DEPARTMENT OF COMPUTER SCIENCE

Hotpants - A software music synthesiser

Jonathan P. Crooke

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering

September 2010 | CSMSC-10

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Jonathan P. Crooke, September 2010

Abstract

Software music synthesisers allow musicians to combine the benefits of electronic synthesiser technology with the scalability and flexibility of software. In recent years, software instruments have become a common, if not central, feature of studios, and at the forefront of continuing research and development.

In this project, we intend to design and implement in software an expressive and professional-sounding digital music synthesiser, competitive with comparable commercial offerings, and suitable for a wide range of offline and real-time applications on standard, home-level computer equipment. The instrument will be implemented in C^{++}, interfacing with the well-established *CoreAudio* instrument and effect plug-in framework, part of Macintosh OS X.

HotPants is named in honour of the late soul musician, James Joseph Brown, Jr.
(May 3, 1933–December 25, 2006)

Contents

Chapter 1

Review

1.1 Introduction

Software music synthesisers are today a popular and powerful tool for musicians of all kinds. They afford the user access to an enormous range of sound creation tools, far in excess of that which was possible with a comparable quantity of traditional musical instruments only a few decades ago. We have, ourselves, been active users of software synthesisers on home computer equipment from their early days as limited off-line sound effect and loop generators, up to their current state as powerful and professional instruments. For this project, we intend to implement this kind of software music synthesiser ourselves, in order to develop skills in signal processing as well as own instrument design ideas.

In the following section we would like to introduce the reader to the historical context of this project, followed by the theoretical—Section 1.3, and technological—Section 1.4. We hope that this bottom-up approach will put the reader in a good position to understand the applications of synthesiser technology we will examine in Section 1.5. From this analysis we hope to derive a viable design and feature set for the HotPants instrument, which will be outlined in Sections 2.2–2.3, along with project management, scheduling, testing and additional implementation details in Sections 2.4–2.6.

1.2 Background

Computer music in software began with research conducted primarily at north American universities commencing in the 1950s. However, for decades the processing requirements of these instruments far outstripped that provided by commercial computer equipment, least not *home* computers. Instead, many musicians were first exposed to electronic instruments through hardware instruments, the ubiquitous analogue synthesisers of the 1960s and 70s, and later specialised digital hardware instruments, first arriving in the 1980s. It was not until the late 1990s that the power provided by the typical home computer became sufficient for electronic music, and brought software instruments within reach of the average musician. The history of hardware and software have, then, tread parallel paths; in order to understand the predecessors of the contemporary software synthesiser we will examine this period in more detail.

1.2.1 Earliest research

The earliest examples of digital sound generation took place in the late 1950s, with research undertaken at Bell Laboratories in Murray Hill, New Jersey [17, pg. 15] [22, pg. 18]. The father of computer music, Max V. Mathews and colleagues demonstrated that a computer was capable of producing sounds according to any pitch scale or waveform, including time-varying frequency and amplitude envelopes [16, pg. 87], elements of sound synthesis which form the foundation of any digital instrument[1]. This research led in 1957 to the development by Mathews of the *MUSIC I* program, capable of producing a single equilateral triangle waveform with a fixed dynamic shape. It would be used by the psychologist Newman Guttman to compose *In a Silver Scale*, the first composition synthesised by the process of digital-to-analogue conversion [17, pg. 15–16].

Mathews furthered his research with the development in 1958 of the *MUSIC II* program, and in 1960 *MUSIC III* in collaboration with Joan Miller [16, pg. 89]. MUSIC III was significant in introducing the *unit generator* (UG/UGen) concept, in that parts of an instrument could be modularised into a system of interconnected, *patched* elements such as oscillators, filters, envelopes and amplifiers [17, pg. 16]. This efficient and flexible approach continues to be the basis of music synthesisers to this day[2].

Research into computer sound synthesis began to diversify in the 1960s, with Mathews's *MUSIC IV* (1963) and in particular *MUSIC V* (1968) [9, chapter 3]. MUSIC V was exported to universities and laboratories around the world, introducing many to digital sound synthesis for the first time. It was significant in being written in standard Fortran IV, whereas earlier incarnations had been written in system-dependent assembly language [17, pg. 17]. This higher-level approach would go on to inspire similar programs and libraries such as *Music BF*, *Music 360*, *Music 7*, *Music 11*, *Csound*, *MUS10*, *Cmusic*, *Common Lisp Music* and others [16, chapter 17].

Off-line and real time instruments

It is important to note that those detailed above were not real-time instruments. During the early years, available processing power was insufficient for real-time processing and instead programs input notational data by way of a specialised scripting language, specifying synthesiser settings as well as musical notes. Programs would produce output to *file*, after considerable processing time. These processed audio files could then be output in real-time by the use of a *digital-to-analogue converter* (DAC), an entirely separate device[3]. Needless to say, these were not performance instruments, and save for exhibition concerts largely remained in universities and laboratories. However, hardware-based instruments were soon being built that were within reach of the average musician.

1.2.2 Analogue hardware synthesisers

The earliest commercial synthesisers were not digital. Even in specialised hardware units, digital synthesis would be prohibitively expensive until well into the 1980s. Instead, the instruments that became such an integral part of popular culture during the 1960s and 70s were analogue *subtractive voltage-controlled* synthesisers [1, chapter 4], most notably those

[1] See Section 1.4

[2] See Section 1.4.1

[3] In the case of Mathews's early experiments, the IBM 704 computer used for sound processing, and the DAC used to produce audio output were located in different cities [17, pg. 15]

designed by Robert Moog [15, pg. 8–11]. We discuss subtractive synthesis in Section 1.4. This approach was relatively cheap, at least in comparison to the astronomical expense of any kind of digital implementation at the time. Most importantly, it could produce instruments suitable for performance on stage and this was the first time that musicians had had wide access to electronic instruments. Sound design techniques developed on these instruments, as well as their hands-on control systems continue to influence digital instruments to this day, despite them having no internal similarities[4].

1.2.3 Digital hardware synthesisers

The 1980s brought the first generation of affordable digital hardware instruments from major electronics manufacturers such as Korg and Yamaha[5]. Digital synthesisers offered a number of advantages over analogue instruments: they were smaller, lighter and more easily maintained; they could easily support multiple *voices*, greatly increasing the polyphonic capability of an instrument[6]; later analogue instruments began to introduce patch[7] storage and recall systems but these methods tended to be rather unwieldy, reliant on linear storage systems such as cassette tape. Digital instruments were entirely better suited for this, making use of floppy discs and small amounts of fast internal RAM for storage. Furthermore, only with digital synthesisers did techniques such as *additive* synthesis and *frequency modulation* synthesis become viable[8]. These sounds would go on to become as popular and significant in their own right as those of analogue subtractive synthesisers.

1.2.4 The soft-synth era

With the popularisation of hardware digital synthesisers it was only a matter of time before software implementations of these designs became viable—*soft-synths*. After all, digital hardware instruments are little more than specialised pieces of microprocessor-controlled computer equipment.

Figure 1.1 illustrates *Simsynth*, a stand-alone instrument, and one of the earliest soft-synths developed for home computer use. Whilst well featured, it is a good example of an instrument limited by the capabilities of desktop computers of the time and able only to produce offline sound-effects. As processor power has increased it has became possible for modest hardware to host powerful real-time instrument comparable with their hardware-based ancestors[9].

Software synthesis offers affordability and maintenance advantages similar to those digital hardware synthesisers had offered over their analogue predecessors: software instruments can be patched and updated along conventional software support lines; they take up no physical space; creating multiple instances of the same instrument is also possible, only limited by available processing power. It is now possible to operate a range of instruments on a machine as portable as a laptop computer, containing devices equivalent in power to whole rooms of hardware equipment only a few decades ago. Popular vintage designs have also been emulated[10].

[4] See discussion of user interfaces in Section 1.4.5

[5] The Yamaha DX7 is discussed with regards to FM synthesis in Section 1.4.2

[6] See Section 1.4.1

[7] Synthesiser programs. More commonly known as *presets* in software synthesisers

[8] See Section 1.4.2

[9] See Section 1.5 illustrates a range of contemporary soft-synths

[10] See the Korg MonoPoly in Section 1.5

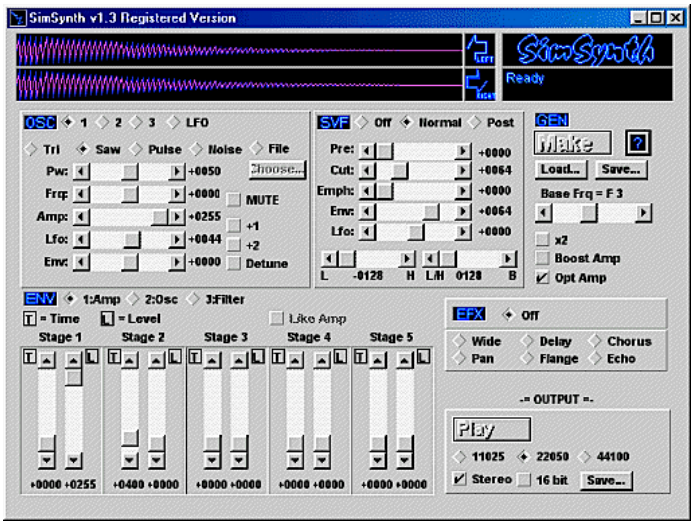


Figure 1.1: Simsynth v1.3, David Billen (circa 1995)

1.2.5 Current state of research

This brings us up to date with regard to the current state of available synthesiser technology. We have seen how the software synthesiser has caught up with its hardware-based predecessors and can now offer the same features and sound quality along with its own advantages. At this point, we would like to spend a little time talking about the the current state of research in the field, the most active being in the techniques of physical modelling and waveguide synthesis, primarily taking place at the *Center for Computer Research in Music and Acoustics* at Stanford University[11]. These techniques are discussed in Section 1.4.2.

1.3 Digital signal processing

To some extent, this project is an application of digital signal processing techniques, such as periodic sampling and the Fourier series. However, lacking experience in this area we consider it to not be a constructive use of time to attempt any original implementation of key signal processing elements of the instrument. In particular, we do not expect to implement our own set of digital filters, since this is an area where quality is of upmost importance and we are not confident of our ability to deliver a satisfactory result. As such, we expect to rely heavily on established and well-documented techniques and concentrate on the architecture of our own instrument. We are, however, wary of some of the common hazards in this kind of project, such as aliasing and the Nyquist frequency, and will attempt as best possible to tackle these issues suitably.

1.4 Synthesiser architecture

Now that we have given an historical and theoretical context for this project, it is possible to discuss in more detail the components of synthesisers. We will begin by further exploring

[11] `https://ccrma.stanford.edu/`

the *unit generator* concept before going on to look at their application in various synthesiser techniques such as those appropriate for HotPants. Introductory discussions for the full range of these technologies can be found in [16].

1.4.1 Unit generators

As first applied in Max Mathews's MUSIC III[12], the use of modularised UG architecture greatly improves the design of all synthesisers, hardware or software. The advantages are manyfold, as with the benefits of using interchangeable components in any system. Below we detail some of these commonly used units.

Oscillator

The oscillator is the most primitive signal generator in a synthesiser, placed at the start of the signal chain, generating some form of raw signal for later use and manipulation. The precise nature of the oscillator is dependent upon the particular synthesis technique in use and its place in the signal chain. For use in this project, we can define the oscillator as one that applies an algorithm to produce a simple periodic signal. Common oscillator waveforms include sine, triangle, saw-tooth, pulse and *noise*[13], illustrated in Appendix E.

Limited by containing physical voltage-controlled oscillators, many early analogue synthesisers were *monophonic*, capable of producing only a single simultaneous note, or *voice*. In order to allow *polyphonic* playback, a synthesiser must clone itself internally in order to generate additional voices. This process is considerably less expensive on a digital synthesiser, and is perhaps the most significant advantage that they offer over their analogue predecessors. We certainly intend to make HotPants a polyphonic, as the range of a monophonic instrument is severely limited.

Wavetable oscillator

Whilst it is possible to calculate an oscillator's signal in real-time, this is inefficient if we wish to produce only simple periodic waveforms. For the purposes of optimised performance, it is preferable to calculate as much as possible in advance, outside of the real-time processing loop. For oscillators, this can be achieved with the use of a table stored in memory, trading storage space for processor time. An oscillator using this technique is known as a *table lookup oscillator*, or *wavetable oscillator*. A single cycle of the desired waveform can be calculated in advance and stored in this table, which can be referenced and looped during playback to produce the desired repeating cycle. In fact, there is no requirement that this stored waveform be restricted to the common sine, triangle, sawtooth and pulse types. Arbitrary waveforms can be created in advance by a developer, even drawn, and stored to disc. With this technique there is a lot of potential for interesting sounds to be generated at the oscillator level, and many instruments make use of it[14].

Equations for indexing the table are straightforward [16, pg. 93]:

$$increment = \frac{L \times frequency}{sampling\_frequency} \tag{1.1}$$

[12] See Section 1.2.1

[13] An oscillator generating a random pattern of samples

[14] See ES2, Albino and Massive, Section 1.5

$$phase\_index = mod_L(previous\_phase + increment) \tag{1.2}$$

$$output = amplitude \times wavetable[phase\_index] \tag{1.3}$$

Where L represents the length of the table and $frequency$ represents the desired note frequency. The table is looped by modulo its length in Equation 1.2 and the resulting amplitude value is scaled with the desired output ratio *amplitude* in Equation 1.3. Rearrangement of Equation 1.1 also illustrates how the table increment value affects the output frequency:

$$frequency = \frac{increment \times sampling\_frequency}{L} \tag{1.4}$$

Reading the table directly, at an increment of 1, will produce a note at the original frequency. Other rates will produce higher or lower note pitches: incrementing through the table at double the rate produces a note of double the frequency, scanning through at half the rate produces a note of half the frequency [10, pg. 81]. In terms of audible frequencies, both cases produce a jump of a musical *octave*. In order to produce the additional eleven musical notes that lie between the octaves it is necessary to index the table at non-integer positions as well, between that of two stored values. Without an effective strategy, error can here result in *table look-up noise* [16, pg. 93]. In this case, there are a number of techniques that can be applied:

1. *Truncation*—the fractional part of an index is discarded
2. *Rounding* the index up or down to the nearest integer value and reading at that position
3. *Interpolating* between adjacent entries in the wavetable to calculate a amplitude value that corresponds to the fractional phase index increment [16, pg. 94]
4. Increased *Table length*—a longer table is more accurate

Interpolation would seem to be the most *correct* method for solving this problem. However, Mitchell [10, pg. 87] shows that in practical applications a long wavetable of over 16K with round-off of the index is nearly equivalent. Generally speaking, most aspects of instrument design involve a balancing of audio fidelity concerns with real-time instrument performance. In this case, if it can be shown that a better performing algorithm has equal or comparable sound quality in comparison to that of a more *accurate* technique, then it is likely that the former will be chosen. This case illustrates that point.

Overall, wavetable oscillators are a very efficient and straight-forward signal generation technique, offering considerable performance benefits with little or no sacrifices in terms of instrument capabilities and audio quality. We intend to use them extensively in HotPants, as well as experimenting with hand-crafted non-generic waveforms.

Low-frequency oscillator

We have been discussing oscillators as generators for signals in the audible range. However, *low-frequency oscillators* (LFOs), those with a frequency below 20Hz, also have uses as sources for *modulations*—varying the signal in some way as a function of time. Modulations are discussed in Section 1.4.3.

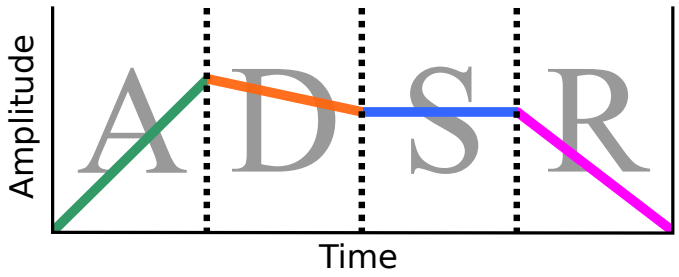


Figure 1.2: Attack, decay, sustain and release (ADSR) envelope

Envelope generator

Having generated a raw signal, the next step towards producing a useful instrument is to exert a degree of control over oscillator output. At the very least, we need to be able to activate and deactivate the signal—known as *gating*—although in order to produce a note that is in any way *expressive*, a degree more of complexity is involved. This is usually done by creating a simple, staged model for the amplitude dynamics of a musical note, forming an *envelope* over the raw waveform:

1. *Attack*—the duration taken for the the note to rise from zero amplitude to its maximum value
2. *Decay*—the duration taken for the note's amplitude to drop from its peak to its *sustain* level
3. Sustain—the amplitude that the note should hold until *released*
4. Release—the duration taken for the note to return to zero from its sustain level when the note is *released*[15]

This common configuration is known as ADSR[16] and is illustrated in Figure 1.2. The settings for these stages are flexible depending upon the effect desired, and some stages may not be used at all—set to zero: a drum sound would have a very short attack and little or no sustain or release; a violin envelope, on the other hand requires all four stages[17].

In any case, this four stage pattern is effective for producing a reasonably wide range of instrument dynamics and is still in common use, along with derivatives such as attack, decay, sustain, *hold*[18], release (ADSHR), AHDSR, AHD, etc. With the emergence of software instruments there can be much more flexibility. Indeed, there need be no limit to the variety of shapes supported, other than the limitations of user interface. The sustain stage, in particular, might contain any number of looping and swooping segments[19]. Mitchell [10, pg. 58] notes that efficient computer graphics line drawing algorithms such as the *Bresenham midpoint algorithm* are also useful for generating amplitude curves in real-time,

[15] It should be noted that the use of the term *release* originated in the analogue synthesiser era. This stage would be triggered when the performer *released* their finger from the keyboard [10, pg. 43]. A percussion instrument, for instance, does not have a *release* of the kind implied here, but the actual sound may have a shape that is still well-served by this model. Keyboard input is still the most common human interface device for electronic music since it is well suited to the task—a keyboard is a row of switches

[16] Attack, Decay, Sustain and Release

[17] An analysis of the envelopes of some common instruments can be found in [2, pg. 80/1]

[18] Limiting the sustain period to a set duration

[19] For a good example of an instrument based on this flexibility, see Absynth (Section 1.5)

an added advantage being that the same algorithms can be reused to present the envelope visually in a graphical user interface.

Whilst primarily used to shape the amplitude of the instrument, envelopes can also be used as modulation sources[20]. We will certainly require the use of envelopes for the HotPants instrument, as both amplitude controllers and modulation sources.

Digital filters

Now that we can generate a signal, and manipulate its amplitude, we turn our attention to how we might shape the audio spectrum in useful ways. Filters are integral to subtractive synthesis[21], but useful applications for them exist in all forms of synthesis. Filter types to implement would include low-pass, high-pass, band-pass, notch, peak, comb and all-pass. EQ types such as high and low shelves may also be useful. Most of all, the subjective quality of the filter's effect on the signal is key in it serving its role in a successful instrument.

1.4.2 Synthesiser techniques

Having detailed the key components of an instrument, we can now look at how these building blocks can be applied to achieve common synthesis techniques. Some have already been mentioned. Each have their particular strengths and weaknesses, and suitable applications and are are candidates for inclusion in the feature set of the HotPants instrument. To follow is a brief summary of each, in a rough order of increasing complexity.

Additive synthesis

Additive synthesis has the longest history of all synthesis techniques, first being used in church pipe organs [16, pg. 134], dating back to the fourteenth century and older. This technique is the most direct audio application of the Fourier series; mixing a large number of simple waveforms, usually sine waves, in order to create a complex spectrum. The major drawback to this approach is that the mathematics of the Fourier series require an infinite number of oscillators, impossible in practice. However, by use of additional modulation techniques, such as *frequency modulation* and *amplitude modulation*[22], it is possible to mitigate this limitation and achieve more satisfactory results with a small number of oscillators.

Subtractive synthesis

The basis for the popular analogue synthesisers of the 1960s and 70s, subtractive synthesis works by generating a rich signal from a small number of oscillators, typically one to three, and attenuating, *subtracting*, parts of the frequency spectrum with one or more filters [16, pg. 184–197]. This method is simple and allows for quick and easy programming of instrument patches. It is, however, not particularly effective at producing realistic instrument sounds and is otherwise rather limited in the range of sounds it can produce. Despite this, its accessibility means that it is still a very popular approach to synthesis.

[20] See Section 1.4.3
[21] See Section 1.4.2
[22] See Section 1.4.3

Frequency modulation synthesis

Frequency modulation, or *FM synthesis* is a family of techniques which achieve many of the theoretical possibilities of additive synthesis with a greatly reduced number of signal generators. In its simplest form, *simple FM* or *Chowning FM*[23], the frequency of one oscillator (the *carrier*) is modulated in the audible frequency range by another, the *modulator*. The equation for generating an FM signal is [5, pg. 527]:

$$e = Asin(\alpha t + lsin\beta t) \tag{1.5}$$

where

e = amplitude of the modulated carrier

α = carrier frequency in rad/s

β = modulator frequency in rad/s

l = d/m = the modulation index, ratio of peak deviation to the modulating frequency

A series of *sidebands* are generated, which spread out from the carrier frequency, creating a harmonically rich signal with just two oscillators. FM synthesis was one of the most popular techniques in the early digital era, most notably with Yamaha's extremely popular DX7 model [16, pg. 226], the first commercial all-digital synthesiser. This unit made use of six oscillators [6] and was capable of producing and extremely wide range of sounds. Use of FM would certainly increase the range of HotPants's capabilities. However, use of a large number of oscillators would likely lead the instrument in the direction of being dedicated FM. As such, it may be more attractive to take the approach of incorporating a Chowning FM oscillator pair instead[24].

Physical modelling and waveguide synthesis

By creating a mathematical model, *physical modelling* synthesis (PhM) seeks to recreate the mechanical and acoustic behaviour of an instrument [4, pg. 30]. Although many PhM applications are mathematically intensive, the *Karplus-Strong string synthesis algorithm* [8] is one that is simple enough to serve well for demonstration purposes. It functions by modifying an initially rich waveform stored in a memory wavetable so that high-frequency note harmonics decay over time towards a pure sine wave. This mirrors the physical effect of a stringed instrument such as a guitar and sounds pleasingly authentic, especially considering its simplicity. Strong's basic algorithm averages two successive samples from the stored wavetable:

$$Y_t = \frac{1}{2}(Y_{t-p} + Y_{t-p-1}) \tag{1.6}$$

Where Y_t is the value of the t^{th} sample, and p is the length of the wavetable, or *periodic parameter*. This technique is likened to a delay line without an input. Adaptations of this basic algorithm introduce randomisation that produces improved simulations of plucked strings, as well as drum timbres [8, pg. 44, pg. 46].

[23] Named after John Chowning, who first explored use of FM for musical applications [5]

[24] This approach is used by the ES2: See Section 1.5

Smith [18] generalises Morse's wave equation [14] for a vibrating string, creating a *traveling wave* solution and the modelling framework known as *digital waveguiding*. Again, this approach has the advantage of being relatively computationally efficient although the latest innovations may not be immediately viable on modest home equipment. Overall, these techniques provide any extremely large scope for future innovation and commercial implementations are already available[25]. We can certainly expect this research to form the basis for the next generation of software instruments.

1.4.3 Modulation

The previous sections have discussed core synthesiser technologies. Next, we turn our attention to modulations, manipulating settings of the synthesiser as a functional of time, in order to create evolving sounds. We will see that many of these common effects are closely related, all being time-delay effects [16, pg. 46], suggesting it may be viable to include a large number of them within HotPants.

LFO and envelope modulations

As noted in Section 1.4.1, low frequency oscillators can be used as internal modulation sources for an instrument. Typical targets for this kind of slow, periodic modulation include overall amplitude (tremolo), pitch (vibrato) and filter settings, such as the cut-off frequency. Polyphonic synthesisers commonly support the use of both monophonic and polyphonic LFOs; in polyphonic mode, each of the synthesiser's voices is modulated by its own LFO instance whereas in monophonic mode all voices are modulated as a group.

Additional envelope generators can also be used to produce modulations that vary in relation to the stages of the musical note. This is especially useful for varying the frequency content of a note as a function of time; many instruments contain a much larger amount of high-frequency content in their attack stage [11, 12, 13] and a short attack-decay filter envelope with can be used to mimic this. Furthermore, by use of the kind of complex, evolving envelopes discussed in Section 1.4.1, complex modulations can be created that are more interesting and organic than static LFO modulations.

Amplitude and ring modulation

Amplitude modulation, or *AM*, is one of the oldest modulation techniques [3, pg. 19], and is very similar to *ring modulation*. As with FM, both techniques make use of a carrier and modulator pair, with the difference being that in ring modulation (RM) two bipolar signals are modulated, whereas AM modulates a bipolar signal with a unipolar signal [16, pg. 216]. In either case, the result is a simple multiplication of the two signals:

$$modulation_t = C_t \times M_t \tag{1.7}$$

Where C represents the carrier, and M the modulator. At very low frequencies this creates a *tremolo* effect. However, when the frequency of M enters the audible range, a series of *sidebands* are created around the carrier. The difference is that in RM the carrier frequency is not present and a frequency shifting effect can be achieved if desired. Both techniques create metallic sounds that were very popular in twentieth-century analogue synthesisers. Since their operation is simple, they are definite candidates for inclusion.

[25]The Sculpture instrument examined in Section 1.5 makes use of the plucked-string algorithm

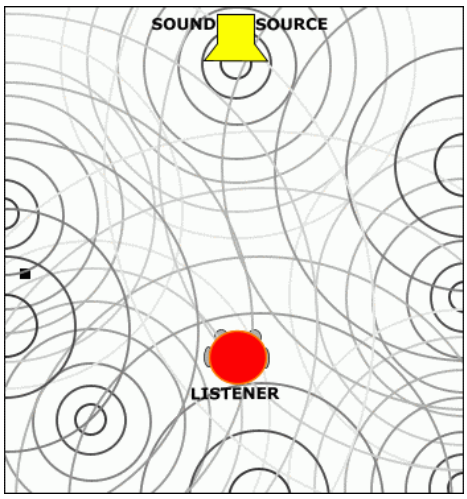


Figure 1.3: Reverberation and delay within a space

1.4.4 Audio effects

Finally, we intend to implement a small number of on-board line audio effects in the instrument. These are usually considered external to the core of a synthesiser, but which nevertheless can be packaged as part of an instrument in order to add to its functionality. They should be a good opportunity for us to broaden the range of the instrument, and our implementation.

Delay and reverberation

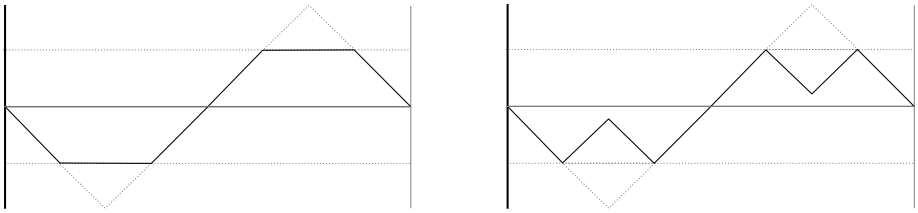
Delay has already been mentioned in the role it plays in digital filters, and it also forms the basis for a number of other common effects. In its most obvious application, delay simulates the effect of echo, sound waves bouncing from the surfaces of a physical space and reaching the listener *delayed* by their extended journey, illustrated in Figure 1.3[26]. Delays can be achieved in software by use of a queue in memory, storing a duration of the input signal and outputting the cycling contents of the buffer, mixed with the continuing input signal. The output of the buffer can optionally be filtered and *fed back* into the input in order to create a decaying sound, similar to echo.

Reverberation, or *reverb* is a similar technique. It differs from delay in that it creates a spatial ambience by use of a very large number of separate short delays, clustered around the input signal. Thus, in reverb, the original sound is not perceived to be replayed as a discernible echo [1, pg. 77-9]. Reverb and delay are typically used together in order to recreate the complex effects of a signal reflecting around a space, and the implementation of at least one of these will do a lot to improve the depth of the instrument's sound.

Digital distortion

Distortion effects drive a signal against a fixed limit, intended to harden the synthesiser timbre, giving it a more aggressive quality, compressing the dynamic range of the signal and boosting perceived volume. There are many techniques available, the most popular of

[26] `http://manual.audacityteam.org/images/9/9f/Reverb_rev1.png`. Audacity User's Manual. Creative Commons Attribution 3.0 license



(a) Hard clip (b) Fold-over

Figure 1.4: Digital distortion types

which model the response of an analogue vacuum tube or transistor amplifier, producing tones similar to guitar amplifier distortions. Digital types, however, are much simpler to produce and also have their uses. Figure 1.4 demonstrates the effect of two simple algorithms, *hard-clipping* and *fold-back*, applied to a triangle wave.

1.4.5 User Interface

We have spent the previous sections detailing the internal components of synthesisers and here we would like to briefly discuss instrument user interfaces, an important area when it comes to software instrument design. As instruments are creative tools, it is essential that our design does not hamper the musician. Ideally, the design should encourage creativity, and there are contemporary designs which could be said to achieve this[27]. Most software instruments take the approach of presenting a user interface which mimics a hardware device: knobs, switches and faders. In this way we can see how the early generations of hardware synthesisers continue to influence software instruments, and these designs are popular due to their simplicity and familiarity.

Manipulating a graphical representation of a synthesiser on-screen using a computer mouse is far from ideal and more ergonomic interface designs are certainly sought after. However, our focus is DSP implementation, not external interface. As will be shown in Section 2.5.3, we will be able to implement an instrument with a purely functional user interface. This basic GUI could then be replaced at a later date with a properly designed interface panel. A truly interesting and effective design would warrant its own project.

1.5 Market analysis

Now that we have introduced a range of common technologies we would like to present examples of their application in complete software synthesiser instruments. We have selected eight popular commercially available instruments, and descriptions for each follow on the next pages. User interface screenshots are provided in Appendix D in order to illustrate the kinds of common user interface patterns discussed in Section 1.4.5. We will use this analysis in Chapter 2 in order to guide the process of deriving a suitable feature set for the HotPants instrument.

[27] See Synplant, Figure D.1h

ES2 (Emagic / Apple) — Figure D.1a *May 2001* [28]

- Three oscillators with the traditional sine, sawtooth, triangle, pulse and noise waveforms as well as 99 *digiwave* wave shapes
- Primarily subtractive synthesiser design
- Chowning FM. Filter cut-off can also be frequency modulated
- Some degree of analogue modelling: controllable detuning of oscillators to emulate unstable analogue circuits
- Reasonably complex matrix-controlled modulation section
- Extremely processor efficient

The ES2 appears to have been designed to combine the best of both the analogue and digital worlds into a flexible and all-round synthesiser. On its introduction, this was not an especially groundbreaking instrument, but remains popular by combining a wide-range of features with ease of use and low power requirements. It is also considered to sound pleasingly *fat* and is suitable for a wide range of applications.

Albino (Rob Papen / Linplug) — Figure D.1b *Dec 2002 (v1). Jun 2006 (v3)* [29]

- Four selectable filters models, each having a unique sound
- Modulation matrix supporting more flexible settings than the ES2, including modulations of modulations
- Separate amplitude envelopes for all four oscillators, including tempo-synchronisation
- Separate envelopes for both filters
- Layering: up to four instances of the instrument can be stacked internally

Albino is comparable to the ES2, with a similarly analogue sound, but with vastly better flexibility thanks to its excellent modulation matrix, not to mention the possibility of layering whole synthesiser patches on top of one another.

FM8 (Native Instruments) — Figure D.1c *Dec 2006* [30]

- Extremely advanced software FM synthesis implementation
- Six FM operators that can be freely combined in a matrix
- Each operator has a dedicated envelope, with looping features
- Equally complex and flexible modulation matrix
- Twelve built in effects that can be combined in an internal rack
- Vector matrix for morphing internally between four instrument patches

The FM8 is the successor to the FM7, an emulation of the Yamaha DX7 synthesiser[31], but takes this foundation and builds on it a horde of new features which make it an enormously powerful and complex instrument with a vast sonic palette.

[28] `http://documentation.apple.com/en/logicstudio/instruments/#chapter=5`
[29] `http://www.robpapen.com/albino-3.html`
[30] `http://www.native-instruments.com/en/products/producer/fm8/`
[31] Section 1.4.2

Absynth (Native Instruments) — Figure D.1g *Oct 2000 (v1). Sep 2009 (v5)* [32]

- Various selectable oscillator types: single and double wavetables as well as FM, ring modulation, fractal and granular
- Basic sampling supported including granular [16, pg. 168–184] processing
- Extremely flexible break point envelope editing with multiple loop points
- Unusual effects such as pipe, multi-comb filter and resonators

Absynth is quite unique. Its feature set is an attempt to create an *organic* instrument unlike other offerings. Whilst it can produce a wide range of sounds, it is best suited to create evolving moods, textures and atmospherics in particular thanks to its support for long and flexible envelopes. Its internal effects—pipes, resonators and echoes—are also geared towards modelling real-world physical sound dynamics.

Sculpture (Apple) — Figure D.1f *Jan 2004* [33]

- Physical modelling synthesis based on a single vibrating string
- Three exciters with varied modes (strike, pick, bow, noise, blow, etc)
- Two movable pickups
- Surface model with morph-able materials (nylon, steel, glass and wood)
- Instrument body-modelling equaliser
- Modulation matrix particularly suited to physical modelling: randomisation, vibrato, jitter
- Recordable two-dimensional morph pad

Sculpture is a very powerful and accessible PhM instrument, an implementation of the Karplus-Strong string algorithm [8]. It is a excellent example of a consumer-focussed physical modelling implementation. It does, however, require a fair amount of processing power which was an issue on its initial release, but is now fairly well catered for by modern machines.

Massive (Native Instruments) — Figure D.1e *Sep 2006* [34]

- Large selection of selectable oscillator waveforms
- Waveforms can be *morphed* in different modes: spectrum, bending and formant modulation
- Very accessible modulation: nearly every synthesiser module has modulation inputs
- Twelve filter modes, including a number of different filter models
- Performance and step LFO modes which allow user-programmable patterns
- A number of built-in audio effects

Massive is an enormously popular instrument and has had a huge impact since its release a few years ago. In particular, some of its more *brash*-sounding features have become

[32] `http://www.native-instruments.com/en/products/producer/absynth-5/`
[33] `http://documentation.apple.com/en/logicstudio/instruments/#chapter=14`
[34] `http://www.native-instruments.com/en/products/producer/massive/`

commonly-heard, such as high-pitched *formant*[35] modulations of its oscillators. Whilst its capabilities are wide, it is this kind of very assertive sound that it has become known for.

MonoPoly (Korg) — Figure D.1d *2004. Original hardware, 1980.* [36]

- Very authentic software modelling of a popular analogue-era hardware synthesiser
- Adds to the original's feature set with conveniences such as preset loading and saving, preset templates and digital effects not included on the original unit.

We have included this instrument as an example of a software modelling of a classic hardware unit. It is acclaimed for sounding very faithful to the original, as well as introducing the benefits of a software instrument such as unlimited polyphony and patch storage.

Synplant (Sonic Charge) — Figure D.1h *Nov 2008* [37]

- Unique, organic approach to user interface and sound design
- Sounds are grown from seeds, which can subsequently be planted to form the basis of further sound generations
- Elements of the sound can be edited by use of a *DNA*-inspired interface

Synplant is an example of a more *evolutionary* instrument concept. Whilst it does not offer any thing especially unique in terms of its internal sound generation ability, it aims to innovate in its unique genetic approach to sound design. As such, it represents an excellent example of a user interface design with creative practicality.

1.6 Conclusion

In this chapter we have introduced the reader to the background for the HotPants project, its historical context and supporting technology. With our selected eight commercial instruments, we have shown the kinds of features common to popular instruments currently found in the marketplace. We will use this analysis in the following chapter, as well as that of the underlying technology detailed in Sections 1.2–1.4, to derive a feature set for the HotPants instrument that meets our requirements for functionality and quality.

[35] Peaks in the frequency spectrum produced by the human human vocal chords [2, pg. 128]

[36] `http://www.korg.co.uk/products/software_controllers/legacyanalog/sc_legacyanalog2007.asp#Monopoly`

[37] `http://www.soniccharge.com/synplant`

Chapter 2

Design

2.1 Introduction

Having discussed software synthesiser technology in Sections 1.2–1.4, and introduced some contemporary instruments in Section 1.5, in the following Sections 2.2 and 2.3 we will turn to the design and feature set that we plan to implement in the HotPants instrument. Following this, we will detail management, implementation and testing details in Sections 2.4–2.6.

2.2 Requirements analysis

To reiterate our stated goal:

> *To design and implement in software an expressive and professional-sounding digital music synthesiser, competitive with comparable commercial offerings, and suitable for a wide range of offline and real-time applications on standard, home-level computer equipment.*

Therefore we present the following requirements for the HotPants instrument:

1. To have a wide feature set; capable of producing a reasonably wide range of sounds
2. To have excellent sound quality, comparable with similar commercial offerings
3. To have excellent computational performance; capable of operating with multiple concurrent instantiations of the instrument plug-in on only modestly powerful computer equipment in real time
4. To have expressive, performance-orientated features and otherwise be a desirable instrument for musicians to use

2.3 Feature set

Due to the modular nature of a synthesiser, the feature set for HotPants is potentially extremely wide and flexible; it is entirely possible to implement the full range of technologies

discussed in Section 1.4 in a single instrument, and do so effectively. However, in order to give this project a more focussed and coherent direction, it is necessary to analyse a narrower design.

In Section 1.2.5 we discussed physical modelling synthesis, currently the cutting-edge of sound synthesis. Whilst this field is exciting and at the forefront of sound design technology, a PhM-based instrument would require an investment into DSP research that is outside of the scope of this project, one which is intended to be software implementation focused. There are also a number of other areas in which a PhM instrument would fail to meet our requirements: PhM instruments tend to be rather specialised—there is no such thing as a general-purpose PhM algorithm and we would like HotPants to be "capable of producing a ... wide range of sounds". With a physical modelling synthesiser we might also fail to satisfy our requirement for an instrument that has real-time performance on modest hardware. On this basis, we assess PhM as unsuitable for HotPants.

For this project, it is more appropriate to design an instrument with a conventional feature set. Technology such as subtractive and FM synthesis are well-established and documented, and well within scope. In light of our market analysis in Section 1.5, project duration and our existing skills, we have assessed the following outline for the instrument:

- A polyphonic, primarily subtractive synthesiser—additional synthesis techniques can feasibly be added later, especially Chowning FM.
- Contain a small number of oscillators—our market analysis reveals that three is a common number and would be feasible to support.
- Contain a small number of filters—the quantity and types of filters offered by our selected instruments vary. A minimum of two filters is common although the types offered differ considerably. At the very least we would like to implement LP, HP and BP filters. More exotic types such as notch, peak and comb are desirable but less necessary, although we would like to incorporate them if possible in the later stages.
- Provide a small number of envelopes—this is one area in which we must be conservative: elaborate envelopes such as those discussed in Section 1.4.1 are not well served by the kind of generic user interface view that we plan to use for HotPants[1]. Instead, it would be better to implement conventional staged envelopes, such as the ADSR, since these can be operated easily in a simple user interface with knobs and faders.
- Support basic modulations—we are developing oscillators and envelopes so it will be a simple matter to reuse these modules as modulation sources[2]. This is another area where our minimal interface may restrict us—effective modulation programming relies upon an intuitive user interface. We may be able to take inspiration from the simple but effective table modulation matrix used in Albino[3]. In any case, we must take care to implement a modulation system that is not overly ambitious.
- Feature a small number of built-in audio effects—as demonstrated in Section 1.4.4, many audio effects are derived from the same unit generators, especially filters. As such, we would like to implement some of these, especially chorus, flanging and phaser. More complex effects such as reverb may prove to be out of scope.
- On top of those features listed above, there are many additional features that we

[1] See Section 2.5.3
[2] See Section 1.4.3
[3] Figure D.1b

might look to include during the development process. The ES2[4], in particular, has a number of tertiary features such as oscillator and filter reset, as well as analogue-style oscillator randomisation that we might wish to include.

Overall, this feature set closest mimics either the ES2 or Albino from our market analysis in Section 1.5. It is important to note that we would like the development of HotPants to be an exploratory process and would like to leave plenty of scope for flexibility. Some features may prove to be challenging, others implemented with ease. Some may prove to be more successful than others. We can use the modular nature of the instrument to our advantage and it will not be necessary to finalise this feature set until the project draws to a close.

2.4 Project management

2.4.1 Risk

In terms of the HotPants project, the greatest risk can be defined as not completing a functioning instrument. This risk is minimal, since a very simple instrument could be considered *functional*:

- One oscillator
- One amplitude envelope[5]

This would be a complete instrument, albeit one with little merit and would certainly fail to satisfy our requirements. It does, however, illustrate that there is a certain hierarchy of feature importance:

1. Oscillators—we must have signal sources
2. Amplitude envelopes—at the very least we must be able to switch the signal on and off
3. Filters—essential for a useful subtractive synthesiser
4. Modulators
5. Effects and performance features

This gives us some idea as to which features we must prioritise. In order to give this project direction, we should take note of this hierarchy, aim to complete a minimally functional instrument as soon as possible and then build upon this foundation to take us towards satisfying our requirements. In order to achieve this, we aim to employ an iterative development model.

2.4.2 Equipment

An additional area of concern is that of risk to our development equipment. Since an audio device requires the use of speakers, some amount of caution is required in order to avoid programming errors that could lead to distorted or maximised signals, and equipment

[4]See Section 1.5

[5]Which could be as little as an open / close gate

damage. By extension, there is also the health and safety risk of hearing damage. In order to avoid this, we intend to develop experimental features away from the audio framework that we will use for integrating the instrument[6]. Before testing the instrument with real audio output we can make use of spectrum analysers to inspect the output signal. Finally, we will integration test the instrument with our audio monitoring system. Use of this process should allow us to avoid any of these risks.

2.4.3 Version control

A version control system will be employed to manage source code-level risk. There are a variety of systems available:

- CVS
- Subversion (SVN)
- Git
- Perforce

All have their merits and the choice is mostly a matter of taste and available support. For Hotpants, we have chosen to use SVN, due to it being supported by all of our chosen development tools as well as being a popular, modern and fully-featured system that is suitable for our needs. SVN is thought by many to be superior to CVS. Git offers a feature set that exceeds that which is required for this project and Perforce is a proprietary system. The project SVN repository will also be set up on a remote server, to minimise risk of source code data loss.

2.5 Implementation Details

We will now discuss the tools and work-flow that we intend to use for the duration of the HotPants implementation.

2.5.1 Language

Our primary implementation language will be C^{++} [21]. It is a standard, well-supported language, extremely powerful, fast and common for this kind of low-level audio work. Features such as its support for template base classes will allow us to define abstract classes for reusable components of the instrument, such as the delay module or oscillators [10, pg. 120]. The modularity of a synthesiser clearly lends itself to an object-oriented approach and C^{++} code is portable, which is desirable and may form an additional implementation task that we may undertake, discussed in the next section.

2.5.2 Environment

An instrument requires some form of input system for musical notes, as well as an audio output stream. As detailed in Section 1.2.1, some implementations make use of a scripting language for note input, and this is still common, being the input method used by Mitchell [10, chapter 24–25]. Output could be in the form of a standard audio file format such as *Wave* or *AIFF*.

[6] See Section 2.5.3

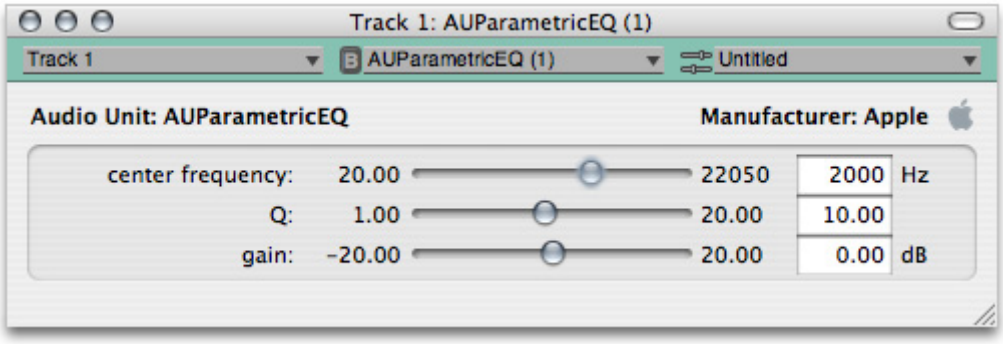


Figure 2.1: AU effect plug-in with generic view

However, another approach is to implement the instrument as a software *plug-in*, interfacing with some existing audio framework. This is the approach taken by most commercial instruments, such as those detailed in Section 1.2.1. Popular formats include *Virtual Studio Technology* (VST), developed by Steinberg[7]; *Real Time AudioSuite* (RTAS), by Digidesigns[8]; and *Audio Units* (AU) by Apple Computer[9]. Input of musical notes is taken care of by the host system's *Musical Instrument Digital Interface* (MIDI) facilities[10], via a device such as a MIDI controller keyboard. Audio output is also provided by the operating system. This approach is ideal for our needs and removes the necessity to implement these supporting systems.

Due to our own preferences, and access to equipment, HotPants will be developed primarily as a plugin for the Apple OS X *CoreAudio* framework in the Audio Unit format introduced above. CoreAudio is a modern and well-supported framework, implemented as a C^{++} class hierarchy. A number of examples and tutorials for the framework are provided online, and at the time of writing we have studied and completed the *Tremolo unit*[11] and *Sinsynth*[12] examples, assuring us that mastery of the framework is not beyond the scope of this project.

As mentioned in Section 1.4.5, we do not intend to develop a full graphical user interface for HotPants. Fortunately, CoreAudio provides a *generic view* for AU instruments that do not define a graphical view, automatically generating basic visual controls that are appropriate for the internal parameters of the instrument. Figure 2.1 illustrates an AU generic view for a parametric equaliser effect plug-in with three internal parameters. Our instrument will no doubt include a great deal more parameters, although the concept will be the same.

2.5.3 Development Tools

Since CoreAudio is a platform-dependent framework, the most appropriate development tool will be the *Xcode* integrated development environment[13], available for free from Ap-

[7] `http://www.steinberg.net/en/company/3rd_party_developer.html`
[8] `http://www.digidesign.com/`
[9] `http://developer.apple.com/technologies/mac/audio-and-video.html`
[10] `http://www.midi.org/techspecs/midispec.php`
[11] `http://developer.apple.com/mac/library/samplecode/TremoloUnit/Introduction/Intro.html`
[12] `http://developer.apple.com/mac/library/samplecode/SinSynth/index.html`
[13] `http://developer.apple.com/tools/xcode/`

ple and purpose-built for developing Macintosh OS X software and plugins. Xcode also supports Subversion, our chosen version control system.

In addition, *Eclipse*[14], a popular and free IDE, will be used for separate class implementations and experimentation with raw data. Eclipse is similarly well supported, with a wide range of available plug-ins as well as C^{++} support. As mentioned in Section 2.4.2, the use of a separate IDE for some stages of development is advantageous. This also allows us to very clearly separate unit development—in Eclipse, and integration—through Xcode.

2.6 Testing

Due to our use of an iterative implementation model we will not have the clear delineation between implementation, unit testing, integration and testing typical of the traditional *waterfall* model. Instead, testing will be a stage in our development workflow, moving between unit development in Eclipse, and module integration in Xcode.

Unit testing

As outlined in Section 2.4.2, unit testing will take place in Eclipse, with inspection of data in the system console as best appropriate for the type of module being developed. This will introduce a certain amount of discipline to our workflow, as well as encouraging us to perfect the performance of our code, essential if we are to satisfy our performance-related requirements. Eclipse is also the appropriate place for the creation of appropriate test cases. Since HotPants modules are discrete, we do not anticipate a great need for regression testing.

Integration testing

Integration testing will take place in Xcode, following a reasonable completion and unit testing of each new module in Eclipse. Our integration focus will be our functional requirements, such as audio quality, as well as performance. We must also take care to avoid potentially dangerous glitches and errors, as noted in Section 2.4.2.

The *AU Lab* application provided as part of the CoreAudio Software Development Kit is a a small and functional tool for testing AU plug-ins during development. We intend to use this application for the majority of the implementation process. As the instrument approaches completion it will be tested in a fully-featured digital audio workstation application, for which we will use the *Logic* package[15], with which we are familiar. This application will also be used to prepare our end of project demonstration piece, with its full provision of multi-tracking and orchestration functions.

2.7 Evaluation criteria

Now that we have fully detailed our implementation plan, we turn our attention to how we feel the finished HotPants instrument might be evaluated. We propose the following procedures and criteria for assessing the success of the completed instrument in light of the requirements stated in Section 2.2.

[14] `http://www.eclipse.org/`
[15] `http://www.apple.com/logicstudio/`

Demonstration

A formal presentation of the instrument will be prepared. The capabilities of the instrument will be demonstrated by the playing of a piece of sequenced music, ideally one that is well-known. There are a large number of pre-prepared MIDI sequences available online and we intend to use one such as this as the basis for our demonstration. We will use the Logic digital-audio workstation application to prepare, orchestrate and mix the piece. We hope that this will allow us to demonstrate the degree to which we have succeeded in meeting our requirements:

- A piece will be chosen in light of the instrument's final feature set in order to best demonstrates its capabilities; we should try to use as many of its in-built facilities as possible. For example, if HotPants is capable of producing percussion sounds then we will incorporate a rhythm section. If the instrument corporates audio effects then we will attempt to include these. We will not make use of external instruments and effects: the entire performance will consist of multiple instantiations of HotPants alone. This will illustrate requirement 1—the breadth of the instrument's feature set.
- In order that comparisons can be drawn with commercial instrument implementations, we shall also prepare arrangements of the same piece using one or more of the instruments highlighted in Section 1.5. Since we imagine that HotPants will have a smaller feature set, it should not be difficult to recreate similar settings. In this way, sound quality can be assessed—requirement 2.
- The demonstration can be arranged in advance, but performed with real-time processing. This will allow us to demonstrate requirement 3, the computational performance of the instrument. The degree to which we are successful in optimising the instrument will also affect the preparation of the piece: poor instrument performance will limit our scope. To simplify the demonstration, our third-party instruments could be recorded to an audio file, unless we feel that it is useful to draw a comparison with HotPants. As expected, the test system for the performance will be a modern laptop computer, but not one that is especially powerful, so as to satisfy requirement 3.

2.8 Conclusion

We have detailed the background and context for this project and assessed a design based on an analysis of existing technology and available instruments. Implementation details have been determined in light of our requirements and risk has been assessed and appropriately provisioned for. In the following chapter, we will review the completed implementation of the project, and assess the degree to which we have been successful in achieving our goals.

Chapter 3

Implementation

3.1 Introduction

In this chapter we document the implementation stage of the HotPants project. We begin by documenting the architecture of the system in Section 3.2, before moving on to discuss topics relevant to the work involved in implementing this design in Section 3.3, such as the kinds of challenges that needed to be overcome, programming techniques and testing strategy. Finally, in Section 3.4 we discuss how we have implemented the instrument as a CoreAudio framework plug-in, and how it might be ported to other platforms.

DSP

Before we begin it is important to note that, as anticipated, novel signal processing has not at all formed a part of this project. Most significantly, the implementation of the instrument's filters, a key component in what is primarily a subtractive synthesiser, is not original work. This is noted fully in Section 3.2.6. We do, however, employ the Fourier series to generate our complex waveforms with additive synthesis, based on standard equations provided in Hartmann [7]. As such, this project should be viewed in light of it being primarily an application of software engineering design and implementation techniques.

3.2 Architecture overview

So as to introduce the reader to the instrument itself, we present a largely top-down view of its key modules. A detailed guide to the directory structure and contents of the source code files themselves can be found in Appendix A, with a system diagram in Appendix B.

Note that for clarity, function arguments have been omitted when not strictly necessary in the following discussions. At other times, our instrument types are used to communicate the role of notable arguments. Also, some local variables have been shortened from those in the original source code.

3.2.1 Instrument core

First of all, we discuss four classes which make up the core of the instrument. In any given instantiation of the synthesiser, only one of each of these modules will be created and they

manage the key functionality of the instrument, its voices and parameters.

HotPantsCore

The instrument core class is `HotPantsCore`. It is intended to fully encapsulate the public interface of the instrument, providing all input and output facilities necessary to integrate with any kind of MIDI-based audio framework[1]. Figure 3.1 illustrates the module connections inside the object which will be detailed in the following pages. Discussion of the core introduces our use of two-part initialisation; in order to allow light-weight construction, as well as changes to the sampling rate throughout the life of the instrument, the following pattern is common throughout much of the instrument:

```
<Component> c;                  // eg. HotPantsCore
SamplingRate r = 44100.0;       // CD-quality
c.initialise(r);
```

For `HotPantsCore`, note and render operations prior to first initialisation are rejected. However, the sampling rate can be re-initialised at any time

```
HotPantsCore c;
c.startNote(1, 64, 64);         // returns FALSE
c.initialise(44100.0);
c.startNote(1, 64, 64);         // returns TRUE
c.initialise(22050.0);
c.startNote(1, 50, 127);        // returns TRUE
```

Re-initialisation causes all active voices to be killed immediately and the full system to be reset for the new sampling rate. This is desirable behaviour, since modification of the sampling rate is an infrequent operation and by definition is not compatible with real-time operation. For internal components, not including `HotPantsCore` itself, this functionality is maintained by the interface `ComponentSetup`[2].

The interesting public methods of `HotPantsCore` itself are the following[3]:

```
bool    startNote(NoteUniqueId, MidiNote, MidiVelocity);
bool    stopNote(NoteUniqueId);
void    pitchBend(MidiValue);
bool    midiCC(MidiCC, MidiValue);
bool    render(float*, float*, u_int32_t);
```

`HotPantsCore` also provides three methods key to parameter control, although this functionality is actually delegated to an instance of `ParameterManager`, discussed below. The five methods above are sufficient to provide MIDI input and audio output for the instrument in any integration context. Notes are activated using `startNote()`, by providing a unique, arbitrary unsigned integer tracking identification number for the note, its MIDI

[1] Further information regarding plug-in integration of the instrument can be found in Section 3.4

[2] See Section 3.2.2

[3] All data types are defined in `Include/TypesConstants.hpp`

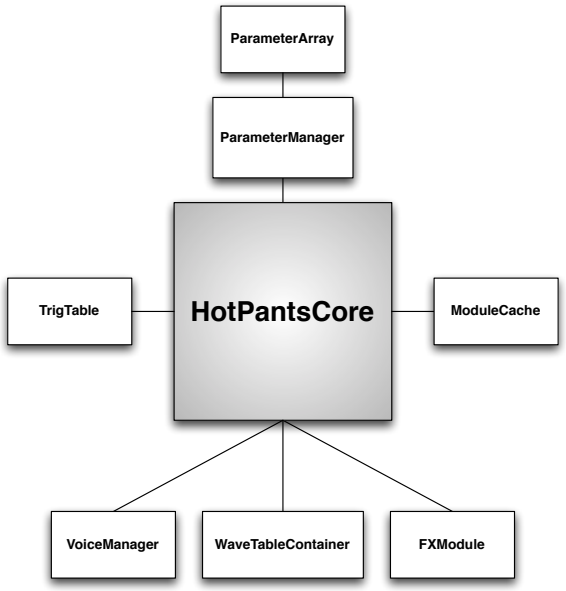


Figure 3.1: `HotPantsCore` object members

note value and MIDI note velocity[4]. Notes can be released by calling `stopNote()` with the identification number previously used to active the note with `startNote()`. The implementation of this note identification system is a consideration for external integration of the instrument; at the very least, the identification number could just be the MIDI note value. However it is more flexible and reliable for an external framework to maintain a list of pressed and released input keys and from this derive a more distinct value.

The MIDI specification also provides for 128 *controls*, sending unsigned 7-bit *control change* values. HotPants provides support for these through `midiCC()`, although only MIDI CC 1—the modulation wheel—is supported internally. The specification outlines other applicable controls, such as 74 and 81 for filter cutoff, 37 for portamento time and 39 for volume, which could easily be implemented using the existing methods, although use of standard control changes is now not a common in software instrument as it once was. MIDI pitch bend, allowing the performer to *bend* the pitch of the note upwards or downwards, is provided with a call to `pitchBend()`, with the desired MIDI control value. Figure 3.2 depicts the standard location of these two controls, directly to the left of the keyboard. Their prominent location explains reasons for specially catering for them when providing expression controls for the instrument.

Figure 3.2: MIDI pitch and modulation wheels

Finally, audio output is provided by the `render()` method, which accepts pointers to two output `float` buffers, representing the left and right channels respectively, of a size

[4]Detailed in the MIDI 1.0 specification, `http://www.midi.org/techspecs/midispec.php`. Generally speaking, MIDI values are unsigned 7-bit.

specified in the third argument. The instrument writes its stereo output to these two buffers in each cycle with a rendering *block size* specified by the third parameter. The CoreAudio framework commonly requests power of two sizes such as 64, 512 and 4096 frames.

ParameterManager

`HotPantsCore` contains a single instance of `ParameterManager`, which in turn contains and manages an instance of `ParameterArray`. `ParameterManager` provides all parameter control functionality, and the public parameter methods in `HotPantsCore` merely pass their arguments on to the `ParameterManager` instance. `ParameterArray` is a simple `struct`-style class with public members, although its constructor does important work, making a copy of the constant static parameter map `ParameterArray::kDefaultParameters[]` to be used as the working array for the instrument. It also copies the default settings of each parameter to their stating values and checks for any human errors that may have been introduced by updating or adding to the instrument's parameters. Similarly, it clones and checks the default module map `kDefaultModuleMap[]`, discussed in Section 3.3.4.

`ParameterManager` provides the actual functionality for the modulation and pitch wheels discussed above, as well as the following methods which provide the interface between the external audio/MIDI framework and the instrument's internal parameters:

```
ParamValue                  getValue(eParameter) const;
bool                        setValue(eParameter, ParamValue);
const _parameterMap&        getParamMap(eParameter) const;
```

The first two methods, `getValue()` and `setValue()` are the public accessor and mutator for a given instrument parameter, enumerated as `eParameter`, defined in `ParamDefines.hpp`. On top of obvious parameter validity checking, a certain amount of additional processing is required in both methods although this will be discussed in proper detail with regard to modulations in Section 3.2.8, and system performance in Section 3.3.4. `getParamMap()` is provided to allow the external plug-in wrapper read-only access to the full range of parameter fields, such as minimum, maximum and default values, the appropriate unit to use and additional data such as whether the control is linear, exponential or logarithmic. It should noted that whilst `ParameterManager`'s function passes a reference, this is converted to a copy by the `getParamMap()` function of `HotPantsCore`. All data is passed by value from `HotPantsCore` to the outside world, so as to ensure complete separation from the external audio framework.

VoiceManager

Of the core modules, `VoiceManager` has the most active responsibilities, providing voices for `HotPantsCore` as well as managing voice rendering. It maintains three containers of instrument voices—class `Voice`, discussed in Section 3.2.3. On initialisation it kills any existing voices and loads its first container—the stack `freeVoices`—with pointers to `kMaxPolyphony` (36) newly free-store allocated voices. The remaining two containers are the standard template library (STL) map `activeVoiceList` and vector `inactiveVoiceList`. `VoiceManager` allocates voices to `HotPantsCore` on `getVoice()` and `stopVoice()` requests by shifting `Voice` pointers between these containers:

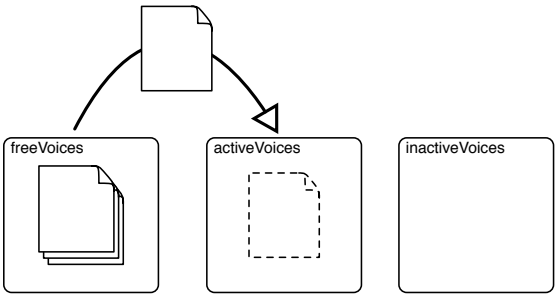


Figure 3.3: Default voice allocation—free stack to active map

First, we must check to ensure that a voice is not already mapped to the specified identification number. If not, voices are allocated by popping a voice pointer from the `freeVoices` stack, wrapping it as a new `ActiveVoice`—described below—and inserting it into the `activeVoices` map, keyed by its specified unique identification number. The normal process for stopping notes is illustrated below:

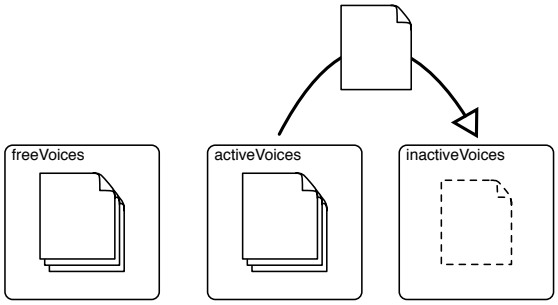


Figure 3.4: Default voice release—active map to inactive vector

The `activeVoices` map is searched for the specified note identification number. If found, the voice pointer is extracted from its `ActiveVoice` wrapper, sent a call to `release()`, has its entry removed from the map, and pointer pushed onto the `inactiveVoices` vector.

Voice stealing

The above two processes are sufficient for the vast majority of note allocation situations. However, the polyphony of the instrument is intentionally limited and occasions can arise where the free voice stack is empty. In order to address this, the `VoiceManager` employs two *note stealing* algorithms. There are two eventualities that we must address:

1. The `freeVoices` stack is empty, but `inactiveVoices` vector is not
2. Both the `freeVoices` stack and `inactiveVoices` vectors are empty

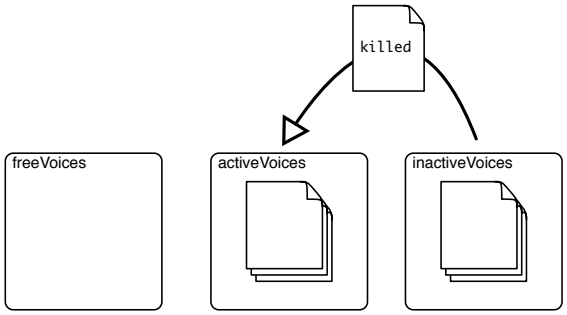


Figure 3.5: Stealing from the inactive voice vector

With regard to case 1, in order to allocate a voice the inactive vector is searched for the quietest note by calling each voice's `getCurrentAmp()` method. This voice is then sent a call to `kill()`, removed from the inactive vector and placed directly into the active voice map. Case 2 is illustrated below:

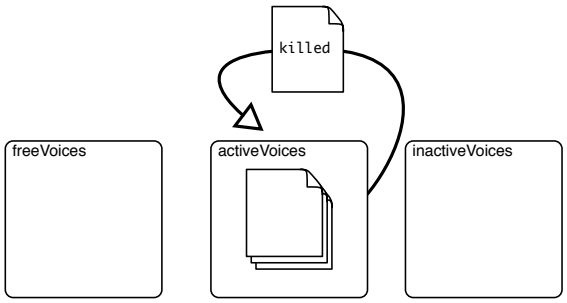


Figure 3.6: Stealing an active voice

Here, an *active* voice, one used by a note the performer has depressed, must be stolen. A different approach is taken here, explaining the use of the `ActiveVoice` wrapper class; the `VoiceManager` maintains a simple incrementing counter which tracks the relative *age* of voices as they are activated. This value is written into each `ActiveVoice` instance created when a voice is added to the `activeVoices` map. The `VoiceManager` uses this value to search the `activeVoices` map for the lowest counter value, and as such the *oldest* voice, and selects it for stealing. We assume that the note that the performer struck *most recently* is currently of the greatest significance, suggesting the note struck *least* recently is the one most expendable.

It is true that this is not the most complex of algorithms and far more intelligent implementations are possible. In particular, the `EnvelopeBase`[5] interface requires a `fastRelease()` method, which could be called when voices were close to exhaustion, beginning the release stage of older notes and meaning that they could be more smoothly *killed* than the current method which cuts them down to zero immediately. In either case, the existing algorithm is reasonably effective.

[5]See Section 3.2.7

VoiceManager rendering

Having discussed how voices are initially allocated, we must now explain how the `VoiceManager` handles rendering each voice. A `Voice` has its own rendering buffer, implemented with an STL `vector`. In the `renderAllVoices()` method, the `VoiceManager` steps through both the active voice map and the inactive voice vector, calling each `Voice`'s `getRenderedBuffer()` method, requesting that the voice render itself to its local buffer, and returning a pointer to the `vector`'s data. The `VoiceManager` object contains a `Mixer`, which receives these pointers with calls to `addBuffer()`, storing them in its own `Voice` vector until such a time as the `VoiceManager` calls its `renderMono()` method. The `Mixer` then mixes all the voices it has received to the output buffer and resets its voice buffer vector for the next cycle.

Freeing Voices

Now that we have explained the rendering process it is possible to fill in the last step of voice management: returning inactive voices to the free stack. This is done during the above described rendering cycle; when voices are moved from the active list to the inactive list, their `release()` method is called, starting the final amplitude envelope stage whereby all voices will eventually become silent. As `renderAllVoices()` steps through the inactive voice vector it calls `isSilent()`; any voices that return `true` have now decayed to silence. Instead of being rendered, they are reset, removed from the inactive voice vector and pushed onto the free voices stack:

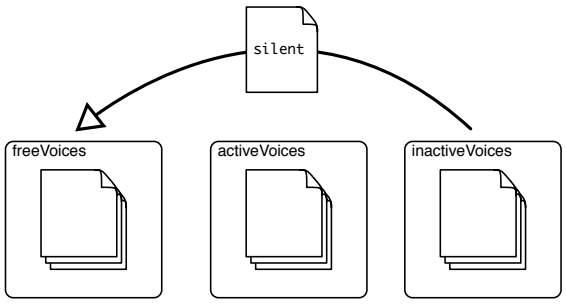


Figure 3.7: Returning silent voices to the free voice stack

Thus explaining the full circle of voice allocation, reuse and rendering in the `VoiceManager`, and completes our discussion of the HotPants core classes.

3.2.2 Interfaces

Whilst we would like to explain the instrument's architecture in a top-down fashion, instead it is necessary to take a step aside at this point and briefly explain the base classes and interfaces used to organise the internal instrument components, since they will need to be referred to in the coming sections. All are defined in `Interface/Interfaces.hpp`.

Component

All internal instrument components, other than `HotPantsCore` sub-class the abstract base `Component`. Each `Component` must be constructed with a reference to the `HotPantsCore`

object at the top of the object tree. By doing so, all instrument objects have a reference back to `HotPantsCore` services such as its `ParameterManager`, wavetables and trigonometry table[6]. This approach is intended to ensure complete object-oriented encapsulation of the design. Without something similar, we would need to rely upon static data, an approach that would make multiple instantiation of the instrument impossible. Having a reference member requires us to define an assignment operator, although this implementation merely returns `*this`, since the reference to `HotPantsCore` will not change throughout the life of any internal components:

```
Component& operator=(const Component& c) { return *this; }
```

ComponentSetup

The base interface for all components that rely on the current sampling rate, which applies to most rendering components. `ComponentSetup` has a protected sampling rate member and pure virtual `initialise()` method. A macro, `_MACRO_SETSAMPLINGRATE_(x)`, is provided in `Interfaces.hpp` for components to use for checking and setting their sampling rate member in an `initialise()` implementation. Whilst this could have been implemented in `ComponentSetup::initialise()`, this would have made `ComponentSetup` a concrete class, where an interface is more suitable. Since the macro includes only two instructions, we consider this a valid use for what is often considered a controversial language construct.

Rendering Interfaces

`MonoRenderer`, `MonoToStereoRenderer` and `StereoRenderer` are fairly self-explanatory, and merely define the pure virtual methods `renderMono()`, `renderMonoToStereo()` and `renderStereo()` for the three types of rendering components: single channel, stereo components, and components that receive a mono input—the left channel—and generate a true stereo output. The majority of current instrument components produce a mono output, with `StereoDelay` the only mono-to-stereo component, and `VolumeStereo`, `DelayHighPass` and `DelayLowPass` the only stereo units.

NoteResponder

Defines an interface for modules that respond to the pitch or velocity of a new MIDI note in some way: oscillators, of course, respond directly to the pitch of a note; the amplifier uses the note velocity to apply velocity sensitivity, and filters can key-track the received note frequency.[7]

Modulator

Interface for the modulation sources—LFOs[8] and modulation envelopes[9]. These components output a single sample for each rendering cycle, used to set modulations before the audio rendering cycle begins. The instrument modulation system is discussed in full in Section 3.2.8.

[6] Section 3.2.9
[7] Section 3.2.4
[8] Section 3.2.5
[9] Section 3.2.7

DynamicComponent

`DynamicComponent` is discussed in Section 3.3.4.

Module and CacheableModule

Other than `ModModule`, which is not a renderer, all voice modules—detailed in the next section—implement the `Module` interface, in itself a combination of `CacheableModule`, `ComponentSetup`, `MonoRenderer` and `NoteResponder`:

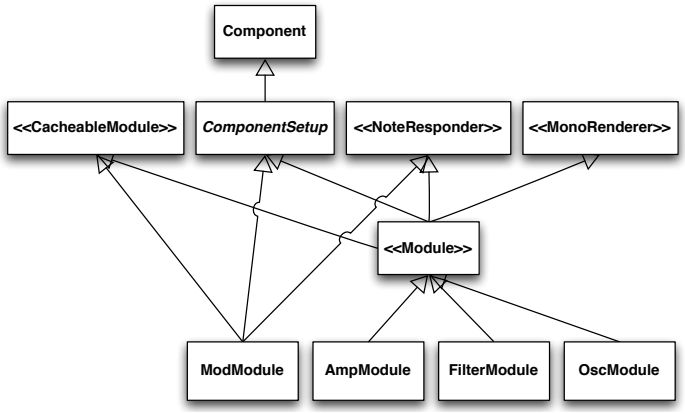


Figure 3.8: Module interface hierarchy

As can be seen from the diagram, `ModModule` is not a renderer, and is instead a combination `NoteResponder` and `CacheableModule`. The role of `CacheableModule` is discussed in Section 3.3.4.

3.2.3 Voice

Having covered the interfaces governing instrument components, we can now move on to describing their function. We start with the highest level: `Voice`. Since HotPants is a polyphonic instrument and each voice must operate independently, each voice requires its own set of oscillators, its own set of filters, its own modulation controller and its own overall amplitude envelope. The class `Voice` encapsulates all of these requirements, containing the four *modules* `OscModule`, `FilterModule`, `AmpModule` and `ModModule`, representing these lower level components. Much of this class's functionality is delegated to these modules, detailed below, although it does have essential features for improving the overall performance of the instrument, discussed in Section 3.3.4.

3.2.4 Modules

The module system, then, serves to divide the behaviour of a voice into four instrument modules. We discuss their function below.

OscModule

`OscModule` contains the instrument's three main audio signal oscillators, `Osc1`, `Osc2` and `Osc3`, specialisations of the `AudioOsc` and `OscillatorBase` classes[10]. `OscModule` conforms to the `Module` interface, and in turn the `NoteResponder` interface, requiring the `noteOn(MidiNote, MidiVelocity)` method, activating all three contained oscillators for a new note. It also contains a `setBend()` method, which is used to deliver a portamento note bend from the last struck MIDI note to the pitch of the current note. `OscModule` renders its oscillators in its `renderMono()` method by calling `renderMono()` on each oscillator in turn. Of note here is that, in order to deliver oscillator modulations such a FM, AM and RM, `OscModule` renders to the buffer in the order, `Osc2`, `Osc1`, `Osc3`. This will be discussed further in Section 3.2.5.

FilterModule

Similarly, `FilterModule` contains the instrument's two filters, `Filter1` and `Filter2`. As with `OscModule`, it conforms to `NoteResponder`, in order to implement filter *key-tracking*. This allows the filter to track the pitch of the note by a specified ratio, maintaining a proportional—or disproportional, if desired—attenuation of frequencies.

Since the filter module allows for both serial and parallel connection of filters, the `renderMono()` method of the `FilterModule` is a little more involved. Serial connection is straightforward, passing the signal first to filter one, and then to filter two. Parallel connection, however, requires the filters to be processed separately using the module's own local rendering buffers. These can then be mixed together using the ratio currently specified by the instrument parameter `kParam_FiltMix`.

AmpModule

`AmpModule` is the simplest of the four `Voice` sub-modules, containing a single amplifier envelope and managing it along with its relevant parameters. Its only independent functionality is to provide note velocity sensitivity, in line with the parameter `kParam_Velocity`, which scales the amplifier based on this parameter and the velocity of the note struck by the performer.

ModModule

The `ModModule` is discussed in Section 3.2.8.

3.2.5 Oscillators

Now that we have described the modules that sub-divide a `Voice`, we can discuss the low-level components of which each module is constructed. It is at this level that signals are actually generated and processed.

OscillatorBase

`OscillatorBase` represents the common link between the types of `AudioOsc` and `LFO`. Recalling our discussion of wavetable oscillators from Section 1.4.1, the basic operation

[10] See Section 3.2.5

involves stepping through an array which holds a single cycle of a particular waveform. We can derive from Equation 1.1 that stepping through the table at an increment of 1—stored as `IndexInc` in `OscillatorBase`—generates:

$$\text{table frequency} = \frac{\text{table length}}{\text{sampling rate}} \tag{3.1}$$

As this value will remain constant for a given sampling rate, it is pre-calculated in `OscillatorBase::initialise()` and stored as `tableFreq`. The method `calcIndexInc()` can be called to multiply this value by the current note, `frequency`, and so calculate the correct index increment for the desired note.

The current oscillator sample is stored locally in the variable `sample`, a performance optimisation explained in Section 3.3.4. Other than this, `OscillatorBase` conforms to the `DynamicComponent` interface and so defines a `refresh()` method, as well as storing a pointer to its owner `HotPantsCore` object's `WaveTableContainer`.

WaveTableBase

The abstract base class `WaveTableBase` provides the framework for all of the instrument's wavetables. All sub-classes must provide a default constructor that fills the protected `table[]` array with a single cycle of their waveform. The size of the array is fixed at compilation to the constant `kTableLength`, currently set to 16,384 entries, given advice by Mitchell [10, pg.86-9]. An example is provided for the simplest of the tables, `SineDirect`:

```
SineDirect::SineDirect()
{
  PhaseValue phaseInc = kTwoPi / kTableLength;
  PhaseValue phase = 0.0;

  for(IndexedValue i = 0; i < kTableLength; i++) {
    table[i] = std::sin(phase);
    phase += phaseInc;
  }
}
```

Sub-classes must also provide a `getWaveForm()` method to advertise the waveform they provide as an enumerated `eWaveform`. Access to the tables is provided via the inherited `getFrameAtIndex()` method, which is sufficient for all *real* wavetables[11].

Provided wavetables are split into two types; direct calculation and additive Fourier series synthesis and we shall here provide an example for both. Direct types involve use of a trigonometric or algebraic equation to generate a pure waveform. The following example is for the type `SawDirect`:

```
SampleInternal ampInc =  (2 * kAmpMax) / kTableLength;
SampleInternal ampVal = -(2 * kAmpMax);
for(IndexedValue i = 0; i < kTableLength; i++) {
  table[i] = ampVal + kAmpMax;
  ampVal += ampInc;
}
```

[11] But not for *virtual* wavetables; see Section 3.2.5

This is, thus, the linear equation:

$$y = \frac{2x}{kTableLength} - kAmpMax \tag{3.2}$$

Producing a single cycle of the saw wave:

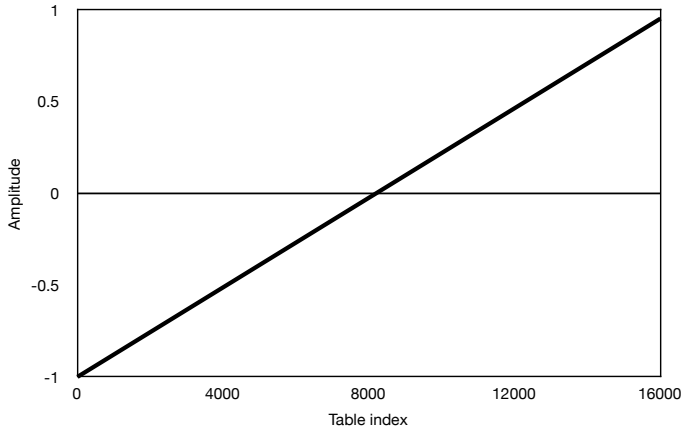


Figure 3.9: Linear equation sawtooth waveform

We have also implemented the wavetable `SawPartial`, which uses additive synthesis, summing sine waves to produce the saw, applying the Fourier series equation [7, pg. 111]:

$$x(t) = -\frac{2}{\Pi}\sum_{n=1}^{\infty}\frac{1}{n}sin(2\Pi nt) \tag{3.3}$$

Here we present a lower resolution plot of the wavetable itself, from a smaller demonstration table of one hundred elements:

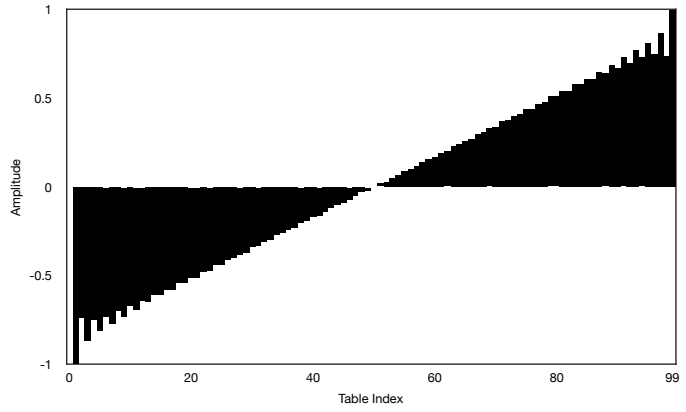


Figure 3.10: Fourier series saw tooth waveform (reduced resolution)

In Figure 3.10, the imperfections caused by summing a non-infinite series can be seen. However, for audio applications this is not of great concern; we wish to limit the number of harmonics in any case, so as to avoid creating aliasing with frequencies that cross the

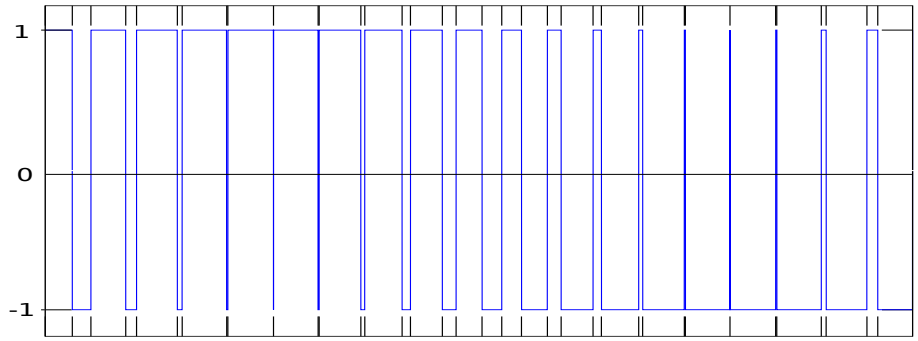


Figure 3.11: Pulse-width modulated square wave

Nyquist limit. In fact, the sharp corners produced by the *perfect* sawtooth in Figure 3.9 render it unsuitable for use as an audio source. Therefore, the distinction for all provided waveforms is that Fourier series waveforms are used an audio sources, with direct calculated types used for modulation source low-frequency oscillators.

Virtual wavetables

As well as the above types, there is a third kind employed in the instrument; virtual, wrapped signal generators. These types do not generate a wavetable and instead produce an output through their own means, overriding the `WaveTableBase::getFrameAtIndex()` method to do so. `NoiseWrapper` contains an instance of the white noise generator `Noise`, which outputs random samples between `kAmpMax` and `-kAmpMax` using the `C` standard library `rand()` function. `NoiseWrapper` ignores the requested table index and instead outputs a frame from its `Noise` object. `PulsePWM` is an implementation of a square wave with variable pulse width, the effect of which is illustrated in Figure 3.11. `PulsePWM` uses the requested table index and current pulse width instrument parameter to decide whether to return `kAmpMax` or `-kAmpMax`:

```
pulseWidthMidPoint = kTableLength *
  core.getParamManager().getValue(kParam_PulseWidth);

if(index <= pulseWidthMidPoint)
  return  kAmpMax;
else
  return –kAmpMax;
```

Due to it needing access to the parameter manager, `PulsePWM` conforms to the `Component` interface. This wrapping approach is a simple way to allow these oscillators to fit into the existing `WaveTableContainer` framework.

WaveTableContainer

In order that oscillators can refer to stored wavetables, `HotPantsCore` contains a `WaveTableContainer` object, which in turn contains an array of pointers to one of each available

wavetable. As such, all instrument oscillators share a single set of wavetables. It would be entirely possible to move the `WaveTableContainer` to `Voice`, thus giving each voice its own store of wavetables. However, this seems rather wasteful in terms of memory; on our main development system, with `double` samples stored in a wavetable size of 16,384 entries, each wavetable occupies 128kB. With eight wavetables available[12] this totals about one megabyte. This is quite acceptable for `HotPantsCore`, but perhaps not for each of thirty-six or more available voices. `WaveTableContainer` provides the method `getFrameFromTableAtIndex()` for public access to any index of any of the available wavetables.

LFO and AudioOsc

Two sub-classes of `Oscillator` provide the full functionality for instrument audible oscillators and modulation low-frequency oscillators, and are presented here.

LFO

An `LFO` differs from an `AudioOsc` in that only a single sample need be calculated for the beginning of a rendering cycle, conforming to the `Modulator` interface. This value is then used by its owner `ModModule` to set the relevant instrument parameters before rendering of components begins in earnest. As such, the `getModValueForBlock()` method prepares to return the oscillator sample for the *start* of the rendering block, then multiplies the current `indexInc` by the size of the block, effectively skipping to the end of the rendering cycle. Further discussion of the reasoning behind this, and its benefits can be found in Section 3.3.4. The concrete classes `LFO1` and `LFO2` simply report their role in the `getThisOsc()` method, simplifying the process of accessing their respective instrument parameters.

AudioOsc

Unlike `LFO`, `AudioOsc` provides a `renderMono()` method for filling a complete buffer with audio samples, as well as `setFrequency()` and `setBend()` methods for recalculating index increment for a new note and setting up note bending portamento using the variables `portoIndexInc`, `portoDuration` and `portoCounter`, which modify the `indexInc` for the current portamento duration.

Most significantly, however, further specialisations of `AudioOsc`, `Osc1`, `Osc2` and `Osc3`, are defined to deliver the unique functionality required to provide oscillator modulations. The reader will recall from Section 3.2.4 that during a rendering cycle, `Osc2` is counter-intuitively written to the output buffer *before* `Osc1`. In Sections 1.4.2 and 1.4.3 we discussed the oscillator-level modulation techniques of frequency modulation, ring modulation and amplitude modulation. These techniques require a *modulator* and *carrier* signal. In this case, `Osc2` provides the modulator signal, and `Osc1` the carrier. Writing `Osc2` first thus greatly simplifies the process of applying modulation to `Osc1`, since the modulation value is waiting in the buffer when we come to calculate the carrier sample. In the case of AM and RM, this is an extremely simple operation:

```
buffer[i] *= sample * amp;              // RM
buffer[i] *= (sample + 1.0) * amp;      // AM
```

[12] Note that the virtual wavetables `NoiseWrapper` and `PulsePWM` inherit the `table[]` array even though they make no use of it

The only difference is that, in the case of AM, the signal must be *unipolar*. Since all oscillators produce a signal between +1.0 and -1.0[13], the addition of 1.0 ensures that the modulator is always above zero.

The process is a little more involved for FM, since we must modulate the *frequency* of the carrier. `AudioOsc` provides the inline function `advanceFrames()` which advances the floating point index counter `indexFP` through the current wavetable cycle. It also accepts a floating point argument, `f`, adding this value to the current `indexInc`:

```
indexFP += indexInc + f;
```

For all other `AudioOsc` renderers this method is called with `f = 0.0`. However, for FM this allows us to modulate `indexFP` by the current modulator value, stored in the buffer:

```
advanceFrames(buffer[i] * kFMmultiplier);
buffer[i] = (sample * amp);
```

The modulator value is then overwritten by the frequency modulated carrier[14], creating the FM effect. Since all kinds of oscillator modulations are more pronounced when the carrier and modulator are not harmonically tuned, `Osc2` also provides a fine-tuning control of ±100% of a semitone and overrides the `refresh()` method to incorporate this parameter.

3.2.6 Filters

`RBJFilter` is the main filter base class for the HotPants instrument. It is an implementation of Robert Bristow-Johnson's *Audio EQ Cook Book* filters[15], adapted for use in the instrument. This base class provides the basic functionality for coefficient calculation and filter rendering, and the pure virtual method `getThisFilter()` ensures that `RBJFilter` must be sub-classed to provide specific functionality for the five concrete sub-class types:

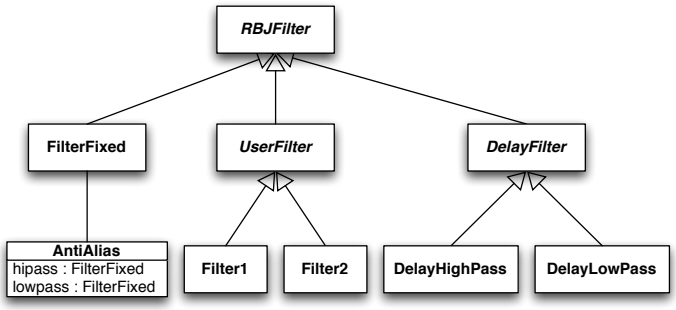


Figure 3.12: Module interface hierarchy

`UserFilter` adds the key-tracking `setKeyTrackNote()` function, as well as overriding `renderMono()` in order to provide the required `DynamicComponent` parameter change

[13] In the source code, this magnitude is represented by the constant `kAmpMax`

[14] Note that `kFMmultiplier` is a constant defined in `TypesConstants.hpp` that allows us to fine-tune the overall amount of FM applied. Currently it is set to 1000, which seems to provide a good level of modulation.

[15] `http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt`, original implementation `http://www.musicdsp.org/files/CFxRbjFilter.h`

checking functionality. `DelayFilter` converts the `RBJFilter` into a stereo filter by creating a second set of filter coefficients and a `renderStereo()` method. `DelayFilters` allow changes to the filter cutoff, but their filter types and bandwidth–Q—settings are fixed.

The user filters, `Filter1` and `Filter2` merely provide `getThisFilter()` methods, identifying themselves. `FilterFixed` makes the `refresh()` method private and the `recalculate()` method public, so that coefficients are calculated once at initialisation, but otherwise are static. As can be seen from Figure 3.12, a fixed low-pass and fixed high-pass filter are employed to create an `AntiAlias` filter, employed in the `FXModule`, in order to improve output audio quality.

DCFilter

HotPants contains a sixth filter type, `DCFilter`, which is not an `RBJFilter` sub-class and is an implementation of a Smith's DC Blocker difference equation[19, pg. 272–278]:

$$y(n) = x(n) - x(n-1) + Ry(n-1) \tag{3.4}$$

Where R is a coefficient selected to produce a desired frequency response:[16]

$$R = \frac{2\pi f}{f_s} \tag{3.5}$$

Where f_s is the sampling rate and f the desired -3dB attenuation frequency. For our purposes, we have select a value of 30Hz. The `DCFilter` is inserted into the signal chain in the `FXModule` after the `AntiAlias` filter. The role of `DCFilter` and `AntiAlias` is discussed further in Section 3.3.1.

3.2.7 Envelopes

HotPants provides a single main envelope type, `ADSRcurve`, and the interface `EnvelopeBase` which declares some protected members and the pure virtual methods required for proper operation with a `VoiceManager`:

```
SampleInternal  getCurrentAmp() const { return currentAmp; }
virtual bool    isSilent() const=0;
virtual void    release()=0;
virtual void    fastRelease()=0;
virtual void    kill()=0;
virtual void    reset()=0;
```

These methods are required for the normal process of releasing notes, as well as note-stealing algorithms and are discussed in Section 3.2.1. The static helper method `timeToSampleDuration()` is also provided for converting envelope parameters in seconds to durations in number of samples.

[16] `http://www.musicdsp.org/showone.php?id=135`

ADSRcurve

`ADSRcurve` itself, is a four stage attack-decay-sustain-release envelope with curved, logarithmic stages.[17] Our first envelope implementation featured linear segments, however, as noted by Mitchell [10, pg. 47], this does not give satisfactory results as either an amplitude or modulation envelope. Instead, the final implementation is effective and reasonably efficient, providing a parameter `kParam_AmpScaling` which allows the user to control the depth of the envelope curve from very deep (100%) to shallow (0.01%).

An envelope is created with a call to `createEnvelope()`, providing an `ADSRcurveParams struct` which contains the desired envelope settings; stage lengths and curve depth. The envelope itself functions as a state machine, with the array `stateMachine[]` managing the four states, including the final pseudo-stage, `_EnvStageNoteOff_`, indicating that the voice is now silent. Its data is populated in the `createEnvelope()` method, converting stage times into sample durations and calculating the values for `scale1` and `scale2` which are used to calculate the amplitude for each frame:

```
// calculate current amp
currentAmp =
  (timeScale - stateMachine[currentState].scale2 + 1.0)
  * stateMachine[currentState].range
  + stateMachine[currentState].offset;

  ...

// update scale2 using scale1
stateMachine[currentState].scale2 *=
  stateMachine[currentState].scale1;
```

The struct `_envState` is defined in `EnvelopeTypes.hpp` and carries the data for each stage; in particular, the counter `duration`, which is decremented in the `renderMono()` method. When zero is reached:

```
currentState = stateMachine[currentState].nextState;
```

This advances to the next stage and is an approach chosen to feasibly allow more complex envelopes to be added to the existing framework; the enum `EnvelopeStage` can be extended with additional stages up to that of the most complex envelope. Transitions between stages can be managed for both simple and complex configurations using this `nextState` variable.

One challenge faced during the implementation of the `ADSRcurve` envelope was an approach for dealing with early note releases. In our earlier discussion of envelopes in Section 1.4.1, depicted in Figure 1.2, the reader will recall that the purpose of the sustain stage is to maintain the voice's amplitude until the key is released. When the envelope is created, all four stages are calculated, assuming that the sustain stage will be reached. However, the performer may release the note during the attack or decay stage, *before* sustain is reached. Should they do so the values previously calculated for the release stage become redundant. In order to solve this problem, early release can be detected and the `calcRelease()` private member function called to recalculate an early release on the fly. This approach appears satisfactory.

[17] See figure 1.2

ModEnvelope

An `ADSRcurve` sub-class, `ModEnvelope` provides `Modulator` interface-compliant behaviour. Ideally, the `ModEnvelope` would provide performance enhancements similar to `LFO`, calculating a single value for the start of a rendering cycle and then skipping the envelope forward by the specified block size. However, due to the stages of the envelope and logarithmic calculations used to calculate each amplitude value, it is possible a solution would be unnecessarily complex, and could not easily be implemented without completely rewriting the existing `ADSRcurve::renderMono()` method. For the time being, the `ModEnvelope` maintains a local buffer and ignores all but the first enveloped sample.

3.2.8 Modulation system

The modulation system for HotPants is delivered through the `ModModule` object and the instrument's `ParameterManager`. `ModModule` encapsulates modulation handling for each `Voice`; two low-frequency oscillators and two modulation envelopes, one of which being a full ADSR, the other a simpler AD—attack, decay. A single sample for each of the four is calculated at the start of a rendering cycle with a call to their `Modulator` interface-declared `getModValueForBlock()` method. Modulations are set with a call to the `ParameterManager::setMod()`. The `ParameterManager` itself does not modify the parameter's value directly:

```
paramMap.settings[p].mod = modValue;
paramMap.settings[p].modified = true;
```

The parameter map contains an additional field, `mod`, which is used to write the modulation value. The `modified` flag is set to allow the `ParameterManager::paramModified()` to report whether or not a value has been modified. The `ParameterManager::setValue()` similarly modifies this flag when a parameter is modified by the user. This is another performance enhancement designed to avoid unnecessary recalculations[18]. Later, when an internal component requests a parameter value with `ParameterManager::getValue()`, the modulation value is combined with the parameter value to deliver the modulated result. As a additional quality improvement measure, a further parameter array field, `delayVal`, is used by the `getValue()` method to apply a simple low-pass moving average filter to the modulated parameter, `p`:

```
value = (value + paramMap.settings[p].delayVal) * 0.5;
paramMap.settings[p].delayVal = value;
```

Although we calculate only one modulation value for each rendering block, expected rendering block sizes of a fraction of a second[19], along with the above parameter smoothing, gives us a result that is not perceptibly quantised to the user. Indeed, the modulation update frequency could probably be reduced further to increase performance without a loss of quality.

[18] See Section 3.3.4

[19] A common CoreAudio rendering buffer size at CD-quality 44.1kHz is 4096 samples, or around a tenth of a second

3.2.9 Audio effects

HotPants also provides three line audio effects, contained within an `FXModule` object. Since all of these effects process all instrument voices together there only need be a single `FXModule`, owned by `HotPantsCore`.

VolumeStereo

`VolumeStereo` is a simple stereo gain control, responding to the instrument parameter `kParam_GlobalVolume`. Its functionality is presented in Section 3.3.4.

DigitalDistortion

As discussed in Section 1.4.4, we have implemented hard-clipping and fold-back distortion. Firstly, the signal is amplified by the gain factor given in the parameter `kParam_DistortGain` and the input buffer is examined. In the case of hard-clipping, any values whose absolute value exceeds `kAmpMax` are limited to that value:

```
if(buffer[i] >  kAmpMax)  buffer[i] =  kAmpMax;
if(buffer[i] < -kAmpMax)  buffer[i] = -kAmpMax;
```

In the case of fold-back, samples that exceed `kAmpMax` are *folded-back* against that limit, by adding or subtracting it:

```
if(buffer[i] >  kAmpMax)  buffer[i] -= kAmpMax;
if(buffer[i] < -kAmpMax)  buffer[i] += kAmpMax;
```

Finally, the volume of both signals is divided by the input gain factor, to maintain a constant overall volume irrespective of the distortion gain amount.

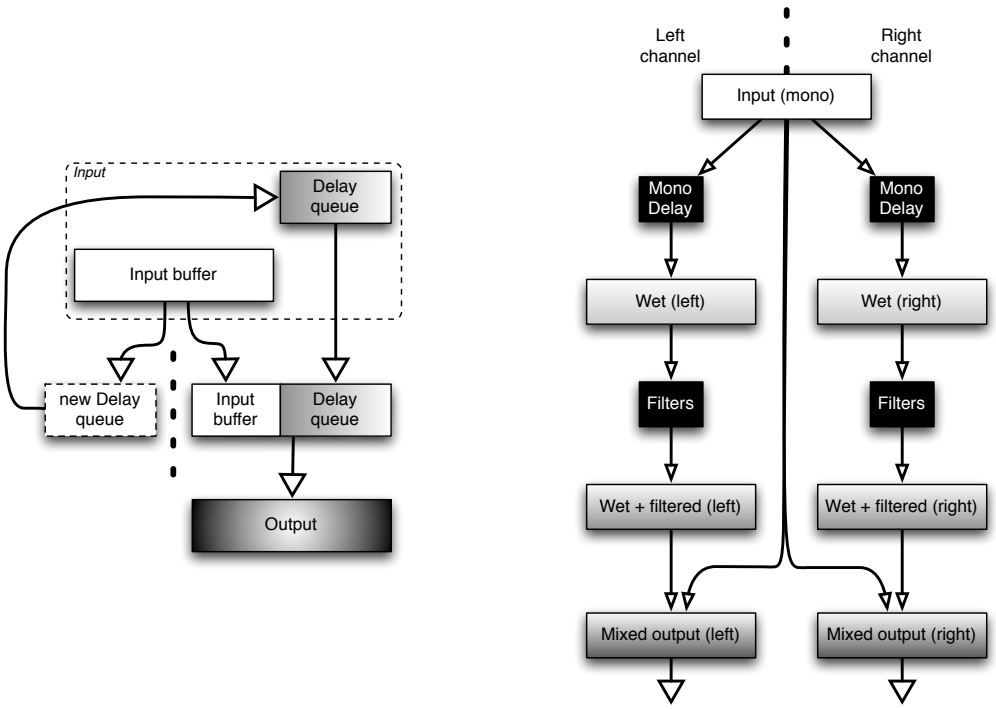
Delay

The more involved of the effects is `StereoDelay`, which combines two `MonoDelay` objects for each of the two stereo channels, providing variable delay time, mix and feedback for each as well as high and low pass filtering for the delayed signal. `MonoDelay` contains a standard library FIFO queue as its delay buffer, as well as its own output buffer, which simplifies the process of mixing the original and effected signal.

`MonoDelay`, Figure 3.13a, iterates through the input buffer, popping samples from the front of the delay queue, writing them to its local buffer, and pushing input samples onto the back of delay, mixing in a percentage of the current delay sample in order to create a feedback loop. Note that in the diagram we represent the delay time as shorter than the output buffer for simplicity of illustration; in practice this may or may not be true. In any case, the process functions identically.

`StereoDelay`, Figure 3.13b, sends its input buffer to both the left and right `MonoDelay`s, receiving back references to their effected, *wet* local buffers. Next, it sends the wet signals through its `DelayHighPass` and `DelayLowPass`[20] filters. Finally, it mixes the wet left and right channels with the dry input buffer, creating a true stereo delay effect. We have

[20] See Section 3.2.6



(a) Mono delay

(b) Stereo delay

not implemented a separate reverb unit for the instrument, but similar results can be achieved with the stereo delay, with very short delay times, high mix, a lot of feedback and appropriately set filters. For a non-dedicated unit, the results are reasonably authentic and don't require an entirely separate effect unit.

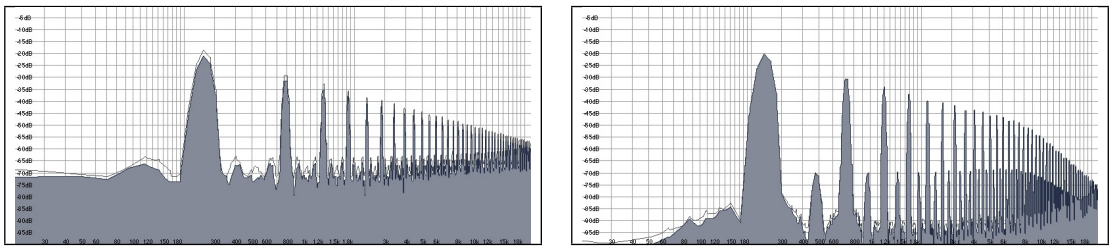
Utilities

Finally, we discuss the `Utility` module, which provides two classes and a namespace of functions that are used in various parts of the instrument. The module is intended to centralise these mostly mathematical operations.

`Utility` is a class with a series of static functions, giving common mathematical operations, including `fpEqual()`, a floating point number comparison operation; `dBGainToPower()` which converts gain factors given in decibels to a signal multiplier; `midiControlToFloat()` which converts a 7-bit MIDI control value to a `float` decimal in the range of 0.0–1.0. `midiNoteToFreq()` uses a `FreqMap` to map MIDI note values to their frequency, and is intended to avoid the need to calculate these values when new MIDI notes are received from the host. Similarly, the class `TrigTable` provides the standard trigonometry functions, `sin()`, `cos()` and `tan()`, implemented using the provided `SineDirect()` wavetable to deliver values. A number of conditional compilation operations control the behaviour of this class[21]:

- `TRIG_TABLE`—enables or disables use of the sine wavetable. When disabled, functions use the standard library `sin()` function

[21] All conditional compilation `#define`s can be found in `Include/Compilator.hpp`



(a) HotPants (b) Albino

Figure 3.13: Comparative signal noise—square wave, C4 (261.63Hz)

- `TRIG_INTERPOLATE`—interpolates values from the wavetable. This greatly improves the accuracy of values calculated
- `DEDICATED_TRIG_TABLE`—if enabled, a `TrigTable` object will have its own `SineDirect` object and wavetable. Otherwise, it will make use of the table belonging to its parent `HotPantsCore`

Finally, `BufferUtils` provides two template functions operating on the buffer types used in the instrumen, including template specialisations to allow them to operator on standard library FIFO queues, such as that used by the `MonoDelay`.

3.3 Implementation details

Now that we have described the instrument design as realised into classes and modules, we turn our attention to certain notable aspects of the implementation process; in particular difficulties that we faced, techniques that were employed and issues that remain, that could be improved upon in future versions of the instrument, or which taught important lessons.

3.3.1 Technical challenges

In terms of this project, we would define *technical* challenges as being those related to the signal processing side of the instrument. The other possibility would be those related to implementing the instrument in C^{++}. Since at least workable solutions to all possible points of discussion were invariably found to complete the instrument, we discuss this as a question of style and technique in Section 3.3.2. In contrast, with regard to DSP we did face some considerable challenges which have had a bearing on the success of the completed instrument.

Aliasing and wavetable signal quality

In spite of a great deal of study and preparation, aliasing produced by our wavetable oscillators proved to be a persistent concern. It proved challenging to reach a good compromise between rich signal quality versus noise caused by aliased frequencies.

Figure 3.13 shows early frequency plots that demonstrated that there was a noise problem in need of addressing, in comparison to Albino, one of our benchmark commercial instruments. In these plots, the first harmonic can be seen at 261.63Hz, and the odd harmonics that constitute the waveform can be seen as the following sharp peaks. Ideally there

should be little or no signal below the first harmonic; this is the case with Albino, but not with this early implementation of HotPants, where a noise floor caused by foldover can be seen throughout the frequency range. Clearly our tables were not sufficiently band-limited.

The use of additive synthesis in our tables allows us a great deal of control over the harmonic content of the signal and Figure 3.3.1 shows a square wave with a much lower number of harmonics than those of Figure 3.13a. The noise floor has been greatly improved but the signal is lacking in high frequencies. Since we are using a single, static wavetable for each waveform any artificial harmonic reduction will be proportional to the note being played meaning that especially low frequency notes will be *dark* and lacking in high frequency colour. The compilation option `WTABLE_CONST_HARMONICS` generates tables based on numbers of partials suggested by Mitchell [10, pg. 96], however we found these signals to be rather bland. A possible solution to this problem could be the use of *multiple* wavetables; one for each of the 7¼ MIDI octaves with a proportional reduction in harmonics for the higher frequency tables. This did not, however, seem an elegant solution.

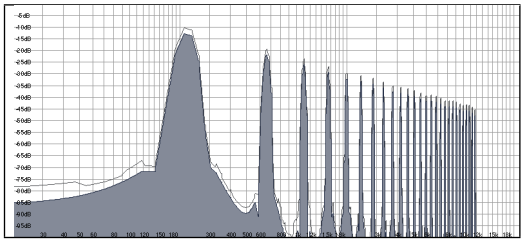


Figure 3.14: Wavetable limited harmonics

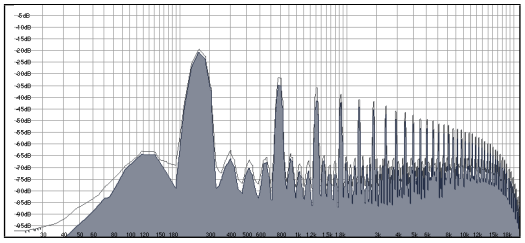


Figure 3.15: Filtered square wave

Another solution, discussed by Mitchell [10, pg. 82], is dynamic summing of sinusoids; discarding our complex waveform wavetables and instead calculating a Fourier series signal on the fly, summing entries from the `SineDirect` wavetable up to the appropriate number of harmonics for the current note frequency. Some initial implementation of this technique was performed, but the performance was found considerably lacking in comparison to static wavetables.

Instead, the approach we have taken is use of the `DCFilter` and `AntiAlias` filter, discussed in Section 3.2.6. As can be seen from Figure 3.3.1, low and high frequency noise have been significantly reduced although a floor is still present throughout the frequency range. It seems that this is also an imperfect solution, but is perhaps a compromise that we must reach for the purposes of this project. A possibility for a complete re-implementation could be the *bandlimited impulse train* method discussed by Stilson and Smith [20] although these are techniques far removed from those implemented in this project thus far.

3.3.2 Programming techniques

In this final section, we present some of the approaches taken in order to realise the instrument in C^{++}, as well as how it has been integrated with an external audio framework.

Object-oriented design

There are elements of the OO design methodology that are heavily used in HotPants, others that are not:

- **Inheritence** is heavily used for the establishment of hierarchy in the instrument, as well as code reuse. The latter appears to be somewhat counter to C++ conventions[22], and quite extensive use of multiple inheritance did cause a number of issues during development until it was greatly reduced in favour of the interfaces discussed in Section 3.2.2. This experience would tend to confirm the above opinion about code reuse in C++. Inheritance of the `Component` constructor did prove to be powerful, if a little unwieldy at first.
- **Polymorphism** is used only in `WaveTableContainer`, which stores an array of pointers to `WaveTableBase` sub-classes. Here it has proved very useful. In other places we have resorted to variations of the `C`-style `switch` approach, a good example of which would be `Osc1::renderMono()`, which `switch`es between the four available rendering modes. This could have been implemented as an additional set of polymorphic classes, with a container stored in `Osc1`. We do, however, feel that this would unnecessarily extend an already large inheritance hierarchy. Perhaps, also, the instrument is not sufficiently dynamic to benefit from this approach. We feel that we have tried to use either the polymorphic or procedural approaches in appropriate places, an approach which seems to be endorsed by Stroustrup [21, pg. 417/727].
- **Encapsulation**—HotPants makes standard use of encapsulation, with members generally private except in instances where classes are more or less `struct`s with some associated convenience code, such as `ParameterArray`. Some use has been made of `friend` functions and classes, however. `UnitTest` is a friend class of `HotPantsCore`, useful for testing and debugging, and `ParameterArray` is a friend of `ParameterManager`, which is the only class that can construct an instance, since its constructor is private. We have tried to use `friend`ship as a tool to enhance encapsulation [21, pg. 278–82].

Operator overloading

One of the most powerful features of C++ is the ability to override the standard language operators. Due to our use of a reference member in the `Component` interface it has been necessary to override the assignment operator and copy constructor in some places. `Component` provides an assignment operator that ignores the reference—since it will never need to be reassigned. `MonoMixer` provides its own version of the assignment operator and copy constructor, functionality necessary for the `ModuleCache` which requires both operations.

Arrays, containers and enums

HotPants makes wide use of many available containers types. On one hand, since our architecture is static, we have made use of a large number of arrays, particularly those used for mapping between enumerated types; for example `kModMap[]` maps between `eModT` modulation targets, and `eParameter`, the associated instrument parameter[23]. This does, of course, expose us to the risk of array out of bounds errors although in practice we were quite able to protect against these bugs since array sizes are defined by and iterated over by our own enumerated types:

[22] `http://www.parashift.com/c++-faq-lite/smalltalk.html#faq-30.4`. Also, Stroustrup seems ambivalent [21, pg. 714-5]

[23] See `Include/ParamMaps.hpp`

```
enum eParameter {                    // Enums.hpp
  kParam_AmpAttack = 0,
  ...
  kNumOfParams
};
const eParameter kFirstParam = kParam_AmpAttack;

...
                                     // operate on array
for(eParameter p = kFirstParam; p < kNumOfParams;
                p = static_cast<eParameter>(p + 1))
{ ... }
```

The above looping approach is something of a bending of C++ enums, although it gives us the desired flexible, but runtime static ranges. Overall, this gives us the performance of static arrays, without the associated risks. In addition, we have also made good use of the C++ STL containers, especially in `VoiceManager`, which employs `std::stack`, `std::map` and `std::vector` as appropriate types for the three voice containers. Elsewhere, `MonoDelay` employs a `std::queue`, a type ideal for its FIFO needs.

Memory management

The modules which deal with pointers to objects are `VoiceManager` and `WaveTableContainer`[24]. It should be possible to remove the `new` and `delete` operations from `VoiceManager` and instead take the addresses from a static array of voices generated at compile-time. There are currently no objects that allocate memory during instrument operation, and to do so would not be desirable in terms of performance. Due to its static architecture, it ought to be possible to redesign the instrument with no heap allocation operations whatsoever, and this would perhaps be desirable, reducing the scope for errors. In any case, we have had very few issues with memory management when implementing and testing the instrument.

Conditional compilation

As noted earlier, HotPants makes use of a number of conditional compilation options, defined in `Include/Compilation.hpp`. This has been particularly useful for *large* features, such as multithreading support, which considerably slows down compilation when enabled. Additionally, there are experimental features such as `WTABLE_CONST_HARMONICS`[25], and instrument signal-flow adjustments such as `VOL_IN_FXMODULE` which plays the master volume control in either the `FXModule`, or `HotPantsCore`. Overall, this has been an extremely useful technique to employ, giving us convenience and flexibility to experiment with features.

Exceptions

HotPants makes no use of C++exceptions. Since the instrument is fairly static, relatively short-lived, and does not allow for low-level user interactions, there are no places where

[24] As discussed above
[25] See Section 3.3.1

they would be of sufficient benefit; indeed, they could only be of use for revealing bugs during development. Exceptions would also reduce performance, and are undesirable for this reason alone.

3.3.3 Testing

We would like now to describe the testing procedures undertaken during this project. An instrument such as HotPants is not a particularly convenient case for software testing since it requires operation in a host, removing the possibility of having absolute control over the testing process. Nevertheless, testing is as necessary as ever, especially since a defective instrument poses a real danger of physical equipment damage. The back-reference non-default constructor requirements enforced by the `Component` interface, as described in Section 3.2.2, would seem to be a hindrance to testing. However, in practice this is not so. Of course a `HotPantsCore` object must be instantiated first, but a testing object can also be created in as little as three lines:

```
HotPantsCore testCore;              // initialisation not req'd
Filter1 testFilter;
testFilter.initialise(44100.0); // CD-quality
```

`Filter1` will, of course, refer to the relevant parameters of the `testCore`'s `ParameterArray`. This is, however, sufficient for testing.

In line with our plans discussed in Section 2.4.1, a separation of unit development and integration was used extensively during development, with integration testing taking place in the `AU Lab` application with frequency analyser plugins taking the place of audio output. Filters, in particular, were an area where care needed to be taken, since incorrect coefficient calculation could lead directly to overloaded signals. A possible alternative could be a failsafe unit, consisting of a short delay queue, similar to the `MonoDelay`; the unit could inspect the state of the queue for a specified number of consecutive samples exceeding `kAmpMax` and send a call to `HotPantsCore` to disable rendering. This would certainly prevent the risks highlighted in Section 2.4.1, and may have saved time if we had planned it from the start.

Testing suite

Towards the end of the integration stage, we developed the command-line testing class `UnitTest`, with the function `run()`. It is intended to take the place of a host as best possible, and simulate normal instrument behaviour as best possible for testing purposes:

1. Create and initialise an instrument for an arbitrary sampling rate and time the process
2. Test limits—for example, no MIDI operations before first instrument initilisation
3. Test instrument reinitialisation
4. `TEST_NOTEON`—running and timing a large number of `startNote()` and `stopNote()` operations with random notes.
5. Buffer output test with error checks for zero output as well as overloaded output
6. `TEST_RENDER`—runs a large number of render cycles and times the output

7. `SYNTH_SIMULATION`—which attempts to simulate normal operation by running a smaller number of rendering cycles with random `startNote()`, `stopNote()` and parameter changes in each cycle. Since the behaviour of this test is entirely random, error messages should be expected since it will attempt to start notes with illegal MIDI note numbers, stop notes that are not active and set illegal parameters

Testing options can be modified in `UnitTest.hpp`. All timing operations use the C standard library `clock()` function and data from these timings is used to quantify performance tests in Section 3.3.4.

Valgrind Tools

Whilst HotPants makes very little use of dynamic memory management, there is a great deal of pointer use in the `VoiceManager`. As such, during the unit testing of this class the Valgrind suite *Memcheck* tool[26] was used extensively to ensure no memory was being leaked. Use of this tool, along with the `TEST_NOTEON` feature of the unit testing suite allowed us to protect entirely against pointer manipulation errors and leaks.

3.3.4 Performance and optimisation

In Section 2.2 we highlighted instrument performance as a key focus. In this section we discuss implementation details that we have taken to deliver on this goal. First we will outline a number of design aspects that have been included with the intention of improving real-time performance. Finally we will discuss our use of profiling tools and compiler code optimisation.

ModuleCache

The method `Voice::attack()` is a potential performance bottle-neck; preparing the four sub-modules, `OscModule`, `FilterModule`, `AmpModule` and `ModModule`, for a new note is a potentially expensive operation. However, any latency in the note-on stage will be especially perceptible to the user. The module caching system is designed to address this. Before describing its operation, it is necessary to understand that there are two kinds of parameter changes within the instrument:

1. Simple values that can be incorporated directly into a rendering cycle
2. Parameters linked to more complex functionality that require the recalculation of locally stored data

An example of type 1 would be the global volume parameter `kParam_GlobalVolume`, which is read from the parameter table at the start of `StereoVolume::renderStereo()` and then used as a multiplier for each sample:

```
ParamValue vol =
  core.getParamManager().getValue(kParam_GlobalVolume);

for(BlockSize i = 0; i < s; ++i) {
```

[26] `http://valgrind.org/info/tools.html#memcheck`

```
    leftBuffer[i] *= vol;
    rightBuffer[i] *= vol;
  }
```

In contrast, as an example of type 2, any change to filter settings requires a complete recalculation of local coefficients. The `ModuleCache` is designed to eleviate the potential for lost performance that might result from constantly recalculating this data unnecessarily; if we assume that only a small minority of the instrument's parameters are modified by the user prior to any rendering cycle then we would like to reuse previously calculated data.

Operation of the `ModModule` involves four separate parts:

1. A `ModuleCache` object, containing its own `FilterModule`, `AmpModule` and `ModModule`—the cached modules themselves
2. The `ParameterManager`'s `ParameterArray`, containing a `moduleMap[]`
3. The `eParamModule` type, and `module` field of the parameter `settings[]` array
4. The `ParameterManager`'s `moduleDirty()` and `moduleSetClean()` functions

If the reader refers to `Include/ParamDefaults.hpp`, the default settings array, and finds the *Module* column, they will find entries from the enum `eParamModule`, defined in `Include/Enums.hpp`:

```
enum eParamModule {
  eModule_None = 0,
  eModule_Filt,
  eModule_Amp,
  eModule_Mod,
  eNumOfModules
};
```

The entries correspond to the three `cacheableModules`:[27] `FilterModule`, `AmpModule` and `ModModule`. This data allows us to relate instrument parameters to these modules. For example, the settings array entry for `kParam_AmpAttack` has `eModule_Amp`, and for `kParam_Filt1Cutoff` it is `eModule_Filt`. When a parameter is modified in the `ParameterManager` functions `setValue()` and `setMod()`, the function `checkAndSetFlagForModule(eParamModule g)` is called:

```
checkAndSetFlagForModule(paramMap.settings[p].module);
...
if(g != eModule_None)
  paramMap.moduleMap[g].dirty = true;
```

The `module` entry for the parameter is checked; if the parameter belongs to a module *and* is a type 2—it is not `eModule_None`—then the parameter manager will set that module's `dirty` flag to `true`. This flag indicates that the module will require refreshing on the next note-on event. It should be noted that the parameter array has been carefully set up, and

[27] See Section 3.2.2

any module parameters such as `kParam_FiltConnect` and `kParam_LFO1Amount` that are set to `eModule_None` are type 1 parameters and do not benefit the module caching system.

The next step in the system involves a particular synthesiser voice `attack()` call, and the `ModuleCache` object. The following example is for the amplifier module and demonstrates the process:

```
if(moduleIsDirty(eModule_Amp)) {
  ampModule.refresh();                    // dirty - refresh
  cache.setAmpModule(ampModule);          // cache module
} else
  ampModule = cache.getAmpModule();       // clean - reuse
```

The module's `dirty` flag is inspected with a call to `moduleIsDirty()`. If the module is *clean*, then the cached copy of the module can be reused with a call to relevant `ModuleCache` getter. On the other hand, if the module is *dirty*, then it will be refreshed with a call to `refresh()` and the cached version will be replaced with a call to the `ModuleCache` setter.

Performance tests of `ModulateCache` using `UnitTest`[28] revealed relative gains of around 4% for note-on operations[29].

The DynamicComponent interface

If `ModuleCache` is designed to improve note-on performance, then `DynamicComponent` is aimed at parameter changes and the normal rendering cycle. The `DynamicComponent` interface applies to all internal instrument components that respond to instrument parameter changes in real-time. We have made the choice, for example, the envelopes are *not* dynamic components; if the user changes an envelope setting during a note they will have to strike another note to hear the effect of their changes. Filters, on the other hand can recalculate their coefficients at the start of any rendering cycle to respond to parameter changes, which is equivalent to real-time response. In order to deliver this functionality, all dynamic components must define a `refresh(eRefreshType)` method that checks any applicable instrument parameters and recalculates local settings accordingly. The enumerated argument `eRefreshType` is of importance here, since dynamic components can be forced to calculate their settings when first initialised—or re-initialised with a new sampling rate—but check for real changes in parameters during a normal rendering cycle:

```
refresh(kFORCE_REFRESH);          // initialisation
refresh(kCHECK_FOR_UPDATES);      // normal rendering
```

These `refresh()` methods make use of the `ParameterManager::paramModified()` function to check whether any related parameters have been changed. `paramModified()` merely returns the state of the `ParameterArray`'s `modified` field for a given parameter; `ParameterManager` mutators such as `setValue()` and `setMod()` modify this flag when real parameter changes are made. `refresh()` methods can thus respond appropriately; it parameters are unchanged then they may reuse locally stored data, otherwise they recalculate.

[28] See Section 3.3.3

[29] 100,000 note-on/off operations of 36 notes each: module caching enabled, 7.76×10^{-6}s/note; disabled, 8.03×10^{-6}s/note

The functionality of the `DynamicComponent` refreshing system can be disabled by commenting out the `REFRESH_CHECK` compilation option in `Compilation.hpp`, forcing `paramModified()` to always return `true`. Testing of this functionality on our development system using `UnitTest` did, however, reveal only modest gains: an average improvement of 2%[30]. This is certainly surprising, and we had certainly hoped for a more dramatic difference. A possible explanation could be that the calculation of filter coefficients is considerably cheaper than it might seem to be.

Envelope stage caching

`ADSRcurve`[31] provides the conditional compilation feature `CACHE_ENV`, which is a performance enhancement similar to the functioning of the `ModuleCache`; in short, it stores the last set of settings and parameters, only updating the envelope stages that have had their lengths altered by the user. Whilst functional, this option has been disabled due to being slightly slower than full calculation with `ModuleCache` enabled and static parameters[32]. It has been largely been superseded in operation by the `ModuleCache`, which provides similar functionality for a range of modules, as discussed above.

Multithreading

An additional avenue explored was the use of multithreading, at least in the rendering stage, to see if performance improvements might be attained. The popular *Boost* C^{++} libraries provide the *Boost thread*[33] cross-platform multithreading system, expected to form the core of language concurrency support in the upcoming *C++0x* standard revision. The most obvious place in the instrument to implement threading support was the `VoiceManager::renderAllVoices()` method; since instrument voices are entirely independent it seemed entirely appropriate that they could render on separate threads. Access to shared data is still required, however, and the `ParameterManager` and wavetables must be serialised by way of the `boost::mutex` with `boost::mutex::scoped_lock` locks on relevant shared data access and mutation functions such as `getValue()`, `setValue()`, `WaveTableBase::getFrameAtIndex()` and so forth.

Whilst this multithreading support has been completed, and can be enabled with the `BOOST`, `THREAD-SAFE` and `MULTITHREAD` compilation flags, as well as linking with the Boost library, profiling revealed surprisingly poor results: an optimised normal build, running 10,000 rendering cycles with a block size of 64 saw a time of $1.69 \times 10^{-4} s/cycle$; with multithreading enabled this dropped considerably to $7.04 \times 10^{-3} s/cycle$, a loss of more than $4,000\%$. We can conclude from this that the overhead required for thread creation and dispatching removes any benefit that might be seen with our current strategy, especially on no more than a dual-core system. A more suitable approach would probably be to create a thread for each `Voice` in `VoiceManager::initialise()`, maintaining these threads for the life of the instrument. This would be vastly more efficient, although again it is doubtful whether real gains would be seen without more available processing cores; hardly the kind of system that we are targeting, at least at present. Nevertheless, the instrument's

[30] Rendering test run of 20,000 cycles, with a block size of 64 frames, 36 active voices: refresh-checking enabled, $6.04 \times 10^{-4} s/cycle$; disabled, $6.11 \times 10^{-4} s/cycle$

[31] See Section 3.2.7

[32] 10,000 note-on/off cycles, 36 voices, enabled: $7.74 \times 10^{-6} s/note$, disabled: $7.55 \times 10^{-6} s/note$

[33] `http://www.boost.org/doc/libs/1_43_0/doc/html/thread.html`

shared data has been serialised and a more efficient system should only require work in the `VoiceManager`. There is plenty of scope here for the future.

Profiling using GPROF

By using the GNU `GPROF` tool we hoped to find performance bottlenecks in the instrument, to focus our efforts on optimisation. Results were certainly enlightening and productive. Whilst `GPROF` highlighted functions in which small coding adjustments could be used to gain some benefit, most of all it demonstrated that altogether *too many* function calls were being made in normal operation of the instrument. These were, then, the primary changes that we made, more structural rather than in terms of individual statements themselves:

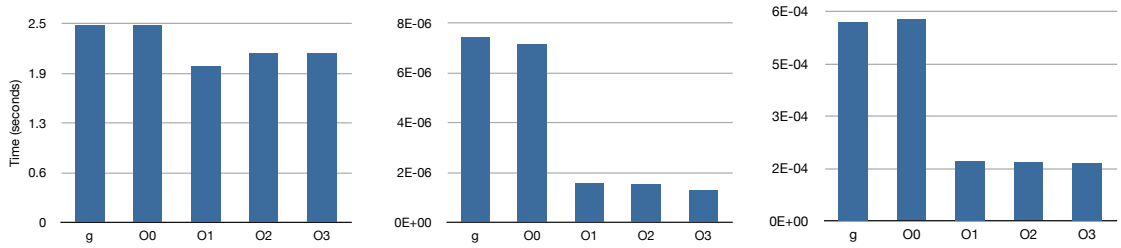
- Profiling revealed that our early instrument versions were hampered by excessive function calls for each component during the rendering cycle. This was as a result of a fundamental design that operated sample-by-sample; for an output buffer size of n frames, with x instrument components, this would require a minimum of nx function calls for each cycle. A change to the component-by-component buffered rendering approach now used reduced this value to just x calls, one for each component. This important change of strategy alone transformed the instrument from one with lacking, to one with satisfactory performance.

- Wavetable reading was also analysed as a significant hot-spot for excessive function calls. This led to the realisation that unnecessary reads were taking place; the wavetable increment value `indexInc` is usually a value less than 1.0; this means that accessing the wavetable is not usually necessary and it is better to store the last sample read from the table and last integer position, `lastTableReadPos`, and not read the table until the floating-point table position, `indexFP` has increased by a whole number:

  if $indexFP - lastTableReadPos \geq 1.0$ **then**
  read new value from wavetable
  $lastTableReadPos = \lfloor indexFP \rfloor$
  else
  return stored sample
  end if

- An effort was made to remove super-class virtual function calls from oscillator and LFO rendering. The intention is that for an one oscillator, in any one rendering cycle, there should be a maximum of two function calls; one to `renderMono()`, and one to `refresh()`. The heavily-used `AudioOsc::advanceFrames()` was improved by making it an `inline` function. Furthermore, two macros, `_MACRO_CALC_GLOBAL_TUNING_` and `_MACRO_GET_AMP_WAVE_` have been used to substitute super-class calls in `AudioOsc` and `OscillatorBase`, respectively

- A number of frequently used `ParameterManager` functions, such as `setMod()` and `getParamMap()` have been inlined to improve performance.



(a) Object creation (b) Note on/off (c) Rendering

Figure 3.16: Compiler optimisation comparison

Compiler optimisation

Having worked to design an efficient instrument, and developed improvements for high-use areas, we finally turn our attention to the use of compiler optimisations to generate efficient code. The GNU C++ Compiler offers the standard `-Ox` optimisation options, and we have profiled `O0`–`O3`, including a debugging build with the `-g` flag only, using the `TEST_RENDER` and `TEST_NOTEON` test suite options. The timing results of the tests can be found in Figure 3.16, where smaller values indicate faster performance. It is clear from the charts that compiler optimisations have a dramatic effect on performance with any of the `Ox` options, with almost a 350% improvement between `O0` and `O3`. It is clear that `O2` or `O3` are extremely attractive for a release build. We could also experiment further with individual optimisation flags, however since the gains from the default options are so significant we imagine that additional gains would be negligible at best.

3.4 Integration and portability

A key goal of this project has been to ensure that the resulting instrument is free and portable to as wide a range of popular software and hardware platforms as possible, and we believe we have been successful in this respect. The final product should be considered as having two parts:

1. The instrument *core*, as detailed throughout Section 3.2, and encapsulated by the `HotPantsCore` class. This can be considered the body of the project
2. The provided example of instrument plug-in integration, in this case for OS X CoreAudio. In terms of implementation, this element is secondary, although necessary in order to use the instrument as intended

In this section we discuss the steps taken to ensure portability, the process of implementing the AU plug-in, and the steps that would be necessary to port the instrument to another platform.

3.4.1 Instrument core

The majority of the instrument core has been implemented using standard C^{++} and the *GNU C++ Compiler*, version 4.2.1. As such, we feel confident that the majority of the instrument is compliant with the standard C++98. We have tried to use standard data

types, and there is little memory management and complex I/O is intended to be dealt with by the external audio framework. There is some use of fixed-length data types, such as `uint32_t` for sizes and `int8_t` from the non-standard `stdint.h`, which may pose some portability issues but which could be fixed with additional compilation switches. In terms of support for 32-bit and 64-bit architecture, at present HotPants can be built as a 64-bit binary with no additional modification, which is a promising indicator of the instrument's portability.

Parameters and presets

In order to ensure as wide a compatibility as possible, the core HotPants parameter definitions use the most generic data types as possible, requiring that they then be converted to more convenient native platform types. This is especially true with names, which have been defined as preprocessor strings in `Include/Strings.hpp`. Our second major consideration is support for built-in factory preset instrument patches. Most audio frameworks provide a native file format for plug-in presets; in the case of CoreAudio it is `.aupreset`. However, this is little more than an XML wrapper for a binary dump, and we would like our core support to be as generic as possible. To do this, we provide preset data as part of the instrument core itself, and which can be adapted for the external platform. The types `kPreset` and `kPresetVal` define a nested structure, and two example presets are defined in `Include/Presets.hpp`, an array for each preset, giving a value to each of the instrument's sixty-seven parameters. Needless to say this is a rather laborious process without taking advantage of additional helper tools. The framework, however, it in place and any number of presets could be designed and included with the instrument core.

3.4.2 AudioUnit implementation

In order to realise the instrument as a plug-in for a audio/MIDI host, we have implemented a CoreAudio AudioUnit wrapper for `HotPantsCore`. The four main source files for this integration can be found in `./AUSource/`. These files are built upon standard templates provided by the CoreAudio SDK, which is required for compilation, although in the final version almost all code in these files is original, providing little more than a wrapper for the contained `HotPantsCore` object. The standard procedure for creating an AU synthesiser is to sub-class `AUMonotimbralInstrumentBase`, although most relevant functionality comes from its super-class, `AUInstrumentBase`. The important input and output functions are:

- `RealTimeStartNote()`—note on.
- `RealTimeStopNote()`–note off
- `HandleControlChange()`—MIDI control changes
- `HandlePitchWheel()`—MIDI pitch wheel

These methods are very simple, passing their arguments on to relevant methods of `HotPantsCore`. The plug-in `render()` method is similarly straightforward, merely preparing appropriately for the `HotPantsCore` render method. Most of the involved work, then, relates to the handling of instrument parameters and presets and outlined in the previous section. The *CoreFoundation* framework upon which CoreAudio is built has support for string objects and dynamic arrays, and our generic types are mapped to these framework types in `ParamMapping.h`. Given enough understanding of the platform's APIs, this

was not a great challenge. The function `checkAUmapping()` is also provided to check any human-errors made in this mapping. As such, the AU plug-in is able to parse our generic data, and provide all of the necessary functionality to create a plug-in implementation of the instrument.

3.4.3 Porting

Having discussed the details of implementing the instrument in CoreAudio, we now present a more generic explanation of how the instrument could be ported to another platform.

1. The instrument core can be built and tested separately, using the `UnitTest` suite on the new platform's console environment. This will reveal any native compiler-related issues. If the suite can be run successfully then it is safe to proceed with the plug-in
2. Instrument parameter strings must be converted to an applicable native format and mapped to the correct parameters
3. Instrument presets can be optionally implemented in line with the platform's own approach to presets
4. The plug-in object itself should contain a `HotPantsCore` object, and follow the initialisation process discussed in Section 3.2
5. MIDI input messages should be passed to the instrument as described in Section 3.2, as well as the CoreAudio example above

Possible issues could include:

1. The use of `stdint.h`, noted above. This could be fixed with conditional compilation `define`s, as part of the shared instrument core
2. The rendering process, which could conceivably differ from the buffered approach we have used up to this point. To our knowledge, however, a buffered output is the most common approach

In light of the small size of our AU implementation, we would not expect porting the instrument to be a lengthy task. Indeed, we would expect that studying the new framework would take up the vast majority of required time, and implementation very little, perhaps measuring in hours.

3.5 Conclusion

We have now discussed our implementation of the instrument designed and outlined in Chapter 2. In the next section we will move on to talk more subjectively about the degree to which we feel that the final project lives up to our plans and objectives.

Chapter 4

Evaluation

4.1 Introduction

In this concluding chapter, we take the opportunity to express our opinions on the degree to which we feel we have been successful in implementing the instrument we originally planned. In line with our list of requirements from Section 2.2, we wish to discuss the instrument in terms of those four areas:

1. Feature set
2. Sound quality
3. System performance
4. Performer-oriented features

4.2 Discussion

Feature set

As described in Section 2.3, our primary focus for the instrument was for a subtractive synthesiser, along the lines of the ES2 or Albino instruments presented in our Section 1.5 market analysis. We certainly have been able to do this; in particular by not attempting to implement our own filters. Robert Bristow-Johnson's designs, offer a range of filters actually *wider* than those of ES2 and Albino, although lacking some of their *warm* audio quality. Other features such as pulse width modulation make the instrument suitable for synthesiser string sounds, and it can certainly create a wide variety of typical vintage synthesiser effect sounds.

We might like to have implemented a wider range of on-board effects, however built-in effects are not considered an essential part of most synthesisers. Plenty of separate, dedicated effects units are available, although it was enjoyable to investigate this aspect since these modules do not have to fit into as complex a system hierarchy as central components. The existing oscillator system could be easily extended to support hand-crafted waveforms loaded from disk, and this would certainly be an interesting feature to experiment with, although not one we have yet seriously investigated. Our plan would be to *draw* the waveforms in a bitmap art package, and convert the images to some raw data file using helper utilities. It should not pose a great challenge.

Overall, if there is one area in which the instrument's feature set falls short, then it is certainly in the area of modulation. HotPants's modulation system is comparable to the ES2, but lacks the flexibility of Albino's matrix design. As mentioned in Section 2.3 we were to a large extent restricted by the instrument's generic user interface, and eventually this did require us to draw a line in the sand; the completed instrument has sixty-seven parameters, far too many to be listed on most computer screens, and in fact renders the instrument difficult to use in hosts that don't allow the parameter list to be scrolled. Of course in an extension of the project a graphical user interface would be first priority, and this would certainly improve the efficiency of the existing feature set, allow it to be extended, and allow the creation of a matrix-base modulation section.

Sound quality

As noted above, the use of the RBJ filters permitted us better sound quality for a subtractive synthesiser than might otherwise have been possible. A poor sounding, or inflexible filter greatly diminishes the usefulness of such an instrument. In other areas we feel less confident: the oscillator modulations, FM, RM and AM function correctly, but seem somewhat lacking, and don't appear to generate the same richness of tones generated by other instruments. The ES2 employs our Chowning FM oscillator routings, but appears to create richer, more interesting tones. Likewise, Albino has oscillator AM with better quality. It is possible the our aliasing issues—noted below—might be interfering with their proper operation somehow.

So, as noted in Section 3.3.1, issues related to aliasing proved our greatest challenge with regard to the quality of the completed instrument. Despite great efforts to combat the issue, the problem persists and is no doubt responsible for reducing the quality of the instrument's output. Particularly when multiple instantiations of the instrument are run in a sequencing host application, the mixing of these undesired frequencies *muddies*—to use audio mixing terminology—the sound. It seems that our current wavetable implementation is naïvely simplistic and this is certainly an issue that we would want to approach entirely differently if we were to commence another synthesiser project. Papers by Stilson and Smith [20], as well as Eli Brandt[1] suggest approaches that might be useful. The use of octave wavetables would be another less theoretical solution. In truth a reworking of this project would evidently require a concerted investment into underlying signal processing theory in order to produce signal quality comparable with professional instruments. We feel that we have merely scratched the surface at this time. That said, we have noted design and implementation of the instrument to be our focus, and is these areas we feel we have been much more successful.

Performance

Instrument performance is one area where we feel that careful investment of time and effort has paid excellent dividends. As noted in Section 3.3.4, the change to fully buffered internal rendering was the single greatest design change that allowed this to happen. Without it, we would have seen performance sink further as we added more components. Instead, performance is quite satisfactory; whilst it may not be the most efficient design possible, it is certainly no challenge for a modest modern computer to support numerous instantiations of the instrument with without taxing the CPU. By way of example, an arrangement with

[1] `http://www.cs.cmu.edu/~eli/papers/icmc01-hardsync.pdf`

seven instruments takes between 20 and 30%, which we feel is adequate. Somewhere in the region of twenty to thirty instantiations should be possible as a current maximum. This is certainly adequate for most musical arrangements and well in line with our stated goal.

With regard to scope for future developed, as we noted in Section 3.3.4, an initial multithreaded implementation has already been completed, although has proved to be unsuitable in its current form. The instrument is already thread safe, but the approach to thread creation and dispatch needs to be properly addressed. We certainly would like to experiment with alternative strategies such as those discussed in Section 3.3.4, at the very least those that offered equal performance to a non-threaded build. This would then create a design that could properly benefit from the increasingly common use of multi-core CPUs in consumer-level hardware, and is not yet a standard feature in other software synthesisers. It would give HotPants something that sets it apart where its other features cannot.

Performer-oriented features

The key areas in which we have attempted to provide this functionality are in the amplifier, with its velocity-sensitivity, as well as in the modulation system, which allows note velocity to be assigned to a modulation target, as well as providing support for the modulation and pitch wheels[2]. These features are comparable to those of vintage synthesisers, as well as more modern models and we are happy that we have this functionality covered. As noted above, the modulation system itself currently leaves something to be desired, and perhaps does not quite behave in the same way as other instrument. This is not to mention the lack of a graphical user interface, which would no doubt be off-putting to all but the most technically-inclined user. We know from our own experience that attractive user interfaces considerable increase the appeal of an instrument, as well as ease of use, should the interface be as functional as it is pretty.

Also, whilst not directly related to the perceived tactility of using the instrument, we feel that some of the instrument's short-comings in the area of sound quality will prove to work against the instrument's appeal as a serious instrument. It would seem that this area is a culmination of the instrument's successes and failures in other areas, an improvements would be seen by addressing concerns in those areas.

4.3 Conclusion

This, therefore, brings us to the end of this project. It has certainly been an educational experience. As expected, its primary benefit has been in thoroughly versing us in the capabilities of the C^{++} programming language, and the kind of design approaches that are most suitable for it, perhaps in contrast with other object-oriented languages. The opportunity to investigate the implementation of a multithreaded design was also particularly enlightening. Whilst gains in our knowledge of signal processing have been slight in comparison we have, however, become far more aware of the kind of approaches taken in real-time data streaming of any kind, no less digital audio. There is much in the final product that we feel proud of; its feature set is strong, as is its performance, and the instrument is extremely stable. As part of a team we already feel ready to take part in a project to develop an instrument that can compete with those available from major software providers.

[2]Section 3.2.1, and Figure 3.2

Appendix A

Source code guide

The following table details the source code directory structure and outlines the contents of each file. Please see Section 3.2 for a discussion of system architecture, and Appendix B for a system diagram.

	Public header files	
./Include	Compilation.hpp	Compile-time program options; use of Boost library, multi-threading, module caching enable/disable, parameter smoothing, debug options, etc.
	ParamDefaults.hpp	The full static parameter defaults definition map. Defines default, minimum, maximum values, units and other configuration flags which aren't intended for external modification.
	Enums.hpp	Parameter enumerations; the main eParameter enumeration, and various others defining available waveforms, parameter units and flags, instrument components and modulation targets.
	Presets.hpp	Built-in factory preset definitions.
	ParamMaps.hpp	Constant maps used to link components and modulation targets to their relative instrument parameters.
	Strings.hpp	Preprocessor strings naming parameters and menu items.
	ParamTypes.hpp	struct definitions used in `ParamMaps.hpp`
	TypesConstants.hpp	Instrument-wide typedefs and constants.

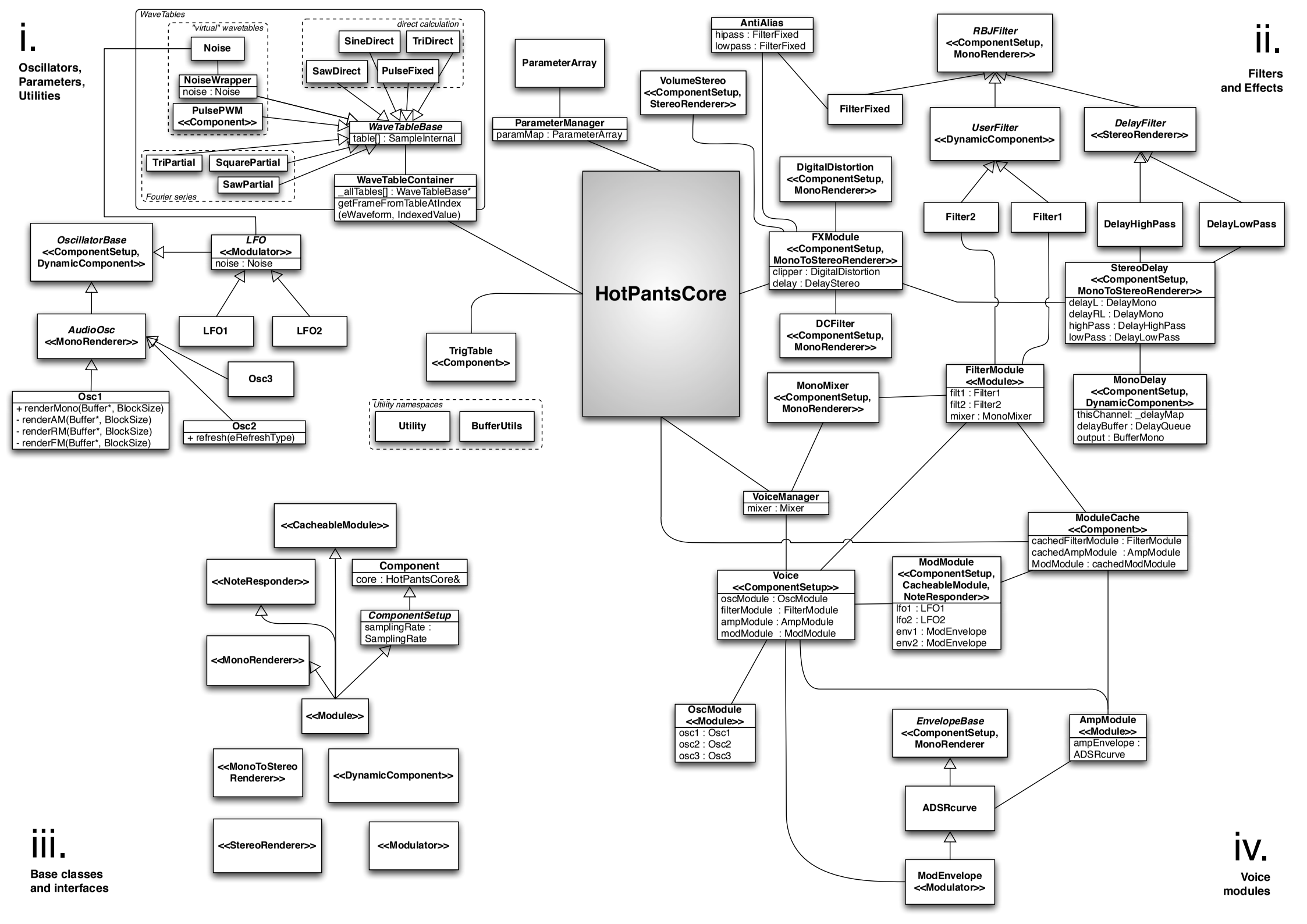
./Src/Core	**Instrument core classes**	
	HotPantsCore.*	The core, public instrument object.
	ParameterArray.*	Contains an instrument parameter array and group map.
	ParameterManager.*	Manages a ParameterArray, with associated parameter mutation and access functions.
	VoiceManager.*	Maintains containers of instrument voices, and provides them to a `HotPantsCore` object as required.
./Src/Effect	**Effect units**	
	AllEffects.hpp	Convenience header for all available effect units.
	DelayMono.*	Mono delay effect.
	DelayStereo.*	Stereo delay, comprised of two `MonoDelay` and two filters.
	DigitalDistortion.*	Hard-clipper and fold-over disortion/-gain unit.
	FXModule.*	Module containing all effect units connected in fixed series.
	VolumeStereo.*	Stereo master gain control.
./Src/Envelope	**Envelope classes**	
	ADSRcuve.*	Logarithmic ADSR envelope.
	EnvelopeBase.*	Abstract base for envelopes.
	EnvelopeTypes.hpp	Envelope data types.
./Src/Filter	**Filters**	
	AntiAlias.*	A high-pass, low-pass combination fixed anti-aliasing filter.
	DCFilter.*	Direct current noise filter.
	Filters.*	Concrete classes for built-in and fixed filters.
	RBJFilter.*	Implementation of Robert Bristow-Johnson's Audio EQ Cook-book filters.
./Src/Generator	**Signal generators**	
	Noise.hpp	Simple `RAND()` random number noise generator.
./Src/Interfaces	**Interfaces**	
	Interfaces.hpp	Component base classes and interfaces.
./Src/Mixer	**Mixer**	
	Mixer.*	Mono `Voice` buffer mixer unit.
./Src/Modulator	**Modulation sources**	
	LFO.*	`OscillatorBase` sub-class providing low-frequency oscillators.
	ModEnvelope.*	`ADSRcurve` sub-class providing an modulation source envelope.

./Src/Module	**`Voice` component modules**	
	AmpModule.*	Master amplifier and note velocity-scaling module.
	FilterModule.*	Filters 1 & 2.
	ModModule.*	Modulation sources module.
	ModuleCache.*	The module caching container object.
	OscModule.*	Oscillator 1–3 container module.
	Voice.*	Instrument voice encapsulation class. Containing oscillator, filter, amplifier and modulation modules.
	VoiceModules.hpp	Convenience header of all `Voice` modules.
./Src/Oscillator	**Oscillator classes**	
	AudioOsc.*	Sub-classes of `OscillatorBase` to provide the audible-range oscillators 1–3.
	OscillatorBase.*	Base class for all oscillator types; eg. `AudioOsc` and `LFO`.
./Src/Utility	**Instrument convenience utilities**	
	Utility.*	Utility classes and namespaces.
./Src/WaveTable	**Wavetables and container**	
	AllWaveTables.hpp	Convenience header of all available real and *virtual wavetables.*
	WaveTableBase.hpp	Base class for wavetables.
	WaveTableContainer.*	Container class for all available wavetable types.
	/Direct/*	Direct-calculated wavetables. Mostly for use by LFOs.
	/Fourier/*	Fourier series calculated wavetables, intended as audible oscillator signal sources.
	/Virtual/*	Virtual, wrapped non-true wavetable classes—`NoiseWrapper`, `PulsePWM`.
./Test	**Unit testing**	
	UnitTest.*	Unit testing suite and `main()` function.
./AUSource	**AudioUnit plug-in source**	
	HotPantsAU.*	Main plug-in instrument class.
	HotPantsVersion.h	Instrument version header.
	ParamMapping.h	AudioUnit-specific const types for converting plain instrument parameters to CoreAudio compatibility structure.

Appendix B

System diagram

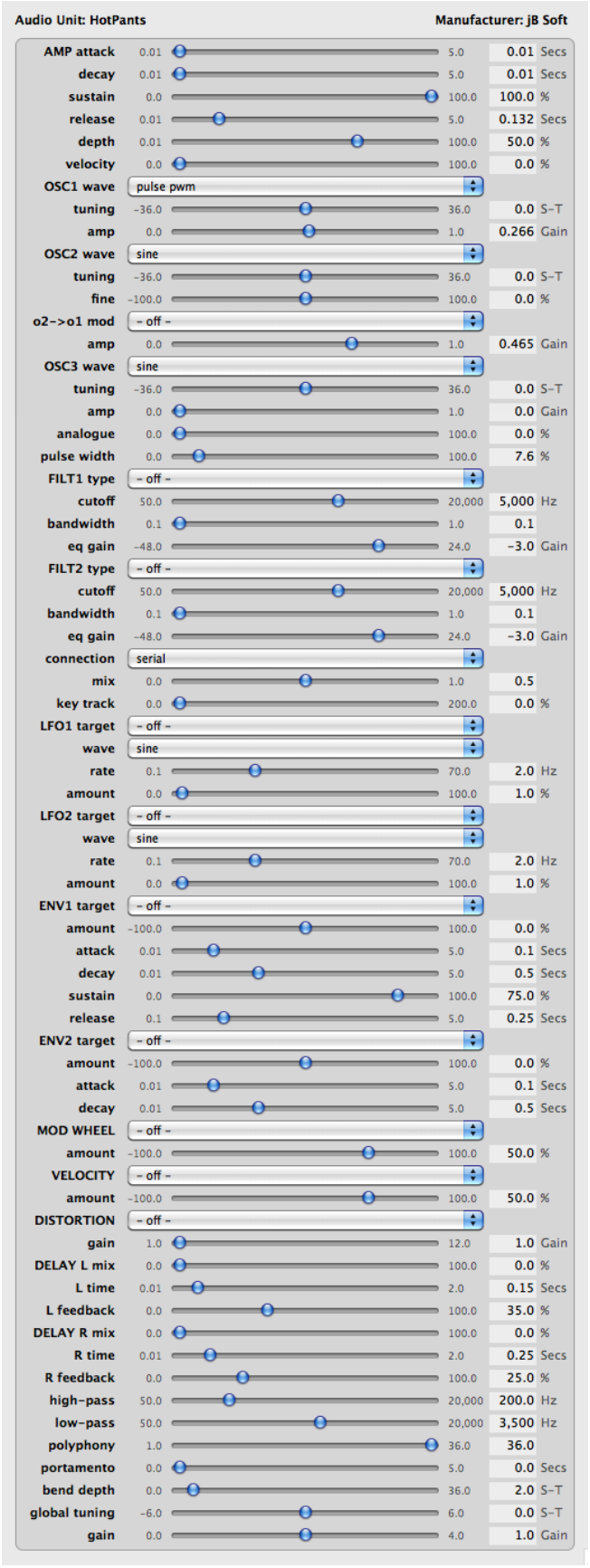
A diagram of the complete instrument system follows. Please see Section 3.2 for a full discussion, and Appendix A for a guide to instrument source code files.


i.
Oscillators, Parameters, Utilities
WaveTables
"virtual" wavetables
direct calculation
Noise
SineDirect
TriDirect
SawDirect
PulseFixed
NoiseWrapper
noise : Noise
PulsePWM
<<Component>>
WaveTableBase
table[] : SampleInternal
TriPartial
SquarePartial
SawPartial
Fourier series
WaveTableContainer
_allTables[] : WaveTableBase*
getFrameFromTableAtIndex
(eWaveform, IndexedValue)
ParameterArray
ParameterManager
paramMap : ParameterArray
VolumeStereo
<<ComponentSetup,
StereoRenderer>>
AntiAlias
hipass : FilterFixed
lowpass : FilterFixed
FilterFixed
RBJFilter
<<ComponentSetup,
MonoRenderer>>
UserFilter
<<DynamicComponent>>
DelayFilter
<<StereoRenderer>>
DigitalDistortion
<<ComponentSetup,
MonoRenderer>>
Filter2
Filter1
DelayHighPass
DelayLowPass
OscillatorBase
<<ComponentSetup,
DynamicComponent>>
LFO
<<Modulator>>
noise : Noise
HotPantsCore
FXModule
<<ComponentSetup,
MonoToStereoRenderer>>
clipper : DigitalDistortion
delay : DelayStereo
StereoDelay
<<ComponentSetup,
MonoToStereoRenderer>>
delayL : DelayMono
delayRL : DelayMono
highPass : DelayHighPass
lowPass : DelayLowPass
AudioOsc
<<MonoRenderer>>
LFO1
LFO2
TrigTable
<<Component>>
DCFilter
<<ComponentSetup,
MonoRenderer>>
Osc3
Osc1
+ renderMono(Buffer*, BlockSize)
- renderAM(Buffer*, BlockSize)
- renderRM(Buffer*, BlockSize)
- renderFM(Buffer*, BlockSize)
Osc2
+ refresh(eRefreshType)
Utility namespaces
Utility
BufferUtils
MonoMixer
<<ComponentSetup,
MonoRenderer>>
FilterModule
<<Module>>
filt1 : Filter1
filt2 : Filter2
mixer : MonoMixer
MonoDelay
<<ComponentSetup,
DynamicComponent>>
thisChannel: _delayMap
delayBuffer : DelayQueue
output : BufferMono
ii.
Filters and Effects
VoiceManager
mixer : Mixer
ModuleCache
<<Component>>
cachedFilterModule : FilterModule
cachedAmpModule : AmpModule
ModModule : cachedModModule
Voice
<<ComponentSetup>>
oscModule : OscModule
filterModule : FilterModule
ampModule : AmpModule
modModule : ModModule
ModModule
<<ComponentSetup,
CacheableModule,
NoteResponder>>
lfo1 : LFO1
lfo2 : LFO2
env1 : ModEnvelope
env2 : ModEnvelope
<<CacheableModule>>
<<NoteResponder>>
Component
core : HotPantsCore&
ComponentSetup
samplingRate :
SamplingRate
<<MonoRenderer>>
<<Module>>
<<MonoToStereo
Renderer>>
<<DynamicComponent>>
<<StereoRenderer>>
<<Modulator>>
iii.
Base classes and interfaces
OscModule
<<Module>>
osc1 : Osc1
osc2 : Osc2
osc3 : Osc3
EnvelopeBase
<<ComponentSetup,
MonoRenderer
AmpModule
<<Module>>
ampEnvelope :
ADSRcurve
ADSRcurve
ModEnvelope
<<Modulator>>
iv.
Voice modules

Appendix C

User interface

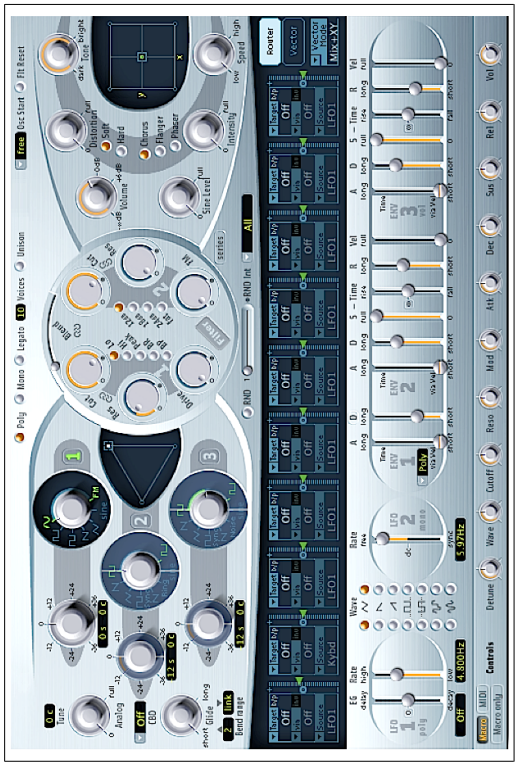
We present here an image of the completed instrument user interface, a CoreAudio generated generic view.


Audio Unit: HotPants
Manufacturer: jB Soft
AMP attack 0.01 5.0 0.01 Secs
decay 0.01 5.0 0.01 Secs
sustain 0.0 100.0 100.0 %
release 0.01 5.0 0.132 Secs
depth 0.01 100.0 50.0 %
velocity 0.0 100.0 0.0 %
OSC1 wave pulse pwm
tuning -36.0 36.0 0.0 S-T
amp 0.0 1.0 0.266 Gain
OSC2 wave sine
tuning -36.0 36.0 0.0 S-T
fine -100.0 100.0 0.0 %
o2->o1 mod - off -
amp 0.0 1.0 0.465 Gain
OSC3 wave sine
tuning -36.0 36.0 0.0 S-T
amp 0.0 1.0 0.0 Gain
analogue 0.0 100.0 0.0 %
pulse width 0.0 100.0 7.6 %
FILT1 type - off -
cutoff 50.0 20,000 5,000 Hz
bandwidth 0.1 1.0 0.1
eq gain -48.0 24.0 -3.0 Gain
FILT2 type - off -
cutoff 50.0 20,000 5,000 Hz
bandwidth 0.1 1.0 0.1
eq gain -48.0 24.0 -3.0 Gain
connection serial
mix 0.0 1.0 0.5
key track 0.0 200.0 0.0 %
LFO1 target - off -
wave sine
rate 0.1 70.0 2.0 Hz
amount 0.0 100.0 1.0 %
LFO2 target - off -
wave sine
rate 0.1 70.0 2.0 Hz
amount 0.0 100.0 1.0 %
ENV1 target - off -
amount -100.0 100.0 0.0 %
attack 0.01 5.0 0.1 Secs
decay 0.01 5.0 0.5 Secs
sustain 0.0 100.0 75.0 %
release 0.1 5.0 0.25 Secs
ENV2 target - off -
amount -100.0 100.0 0.0 %
attack 0.01 5.0 0.1 Secs
decay 0.01 5.0 0.5 Secs
MOD WHEEL - off -
amount -100.0 100.0 50.0 %
VELOCITY - off -
amount -100.0 100.0 50.0 %
DISTORTION - off -
gain 1.0 12.0 1.0 Gain
DELAY L mix 0.0 100.0 0.0 %
L time 0.01 2.0 0.15 Secs
L feedback 0.0 100.0 35.0 %
DELAY R mix 0.0 100.0 0.0 %
R time 0.01 2.0 0.25 Secs
R feedback 0.0 100.0 25.0 %
high-pass 50.0 20,000 200.0 Hz
low-pass 50.0 20,000 3,500 Hz
polyphony 1.0 36.0 36.0
portamento 0.0 5.0 0.0 Secs
bend depth 0.0 36.0 2.0 S-T
global tuning -6.0 6.0 0.0 S-T
gain 0.0 4.0 1.0 Gain

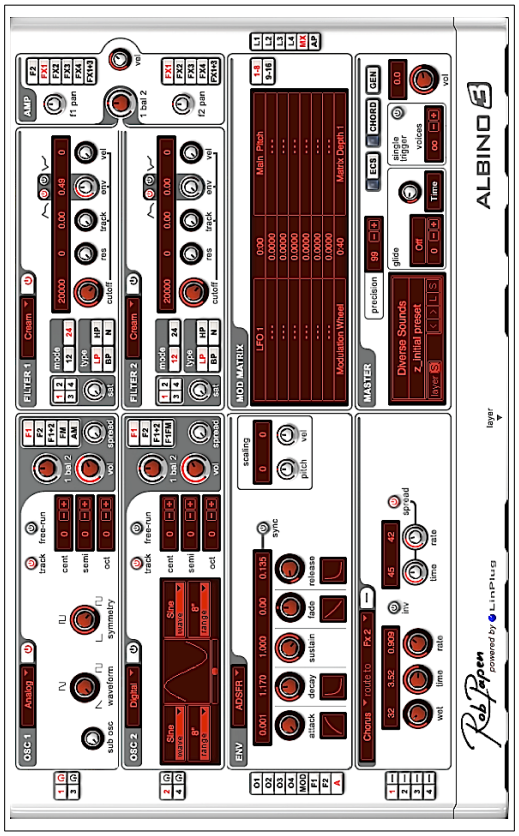
Appendix D

Synthesiser Screenshots

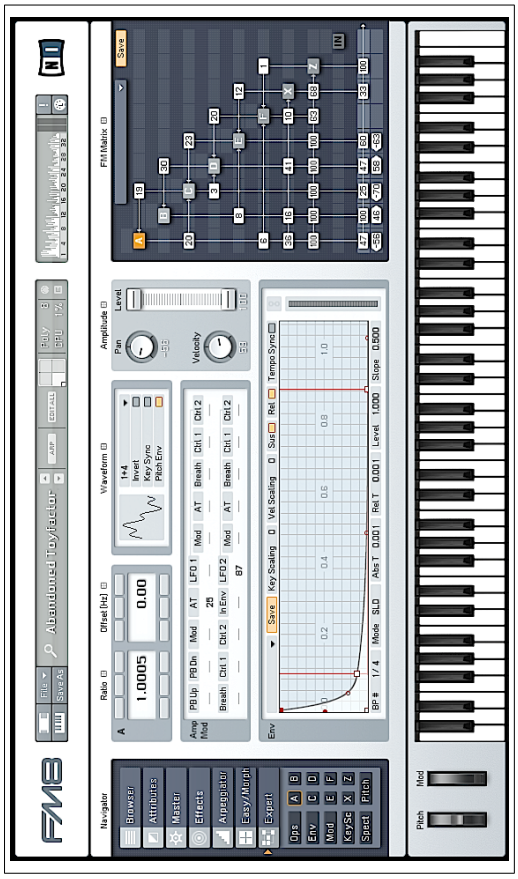
For a discussion of these instruments, please refer to Section 1.5.



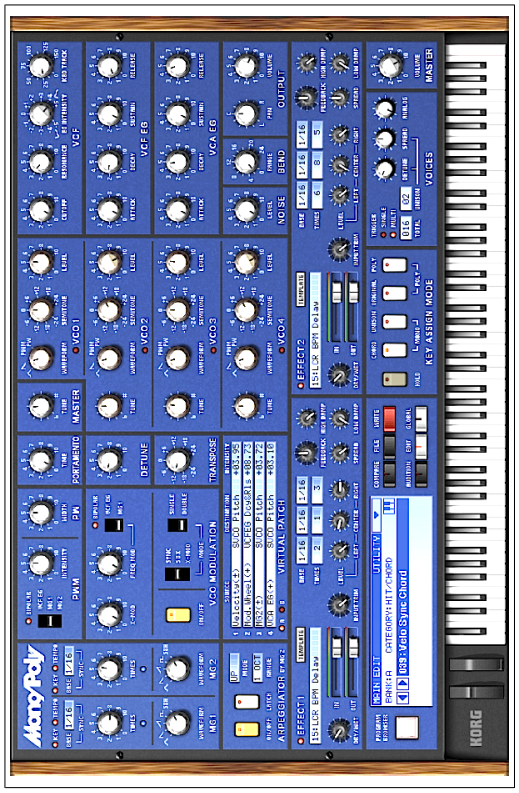
(a) ES2



(b) Albino



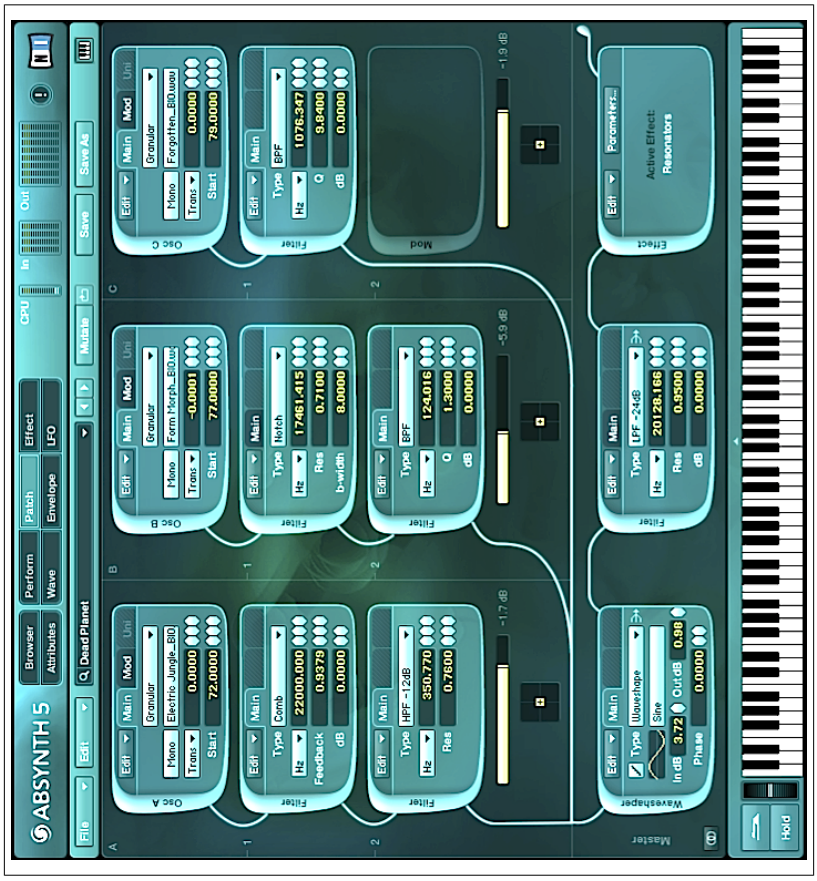
(c) FM8



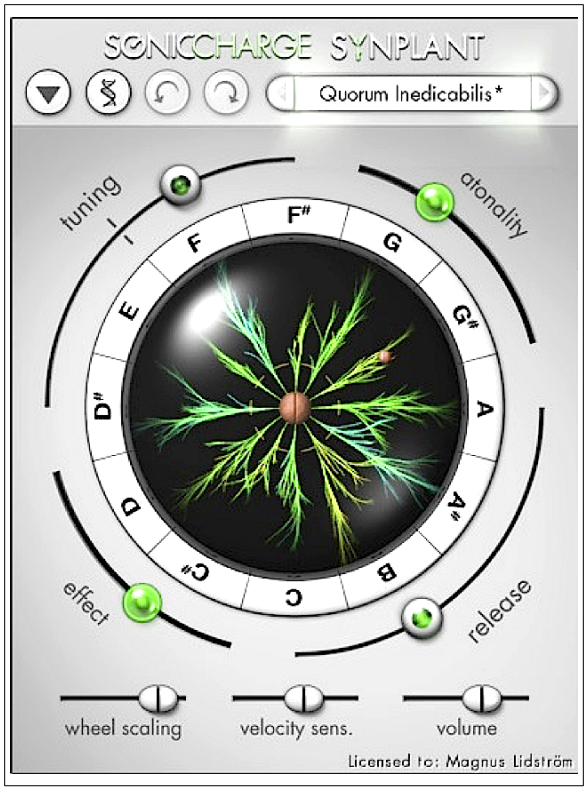
(d) Monopoly

(e) Massive

(f) Sculpture



(g) Absynth



(h) Synplant

Appendix E

Oscillator waveforms

Primitive waveforms types discussed extensively throughout this document.

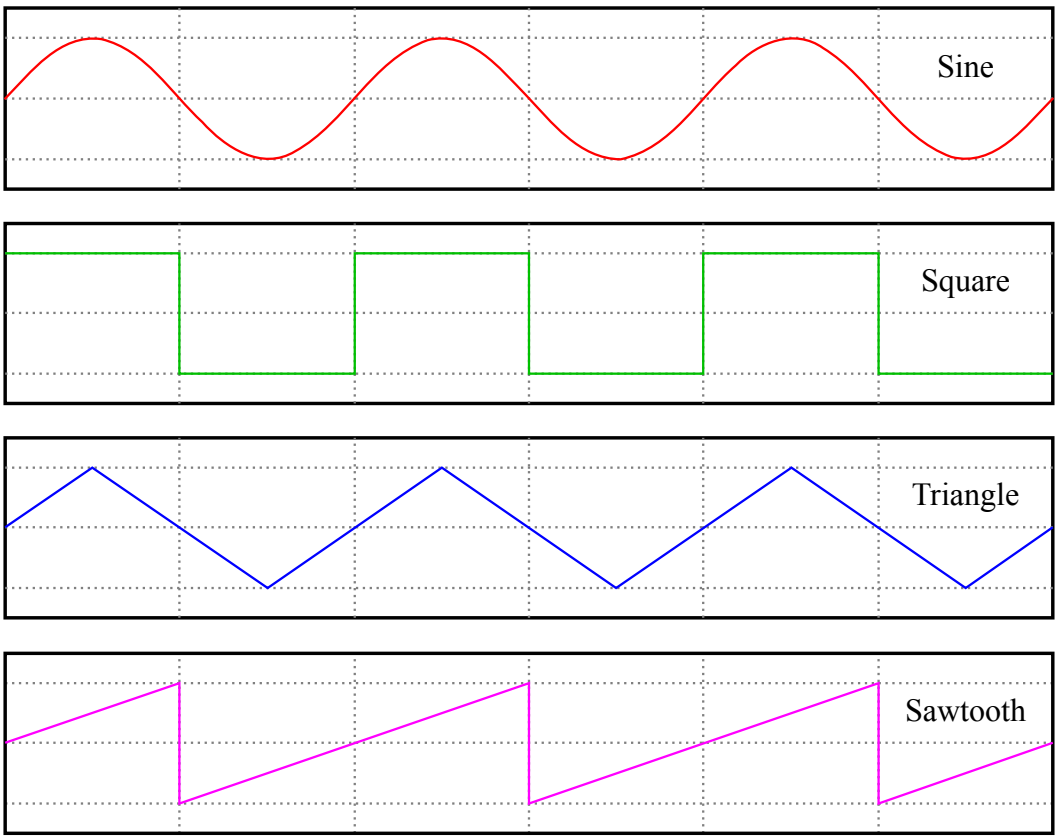


Figure E.1: Oscillator waveform primitives

Bibliography

[1] Appleton, J., and Perera, R. *The development and practice of electronic music.* Prentice Hall, 1975.

[2] Bateman, W. A. *Introduction to computer music.* John Wiley & Sons, 1980.

[3] Black, H. *Modulation theory.* Van Nostrand, 1953.

[4] Borin, G., Poli, G. D., and Sarti, A. Algorithms and structures for synthesis using physical models. *Computer Music Journal 16*, 4 (1992), 30–42.

[5] Chowning, J. The synthesis of complex audio spectra by means of frequency modulation. *Journal of the Audio Engineering Society 21*, 7 (1973), 526–534.

[6] Darter, T. John chowning, an interview. *Aftertouch Magazine Volume 1* (date unknown, reproduced at `http://www.maths.abdn.ac.uk/~bensondj/html/Chowning.html`).

[7] Hartmann, W. *Signals, sound, and sensation.* Amer Inst of Physics, 1997.

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

[9] Mathews, M. *The Technology of Computer Music.* The MIT Press, 1969.

[10] Mitchell, D. *BasicSynth.* Lulu.com (self-published), 2008.

[11] Moorer, J. A., Grey, J., and Strawn, J. Lexicon of analyzed tones. part 1: Violin tone. *Computer Music Journal 1*, 2 (1977), 39–45.

[12] Moorer, J. A., Grey, J., and Strawn, J. Lexicon of analyzed tones. part 2: Clarinet and oboe tones. *Computer Music Journal 1*, 3 (1977), 12–29.

[13] Moorer, J. A., Grey, J., and Strawn, J. Lexicon of analyzed tones. part 3: The trumpet. *Computer Music Journal 2*, 2 (1978), 23–31.

[14] Morse, P. *Vibration and Sound.* American Institute of Physics for the Acoustical Society of America, Woodbury, New York USA, 1936.

[15] Pinch, T., and Trocco, F. *Analog days: The invention and impact of the Moog synthesizer.* Harvard Univ Pr, 2004.

[16] Roads, C. *The Computer Music Tutorial*. The MIT Press, February 1996.

[17] Roads, C., and Mathews, M. Interview with max mathews. *Computer Music Journal 4*, 4 (1980), 15–22.

[18] Smith, J. O. Physical modeling using digital waveguides. *Computer Music Journal 16*, 4 (1992), 74–91.

[19] Smith, J. O. *Introduction to Digital Filters with Audio Applications*. W3K Publishing, 2007.

[20] Stilson, T., and Smith, J. Alias-free digital synthesis of classic analog waveforms. In *Proceedings of the International Computer Music Conference* (1996), Citeseer, pp. 332–335.

[21] Stroustrup, B. *The C++ Programming Language: Special Edition*, 3 ed. Addison-Wesley Professional, 2000.

[22] Wood, P., and Pierce, J. Recollections with john robinson pierce. *Computer Music Journal 15*, 4 (1991), 17–28.