ugr | 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: Un framework para la síntesis de audio modular, interactiva y colaborativa y la ejecución de música en directo.

Juan Pedro Bolívar Puente

Palabras clave: síntesis modular, interactividad, sistemas colaborativos, programación genérica, tiempo real, redes, C++0x, software libre, GNU

Resumen

En este proyecto de fin de carrera desarrollamos un sístema de síntesis de audio modular, interactiva y colaborativa orientado a la interpretación de música en directo. El sistema puede mezclar y manipular sonidos desde ficheros, una amplia variedad de generadores y aplicar filtros y efectos de toda clase. La interactividad se consigue mediante la técnica de *conexionado dinámico*, que permite alterar la topología del grafo de síntesis con simples movimientos de los módulos de síntesis, así como prestando especial atención en la implementación para que cualquier operación sea aplicable en tiempo real sin generar distorsiones apreciables. La colaboratividad se logra con un sistema de sincronización en red que permite configurar la generación del sonido desde varios ordenadores simultaneamente conectados en red.

En este desarrollo nos centramos en el *framework* que provee el sistema. Por un lado, desarrollamos una bilioteca de procesamiento de audio utilizando los últimos avances en programación genérica. Además, construimos un entorno de síntesis modular, jerárquico y desacoplado con novedosas abstracciones de comunicación entre hebras para facilitar el desarrollo de procesadores digitales de señales altamente interactivos.

Todo el software desarrollado es libre y sigue una metodología de desarrollo continua y abierta. Es parte del proyecto GNU.

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

Juan Pedro Bolívar Puente

Keywords: modular synthesis, interactivity, collaborative systems, generic programming, real-time, networking, C++0x, free software, GNU

Abstract

In this mater's thesis project we develop a system for modular, interactive and collaborative sound synthesis and live music performance. The system is capable of mixing and manipulating sounds from files, a wide range of generators and can apply filters of all kinds. Interactivity is achieved with the *dynamic patching* technique, that allows altering the topology of the synthesis graph with simple movements on the synthesis modules, and also taking special care on the implementation such that any operation can be done in real-time without introducing noticeable distortions. Collaborativity is managed through a network based synchronisation mechanism that enables modifying the sound generation through several computers simultaneously connected through the network.

In this development we concentrate on the *framework* that the system provides. On the one hand, we develop a library for audio processing using latest advancements in generic programming. Then, we build an modular synthesis environment that is hierarchical and lowly coupled with novel inter-thread communication abstractions to ease the development of highly interactive digital signal processors.

All this software is free — as in freedom — and follows a continuous and open development methodology. It is part of the GNU project.

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.

Informa:

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 autoriza la defensa de dicho proyecto ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a 7 de Julio de 2010.

El director:

D. Joaquín Fernández-Valdivia

AGRADECIMIENTOS

Añadir agradecimientos.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE — A PERSONAL, HISTORICAL AND AUDIOPHILE DISSERTATION

> SIMPLICO: You need to walk before you can run.
> SALVIATI: No, you need something to run towards!
>
> *A mathematician's lament*
> PAUL LOCKHART

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 interested in the wild 70's explosion of musical experimentation, and specially 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 flats, with rather high ceilings and wooden windows, where many students live in Granada — when we played a video on Youtube 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?

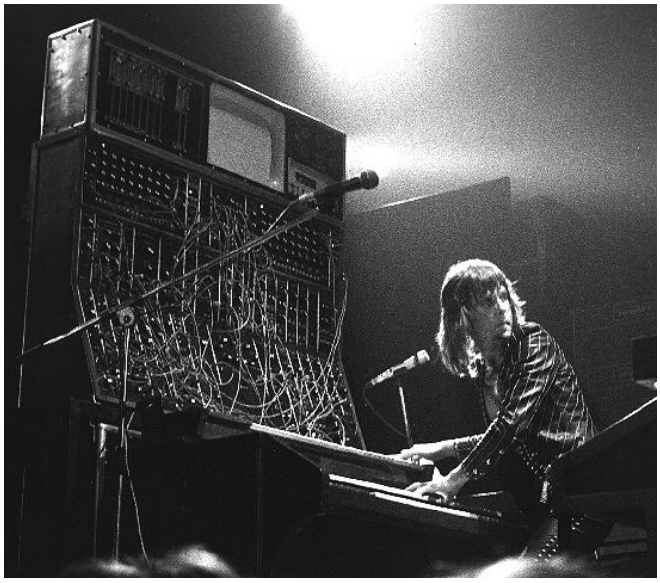


Figure 1.: Keith Emerson playing a modular Moog. The keyboard is connected to a big rack of modules, interconnected with eachother with wires. Each module generates or processes the sound carried in analog form in the interconnecting wires, therefore having an exponential number of possibly different 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 chronological development. Beginning with some *musíque concrete*, then moving 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. Early electronic music artists show a curiosity and naivety that is specially attractive for the educated in rock or classical music. They started by adding subtle electronic sounds but then soon played — in the most childlike sense of the words — in long composi-

tions delighting the listener with a lot of knob-twisting, pushing buttons and (now primitive) effects. It was the birth of post-modernism and continuous fusion — Spanish band Triana with its merge of flamenco, progressive rock and electronic music is a good example. In the conservatories Wendy Carlos and others broke the boundaries often imposed by their colleagues opening ground for exploration with a more scientific approach. "The well tempered synthesiser" is a good example of this struggle for conceiving synthesisers as just instruments, bridging the gap between an engineer and luthier. Kraftwerk's own evolution from rock to drum-pads, vocoders and synthesisers allowed me to open my ears to more modern electronic music. Their repetitive evolving patterns, symbolise a sort of a nostalgia for fordism in the transition to a *liquid society*, using Zygmunt Bauman terms.

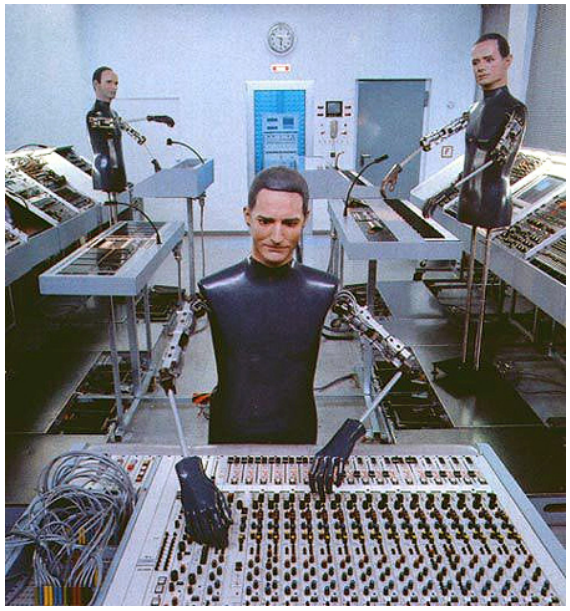


Figure 2.: Artwork for the music of Kraftwerk, with robotic representations of the band members playing with electronic instruments. On the front we can see an old analog sequencer.

Note 0.1 (Kraftwerk and the first analog sequencers)
In figure 2 can see an old analog sequencer, As opposed to a synthesiser — 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 (synthesisers) it is connected to. While current digital sequencers are very powerful, at that time they had important limitations that influenced Kraftwerk's robotic yet 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.*

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 [39]. In a dark room only illuminated by a bright blue table, several performers placed tagged objects on the table. It 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, "follow the old hacker saying," I thought: *"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 the 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

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

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 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 again and again and again — this project is actually just yet another iteration in this cyclic process.

Note 0.2 (Autechre and generative music)
Figure 3 show a couple of modular synthesis patches related to Autechre for doing generative *music. 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 pioneered generative music, using fractal-like sound structures that follow stochastic processes to create futuristic sonic atmospheres. Indeed, they are considered one of the most innovating bands in the 90's. We should do some justice mentioning Spanish band making experimental industrial music:* Esplendor Geométrico. *The careful listener can find a lot of influences from this band in Autechre's work, something they have acknowledged themselves.*

Max/MSP and its Free Software counterpart Pure Data are graphical data-flow programming software, *which can be considered a sort of general low-level modular synthesizer. They are 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

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.

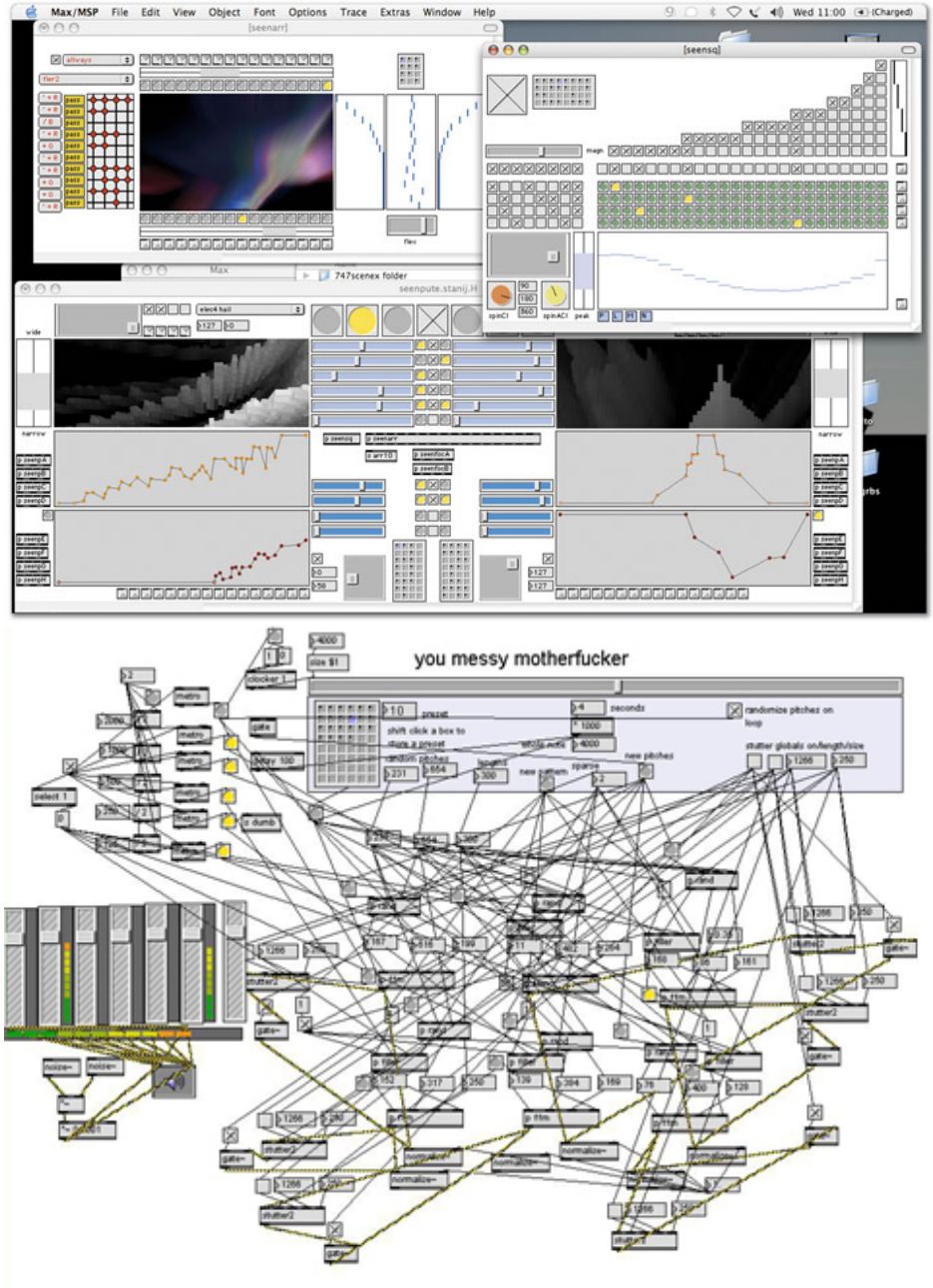


Figure 3.: The first picture is a Max/MSP claimed to be made by the electronic music duo Autechre [83]. The second is a Max/MSP patch made by Robbie Martin that *generatively* reverse-engineers Autechre's song Powmod.

by Autechre's fractal glitchs and Aphex Twin and Squarepusher's spastic patterns[3]. 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 only to realise that the road to get something useful out of Psychosynth was rather long yet.

The project won a price in that Free Software contest and it got some reviews in blogs and online journals [64, 44]. 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 instigated by an increasing interest in programming language theory and functional and generic programming, I also became more and more conscious of the flaws of the code — the lack of synchronisation mechanisms, MIDI (Music Instrument Digital Interface) support, pluggable modules, patch persistence, etc. turn 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 the 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 [6]. Also, the people at

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 crazy recursively modulated sinusoids.

4 http://soundcloud.com/shaker08

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, what's most needed, insight from a musician point of view.

So here we are now, in the fall of 2010. We are still quite ignorant in music making but pretending to be a "digital luthier" motivated by passion for music — and attempting to turn all this personal playground game into a final master's thesis project. Lets see how it goes...

5 http://www.artquimia.net/

1 INTRODUCTION, DEFINITION, 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 contained 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 usually come with similar modules too, but sometimes they use different terminology that, unlike in this list, is not related to voltage based signal processing.*

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 [56]. There, the sound signal is most 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 technique 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 keyboard-controlled instrument with a natural sound 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 to represent signals of unconstrained kinds. We can also instantiate copies of the modules as we wish only bounded 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 of the synthesis graph.

1.1.2.1 *Dynamic patching*

The *dynamic patching* [42] 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 4. 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 surface can help the artist to achieve new innovative ways of producing variations in his music.

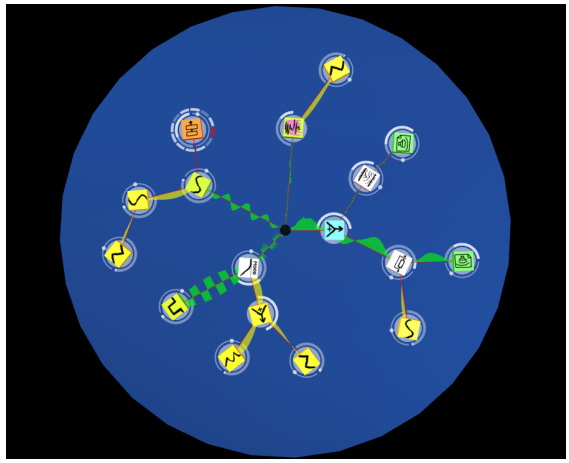


Figure 4.: A screenshot of Psychosynth 0.1.1 using dynamic patching to connect a complex graph.

1.1.2.2 *Touchable interfaces*

When striving for interactivity, 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.

There is an increasing availability of multi-touch interfaces, either specifically designed for music performance like the Lemur (figure 5) or general purpose ones like Apple's IPad or Samsung's Galaxy. Using those, 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 because several functions can be reached in the same space.

Figure 5.: 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.

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 directly telling the instrument

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

what notes to play [51]. There exists many kinds of electronic devices that simulate the feel of a keyboard or a drum-kit (figure 6) but instead of producing any sound they send MIDI [55] messages to a synthesiser that reproduces the notes.

Our software should, in the future, 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.

Figure 6.: 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.

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 are added implementing not only synthesis, but also sampling and sequencing.

This integrated environment should be able to be manipulated by several people at the same time to allow a 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 7, with such an intuitive and physically unconstrained interface several people can stand around the device and manipulate it.

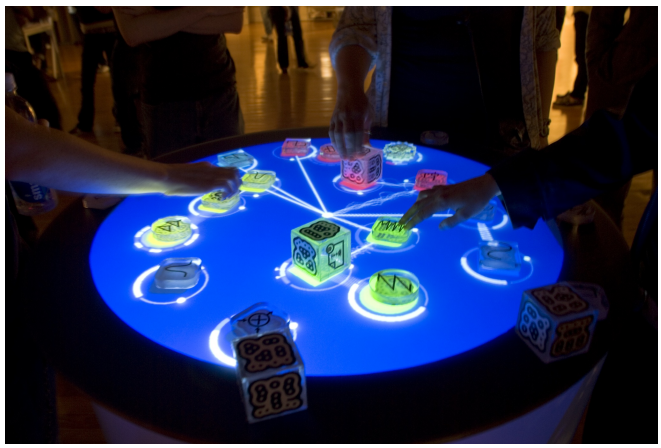


Figure 7.: An example of the Reactable being used collaboratively by several people.

1.1.3.2 *Networking*

Networking support releases a device even further 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 performing in a reasonable distance 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 8 shows an example of this feature being used live.

Figure 8.: 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 [76] 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. Not all of these objectives are to be accomplished in this mater's thesis project though, as we will further discuss in the analysis section. However, they provide a long-term perspective of the project and, on every step we take, we should take into account that these is what we are striving for.

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, mostly, 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 would be, 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. On the other hand, 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 student 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.*

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

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

techniques of the underlying programming language C++, a fact that is even 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 main objectives during the project and play a leading role in its overall success.

1.2.3 Partial objectives

A more precise 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 [56]. Since then, a wide variety of analog modular synthesisers have been developed commercially, but dragging some limitations as described in the introductory 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 [67]. This is a data-flow 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 3 in the preface showed an example of its functioning. The author of Max/MSP later developed a Free Software counterpart called Pure Data [66] that also has video processing and synthesis support and is widely used nowadays.

Since then, many user-oriented commercial modular synthesisers have been developed. One of the most famous ones is Native Instrument's Reaktor (figure 9). 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 multi-touch graphical interface on top of a modular synthesis environment.

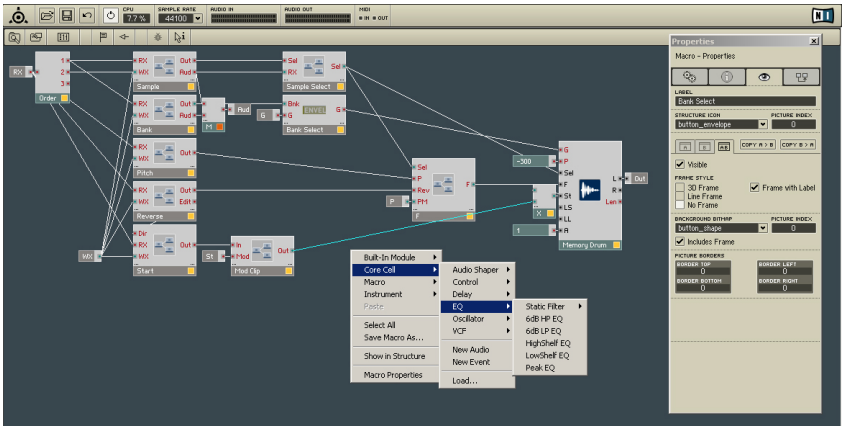


Figure 9.: Native Instrument's Reaktor editing a *core cell* based patch.

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 has been quite inspiring in our work. 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 system management that has a similar rival only in latest OSX versions: Jack [48]. 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 [34].

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 broad but not very updated

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

survey that gives some taxonomical background and analyses a wide variety of products can be found in [15]. One of the first attempts in using them for improved interaction in musical software is the Audio Pad [63], 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 [14]. Many videogames and other kinds of applications 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[6].

Maybe the most interesting example, which inspired the whole development of GNU Psychosynth, is the Reactable project [39]. 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 [9]. In the Reactable system a translucid round surface hides a projector and a high resolution camera underneath, as shown in figure 10. 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 (Open Sound Control) [20] based messages to a synthesiser following the TUIO (Tangible User Interface OSC) protocol [41]. The literature describing the initial prototypes recall Pure Data as the underlying implementation language for the 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.

The Reactable has been very successful and a very gratifying fact is that the computer vision component of the system — the one used to

6 A rather comprehensive listing can be found here http://mtg.upf.edu/reactable/related.htm

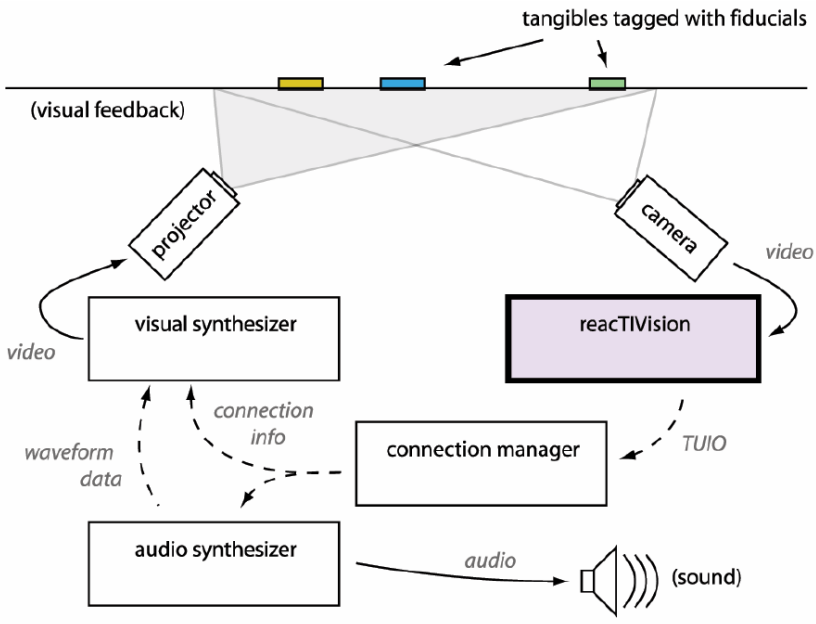


Figure 10.: A typical Reactable setup. Source: The Reactable project.

track the fiducial and finger tips on the table — 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 [60]. The former offers a fully customisable OSC based multi-touch interface. Using that, 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 11).

Nowadays, multitouch interfaces are the main trend, specially after the explosion of the tablet market, but big music software companies still have to catch up with the latest hardware developments. Specially interesting for the GNU Psychosynth projects it the Indamixx 2 tablet,

Figure 11.: Electronic music artist Four Tet performing with a Tenori-On

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 in the future.

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) [70] but it lacks proper dynamic signal routing mechanisms and some subtleties, like the sample format being hard-coded to float, seem too constrained for our purpose.

A popular sound synthesis and music composition DSL (Domain Specific Language) is CSound, which later was extended with a C and C++ programming API [17, 18]. 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 to be built on top of GNU Psychosynth. Still, some of them are worth mentioning. Notable examples are SuperCollider [52], which was for a long time the reference sound synthesis DSL; Chuck [92], that adds concurrency and time control

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

in an elegant and boilerplate-less fashion, and the newer Faust [62], 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 it 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 [82] 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), an award winning library based on modular synthesis [5]. Its design will be carefully taken into account during the redesign of few of Psychosynth core components. Still it does not fully fit our needs as it is too oriented towards research simulations more than live interaction, and it does not support polyphony.

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

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

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 our sound synthesis 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 and that will guide our 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 2.4. We will try to avoid describing the requirements related to those ready facilities in this section, as the only purpose of the requirement modelling 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 for a productive usage.

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 asynchronous 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.*

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

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 jump to zero assumed by the sound-card during the period that 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 because 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 an 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 several 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 tradeoffs, 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 requires some sort of automatic semantic impedance resolution system among the interfaces of Psychosynth nodes and the third-parties'.*

Requirement 13 *The system should be able to dynamically load modules — i.e. plug-ins — 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 a 2D euclidean space) input port of the same type not belonging that is free — i.e. is not connected already to a closer node — and that does not create cycles.*

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, some 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, in order to perform the dynamic patching.*

While this may change due to design considerations, we suggest specifying a set of tags for each port, with the following connectability rule for ports A and B:

$$connectable(A, B) \Rightarrow tags(A) \cap tags(B) \neq \emptyset \tag{2.1}$$

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 without explicit interaction.*

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.*

For example, the parameters of an LFO should be fixable such that the wave length is a factor of the tempo, and the phase is synchronised to a bar 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 random 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 supporters of the Free Software movement. We believe that respecting user freedom is specially necessary in an academic and public environment like ours, where open access to the generated knowledge should be expected by the taxpayers that are, indeed, the founders of our work. 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. Also, 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 license text

can be downloaded from: http://gplv3.fsf.org/. It can be found in the LICENSE file in the source code tarball included in the CD addendum.

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 third-party software that is attracted by these to be released as Free Software too [74]. 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 [76]. 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 several 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 at the Savannah software forge — the GNU project official forge — and its project page can be accessed here:

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

The project is using GNU Bazaar, a distributed revision control system, as source code repository. 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 volunteer driven Free Software 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 process. Most of the time, it is technically oriented and it can serve as source of motivation for external developers to contribute to the project. It also provides a fresh summary of the current status of the project. It can be accessed through:

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

More recently we started a micro-blog in the free and federated *StatusNet* microblogging network:

`http://indeti.ca/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 email management issues to some users, the Gmane project[7] offers a newsgroup

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

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:

- Users mailing list `nntp` interface:
  `gmane.comp.gnu.psychosynth.user`
- Developers mailing list `nntp` interface:
  `gmane.comp.gnu.psychosynth.devel`

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 its evolutionary design. Some of this flaws are nonetheless going to be solved in the next standardisation of the language, to be released in 2011 [79].

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.

Note 2.4 (On the new C++0x standard)
This section was written a the beginning of the project in Autumn 2010. During the development of the project, the ISO standardisation process for the new C++ standard is almost over. There is a final draft (a FDIS, Final Draft International Standard) so no modifications in the standard are to be expected [80]. The FDIS will get ISO's final signature and be available for sale within weeks or months — since late June 2011 when this note is being written.

Also, at this time, GCC 4.6 is the default compiler in Debian Sid and the last Ubuntu version, so its new supported features — like proper weakened conditions for container value types, range based `for`*, full lambda support, type inference, among others, are being used in parts the new code. More specifically, code written for the work described in chapter 3 depends on GCC 4.5, and this requirement was changed for the code developed for chapter 4 to GCC 4.6, when it had already become widespread in Debian based distributions.*

Note also that because the fact that the new C++ had a final standard was only known during half of the development of this project, it is called both C++ox and C++11 in this document, and the terms are used interchangeably.

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 as suggested by requirement 25. 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 to satisfy this constraint.

Still, we will try to comply with the POSIX standard [65] 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 [51][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.

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.

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 paradigm 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.

We use few of the Boost facilities, some of them including `boost::filesystem`, `boost::thread`[10], `boost::mpl`, `boost::test` — a wonderful unit testing framework — among others.

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

2.3.3.2 *Libxml2*

We use Libxml2[11] 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[12] 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 soft copyleft LGPL 2 and 3 licenses.

2.3.3.4 *LibVorbis*

We use LibVorbis[13] for reading OGG files. It is released under the permissive BSD license and is very portable.

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

2.3.3.5 *SoundTouch*

We use SoundTouch[14] 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[15] [16] 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. All of them are distributed under the LGPL2+ license. 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[17] is used for OSC support. It is licensed as LGPL2+ and is POSIX compliant.

2.3.3.8 *The user interface libraries*

The user interface included at the beginning of the project is based on Ogre3D[18], using OIS[19] (Open Input System) for keyboard and mouse input and CEGUI[20] as widget toolkit.

14 http://www.surina.net/soundtouch/
15 http://www.breakfastquay.com/rubberband/
16 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
Ardour recently moved tater moved to this library: http://www.ardour.org/node/1455
17 http://liblo.sourceforge.net/
18 http://www.ogre3d.org/
19 http://www.ogre3d.org/
20 http://www.cegui.org.uk/

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 graphically appealing, 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 doors to the yet-to-come wide range of Meego based tablets.

2.3.3.9 *Build system*

We use GNU Autotools[21] 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. Moreover, Autotools are suggested in the GNU Coding Standards [75] that we shall follow during our development due to our affiliation to GNU.

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[22] to embed the API documentation in code comments. A reference manual generated by Doxygen will be included in the CD addendum.

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

22 www.doxygen.org

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 12 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 [23] or in appendix **??**. In the project web page and the addendum CD demonstration videos and other multimedia material will be included. The article [68] published in the Novatica journal in 2008 gives a good introduction to the project too.

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 [27].

We use the term *library* when the root of the call graph is on the client side and the client invokes the library functions 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.".

23 http://psychosynth.com/index.php/Documentation

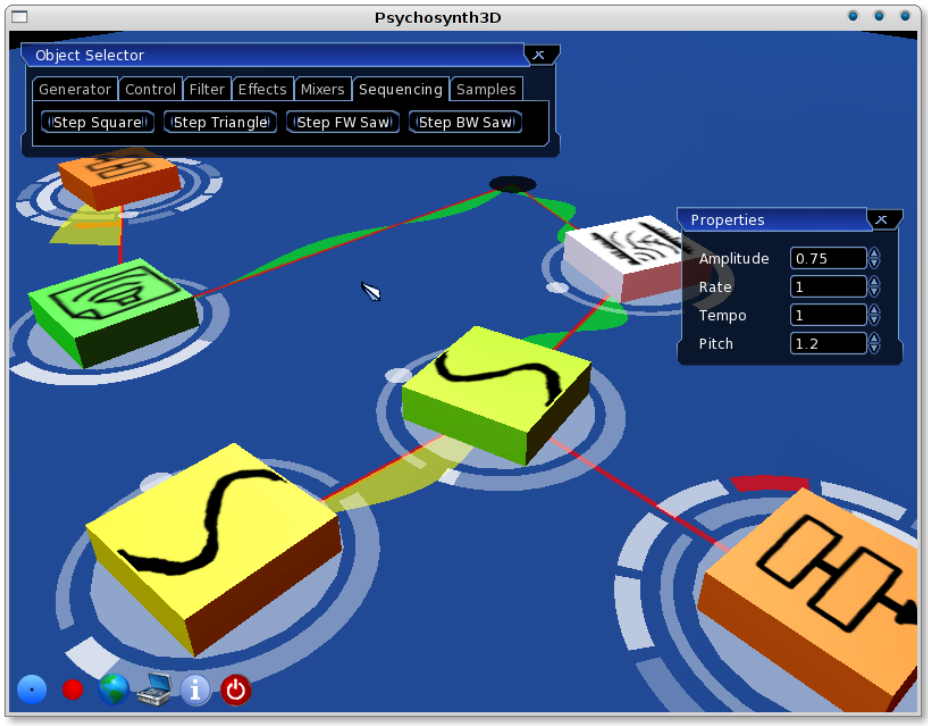


Figure 12.: A screenshot of GNU Psychosynth 0.1.4.

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 [28]. This tries to achieve 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 she is only using some bottom layers. Still, note that the library is currently compiled will all layers bundled in the same shared-object file, so this is not an advantage yet. Because the heavy

redesign ongoing during this project, we shall postpone that until later development stages when layer interactions are clear and stable.

Figure 13 represents the current layered architecture. The vertical arrows represent function calls. The big arrow that crosses through all layers represents the fact that all layers are allowed to use the facilities of the *base* layer. Thus, a layer is allowed to use the facilities in the layer below and the *base* layer only — e.g. the *graph* layer uses the `synth` and `base` API's in its implementation. Lets make a bit more in-depth discussion of each layer.

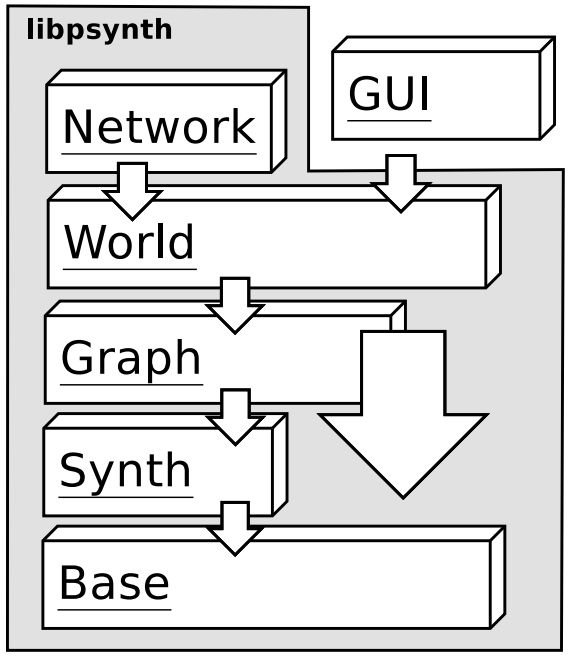


Figure 13.: The GNU Psychosynth layered architecture.

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 are:

CONFIGURATION PERSISTENCE Classes for representing program settings in a hierarchical manner. This configuration system can use interchangeable backends and has an observable interface.[24]

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. Maybe in the future it will be moved to the thin *app* module, but it is kept 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 implementable generically in some cases. Andrei Alexandrescu proves this point for C++ in [3]. This layer provides design pattern generic implementations inspired by Alexandrescu's approach. Some of the included facilities are implement *observer*, *singleton* and *composite*.

COMMAND LINE ARGUMENT PARSING While we have considered moving to Boost's Program Options library[25], 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.

24 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[27].

25 http://www.boost.org/doc/libs/release/doc/html/program_options.html

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. Indeed, this layer should be divided in three parts: the sound representation library, the IO classes and the synthesis algorithms.

2.4.2.3 *The graph layer*

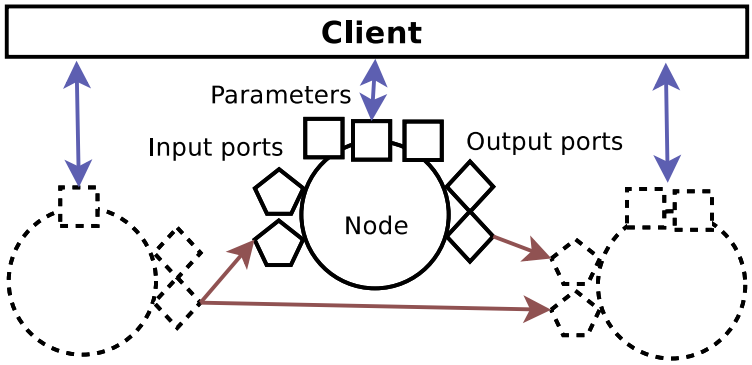


Figure 14.: Representation of the node graph as in Psychosynth 0.1.7. Input ports are represented as pentagons, output ports as rombus and parameters as squares.

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 14 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 15. Values are copied to and from an intermediate channel between the audio processing blocks.

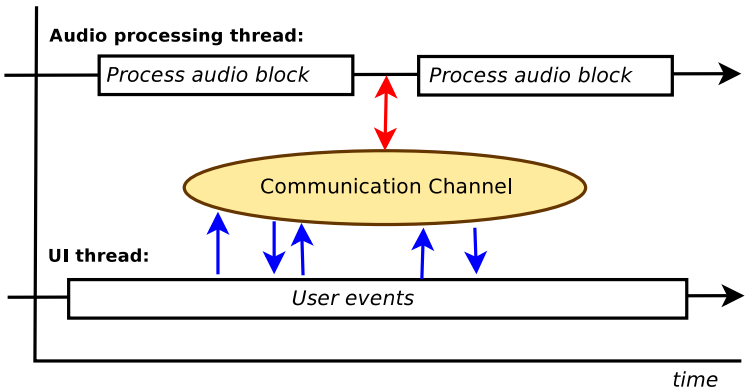


Figure 15.: Communication between the audio and user interface thread as in Psychosynth 0.1.7

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 Model View Controller that the system implements. Figure 16 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.

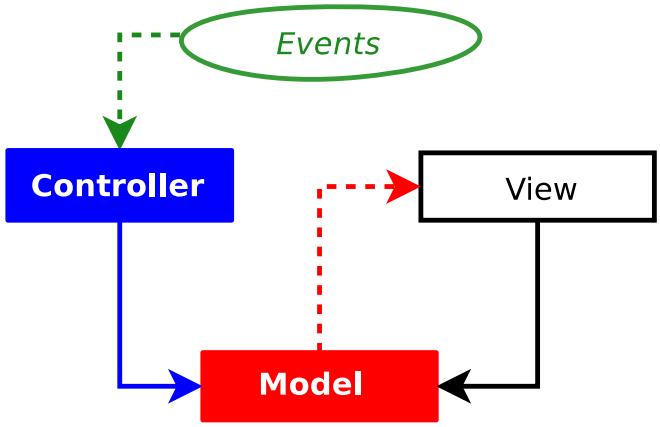


Figure 16.: The Model View Controller arquitectural style. Dashed arrows represent indirect invocation —i.e. via the *observer* design pattern— and solid lines represent normal method calls and data access.

The model changes when controllers invokes methods on it, several 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 in 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 some built-in views and controllers*

There is a thin layer, instance of the *facade* pattern, called `app`. It was hidden for simiplicity in figure 13 representing the layered architecture. It sits on top of the `world` layer and is in charge of initialising the *world*,

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`. This in turn delegates the heavy work to user code that is hooked as method overrides of that same class.

Orthogonal to this layer, and also sitting 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 — thus enabling 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 component to function properly. One could develop a new experimental and cool UI and it would automagically be optionally able to work with third party clients over the network, even potentially using a different user interface.

In its current version, on top of all this, there is the code of the 3D user interface and some 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. Instead, the framework remain user interface agnostic, so we will not further describe the UI code as it falls outside the scope of this project.

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

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's thesis project. In our school we study with most detail the Waterfall Model [10] and the Rational Unified Process [47].

Those development processes propose a fordist software production model, targeted at huge development teams and the development of stable code bases in non-experimental, well defined, fields. Many of their proponents state that software engineering is like any other engineering where creative analysis and design is only the first step — thus they believe programming is analogous to construction. Fowler makes a great point [24] criticising this argument, 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 indeed. 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, these 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 completely substitute a software development process 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. Agile methodologies put most 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, providing no additional value. 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.

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.

Consequently, 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.

27 http://agilemanifesto.org/

2.5.2.1 *The design phase*

In the design phase we shall define the API that we would like the current subsystem to 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 programming 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 documentation should include rationale explaining why the design decision lead to the satisfaction of the requirements assigned to that 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.

2.5.2.2 *The implementation phase*

During the implementation phase the code implementing the design should be written. It is possible and even sometimes 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 in the design document — even though an API may be designed through an inductive empirical process, a deductive rational description may be more useful for its clear understanding indeed.

We are keen on Test Driven Development [8]. This methodology suggests that unit tests should be written before the actual implementation for the tested interface is written at all. Instead, a mock implementation

is to be provided. Once the tests compile — and fail — implementation is started concentrating on making the tests pass. Once the tests do pass, the code and internal design is improved via *refactoring*. We will not follow this methodology dogmatically — specially when exploratory programming is required — but we suggest to follow it whenever possible. Thus, we can consider than the verification phase and implementation phase are, or at least should be, overlapped.

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 must be modified such that they use the new code. This might even be sometimes considered part of the verification, as older tests working on the upper layers should be checked to be working after the integration.

Informal integration tests should be done on the final user interface to make sure that the properties of the older implementation 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

In the following we propose a project plan to accomplish the requirements specified at the beginning of this chapter. As we stated in the introductory chapter, this is a long-term project, and the required effort to fulfil all the proposed objectives clearly exceeds what a student can do in parallel to his normal studies[28]. *Thus, only the two first iterations are to be developed in this mater's thesis*. This structure fits very well in the Spanish university course structure; the first iteration shall be developed during the first semester and the second iteration during the second and last semester.

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

DESCRIPTION In this iteration involves a deeply re-design the core data structures using the latest techniques in C++. This requires special research. Performance requirements deeply rely on the success of this iteration.

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

ESTIMATED TIME COST: 4 months.

28 The student involved in this project, apart from being studying the normal courses for the 5th year in "Ingeniería Informática", he is also collaborating with the Computer Vision Group in the development of Moodle plug-ins (http://nongnu.org/cvg-moodle/), and has been hired by the Institute of Astrophysics of Andalucia to held a course in Advanced Python Programming in May (http://sinusoid.es/python-avanzado/). These time constraints must be taken into account in the project planning.

2.5.3.2 *Second iteration: Redesign of the `node` layer*

DESCRIPTION The `node` layer requires a redesign if we want it to scale to satisfy all our long-term 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. The whole layer should be reimplemented and validated.

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

ESTIMATED TIME COST 4 months.

2.5.3.3 *Third iteration: Dynamic loading of nodes*

DESCRIPTION In this iteration the plugin system is to be developed. This involves loading of nodes with our own interface — as designed in the previous iteration — and with external interfaces, implementing interface adaptors when needed.

CRITERION Requirements 12 to 15 should be satisfied for its success.

ESTIMATED TIME COST 3 months.

2.5.3.4 *Fourth iteration: Adding MIDI and synchronisation*

DESCRIPTION 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.

CRITERION Requirements 17 to 22 should be satisfied for its success. Requirement 11 might be implemented in this iteration too.

ESTIMATED TIME COST 5 months.

2.5.3.5 *Post mortem analysis*

After the development of those previous iterations, we should write a conclusive report and evaluation of the project's success. Also, we should prepare a final presentation for the projects evaluation.

Note 2.5 (Structure of the rest of the documment)
The rest of the documment is devoted to documenting the modules developed in the iterations that are expected to be developed within this master's thesis. Chapter 3 describes the first iteration involving the new sound system. Chapter 4 explains the new graph layer as developed in the second iteration. Finally, chapter 5 recapitulates and will provide some sort of post-mortem analysis — of these two iterations — also preparing the ground for the future development of the project.

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 is: 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, we shall debate it with care.

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 [29] — of sound by measuring the successive oscillating values of air pressure in the observer's point in space. Note that a constant 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*, and 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 vice-versa [23]. The *sound pressure level* (SPL) is a logarithmic measure of the effective sound pressure of a sound relative to a reference value. It is measured in *decibels (dB)* above a standard reference level.

Most of the time, in hardware, we represent the SPL value as a voltage value, that varies within some range — e.g. $[-5, 5]V$. This is an analog 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 measure the current voltage or air pressure value. Figure 17 illustrates this, every equally spaced vertical line is a discrete instantaneous sample and the information in between is lost. The value of the signal between two 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

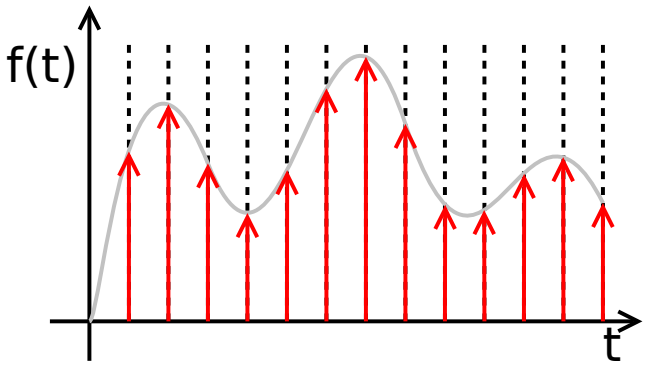


Figure 17.: Temporal quantisation of continuous 1-D signal.

equivalently, when the *Nyquist frequency* (half the sample rate) exceeds the highest frequency of the signal being sampled. Because the audible frequency range in best hearing 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 (DVD), 96 kHz (DVD-Audio, Blue Ray) and 192 kHz (High Definition DVD-Audio, Blue Ray). Sampling rates bellow 44.1 kHz are used also in old computer games that were limited by the computing power, and in 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 a finite and discrete scale. Figure 18 shows a linear spatial quantisation, represented by the dotted lines parallel to the abscissa axis. That is a best fit quantisation assuming a time continuum, in practise, spatial quantisation is applied over an already discreet time,

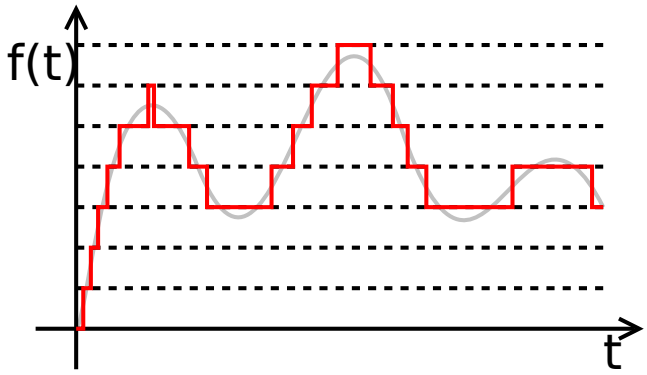


Figure 18.: Spatial quantisation of continuous 1-D signal.

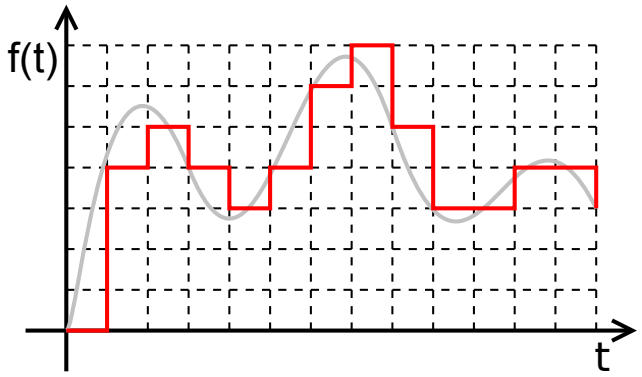


Figure 19.: Fully quantised continuous 1-D signal

figure 19 combines the quantisations of figures 17 and 18. In a computer system, spatial quantisation is mostly determined by the size in bits of the underlying type used to store the samples. Also, the quantisation is affected by how we space the different plausible values. In practise, linear quantisation — i.e. equally spaced samples — is most used in audio formats targeted at music and multimedia, but logarithmic and other non-linear divisions are common in telephony and other lo-fi systems. Note that using floating point values to represent samples, as it is often done in software synthesis, provides an implicitly logarithmic audio quantisation. 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 find 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 (SNR) 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 [26]. Digital audio CD have a theoretical dynamic range of 96 dB — actual value is around 90 dB due to processing. Human hearing pain threshold 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 [50], 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, the algorithms implementation is much harder to encode with fixed point arithmetic because products yield greater values and there are many issues 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 [72]. Another factor is the *endianess* of fixed point values but is this relevant only when interfacing with file formats and output devices.

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

The last concern related to quantisation is *dithering*. Dithering is the process of adding a white noise signal in a range similar to that of the quantisation step — e.g. $[-2^{-n}, 2^{-n}]$ for a linear n-bit quantisation — prior to doing the quantisation itself. While this increases the overall SNR, the perceived subjective distortion is much smaller as harmonics introduced by the regularity of quantisation error are softened; with a proper noise model we can even push the harmonic artifacts outside the audible range to achieve minor distortion [87, 49].

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 sound has a 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* to characterise the temporal quantisation regardless of the number of channels in the sound signal. Under this definition:

$$sample\_rate = num\_channels \times frame\_rate \tag{3.3}$$

Still, many texts use the term samples rate and frame rate interchangeably, so one must be careful and pay attention to the context.

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.

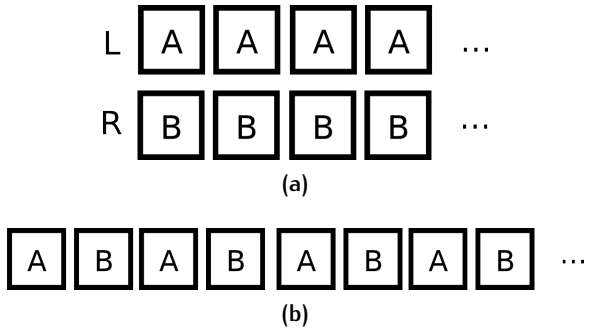


Figure 20.: Multi-channel data in planar (a) and interleaved (b) form.

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 20 compares the interleaved and planar formats visually.

Another issue is the order in which the data from different semantic channels is stored. We call *channel!layout* to 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 is 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 factors affect audio representation inducing a combinatorial explosion of format conversion functions. If the reader wants more evidence, we invite she to take a look at the 128 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, are simply not an option in any audio software which supports real-time operation because of the associated overhead.

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 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 problems and thus share a lot in the representational issue. By realising a proper conceptual mapping between both worlds (table 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.

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 *colours* that, for example, capture the

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

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

Table 1.: Terminology map from `boost::gil` to `psynth::sound`

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 all the statically arranged sound synthesis and processing.

3.2.1 Core techniques

The Boost.GIL and thus the Psynth.Sound modules design make 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 [40] 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 related free functions, operators, nested types, etc. (2) While dispatching through interfaces requires, at least, a dereference, addition and function call [22], 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 `instance` doing the job of `concept_map` [11].

Concepts are an extension to the template mechanism to add some sort of type checking for its template parameters. In fact, without concepts, checking and dispatching based on requirements can be achieved with techniques like SFINAE (Substitution Failure Is Not an Error) [88]. Property (5) of our concepts can be simulated with *traits* [57]. 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" [78]. 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] [71] 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 documentation we will use the concept syntax proposed in the proposal N2081 to the standardisation committee [30].

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

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

```
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 satisfy a 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 [40] being the most updated and useful from a programmer's point of view.

3.2.1.2 *Metaprogramming*

The C++ template system is Turing complete [90], 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 meta-program that outputted the first N prime numbers on the console using compiler errors [84]. 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 [3], where Alexandrescu uses them to instantiate design patterns as generic C++ libraries. A deeper reference is Abraham's [1], 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 of a template 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 compile-time function. It is up to the programmer to choose the naming convention for the result members of the metafunc-

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

tions. In the following, we will use Abraham's style calling `type` for result values that are a type, and `value` for integral value results. Listing 1 illustrates how we can write and use a metafunction for computing the n-th Fibonacci number.

Listing 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;
}
```

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 a recursive template to define the metafunction for the general case and then 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

Listing 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;
};
```

three meta types: types (which 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 are type checked 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.

In order to simplify our meta type system we shall wrap constants in a type like in listing 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 — like `constexpr` in C++0x supporting compilers. 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 meta-function is any type — template or not — that has a nested type called `type`.

Listing 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;
}
```

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 also illustrates the metafunction forwarding 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 modelling 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 use 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 [78].

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 `MPLIntegral-Constant` (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:

$$Min < Zero < Max \wedge \forall x \in Type, x + Zero = x \tag{3.4}$$

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 with the `sample_type` metafunction.

```
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 to 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 `HomogeneousFrameValueC` 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`. These iterators must provide some extra metafunctions, as in:

```
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 (i.e. a pair of iterators determining the *begin* and *end* of a sequence), 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 one 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 referred 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 21. This

data structure is specially important for passing audio data through different threads.

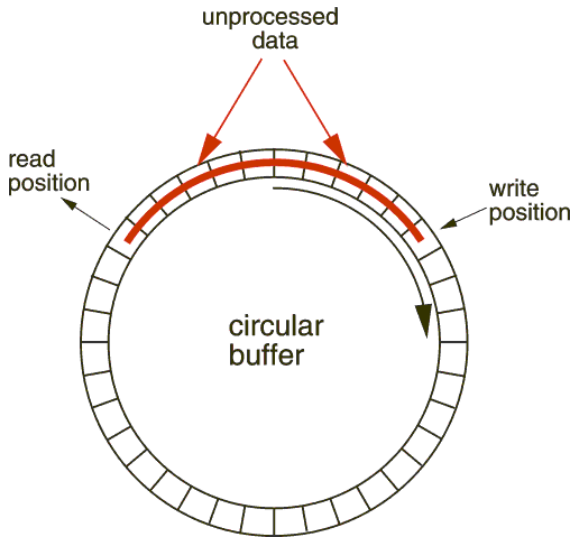


Figure 21.: Ring buffer operation.

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. There might be multiple readers at different pace in our ring buffers.

```
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; where PositionConcept<unsafe_position>;
    typename position; where PositionConcept<Position>

    typename unsafe_iterator;
```

```
    where RandomAccessIteratorConcept<unsafe_iterator>;
typename iterator;
    where RandomAccessIteratorConcept<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 <PositionConcept Position> range
R::sub_range_one (const Position& p, size_type slice) const;
template <PositionConcept Position> range
R::sub_range_two (const Position& p, size_type slice) const;

template<PositionConcept Position, class Range2>
    where RangesAreCompatibleConcept<R, Range2>
size_t R::read (Position& r, const Range2& range) const;
```

```
    template<PositionConcept Position, class Range2,
             class CC = default_channel_converter>
        where FrameConvertibleConcept<R::value_type, Range2::value_type>
    size_t R::read_and_convert (Position& r, const Range2& range,
                                CC cc = CC ()) const;

    template <class Range2>
        where RangesAreCompatibleConcept<R, Range2>
    void R::write (const Range2& range);
    template <class Range2, class CC = default_channel_converter>
        where FrameConvertibleConcept<R::value_type, Range2::value_type>
    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. Ring buffers support 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 whether is must 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.

Also, note that our ring buffers support backwards operation. This is useful when 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 usually do not make sense. The sub ranges should be used to pass its data to non mutating algorithms.

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 [77]. Hand coding most loops often makes user code not generic, because one easily 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 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 overloads 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` functions. 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 associated metafunctions

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 instantiations. The naming pattern for these concrete types is:

$$\begin{aligned} ColorSpace + BitDepth + [\texttt{f}] + [\texttt{s}] + [\texttt{c}] + \\ [\texttt{\_planar}] + [\texttt{\_step}] + ClassType \end{aligned} \tag{3.5}$$

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 and 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 `variant` subclasses `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` 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 polymorphism — 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 the concrete type that we want as an argument. This function object should provide an `operator()` overload for every 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;
```

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

```
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 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 [19] 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. Basic support for this technique is provide by our library. Anyway, whether by using this technique or by just avoiding dynamic buffers over too many types, special care should be taken when using dynamic buffers.

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

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

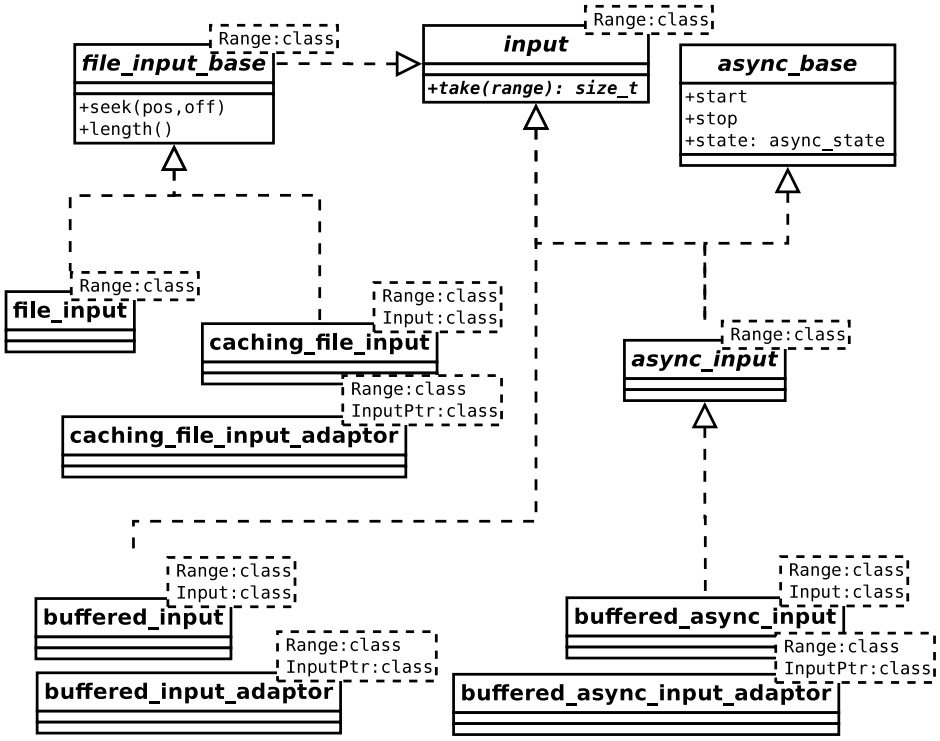


Figure 22.: UML class diagram of `psynth::io` input facilities.

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 22 and 23 show the class diagrams for the input and output parts of the module, respectively. We use the Unified Modelling Language notation. The doted boxes over the corner of some classes list the formal

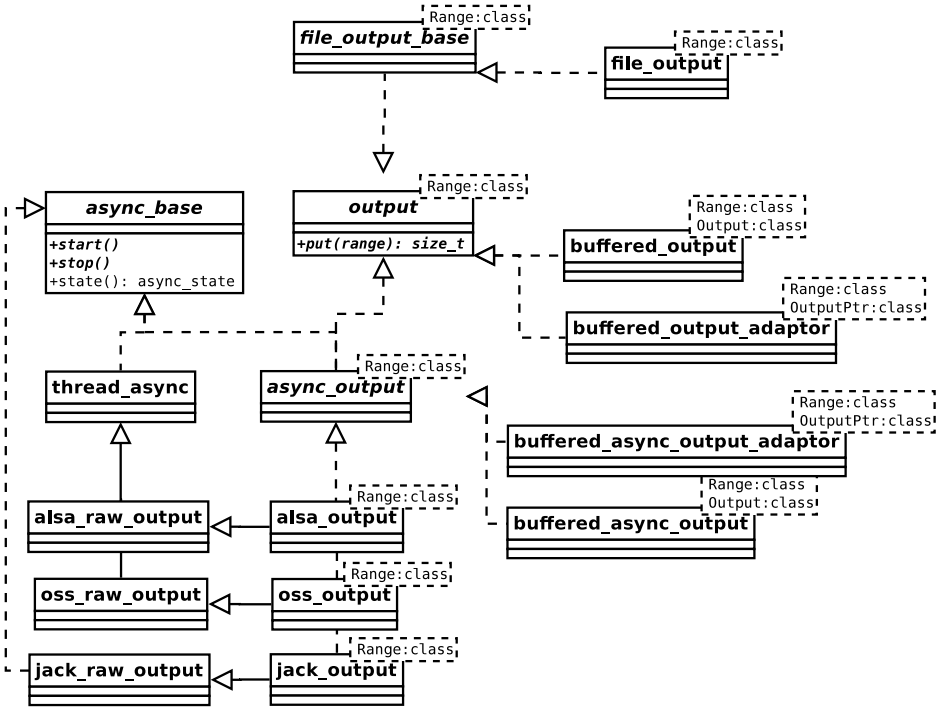


Figure 23.: UML class diagram of `psynth::io` output facilities.

parameter of generic entities, an extension that was introduced in the 2.3 version of UML [31].

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 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 supports 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 (Resource Adquisition Is Initialiation) — i.e. 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 the 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 — this contradicts dynamic buffers mode of operation with visitors that operate on a family of types. Indeed, the basic I/O constructs require their `Range` parameter to model `FrameBufferRangeConcept`. 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 I/O system 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.

In order to convert among formats, buffering uses the *sound::copy_and_convert_frames* function that we described earlier. This allows the user to provide a custom conversion functor to convert among very specific formats.

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 I/O systems 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.

The `thread_async` class uses a thread to provide this kind of asynchronous operation by and delegates the actual iteration inside an infinite loop to the derived classes. Optionally, this class will try to get real-time priority if requested[9]. Actually, we do request real-time priority in all the I/O devices based on the `thread_async` class.

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 were pro-

9 The `std::thread` standard class does not support changing its priority. We use its `native_handle` hook to request the new priority using the `pthreads` API directly when POSIX threads is the underlying implementation. At this moment, we do not support real-time priority on other platforms.

grammed 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, avoiding to use a worthless `buffered_*_output` adaptor.

Our current implementation uses mutexes and condition variables, which should be avoided, as we will deeply discuss in the next chapter. It is planned to fix this in the future, section 3.4.2.1 further discusses several solutions to this issue. Also, its implementation is designed to aid writing *samplers*. It has interesting benefits, such as allowing to read the file backwards. Also, when the change from forward to backwards reading is made, the front and back pointers of its ring buffer are flipped and the direction of reading and writing is reversed, such that the data that was already loaded in those big chunk pre-fetchs can be reused.

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) being 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.

The modules described in this chapter are evaluated with a total of 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 suites involving these modules. Not all of those unit tests have been written manually. Using `BOOST_UNIT_TEST_TEMPLATE` one can define a test case parametrised over some type variable, 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. This kind of report may be generated passing the `--report=detailed` parameter when running the test script directly.

```
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 4 shows an example of such generic function that you can compare with its concrete implementation in 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 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};
});
```

Listing 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;
}
```

We check the performance on some different cases that try to stress different potential overhead corners, like using a different layout from the natural order in the channel space, planar and interleaved buffers, etc. Each test case tests a kind of loop abstraction 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 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 the code with different versions of GCC options `-O3 -std=c++0x`. Tables 3 and 4 show the results for GCC 4.5.2 with 4096 and 32 buffer size respectively and 5 and 6 show the results for GCC 4.6.0. The acronyms for the concrete parameter types are expanded in table 2. All the timings are represented in *milliseconds* computed as the mean of all iterations. The "gain" is defined as:

$$Gain = \frac{T_{non\_psynth}}{T_{psynth}} \tag{3.6}$$

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 large 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. Indeed, the results

are more even with this compiler in a probable tendency towards 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 then 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[10] 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 and deployment

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.

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

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.: 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 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 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 6.: Performance tests for 32 buffer size with GCC 4.6.

The system was then *peer reviewed* by project collaborators, mainly by the maintainer of the Ubuntu/Trinux packages Aleksander Morgado[11]. After minor bugfixing, we agreed to make a new Psychosynth 0.2.0 release that included the 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.*

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

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 research oriented, and has a lot of value in its novelty. This, admittedly, delayed our development more than expected in our original plan too.

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. This leads 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 that of 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 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* [4].

Even though we believe that 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. Falling outside this project's scope, there are still some possible improvements such as:

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 [43]. Our output subsystems execute the audio processing callback in a real-time mode thread whenever possible, and thus the problem can become significant.

Locking is done with care and in practice we haven't experienced any buffer underrun due to this problem, even with high number of caching file readers in execution. However, for correctness and better support of Jackd, we should implement a lock-free ring buffer [85, 54] that does not require using special synchronisation to support one reader and one writer on different threads. 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 — if not probably making a lock-free version completely impossible.

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 virtual ranges because, in GIL, 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[12] and Boost.Range[13] should be carefully evaluated.

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, at least, when making some compromises. Some of the issues Lubomir states have different answers for sound processing. In fact, allowing out of range values seems to be 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

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

a fixed point representation. Maybe, not all those questions have to be answered in order to improve the arithmetic support anyway.

If we where to implement such support, one of the main drawbacks when writing a generic sample 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, specially when nesting complex arithmetic expressions.

This is not a dead end. Using the *expression template* [89] 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[14] [59] could be of great help. Moreover, with careful studying of the audio DSL's discussed in section 1.3.3 and trying to mimic some of their interface, the scope of such effort could be broadened to build a full sound synthesis and processing EDSL for C++. Indeed, this would allow writing specific optimisations in this arithmetic layer itself. It would be specially interesting to optimise for the usage of SIMD[15] instructions of modern processors. While this can be done now in a per-algorithm basis, doing so in a frame-arithmetic layer would hide all this complexity to the DSP developer providing a significant performance boost for free.

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

15 Single Instruction Multiple Data. These instructions can operate on several arguments at the same time. They are very useful in multimedia applications. For example, all the samples in a multi-channel audio frame could be updated with one single instruction, multiplying the performance. The SSE family of multimedia extensions provided by Intel processors are is the most widespread set of SIMD instructions.

3.4.2.4 *Submission to Boost*

In our conversations with Lubomir Bourdev he suggested submitting our library for inclusion in the Boost package. 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 compatible with the C++03 standard, while our code requires C++0x. Moreover, our code has dependencies with other submodules in `psynth::base`, specially the exception 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.

Such a system can be characterised 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 24. We can then taxonomise its parts as in the following; we will later use this terminology in our design.

INPUT PORTS These are sockets for signals incoming 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 consider anything that can go through a wireas an input. Thus we are not limited to just analog time-domain signals, but, for example, 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 program 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 sockets for the signals that the module sends to others. 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 can be used instead if safe and applicable.

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.

Note that, while on a 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 —, software can relate them in a one-to-many fashion. This is so because 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 whatsoever —, but it is instead an abstraction that 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. Most of what have been said about parameter control applies here. Once again, what is important is the abstraction not

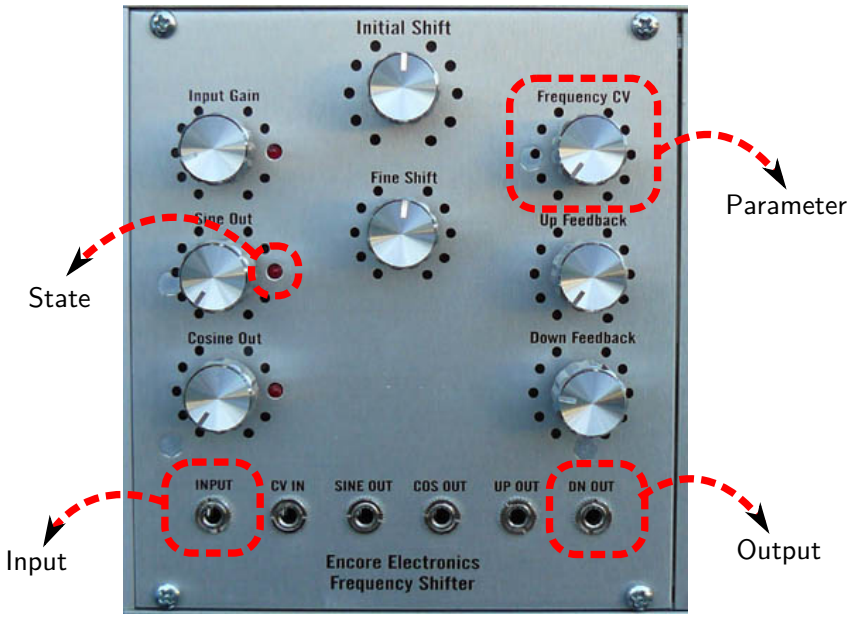


Figure 24.: Components of a synthesis module illustrated in a standard *frequency shifter*.

the visual representation; those the aforementioned example would 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 it is easy to 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 25 summarises all this. Note that this is a conceptual class diagram, not a design one, so there is not a direct match to the classes in the actual code, not even terminologically.

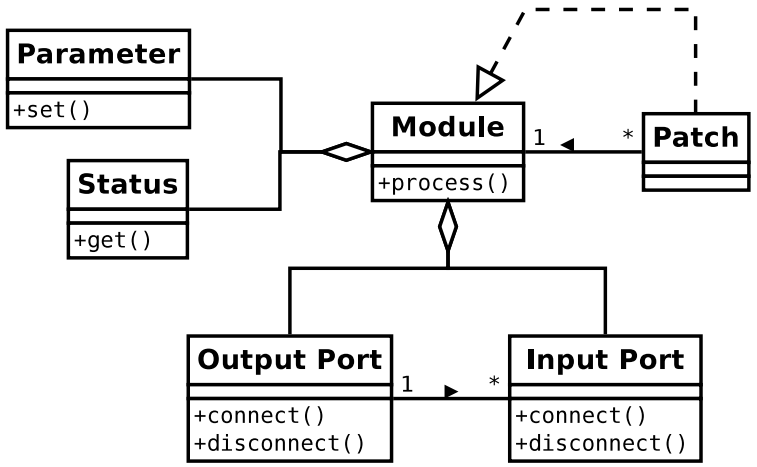


Figure 25.: Conceptual class diagram of the modular synthesis main entities.

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 4.2.5. 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 study 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 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. 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 — in a consumer quality system, we would need to produce 44000 samples per second! 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. Hence, 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's processing power. A sensible block size is 64 samples, like Pure Data's default. Using the terminology in [18], 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:

$$control\_rate = \frac{audio\_rate}{block\_size} \tag{4.1}$$

Note that control signals are not restricted to what we labelled as *controls* in the previous sections. In fact, through a *port* signals may flow 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 sound generation 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 last chapter, some audio output interfaces like Jackd control the audio thread themselves invoking the processing function through a callback. Our own device API wrapper that was developed in the previous chapter promotes this kind of asynchronous output and simulates it whenever the output system only provides a blocking I/O interface. The inverse of the control rate, that we may call the *control period*, provides a very explicit deadline for the 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]. Hence, our system can be categorised as *soft real-time* [81]. This implies that, 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:

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 [43]. 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 [25], 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 [86] or do conditional locking instead.

2 In fact, in the middle of a live performance with a 10^5 watts sound system, consequences can be more severe as it may seem superficially ...

3. Because the deadline depends directly on the block size n, algorithms running in the audio thread should be in $O(n)$. 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, like same-sized objects, a custom memory allocator can do the job [3]. 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 other ad-hoc solution.

4.1.3 Anticipating future needs

For the sake of proper workload balancing, several features that deeply affect the structure of this layer are delayed for later development iterations. However, to prevent rewriting the code at those stages, we should anticipate what characteristics are required from the basic constructs in order to later extend the system with such features.

4.1.3.1 *Polyphony*

Polyphonic audio allows building chords and enrich the whole harmonic depth of the music [38]. Polyphony is produced when several notes are played at the same time; for example, by pressing simultaneously several keys of a keyboard. Actually, polyphony is also present without simultaneously pressing several keys, as instruments usually have a

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

significant *decay time* — i.e. the time between the key release and the sound fully fades out — blending the sounds of successive key strokes.

Because in a modular synthesiser a generator is producing a single tone at a time, it is not trivial to implement polyphony. In practise, several copies of the processing graph are to be maintained, each one is called a *voice*. When a new note is to be played, the system has to allocate a voice for it and trigger the processing; when the key is released and the note fully faded out, the voice is released. Voice allocation is in many ways similar to other allocation mechanisms, like page allocation in an operating system, and different stragies have been proposed: *round-robin*, *last-in-first-out*, *priority based algorithms*, etc. The number of of voices is called the *level of polyphony*, which is usually user-controllable but fixed during a performance parameter of the system. Modern computers allow levels of polyphony high enough (from 16 to 128) such that the sophistication of the allocation mechanism is not too relevant for the overall qualitative experience of the sound produced by the synthesiser. Also, in a modular software synthesis engine, a whole network needn't be polyphonic. For example, some dynamic range compression or reverb filters are usually applied to the final mixed sound of all voices, yielding not only better performance but also a richer sound.

It is not worth to concentrate on the actual management of the different voices and the triggering mechanisms until sequencing and or MIDI support is developed in later stages. However, there are some questions that we do have to address now:

- We have to split the responsibilities of the different classes properly predicting which classes will need to be extended, preferably via inheritance, to provide the polyphony support.
- We have to define, at least partially, an interface that polyphony enabled DSP nodes will have to implement.

4.1.3.2 *Plug-in system*

At a later stage, we plan to implement a plug-in system as defined by requirements 12 to 15. These plug-ins shall be in several formats, like LV2, LADSPA, or our own defined interface. We should take this into account in several ways:

1. For consistency, the interface for implementing modules that we will define in this chapter shall be the same that our own plug-ins will implement later. The only significant difference should be the linking method of the code and the registration mechanism.
2. Even for statically linked DSP modules, bject construction should be done in an indirect way, via object *factories* [27]. Thanks to this, the system can later be transparently extended by modifying the factory manager to delegate the construction of specially-identified objects — objects identified by a resource URI [53] — to a plug-in loader.
3. Parameters, states, inputs and outputs of an object should be accessed via dynamic means of introspection. In practise, they should be identified by strings ands modules should provide some way of iterating through them. Ideally, there should be some hooks to provide some meta-data that can be useful for an automatic user interface builder; for example, a parameter's range, whether it is naturally linear or logarithmic, etc.

4.2 DESIGN

4.2.1 Overview

Figure 26 shows an overview of most important classes in this layer from the musical point of view. Most of the code implementing these classes lives in the `psynth::graph` namespace, named after the fact that this layer implements the notion of sound processing described as a graph of

interconnected processing modules. The `node` class is the base class of all kinds of vertexes in this graph. Therefore, *node* is a synonym for what we called *module* or *DSP node* in the analysis section; i.e. it is the basic unit that processes some input signals producing new output values. In the rest of this chapter, unless otherwise stated in its close context, we will use the term node with this meaning.[4]

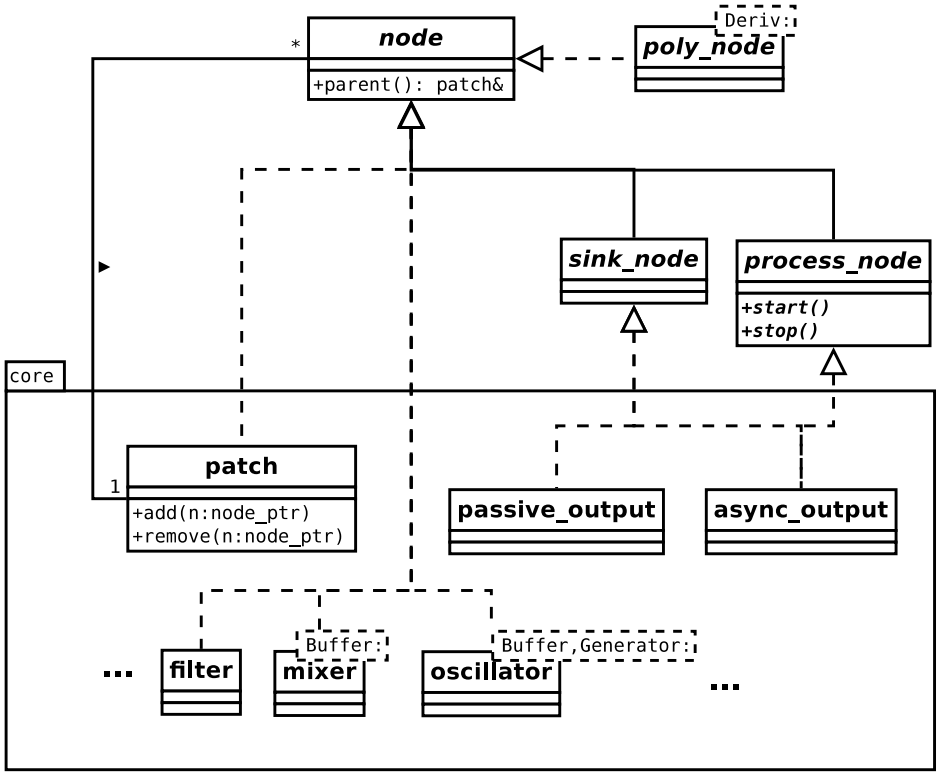


Figure 26.: Overview of the graph layer

The `psynth::graph::core` namespace contains the *concrete* nodes that are implemented in our library. All these shall be registered in the object factory that we will describe in section 4.2.6 and most of the code should not instantiate these modules directly. This introduces the notion of *core node*, i.e. a node that is built-in — statically linked — inside the library, thus it is not a plug-in. Other sorts of abstract nodes that are there only to

4 Indeed, we prefer this term as the name *module* has a different specific meaning in the code, similar to that of *translation unit*.

offer certain basic functionality should live directly in the `psynth::graph` namespace, and they are considered basic infrastructure and may be useful in the implementation of plug-ins.

We will explain other parts of this diagram later. What should be clear now is the relationship between a `node` and a `patch`. This is an instance of the *composite* design pattern [27, 91]. A *patch* is a collection of nodes but it is a node by itself, building tree-like structures that most of the time have non-patch nodes as leafs, and a patch on its root. To ease memory management, `node_ptr` is an alias for `std::shared_ptr<node>`[5]. A node keeps a weak reference to its parent easing traversal and some special operations, accessible via the `parent ()` method.

We will next explain the execution model of the synthesis network defined by these classes, but first, lets make a parenthesis to explain a novel data-structure that is deep in its core.

4.2.2 Heterogeneous deques — bridging polymorphism and real-time processing

As we explained in the analysis section, avoiding heap allocation is crucial in real-time processes. Proper runtime polymorphism is said to require heap allocation because the code paths in charge of managing the lifetime of some object do not know its full type — specially the actual object size — and thus the object can not be allocated in the stack. While true in general, we can constrain the problem in several ways to obtain $O(1)$ allocations if the programming language provides enough low level constructs to implement these solutions.

A possible solution is to override the operator `new` for certain types to provide constant-time allocation. Alexandrescu provides a clever way of doing this [3]. This, however, have several problems:

1. *Concurrency*. We have, in general, several threads creating objects of these kinds. Alexandrescu solves the problem by using mutexes.

5 This is a recurring pattern, also explained in the "Programmer guide".

This is forbidden in our real-time thread; which is just what we are trying to avoid. Maybe a lock-free data-structure could be used to allocate this memory, solving the issue, but we do not any that would fit this purpose.

2. *Explicitness*. We would like to easily know what operations are safe to be performed in the real-time thread. Any programmer should get worried when she reads something like this in a real-time code path:

```
some_type* x = new some_concrete_type (...)
```

To regain confidence about the correctness of that code, she should go and read the implementation of `some_concrete_type` and discover that it has an overloaded operator `new`.

Another plausible solution is to constrain the set of types acceptable in some variable. By doing so we are, in fact, changing the open world assumption of universal polymorphism by the closed-world assumption of ad-hoc polymorphism. Thanks to this constrain, we can, via template metaprogramming or the new C++11 `constexpr` facility, compute at compile time the maximum storage needed to hold values of any of those types and allocate it in the stack; then construct the objects in this space with *placement new*. This is actually how *Boost.Variant*[6] is implemented [2], and as we are depending on Boost already, using it is quite safe in real-time code and can be a solution when ad-hoc polymorphism is enough.

In some other cases ad-hoc polymorphism is not enough, but the pattern of object construction and destruction can be constrained instead. We can think of *the heap* as a global data-structure — a data-structure that is so useful that its insertion and deletion operations are keywords of the language itself — i.e. the global operators `new` and `delete`. In fact, its interface is very similar to that of a multi-set, but without iteration and with the ability to hold elements of varying size. If we restrict its

6 www.boost.org/doc/html/variant.html

interface such that elements can only be allocated and released in FIFO or LIFO order, we get a *deque*.

Listing 1: Example of usage of heterogeneous deques

```
struct base {
    virtual void method () { std::cout << "b"; };
};
struct deriv : base {
    deriv (int x = 0) {}
    virtual void method () { std::cout << "d"; };
};

hetero_deque<base> q;
base b;
deriv d;

// 1. Copying
q.push_back (b); q.push_front (d);

// 2. Moving
q.push_back (deriv ()); q.push_front (std::move (b))

// 3. Constructing, even with params!
q.push_back<base> (); q.push_front<deriv> (1);

// Error checking
// static_assert error: 'int' is not a subtype of 'base'
q.push_back (1);

// Access is polymorphic!
// Output: bddbbd
for (Base& x : q)
  q.method ();
```

The class `psynth::base::hetero_deque<Base>` is a constant-size deque that can hold elements of any subclass of its type parameter `Base`. The concrete type of the object must be known at insertion time for the deque to be able to allocate space for it in its internal buffer. Actually, the data structure API offers three ways of inserting a concrete derivative of `Base`

into it: (1) copying it, (2) moving it using R-value references or (3) directly constructing it into the data-structure, using perfect forwarding to pass the parameters. The inserted element must be a subtype of `Base` and this is statically checked rising a `static_assert` error otherwise. All this is illustrated by the example in listing 1.

The data-structure interface is similar to that of an STL deque, with some differences. Because access is done polymorphically via a reference of type `Base&`, `value_type` semantics, which maps to `Base`, are slightly different than in most containers because the actual elements are of different heterogeneous concrete types. Also, because objects can be directly constructed inside the data structure with arbitrary parameters, they needn't be regular in any sense, this is, they don't have to be copy-constructible, move-constructible, default-constructible, copyable or moveable. This is quite convenient because classes that are designed to be used polymorphically, more often than not, they do not satisfy many of these properties. The only restriction is that `Base` should have a virtual destructor if any of its derivates has a non-trivial destructor.

Note that the data structure itself is not copyable, nor moveable, nor resizeable. While it would be feasible to implement it otherwise, it has a slight memory overhead. As our current use-cases do not need it, we decided to stick to the current simpler implementation. This is so because to be standard compliant, an object identity — i.e. the memory address where it lives — must remain constant during its livetime; in order to copy or move it, we need to call the proper copy or move constructor, and in the later case call the destructor of the moved object if it is to be released [7]. Calling the proper copy or move operation requires, at least, one function pointer — or a *vtable* pointer for a box type. We already have a plan to implement this feature using policy-based design [3] so it would not cause an overhead to those who do not need it.

The data structure is implemented by storing the objects in a constant sized memory block organised in a circular buffer fashion. We append a header like the one in listing 2 at the beginning of each a object. An empty header at the end of the list allows to keep track of the beginning of the

Listing 2: Header of heterogeneous deque elements

```
template <class Base>
struct header {
    header* prev;
    header* next;
    Base* access;
};
```

free space. The `prev` and `next` pointers organise the data in the deque in a doubly-linked list fashion. Even though the storage is continuous, this is needed because the size of the objects is variable and only known at object insertion time. The `access` pointer keeps a reference to the object itself casted to `Base*`. This is required because, in the presence of multiple-inheritance, the `Base` part of the object might not be aligned at the beginning of the object's layout. If we think about it, the `next` and *access* pointers are just a way of keeping track of the static information that is lost in the *type erasure* that happens when adding a new element to the data structure: its *size* and the *offset* to its still known attributes and *vtable*; the `prev` pointer just aids reverse traversal and addition at the front.

Figure 27 represents an instance of the data structure containing two objects. The `front` and `back` pointers keep track of the first and last valid element in the container. We can see how adding a new element that does not fit at the end of the memory space, causes that space to be wasted — marked as *offset* in the figure — and the object is instead added at the beginning of the memory block, where there is enough free space available in this case, in a circular buffer fashion.

So, thanks to low level facilities provide by C++, we are able to use different techniques to get the benefit in expressiveness and maintainability of polymorphism in a real-time constrained environment. Our most prominent use-case, passing active events among different threads, gets very benefited from our last approach, as we shall see in the rest of this section. In other parts of the system, disjoint unions as implemented by *Boost.Variant* are enough, and we actually do use them, for example, to

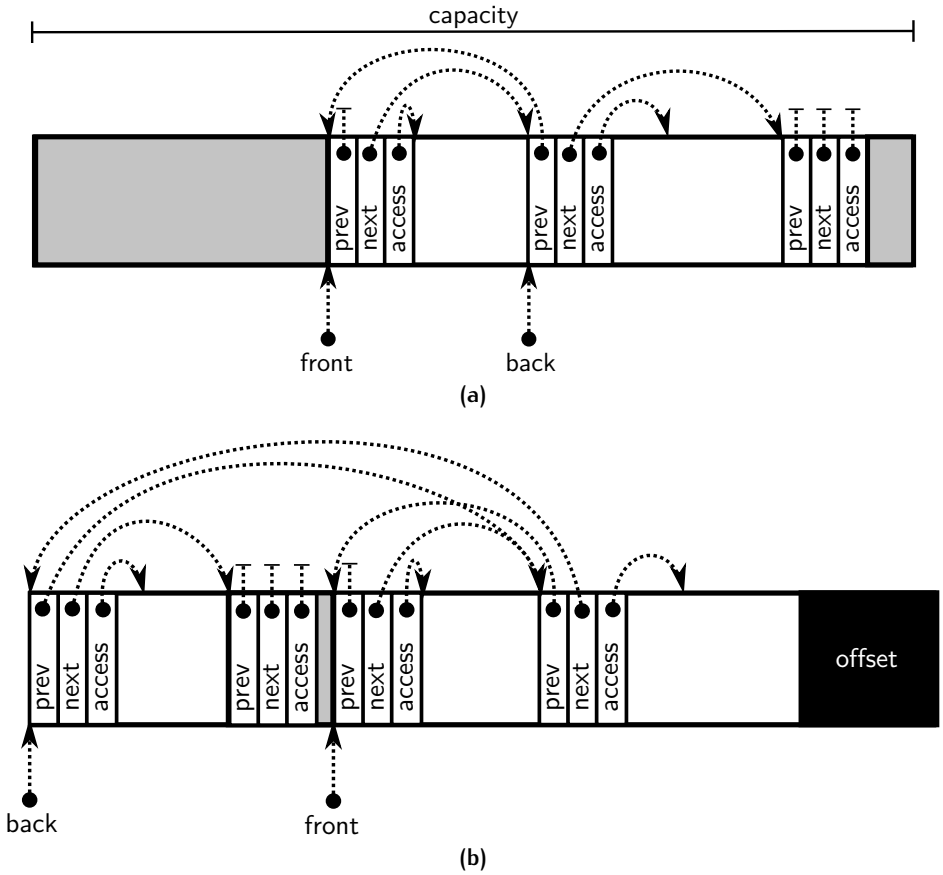


Figure 27.: An heterogeneous deque with two elements (a), and after adding a third element (b).

treat a closed family of oscillator objects polymorphically in our oscillator node implementation. A custom object allocator is a valid solution in some other situations, but as we have already analysed, it is mostly unsuitable for our quite specific needs.

4.2.3 Execution model

Figure 28 introduces some new classes that are the coordinate the execution of the synthesis graph. Because there are several concurrent — and potentially parallel — threads of execution, writing code without

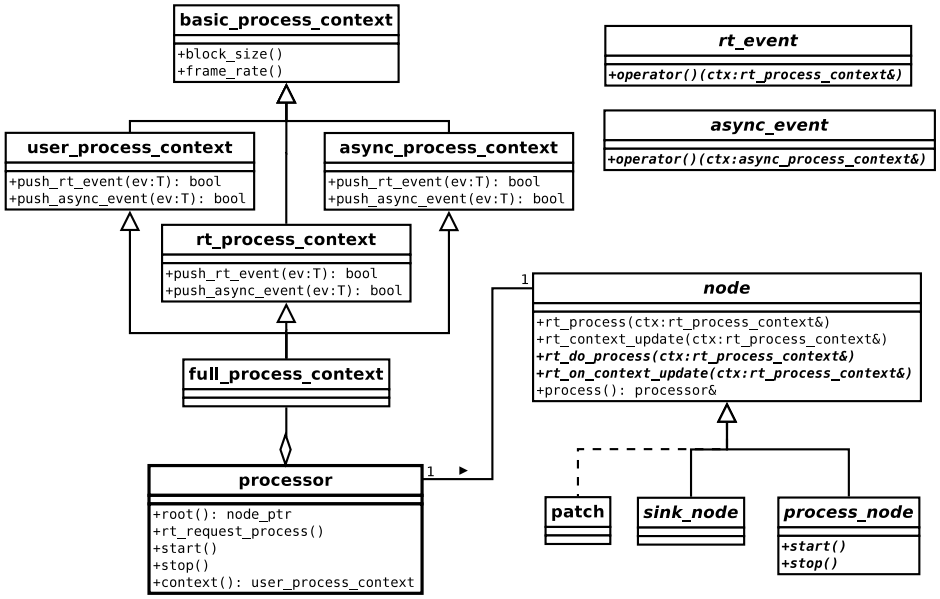


Figure 28.: The graph processor, its context and the processing methods of nodes.

race conditions and other synchronisation problems can get quite hard if we do not establish some conventions. Unless otherwise specified in its documentation, a method is expected to be executed in only one *kind* of thread, sometimes even in just only one specific thread. There are three kinds of threads and some naming conventions allow us to determine in which thread some method is expected to be used:

USER THREAD The user thread is the one where most objects are created and where, usually, the user interface lives. In some way, this can be considered the *normal thread*, and it is the root of the thread tree — i.e. the one that creates the other threads.

If a method does not have any special prefix nor belongs to a class named with a special prefix, we shall assume, unless otherwise specified in its documentation, that it is safe to execute it only in the user thread. In the current implementation, these methods, while reentrant, they are not fully thread-safe. This means that most non-const methods of an object and related objects — i.e. objects that keep references among themselves — shall be executed in the same

thread, but it is safe to operate with unrelated objects in different user threads.

REAL-TIME (RT) THREADS These are the threads where the actual work of processing the synthesis graph is done. We sometimes call these just the *audio thread*. As we shall see later, these threads are controlled by the `process_node` derivates, but that is not relevant now. Methods beginning by the `rt_` prefix, and all methods of classes named with that prefix, are expected to be executed in this thread. While it can be normal for several RT threads to coexist, implementers of RT-methods and objects should not worry about this, because, unless they are actually implementing a `process_node`, the system itself serialises calls to dependent methods — in the current implementation, it just serialises calls to `rt_request_process`, we will discuss this further soon.

ASYNCHRONOUS OPERATION THREADS These threads, that we may call just *async threads*, ease the execution of operations that are forbidden in the RT thread. The `processor` controls the execution of these threads. Methods and classes prefixed with `async_` are expected to be executed in these thread.

4.2.3.1 *Processing the graph*

The `processor` class is a *facade* that manages the execution of the synthesis graph described by a given root node. Of course, a whole graph can be attached to one and only processor and an exception would be thrown if we try to attach it to several.

The `rt_request_process` method triggers the execution of one iteration of the graph processing, this is, it processes one block of audio rate samples or one control sample. The processing algorithm is as follows. The processor keeps a list the nodes that have side-effects outside the graph itself, for example, a node for outputting to a device. These are the nodes that inherit from `sink_node`, and we can refer to them as just *sinks*. For every sink, the processor invokes the recursive algorithm 1 implemented by `rt_process` in the `node` class. This recursively calls

Algorithm 1 Process a control iteration of the graph, $rt\_process(n, ctx)$

Require:
$n \in$ `node`
$ctx \in$ `rt_process_context`
Ensure:
Every node in the graph is processed and its results ready in their outputs, if any, in the right order.

if $\neg visited(n)$ **then**
  for all $i \in inputs(n)$ **do**
    if $connected(i)$ **then**
      $n_i \leftarrow source(i)$
      $rt\_process(n_i, ctx)$
    end if
  end for
  $rt\_do\_process(n, ctx)$
  $visited(n) \leftarrow \top$
end if

the process method of the nodes connected to one's input. Because the synthesis graph may have loops and input ports may be connected to several outputs, it marks every node it visits. In this way, every node is processed only once and nodes that may have no side effects are discarded — because they do not interact wit the "real world" nor any node that does depends on it. When a node is being visited, the pure virtual `rt_do_process` methods is invoked. That is the most important method that implementers of their DSP modules have to take care of; in it they should fetch the input data, if any, process it, and write some output, if any. We often just call this the *worker method* of a node.

4.2.3.2 *Proactive nodes*

`rt_do_process` is usually executed by implicitly triggered by *process_nodes*. These nodes represent devices that may require processing from the graph, and they have and extra `start ()` and `stop ()` methods. They are in charge of managing their own thread where they request process-

ing whenever they need frames. The start and stop methods manage the status of this thread.

A canonical concrete *process_node* is *core::async_output*, which, when associated to an asynchronous output device (see section 3.2.6.2), requests processing in the output callback and passes what it gets from its single input port to the device. In order to allow several asynchronous output devices, calls to `rt_request_process` are serialised. Then, in their worker method they just accumulate their input in an internal ring buffer, but they do not directly output to the device, because that operation may block and delay other devices that are waiting on this request. In their own device callback, when there is enough information in their internal ring buffer, this is sent to the device.

Calls to `rt_request_process` are serialised using a mutex in a special way. When we try to get the lock and it is owned already, we do wait for the ongoing request to finish but we do not perform a new request, because the one that was already on the way might have been enough for the device to get sufficient information. This use of locks does not violate the RT rules, because all threads that are expected to contend for it are real-time so no priority inversion can occur. It may introduce some context switches, but that is unavoidable, and in practise they do not add much overhead, and in modern processor the gained parallelism compensates. While we are doing the output operation on the device, another process node might have triggered another request, preparing our next bunch of frames already. When we are locked on the mutex, another RT thread is already doing what we wanted to do, and if the OS scheduler is not dumb, it should preempt our RT thread soon at it will happily find that its request have been processed already. Thanks to this mechanism we can output to several devices at the same time, even if they are in different sound cards. What information is set to each one depends on how the graph is interconnected, and it might even be different, of course. This is a common usecase, for example, DJ's peek at how the next track mixes in while they adjust its tempo on their headphones without altering what is still being played on the loudspeakers. Figure 29 shows how a typical DJ setup could be built with Pyschosynth's engine

and illustrates the most important aspects of the algorithm that we just described.

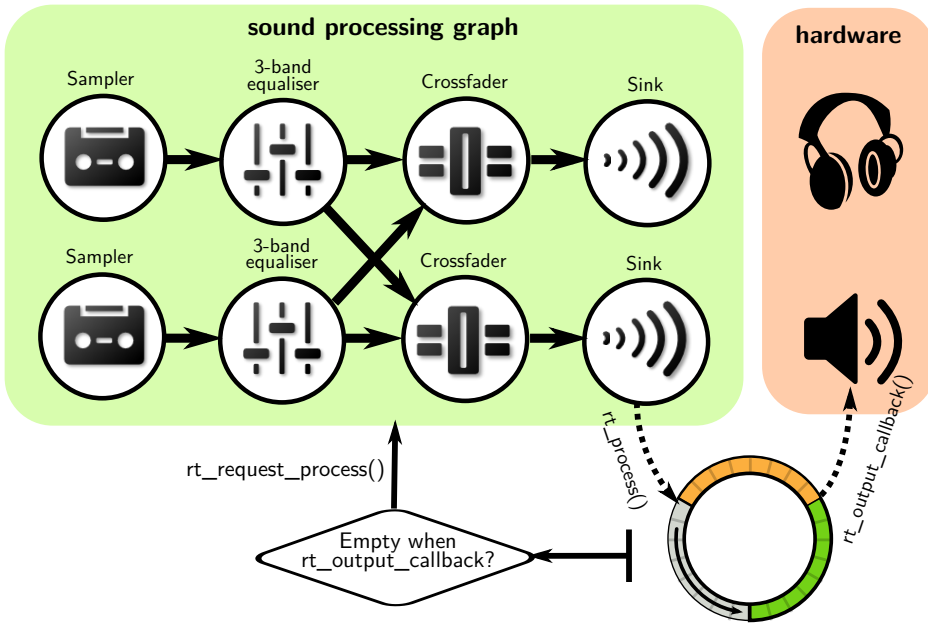


Figure 29.: A DJ setup built with Psychosynht's engine to illustrate the multiple device output system.

The `start ()` and `stop ()` methods in the `processor` class are in charge of starting and stopping all the process nodes in its associated graph. This class also starts nodes that are added after the processor have already been started and stops them when removed. Usually, we can just start the processor at the beginning of our program and start manipulating its graph. These methods also control the execution of the *async thread*.

For completeness, we shall mention that there are passive output nodes that only write data in the event of an external request for processing. We can use this, for example, to record a live session driven by a second active device to a file. Also, if we only want to generate the audio directly in the file without the presence of any proactive sink, we may directly call `rt_process_request` in the user thread as many times as needed,

taking into account that the duration of the file resulting from calling it i times is, in seconds:

$$duration(i) = \frac{i \times block\_size}{frame\_rate} \tag{4.2}$$

4.2.3.3 *Communication between threads*

Creating proper abstractions for communicating among the different threads is essential for the extensibility of the system. This is, in fact, one of the main flaws of the former implementation of this layer. The usage of ad-hoc mechanisms for every different feature made the system harder to understand, leading to subtle race conditions, the overuse of locks and unnecessary busy-wait checks in the RT thread.

It should be almost clear already why we need our own thread communication abstraction instead of just using some traditional blocking mechanism — message passing, monitors, etc. First, the RT threads should not wait on non-RT threads like the user thread or the async threads. Moreover, most of the processing in the RT thread should be done in tight, simple, loops; even if the parameters coming from other threads where implemented with atomic variables, this state should not change during the execution of these loops. What we do instead is to delegate the execution of actions with side-effects visible by another thread to that thread itself. We call these actions *events*, and they are an instance of the *command* design pattern [27]. Going back to figure 28, we can distinguish two kinds of events, `rt_event`, that are to be executed by the RT thread, and `async_event`, that are executed in the asynchronous thread. Note that there is no notion of `user_event`, because that would add unnecessary complexity. If the async thread wants to produce side-effects visible to the user, they can just coordinate using locks, because they are not forbidden in the async thread. If the RT thread wanted to do so, he can just send an event to the asynchronous thread that does the synchronised update in that unconstrained environment.

The events are just functors with its overloaded `operator()` made virtual. The *fn_*_event* family of concrete events just wrap a non-polymorphic

functor to match the corresponding interface. This recurring pattern of embedding a object with a non-polymorphic interface into an adaptor with the same interface implemented with virtual methods is called *boxing*. The `make_*_event` family of box factories use template argument deduction to automatically box its parameter without explicitly writing its type. This is extremely useful with the new C++11 lambdas, because to have a compile-time determined size[7], every lambda expression has a distinct, anonymous, compile-time determined type [37]. Using lambda expressions to create events rises the conciseness and clarity of the code, because the code that asynchronously produces the side-effects follows directly to the source of this event. If you are eager to see this pattern of control flow abstraction in use, skim to example 3.

The events are stored inside the *process context*. This is where the common information that characterises the processing is stored, such as the block size or the sample rate. Also the event deques are stored there. The `basic_process_context` holds all this information. But, instead of directly exposing access to the event queues, three classes that virtually inherit from it offer different access methods for them, even though with the same signatures. These are the `user_process_context`, `rt_process_context` and `async_process_context` classes. Each event deque is split in a triple-buffer fashion. Which internal buffer should we push events in, and what kind of lock, if any, should we use when doing so, depends on the thread that is the source of the event. These three views on the process context manage this implementation logic transparently to the user. We designed our API carefully to ensure that the code executed in each process has access only to the proper view of the context; in this way, the type system ensures that events are sent in the right way, giving confidence to the programmer about his own code.

The events received in the RT thread from other threads are processed at the beginning of each `rt_request_process` just before executing the synthesis graph. Events that are sent from the RT thread to the RT thread itself are instead processed at the end of the request. Recursive events are

7 Something that gracefully allows us to use them even in our RT-thread as long as we do not erase its type with a `std::function<>`

delayed to the next request. Events are accumulated and sent in blocks, in this way we avoid the unlikely situation in which the user thread is constantly sending events and the RT thread is stuck processing them as they arrive, and never advances to execute the synthesis graph.

The async thread is in an infinite loop waiting to receive and process events. Once again, it receives events from the RT thread in blocks that are dispatched at the end of every RT process cycle.

There two triple-buffer structures, one associated to the RT thread and another one for the async thread. Each triple buffer contains three `hetero_deques`[8] where one can push events of any derivate of the `rt_event` and `async_event` respectively. Multiple buffering is a technique often associated to computer graphics. Each buffer is accessed through a pointer; one part of the system is writing data to one of the buffers (the back buffer) and another one is concurrently reading from another (the front buffer). When the reader is done processing the current buffer, a simple pointer swap operation (called the buffer "flip") feeds him with a bunch of new information coming from the back buffer, and the writer gets a new fresh blank buffer where to dump new data. The three buffers associated to each of our triple-buffer structures have the following semantics, as exemplified by figure 30.

FRONT BUFFER This is the buffer where the events are popped from for processing. Only the RT thread should access the front of its triple buffer, and the same happens in the async thread.

BACK BUFFER This is, in general, the buffer where events coming from other threads are written to. In the RT triple buffer case, the both the user thread and the async thread do write into it, but they synchronise their insertions with a mutex — because they can! The `flip_back ()` method exchanges the front and back buffers. The flip is done in the RT thread, at the beginning of each process request, thus should can not contend for the mutex. Instead it does a conditional lock, and if it fails, it skips the flip. This may

8 The interested reader can take a look at the `graph::triple_buffer` class, which is parametrised over the buffer type and the synchronisation policy for the "flip" operations.

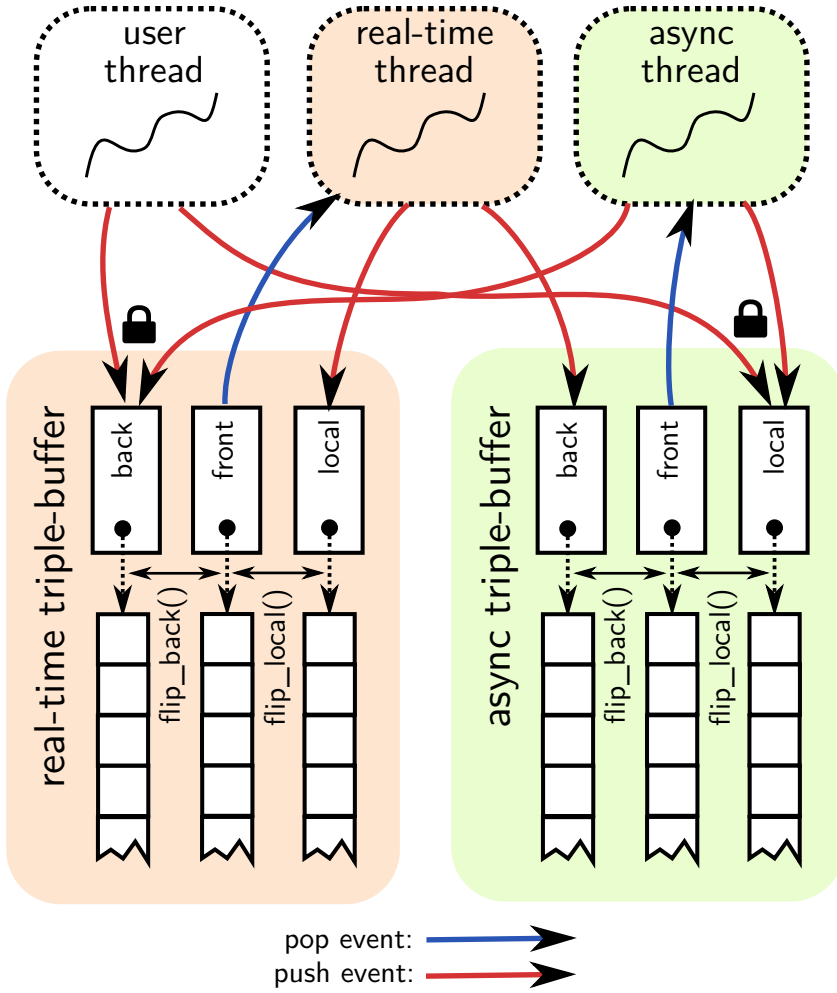


Figure 30.: Communication with triple buffer event deques.

cause some events to be delayed more than one request if there is a lot of contention on the mutex — i.e. the user and async threads are writing a lot of events — but that does not happen in practise and informal experiments suggest that the approach is enough. However, this could be completely solved by implementing lock-free `hetero_deques`, at least lock-free for one writer and one reader. We develop this idea in section 4.4.2.1. In that case, the user and async thread would still need to serialise their writes, but the RT can flip and read safely without caring about the mutex at all.

The back of the async triple buffer is slightly different. Here, only the RT thread writes, because if the user thread could write on it at the same time we would need a lock. Indeed, it is also the RT thread who does the flip after filling it, as long as the async thread is not still processing the current front buffer, otherwise the flip is postponed.

LOCAL BUFFER The local buffer is where a buffer writes events that he sends to itself. Thanks to this, we can have a notion of "post-processing" and also infinitely recursive events that generate a new event each time. This is very useful for implementing *tasks* and state machines. If a thread wrote his own events in its front buffers, this kind of behaviour would lead to an infinite loop and events sent to the back buffer would never be processed. The `flip_local()` method exchanges the front and local buffers.

Note that in the local buffer of the async thread is a bit special. Because the RT thread has exclusive non synchronised access to the back buffer, events from the user thread are sent to the async thread through the local buffer, and they use a mutex to avoid data races.

While this structure might seem a bit baroque, all the complexity is hidden in the `processor` and `*_process_context` classes. In most cases, one should only care about making the right choice of which thread should execute our event; all the dispatching and processing happens transparently and the type system ensures we can only do it in a thread safe way — thanks to the three-tier processing contexts.

Listing 3 shows an example on how to use this event passing system. The frequency parameter is set from the user thread, but it must be read from the RT thread and the related variable *rt_dt* must be updated. Doing this in a portable and thread-safe way without using mutexes is tricky. Thanks to our structure, we just send an event that does the job of updating the variables. There are no possible data races even if reading/writing float values is not atomic, because the lambda expression captures a local copy of the `freq` variable. This is a very convenient fact: the event's own state provides an intermediate buffer for communicating

value updates without data races; the lambda semantics in C++0x just provides us the syntactic sugar to make the code pleasant to write and read. Please remember that this code is here only to illustrate the usage of the event passing system; in practice, passing parameters from the user thread to the RT thread is even simpler, as we will explain in the following.

4.2.4 Node components

In the last section we have seen how we can easily build a notion of *parameter* similar to the one abstracted in the analysis section on top of the event passing infrastructure. However, there are a few ways we can criticise the approach in listing 3:

1. Even though the code is not too large, in the presence of nodes with many parameters we could easily devise a recurring pattern calling for a refactoring. Indeed, we always do the same to pass information from the user to the node's processing state: store a copy for the user and another one for the node's internal processing in the RT thread — to update the later from the user thread, we send a RT event. For controls working in the other direction, this is, controls intended to send information from the RT thread to the user, this is a bit less trivial because it involves sending the update through the async thread and, if the status control type does not have an atomic assign operation, locking a mutex to avoid data races between the user and async thread. It seems reasonable to abstract this behaviour and relieve the node implementer from rewriting this patterns *ad-nauseum*. Moreover, if abstracted as a class, these controls can be later extended by themselves, for example, to add polyphonic support.
2. There are some corner cases that the previous code does not address. Actually, if the node is not attached to a processor that code would throw a `node_attachment_error` exception when it invokes the node's `process ()` method inside `set_frequency ()`. It does make

Listing 3: A thread communication use-case

```
struct example_oscillator_node : public node
{
    example_oscillator_node ()
        : _freq (440.0f) , _rt_freq (_freq), _rt_phase (0)
        , _rt_dt (0) // rt_on_context_update will be executed before
                     // the first process so we can set properly this
                     // variable that depends on the frame rate.

    void set_frequency (float freq) {
        _freq = freq;
        auto& ctx = process ().context ();
        ctx.push_rt_event (make_event ([=] (rt_process_context& ctx) {
            _rt_freq = freq;
            _rt_dt = 1 / (freq * ctx.frame_rate ());
        }));
    }

    float frequency () const
    { return _freq; }

protected:
    void rt_on_context_update (rt_process_context& ctx)
    { _rt_dt = 1 / (_rt_freq * ctx.frame_rate ()); }

    void rt_do_process (rt_process_context& ctx) {
        auto frames = ctx.block_size ();
        while (frames -->> 0) {
            // We suppose M_PI is defined. The output_sample function
            // does not actually exist, it is just an example.
            output_sample (std::sin (_rt_phase * 2 * M_PI));
            _rt_phase += dt;
        }
    }

    float _freq, _rt_freq;
    float _rt_phase, _rt_dt;
}
```

sense to alter the node frequency before attaching it to a process, so this limitation should be avoided. Moreover, even if the node is attached but the process is not started, events would accumulate in the queue. If the node is detached from the processor before the network is run, the internal frequency would never be updated indeed. This shows how there are many corner cases, and isolating this complexity in a class with its own responsibilities yields safer code.

3. Manipulating the frequency parameter as implemented in that code requires knowing the node's concrete type. This is a show stopper for the *plugin* that we shall implement in the next iteration. Instead, parameters should be accessible directly from the node's basic interface, and identified with strings, such that using them requires no compile-time binding at all.

This results in the notion of *component*: an attribute associated to a node, that may be of four kinds; an input port, an output port, a input control or an output control. These four kinds of components have different properties but there are some commonalities among them too:

1. They have an unique name (unique within a node and kind of component) that identifies them.
2. They have an associated regular type. Regular means that it models the `Regular` concept, i.e. is default constructible, copy-constructible and copyable. This is the type of the data that they "hold" or "pass".
3. They have some associated metadata, which is just a $String \to String$ map providing information that may be useful to the user, most of the time to the human user of the application.
4. They can be queried by their name and enumerated from the node interface. This is required to allow the implementation of plug-ins.

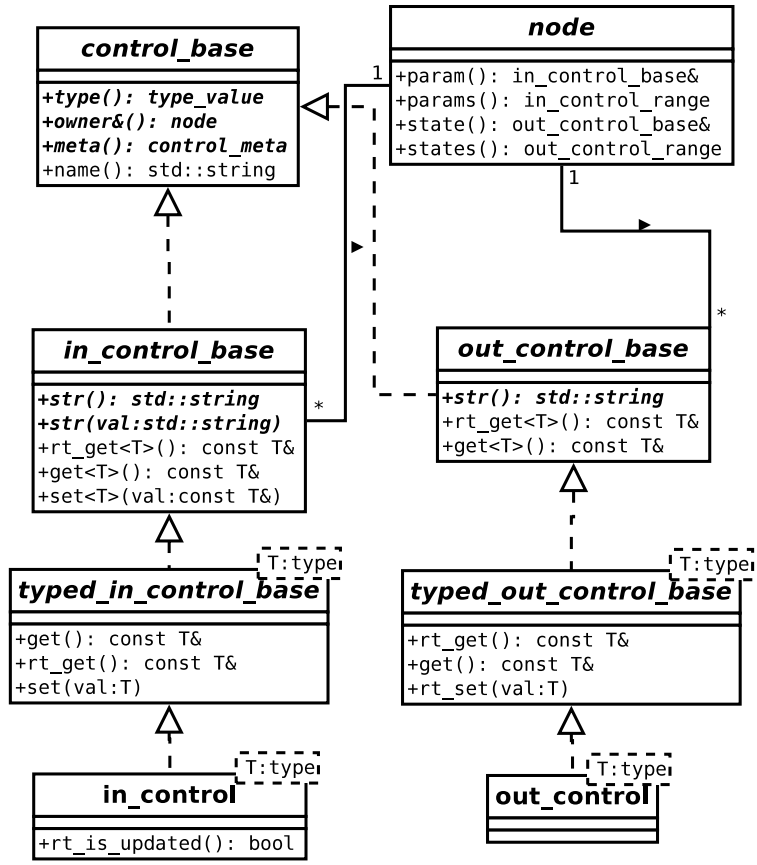


Figure 31.: UML class diagram of the most important classes for the port components.

4.2.4.1 *Control components*

Figure 31 shows the most important classes and interfaces implementing the control components — i.e. parameter and status variables. Parameters or input controls are used to pass information from the user thread to the node's internal processing state. States or output controls work in the other way, they expose some internal processing status to the user.

Note that in the current implementation there is some special requirement that somehow couples the control bases and their typed version. If you access the control through the control base access template method with a type `T`, and for that control `type ()` method returns `typeid (T)`,

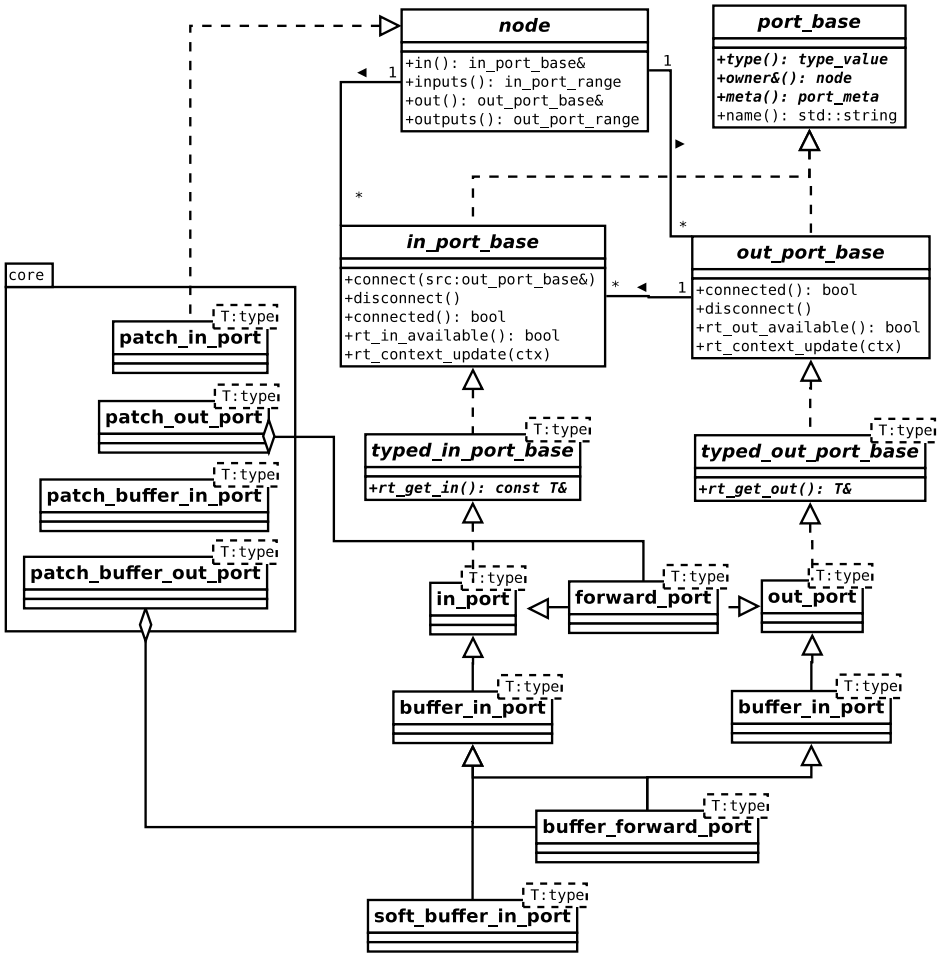


Figure 32.: UML class diagram of most important classes for the control components.

then that object must be a typed control of type T; otherwise this is undefined behaviour. In practice, this means that you must never inherit from `in_control_base` or `out_control_base` directly, but you should inherit from their typed versions instead.

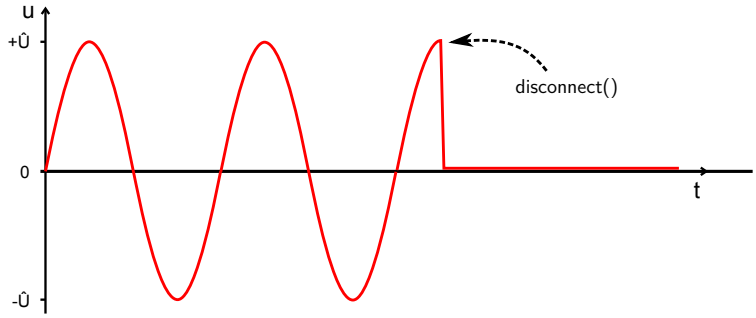
4.2.5 Port components

Figure 32 shows the most relevant classes implementing the notion of *ports*. There are two symmetric kinds of ports, input and output.

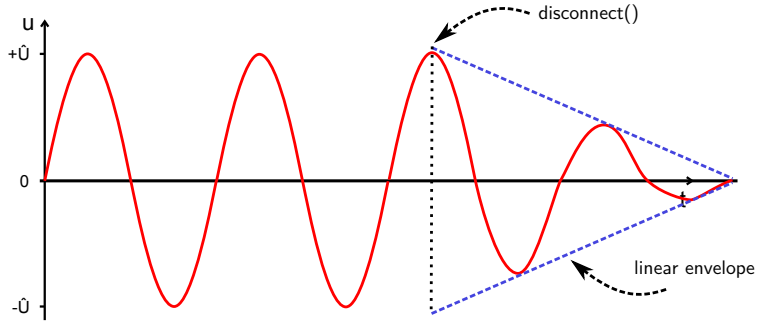
Following the analysis in the first section, these ports are related in a one-to-many manner — an output port may be connected to many input ports. These ports thus give the graph shape to patches. As we can see in the diagram, concrete ports are parametrised over the signal type but a polymorphic interface exists providing its basic functionality. While this diagram is very similar to that describing the *controls* in figure 31, ports have some added complexity and there are some extra classes that are worth mentioning.

First, we have to distinguish between audio-rate and control-rate signals. Usually, we store audio-rate signals in sound buffers modelling the `BufferConcept` we developed in the previous chapter. These buffers should be able to allocate a whole bunch of `block_size` frames; to enforce this invariant, some mechanism should react on process context update notifications to ensure that the buffer size matches the current processing block size. The `buffer_out_port` and `buffer_in_port` do this transparently to the user, but they impose a `BufferConcept` requirement on their type parameter T. For control rate signals, using a simple `in_port` or `out_port` suffices.

Another important issue that we have to address is: how do we add ports to a patch node? Note that the relation between components and nodes is somewhat static — actually the port or control constructor takes care of establishing the relationship. We should expect that, like in most modular synthesis software, the user can arbitrarily manipulate which ports are exposed from within a patch. The most natural way of manipulating a patch is by manipulating nodes, thus, ports are exposed by special kinds of concrete nodes like the ones in the `core` package in the diagram. The `patch` class detects whenever a port node is added and exposes it in its interface. A "patch input port node" would show as an input port in the patch that contains it, and the *"port-name"* parameter of the node controls the name that the port has in the patch interface. The port node itself has an output port, that can be connected to any other node within the patch. A "patch output port node" works in the opposite way: it exposes an output port externally and an input port internally. These kind of nodes are implemented using the *forwarding ports*, which



(a) A click is produced on a disconnect.



(b) Solution by delaying the actual disconnection and enveloping the signal.

Figure 33.: The problem of softening a signal disconnection.

allow implementing a sort of *identity function* node without the overhead of copying the input into the output. When its output is requested, they just return a reference to its input data.

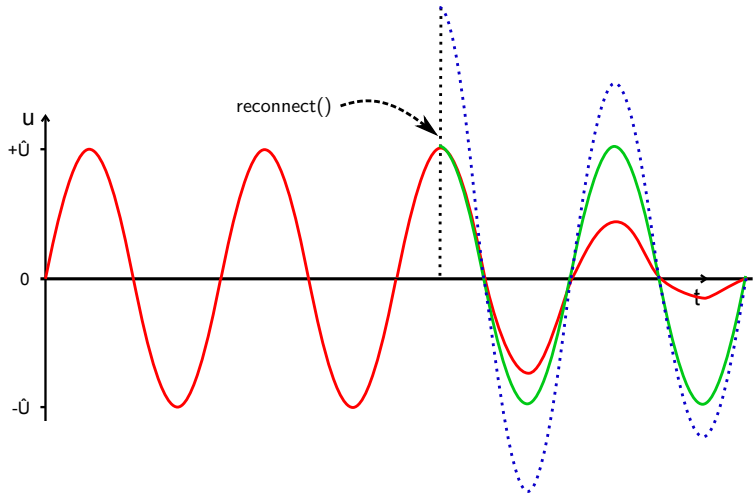
More problems arise when we consider that ports may be disconnected and reconnected at any point in time, a precondition for the high interactivity that we want to achieve with our system. As a matter of bad luck, it may happen that we disconnect a port just when the incoming sound signal is at a "high" value. Figure 33 (a) represents the actual signal that is going through an hypothetical input port which is connected to a sinusoid generator and then suddenly disconnected. If this signal is emitted to the speakers, we would hear a very disruptive "click" noise at the point of disconnection because an instantaneous change can be consider of infinite frequency that the DAC in your sound card will just mess up.

There are few ways to solve this problem. For example, the Audacity [9] sound editor automatically modifies selection edges on the sound wave slightly to the closest positive-slope zero crossing when the *Find Zero Crossings* option is enabled. In a similar way, we could delay a disconnection until the signal goes back naturally to zero.

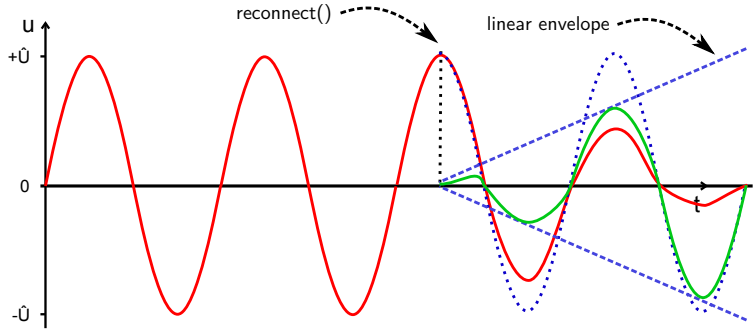
While we might implement that behaviour in the future to make it optionally available when it makes sense, we have implemented a bit more general solution that works sufficiently well even with LFO's where the former approach would introduce too much latency. Instead of waiting for the source signal to go back to zero, we directly apply a linear envelope that goes from 1 to 0 and only wait for the envelope to reach the 0. Figure 33 (b) illustrates this effect. The length of the envelope is fixed and thus the latency introduced by the anti-click mechanism is deterministic and we can keep it as small as we want. The length of the envelope defaults to a conservative value of $4ms$. The reasoning behind this is that in the worst case it would introduce a slope from 1 to 0 of length $4ms$, which if it were a signal by itself would have a dominant frequency of $\sim 50hz$ that lies around the low threshold that human hearing can perceive [29]. The actual artifacts that this mechanism may introduce in the general case are hard to analyse formally, but subjective experiments show that it introduces no noticeable defects on its own and eliminates the click artifact that we wanted to avoid. The `soft_in_port` implements this behaviour and the client can set the envelope length and the stable value — whether the signal should fade out to 0, 1 or some other value — to adapt this mechanism taking the actual nature of the expected input into account.

Note that the envelope must be applied both on disconnection and connection, otherwise a disastrous situation like the one in figure 34 (a) may occur. There, at the marked point in time the input port of the red signal is disconnected, but the same signal is connected back into a second port, represented in green. A dotted blue line show the sum of both the old port that is fading out and the new port. First of all, the click appears again at the new input end. Even worse, the total amplitude is

9 http://audacity.sourceforge.net/



(a) A click and a power artifact is introduced because of coupling a disconnection and a new connection.



(b) Solution by enveloping the new connection.

Figure 34.: The problem of softening a signal connection.

unexpectedly big — maybe even introducing clipping — during the input signal fadeout. If we apply a rising envelope when a new connection is made with the same properties that we described previously, the result is more close to what the user would usually expect, as shown in 34 (b).

In the future we may change the linear envelope with an exponential envelope, which is suggested to have a better perceived accuracy [18]. Also, it seems reasonable to use the second half of a Hann window[10] as envelope, as it is designed to produce minimal aliasing [13].

10 Like the one used to slice a signal when applying the Short-time Fourier transform.

4.2.5.1 *The oscillator node example revisited*

Listing 4: The oscillator node implemented using components

```
struct better_example_oscillator_node : public node
{
    buffer_out_port<mono32sf_buffer> out_output;
    soft_in_port<mono32sf_buffer> in_modulator;

    in_control<float> param_frequency;
    in_control<float> param_amplitude;

    better_example_oscillator_node ()
        : out_output ("output", this)
        , in_modulator ("modulator", this, 1.0f)
        , param_frequency ("frequency", this, 440.0f)
        , param_amplitude ("amplitude", this, 1.0f) {}

protected:
    void rt_do_process (rt_process_context& ctx)
    {
        _osc.set_frequency (param_frequency.rt_get ());
        _osc.set_amplitude (param_amplitude.rt_get ());
        _osc.update_fm (out_output.rt_out_range (),
                        in_modulator.rt_in_range ());
    }

    void rt_on_context_update (rt_process_context& ctx)
    { _osc.set_frame_rate (ctx.frame_rate ()); }

    synth::oscillator<synth::sine_generator> _osc;
};
```

We are now ready to devise a more realistic implementation of the oscillator node that we introduced in listing 3. The new code in listing 4 shows that by using the new constructs that we built in this section, not only is the code safer and more extensible and featured, it is also clearer and more concise. This code introduces the `synth::oscillator` class (section 3.2.7). Also, this oscillator allows frequency-modulation, this is, to vary the oscillator frequency using an optional input signal.

We do not need to check whether the optional modulator is connected, because an input port with softening defaults to its *stable value* when it is not connected.

4.2.6 Indirect object construction

In the presence of plug-ins, the main control flow of the program never gets to know at all the concrete type of some objects, and they are used only through its abstract interface. Even without plug-ins, it is quite convenient when writing a user interface to be able to choose the concrete type of an object dynamically — identified by a run-time value like a integer or a string. The *factory method* and *abstract factory* design patterns coordinate to provide this feature [27, 91].

We wrote the pattern as a set of reusable template classes, following "Modern C++ Design"s advice [3]. We extended his implementation with improved safety when using smart pointers, decoupled abstract factories and methods, and customisable constructor signatures. Many times factories make more sense as *singletons*, and some of our factories — *restricted static factories* — can only be instantiated as a singleton. The `base::singleton_holder<T, CreationPolicy, LifetimePolicy, ThreadingPolicy>` may be used build a singleton type from any type T, offering control over the life-time of the object, its construction and the thread safety of the singleton access method. The interested reader can read the attached source code or the reference manual for further information on these classes.

A singleton factory manager for creating nodes is defined as:

typedef
base::restricted_global_factory_manager<std::string, node_ptr>
node_factory;

The `self ()` method returns the singleton instance and then the `create (std::string id)` method returns a new instance wrapped in a safe shared pointer `node_ptr` — remember that this is an alias for

`std::shared_ptr<node>`. Creating a new instance thus does not require including the header that declares the concrete type, and it can be chosen based on a value determined by external input. For example, we can create an oscillator as simply as in:

```
node_ptr osc = node_factory::self ().create ("audio_sine_oscillator");
```

New types can be registered in the factory only by instantiating an instance of the `node_factory::registrant<T>` type and its constructor takes the name that should identify the type. This is so because it is a *restricted* factory, other kinds of factories have a `add ()` and `remove ()` methods for registering types at any time. This restricted singleton factory enforces doing the registration of the types before the `main ()` function starts by instantiating the registrant as a global variable. The following macros ease this common pattern of use:

```
#define PSYNTH_REGISTER_NODE_STATIC(node_type) \
    static ::psynth::graph::node_factory::registrant<node_type> \
    node_type ## _registrant_ (#node_type);

#define PSYNTH_REGISTER_NODE_STATIC_AS(node_type, node_name) \
    static ::psynth::graph::node_factory::registrant<node_type> \
    node_type ## _registrant_ (node_name);
```

To register the node that we defined in listing 4, we would write in its `.cpp` implementation file at global or namespace scope:

```
PSYNTH_REGISTER_NODE_STATIC(better_example_oscillator_node);
```

This mechanism can be used as a primitive plug-in system. If we open a shared library file (e.g a `.so` file in GNU/Linux) using the Unix `dlopen ()` function it would automatically instantiate any global registrant and we could already create any object provided by the plug-in passing their name as a string to the factory. Indeed, some authors propose this mechanism for dynamic loading of C++ classes [61, 12]. However, a practical plug-in system must be a bit more sophisticated to do proper

versioning, ease metadata access and put the main program — and not the plug-in — into control of the registration[11].

4.2.7 Going Model-View-Controller

So far, we have concentrated on devising the core of the audio processing engine. At some point, we will need to write user interfaces — indeed, we already have written a 3D experimental user interface and a command line user interface that can inter-operate over the network, and we have plans to build a custom user interface for tangible devices and another one for multi-touch pads. These user interfaces deserve a whole project on their own; as our main objective states (section 1.2.1), we shall concentrate on the basic framework *enabling* and supporting the rapid development of these interfaces. For this purpose, requirement 16 suggests that we provide the hooks to independently write collaborating views.

The Model-View-Controller, which is often regarded as an architecture, sometimes a pattern, or in its original paper as *paradigm* [46], provides the basic structure that we shall implement to satisfy requirement 16. When decoupling a system into a MVC arquitecture, there is this simple guideline, quoted in a software arquitects wiki as *the best rubric ever*:[12]

> "We need *smart* models, *thin* controllers, and *dumb* views"

This translates in our system as the following:

MODEL The core of our model is the synthesis network, this is, a `patch` and related classes. These classes maintain the invariants and do the work of processing the sound.

However, the `patch` class is not suitable as a model by itself. One of the main properties of a model is that it should be *observable*. This means that it should use some *inversion of control* mechanism to notify the views on any externally visible change of its state.

11 In these slides we discuss the tricky corners of implementing a plug-in system in C; a lot of what it states applies to C++: http://colinaroja.org/files/charla_c.pdf

12 http://c2.com/cgi/wiki?ModelViewController

This is usually an instance of the *observer* design pattern [27, 91]. A separate set of classes wrap the `psynth::graph` layer providing this observable interface — we call this the `psynth::world` layer, that should be renamed in the next iteration as the `psynth::model` layer. Figure 35 gives an overview of this layer. Observability is decoupled from the graph layer itself because (1) we get an extra design indirection that allows more API flexibility during the fast development of the graph layer and (2) avoids the sometimes significant overhead and bookkeeping that the observable interface requires, thus enabling the usage of the graph layer in isolation where no flexible UI's are required — e.g. in embedded systems or integrated in some other third party system.

VIEWS Views react to the model events producing, usually, visible side-effects in the real-world. Note that this notion of view is abstract and does not necessarily imply producing a visual representation of the model. An example of such non-visual view is the component that sends network events in reaction to local changes to keep remote synthesisers synchronised. The `psynth::net` namespace contains the views and controllers required for network synchronisation, using the OSC protocol [20]. Another example of view maintains a 3D visual representation of the synthesis state and is implemented in the `gui3d` module.

CONTROLLERS Controllers are usually what glues views and models together. They are usually do not require any special pattern or indirection; controllers are the set of objects that invoke mutating actions on the model. Because the observer pattern is used to notify the views, controller-view pairs are completely independent of each-other and each should work properly regardless or whatever views or controllers are also registered in the model. This clearly simplifies the implementation of a user interface or a network synchronisation system in a completely orthogonal and optional manner.

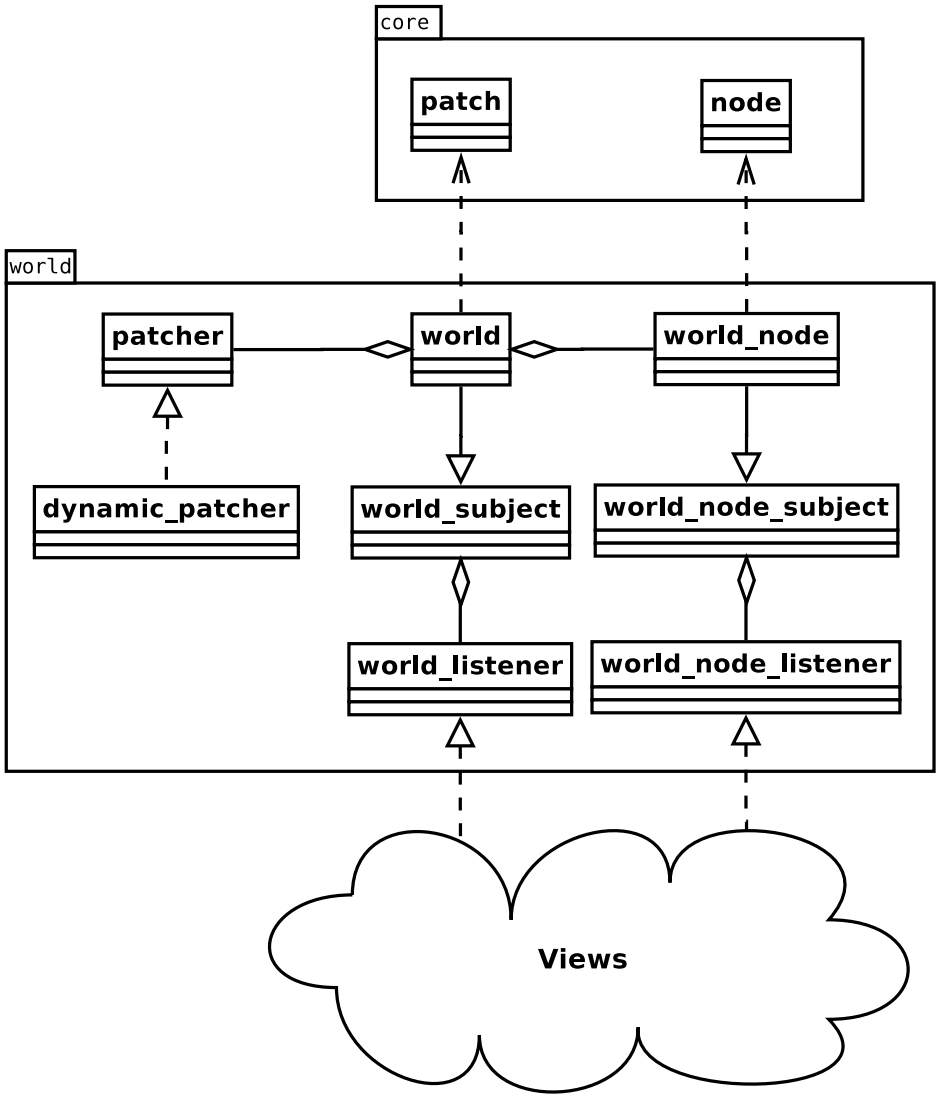


Figure 35.: UML class diagram of the `world` layer.

As we explain in section 4.3.3, integration of the new graph layer in the rest of the system has not been done already. This means that the work of modifying the *world layer* to use the new system is yet to be accomplished. Because the views and controllers *never* use the graph layer API directly, but they call the methods provided by the `world` and related classes instead, once this is done the current user interfaces and networking modules would require no modification.

The world layer hides directly manipulating the synthesis graph in its public API and delegates the topology management to a `patcher` derivative. This implements the *strategy* design pattern, because the *patcher* can be of different kinds — e.g. dynamic, manual, etc. The *dynamic patcher* arranges the network such that closest compatible connections are established.

The graph is connected constraining port connectivity in a one-to-one fashion using distance as criterion to choose connections among the possibilities, and it also avoids cycles. The resulting graph looks a lot like the *minimum spanning tree* of the graph constructed by nodes and all possible connections and edges weighted on the spatial distance of the interconnecting nodes in the visual representation. In fact the problem is almost an instance of the dynamic MST problem [73] which can handle fast a recomputation of the MST when a small update of the input data happens — e.g. the weight of one edge changes, for example, when moving an node.

The main impediment to trivially adapt the world layer to use the new graph layer is the *dynamic patcher*, because it uses a matrix based on static information about all possible nodes to check whether two ports are compatible. This static information is no longer available and dynamic metadata is provided instead to ease implementing a plug-in system, as we long discussed in this chapter. We will delay adapting the dynamic patcher until the plug-in system is fully implemented in order to avoid rewriting it in the presence of unexpected API changes.

4.2.8 Polyphonic nodes basic interface

While a full implementation of polyphonic audio will be later done when implementing MIDI support, an interface for polyphonic nodes has already been designed.

The most important aspect is how to decouple the state that is common to all voices and the per-voice internal state. Listing 5 shows an example of how a polyphonic oscillator could be implemented on top of our

proposed interface. The reader can compare it with listing 4 to assure that there is minimal boilerplate compared to a equivalent non-polyphonic node.

Listing 5: Example polyphonic node.

```
struct poly_oscillator_node : public poly_node<poly_oscillator_node>
{
    poly_buffer_out_port<mono32sf_buffer> out_output;
    poly_soft_in_port<mono32sf_buffer> in_modulator;

    in_control<float> param_frequency;
    in_control<float> param_amplitude;

    struct voice
    {
        void rt_do_process (poly_oscillator_node& n,
                            voice_id idx,
                            rt_process_context& ctx)
        {
            _osc.set_frequency (n.param_frequency.rt_get ());
            _osc.set_amplitude (n.param_amplitude.rt_get ());
            _osc.update_fm (n.out_port.rt_out_range (idx),
                            n.in_modulator.rt_in_range (idx));
        }

    private:
        synth::oscillator<synth::sine_generator> _osc;
    };

    poly_oscillator_node ()
        : out_output ("output", this)
        , in_modulator ("modulator", this, 1.0f)
        , param_frequency ("frequency", this, 440.0f)
        , param_amplitude ("amplitude", this, 1.0f) {}

};
```

The `poly_node<Deriv>` template class is the common base of all polyphonic nodes. It uses the "Curiously Recurring Template Pattern" , first documented by Coplien [21], to get full type information about its

subclass. This allows it to both handle voices efficiently and minimise boilerplate code. Then, there are three kinds of objects that intervene.

1. State that belongs naturally to the node itself and is common to all voices. A clear example of these are *controls*. As we can see in the above code, no special classes are implemented for polyphonic controls, because in fact there is no way that polyphony makes a difference for them, and they live in the node's common state.
2. Objects that keep both some per-voice state and some common state. A clear example of these are the *ports*. In the presence of polyphony, a port should handle a different buffer for each voice. However, the connection information is still common for all voices and should not be repeated. The `poly_*_port` family of port classes provides ports for polyphonic nodes. The main difference with normal ports in its interface is that the access methods take the current voice index as parameter.
3. State that changes with each voice. A nested `voice` class stores this state and its instances are managed transparently by the `poly_node<>` class. An example of a per-voice state is the current oscillator position. This *voice* class has a `rt_do_process` method similar to that in normal nodes; however, this method takes the actual node it belongs to as first parameter — such that it can access its common state — and its voice index — such that it can access its buffers from the ports, and any other information provided by objects with both per-voice and common data.

Implementing polyphony in this manner requires minimal, if any, modifications in the code that we have already implemented in the rest of this chapter. In fact, only implementing the `poly_*_port` and `poly_node<>` classes, that should specialise their non polyphonic counterparts, is needed. The user code is kept as simple as and type-safe as possible, yet allowing the most efficient implementation because no dynamic polymorphism is involved in the voice management.

4.3 VALIDATION

4.3.1 Unit testing

Most of the reasoning favouring the implementation of unit tests in section 4.3.1 applies here. Note 4.1 shows a briefing of the test suite results for the modules described in this chapter. Note that some suites in the `base` layer that were developed as support for this module are resented there too — like, for example, generic singletons, factories and heterogeneous deques. A total of 206 assertions are checked in 54 unit tests.

Note 4.1 (`psynth::graph` and some `psynth::base` 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. This kind of report may be generated passing the `--report=detailed` *parameter when running the test script directly.*

```
Test suite "c3_class_test_suite" passed with:
  10 assertions out of 10 passed
  10 test cases out of 10 passed

Test suite "base_exception_test_suite" passed with:
  16 assertions out of 16 passed
  4 test cases out of 4 passed

Test suite "base_hetero_deque_test_suite" passed with:
  87 assertions out of 87 passed
  9 test cases out of 9 passed

Test suite "base_factory_test_suite" passed with:
  8 assertions out of 8 passed
  4 test cases out of 4 passed

Test suite "graph_processor_test_suite" passed with:
  23 assertions out of 23 passed
```

```
  9 test cases out of 9 passed

Test suite "graph_core_test_suite" passed with:
  1 assertion out of 1 passed
  1 test case out of 1 passed

Test suite "graph_port_test_suite" passed with:
  1 assertion out of 1 passed
  1 test case out of 1 passed

Test suite "graph_control_test_suite" passed with:
  36 assertions out of 36 passed
  12 test cases out of 12 passed

Test suite "graph_patch_test_suite" passed with:
  24 assertions out of 24 passed
  4 test cases out of 4 passed
```

4.3.2 Performance

Quoting Donal Knuth is the best way to illustrate current trends towards software optimisation [45]:

> *Premature optimisation is the root of all evil.*

The reasoning behind this phrase is that micro-optimising every single part of our code is futile. It does not lead to faster programs, instead, doing so increases the complexity of the code, leading to subtle bugs and making harder, if not impossible, to later redesign the code to implement better algorithms that would give a much larger performance boost. A much better methodology suggests to concentrate on program correctness. When the program has been validated, *profiling* provides a way to detect, empirically, what parts of the program are consuming most time. Then, if needed, we can optimise the slow parts, which usually are deep in the call graph, just inside the most nested loops.

We developed a program that, we believe, is a good stress test for our system and executes most relevant code paths of our system. This program runs the synthesis graph in figure 36 for 10 seconds. Note that dashed lines represent patches and big black dots are patch ports. The other spheres are other kinds of nodes. The following events are generated during the execution:

- Every 0.1s it randomly changes the value of all the oscillators.
- Every second it randomly connects or disconnects the links between modulators and generators.

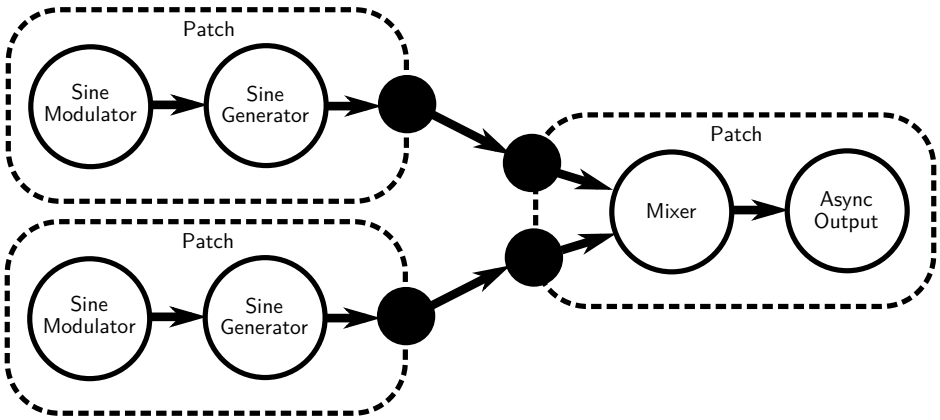


Figure 36.: Synthesis graph of the performance test program.

The program source code is in the `src/test/examples/graph_perf.cpp` file. The high amount of user events stresses the synchronisation channels to find potential implementation problems. It also has enough nodes to check if graph traversal consumes too much time, and the presence of several patches can show whether port forwarding is causing a significant overhead — it should not!

4.3.2.1 *Call graph profiling*

The Valgrind[13] [58] binary instrumentation framework provides many tools for evaluating the performance and correctness of programs by executing them in an instrumentated environment — a sort of machine

13 http://valgrind.org

simulator. In this section, we are most interested in Callgrind [93], a profiler that generates an annotated call graph of a program's execution, indicating the amount of time spent under each function and within the function itself. We should expect that the time spent in the functions that we developed in this chapter is little compared to that of doing the actual sound processing and I/O. Because in section 3.3.2 we already showed that our sound processing facilities are quite efficient, that would evidence that the overall performance of the system is acceptable.

Note 4.2 (Other kinds of profilers)
The Gperf[14] tool does a similar job with a much more lightweight instrumentation. Heavy instrumentation can be a problem when running a program with real-time conditions like ours causing fake device buffer underruns*, but a powerful enough machine often compensates. Indeed,* `gperf` *was not valid in our case because of its limitations with multithreading.*

Statistical profilers like Sysprof[15] and Oprofile[16] hook in the operating system and sample the current running process from time to time providing the least intrusive profiling. However, they provide much less accurate data and require long running times to yield good results. Actually, because of the barriers that the OS imposes in interrupting real-time threads, they are particularly unsuitable for our needs.

We indeed tried all these tools and Valgrind provided the most accurate and useful results. The ease of use and polishment of the tools for visualising Callgrind's output also influenced our decision.

We ran the program with the command:

```
$ valgrind --tool=callgrind ./example_graph_perf
```

This generates a `callgrind.out.$PID` file that can be visualised and navigated with the Kcachegrind[17] tool. It is usually hard to summarise a

17 http://kcachegrind.sourceforge.net/

whole profille data in a simple graph or table. Still, the resulting output supports our thesis: the graph maintenance and thread communication system produce negible overhead and most of the time is spent doing the sound processing itself as we expected. Around a 18% of the time was spent in the `soft_in_buffers` because we apply the envelopes in a sort of brute-force manner, but this is still little compared to the 35% that was spent generating sinusoids or the 38% of time that involved I/O with the sound card. Hence, it is not urgent to optimise that and it would be relatively easy to do that anyway. The Callgrind output file is included in the CD addendum such that the skeptical reader can validate the output by herself. Figure 37 attempts to present the most significant parts of the callgraph too.

4.3.2.2 *Validating real-time constraints satisfaction*

The `mutrace` and `matrace` tools profile the program to find much more specific patterns of resource usage. The first one traces a program to find which locks are used in a real-time thread, the number of contentions and the amount of time spent both contending and withing the lock's critical section. The later finds allocations and deallocations that happen within a real-time thread. It uses a much more lightweight kind of instrumentation that only relinks the application with wrapped versions of *pthreads* and *libc* that allow the profiler to collect the statistics. In order to run the program with these tools we passed the following extra compile flags when building our program:

```
-rdynamic -DPSYNTH_NO_LOG_RT_REQUEST
```

The first option generates extra hooks for accessing the stack trace at runtime allowing `mutrace` and `matrace` to provide better information. The second disables logging a message into the Psychosynth log system to inform the user that we succesfully got real-time priority for the audio thread. When logging that message several allocations are performed and critical sections are entered. All this happens in the real-time thread,

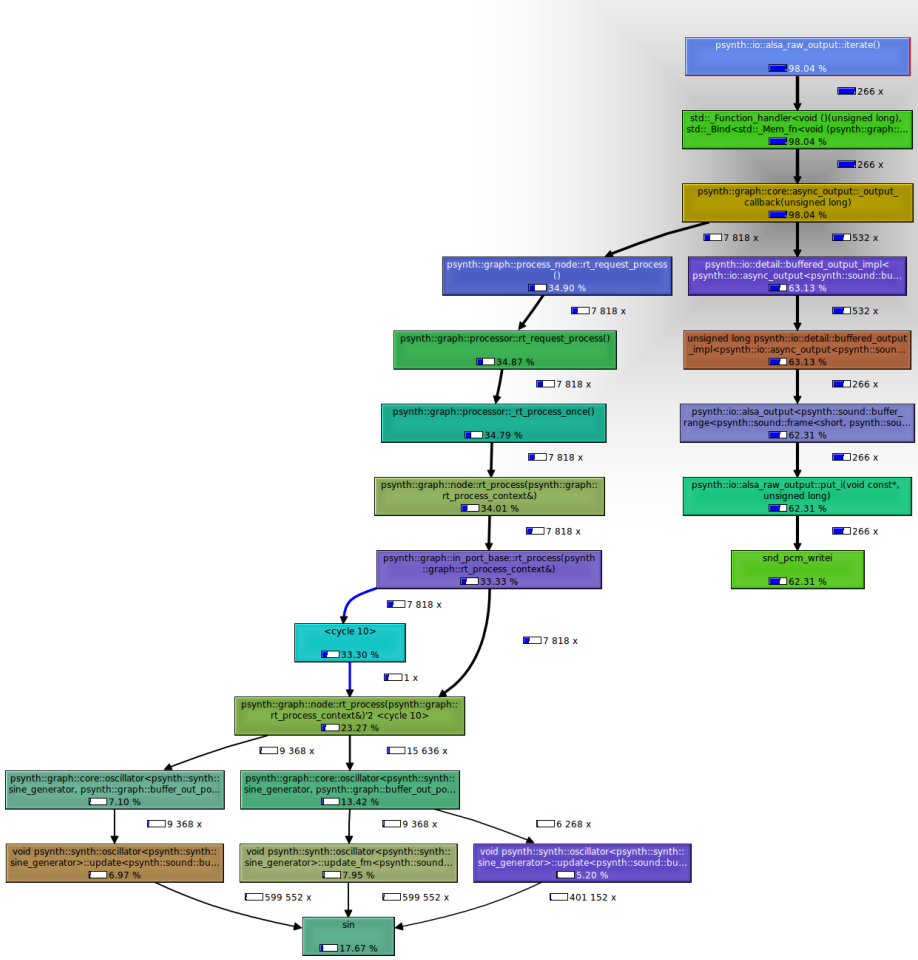


Figure 37.: Overview of the profiling call graph in the sound processing thread.

but it is done just after it is launched and before we do anything useful, and thus it only bloats the profiler output unnecessarily.

Then, we run the program with parameter `-r` to track real-time threads and `--all` to display information about all mutexes:

```
$ mutrace -r --all ./example_graph_perf 2> mutrace-output
```

The most relevant bits of its output are illustrated in note **??**. As we can see, none of the three mutexes used in the real-time thread is ever contended. This is not chance, repeated execution of the program yields

the same results; it is so because we always `try_lock` from the real-time thread.

Note that the third mutex in the list — mutex #7 — is the mutex that synchronises calls to `rt_request_process` — this can be elicited from the `mutrace` output that was removed from this summary. Because this mutex is locked exactly during each iteration of the audio processing, two important performance facts can be derived from its numbers:

- It was locked for a sum of 248.458*ms* out of a total of 10065.873*ms* running time. We can conclude from this that the sound processing took only a 2.465% of the total running time, which insinuates that our synthesis graph can grow much larger.
- It was locked 0.036*ms* on average and a peak of 0.709*ms*. The program was run with a block size of 64 frames and a frame rate of 44100*hz*, giving a maximum latency of 1.45*ms*. This suggests that it is extremely unlikely for buffer underruns due to software overload.

However, it is rather intriguing the presence of inconsistent mutexes[18]. The fact is, we are *always* locking mutexes with `unique_lock`, so it is impossible for a mutex to remain unlocked on scope exit. The further stack traces that `mutrace` dump and that was removed from the fragment — it can be checked in the CD too — suggest that it is an incompatibility between `mutrace` and the sometimes subtle internal use of `pthreads` in the C++0x threads API GCC implementation.

Note 4.3 (`mutrace` output for the performance test program)

```
...

mutrace: Showing 12 most contended mutexes:

 Mutex #   Locked  Changed  Cont. tot.Time[ms] avg.Time[ms] max.Time[ms]  Flags
       3     7435      205      0        1.820        0.000        0.011 M-R--.
```

18 A mutex is said to be inconsistent when it remains locked after the owner thread terminates. The threading system unlocks it automatically, but it can signify of a software logic problem.

```
       6     7028        3      0        1.862        0.000        0.020 M-R--.
       7     7025        0      0      248.458        0.035        0.709 M-R--.
      11     1065        0      0        4.427        0.004        0.099 M!---.
       2      429        0      0        0.294        0.001        0.022 M----.
       1       26        0      0        0.265        0.010        0.215 M----.
       0        8        0      0        1.093        0.137        1.046 M----.
       9        5        0      0        0.087        0.017        0.070 M!---.
       4        4        0      0        0.011        0.003        0.007 M----.
       8        4        0      0        0.007        0.002        0.004 M----.
      10        1        0      0        0.023        0.023        0.023 M----.
       5        1        0      0        0.000        0.000        0.000 M----.
                                                                        ||||||
                                                                        /|||||
          Object:                                  M = Mutex, W = RWLock /||||
           State:                               x = dead, ! = inconsistent /|||
             Use:                               R = used in realtime thread /||
      Mutex Type:                 r = RECURSIVE, e = ERRRORCHECK, a = ADAPTIVE /|
  Mutex Protocol:                                   i = INHERIT, p = PROTECT /
    RWLock Kind: r = PREFER_READER, w = PREFER_WRITER, W = PREFER_WRITER_NONREC

mutrace: Total runtime is 10065.873 ms.

...
```

The twin tool `matrace` also provides relevant evidence on the correctness of our program. We ran it typing:

```
$ matrace ./example_graph_perf
```

Note 4.4 shows the most relevant parts of its output. There are 5 allocations and 4 frees in the real-time thread but they are not alarming at all. The stack traces removed from that output summary reveal that they were performed just when the thread started — as a product of copying the `std::function` that the `std::thread` constructor takes as argument — and just when the thread is about to be released — performed by the *pthreads* internal data structures management.

Note 4.4 (`matrace` output for the performance test program)

```
matrace: Total of 7381 allocations and 5747 frees in non-realtime
```

```
threads in process lt-example_grap (pid 1617).

matrace: Total of 5 allocations and 4 frees in realtime threads.
```

This section provided enough evidence that the performance of the program is quite acceptable and that it meets our minimum guarantees to control non-determinism and prevent clicks due to buffer underruns.

4.3.3 Integration and deployment

Section 4.2.7 discussed how we chose not to integrate this new module into the upper layers of the application yet. We shall wait for the next iterations of the project — which are out of the scope of this master's thesis project — to fully stabilise this API and make the work of adapting the UI worth. However, we wrote some example test programs that test interactions between modules further than the unit tests. These programs are built in the `src/test` directory and are:

EXAMPLE_GRAPH_SCALE Plays an *A* note jumping among octaves among other events. It was one of the firsts test programs developed and helped to ensure that the control classes works properly.

EXAMPLE_GRAPH_PATCH Plays some very strange sounds with patches nested up to several levels, with some patch ports that even change names during the playback and some other things. It helped to ensure that the interacting actors in the hierarchical patches system do work properly.

EXAMPLE_GRAPH_PERF This is the program that we described in the last section. It tries to simulate a rather dynamic and stochastic interaction with the system and ease profiling.

EXAMPLE_GRAPH_OUTPUT Tests outputting to two different proactive devices at the same time. Its synthesis network is very similar to that of the profiler program back in figure 36. Apart from the output

device that is already shown there, there is a second *async output* connected on the output of the first synthesis patch. There are no random events; each generator-modulator has fixed constants with different out of phase frequencies. On one of the output devices one should listen to the sum of the two vibrating sinusoids, on the other just one of them. The test succeeded emitting at the same time on an on-board "Intel 5 Series/3400" sound card and an "M-Audio Xponent DJ Controller" USB integrated audio card. We ran the test for as long as 15 minutes to check whether clicks would be produced eventually — due to, for example, some small speed differences in such dissimilar devices — with no single clicks.

EXAMPLE_GRAPH_SOFT This program ensures that the ports with automatic reconnection softening actually do work properly. The network it executes is represented in figure 38. It generates three files, `graph-soft-N.wav`.

As illustrated in the figure, the program starts with a sine generator connected at the first of the two inputs of the greater patch — these inputs do connetion softening. It generates 1024 samples of a $880hz$ sinusoid — a pure A_5 in the ISO standard tuning [36] — and then it changes the connection to the second port that takes the signal into the patch. The audio output in the three files should then be something like what we described back in figure 34 in section 4.2.5. We should expect the `graph-soft-1.wav` file to get something like the red line in that figure, a fade out after the disconnection. Then, the `graph-soft-3.wav` should show something like the green signal, a fade in when the generator is changed from one port to the other. Finally, the `graph-soft-2.wav` file should contain the sum of those two signals, similar to the dotted blue line in the plot — it should remain as a stable pure tone without artifacts when the reconnection is made.

The system passes this test as shown in figure 39 where the output files are visualised with the Audacity program. When changing the

connection, the signal is properly faded out and in avoiding any audible distortion.

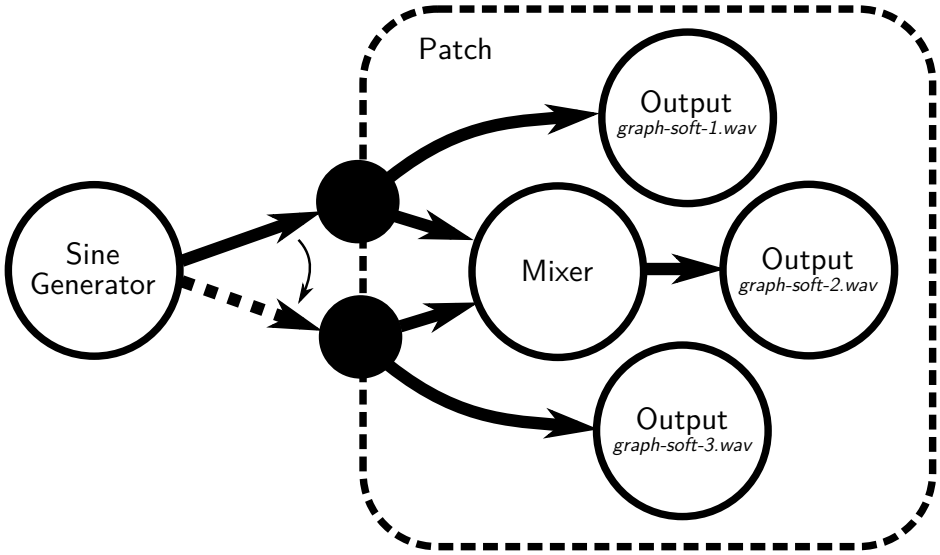


Figure 38.: Synthesis network of the reconnection softnening test program.

There have been several bug fixes that has happened parallel to this development. Also, while not used by the user interface code yet, it seemed reasonable to make the new API available to the public such that it can be validated by third parties. Thus, Psychosynth 0.2.1 was released at the end of this iteration. It was peer reviewed by project collaborators, mainly by the maintainer of the Ubuntu/Trinux packages Aleksander Morgado. The official *changelog* briefing for this release is included in note 4.5. Note that Ubuntu packages have not been built yet for this release because we have to wait until Oneiric's release in October to get GCC 4.6 as the default compiler in Ubuntu.

Note 4.5 (Changelog of Psychosynth 0.2.1)

- *Now real-time priority is properly acquired for the audio thread in POSIX systems whenever possible.*
- *New modular synthesis engine included in the framework — not yet used by the UI, providing:*

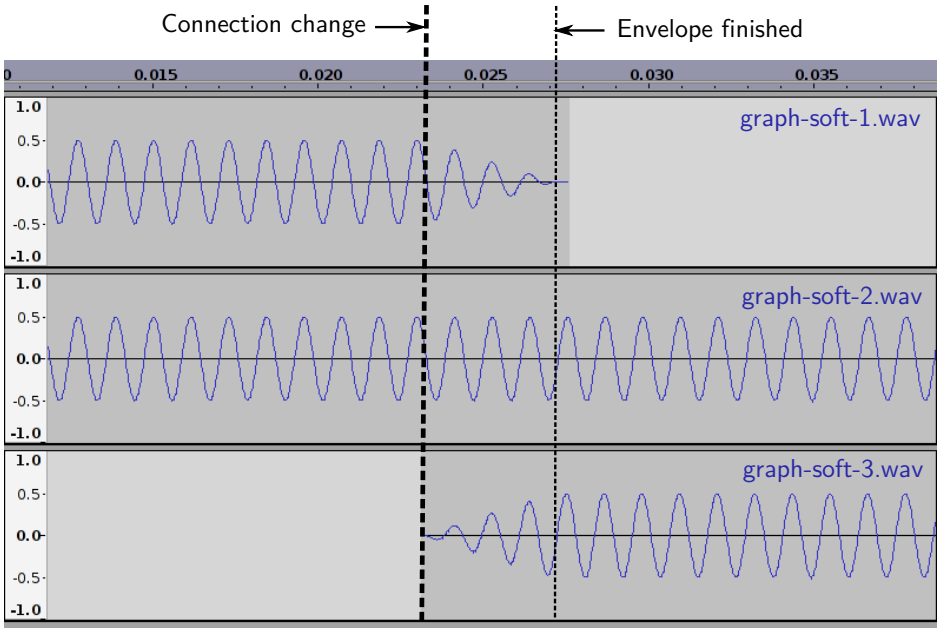


Figure 39.: The three output files in the reconnection softening test program. The signals are, in this top-down order: `graph-soft-1.wav`, `graph-soft-2.wav` and `graph-soft-3.wav`

.

- *Hierarchical patches. One can have a full patch inside one node that may be later store and or loaded from the hard drive.*
- *Any kind of signal can be processed now.*
- *Multiple output and input ports can coexist.*
- *Plug-in system is much easier to implement on top of the new system.*

- *New code uses some new C++0x features implemented in new compilers only. With this release, GNU Psychosynth requires GCC 4.6 to compile.*

4.4 CONCLUSION

In this second and last iteration of this project we rewrote the whole modular synthesis engine of the Psychosynth software in an attempt to solve the design problems of the previous implementation and set up a solid base capable of being extended with all the new features that are planned for the future. Influenced by Van Roy's work [69], multi-paradigm programming aided in writing an scalable design: (1) generic programming and policy based design helped in avoiding redundancy in the implementation, improves type safety thanks to compile-time bound polymorphism and abstracts factors orthogonally, (2) object-orientation provides dynamic reconfigurability and aided in large-scale design and global system modularity, and (3) functional programming allowed abstracting the control flow and helps in controlling state and the exponential complexity growth when mixing concurrency with stateful algorithms.

4.4.1 Benefits and caveats

There are few advancements in the new code that we should highlight:

1. Writing thread-safe code is usually quite challenging. Doing so in a real-time system is even harder. We abstracted the control flow using an event passing system that eases extending the program with new facilities that require inter-thread communication.
2. The new system is much more general and type safe in the processing. It is capable of processing signals of any kind in a type-safe manner — whenever possible — or throwing exceptions on type violations where dynamism is unavoidable.
3. The new design is much more decoupled, permitting the extension of the different components individually. For example, the fact that ports and controls are now objects instantiated by the concrete node type by itself, allows the development of new patterns of inter-node communication by deriving from these classes. A concrete

example of this is the connection softening. That used to require a lot of work from the DSP implementer and it is now transparent and optional because an specific port object does the job. Another positive consequence of this new design is that implementing a plug-in system is now much more straightforward.

There are some many other minor implementation improvements, like much better error control, improved compilation process via the usage of forward declarations and the absence of data races and dead locks that the former implementation suffered.

This new implementation uses C++0x features and advanced C++ implementation techniques, so most of the reasoning in the conclusions of the last chapter on the barrier that this might be for novel developers applies up to some extent (section 3.4.1). Still, in this layer, most of this complexity is hidden in the implementation and does not get through the public interface. Most of this complexity was caused by how template programming errors can not be easily transcribed by compilers, but most of the time, writing new DSP nodes or using the external graph and processing interface relies on simpler and easier to use object-oriented concepts that do not yield cryptic error messages.

4.4.2 Future work

Our requirements being already met, there are few ways we can later develop this module. Some small advancements have been proposed already during the design description, yet we can recapitulate some larger side projects that could implemented in the future.

4.4.2.1 *Lock-free heterogeneous deques*

Our triple-buffered event deques prevent locking from the RT thread as long as the "flip" can be done conditionally. This means that if a event is being added to back deque from the user thread just when the RT thread is attempting the flip, the flip might be skipped and events

in the back buffer would need to wait for the next control iteration. Inserting user event happens sparely enough for this pattern to work, as our experiments show. However, this prevents implementing some other techniques that are more demanding — we assumed that events are caused by user interaction most of the time, but it can be interesting for some applications to generate huge amounts of events programatically.

If our heterogeneous deques where lock-free — i.e. they support one reader and one writer at the same time in different threads — the flip would need no locking at all and it could be always achieved regardless of what the user thread is doing. We already cited Valois' work on lock-free data-structures [86, 54]. Our heterogeneous deques unique nature do not have a direct mapping to any of the data structures in the bibliography, but some authors provided design techniques to elicit our own lock-free datastructures. Harris improvements for node-based structures can be of great help [32]. There is plenty of other bibiography we can refer to[19].

4.4.2.2 *"Futures" API for the asynchronous events*

We described how the "asynchronous operation" thread can be used by the RT thread to delegate costly not urgent operations — like allocating heap memory or doing I/O. Very often, that requires returning a result back — for example, the pointer to the allocated memory. It is not hard to do so, but it requires adding extra redundant code. The C++0x standard includes the notion of *future*, a value that is computed, potentially, in a separate thread, leading to more declarative concurrency [33]. Still, these are not usable in a real-time context, because computing a future requires creating a dynamic functor, launching threads, allocating memory and so on. However, its API can be emulated on top of our thread communication infrastructure, simplifying this common pattern of requesting some computation in the "async thread" and using its result later. In some way, this is just extending the event system with a notion of "event with a return value".

19 We do not want to flood this document with references to lock-free algorithms. A good compilation has already been made by Ross Bencina: http://www.rossbencina.com/code/lockfree

4.4.2.3 *Compiling static networks*

In this iteration we concentrated on the dynamic features of our synthesis system. This often trades-off performance. This is ok as long as enabling live music is our most important objective. However, getting some nice sounds often requires sitting down in the studio and doing *sound design* and implementing complex mathematical functions by assembling lots of low-level modules into a patch. These patches are too low-level to be manipulated in a live performance and thus dynamism can be exchanged for a performance gain, needed to handle such big amounts of small nodes. Some software implementing these kind of features, like Reaktor's "Core-Cell Modules", traduce the network into a compiled form optimised for fast execution. This can be a very interesting development to broaden the scope of the project. However, it requires a lot of effort and is not very urgent — sound design and DSP programming can already be done either directly in C++, or in other synthesis systems and then plugged in when the plug-in system is ready — so we propose it as later future work.

5 CONCLUSION

> La dialectique est une machine amusante qui nous conduit / d'une manière banale / aux opinions que nous aurions eues en tout cas.
>
> *Manifeste Dadá* (1918)
> Tristan Tzara

This master's thesis project served, mainly, two purposes. First, to recapitulate, rationalise and re-focus a long-term project that has served its author as learning sketchbook for many years. Second, to give the first big steps in lifting the software to a professional quality grade. Of course, there is a lot of work to be done, yet crucial achievements have been made during the last year. In the analysis chapter the project was planned for a long-term development in an iterative process. This allows us to concentrate on feasible mid-term goals while building a broadly scoped system. Two of these iterations have been successfully accomplished.

Coming from a computer-science background, admittedly, we are still quite amateur in many musical and even digital signal processing aspects. Instead, we focused in what we can do better — exploiting programming languages abstraction facilities, making scalable architectures and applying and eliciting design patterns in order to write shorter, safer and faster code. The upcoming new C++ standard provided us with a big open ground for research in new ways of using multi-paradigm programming for our purpose. The most novel part of our work is concentrated in our first iteration, where template metaprogramming and concept-based design allowed us to write a sound processing library that is generic yet efficient and safe. The last stage also included relevant novelties —

like our polymorphic event system for real-time processing — yet being more pragmatical and focused on requirement satisfaction and enabling concrete features, like a hierarchical patches, a plug-in system, and so on. We should not, though, look at these advancements indulgently. Their own chapters further discusses their benefits, caveats, and open ground for improvement.

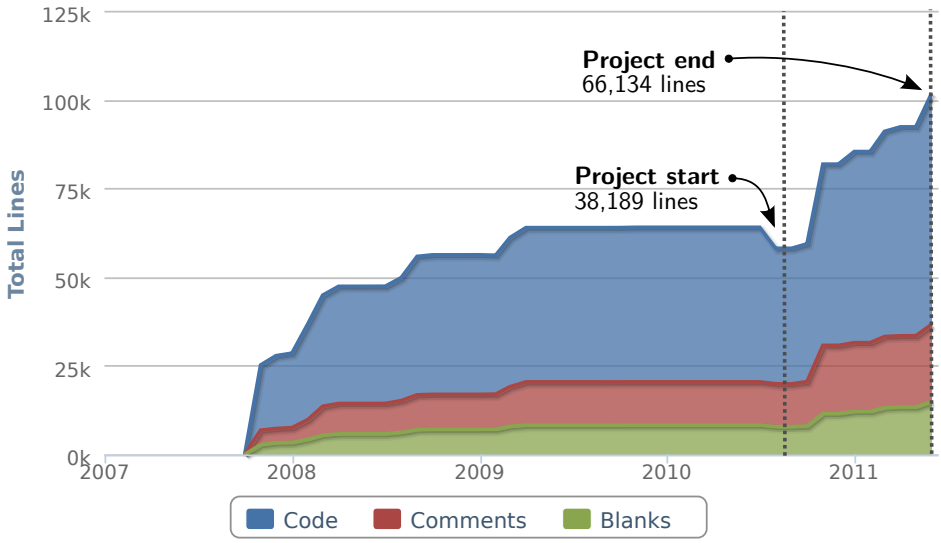


Figure 40.: Development progress in code lines.

Figure 40 shows the project growth in lines of code, measured automatically from the sources repository by the Ohloh webpage[1]. Around 30000 new lines of code have been added during the last year. However, most of our work have involved fully rewriting legacy code — in other statistics, Ohloh states that around 100000 lines of code have been changed during all the commits. This have been the biggest effort since the participation of the project in the Free Software Contest. Including *"logic code only"* Ohloh estimates that the total effort of the project — since its beginning in mid 2007 — in 12 person-years. It used the *Basic COCOMO* [16] method with coefficients $a = 2.4, b = 1.05$ to yield such result.

When looking into the future there are many possibilities. We are half way to fulfil the long-term goals that were proposed in the first two

1 Ohloh is a social network for Free Software development. Visit Psychosynth's page for further statistics: https://www.ohloh.net/p/12930

chapters. Apart from following our current plan blindfolded, there are some developments that may make us re-plan part of our future work.

Figure 41.: GNU Psychosynth's new webpage design.

First, there is the *Collaborative Roadmap*. As part of the redesign of the project's web-page that we did in March (see figure 41) we included a feature request voting system based on micro-donations. The Flattr[2] social micro-payment network works as follows: each person puts some money into her account each month. Then, different *things* — music, software, pictures — get a web button that users can press to show support. At the end of the month, the money in someones' account is distributed among all the creators of the things she clicked. Usually, each individual click does not give much, but very popular things can get a significant amount of money. Also, each Flattr button has a counter with the number of people that pressed it. We use this system in an special

2 http://flattr.com

way: each potential feature not yet developed has its own Flattr button. If someone thinks some feature is more urgent than another, he can push for his opinion to be taken into account by showing some commitment by making a micro-donation in this system. Table 7 shows the current ranking in the Collaborative Roadmap.

Votes	Thing
11	GNU Psychosynth — general
6	Audio input
4	MIDI support
1	Plug-in system
1	Tangible user interface
1	VST plug-ins
	...

Table 7.: Current ranking in the *Collaborative Roadmap*, as in June 2010

The Collaborative Roadmap credibility depends on whether we actually take it into account. Thus, now that this master's thesis project is over, we will devote some time to the most voted items. Indeed, the "Audio Input" feature seems quite feasible and fun to have, so... why not?

Moreover, the growing interest in the project are opening the doors for quite interesting collaborations. The ArtQuimia Music Production School[3] has been supporting the project since we started our requirement elicitation a year ago, and we are grateful for this. More recently, Miguel Vazquez-Prada contacted us to join efforts in building experimental tangible instruments. He has already built a ReacTable clone called Tangiblex[4] with impressive results (figure 42). Hopefully this is the beginning of a very fruitful collaboration. Indeed, a hardware-software pair is probably one of the best ways to build a business model for a consumer oriented application while respecting user's freedom with non-privative licenses. Also, Intel is offering a wide range of grants and other kinds of support and partnership programs to promote their Meego platform and Atom-based devices. That might be a good chance to

3 http://www.artquimia.net/
4 http://www.tangiblex.net/

prioritise a port for multi-touch devices — a very interesting development with open grounds for experimentation and innovation.

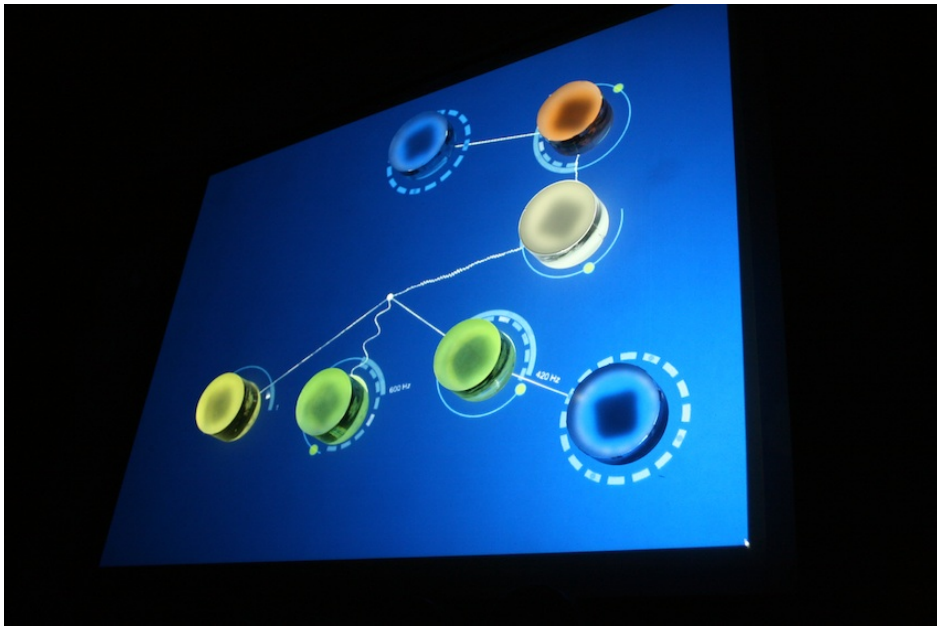


Figure 42.: The Tangiblex ReacTable-alike tangible instrument.

Anyway, whatever happens next, we have enjoyed and learnt a lot during the development of this project. That is, indeed, all what one should expect from a master's thesis project.

Thanks a lot for reading this document.

Appendices

A USER MANUAL

This document constitutes the user manual for Psychosynth, it is intended to help resolve issues in the use of the Psychosynth software package.[1]

A.1 INSTALLATION

A.1.1 Installing from the source code

First of all you must get a copy of the sources of this program. Go to the *download section*[2] if you do not have them yet. Note that this section describes only how to install from the source code — if there is a binary package for your distribution, use that instead.

A.1.1.1 *Dependencies*

Then, to try the software you will need these third party libraries and programs:

- GNU Autotools (only for the development version)
- Ogre (needed by the 3D interface)
- CEGUI (needed by the 3D interface)

1 This document was first written in Spanish, available here:
http://psychosynth.com/index.php/User_manual/es.
This document is a modified version of Ben Mullet's translation of that document, which is also accessible in the software's web page, integrating other older documents like the "Installation Guide".
http://psychosynth.com/index.php/User_manual

2 http://psychosynth.com/index.php/Download

- OIS (needed by the 3D interface)
- liblo (needed for the network support)
- libxml2 (needed for XML config support)
- Alsa (needed for ALSA sound output)
- Jack (needed for Jack sound ouput)
- libsndfile (needed for pcm file support)
- libvorbis (needed for OGG vorbis file support)
- SoundTouch (needed for sample stretching)
- Several Boost libraries

In Debian and Ubuntu you can install all those dependencies with the following command. Anyways, I suggest installing *liblo* from the original sources because the version in the repositories is outdated and contains a bug:

```
# apt-get install automake libtool libogre-dev \
      libceguiogre-dev libois-dev libcegui-mk2-dev \
      libasound2-dev libjack-dev liblo0-dev \
      libsndfile-dev libxml2-dev libsoundtouch1-dev \
      libvorbis-dev libboost-all-dev
```

A.1.1.2 *Installing*

If you downloaded the program from Bazaar you will first need to generate the compilation scripts:

```
$ autoreconf
```

Now you will need to run the configuration script to detect the the libraries and set up the compilation settings.

```
$ ./configure
```

Check that everything has been detected correctly and that everything that you want to install is going to be actually built — for example, if

you did not install Ogre, the library and CLI interface will install anyway but you will miss the 3D interface. Then compile the program:

```
$ make
```

At last we must run these commands with superuser privileges to install:

```
$ make install
$ ldconfig
```

We can now run the 3D simulator and enjoy:

```
$ psynth3d
```

A.1.1.3 *Troubleshooting*

Trouble 1 *On Ubuntu 8.10 or Debian Sid I get a linkage problem related to CEGUI::Exception*

You must add this repository to your `sources.list` and update Ogre and Cegui to their latest versions:

```
deb http://ppa.launchpad.net/andrewfenn/ubuntu hardy main
```

Trouble 2 *On Ubuntu 8.04 I can't find Boost 0.35*

You can find proper `.deb` packages here: https://launchpad.net/ubuntu/intrepid/+source/boost1.35/

Trouble 3 *On Ubuntu 8.04 I have properly installed* libsoundtouch *but it is not detected*

Run this command before executing configure:

```
$ sed s/soundtouch-1.0/libSoundTouch/ configure > configure
```

A.1.2 Installation for Ubuntu and derivatives

Follow these instructions to install Psychosynth if you are using Ubuntu or any other derivative (like Trisquel GNU/Linux[3], a fully free, as in freedom, distribution).[4]

These packages have been tested in Ubuntu Lucid (10.04), Ubuntu Maverick (10.10) and Ubuntu Natty (11.04), as well as other older releases not covered in this PPA.

Please note that since Maverick, the Ubuntu kernel comes without OSS support, so only ALSA or JACK output are supported. See the Troubleshooting section below.

A.1.2.1 *Easy setup*

If you just want to go the easy way, execute these commands:

```
$ sudo apt-add-repository ppa:gnu-psychosynth-team/ppa
$ sudo apt-get update
$ sudo apt-get install psychosynth-gui psychosynth-samples
```

A.1.2.2 *Manual setup*

In case you maintain manually your sources.list, here you have the detailed manual setup instructions. More information can be found in the GNU Psychosynth PPA webpage.

```
deb https://launchpad.net/~gnu-psychosynth-team/+archive/ppa lucid main
deb-src https://launchpad.net/~gnu-psychosynth-team/+archive/ppa lucid main
```

Where it says `lucid` you may also write a later version of the Ubuntu distribution (e.g Maverick). You may need to tell apt to accept those repos without issuing a warning when running apt-get update. For that, you can just do the following for each warned key:

3 http://trisquel.info/
4 Thanks a lot to Aleksander Morgado who made these packages and wrote this section.

```
$ gpg --keyserver subkeys.pgp.net --recv 4A3B72F02ADA7053
$ gpg --export --armor 4A3B72F02ADA7053 | sudo apt-key add -
```

Once you run `sudo apt-get update`, install the packages:

```
$ sudo apt-get update
$ sudo apt-get install psychosynth-gui psychosynth-samples
```

A.1.2.3 *Troubleshooting*

Trouble 4 *Sound comes terribly delayed (up to 30s).*

If you have ALSA configured in *Settings* → *Audio* → *Output* and you get the audio veeeery delayed, then try to use another Alsa device instead of the `'default'` one, like `'hw:0,0'`:

```
$ psynth3d -o alsa --alsa-device hw:0,0
```

Trouble 5 *No sound yet!*

Try to use JACK output, this always fixes the problem, in *Settings* → *Audio* → *Output*:

```
$ sudo apt-get install jackd
$ jackd -d alsa &
$ psynth3d -o jack
```

A.2 3D USER INTERFACE

Psychosynth comes with different programs that allow you to graphically manipulate synthesizer elements or communicate among them. Right now there are two user interfaces, a three-dimensional graphical interface and the command line.

To run the 3D interface enter the command:

```
$ psynth3d
```

Then you should see a window like in figure 43.

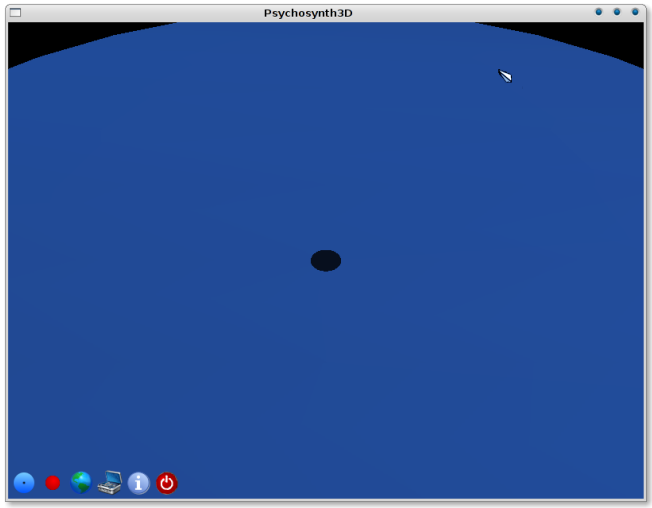


Figure 43.: Initial state of 3D the user interface.

At the bottom of the screen are a series of icon buttons that will enable us to deploy different windows to control Psychosynth. In the following order, the buttons are:

OBJECT SELECTOR Selects objects to place on the workspace.

SOUND RECORDER Records to a file all the sound being generated.

NETWORK SESSIONS Connects the synthesizer with others across a network.

INFO PANEL Displays information panels about the program and some help.

EXIT Exit the program.

In the center of the screen is a blue table where Psychosynth objects are placed to create and manipulate sound, as we shall see below.

Now let's see how we might do this in a three-dimensional environment.

A.2.1 The camera

The first thing we must get used to is manipulating the *camera*. It might seem that the environment is only 3D eye candy; in practise it adds important aspects of control over the synthesiser.

3D provides an intuitive means to adjust a Psychosynth object's controls: when zoomed out we can see many objects and easily move them between relatively distant points. On the other hand it is better to zoom in when we wish to modify an object parameter precisely. Mouse movements will represent shorter distances and thus increase the resolution.

Camera movements include:

MOVE To move the camera simply click the left mouse button on the table. The whole environment can be dragged around with the mouse. We can also focus the camera on a point by pressing the shift key while clicking the left button where we want to focus on the table.

ROTATE With no object selected we can rotate the camera focus around to a given point by moving the mouse while pressing the left button

ZOOM We can zoom the point that we have focused by either using the mouse wheel or by pressing the middle mouse button while moving the mouse forward and backward.

A.2.2 Creating sounds

To create sounds we will be placing some Psychosynth objects on the table. The central spot represents the main audio output, which will normally be our speaker.

To **place an object** on the table, click on the first icon button at the bottom of the screen. An object selector appears, the buttons are organized by category - allowing us to select and place different synthesis objects.

As the library of objects is growing rapidly there is no documentation yet on individual elements. Fortunately Psychosynth quickly allows you

to experiment with the objects — discovering what each one does and how they interact with each other — learning about audio synthesis in a fast, fun and highly intuitive manner.

When you click on one of the selector buttons you'll see a yellow circle that follows the mouse, indicating where the object will be placed. Then when we click on the table the object will appear there. If a sinusoidal oscillator is placed it will automatically connect with the center making a pure tone of 220 Hz.

The situation would then look like in figure 44

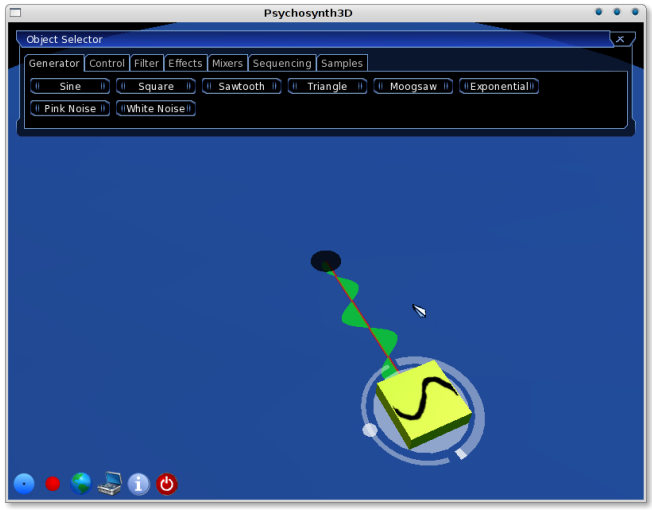


Figure 44.: A simple pure tone generator.

A connection between two objects will show us the wave being broadcast through it in real time, green if an acoustic signal and yellow if a control signal.

A.2.3 Modifying the sound

Psychosynth objects are connected with each other automatically. Two objects will be connected if they have compatible input and output ports, and if those ports are not already being used for other, shorter connections. Thus we can very quickly and intuitively alter the topology

of the synthesizer simply by moving the objects that we want to connect more closely.

To **move an object**, click the left mouse button and drag it onto the table.

Each object also has a number of **parameters**. For example, an oscillator object like the one we used earlier has two parameters — frequency and amplitude (or volume). The frequency is the **main parameter** of this object and we can see the relative value on an indicator that surrounds the subject on the right side.

To modify the main parameter simply move the mouse around the selected object while holding down the right mouse button, which rotates the object.

The other parameter — in this case the amplitude — is represented by a slider on the other side of the object. To change it click the link to the left of the object and move the slider.

There may be other parameters that are not directly represented on the object, sometimes because they are less important and sometimes because it is not yet decided how to represent them visually. For example, the main parameter of a sampler is the playback speed — the amplitude is *secondary*. But we can also alter the Tempo independently — that is, change the playback speed without changing the key — and alter the pitch — play in a different key without changing the playback speed.

To list all of an object's parameters and to view and modify the numerical values, press the *'e'* key to display a parameters window. An example would be like in figure 45, this time showing a more complex scenario.

To **remove an object** from the table use the *'delete'* key.

We can also **select multiple objects** simultaneously which is useful if modifying their main parameters all at once — or deleting them — or perhaps moving them. To select multiple objects, hold down the shift key while selecting. To deselect an object in the group press the control key and click on the object.

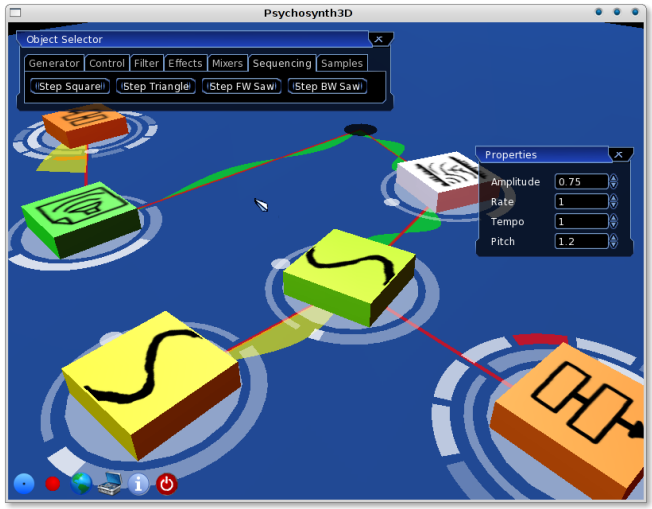


Figure 45.: A complex patch showing the full parameter editor.

Finally, we can temporarily **mute a connection** by clicking on it. It then turns black and any objects that attach to that connection in the hierarchy from the center will not be updated until we click on that link again.

A.2.4 Recording sound

To **record a session** just click on the red button that appears in the second position below. A window appears with the file name to which the sound is to be saved — and a start button. Figure 46 represents this situation.

We can modify the file name to be created, using any valid route from the current working directory. To start recording, click on Start and once finished, click on Stop.

A.2.5 Network sessions

To connect multiple synthesisers we need to establish a server computer and create one or more clients.

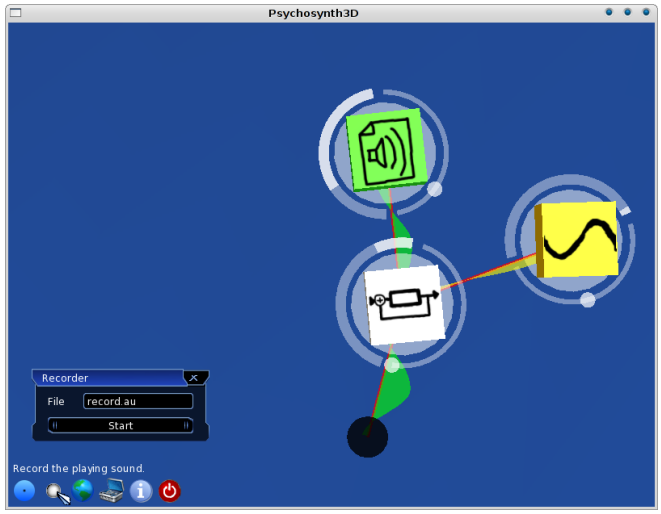


Figure 46.: The session recording dialog.

During a network session all events will be synchronized between connected synthesisers, but they will not be aware of any events that occurred before the connection. Therefore, it is recommended to clear the table before all computers are connected.

The third icon button in the row below the window opens network handling, which has two tabs, *client* and *server*. First, we need a **server** computer, so we go to the *server* tab. Then we can alter the listening port number - usually the default port is used. When you start the server it will be ready to receive connections and the box in the window displays: *"Server listening"*. The same box will show when someone is connected or disconnected — and similar events.

The other computers use the tab *client* to connect as **clients**. There we enter the IP address of the computer server where it says *Remote host* and the port in *Remote port*. We can also change the client listening port in *Local port*. Once we are ready to go, hit the *start* button.

A message that says *"Connecting..."* should appear. Once the connection has been made successfully we will see the message *"Connection accepted"*, or an error message. Figure 47 shows a successful connection.

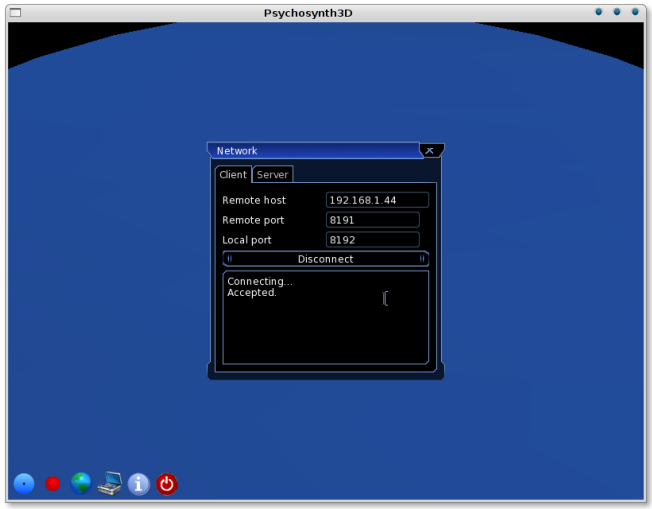


Figure 47.: Successful connection as client in a networked session.

A.2.6 Editing options

The program has various options that we can edit. To do so, we click on the *toolbox* icon button in the row below. These are the options that we will find in each category.

A.2.6.1 *Audio*

Here we can modify the sound generation options. Some of these are:

SAMPLE RATE The number of audio samples taken per second. Higher values give higher quality but need more computer power; it is recommended to use the default value. Please note that most sound cards only support a specific set of values.

BUFFER SIZE The number of samples that are computed in one block. Influences the latency directly, which can be calculated as:

$$latency = \frac{buffer\_size}{sample\_rate} seconds \tag{A.1}$$

Almost all sound cards require that this number to be a power of two. Note that a very low number will cause a greater CPU load and potentially cause buffer underruns with annoying clicks at high load, while very high values will slow the transfer to the audio generation system.

CHANNELS The number of audio channels. By default stereo sound is used.

OUTPUT Choose the output system.

The relative advantages of each setting depend on the platform used and the use it will give Psychosynth. Normally, depending on the output system we choose there will be some options available to the chosen output device.

A.2.6.2 *Video*

In this section we can change options for the on-screen 3D environment. Specifically:

WIDTH AND HEIGHT The width and height of the window in pixels. fullscreen: Select this for full screen. Note that the apply button does not work on systems using GLX due to problems in the Ogre library version.

FPS The refresh rate of the screen in samples per second. Higher values consume more CPU power and lower values make the animation jerky.

A.2.6.3 *Paths*

Here we configure the paths where the program searches for data. For now we can only change where samples are being sought, eg audio files that we can find in the samples section of the objects selector.

Once we have finished altering the list of search paths click *refresh* to update the list of samples in the objects selector. Note that Psychosynth only uses samples that are in `.wav`, `.au`, `.aiff`, `.ogg` or `.flac` format.

A.3 THE COMMAND LINE INTERFACE

There is a version of the synthesizer that runs from the command line.

In this version we cannot change the environment directly, but we can use it as a network server, using the option `-server` or as a client in a network using `-client`.

We can also hear all the sounds being generated, it is a real synthesizer; only the state of the scenario is not displayed and parameters can only be changed through the network.

For complete information on available command line options use `-help`.

B GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

http://fsf.org/

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a

specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with

all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the

Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a

disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

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, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation 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. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part

into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

BIBLIOGRAPHY

[1] Abrahams, D., and Gurtovoy, A. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004. (Cited on page 71.)

[2] Alexandrescu, A. An implementation of discriminated unions in c++. In *Second Workshop on C++ Template Programming* (2001), OOPSLA 2001, ACM. (Cited on page 131.)

[3] Alexandrescu, A. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. (Cited on pages 46, 71, 126, 130, 133, and 156.)

[4] Alexandrescu, A., and Marginean, P. Generic<Programming>: Change the Way You Write Exception-Safe Code—Forever. *C/C++ User Journal 18*, 12 (Dec. 2000). (Cited on page 112.)

[5] Amatriain, X., Arumi, P., and Garcia, D. Clam: a framework for efficient and rapid development of cross-platform audio applications. In *Proceedings of the 14th annual ACM international conference on Multimedia* (New York, NY, USA, 2006), MULTIMEDIA '06, ACM, pp. 951–954. (Cited on page 26.)

[6] Andersen, T. H. Mixxx: towards novel dj interfaces. In *Proceedings of the 2003 conference on New interfaces for musical expression* (Singapore, Singapore, 2003), NIME '03, National University of Singapore, pp. 30–35. (Cited on page 7.)

[7] ANSI-ISO-IEC. *Programming Languages — C++, ISO/IEC 14882:2003(E) International Standard*, ansi standards for information technology ed., 2003. (Cited on page 133.)

[8] Beck, K. *Test Driven Development: By Example*. Addison-Wesley

Longman Publishing Co., Inc., Boston, MA, USA, 2002. (Cited on page 53.)

[9] Bencina, R., Kaltenbrunner, M., and Jorda, S. Improved topological fiducial tracking in the reactivision system. *Computer Vision and Pattern Recognition Workshop 0* (2005), 99. (Cited on page 23.)

[10] Benington, H. D. Production of large computer programs. In *Proceedings of the 9th international conference on Software Engineering* (Los Alamitos, CA, USA, 1987), ICSE '87, IEEE Computer Society Press, pp. 299–310. (Cited on page 51.)

[11] Bernardy, J.-P., Jansson, P., Zalewski, M., Schupp, S., and Priesnitz, A. A comparison of c++ concepts and haskell type classes. In *Proceedings of the ACM SIGPLAN workshop on Generic programming* (New York, NY, USA, 2008), WGP '08, ACM, pp. 37–48. (Cited on page 69.)

[12] Beveridge, J. Self-registering objects in c++. *Dr. Dobb's Journal 23* (August 1998). (Cited on page 157.)

[13] Blackman, R., and Tukey, J. *The measurement of power spectra, from the point of view of communications engineering*. Dover Publications, 1959, ch. Particular pairs of windows, pp. 95–100. (Cited on page 154.)

[14] Blaine, T. The jam-o-drum interactive music system: a study in interaction design. In *In DIS '00: Proceedings of the conference on Designing interactive systems* (2000), ACM Press, pp. 165–173. (Cited on page 23.)

[15] Blaine, T., and Fels, S. Contexts of collaborative musical experiences. In *NIME '03: Proceedings of the 2003 conference on New interfaces for musical expression* (Singapore, Singapore, 2003), National University of Singapore, pp. 129–134. (Cited on page 23.)

[16] Boehm, B. W., Clark, Horowitz, Brown, Reifer, Chulani, Madachy, R., and Steece, B. *Software Cost Estimation with Cocomo II with Cdrom*, 1st ed. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000. (Cited on page 182.)

[17] Boulanger, R., Ed. *The Csound Book: Perspectives in Software Synthesis,*

Sound Design, Signal Processing,and Programming. The MIT Press, March 2000. (Cited on page 25.)

[18] Boulanger, R., Lazzarini, V., and Mathews, M. *The Audio Programming Book*. The MIT Press, 2010. (Cited on pages 25, 124, and 154.)

[19] Bourdev, L., and Järvi, J. Efficient run-time dispatching in generic programming with minimal code bloat. *Sci. Comput. Program. 76* (April 2011), 243–257. (Cited on page 98.)

[20] Center, M. W., and Wright, M. Opensound control: State of the art 2003. In *In Proceedings of the 2003 Conference on New Interfaces for Musical Expression (NIME-03* (2003), pp. 153–159. (Cited on pages 23 and 159.)

[21] Coplien, J. O. Curiously recurring template patterns. *C++ Rep. 7* (February 1995), 24–27. (Cited on page 162.)

[22] Driesen, K., and Hölzle, U. 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* (New York, NY, USA, 1996), OOPSLA '96, ACM, pp. 306–323. (Cited on page 68.)

[23] Fletcher, H. Loudness and pitch. *Bell Laboratories Record 13*, 5 (January 1935), 130–137. (Cited on page 60.)

[24] Fowler, M. *Is design dead?* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001, pp. 3–17. (Cited on page 51.)

[25] Franke, H., Russell, R., and Kirkwood, M. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Proceedings of the Ottawa Linux Symposium* (2002), OLS '02, pp. 479–595. (Cited on page 125.)

[26] Fries, B., and Fries, M. *Digital Audio Essentials*. O'Reilly Media, Inc., 2005. (Cited on page 63.)

[27] Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, March 1995. (Cited on pages 43, 46, 128, 130, 141, 156, and 159.)

[28] Garlan, D., and Shaw, M. An introduction to software architecture. Tech. Rep. CMU-CS-94-166, Carnegie Mellon University, January 1994. (Cited on page 44.)

[29] Goldstein, E. B. *Sensation and Perception*, 6th edition ed. Wadsworth Publishing Compan, Belmont, California, 2001. (Cited on pages 60 and 153.)

[30] Gregor, D., Stroustrup, B., Järvi, J., Reis, G. D., Siek, J., and Lumsdaine, A. Concepts: Linguistic support for generic programming in c++. In *SIGPLAN Notices* (2006), ACM Press, pp. 291–310. (Cited on page 69.)

[31] Group, O. M. OMG unified modeling language (OMG UML) infrastructure version 2.3. Tech. Rep. formal/2010-05-03, 2010. (Cited on page 100.)

[32] Harris, T. L. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing* (London, UK, UK, 2001), DISC '01, Springer-Verlag, pp. 300–314. (Cited on page 178.)

[33] Hinnant, H. E. Multithreading api for c++ 0x — a layered approach. Tech. Rep. N2094=06-0164, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, September 2006. (Cited on page 178.)

[34] Hohn, T., Carot, A., and Werner, C. Netjack: Remote music collaboration with electronic sequencers on the internet. In *In Proceedings of the Linux Audio Conference 2009* (2009). (Cited on page 22.)

[35] Humphrey, W. S. Using a defined and measured personal software process. *IEEE Software 13* (1996), 77–88. (Cited on page 52.)

[36] ISO. *Acoustics – ISO 16:1975, Standard tuning frequency (Standard musical pitch)*, international organization for standardization ed., 1975. (Cited on page 173.)

[37] Järvi, J., and Freeman, J. C++ lambda expressions and closures. *Sci. Comput. Program. 75* (September 2010), 762–772. (Cited on page 142.)

[38] Johnston, I. H. *Measured tones: the interplay of physics and music*, 2nd ed. ed. Institute of Physics Publishing, Bristol; Philadelphia, 2002. (Cited on page 126.)

[39] Jordà, S., Geiger, G., Alonso, M., and Kaltenbrunner, M. 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* (2007), ACM Press, pp. 139–146. (Cited on pages 4 and 23.)

[40] Järvi, J., Marcus, M., and Smith, J. N. Programming with c++ concepts. *Science of Computer Programming 75*, 7 (2010), 596 – 614. Generative Programming and Component Engineering (GPCE 2007). (Cited on pages 68 and 71.)

[41] Kaltenbrunner, M., Bovermann, T., Bencina, R., and Costanza, E. Tuio: A protocol for table-top tangible user interfaces. In *In Proceedings of the 2 nd Interactive Sonification Workshop* (2005). (Cited on page 23.)

[42] Kaltenbrunner, M., Geiger, G., and Jordà, S. Dynamic patches for live musical performance. In *Proceedings of the 2004 conference on New interfaces for musical expression* (Singapore, Singapore, 2004), NIME '04, National University of Singapore, pp. 19–22. (Cited on page 13.)

[43] Kim, K. H. K. Basic program structures for avoiding priority inversions. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (Washington, DC, USA, 2003), ISORC '03, IEEE Computer Society, pp. 26–. (Cited on pages 114 and 125.)

[44] Kirin, P. Psychosynth: Free 3d music interface, as a virtual reactable. *Create Digital Music, online edition: http://createdigitalmusic.com/2008/12/psychosynth-free-3d-music-interface-as-a-virtual-reactable/ 2008* (December 2008). (Cited on page 7.)

[45] Knuth, D. E. Structured programming with go to statements. *Com-*

puting Surveys 6 (1974), 261–301. (Cited on page 165.)

[46] Krasner, G. E., and Pope, S. T. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program. 1* (August 1988), 26–49. (Cited on page 158.)

[47] Kruchten, P. *The Rational Unified Process: An Introduction*, 3 ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. (Cited on page 51.)

[48] Letz, S., Arnaudov, N., and Moret, R. What is new in jack 2? In *In Proceedings of the Linux Audio Conference 2009* (2009). (Cited on page 22.)

[49] Lipshitz, S. P., Vanderkooy, J., and Wannamaker, R. A. Minimally audible noise shaping. *J. Audio Eng. Soc 39*, 11 (1991), 836–852. (Cited on page 64.)

[50] Ludwig, A. Music and the human hearing. *Self published: http://www.silcom.com/~aludwig/EARS.htm, last accessed 03 of March 2011.* (2009). (Cited on page 63.)

[51] Magnusson, T., and Mendieta, E. H. 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* (New York, NY, USA, 2007), NIME '07, ACM, pp. 94–99. (Cited on pages 15 and 39.)

[52] McCartney, J. Rethinking the computer music language: Supercollider. *Computer Music Journal 26*, 4 (2002), 61–68. (Cited on page 25.)

[53] Mealling, M., and Denenberg, R. Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations. RFC 3305, Aug. 2002. (Cited on page 128.)

[54] Michael, M. M., and Scott, M. L. Correction of a memory management method for lock-free data structures. Tech. rep., University of Rochester, NY, USA, 1995. (Cited on pages 114 and 178.)

[55] MIDI Manufacturers Association Incorporated. *Complete MIDI*

1.0 Detailed Specification, 1999/2008. (Cited on page 15.)

[56] Moog, R. A. Voltage-controlled electronic music modules. In *Audio Engineering Society Convention 16* (10 1964). (Cited on pages 11 and 21.)

[57] Myers, N. C. Traits: a new and useful template technique. *C++ Report* (June 1995). (Cited on page 69.)

[58] Nethercote, N., and Seward, J. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 89–100. (Cited on page 166.)

[59] Niebler, E. Proto: a compiler construction toolkit for dsels. In *Proceedings of the 2007 Symposium on Library-Centric Software Design* (New York, NY, USA, 2007), LCSD '07, ACM, pp. 42–51. (Cited on page 116.)

[60] Nishibori, Y., and Iwai, T. Tenori-on. In *NIME* (2006), N. Schnell, F. Bevilacqua, M. J. Lyons, and A. Tanaka, Eds., IRCAM - Centre Pompidou in collaboration with Sorbonne University, pp. 172–175. (Cited on page 24.)

[61] Norton, J. Dynamic class loading in c++. *Linux Journal 2000* (May 2000). (Cited on page 157.)

[62] Orlarey, Y., Fober, D., and Letz, S. *FAUST: an Efficient Functional Approach to DSP Programming*. 2009, pp. 65–96. (Cited on page 26.)

[63] Patten, J., Recht, B., and Ishii, H. Audiopad: a tag-based interface for musical performance. In *Proceedings of the 2002 conference on New interfaces for musical expression* (Singapore, Singapore, 2002), NIME '02, National University of Singapore, pp. 1–6. (Cited on page 23.)

[64] Phillips, D. Give me 3 synths, part 3. *Linux Journal, online edition: http://www.linuxjournal.com/content/give-me-3-synths-part-3 2008* (May 2008). (Cited on page 7.)

[65] POSIX.1-2008. The Open Group Base Specifications. Also published

as IEEE Std 1003.1-2008, July 2008. (Cited on page 39.)

[66] Puckette, M. Pure data: another integrated computer music environment. In *in Proceedings, International Computer Music Conference* (1996), pp. 37–41. (Cited on page 21.)

[67] Puckette, M. Max at 17. In *Computer Music Journal* (10 2002), vol. 26 (4), pp. 31–43. (Cited on page 21.)

[68] Puente, J. P. B. GNU Psychosynth: Un sintetizador de audio por Software Libre. *Novática 195* (September 2008), 47–50. (Cited on page 43.)

[69] Roy, P. V., and Haridi, S. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA, 2004. (Cited on page 176.)

[70] Scavone, G. P., and Cook, P. R. Rtmidi, rtaudio, and a synthesis toolkit (stk) update. In *In Proceedings of the International Computer Music Conference* (2005). (Cited on page 25.)

[71] Siek, J. G., and Lumsdaine, A. Concept checking: Binding parametric polymorphism in C++. In *Proceedings of the First Workshop on C++ Template Programming* (Erfurt, Germany, 2000). (Cited on page 69.)

[72] Smith, S. W. *The scientist and engineer's guide to digital signal processing*. California Technical Publishing, San Diego, CA, USA, 2002. (Cited on page 63.)

[73] Spira, P. M., and Pan, A. On finding and updating spanning trees and shortest paths. *SIAM Journal on Computing 4*, 3 (1975), 375–380. (Cited on page 161.)

[74] Stallman, R. 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). (Cited on page 35.)

[75] Stallman, R. M. *GNU Coding Standards*. FSF, http://www.gnu.org/prep/standards/standards.html, last visited January 5th 2011, 1992–2010. (Cited on page 42.)

[76] Stallman, R. M., and Gay, J. *Free Software, Free Society: Selected Essays of Richard M. Stallman*. Free Software Foundation, June 2002. (Cited on pages 17 and 35.)

[77] Stepanov, A., and Lee, M. The standard template library. Technical Report HPL-94-34, Hewlett-Packard laboratories, 1994. (Cited on page 94.)

[78] Stepanov, A., and McJones, P. *Elements of Programming*, 1 ed. Addison-Wesley Professional, June 2009. (Cited on pages 69 and 75.)

[79] Sutter, H. 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.* (March 2010). (Cited on page 38.)

[80] Sutter, H. We have fdis! (trip report: March 2011 c++ standards meeting. *Sutter's Mill Blog: http://herbsutter.com/2011/03/25/we-have-fdis-trip-report-march-2011-c-standards-meeting/, last accessed 01 of July 2011.* (March 2011). (Cited on page 38.)

[81] Tanenbaum, A. S. *Modern Operating Systems*, 3rd ed. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007. (Cited on page 125.)

[82] Taube, H. An Introduction to Common Music. *Computer Music Journal 21*, 1 (1997). (Cited on page 26.)

[83] Tingen, P. Autechre, recording electronica. *Sound on Sound* (2004). (Cited on page 6.)

[84] Unruh, E. Compile time prime number computation. ANSI X3J16-94-0075/ISO WG21-462. Available online: http://www.erwin-unruh.de/primorig.html, 1994. (Cited on page 71.)

[85] Valois, J. D. Implementing lock-free queues. In *In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV* (1994), pp. 64–69. (Cited on page 114.)

[86] Valois, J. D. *Lock-free data structures*. PhD thesis, Troy, NY, USA, 1996. UMI Order No. GAX95-44082. (Cited on pages 125 and 178.)

[87] Vanderkooy, J., and Lipshitz, S. P. Dither in digital audio. *J. Audio Eng. Soc 35*, 12 (1987), 966–975. (Cited on page 64.)

[88] Vandevoorde, D., and Josuttis, N. M. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, Nov. 2002. (Cited on page 69.)

[89] Veldhuizen, T. Expression templates. *C++ Report 7*, 5 (June 1995), 26–31. (Cited on page 116.)

[90] Veldhuizen, T. L. C++ templates are turing complete. Tech. rep., Indiana University of Computer Science, 2003. (Cited on page 71.)

[91] Vlissides, J. *Pattern hatching: design patterns applied*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1998. (Cited on pages 130, 156, and 159.)

[92] Wang, G., and Cook, P. R. Chuck: a concurrent, on-the-fly audio programming language. In *Proc. ICMC* (2003), pp. 219–226. (Cited on page 25.)

[93] Weidendorfer, J., Kowarschik, M., and Trinitis, C. A tool suite for simulation based analysis of memory access behavior. In *In Proceedings of International Conference on Computational Science* (2004), Springer, pp. 440–447. (Cited on page 167.)

INDEX