University *of* Limerick

Thaw

Software for the Creation and Manipulation of Static and Drone Sounds Using the Phase Vocoder

Cormac Daly B.Sc.

M.Sc. Masters in Music Technology

Supervised by Mikael Fernström

Submitted to the University of Limerick, September 2006

DECLARATION

Thaw

Software for the Creation and Manipulation of Static and Drone Sounds Using the Phase Vocoder

Supervisor: Mikael Fernström

This Thesis is presented in partial fulfillment of the requirements for the degree of Master of Science in Music Technology. It is entirely my own work and has not been submitted to any other University or higher education institution, or for any other academic award in this University. Where use has been made of the work of other people it has been fully acknowledged and fully referenced.

Signature:_____

Cormac Daly

September 1st, 2006

“The drone is the eternal voice of the universe.”

- Greg Davis

ACKNOWLEDGEMENTS

I would like to extend my sincere gratitude to:

- All the staff at The Centre for Computational Musicology and Computer Music (CCMCM) in UL, for sharing their collective wisdom. Thanks in particular to Mikael Fernström for numerous forms of support in his role as my Supervisor, and to Jürgen Simpson for contributing to the ideas which formed the basis of this work. The initial guidance provided by Margaret Cahill and Donncha Ó Maidín, as well as the support of David Carugo, is also greatly appreciated.

- Richard Dobson, whose work provided a solid foundation for parts of the Thaw software. His generous personal advice was most invaluable.

- The largely anonymous and unpaid community of online audio software developers and enthusiasts, whose patient tuition has greatly improved my programming skills. Thanks must also go to my team of beta testers, whose valuable feedback has significantly improved the final product.

- All of my classmates, for stimulating my imagination as well as providing me with a year of unique education, inspiration and friendship.

- My family, whose advice and support is perpetually available and gratefully received.

- And to Jen, to whom I dedicate this work, for her encouragement, patience and support.

TABLE OF CONTENTS

TABLE OF FIGURES **3**

ABSTRACT **4**

Chapter 1 INTRODUCTION AND BACKGROUND **5**

1.1 Conception ***5***

1.2 Goals ***7***

Chapter 2 CREATIVE ORIGINS **8**

2.1 Static Sound ***8***

2.2 Early Traditions ***9***

2.3 Drone Minimalism and Modern Dronology ***10***

2.4 Spectral Music ***12***

2.5 Drone Instruments ***13***

Chapter 3 EXISTING WORK **16**

3.1 Existing Products – The State of the Art ***16***

3.1.1 GRM *Freeze* 16
3.1.2 Smartelectronix *Ambience* 17
3.1.3 *Spectral Monkeyage* 18
3.1.4 Smartelectronix *Sloper* 19
3.1.5 Smartelectronix *FlitchSplifter* 19
3.1.6 CDP Phase Vocoder plug-ins 20
3.1.7 Further Examples 21
3.1.8 In Summary 21

3.2 Potentially Suitable Signal Processing Techniques ***22***

3.2.1 Looping 24
3.2.2 Granular Synthesis 24
3.2.3 Time Domain Harmonic Scaling 24
3.2.4 Linear Predictive Coding 25
3.2.5 Phase Vocoder 25

3.3 Choosing a Signal Processing Solution ***26***

Chapter 4 THE PHASE VOCODER IN DETAIL **28**

4.1 History of the Phase Vocoder ***29***

4.2 Functionality of the Phase Vocoder ***29***

4.2.1 Overview 29
4.2.2 Filtering 30
4.2.3 The Fourier Transform 31
4.2.4 The Fast Fourier Transform 32
4.2.5 The Filter Bank 32
4.2.6 Windowing and Overlapping 33
4.2.7 Phase 36

4.3 The Phase Vocoder's Role in Digital Music ***37***

4.3.1 Time Scaling 37
4.3.2 Pitch Shifting 38
4.3.3 Further Effects and Future Possibilities 38
4.3.4 Software Implementations of the Phase Vocoder 39
4.3.5 The CARL Phase Vocoder and the Composers' Desktop Project 41

Chapter 5 SOFTWARE DESIGN **42**

5.1 Platform ***42***

5.2 Stand-Alone vs. Plug-In. *43*

5.3 The Virtual Studio Technology Plug-In Format in Detail *44*

5.4 Programming Language and Environment *45*

5.5 Implementation Fundamentals and Strategy *46*

Chapter 6 SOFTWARE IMPLEMENTATION **47**

6.1 Thaw Sound Transformations and their Parameters *47*

6.2 Software Structure *51*
6.2.1 File Structure 51
6.2.2 Class Structure 51

6.3 Initialisation Methods *54*
6.3.1 Thaw Initialisation 54
6.3.2 Phase Vocoder Initialisation 54

6.4 Signal Processing Algorithms *55*
6.4.1 The `processReplacing()` method 55
6.4.2 Phase Vocoder Incrementation and Freeze Effect 57
6.4.3 Creating the Frozen Frame 58
6.4.4 The Sound Transformation Algorithm 59
6.4.5 Further Noteworthy Algorithms 62

6.5 Further Implementation Particulars *64*
6.5.1 Phase Vocoder Parameters 64
6.5.2 Software Testing and Refinement 65

Chapter 7 CREATIVE IMPLEMENTATION **66**

7.1 Concept and Goals *66*

7.2 Composition, Production and Post Production *67*

7.3 Aesthetic *68*

Chapter 8 CONCLUSIONS AND FUTURE POSSIBILITIES **70**

8.1 Observations on Sonic Performance *70*

8.2 Achievement of Goals in Summary *71*

8.3 Further Proposed Developments to Thaw *72*
8.3.1 User Interface 72
8.3.2 Optimisation 73
8.3.3 Multiple Platform Support 74

8.4 Scope for New Creativity *74*
8.4.1 An Enhanced Implementation Context 75
8.4.2 Vector-Based Frame Morphing 75

REFERENCES **77**

APPENDICES **82**

Appendix I. C++ Code Examples **83**

Appendix II. Excerpts from the Development Log **88**

Appendix III. Sample Beta Testing Reports **89**

Appendix IV. Software Screenshots **92**

TABLE OF FIGURES

Figure 1: The GRM Freeze interface. *17*
Figure 2: The Spectral Monkeyage interface. *18*
Figure 3: The Sloper interface. *19*
Figure 4: The FitchSplifter interface. *20*
Figure 5: A simple model of the Phase Vocoder *30*
Figure 6: DC offset *32*
Figure 7: FFT windowing *34*
Figure 8: Rectangular windowing *34*
Figure 9: A comparison of window performance in artefact reduction. *36*
Figure 10: Phase. *37*
Figure 11: Phase vocoders in Max/MSP and Pure Data *40*
Figure 12:The SPEAR interface. *40*
Figure 13: The Thaw project's file structure. *51*
Figure 14: The relationships between the components of the Thaw software. *52*
Figure 15: The production environment for What Goes Around *68*

ABSTRACT

***Thaw*: Software for the Creation and Manipulation of Static and Drone Sounds Using the Phase Vocoder**

Cormac Daly B.Sc.

Audio processing software entitled 'Thaw' is presented. The aim for the development of Thaw is to allow a computer musician or sound designer to create a perceptual 'snapshot' of an instant of any sonic source material, and allow the manipulation of this snapshot to create drones and other static and slowly-moving sounds.

The aesthetic motivation for the development of Thaw is inspired by the paradoxical concept of music stopped in time, and endeavours to facilitate the practices of drone minimalism and related drone-based musical aesthetics. A review of the background to such musical practices, including spectral music and more recent drone-based traditions in electronic music, is therefore presented. This investigation indicates the need for such a tool as Thaw, in order to continue the drone tradition in modern electronic music production environments.

Bearing in mind the proposed aesthetic, the unique audio spectrum modification capabilities of the phase vocoder are harnessed in order to create the initial sonic snapshot, and enable subsequent effects. These effects, including low frequency oscillator, pitch shifter, spectral degrader and parametric spectral filter, are implemented in order to 'thaw' the initial 'frozen' sound and allow for the creation of the aforementioned drones.

Thaw is successfully implemented in the form of a Virtual Studio Technology (VST) effect plug-in, enabling its use within a broad variety of host applications and popular computer music production contexts. This thesis documents the power of the phase vocoder for the creation of such effects and the subsequent design, implementation and refinement of Thaw. As well as examining the originality and significance of Thaw, this document concludes by proposing future enhancements to the software, and by commenting on the significant potential inherent in the phase vocoder for the future exploration of the drone aesthetic.

A short composition entitled *What Goes Around* is also presented. This was inspired by the drone aesthetic and its creation was enabled by the functionality of Thaw, thus demonstrating the role of the software in allowing the development of this tradition in the era of electronic music.

Chapter 1 INTRODUCTION AND BACKGROUND

1.1 Conception

The conception for this Masters project evolved from a fascination with static sound and the potential for creation of an illusionary snapshot of sound. Peripheral ideas related to this concept were soon developed, refined and implemented, and later extended to the exploration of the creation and manipulation of static timbres.

Sound, described in a basic physical sense, consists of a wave-like motion. This motion exists in its natural form as waves of periodically-varying pressure in a medium such as air, or may be represented electronically as continuous streams of analogue or digital data. Sound by its nature is therefore dynamic and exists only within the context of the time domain. A true 'snapshot' of any sound is impossible.

Initially, this project sought to implement a piece of software which would create the perceptual illusion of such a paradoxical snapshot; that is, a static, 'infinitely time-stretched', unmodulating sound which would retain the timbre, pitch and amplitude of the original sound at the point where it is 'frozen'.

Initial research, however, unearthed an array of pre-existing software products which sought to achieve this effect to varying extents and in a variety of contexts; a representative selection of these products are discussed in detail in Section 3.1 and an analysis of the different digital signal processing techniques commonly used to create these techniques is also presented. The initial plan to create a simple 'freezer', as the effect is commonly known, was therefore extended and this resulted in the software development phase of this work culminating in the creation of a plug-in, entitled 'Thaw', which allow a composer to explore the paradox of effectively removing the time dimension from the manipulation of a piece of digital audio in ways that allow new and unique forms of control over the flow of the music. The title 'Thaw' was chosen

to represent the nature of the sounds produced by the software; initially ‘frozen’ and static, yet capable of becoming ‘thawed’, dissolved, and slowly-moving.

The evolution of the final design for this plug-in was heavily influenced by the unique audio manipulation possibilities offered by the phase vocoder. This digital signal processing technique, which is discussed in detail in Chapter 4, has attracted much interest amongst both musicians and programmers in recent years. Therefore, the processes of researching and designing Thaw evolved to incorporate an attempt to take advantage of, and more importantly to build upon, an open source implementation of the phase vocoder and thereby further extend a fascinating vein of research into this tool. As well as creating the initial static snapshot of sound, the phase vocoder facilitates the manipulation of the resulting audio spectrum in a unique manner, further allowing for exploration of the creation of static sounds and drones.

Other conceptual factors which contributed towards the final shape of this work included a high degree of initial trial and error, experimentation using several diverse software packages such as *Pure Data* and *Csound*, and digital signal processing techniques other than the phase vocoder, such as looping and granular synthesis. The decision to implement the software in the form of a plug-in also meant that the product would need to function in real-time; this implementation scenario offered another set of constraints and exciting possibilities.

Apart from practical concerns, it is important to note that this project began as a creative concept; aesthetic concerns were therefore highly influential throughout the lifecycle of this work. With this in mind, a discussion on the history of static sounds in music, and hence the role of the proposed software in the creation of such sounds, is presented in Chapter 2.

A short composition entitled *What Goes Around*, inspired by the drone music tradition and intended to demonstrate the capabilities of the Thaw software, is also presented amongst the deliverables of this project. The development of this composition is outlined in Chapter 7.

1.2 Goals

As outlined above, the project conceptualisation evolved during the course of the work and the goals therefore evolved accordingly, especially owing to the large and broad selection of pre-existing work which was unearthed. Following a period of refinement and experimentation, a number of clear aims have emerged which expand upon the initial concept of a 'freezer'. The fundamental goal of creating a static sound remains intact, however, and has been augmented to encompass a broader scope of research and creativity.

Therefore, the primary goals of this work are as follows:

- **To develop software which will allow computer musicians and composers to create and manipulate high quality static and slowly-moving sounds.** This software will extend the capabilities of similar pre-existing products by focusing exclusively on the creation of such sounds using the phase vocoder technique.

- **To exploit the power of the phase vocoder in the implementation of this software.** The distinctive possibilities inherent in this digital signal processing technique have been explored previously in the manipulation of musical time and it is widely recognised that the phase vocoder offers a wealth of unexplored possibilities for the creation of unusual sounds. A degree of experimentation with such possibilities has been a major factor in determining the form of the innovations achieved in the first goal above.

- **Inspired by Dronology and other such musical tradition, to compose a short piece of music which would exploit the functionality of the software developed.** This piece would also serve to illustrate the added creative dimension which the Thaw software makes available to computer musicians.

With these goals in mind, it is necessary to ascertain the need for the Thaw software. This process begins with an exploration of the creative background of the aesthetic of static sound, in Chapter 2.

Chapter 2 CREATIVE ORIGINS

The musical or practical need for any software audio manipulation tool needs to be explored in order to determine whether anything useful is to be achieved by its development. This chapter will, therefore, document the long and fascinating thread of musical tradition which explores static and drone-like sounds, thereby attempting to demonstrate that the need exists for such a tool to continue this tradition in the era of digital music. Several existing acoustic instruments which previously attempted to explore the creation of static sounds are also discussed; this aims to further demonstrate the available niche and necessity for analogous software tools to create similar sounds.

2.1 Static Sound

While a completely static, unmodulating sound is generally quite uninteresting[1], drone-like sounds are often used in music. Musical drones, which typically consist of continuous, static sounds with slowly-moving harmonics or timbres, have frequently been incorporated into many and varied types of music. Drones may be harmonic or monophonic, or occasionally noise-based, and have served as either fundamental components of musical pieces or, perhaps more commonly, as peripheral embellishments which are not intended to capture or maintain the interest of the listener. A typical peripheral implementation of the drone functions as a harmonic centre for a piece of music. Almost universally, drone music is rhythmically static or very slow and, instead of drawing attention to conventional facets such as rhythm and tonality, allows the listener to focus their attention on the microscopic, internal aspects of the individual sounds which comprise the musical whole. Because of this, interesting facets of such internal aspects of music which would not be apparent in a denser musical context may be revealed due to the static nature of the drone; for example, the emphasis of slight mistunings caused by the equal temperament tuning system.

[1] This is often the case with any aspect of music at any level. For example, static or repetitive musical ‘form’, at a macroscopic (high) level, is often to blame for causing a piece of music to be perceived as boring, as it causes the listener to rapidly lose interest.

These, and other dimensions of drone sound, have been summarised as follows:

> …sustained intonation that establishes a harmonic center for its accompanying elements…the drone might utilize a single note repeated indefinitely or, at the opposite extreme, all of the scale's notes spread across numerous octaves. Other key aspects include extended duration, modular repetition, and a focus on overtones…the trance-inducing drone with its extended tones and layered pitches does change but glacially…
>
> (Textura, 2005).

2.2 Early Traditions

The idea of creating static and slowly-moving sounds is certainly not new and has originated in several forms of ancient and 'world' music; several representative examples of such genres will now be discussed briefly.

The Japanese gagaku tradition originated in 7th century Imperial Japan and incorporated drone-creating instruments as well as conventional tempered instruments. Gagaku music is still played in Japan to this day and has influenced 20th century composers including Henry Cowell (*Ongaku*, 1957) and Benjamin Britten (*Curlew River*, 1964). Hindustani and Indian classical music is also frequently accompanied by the tambura, which is capable of playing only drones, or by the sitar; such music often features infinitely cyclical rhythmic structures known as 'talas', which serve to further accommodate the drone aesthetic. The indigenous didgeridoo music of Australia is a well-known example of drone-based music which continues to be played in the modern era. One-note drone-like Gregorian chants which predate polyphonic Western music were commonplace in Medieval Europe. Similarly, the Shamanist spiritual tradition has used the singing of structurally static songs known as 'icaros' in ritualistic contexts.

These musical traditions, while somewhat obscure in the modern Western world, serve to demonstrate the ubiquitous penetration of drones and static sounds in musical tradition. Some of these old or ancient traditions remain alive

today, and perhaps more importantly, they have served to inspire contemporary artists and extend the use of drones into modern Western music.

2.3 Drone Minimalism and Modern Dronology

The practice of incorporating drones into a multitude of musical genres has extended into the 20th and 21st centuries and has become known as 'Dronology'. Although the early 20th century works of atonal composer Anton Webern feature long tones, such as in the third movement of *Five Pieces for Orchestra, Op. 10* (1910-1913), any discussion of dronology in modern Western music invariably begins with La Monte Young's pioneering works of the 1960s. Young has cited Webern and gagaku in particular as the primary influences for his groundbreaking *Trio for Strings* (1958) (Strickland, 1993, p.125) – he also famously claimed that this was the first piece of music to have ever been created with nothing but long, sustained sounds[2] (Morgan, 1991, p.424).

Although the work of Young and his contemporaries Terry Riley and John Cale is known as Minimalism, their music and that of the more prominent minimalists Steve Reich and Philip Glass occupy two considerably different musical subsets[3]. Tony Conrad, another early pioneer of dronology, has described minimalist music (of the drone variety) as involving "tonality, repeating modes, and long pieces with middles but no endings or beginnings" (Textura, 2005).

This flavour of minimalism, known as 'drone minimalism', first began to emerge in the late 1950s and early 1960s from the California-based group 'Theater of Eternal Music' (or 'Dream Syndicate'). This ensemble, which focused on the creation of drone-based music, included Young, Cale, Riley,

---

[2] The variety of older music discussed in the previous section would appear to contradict this claim. This piece is nonetheless a highly significant work, and is often cited as the work which founded minimalism (Strickland, 1993, p. 122).

[3] The minimalist musical genre emerged during the 1950's. Early minimalism can be roughly broken down into two subgenres: the drone variety, practiced by Young, Riley, Cale and their contemporaries; and a more tonal, repetitive and structured variety pioneered by Reich and Glass. A wider discussion of the latter type of minimalism is outside the scope of this work – Edward Strickland's *Minimalism: Origins* (1993) provides such a discussion.

Conrad and a number of others. The processes used to create these early works varied from piece to piece, but typical traits included static instrumentation, linear transformations (i.e. a lack of discrete structural sections), pure tuning ratios (i.e. scales which are not used in conventional Western music, such as 'just intonation'), and non-Western musical influences, such as Indian classical music, Indonesian gamelan and other such traditions as discussed in Section 2.1. Noteworthy works produced by Young and the Theater of Eternal Music, around the same time as *Trio for Strings* (1958), include Young's *Second Dream of the High Tension Line Step-down Transformer* (1962) and the Dream Syndicate's *The Tortoise Recalling the Drone of the Holy Numbers as they were Revealed in the Dreams of the Whirlwind and the Obsidian Gong* (1964).

Although the work of the Theater of Eternal Music and other drone minimalists is often dismissed as esoteric experimentation, it occupies a far more prominent role in music as far as dronology is concerned. This work would inspire further composition in the dronology mode in art music circles in the latter half of the 20th century, and would even go on to inspire mainstream and popular artists from the 1960s onwards.

This influence was exemplified by the rock band The Velvet Underground, one of whose members was violist and Theater of Eternal Music founder, John Cale; they frequently embellished their early works in the mid 1960s with drone sounds. Similarly, Kraftwerk – a hugely influential electronic group – featured several pieces which were based around static and lengthy instrumental drones in their first album released in 1970. Other examples of drone-inspired music to emerge during the 1970s included Tangerine Dream's double prog-rock album *Zeit* (1972), and Brian Eno and Robert Fripp's collaborative ambient albums *No Pussyfooting* (1973) and *Evening Star* (1975). Drones have also found their way into the music of contemporary electronic artists Aphex Twin (*Selected Ambient Works Volume II*, 1994) and Boards of Canada (*Corsair*, from the *Geogaddi* album, 2002). The ambient electronic group Minit, founded in 1997, uses electro-acoustic and digital processing techniques to create abstract meditative soundscapes, such as in the album and title track *Now Right Here* (2004).

Dronology has not only inspired individual artists and composers, but has heavily influenced, or even spawned, musical genres. Ambient music, a super-genre of several flavors of drone music, inherently features static or drone-like sounds, and has cross-fertilized and influenced different or derivative genres, such as New Age music, ambient techno, and IDM[4]. 'Drone Doom' or 'Drone Metal' (a sub-genre of doom metal music) marries rock instrumentation with the drone aesthetic. This little-known style features down tuned guitars and basses, lengthy static musical pieces and large amounts of distortion and reverberation.

2.4 Spectral Music

A phase vocoder will be used to implement the Thaw software – this digital signal processing tool, discussed at length in Chapter 4, has served as one of the many new means for creating a relatively recent phenomenon known as spectral music. A short introduction to spectral music, and its relevance to this work, is therefore now presented.

Spectral music emerged in the latter part of the twentieth century, notably in the works of Tristan Murail and Gérard Grisey. Spectral music is not limited to one particular style or genre; its composition usually draws from the composer's understanding of acoustics, the internal spectral structures of his/her source material, and the understanding of how psychoacoustic phenomena may be heard and interpreted by the listener. It has been described by Murail as "an attitude towards music and composition, rather than a set of techniques" (Fineberg, 2000, p.2), and usually involves the use of modern computer technologies to understand and exercise a unique and precise form of control over these musical spectra. This attitude thereby allows a composer to sculpt individual sounds in a manner which allows for composition modes which focus on surface texture and new timbres in preference to conventional musical traits such as rhythm and tonality. In this manner, spectral music may be seen in some

[4] IDM - 'Intelligent Dance Music', an electronic genre which has particular inspirational value in this work due to the widespread trend of using software plug-ins for timbre composition in the genre.

cases as an aesthetic extension of the core methodologies of the drone minimalism genre; and, notwithstanding spectral music's inability to be tied down to any single genre, the works of La Monte Young and his contemporaries have been cited as the precursors of spectral music (Anderson, 2000). Fineberg makes the following observations which commonly, but not universally, apply to spectral music:

> The music has made colour into a central element of the musical discourse, often elevating it to the level of principal narrative thread…orchestral fusion is often a main feature of its surface texture, so that individual voices are subsumed in the richness of the overall texture and colour…the basic sonic image is often sonorous and resonant giving the music a sort of acoustic glow that comes from the coherence – in the domain of frequencies – of the different constituent pitches…this music simply sounds profoundly different than other musics.
>
> (Fineberg, 2000, p.3).

The relevance of spectral music to the development of Thaw will be further explored in Chapter 4, including an examination of the unique attributes of the phase vocoder that make it suitable for creating the sort of sonic transformations common to spectral music. Hence, the finalized Thaw software will exploit the phase vocoder's possibilities in order to enable its use as a tool for the creation of such music.

2.5 Drone Instruments

The means used by musicians to create static and drone sounds are numerous and varied. The instruments in question, like all established musical instruments, have evolved over centuries and have had their individual designs influenced by the requirements of the types of music they aimed to create. Several examples of such instruments are now presented:

- **The Didgeridoo**. This is a wind instrument (or aerophone) of the indigenous Australians. Played using circular breathing, it is capable of producing only a single pitch. However, interesting and unique harmonics may be created

and slowly manipulated by the contortions of the player's mouth and vocal tract.

- **The Bagpipes and Uilleann pipes**. These wind instruments, native to Scotland and Ireland respectively, commonly feature a 'drone pipe' which accompanies the melody played by the piper.

- **The Sitar and Tambura**. These stringed Indian instruments incorporate 'drone' strings as well as conventional 'melody' strings.

- **The Bluegrass Banjo**. This instrument is a variation of the banjo; although derived from an African instrument it is now primarily used by American Bluegrass musicians. The fifth string is typically tuned to the pitch of the next string at the fifth fret and is seldom fretted whilst playing.

Although these examples are largely designed to incorporate or specifically create drones, a large variety of other instruments have been used in imaginative ways to also create drones. These include the violin or viola, such as in La Monte Young's *Trio for Strings* (1958); muted trumpets, such as in one particular interpretation of Young's *Second Dream* composition (performed in 1991); and the Jew's harp.

As the 20th century progressed, musicians turned increasingly towards new electronic methods of sound production and this practice naturally appealed also to musicians who wished to create 'dronal' music. Although tape loops had been used by Musique Concrète[5] practitioners as early as the 1950s, in the 1980s drone artists such as The Loop Orchestra began to use reel-to-reel machines and magnetic tape in order to create drone music, which, being "far from static, ebbs and flows hypnotically… the loops are simple, organic and very human" (Textura, 2005). The e-bow, a small electromagnetic device which allows guitarists to create 'infinite sustain' with their instruments, is another

[5] Musique Concrète – the practice of composing music using 'found sounds'. The birth of this movement was facilitated by the introduction of magnetic tape as a medium for sound in the 1940s.

example of an electronic technology which has been developed for the purpose of creating drones.

As technology has progressed, digital music production techniques have eventually succeeded tape-based music and extended the possibilities offered by acoustic instruments. The potential means for the electronic production of drone music have therefore mushroomed into a seemingly infinite scope of tools and techniques. Although composers have turned to software such as *Max/MSP* and similar musical programming environments in order to manually create drone music, this research has found very little evidence of previous software development with the sole purpose of facilitating this particular musical aesthetic. Chapter 3 will now present the 'State of the Art' in existing software products which, to some extent, explores the creation of static and drone sounds, thereby setting the scene for the development of the Thaw software; one of goals of which is to address this deficit.

Chapter 3 EXISTING WORK

3.1 Existing Products – The State of the Art

Digital computers have vastly increased the scope of means of expression available to composers and musicians who seek to create static or drone-like sounds. Software instruments and effects have presented a new sound-producing paradigm, and therefore a number of products which achieve the exploration of the static and drone sound phenomenon to various extents now exist. These products, which have been developed both commercially and academically, vary greatly with regard to the type of aesthetic they produce, the digital signal processing techniques they utilise in order to create this aesthetic, the way in which they allow the user to control the parameters of the sound-manipulation algorithms, and the manner in which they take control, or allow the user to take control, over the flow of the music.

This section will, therefore, initially present a summary of a review carried out of the overall goals and functionality of a representative range of the very large array of 'freezer' software products currently available to the public. Furthermore, this summary will draw attention to unique and/or original features of these products, thereby setting the scene for discussion later in the thesis on how this large feature set can be enhanced.

3.1.1 GRM *Freeze*

Freeze is a Virtual Studio Technology (VST)[6] standard plug-in created by the GRM (Groupe de Recherches Musicale, 2006). It allows a user to visually select a piece of streaming audio and create very short loops which may be changed in size or temporal location using an X-Y controller. The visual aspect of the control is useful as it allows the user to anticipate attacks, decays and other

[6] A popular plug-in standard developed by Steinberg which is compatible across a wide range of host applications. The VST format is described in detail in Section 5.3.

aspects of the amplitude in the music, and freeze them as desired. Its unique mode of control over the time domain also allows for subtle, continuous timbral change. Although the sound produced by the *Freeze* plug-in is not entirely static, as the looping technique used is often audible, this product is nevertheless highly popular among users (KVR Forums, 2006) and presents a fascinating mode of control over the flow of the sound.

Figure 1: The GRM *Freeze* interface[7].

3.1.2 Smartelectronix *Ambience*

Ambience (Jonsson, 2005) is a freely-available, flexible and high-quality reverberation VST plug-in which offers a 'Hold' function as a peripheral feature. This feature allows the user to instantly and abruptly freeze the output of the plug-in, producing an effect which is somewhat similar to the initial static sound aspired for by the plug-in which is the subject of this research. *Ambience* fails, however, to offer parameters which intend to facilitate the subsequent creation of drones. Furthermore, the 'Hold' effect may only be applied to the 'wet' reverberation output and not the 'dry' signal, hence reducing its potential scope for use. The digital audio manipulation technique used by the author of this plug-in to produce the 'Hold' effect is unknown. It is suspected that a phase

[7] Each of these plug-ins requires a host application in order to function – in these screenshots, each plug-in is pictured within Tobybear *MiniHost* (Section 6.5.2**Error! Reference source not found.**).

vocoder is used, however, as there is a similarity here with the sound quality of known phase vocoder-based plug-ins.

3.1.3 *Spectral Monkeyage*

Spectral Monkeyage[8] is a simple plug-in which allows for the production of innovative and abstract sounds through manipulation of the audio in the frequency domain. It offers an 'infinite timestretch' feature which may be introduced instantaneously or gradually via a 'spectral blurring' parameter. This produces an odd smearing effect that leads seamlessly to the impression of the infinite timestretch. Other possible effects created by *Spectral Monkeyage* include drastic pitch scaling, control of individual frequency dynamics, and further exotic effects which are not easily described. An undesirable side-effect produced by this plug-in, however, is a slight smearing effect on the sound even when no effect is being applied. It would appear that *Spectral Monkeyage* uses a phase vocoder technique in order to produce its effects, but this cannot be confirmed as the anonymous author of the effect could not be contacted. It is certain that *Spectral Monkeyage* utilises a Fourier Transform, however, as it offers the user a choice of window sizes and techniques; this, along with the slight audible smearing, would suggest that a phase vocoder is used.

Figure 2: The *Spectral Monkeyage* interface.

[8] No reference is available for *Spectral Monkeyage* as it is no longer supported and does not have an official website.

3.1.4 Smartelectronix *Sloper*

Sloper (Schnetzler, 2006) is another VST plug-in which functions as a 'stutter-stretch delay'. The innovative control mode allows a user to create a 'stuttering' flow through the music by varying a variety of nested loop speeds, loop sizes, delay parameters and scan rates; a static or slowly-moving sound may be achieved through careful manipulation of these parameters. The plug-in takes a tempo parameter, supplied by the host, and the inter-loop scan rate and pseudo timestretch effects are dependent on this.

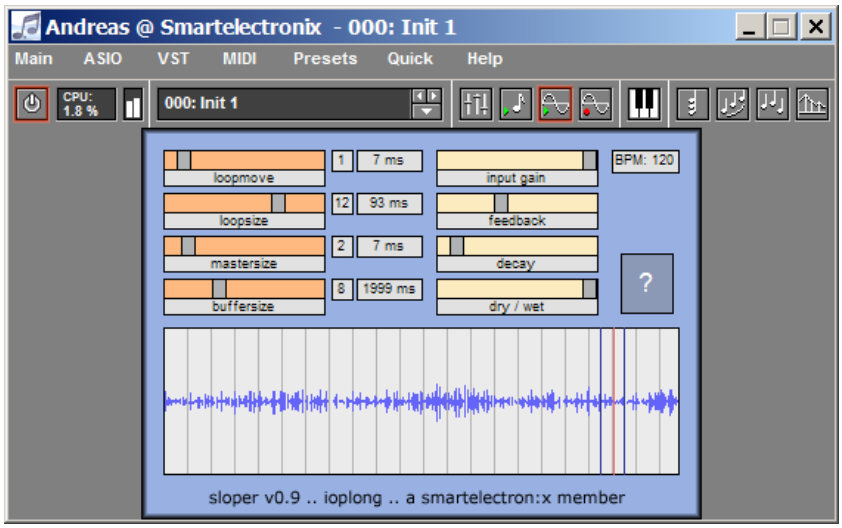


Figure 3: The *Sloper* interface.

3.1.5 Smartelectronix *FlitchSplifter*

FitchSplifter, by the same author as *Sloper*, is described as a "midi-playable dynamic buffer freezer fx plug-in" (Schnetzler, 2003). It uses a looping technique to achieve a somewhat frozen sound. Its innovative interface (Figure 4) allows for a large degree of control over the flow of the sound. The input signal is passed through an envelope detector and gate. If the signal exceeds the gate threshold, the input is recorded in a buffer and repeated; the number of loops being determined by user-controlled parameters. A host of other parameters allow control over wet/dry mix, 'smoothing' factor, bit reduction, resonance, a 3-mode filter and envelope control. The plug-in also allows the

user to ‘play’ the buffered loops using a MIDI keyboard. *FlitchSplifter*, while not aiming to create a wholly static sound or drone aesthetic, takes the fascinating approach of trying to combine a ‘freeze’ aesthetic with a ‘glitch[9]’ aesthetic, and the results are unique.

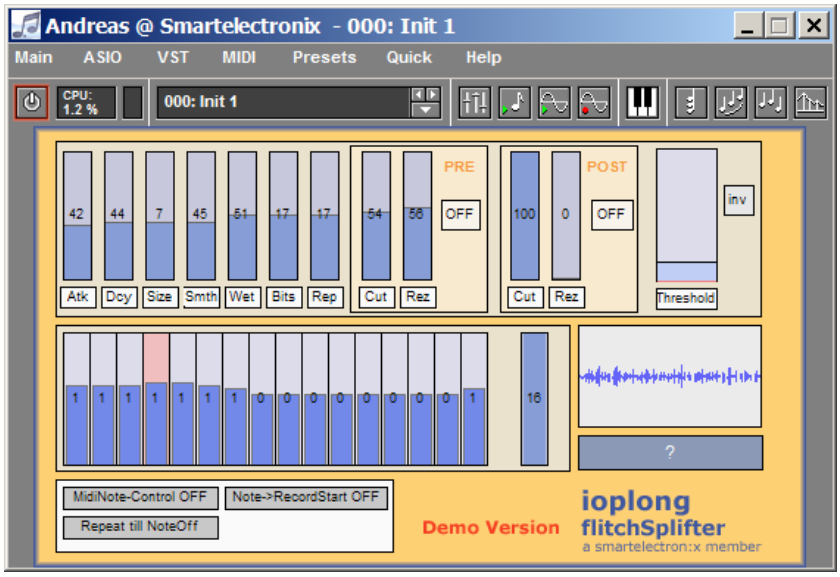


Figure 4: The *FitchSplifter* interface.

3.1.6 CDP Phase Vocoder plug-ins

This untitled suite (Dobson, 2001) consists of three simple experimental VST plug-ins: *‘pvtran’*, *‘pvexag’* and *‘pvaccu’*. The initial work around which these effects are based was originally created by Trevor Wishart and his collaborators for the Composer’s Desktop Project (Miranda, 2002)[10] and later adapted for streaming purposes in a VST environment by Richard Dobson (Dobson, 2001). Although none of these aims to create a freeze effect, they are included in this review as they all utilise the phase vocoder in creative and innovative ways – the phase vocoder is the digital signal processing technique used to implement the Thaw plug-ins; it is discussed in detail in Chapter 4. *pvtran* is described as a

[9] ‘Glitch’ in this context refers to an intentional (albeit seemingly chaotic) musical aesthetic, as opposed to the undesired sounds caused by equipment or programming errors, which will be mentioned later in this thesis.

[10] The phase vocoder contained within the Composer’s Desktop Project package is a remarkable piece of work; it contains over 60 tools for the manipulation of sound using the phase vocoder. This is explored in further detail in Section 4.3.5.

‘simple pitch shifter’, and performs this basic function with excellent sonic results. *pvexag* exaggerates the spectral envelope. *pvaccu* performs ‘spectral accumulation’ with glissando. The latter plug-in, in particular, was of interest due to its inherent potential for producing slow-moving or near static sounds.

The further importance of this software’s role in the development of Thaw is documented below, in Section 3.1.8.

3.1.7 Further Examples

- ***Max/MSP* groove~** – an object for the modular *Max/MSP* musical programming environment which operates using the same principles as GRM *Freeze* and produces similar results – groove~, however, does not offer an effective interface or suitable peripheral controls, as *Freeze* does.

- **Ohmforce *Symptohm*** – a comprehensive synthesis plug-in instrument whose features include the ability to create a freeze effect using granular synthesis. This is a tedious, manual process however, and it is not the sole application of the software.

3.1.8 In Summary

This range of software products utilise a broad range of techniques, interfaces and musical aesthetics and share the common attribute of being capable of creating static sounds to some extent, but with varying degrees of limitation. This is understandable given that these products have not been designed with the sole intention of creating and/or manipulating static or drone sounds. The notable exception is perhaps GRM *Freeze*, whose ability to sweep backward and forward through a time-domain signal enables it to produce drone-like sounds. However, deficiencies in achieving the static/drone aesthetic have already been highlighted in *Freeze* – notably, the aesthetic drawbacks of the looping technique and the lack of potential relevant sonic modifications which may be performed within the loops themselves.

The examination of the CDP Phase Vocoder plug-ins, however, had a number of valuable advantages. Firstly, they demonstrated the unique power of the phase vocoder and provided an incentive to further explore this power. Secondly, the source code for this plug-in suite is freely available and reproducible under the GNU Public Licence (GPL) (Free Software Foundation, 1991). This offers a unique opportunity for interested parties to explore the workings of the phase vocoder in a VST framework, either at a general, theoretical and functional level or at a deeper, mathematical level. The freedom of the GPL also allows for the modification and/or redistribution of the software. This source code, based around an implementation of the Computer Audio Research Laboratory (CARL) phase vocoder (Loy, 2002), was to prove invaluable in founding the development of the Thaw software. Its role is discussed in further detail in Section 4.3.5.

This review therefore set the scene for a study on how the more desirable attributes of these products could be combined, developed and/or refined in order to design a piece of software whose primary goal would be to create and manipulate high quality static and drone sounds. Section 3.2 will now discuss a consideration of which digital signal processing technique(s) might be used in order to achieve this goal.

3.2 Potentially Suitable Signal Processing Techniques

As the authentic digital reproduction of a given instant of sound is impossible, as explained in the Introduction of this thesis, the final aural output of the proposed software product merely needs to be perceptually convincing. Although the sound created by existing ‘freezers’ is subjective and best described in qualitative terms, it is at least necessary to preserve amplitude, pitch and timbre in order to recreate a convincing sound, depending on the authenticity and overall aesthetic of the sound that is aspired to. Whereas pitch and amplitude are relatively straightforward to measure and reproduce, it is well documented that the perception of timbre is highly dependant upon ‘timbre

envelope'[11] (Howard & Angus, 2001, p.222). For example, research has shown that it is difficult for a listener to differentiate between instruments as diverse as the piano and the trumpet if the attack and/or decay portions of the sound are removed (Howard & Angus, 2001, p.221). This poorly-understood role of the time-domain in timbre perception may pose a challenge given the static nature of the desired sound.

According to Grey (1977), timbre does not have a one-dimensional subjective scale, such as those scales on which loudness and perceived pitch can be measured on. Changes in harmonic relationships in the time domain are, therefore, vital in preserving timbre. Pollard and Jansson (1982) described a 'tristimulus' method which is an approximate means of tracking the changes in the spectral content of sounds over time; in other words, an attempt to quantify the temporal dynamics in timbre. As such established means of perceptually describing timbre are dependant on time, the implementation of the proposed system needs to attempt to perceptually bypass this requirement for dynamic change.

The reproduction of pitch is also highly dependent on the time domain – pitch cannot be measured instantaneously due to its periodic nature, and psychoacoustic experiments have identified that the human auditory system must be exposed to a sound for at least 20ms (approx.) in order to successfully identify pitch. The chosen signal processing technique must therefore also allow for the identification of pitch.

The success of faithfully capturing a snapshot which preserves pitch and timbre may lie in the careful choosing of one of a number of suitable audio analysis, manipulation and reproduction techniques. The pre-existing examples explored in the previous section utilise a variety of such techniques to achieve their goals. Each of these, as well as a number of other potential strategies, has advantages and drawbacks which will now be discussed.

---

[11] Timbre envelope - the variation of spectral energy distribution (i.e. frequency components of a sound) over time.

3.2.1 Looping

This technique, used by GRM *Freeze* and *Max/MSP* groove~ as well as many other products, creates very short loops within the streaming audio to create the freeze effect. While this technique is relatively simple to implement and does not require large amounts of computing power to perform its function, its use for creating static sounds is not quite convincing. This is due to the fact that, depending on how well the technique is applied, the loops may contain 'rough edges'. This causes audible artefacts, and an impression of periodicity which may not be desirable in our context. This 'glitchy' effect may be successfully avoided, however, perhaps by applying a feature akin to 'windowing'[12].

3.2.2 Granular Synthesis

This is another time-domain technique which is not dissimilar in principle to the looping technique. Tiny 'grains' of sound of specified quantity and duration are randomly assembled in the form of either raw, synthesised sounds, or samples taken from an input signal and reassembled in a manner which may be determined by the user/programmer; the manner in which this is done determines the type of resulting sound. Although a number of existing granular synthesis-based products which are capable of producing 'freeze' sounds exist, none of them are dedicated solely towards creating this effect.

3.2.3 Time Domain Harmonic Scaling

Time Domain Harmonic Scaling (TDHS) is another time-domain technique commonly used for pitch/time scaling. It is computationally quite fast, as no Fourier Transforms are used. Its time-stretching role functions by obtaining the fundamental frequency (f_0) of a sound, and subsequently overlapping or cross-fading different sections of the sound in order to create time-stretching.

---

[12] A technique used by the phase vocoder which reduces the amplitude at the beginning and end of a portion of analysed audio. This technique is described in more detail in Section 4.2.6.

However, it only works well with well-pitched monophonic sounds. Also, it commonly produces poor quality results and its potential for 'infinitely' time-stretching a piece of audio seems to be unexplored.

3.2.4 Linear Predictive Coding

Linear Predictive Coding (LPC) analyses a sound (usually human speech) and creates a data representation of its spectral content. It separates what it perceives to be noisy sounds (or residue) from harmonic or pitched sounds and creates separate representations of these. Timbre, pitch and rhythm can therefore be separated and treated independently; time-stretching and pitch changing are possible applications of this feature. It is commonly used for compression/resynthesis of telephone signals. Again, however, the resulting sound quality is quite low and the potential offered by this technique for achieving the desired freezing function is unknown.

3.2.5 Phase Vocoder

The phase vocoder uses Fast Fourier Transforms to convert audio signals to frequency-domain representations where both frequency and phase information are preserved. These representations may be manipulated while in the frequency domain, and then resynthesised. If the frequency information from the sound is interpolated at resynthesis at a different rate than it was during analysis, time-stretching without pitch change may be achieved. Other unusual effects may be achieved by manipulating the signal within the frequency domain.

Like all the other techniques discussed above, the phase vocoder introduces audible artefacts (or 'side effects')[13] during resynthesis, although these effects may not be as severe as those produced by the other techniques. They may be kept to a minimum by carefully choosing suitable parameters; different phase vocoder characteristics (discussed in Chapter 4) suit different types of sounds.

[13] Usually reverberation and a slight audible 'smearing', in the case of the phase vocoder.

The phase vocoder is quite powerful in its potential and also produces fairly consistent results regardless of the type of sound being processed.

3.3 Choosing a Signal Processing Solution

All of the potential solutions discussed above offer various advantages and disadvantages. In the worst cases, the use of the time domain harmonic scaling or linear predictive coding techniques for the creation of static and drone sounds seems to be previously completely unexplored. Apart from their previous uses as time stretching and time compressing tools, there appears to be little evidence available to support the notion that there is good potential in these techniques for producing our desired aesthetic. It follows, therefore, that the use of one of these techniques would be unwise. Similarly, the timbres produced by the granular synthesis technique may be unpredictable given the hugely varied nature of sounds which may be fed into the plug-in. Granular synthesis may therefore not be the best option for this project.

The remaining techniques – the phase vocoder and looping methods – appear to be the most suitable; at least, they have been used more than any other technique in previous implementations of software freezers. The looping technique offers the significant advantage of being very straightforward to implement. This may allow the programmer to place the looping in a control context of high complexity, which could allow for very innovative and hands-on means of control (as demonstrated by the *Freeze*, *Sloper* and *FitchSplifter* plug-ins in particular). Within the actual loops of sound themselves, however, there is not a large amount of scope for innovation. Furthermore, unless they are very carefully implemented, the loops could give the perceived impression of periodicity; this would not conform to the initial ‘static’ aesthetic that the goals of this work call for.

The phase vocoder, on the other hand, has the potential to create high fidelity static sounds regardless of the type of sound which is being analysed, as demonstrated in particular by the *Ambience* and (as far as can be told) *Spectral*

Monkeyage plug-ins. Perhaps more significantly, the unique amplitude/frequency data representation used by the phase vocoder allows for unparalleled opportunities for the creation of new and innovative types of audio spectral manipulation. The fact that it is less explored than more conventional signal processing techniques means that its power may be largely untapped, allowing for a great degree of potential innovation in this work.

The phase vocoder does offer disadvantages, of course; it is notoriously difficult to implement, especially in real-time systems. This may not be a significant problem however, as the coding of a phase vocoder from scratch would be outside the scope of this work. It is also highly computationally expensive, due to the large number of Fast Fourier Transforms used. This is becoming less of a problem, however, due to the relentlessly increasing power of modern computers.

Despite these disadvantages, the background research into potentially suitable signal processing techniques outlined in this section indicated that the phase vocoder was the most promising solution for creating the Thaw software, largely owing to its potential for manipulating sounds in new and innovative ways that will be outlined in further detail in Section 4.3.3. As a more detailed description of the phase vocoder is necessary in order to understand the development and workings of the Thaw software, a fuller investigation now follows in Chapter 4. This draws on a large body of work exploring the use of the phase vocoder which has emerged in recent decades (including Dolson, 1986; Dobson, 1993, 2001; and Bernsee, 2005).

Chapter 4 THE PHASE VOCODER IN DETAIL

Any understanding of the inner workings of the Thaw software requires knowledge of the workings of the phase vocoder. However, the implementation of this software has not necessitated in-depth investigations or manipulations of the inner workings of the phase vocoder or the Fast Fourier Transform. Therefore, although a seemingly lengthy description of its workings is presented here, the following discussion barely scratches the surface of the extensive creative, engineering and mathematical work which has been carried out on the phase vocoder to date. Although a lengthy study of the fundamentals of digital signal processing and the phase vocoder was undertaken by the author in preparation for this work, detailed analyses of the mathematical and physical aspects of the nature of the phase vocoder remain outside its scope.

In addition to its relevance to the development of Thaw, it is hoped that the scope of the description of the phase vocoder presented here would be of particular interest to computer musicians and programmers; while the use of the phase vocoder in non-conventional musical applications was largely untapped in the past (Dolson, 1986), interest in the technique appears to have grown in recent years.

Following an introduction to the history and functionality of the phase vocoder, a summary of its role in manipulating and creating music will be discussed in this chapter. It will then conclude with a number of examples of previous software implementations of the phase vocoder. Sources consulted in preparing this presentation of the phase vocoder include Richard Dobson's *The Operation of the Phase Vocoder* (Dobson, 1993), Stephan M. Bernsee's *The DFT "à Pied"* (Bernsee, 2005) and Mark Dolson's *The Phase Vocoder: A Tutorial* (Dolson, 1986), as well as many others.

4.1 History of the Phase Vocoder

The phase vocoder has had a long and distinguished history as a tool for the analysis and manipulation of digital signals in a range of engineering, musical and other applications where the frequency-domain analysis of periodic signals is necessary. First described in a technical paper by Flanegan and Golden of Bell Laboratories (Flanegan & Golden, 1966), the phase vocoder extended Homer Dudley's Channel Vocoder (Dudley, 1939) by describing both frequency and phase components of an analysed signal. The target application of this new technique was to be the encoding of voice signals for reducing transmission bandwidth. Unfortunately, the fact that the analysis data produced by the phase vocoder was much greater than the original time-domain signal meant that it was unsuitable for this purpose.

However, the advent of powerful digital computers meant that the phase vocoder was later adopted as a digital audio processing tool. Many contemporary composers have experimented with such software phase vocoders, leading to the revelation of radical new possibilities for the manipulation of sound. It has become one of the most uniformly reliable and flexible techniques for performing time scale modifications (i.e. time stretching and compressing) of digital audio. Subsequently, in the last several decades, a variety of software implementations of the phase vocoder have been developed. Such implementations, and potential future uses of the phase vocoder, are presented in detail at the end of this chapter.

4.2 Functionality of the Phase Vocoder

4.2.1 Overview

The operation of the phase vocoder (Figure 5) occurs in two stages: analysis, whereby a time-domain signal is converted to a spectral representation, and resynthesis, whereby the reverse process is carried out. The scope for manipulation of the analysis data between these two stages is one of the

attributes of the phase vocoder that gives it its potential for creating unique audio manipulations.

The phase vocoder steps through the time-domain signal to be analysed in overlapping steps, or 'windows'. During the analysis of each window, a bank of band pass filters uses Fast Fourier Transforms to create frequency/amplitude representations of the spectral content of the window. Frequency information may hence be manipulated without affecting the temporal characteristics of the signal, and vice-versa, before similar overlapping windows use a bank of oscillators to resynthesise the original or modified signal.

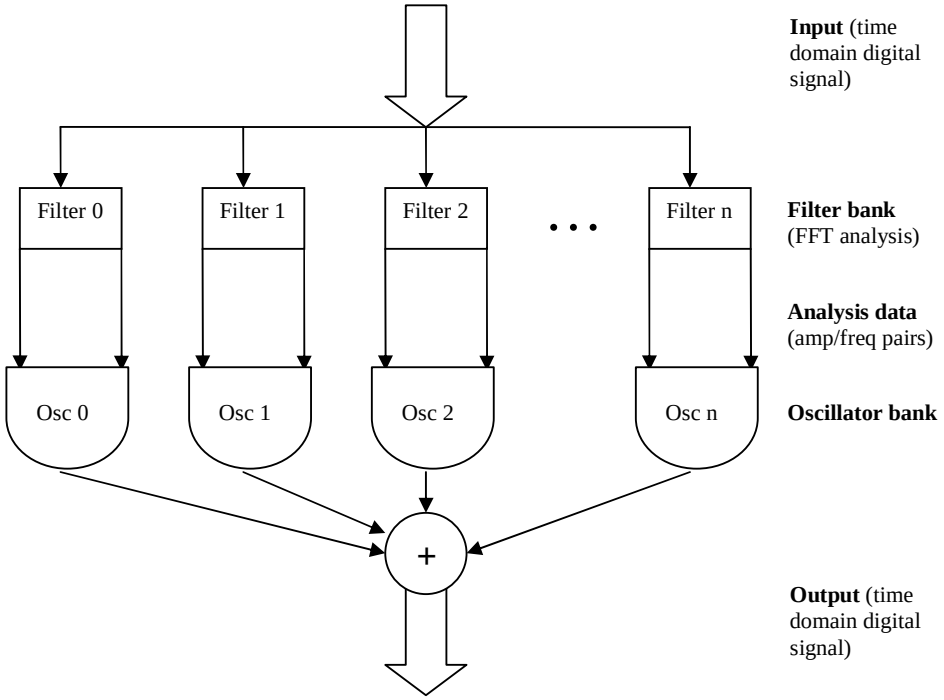


Figure 5: A simple model of the Phase Vocoder (adapted from Cunningham, 2003).

4.2.2 Filtering

During the analysis stage, a bank of band pass filters analyses a time-domain digital signal. This series of filters must accommodate all frequencies which are present in the incoming signal and their centre frequencies are therefore spaced equally from 0Hz to half the sample rate. The individual band pass frequency response must be identical, ensuring that the overall frequency response of the

filter bank is flat. This means that no frequency band may be reproduced disproportionately and the resynthesised signal may be produced as faithfully as possible. The output of these filters consists of pairs of data (or bins) representing amplitude and frequency. The manipulation of these bins will be discussed presently.

4.2.3 The Fourier Transform

A Fourier Transform is used to implement each digital filter. This method is based on principles developed by Jean Baptiste Fourier (1768-1830) which state that any periodic waveform, regardless of its origin or degree of complexity, can be represented by the sum of a set of harmonically related sinusoidal waves. In the case of harmonically related sine waves, each wave is said to be an integer multiple of the fundamental frequency. The harmonic structures of common waveforms such as square, triangle, saw-tooth, and so on, as well as arbitrary complex signals, can therefore be measured with a high degree of precision. The principle was originally intended to deal with the conduction of heat in materials.

The Fourier Transform detects the magnitudes of the frequencies present using a pattern matching method, analogous to the additive synthesis technique of ring modulation. When two sine waves are multiplied, the resulting frequencies represent the sum and difference of the combined input frequencies. For example, if a signal of 440Hz is multiplied by a signal of 460Hz, the output consists of a sine wave of 900Hz and a sine wave of 20Hz. If the inputs are equal, then the difference is zero and the output consists of a sine wave whose frequency is equal to twice the sum of the input. However, it should be noted that a DC offset (Figure 6) is introduced – that is, the resultant signal lies above the zero line. The magnitude of this offset is directly proportional to the amplitudes of the input signals. Therefore, if one of these inputs is at a fixed reference level, the amplitude of the other may be determined. The Fourier Transform exploits this principle by sweeping through the waveform which is to be analysed with a ‘reference’ sine wave of fixed amplitude and gradually

sweeping frequency. The resultant, continuously fluctuating amplitude is recorded. Any non-zero amplitude in this signal flags the presence of a corresponding input harmonic at that frequency and amplitude.

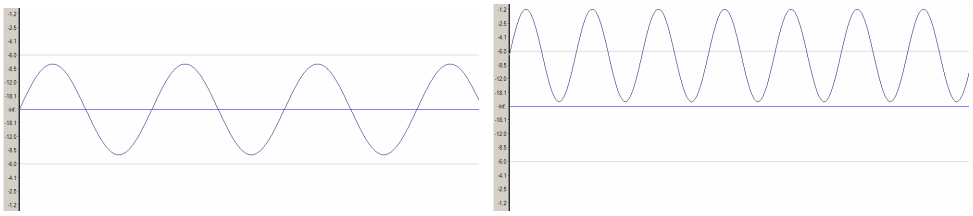


Figure 6: DC offset in the wave to the left is zero; the DC offset is positive in the wave to the right

4.2.4 The Fast Fourier Transform

As a truly analogue, continuous analysis sweep would be hugely computationally expensive and unsuitable for use in a digital system, the Fast Fourier Transform (FFT) is used. This streamlined version of the Fourier Transform discretely samples the analysis waveform at multiples of its own fundamental frequency. As long as this sampling rate is high enough, a good spectral analysis of the signal may be reproduced by the FFT.

4.2.5 The Filter Bank

The operation of the FFT is similar to that of a bank of finely tuned band pass filters, and is therefore ideal for what is required by the phase vocoder. As the overall bandwidth of the filter bank and the individual frequency responses of the individual filters are predetermined[14] the only factors to be decided in the design of the filter bank are the number of filters to use and the frequency response of each filter. If a high degree of precision in measuring frequency is required, the bandwidth of each filter must be as narrow as possible. Correspondingly, the number of filters required to cover the audible range becomes larger.

[14] That is, the bandwidth must correspond to the audible range of frequencies – approximately 20Hz-22.5kHz – and the individual frequency responses must be uniform.

A bank of filters with bandwidth of, for example, 20Hz, would provide adequate coverage at 440Hz, where 20Hz corresponds to approximately one semitone. Spaced linearly, however, 1000 such filters would be required to cover the audible spectrum. At 4kHz and above, this 20Hz resolution is unnecessarily high; and at the lower end of the spectrum, 20Hz can cover up to an octave and is therefore not high enough. This problem occurs because of the logarithmic relationship between frequency and pitch.

In this manner, the FFT is unfortunately not very economical[15]. Simple periodic waveforms may be analysed efficiently enough, as monophonic sounds do not contain partials in close proximity to each other. The harmonic content of the spectra of arbitrary complex sounds (which comprise the input sounds of the phase vocoder the vast majority of the time, at least where musical applications are concerned) is much more uncertain, however, and therefore the sideband frequency responses which are unavoidable artefacts of the band pass filters become more unpredictable and potentially problematic. Input frequencies which do not correspond exactly to the centre frequency of any of the band pass filters will register in the outputs of peripheral filters; this effect is known as 'spectral leakage' and its effects may be overcome by 'windowing' the incoming analysis data.

4.2.6 Windowing and Overlapping

Whilst the amplitude of a piece of digital audio may be measured at any given instant (i.e. by measuring the amplitude of a single sample), the instantaneous measurement of frequency may not be achieved due to its periodic nature. As discussed in Section 3.2, psychoacoustic experiments have identified that it takes approximately 20ms for the human ear to accurately identify a pitch (Howard & Angus, 2001). Therefore a 'window' of at least this size is generally used to analyse the signal. At a sample rate of 44.1kHz, 20ms corresponds to 882 samples. As a power-of-two window size is generally more efficient in FFT

[15] Unless the frequency content of the input signal is known beforehand, in which case the filter bank can be 'tuned' to perform an optimal analysis of the prominent frequencies in the signal. However, this is not the case in most applications.

implementations, a minimum window size of 1024 samples is therefore commonly used.

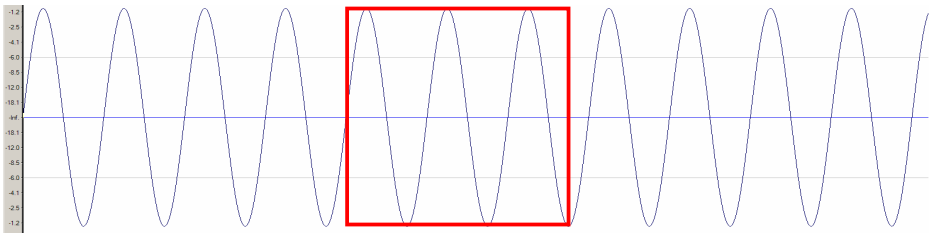


Figure 7: FFT Windowing

Such a window is considered a slice of sound: a useful analogy is to think of an FFT analysis window as a single frame of optical film – a single component in a sequence that, when reassembled, will form a continuous moving picture.

The problem with the rectangle-shaped window in Figure 7 is that its edges may fall on any sample. The resulting abrupt changes in amplitude may cause audible clicking in the resynthesised sound, as illustrated in Figure 8. As the FFT considers the windowed analysis frame to be exactly one cycle of a periodic waveform, the discontinuity in the cycle caused by the irregular cut off by the window edges of the sine wave (in this example) would lead to a significantly inaccurate resynthesis with unwanted sideband responses.

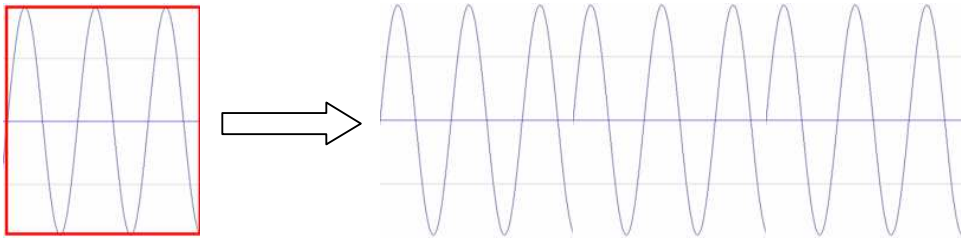


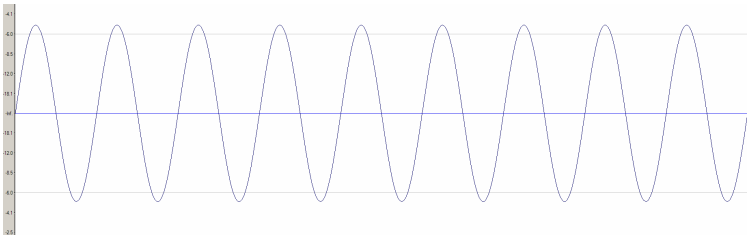
Figure 8: A rectangular window causes unpredictable changes in amplitudes and hence audible glitches in the output sound

The solution to this problem is to ‘squash’ the amplitudes at the edges of the analysis window by using a different window shape. This may be thought of as a symmetrical amplitude envelope which is applied to the window. A number of various types of windowing envelopes are used, including rectangular, Blackman, Kaiser, Bartlett and Hamming, and others. Different window types

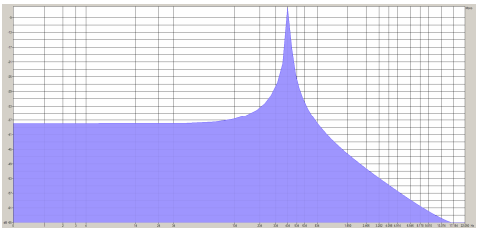
are suited for different purposes[16]. In terms of reducing the emergence of unwanted spectral artefacts, most of these designs are superior in performance over the rectangular window.

In Figure 9, a brief graphical comparison of spectral analyses of the same time domain signal (top) using several different window types is presented[17]. In the spectral analyses, the amplitude of the signal is represented on the vertical axis, and frequency on the horizontal. In each analysis, a peak is produced at 440Hz; however, it may be observed that the Rectangular window shape produces far more sideband responses than any other window:

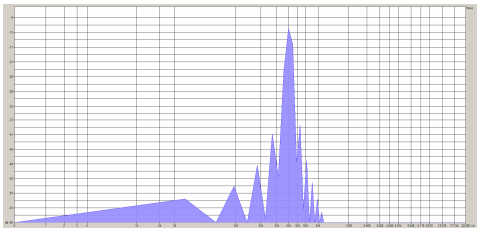
Time-domain representation of 440Hz Signal:



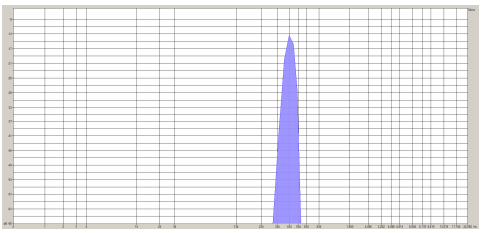
Spectral analyses of the signal:



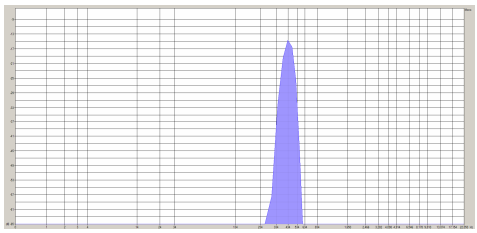
Rectangular



Triangular



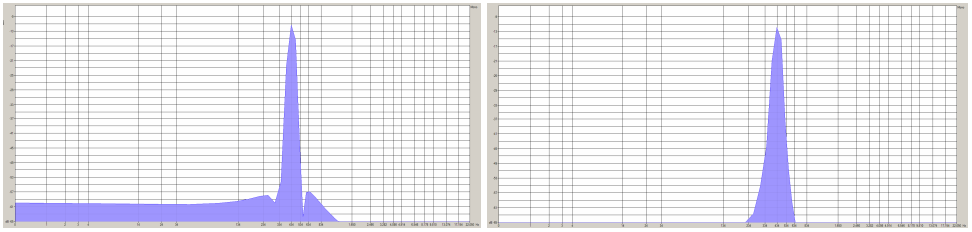
Blackman



Blackman-Harris

[16] The Thaw software uses a Hanning window by default, partially due to its suitability for analysing different types of sound and partially due to the limitations of the CARL phase vocoder which was used.

[17] These spectral analyses of a 440Hz sine wave at -5dB were generated in Sony Sound Forge using a phase vocoder of window size 1024 and overlap of 75%.



Hamming **Hanning**

Figure 9: A comparison of window performance in artefact reduction.

Another factor to be determined in the implementation of the phase vocoder is the overlap factor: in order to reduce the trade-off between frequency resolution and time resolution when determining the window length, the successive windows in the FFT analysis are overlapped. For example, for a window size of 1024 samples, successive FFTs may be applied to windows starting at 0, 64, 128, 192 samples, and so on. The analysis rate is therefore multiplied eightfold, leading to a much greater frequency and time resolutions and, of course, increased computational demands.

4.2.7 Phase

If the relative phases of the different amplitude/frequency bins are not known, the reconstruction of the overlapping windows during the resynthesis may produce undesirable clicks and glitches, or in the worst case, may bear very little resemblance to the original signal. If the phase vocoder is simply reproducing exactly what it has analysed, this is not an issue as it already has all the information it needs. However, a straightforward analysis-resynthesis is somewhat musically pointless – if the spectral data or rate of resynthesis is modified, as it usually is in musical applications of the phase vocoder, then the phase vocoder needs to supply additional phase information.

The phase of a signal relates to the ‘start point’ in the cycles of the waves describing its frequency components; one cycle, or 360°, is equivalent to a single wavelength. A sine wave and cosine wave, for example, begin one

quarter of a cycle (or 90°) apart – the sine wave is considered the 'imaginary' part of a signal and the cosine represents the 'real' part. A mathematical complex number stores each part. Plotted on an X-Y graph, the real and imaginary parts may be measured from the origin in terms of amplitude (distance) and phase (angle). This 'polar' representation is preserved by the phase vocoder and is central to the operation of the FFT and inverse FFT (iFFT).

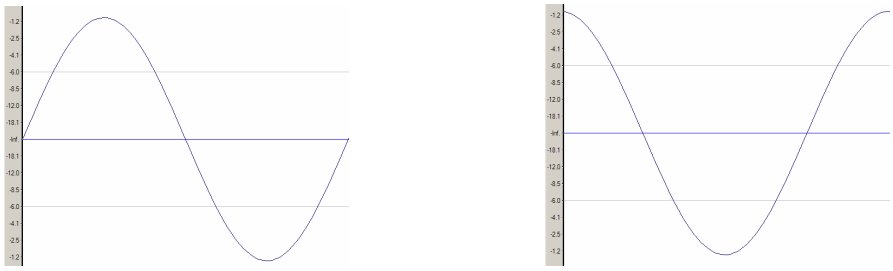


Figure 10: Single cycles (360°) of a sine wave, left, and cosine wave, right. The signals are 90° out of phase.

4.3 The Phase Vocoder's Role in Digital Music

Once the processes which have been described above are complete, the phase vocoder has all the information it needs to perform a resynthesis. It is at this point that the musical uses of the technique become apparent and the phase vocoder therefore becomes much more interesting to the computer musician (Laroche & Dolson, 1999).

The most common use of the phase vocoder in musical applications to date has been the enablement of the independent manipulation of either the temporal characteristics or pitch characteristics of a sound. The methods used to achieve these utilitarian effects are now described briefly.

4.3.1 Time Scaling

Resynthesis, the final phase of the operation of the phase vocoder, is subsequently carried out by a sequence of overlapping windows; within each window a bank of oscillators produces the specified frequency components at

the correct amplitudes. Time-domain scaling may be achieved by simply changing the rate of overlap between these successive windows. For example, if the resynthesis windows are interpolated at a faster rate than the rate at which the analysis windows were overlapped, time compression is achieved. The frequency/amplitude/phase information describing the original signal is left untouched[18] and the time-stretched or compressed reconstructed wave therefore retains most of these original characteristics.

4.3.2 Pitch Shifting

Pitch shifting of a signal upon resynthesis is simply an extension of the time scaling technique. After the overlap rate has been changed, the output may be resampled to a specified rate, thereby restoring the time domain factor but changing the pitch without further degradation of the signal. Significant research on improving the quality of the phase vocoder in pitch shifting is ongoing, but this work is outside the scope of this discussion.

4.3.3 Further Effects and Future Possibilities

Although time and pitch modifications have been the primary uses of the phase vocoder in digital music to date, a variety of other exotic effects have been achieved. It is important to note that the frequency-domain representation used by the phase vocoder offers a very new, unique and powerful means of manipulating sound. Such frequency domain manipulations would not be possible with a lesser signal processing technique. The examples of phase vocoder implementations for achieving such effects presented in this work only serve as demonstrations of the capabilities of the technique; the possibilities for future work doubtlessly extends far beyond these.

---

[18] Unless desired otherwise; see Section 4.3.3 for examples of how this information may be manipulated with interesting musical consequences.

Such innovative manipulations include "chorusing, harmonising, partial stretching" and other radical modifications, described by Laroche and Dolson (1999). The software suite developed by the Composer's Desktop Project (Wishart, 1994; Dobson, 2001) includes a phase vocoder which has been adapted to perform hundreds of similar feats such as 'spectral exaggeration' and 'spectral accumulation'. Some of the software effects discussed in Section 3.1 (e.g. *Spectral Monkeyage*) provide further examples of the types of audio manipulations that the phase vocoder has been used to create.

Apart from the variety of new sounds produced by the phase vocoder, another remarkable facet of its adoption by composers has been an attempt to develop alternative means of interfacing with the functionality of the phase vocoder, or controlling the flow of the music with a phase vocoder which has been inherently 'automated' to an extent.

An example of such a new means of interfacing with the phase vocoder is *SPEAR*[19] (Klingbeil, 2005), a stand-alone program which analyses a sound file and allows a user to graphically manipulate individual partials (frequency components) before dynamically resynthesising the sound. IRCAM's *AudioSculpt* performs a similar function using a phase vocoder called SuperVP (Bogaards, 2005). These programs, which do not function in real time, have created a stir of interest amongst the electronic and electro-acoustic music production community in recent years due to the fact that such a multidimensional and graphical means of frequency control is so new and innovative.

4.3.4 Software Implementations of the Phase Vocoder

Thanks to efforts by composers and programmers, software implementations of the phase vocoder now appear in many forms – amongst many, these developers have included the Composers' Desktop Project (Wishart, 1994), Dobson (1993, 2001), Dolson (1986), Bernsee (2005), and IRCAM (Bogaards, 2005). The outcomes of the work of these parties include freely-available C++ source code,

[19] *SPEAR* - Sinusoidal Partial Editing Analysis and Resynthesis.

hands-on examples in *Csound*, *Pure Data*, and *Max/MSP* (Figure 11), along with countless mainstream audio applications and plug-ins. Examples of such applications include *SPEAR* (Figure 12) and *AudioSculpt* as described in the previous section, and *Spectral Monkeyage* as described in Chapter 3.

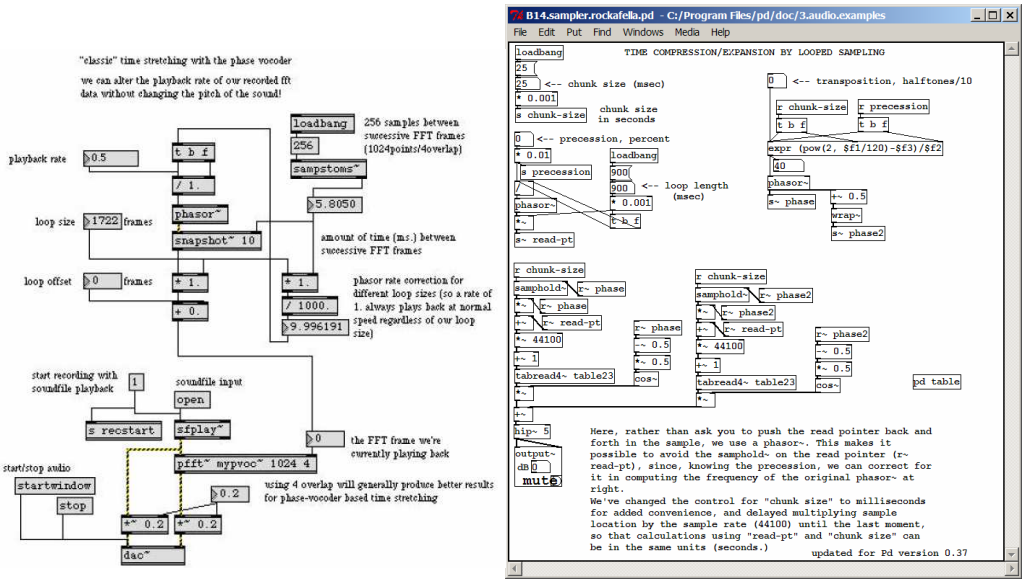


Figure 11: Phase vocoders in *Max/MSP* (left) and *Pure Data* (right).

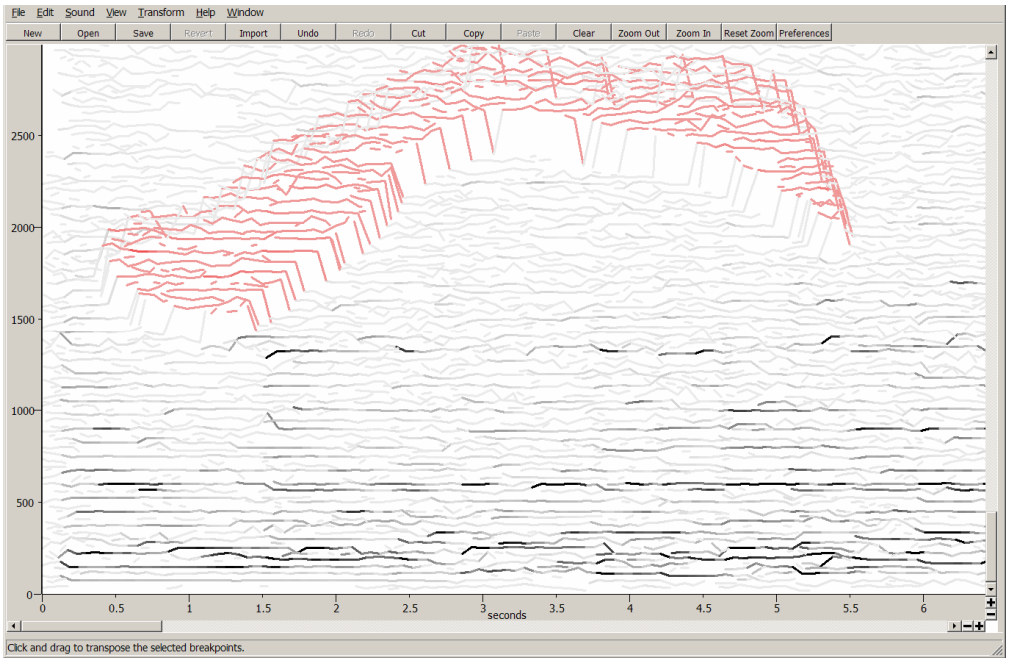


Figure 12: *SPEAR*'s innovative interface, which allows control over individual partials.

4.3.5 The CARL Phase Vocoder and the Composers' Desktop Project

An implementation of the phase vocoder of particular interest in this work is the Computer Audio Research Laboratory (CARL) phase vocoder (Loy, 2002) and its subsequent use by the Composers' Desktop Project (CDP). Trevor Wishart (Wishart, 2000) and a number of collaborators undertook the development of this large phase vocoder-based system at the IRCAM audio research institute from 1986. Basing his work upon the CARL implementation of the phase vocoder, Wishart implemented a number of software 'instruments' which manipulated the audio spectrum in ways never achieved before: these included "stretching the spectrum, spectral morphing, waveset manipulation, grain manipulation, sound shredding, spectral cleaning, spectral banding and brassage" (Wishart, 2000), and many more.

The CARL phase vocoder and CDP software was later adapted by Richard Dobson (Dobson, 2001) in order to function in real-time. Dobson also developed the VST versions of the CDP Phase Vocoder plug-ins described in Section 3.1.6. The opportunity to learn the intricacies of, and later experiment with these open-source plug-ins, offered an excellent starting point for the development of the Thaw software. The design of this is now documented in Chapter 5.

Chapter 5 SOFTWARE DESIGN

This chapter will focus on the conceptual aspects of the Thaw software development lifecycle. Although many details of the design have naturally evolved and changed during the course of the development, a number of fundamental aspects became fixed from the start of the design process. These include the choice of digital signal processing solution – the phase vocoder had already been chosen for this task, as discussed in Section 3.3. Other such practical aspects of the software development process, as well as the reasoning behind their selection, are now documented in this chapter.

5.1 Platform

The primary operating systems used on modern computers are Microsoft Windows, Apple Macintosh and Linux. All of these platforms are used to some extent by computer musicians and, although Linux is enjoying a growing popularity for computer music purposes, Windows and Mac attract the vast majority of users.

Microsoft Windows was the platform chosen on which to initially develop the Thaw software. Windows was chosen mostly for reasons of convenience; these include the author's pre-existing familiarity with Windows programming, and the fact that Windows enjoys a far greater level of support from the pre-existing software packages used during the development[20] than either Mac or Linux.

Notwithstanding these considerations, the choice of development platform is not central to the completion of this work – all of the aforementioned operating systems mentioned support the same signal processing techniques and sound production capabilities (theoretically, at least) and the potential performance differences attainable across the systems are usually negligible. The

[20] Notably the VST Software Development Kit (Section 5.3) and the CARL phase vocoder (Section 4.3.5).

convenience of using Windows as a development platform allowed the software development work to focus on the important, internal aspects of the design. Furthermore, the 'porting' of the software to accommodate Mac and/or Linux users remains a future possibility (Section 8.3.3).

5.2 Stand-Alone vs. Plug-In.

Sound creation and manipulation software typically takes one of two forms: stand-alone, or plug-in[21]. Stand-alone applications offer the advantage of not requiring a host within which they must perform their functions. Effect hosting applications, musical sequencers and waveform editing tools generally take the form of stand-alone applications. The plug-in format, however, offers large advantages over stand-alone applications, particularly with regard to software whose function is to create or manipulate sound. Such plug-in 'instruments' and 'effects' may be implemented within complex software structures which allow for a high degree of flexibility in the routing of signal and control paths. Effects may be chained together in series (as 'Insert' effects) or in parallel (as 'Send' effects) within the same host application, and may be automated[22].

Audio processing plug-ins are available in a wide variety of formats, which vary with regard to which features and operating systems they support. Such formats include DirectX for Windows, Audio Units (AU) for Macintosh, and Linux Audio Developer's Simple Plug-in API (LADSPA) for Linux. However, the most popular plug-in format (Steinberg GMBH, 2006) in use today is Virtual Studio Technology (VST). For a number of reasons, which are now discussed in Section 5.3 below, it was decided that the Thaw software would take the form of a VST plug-in.

---

[21] A large body of recent audio software development has begun to blur the lines between these formats, however. Increasingly, commercial vendors are making their products available in both stand-alone and plug-in formats. Furthermore, the number of modular music systems is on the increase; these allow for a huge degree of flexibility in the interoperability of different products and the routing of audio signals, further breaking down the traditional host/slave paradigm. Examples of systems which support such modular structuring include Cycling '74 *Max/MSP*, Native Instruments *Reaktor*, Propellorhead *ReWire*, Plogue *Bidule*, and many others.

[22] Automation refers to the 'recording' of parameter values over time, allowing for non-real time, or 'off-line', mixing and editing of music.

5.3 The Virtual Studio Technology Plug-In Format in Detail

The Virtual Studio Technology (VST) plug-in format was first developed by Steinberg in 1996. The format was first incorporated into Steinberg's *Cubase* Digital Audio Workstation (DAW)[23] host software in an attempt to allow for the amalgamation of 'virtual' studio resources (such as effects, instruments, mixers and automation) into the same desktop environment, thereby emulating the structure and functionality of a 'real' studio which typically consists of the same elements in hardware form. In 1997, an SDK[24] for the development of VST-compatible software was released to the public. This open-source kit, as well as a freely-available development licence from Steinberg, meant that hobbyist and commercial software developers began to take up the development of VST-compatible software. Furthermore, VST software is cross-platform, meaning it may be created for Windows, Mac or Linux platforms, a feature which further enhanced its popularity. Given these benefits, VST-compatible plug-ins and hosts numbered in their thousands within several years.

VST plug-ins function in real-time and generally take one of two forms: instruments, which produce a sound (generally using synthesis, sampling or a combination of both); or effects, which modify existing sounds and may be chained together. The Thaw software is an example of a VST effect. VST plug-ins require a host within which they must function. A great number of such suitable hosts exist, allowing for the use of VST plug-ins within a broad and flexible variety of music production contexts. Furthermore, since the release of version 2.0 of the SDK in 1999, VST plug-ins have supported the reception of MIDI messages – this has allowed for further flexibility in the control of VST software, particularly for software synthesisers.

---

[23] DAW – a desktop music production environment. Usually consists of a central host application, such as *Cubase*, which enables recording, editing, arranging, sequencing and playback of audio and MIDI tracks, as well as the capability to use plug-ins.

[24] SDK – Software Development Kit. A collection of resources, including source code and/or precompiled libraries and documentation, aimed at assisting the development of software.

Perhaps the most significant advantage of developing the Thaw software in the form of a VST compatible plug-in has been the ability to focus on the internal and innovative processes of the software – the development of a stand-alone product, or indeed a lesser-supported plug-in format, would have involved spending a significant amount of time on aspects of the software which are not required by the VST standard. A Graphic User Interface (GUI), for example, does not necessarily need to be included when developing a VST plug-in – although the usability and visual aesthetics of a plug-in are generally enhanced by the inclusion of a GUI, the host application supplies a generic set of graphical slider controls if one is not included. Similarly, the VST standard does not require the programmer to communicate with the computer's audio hardware or drivers. Instead, the host simply passes a stream of digital numbers[25] to the plug-in, which may be manipulated as the programmer wishes. The VST SDK provides a framework for the programmer's definition of all the functions and processes which are central to the operation of the VST.

The architecture and language of the VST standard SDK version 2.4, as well as its role in the development of Thaw, will be explored in further depth in Chapter 6.

5.4 Programming Language and Environment

A number of programming languages including Visual Basic, Java, Pascal, Delphi, and even machine/assembly code, have successfully been used in the development of audio processing software. The majority of these, however, suffer from deficiencies such as low efficiency and a lack of support from SDKs. Most of these languages are suited to the development of musical programs which do not involve processing large quantities of digital audio data – for example, notation software or software for handling MIDI signals.

[25] Each of these floating point (i.e. real, non-integer) numbers in the range of -1.0 to +1.0 represents an individual digital audio 'sample' – at CD standard audio quality, the VST will receive 44,100 such samples per channel per second.

The most commonly used languages for developing audio applications are C and C++ – these languages have a wide feature set, produce fast and highly efficient code, and are supported by the majority of audio SDKs. Despite a number of weaknesses such as having a difficult learning curve and a tendency to produce bewildering and esoteric errors, it was decided that the Thaw software would be written using a combination of C and C++[26].

The Microsoft Visual C++ 6.0 IDE[27] was used in order to facilitate the C and C++ software development. Although different IDEs are available, as well as newer versions of Visual C++, Visual C++ 6.0 was chosen as it has a proven track record and benefits from high levels of online support in the development of VST software.

5.5 Implementation Fundamentals and Strategy

The process of VST development began once the goals of the Thaw software development were eventually crystallised, as discussed in Chapter 1, and following decisions on the overall design, as discussed in this chapter. No project timeline was drafted prior to the commencement of development due to the previously unexplored nature of this particular project. However, in accordance with good software engineering principles a detailed project log, excerpts of which are reproduced in Appendix II., was maintained throughout the development process. This, along with a comprehensive backup repository of the software itself, ensured that the development process proceeded at a good steady pace without major setbacks.

The final form of the completed product, including an explanation of the algorithms which are central to its innovative methods of sound transformation, is now documented in the following chapter.

---

[26] The C++ language is an extension of the earlier C language. While C is faster, the object-oriented nature of C++ is generally better for producing programs with any non-trivial degree of complexity (Deitel & Deitel, 2001).

[27] IDE – Integrated Development Environment. A program which integrates the elements necessary for software development, such as text editor, compiler, linker and debugger.

Chapter 6 SOFTWARE IMPLEMENTATION

This chapter documents several of the important algorithms and methods used within the Thaw software. Pseudo code[28] is used to explain much of the Thaw functionality in this chapter; C++ literate readers may consult Appendix I. for C++ code snippets which correspond to the pseudo code in this chapter, or may access the accompanying CD in order to view the entire source code of the project.

The Thaw software utilises three pre-existing code sources which were necessary for the implementation of the plug-in. These consist of the Steinberg VST SDK version 2.4 (Steinberg GMBH, 2006) – see also Section 5.3; an implementation of the CARL phase vocoder which was previously modified by Richard Dobson and the Composers' Desktop Project for real-time functionality (Dobson, 2001) – see also Section 4.3.5; and a Low Frequency Oscillator class by Remy Muller (Muller, 2003).

6.1 Thaw Sound Transformations and their Parameters

Thaw allows controls over its functionality via eleven parameters which may be manipulated in real-time by the user using either a computer mouse or MIDI controller[29], or by the host software using automation. These parameters specify the nature and magnitude of the sound transformations executed by the software. The function of each of these parameters and their subsequent effects is now explained:

- **Dry/Wet mix:** this parameter allows the user to cross-fade the frozen/thawed or otherwise modified 'wet' sound created by the plug-in

[28] Pseudo code – an explanation of software algorithms in plain terms, without the use of a specific computer language. In this chapter, the pseudo code contains snippets of Thaw code – class, object, method and variable names, as well as other key words which appear in the C++ code itself, are highlighted here in the `Courier New` font.

[29] The ability to use MIDI controllers for changing Thaw's parameters is dependent on the capabilities of the host software.

with the 'dry,' unaffected sound which is being fed to the plug-in from the host. This parameter is not strictly necessary as most hosts will allow the plug-in to be used as a 'send' effect, thereby allowing these sounds to be mixed manually outside the plug-in. The inclusion of a Dry/Wet mixer within the plug-in, however, allows for easier structuring within many modular music production environments and was therefore included following requests from several beta testers (Appendix III.). It is worth noting at this point that Thaw's phase vocoders are constantly active, whether or not the freeze effect is On or any other effect is active – this results in slight audible artefacts (as mentioned in Chapter 4) when the Dry/Wet mix is set to 100%. Furthermore, because of a slight latency in the operation of the phase vocoder, a very short echo is audible when the Wet and Dry signals are mixed together and the effect is Off. This is not a major issue, however, as the user would have little reason to blend these signals when the effect is Off.

- **Off/On:** this parameter allows the 'freeze' functionality to be switched on. When this happens, the phase vocoder captures a single frame of frequency/amplitude data[30] and resynthesises this instead of the usual overlapping analysis windows. This important functionality is central to the operation of Thaw and contributes strongly to its originality – the algorithm is explained in detail in Section 6.4.2. The user may still manipulate the (non-frozen) sound with all of the other parameters, even when this parameter is set to Off.

- **Pitch Shift**: this simple pitch shifter raises the values of the frequency components in the phase vocoder analysis frames. It does not preserve the relative relationships between the harmonic components of the sound, as most conventional pitch-shifting algorithms do, thereby allowing this function to create unusual sounds which bear little resemblance to the non-pitch shifted original. Furthermore, this function is cyclical – the pitch-

[30] In Thaw, the frame is approx. 20ms or 1024 samples long by default; therefore, when the freeze effect is On, the listener is essentially hearing a synthesised reproduction of the average timbral characteristics of a 20ms segment of sound.

shifted sound ‘wraps around’, returning to the original pitch and allowing for the creation of a sound akin to Shepard tones[31].

- **Spectral Degradation**: this feature, which is made possible by the unique nature of the phase vocoder, takes a user-specified proportion of random amplitude components of the spectral data and sets them to zero. This effectively cuts a proportion of random frequencies from the synthesised sound and produces a noisy, warbling effect. This parameter is suitable for producing noise-based drones.
- **Low Frequency Oscillator (LFO)**: an LFO is a continuously modulating value corresponding to one of several wave shapes. While oscillators of much greater rate/speed are commonly used to produce audio tones, LFOs are typically used to modify existing sounds; often in order to produce tremolo, vibrato or similar effects. In Thaw’s case, the LFO is used to modify the frequency components of the phase vocoder data, producing rising and falling pitches. The operation of the LFO depends on the following parameters:
    - **LFO Modulation Factor**: this parameter specifies the proportion of frequency components which are to be affected by the LFO. At low values, its effects are minimal. At the Modulation Factor’s maximum setting, the LFO affects all of the frequency components of the sound.
    - **LFO Wave Shape**: the user may select one of a common number of wave types in order to vary the nature of the movement of the LFO – the options consist of sine, triangle, sawtooth, square and exponent waves. These wave shapes were included in the pre-existing LFO algorithm; however, the author has extended these by implementing a random LFO wave shape. With the proposed drone aesthetic in mind, this random wave shape is intended to allow the user to create a more organic sound, thus offering an alternative to the other periodic and

[31] Shepard tones are an auditory illusion that gives the impression of an infinitely rising or falling pitch.

rhythmic wave shapes. The original algorithm for producing the random wave-shape is documented in Section 6.4.5.

- **LFO Rate**: the speed/frequency at which the LFO operates. As Thaw is intended to produce slowly-moving sounds, the maximum LFO speed is limited to 24Hz (i.e. 24 cycles per second); this allows very fast vibratos and a slight ring modulation effect, however, user control for this parameter is logarithmic to allow for more precision when selecting very low frequencies. This logarithmic relationship is described in more detail in Section 6.4.5.
- **LFO Depth**: the degree to which the LFO will alter the frequencies in the phase vocoder analysis windows, corresponding to the pitch of the sound produced. The LFO Depth is essentially a more flexible, automated version of the manual Pitch Shifter parameter.

- **Spectral Filter**: allows the user to boost or cut a specified band of frequencies. Although the functionality of this feature is similar to that of a parametric equaliser, this Spectral Filter is not quite the same as it modifies the spectral data produced by the phase vocoder as opposed to the time-domain signal which is affected by an equaliser.
    - **Spectral Filter Centre Frequency:** specifies the frequency in the audible range around which frequencies will be cut or boosted. Like the LFO Rate parameter, this parameter is logarithmically controlled – this is to allow a greater degree of control in the lower, musically important end of the audible spectrum.
    - **Spectral Filter Gain:** this parameter specifies whether the sound is to be cut or boosted by the filter. Negative values indicate a cut, positive value indicate a boost.
    - **Spectral Filter Bandwidth:** specifies the range of frequencies around the centre frequencies which will be affected. This parameter is analogous to the ‘Q’ parameter found on most parametric equalisers.

6.2 Software Structure

6.2.1 File Structure

Figure 13 (below) illustrates the hierarchy of the files used in Thaw's Visual Studio project. The nature of the interaction between the classes in these files is now discussed in Section 6.2.2.

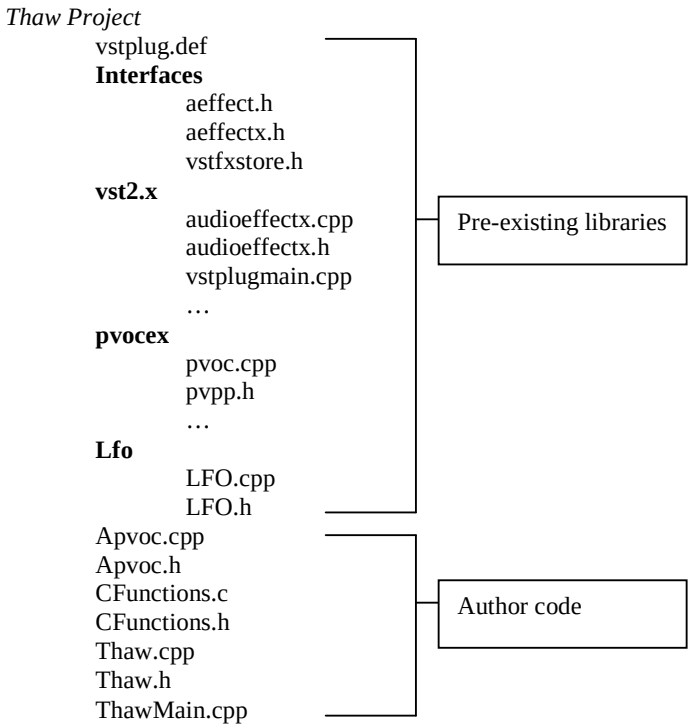


Figure 13: The Thaw project's file structure. Bold names indicate folders.

6.2.2 Class Structure

Like the majority of similar modern software products, an object-oriented design strategy was applied to the development of Thaw. While this design principle assists in the manageable construction of sizeable software products and the relatively easy integration of pre-existing software components, it may not be familiar to the uninitiated reader. Furthermore, as a wider discussion of object-oriented design principles is outside the scope of this work, this section will not attempt to explain the Thaw software architecture in formal detail.

Rather, it will attempt to provide the reader with a good high-level understanding of how the different elements of the software integrate with each other and function in real-time.

As previously mentioned, Thaw is constructed using a number of components. These include a large amount of original code, code derived from the Steinberg VST SDK, instances of a modified CARL phase vocoder, and a simple LFO class which has been extended to incorporate a new wave shape specifically for Thaw. The nature of the interaction between these components is outlined in Figure 14:

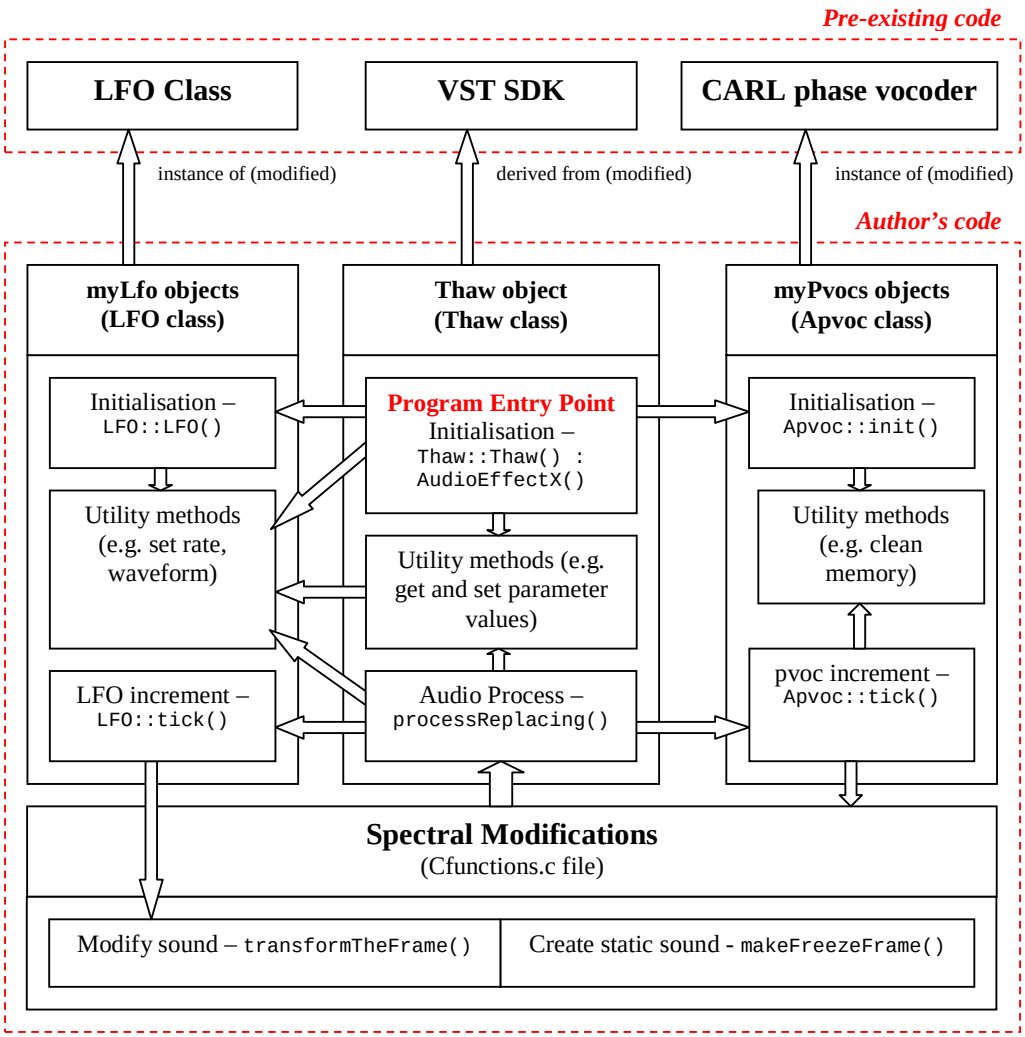


Figure 14: The relationships between the components of the Thaw software[32].

[32] This diagram is an informal and non-exhaustive illustration of the Thaw software architecture and its components. Interested readers are encouraged to read the relevant C++ header (.h) files on the accompanying CD in order to gain an insight as to the formal structure of the Thaw software.

An object named `Thaw` is created upon the opening of an instance of the plug-in and is therefore the entry point of the software. It is derived from a base class defined in the VST SDK – this means that not only does it provide access to the functionality of the SDK in order to perform all of the necessary interaction between the host and the plug-in, but it also provides the ability to extend the capabilities of the SDK to suit the intended purposes of the software.

Upon its creation, the `Thaw` object is called by the host software. It creates and initialises two instances of the `Apvoc` class, named `myPvoc1` and `myPvoc2`. Each of these incorporates phase vocoders which analyse and resynthesise the audio streams passing through each of two audio channels. For every numerical sample received by `Thaw` from the host, methods called `tick()`, members of `myPvoc1` and `myPvoc2` are called, thereby incrementing the phase vocoders.

`Thaw` also creates one instance of the `LFO` class named `myLfo`. `myLfo` is similarly called every time `Thaw` receives a new sample. `myLfo` is initialised and thereafter returns a value back to `Thaw` representing a low frequency wave of variable shape. This value may be used as desired; its use in the Thaw software is explained in Section 6.4.4.

The methods/functions of each of the `Thaw`, `myLfo`, and `myPvocs` objects may be roughly subcategorised as Initialisation, Utility and Processing. Upon the creation of each new object, Initialisation methods are called – in Thaw's case, this occurs when the plug-in is initially loaded. Utility methods may be called as required – for example, in the case of a user-defined change of plug-in parameters or a change in preset/program. Processing methods are called in real-time and are therefore central to the processing of the real-time stream of digital data received by the host. All other Processing methods are called from `Thaw`'s `processReplacing()` method.

Several of Thaw's most important methods will be explained in the following two sections. The source code for all of these algorithms is available in Appendix I.

6.3 Initialisation Methods

6.3.1 Thaw Initialisation

Method name: `Thaw::Thaw(...) : AudioEffectX(...)`

The constructor[33] for the `Thaw` class, which is derived from the fundamentally important SDK-defined `AudioEffectX` class[34], performs the following functions upon the loading of the plug-in:

Algorithm:

- Get the sample rate from the host.
- Create and initialise a `ThawPrograms` object (for storing plug-in presets[35]).
- Initialise all of the plug-in parameters to their default values.
- Set the number of inputs and outputs.
- For each audio channel, create and initialise an object of type `Apvoc`. Pass the sample rate, FFT length, overlap factor and window type to each object. Perform some error checking.
- Set the default values for `freezing` and `freezeOnce` to tell the phase vocoders that the plug-in is not yet in 'freezing' mode.
- Create an object of type `LFO` – initialise the rate, phase and waveform.

6.3.2 Phase Vocoder Initialisation

Method name: `Apvoc::init(sampleRate, fftlen, overlap, mode)`

This method is called upon the creation of the `myPvocs` objects, immediately following the initialisation of the `Thaw` object. It prepares the phase vocoder for its subsequent task of analysing and resynthesising the incoming audio samples:

---

[33] Constructor – a method which is called when an object is first created. The constructor is typically used for initialising an object and its member functions.

[34] `AudioEffectX` is essentially the host's means of communicating with the plug-in.

[35] The VST plug-in architecture allows for the definition of preset parameters. This is often useful to allow users of many types of plug-ins in getting started. However, it was decided that Thaw would not include presets, due to its experimental nature and suitability for nurturing experimentation from the user.

Algorithm:

- Perform some error checking.
- Initialise some variables indicating the size and overlap of the phase vocoders and the number of frequency bins[36] in each phase vocoder frame. Define buffers to hold phase vocoder bins and initialise them to zero.
- Create CARL phase vocoder objects; one each for analysing and resynthesising the streaming audio. Initialise them and perform some error checking.
- Initialise some user-defined flags which will later be used to freeze/unfreeze the sound.

6.4 Signal Processing Algorithms

6.4.1 The `processReplacing()` method

Method name: `Thaw::processReplacing (inputs, outputs, sampleFrames)`

This method is called each time the plug-in receives an audio sample from the host and therefore is the fundamental driving force behind all subsequent signal processing algorithms. `processReplacing()`is derived from a base class which is defined by the VST SDK and is necessary in all VST plug-in effects. It operates in real-time, and great care must therefore be taken to make its operation as efficient as possible:

Algorithm:

- Create four buffers: for storing the input and output samples in each of two audio channels. Assign these buffers to the inputs and outputs used by the host.
- **If** the `OnOff` parameter is set to on:
    - `freezing` is true. `freezeOnce` is false. [1]
- **Else**:
    - `freezing` is false. `freezeOnce` is true. [2]
- **End If**
- **If** the `LFO` shape is changed by the user:
    - Change the `LFO` to the specified Shape.
- **End If**
- **If** the `myPvoc` objects are functioning correctly:

[36] Frequency bins are pairs of data representing the amplitude and frequency of spectrally analysed signals, as discussed in Chapter 4.

- **While** samples are received from the host:
    - Increment the LFO - `LFO->tick()`. [3]
    - Request the next sample from the host.
    - Increment the `myPvocs` objects and receive a value in return. [4]
    - Multiply this returned value by the `CrossFade` Factor and send to the host. [5]
- **End While**

- **End If**

Notes:

[1] If the user specifies that the freezing function is currently on, a variable named `freezing` is told so. The `freezeOnce` variable is set to false, so the phase vocoders will simply resynthesise the original sound. This functionality is explained in further detail in Section 6.4.2, below.

[2] If `freezing` is true, the `freezeOnce` variable will inform the phase vocoder to continuously resynthesise a single captured frame of audio rather than stepping through the audio in overlapping frames in its usual manner (as explained in Chapter 4). This algorithm is fundamental to Thaw's freeze functionality and will be explained in greater detail in Section 6.4.2.

[3] The `LFO` object will return a value which may be used as desired – in Thaw's case, this value will be sent to the `transformTheFrame()`method via the `myPvocs->tick()` method, where it will perform modifications on the frequency components of the phase vocoder data.

[4] `myPvocs->tick()` is called in real time. It receives the following values from `processReplacing()`:

- The audio sample which is to be processed.
- The `Freezing` flag.
- Values relating to the functionality of the Spectral Degrader, Spectral Filter and LFO effects – a comprehensive list of these parameters is detailed in Section 6.4.4.

Tick passes a value back to `processReplacing()` in return.

[5] This returned value is multiplied with the original input signal using a cross fader (corresponding to the Dry/Wet parameter) and sent back to the host.

6.4.2 Phase Vocoder Incrementation and Freeze Effect

Method name: `Apvoc::tick (insamp, freezing, lfoFac, lfoModFac, pitchShift, spectDeg, notchCF, notchGain, notchQ)`

This method is responsible for filling the phase vocoder buffers (or frames) with incoming samples, analysing each frame when it is filled, calling the function which performs spectral modifications (as detailed in Section 6.4.4, below) on the information in these frames, 'freeze' a single frame if told to do so, resynthesise the spectral data contained in the frames, and pass audio samples back to the `processReplacing()` method, and hence the host, one by one:

Algorithm:

- **If Not** `freezing:`
    - `freezeOnce` is set to true. [1]
- **End If**
- Increment the current position in the output buffer.
- **If** the position of the input buffer is equal to the size of the analysis frame: [2]
    - Using the phase vocoder, generate an analysis frame of spectral data from the input buffer.
    - Reset the current frame position to zero.
    - **If** the `freezing` effect is currently on:
        - **If** `freezeOnce` is true:
            - Call `makeFreezeFrame()`. [3]
            - Set `freezeOnce` to false.
        - **End If**
        - Call `transformTheFrame()`, passing the frozen frame. [4]
        - Using the phase vocoder, resynthesise this frozen analysis frame and add the results to the output buffer.
    - **Else**
        - Call `transformTheFrame()`, passing it the streaming, non-frozen frame. [5]
        - Using the phase vocoder, resynthesise this analysis frame and add the results to the output buffer.
    - **End If**
- **End If**
- Increment the input buffer, filling it with the next sample from the host.
- Return the current value in the output buffer to the host.

Notes:

[1] If the freezing effect is currently off, this flag needs to be set to true in order to continuously prepare a buffer for when the effect is switched on.

[2] The code within this **If** condition (which comprises the majority of the code in this function) is only executed when a new full analysis frame has been filled with samples from the host.

[3] The `freezeOnce` flag will only be true whenever the freezing effect is initially switched On. A single frame of analysis data is captured using the `makeFreezeFrame()` function.

[4] This frozen frame is passed to the `transformTheFrame()` function, which applies all of the subsequent spectral modifications – Section 6.4.4, below.

[5] If the freezing effect is NOT currently on, the continuous stream of overlapping non-frozen analysis frames is passed to `transformTheFrame()`, thereby allowing the user to apply effects to the non-frozen sound if wished.

6.4.3 Creating the Frozen Frame

Method name: `makeFreezeFrame(*streamingFrame, *freezeFrame, nbins)`

This method is called each time the Off/On parameter is set to On. It is only called at the instant the parameter is switched On, and thereafter cannot be called again until Off/On has been set to Off and back On again. Its function is to copy the contents of the last phase vocoder frame which has been processed (`streamingFrame`) into a new buffer called `freezeFrame`. The `freezeOnce` and `freezing` flags, discussed in previous algorithms, will then determine when this frozen frame, as opposed to the usual stream of continuously overlapping frames, will be resynthesised. The algorithm is quite simple; it simply copies the contents of one buffer into another:

Algorithm:

- Create variables `a`, `a1`

- **For** `a = 0, a1 = 0; a < nbins; a++, a1+=2:` [1]
    - `freezeFrame[a1] = streamingFrame[a1]` [2]
    - `freezeFrame[a1+1] = streamingFrame[a1+1]` [3]
- **End For**

Notes:

[1] Two indices are used by the **For** loop due to the interpolation of the values in the frequency bins; i.e. amplitude and frequency values are stored one after the other.

[2] Amplitude values are copied individually from the phase vocoder analysis frame into the frozen buffer.

[3] Similarly, frequency values are copied here.

6.4.4 The Sound Transformation Algorithm

Method name: `transformTheFrame(*inFrame, *outFrame, lfoFac, lfoModFac, pitchShift, spectDeg, specFiltCF, specFiltGain, specFiltQ, nbins)`

`transformTheFrame()` is one of only two C (as opposed to C++) methods used within the Thaw software. As this method is called very frequently (every 1024 samples) and performs multiple modifications on hundreds of analysis bins every time it is called, efficiency is of the utmost importance – the C language is hence more suitable in this context.

`transformTheFrame()` is similar in operation to the `makeFreezeFrame()` algorithm discussed above, but is much more complex. It takes an input analysis frame – which may be either the frozen frame or the normal streaming sound, as specified by the user – modifies it, and returns the modified frame. It receives the following parameters from `Apvoc::tick()` (most of which originated from the `Thaw` object and hence the user-controllable parameters) when it is called:

- `*inFrame` – a pointer to a buffer where the current analysis frame is stored

- `*outFrame` – a pointer to a buffer where the modified analysis frame will be stored.
- `lfoFac` – the magnitude of the current LFO value (depth multiplied by the current rate factor received from the `LFO` object).
- `lfoModFac` – the LFO modulation factor.
- `pitchShift` – the pitch shifting factor.
- `spectDeg` – the spectral degradation factor.
- `specFiltCF` – the spectral filter centre frequency.
- `specFiltGain` – the spectral filter gain.
- `specFiltQ` – the spectral filter bandwidth.
- `nbins` – the number of bins in the analysis frame.

Algorithm:

- Define variables `a` and `a1` which will be used in the **For** loop.
- Define a variable `shiftFac = pitchShift * 170`. [1]
- Define a variable `multFac = (1 – lfoModFac) * 100`. [2]
- Define a variable `degFac = (1- spectDeg) * 100`. [3]
- Define a variable `halfFilterWidth = (nbins*(1-specFiltQ)/2)`. [4]
- Define a variable `filterFacHi = (specFiltCF*nbins) + halfFilterWidth`. [4]
- Define a variable `filterFacLo = (specFiltCF*nbins) – halfFilterWidth`. [4]
- Define a variable `filterFacGain = specFiltGain * 2`. [4]
- **For** `a = 0, a < nbins, a++`: [5]
    - **If** `a < 5 * rand()%(101-degFac)`: [6]
        - `outFrame[a] = 0`. [7]
    - **Else If** `a > notchFacLo && a > notchFacHi:`
        - `outFrame[a] = inFrame[a] * notchFacGain` [8]
    - **Else**
        - `outFrame[a] = inFrame[a]` [9]
    - **End If**
    - **If** `a < 5 * rand()%(101-multFac)`: [10]
        - `outFrame[a+1] = inFrame[a+1] + lfoFac + shiftFac` [11]
    - **Else**
        - `outFrame[a+1] = inFrame[a+1] + shiftFac` [12]
    - **End If**
- **End For**

Due to the complexity of the preceding algorithm, C++ literate readers may wish to consult Appendix I. or the relevant file on the accompanying CD in order to gain a better insight into its operation.

Notes:

[1] This simply scales the pitch shifting value so that the pitch will return to unison when the parameter is set to 100%.

[2] The use of this variable will be explained in note [10], below.

[3] The use of this variable will be explained in note [6], below.

[4] These variables simply calculate how many bins on each side of the filter's centre frequency will be modified by the filter gain parameter.

[5] This **For** loop iterates through all of the bins in the analysis frame. Its operation is similar in operation to that of the `makeFreezeFrame()` function's **For** loop.

The lines of code from note [6] to note [9] perform modifications on the **amplitude** components of the spectrally modified sound:

[6] This condition selects random amplitude bins, the proportion of which is decided by the `degFac` variable – this action corresponds to the **Spectral Degradation** effect.

[7] If an amplitude bin is randomly selected, its value is set to zero.

[8] If an amplitude bin has not been selected, it may still be affected by the **Spectral Filter** – if the bin falls between the `filterFacLo` and `filterFacHi` parameters, this modification (i.e. the cutting or boosting of a specified range of amplitudes) is performed here.

[9] If neither the spectral degradation nor spectral filter modifications apply to this bin, it is copied to the output frame verbatim.

The modifications carried out from note [10] to note [12] correspond to changes in the **frequency** components of the analysis frame:

[10] This selects random frequency components, the proportion of which are decided upon by the `multFac` variable – this will decide how much of the

sound is to be affected by the LFO and corresponds to the **LFO Modulation Factor** parameter.

[11] If this frequency bin is to be affected by the LFO, `lfoFac` is added. `shiftFac`, the **Pitch Shifter** factor, is also added.

[12] If the bin is not affected by the LFO, just the `shiftFac` is added.

6.4.5 Further Noteworthy Algorithms

This section briefly describes several algorithms which, while not fundamental to Thaw's functionality, have been implemented in order to improve or enhance the aesthetics or usability of the plug-in.

Random LFO Wave Shape

(Excerpt from) Method name: `LFO::setWaveform(index)`

This method is called when the user changes the LFO wave shape to Random. It produces a 256 point wave table using a 'drunken walk' algorithm[37] which was written by the author to function in a suitable manner within Thaw. This table is accessed by the `LFO::tick()` algorithm, thereby passing its points sequentially to the `transformTheFrame()` method at the specified rate and depth.

- Declare a variable `i`. [1]
- Declare a variable `place` and initialise to zero. [2]
- **For** `i = 0; i < 256; i++`: [3]
    - **If** a random value from 0-4 is equal to zero: [4]
        - `place` is assigned the value of `place` plus a random value: either zero, -0.1 or + 0.1. [5]
        - `table[i]` is assigned the value of `place`. [6]
    - **End If**
- **End For**
- `table[256]` is assigned the value of zero. [7]

Notes:

[37] The drunken walk algorithm starts at a particular point (in this case zero) and thereafter takes a random 'step' to the left or right – in the case of this LFO, it will produce a continuous wave shape which, while somewhat random, may not become so noisy as to distort the sound.

[1] `i` will be used to increment the **For** loop, below.

[2] `place` will be used to store the current position of the wave table points.

[3] This **For** loop iterates through the wave table, filling it with new points.

[4] This condition is in place to ensure the wave table is not filled at every new point, hence reducing noise.

[5] This is the central part of the drunken walk algorithm. The previous value of `place` is randomly either increased, decreased or untouched.

[6] The next point in the table is filled with the current value of `place`.

[7] The last point in the table is reset to zero.

Logarithmic Scaling of LFO Rate

(Excerpt from) Method Name: `setParameter(index, value)`

As Thaw is intended to create slowly moving sounds, it was decided that the rate of the Low Frequency Oscillator should be kept low. However, higher LFO rates are capable of producing ring modulation[38] effects and hence would add another dimension to Thaw's sound modification capabilities.

A short function was therefore written in order to logarithmically scale the user-controllable LFO Rate parameter to a range of values which would be capable of allowing higher LFO rates (up to 24Hz) while offering a much finer degree of control at the slower LFO rates (below approx. 4Hz) in order to produce slow drones. Another function – which may be examined in Appendix I. – logarithmically scales the Spectral Filter Centre Frequency parameter in a similar manner, thereby allowing more control at the lower end of the audio spectrum.

Function:

- Declare a variable `logVal` and assign it the LFO Rate `value` currently specified by the user.
- **If** `logVal` equals zero:

[38] Ring modulation is an unusual audio effect which is achieved by combining two audio signals, one of which is usually a simple wave form, such as those which Thaw's LFO is capable of producing. At low rates (below 20Hz) the effect is perceived as tremolo or vibrato. However, at much higher rates, an unusual and distinctive bell-like sound may be achieved.

    - Assign a very low non-zero value to `logVal`. [1]
- **End If**
- Thaw's LFO Rate is assigned the value `(-log10(1 – logVal)) * 5`. [2]

Notes:

[1] The `log10()` mathematical function requires a non-zero value; this line of code will replace any zero value with an extremely low non-zero value.

[2] `log10()` is used here to provide logarithmic scaling and is inverted and scaled in order to provide the 0-20Hz range required.

6.5 Further Implementation Particulars

6.5.1 Phase Vocoder Parameters

During the development process, it was hoped that Thaw's functionality would allow the user to select different phase vocoder window types, sizes and overlap factors (as specified in Chapter 4) via several additional parameters. However, due to the nature of the CARL phase vocoder used, it was not possible to allow the user to change these options after the phase vocoder has been initialised, i.e., immediately after the initial loading of the plug-in. Nevertheless, changing these parameters within the source code itself is a trivial matter: by default, the phase vocoder uses a 1024 sample long Hanning window with an overlap factor of four. Consequently, it is envisaged that Thaw will be compiled and distributed with a number of variations of these parameters. Furthermore, the software has been structured to allow users to easily define their own parameters using `#define` flags. This flexibility should allow users of the software – whose source code will be made openly available – to choose which combination suits best, depending on their computational resources and the types of sound they wish to process.

6.5.2 Software Testing and Refinement

Thaw was developed on a Dell Inspiron 6000 PC running Windows XP Service Pack 2 with Intel Centrino 2.2GHz processor, 1 GB RAM, ASIO audio drivers and E-MU 1616 audio hardware. During this process, Thaw was continuously refined, debugged and tested by the author on an ongoing basis using Tobybear *MiniHost* v.1.64 and Steinberg *Cubase SX* v.2. *MiniHost* is a simple application whose sole aim is to host VST plug-ins; its simplicity and speed were the main reasons it was chosen as the primary alpha test host platform for Thaw. Although *MiniHost* allowed for speedier testing, *Cubase* offered the opportunity to test the plug-in in a much more common and popular music production environment which offers multiple track sequencing, multiple plug-in instances, automation, and other features which would test the plug-in's capabilities and reveal bugs. Thaw was also tested by the author in Ableton *Live*, Cycling '74 *Max/MSP* and Jørgen Aase's *energyXT*.

Approximately one month prior to the submission of Thaw, copies of the plug-in were sent to a small group of voluntary beta testers. This group were asked to try the plug-in in a variety of different hosts and test machines and to test its functionality and reliability, based on a number of guidelines provided by the author. This beta testing process and its outcome is detailed in Appendix III.

During beta testing, the ongoing development of the plug-in temporarily ceased and the author used the opportunity to further test the software in a real music production environment. This process of using Thaw in composition is documented now in Chapter 7.

Chapter 7 CREATIVE IMPLEMENTATION

7.1 Concept and Goals

A short composition entitled *What Goes Around* is presented. The piece, which makes abundant use of several instances of the Thaw plug-in, was written under the influence of both practical and creative concerns.

From a practical point of view, the composition of this piece has had several benefits. Firstly, it is intended to be a demonstration of the sound-producing capabilities of Thaw. In this regard, it was composed almost exclusively using automation of the Thaw plug-in and intends to demonstrate the scope and breadth of sounds which Thaw is capable of producing and manipulating.

Secondly, as its composition was undertaken in parallel with the latter stages of development of the Thaw software, the composition process served to thoroughly test the plug-in in the type of DAW environment which is regularly used by a large majority of computer musicians. Furthermore, this composition was in its development stages during the final stages of alpha testing and during the entire process of beta testing, thus offering the author the unique ability to alter Thaw's source code and recompile the plug-in whenever a specific aesthetic was required or where a previously undetected bug was uncovered. Thaw's final form therefore evolved partially as a result of the personal experience of using it in a real computer music production environment.

From a creative point of view, *What Goes Around* may be interpreted as a continuation of the drone-type aesthetic which has been intertwined within many and varied types of music throughout history, as discussed in Chapter 2. Although the composition's function as an expression of artistic statement is secondary to its role as a demonstration of Thaw, it is hoped that the lack of usual musical traits such as rhythm, tonality and thematic development are

compensated for by a wealth of unique timbres and the meta-music[39] which results from interaction between the textures of the component sounds, thereby providing the piece with artistic and aesthetic merit as well as a purely functional purpose.

7.2 Composition, Production and Post Production

What Goes Around was composed using Steinberg's *Cubase SX* DAW software. Three audio tracks were used; each of these used an instance of Thaw as an insert effect plug-in. No other plug-ins were used in *Cubase* during this process of composition – this decision, along with the decision to make sparse use of audio source materials, was taken in order to make the composition as transparent as possible in revealing the modifications performed by Thaw.

The source material for the composition consisted of a cello sample, a triangle sample, and a very brief quotation from Beethoven's *Moonlight Sonata* for solo piano. Each instance of Thaw was used to perform sound modifications on its respective audio sample throughout the duration of the piece; a frozen snapshot of each instrumental sample was taken in the opening seconds of the composition and was manipulated with Thaw's other effects thereafter until the end of the piece. This resulted in the creation of consistent yet constantly evolving drones. These three samples were carefully chosen in order to present as varied an audio spectrum as possible to the plug-ins.

Once the process of composing using automation was complete, each of the three audio tracks was exported in a raw form and the post production process began. *What Goes Around* was edited, mixed and mastered in surround sound using Apple's *Logic Pro*. The use of post-production effects was kept to a minimum; again, this tactic aimed to allow the listener to perceive the effects of the Thaw software without distraction and hindrance from other effects. Some

[39] Meta-music – may be described as the rhythms, harmonies, overtones and other musical traits which unintentionally emerge at the 'surface' of a piece of music as a side effect of the interaction between the intended, structured musical components. Meta-music is especially noticeable, even intentionally applied, in the minimalist genre and in spectral music.

reverberation was added to the mix on a 'send' effects bus, and equalisation was added to each track in order to remove some of the noisy high-frequency artefacts caused by the phase vocoder. A noise reduction plug-in was also used, in order to improve the quality of the unfrozen *Moonlight Sonata* track at the end of the piece.

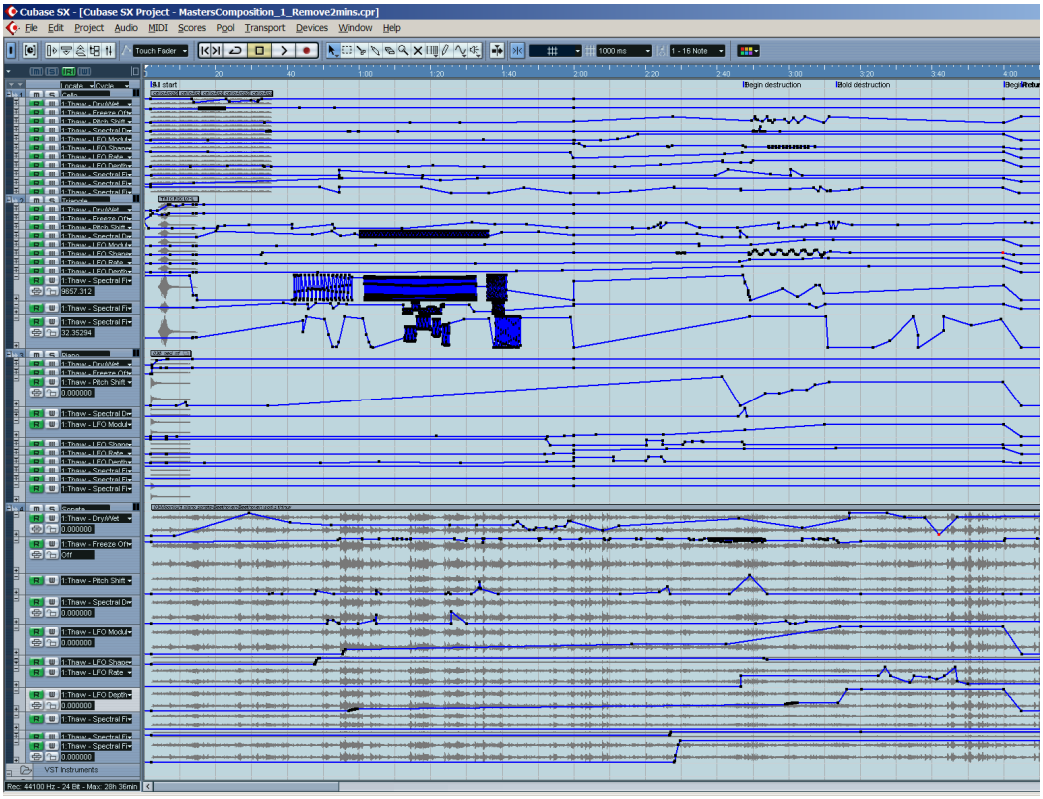


Figure 15: A screenshot of the *Cubase* production environment in which *What Goes Around* was composed. The horizontal blue lines represent the automation curves which comprise almost the entire composition.

7.3 Aesthetic

What Goes Around begins with a slow fade-in revealing three concurrent, spatially distributed and largely static drones. Over several minutes, Thaw's effects are used to linearly change the drones from largely static through a variety of stages which are mainly characterised by different LFO settings[40] and

[40] In particular, the Random LFO shape (which was specifically written with the drone aesthetic in mind) offers a highly organic feel to the composition by creating random, jerky pitch changes at the specified magnitude.

pitch changes. The spatial distribution of the drones is varied from time to time by means of surround panning, and each track's dynamic level occasionally changes, usually in order to reveal noteworthy sounds in certain drones at certain times.

The magnitude of the effects is gradually increased over time; continuously rising pitches, rising levels of distortion and ever-increasing LFO rates lead to an agitated, 'wall of sound' type aesthetic by the fourth minute of the composition. At approximately this point, all effects – except for the fundamental 'freeze' effect – are abruptly removed, and the drones again become static. This aesthetic is sustained for approximately one minute, during which more of the meta-music caused by the interacting drones becomes apparent, lending a gentle, pulsating feel to the music. At the end of the composition, a final pitch change leads seamlessly into the 'unfreezing' of each drone, thereby allowing the listener to recognise the instrument which was used to initially create each drone.

Chapter 8 CONCLUSIONS AND FUTURE POSSIBILITIES

This chapter will summarise the sonic and functional performance of the Thaw software. The extent to which the goals of the work were achieved will then be discussed. Finally, proposals for future developments and extensions to Thaw, as well as outlines of several broader conceptions based on this work, will be presented.

8.1 Observations on Sonic Performance

Regarding the aim of creating completely static and unmodulating 'snapshot' sounds, as discussed in Chapter 1, Thaw's performance ranges from acceptable to highly convincing. The end result depends to a large extent on the phase vocoder's attributes; that is, by varying the choices of window type, length and overlap factor – often at the expense of computational performance – the plug-in's resulting aesthetic will change. The source sound's spectral characteristics are, however, a much greater factor in this regard. Well-pitched monophonic sounds may be 'frozen' with excellent results; a musical note may seamlessly and convincingly be sustained indefinitely where a listener would normally expect it. Polyphonic sounds occasionally do not successfully convey the original impression of harmony or pitch in the resynthesised signal. However, amplitude and timbre are almost always successfully preserved and, thanks to the phase vocoder's use of a bank of oscillators rather than a looping technique, the impression of periodicity is avoided.

As a tool for the not-quite-static manipulation of drones, Thaw introduces a number of original effects which serve to give it character, or to give it a 'signature' sound. The Low Frequency Oscillator may be used either to discreetly and slowly modulate certain frequencies of a sound, to randomly send its components askew, to generate pulsing meta-rhythms, or, at high rates, to introduce distortion and ring modulation into the sound. The Spectral Filter may

accentuate certain frequency bands and create barely audible melodies through slow manipulation of its parameters. The Pitch Shifter may be use to gradually skew the timbre of a sound until it is beyond recognition, or, in conjunction with the Dry/Wet mix, introduce the phenomenon of rhythmic 'beats' which result from simultaneously playing closely adjacent pitches. These effects, as well as the others included with Thaw, may be used either alone, in multiple instances, or in combination with other plug-in instruments and effects, either in off-line production contexts – as demonstrated by *What Goes Around* – or in live performance situations.

8.2 Achievement of Goals in Summary

Overall, the goals of this work were achieved. The Thaw software, incorporating a phase vocoder and a variety of original effects, has successfully been implemented. A composition which serves to demonstrate Thaw's use as a tool for drone creation and sound design has been presented.

It has been demonstrated that the Thaw software is an original, distinctive and capable means of producing static and drone sounds and manipulating them in a number of innovative ways which, heretofore, have not been available to computer musicians. Furthermore, the fact that the effect is encapsulated in a VST plug-in format offers novice or inexperienced musicians to use its functionality in an environment with which they are familiar.

Perhaps most significantly, as demonstrated by the review of existing products in Chapter 3, Thaw is believed to be the only phase vocoder-based product whose central goal is the creation of drone-like sounds. This achievement is highlighted by the composition of *What Goes Around* and by the emergence in the software of a distinctive, yet highly drone-based and characteristic aesthetic, as observed in Section 8.1, above.

Notwithstanding these developments, Thaw is not intended to be a definitive, all-encompassing solution for the creation of static and drone sounds. Indeed, it

is likely that many experienced computer musicians or sound designers would prefer to utilise ‘home brewed’ methods in order to achieve a favoured aesthetic.

One of the most significant aspects of the Thaw development has been the opportunity to explore and further develop the power of the phase vocoder. As previously mentioned, it is widely acknowledged that the phase vocoder’s sound modification abilities have only partially been explored; although several original effects which harness this power have been implemented in this work, it is now even more apparent that much more can be achieved, particularly where a drone aesthetic such as Thaw’s is concerned. Following a number of suggestions on how the existing Thaw product may be expanded in Section 8.3, Section 8.4 will propose a number of conceptual innovations which would take advantage of these facets of the phase vocoder, thereby involving significant changes to the product.

8.3 Further Proposed Developments to Thaw

It is anticipated that further developments to the version of Thaw presented here will soon be undertaken; some of these are now outlined. These developments would not require a significant change to the structure of the existing product.

8.3.1 User Interface

During the latter stages of development, the development of a Graphic User Interface (GUI) for Thaw was undertaken in the hope that the usability and visual aesthetics of the product could be enhanced. Due to time constraints and unforeseen programming-related issues, this GUI was not completed to a satisfactory standard prior to submission. This is not considered a major issue as hosts will automatically assign suitable graphical slider controls for any VST where a GUI is not supplied. However, these host-assigned GUIs often do not offer an ideal user interaction experience; for example, a button-type control for

Thaw's Off/On parameter would be much more suitable than the current slider-type control.

More complex graphical controls are also possible. It is widely recognised that control of music software via the computer mouse is largely restrictive, one-dimensional and is generally a poor medium for conveying musical expression. Although control of VST parameters is also available via MIDI and automation, a number of mouse-based control innovations should greatly improve the usability experience of Thaw users. Such an innovation could involve the pairing of comparable parameters in a two dimensional controller (as exemplified by GRM *Freeze*, Section 3.1.1).

It is hoped that Thaw's GUI, inclusive of such innovative control methods, may be completed as soon as possible. The completion of this GUI should also serve to eliminate an existing glitch which was discovered in two of Thaw's host-assigned GUIs during beta testing (detailed in Appendix III.). A prototype for the graphics of a partially completed interface for Thaw, as well as examples of the aforementioned host-assigned GUIs, may be inspected in Appendix IV.

8.3.2 Optimisation

Although the alpha and beta testing processes indicated that Thaw performed without significant flaws on a variety of modern PCs, it was noted that the plug-in was quite heavy in its CPU usage; whilst processing a stereo file, each instance of the plug-in would typically consume 20-30% of the resources of a Pentium Centrino 2.2GHz processor. As most VST-based desktop music production environments would typically use many and varied plug-ins, this scenario may deter a musician from using something as CPU-heavy as Thaw.

Improvements in Thaw's performance efficiency would therefore seem wise. A significant improvement may be achieved through the use of more efficient FFT routines; in personal correspondence from Richard Dobson, who first implemented the CARL phase vocoder in a VST environment, it was

recommended that the Fastest Fourier Transform in the West (FFTW) be implemented in place of the generic, inefficient FFT libraries which are currently used. The FFTW architecture (Frigo & Johnson, 1998) optimises itself to take advantage of the hardware architecture of the machine on which it is run, therefore offering much improved performance. It is hoped that the successful implementation of the FFTW libraries and other such optimisations can soon take place.

8.3.3 Multiple Platform Support

As previously mentioned in Section 5.3, the VST standard allows for the development of plug-ins for multiple platforms. Although the current incarnation of Thaw has been developed for the Windows-based PC, it is theoretically a trivial matter to 'port' the existing source code so that Thaw can run on other platforms, notably Apple Macintosh and Linux. This porting, which is planned for the near future, should significantly increase Thaw's potential user base.

8.4 Scope for New Creativity

It is envisaged that the Thaw software will be released as a free, modifiable, and open-source plug-in following the completion of this work. This action will be taken with the objective of making Thaw's functionality available to any interested party who may wish to build upon its achievements, whether these interests lie in the creation of drone music, or merely in the functionality of the phase vocoder in a real-time environment. It is hoped that this release will be the first and most important step in prompting future innovative work which might expand upon Thaw's achievements. This section will now propose several of the author's ideas for such expansions.

8.4.1 An Enhanced Implementation Context

While the inner workings of Thaw's phase vocoder effects are quite complex, the context in which they are implemented is relatively simplistic. In other words, there is a hands-on, linear, and straightforward relationship between the user controls and the sonic end result. While this straightforward control path allows for a high degree of user control which does not require a steep learning curve, the implementation of a more complex means of controlling the effects could lead to a large broadening of the scope of sounds which Thaw is capable of producing.

Such an implementation could involve internally automating certain Thaw effects. For example, an algorithm which would automatically switch on or off Thaw's Freeze effect at certain rhythmic intervals specified by the user might be implemented. This could take the tempo parameter from the host, thereby freezing and unfreezing the sound in a manner which would align Thaw's resultant sound with the rhythm of the relevant composition. This would be a relatively simple algorithm to implement; however, it may have the unintended side-effect of destroying Thaw's role as a drone tool and instead result in a tool for the creation of glitch-type music!

Nevertheless, such inherent automated modes of control are on the increase in VST plug-ins (as demonstrated by products such as *Sloper* and *FitchSplifter* in Chapter 3) and, with careful implementation, such control modes would doubtlessly enhance the scope and type of sounds Thaw is capable of producing as well as improving its usability.

8.4.2 Vector-Based Frame Morphing

This proposed effect would extend Thaw's current drone-producing abilities by further exploiting the nature of the phase vocoder. Its implementation would involve a significant change to Thaw's software architecture.

Instead of creating a single ‘frozen’ snapshot frame, as is the case with the current incarnation of Thaw, this effect would involve the creation of two or more such frames over time, and a subsequent transformation from one frame to the next involving vectors. In this case, vectors would refer to a number of ‘lines’ drawn from each frequency/amplitude bin in the first frame to the corresponding bin in the next frame, thereby causing linear transformations from one frozen state to the next. The rate at which these transformations occur, as well as the number of frozen states which would be involved in the transitions, would be specified by the user.

It is possible to achieve this effect in a manual and tedious manner using software such as *SPEAR* (Section 4.3.3). Furthermore, the implementation of this effect in a real-time environment could be difficult and may not allow for a predictable degree of user control. Nevertheless, this proposed effect could allow for the easy creation of a highly innovative type of drone.

REFERENCES

Anderson, J. (2000). 'A Provisional History of Spectral Music', in *Contemporary Music Review*, 19(2), p. 7-22.

Bernsee, S. (2005). *The DFT "à Pied": Mastering The Fourier Transform in One Day* [online], available: http://www.dspdimension.com/ [accessed 7th June 2006].

Bogaards, N. (2005). 'Analysis-Assisted Sound Processing with AudioSculpt', in *Proceedings of the 8th International Conference on Digital Audio Effects*, Madrid, Spain, Sept 20-22, 2005.

Cunningham, A. (2003). *Maul: A tool for sound creation and rhythm correction*, unpublished thesis (M.Sc.), University of Limerick.

Deitel, H. & Deitel, P. (2001). *C++: How to Program*, 3rd ed., USA: Prentice Hall.

Dobson, R. (1993). *The Operation of the Phase Vocoder: A non-mathematical introduction to the Fast Fourier Transform* [online], available: http://www.bath.ac.uk/~masjpf/CDP/operpvoc.htm [accessed 13th July 2006].

Dobson, R. (2001). 'A Prototype Real-time Plugin Framework for the Phase Vocoder', paper presented at *Music Without Walls? Music Without Instruments?*, De Montford University, 21-23 June 2001.

Dolson, M. (1986). 'The Phase Vocoder: A Tutorial', in *Computer Music Journal*, 10(4), p. 14-27.

Dudley, H. (1939). 'The Vocoder', in *Bell Labs Record*, 17, p. 122-126.

Fineberg, J. (2000). 'Spectral Music', in *Contemporary Music Review*, 19(2), p. 1-5.

Flanagan, J., & Golden, R. (1966). 'Phase Vocoder', in *The Bell System Technical Journal*, 45(11), p. 1493-1509.

Free Software Foundation (1991). *GNU General Public License* [online], available: http://www.gnu.org/licenses/gpl.html [accessed 24th July 2006].

Frigo, M. & Johnson, S. (1998). 'FFTW: an adaptive software architecture for the FFT', in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, Seattle, USA, 12-15 June, 1998.

Gann, K. (2001). *Minimal Music, Maximal Impact* [online], available: http://www.newmusicbox.org/page.nmbx?id=31tp00 [accessed 25th July 2006].

Grey, J. (1977). Timbre discrimination in musical patterns, in *Journal of the Acoustical Society of America*, 64, p. 467-472.

Groupe de Recherches Musicale (2006). *GRM Tools Freeze VST* [online], available: http://www.grmtools.org/quicktour/vstqtclassic/freeze.html [accessed 8th August 2006].

Howard, D. & Angus, J. (2001). *Acoustics and Psychoacoustics*, 2nd ed., UK: Focal Press.

Jonsson, M. (2005). *Ambience Reverb Documentation* [online], available: http://www.smartelectronix.com/~magnus/#Ambience [accessed 22nd March 2006].

Klingbeil, M. (2005). *Software for Spectral Analysis, Editing and Synthesis* [online], available: http://www.klingbeil.com/papers/spearfinal05.pdf [accessed 24th July 2006].

KVR Audio (2006). *KVR DSP and Plug-In Development Forum* [online], available: http://www.kvraudio.com/forum/viewforum.php?f=33 [accessed March 2006 – September 2006].

Laroche, J. & Dolson, M. (1999). 'New Phase-Vocoder Techniques for Pitch-Shifting, Harmonizing and Other Exotic Effects', in *Proceedings of the 1999 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, New Paltz, New York, Oct 17-20, 1999.

Loy, G. (2002). 'The CARL System: Premises, History and Fate', in *Computer Music Journal*, 26(4), p. 52-60.

Miranda, E. (2002). *Computer Sound Design: Synthesis Techniques and Programming*, 2nd ed., UK: Focal Press.

Morgan, R. (1991). *Twentieth-Century Music: A History of Musical Style in Modern Europe and America*, USA: Norton & Company.

Muller, R. (2003). *Implementation file for LFO* [copyright notice within source code – online], available: http://www.musicdsp.org/showone.php?id=152 [accessed 14th August 2006].

Niemitalo, Olli (2001). *Yehar's digital sound processing tutorial for the braindead* [online], available: http://www.eumus.edu.uy/docentes/jure/docs/Niemitalo_DSPForTheBraindead.html [accessed 23rd June 2006].

Pollard, H. & Jansson, E. (1982). 'A tristimulus method for the specification of musical timbre', in *Acoustica* 51, p. 162-171.

Pressnitzer, D. & McAdams, S. (1999). 'Acoustics, Psychoacoustics and Spectral Music'. *Contemporary Music Review* 19, p. 33-60.

Schnetzler, A. (2003). *FlitchSplifter Documentation* [online], available: http://andreas.smartelectronix.com/index.php?nav=9&p=6&kat=0 [accessed 24th July 2006].

Schnetzler, A. (2006). *Sloper Documentation* [online], available: http://www.smartelectronix.com/~andreas/index.php?nav=9&p=4&kat=0 [accessed 08th August 2006].

Smith, S. (1999). *The Scientist and Engineer's Guide to Digital Signal Processing*, 2nd ed., USA: California Technical Publishing.

Steinberg GMBH (2006). *Steinberg Virtual Studio Technology (VST) Plug-in Specification 2.4 Software Development Kit Documentation* [online], available: http://www.steinberg.de/324+M52087573ab0.html [accessed 23rd March 2006].

Strickland, E. (1993). *Minimalism: Origins*, USA: Indiana University Press.

Strom, R. (2005). *VST Audio Effect Plug-in Programming in C++* [online], available: http://www.stromcode.com/modules.php?name=Glowdot_Tutorials&op=list&cat=3 [accessed 15th May 2006].

Textura (2005). *A History of Dronology* [online], available: http://www.textura.org/newreviewspages/dronesarticle.htm [accessed 23rd March 2006].

Wishart, T. (1996). *On Sonic Art.* Amsterdam: Harwood Academic Publishers.

Wishart, T. (2000). *Computer Sound Transformation: A personal perspective from the UK* [online], available: http://www.trevorwishart.co.uk/transformation.html [accessed 9th August 2006].

APPENDICES

Appendix I. C++ Code Examples

Corresponding pseudo code and notes for all methods described here may be found in Chapter 6. Comments in the following code are highlighted in bold.

Thaw Initialisation

From Thaw.cpp

```
Thaw::Thaw (audioMasterCallback audioMaster)
        : AudioEffectX (audioMaster, kNumPrograms, kNumParams)
{
        //get the sample rate from the host
        lSampleRate = audioMaster(&cEffect, audioMasterGetSampleRate, 0, 0, 0,
0);
        lSampleRate = sampleRate;

        size = lSampleRate;
        buffer = new float[lSampleRate];

        //Initialise programs/parameters
        programs = new ThawProgram[kNumPrograms];
        if (programs)
                setProgram (0);

        setParameterAutomated(kCrossfade,1.0f);
        setParameterAutomated(kOnOff,0.0f);
        setParameterAutomated(kLfoRate,0.0f);
        setParameterAutomated(kLfoDepth,0.0f);
        setParameterAutomated(kModBins,0.0f);
        setParameterAutomated(kPitchshift,0.0f);
        setParameterAutomated(kAmpMod,0.0f);
        setParameterAutomated(kLfoShape, 0.0f);
        setParameterAutomated(kSpecFiltCF, 0.0f);
        setParameterAutomated(kSpecFiltGain, 0.5f);
        setParameterAutomated(kSpecFiltQ, 0.0f);

        //Initialise I/O
        setNumInputs (NUMINPUTS);
        setNumOutputs (NUMOUTPUTS);

        canProcessReplacing();

        setUniqueID ('cda1');

        resume ();

        //create myPvocs: objects based on the Apvoc class
        myPvoc1 = 0;
        myPvoc1 = new Apvoc();
        myPvoc2 = 0;
        myPvoc2 = new Apvoc();

        //initialise myPvocs
        if (myPvoc1 == 0 || (!myPvoc1->
init(lSampleRate,FFTLEN,BUFLEN,PVPP_STREAMING)))
        {
                delete myPvoc1;
                myPvoc1 = 0;
                return;
        }
        if (myPvoc2 == 0 || (!myPvoc2->
init(lSampleRate,FFTLEN,BUFLEN,PVPP_STREAMING)))
        {
                delete myPvoc2;
                myPvoc2 = 0;
                return;
        }
```

```
        //initialise freeze flags
        freezing = false;
        flagOnce = true;

        //initialise LFO
        lfoRate = 0;
        lfoDepth = 0;
        lfoModBins = 0;
        lfoFac = 0;
        myLfo = 0;
        myLfo = new LFO(lSampleRate);
        if (myLfo == 0)
                return;
        myLfo->setRate(getParameter(kLfoRate));
        setLfoShape(getParameter(kLfoShape));
}
```

Phase Vocoder Initialisation

From Apvoc.cpp

```
bool Apvoc::init(long sampleRate, long fftlen, long overlap, pvocmode mode)
{
        int i;
        if(inpv)
                return false;
        if(fftlen <= 0 || sampleRate <= 0)
                return false;

        if(overlap > fftlen/2 || overlap <= 0)
                return false;

        long windowsize = fftlen;
        decfac = overlap;
        inptr = outptr = 0;
        nbins = (fftlen + 2) / 2;

        pvoc_frametype outframetype = PVOC_AMP_FREQ;
        pvoc_frametype inframetype = PVOC_AMP_FREQ;

        try
        {
                //create dynamic storage for analysis frames and I/O buffers
                frame = new float[fftlen + 2];
                freezeFrame = new float[fftlen + 2];
                morphFrame = new float[fftlen + 2];
                inbuf = new float[decfac];
                outbuf = new float[decfac];

                //create phase vocoders for analysis and resynthesis
                inpv = new phasevocoder();
                outpv = new phasevocoder();

                if (inpv == NULL || outpv == NULL)
                        return false;
        }
        catch(...)
        {
                cleanup();
                return false;
        }

        //try initialising the phase vocoder objects
        if(!inpv->init(sampleRate,fftlen,decfac,mode,FFTTYPE))
        {
                cleanup();
                return false;
        }
        if(!outpv->init(sampleRate,fftlen,decfac,mode,FFTTYPE))
        {
                cleanup();
                return false;
        }

        //initialise the IO buffers and analysis frames
        for (i = 0; i < decfac; i++)
```

```
		inbuf[i] = outbuf[i] = 0.0f;
	for (i = 0; i < fftlen + 2; i++)
		frame[i] = freezeFrame[i] = morphFrame[i] = 0.0f;

	//initialise the freezer flags
	freezeOnce = false;
	freezing = false;

	return true;
}
```

The `processReplacing()` Method

From Thaw.cpp

```
void Thaw::processReplacing (float** inputs, float** outputs, VstInt32
sampleFrames)
{
	float* in1 = inputs[0];
	float* out1 = outputs[0];
	float* in2 = inputs[1];
	float* out2 = outputs[1];

	//sets the freeze flags to send to the myPvocs..
	if (fOnOff == 1.0f)
	{
		freezing = true;
		flagOnce = false;
	}
	else
	{
		freezing = false;
		flagOnce = true;
	}

	setLfoShape(fLfoShape);

	if(myPvoc1 && myPvoc2)
	{
		while(--sampleFrames >= 0)
		{
			myLfo->setRate(fLfoRate);
			lfoRate = myLfo->tick();
			lfoFac = lfoRate * fLfoDepth;

			t1 = *in1++;
			t2 = *in2++;
			*out1++ = ((myPvoc1->tick(t1, freezing, lfoFac, fModBins,
fPitchshift,fAmpMod,fSpecFiltCF,fSpecFiltGain,fSpecFiltQ) * fCrossfade) + (t1 *
(1 - fCrossfade)));
			*out2++ = ((myPvoc2->tick(t2, freezing, lfoFac, fModBins,
fPitchshift,fAmpMod,fSpecFiltCF,fSpecFiltGain,fSpecFiltQ) * fCrossfade) + (t2 *
(1 - fCrossfade)));
		}
	}
	else
	{
		while(--sampleFrames >= 0)
		{
			*out1++ = ((*in1++) * 0.1f);
			*out2++ = ((*in2++) * 0.1f);
		}
	}
}
```

Phase Vocoder Incrementation and Freeze Effect

From Apvoc.cpp

```
float Apvoc::tick(float insamp, bool freezing, float lfoFac, float lfoModFac,
float pitchShift, float spectDeg, float specFiltCF, float specFiltGain, float
specFiltQ)
{
	long a;
	if (!freezing)
```

```
                freezeOnce = true;

        if (!outbuf)
                return 0.0f;

        //increment the pointer to the next sample in outbuf
        float outval = outbuf[outptr++];

        if(outptr==decfac)
        {
                //generate an analysis frame of window using phase vocoder
                a = inpv->generate_frame(inbuf,frame,decfac,PVOC_AMP_FREQ);
                inptr = outptr = 0;

                if (freezing) // the freeze effect is switched on
                {
                        if (freezeOnce) //snapshot frame not filled, do so now...
                        {
                                makeFreezeFrame(frame,freezeFrame,nbins);
                                freezeOnce = false;
                        }
                        //Sound transformations to frozen frame are applied here:
                        transformTheFrame
(freezeFrame,morphFrame,lfoFac,lfoModFac,pitchShift,spectDeg,specFiltCF,
specFiltGain,specFiltQ,nbins);

                        //resynthesise freezeFrame and send to outbuf
                        a = outpv->process_frame
(morphFrame,outbuf,(pvoc_frametype)PVOC_AMP_FREQ);
                }
                else
                {
                        //transformations to non-frozen frames are applied here:
                        transformTheFrame
(frame,morphFrame,lfoFac,lfoModFac,pitchShift,spectDeg,
specFiltCF,specFiltGain,specFiltQ,nbins);

                        //Resynthesise continuous frame, and send to outbuf
                        a = outpv->process_frame
(morphFrame,outbuf,(pvoc_frametype)PVOC_AMP_FREQ);
                }
        }
        inbuf[inptr++] = insamp;
        return outval;
}
```

Creating the Frozen Frame

From Cfunctions.c

```
void makeFreezeFrame(float *streamingFrame, float *freezeFrame, long nbins)
{
        int a, a1;
        for (a = 0, a1 = 0; a < nbins; a++, a1 += 2)
        {
                freezeFrame[a1] = streamingFrame[a1];
                freezeFrame[a1+1] = streamingFrame[a1+1];
        }
}
```

The Sound Transformation Algorithm

From Cfunctions.c

```
void transformTheFrame(float *inFrame, float *outFrame, float lfoFac, float
lfoModFac, float pitchShift, float spectDeg, float specFiltCF, float
specFiltGain, float specFiltQ, long nbins)
{
	int a, a1;

	int shiftFac = (int)(pitchShift*170); //manual pitchshift factor
	int multFac = (int)((1-lfoModFac) * 100);//LFO modulation factor
	int degFac = (int)((1-spectDeg) * 100);//amp bins degradation factor

	float halfspecFiltWidth = nbins*(1-specFiltQ)*0.5f;
	float specFiltFacLo = (specFiltCF*nbins) - halfspecFiltWidth;
	float specFiltFacHi = (specFiltCF*nbins) + halfspecFiltWidth;
	float specFiltFacGain = specFiltGain * 3;

	for (a = 0, a1 = 0; a < nbins; a++, a1 += 2)
	{
		//amplitude modifications
		if (a1 < 5 * (rand()%(101 - degFac)))
			outFrame[a1] = 0;
		else if (a1 > specFiltFacLo && a1 > specFiltFacHi)
			outFrame[a1] = inFrame[a1] * specFiltFacGain;
		else
			outFrame[a1] = inFrame[a1];

		//frequency modifications:
		if (a1 < 5 * (rand()%(101 - multFac)))
			outFrame[a1+1] = inFrame[a1+1] + lfoFac + shiftFac;
		else
			outFrame[a1+1] = inFrame[a1+1] + shiftFac;
	}
}
```

Random LFO Wave Shape

From LFO.cpp

(**Excerpted from**) `void LFO::setWaveform(waveform_t index)`

```
{	...
	int i;
	float place = 0.0f;
	for(i=0;i<256;i++)
	{
	    if (rand()%5 == 0)
			place += ((rand()%2)-0.5)* 0.2f;
	    table[i] = place;
	}
	table[256] = 0.0f;
	...
}
```

Logarithmic Scaling of LFO Rate and Spectral Filter CF

From Thaw.cpp

(**Excerpted from**) `void Thaw::setParameter (VstInt32 index, float value)`

```
{
	//ensures 1-logval (for scaling of fLfoRate and fspecFiltCF) is nonzero
	double logval = value;
	if (logval == 1)
		logval -= 0.0001;

	switch (index)
	{
		case kLfoRate:		fLfoRate = (-log10(1-logval))*5;
			break;
		...
		case kSpecFiltCF:	fSpecFiltCF = (-log10(1-logval))*0.25;
			break;
	}
}
```

Appendix II. Excerpts from the Development Log

05-06-2006
- Working version of aDelay sample program.

04-07-2006
- Developed Apvoc class, similar in structure and context to equivalent classes in the CDP plug-ins. The plan is this: Apvoc class contains methods including init to declare and initialise in and out phasevocoder() objects. An object called myPvocs is declared and initialised in the AudioEffectX method of the Thaw object, i.e. it is created as soon as the plug-in is started.
- The main work, of course, will be done inside the processReplacing method. Am currently working on Apvoc::tick() method which will encapsulate phasevocoder::generate_frame and phasevocoder::process_frame methods. Not sure how it will work yet but this is how Dobson's spectral plug-ins work.

10-07-2006
- Bad stuff has been happening…Still facing the tick() problem, and I've done something that makes MiniHost crash every time I try to run it. New approach: I am going to try to use Dobson's adapted CARL software routines instead of the generic ones I downloaded which do not appear to be suitable for streaming pvoc uses.

11-07-2006
- Finally, a working pvoc! Sorted out the Access Violation Exception mentioned previously by sorting out some dodgy pointers. Now have a working analysis/resynthesis algorithm although I haven't figured out how to do anything interesting with it yet. Have defined 2 parameters for the VST, one to control the volume, the other will now be used for experimentation purposes...

22-07-2006
- The freezer works! Took the approach of storing a single analysis frame in a buffer and spitting it out while the freeze effect is on.
- Got some correspondence from Richard Dobson in reply to my email. Looks like there will be no licensing problems with the CARL pvoc, thankfully. He recommended I use the FFTW libraries for speed, had a look at them, they are quite daunting.

01-08-2006
- An LFO with controllable rate and depth has been added to the software. At the moment it is modulating a specified proportion of the frequency bins in the freeze frame. The effect is quite cool but the code to select the proportion of bins needs work

09-08-2006
- Hope to sort out some issues with the Modulation and Amp deg factors - they are not very linear at the moment.

15-08-2006
- Been trying to implement the GUI but with no luck.
- Realised it might be a good idea to allow the user to apply the pvoc sound transformations even when the effect is not 'On' - while this may not make the plug-in as aesthetically static as desired, it does offer a lot more flexibility with regard to functionality.

24-08-2006
- Code tidied up a great deal. #DEFINE headers put in to allow programmers to quickly change pvoc parameters and recompile as desired.
- Awaiting more beta tester responses.
- Implemented some more tiny functionality changes in response to tester feedback - LFO max rate is now 24Hz. LFO depth in random algorithm has been increased.

Appendix III. Sample Beta Testing Reports

'Beta testing' refers to one of the latter stages of software testing whereby a product in development is released to a small group of users outside of the core development environment in an effort to uncover bugs. Although formal beta testing standards and procedures exist, the resources and expertise for conducting such processes were limited at the time of testing. Nevertheless, the importance of rigorous and structured testing for the purposes of improving the final product on many levels was acknowledged.

Therefore, a small group of voluntary beta testers received a copy of the Thaw plug-in on 11th August 2006 and were asked to comment on the usability, functionality, aesthetics and reliability of the software, following a week-long period of testing using various hosts and test machines. In particular, the volunteers were asked to perform testing in somewhat unusual scenarios, including the use of different sample rates, mono and stereo sound sources, the use of multiple instances of the plug-in, automation of the plug-in, and any other action which would stretch the functionality of the software to its limits. The beta testers' responses were noted in a log. The table on the following page summarises some of this feedback. A list of corrections and enhancements that were implemented following the receipt of this feedback is then presented.

Tester #	Machine Specifications	Host Software	Technical Issues	Sample Comments on Usability & Aesthetics
Tester 1	Windows 2000, Pentium IV, 512 Mb RAM	Ableton Live v.4	[No issues]	"left it running for a few hours, no problems there… on occasion the pitch can rise by a semitone when frozen, this might be some kind of weird alias?" **[1]**
	Windows XP	Ableton Live v.5		
Tester 2	Windows XP, Pentium 1.8GHz, 1 Gb RAM, M-Audio sound card	Tracktion, Max/MSP	- Tracktion: graphical glitch from host-assigned GUI (functionality unaffected) **[2]** - Audible glitches when rapidly changing wet/dry mix **[3]**	"why stop LFO Rate at 8Hz? Having more than 20Hz would create cool ring mod effect… no crashes" **[4]**
Tester 3	AMD 2GHz, 1 Gb RAM, Windows XP, Delta 44 sound card	Audiomulch	[No issues]	"well impressed with it… I couldn't get it to break… creating some crazy drones!"
Tester 4	Intel Centrino 2.1GHz, 1 Gb RAM, Windows XP (SP2), Indigo sound card	FL Studio 6, Bidule, energyXT, Usine	- FL Studio: graphical glitches from host-assigned GUI (functionality unaffected) **[5]** - Excessive LFO speed at 96kHz sample rate**[6]**	"a wet/dry mix would be useful, for using this in Bidule" **[7]**

Notes:

[1] This perceived change in pitch is in fact the result of 'freezing' some types of polyphonic sounds using the phase vocoder, as discussed in Section 8.1. It may be avoided by using a longer phase vocoder window length than the default 1024, at the expense of more CPU power.

[2] This graphical error caused the parameter displays in Thaw's host-assigned GUI to display incorrect values for a particular parameter, without actually affecting the sounds produced by the plug-in. It is hoped that this problem will be rectified in a future version of Thaw, with or without the development of a custom GUI for the software.

[3] Thaw's cross-fading function does not interpolate between subsequent extreme values, therefore sudden extreme changes of the Dry/Wet mix parameter can result in audible glitches. It is foreseen that this problem may be removed by introducing such interpolation.

[4] In response to this tester's request, the maximum LFO rate was raised to 24Hz, thereby allowing rapid vibratos and slight ring modulation. A logarithmic scaling function, which would allow the user to retain finer control at low frequencies, was also subsequently implemented (Section 6.4.5).

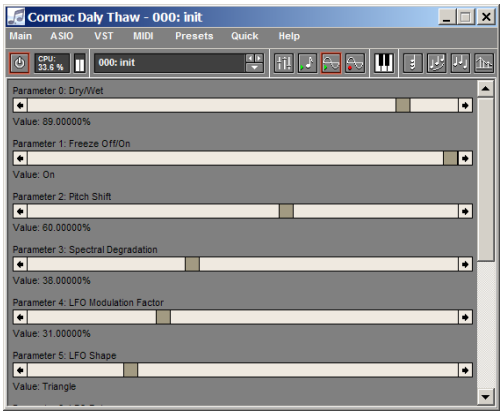
[5] This appears to be the same glitch reported in Note [2].

[6] This tester uncovered this elusive bug by using the software with a higher sample rate (96 kHz) than the default 44.1 kHz. It was subsequently discovered that, in the LFO, the sample rate was 'hard coded' at 44.1 kHz rather than requesting the sample rate from the host, as it should have been. This had the effect of making the LFO rate go twice as fast as intended. This error was duly rectified without difficulty.

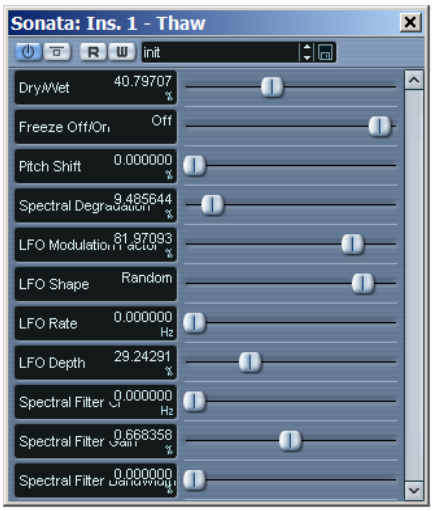
[7] The Dry/Wet mix was introduced at an early stage of development, following this tester's request.

Appendix IV. Software Screenshots

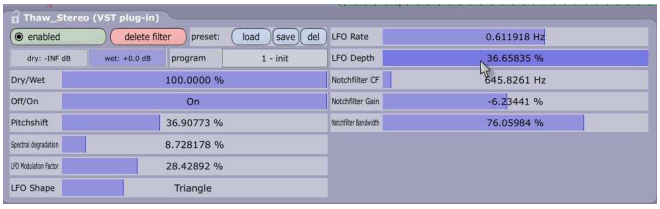
Sample of Host-Assigned User Interfaces for Thaw



MiniHost



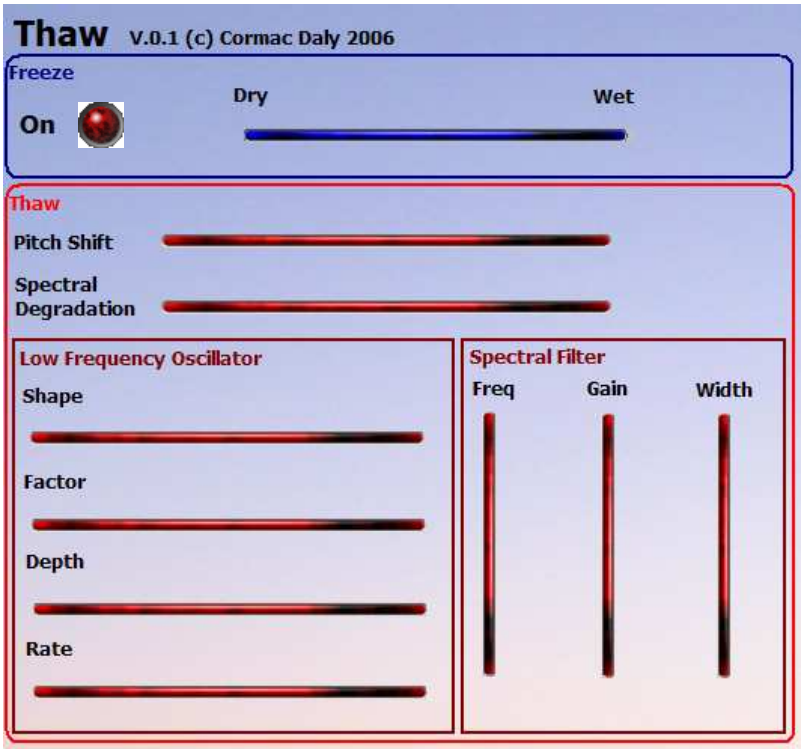
Cubase



Tracktion

energyXT

Prototype Custom User Interface



A prototype design for a custom user interface for Thaw. The plug-in's parameters have been grouped into logical compartments in order to improve usability. Slider 'handles' and parameter displays are not included in this image.