-Valdivia**, Catedrático del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

Informan:

Que el presente proyecto, titulado ***GNU Psychosynth: A framework for modular, interactive and collaborative sound synthesis and live music performance*** , ha sido realizado bajo su supervisión por **Juan Pedro Bolívar Puente**, y autorizamos la defensa de dicho proyecto ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 1 de Junio de 2010.

El director:

D. Joaquín Fernández-Valdivia

AGRADECIMIENTOS

Añadir agradecimientos.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

TODO LIST

PREFACE — A PERSONAL, HISTORICAL AND AUDIOPHILE DISSERTATION

I am going to let the formalities, both in form and content, of a final project aside in this section to give an initial background on the historical development of the conception of GNU Psychosynth. After all, the story of this project is, in many ways, the story of my own learning and maturing process and specially the evolution of my interest in music.

In 2006 I was a young computer science student who had just moved to Granada, a city full of youth, joy and cultural activities. At that time, I was not keen at all in electronic music —maybe biased by my prejudices on the rave subculture that surrounds a wide part of it, even though eventually I happened to appreciate it in some way. At that time I was more of a punk and ska fan and rejected the artificiality and production process of electronic music; this was actually a contradiction with my interest in programming. However, I eventually got specially interested in the wild 70's explosion of musical experimentation, and concretely in progressive rock.

I can vividly remember my first positive contact with electronic music. It was a chilled and psychedelic evening at a friend's place —one of those old and rundown but magical flat, with rather high ceilings and wooden windows, where many students live in Granada— when we Youtubed a video where Keith Emerson virtuously performed "Knife Edge" on a keyboard attached to a big wooden box full of

wires and knobs. By rearranging the wires or playing with the knobs, he would radically change the sound that his keyboard emitted. That magical sound machine was a Moog modular synthesiser, and that was the birth of a true love for electronic music that would later conceive Psychosynth —is not love, they say, the true means for conception?

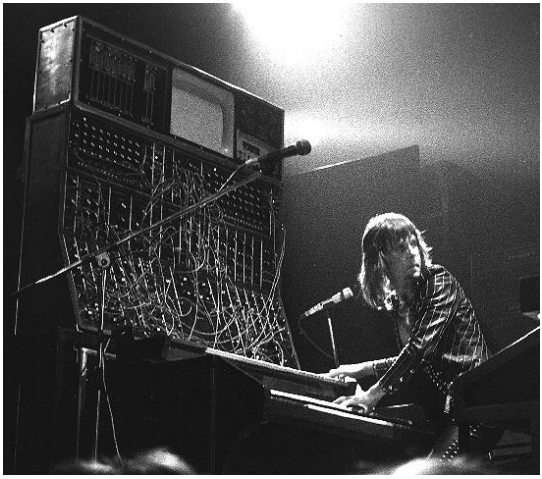


Figure 0.1.: Keith Emerson playing a modular Moog. The keyboard is connected to a big rack of modules, interconnected with eachother with wires. Each module generates or processes the sound carried in analog form in the interconnecting wires, therefore having an exponential number of possibly different sounds by combining modules and settings.

After that, I started to listen to more and more music made with electronic devices — an exploration that happened, actually, following electronic music's own history. From the synthesiser-full rock of Soft Machine, King Crimson or The

Nice I opened my ears to the purely synthetic orchestral compositions of Wendy Carlos, Vangelis or the early Jean Michelle Jarre. Kraftwerk's own evolution from rock to drum-pads, vocoders and synthesisers allowed me to open my ears to more modern electronic music.

It was still my first year of university when I watched a video, under circumstances probably similar to that before, of a new device being developed in the University Pompeu Fabra, in Barcelona: the ReacTable [1]. In a dark room only illuminated by a bright blue table, several performers placed tagged objects on the table. The table automatically arranged a connection graph among the objects based on their nature and relative distance and it displayed it but, more interestingly, the sound was evolving as this graph did. I just thought: wow, that must be fun, I want to play with that! — well, Do It Yourself.

That was the birth of Psychosynth. At the beginning it was just a toy. I did know nothing on how to process real-time audio, so I wrote many experiments. When I was bored of testing the dynamic range of my speakers and ears with all sorts of sinusoids, triangle and square signals I started to read more and more source code of other audio Free Software and eventually started to write a digital modular synthesizer and play with Ogre3D. [1] By the summer of 2007 I had some very primitive synthesis code and also some user interface experiments[2]. While I was an average C developer when I started my studies, I did not have any clue on C++. During the development of these initial experiments, I also

1 Torus Knot Software Ltd. `http://www.ogre3d.com`

2 As this video can show, `http://blip.tv/file/325103`, the software was just a bunch of buttons with a 3D blue table like Reactable's and no sound.

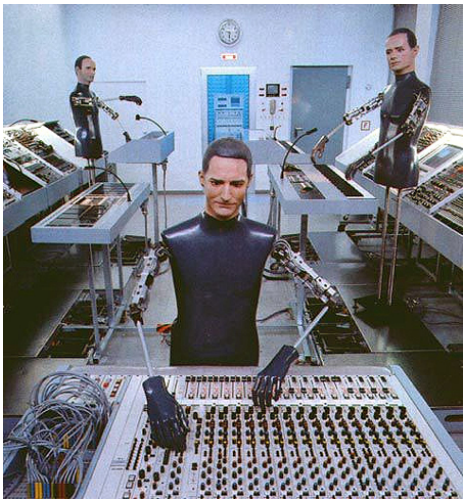


Figure 0.2.: Artwork for the music of Kraftwerk, with robotic representations of the band members playing with electronic instruments. On the front we can see an old analog sequencer.[a]

a As opposed to a synthesizer —this is, a virtual instrument, which is in charge of producing sound of whatever note you tell it to— a *sequencer* is an electronic score that can not produce sound by itself. The notes are programmed in the device —in an analog sequencer, by toggling switches and knobs— and it sends them at the appropriate time and duration to the electronic instruments (synthresisers) it is connected to. While current digital sequencers are very powerful, at that time they had important limitations that influenced Kraftwerk's robotic but charm sound. Kraftwerk was one of the first pop bands to produce its music entirely with electronic devices and is considered the father of modern electro and techno and has heavily influenced many other styles like house and hip-hop.

had to learn about inheritance, what `virtual` means, etc. but of course the design was flawed all the time and I had to rewrite the code many times.

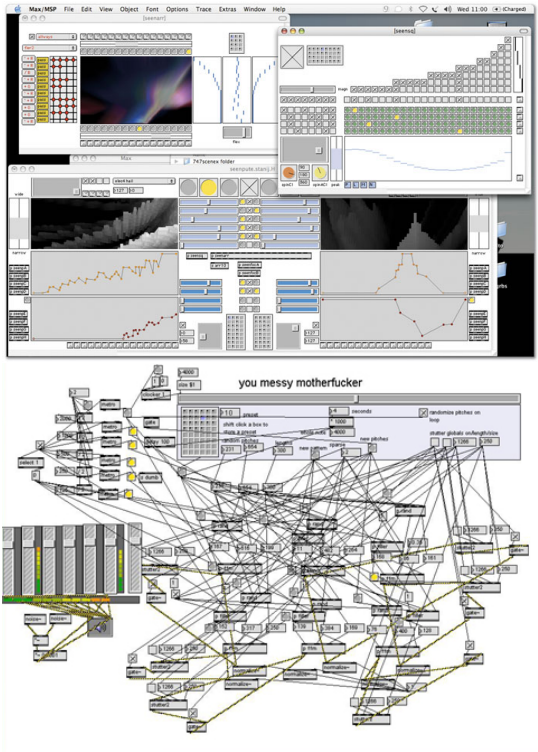


Figure 0.3.: The first picture is a Max/MSP claimed to be made by the electronic music duo Autechre[2]. The second is a Max/MSP patch made by Robbie Martin that *generatively* reverse-engineers Autechre's song Powmod.[a]

a Music is said to be *generative* when the whole composition itself is performed by pseudo-random algorithms programmed to produce well-sounding evolving non-repeating patterns. Autechre are known to have explored generative music. Max/MSP and its Free Software counterpart Pure Data are graphical data-flow programming software, which can be considered a low-level modular synthesizer, that is often used to write arbitrarily complex musical software and is specially interesting for generative music.

Something happened then at the beginning of my second year of university: I applied as a contestant to the Spanish Free Software Contest for University Students, and Psychosynth was the project I would develop. At that time I was already interested in experimental electronic music from the 90's, and late programming nights were accompanied by Autechre's fractal glitchs and Aphex Twin and Squarepusher's spastic patterns[3]. At that time, this cruise in the most experimental side of electronic music and my ignorance in proper music making made me believe in scoreless generative music produced from evolving continuous signals, and that influenced the lack of proper synchronisation mechanisms in Psychosynth. At some point, Shakero8[4], a music producer and DJ from Malaga, developed some beat loops to distribute along with the software and helped in early showcase performances. He also tough me a lot on how music is traditionally made.

The project won a price in that Free Software contest and he got some reviews in blogs. it then became part of the GNU project — an attempt to assure its long-term development and that it would remain Free Software in the future.

After that, however, the development stalled a bit. I had big refactoring ideas that never got the motivation to be accomplished. In that seek for perfect code motivated by an increasing interest in programming language theory and functional and generic programming, I also became more

3 If there is something I am grateful for during the early development of Psychosynth is the patience of my flatmates during those noisy programming nights. Psychosynth was long-time nicknamed "the ambulance siren sound maker" because it was only able to produce recursively modulated sinusoids.

4 `http://soundcloud.com/shaker08`

and more conscious of the flaws of the code —i.e. the lack of synchronisation mechanisms, MIDI (Music Instrument Digital Interface) support, pluggable modules, patch persistence, etc. make any serious non-experimental attempt to make music with it very hard. During these last 2 years I have learnt much more on the music production workflow and terminology, a process parallel to a final step in opening my ears to current electronic music, specially Drum and Bass, Dubstep, and even some Minimal and Techno. During last summer I got a MIDI DJ controller that got me to better understand the limitations and possibilities of Psychosynth for music mixing and I became a casual contributor of the best Free Software DJ software: Mixxx [3]. Also, the people at ArtQuimia[5], the music production school where Shakero8 was educated, became interested in the project and has offered lending gear for testing and supervision, guidelines and insight from a musician point of view.

So here we are now, in the fall of 2010. Still quite ignorant in music making but pretending to be a "digital luthier" motivated by passion for music. And trying to turn all this personal game into a final master thesis project. Lets see how it goes...

5 http://www.artquimia.net/

1

INTRODUCTION, DEFINITION AND GOALS

> La utopía está en el horizonte. Camino dos pasos, ella se aleja dos pasos y el horizonte se corre diez pasos más allá. ¿Entonces para que sirve la utopía? Para eso, sirve para caminar.
>
> EDUARDO GALEANO

1.1 PROBLEM DEFINITION

Our problem definition is well expressed in the title of this project: "a framework for modular, interactive and collaborative sound synthesis and live music performance". Lets elaborate this by describing its parts.

1.1.1 *A modular synthesiser*

A modular synthesiser is one where the sound generation and manipulation process is described as a directional graph where each link represents the flow of sound signal from one node to another, and each node transforms or generates those signals.

A node with no input is often called a *generator*. A node with one input and one output is often called a *filter*. A node can have different *parameters* that in analog hardware can be set with knobs and potentiometers. Often, these parameter can be controlled with other optional input signals that are called *modulators*. A concrete interconnection of a set of modules is commonly referred as a *patch*. Note 1.1 lists some of the most common modules in analog modular synthesisers.

Note 1.1 (Standard modules in an analog modular synthresizer) *Software modular synthesizers have usually similar ones in their default module too, while they quite often use different terminology not related to this voltage based signal like in this case.*

[The following text is extracted from the Wikipedia article on "Modular Synthesizer," as checked on December 12th 2010.]

VCO *Voltage Controlled Oscillator, which will output a pitched sound (frequency) in a simple waveform (most usually a square wave or a sawtooth wave, but also includes pulse, triangle and sine waves).*

NOISE SOURCE *A generator that supplies "hiss" sound similar to static, which can be used for explosions, cymbals,*

or randomly generated control signals. Common types of noise offered by modular synthesizers include white, pink, and low frequency noise.

VCF *Voltage Controlled Filter, which attenuates frequencies below (high-pass), above (low-pass) or both below and above (band-pass) a certain frequency. VCFs can also be configured to provide band-exclusion, whereby the high and low frequencies remain while the middle frequencies are removed.*

VCA *Voltage Controlled Amplifier, which varies the amplitude of a signal in response to a supplied control voltage.*

EG *Triggering an Envelope Generator produces a single, repeatable shaped voltage pulse. Often configured as ADSR (Attack, Decay, Sustain, Release) it provides the means to shape a recognizable sound from a raw waveform. This technique can be used to synthesize the natural decay of a piano, or the sharp attack of a trumpet. It can be triggered by a keyboard or by another module in the system. Usually it drives the output of a VCA or VCF, but the patchable structure of the synthesizer makes it possible to use the envelope generator to modulate other parameters such as the pitch or pulse width of the VCO. Simpler EGs (AD or AR) or more complex (DADSR—Delay, Attack, Decay, Sustain, Release) are sometimes available.*

LFO *A Low Frequency Oscillator is similar to a VCO but it usually operates below 20 Hz. It is generally used as a control voltage for another module. For example, modulating a VCO will create vibrato while modulating a VCA will create tremolo.*

RM *Ring modulator, two audio inputs are utilized to create sum and difference frequencies while suppressing the original signals. This gives the sound a "robotic" quality.*

MIXER *A module that combines multiple signals into one.*

S&H *Sample and hold, which takes a "sample" of the input voltage when a trigger pulse is received and "holds" it until a subsequent trigger pulse is applied. The source is often taken from a noise generator. Sequencer, which produces a sequence of notes, usually a music loop.*

SLEW LIMITER *Smooths off the peaks of voltages. This can be used to create glide or portamento between notes. Can also work as a primitive low-pass filter.*

CUSTOM CONTROL INPUTS *Because modular synthesizers have voltage-driven inputs, it is possible to connect almost any kind of control. Pitch can be varied by room temperature if you wish, or amplification varied by light level falling on a sensor.*

Analog modular synthesisers where invented in parallel 1968 by R. A. Moog Co. and Buchla in 1963 [4]. There, sound signal is more often represented by oscillating voltage levels running through wires —the links— and manipulated by analog signal processor modules. Usually these modules where arranged in big racks.

One of the biggest problems of modular synthesisers is their limited ability to cope with *polyphony*. We say that a synthesiser is polyphonic when several different notes can be played at the same time. The basic implementation tech-

nique for this is to have several copies of the synthesis logic —each is called a *voice*— and dispatch each keystroke to an unallocated voice (if present, otherwise, some note priority logic is to be implemented). In old analog modular synthesisers this was achieved by having multiple root oscillators, maybe 2 or 4, but this multiplied the complexity of connecting the synthesis graph. That is one of the main reasons why modular synthesis was gradually abandoned for analog devices, as a naturally sounding keyboard-controlled instrument should have a much higher grade of polyphony and should remain usable —note that the required polyphony level is higher than the maximum number of keys that we want to be able to press simultaneously, because a note remains playing after the key is released during a *decay time* while the sound softly fades out.

Nowadays, the increasing power of computers allows us to build modular synthesisers by software. Even further, we are no longer limited by wires and we can use arbitrary data types as processing and instantiate copies of the modules as we wish only constrained by our memory and computation power. On software, it is easier to achieve polyphony but it is still a non-trivial problem to make it efficiently.

1.1.2 *An interactive synthesiser*

Even though Keith Emerson virtuously manipulated the wires and knobs of his Moog in the middle of his performances, old-school modular synthesisers have the inconvenience that they are rather static. It is very hard to control all the parameters during the performance. Changing the topology of the synthesis network is almost impossible, and in

most systems it causes clicks and other inconvenient noises as a result of abruptly connecting and disconnecting the wires —whether software or analog.

We should design our engine with care so no noise is generated as a result of manipulating the system live. But we should also provide other means to enable easier manipulation of the topology.

1.1.2.1 *Dynamic patching*

The *dynamic patching* [5] technique was first introduced in the ReacTable project and offers means to automatically interconnect the modules of a software modular synthesiser. When present, modules are laid out in a bi-dimensional space and each output automatically connects to the nearest available input of its same kind. The sink node that leads the output to the speakers is situated in the centre and the synthesis graph grows as a radial tree around it, as shown in figure 1.1. By simply moving one module from one position to another the user can radically change the topology of the network without doing a lot of plugging and unplugging of wires. A clever disposition of the modules on the space can help the artist to achieve new innovative ways of producing variations in his music.

1.1.2.2 *Touchable interfaces*

An specific problem for music software is that its interface is highly limited by the keyboard and mouse as interface. While a person has 20 fingers[1] she is limited to manipulate only one parameter at a time with the mouse.

1 She must be very skilled to use all of them at the same time!

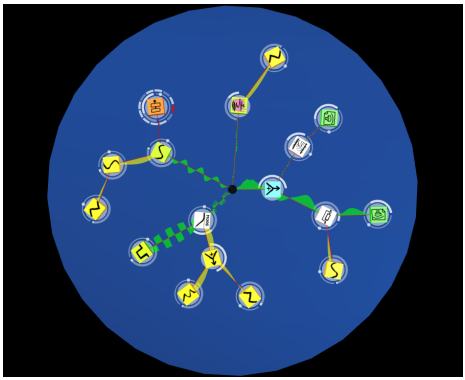


Figure 1.1.: A screenshot of Psychosynth 0.1.1 using dynamic patching to connect a complex graph.

There is an increasing availability of touchable interfaces, either specifically designed for music performance like the Lemur (figure 1.2) or general purpose ones like the IPad. There, one is no longer constrained by the single-click paradigm and not only can she manipulate various parameters at a time, different multi-tap gestures can expand the possibilities of a spatially limited user interface as several functions can be reached in the same points.

1.1.2.3 *Instrument simulating controllers*

While touchable interfaces might be better for manipulating the continuous parameters of the synthesis and dynamically manipulate a sequencer, many musicians would prefer more traditional interfaces to interpret part of their work in the "old-fashioned" way of playing an instrument [6]. There exists many kinds of electronic devices that simulate the feel of a keyboard or a drum-kit (figure 1.3) but instead

Figure 1.2.: The JazzMutant's Lemur touchable music interface. All the on-screen controls can be configured with an UI designer program, and later mapped to any program via OSC messages.

of producing any sound they send MIDI [7] messages to a synthesiser that reproduces the notes.

Our software should be able to interpret MIDI messages such that it can be controlled with such a device, making it more natural and creative for many musicians.

1.1.3 *A collaborative environment*

Since the beginning of times, music performance have been a collaborative act, with each performer contributing to the global rhythm and harmony by playing one instrument. Psychosynth should serve the purpose of producing arbitrarily complex music by itself as modules that implement not only synthesis, but also sampling and sequencing are added.

This integrated environment should be able to be manipulated by several people at the same time to allow a

Figure 1.3.: A conga-alike MIDI controller. When pressing different parts of its surface it will emit different MIDI note messages, that a synthesiser or sampler could use to either simulate a real conga or to produce any other sound.

collaborative experience. After all, it would be very hard for one person to control in a live performance all the subtleties of the sound generation by herself.

1.1.3.1 *Tangible interfaces*

One approach to achieve this is by using a user interface that is big enough to accommodate several performers around it. A tangible interface is one where the different elements are represented by physical objects that one can touch and move in physical space. The ReacTable implements such an interface where the modules of its synthesis engine are plastic blocks that are laid out over a round table that provides visual feedback thanks to a projector under it. As shown in figure 1.4, with such an intuitive and physically unconstrained interface several people can stand around the device and manipulate it.

Figure 1.4.: An example of the Reactable being used collaboratively by several people.

1.1.3.2 *Networking*

Networking support even further releases a device from the space constraint by allowing several instances of the software intercommunicate over a computer network —i.e. IP. At some point, latency problems can still be a drawback for this technique, but it can be useful for some kinds of collaboration that do not require perfect timing. When playing in a local range, this becomes perfectly valid even under high latency requirements.

This has long time been a main goal in Psychosynth. When running in collaborative mode, all the clients that connect to a server share the same synthesis graph and whenever and object is moved, added or deleted, or a parameter is changed, all are notified such that they keep the same view of the scene. Figure 1.5 shows an example of this feature being used live.

Figure 1.5.: An example of Psychosynth 0.1.4 being used collaboratively over the network. The picture was taken in a showcase performance held by Shakero8 and the author in the Open Source World Conference in 2008.

1.1.4 *A framework*

Music making and audio processing software and interfaces are evolving very fast. Abstracting the common core functionality needed by developers in a layered and abstracted programming API and development of a Free Software [8] framework is crucial to enable further focused research on the different areas previously discussed. Results of that research can be later integrated in the framework as they are stabilised.

Such a framework should enable the development of modular audio applications with an abstracted interfacing mechanism, probably following a Model-View-Controller paradigm, general enough to be able to support all the previously described qualities. If this task is properly accomplished, the framework could become the basis of a wide range of unexpected future Free Software applications.

1.2 OBJECTIVES

Taking all that into account, we should next define the concrete objectives for our present project. We should note that we depart from the basis of the current state of the GNU Psychosynth project —as of its version 0.1.7— and we assume that as previous work, not our current target.

1.2.1 *Main objective*

Objective 1 *Re-focus the GNU Psychosynth project as a development framework for the development of professional-quality modular, interactive and collaborative synthesisers.*

In the long-term we would like GNU Psychosynth to include innovative user interfaces, and some might even be developed as a side effect of this project —or we might just update the older one to rely on the new framework. However, that is not the purpose of this project, instead we will concentrate on the development of the underlying core architecture and implementation of its API.

This is so mainly because of time constraints. Also, if we were to miss some features in the core in order to allocate more time for the user interface, we have to take into account that this will probably hard to fix afterwards when a lot of code depends on the broken design. Also, user interface development is easier to do in a non-disciplined, voluntary and heterogeneous team. If we achieve a nice framework now we can still develop the user interfaces later with the help of other the people collaborating on the Internet; but, as these two years of stalled development have shown, it

is hard without the pressure of an external deadline and a project plan to invest a lot of time in rewriting the "invisible" but crucial parts of the system.

1.2.2 *Preconditional objectives*

There are two objectives of this project that can also be considered as a precondition for the success of our main goal. These are:

Objective 2 *Collaborate with professional musicians to get a defined understanding of the meaning of "professional quality" and their real needs.*

The students participating is this project have an amateur knowledge of music production. It is important to communicate and allow supervision by professional musicians and experts in digital music to assure the suitability of the software for use in a professional environment.

We are working in collaboration with the ArtQuimia Music Production School[2], which has long time been educating successful producers and is currently participating in the European Cross-Step project[3], and his director David García, a musician with professional experience in the industry as music composer and sound designer for video games.

Objective 3 *Research and apply the latest techniques in modular design and implementation and explore the boundaries of the underlying implementation devices.*

2 http://www.artquimia.net/

3 http://www.cross-step.org/node/4

The success of an framework relies on the proper decomposition of its features and its extensibility. Moreover, the authors of this project have a special fascination for programming languages, design patterns and software modularity. Even more, there is an active research community questioning and re-developing the modularisation and abstraction techniques of the underlying programming language C++, a fact that is more true as we approach the final resolution of the standarisation committee on the new C++0x standard. All this suggests that research and application of the state-of-the-art and even development of our new design and coding patterns will be one of the leading objectives during the project and play a leading role in its overall success.

1.2.3 *Partial objectives*

A more concrete subdivision of our main goal should be given. Note that these are not yet the detailed requirements, but an overall initial objectives vaguely elicited from the problem definition, the previous experience with the GNU Psychosynth software and our personal interests.

Objective 4 *Improve the framework to be able to load modules dynamically from plug-ins, satisfying our own API and/or third-party standards.*

This is a common feature in most industry standard applications and ours should support it. Many layers of the framework, specially those related to the dynamic patching, will require vast modifications to enable customisation to understand third-party modules.

Objective 5 *Improve the framework to be able to communicate with music controllers via MIDI or other industry standards.*

While not explicit in this wording, this adds the requirement for *polyphony* as in most cases such feature would be useless without it.

Objective 6 *Add basic synchronization and sequencing support to the framework.*

If we want to understand the software as a full live performance environment and not a bare synthesiser, this currently lacking feature is fundamental.

Objective 7 *Include common modular audio system utilities into the framework. Some of the most important being patch hierarchy and persistence.*

1.3 BACKGROUND AND STATE OF THE ART

1.3.1 *Modular synthesis*

The history of modular audio generation starts with the Moog analog synthesiser in 1967 [4]. Since then, a wide variety of analog modular synthesisers have been developed commercially, but retaining some limitations as described in the introduction in section 1.1.1.

Modular synthesis became more interesting with the uprising of computer based electronic music. One of the most important examples in this development is Max/MSP [9]. This is a dataflow based visual programming environment mainly targeted at music applications. In such software one

can add boxes where one types the name of the function that it should perform. When the name has been written some connection plugs appear on its corners depending on the module name, and one can draw lines connecting those plugs. Figure **??** showed an example of its functioning. The author of Max/MSP later developed a Free Software counterpart called Pure Data[10] that also has video processing and synthesis support.

Since then, many user-oriented commercial modular synthesisers have been developed. One of the most famous ones is Native Instrument's Reaktor (figure 1.6). In its version 5 it included the new *core-cell* technology, which allows the visual and modular design of the lower level parts of the DSP (Digital Signal Processing) programming, which transparently compiled to efficient machine code. Later, Plogue's Bidule is gaining special recognition for its simpler interface. A new interesting software is XT Software's EnergyXT, a DAW (Digital Audio Workstation) that has a "modular view" where one can route through wires all the MIDI and audio signals that flow behind the traditional sequencing view. Finally, Sensomusic's Usine is remarkable for introducing a highly customisable multitouch graphical interface on top of a modular synthesis environment.

On the Free Software side few modular synthesisers exist. Alsa Modular Synth[4] is one of the most popular. Ingen[5] is a more modern one whose code we praise for its quality. Most of these software still lack some of the features of their privative counterparts, with the ability to create customised user interfaces being the most relevant. Still, Free Software offers a very interesting modular approach to the sound

4 http://alsamodular.sourceforge.net/
5 http://drobilla.net/blog/software/ingen/

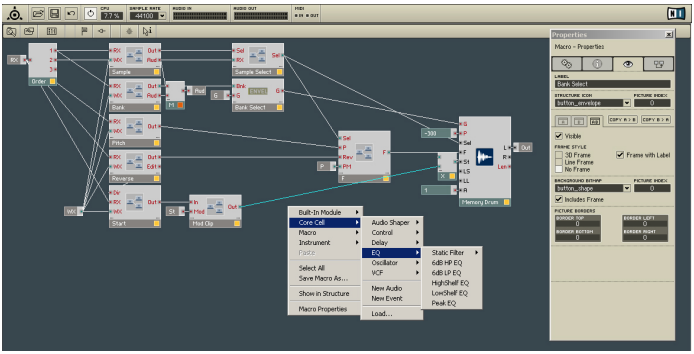


Figure 1.6.: Native Instrument's Reaktor editing a *core cell* based patch.

system management that has a similar rival only in latest OSX versions: Jack[11]. Jack is an audio server providing zero-latency interprocess communication that acts as a patch-bay among any supporting audio applications. The user can route the output of one program to the input of any other program, or the soundcard sink, or whatever exposes a Jack port. It can be used to route MIDI signals too and recent versions include very nice features, like collaborative routing over the network for distributed performances [12].

1.3.2 *Touchable and tangible interfaces*

In the last decade, the development of touchable and tangible user interfaces have been of rising interest. An interesting but not very updated survey that gives some taxonomical background and analyses a wide variety of products can be found here [13]. One of the first attempts in using them for improved interaction in musical software is the Audio Pad[14], where the user places and moves tagged pucks

on a bidimensional surface. This is an example of an interface with active tangibles, because the pucks have an RF transmitter to allow their recognition. The Jam-o-Drum, on the other hand, offered a percussive collaborative approach where performers sit on the corner of an interactive table [15]. Many videogames and other kinds of application have been developed on top of the Jam-o-Drum hardware too. Since then, a huge number of different table based interfaces have been made, an example listing can be found here[6].

Maybe the most interesting example, which inspired the whole development of GNU Psychosynth, is the Reactable project[1]. Its user interface is based on modular synthesis and uses the dynamic patching technique for easily manipulating the patch topology. It uses innovative digital image processing techniques to detect the position and rotation of passive tangibles [16]. In the Reactable system a translucid round surface hides a projector and a high resolution camera underneath, as shown in figure 1.7. Finger-tips and the specially designed tags called *fiducials* that are placed on the table surface are captured by the camera and recognised by the ReacTIVIsion system on a computer. This system sends OSC[17] (Open Sound Control) based messages to a synthesiser following the TUIO (Tangible User Interface OSC) protocol [18]. The literature describing the initial prototypes used Pure Data as the underlying implementation language for this synthesis engine, but conversations with the authors of the Reactable suggest that the current commercial implementation is written in C++. This synthesis engine is connected to a visual engine that generates visual feedback and sends it through the projector. The picture is reversed so it can be correctly visualised through the translucid surface.

6 `http://mtg.upf.edu/reactable/related.htm`

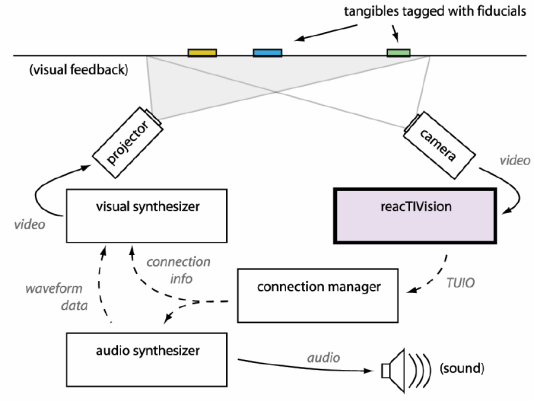


Figure 1.7.: A typical Reactable setup. Source: The Reactable project.

The Reactable has been very successful and a very interesting fact is that the computer vision component of the system is Free Software, so it could be integrated with GNU Psychosynth in the future.

Some other remarkable tangible and highly interactive user interfaces that where released around 2007 too are the JazzMutant's Lemur and the Iwai-Yamaha's Tenori-On [19]. The former offers a fully customisable OSC based multitouch interface, where one can design interfaces with sliders, virtual-knobs, X-Y pads and all kinds of multi-touch controls that are mapped to OSC messages that can be interpreted by the audio engine of choice. The later is a 16x16 matrix of LED illuminated buttons that can be configured in different modes that provide innovative sequencing mechanisms (figure 1.8).

Nowadays, multitouch interfaces are the main trend, specially after the explosion of the tablet market, but big music

Figure 1.8.: Electronic music artist Four Tet performing with a Tenori-On

software companies still have to catch up the latest hardware developments. Specially interesting for the GNU Psychosynth projects it the Indamixxx 2 tablet, which is based on the Meego OS[7] and oriented towards music production and live performance — we should definitely keep an eye on it and develop a multi-touch enabled user interface for it.

1.3.3 *Audio synthesis libraries and frameworks*

One of the first synthesis libraries that where evaluated before the development of GNU Psychosynth started is the STK (Synthesis ToolKit) [20] but it lacks proper dynamic signal routing mechanisms and some details, like the sample format being hard-coded to float, seem too constrained.

A popular sound synthesis and music composition DSL (Domain Specific Language) is CSound, which later was ex-

7 Nokia and Intel's joint effort to provide a GNU/Linux based operating system for embedded devices, mobile phones, netbooks and tablets.

tended with a C and C++ programming API[21]. Many DSP oriented DSLs have been developed, but they are not general enough to support the development of the fully featured applications that we wish on top of GNU Psychosynth. Still, some of them are worth mentioning. Notable examples are SoundCollider [22], which was for a long time the reference sound synthesis DSL; Chuck [23], that adds concurrency and time control in an elegant and boilerplate-less fashion, and the newer Faust [24], which is based in a functional and data-flow paradigm and interfaces easily due to its compilation to C++.

While we are focusing on C and C++, many languages have music related libraries. Impromptu[8] is a Scheme environment for live-coding, this is, live music performances where the programming is done in front of the audience while the programmer writes and mutates the code[9]. It supports video manipulation too, but sadly is available for OSX only. Its heavy modularity and dynamism is what make Lisp dialects so interesting for live-coding and music composition. Common Music[25] was started in 1989 and it is a highly featured Common Lisp framework that, among other things, provides a very elegant embedded score notation.

Maybe the most similar project to GNU Psychosynth in its approach is CLAM (C++ Library for Audio and Music) and award winning library based on modular synthesis[26]. Its design will be carefully taken into account during the redesign of few of Psychosynth core components. Still it

8 `http://impromptu.moso.com.au`

9 The video *Algorithms are thoughts, chainsaws are tools*, by Stephen Ramsay is a great introduction to this technique.
`http://createdigitalmusic.com/2010/07/thought-and-performance-live-coding-music-explained-to-anyone-really/`

does not precisely fit our needs as it is too oriented toward research simulations and does not support polyphony.

2

ANALYSIS AND PLANNING

> As I see it, criticism is the prime *duty* of the scientist and of anyone who wants to advance knowledge. Seeing new problems and having new ideas, on the other hand, are *not* one's duty: originality is, rather, a gift of the gods.
>
> 1982 Preface to *Quantum theory and the schism in physics*
> Karl Popper

2.1 REQUIREMENT MODELLING

In the following section we discuss the requirements that we want to impose on the resulting framework. Note that functional requirements are hard to express for an API, but a vague description of the desired qualities and features is still possible will guide or development process.

Quite often, we will focus on the final feature that should be implementable through the framework. Also, note that the framework already had many features at the beginning of the project, some of which we will discuss in section **??**. We will try to avoid describing the requirements related to those ready facilities in this section, as the only purpose of those is to aid the actual development of this thesis project. Still, some already satisfied requirements might be made explicit when some other related requirement follows, or because of their relevance to the user or just for the global consistency of the text.

As we suggested in our objective 2, these requirements have been elicited with the assistance of the ArtQuimia Music Production School in order to ensure the suitability of the software in a productive usage context.

2.1.1 *Functional requirements*

2.1.1.1 *Basic audio support*

Requirement 1 *The library must provide data-structures and basic algorithms for manipulating audio signals.*

Requirement 2 *The library must provide means to output and input audio data to and from files in various formats, including, at least: Microsoft WAV, Apple AU and OGG Vorbis.*

Requirement 3 *The library must provide means to output and input audio data to and from sound-card devices in an asyn-*

chronous real-time fashion, supporting, at least, the following audio systems: ALSA[1], OSS[2], Jack[3].

2.1.1.2 *Node graph support*

Requirement 4 *The library must include means for producing the audio as the result of executing a dataflow-graph.*

Requirement 5 *The library user must be able to define his own processing nodes.*

Requirement 6 *Each node should have an arbitrary number of named signal and control input and output ports. The difference between signal and control ports are that the later are sampled at much less frequency.*

Requirement 7 *Both signal and control ports should be able to process information in arbitrary datatypes. Signal ports may have practical limitations as for the real-time constraints is concerned.*

> **Note 2.1 (Realtime constraints)**
> *All the processing done inside the nodes should satisfy soft real-time constraints. This is so because in order to produce sound with low latency (see requirement **??**) the sound processing should be done in blocks (buffers) as small as possible that are delivered to the sound-card as soon as possible. If the deadline is not met, a disturbing "click" sound will be heard because of the zeroes assumed by the sound-card during the period that*

1 Advanced Linux Sound Architecture: `http://www.alsa-project.org/`
2 Open Sound System: `http://www.opensound.com/`
3 `http://jackaudio.org/`

it did not have any audio to output. For example, for a 44100 Hz sampling rate and a 128 frames block size, it should take less than

$$\frac{1s}{44100\ frames} \cdot 128 \frac{frames}{block} = 2.90 \frac{ms}{block}$$

to process and deliver an audio data block.

In practice, this means that the processing algorithms should take an amount of time proportional to the amount of data to process —i.e. they are $O(n)$. This in turn disallows writing and reading files and most operations that cause non deterministic hardware operations or force context switching (mutexes might be unavoidable but should be used scarcely). Also, allocation of memory on the heap should be avoided as it has a non-predictable and potentially non-linear time consumption. Of course, the framework should provide hooks to do those forbidden operations outside the audio processing thread, but we consider that a design issue that should be addressed later.

Requirement 8 *Each output port must be connectable to arbitrary number of input ports. Each input port must be connectable to one output port.*

Requirement 9 *Ports may be defined statically —i.e. at compile time— or dynamically —i.e. at runtime.*

Requirement 10 *The system must allow the hierarchical composition of nodes, with special* input *and* output *nodes that are exposed as ports in the parent level.*

Requirement 11 *Nodes can be either monophonic or polyphonic. A polyphonic node internally has a number of copies of its processing state, called* voices*, that are dispatched accordingly trough trigger signals.*

> **Note 2.2 (On the scope of polyphony)**
> *We will avoid specifying here details on how polyphony works that are still quite important as a* usability *concern. For example, should monophonic ports be connectable to and from polyphonic ports? Should there be polyphonic and monophonic nodes in the same patch hierarchy level? How are the voices dispatched and mixed? Because answering this issue highly affects performance and implementability trade-offs, these issues are left open until the design stage of these components.*

2.1.1.3 *Dynamic loading of nodes*

Requirement 12 *The system must be able to dynamically load modules developed with standard interfaces, at least using the LADSPA standard*[4]*, but LV2*[5] *and VST*[6] *are proposed too in this order of priority. Note that in some cases this implies the automatic fixing of the semantic impedance among the interfaces of Psychosynth nodes and the third-parties'.*

Requirement 13 *The system should be able to dynamically load modules developed using the same native interface exposed to library users to define their own modules.*

4 Linux Audio Developer's Simple Plugin: `http://www.ladspa.org/`
5 LADSPA Version 2: `http://lv2plug.in`
6 Steinberg's Virtual Studio Technology: `http://ygrabit.steinberg.de/`

2.1.1.4 *Dynamic patching*

Requirement 14 *The system should optionally release the library user from arranging the interconnections among nodes by using* dynamic patching. *When using* dynamic patching *an output port is connected to one input port. The input port is the closest (in the sense of euclidean distance, we assume that the modules are laid out in the space) input port of the same type not belonging to the same node that is free —i.e. is not connected already to a closer node.*

When combining dynamic-patching with dynamically loaded modules using standard interfaces like LADSPA, one might need extra information to correctly perform the dynamic patching.

Requirement 15 *The system should be able to locate and automatically load if present special description files containing the required information for some dynamically loaded modules to work correctly. This file should specify which ports are eligible and to which ports they are connectable to.*

While this may change due to design considerations, we suggest specifying a set of tags for each port, with the following connectability rule for port A and B: $connectable(A, B) \Rightarrow tags(A) \cap tags(B) \neq \emptyset$.

2.1.1.5 *View independence*

The following requirements are here to suggest an observable interface for the synthesis model satisfying a model-view-controller architecture. This is one of the key concepts in achieving a network-based shared environment that is

transparent to a wide range of graphical interface experiments or external MIDI controls.

Requirement 16 *The system must enable the development of graphical interfaces —or any other instance of the more abstract concept of* view*— that can coexist with each-other.*

2.1.1.6 *Synchronisation and sequencing*

Requirement 17 *The system should include a notion of* tempo *such that external imprecise events can be quantised and synchronised to.*

Requirement 18 *Parameters of various node kinds, specially time related ones, should be controllable as a factor of the current tempo.*

I.e. the frequency of an LFO should be fixable such that the wave length is a factor of the tempo, and the phase should be synchronisable to beat.

Requirement 19 *The system should be able to synchronise to the MIDI clock.*

2.1.1.7 *Collaborative support*

Requirement 20 *The system must be able to receive, process and use as control source MIDI events coming from other software or external hardware devices.*

Requirement 21 *The system must be able to receive, process and use as control source OSC events coming from other software, computers or external hardware devices.*

Requirement 22 *The system must be able to use a specially crafted OSC based protocol to enable the collaborative manipulation of the same node graph among different computers connected through the network.*

Note 2.3 (Relaxing the "shared-environment" requirement) *Requirement 22 is currently implemented by broadcasting all events among all the participants in a shared session. In presence of audio input nodes, dynamically loaded modules and other sources of non-determinism —this is, sound that might not depend only on the sequencing of certain events— this gets harder to implement. We do have some solutions in mind, like placeholder nodes that stream the local non-deterministic signals through the network too, but it might be too hard to implement in the context of this master thesis project and only a proof-of-concept implementation will be required.*

2.1.1.8 *Persistence*

Requirement 23 *The system must be be able to store and load the graph layout and control values the node graphs. Sub-graphs in the graph hierarchy should be storable and loadable individually.*

2.1.1.9 *Optional functionality*

Requirements in this section are not such, but instead they are bare ideas that would be nice to have but are considered too time consuming, hard or not urgent enough to be considered a measure of the project success.

Requirement 24 *The highest level part of the API should have a Python —or any other dynamic language of choice— binding for the rapid prototyping of audio applications and user interfaces on top of it.*

2.1.2 *Non-functional requirements*

2.1.2.1 *Free Software*

Requirement 25 *Unless constrained by very specific hardware, the system should not add any non-Free Software dependency — i.e, it must be able to compile and run without executing any privative software bit.*

2.1.2.2 *Professional quality sound*

Requirement 26 *The software must be able to work at different sampling rates up to 96000 Hz.*

Requirement 27 *The software should be able to use arbitrary precision samples. Up to 64 bit floating point samples are required.*

2.1.2.3 *Performance*

Requirement 28 (Latency) *The software should be able to work with a block size as low as possible down to 64 frames, as long as the underlying hardware permits it.*

2.2 OPEN, COMMUNITY ORIENTED DEVELOPMENT

The developers of this software are strong proponents of Free Software development. This is specially relevant in an academic and public environment like ours. Therefore, the software not only is distributed under a Free Software license, it also follows an open community development model, where everyone can read, use and modify the source code as it is developed and there are means for online communication promoting development among volunteering distributed peers.

Previous versions of the software are available on the Internet for download and it has an official web page: `http://www.psychosynth.com`

2.2.1 *Software License*

The software is licensed under the GPL license version 3, offering strong copyleft requirements — i.e. derived works and software linking against the library must be distributed under the same terms. The full description of the license is included in the appendix B.

While the GPL3 is often misunderstood as inadequate for a library, that is not true in the context of libraries that provide unique features, as it motivates the release of third-party software that is attracted by these as Free Software too [27]. This is not only personal belief, it is also the official guideline in the GNU project.

2.2.2 *The GNU project*

Since October 2008, Psychosynth is part of the GNU project. GNU was started in 1984 by Richard Stallman with the long term goal of providing a fully free —as in speech— operating system [8]. Under the umbrella of GNU, Psychosynth gets access to a broader community, technical support and it is a recognition of its attachment to the Free Software ideals.

2.2.3 *Software forge and source code repository*

A *software forge* is an online service targeted at aiding the development of Free Software. It offers a series of tools to aid the community participation and distributed development, such as *bug trackers*, *support trackers* and *mailing lists*. One of the most important features is the *revision control system* that serves as source code repository.

GNU Psychosynth is hosted in the Savannah software forge, and its project page can be accessed here:

```
http://savannah.gnu.org/projects/psychosynth
```

The project is using GNU Bazaar as distributed revision control system. One can fetch a copy of the latest version of main development branch by executing the command

```
bzr branch http://bzr.sv.gnu.org/psychosynth/trunk
```

2.2.4 *Communication mechanisms*

Fluent distributed communication is essential for the advancement of a free project. For this purpose we offer the following tools.

2.2.4.1 *A blog*

The blog serves as an informal and easy way of getting the latest news on the development. It is most of the time technically oriented and can serve as source of motivation for external people to contribute to the project and as a summary of the current status of the project. It can be accessed through:

`http://planet.gnu.org/psychosynth/`

2.2.4.2 *Mailing lists*

Mailing lists are multi-directional broadcast based communication means and the main spot for discussion of development (from the developer point of view) and getting news or asking for help (from the spare user point of view).

GNU Psychosynth has two mailing lists.

- Users mailing list:
  `http://lists.gnu.org/mailman/listinfo/psychosynth-user`
- Developers mailing list:
  `http://lists.gnu.org/mailman/listinfo/psychosynth-devel`

Because being registered in many mailing lists can cause management issues to some users, the Gmane project[7] offers

7 `http://www.gmane.org/`

a newsgroup gateway that can be used by Free Software projects to allow participation in their mailing lists with Usenet capable software. Psychosynth mailing lists are registered there and can thus be accessed through:

- `gmane.comp.gnu.psychosynth.user` for the users mailing list.
- `gmane.comp.gnu.psychosynth.devel` for the developers mailing list.

2.3 DEVELOPMENT ENVIRONMENT

The development environment is very important for the project success. This section should clarify our choices and explain the rationale behind such decisions.

2.3.1 *Programming language*

Psychosynth is developed using C++. This decision is based on the following facts:

- It is a stack based language with fine grained control over memory management. As we introduced in note 2.1, this is crucial during the development of live audio processing software.
- It is a mature language with widespread tool support and a very good Free Software compiler: GCC.
- Apart from its low-level mechanisms it has powerful means for abstraction, most of which are designed to pay zero cost.

- It is multi-paradigm, and as such it can easily adapt to the natural way of expressing the concepts of a heterogeneous system as this, where we want to go to from low level DSP programming to high level interface design.
- It is compatible with C, which gives us direct access to the widest range of libraries available. Most audio-processing libraries are written in C and thus we can save a lot of time in implementing mathematically complex algorithms.

Of course, it also has its flaws, like unnecessary complexity and unsafety in some corners, most of which are justified by its backward compatibility to C and evolutionary design. Some of this flaws are nonetheless going to be solved in the next standardisation of the language, to be released in 2011 [28].

Compilers are starting to support it and we are very interested in exploring the benefits and new programming patterns and design benefits that it can provide. Because Psychosynth is an ongoing and forefront project we do not fear portability — and after all, GCC is very portable! — and as such we are going to use the facilities in C++0x as soon as they are supported by the latest version of GCC included in the Debian Sid.

2.3.2 *Operating System*

The project is mainly targeted at GNU/Linux, which is the most widespread free operating system, therefore, compliance with it is the highest priority. That is also the operating

system of choice of the authors of this project so it feels like a natural environment and there is no extra effort needed.

Still, we will try to comply with POSIX such that porting to other Unix operating systems is easy. Sadly, there is no universal high performance and fully featured cross-platform audio output engine and thus that is an important portability boundary. In the future, maybe with some financial aid, we might be able to port the software to OSX, which is near to be the most used operating among musicians [6][8].

2.3.3 *Third party libraries*

In this section we give an overview of the external libraries used in the software. Note that they have to be chosen in compliance with requirement 25.

2.3.3.1 *Boost*

Boost[9] is a set of libraries for C++ that are peer-reviewed and specially crafted to integrate well with the paradigms and abstraction techniques of the standard library. Many of its modules are, actually, going to be part of the standard library in the future C++0x standard.

8 The cited survey dates back to 2006. Given the recent rise in popularity of Apple products, we speculate that OSX might be even more popular that Windows among musicians nowadays.

9 `http://www.boost.org`

We use few of the Boost facilities, some of them including `boost::filesystem`, `boost::thread`[10], `boost::mpl`, `boost::test`[11] among others.

It is extremely portable and licensed under the permissive Boost Software License.

2.3.3.2 *Libxml2*

We use Libxml2[12] for parsing the XML configuration files. It is very portable and licensed under the permissive MIT License.

2.3.3.3 *LibSndFile*

We use LibSndFile[13] for loading different uncompressed sound formats. Note that from version 1.0.18 it also supports OGG and FLAC formats, and thus we plan to use this instead of LibVorbis in the future. It is very portable and is licensed under the LGPL 2 and 3.

2.3.3.4 *LibVorbis*

We use LibVorbis[14] for reading OGG files. It is released under the BSD license and is very portable.

10 We plan to replace this with the new threading facilities in the standard as soon as they are implemented in GCC.
11 A wonderful unit testing framework.
12 `http://xmlsoft.org/`
13 `http://www.mega-nerd.com/libsndfile/`
14 `http://xiph.org/vorbis/`

2.3.3.5 *SoundTouch*

We use SoundTouch[15] for time-stretching, this is, changing the pitch and tempo of a song independently. It is licensed as LGPL and works on mayor operating systems.

Some people claim to obtain better results in performance and sound quality with the Rubber Band library[16] [17] and we will probably replace SoundTouch by this one in the future.

2.3.3.6 *ALSA, OSS and Jack*

In accordance to requirement 3 we use ALSA, OSS and Jack respectively. Licenses are LGPL2+ for all of them. ALSA is Linux only, Jack works on GNU/Linux and OSX and OSS works on Linux and some other POSIX operating systems.

2.3.3.7 *LibLO*

LibLO[18] is used for OSC support. It is licensed as LGPL2+ and is POSIX compliant.

---

15 `http://www.surina.net/soundtouch/`

16 `http://www.breakfastquay.com/rubberband/`

17 The DJ software Mixxx, where sound stretching quality is very relevant, is moving towards RubberBand and their developers support the above claims: `http://www.mail-archive.com/mixxx-devel@lists.sourceforge.net/msg03103.html`
Also, Ardour later moved tater moved to this library: `http://www.ardour.org/node/1455`

18 `http://liblo.sourceforge.net/`

2.3.3.8 *The user interface libraries*

The user interface included at the beginning of the project is based on Ogre3D[19], using OIS[20] (Open Input System) for keyboard and mouse input and CEGUI[21] as widget toolkit.

During the development of the previous Psychosynth versions CEGUI has proven to be extremely painful with an overengineered API and bugfull implementation. Also, the 3D interface, while being fancy it is confusing for some users and did not offer anything new to the experienced musician, as some blog reviews showed. It contains some interesting concepts — zoomability being the most important — but it is too distracting after-all. Thus, while we are going to keep this interface during the development of this project, we want to later rebuild the GUI using Qt, which is multi-touch enabled and could open us the door to the yet-to-come wide range of Meego based tablets.

2.3.3.9 *Build system*

We use GNU Autotools[22] as the build system for the project. It is extremely portable and the de-facto standard among Free Software, even though some interesting alternatives are emerging. Nonetheless, Autotools are suggested in the GNU Coding Standards[29] that we shall follow during our development due to our affiliation to GNU.

19 `http://www.ogre3d.org/`

20 `http://www.ogre3d.org/`

21 `http://www.cegui.org.uk/`

22 We use mainly Autoconf (`http://www.gnu.org/software/autoconf/`), Automake (`http://www.gnu.org/software/automake/`), and Libtool (`http://www.gnu.org/software/libtool/`).

2.3.3.10 *Documentation system*

Because we are developing a framework, it is specially important that the public interfaces are properly documented. We are going to use Doxygen[23] to embed the API documentation in code comments. A reference manual generated by Doxygen should be attached to this document.

2.4 ARCHITECTURE AND CURRENT STATUS

The Psychosynth project was started in 2007 and since there has been some relevant developments. The screenshot in figure 2.1 gives a broad perspective of the current features of the project. On the bottom we can see some buttons for poping up different menu windows, such as for settings, recording the sound into a file or connecting several instances over the network. The window on the top of the screen is used to add elements to the processing graph. The smaller *properties* windows contains a listing of all the actual parameters of a node and allows us to give numeric values to them. All around the screen, sinusoids, step sequencers, file samplers and echo filters are interconnected as represented by the 3D blocks. For a more detailed description from the user point of view, please refer to the user manual that can be checked online on the project webpage [24], where there are also demonstration videos and other multimedia material. The article [30] can serve as an introduction to the project too.

23 `www.doxygen.org`
24 `http://psychosynth.com/index.php/Documentation`

Figure 2.1.: A screenshot of GNU Psychosynth 0.1.4.

2.4.1 *A critique on the term framework*

The term *framework* is used many times in this and other projects and is becoming a techie buzzword. In many contexts it is abused as a synonym for the term *library*. Instead, we believe that a framework is something different, following the definition given in the famous design patterns book by the gang of four [31].

We use the term *library* when the root of the call graph is on the client side and she invokes the library function sparely to obtain some concrete functionality. Instead, a *framework* stands in the root of the call graph and the client code is called through extension hooks in the framework,

following the "Hollywood principle" — "Don't call us, we'll call you.".

Because the Psychosynth system is layered, one can just use the bottom layers as a library, or rely on the upper layers that properly use inversion of control like a framework.

2.4.2 *The layered architecture*

At the current stage, GNU Psychosynth implements a layered architecture [32]. This intends to promote a more decoupled design, as calls between modules are only allowed from top down.

Also, the library has many features, many of which some users may not need. This layered approach could allow a potential user to avoid any special overhead when he is only using some bottom layers. Still, note that the library is now compiled will all layers bundled in the same shared-object file, so this is not a fact now. Because the heavy redesign ongoing during this project, we shall postpone that until the late development stages when the layer interactions are clear and stable.

Figure 2.2 represents the current layered architecture. Lets make a bit more in-depth discussion of each layer.

2.4.2.1 *The `base` layer*

The base layer includes basic facilities that may be reusable in any other part of the application. Some of the most relevant examples:

La capa node hay que cambiarlo por `graph`

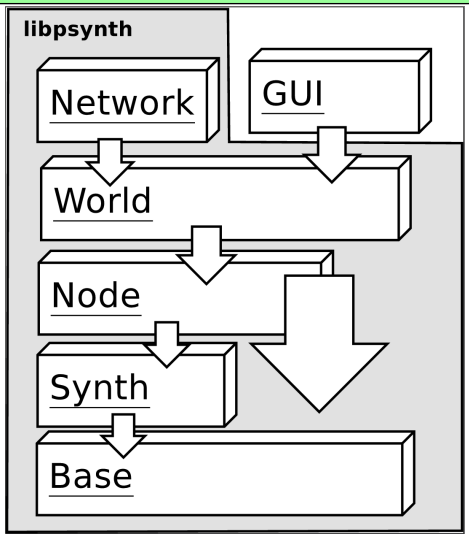


Figure 2.2.: The Psychosynth layered architecture.

CONFIGURATION PERSISTENCE Classes for representing program settings in a hierarchical manner. This configuration system can use interchangeable backends and has an observable interface.[25]

In fact, we do not recommend using this module in the core of the intermediate layers of the library because it can cause unexpected coupling, but this is not too clear and we keep it here for historical reasons.

IMPLEMENTATION OF DESIGN PATTERNS While the term *design pattern* means reusable design structure, not reusable code, language abstractions can make them

25 We use quite often the term *observable interface* which is rare in the literature. By this, we mean that it provides signals, listeners or other event mechanisms, instances of the *observer* design pattern[31].

implementable as code in some cases. Andrei Alexandrescu proves this point for C++ in [33]. Thus, we provide implementations, quite often similar to Alexandrescu's, to various recurring design patterns such as *factory* or *singleton*. Some implementations are not inspired in Alexandrescu's, like the generic *composite*.

COMMAND LINE ARGUMENT PARSING While we have considered moving to Boost's Program Options library, our own implementation have different trade-offs and is rather extensible.

LOGGING SYSTEM A hierarchical and multi backend logging system for registering messages from other modules. It should be used instead of direct output to `std::cout/cerr` in all the code.

FILE MANAGEMENT TOOLS That ease the task of finding resources in the hard-drive and can cache results.

Some other minor classes and tools are excluded from this list. During the development of the project we will drop in this layer classes that feel interesting at any abstraction level.

2.4.2.2 *The `synth` layer*

This layer contains classes for the *static* construction of synthesisers and sound processing. The audio input and output facilities are considered to be in this layer, and as well audio processing data structures —like ring buffers, multi channel buffersm, etc.—, basic implementations of filters, oscillators and audio scalers.

By static, we mean that this code does not provide any dynamic routing facilities, instead, the programmer is in

charge to assign buffers and call the processing elements manually.

Requisites 1 to 3 should be implemented here. Non functional requisites 26 to 27 are specially relevant in this layer too.

2.4.2.3 *The `node` layer*

This layer provides the facilities for the *dynamic* construction of synthesisers. It includes the mechanisms for describing and executing the modular synthesis graph with the signal flow and so on. Figure 2.3 represents the main concepts behind the current design. Ports are considered as "signal ports" using the terminology in requirement 6 — "control ports" are similar to "parameters", but parameters are not a precise model of "control ports" as they can not be routed and are intended for communication between the client user interface code and the audio thread state.

The communication system used to propagate values between the audio processing thread and the client thread is represented in figure 2.4. Values are copied to and from an intermediate channel between the audio processing blocks.

Requisites 4 to 11 should be implemented in this layer. A heavy redesign of its API and many of its internal implementation is to be expected for that to be accomplished.

2.4.2.4 *The `world` layer and the Model View Controller architecture*

This layer simplifies the interface exposed to the previous layer and makes it *observable*. This is fundamental for the

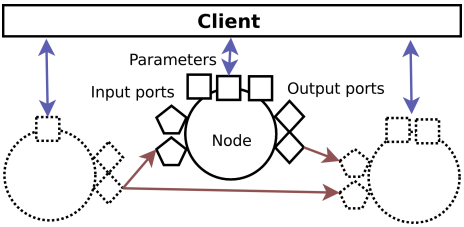


Figure 2.3.: Representation of the node graph as in Psychosynth 0.1.7. Input ports are represented as pentagons, output ports as rombus and parameters as squares.

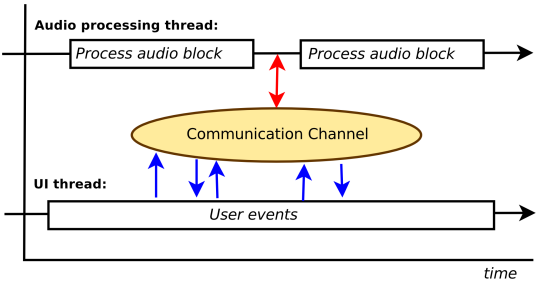
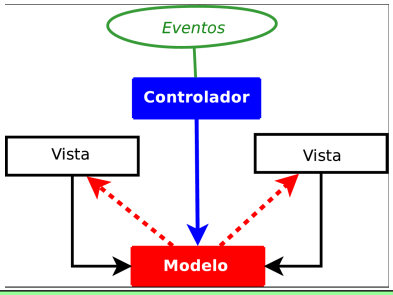


Figure 2.4.: Communication between the audio and user interface thread as in Psychosynth 0.1.7

Model View Controller that the system implements. Figure 2.5 represents this architectural style. On the following, we can refer to this observable interface abstracting the synthesis engine as *the model*.

Several views can coexist independently —for example, a GUI user interface and a web client—, that get updated whenever a value has changed in the model. They register themselves on the model at the beginning and then become passive entities that get called by the model. The model changes when controllers invokes methods on it, several



Mejorar y traducir esta figura.

Figure 2.5.: The Model View Controller arquitectural style. Dashed arrows represent indirect invocation — i.e. via the *observer* design pattern— and normal lines represent normal method calls and data access.

controllers can coexists too. Usually, models and views come in pairs. For example, a GUI view has an associated controller that triggers the manipulation of the model in the eventuality of certain user actions like clicking a button; in this case the representation (the buttons) and the action (clicking it) are strongly related, but this is not necessarily true for other situations.

This layer also abstracts the graph interconnection mechanism using the *strategy* design pattern. Concretely, dynamic patching is implemented here and the interface exposed in this layer hides the manual node interconnection mechanisms but provides observability for topological changes.

This layer should implement requisite 16.

2.4.2.5 *The application framework layer and the view and controller layers*

There is a thin layer, instance of the *facade* pattern, called `app`, that was hidden for simiplicity in figure 2.2 representing the layered architecture. It sits on top of the `world` layer and is in charge of initialising the *world* and defines a configuration tree, using the facilities in the `base` layer, and setups the machinery using the observability of the configuration system to keep coherence between the user preferences and the status of the synthesis model — for example, if the "`psynth.output`" setting is changed, it automatically creates the appropriate output object instance, sets its values and substitutes the output node in the synthesis graph. This layer also sets up the command line argument parsing and installs some common audio setting arguments in the parser. This layer is where Psychosynth becomes a framework at its most pure level, as it offers a `psynth_app` class whose `run` method should be the only call in the program `main`, and in turn delegates the heavy work to user code that is hooked as method overrides of that same class.

Orthogonal to this layer and sitting also on top of the `world` layer the networking layer offers a set of views and controllers[26] that can be used to create the shared environment described in requirement 22 and thus allowing collaborative performances. This is an example of the value of the MVC architecture: because views and controllers are orthogonal, the user interface does not need to know about the presence of this to function properly; one could develop a new experimental and cool UI and it would automagically

26 A primitive implementation of such at this stage of the development.

be able to work with third party clients over the network, even potentially using a different user interface.

On top of all this there is in the current version of Psychosynth the code of the 3D user interface and small and simple command line tools intended to be used as GUI-less network capable clients and servers. But all this code is not part of the framework, as it wants to be user interface agnostic, so we will not further describe that code.

2.5 PROJECT PLANNING AND METHODOLOGY

2.5.1 *Rationale — A critique on software engineering*

Choosing a well known software engineering process is considered one of the first steps to be taken in a final master thesis project. In our school we study with most detail the Cascade Process and the Unified Rational Process.

Those development processes propose a fordist software production model, targeted at huge development teams and the development of stable code bases in non innovative well defined fields. Martin Fowler makes a great point [34] criticising the often repeated argument stating that software engineering is like any other engineering where creative analysis and design is only the first step and thus coding is analogous to construction. As he says, the construction is done by the compiler and people involved in programming are actually doing an intellectual and creative work too — in computing, any systematic task can and must be automated. The *programming = construction* metaphor is alienating for the programmer, who is completely excluded from the task

of criticising and improving the software design, and thus this metaphor often leads, in the end, to bad software.

Moreover, fordist development models take risk control and client requirements satisfaction as most important factors. Because we are in an academic environment, there are two more important factors: the pedagogical value of the project —this is, that the student involved takes the risk of exploring the unknown by himself— and the research value —this is, that the student involved takes the risk of exploring the unknown by humanity.

Of course, this is neither a pure research project, so we can not substitute a software development project by the scientific method. But we can choose a more dynamic methodology that includes *falsification* in one way or the other; *agile* methodologies[27] propose many alternatives that could be valid for a master thesis project.

Still, these methodologies are, we believe, inadequate for this concrete project. The main reason is that this project is developed by only one person. Most methodologies, specially agile ones, put emphasis on the developer communication methods and collective decision making, so they are often inadequate and too constraining and time consuming for an unipersonal team. The Personal Software Proccess [35] proposes a methodology that is specially targeted at personal software developed by engineering students. Sadly, we are not very familiar with it —and do not have enough time to make that happen within the time constraints of the project— and it seems too be to specific and time consuming in its time tracking proposal.

27 http://agilemanifesto.org/

Because we still believe that some rational planning and methodology is needed, we propose in the following a defined but unconstrained methodology that is specially tailored for our circumstances, capturing the most common elements in other software processes.

2.5.2 *An iterative development model*

Because of the size and complexity of the project, we should not consider developing it all at once. Moreover, the layered architecture of the starting code base and the variety of requirements that we want to satisfy favour an iterative development.

For all this, we want to split the development in iterations. Each iteration is composed by the following phases: *design*, *implementation*, *verification* and *integration*. Each iteration shall be assigned a set of requirements from the specification in section 2.1 that are to be satisfied after the successful accomplishment of that iteration.

2.5.2.1 *The design phase*

In the design phase we shall define the API that we would like the current subsystem have. Because we are developing a library and framework with a public interface, the design phase is specially relevant shall be done with care.

We do not enforce a particular method for documenting the design as different paradigms favour different documentation means. For example, in the first iteration we will develop a library heavily based on metaprogramming, where UML does not fit very naturally. Still, the documen-

tation should include rationale explaining why the design decision lead to the satisfaction of the requirements that we want to satisfy after this development iteration. Also, it may be found that a requirement may be impossible to satisfy on the current iteration or that this requirement is to be better integrated in some other iteration. The developer is free to reassign that requisite for later iteration properly documenting this as a post-analysis plan fix.

What we do enforce is that all the API is documented with Doxygen for the sake of completeness of the reference manual that you should receive along with this document.

2.5.2.2 *The implementation phase*

During the implementation phase the code implementing the design should be written. It is possible and even recommended to modify the design during this phase as inconsistencies and fundamental problems are found. Sometimes, this may even start as soon as design, specially when it is unclear the properties that such API should have and some "exploratory programming" is needed. This fact may or many not be documented at the beginning of the section — even though an API may be designed through an inductive empirical process, a deductive rational description may be more useful for its clear understanding.

2.5.2.3 *The verification phase*

In the verification phase we perform *unit tests* on the most important parts of the system. No iteration should be considered finished unless proper unit tests are written and

satisfied for its core components. For writing such tests the Boost Unit Testing Framework should be used.

When some elements are considered relevant to performance requirements, *performance tests* should be included. While we do not enforce a specific performance testing technique here, the tests should be reproducible and automatable whenever possible.

2.5.2.4 *The integration phase*

When a subsystem is added and it is to replace an existing subsystem in the project, the older code should be removed and the layers on top modified such that they use the new code. This might even be considered part of the verification, as older tests working on the upper layers should be checked to be working still.

Informal integration tests should be done on the final user interface to make sure that the older properties are preserved. Note that in most cases, we do not recommend to lose time editing the old user interface such that the new features in the framework are exposed to the user. Of course, that the new features are usable is the final objective, but as it was justified in 1.2.1, a completely new user interface will be developed as part of a future project.

2.5.2.5 *Recursive decomposition of iterations*

In practice, some of the expected requirements to be satisfied may be found orthogonal or maybe too big to be addressed at one. It is thus allowed to recursively decompose an iteration in sub-iterations when a first evaluation during the design phase suggests that.

2.5.3 *A project plan*

2.5.3.1 *First iteration: A metaprogramming based sound processing foundational library*

This iteration is here to deeply re-design the core data structures using the latest techniques in C++. This requires special research and performance requirements deeply rely on the success of this iteration.

Requirements 1 to 3 and 26 to 27 should be satisfied for its success.

Etimated time cost: 6 weeks.

2.5.3.2 *Second iteration: Redesign of the `node` layer for hierarchy and polyphony*

The `node` layer requires a redesign if we want to satisfy all our purposes. Polyphony and hierarchy would be specially tricky to implement directly on top of the current code base. A special evaluation of how the new design interacts with the MVC architecture and networking is required. The `world` layer may be affected too. All the design changes should be implemented too.

Requirements 4 to 11 should be satisfied for its success. Requirement 23 may be considered for its implementation in this iteration too.

Estimated time cost: 6 weeks.

2.5.3.3 *Third iteration: Dynamic loading of nodes*

In this iteration the plugin system is to be developed.

Requirements 12 to 15 should be satisfied for its success.

Estimated time cost: 4 weeks.

2.5.3.4 *Fourth iteration: Adding MIDI and synchronisation*

Synchronisation and MIDI support is one of the most important features and it is also one of the features we know the least about, thus, we should put special care on research and design. This will affect the node and world layers mostly.

Requirements 17 to 22 should be satisfied for its success.

Estimated time cost: 8 weeks.

2.5.3.5 *Post mortem analysis*

After the conclusion of all the previous iterations, we should write a conclusive report and evaluation of its success. Also, we should prepare a final presentation for its evaluation.

3

A GENERIC SOUND PROCESSING LIBRARY

Numbers it is. All music when you come to think. Two multiplied by two divided by half is twice one. Vibrations: chords those are. One plus two plus six is seven. Do anything you like with figures juggling. Always find out this equal to that. Symmetry under a cemetery wall. He doesn't see my mourning. Callous: all for his own gut. Musemathematics. And you think you're listening to the etherial. But suppose you said it like: Martha, seven times nine minus x is thirtyfive thousand. Fall quite flat. It's on account of the sounds it is.

Ulysses
James Joyce

3.1 ANALYSIS

Requirements 1 to 3 and 26 to 27 refer to the second layer of our system — in a bottom-up approach. The most crucial question here: how do we represent a sound in a computer? Then, a new question arises: how do we get the sound to the speakers? The later question has a trivial answer — use whatever API your operating system exposes for delivering

sound to the soundcard — but the first question is still to be answered. Actually, the solution to this first question mostly subsumes the issue of how to interface with these external interfaces, thus, shall we debate it with care.

Añadir más dibujitos.

3.1.1 *Factors of sound representation*

A sound signal is a longitudinal wave of air in motion. We can analogically record the *proximal stimuli* — i.e. the physical stimuli leading to the subjective act of perception [36] — sound by measuring the successive oscillating values of air pressure in the observer's point in space. Note that an static air pressure value can not be perceived and the sensation of sound is caused by the relative oscillation of this measure. The *amplitude* of this change is associated to our perception of *loudness*, the frequency of this oscillation mostly logarithmically determines our perception of pitch. We phrased this conditionally because these two variables are actually interrelated and our actual subjective perception of loudness might vary with pitch and otherwise [37].

Most of the time, we represent the relative air pressure value as a voltage, that varies at some range — e.g. $[-5, 5]V$. This is an analogous signal that we have to discretise somehow in order to manipulate it computationally.

3.1.1.1 *Temporal quantisation*

Temporal quantisation relates to how many times per second do we store the current voltage or air pressure value. The

value of the signal between to equally spaced in time samples is unknown, but we can use some interpolation method to *upsample* a signal — i.e. to figure out what is between two samples. Most of the time we refer to the *sampling rate*, in hertz, as the frequency of the temporal quantisation.

We know from *Niquist-Shannon sampling theorem* that perfect reconstruction of a signal is possible when the sampling frequency is greater than twice the maximum frequency of the signal being sampled, or equivalently, when the *Nyquist frequency* (half the sample rate) exceeds the highest frequency of the signal being sampled. Because the hearing range in most human beings is 20 Hz–20 kHz, audio compact discs use a 44.1 kHz sampling rate. Other popular rates in audio production are 48 kHz, 96 kHz and 192 kHz. Sampling rates bellow 44.1 kHz are used also in old computer games that were limited by the computing power and low bandwidth systems such as telephone, where low cost and proper understanding of human speech is more important than audio fidelity.

The sound representation mechanism itself does not vary with the sampling rate, and thus supporting various rates depends more on the implementation of the signal processing units and the overall performance of the system, with the CPU being able to process so many samples per second being the biggest constraint.

3.1.1.2 *Spatial quantisation*

Spatial quantisation determines how many possible values can a sample take in our finite and discrete scale. In a computer system, this is determined by the size in bits of the underlying type used to store the samples. An audio CD

uses a *bitdepth* of 16 bit with samples that can take 65.536 possible values while professional audio uses 24 bit or 32 bit samples. We can even see systems using 64 bit samples during the processing to avoid accumulative rounding problems due to heavy arithmetic. The *dynamic range* of a signal with Q-bits quantisation is:

$$\mathrm{DR}_{\mathrm{ADC}} = 20 \times \log_{10}(2^Q) = (6.02 \cdot Q)\,\mathrm{dB} \tag{3.1}$$

The maximum *signal-to-noise* ratio for such a system is:

$$\mathrm{SNR}_{\mathrm{ADC}} = (1.76 + 6.02 \cdot Q)\ \mathrm{dB} \tag{3.2}$$

Most analog systems are said to have a dynamic range of around 80 dB [38]. Digital audio CD have a theoretical dynamic range of 96 db — actual value is around 90 db due to processing. Human hearing pain level is at 135 db but actually prolonged exposure to such loud sound can cause damage. A loud rock concert is around 120 dB and a classical music performance is at 110 db [39], thus requiring bitdepth of at least 24 bit (theoretical dynamic range of 144 dB) for perfect fidelity.

There are some other aspects related to representation of samples in a computer, such as the *signedness* of the underlying type. Signed types are usually considered more convenient for audio signals as 0 can be easily recognised as the still no-sound value simplifying computations. Another important factor is whether we use *fixed point* or *floating point* arithmetic. While fixed point is used in low-cost DSP hardware, floating point is the most common representation in current audio software as nowadays processors are optimised for SIMD[1] floating point arithmetic. Moreover,

1 Single Instruction Multimple Data, as supported by MMX, 3D Now! and SSE extensions in Intel and AMD chips

the algorithms implementation is much harder to encode because products produce greater values and there are many ways on how to account the carry. Actually, even while the actual bitdepth (the bit for the mantissa) of a 32 bit floating point is the same of a 24 bit fixed point, then a 32 bit fixed point will have a quite lower *quantization error*, but the dynamic range and SNR of a floating point is much higher because the values are spaced logarithmically over a huge range [40]. Another factor is the *endianess* of fixed point values but is relevant only when interfacing with file formats and output devices.

3.1.1.3 *Channel space*

Because our hearing system is dicotomically symmetric, audio engineers discovered that much better fidelity can be achieved by reproducing the sound with some differences from two separate loudspeakers. This is the well-known *stereophonic* sound, commonly named just *stereo*.

For representing such a signal, two different streams of information are needed for the left and right channels. Moreover, nowdays *quadraphonic*, and *surround* sound with varying numbers of channels up to 20.2 are used in different systems.

We call *channel space* to the set of semantic channels we use in some sort of audio representation — e.g. stereo audio has channel space with $left$ and *right* elements. We use the term *frame* to call a set of samples coincident in time, this is, the samples of the various channels at a given time point. Thus, we will use most of the time the more accurate term *frame rate*.

This rises the problem on how to linearise the multi channel data. The most common mechanism in domestic hardware is by *interleaving* the samples of different channels, this is, by storing the frames sequentially. However, high-end hardware often accepts data in non-interleaved form where the samples of each channel is stored in a separate sequence. In this document, we borrow from the image processing world the term *planar* to refer to non-interleaved data. Software doing a lot of processing of the audio signal often chooses this representation as it is easier to scale to varying number of channels and split the signal to do per-channel filtering. Figure 3.1 compares visually the interleaved and planar formats.

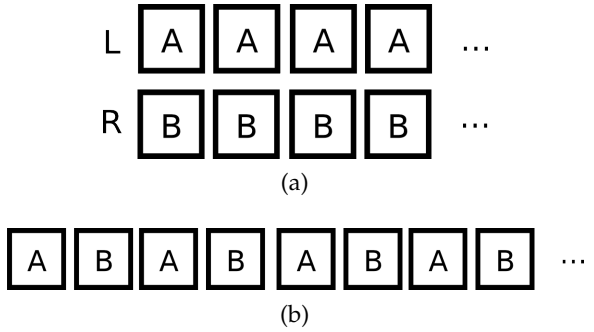


Figure 3.1.: Multi-channal data in planar (a) and interleaved (b) form.

Another issue is the order in which the data from different semantic channels is stored. We call a *channel layout* a bijection $L : C \to \mathbb{Z}_{\|C\|}$, where C is a given *channel space*. For example, the mapping $\{left \mapsto 0, right \mapsto 1\}$ is a common layout for stereo sound, but $\{left \mapsto 1, right \mapsto 0\}$ is sometimes used too.

3.1.2 *Common solutions*

As we have already noticed, 32 bit floating point sound with planar left-right layout the most common in software of our kind during internal processing. As most of this software is written in C, a simple `float**` does the job. This was, actually, the internal representation used in GNU Psychosynth in versions prior to 0.2.0, wrapped in the `audio_buffer` class.

However, this design starts to wobble whenever one has to interface with some other library or hardware using a different format. Thus, the `audio_buffer` class provided different `interleave_*` and `deinterleave_*`, where the asterisk can be substituted by different sample formats like `s16` or `s32` (fixed point signed 16 bit and 32 bit respectively). This is very inconvenient because, as we have seen through this section, many orthogonal factor affect audio representation inducing a combinatorial explosion of format conversion functions. Take a look at the 64 different read and write functions in the `pcm.c` file of the LibSndfile[2] library.

This is a maintenance hell, but using the common means for abstracting orthogonal behaviour variability, i.e. dynamic polymorphism, is simply not an option in any audio software which supports real-time operation.

3.1.3 *A generic approach: Boost.GIL*

However, there is a piece of software that proved that this issue can be solved in C++ using static polymorphism. This is the Generic Image Library[3] which was developed by

2 `http://www.mega-nerd.com/libsndfile/`
3 http://stlab.adobe.com/gil

Lubomir Bourdev et. Al inside Adobe Software Technology Lab that was later include inside the Boost library distribution.

While sound and image manipulation are quite different, specially from the psycho-perceptive point of view, they are both a signal processing problem and thus share a lot in the representational issue. By realising of a proper conceptual mapping between both worlds (table 3.1), most of the library design and even quite a lot of code of Boost.GIL can be reused to build a unique state-of-the-art sound processing library that addresses the aforementioned issues in an orthogonal generic manner while maintaining near-optimal performance.

Boost.GIL	Psynth.Sound
Channel	Sample
Color	Channel
Color Space	Channel Space
Color Layout	Channel Layout
Pixel	Frame
View	Range
Image	Buffer

Table 3.1.: Terminology map from `boost::gil` to `psynth::sound`

An *image* is bidimensional matrix of *pixels*, that capture the properties of light electromagnetic waveform at those discrete points. Each pixel, however, is decomposed in several *colors* that, for example, capture the intensity in the red, green and blue sensors of a CCD camera. As there are different ways of decomposing an audio frame (e.g, stereo, surround, etc.), there are different ways of decomposing a

pixel into several values, known as the *color space* (e.g, RGB, CMYK, YUV, etc.). Boost.GIL uses the term *channel* to name the individual value of one those color components.

In our audio framework, a *buffer* is unidimensional array of *frames* that represent a sound or part of a sound — sound is continuous and thus we usually process it in chunks. The reader might note that the the data in a buffer being arranged along the *time* dimension while the dimensions of an image represent *physical space* makes these entities completely different from the processing point of view. However, they share most representation problems, with sound representation being actually a sub-problem of image representation, as we have one dimension less. The samples in a series of audio frames can be stored in an interleaved or planar fashion as happens with the channels of a pixel. Also, both channels and samples can vary in signedness, fixed/floating point, bitdepth, etc.

Those already familiar with Boost.GIL can thus already understand easily our Psynth.Sound module design and implementation that we are to describe in the following section.

3.2 DESIGN

This section describes the design of the modules implemented in this layer. We will first introduce some advanced techniques used in the design that the reader might be unfamiliar with. Then, sections 3.2.2 to 3.2.5 will describe the classes in the `psynth::sound` namespace, section 3.2.6 will be dedicated to the `psynth::io` namespace and finally section 3.2.7 to the `psynth::synth` namespace. `psynth::io`

and `psynth::synth` are independent from eachother and both depend on `psynth::sound`. Still, from the architectural point of view they are all considered to be on the same layer — the "synth" layer dedicated to mostly statically arranged sound synthesis and processing.

3.2.1 *Core techniques*

The Boost.GIL and thus the Psynth.Sound modules design makes heavy use of static polymorphism and generic programming via C++ templates to achieve generality without runtime overhead. We are going to introduce advanced techniques used in generic programming for the reader unfamiliar with this programming paradigm.

3.2.1.1 *Concepts*

.

Concepts [41] are to generic programming what *interfaces* — pure abstract classes in C++ — are to object oriented programming: they specify the requirements on some type. However there, are few substantial differences. (1) While interfaces can only specify the method signatures of its instances, a concept can specify most syntactic constraints on a type, like the existence of free functions, operators, nested types, etc. (2) While dispatching through interfaces requires, at least, a dereference, addition and function call [42], when using concepts the concrete function to be executed can be determined and even inlined at compile-time. (3) One can not declare that a type satisfies an interface separately from the type definition, but one can say that a type models a

concept at any point of the program. (4) Thus, no primitive type defines any virtual interface, but one can turn any primitive type into an instance of any concept via a `concept_map`. (5) Actually, the syntactic properties defined by a concept its models may differ, but they are matched via the `concept_map`. In fact, C++ concepts are more similar to Haskell *type classes*, with `instace` doing the job of `concept_map` [43].

Concepts are an extension to the template mechanism to add type checking for it. In fact, checking and dispatching on requirements can be achieved with techniques like SFINAE (Substitution Failure Is Not an Error) [44]. Property (5) of our concepts can be simulated with *traits* [45]. However, both compiler errors and the code using templates without concepts is usually much more unreadable.

The proposal of adding concepts to the C++ language was rejected last year by the standardisation committee and thus we can not use them in our code. However, Boost.GIL is very influeced by Alexander Stepanov's deductive approach to computer programming using generic programming and modeling with concepts, that he elegantly describes in his master-piece "Elements of Programming" [46]. Actually Stepanov worked several years in Adobe where he held a course "Foundations of Programming" based on his book. Thus, the *modeling* of the library extensively uses concepts. Its implementation uses a limited form of concept checking via the Boost.ConceptCheck[4] [47] library, however, enabling this library in release mode can affect performance and its syntax is quite more cumbersome than the concepts in the C++ standard proposal. For consistency with the Boost.GIL

4 Boost.ConceptCheck: `http://www.boost.org/doc/libs/release/libs/concept_check/concept_check.htm`

documentation we will use the concept syntax proposed in the proposal N2081 to the standardisation committee [48].

The following example defines a concept that is satisfied by every type that has an `operator<`:

```
concept LessThanComparable<typename T> {
  bool operator< (T, T);
};
```

This allows us to write a generic function that depends on the existence of a less-than comparator for the parametrised type:

```
template<LessThanComparable T>
const T& min (const T& x, const T& y) {
  return x < y? x : y;
}
```

An alternative syntax for specifying that `T` must satisify the `LessThanComparable` concept is the `where` clause:

```
template<typename T>
    where LessThanComparable<T>
const T& min (const T& x, const T& y) ...
```

In fact, this is the only valid syntax when the concept affects multiple types. Also, the `where` clause can be used inside concept definitions to provide specialisation.

Specifying that a type models a concept is done with the `concept_map` device. If the type naturally models the concept, we can just use:

```
concept_map LessThanComparable<int> {}
```

Note that these trivial concept mappings can be avoided by using the `auto` keyword in front of the `concept` keyword in the concept definition. However, it might happen that a

type requires some wrapping to satisfy the concept. We can do this in the concept map definition itself.

```
concept_map LessThanComparable<char*> {
  bool operator< (char* a, char* b) {
    return strcmp (a, b) < 0;
  }
}
```

Note that this last piece of code is an example of a bad usage of concept maps, as this specialises the mapping for pointers changing the expected semantics.

This should suffice as an introduction to concepts in order to understand the concept definitions that we will later show when modelling our system. A more detailed view can be read in the cited bibliography, with [41] being the most updated and useful from a programmer point of view.

3.2.1.2 *Metaprogramming*

The C++ template system is Turing complete [49], thus it can be used to perform any computation at *compile time*. This was first noted in 1994 by Erwin Unruh who, in the middle of a C++ standardisation committee, wrote a template metaprogram that outputted the first N prime numbers on the console using compiler errors [50]. Even though this might seem just a crazy puzzle game, it can be used in practise and actually new Boost libraries use it extensively. A very gentle introduction to template metaprogramming can be found in [33], where Alexandrescu uses them to instantiate design patterns as generic C++ libraries. A deeper reference is Abraham's [51], which focuses on the Boost Metaprogramming

Library[5] and introduces the usage of metaprogramming for building Embedded Domain Specific Languages (EDSL) in C++. This Boost.MPL, providing reusable meta data structures and algorithms, is the de-facto standard library for template metaprogramming[6] and we will use it in our implementation.

Template metaprogramming is possible thanks to *partial template specialisation*, that allows giving an alternate definition for a pattern matched subset of its possible parameter values. A *metafunction* is thus just a template `class` or `struct` with a public member that holds the result of the function. It is up to the programmer to choose the naming convention for the result members of the metafunctions, in the following, we will use Abraham's style calling `type` for result values that are a type, and `value` for integral values. Listing 3.1 illustrates how can we write and use a metafunction for computing the n-th Fibonacci number.

The program returns the forty-second Fibonacci value. However, it will take no time to execute, because the number is computed at compile time. We use recursion to define the metafunction for the general case and the specialise for the base cases.

If we consider the template system as a meta-language on its own, we should describe its most outstanding semantic properties. It is a pure functional programming language, because variables are immutable. It is lexically scoped. It supports both lazy and strict evaluation, depending on whether we choose to access the nested `type` result name at call site or value usage type. When we look at the meta type system, we find three meta types: types (which

5 The Boost.MPL: www.boost.org/doc/libs/release/libs/mpl
6 It is often called "the STL of template metaprogramming".

Listing 3.1: Metaprogram for computing the Nth Fibonacci number

```
template <int N>
struct fib {
  enum {
    value = fib<N-1>::value + fib<N-2>::value;
  };
};

template <>
struct fib <0> {
  enum { value = 0 };
};

template <>
struct fib <1> {
  enum { value = 1 };
};

int main () {
  return fib<42>::value;
}
```

are duck-typed records), integrals (e.g. `int`, `char`, `bool` ...) and meta-functions (i.e. templates).

The fact that records are duck typed but integrals and metafunctions cause several inconveniences in practice, specially when dealing with the later. For example, in the absence of template aliases, returning a metafunction produced by another function requires defining a nested struct that inherits from the actual synthesised value. Also, the template signature should be specified on a template parameter expecting a template.

Listing 3.2: Integral constant nullary metafunction wrapper.

```
template <typename T, T V>
struct integral_c
{
  BOOST_STATIC_CONSTANT(T, value = V);
  typedef integral_c<T, V> type;
};
```

In order to simplify our meta type system we shall wrap constants in a type like on listing 3.2.

There are a couple of issues regarding this definition worth explaining. First, the `BOOST_STATIC_CONSTANT` macro is used to define a constant. Internally, it will try to use `enum` or any other mechanism available to actually define the constant such that the compiler is not tempted to allocate static memory for the constant. Second, the `typedef` referring to itself turns a constant value into a self returning nullary meta-function. This can be very convenient because, for example, it allows using *value*`::type::value` always on the value usage point, allowing the caller or producer of the value to choose whether he wants to evaluate the value lazily.

Because we just wrapped values into a type, we can simplify our conventional definition of *metafunction*: a metafunction is any type — template or not — that has a nested type called `type`.

Now we should also turn metafunctions into first class entities of the meta-language. We just add a new level of indirection and define a *metafunction class* as a type with a nested template metafunction called `apply`. The example in listing 3.3 also illustrates the metafunction forwarding

Listing 3.3: Metafunction class for computing Fibonacci numbers. We suppose that the previous `fib` definition uses `integral_c` to wrap its parameters and return types.

```
struct fib_class {
    template <class N>
    struct apply : public fib<N> {};
};

int main ()
{
  return fib_class::apply<integral_c<int, 42>>::type::value;
}
```

technique when defining the nested `apply` metafunction by inheriting from `fib`.

Using this convention the MPL library defines many useful high order metafunctions that take metafunction classes as input, like `mpl::fold` and `mpl::transform`. Note that it is not needed to define metafunction classes for all our metafunctions, instead, we shall convert them when needed using the `mpl::quote`*N* functions and the `mpl::lambda` facility.

3.2.2 *Core concepts*

We are now ready to understand the main design and implementation techniques used in our generic library. Because the library is *generic*, in the sense of generic programming, most algorithms and data structures are parametrised such that they can be instantiated with any concrete type mod-

elling some concepts as we suggested in section 3.2.1.1. Thus, traditional modelling techniques like the Unified Modelling Language are not useful since they are intended for object oriented design.

We are going to follow the following methodology for describing the library. First, we will name a concept and give a brief description of its purpose. Then, we will define the concept using the notation described in section 3.2.1.1 and finally we will enumerate and describe some models for such concept.

For brevity, we will omit basic concepts such as `CopyConstructible`, `Regular`, `Metafunction`, etc. Their complete definition should be evident and an interested reader can find most of them in [46].

3.2.2.1 *ChannelSpaceConcept*

A channel space is a MPL sequence (like `mpl::list`) of whose elements channel tags (empty types giving a name for the semantic channel).

```
concept ChannelSpaceConcept<MPLRandomAccessSequence Cs>
{};
```

Some example models include `stereo_space` or `surround_space`. An example on how a user of the library can define his own channel space follows.

```
struct left_channel {};
struct right_channel {};

typedef mpl::vector<left_channel, right_channel> stereo_space;
```

A related trivial concept is `ChannelSpaceCompatibleConcept`. Two channel spaces are compatible if they are the

same. In fact, this leaks the underlying MPL sequence type used in the channel space through the abstraction, because spaces with the same set of semantic channels might be found incompatible, but it suffices in practise.

3.2.2.2 *SampleMappingConcept*

An MPL Random Access Sequence, whose elements model `MPLIntegralConstant` (like `mpl::int_`) representing a permutation of the channels in the channel space, thus specifying the layout.

```
concept SampleMappingConcept<MPLRandomAccessSequence CM> {
};
```

The *layout* of a frame based type is a channel space plus a sample mapping, as defined by:

```
template <typename ChannelSpace,
          typename SampleMapping = boost::mpl::range_c<
              int, 0, boost::mpl::size<ChannelSpace>::value> >
struct layout
{
    typedef ChannelSpace channel_space;
    typedef SampleMapping sample_mapping;
};
```

The sample mapping is usually directly defined in the layout, if needed at all — the default sample mapping is the normal order in the channel space. For example, the reversed stereo layout is defined as:

```
typedef layout<stereo_space, mpl::vector2_c<int, 1, 0>> rlstereo_layout;
```

3.2.2.3 *SampleConcept*

A *sample* is the type we use to represent the amplitude of a channel at certain point in time.

```
concept SampleConcept<typename T> :
            EqualityComparable<T> {
    typename value_type = T;
    // use sample_traits<T>::value_type to access it
    typename reference = T&;
    // use sample_traits<T>::reference to access it
    typename pointer = T*;
    // use sample_traits<T>::pointer to access it
    typename const_reference = const T&;
    // use sample_traits<T>::const_reference to access it
    typename const_pointer = const T*;
    // use sample_traits<T>::const_pointer to access it
    static const bool is_mutable;
    // use sample_traits<T>::is_mutable to access it

    static T min_value();
    // use sample_traits<T>::min_value to access it
    static T zero_value();
    // use sample_traits<T>::zero_value to access it
    static T max_value();
    // use sample_traits<T>::min_value to access it
};
```

Built-in scalar types like `char`, `int` or `float` model `SampleConcept` by default.

The `scoped_sample_value<Type, Min, Max, Zero>` template class models the concept whenever `Type` is a scalar type and `Min`, `Zero` and `Max` satisfy:

$$Min < Zero < Max \wedge \forall x \in Type, x + Zero = x \qquad (3.3)$$

Note that, in order to avoid the limitation of floating point values not being usable as template arguments, `Min`, `Zero` and `Max` should be a type with a static method `apply()` that returns the actual value. It should be used to constraint the "clipping thresholds" of floating point types. For example, the `bits32sf` model defined as:

```
typedef scoped_sample<float,
                      float_minus_one,
                      float_zero,
                      float_one> bits32sf;
```

User defined types should specialise `sample_traits` to map the concept.

Related trivial concepts are `MutableSampleConcept` and `SampleValueConcept` (a sample that is also `Regular`).

3.2.2.4 *SampleConvertibleConcept*

Because casting does not suffice in most cases, one should override a `T sample_convert (U)` function for `U` to be convertible into `T`.

```
concept SampleConvertibleConcept<SampleConcept SrcSample,
     SampleValueConcept DstSample> {
    DstSample sample_convert (const SrcSample&);
};
```

The library provides overrides for `sample_convert` making most supplied sample types being convertible too.

3.2.2.5 *ChannelBaseConcept*

A *channel base* is a container of channel elements (such as samples, sample references or sample pointers).

The most common use of channel base is in the implementation of a frame, in which case the channel elements are sample values. The channel base concept, however, can be used in other scenarios. For example, a planar frame has samples that are not contiguous in memory. Its reference is a proxy class that uses a channel base whose elements are sample references. Its iterator uses a channel base whose elements are sample iterators.

```
concept ChannelBaseConcept<typename T> :
      CopyConstructible<T>, EqualityComparable<T>
{
    // a Psynth layout (the channel space and element permutation)
    typename layout;

    // The type of K−th element
    template <int K> struct kth_element_type;
    where Metafunction<kth_element_type>;

    // The result of at_c
    template <int K>
    struct kth_element_const_reference_type;
    where Metafunction<
            kth_element_const_reference_type>;

    template <int K>
    kth_element_const_reference_type<T,K>::type at_c(T);

    // Copy−constructible and equality comparable
    // with other compatible channel bases
    template <ChannelBaseConcept T2>
        where { ChannelBasesCompatibleConcept<T,T2> }
        T::T(T2);
    template <ChannelBaseConcept T2>
        where { ChannelBasesCompatibleConcept<T,T2> }
        bool operator==(const T&, const T2&);
    template <ChannelBaseConcept T2>
```

```
        where { ChannelBasesCompatibleConcept<T,T2> }
        bool operator!=(const T&, const T2&);
};
```

A channel base must have an associated layout (which consists of a channel space, as well as an ordering of the samples). There are two ways to index the elements of a channel base: A physical index corresponds to the way they are ordered in memory, and a semantic index corresponds to the way the elements are ordered in their channel space. For example, in the stereo channel space the elements are ordered as {`left_channel, right_channel`}. For a channel base with a RL-stereo layout, the first element in physical ordering is the right element, whereas the first semantic element is the left one. Models of `ChannelBaseConcept` are required to provide the `at_c<K>(ChannelBase)` function, which allows for accessing the elements based on their physical order. Psynth provides a `semantic_at_c<K>(ChannelBase)` function (described later) which can operate on any model of `ChannelBaseConcept` and returns the corresponding semantic element.

A related concept is `MutableChannelBaseConcept` and `ChannelBaseValueConcept` with the excepted definition. There is also the concept `ChannelBasesCompatibleConcept`. Two channel bases are compatible if they have the same channel space and their elements are compatible, semantic-pairwise.

3.2.2.6 *HomogeneousChannelBaseConcept*

Channel base whose elements all have the same type.

```
concept HomogeneousChannelBaseConcept<
            ChannelBaseConcept CB>
{
```

```
    // For all K in [0 ... size<C1>::value−1):
    // where SameType<kth_element_type<CB,K>::type,
    // kth_element_type<CB,K+1>::type>;
    kth_element_const_reference_type<CB,0>::type dynamic_at_c(
            const CB&, std::size_t n) const;
};
```

Related concepts `MutableHomogeneousChannelBaseConcept` and `HomogeneousChannelBaseValueConcept` have the expected definition.

The library provides an `homogeneous_channel_base` class that models the concept.

3.2.2.7 *FrameBasedConcept*

Concept for all frame based constructs, such as frames themselves, iterators, ranges and buffers whose value type is a frame. A `FrameBased` type povides some metafunctions for accessing the underlying channel space, sample mapping and whether the frame representation is planar or interleaved.

```
concept FrameBasedConcept<typename T> {
    typename channel_space_type<T>;
        where Metafunction<channel_space_type<T> >;
        where ChannelSpaceConcept<channel_space_type<T>::type>;
    typename sample_mapping_type<T>;
        where Metafunction<sample_mapping_type<T> >;
        where SampleMappingConcept<sample_mapping_type<T>::type>;
    typename is_planar<T>;
        where Metafunction<is_planar<T> >;
        where SameType<is_planar<T>::type, bool>;
};
```

There are many models for this in the library, like `buffer`, `buffer_range`, `frame`, `bitaligned_frame_iterator`, `bitaligned_frame_reference`, `packed_frame` and so on.

3.2.2.8 *HomogeneousFrameBasedConcept*

Concept for homogeneous frame-based constructs — frame based with the same sample type for all its channels. These should allow access to the underlying sample type.

```
concept HomogeneousFrameBasedConcept<FrameBasedConcept T> {
    typename sample_type<T>;
        where Metafunction<sample_type<T> >;
        where SampleConcept<sample_type<T>::type>;
};
```

Most container and alike constructs in the library — iteators, buffers, ranges — model this whenever the underlying frame is homogeneous.

3.2.2.9 *FrameConcept*

A set of samples coincident in time, one per channel in the given channel space.

```
concept FrameConcept<typename F> :
            ChannelBaseConcept<F>, FrameBasedConcept<F> {
    where is_frame<F>::type::value==true;
    // where for each K [0..size<F>::value−1]:
    // SampleConcept<kth_element_type<F,K> >;

    typename F::value_type;
        where FrameValueConcept<value_type>;
    typename F::reference;
        where FrameConcept<reference>;
    typename F::const_reference;
```

```
        where FrameConcept<const_reference>;
    static const bool F::is_mutable;

    template <FrameConcept F2> where { FrameConcept<F,F2> }
        F::F(F2);
    template <FrameConcept F2> where { FrameConcept<F,F2> }
        bool operator==(const F&, const F2&);
    template <FrameConcept F2> where { FrameConcept<F,F2> }
        bool operator!=(const F&, const F2&);
};
```

Related concepts are `MutableFrameConcept` and `FrameValueConcept`, defined as usually.

Frame compatibility should be tested with the `FramesCompatibleConcept`. Frames are compatible if their samples and channel space types are compatible. Compatible frames can be assigned and copy constructed from one another.

Provided models include `frame`, `packed_frame`, `planar_frame_reference` and `bit_aligned_frame_reference`.

3.2.2.10 *HomogeneousFrameConcept*

A frame with all samples of the same type should also provide an indexed access operator.

```
concept HomogeneousFrameConcept<FrameConcept P> :
            HomogeneousChannelBaseConcept<P>,
            HomogeneousFrameBasedConcept<P>
{
    P::template element_const_reference_type<P>::type
    operator[] (P p, std::size_t i) const
    { return dynamic_at_c(p,i); }
};
```

Related concepts are `MutableHomogeneousFrameConcept` and `HomogeneousFrameValueConcept`, defined as usually.

Provided models include `planar_frame_reference` and `frame`.

3.2.2.11 *FrameConvertibleConcept*

A frame type is convertible to another frame type if there exist a `channel_convert` overload. Convertibility is non-symmetric and implies that one frame can be converted to another, approximating the value. Conversion is explicit and sometimes lossy.

```
template <FrameConcept SrcFrame, MutableFrameConcept DstFrame>
concept FrameConvertibleConcept {
    void channel_convert(const SrcFrame&, DstFrame&);
};
```

Frame types provided by the library are convertible.

3.2.2.12 *FrameDereferenceAdaptorConcept*

Represents a unary function object that can be invoked upon dereferencing a frame iterator. This can perform an arbitrary computation, such as channel conversion or table lookup.

```
concept FrameDereferenceAdaptorConcept<
            boost::UnaryFunctionConcept D>
        : DefaultConstructibleConcept<D>
        , base::CopyConstructibleConcept<D>
        , AssignableConcept<D>
{
    typename const_type;
        where FrameDereferenceAdaptorConcept<const_t>;
    typename value_type;
```

```
        where FrameValueConcept<value_type>;
    typename reference; // may be mutable
    typename const_reference; // must not be mutable
    static const bool D::is_mutable;

    where Convertible<value_type,result_type>;
};
```

The `channel_convert_deref_fn` provides a model that performs channel conversion.

3.2.2.13 *FrameIteratorConcept*

An STL random access traversal iterator over a model of `FrameConcept`. Our iterators must provide some extra metafunctions.

```
concept FrameIteratorConcept<typename Iterator>
        : boost_concepts::RandomAccessTraversalConcept<Iterator>
        , FrameBasedConcept<Iterator>
{
    where FrameValueConcept<value_type>;
    typename const_iterator_type<It>::type;
        where FrameIteratorConcept<const_iterator_type<It>::type>;
    static const bool iterator_is_mutable<It>::type::value;
    static const bool is_iterator_adaptor<It>::type::value;
    // is it an iterator adaptor
};
```

Related concepts include `MutableFrameIteratorConcept`, defined as usually. The related `HasDynamicStepTypeConcept` is modelled by those iterator types with an overload for the `dynamic_step_type` metafunction returning a similar iterator that models `StepIteratorConcept`.

Models include `T*` where `T` is a frame or `bitaligned_frame_iterator`, `memory_based_step_iterator` and `planar_frame_iterator`.

3.2.2.14 *MemoryBasedIteratorConcept*

Iterator that advances by a specified step. Concept of a random-access iterator that can be advanced in memory units (bytes or bits).

```
concept MemoryBasedIteratorConcept<
          boost_concepts::RandomAccessTraversalConcept Iterator>
{
    typename byte_to_memunit<Iterator>;
    where metafunction<byte_to_memunit<Iterator> >;
    std::ptrdiff_t memunit_step(const Iterator&);
    std::ptrdiff_t memunit_distance(const Iterator& , const Iterator&);
    void memunit_advance(Iterator&, std::ptrdiff_t diff);
    Iterator memunit_advanced(const Iterator& p,
    std::ptrdiff_t diff)
    { Iterator tmp; memunit_advance(tmp,diff); return tmp; }
    Iterator::reference memunit_advanced_ref(
    const Iterator& p, std::ptrdiff_t diff)
    { return *memunit_advanced(p,diff); }
};
```

Iterators defined by our library are memory based.

3.2.2.15 *StepIteratorConcept*

Step iterators are iterators that have can be set a step that skips elements.

```
concept StepIteratorConcept<
        boost_concepts::ForwardTraversalConcept Iterator> {
    template <Integral D> void Iterator::set_step(D step);
```

```
};
```

A related `MutableStepIteratorConcept` is defined as expected. The class `memory_based_step_iterator` models the concept.

3.2.2.16 *FrameIteratorConceptIteratorAdaptorConcept*

Iterator adaptor is a forward iterator adapting another forward iterator.

```
concept IteratorAdaptorConcept<
            boost_concepts::ForwardTraversalConcept Iterator>
{
    where SameType<is_iterator_adaptor<Iterator>::type,
                   boost::mpl::true_>;

    typename iterator_adaptor_get_base<Iterator>;
        where Metafunction<iterator_adaptor_get_base<Iterator> >;
        where boost_concepts::ForwardTraversalConcept<
                  iterator_adaptor_get_base<Iterator>::type>;

    typename another_iterator;
    typename iterator_adaptor_rebind<Iterator,another_iterator>::type;
        where boost_concepts::ForwardTraversalConcept<another_iterator>;
        where IteratorAdaptorConcept<iterator_adaptor_rebind<
                  Iterator,another_iterator>::type>;

    const iterator_adaptor_get_base<Iterator>::type&
    Iterator::base() const;
};
```

There exist a related `MutableIteratorAdaptorConcept` with the usual definition. Classes `dereference_iterator_adaptor` and `memory_based_step_iterator` do model the concept.

3.2.2.17 *RandomAccessBufferRangeConcept*

This is a range, similar to the new STL range defined in C++0x, but with some extra members for random access.

```
concept RandomAccessBufferRangeConcept<base::Regular Range>
{
    typename value_type;
    typename reference; // result of dereferencing
    typename difference_type;
    // result of operator−(iterator,iterator)
    typename const_type;
        where RandomAccessBufferRangeConcept<Range>;
    // same as Range, but over immutable values
    typename iterator;
        where RandomAccessTraversalConcept<iterator>;
    // iterator over all values
    typename reverse_iterator;
        where RandomAccessTraversalConcept<reverse_iterator>;
    typename size_type; // the return value of size()

    // Defines the type of a range similar to this type, except it
    // invokes Deref upon dereferencing
    template <FrameDereferenceAdaptorConcept Deref>
    struct add_deref {
        typename type;
        where RandomAccessBufferRangeConcept<type>;
        static type make(const Range& v, const Deref& deref);
    };

    Range::Range(const iterator&, const size_type&);

    // total number of elements
    size_type Range::size() const;
    reference operator[](Range, const difference_type&) const;

    iterator Range::begin() const;
```

```
    iterator Range::end() const;
    reverse_iterator Range::rbegin() const;
    reverse_iterator Range::rend() const;
    iterator Range::at(const size_type&);
};
```

There exists a `MutableRandomAccessBufferRange` with the usual semantics.

3.2.2.18 *BufferRangeConcept*

A random access range over frames, it has extra information to get the number of channels of the underlying frame type.

```
concept BufferRangeConcept<RandomAccessBufferRangeConcept Range>
{
    where FrameValueConcept<value_type>;
    std::size_t Range::num_channels() const;
};
```

There exists a related `MutableBufferRangeConcept` with the expected definition. Also, there is a `RangesAreCompatibleConcept`. Ranges are compatible if they have the same channel spaces and compatible sample values. Constness and layout are not important for compatibility.

The library provides the `buffer_range<Iterator>` model. There exists also a whole family of *range factories* that build a range or transform one kind of range into another. These allow to build a buffer range on top of raw data provided by an external library, lazily converting from once frame type to another, obtaining sub parts of a frame, etc.

3.2.2.19 *RandomAccessBufferConcept*

A container of values. The values are accessible via an associated range. Buffers are not ranges by themselves, because that generates boilerplate due to constness problems — a `const` range may give mutable access to its frames, but a `const` buffer does not, thus an algorithm that may mutate the frames but not the range would have to provide two overloads.

```
concept RandomAccessBufferConcept<typename Buf> :
            base::Regular<Buf> {
    typename range;
        where MutableRandomAccessBufferRangeConcept<range>;
    typename const_range = range::const_type;
    typename value_type = range::value_type;
    typename allocator_type;

    Buf::Buf(point_t dims, std::size_t alignment=1);
    Buf::Buf(point_t dims, value_type fill_value, std::size_t alignment);

    void Buf::recreate(point_t new_dims, std::size_t alignment=1);
    void Buf::recreate(point_t new_dims, value_type fill_value,
                       std::size_t alignment);

    const const_range& const_range(const Buf&);
    const range& range(Buf&);
};
```

3.2.2.20 *BufferConcept*

A buffer containing frames.

```
concept BufferConcept<RandomAccessBufferConcept Buf> {
    where MutableBufferRangeConcept<Buf::range>;
};
```

The `buffer` class models the concept.

3.2.2.21 *RandomAccessRingBufferRangeConcept*

Circular or ring buffers provide a virtual continuous flow of data where one can write at one end and read from the other. There ring buffer has a fixed size and the amount of available data plus free space in the structure remains constant during operation, as seen in figure 3.2. This data structure is specially important for passing audio data through different threads.

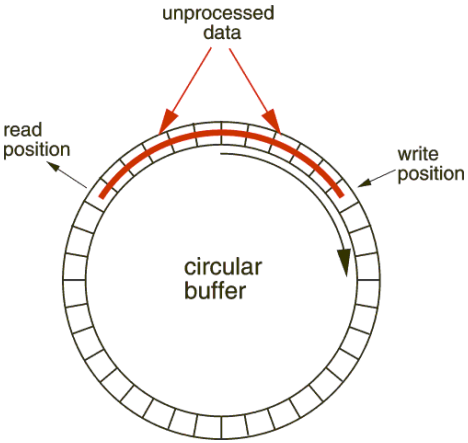


Figure 3.2.: Ring buffer operation.

Our ring buffers wrap and underlying buffer range. Note that each ring buffer range keeps its own write pointer, so using multiple ring buffer ranges on the same data might lead to subtle problems.

A este concepto le faltan restricciones

.

```
concept RandomAccessRingBufferRangeConcept<Regular R>
{
    typedef range; where RandomAccessBufferRange<range>;

    typename reference = range::reference;
    typename size_type = range::size_type;
    typename value_type = range::size_type;
    typename difference_type = range::difference_type;

    typename unsafe_position;
    typename position;

    typename unsafe_iterator;
    typename iterator;

    R (const range& range);
    R& R::operator= (const range& r);

    size_type R::size () const;
    unsafe_iterator begin_unsafe () const;
    unsafe_iterator end_unsafe () const;
    iterator begin () const;
    iterator end () const;
    reference operator [] (difference_type i) const;
    iterator at (difference_type i) const;
    unsafe_iterator unsafe_at (difference_type i) const;

    unsafe_position R::begin_unsafe_pos () const;
    unsafe_position R::end_unsafe_pos () const;
    position R::begin_pos () const;
    position R::end_pos () const;

    size_type R::available (const position& r) const;
    size_type R::available (const unsafe_position& r) const;
    size_type R::available () const;
    ring_buffer_error R::check_position (const position& reader) const;
```

```
    template <class Position> range
    R::sub_range_one (const Position& p, size_type slice) const;
    template <class Position> range
    R::sub_range_two (const Position& p, size_type slice) const;

    template<class Position, class Range2>
    size_t R::read (Position& r, const Range2& range) const;
    template<class Position, class Range2,
             class CC = default_channel_converter>
    size_t R::read_and_convert (Position& r, const Range2& range,
                                CC cc = CC ()) const;

    template <class Range2>
    void R::write (const Range2& range);
    template <class Range2, class CC = default_channel_converter>
    void write_and_convert (const Range2& range, CC cc = CC ());

    bool is_backwards () const;
    void set_backwards ();

    // Total of data ever writen to the buffer.
    difference_type count () const;

    // Fix iterators after using set_backwards.
    position sync (const position& r) const;
};
```

There are some things worth mentioning about this concept. It supports two different ways of manipulation. The STL alike iterator based interface is provided for compatibility, but one should try to avoid it for performance issues — it has to check for having to wrap around the end of the buffer on every iterator increment. Instead, data should be added in chunks as large as possible with the *position* based interface — the `read*` and `write*` functions. When this is not possible, sometimes the `sub_range_*` functions can provide

a solution, allowing to obtain the slices of the underlying range that represent a sub range of the ring buffer.

Tal vez podrían desacoplarse las dos formas de manipular en dos conceptos diferentes...

Also, note that the ring buffer supports backwards operation. This is used such that whenever we are reading a file and we eventually want to read it backwards, we can reuse the data already in the buffer.

Note that non mutable ring buffers do not make sense.

3.2.2.22 *RingBufferRangeConcept*

A ring buffer range of frames.

```
concept RingBufferRangeConcept<
            RandomAccessRingBufferRangeConcept Buf> {
  where MutableBufferRangeConcept<Buf::range>;
};
```

The `ring_buffer_range` type provides the reference model for this concept.

3.2.2.23 *RandomAccessRingBuffer*

A random access ring buffer is a buffer that has an associated ring buffer range.

```
concept RandomAccessRingBuffer<RandomAccessBuffer Buf> {
  range& range(Buf&);
}
```

Note that it returns a non const referencce in the `range` function. This is such that one can mutate the internal write pointer of the ring range associated to the buffer.

3.2.2.24 *RingBuffer*

A ring buffer over frames.

```
concept RingBufferConcept<RandomAccessRingBufferConcept Buf> {
    where MutableBufferRangeConcept<Buf::range>;
};
```

The main model for this is the `ring_buffer` template.

3.2.3 *Algorithms*

The library provides a series of generic algorithms similar to those of the Standard Template Library [52]. Hand coding most loops often makes user code not generic as it is easy to be tempted to make assumptions on the data format when doing so. Using these algorithms it is easier to write concise generic code with no performance overhead. We can distinguish between *sample algorithms*, *frame algorithms* and *range algorithms*.

The first include sample conversion functions and limited arithmetic support — proportional multiplication and inversion.

Frame algorithms allow abstract iteration over the samples of a frame. They are the `static_*` alternatives, including STL alike *transform*, *for each*, *fill*, *generate*, *equal* ... Because the number of channels is encoded in the type, they do perform static unrolling such that no looping overhead is added. Also, there are the aforementioned frame conversion overload that make our frame types convertible.

The last family of algorithms include STL alike *copy*, *for each*, etc. over buffer ranges. They are the `*_frames` func-

tions. Also, those algorithms can mostly be safely used with their original iterator based versions in the `std` namespace because optimised overloads are provided.

3.2.4 *Concrete types*

A bunch of macros generate a whole range of typedefs for concrete types such that user code does not need to mess with long template instantiation. The naming pattern for these concrete types is:

$$\begin{aligned} ColorSpace + BitDepth + [\texttt{f}] + [\texttt{s}] + [\texttt{c}] + \\ [\texttt{\_planar}] + [\texttt{\_step}] + ClassType \end{aligned} \tag{3.4}$$

ColorSpace may be `mono`, `stereo`, `quad` or `surround`. The optional *f* indicates floating point arithmetic samples, *s* is for signed samples, the *c* denotes immutability, *planar* indicates that the data is in non interleaved form. *Step* indicates that it is a type with a dynamic step. *ClassType* may be `ptr` for iterators, `range` for buffer range, `buffer` for buffers, `ring_range` for ring buffer ranges, `ring_buffer` for ring buffers, `frame` for frames, etc.

For example:

```
lrstereo8_buffer a;
surround16_frame; b;
surround16c_planar_ref_t c(b);
stereo32sf_planar_step_ptr_t d;
```

Types in the library are very interrelated with one another. Thus, wide range of metafunctions are provided to map among types, like the `*_from_*` family (e.g. `range_type_from_frame`). The `derived_*_dype` metafunctions can be used to obtain a new type that is mostly like another but

changing some of its parameters. There are metafunctions for obtaining the channel space, number of samples and the rest of properties from types. A full list of the provided metafunctions can be found in the reference manual.

3.2.5 *Going dynamic*

Until now, we have described a very generic system for sound data representation and manipulation that uses static polymorphism. However, we do not always know what kind of sound data we need to use at compile time.

For this purpose we use the `variant` class, which implements a generic type safe disjoint union. Our variant class, taken from GIL, is very similar to the Boost.Variant class[7] — the main differences being that our library takes a MPL sequence as parameter while Boost takes the types directly on the template argument list, and that our visitation function is `apply_operation` in contrast with Boost.Variant's `apply_visitor`.

We provide the subclasses of variant `dynamic_buffer` and `dynamic_buffer_range`. Note that the interface of these types is more limited than the one of `buffer` and `buffer_range`. Specifically, they do not model `RandomAccessBuffer` and `RandomAccessBufferRange` or nor their more concrete frame based counterparts. There are obvious reasons for this: (1) there is no specific associated frame type associated to these so the static metafunctions associated to these models would have no sensible definition and (2) there is no efficient way to implement iteration with dynamic polymor-

7 The Boost.Variant library: `www.boost.org/doc/html/variant.html`

phism — this is the very same reason why we used generic programming in the first place!

However, there are overloads taking a dynamic buffer or range for most algorithms and buffer factories supplied by the library, so they can be used as if they where a concrete static type most of the time. Moreover, with the `apply_operation` function we can execute a function object "inside" the variant, this is, taking as an argument the concrete type that we want. This function object should provide an `operator()` overload of every file format from the variant that it supports. The `operator ()` can of course be a template for improved generality.

A clarifying example follows:

```
using namespace psynth::sound;
namespace mpl = boost::mpl;

// A dynamically determined buffer
typedef sound::dynamic_buffer<
    mpl::vector<
        mono16s_buffer,
        mono32sf_buffer,
        stereo16s_buffer,
        stereo32sf_buffer>>
    my_buffer;

// A generic operation
struct set_to_zero
{
  template <typename R>
  void operator () (const R& data)
  {
    psynth_function_requires<BufferRangeConcept<R>> (); // Works!
    typedef typename sample_type<R>::type sample;
    typedef typename R::value_type frame;
```

```
    const auto zero = sample_traits<sample>::zero_value ();

    fill_frames (data, frame (zero));
  }
};

int main ()
{
  my_buffer buf; // Now it holds a mono16s_buffer;
  buf.recreate (1024);
  apply_operation (range (buf), set_to_zero ());

  buf = stereo32sf_buffer (1024); // Now a stereo32sf_buffer
  apply_operation (sub_range (range (buf), 128, 256),
                   set_to_zero ());

  // These two sentences generate compile errors:
  buf = surround32sf_buffer (1024);
  psynth_function_requires<BufferRangeConcept<buf::range>> ();

  return 0;
}
```

In this example we defined a dynamic buffer type that can hold mono and interleaved stereo buffers of 32 bit signed floating points or 16 bit signed fixed point frames. As the example illustrates, we can assign into it any of these types but static errors are generated when trying to assign other types. The example shows that inside the `set_to_zero` operation we can access to all the static information of the type that is actually in the buffer and write real generic code.

Because dispatching is internally done via a `switch`[8] there is minimal performance overhead — ideally one switch

8 Generated with preprocessor meta programming, which is kind of a hack, but pure template metaprogramming solutions have worse performance.

per operation (because iteration itself is not done via the dynamic interface) which is negligible. However, careless use of this facility produces object code bloat.

This is so because whenever `apply_operation` is used with a generic algorithm, it is instantiated for every possible type that the variant can hold. Bourdev presents a technique [53] that avoids a lot of the bloat using a `reduce` metafunction to partition the types and choose a representative for each subset of the types — the generic algorithm is then instantiated only for the representatives. This escapes the C++ type system and thus the safety relies on the programmer properly matching the types taking into account the properties of the algorithm. Whether by using this technique or by just avoiding dynamic buffers over too many types, special care should be taken when using dynamic buffers.

Note that there exist a `dynamic_ring_buffer` and `dynamic_ring_buffer_range` that weaken their requirements on the underlying buffers and ranges such that they can be used over dynamic buffers and ranges.

3.2.6 *Input and Output module*

When talking about input and output, runtime polymorphism is more important than static generality. We have to map the previous facilities with the data that is out there in formats not always known beforehand and provided by devices and interfaces whose availability depend from one user system to another. Object orientation comes back into play.

Figures 3.3 and 3.4 show the class diagrams for the input and output parts of the module. We use the Unified Mod-

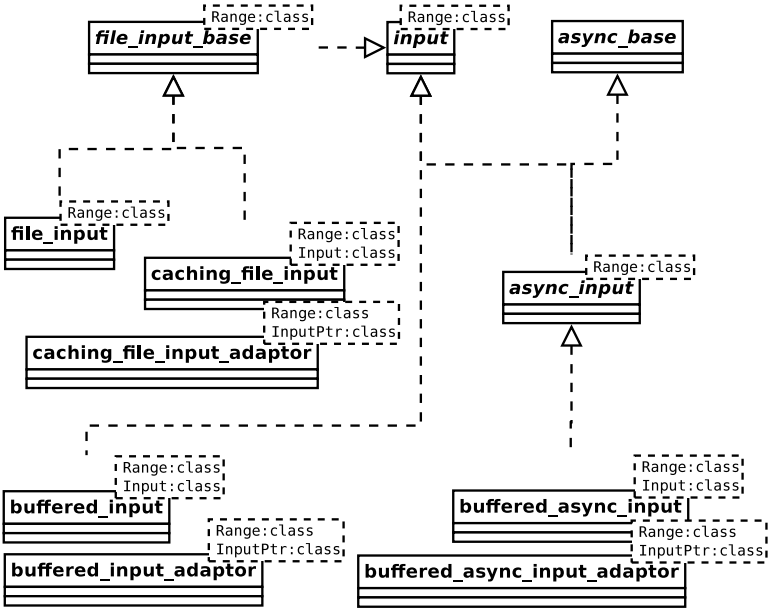


Figure 3.3.: UML class diagram of `psynth::io` input facilities.

elling Language notation [54]. The doted boxes over the corner of some classes list the formal parameter of generic entities, an extension that was introduced in the 2.3 version of UML.

One should start reading the diagrams from the `input` and `output` classes. They are the basic interfaces on which we build the system. They are very simple, providing just a `take(const range&)` and `put (const range&)` abstract method respectively — and a typedef for accessing the actual argument for its `Range` parameter. Because virtual functions can not be templates themselves, these classes, when instantiated, provide through those methods a hook for

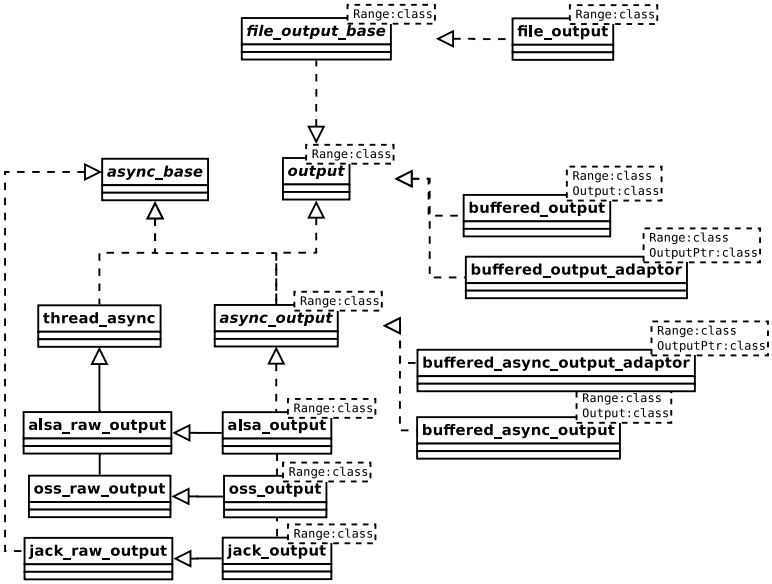


Figure 3.4.: UML class diagram of `psynth::io` output facilities.

polymorphically sending or receiving data to and from the external systems *in only one format*. The next section shows how to work around this. Note also that not every device support every format. Sometimes this is known only at runtime — and an exception will be thrown if trying to open it in the wrong format — but sometimes it can be known at compile time (e.g., the OSS interface itself can not output planar data)

Note that none of our I/O devices support *open* and *close* functions. After our experience in previous versions of Psychosynth, we have decided to simplify the interface and force specialisations to use RAII — opening the resource occurs in the constructor and closing it in the destructor.

This makes life easier to implementer of specialisations (who are released from having to implement a state machine) and for users of the library (who can be sure that a living object is ready to use). If the same variable is to be used for opening several files, one can use `swap` or the *move constructor*. If the lifetime of variable does not match the one of the resource `boost::optional` or a smart pointer should be used.

3.2.6.1 *Buffering*

For the reader who paid attention to our discussion on dynamic buffers and ranges in section 3.2.2 it might seem that constraint of having to interface with only one type can be easily solved by instantiating the I/O devices with a dynamic range. However, in most cases that is not possible, because the device is opened to be used in one mode of operation only and it will not be able to change it until it is reopened. The most sensible solution is to use an intermediate buffer where we convert the data before sending it or receiving it. In order to avoid overhead due to buffering when unneeded this behaviour is not provided by default.

The `buffered_*` family of classes are adaptors that match from a I/O interface in one format to another desired format. Because of buffering, the target interface can be have a `dynamic_range` as parameter. The `buffered_*_adapter` types *aggregate* the adaptee via a pointer or smart pointer such that an already living device can be adapted, or the buffer can be reused for several devices. The other types *compose* the adaptee managing their lifetime — the constructor parameters are forwarded from the buffered adaptor constructor.

3.2.6.2 *Asynchronous operation*

When data is to be provided in real time to or from a sound card, chunks of audio data are to be processed as requested by the device. All the descendants of `async_base` are of this kind. That interface provides `start ()` and `stop ()` methods for controlling the state of the asynchronous operation, plus others for checking its current status.

Whenever new data is sent or required by the device, a user specified callback is executed with the number of frames as parameter, which should in turn call `put ()` or `take ()` once the data is ready or it can consume new information. Unless otherwise specified by an specialisation, we assume that these functions can only be called from the asynchronous callback. Also, we shall assume that the callback is running under real time conditions, and thus it is expected to have a time $O(n)$, where n is the number of requested samples, not block, perform no system calls and allocate no dynamic memory.

3.2.6.3 *Raw I/O*

Most device management code does not depend on the audio format type, and unnecessary object code bloat would be produced if it where in the templates over the buffer range type. It is then abstracted in the "raw" I/O classes — named after the pattern `*_raw_{input|output}`. They can perform the I/O in an type unsafe manner with incoming data in a void pointer. They are mostly an implementation detail, but some users might find them handy as a lightweight RAII wrapper over the underlying C interface.

3.2.6.4 *Caching input*

As we have noted, system calls should be avoided in the audio processing thread. However, files should be read from disk, and potentially decoded from hard formats, in order to play pre-recorded samples or mix full songs. The `caching_file_input` family of adaptors allow this kind of operation by reading the data in big chunks on a separate thread. Because it has to do buffering anyway, it can be used as an interface adaptor from one kind of buffer range to another too.

Our current implementation uses mutexes and condition variables, which should be avoided. It is planned to fix this in the future, section 3.4.2.1 further discusses this issue.

3.2.7 *Synthesis module*

In the `psynth::synth` namespace there is a whole bunch of algorithms for sound synthesis, including oscillators, wave tables, filters of all sorts, time stretchers, etc. Most of them are objects that just provide an `update (...)` method taking buffer ranges as needed plus several manipulators for their parameters.

We feel that their interface can be extensively improved, as we discuss in section 3.4.2.2. For this same reason, we will avoid giving a detailed discussion of its design now.

3.3 VALIDATION

3.3.1 *Unit testing*

Unit testing serves to ensure that the actual behaviour of a function matches its documented requirements. In this code it is very important because: (1) because it is a library to be used by third party developers, the proper functioning of every single method is as important as the observable behaviour of the final application that we may deploy alongside; and (2) because of the duck typing in template metaprograms we need to ensure that all "instantiation paths" — i.e. code paths of the metaprogram — compile without errors and lead to correct execution.

Recordar actualizar esto si más adelante se añaden pruebas por lo que sea

The modules described in this chapter are evaluated with a total 223 unit tests. All of them pass with a total of 1478 successful assertions. Note 3.1 includes a more detailed summary of the test cases involving these modules. Not all of those unit tests have been written manually. Using `BOOST_UNIT_TEST_TEMPLATE` once can define a parametrised over a type and that is later instantiated for every type in a given MPL sequence. For example, we can compute the product of a MPL sequence of buffer types with a metafunction and pass the result to the templated unit test that checks proper conversion among buffer types, simplifying a lot the amount of testing code — avoiding combinatorial explosion of code to test all instantiation paths.

Note 3.1 (`psynth::sound` and `psynth::io` unit tests) *The user can run the unit tests by herself by running `make check` or running the `psynth_unit_tests` in the `src/test` folder.*

```
Test suite "io_input_test_suite" passed with:
  177 assertions out of 177 passed
  110 test cases out of 110 passed

Test suite "io_output_test_suite" passed with:
  140 assertions out of 140 passed
  81 test cases out of 81 passed

Test suite "sound_ring_test" passed with:
  10 assertions out of 10 passed
  4 test cases out of 4 passed

Test suite "frame_iterator_test_suite" passed with:
  5 assertions out of 5 passed
  2 test cases out of 2 passed

Test suite "sound_peformance_test_suite" passed
with:
  38 assertions out of 38 passed
  4 test cases out of 4 passed

Test suite "buffer_test_suite" passed with:
  14 assertions out of 14 passed
  2 test cases out of 2 passed

Test suite "sound_frame_test_suite" passed with:
  81 assertions out of 81 passed
  4 test cases out of 4 passed

Test suite "sound_sample_test_suite" passed with:
```

```
1073 assertions out of 1073 passed
14 test cases out of 14 passed
```

3.3.2 *Performance*

We claimed that static polymorphism and optimal algorithm selection via template metaprogramming allow genericity with no overhead over non-generic implementations. Given the performance constraints on the real time audio processing thread, satisfying this property is a must.

To ensure that there is no overhead, we include a test suite that compares the efficiency of the generic algorithm building blocks provided by the library with hand-rolled loops performing the same function. Listing 3.4 shows an example of such generic function that you can compare with its concrete implementation in 3.5. Note that for simplicity and because it does not affect the results the non generic version is still parametrised over the sample type T. One can read the whole performance test suite in `src/test/psynth/sound/performance.cpp`.

Listing 3.4: Generic `for_each` that asigns $(0, 1)$ to every frame over non interleaved data

```
for_each_frame (_v, [] (F& f) {
    f = F {0, 1};
});
```

We check the performance on some different cases that try to stress different potential overhead corners, like using a

Listing 3.5: Non generic `for_each` that asigns $(0, 1)$ to every frame over non interleaved data

```
T* first = (T*)_v.begin ();
T* last = first + _v.size () * 2;
while (first != last) {
    first [0] = 0;
    first [1] = 1;
    first += 2;
}
```

different layout from the natural order in the channel space, planar and interleaved buffers, etc. Each test case test a kind of loop over a buffer of a certain size. We test sizes of 32 and 4092 samples, as those are the common bounds of buffer sizes used in audio (the lower the buffer size, the better —lower— latency). This way we ensure that both the per buffer and per sample overhead is minimal. Each test is repeated 2^{21} times for buffers with 4096 samples, and 2^{26} times for buffers of 32 samples.

Name	Meaning
`s8b`	Interleaved stereo buffer with 8 bit samples.
`rs8b`	Interleaved reversed stereo buffer with 8 bit samples.
`s8pb`	Planar stereo buffer with 8 bit samples.
`s8f`	Stereo frame with 8 bit samples.
`rs8f`	Reversed stereo frame with 8 bit samples.

Table 3.2.: Acronyms for the parameter types in performance test result tables.

We ran the tests in a Intel i5 M460 with four 2.53FHz cores. We compiled tested the code with different versions of GCC compiled with options `-O3 -std=c++0x`. Tables 3.3

and 3.4 show the results for GCC 4.5.2 with 4096 and 32 buffer size respectively and 3.5 and 3.6 show the results for GCC 4.6.0. The acronyms for the concrete parameter types are expanded in table 3.2. All the timings are represented in *milliseconds* computed as the mean of all iterations. The "gain" is defined as:

$$Gain = \frac{T_{non\_psynth}}{T_{psynth}} \tag{3.5}$$

Such that a value greater than one signifies that the generic version is faster and a value smaller than one implies that there is some overhead.

The tables show interesting results. With GCC 4.5 the generic versions performs as efficiently or better most of the time for larger buffer sizes, however, there seems to be an additive constant that makes the hand coded version slightly better when the buffer size is very small. Surprisingly, GCC 4.6 changes this tendency and the generic version actually gets more favourable results with short buffer sizes. Actually the results are more even with this compiler in a probable tendency in making optimisation techniques more general — i.e. dependent on the actual semantics of the code and not on how you express it — which is very desirable for generic code.

In any case, these results supports our claims that the performance overhead, if any, is negligible. Sometimes the generic code is even more efficient than a carefully hand-coded algorithm for a specific audio format. The results also show that as soon as you add arithmetic computations like in the `transform` test micro-optimisation in the looping constructs is futile. Maybe testing with a wider variety of compilers should be done, but that is not possible because

few support C++0x. Anyway, it is to expect that a compiler supporting C++0x also has a decent optimiser and the we would get similar results. Comparison of the generated object code would can be another interesting mechanism for assuring the lack of overhead due to generality. We did informal tests confirming our hypotheses that can be repeated by the reader. Also, there are some similar object code comparisons in the Boost.GIL video tutorial[9] that the interested reader can check — given the similarities in GIL and `psynth::sound` we can expect equivalent results.

We can safely state that the library passes the performance requirements.

3.3.3 *Integration*

The module was first developed separately from the main development branch in a branch called `gil-import`. Once the previous tests were passed, the former signal representation classes and I/O code was removed from the code base. The upper layers — mainly the `graph` layer — was adapted to use the new library. Note that, given that we will mostly rewrite the `graph` layer in the next iteration we tried to make minimal changes to get the project compile and run properly again.

The system was then *peer reviewed* by project collaborators, mainly by the maintainer of the Ubuntu/Trinux packages Aleksander Morgado[10]. After minor bugfixing, we agreed to make a new Psychosynth 0.2.0 release that included the

9 Boost.GIL presentation: `http://stlab.adobe.com/gil/presentation/index.htm`

10 Aleksander Morgado's web blog: `http://sigquit.wordpress.com`

Algo.	Parameters		Psynth	Non-Psynth	Gain
fill	s8b	s8f	0.38147	0.38147	1
		rs8f	0.376701	0.38147	1.0127
	s8pb	s8f	0.195503	0.190735	0.9756
		rs8f	0.190735	0.190735	1
for_each	s8b		0.371933	0.376701	1.0128
	s8pb		0.286102	0.281334	0.9833
copy	s8b	s8b	0.38147	0.38147	1
		rs8b	1.00136	1.05381	1.0524
		s8pb	0.753403	0.753403	1
	s8pb	s8pb	0.748634	0.762939	1.0191
		s8b	0.38147	0.38147	1
transform	s8b	s8b	23.4795	23.4842	1.0002
		s8pb	23.4842	23.4842	1
	s8pb	s8b	23.4842	23.4842	1
		s8pb	23.4842	23.4842	1

Table 3.3.: Performance tests 4096 buffer size with GCC 4.5.

Algo.	Parameters		Psynth	Non-Psynth	Gain
fill	s8b	s8f	0.0103563	0.0104308	1.0071
		rs8f	0.0103563	0.0104308	1.0071
	s8pb	s8f	0.0125915	0.0104308	0.8284
		rs8f	0.018999	0.0139326	0.7333
for_each	s8b		0.00149012	0.00141561	0.9499
	s8pb		0.0090152	0.0064075	0.7107
copy	s8b	s8b	0.012517	0.012219	0.9761
		rs8b	0.0112504	0.0108033	0.9602
		s8pb	0.0179559	0.0175089	0.9751
	s8pb	s8pb	0.0239909	0.0243634	1.0155
		s8b	0.0136346	0.00394881	0.2896
transform	s8b	s8b	0.183433	0.183508	1.0004
		s8pb	0.183508	0.183433	0.9995
	s8pb	s8b	0.183433	0.183508	1.0004
		s8pb	0.183508	0.183433	0.9995

Table 3.4.: Performance tests for 32 buffer size with GCC 4.5.

Algo.	Parameters		Psynth	Non-Psynth	Gain
fill	s8b	s8f	0.557899	0.562668	1.0085
		rs8f	0.557899	0.557899	1
	s8pb	s8f	0.190735	0.190735	1
		rs8f	0.190735	0.185966	0.9749
for_each	s8b		0.557899	0.562668	1.0085
	s8pb		0.276566	0.286102	1.0344
copy	s8b	s8b	0.739098	0.743866	1.0064
		rs8b	1.05381	0.934601	0.8868
		s8pb	0.753403	0.753403	1
	s8pb	s8pb	0.867844	0.743866	0.8571
		s8b	0.38147	0.386238	1.0125
transform	s8b	s8b	23.4842	23.4795	0.9997
		s8pb	23.4842	23.4842	1
	s8pb	s8b	23.4842	23.4842	1
		s8pb	23.4842	23.4795	0.9997

Table 3.5.: Performance tests for 4096 samples with GCC 4.6.

Algo.	Parameters		Psynth	Non-Psynth	Gain
fill	s8b	s8f	0.00625849	0.00625849	1
		rs8f	0.00499189	0.00499189	1
	s8pb	s8f	0.0103563	0.0104308	1.0071
		rs8f	0.0104308	0.0104308	1
for_each	s8b		0.00618398	0.00625849	1.0120
	s8pb		0.0064075	0.0064075	1
copy	s8b	s8b	0.012219	0.0125915	1.0304
		rs8b	0.00782311	0.0111014	1.419
		s8pb	0.0169873	0.0172853	1.018
	s8pb	s8pb	0.0243634	0.0219047	0.899
		s8b	0.0114739	0.0160933	1.4026
transform	s8b	s8b	0.183433	0.183508	1.0004
		s8pb	0.183433	0.183582	1.0004
	s8pb	s8b	0.183433	0.183433	1
		s8pb	0.183433	0.183508	1.0004

Table 3.6.: Performance tests for 32 buffer size with GCC 4.6.

new code described in this chapter and some other fixes and modifications developed alongside.

The official changelog briefing for this release is included in note 3.2.

Note 3.2 (Changelog of Psychosynth 0.2.0)

- *New audio processing and I/O subsystem based on template programming for generic yet efficient sound signals.*
- *The extreme latency when using ALSA bug seems to be fixed in some cases.*
- *No longer depend on libvorbis, libsndfile can now handle ogg and flac files too.*
- *No longer depend on* `libsigc++`*, using* `boost::signals` *which, which is a bit slower but neglibe and this simplifies the dependencies.*
- *The mouse wheel now scrolls in the object selector.*
- *The object selector no longer lets mouse clicks pass through.*
- *Backwards reproducing a sample works a bit better now too.*
- *Some new niceties in the framework base layer, including some experiments on applying the C3 class linearisation algorithm in raw C++.*
- *C++ox features are being used in the code. For GCC, this means version 4.5 shall be used. We doubt it will compile with any other compiler (maybe latest VS), but users are welcomed to try and report.*

- *For this same reason, Boost.Threads is no longer a dependency, we use STL threads instead.*

3.4 CONCLUSIONS

In this iteration we developed a generic approach to representation and input and output of audio data. We do not have any records of any audio software using such paradigm in their code, so this development have been exploratory and has a lot of value in its novelty. This however delayed our development more than expected in our original plan.

3.4.1 *Benefits and caveats*

The three main advantages in the new code are:

1. Code can be mostly abstracted from the audio format representation while retaining near optimal performance. Because generality allowed decoupling orthogonal concepts, each audio representation factor can be optimised and tuned for computational accuracy on its own, leading to higher quality code with lower maintenance cost as we avoid the combinatorial explosion that happens otherwise.
2. Algorithms correctly written with our generic facilities have a performance equivalent to the hand-written code. When a certain algorithm has not general interpretation or can not be efficiently implemented generally, the library still allows for the algorithm to be

written with for a concrete or a constrained family of audio formats.

3. Because the signal format is encoded in the data type, we can either statically check that the data is in the correct format through our processing chain, or trivially enforce runtime checks when the format is unknown at compile time (via `dynamic_buffer` and similar tools), leading to more secure code.

Also, because a lot of learning have happened since the old code base was written, the new code is better written and quite safer, making use of exceptions and scope guards [55].

Even though we believe the benefits outweight the drawbacks, we have to acknowledge the caveats of our new approach, the most relevant being:

1. The new code uses advanced C++ programming techniques that many programmers find hard to understand. Thus, it might be harder for casual contributors to join the project in the future.
2. In the absence of real language support for concepts, template metaprograms leak their implementation in user code's compilation errors. This is so because, actually, by expanding the type instantiations in the compilation error, the compiler is actually showing a full backtrace of the metaprogram. This leads to cryptic error messages that often obfuscate the real source of the problem, discouraging novel developers.
3. Template metaprograms take longer to compile. However, proper usage of the new `extern template` facility should avoid redundantly instantiating templates

in different translation units only to be discarded by the linker mostly solving this issue. Also, because the compiler generates different object code for each audio format, thoughtless use of the library can lead to code bloat and too large binary size.

3.4.2 *Future work*

While the current status of the library is quite satisfactory for our needs, there is still a lot of room for improvement that we will postpone since they fall outside this project's scope. Nonetheless it is worth enumerating them:

3.4.2.1 *Lock free ring buffers*

The audio processing thread has real time constraints. As we introduced in note 2.1 this implies, among other things, forbidding the usage of mutexes. However, our ring buffers have not been tested for thread safety and mutexes should be used when shared among different threads. This happens in our `caching_file_input` implementation. The problem is specially severe when the audio processing thread is running with higher priority, because *priority inversion* occurs [56]. While our current Alsa and OSS output systems do not attempt rising their thread priority, the Jackd subsystem executes the audio processing callback in a real-time mode thread and thus the problem can become significant.

Locking is done with care and in practice we have experience no buffer underruns due to this problem, even with high number of caching file readers in execution. However, for correctness and better support of Jackd, we should im-

plement a lock-free ring buffer [57, 58] that does not require using special synchronisation. Jackd actually provides a C implementation that can serve as a basis for ours. The most important interface change is that only one reader should be permitted — we can embed the read pointer inside the data structure. Also, the "backwards" operation mode in our current implementation might introduce an unexpected complexity in the implementation.

3.4.2.2 *Virtual and adapted iterators and ranges*

Boost.GIL included a "locator adapter" and "virtual locator" notions that allowed creating or modifying images lazily via a function object. We discarded implementing the virtual ones because they were coupled to their `locator` concept which is specific to the problem of image representation — locators are in practice 2D iterators. Moreover, they used the indexed position in the image as the parameter to the function object that synthesised the image. However, because audio is processed in chunks, the position in the audio buffer is meaningless for the synthesis or filter function — instead, some stateful function object which includes a notion of time position related to the frame rate is needed. Thus, many unexplored design decisions should be taken, and the interactions with other similar libraries like Boost.Iterator[11] and Boost.Range[12] should be carefully evaluated.

11 The Boost.Iterator Library: `http://www.boost.org/doc/libs/release/libs/iterator`

12 The Boost.Range Library: `http://www.boost.org/doc/libs/release/libs/range`

3.4.2.3 *Better arithmetic support*

The library includes some basic arithmetic support for samples. There are few complications when developing full generic arithmetic support for samples and frames. As Lubomir Bourdev, lead developer of Boost.GIL, stated it in an email conversation with us:

> "Doing arithmetic operations is tricky for a number of reasons:
>
> - What do you do on overflow? Clip, throw exception, allow out of range values?
> - What is the type to be used during conversion? Even if the source and destination have the same type, the operation might need to be done in another type and then cast back.
> - Certain arithmetic operations have no meaningful interpretation as far as color is concerned, such as multiplying one pixel by another.
>
> Because of issues like these we have not tackled the problem of providing arithmetic operations, but we have provided generic operations that can be done per channel or per pair of channels which could be the basis for arithmetic operations."

Nonetheless with time and effort the problem could be approached making some compromises. Some of the issues Lubomir states have different answers for sound processing, in fact, allowing out of range values is the best answer for the first question given the fact that sound amplitude is not

naturally constrained and clipping is introduced only by the DAC hardware or when moving from floating to a fixed point representation. Maybe, not all those questions have to be answered in order to improve the arithmetic support.

One of the main annoyances when writing a generic algorithm is using the per sample `static_*` algorithms. Using them we could write a simple arithmetic layer for frames that would simplify the user code. However, even though we do not have experimental data, we believe this straightforward solution could introduce overhead. This is because every `static_*` unrolls one statement per channel. Thus, a simple frame arithmetic expression would in fact result into many sequence points that is yet to be tested whether compilers can optimise properly.

This is not a dead end. Using the *expressions templates* [59] technique and r-value references we can perform transformations with the aid of metaprogramming such that sequence points are not introduced by the arithmetic expression itself. A expression template framework like Boost.Proto[13] [60] could be of great help. Moreover, with careful studying of the audio DSL's studied in section 1.3.3 and the usage of these same techniques the scope of such effort could be broadened to build a full sound synthesis and processing EDSL for C++.

3.4.2.4 *Submission to Boost*

In our conversations with Lubomir Bourdev he suggested submitting our library for inclusion in the Boost package.

13 The Boost Proto library: `http://www.boost.org/doc/libs/release/libs/proto`

However, there are few issues that we should tackle before that:

1. A lot of code is algorithmically identical to that of Boost.GIL with changes only in terminology. A lot of work in properly abstracting such common parts should be made to avoid code repetition and doubled maintenance effort inside Boost.
2. As we said earlier, interesting interactions can emerge with the Boost.Iterator and Boost.Range libraries. We believe that any possible issues and unneeded incompatibilities with those libraries should be solved before submission into Boost.
3. Boost is written in C++03 standard, while our code uses C++0x. Moreover, our code has dependencies with other submodules in `psynth::base`, specially the expection and logging system. While these dependencies are not too strong, the effort made to polish these corners is outside the scope of the current project.

4

A MODULAR SYNTHESIS ENGINE

> We can now see that the whole becomes not merely more, but very different from the sum of its parts.
>
> *More is Different: Broken Symmetry and the Nature of the Hierarchical Structure of Science*
> Philip Warren Anderson

4.1 ANALYSIS

In the previous chapter we built a system for sound representation, processing, and interfacing. In such system, interactions between different processing elements is hard-coded in the control flow of the program itself and the relations among the statically parametrised types that intervene.

As described by requirements 4 to 11, we shall develop a system where the basic DSP units can be composed orthogonally and hierarchically to build complex devices *at runtime*. With the applications built on top of our framework being targeted at live performances, it should be particularly dynamic.

4.1.1 *An abstract model of a modular synthesiser*

In a modular synthesiser, the sound generation is made by interconnecting basic processing units. Each of them might generate sound, filter it, and can have a varying number of inputs and outputs of different kinds. Because a module can apply virtually any mathematical function to its inputs; we can realise any synthesis technique — i.e. additive, subtractive, FM/PM — by just wiring the available modules in an appropriate way.

We can characterise such a system by abstracting the parts of one of these processing units. A hardware modular synthesiser can be used to illustrate the concepts behind this, as shown by the *frequency shifter*[1] in figure 4.1. We can thus taxonomise its parts as in the following; we will later use this terminology in our design.

INPUT PORTS These are the signals that come from another module. In the example figure we can see a *input* signal that carries an arbitrary sound wave to be frequency-shifted, and a *CV in*, which is used to modulate the shift parameter. In a hardware module we can con-

1 A frequency shifter is a module that produces oscillating modifications on the frequency components of a given input signals, producing interesting Doppler effects, binaural effects, vibratos, and so on.

sider as an input anything that can go through a wire, thus we are no limited just to analog time-domain signals, but we can consider a digital MIDI input of a synthesiser as an input signal too.

In our software system we are even less constrained, and our system should cope with signals of any kind — i.e. of any type, in the programming language sense. Note also that an input might remain disconnected during execution of the synthesis graph, and a proper default behaviour should be implemented in that case.

OUTPUT PORTS These are the signals that the module sends to other modules. They are of the same nature than their input counterparts.

In order to connect an output to an input, the kind of signals that flows through them must match; in a computer based digital system this should be checked and proper error handling must come into action if necessary or maybe some automatic conversion mechanism if safe and applicable.

Note that while on and hardware synth inputs and outputs are generally related in a one-to-one manner — unless we do not consider a hub/split or a mixer a module by itself but a connection device — in software we can relate them in a one-to-many fashion, as the value produced in an *output port* can be read several times by different modules if it has its own memory.

PARAMETER CONTROLS These allow the user to tune the different settings of the device. In a hardware device, they are most of the time represented by a knob or a slider, but modern synthesisers include bidimensional

touchpads and other input devices for controlling the process.

Note that, at this stage, the notion of a *parameter* or *control* is not directly related to how it might be represented to the user — like a text-box, virtual knob, or whatosever — but to the abstractions that the a DSP module uses to get input from outside of its inner processing function. Just like an input port, it should be of any type. For example, any control that is naturally representable with a knob or slide is quite often a *float* parameter.

The reader might wonder how is this different from an input port in a software system. On the one hand, controls do not have any built-in interconnection system. On the other hand, the synchronisation mechanisms used to pass the values through controls and ports are quite different. This is so because while the information that goes through ports is to be used only by the DSP modules, controls get input from the user interface thread, thus requiring special care. We will discuss this further in the next section.

STATUS CONTROLS These provide feedback to the user about relevant parts of the state of the processing. In the example figure, a red led is light up when the output signal exceeds the *clipping value* — i.e. the upper threshold of the admissible range of the amplitude of the signal — suggesting the user to lower the gain to avoid annoying distortion. Everything that have been said about parameter control applies here. Once again, what is important is the abstraction not the visual representation; those the aforementioned example would

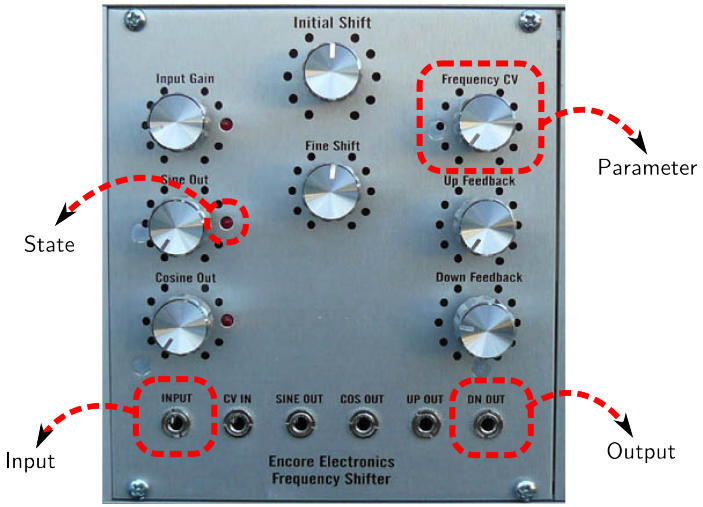


Figure 4.1.: Components of a synthesis module illustrated in a standard *frequency shifter*.

be a *boolean* status control in our system, that might be represented by any other means.

A collection of modules and the connections among them is called a *patch*. In a hardware modular synthesiser, these are assembled in special racks that have slots satisfying some standard to place the modules in them — for example, the module in the figure fits in *eurorack* slots. In many software synths, and specifically in the one we are developing, patches can be arranged hierarchically. We can visualise this as if we could put a whole rack into a black box with holes to expose certain controls and ports to the outside, and then place this box in a larger rack. This is very useful in a software synthesiser; for example, one could build a complex patch out of low-level processing primitives and expose a simplified interface. Some software, like Reaktor,

call these patches *instruments*. This arrangement can then be stored into a file for later use. On the Internet one can find many collections of these ready-to-use patches, and there are even commercial packages developed by professional sound designers.

The *conceptual class diagram* in figure 4.2 summarises all this. Note that this is a conceptual class diagram, not a design one, so there is not necessarily a direct match to the classes in the actual code, not even terminologically.

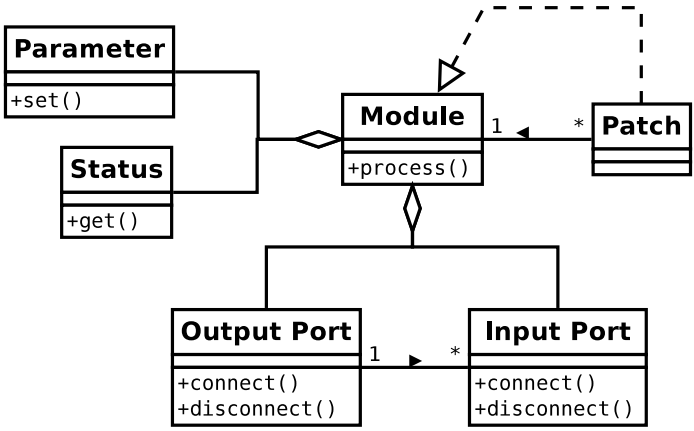


Figure 4.2.: Conceptual class diagram of the modular synthesis main entities.

4.1.2 *Software real time synthesis concerns*

Our system generates the audio in real-time sending it directly to an output device — a sound-card. Even more, it is specially targeted at live performances, so every operation should be designed such that it does not disrupt the audio generation process and generates no kind of noise.

For example, in some software modular synths changing a connection among ports triggers a recomputation of the dependencies among modules to generate a compiled form of the patch for easier execution; but that often that takes long enough to produce a small buffer underrun — i.e. an audible click — as it might involve doing non-linear computations or locking certain mutexes to synchronise the state with the user thread. Actually, the port reconnection issue requires special care, as we will discuss in section 4.2. Because of *dynamic-patching* we should expect the topology of the synthesis graph to change a lot during a live performance.

To better understand this problem we should understand how audio is usually processed in real-time, thus feeding us with proper terminology and knowledge to later tackle the issue. While this might seem like a design issue to some, this is such a universal structure that it shall be consider as a fixed constraint to be analysed than a design decision itself.

Ideally, we would produce each sample one at a time and send it to the output device. However, because we have to produce, at least, 44100 samples per second — more in some professional applications — traversing all the synthesis system for every frame involves many function calls, some of them even require dynamic dispatch in an extensible system like ours, leading to a too low performance to deliver the samples on time. For this reason, audio is processed blocks of *block size* samples in tight loops. This *block size* might match the device's buffer size or it might be smaller. Parameter and status values are updated only in between blocks. Thus, we should try to keep the block size as low as possible to avoid noticeable latency, and most professional audio software allow changing this parameter to fit the machine processing power. A sensible block size is 64 samples, like

Pure Data's default. Using the terminology in [61], we can distinguish between *audio rate* — i.e. the frame rate as described in the previous chapter — and *control rate*, which is the sampling frequency of control signals — i.e. the signals that produce only one sample per processing block:

$$control\text{-}rate = \frac{audio\text{-}rate}{block\text{-}size} \tag{4.1}$$

Note that control signals are not restricted to what we labelled as *controls* in the previous sections. In fact, through a *port* may flow signals at audio rate if their signal type is that of an audio buffer that holds a block of samples, or at control rate if their signal type is that of a single sample, like a `float` or `double`.

Interleaving the audio computation within the user-interface loop is not plausible; thus the audio processing lives in its own thread/s. In fact, as we saw in the previous system, some audio output interfaces like Jackd control the audio thread themselves invoking the processing function through a callback. Our own device API wrapper that we developed in the previous chapter promotes this kind of asynchronous output. The inverse of the control rate, that we may call the *control period*, provides a very explicit deadline for block processing function. Missing the deadline might not kill people, but it would produce an unpleasant clicking noise and thus we have to do as much as possible to meet the deadline[2]. Our system can be categorised as *soft real-time* [62]. Apart from trying to get real-time priority in the audio thread, we have to take special care when developing the program, and this will deeply influence the design:

2 In fact, in the middle of a live performance with a 10^5 watts sound system, consequences might be more severe as it seems superficially :)

1. Avoid system calls in the audio thread. System calls produce a context switch and it can take quite long until the audio thread is preempted back; at least too much to meet the deadline. This is specially true for I/O system calls. In the previous chapter we already provided some devices to avoid this problem, like the caching file readers.
2. Avoid contending for a *mutex* or some other kind of lock with the user thread. Without special support from the operating system — like priority inheriting mutexes — this can lead to priority inversion [56]. Even in the later case, the context switch produced by the contention gives good chances to miss the deadline until the audio thread is preempted. The overhead of locking a mutex when there is no contention is negligible in most operating systems, and it definitely is in Linux which uses *futexes* (Fast User-space Mutex) to implement them [63], so using a *try-lock* operation is usually acceptable. The rule of thumb is to never wait on a mutex in the audio thread, and use lock-free data structures [64] or do conditional locking instead.
3. Because the dead line depends directly on the block size n, algorithms running in the audio thread should be $O(n)$. A special consequence of this restriction is that allocating memory in the heap, at least with the default allocator — i.e. using `new` — is forbidden. This is so because memory allocation algorithms are not proportional to the size of the requested block, but instead depend on non-deterministic properties like the pattern of memory usage and use complex search algorithms to find a fitting block of spare memory. This restriction implies that manipulating STL containers

is forbidden too. For some special kinds of objects, a custom memory allocator can do the job [33]. Sometimes, an intrusive data structure, like *Boost.Intrusive* STL counterparts [3] are enough to avoid the allocations. In other situations, we can release the job of allocating memory to the user thread, use custom data structures or any ad-hoc solution.

4.2 DESIGN

4.3 VALIDATION

4.4 CONLUSION

3 `http://www.boost.org/doc/libs/release/doc/html/intrusive.html`

BIBLIOGRAPHY

[1] Sergi Jordà, Günter Geiger, Marcos Alonso, and Martin Kaltenbrunner. The Reactable: Exploring the synergy between live music performance and tabletop tangible interfaces. In *In Proceedings of the first international conference on "Tangible and Embedded Interaction", Baton*, pages 139–146. ACM Press, 2007.

[2] Paul Tingen. Autechre, recording electronica. *Sound on Sound*, 2004.

[3] Tue Haste Andersen. Mixxx: towards novel dj interfaces. In *Proceedings of the 2003 conference on New interfaces for musical expression*, NIME '03, pages 30–35, Singapore, Singapore, 2003. National University of Singapore.

[4] Robert A. Moog. Voltage-controlled electronic music modules. In *Audio Engineering Society Convention 16*, 10 1964.

[5] M. Kaltenbrunner, G. Geiger, and S. Jordà. Dynamic patches for live musical performance. 2004.

[6] Thor Magnusson and Enrike Hurtado Mendieta. The acoustic, the digital and the body: a survey on musical instruments. In *Proceedings of the 7th international conference on New interfaces for musical expression*, NIME '07, pages 94–99, New York, NY, USA, 2007. ACM.

[7] MIDI Manufacturers Association Incorporated. *Complete MIDI 1.0 Detailed Specification*, 1999/2008.

[8] Richard M. Stallman and Joshua Gay. *Free Software, Free Society: Selected Essays of Richard M. Stallman*. Free Software Foundation, June 2002.

[9] Miller Puckette. Max at 17. In *Computer Music Journal*, volume 26 (4), pages 31–43, 10 2002.

[10] Miller Puckette. Pure data: another integrated computer music environment. In *in Proceedings, International Computer Music Conference*, pages 37–41, 1996.

[11] Stephane Letz, Nedko Arnaudov, and Romain Moret. What is new in jack 2? In *In Proceedings of the Linux Audio Conference 2009*, 2009.

[12] Torben Hohn, Alexander Carot, and Christian Werner. Netjack: Remote music collaboration with electronic sequencers on the internet. In *In Proceedings of the Linux Audio Conference 2009*, 2009.

[13] Tina Blaine and Sidney Fels. Contexts of collaborative musical experiences. In *NIME '03: Proceedings of the 2003 conference on New interfaces for musical expression*, pages 129–134, Singapore, Singapore, 2003. National University of Singapore.

[14] James Patten, Ben Recht, and Hiroshi Ishii. Audiopad: a tag-based interface for musical performance. In *Proceedings of the 2002 conference on New interfaces for musical expression*, NIME '02, pages 1–6, Singapore, Singapore, 2002. National University of Singapore.

[15] Tina Blaine. The jam-o-drum interactive music system: a study in interaction design. In *In DIS '00: Proceedings of the conference on Designing interactive systems*, pages

165–173. ACM Press, 2000.

[16] R. Bencina, M. Kaltenbrunner, and S. Jordà. Improved topological fiducial tracking in the reactivision system. 2005.

[17] Matthew Wright Center and Matthew Wright. Open-sound control: State of the art 2003. In *In Proceedings of the 2003 Conference on New Interfaces for Musical Expression (NIME-03*, pages 153–159, 2003.

[18] Martin Kaltenbrunner, Till Bovermann, Ross Bencina, and Enrico Costanza. Tuio: A protocol for table-top tangible user interfaces. In *In Proceedings of the 2 nd Interactive Sonification Workshop*, 2005.

[19] Yu Nishibori and Toshio Iwai. Tenori-on. In Norbert Schnell, Frédéric Bevilacqua, Michael J. Lyons, and Atau Tanaka, editors, *NIME*, pages 172–175. IRCAM - Centre Pompidou in collaboration with Sorbonne University, 2006.

[20] Gary P. Scavone and Perry R. Cook. Rtmidi, rtaudio, and a synthesis toolkit (stk) update. In *In Proceedings of the International Computer Music Conference*, 2005.

[21] R. Boulanger, editor. *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing,and Programming*. The MIT Press, March 2000.

[22] James McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[23] Ge Wang and Perry R. Cook. Chuck: a concurrent, on-the-fly audio programming language. In *Proc. ICMC*, pages 219–226, 2003.

[24] Yann Orlarey, Dominique Fober, and Stephane Letz. *FAUST: an Efficient Functional Approach to DSP Programming*, pages 65–96. 2009.

[25] Heinrich Taube. An Introduction to Common Music. *Computer Music Journal*, 21(1), 1997.

[26] Xavier Amatriain, Pau Arumi, and David Garcia. Clam: a framework for efficient and rapid development of cross-platform audio applications. In *Proceedings of the 14th annual ACM international conference on Multimedia*, MULTIMEDIA '06, pages 951–954, New York, NY, USA, 2006. ACM.

[27] Richard Stallman. Why you shouldn't use the Lesser GPL for your next library. *The GNU Project Philosophy webpage: http://www.gnu.org/licenses/why-not-lgpl.html, last accessed 4 of January 2011.*, 1997.

[28] Herb Sutter. Trip report: March 2010 iso c++ standards meeting. *Sutter's Mill Blog: http://herbsutter.com/2010/03/13/trip-report-march-2010-iso-c-standards-meeting/, last accessed 04 of January 2011.*, 2010.

[29] R. M. Stallman. *GNU Coding Standards*. FSF, `http://www.gnu.org/prep/standards/standards.html`, last visited January 5th 2011, 1992–2010.

[30] Juan Pedro Bolívar Puente. GNU Psychosynth: Un sintetizador de audio por Software Libre. *Novática*, 195:47–50, September 2008.

[31] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, March 1995.

[32] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[33] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[34] Martin Fowler. *Is design dead?*, pages 3–17. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[35] Watts S. Humphrey. Using a defined and measured personal software process. *IEEE Software*, 13:77–88, 1996.

[36] E. Goldstein. *Sensation and Perception*. Wadsworth Publishing Compan, Belmont, California, 6th edition edition, 2001.

[37] Harvery Fletcher. Loudness and pitch. *Bell Laboratories Record*, 13(5):130–137, January 1935.

[38] Bruce Fries and Marty Fries. *Digital Audio Essentials*. O'Reilly Media, Inc., 2005.

[39] Ars Ludwig. Music and the human hearing. *Self published: http://www.silcom.com/~aludwig/EARS.htm, last accessed 03 of March 2011.*, 2009.

[40] Steven W. Smith. *The scientist and engineer's guide to digital signal processing*. California Technical Publishing, San Diego, CA, USA, 2002.

[41] Jaakko Järvi, Mat Marcus, and Jacob N. Smith. Programming with c++ concepts. *Science of Computer Programming*, 75(7):596 – 614, 2010. Generative Programming and Component Engineering (GPCE 2007).

[42] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in c++. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, pages 306–323, New York, NY, USA, 1996. ACM.

[43] Jean-Philippe Bernardy, Patrik Jansson, Marcin Zalewski, Sibylle Schupp, and Andreas Priesnitz. A comparison of c++ concepts and haskell type classes. In *Proceedings of the ACM SIGPLAN workshop on Generic programming*, WGP '08, pages 37–48, New York, NY, USA, 2008. ACM.

[44] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, November 2002.

[45] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[46] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, 1 edition, June 2009.

[47] Jeremy G. Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *Proceedings of the First Workshop on C++ Template Programming*, Erfurt, Germany, 2000.

[48] Douglas Gregor and Bjarne Stroustrup. Concepts (revision 1). (N2081=06-0151), 10/2006 2006.

[49] Todd L. Veldhuizen. C++ templates are turing complete. Technical report, Indiana University of Computer Science, 2003.

[50] Erwin Unruh. Compile time prime number computation. ANSI X3J16-94-0075/ISO WG21-462. Available

online: http://www.erwin-unruh.de/primorig.html, 1994.

[51] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[52] Alexander Stepanov and Meng Lee. The standard template library. Technical Report HPL-94-34, Hewlett-Packard laboratories, 1994.

[53] Lubomir Bourdev and Jaakko Järvi. Efficient run-time dispatching in generic programming with minimal code bloat. *Sci. Comput. Program.*, 76:243–257, April 2011.

[54] Object Management Group. OMG unified modeling language (OMG UML) infrastructure version 2.3. Technical Report formal/2010-05-03, 2010.

[55] Andrei Alexandrescu and Petru Marginean. Generic<Programming>: Change the Way You Write Exception-Safe Code—Forever. *C/C++ User Journal*, 18(12), December 2000.

[56] K. H. (Kane) Kim. Basic program structures for avoiding priority inversions. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '03, pages 26–, Washington, DC, USA, 2003. IEEE Computer Society.

[57] John D. Valois. Implementing lock-free queues. In *In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV*, pages 64–69, 1994.

[58] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical report, 1995.

[59] Todd Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[60] Eric Niebler. Proto: a compiler construction toolkit for dsels. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, LCSD '07, pages 42–51, New York, NY, USA, 2007. ACM.

[61] R. Boulanger, V. Lazzarini, and M. Mathews. *The Audio Programming Book*. The MIT Press, 2010.

[62] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[63] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Proceedings of the Ottawa Linux Symposium*, OLS '02, pages 479–595, 2002.

[64] John David Valois. *Lock-free data structures*. PhD thesis, Troy, NY, USA, 1996. UMI Order No. GAX95-44082.

Appendices

A

GLOSSARY

¿Añado un glosario recopilando las definiciones que doy de diferentes términos específicos del dominio del audio y para expandir y aclarar siglas y acrónimos? ¿O mejor paso? — JP

B

GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

B.1 PREAMBLE

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software;

it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, al-

though the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

B.2 terms and conditions

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed

as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

   The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding

Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the

covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-

permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

   b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however,

if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

   a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

   b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the

object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where

> the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permis-

sions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

   You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as

a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation

(including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent

license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted un-

der this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

    If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of

a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT

LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

End of Terms and Conditions

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it do

Copyright (C) <textyear>  <name of author>
```

```
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.