New Electronic Performance Instruments

For Electroacoustic Music

Dylan Menzies

Department of Electronics,

University of York

A dissertation submitted for

the degree of Doctor of Philosophy.

February 1999

Acknowledgements

Thanks to my supervisor, David Howard, for enduring me for three years and making sure this thesis was finished on time and understandable. I am also grateful for the patient assistance provided by the technical support staff, particularly Ross Oliver. To room T223 and its past inhabitants I say farewell, and fondly remember time spent talking over countless cups of tea.

Thanks to the Music Department for letting me use their facilities, and drill the occasional hole in a wall. I feel privileged to have made some good friends among the music postgraduates. May nerve8 continue to prosper!

Finally, thanks to all my family.

Abstract

Musical performance is an ancient art form covering many different musical styles and instruments. Since the turn of the century the rise of electricity and then electronics has opened up opportunities to create and control sound in fundamentally new ways. This process continues today through the rapid development of computers.

The use of modern technology in musical performance is first examined, focusing on the electroacoustic style, and with reference to traditional forms of preformance. From this guidelines are constructed for designing new performance instruments. The characteristics of traditional instruments play a central role in these guidelines.

Four projects are then used to demonstrate how the principles can be incorporated with contemporary synthesis and spatialisation techniques. Considerable effort is devoted to the practical as well as theoretical aspects. Some CD recordings are included to illustrate the work.

A broad range of practical techniques are developed, and deeper insights into the performance problem are gained. The success of the designs supports the original design principles, in particular those based on traditional instruments. There remain practical obstacles to more widespread application of these ideas, but with time we may see the emergence of a new class of high quality electronic instrument.

Contents

List of Figures

Chapter 1

Introduction

Critical Analysis of Instrument Design

Music began as a performance art. The only way to hear music was to listen to a live performance. This is no longer true. Music can be played back from a recording machine, broadcast onto radio sets or generated by a computer. Technology has also directly affected the ways in which people perform. **Chapter 2** traces this process and identifies areas of electronic instrument design where more development would be profitable. Acoustic instruments are repeatedly found to be valuable in assessing electronic instruments, to the extent that **Chapter 3** is devoted to a detailed survey of acoustic instruments.

The Central Hypothesis

The scope of all the design guidelines created is wide, but those concerned only with acoustic instruments are so pervasive that they form a central theme, which may be stated as a hypothesis. *Electronic instrument design can benefit greatly from careful abstraction of high level design principles inherent in traditional acoustic instruments.* This may at first seem confusing as many electronic instruments superficially resemble acoustic instruments. The full evaluation of an instrument does, however, require the examination of many abstract qualities besides physical appearance. A detailed elaboration of the design principles can be found in the summary of **Chapter 3**. The following is an indication of the contents:

- *Ergonomics and sensing* - The physical interface should exploit the strengths of human motor capabilities, and the sensing technology must be carefully matched to provide adequate response and resolution where it is required.
- *Control-synthesis dynamics* - Subtle complexities in the relationship between control and sound output add much interest to an instrument without making it too complex to play.

- *Global integration* - The components of the man-instrument system are heavily interdependent, hence an integrated design approach is paramount.

Electroacoustic Music

The observations on modern performance technology apply across a range of styles. However, for the purposes of this thesis the focal style is *electroacoustic* music, which encompasses the tradition of processed and synthesized music dating back to the early experiments in Europe in the late 1940's, which were partly funded by radio stations. While the setting was not in university, many of those involved had previously received a high level of academic training in music and other disciplines, which is reflected in the depth of analytical thinking in the work. Electroacoustic music is now firmly in the public domain. Anyone with desire and talent can create it with a minimal financial outlay. It retains a wide following, but not on such a large scale that it is subject to commercial pressures.

Throughout its existence electroacoustic music has been associated with live performance. Recent technological innovations have further raised interest in live performance, whereas in other musical styles live performance increasingly takes second place to studio production. Electroacoustics provides a broad testbed for expanding the definition of performance.

Performance Technology Review

In **Chapter 4** audio signal processing techniques and tools are surveyed, with particular attention to live performance applications. Included are introductions to the waveguide and Ambisonic technologies, which are employed later.

Project Work

Four projects were undertaken to develop new performance instruments for electroacoustic music, each of which incorporates some of the guidelines established earlier.

The MIDI keyboard is ubiquitous and therefore defines a global performance interface standard. **Chapter 5** looks at how the interface can be applied in a generalised way to electronic performance, so that the concept of *note* is no longer dominant. Some examples are given in Csound. The novelty lies in the control processing concepts and to a lesser degree the techniques used to implement the concepts in Csound. Dispensing with the *note* allows some interesting acoustically inspired behaviour to be introduced.

Ambisonics is a methodology for encoding the soundfield about a point and decoding it onto a speaker array so that listeners perceive the original soundfield. It has been applied to the live diffusion of sound, but using tools which were originally designed for studio production work rather than performance. In **Chapter 6** a new purpose built performance system is described. Novel signal processing is combined with an unusual method of control, using a MIDI keyboard as the performance interface. The traditional point synthesis is extended with an efficient for technique for encoding object width. New algorithms are described for efficient rendering of spatialised reflections, and for the spatial spreading of frequencies from a mono source.

The concept of an instrument has broadened within live electronics, to mean a system which processes human performance to produce sound, and whose behaviour may change over time. **Chapter 7** presents *SidRat*, a piece for live electronics which embraces this most general definition. It can be viewed as a piece *and* an instrument. The main achievment is to show how Csound can be used effectively for implementing instruments of this type, by assembling a number of non-standard Csound techniques. In doing so one of the principal design guidelines is adhered to, that of global integration.

In the first three projects no new interface hardware has been built, because a conscious decision was made to make the best use of common control devices. There is clearly much potential, however, for designing interfaces with new properties. This approach requires an extended level of integration discipline. **Chapter 8** describes the design of the *CyberWhistle*, an electronic instrument based around the Penny Whistle. The main feature of the interface is the continuous tracking of finger movement above the fingerholes, allowing for more expressive control than switches. Synthesis is developed in C++ using waveguides, extending on standard techniques to include efficient models for the bore. This leads to a much more open consideration of control-synthesis dynamics inspired by acoustic instruments, which may yet play an important part in the future of instrument design.

Conclusions

The projects affirm the original arguments over design guidelines, and in their course provide a wealth of practical methods which may be of future use. **Chapter 9** provides a summary of all the work and outlines future developments.

Chapter 2

Technological Developments in Musical Performance

Here we examine the developments of music technology for performance associated with the modern industrial age, those employing powered mechanisms, electricity, electronics and finally computers. The aim is not to provide an exhaustive history, but to instead trace the notable successes and failures, and supply some analysis for them. The main reference is [Man85]. Additional relevant material can be found in [DJ85] and [RSA$^+$95].

2.1 Aspects of Performance Pre 1900

2.1.1 Classical Music

In the West a strong tradition has evolved in which music is written to score by a composer and then performed by an ensemble or orchestra in front of an audience. It is worth examining this process in more detail before considering the changes that have occurred to it. The composer writes to a score using expert knowledge of the sounds that can be produced by different instruments, and the kinds of articulations and phrasing that players can produce. [1]

In smaller groups the details of performances by each individual become more noticeable. The term *musical interpretation* describes the information which each player adds to the basic content of the score. The extra information is implicitly required but cannot be practically codified in the score. This is just as well because it allows for the freedom of individual expression which is a strong motivation for players. The concept of *virtuoso* in western music is an indicator of the importance of individual expression. Even in a large orchestra, where the individual players may not be heard

[1] In some instances the composer may envisage novel effects resulting from unusual articulations or combinations of instruments. It is then more important for the composer to guide the rehearsal process so that the imagined effects are better realised.

clearly, many elements in the overall sound give away the presence of human performers.

The listeners are passive, except that they may react positively to the performers, and encourage them to play well. It is common for musicians recording in a studio today to complain about the lack of a ‘live atmosphere’.

2.1.2 Folk Music and Improvisation

The other broad thread of musical practice derives from folk music, of its many kinds through out the world. Here there is less emphasis on composition. The musicians *are* the composers. They take old music, reinterpret it and write new music. More importantly, there is the element of improvisation, that is the ability of the player to compose, to some degree, while playing. The scope for composition is more limited with improvisation, but it does allow for the creation of music with spontaneity, surprise and unity of dynamic and melodic phrasing. This can be viewed as an extension of the freedom of expression in the performance of classical scores. Many well known western composers have been keen improvisers although without recordings, improvisation is not part o their musical legacy. Mozart was known for improvising entire cadenzas when performing his piano concertos. As we shall see later, improvisation has become a functional part of contemporary music.

2.1.3 Changing Views Towards Performance

Wagner’s Orchestra

The 19th century saw some experimentation with the notion of music performance. Wagner introduced the covered orchestra pit and so gave orchestra a more abstract role. [Fre89].

The Player Piano

The application of mechanical technology in clockwork music devices led to a new complexity in music technology. The popularity of the *Player Piano* at the turn of the century demonstrated a willingness to embrace new musical devices [OH84]. The player piano developed to a stage where it could faithfully reproduce performances recorded onto paper rolls. More surprising are the controls that were developed so that someone sitting at the piano could ‘interpret’ the rolls. The foot operated bellows were not used just to power the system, but also to add dynamics. The tempo lever controlled the rate of roll advancement and other levers provided for mechanical frequency equalization. The player piano is thus reminiscent of modern digital keyboards with one finger chord features and built in rhythms. The idea is to allow people with little keyboard experience to become musically involved. Judging by the high regard of the player piano amongst composers and performers of the day, it seems that it was more successful than its complex digital relatives of today. Stravinsky for one, was impressed with the fidelity, and prepared nearly 50 rolls of his orchestral works in order to ‘express his musical intentions more clearly to future conductors’.

2.2 New Technology

2.2.1 Electricity

By the end of the 19th century, the acoustic science of wood, glue, reed and brass had matured to the point where no significant developments were being made. The harnessing of electricity at this time offered a wealth of new musical possibilities. A patent was filed in 1897 by the American, Thaddeus Cahill for an electrically based sound generation system subsequently known as the Dynamophone or Telharmonium. The first fully developed model was presented to the public in 1906, at a cost of $200 000. The Telharmonium was keyboard operated and used dynamos as oscillators. The composer Ferruccio Busoni was interested and wrote an essay *Sketch of a New Aesthetic of Music* discussing the potential of the Telharmonium for novel sound and harmony. Busoni clearly foresaw the potential of electricity to produce and control new sounds. Cahill had plans to send live music across the telephone system, where it could be received in lounges and bars[2]. Unfortunately the signals were found to interfere with other telephone signals and the venture fell, just before the outbreak of World War I.

2.2.2 Early Electronic Instruments

The *thermionic valve* patented by Lee De Forest in 1906, marked the start of the electronic age. Progress was initially slow, but between the Wars a number of electronic instruments were produced, mainly as additions to the concert orchestra. The two most remembered are the *Thérémin* (1924) and the *Ondes Martenot* (1928). The first of these has an unusual method of control. The pitch and amplitude are controlled separately by the proximity of each hand to separate metal loops (It is said to be the result of a building a faulty radio). In skilled hands it is capable of soprano like vibrato and tonal variation. It has been used in popular music by the *Beach Boys*, and today enjoys a renaissance as a commercial instrument. The Ondes Martenot has a distinctive, resonant sound and is played by keyboard, and a ribbon controller allowing continuous glissandi. Messiaen used it to good effect in the *Turangalila symphony*. Several other composers employed such devices : Hindemith, Honegger, Koechlin, Milhaud, but it was only a brief trend. Film music established a longer lasting relationship with the instruments. Unusual sonic material has since been common in film scores.[3]

[2]This is before recording had really caught on.

[3]One of the first powerful digital audio processors was built for Lucasfilm, [Mor 82].

2.2.3 Radio Broadcasting

Marconi made his first successful wireless broadcast over 2 km in 1895. Wireless telegraphy developed, rapidly followed by audio broadcasting. For the first time people could listen to live performances of music from their living room. This marked a major change in perception about live performance, that it could be abstracted away from the human performers. Through broadcasting music gained a wider audience but possibly at the cost of weakening, in some sense, the musical listening experience.

2.2.4 Recording

The phonograph, patented by Edison and Berliner in 1877, was the first method of sound reproduction, beginning a long line of technical improvements which continue to this day.[4] Recording brought the second major change in the way we listen to music. Not only is the performance reproduced at any place, from a box, it is also reproduced at any time. Records enabled radio stations to play much more music, which encouraged more radio listening and buying of records, fueling more music recording. A vast new industry was created. Recording further encouraged the move away from concert listening that began with radio.

Before recording developed, every piece of music heard was different. A recording of classical music today may contain numerous edits in a mix taken from several microphones. Repeated listening means that imperfections can not be tolerated. The perception of a natural performance, in this case, is an illusion. Jon Frederickson summarises the impact of recording as follows [Fre89]

- The performance need not be live.
- The performer relates to the microphone not to an audience.
- The audience relates to a machine not a performer.
- Musical time is replaced by clock time.
- The 'aura' of the musician, his sound, becomes transformed into electronic information: The artists unique voice becomes a universal possession.
- The act of making music together becomes transformed into an electronic function of a computer track which coordinates all other tracks.
- Technology simulates live music so well that the simulated reality often becomes a new definition of reality music.

The relationship between popular music and recording technology is discussed later. Attention is now directed towards one of the first movements to use recording technology creatively.

[4]Apparently Edison was not interested in recording music.

2.2.5 Musique Concrète

A group of musicians in Paris led by Pierre Schaeffer developed methods of composing music using recordings of natural sounds. A new kind of composer emerged, capable of innovating with technology to a degree previously unknown (Schaeffer was trained as a physicist). In the winter of 1949 Schaeffer gave a live concert of musique concrète which involved record players, mixing units and loudspeakers. Despite lack of rehearsal the event was well received. It wasn't until 1951 that Schaeffer could replace the inconvenient process of record cutting with tape recording.

The concept of a *concert for tape* became increasingly common as tape recorders became more widespread. Another format that established itself early on is the *concert for tape and acoustic instrument(s)*, in which the player must follow the lead set by the tape part. This creates new problems of synchronisation. In some cases exact timing is not required and the performers can gain sufficient cues from the tape. Otherwise a *click-track* may be used. Either the performers or a conductor hear the clicks through a small earpiece. A conductor is often preferred as the clicks may be distracting to the performers.

2.2.6 Elektronische Musik

The musique concrète movement was driven by the study of natural sound and its psychoacoustics. By contrast an informal network of composers grew in Cologne in the early 50s who were motivated by pure electronic sounds. This began when Dr Werner Meyer-Eppler at the Department of Phonetics at Bonn University was visited by Homer Dudley of Bell Labs, USA. Dudley demonstrated the first vocoder, which was capable of synthesizing crude speech by filtering a noise source. Those associated with *Elektonische Musik* went on to develope a wealth of techniques for generating and processing electronic sound, both for tape and live performance.

In 1955 Herbert Eimert, one the pioneers of electroacoustic music at Cologne wrote:

> "Here we touch on the most widespread misconception: namely that one can make music 'traditionally' with electronic means. Of course one 'can'; but electronic concert instruments will always remain a substitute. The fact that practically no music which can be taken seriously, artistically , has been written for electronic concert instruments is due precisely to the fact that its use either as soloist or ensemble instrument does not transcend the old means of performance. New ways of generating sound stipulate new compositional ideas and these may only be derived from sound itself which in turn must be derived from the general 'material'."

Of this Jon Appleton observes [App86]

> "Eimert was attempting to justify the absence of the performer because thirty years ago instruments did not exist which could offer in real-time the universe of sound discovered

in the tape studio."

Stockhausen, who later became head of the Cologne studio, produced the classic electronic piece, Kontakte, in 1958-60. Scored for tape with live piano and percussion, Kontakte creates subtle points of contact between the real and synthetic parts, without resorting to imitative sounds. Stockhausen created new standards for live performance-with-electronics that are still relevant today, [Hei72]. From 1964 he produced numerous pieces in the genre. In the piece *Solo* (1966) the sound from a solo performer is passed through an arrangement of delays onto four speakers. Four assistants are required to manipulate the tapping points, switches and volume levels according to a precise sequence. The result is added sonic structure which correlates in a predetermined way with the scored solo performance. The volume control levels are not specified exactly, which allows an element of performance interaction between one of the assistants and the solo performer, although the assistant is not really 'performing in a musically intimate way'. The arrangement is typical of many which require an engineer to assemble a number of separate items to create a performance. This adds an element of theatre to the work, but possibly the inconvenience of restaging work reduces the chances for future performance. The careful use of multiple loudspeakers created new possibilities for the control of the sound in 3-dimensional space.

Up until *Ylem* (1972), Stockhausen relied mainly on the signal processing of natural, performed sound. Thereafter, he began incorporating more live synthesis. Manning, [Man85], considers that Stockhausen has succeeded most in his older style, and has been reluctant to follow developments. This may be the price for establishing a solid aesthetic base in a technological environment which is rapidly changing. It may also reflect Stockhausen's dissatisfaction with digital systems.

2.2.7 The Growth Of Live Electronics

Italy and Musical Theatre

The 60's saw a large growth in the use of live electronics in performance. In Italy two ensembles were established which took a strong interest in improvisation with electronics. *Gruppo di Improvisazione Nuova Consonanza* began by producing music for acoustic instruments which had electronic qualities, and later introduced live electronics and formal tape parts. *The Musica Elettronica Viva* explored freer styles of improvisation using a mixture of live electronic and acoustic sound. They promoted the use of *musical theatre* in electronic music. The flexibility of electricity lends itself to unusual methods of performance interaction such as controlling aspects of someone else's performance. Theatre can be used to exploit this fully. By contrast Stockhausen's performances have been very formalised.

Britain and The Jazz Avant-Garde

In Britain the group AMM founded in 1965 by Lou Gare, Keith Rowe and Eddie Prevost combined jazz and contemporary music, with a strong element of live electronics. Two similar ensembles *Gentle Fire* and *Naked Software* employed a range of unusual instruments constructed from inexpensive materials with electrical pickups.

Roger Smalley and Tim Souster formed the group *Intermodulation* in 1969 which initially produced work similar in style to Stockhausen's. In fact *Transformation I* predated Stockhausen's similar *Mantra* by a year. Souster went on to found *0 dB* which incorporated yet more styles of music outside the classical western mainstream.

John Eaton was successful in combining early synthesizers with acoustic performers in such works as *Mass* (1970) for soprano, speaker, clarinet, three *synkets* and a Synmill-Moog synthesizer.

Japan and Environmental Music

Toshi Ichiyanagi was a pioneer of *Environmental Music* and *Installation Music*. In 1964 he constructed a system for an art gallery in Takamatsu where an exhibition of sculpture was accompanied by a soundscape of photo-cell controlled oscillators, operated by the movement of passers-by. This is live performance of a kind, but one in which the performers may be only partially aware of how their actions affect the music.

America

In America, one name stands out in the early development of electronic and other experimental musical forms, that of John Cage. 'Music for Amplified Toy Pianos' and 'Cartridge Music' are live pieces employing the amplification of very quiet sounds. In 'Variations V' (1965), Cage includes dancers who control audio-visual elements by breaking light beams focused on photoelectric cells.

Fluorescent Sound (1964) by David Tudor utilises the resonances of fluorescent light tubes, amplified and distributed via loudspeakers. In *Rainforest* (1968) Tudor has replaced loud speaker cones with various kinds of resonator. Banks of oscillators, performed live, constitute the speaker feeds.

Wave Train (1966) by David Behrman relies on the feedback between a piano fitted with guitar pickups and a monitor speaker to extend the decay of the strings, which are performed.

Gordan Mumma developed a *cybersonic controller*, a kind of effects processor that the performer could wear around the neck, and control, whilst performing an acoustic instrument. Cybersonics was used in *Medium Size Mograph* (1963) for piano and four hands, and *Mesa* (1966) for accordion, in which simple melodic phrases are tranformed into complex successions of sounds rich in non-harmonic partials.

Alvin Lucier was particularly interested in the theatrical possibilities of live electronics. *Music for Solo Performer* (1965) included live, amplified alpha brain waves. *North American Time Capsule* (1967) employed eight prototype digital vocoders to process the sounds of a chorus and various theatrical objects. In *Vespers* (1968) performers carry location detectors which react sonically to objects placed around a darkened stage.

The *Minimalist* movement associated with Steve Reich, Terry Riley and later Philip Glass, has had a much simpler relationship with live electronics. Keyboard synthesizers have been used to produce subtle contrasts of timbre to acoustic instruments, possibly with slowly varying elements within a cycle of notes. The keyboard bass in *Floe* by Philip Glass is an example.

2.2.8 Computer Music

When computers were first used for musical purposes they were too slow for live or real-time work, and were only available in universities and corporations. Technology has progressed rapidly over 30 years to a stage where an affordable personal computer is capable of complex live synthesis. The basic principles have not changed. All digital computers are *finite Turing machines*. They accept information, process it according to a program and output the result. The behaviour of the machine is defined by its program, which has no physical substance. This immediately establishes a very different design aesthetic to acoustic instruments whose musical character is entirely dependent on the physical construction from different materials. With the live electronics discussed previously, instruments are constrained to the particular properties of the electronics available. The only theoretical constraint on computers is speed of execution, which determines the complexity of processes that can be sustained in real-time. In practice other important constraints arise from compromises in the construction of the audio I/O system. These include *I/O latencies*, the delays incurred passing audio data from the input to the processor and passing it to the output. The delays are usually maintained deliberately as buffers against breaks in audio activity, to reduce the chance of the audio output being interrupted. The faster the processor, or the simpler the real-time audio process, the smaller can the buffers be made while not increasing the chance of audio output breakage.

Non real-time environments

Max Mathews is remembered as the father of computer music. Working in the Bell Telephone Laboratories, USA, Mathews wrote programs for synthesizing digital sound samples. MUSIC I appeared in 1957, leading to many future developments in the MUSIC series and programs by others. In 1963 Mathews wrote in the Science article *The Digital Computer as a Musical Instrument* that 'Sound from numbers' offered unlimited possibilities for sound creation: 'any perceivable sound can be so produced' [Mat63]. However, in *The Technology Of Computer Music* [Mat69], he wrote:

> "The two fundamental problems in sound synthesis are (1) the vast amount of data needed to specify a pressure function-hence the necessity of a very fast program-and (2)

the need for a simple, powerful language in which to describe a complex sequence of sounds”

Csound

Csound developed by Barry Vercoe of MIT is the most widespread MUSIC related program available today [Ver92]. The basic mode of operation is to supply the program with a text *score* and *orchestra*. The orchestra contains the definition of different *instruments* used by the score. Variables, operators and audio processing primitives allow complex synthesis functions to be created rapidly. The score triggers instruments and sends parameters which can be used within instruments to modify their performance. This structure enables the composer to write in flexible manner whilst retaining something of the style of traditional written scores. The orchestra, of course, has none of the human musical intelligence of a real orchestra, ensemble or solo performer. As a result much more information needs to be specified explicitly in the score to achieve a satisfactory computer performance. The digital orchestra makes up for this ‘stupidity’ by giving the composer precise control over every aspect of the performance; precision timing, dynamic phrasing and timbral control. The last of these, timbre control, is the most challenging. The composer can synthesize sound or process natural sound. The variety of sound produced in a human performance is largely due to subtle variations in articulation. It is impractical for the composer to explicitly enter into the score such detailed articulations. Instead, attention is usually directed towards varying the *underlying* sonic material, by finding new synthesis or processing techniques.

Csound is fast to use, but a side effect of this is that it imposes a large amount of structure. Other computer music environments such as Cmusic by Richard Moore allow the composer to use libraries of C functions to build up an orchestra, and a score processor for accessing the orchestra. The composer has the flexibility of working directly with C and the ability to define new base level functions. However, composers with little programming experience will find working with a Csound script much easier initially.

Real-time environments

The first use of computers for real-time music was to control analog oscillators, since useful sampling rates for the control signals are much lower than audio rates. The first ‘hybrid’ system, known as GROOVE, was developed by Max Mathews and was unveiled in 1970. The control interface consisted of joysticks and knobs. A graphical editing facility was available for non real-time input and editing. Mathews characterised GROOVE as follows (Mathew and Moore 1969).

> “The desired relationship between the performer and the computer is not that between a player and his instrument, but rather that between the conductor and his orchestra.”

A conductor is, at least in part, a live performer but GROOVE was used for interactive composition, not live performance. The principle of interactive composition is that the composer can quickly experiment with different ways of generating a sound object, and also impart some ‘live performance character’ into the sound. In the last section it was noted that human articulations are impractical to synthesize directly into a script.

Jean Claude Risset gives the following caution regarding GROOVE [Ris94]:

> “The human operator can only specify a limited amount of information in real-time, so there must be a careful balance between the informations specified in advance and those controlled by gestures in real-time.”

This is in contrast to traditional instruments, which can be used to sustain rich solo performances. In this we see an important difference between interactive composition and live performance. For interactive composition the same intense level of real-time music expression is not required, since the music can be built by accumulating layers of less intense real-time performance. The result is muscially satisfying in a different way. The accumulated relationships between the layers cannot be achieved by a single solo performance. Even if several musicians are playing together live, they don’t have the luxary of editing their performance layers. The best they can do is practice together several times before hand.

Digital Synthesizers

By the early 1970’s a number of all digital real-time synthesis systems had been developed. Cameron Jones produced the prototype for a self-contained digital synthesizer in association with the New England Digital Corporation, which was subsequently marketed successfully as the *Synclavier*. The structure consisted of a bank of *timbre generators* each of which contained of 24 independently controlled sinusoidal oscillators. As well as additive synthesis a frequency modulation function allowed the combination of 4 voices to generate complex tones. From the viewpoint of control, the Synclavier was the first all-digital instrument targeted at live performance rather than interactive composition. The five octave piano-style keyboard could be split for different voices and a touch ribbon permitted continuous pitch variation in the manner of the Ondes Martenot. Access to the system software via keyboard, and a VDU were later added enabling interactive composition.

The composer Joel Chadabe and Roger Meyers developed a special performance program *PLAY*. This permits continuous variations to be superimposed on a pre-programmed sequence of events by the performer. Chadabe used capacitive field detectors to allow control by hand movements.

Commercial Digital Synthesizers

Other commercial synthesizers soon appeared, such as the Fairlight CMI, but the commercial market is inevitably oriented mainly towards popular music. As a result such systems have been of limited value to the composer interested in interactive composition, and live performance. The MIDI (Musical Instrument Digital Interface) standard arose from the practical need to control several synthesizers with one keyboard. It has led to communication with computers and is responsible for many techniques used in studios for pop music, television and film. In his Keynote Paper at the ICMC-91, Julius Smith has this to say:

> "The ease of using MIDI synthesizers has sapped momentum from synthesis-algorithm research by composers. Many composers who once tried out their own ideas by writing their own unit generators and instruments are now settling for synthesis of a MIDI command stream instead. In times such as these, John Chowning would not likely have discovered FM synthesis: a novel timbral effect obtained when an oscillator's vibrato is increased to audio frequencies .... MIDI was designed to mechanise performance on a keyboard-controlled synthesizer."

Here Smith is primarily referring to composers generating tape music, as few composers have ever had access to powerful digital real time synthesis systems. Most composers producing live electronic music today still rely on MIDI synthesizers. Up until very recently they had no other option. The increased power of the affordable workstation is changing this.

A more careful consideration of individual commercial synthesizers is given under the section on popular music, where some of the promising new developments in MIDI synthesizers are discussed.

One example of a commercial system that was aimed at the serious composer is the DMX-1000. This was produced in the USA by Digital Music Systems. A later version influenced Barry Truax's POD program. The DMX provided a real-time interactive environment for processing and synthesizing. This complemented the MUSIC programs of the time which were dedicated to producing complex sounds in non real-time.

2.3 Recent Developments In Live Computer Performance

Today the situation for live performance with electronics is centred on computer workstations and MIDI equipment. Non-computerised electronics has been relegated to sensory functions. The ease of use and flexibility and low cost of digital electronics has encouraged composers to create their own systems for working with, resulting in large numbers of such systems. The situation is further complicated by the large number of more technologically minded people attracted, as is abundantly clear from International Computer Music Conference proceedings. Simon Emmerson 'identifies two types of computer composition to have emerged from the studio into the performance space .. one more interested in event processing, the other in signal processing' [Emm91].

2.3.1 Event Processing

Event processing can be traced back to the work of Xenakis in 1956 using the computer as an abstract compositional aid; 'an algorithmic composer' The output of a typical program generates the score for acoustic players or a synthesis program such as Csound. To adapt this concept to live performance it is natural to supply real-time event data, to an algorithm which generates real-time score as output. The score is then 'performed' by a real-time synthesizer. [5] The event data can be generated directly from physical controllers, or indirectly from the real-time analysis of an acoustic instrument sound. Commercial synthesizers were initially the only practical method of performing the score. It has already been noted that the attributes of these are often less than ideal. Risset has provided an unusual solution to this problem by using only acoustic sound [Ris90]. In his *Duet for one pianist* the event source is the MIDI output of a Yamaha Diskclavier keyboard and the score output destination is the MIDI input. The Diskclavier is a modern day pianola which contains velocity sensors and electromagnets. The algorithm adds notes to those played by the pianist. The piece is divided into eight sketches. The computer behaves differently in each, ranging from simple interval generation to giving the impression of a second, invisible player. There is certainly an element of 'live algorithmic composition' in Risset's work. However, the human and computer parts are sufficiently well bound to suggest an unusual instrument rather than an algorithmic 'animal' being provoked by the performer.

There are many others working in the field who concentrate on developing general purpose languages for interactive music. In *Real-Time Performance via User Interfaces to Musical Structures* [Pop93], a number of graphical computer music languages are presented, each with emphasis on different algorithmic techniques. Pope concludes:

> "I believe that most of them [the systems] could effectively be applied in demanding real-time environments to produce flexible, powerful and abstract performance instruments."

Unfortunately no reference is given to live performances using these systems, or the nature of physical interfaces that would be appropriate. Pope finishes with:

> "I believe it will soon be unacceptable to offer performance tools based on the current, low-level representations based on weak abstraction of events and signals.."

There may be truth here regarding algorithmic composition, and to some degree interactive composition, but this is not enough for live human performance. If performance on an instrument results in sound that could be produced by non real-time computer composition, surely the worth of the instrument is greatly reduced in comparison to another which produces sound which can only be produced by live human performance? Pope's instruments clearly fall into the former category. The question

[5] Or an acoustic performer perhaps?

is how to structure different levels of abstraction within a single elegant interface and instrument which exploits the uniqueness of human performance.

Score Following

Another aspect to live event processing is *score following*. A live performer performs a score written beforehand, and a computer attempts to follow the performance by reference to an internal representation of the score. Events are triggered at particular points in the score. The problem is first to find the pitch of the instrument and second to correlate the pitches to the score. Discussion on the latter part can be found in Vercoe [Ver84] and Dannenberg [Dan84]. In one instance the first part was resolved in the *instrumented flute* of Lawrence Beauregard. Switches relay the fingering. In combination with a simple acoustic pitch detector, the pitch can be reliably tracked in real-time.

2.3.2 Signal Processing

Digital audio signal processing is necessarily a much more demanding task than event processing, and so has a more recent history. Developments are now occuring at a rapid pace.

IRCAM and the 4X

At IRCAM the 4X workstation was developed amid increasing demand amongst composers for live signal processing. In 1988 the 4X was fitted with a MIDI interface allowing control from an Apple Macintosh. The ‘MAX’ graphical program was developed to ease the design of event processing programs which interfaced with the signal processing capabilities of the 4X [Puc86]. MAX has since been made commercially available, and been widely used for performing general event processing with MIDI synthesizers.

The Samson Box

At Stanford University California, dedicated synthesis hardware was developed through the 70’s leading in 1977 to the commercial release of the *Samson Box*. This was used successfully by many composers over a decade. Although fast and elegant in design, implementing basic new synthesis structures such as physical models proved in many cases impossible. The discrete logic hardware shared the same basic programming problems as the DSP chips which appeared in the early 80’s. Another, indirect, offspring of Stanford was the Lucasfilm Digital Audio Processor capable of processing 64 channels of sound at up to 50-kHz sampling rate. Functions included those of a mixing desk fitted with synchronisation controls for video, plus sound effects capabilities.

RISC vs DSP

More recently IRCAM developed the successor to the 4X based on two Intel i860 RISC processors. This marked the beginning of a new trend towards the use of fast, general purpose RISC and Super Scalar processors, rather than DSP chips, despite DSP chips being cheaper for a given computational speed. Because of its flexibility, RISC is used in many more applications than DSP, which means the corresponding high-level development tools are cheaper and more widespread. Compilers for DSPs often find it difficult to produce optimal code because the instructions have many side effects. Hand coding is frequently necessary. By contrast, C compilers with rigorous optimisation are standard tools for RISC chips. Using C has the usual advantage that it is portable across the many supported processors.

Desktop Signal Processing

Today composers have access to affordable personal computers capable of real-time signal processing and synthesis. The introduction of RISC and associated technologies is creating a renaissance in experimentation with synthesis, where previously rigid DSP instrument design systems dictated only certain possibilities. The machines are compact enough to be transported to a live performance, and an ever widening range of multichannel audio interface cards is available. The *C* programming language can be used to directly code programs incorporating event and signal processing. While this approach offers the greatest flexibility and efficiency, the composer may consider using some of the following higher level systems to achieve results more quickly.

Real-time Csound

Csound developed MIDI compatability in 1991. The result is primarily a real-time synthesis and signal processing tool, with little scope for event processing. One problem with moving Csound into the real-time interactive domain is that it was conceived precisely for non-interactive use. Nevertheless, Csound contains a powerful and friendly signal processing structure, and with ingenuity interesting performance instruments can be built.[6]As desktop computers become ever faster, so the complexity of the instruments can be scaled upwards. The open availability of Csound for use and development has certainly been a factor in its popularity. It may become a victim of its own success however, as the development of several non-standard versions is overtaking the original source from MIT. This is partly because MIT have developed a commercial version for use with specialist hardware. Such a move appears shortsighted, not least because any hardware specific development must now have a very short life span against the background of rapid personal computer improvement. The technical details of real-time Csound are discussed in the Chapter 4.

[6]See for example the instruments described by Menzies, [Men95].

Real-time Graphical Signal Processing

MAX has been extended into a self-contained real-time signal processing and instrument design system. *Synthbuilder* designed at CCRMA, Stanford University, is a purpose built instrument design and realisation package for the NEXT computer which includes a 56001 DSP, [Wat96]. Full graphical editing of audio and MIDI unit generator networks, graphical editing of function curves and test-probing of lines provide a friendly and comprehensive development environment. A version for Windows'95 is shortly to be released commercially. Synthbuilder was the testing ground for many of the advances in physical modelling which originated at Stanford. These advances have now been licensed out to companies producing 'synth design' packages of their own, *Reality Systems*, for example. A number of shareware software synthesisers have been produced, of varying flexibility, many dedicated to the simulation of analog synthesizers. The result is a rich, if somewhat chaotic, environment for the composer of live electronics.

Physical Modelling

Over the last 20 years considerable effort has been directed towards implementing physical models of instruments on computer. The initial intention was to compare the behaviour of models with real instruments, but not in real-time. Waveguide implementations of reed and bowed instruments by Smith, [Smi86], made real-time physical modelling feasible. Yamaha now own the rights to much of the underlying technique, which was first exploited in the form of the VL1 synthesizer, released in 1993. Other manufacturers such as Korg and Technics have produced models incorporating physical models licensed from Yamaha.

Smith has predicted that physical models will become increasingly important in comparison to more abstract synthesis methods, because they provide a more intuitive basis for the composer to explore new sounds. As yet physical models have not featured strongly in electroacoustic music, despite the availability of hardware and software implementations. The main obstacle appears to be the lack of suitable control interfaces. This applies equally to commercial synthesizers and applications such as Synthbuilder. For example a flute like sound can be controlled from a MIDI keyboard by triggering suitable breath envelope and embouchure information to give a stable tone. Using a breath controller to control breath pressure gives more control, but that still leaves the many details of embouchure and fingering which a performer of a real flute-like instrument knows must be controlled precisely to achieve an expressive musical performance.

2.3.3 Hyperinstruments

There has been a growing tendency to combine elements of signal and event processing in live work. The Beauregard flute was developed to enable live score following. The events generated by note recognition were used to control aspects of signal processing applied to the flute. At MIT, work of this kind by the composer Tod Machover, notably with the cello, has led to the term *Hyperinstru-*

ment. This is an acoustic instrument which has been *extended* by attaching electronic sensors. The sensor information can be processed in an event-like or signal-like way to generate electronic sound which adds to the natural acoustic sound. As well as microphones recording the sound generated by the cello, sensors attached to the bow and elbow of the player transmit bow position and angle information. Machover has endeavoured to extend solo performance to orchestral levels of complexity by generating complex electronic parts. The overall identity of the instrument remains unclear however, as the electronic sound generation processes are subject to major variations between pieces. There is the associated problem of reconciling synthetic or processed sounds with the natural sound of the instrument. The natural sound of the hypercello is deliberately suppressed by modifications to the soundboard. If this process were taken to its conclusion the player would no longer hear the acoustic sound, but the bow and string dynamics would be unchanged. The cellist Yo Yo Ma has performed works with the hypercello, to critical acclaim. For a review see *Taming the Hypercello* [Lev94].

2.3.4 Commercial Physical Controllers

Although commercial controllers are subject to market pressures, they should still be considered for use in electroacoustic music, especially as the instrument designer is free to develop his own, non commercial, synthesis system to use with the controllers. Commercial controllers contain some good ideas even if they are not constructed to the high standards expected from acoustic instruments.

Comparing Manufacturing Costs

The main part of the cost of a synthesizer comes from the development of its integrated circuits and software. For a commercial acoustic instrument most of the cost is in manufacturing to a high standard an established successful design. A large part of this is the actual physical interface with the player. The key and hammer action of a piano, the complex key arrangement of a clarinet and the precise dimensions of the mouthpiece. It remains to be seen whether a market for electronic instruments can be created in which a larger part of the cost is spent on manufacturing a higher quality physical interface.

The MIDI Keyboard

Commercially, physical controllers for electronic music have been dominated by the MIDI keyboard, which has been strongly associated with the characteristics of MIDI synthesizers. The serious musician can choose to ignore these associations and treat the keyboard purely as an interface, as the controls generated by the keyboard are directly available at the MIDI out socket, independent of any synthesis function the keyboard has. Keyboards with delicate physical properties such as the Kurzweil or Fatar weighted keyboards are particularly fine. These contain a hammer mechanism that emulates the tactile response of a piano very well in a compact form. The important aspects of the piano mechanism from the control viewpoint, are that a succession of events can be generated with high resolution in both time and velocity. This will be discussed more fully in Chapter 3.

Aftertouch

As well as velocity information, most MIDI keyboards can send *aftertouch*, the pressure variation of fingers on depressed keys. Usually the maximum pressure over all keys is sent, but some models can send *polyphonic aftertouch*, which means that independent pressure readings are sent for each depressed key. Examples include the Ensoniq EPS and EMU Proteus series. Polyphonic aftertouch opens up very interesting new performance possibilities, but is rarely used, partly because commercial synthesizers have inadequate architectures for making use of many controllers, and also because MIDI bandwidth restricts the number of simultaneous controllers. Menzies gives an example in Csound, [Men95], in which a live sound source is processed by a bank of band-pass filters, with each filter controlled by the finger pressure on one key.

Expression Wheels, Joysticks and Ribbons

Other common performance features of MIDI keyboards are the *expression wheels*. These are generally positioned at the left end of the keyboard so the player can quickly reach with the left hand while continuing to play with the right hand.

Moving the wheels generates MIDI controller messages which are by default assigned to pitch bend and modulation. The pitch bend wheel usually has a spring return to the centre to help the player return to normal pitch, while the modulation wheel is unsprung so that it can be left untouched at any value.

The attraction of using the wheels is diminished by the strong associations with ways in which they are normally used. The direct control of pitch with the pitchbend wheel creates crude and cliche effects, likewise for the routing of the modulation to vibrato. There is no practical reason why the wheels cannot be used more creatively, by using the control signals as input to an external process.

Some keyboards include joysticks which may send external MIDI control messages. The quality varies but good examples are found on the Yamaha SY series. The joystick is a convenient of way a 2-dimensional variable with one hand. Recently there has been a trend to other forms of multi-dimensional control. *Ribbons* are flat plastic surfaces on which the player moves a finger. They can detect the position of one finger and in some cases the applied finger pressure as well. The Korg Prophecy includes a ribbon mounted on a rockable barrel so that one hand could potentially send 4 continuous control signals.

MIDI Windcontrollers

Windcontrollers mimic the control interfaces of real wind instruments, which consist of finger control, breath control and possibly additional embouchure controls. The motivations are two-fold, first to give the wind player a way of applying their skills to electronic music and second to extend the control possibilities of the keyboard. Several have been produced by different manufacturers. Their use has been mainly restricted to modern electronic jazz. Programming a commercial synthesizer to work successfully with a wind controller is not straight forward, as most synthesizers are only designed to work with keyboards. Even with special programming effort the results are disappointing in comparison with a real wind instrument.

A common design problem of wind controllers is how to decode fingerings. As a player moves from one fingering to another, a number of other fingerings may be crossed on the way. The decoder must decide which fingerings will cause a note event, without generating any false notes. If the decoder waits too long for a fingering to stabilise, the note event will be delayed. The root cause of the problem is that the windcontroller simply determines whether a finger is open or closed, whereas the state of a real wind instrument is governed by the continuous position of the fingers or keypads.

The AKAI EWI (Electronic Wind Instrument) windcontroller and analog wind synthesizer have been available since 1989, and provide a rare example of a windcontroller with a specially developed synthesizer.[7] The EWI has pressure sensitive keys which are used to add continuous expression to the analog synthesis. Opinion[8] on pressure keys appears divided, some people finding pressure control difficult to combine with rapid keywork and tiring to sustain. AKAI also market an instrument modelled on the acoustic trumpet called the EVI (Electronic Valve Instrument). This has valve like switches for controlling pitch, but more importantly, a *lip pressure* sensor in addition to the breath pressure sensor. The lip sensor can be used to trigger harmonics.

Yamaha released the VL series of physical modelling synthesizers which work fairly well with the Yamaha WX range of wind controllers. These have keys which function only as switches. The key arrangement is similar to a saxophone with extra register keys for changing octaves. MIDI note messages are generated by the same combinations of keys used to generate the notes on a saxophone. This is convenient, but it means that some combinations cannot be detected. It would be useful to include a mode for sending the state of each key, as on a keyboard, for instance by sending a note event when ever a key is pressed or released. Musicians often criticise the keys for being too light. The WX mouthpiece includes a breath pressure sensor and a plastic *reed* in the style of a clarinet, which responds to lower jaw pressure. Note velocity information is generated using the early part of the breath pressure signal. It is generally better to turn off velocity sensitivity if possible and just use the breath signal to control volume. The breath sensor does not detect the breath pressure applied by the player directly. As the reed closes, air pressure falls over the gap, and the pressure registered by the breath sensor inside the mouthpiece is less than in the player's mouth. The pressure in the mouth corresponds better to the energy that the player would be injecting into an acoustic instrument. The mouth pressure can be recovered from the controller breath and reed signals, see [Men95].

To quote the saxophonist Courtney Pine (*Sound On Sound*, Sept 96): "I'm still waiting for the perfect

[7]The *Casio Horn* integrated control, synthesis and sound production into a single unit, but it was produced as a toy. It would be interesting to apply the same concept but with a much higher design standard.

[8]Taken from informal conversations from the *WIND* list, *wind@morgan.ucs.mun.ca*

windcontroller to arrive." This sentiment reveals the common misconception that the control process is the only problem, whereas it is more helpful to criticise the controller and synthesis as one object.

The MIDI guitar

The great popularity of the electric and acoustic guitar in popular music has led to many designs for a MIDI guitar controller. The earliest of these had plastic strings and switches on the freeboard, for example the Casio guitar controller. More recently attempts have been made to use electric and acoustic pickups on real guitars to extract control information. These were initially hampered by the difficulty of following pitch quickly. This has been overcome in the Roland GR1 by triggering notes when onset transients are detected, and then mixing pitched voice information as the pitch becomes well established. In this respect the GR1 is a successful *hyperinstrument* interface, although it is likely that in a popular music setting the GR1 would be used mainly to allow competent guitarists access to keyboard sounds.

The Yamaha Diskclavier

The Diskclavier takes the hyperinstrument concept a step further by incorporating electromagnets which can play the piano in manner analogous to human fingers. MIDI output records human playing measured with the sensors, while MIDI input triggers the electromagnets. Its use has already been discussed in connection with Risset's piece 'Duet for one pianist' on Page 16.

The MIDI Drum

These have been around from the early days of MIDI and vary from practice pads, and shock sensing sticks, to expensively constructed pads that can measure a wide range of dynamics. In 1994 Korg introduced a sophisticated drum pad as part of the *Wave Drum* physical modelling drum. This pad can accurately, and independently, locate impact and pressure over its surface. The high price tag restricts the wavedrum's use to large studios. A little more affordable is the *V drum* series from Roland which has received high critical acclaim. Detection is with piezo microphones positioned at different points around the perimeter of the drum heads.

2.3.5 Non-Commercial Physical Controllers

The following controllers are mainly the product of research in academic environments and were designed for use in live electroacoustic music. The hyperinstruments previously described are related to this category by their sensor systems. In addition some designs are presented from outside academia and these are usually intended for use in popular music.

Few composers will be able to secure the use of these instruments. The value of this survey is to provide background information on a range of designs to help in the design of a new instrument.

The Radio Drum

The *Radio Drum* is the last in Max Mathew's series of drum controllers [Mat91]. The *Sequential Drum* detected 2-dimensional position and hitting strength of a mallet using a resistive mesh. The *Daton* was a more elegant version in which position and strength are found by measuring and correlating the stress measured at each corner of a rectangular surface. The Radio drum uses a lattice of aerials beneath the drum skin to detect the 3 dimensional position of a drum stick containing a low frequency radio transmitter. By using different frequencies more than one drum stick can be independently tracked. A natural use for detecting height above a drum skin is for generating a lead sound preceding the main strike. Mathews original intention for the drum was as a 'baton' which can be used to conduct an electronic score, thereby strengthening the role of the conductor as an active performer. The recognition and interpretation of baton gestures presents a difficult problem however, and it is clear that this approach is yet to reach maturity.

The Zeta Violin

The *Zeta Violin*, by Keith McMillan, is similar in concept to the Roland GR1 guitar controller. It is a solid-body electric violin with audio and MIDI output. Each string has its own pickup and pitch detector. Pitch information is available on the MIDI output. Schloss gives an example of a piece for Zeta Violin and Radio drum, in which the performances interact [SD93]. In one instance, pitches detected on the violin determine the pitches of chords triggered by the drum. Both instruments have been manufactured in small numbers, and have enjoyed growing familiarity and accessibility amongst composers. The Zeta violin is also occasionally heard in popular music.

The Hands

The Hands, developed by Michael Waisvisz at STEIM, Amsterdam, in the mid-80's are gloves fitted with sensors to detect pressure and movement of fingers and hands relative to each other [Kre90], [And94]. This approach has been repeated several times elsewhere, especially in the context of total immersion virtual reality in which the gloves may additionally create pressure mechanically on parts of the hand to simulate touching or lifting solid objects.

The Web

The Web is similar to The Hands in that it responds to individual finger movements, but is, perhaps, more visually interesting [Kre90], [And94]. A spider-like web with a number of tension sensors in the interconnecting segments is manipulated by pulling with the fingers. The tensions are each affected by all the fingers to varying degrees. The natural vibrations of the web add dynamic structure to the control output which can also be sensed with fingers and seen by the player and audience.

The Light Baton

In *The Light Baton System* [BC93], a controller is described for use mainly with interactive composition; ‘The aim is to endow electronic music with the expressiveness and feeling which are characteristic of live performances’. There is no reason why the baton could not be used in actual live performance. In this respect it would be novel, because a traditional conductor’s baton has only an indirect effect on the music via the attention of the players. Another use is for training people in conducting technique. The light baton is essentially just a normal baton with a light at its tip, the position of which is tracked with a video system. The efforts of the developers have been concentrated on extracting beat information from the video data by image analysis and gesture recognition.

Modular Feedback Keyboard

The *Modular Feedback Keyboard* developed by Cadoz et al, is a no-compromise approach to developing a keyboard interface [CLF90]. Each key is equipped with a *sensor-motor* module which has two functions. First to precisely detect the angular position of the key and second to generate a controllable force on the key. The specifications of force control in terms of bandwidth and maximum force magnitude are exacting, in order to simulate the inertial reaction of a grand piano from pianissimo to fortissimo. To achieve these, a patented *slice motor* has been developed in which a strong magnetic field is created by alternately linking the fields from the separate modules across the keyboard. The generality of the system means that the arbitrary key dynamics can be simulated, which leads to the concept of *instrument synthesis* discussed in the paper.

> “We are not simply aiming at improved ergonomics of gestural control in sound synthesis, but rather at a fundamentally new insight into musical synthesis itself: our approach is focused on the importance of the instrumental relation in both the learning and the intrinsic process of musical creation. We therefore were led to propose not only a synthesis of the sound but also of the instrument.”

The modular feedback keyboard is a bulky and costly item, useful for research, but impractical for manufacture. Even if it could be cheaply reproduced, it is likely that musicians would not want to alter the touch of the keyboard much, so its flexibility would be wasted.

Bio Electric Controller

Bio Muse is an example of a bio electric controller system that has been used for live music generation [KL90]. EMG signals are obtained from the head and limbs using electrodes help in contact with velcro straps. A substantial amount of noise must be filtered electronically to produce signals that can be correlated easily with muscle contraction and brainwaves, and thus form part of a musical feedback system. The awkwardness of the equipment and its limited sensitivity make it an unlikely candidate for live music. BioMuse has, however, been used successfully to give severely handicapped people an opportunity for musical performance, and work has begun to use it in the rehabilitation of paralysis.

preFORM

The *preFROM* system by Xavier Chabot et al [Cha90], is an integrated system for live electronic performance. For physical sensors, Chabot appeals to the aesthetic of dance.

> “With the electronic instrument, sound and musical discourse are unified and internalised in the performer’s body; the body itself becomes instrument, sound and music. We are now far removed from the concept of the instrumentalist as triggering actor, which is so deeply anchored in peoples’ minds that it is still the basis for most so-called performance electronics applications.”

The physical sensors used by preFORM are as follows:

Sonar System

This is simply used to measure 1-dimensional distance to the performer. Chabot describes the control available using this as a ‘straight and dense linear space’.

Angular Detector

This takes the form of a pendulum in oil, encased in a unit which can be worn by the performer. The movements of the performer combine with the dynamics of the damped pendulum to provide a ‘physically filtered’ form of the motion of the performer. Under certain conditions of motion, the pendulum provides a fairly accurate measure of rotational position. The indirectness of the unit output is not so great as to confuse the performer. On the contrary, it adds some musical interest.

Airdrum

The Airdrum is essentially two sticks each with a 3-dimensional accelerometer at their tip. Theoretically drum like gestures can be simulated. Additionally the more subtle accelerations occuring before impact can be used for musical purposes, in the manner of the Radio Drum.

The HIRN breath controller

The *HIRN* is a ‘Meta Wind Instrument Controller’ by Perry Cook, [Coo92]. It was built for use with a ‘Meta-wind Physical Model’, which is a simplified generic synthesis model that can be adapted to reed, brass or flute instruments. In addition to a breath sensor and switch-keys a number of other controls are incorporated. The mouthpiece has bite sensor and its rotation about an axis perpendicular to the bore is also measured. The lower set of keys is mounted on a sliding, rotating barrel. HIRN is impressive in its seemingly exhaustive use of the available mechanical degrees of freedom, with the exception of the keys, which are switches only and cannot detect continuous movement.

The LightHarp and LaserLyre

The *LightHarp* and *LaserLyre* are instruments by Stuart Favilla which use LDR light sensors to detect the interruption of light by the player’s fingers, causing MIDI note messages to be triggered [Fav94]. The LaserLyre uses several lasers each aimed at a separate LDR. With smoke added the ‘light strings’ become visible. Favilla has been greatly concerned with the ergonomics and aesthetic qualities of his instruments, borrowing from the Yogic symbolism of the Indian *Vina* instrument.

> “The aesthetic design of commercial controllers reflects a generic high-tech commercialised Western culture, which contradicts the cultural aesthetic of Asian musics.”

2.4 Popular Music

Popular music is not the prime concern of this report, but since it has influenced the development of contemporary music, technologically and musically, it must be considered. Peter Manning summarizes this influence concisely, if a little gloomily, in the opening paragraph of the chapter *Rock and Pop Electronic Music* from his book *Electronic and Computer Music* [Man85]:

> “Electronic jingles invade the lives of millions everyday via the media of radio and television. These ephemera, while familiarizing the public with the nature of synthesized sounds, have debased electronic music to the level of an advertising aid. Such a widely recognized application, however, is only one step higher in the public consciousness than rock and pop music which uses electronically generated material as a matter of course. Today the commercial digital synthesizer is ubiquitous, serving in many instances both the serious and popular music sectors with equal success. As a result the primary distinctions have become those of application rather than of technology. This situation contrasts sharply with the circumstances surrounding the pioneering phase of synthesised rock and pop music. Here experimentation by a number of leading artists with new methods of sound production inspired a range of technical developments which were to prove highly influential on the evolution of the electronic medium

as a whole.”

2.4.1 Recording and Live Concerts

Popular music began with the commercial development of recording techniques. Records effectively advertised themselves on the radio. The aesthetics of popular music are mainly folk in origin but have been heavily affected by the cultures surrounding the recording industry. The development of magnetic tape opened up new possibilities for recording and processing sound. Editing could be achieved by cutting and joining tape, *splicing*, or re-recording over sections of tape. The number of concurrent channels is, in principle, unlimited. This led to the concept of the multi-track recording studio, first suggested by Les Paul. The studio quickly became a creative environment for recording and producing records. Musicians speak of recording a ‘performance’ in the studio, but in many cases this will be very different to a live performance in front of an audience. The studio is a place of introspection where the musician works more like a craftsman rather than a showman.

2.4.2 The Electric Guitar

Electronic amplification enabled performance to large numbers of people, furthering the influence of popular music. The electric guitar was invented with this in mind, notably by Les Paul. It has the special advantage of having negligible acoustic feedback, permitting very great amplification in a live setting. The vibration of metal strings is converted directly to an electrical signal with electromagnetic pickups. This facilitates signal processing in many forms. The *Wah Pedal* is a foot controlled effect which filters the guitar signal. Details vary between models, but the general effect is 12 dB/octave or more low pass filtering with resonance at cutoff. Pressing the pedal raises the cutoff frequency, creating the characteristic vocal ‘wha’ sound. *Distortion* has been very important in building the identity of the instrument. It was originally found accidentally by overdriving valve pre-amplifiers, which creates soft-knee waveform clipping effects. When transistors replaced valves, it became clear that it was not easy to recreate the richness of the valve distortion, and that more subtle processes were at work than just waveshaping causing harmonic distortion. FET devices have proved more popular than bipolar transistors, and there are now some digital implementations. The guitar itself is very important for making a particular distortion. The *humbucker* pickup consists of two parallel lines of poles wired out of phase. The original purpose was to reduce mains hum interference under high amplification, but now the *out of phase* sound from the distorted humbucker had become popular in its own right. The spectral simplicity of the guitar pickup signals is important for good distortion effects. Other instruments with a more complex spectrum quickly loose their harmonic coherence as the distortion is increased.

The electric guitar is a modern example of an instrument created by a musical genre which has then gone on to promote the development of the genre. This demonstrates the way in which instruments interact with the musical styles they inhabit.

2.4.3 Synthesizers, MIDI

The Thérémin discussed previously on page 7, was the first commercially available music synthesis instrument, and has been used by a number of popular artists over the years. The keyboard, however, has proved to be the dominant vehicle for synthesizers. The technical details of the synthesis techniques presented here are discussed in greater detail in the next chapter.

The Hammond Organ

The earliest of these were electro-mechanical in design. The original Hammond Organ generated tones by rotating metal plates in magnetic fields. Each plate has a varying radius about its circumference. By rotating the plates together harmonics can be readily generated. The strength of different harmonics can be controlled with *draw bars*. Effects can be applied to the sound, most notably with the *Leslie tremolo* unit. A distinctive tremolo can be progressively applied with a draw bar. The overall result is a subtle organ-like instrument. The modern Hammond, while implemented with digital electronics, faithfully retains the features of the original including the high price tag.

The Mellotron

The *Mellotron* was unlike any other keyboard instrument. Each key controlled the playback of a tape loop sample. It was the forerunner of the modern sampler.

The Fender Rhodes

The *Rhodes* is the classic electric piano'. Strictly this is not a synthesizer but a hybrid like the electric guitar. Short piano-like wires are hit by key hammers, and the vibrations are transduced with electromagnetic pickups. The tonal qualities introduced by the pickup, result in a wide variation of tone from bell like piano to a hard, almost distorted forte. The vibrations of the strings, although not heard directly, can be felt via the keys.

Transistor Synthesizers

The thermionic valve was used to synthesize sound in the early electronic music studios, but it was not until the advent of the transistor in the late 50's that analog electronic synthesis became widespread. The small size, low power and relative stability of transistor circuitry led to the rapid commercial availability of keyboard controlled synthesizers, based on the synthesis designs previously developed in the studios. The electronics were separated into modules which could be physically interconnected with patch chords or peg boards. A lucrative period of expansion followed, dominated mainly by the companies *Moog*, *Buchla*, *ARP* and *EMS*. Although intended for popular music, contemporary composers benefited from these developments as well. The open, visual structure of control decks proved very popular. As well as patching options, knobs and sliders provided a direct means to experiment and perform with synthesis. In retrospect, another asset of analog synthesizers was their lack of predictability. This gives the music characteristics which are reminiscent of acoustic instruments, for instance the non-uniformity of starting transients and note transitions, and slight pitch instability.

Hybrid Synthesizers: The Oberheim OBX and Roland Juno

As integrated circuit technology found its way into synthesizers, the control circuitry was the first to be replaced. This made polyphony management practical and allowed patch arrangements to be stored in memory and quickly restored. The Oberheim OBX and Roland Juno series represent the hybrid class well. The Oberheim Matrix 1000 is still successfully in production, after 10 years of production. It is derived from the OBX series but replaces analog oscillators with digital, while retaining the analog filters. Thus it combines many of the luxuries of modern synthesizers such as control mapping and multi-timbrality with the analog 'sound' created by using analog filters. Analog filters are often described as being musically 'warmer' than digital clones. This is due to slightly non-linear response characteristics combined with the complete absence of alias distortion, which is most noticeable with wavetable synthesis when notes are transposed downwards.

One unfortunate side effect of digitization is that the number of knobs and sliders rapidly diminished on many keyboards, replaced with a few buttons and some 7-segment displays. The same trend is found in the development of all-digital synthesizers.

Digital Synthesizers

At the start of the 80's the market was dominated by analog equipment. The only all-digital commercial machines were very expensive; the CMI Fairlight and the New England Synclavier. Improvements in VLSI chip design and fabrication, enabled cheaper synthesizers to be produced. Casio was first to break the market with the *VL-1* costing less than a hundred dollars. Including basic, low quality synthesis with preset rhythms, the VL-l heralded a design concept which is still commercially successful today. This is an example of the 'gizmo' factor which is a widespread phenomenon in digital consumer electronics.

On analog and hybrid systems, analog controls such as sliders were a cost effective means for controlling oscillators and filters because they interfaced directly to the analog circuits. Complete digitization favoured digital input methods, such as switches and 7-segment displays. To implement an analog-style control interface would require a relatively expensive multichannel analog to digital conversion system, with additional processing concerns about rate conversions. After years of complaints about the lack of analog controls on digital synthesisers, there is now movement back to the older style.

The Yamaha DX series

Yamaha entered the market in the mid/high price range, with the *GS1* and *GS2* synthesizers. These led the way to the development of the classic *DX* series, which featured innovative use of FM synthesis. In conjunction with John Chowning of Stanford University, Yamaha took out a number of patents on FM synthesis, which effectively blocked competition. The DX7, launched in 1983, is a 5 octave, 16 note polyphonic, mono-timbral synthesizer. Each voice has 6 oscillators, ’operators’, which can be configured in one of 32 preset arrangements. Allowable connections between oscillators are frequency modulation, mixing and a single feedback path. Each oscillator has its own programmable envelope which modulates the amplitude and a programmable static frequency modulation depth. The frequency of the oscillators can be fixed as a multiple of the voice frequency with an optional frequency offset. A single low frequency oscillator can be applied to the operator amplitudes using an external controller. A number of external controllers are recognized; aftertouch, modulation wheel, foot pedal, and breath controller. These can each be assigned to differently affect the LFO and oscillator amplitudes. Unfortunately each oscillator cannot be independently assigned to each controller, which would have offered many more performance possibilities.

The DX7 is an impressive machine for timbral experimentation and performance, although the programming features have been largely ignored by owners due to their complexity and lack of an intuitive interface. Its very great commercial success, which contributed to the funding of CCRMA at Stanford University, lay in the large numbers of high quality *patches*, preset voice programs, that became available on plug-in cartridges. These included some convincing natural sounds, especially bells and electric pianos, and many unusual sounds.

From the viewpoint of keyboard performance, the DX7 offers wide and sensitive tonal variation according to key striking velocity, which makes it particularly suitable for percussive sounds. The variations are not created with simple filtering or cross-fades as on many later keyboards, but result from complex spectral variations occuring in FM systems when the modulation depths are increased. Small variations between two notes struck with similar velocity are enough to give a sense of acoustic-like unpredictability. As well as percussive sounds, multi textured, slowly evolving sounds can be generated and expressively controlled with continuous controllers.

The recognizable sound of the DX, which has been its strength has also to some extent been its weakness. There have been frustrated attempts to match FM synthesis to general sounds. This is to overlook the value of FM as an abstract sound generation technique that is well suited to performance control, as evidenced by the DX enduring changes of musical style for over a decade.

MIDI

The *Musical Instrument Digital Interface* was the result of a collaboration between several synthesizer manufacturers in the summer 1982, following a meeting at the *National Association of Music Merchants* the previous year. The aim was to connect different synthesis and controller products regardless of manufacturer, so that for example one keyboard controller could be used to access a number of different synthesizers. The specification is heavily biased towards keyboard control. In addition to *note on/off* messages there is provision for 128 different *controller* messages which can each hold 7 bits of data. Other short messages exist for synchronisation and *active sensing* used to detect the presence of broken MIDI connections between devices. Other longer, and therefore slower, messages can be sent via the *system exclusive format*, the main part of which is machine dependent. MIDI has been a great success commercially and has created new markets for stage and studio equipment. The *MIDI sequencer* was developed as a multi-track recording and editing system for MIDI information. Initially in the form of portable hardware devices such as the *Alesis MT-8*, sequencers soon spread onto personal computers, gaining greater flexibility. One of the earliest software sequencers was *Steinberg Pro24* from 1985, which made use of the built in MIDI hardware on the *Atari ST*. Devices other than synthesizers such as effects and mixing desks acquired MIDI inputs so that everything could be run in an organised fashion from one sequencer. MIDI↔SMPTE conversion devices allowed MIDI to be synchronized with audio tracks on tape. As personal computers became more powerful it was possible to integrate the recording and playback of audio files direct from hard-disc with MIDI recording and playback. More real-time effects and falling latency mean that the computer is increasingly becoming the centre of the studio.

MIDI was originally designed to help live performers reduce their equipment count. Today, you hardly ever see a MIDI cable in live use. The studio has become the dominant force driving developments in MIDI. This has had an adverse effect on the development of devices for genuine live performance. The Earlier analog systems were much more convincing as live devices.

Wavetable Synthesis

To counter the DX7 competitors had to develop new synthesis techniques. The most enduring of these has been the wavetable synthesis method. The basic operation of wavetable synthesis is to loop a digital sample of a natural or even a synthesized sound. Being memory intensive, the development of wavetable synthesis has progressed as the world prices of memory have fallen. It was first demonstrated on the *Fairlight*, which incorporated hardware for sampling sound. The *E-mu Emulator I* was the first widespread wavetable synthesizer, with wavetables in ROM. The Kurzweil 250 marked the first of the prestigious Kurzweil synthesizers. These were unique in that all the internal audio processing was implemented in floating point hardware, resulting in less noise in small signals. Kurzweil have also designed a flexible architecture called VAST, which allows control parameters to be cross-processed with programmable functions that are fed to the synthesis components

The Sampler

Parallel to the commercial development of wavetable synthesis was the sampler. This is effectively a wavetable synthesizer with RAM for holding samples, AD converters for sampling sound, and editing software for guiding the user through the sampling process. Samplers have traditionally been aimed at a higher, more professional budget, due to the high early cost of RAM, and the need for good editing software and display hardware to make them worthwhile. The *Akai 900* and the *Ensoniq EPS* released in 1986 are good early examples. The EPS is entitled ‘performance sampler’, and comes as a 5-octave keyboard which significantly includes polyphonic aftertouch and ‘performance switches’ near the modulation and pitch wheels. The switches can be used to select layers of samples with the left hand whilst playing the keys with the right. For instance, a flute like sound can be played in different styles. The layout and design of the editing software facilitates rapid work and the operating system is held in RAM allowing easy updates.

Mass Market Wavetable Synthesis

Towards the end of the eighties a number of manufacturers had produced cheap synthesizers with essentially similar wavetable systems, using short samples. Roland introduced the *LA* synthesis system in which four oscillators each reading from a wavetable are combined into a single keyboard voice. Detuning, amplitude modulation and envelopes constitute the bulk of the remaining architecture. Extra features such as amplitude modulation, often omitted from samplers, are incorporated to enrich the use of short samples.[9]

Multi Timbrality

Multi-timbrality the ability of a synthesizer to play different patches on different voices at the same time. Initially this came in the form of simple keyboard splits, such as bass in the left hand and the lead in the right. With the increased use of multi-track sequencers and backing tracks for home-organs, multi-timbrality rapidly became standard.

The Korg M1

In 1990 Korg introduced the M1, which was notable for the use of substantial transient samples in conjunction with loops. The transients allow a much more convincing synthesis of natural sounds. Also significant was the inclusion of on-board effects processing for reverberation and 8-part multi-timbrality.

Vector Synthesis

Prophet introduced a system in the *VS* for dynamically mixing several samples. This led to a number of manufacturers including joysticks on keyboards for mixing. *Vector Synthesis* has proved a successful method of integrating performance and synthesis.

[9]Samplers are wavetable synthesisers as well, but the emphasis has always been on using long samples which can be changed from one piece to another, rather than combining and processing fixed samples.

The Korg Wavestation

Korg introduced a vector synthesis system based on the earlier Prophet system, and synchronized the sequencing of samples with vector movement. The control flexibility of the wavestation has won it many admirers, but predictably, programming is complex and requires a computer editor.

The E-mu Z-plane synthesis

The *Morpheus* synthesizer by E-mu introduced a 14 pole dynamically variable filter called the *Z-plane*. The high order of this filter is used to perform spectral-morphing like effects between different samples. This is essentially an extension of vector synthesis technique, with good control flexibility .

Physical Modelling Synthesis, the Yamaha VL-1

The efforts of the academic community in synthesis by physical modelling, principally by Julius Smith of Stanford University, have led to a number of recent commercial products. The first of these, the *VL-1 Virtual Lead Synthesizer* released in 1994, incorporates models of reeds, pipes, strings and bows, but is only duophonic. By playing the keyboard it is immediately apparent that the sounds contain unstable and unpredictable elements. The real value of the VL-1 is in producing completely new instruments which combine some acoustic 'feel' with unusual sounds. A high quality digital effects processor complements the raw VL sound very well.

The Korg Wavedrum

The Korg Wavedrum is a synthesizer physically modelling drums, using techniques licensed from Yamaha. It was previously discussed as a controller in Section 2.3.4, as it incorporates an advanced drum head capable of position and static/dynamic pressure sensing. It is critically acclaimed but also very expensive.

The Korg Prophecy

The Prophecy is the first Korg synthesiser to combine a number of synthesis structures including FM and physical modelling with a physical interface designed for rapid control of parameters. A row of knobs can be assigned to act on the sound in different ways and a ribbon detects position and pressure. Despite being only monophonic and originally costing around 1000 pounds when released in 1995, the Prophecy became an instant success. The subtle instabilities in the sound and the controllability are the two important factors.

The Nord Clavier Lead

This is another monophonic keyboard synthesizer which operates and sounds like a traditional analog synthesizer, except that the analog circuitry has been replaced with DPS chips modelling analog circuitry. The result is authentic analog quirkiness combined with easy experimentation and reliability.

The Yamaha CS-1

The CS-1 was the first of a recent development in *controller keyboards*, with emphasis on live performance control of conventional digital synthesis. A generous number of knobs and an ergonomic layout make it very accessible. This is very much a step backwards to older analog style keyboards, but with the quartz precision of modern digital electronics.

Quote

> "Build synthesizers with a few sounds but with lots of ways of articulating them."

This old advice from Brian Eno to the synthesizer manufacturers is being heeded. Richness of articulation in a single object is the key to successful live performance. Although synthesiser design now appears to be improving generally, it is easy to be dazzled by technological advancement without assessing the true musical worth of new equipment.

2.4.4 Hiphop

The *Hiphop* style of popular music was born in the ghettos of New York in the seventies. Hiphop traditionally requires at least two performers, a DJ using record decks and a *rapper* who talks and sings with a distinctive rhythm. A traditional DJ merely selects records, fading one into another, possibly with the tempo or pitch matched by applying small variations in playback speed. The Hiphop DJ extends this technique into a performance art.

The records are chosen to provide rhythmic backing loops and effects for the rapper. To create a loop, the DJ plays a section on one deck then switches to the start of another on the other deck, while rewinding the first deck manually to the start of its section. The DJ's mixer has evolved into a specialist device containing a fast moving lever for switching rapidly from one deck to another without glitching. *Scratching* is the technique of manually oscillating a record to create rhythmic sound effects containing wide pitch variations. The mixer lever can be used in synchronization with the oscillations to enhance the technique. The inertial dynamics of the turntable lend scratching a distinctive character. It is remarkable that such well established and influential style of popular music has developed by 'misusing' record players!

2.4.5 House

Like Hiphop, *House*, was born in Chicago in the 70's and is based around a DJ performer. The original House DJs combined processed vocal tape recordings with an uninterrupted high energy rhythm, very different to the flexible, broken rhythms in Hiphop. Tape recorders were used to edit and process sections for later live used. Today House is synomynous with any mainstream dance music containing a vocal component, whether the DJ performs to any extent or not.

2.4.6 Electronic Music

Within popular music, sub-genres of studio-produced instrumental music exist which share some of the aesthetic preoccupations of electroacoustic music with timbral control. There are two broad divisions. *Ambient* stems from Brian Eno's work in the early 1980's, characterised by slowly evolving textures. *Techno* is based on powerful electronic rhythms and originates from German bands of the late 70's such as *Tangerine Dream* and *Kraftwerk*. Today there are many derivatives; *Hardcore*, *Trance* and some hybrids such as *Drum and Bass* which combines electronics with the Jamaican *Ragga* beat. Some of the more experimental artists have worked with with well known contemporary composers, for instance Richard James with Philip Glass.

Live Techniques

It is difficult to recreate the complexity of a carefully prepared studio piece on a stage convincingly. It is the same problem that faced Pierre Schaeffer when he tried to give a live concert of musique concrète in 1949. Modern technology offers some practical solutions. *The Utah Saints* produce fast, sample based music. They perform live by having pre-prepared sequences which they mix and alter while the sequences are playing using MIDI keyboards connected to computers and samplers. Rather than expression with subtle muscle gestures, the excitement of improvising with structures is being communicated. *The Prodigy* write music in studios with live performance very much in mind. They combine the repeated playing of 4 or 8 bar samples of their records with live guitar, vocals and sequencing. Bristol is known for a characteristic live sound created by bands such as *Massive Attack*. Hiphop style techniques are combined with live use of samplers and sequencers. Other artists such as the *The Orb* have taken more theatrical attitudes towards live performance by playing chess on the front stage after starting their music. Besides mainstream recording bands, there a number of artists who exist as performing DJs, earning their living by touring prestigious nightclubs.

2.5 Summary

Real-time vs Live

The concepts of interactive and algorithmic composition have been manifestly beneficial for the production of tape work. However, their influence has led to the creation of numerous ‘Real-Time Performances’ in which the presence of the human performer is almost incidental. Emmerson supports the view that more effort should be given to make the performance of electronic music more *human*,[Emm91]:

> “Now I want to argue for a difficult distinction. ‘Real Time’ is not the same as ‘live’. At the ICMC in Glasgow (September 1990), one had the spectacle of four performers sitting giving serious attention to VDUs, and rather in the manner of an advanced weapons interception system ‘reacting’ to emanations from each other. ‘Live’ it was only in the sense that it was not pre-recorded - the actual actions of the performers were almost exclusively finger on QWERTY keyboard, even the few times MIDI keyboards were used the actions often had the same trigger function. Thus ‘instrumental’ and ‘human’ gesture were replaced by algorithm, the human relegated to the level of a sophisticated trigger/response mechanism. This description is independent of the specific software this group was using, it might have been anything from a simple choice list to one based on learning procedures. The listener had no way of knowing.”

Live Music

Emmerson identifies two tangible characteristics of his notion of ‘live’ music. First there is the correlation between the visible performance and the audio performance, and second, possibly more important, are the inherent qualities of the sound which associate it in the minds of the listener with human performance, even without visuals. To illustrate the second point consider a simple melody such as ‘three blind mice’ played from a score by Csound. Even with quite elaborate devices for introducing vibrato, random factors, phrasing and numerous other considerations will the result compare with the same played by a violinist. The violin is a medium through which many human psychological traits are expressed, in much the same way as the human face communicates human thought processes by different facial expressions. These reveal the underlying complexity and intelligence of the human mind even when, on the surface, the information does not seem inherently different to the information generated by a mechanical process.

MIDI Physical Limitations

Some identify the MIDI standard as a barrier to expressive control. The bit rate is 31Kbits/s. Each byte is sent with a start and stop bit added. So, to transmit a stream of control messages using the same controller, each new message is transmitted using 10 bits. Frequent rests are required between messages, otherwise the receiver will become confused. Using a safe 100% rest cycle of another 10 bits after each message gives a maximum sample rate of $31000/(10+10) \approx 1600Hz$. This is ample bandwidth for one parameter. If several controllers are required they will share this rate.[10] However, in real performance the separate channels rarely, if ever, require full bandwidth simultaneously[11]. Since MIDI messages are only transmitted when a state change occurs, unused bandwidth is not wasted, and the available bandwidth for any single channel remains high.

MIDI Interpretation

While increasing the bandwidth would certainly help, especially for multi channel performances, many controllers and synthesisers use MIDI inefficiently and unimaginatively. In the first instance the scope for control of commercial synthesizers has been quite narrow in the past, often involving bandwidth-unfriendly *system exclusive* commands, which can only control a few parameters. Happily the situation is changing with the latest generation of synthesizers and effects units, many of which have a healthy number of assignable MIDI control numbers. This has exposed another problem, namely the processing of control signals. To ensure no audio artifacts, control signals should be upsampled by band-limited resampling. The cost of such a process is unjustified in all but the most expensive studio equipment. On the other hand, many so-called ‘real-time controllable’ devices do not even employ a cheap upsampling algorithm such as linear interpolation. The resulting *zipper* distortion effectively rules out real-time control.

MIDI Mapping

Interchanging controllers and synthesisers using MIDI connections has led to the term *mapping*, which is the process of assigning one control parameter in the transmitter to one in the receiver. This is necessary to improve the compatability of different devices. Unfortunately, mapping has become strongly associated with the *creative* process of combining controllers with synthesizers. There are many instances in the academic and commercial literature where *flexible mapping* is implied to be a full specification for combining a controller with a synthesizer. However, from the pure engineering control viewpoint the only constraint on the relationship between controller and synthesizer is the obvious one that it should be /em causal. That is, the output cannot be affected by what happens in the future. Mapping is a *very* special case of this, implying that the output is dependent only on the inputs at that time rather than the input state at several previous times, or even continuum of times.

[10]There is also a reduction due to the status bytes required to change the running status.

[11]If the controls *are* worked at full the rate the resulting sound may not suffer from bandwidth loss because it appears too incoherent anyway.

The Computer Society

Electronic music today is virtually synonymous with the use of computers. The computer is a faceless tool. Its only important characteristics are speed, memory and reliability. The standardisation of languages across platforms means that perfect replicas of tools such as Csound exist. At first sight this would seem to be a sterile environment for development. What would happen if pianos were all exactly the same? On the other hand standardisation allows software to be rapidly circulated, promoting the development of new artistic societies and therefore new art.

The physical aspect of instruments are not so readily distributed. The MIDI keyboard is virtually the only widespread standard. Relatively few wind and guitar controllers exist. The ready accessibility of software, particularly *shareware*, creates an environment of impatience towards physical devices.

Computer Musical Instrument Design and Designers

Computer music instruments are not just physical instruments, they contain an abstract core of computer code embodied physically in silicon memory which is totally separated from the player and listener. There is a dislocation between the physical form of control and the 'guts'. The designer is faced with the difficult task of unifying the musical behaviour, determined by the guts, with the outward physical gestural control. The temptation is to concentrate on the guts and pay little attention to the physical side or its relationship with the guts.

The traditional instrument designer was by necessity a musician, a sculptor and a scientist. In today's commercial environment it is too easy for the tasks to be disintegrated. In acoustic instruments the physical controls merge with the physical sound generating structure in a way that is satisfying for the player and listener alike.

Artistic constraints

Computers present a particularly acute form of the old problem of artistic creation within an underconstrained system. The principal method by which constraints do appear, is the creation of software tools such as Csound, sequencers, or synthesizers. Each leads to a particular artistic trend or trends. These trends then influence the development of the tools, so creating a natural progression.

Computers Forever?

One may conjecture that some new non-electronic technology might revolutionise music. Certainly it would be unwise to let the computer be the last word on advances in musical instruments. However, computers have a strong abstract appeal which goes beyond their individual functionality. So far their widespread use in music has been brief but far reaching. We should expect our relationship with computers to mature as time progresses.

Research Recommendations

The outcome of this chapter's investigation has been a mixture of optimism and reservation. In many cases instrument design is diverging away from traditional acoustic design principles. This can be refreshing, but it has also empasized the value of the old principles. It is proposed that research be undertaken to combine the principles with contemporary technology according to the following guidelines. More detailed guidelines are given in reference to acoustic instruments at the end of Chapter 3.

- *Live Performance* The instruments reviewed fall mainly into two classes, those designed exclusively for live performance and a vague group for which the conditions of use are not clear or described fully. The lesson seems to be that instruments should be designed with a clear purpose as to their eventual role, which in this thesis is live performance.
- *Ergonomic Design* The physical interface must be given full consideration for its mechanical properties and sensing capabilities. Ergonomics is traditionally about making objects physically easy to handle. In the context of musical instruments an expanded definition is required, in the sense that the physical interface must ease the communication of *musical expression*. An interface may be simple and ergononmic in the traditional meaning, and yet would frustrate the communication of significant expression, regardless of any synthesis technique used. Although musical expression is subjective, it provides an important focus for the designer.
- *Sympathetic Synthesis* The synthesis process and interface should weld together seamlessly. Without this the musical integrity of the instrument is lost.
- *Open Architecture* Much current work is locked into a narrow set of models for the audio control process, for instance the *note paradigm*. The only essential limitation is causality.
- *Practicality* The hardware and software design must take account of basic requirements such as portability, cost, operating conditions. An elaborate design may be interesting but lead to no future development.

Chapter 3

Acoustic Instruments

This chapter examines the playing qualities of acoustic instruments which are attractive to players and listeners. Information presented here is based mainly on simple observations, and is part of the standard body of knowledge acquired by musicians. [Fle91] is a useful reference for technical details, and [SG84] provides a wealth of general information.

It is hoped that lessons can be learnt which may be applied in the design of new electronic instruments. One is respectfully aware that in most cases acoustic instruments are the product of many generations of musical and engineering evolution. Digital electronic instruments have no such heritage. Acoustic instruments are adaptations of mechanical devices which are subject to physical constraints. The digital instrument has no such constraints, as any sound can be produced, and controlled in any way. The negative side to this freedom is that lack of constraints makes it harder to justify an initial design approach from countless possibilities. In practice the designer will be constrained to a degree by the speed of real-time processes, the structure of languages used to build processes, and the hardware and sensors available for constructing control interfaces.

Classification

The full breadth of acoustic instruments is too great to represent here, so only a coarse cross-section is presented. The popular classification system of Curt Sachs divides instruments into four groups;

- *Idiophones* for which vibrations are generated directly in a solid body; cymbals, the triangle, xylophone, gongs and the harmonica.
- *Membranophones* in which vibrations originate in a stretched membrane; drums.
- *Chordophones* in which string vibration is the source; the violin family, the guitar, harp, harpsichord and pianoforte.
- *Aerophones* in which the passage of air into a tube results in vibration; trumpets, horns, single and double reed, flutes and recorders.

3.1 Idiophones

3.1.1 The Vibraphone

The vibraphone consists of bars of wood or metal tuned to resonate to chromatic frequencies when hit with padded hammers. Under each bar a metallic tube is tuned to resonate at the same frequency. Foot pedals are available to dampen the bars or pipes independently. The build up of resonance in the tubes is gradual, and determined by how frequently and hard the player hits the corresponding bars. Thus the player can choose to emphasize the bar sound or the tubes. The slowly decaying, smooth tone of the tubes contrasts well with the percussive characteristics of the bars.

The player uses one or two hammers in each hand, creating a very clear visual spectacle for the audience, especially in fast, energetic music.

3.1.2 The Blues Harp

This is a variation on the harmonica with a pentatonic rather than chromatic scale. The metal reeds are excited to resonate by direct breath flow. The case acts as a soundboard, not a resonator. The energizing of the reeds is a cumulative process. When breath is reduced the reed vibrations decay smoothly. The reeds are arranged alternately side by side so that forward and reverse breath cause adjacent notes. Drawing breath in sharply with the lips almost closed and sealed against the harp causes the pressure drop across the reed to increase, resulting in a lowering of pitch by up to a semi-tone, and a characteristic change of timbre. The bending of pitch in this way is essential to the blues idiom, as it allows the so-called *blues note* to be played. Blues harps are designed with ease of bending in mind.

If the player broadens the area of open mouth in contact, several adjacent reeds can be energized at once resulting in a chord. If the pitch is bent, different parts of the chord will bend at different rates due to pressure variations, creating interesting harmonic ‘tension’ effects. Another technique, common in blues playing, involves cupping the harp with the hand. The ‘cup’ acts like a resonant low pass filter. Closing the cup lowers the cut off frequency.

3.2 Membranophones

3.2.1 The Tabla

The Tabla is an important percussion instrument in classical Indian music. It consists of two drums connected together, played by hitting and rubbing with different parts the hands on different parts of the drum surfaces. The great variety of sounds over a wide frequency range allows a skilled player to sustain complex rhythms.

Treble Drum

The smaller drum produces the high, ‘snapping’ sounds. Hitting smartly with two or three fingers near the edge produces louder, well pitched sounds. Hitting near the centre and holding the fingers down gives a contrasting muted sound. The fingers of one hand can be tapped across the surface in sequence to give a rapid succession of hits.

Bass Drum

The larger drum has a special construction. The membrane consists of layers of attached concentric leather, such that the membrane is thicker and therefore heavier at its centre (This cannot be seen from the top). When the membrane is hit by the underside of the wrist, the player can modulate the pitch and volume of the ringing low note by rubbing his hand along a radius on the membrane, creating a characteristic bass sound. The effort required is clearly visible and complements the heavy penetrating bass sound.

3.3 Chordophones

3.3.1 The Pianoforte

Keyboard instruments are designed primarily for convenient polyphonic performance. The arrangement of black and white notes is based on the principals of Western harmony. It has also effected the development of harmonic thinking. Bach was the first to seriously apply equal-tempered tuning, as a compromise to allow free modulation on keyboards. Subsequently even tempered harmony has come to dominate all kinds of instrumental music. On the piano the tone is sufficiently rich that the harmonic impurity of the intervals does not distract the listener.

Key Control

The piano is separated from other keyboard instruments in its capacity for the dynamic control of each note. The process of hitting a key begins with the acceleration of the key downwards. During this period of acceleration, the player can feel the inertial resistance of the hammer through the key. The acceleration is too brief for the player to respond to the resistance and adjust significantly in that time. It will, however, be possible for the player to associate the resistance with the loudness of the note, and so help to calibrate the accurate playing of future notes. The very wide range of dynamic control possible on the piano is due to the ‘resistance’ being remembered in terms of muscular effort *and* a time over which that effort is applied. Thus a single parameter is controlled using two, and has resolution which is, roughly speaking, the multiple of resolutions of the component parameters.

The hammer does not return immediately to its resting position so rapid repetitions of the note can be made without repeating the effort of the initial strike, allowing *trills* and *shaking*. The rapid bounce of hammer as it leaves the string helps in this process by quickly bringing the hammer back to a point where it can be usefully projected towards the string again.

Pedal control

The sustain allows all the strings to vibrate freely. Notes that have been hit sustain when released, but also set other strings vibrating in sympathy according to their harmonic relationship, creating the hall-like resonance sometimes referred to as *breathing*.

3.3.2 The Violin

The violin lacks the polyphonic versatility of the piano, but excels in the variety and precision of articulation possible. The open ended nature of its articulation is precisely why it is so difficult to play well, as the player must constrain their movements within marrow margins. These observations apply to other members of the violin family.

Bowing

Many styles of bowing are possible. To achieve a smooth start the bow-string velocity must be increased continuously from zero. Louder sounds are made by bowing faster or nearer to the bridge. Bridge bowing deemphasizes the low harmonics, and a hard, biting attack can be produced by moving the bow a little as it comes into contact with the string. Bowing nearer the bridge requires more down-force to be applied with the bow on the string, and for this to be held between narrower limits. The extra effort required to do this increases the psychological tension in the performer and this is reflected in the way the sound is articulated by the player. Varying bowing velocity allows a continuous, controlled dynamic, which is reflected visually in the movement of the whole bowing arm.

Bowing two strings together allows intervals to be played. If one or more of the strings is open the sound is more confident. If both strings are fingered, or *doublestopped*, then even in good playing the intervals becomes noticeably unsteady.

Fingering

Fingers press strings against the fingerboard to change the pitch of each string. Sliding fingers produces a continuous change in pitch. Producing accurately pitched notes with a sharp attack is difficult. A small amount of pitch adjustment is necessary during the note onset, creating a subtle but characteristic aspect to the sound of violin playing. Fine pitch control also allows a player to modify equally-tempered pitches to produce perfect intervals, or *just intonation* with other players.

This is a subtle but important effect, which players execute instinctively.

Rocking fingers backwards and forwards whilst pressed firmly against the strings produces an expressive, controllable vibrato. This is so integrated in the violin family that it is difficult to separate it from the rest of the playing technique.

3.3.3 The Guitar

The guitar combines the intimate control of excitation, damping and pitch bend of each note with direct access to chromatic pitches and harmony. Some features are shared with the violin. Several strings are arranged side by side, the pitch of a string is raised by shortening the vibrating section with a finger and a resonant cavity is used. On the guitar the strings are shortened by pinning the string against a fret with a finger held just behind the fret.

Guitar Harmony

The fretted pitches are taken from the even-tempered scale, the same used for the piano. Sounding different combinations of fretted strings provides a kind of harmonic variety that is unique to the guitar, characterized by wide intervals. With standard guitar tuning, producing pitch clusters is physically impossible for the fretting hand. Changing the tuning can be used to give a different harmonic emphasis.

Picking styles

Many styles of guitar construction and playing have developed. The classical guitar is designed for keyboard-like playing in which the finger tips or nails of the right hand are used to *pick* different combinations of strings. The ‘Flamenco’ style incorporates rapid scales played by alternately picking with one or more picking fingers. *Strumming* all or some the strings with the finger tops, rhythmically back and forth provides a contrasting technique to precise finger picking.

Vibrato

Methods of vibrato vary depending on string type. For lighter gut or nylon strings, a delicate vibrato is produced by rocking the finger behind the fret. This technique is an essential part of classical and South American guitar playing.

For metal strings, which are under much greater tension, the string must be moved laterally to have a noticeable effect. The depth of vibrato relates directly to the physical effort required. Wide, fast vibrato is harder to sustain in a smooth fashion and so becomes even more associated with musical intensity.

Muting

Muting is the damping of strings with light hand contact so that they are not quite fretted, and is a very important part of left and right hand technique in many guitar styles. When strumming, muting on individual strings can be used to play a wider range of chords. Strumming over several muted strings produces a distinctive percussive sound. Coordinating the strumming and the muting hand gives scope for interesting rhythmic gestures.

Electric Guitar

The electric guitar is an acoustic hybrid, discussed here because of the similarities in playing technique with an acoustic guitar. The *semi acoustic* guitar, commonly used in jazz, is an electric guitar with a hollow body, giving it a more resonant sound. Electric guitars are most commonly played with a plectrum, a small flat piece of hard material used for plucking or strumming. This gives a harder attack than can be achieved with fingers. Many new techniques have been developed on the electric guitar as a result of the high gain that can be used without feedback problems. A slight tap with a finger which is then held on a fret produces a solid tone. Pulling a fretting finger away makes the string jump to the pitch given by the next fret held down. The energy of the *pull off* determines the volume of the following note. A combinations of taps and pull-offs allow very fast passages to be played, while retaining control of pitch bend and vibrato.

3.4 Aerophones

Finger Control

All wind instruments share some basic properties. The fingers are used to control pitch either by direct contact with tone holes in the instrument body, or indirectly via finger pads connected to levers and hole pads. Different pitches are selected by holding down different combinations of fingers. If the breath is already sustained legato notes can be executed with precise timing accuracy by moving fingers rapidly on or off finger holes. There is usually one point in the scale of pitches, the *register change* where the change in combination is particularly awkward. This point presents a challenge even to experienced players, and places extra melodic constraints.

Sometimes there are alternative fingerings for pitches which may be more convenient or have a more suitable timbre variation for a particular passage than the standard fingering. Other fingerings give slight variations of pitch. These are often produced when open holes are surrounded by closed holes, know as *forked fingerings*. Finally there are fingerings producing complex sounds such as *multiphonics* where separate pitches can be discerned.

So far in this description, each finger has been viewed as either on or off. A more accurate description is that the fingers and key pads move continuously between their extreme positions. In some wind the intermediate positions, know as *shadings*, can be used in a similar way to forked fingerings. In some cases it is practical to smoothly control the degree of shading whilst playing. The opening clarinet part to the *Rhapsody in Blue* by Gershwin is an example of this.

Mouth Control

Some pairs of notes are more awkward to change between because of the additional changes of embouchure or lateral movement of fingers required.

Mouth control consists of two components; *tonguing* and *embouchure*. Tonguing is the movement of the tongue to restrict air flow. It can be used to start and stop notes very precisely, and when used sparingly, to provide some phrasing between legato notes. Embouchure is the shape of the mouth and its position relative to the mouthpiece, determined by a set of continuously controlled facial muscles. As such it is often the most complex but expressive part of wind instrument control. Embouchure must often be carefully controlled at the onset of a note to produce a resonant or *centered* note. Conversely embouchure can be used to start special effects, especially when used with non-standard fingerings. Over the duration of the note embouchure can be used to continuously vary timbral characteristics of the note, and also vary the pitch over a small range. In some cases embouchure pitch adjustment is a necessary part of playing in tune. It also allows groups of players to play just intervals.

Onset Sound and Control

The onset sound is generally a very distinctive part of the overall sound in comparison with other types of instrument. Initially there is often a high component of noise in the sound which evolves into a steady pitch via a short period of harmonic fluctuation. Precision of control is critical at this point. The register and tone of the steady note are largely determined by the short period of initial control.

3.4.1 The Saxophone

The saxophone is unusual in several ways. It is a rare example of an established instrument that was originally developed by a single designer. It was initially heralded as a revolutionary orchestral and chamber instrument because the rich, sonorous tone. After coming to prominence in the Jazz era as a solo instrument, the modern saxophone is popular in jazz and contemporary music. Ironically, the reason for success in jazz is partly due to qualities of the saxophone which were definitely not intentional, and may well have been considered a hinderence. The player has a much wider control of the sound through breath, tongue and embouchure than is the case for other single-reed instruments.

Starting, Ending and Changing Notes

The upper teeth bite into the upper surface of the mouthpiece, the lower lip folds over the lower teeth and presses against the reed. The sides of the mouth are sealed to the mouthpiece. The tongue is extended slightly so that the tip pushes the underside of the reed. In one smooth gesture a positive pressure of air is built up in the mouth by constricting the lungs, and tip is withdrawn, introducing a pulse of air into the mouthpiece. The tongue tip is the principal means for precisely controlling the timing of musical events on the saxophone. When changing notes the tip is briefly extended to give a definite accent to the transition. The degree of accent can be reduced progressively towards a slur by extending the tongue less, so that it doesn't quite meet the mouthpiece. When a 'relaxed tongue' approach is used to start a note, a large amount of air noise is produced which is particularly audible in the tenor saxophone. The air noise can be extended into the later pitched part of the note, and is a common technique in jazz for providing an effective contrast to clean resonant notes. A note can be stopped emphatically by extending the tongue fully.

During a Note

All the notes require a minimum clean pulse of breath to start cleanly. This minimum is much greater for the lowest notes. Once a note has begun the breath can be reduced and the note is sustained. Very low breath produces pure tones. As the breath is increased the sound becomes increasingly rich. The tone can be made much more complex by singing while playing, or *growling*. The variation of tone across amplitude and scale, is distinctive for each register of saxophone. The soprano saxophone has a tonal transition in the upper register which is remarkably similar to the oboe's. Between the lower and upper registers, the tenor saxophone has voice-like tonal qualities, matched by a wide variation in tone with breath strength. In practice, there is wide variation between different instruments and mouthpieces, and how people play them. This is part of the appeal of the saxophone.

Changing the embouchure slightly whilst holding a note creates subtle 'moving' effects in the tone, similar to vocal formant transitions. Reducing lip pressure makes the pitch fall up to 1.5 tones, accompanied by diminishing tonal strength. Such large pitch variations can be used effectively in blues and jazz music. Slight oscillations of lip pressure produce a distinctive vibrato.

3.4.2 The Oboe

Control of the oboe is very subtle in comparison with the saxophone. The embouchure must be rigorously maintained within very narrow limits to achieve a satisfactory tone. The loudness is modulated with breath pressure, but the dynamic range is not very great. It is difficult to sustain good intonation for the quieter notes, especially in the lower register. Vibrato may be added by pulsing the air pressure with abdominal movement. The subtle application of vibrato has a dramatic effect on the clear, solid sound and the tonal variation across the scale is highly colourful and characteristic.

3.4.3 The Trumpet

Pitch Control

The player's lips form a vibrating source in contact with the mouthpiece. Increasing lip tension raises the natural lip frequency. To resonate the trumpet successfully and produce a good tone, the lips must resonate near a harmonic of the fundamental of the trumpet. The finger valves are used in combination to change the fundamental by combining different lengths of tubing. The complexity of tubing and valve work limit the accuracy of the fundamental pitches. The resulting set of scale pitches require adjustments of embouchure to be accurate.

One characteristic effect of this pitch system on phrasing is that there are two kinds of slur which sound different; one executed by finger movement and another only requiring lip tension change.

Embouchure

Muscle tension in the lips is adjusted to tune the resonant frequency of the lips to a harmonic of the fundamental resonance of the tube. Because the lips are the vibrating source, like a reed, the player has very intimate tonal control via the different muscles connecting to and within the lips. This puts a considerable burden on the player to keep their lips in good condition. Conversely, the mouthpiece is very simple and robust, requiring little attention.[1]

3.4.4 The Penny Whistle

The penny whistle is a very simple but surprisingly expressive instrument. A whistle mouthpiece connects to a metal tube drilled with finger holes. Vibrato is achieved by modulating breath pressure, and requires considerable skill. Higher pressure forces the pitch to the upper register where the sound becomes louder and brighter. The lower register, therefore, is necessarily quieter. The finger holes are large enough to make performance using *shading*[2] feasible. Static shading can used to generate a full chromatic scale. Dynamic shading applied at low speeds creates expressive pitch slides. At higher speeds it is effective in accenting notes in ornaments. A good example of this is found in traditional Irish music, where ornamentation often plays an important part.

3.5 The Acoustic

The acoustic used for playing an instrument is frequently an important consideration. There are many kinds of acoustic, some of which are suited to particular kinds of instrument or style of music.

[1]The single-reed player on the other hand has a less demanding embouchure but must ensure the mouthpiece assembly is in excellent condition. The right reeds must be selected, and the ligature adjusted according to the reed and the music.

[2]Partial coverage of holes

Essentially the acoustic can be seen as a transformation of an instrument's behaviour, to add spatial depth, change tonal characteristics and introduce a new *dynamic* relationship between control and sound output.

The violin provides an example of an instrument which is particularly sensitive to the acoustic, as the direct violin sound has a short decay and can be unpleasantly 'raw'. A slightly reverberant or *wet* acoustic will consolidate the tone of the instrument and smooth over rough edges by virtue of the *reverberant tail*, in an analogous manner to a liquid applied to a dry surface. A wetter acoustic encourages sparser playing. Conversely a dryer acoustic calls for well supported phrasing, for example legato transitions require the dynamics of notes to be controlled carefully throughout. In either case the acoustic becomes an extension of the instrument in the perception of the player and, even more so, of the listener. Instruments such as the piano have an inherently wetter sound than the violin, due to soundbox, soundboard and sympathetic string resonances. For these the acoustic is not as critical, although it remains important.

3.6 Summary

This chapter concludes by rounding up some of the important general qualities found in the range of instruments surveyed.

- Physically the instrument is a single object, or possibly two objects intimately associated, as in the case of the violin family. There is often a simplicity and economy of function that makes the instrument seem more object-like in abstract terms. The strings of the violin are a sound source but also provide points of direct control interaction. The body holds the strings taut but also provides a resonator and soundboard. It has been quite difficult to dissect instruments for linear description in this chapter because of the inter-dependence of their many qualities.

- The coordination of a variety of muscular actions is required in playing an instrument. The muscle actions are appropriate to the quality of control required. Fingers can be controlled through small distances well. A bowing arm requires more free movement to achieve a similar degree of control resolution. Fingers are particularly sensitive to small contact forces. The dynamics of the bowing arm are felt by the player in the tensions and inertial stresses in the upper arm, elbow joints and the fine stresses on each of the fingers holding the bow.

- Pitch control layouts tend to bias the instrument towards certain note sequences or chords. Guitar chords are often difficult or impossible to play on a piano and vice versa. Similarly, some kinds of articulation are only possible at certain pitches, for example lip slurs on the trumpet. This lack of uniformity, far from being an obstacle, gives instruments natural character, which often forms the basis for phrasing. This is why a successful phrase on one instrument can rarely be played on a different kind of instrument while accurately retaining the same musical sense.

- There are aspects of the control mechanisms which have interesting dynamic properties, eg the piano hammer mechanism, bow friction, breath through a reed, guitar string tension. Such dynamics often improve the communication of the instrument state by tactile feedback. On a psychological level they also provide another level of interest for the player, who will instinctively want to understand them.

- Many control systems combine one part containing the means for precisely timed musical events and another allowing accurate continuous information. For example on the violin the player can jump between strings or fingers to change note but can also control tone and dynamics continuously with the bow. On the clarinet fingers and tongue move to change note, and the player's embouchure and breath control tone and dynamics continuously. The piano does not have a direct means for continuous control, but it does allow musical events to be associated with precise dynamics. A stream of notes can be played whose dynamics approximate a continuous function.

- Instruments often display a hierarchal behaviour. In a coarse-grained view, sound qualities relate to the controls very simply, for instance hitting a piano key gives a specific pitch. In a fine-grained view, the relation is enriched by dynamical processes, some of which are connected with the mechanical dynamics of control mentioned above. The precise time of hammer impact, and hence note attack, given a control pulse depends on the previous history of hammer activity. The interaction of the hammer with the strings depends on the previous state of excitation of the strings. Resonances in free strings and the piano body depend on the cumulative effect of hammer strikes.

- Another attribute of the fine-grained viewpoint is that each quality of the sound is effected in varying degrees by several different controls. For instance the variation of pitch with breath strength *and* embouchure in aerophones. The mixing of controls puts emphasis on unique mixed sound gestures which help to define an instrument's character.

- The coarse-grain view is valuable because it helps beginners make a start on their instruments. Human intuition is well developed for learning fine-grain properties without supervision.

- Apart from the complexity introduced into the sound by the mechanical-dynamics of control and dynamic processing of control, sound variations occur due to dynamic processes which are effectively independent of control. An obvious example is the continuously changing sound of the piano once a key is hit. A similar hit produces a similar note but on close examination the sounds are subtly different. This effect is due to the complex nature of the hammer hit and the interactions in and between the strings. All acoustic instruments show this kind of sonic variation, as a general feature of natural complex physical objects.

- The fine-grained complexity and unpredictability of acoustic sound extends to the way in which it is radiated from the instrument. Our hearing system has evolved to extract spatial information about sound sources from their complex sound fields.

- There are instrument sounds which are often used to portray particular *emotional states*.[3] For instance smooth, quiet tones for tranquillity, loud, rich tones for boldness. A biting violin onset sound suggests determination, a wide unsteady vibrato, intensity and so on. All these effects are natural to the instrument and not contrived by the player. It is as if these are emotional states of the instrument itself. This can be explained by an evolutionary process in instrument design, with positive selection for instruments which can be used to better express emotional states.

- The use of acoustics shows the importance of spatialising performed sound, and also emphasises how subtle dynamic relationships between control and sound output can contribute strongly to the character of an instrument.

[3]It is difficult to talk of emotional states without complex discussion first. In this case an emotional state is seen as culturally related. This may partly account for the variation in musical instruments over time and around the world.

Chapter 4

Digital Signal Processing for Live Performance

This chapter reviews the ways in which digital signal processing has been applied in musical performance, and comments on how it might be applied in the future. The techniques naturally divide into sound synthesis and sound processing. Complete musical instruments can be viewed as signal processors converting control rate input to audio rate output. The recent interest in physical modeling synthesis has emphasized the pure signal processing outlook on musical instruments.

We begin by looking at synthesis techniques. The classification is due to Smith [Smi91]. This reference also contains an useful comparison of the detailed technical aspects amongst the techniques. Useful general references are [DJ85] and [RSA+95].

4.1 Abstract Algorithm Synthesis

This section includes the older forms of synthesis first implemented with discrete electronics. Although based on simple principles, they remain useful today, as part of more elaborate architectures.

4.1.1 Additive Synthesis

Fourier analysis shows that any signal can be decomposed with arbitrary precision[1] into a sum of sinusoids. In the earlier days of synthesis only enough sinusoids could be generated to make abstract sounds. Some of the first realistic additive sounds were from Risset's *Bell* instruments which combined around 10 carefully chosen sinusoids with non-harmonic frequency relationships. An important part of these instruments was an overall amplitude envelope. The same technique can be used with sounds with dominant harmonic components, such as wind instruments. Global vibrato

[1] Not withstanding the Gibbs phenomenon.

can be used to further bind the perception of the harmonics as a single sound. However, to generate convincing re-synthesis of complex and varying instrumental sounds individual control of the sinusoid pitches and amplitudes is required. Risset demonstrated this first with his additive trumpet instruments containing 5 or 6 sinusoids. The *phase vocoder* and *heterodyne analyzer* can be used to generate data for such sinusoids, but this raises questions about about how such an instrument can be controlled, whether from score or live performance. Until recently detailed real-time additive synthesis was not feasible. New hardware architectures based on stable recursive sine wave oscillators, like Smith's waveguide oscillator, [SC92], have resulted in commercial instruments such as a the Kawai K5000 introduced in 1997. Recursive oscillators have superior bandlimiting properties to wavetable oscillators, and require only a small amount of localized memory. The K5000 uses around 50 oscillators to generate a high quality realistic voice. It is likely that the popularity of additive live synthesis will increase in the future as machine speeds increase, because of the synthesis flexibility and the inherent low aliasing properties.

4.1.2 Waveshaping

Waveshaping is simply the application of a time invariant non-linear function to an audio signal. If the signal is periodic the waveshaped signal is necessarily also periodic, and so in this case waveshaping can be viewed as a form of harmonic distortion. Waveshaping can be extended by varying the transfer function over time, both in the short and long term. Simple uses for waveshaping include cheap filtering or frequency multiplying. Waveshaping is used commercially with notable success in the Kurzweil K2000, where it is used for such tasks as 'crunching' the attack portion of a cello sound by a controllable amount.

4.1.3 Amplitude Modulation (AM)

An audio signal, the *modulated* signal, is multiplied by another audio signal, the *modulating* signal. The effect of modulating with a sine wave, is to map each frequency component of the modulated signal to one displaced upwards by the modulating frequency and one displaced downwards by the same amount. By high pass filtering the result, the overall effect is a spectral shift upwards in frequency. This destroys harmonic relationships which may exist between frequency components. Multiplying two complex signals produces much greater spectral complexity. In pure synthesis AM has been used occasionally to colour fairly simple sounds, but its main use has been in the context of effects processing (see later). Although AM can be used to generate complex sounds, interest is limited because harmonic relationships are always destroyed.

4.1.4 Frequency Modulation (FM)

The frequency of a periodic audio signal is offset by another audio signal. Here *frequency* is instantaneous, defined as the rate of change of phase with time. Clearly we can extend the concept of frequency modulation to aperiodic signals, such as recorded or delayed signals, by defining phase as an increasing measure of position within the signal.

With just a sine wave modulating another sine wave, a complex set of new frequencies is produced. These can be further enriched by applying more stages of frequency modulation. By varying the harmonic relationships between the modulating frequencies, a surprisingly wide variety of timbres can be obtained. Several FM generators can be mixed and further processed with envelopes to generate a wide variety of musically useful waveforms. The net result is a complex sound generated from very simple basic waveforms. John Chowning made an extensive exploration of these techniques in the 60's and 70's, [Cho73], with particular application to acoustic instruments. Not only did he manage to recreate many sounds realistically, but he identified ways of controlling the sounds so they respond realistically. For example the *modulation depth* of an FM operator is the parameter which scales the modulating signal before addition to the modulated frequency. Increasing modulation depth creates a *brighter* sound. This effect can be used as the basis for simulating the brightening of natural sounds with increased energy input, for instance the brightening of a trumpet with increased blowing effort. The development of such simple control techniques contributed greatly to the success of the DX7 synthesizer.

Unlike AM, FM synthesis can produce a complex and yet harmonically rich signal. This is determined by the *modulation index* which is the ratio between the frequencies of the modulating and modulated signals.[2] For indices that are simple integer ratios, the resulting signal has a clear harmonic structure. Small continuous changes of the modulation index have large discontinuous effects on the sound, and so the index is not often used as a control parameter.

After its initial success, FM synthesis was soon eclipsed by new emerging synthesis techniques. Attempts have been made to find generalized methods for re-synthesizing natural sounds using FM synthesis, [TL96], but these make no consideration for the control of instrument sounds, and appear more as an academic exercise rather than an attempt to create something musically useful.

Today FM retains some importance, not for its imitative sounds but for abstract sounds which have unique recognizable character, and some of the control characteristics of the more imitative sounds. To designers, FM synthesis has an aesthetic which goes beyond the sound to the simplicity and economy of the algorithm.

Although FM is very successful at simulating a few natural sounds, there are subtle qualities in any FM sound which unite them, and undermine general attempts to recreate sounds. The real strength of FM is to create a class of abstract, complex sounds with these qualities.

[2]For this definition to make sense, the two signals *must* be periodic, or nearly periodic.

4.1.5 FM-related Modulation

Other forms of modulation, similar to FM, have been used. In *phase modulation*, used by Casio, the modulating signal offsets the phase of the modulated signal, rather than offsetting the phase rate of change. These methods have been devised out of necessity to overcome the FM patents. They have similar timbral properties, but programming may be less straight forward.

4.2 Processed-Recording Synthesis

The following methods of synthesis rely on recordings of real sound held in digital memory. They can be regarded as live tools for 'digital musique concrète', although the widespread usage in poor quality synthesizers doesn't really do justice to this term.

4.2.1 Wavetable Synthesis

A voice is created by playing back a sample, possibly with looping to sustain the sample. This forms the basis of many commercial synthesizers. The voice can be enriched by applying envelopes and low frequency oscillators for amplitude, frequency and filtering, and grouping voices together.

4.2.2 Group Synthesis

The aim is to synthesize a variety of sounds by carefully mixing a set of *basis* wavetables in different ways. Additive synthesis is a special form of group synthesis in which the basis waveforms are sines and cosines. The *Bradford Musical Instrument Simulator* is based on group synthesis, [CE88], and was designed to simulated organ sounds. This is especially appropriate as many organ sounds are summations of simpler organ sounds derived from single pipes. Methods have been devised to produce optimal basis sets which can cover a given set of target sounds, [HBH93]. Little attention seems to have been paid to the manipulation of these waveforms for performance, or the generation of basis sets which can be controlled to re-synthesize a variety of recorded performance variations of one sound.

Commercially, group synthesis is present in a weaker form in many synthesizers. For instance, the Roland *LA* system allows the mixing of four separate wavetables for each voice. *Vector synthesis* is the direct performance control of group mixing. The emphasis is on letting the sound programmer combine and control sounds in whatever form they like. This is ultimately a more musically rewarding approach than the group-basis approach outlined above, at least for real-time performance situations.

4.2.3 Granular Synthesis

Many small samples of sound, *grains*, are taken from a larger sound sample. Window envelopes are applied to smooth the ends. The grains can be remixed to produce a range of sound effects. In the simplest case the original sample can be reconstituted. Moving the grain positions closer together or further away uniformly, makes the total length smaller or greater without changing the pitch. More complicated mappings move grains backwards and forwards. A great number of parameters can be devised for controlling the mapping process. The program *Granular*, [Beh96], is one example of the growing number of real-time granular processing tools. These operate very much in the interactive composition mode, but it is possible that they could be adapted to live performance.

4.3 Spectral Synthesis

The following methods of musical synthesis are based on alternate representations of audio signals, loosely referred to as *spectral*. The audio signal is synthesized in the audio representation then converted to the time domain. *Analysis and re-synthesis* is the term applied to the special case where the synthesis procedure uses data taken from the analysis of sound, usually natural.

The representations may be *homomorphic* in which case they can be converted to and from the time domain with any change, or alternatively the new representation may be *lossy*, and information is lost. Most audio compression techniques fall into the latter category.

4.3.1 Linear Predictive Coding

LPC has a convenient structure for analysis and re-synthesis of sound. It was originally considered as a technique for compressing speech. The analysis procedure divides the speech into short frames. For each of these a set of optimal filter coefficients is found up to the desired order, together with excitation parameters representing the voiced fundamental and noise components of the speech. The speech is reconstructed, approximately, simply by filtering the reconstructed excitation signal. The order of the filter for reconstruction of speech at 15kHz should be about 19. The excitation parameters can be compressed using frequency and amplitude traces.

Speech responds well to this form of compression, because the separation into excitation and filter bank corresponds naturally to the vocal cords being filtered by the vocal cavity.

For musical purposes the principles of LPC can be abused a little to produce some intriguing results. Filters from an LPC analysis can be applied to sources other than the LPC excitation, to create *cross-synthesized sounds*. This is essentially the operation of the *vocoder* used occasionally in popular music as means for imposing vocal formants on other instruments.

The *naturalness* of LPC re-synthesis is improved if the pulse is replaced with a ‘fuzzy pulse’ consisting of a cluster of pulses. LPC analysis is expensive for real-time applications, although synthesis is much less so. Future applications might make use of pre analyzed data during performance.

4.3.2 FFT and The Phase Vocoder

The computational load of additive synthesis with many components can be reduced by using the *inverse fast fourier transform*, which has the same complexity as the normal FFT. Complexity O(n) per sample, using additive synthesis, is reduced to O(log n) per sample, where n is the number of frequency components.

The Phase Vocoder *is* essentially the FFT algorithm, but the parameters are more carefully interpreted. On the surface the FFT has frequency resolution limited by the size of the frequency bins. However, if a bin contains a single sharp frequency component then its position within the bin can be determined from the phase component for that bin. Since many signals of interest, such as harmonic sounds, have widely spread spectral peaks, they can be analyzed using the phase vocoder.

Musical applications of the phase-vocoder have traditionally been off-line transformations of sounds or cross-synthesis. Although the FFT is efficient it is still a difficult real-time task if high quality is to be retained. There is also the inherent latency problem in that the input signal must be binned. If the bin is too short the frequency resolution becomes too coarse, and frequency components become confused within the bins.

Wavelets

The wavelet representation is designed to give a viewpoint in between the frequency and time domain representations which is more natural to our perception of sound. The basis waveform set or *wavelets* consists of windowed wave packets of different length and frequency. This gives a two dimensional representation with fast response to impulse like changes at one side and sensitive response to sustained oscillations at the other.

No fast techniques have been found for wavelet decomposition, as yet. However decomposition for just a few wavelets could still provide useful analysis information in a real-time situation. An important advantage over the phase vocoder is that fast changing events need not be hampered by a having to perform a long overall decomposition.

4.4 FOF and VOSIM

These techniques are forms of direct time-domain synthesis which do not fit into the other classifications easily. They are both designed for synthesizing vocal formants, by direct synthesis rather than the filtering of a source. FOF achieves this by controlling the envelopes of a harmonic series of oscillators, which are pulsed to simulate the vocal pulse, [Rod79]. VOSIM uses a single waveform for each formant, one period of which consists of several pulses and a flat section, [KWT78]. By controlling the shape and number of pulses of the waveform, it is possible to change the formant characteristics. VOSIM is particularly well suited to live applications. There would seem to be many possibilities for developing it for general purpose live synthesis.

4.5 Physical Modeling

Physical models are, of course, the basis of physical science. Over the last 10 years interest has increased in implementing physical models of acoustic instruments on computers, spurred on by the continuing rapid increases in speed and memory capacity. The three approaches given here represent three ways of implementing the same basic mathematics contained in the physical model, although practical differences result.

Cellular models are based on low level physical models. The other two are based on higher level physical models which are constructed from the same low level physical models. The concept of a resonator follows from the wave equation which in turn is derived from the low level properties of the materials.

From the view point of real-time synthesis, physical modeling synthesis offers a link with traditional expectations of musical response to control, while at the same time providing exciting possibilities for new sounds.

4.5.1 Cellular Models

The basic local laws of physics applying to each point of the instrument and the air are approximated by calculating at a finite number of evenly distributed points, and times. The advantage of this method is that interactions with the instrument by the performer can be modeled without confusion by applying local laws. Unfortunately, cellular models are currently intractable in real-time with reasonable digital resources. Examples of cellular models are given by the systems CORDIS-ANIMA, [CLF93], and TAO, [Pea96].

4.5.2 Waveguide Models

Waveguide models were developed originally for instruments based on 1-dimensional waveguides, such as wind instruments and string instruments. The general solution of the 1-dimensional wave equation can be represented as two traveling waves moving in opposite directions. Traveling waves are very cheaply implemented on computer with delay lines, and so waveguide models are highly suited to real-time synthesis. **Figure 4.1** provides a simple overview of a waveguide model of a violin or a clarinet.

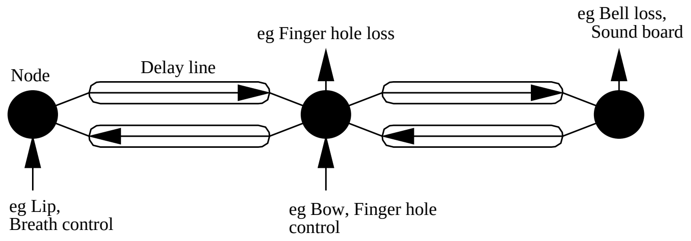


Figure 4.1: Simplified overview of a Waveguide Instrument

The whole instrument model consists of waveguide sections connected by nodes which mark interruptions in the waveguide. The violin bridge, bow point, nut, and clarinet mouthpiece, tone holes and bell correspond to different nodes. Any dampening or dispersion which occurs along the waveguide sections can be accumulated at the nodes. The modeling of physical processes at the nodes presents the major problem. The acoustic literature is rich in such models, but compromise is required in finding models that are cheap enough for real-time instruments, and accurate enough to be musically useful. Particularly sensitive is the interaction of control signals such as breath pressure or bow motion with the instrument.

2D and 3D extensions

Waveguides can be extended to model 2-dimensional and 3-dimensional objects by connecting waveguide sections in lattices. This is a cellular approach, but offers much reduced computation over exhaustive cellular models.

4.5.3 Modal Models

Modal models are even more abstracted than waveguides. The resonant frequency components of a particular physical configuration of an instrument are updated dynamically according to nonlinear functions controlling their interaction. Although much more efficient than cellular models, modal models are less efficient than waveguide models in many cases, and also more difficult to construct and control from the physical data of the instrument. *MOSAIC: a framework for modal synthesis* , [MA93], describes a non real-time Modal synthesis system which has been in use at IRCAM, Paris.

4.6 Effects Processing

Effects processors transform an audio input signal into an audio output signal in real-time. They are often used in a live context for altering the amplified sound from an acoustic or electronic instrument while it is played. At the simplest level this can be to restore a natural room acoustic that may be lost in amplification. Effects can be used to radically change the sound and playing *feel* of an instrument. The application of effects need not be time invariant. Other processes can influence the effects, for example from a score or another player.

4.6.1 Delay

Delay has a simple but powerful musical effect. In the simplest case, the signal is delayed by a fixed time and mixed with the original. Feedback can be applied to create diminishing echoes. Mixing multiple delays allows more complex echoes. Rhythmic instrument performance interacts very strongly with delay processing, and is used extensively in popular music. Variable delays can be used for Doppler effects discussed under *Spatialisation*, and also the chorus effects below.

Delay may also be used in an *event processing* context. Samples of performed sound are played back according to sequenced or performed events. Technically, delay is just a linear filter, but with the unique property that it does not affect timbre.

A useful quality of delay is that it does not change the timbre of the signal.

Delay is implemented very simply by writing samples direct to digital memory in a cyclic queue, making elaborate manipulations very easy.

4.6.2 Early Reflections and Reverberation

Early reflections are multiple short delays designed to simulate the early reflections of a real room. Simulations vary from crude multi-tap delays to precise spatialisation discussed later. Similarly reverberation simulates the latter part of a large room response, in which the reflections have blended to form a continuum. The common *Schroeder* reverberation networks consist of comb filters in parallel followed by allpass filters in series.

4.6.3 Chorus

This effect attempts to simulate a number of instruments or singers all playing the same notes. The slight variations of pitch, static and dynamic, create a *richer* sound. The simplest form of chorus consists of a short time varying delay. The delay time is varied with a simple waveform. The crests and troughs of the waveform can produce an unnatural, repetitive transition in the sound, which is less pronounced in more complex chorus effects.

4.6.4 Flange

The flange is a chorus with feedback. The result is complex, and includes a sharp filter sweep which follows the chorus waveform. Flanging is very strongly associated with electric guitar music, particularly rhythmic playing, in which the obvious cycle of harmonics is broken up by the amplitude variations of the guitar signal.

4.6.5 Pitch-shifting

Pitch-shifting shifts pitch by a uniform amount across the spectrum. In contrast to frequency shifting accomplished with amplitude modulation, pitch-shifting maintains harmonic relationships. Changing the playback rate of a recording shifts the pitch, but also changes the duration. Pitch-shifting is achieved in most commercial equipment using granular synthesis; windowing successive grains of input sound. The grains are stretched or compressed and remixed with sufficient overlap to ensure a smooth signal. Higher quality pitch-shifting can be achieved with phase-vocoding this is not practical for real-time purposes at present.

The *Harmonizer* is a pitch-shifting effect in which the fundamental pitch of the input signal is found, by a pitch tracker, and used to determine a number of harmonizing pitch-shifts. These are applied on separate pitch-shifting channels and mixed to produce the combined harmony. The harmonies can be *static*, where the pitch-shifts are fixed, or *intelligent* in which case the shifts change according to the detected pitch.[3]

4.6.6 Vocoding

Vocoding is an old technique originally developed by the Bell laboratories to try and compress bandwidth for intelligible speech. The formant structure of an input signal is extracted and is applied to a second signal with filters. A range of distinctive effects is possible. Pop musicians occasionally use a vocoder to modulate an instrument sound with a voice sound, and so make it ‘speak’.

4.6.7 Envelope following

Like vocoding, two input signals are required. The amplitude envelope of the modulating signal multiplies the modulated signal. This is a simple technique which is effective if the inputs are carefully chosen.

4.6.8 Gating

Gating is a more extreme version of envelope control. Either the modulated signal is passed or blocked. It is primarily used in studios to limit noise. When the signal level, and hence the signal to noise ratio, falls below a low threshold the noise gate blocks causing silent output.

A very common sound in ’80s popular music was the gated-reverberated-drum. The reverberation gives the drum a sense of space, and the gate prevents the reverberation tail from obscuring the following beats.

A more unusual application, sometimes found in popular music, is to use the sound from one instru-

[3]For instance, diatonic chords can be produced from different root notes in a scale.

ment to gate another instrument. Gating an electric guitar with a drum kit gives the impression of an *extended* guitar.

4.6.9 Compression

Compression is a form of automatic gain control. The gain is determined by a nonlinear amplifier acting on the input. In normal operation the gain is reduced quickly when the input level rises quickly, but rises more slowly as the signal level decreases.

On a practical level compression can be used to prevent overload in a recording situation while maintaining best use of the available resolution most of the time.

As a musical effect compression is commonly used to bring out musical detail. This can be applied with specific settings for different voices within a mix, for example the human voice and acoustic bass. Extra compression is frequently applied to the overall mix to help the different component sounds stand out, and improve the clarity on poor quality reproduction systems such as the radio. The unfortunate tendency has been for different programs to compete for attention by applying ever higher degrees of compression. Too much compression is tiring to listen to and is best used sparingly in contrast with uncompressed or differently compressed material.

From the electroacoustic viewpoint there is potential for dynamic control of compression parameters. Compression is a simple process and can be efficiently implemented digitally.

4.6.10 Equalization

Equalization is a general term for static audio filtering, either for technical or musically creative reasons. Parametric equalizers on mixing desks can be used to alter components within a mix. They are compositional tools.

Similar kinds of filtering can be used dynamically in a performance situation. The oldest of these is the *wha pedal* used with electric guitars. This distinctive effect is a low pass filter with resonance at cutoff and controllable cutoff.

4.6.11 Exciters

Exciters generate new harmonics or sub harmonics of frequencies present in the input. They can be used to ‘liven up’ a final recording, or to greatly increase the bass response of a signal. This is sometimes justified as a compensation for recorded frequency response.

4.6.12 AM and FM

AM can be used as an effect, rather than a synthesis technique, by deriving one or both of the two multiplied signals from signal sources. In this context the process is more commonly referred to as *ring-modulation*, and was often used in early electronic music. Small amounts of ring-modulation produce a distinctive distortion of harmonic signals.

In the synthesis section it was explained that FM can be extended to the modulation of aperiodic waveforms, which can derived from an input signal. The result is more complex than ring-modulation, but worthy of investigation.

4.6.13 Granulation

Granulation has already been mentioned as a synthesis method. It is also possible to use granulation as a real-time effect provided audio input and output are sustained at the chosen sampling rates. It is inherent in the granulator effect that the output will reference earlier times in the input. To minimize the control feedback delay or *latency*, it is important to maintain constant reference to the current input. In the non real-time case there are no such restrictive considerations.

4.7 Spatialisation

Spatialisation is the transformation or synthesis of sound to create the impression of sound generated from a natural distributed physical system occupying a volume of space. Since the human senses and thought processes are highly optimized for natural systems, accurate simulation of them is of use to both practical and musical endeavours.[4]

Spatialisation can be achieved by an exhaustive physical model of the natural system, but this approach is rarely taken because of the computational cost and the difficulty of controlling the system. Instead, the designer can focus an elements of the model which are of most importance to the listener, psychoacoustically and psychologically. These psychcological parameters relate more directly to musical parameters than the raw physical variables.

Monophonic reproduction can be used to convey indirect spatial cues such as room acoustics, but it cannot provide cues for several directions at once. There are two methods of multichannel reproduction currently in operation:

- Several speakers are statically positioned facing the listener.
- The listener wears headphones.

Headphones are theoretically ideal since the sound in the each ear can be controlled exactly. How-

[4]The growing field of *auditory visualization* explores how sound can be used to convey spatial and abstract information.

ever, for live performance it is impractical, to equip every listener with headphones.[5] Further limitations will be discussed later. The speaker array may not have the same degree of control over the sound in the listener's ears, but it is a lot more practical for an audience.

4.7.1 Live Electronic Diffusion

Musicians working with electronics were quick to realize the potential of loudspeakers for generating sound with spatialisation as a significant compositional component. Early methods of control depended on specially constructed fading devices, such as those of Jacques Poullin. The process of performing the spatialisation is termed *diffusion*. More recently, electroacoustic groups often use commercial mixing desks. A common configuration is to split each channel of a stereo source into several channels and feed these to the line inputs. Post-fader channel sends are then fed to the speakers. Each speaker can only deliver one of the stereo channels at a time /footnoteEach speaker feed could be mix of left and right using the pan control, but this is rarely used., but with a careful speaker setup and stereo arrangement the configuration is musically useful.

The acoustics of the concert space are usually very important in deciding on the speaker layout and even the material to be performed. Speakers further away are more obscured by the natural reverberant sound of the surrounding space, while close speakers naturally sound clearer and nearer. The variety in room acoustics and the availability of speaker equipment make it difficult to reproduce concerts accurately in different locations, so the issue of interpretation becomes important. On the positive side different situations provide the opportunity to bring new life to old works.

4.7.2 Audio Engineering

Early on, the engineering community became interested in faithfully reproducing the spatial qualities of sound. This is a very different approach from the aesthetically motivated methods of diffusion just discussed. Some of the engineering techniques have crossed into live electronics, but with limited success due to the engineering focus on recording and reproduction, rather than creative tools for studio work and live performance.

4.7.3 Huygens Principal

Huygen's principal is a mathematical property of the wave equation, and so is closely applicable to sound. It states that the wave field inside a closed surface can be recreated by integrating omnidirectional radiators across the surface. The property is useful because the integration can be approximated to a finite number of radiators without ruining the interior field. This was first demonstrated in the 1920's with a curtain of microphones connected to a distant curtain of speakers.

[5]Even if headphones could be made very light and non-invasive, there is a more delicate question of whether headphones are too *unnatural*. Traditional concerts are the shared experience of sound from common sources.

Prof Diemer de Vries of Delft University has recently experimented with 160 speakers placed in a circle about the audience, and driven from a signal processor. To simulate a single source each speaker is fed with a delayed and gain adjusted copy of the source. The audience is free to move and reorient within the circle without disturbing the consistency of the image.

The system has been installed in a cinema. The system has not been designed with electronic music performance in mind, so no tools are yet available for this. While it is just feasible to apply to 2 dimensions, a 3 dimensional array would pose many more problems.

4.7.4 Stereo

stereo is a recording and playback technique, based on psycho-acoustic principals.[6] The recording is made with a pair of coincident microphones angled to the left and right. Ideal listening is symmetrically in front of two speakers with an angle of 60 degrees from left speaker to right. Stereo produces a convincing image for a listener close to the optimal spot and facing the speakers. If the listener rotates sideways the image eventually becomes confused. For a large audience the benefits of stereo will be limited to only a few people.

Today *stereo* is often used as a general term for two channel sound reproduction. Pure stereo recording techniques have become rare. Multiple distributed microphone techniques enable producers to experiment quickly with the balance and editing of a recording, but at the expense of loosing the convincing spatial qualities of natural stereo.

Transaural stereo is another 2-speaker system, but operates on very different principles, and will discussed later in relation to binaural reproduction.

4.7.5 Quadraphonics

Technical attempts to develop stereo techniques for more speakers, which surround the listener, resulted in the *quadraphonic* systems of the seventies. These worked by the *pair-wise intensity panning* of sound, meaning each sound was located using only two speakers at a time. This system was not based on any serious psycho-acoustical research, and suffered from uneven image stability.[7] These techniques are similar to the spatialisation methods used by Chowning in electronic performance. Pair-wise panning has been used successfully in other applications. *Level Control Systems* produce diffusion systems for theatres in which sounds are panned across a 3-dimensional speaker array using 3 speakers at a time.

[6]Developed in the 1930's by Alan Blumlein.

[7]This was later explained by Michael Gerzon, [Ger92], where he showed the central region of speakers subtending 90 degrees to the listener is unstable.

4.7.6 Dolby Surround

Dolby surround is the multi speaker standard that has been widely used in cinemas. It is a specialized system which extends on pure stereo by allowing extreme left and right signals to be panned onto speakers which are mounted along the side walls. It is downwardly compatible with stereo, enabling stereo only cinemas to directly use a Dolby surround track.

More recently Dolby released the *Dolby Digital* (5.1) standard. This is actually a multichannel audio compression scheme for spatial sound rather than a method of encoding and decoding spatial sound. 5.1 stands for 5 full bandwidth channels and one low frequency, low bandwidth channel. 5.1 is mainly aimed at the growing home cinema and digital TV market. The exact speaker placements are not specified but the overall arrangement is one forward speaker, one left, one right, back-left, back-right and one rear bass speaker. Although theoretically uninteresting from the surround viewpoint, 5.1 is a promising development because it leaves the spatialising technique open, as the channels are all independent.

4.7.7 Ambisonics

Ambisonics is an integrated system for the production and projection of spatial audio using speaker arrays. The original ideas appeared independently in the early 70's, and were developed primarily by Michael Gerzon. Like the original stereo format, Ambisonics is based on solid psychoacoustic foundations. Its many elegant theoretical properties are of direct practical utility.

Design Criteria

The fundamental problem of Ambisonics is how to feed a speaker array surrounding a central listener such that the perceived absolute direction of the sound source appears unchanged by head movement. This is sometimes termed *sound holography*, but note that we do not additionally require consistent perception of sound direction when the listener changes position. Experiments leading into the 70's had showed that two psychoacoustic cues, the so called *Makita* and *energy* or *intensity* cues, were largely responsible for the perception of direction, [Lea59, Mak62, Ber73]. Each cue alone can contribute to direction perception by an amount dependent on the soundfield in question. If the cues agree the perception of direction increases dramatically. Gerzon proved that for a diammetrically symmetrical array the requirement for consistency was simply that the sums of signals feeding each opposed pair of speakers should all be the same. This is the first theorem in *A General Metatheory of Auditory Localization*, [Ger92], based on work originally completed in 1976. The remainder of this important paper goes on to develop the theory of Ambisonics as the first stage of series of increasingly refined spatialisation systems.

Encoding

Stereo is played directly onto two speakers from the recording. For a many-speaker surround system it is practical necessity to find an *encoded* recording format which uses just a few channels. This can then be *decoded* to generate all the channels required to feed the speakers. It turns out a very natural encoded format, known as *B-format*, can be used which decodes easily onto a symmetric Ambisonic array. The decoding process ensures that signals representing directed sound will give the maximum possible strength of directional perception.

The B-format signal consists of the components of the first order spherical approximation of a sound-field about a point. The *W* component is the zero order component. This is the signal that would be measured by an ideal omnidirectional microphone, with a gain of $1/\sqrt{2}$.[8] The *X,Y and Z* components are the first order components corresponding to the signals from three cosine, 'figure of eight' response microphones, oriented orthogonally to one another. One lobe of the eight is a positive response, the other is negative. **Figure 4.2** shows a representation of the responses and how they relate spatially.

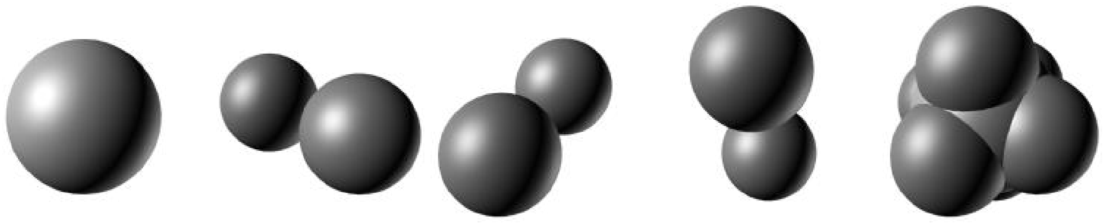


Figure 4.2: B-format W, X, Y, Z responses separately and together

It follows immediately from the 1st order cosine functions that the components of a single source are related by:

$$2W^2 = X^2 + Y^2 + Z^2 \tag{4.1}$$

Theoretically it is possible to extend B-format to include 2nd order harmonics/footnoteThese are simply more complex angular distribution functions, with more lobes. In three dimensions there are 5., and accordingly modify the decoding procedures. This would allow independent signals to assume more complex directional profiles, which appeal to more subtle psychoacoustic mechanisms than the Makita and Energy profiles. The general consensus amongst the audio community appears to be that the improvement in quality would not justify the additional channels required.

UHJ

If we restrict the soundfield to 2 dimensions, the B-format signal has only three components. With careful use of phase encoding, these three signals can be compressed satisfactorily onto a 2-channel

[8]This response is adjusted so that the dynamic range of the W,X,Y and Z signals is closer for most natural soundfields. This ensures the best possible signal to noise ratio when the signal is eventually decoded.

format called *UHJ*. This can be broadcast or recorded on standard stereo media. In Finland this system has a commercial footing in several radio stations. Played on a normal stereo system UHJ behaves well.

Decoding

Using speakers arranged in opposite pairs, according to the first decoder theorem, is very straightforward. A simple matrix is applied to the B-format signal to generate the speaker feeds. To compensate for bass proximity effects shelf filters can be applied to the encoded signal.

Although the theory was originally applied to a central listener, Gerzon has since shown theoretically that the holographic image quality is high over a wide area.[9] Such a wide sweet spot makes Ambisonics suitable for audiences such as those found at electroacoustic concerts. The beneficial effect of shelf filtering is much reduced for off centre listening, and can even worsen the image, and so they are generally not employed in concert situations.

For 3-dimensional arrays, strength of directional perception can be traded between the vertical and the horizontal. For horizontal only arrays the strength of horizontal perception is always greater than that for a 3-dimensional array.

Irregular arrays containing speakers which are not pair members, can also be used to satisfy the Ambisonic criteria closely but the decoding procedure is more elaborate and sensitive to error. The solution was first presented in the context of speaker arrays for surround sound TV, [GB92].

Recording

The simplicity of the B-format signal lends it to direct generation from a microphone. Gerzon proposed the following design, which is called the *soundfield microphone* and has been produced commercially by several manufacturers.

Accurate omnidirectional and cosine microphones do not exist, so the B-format components are formed by the matrix summation of 4 cardioid microphones, whose response is closely approximated by a sum of omnidirectional and cosine responses. The microphones capsules are arranged tetrahedrally and mounted in a single microphone. The matrix hardware can conveniently incorporate other effects such as stereo output width control.

When recorded B-format is played back over an Ambisonic array, the power and economy of Ambisonic methodology becomes clear. The directional complexity of natural soundfields is convincingly portrayed, and the sound does not appear biased in any way in the direction of the speakers. In ideal listening circumstances with very little room acoustic, the perception of the image can be strong enough to provoke the same involuntary responses as might be provoked by the original. [10]

[9]This does *not* however mean the image transforms correctly as the listener moves.

[10]Clearly, interesting possibilities exist for working with recorded Ambisonic sound to produce tape music. Using only 3

B-format Processing

The group of linear operations which conserve the relation 4.1 are of special interest, since they maintain the correct localization of all the components in natural sound while transforming their directions. A subgroup is the well known group of *3-dimensional rotations*. Rotation is of immediate practical importance because it can be used to simulate rotation of the head.

The complete group consists of the rotations extended by the *dominance* transformation. Dominance has the effect of moving the directions towards or away from a special direction, *the direction of the dominance*. If the *dominance factor* is zero, then it the identity. At maximum dominance the directions are all changed to the direction of dominance. Associated with the change of directions is a gain factor. For initial directions further from the dominance direction, the gain is relatively less than for closer directions. At maximum dominance opposed directions are eliminated completely by zero gain. Dominance has been used as a kind of zooming tool to give gain emphasis locally while maintaining B-format integrity.

B-format Synthesis

Before continuing we must carefully examine what it means to synthesize a B-format signal. Theoretically, a soundfield can be constructed for any given B-format signal. Natural soundfields are more constrained, because they consist mainly of discrete sound sources radiating uncorrelated signals. A B-format signal for a single source direction is found very simply. If the source direction makes angles $\theta_x, \theta_y, \theta_z$ with the B-format axes then the 1st order responses are, by definition, $\cos\theta_x, \cos\theta_y, \cos\theta_z$. Coincidentally, these are also the components of a normal vector in the direction of the source, because the axes are orthogonal. The W response is a constant $1/\sqrt{2}$. The actual B-format signal is the product of the source signal with the response vector, $(1/\sqrt{2}, \cos\theta_x, \cos\theta_y, \cos\theta_z)$. By adding together the B-format signals of separate sources a more complex audio scene is created.

Analog devices have been constructed for synthesizing single sources, [Ger75a]. These were originally intended for a studio production environment, but have also found use in live electroacoustic music.[11] Using joysticks, two sources can be directed at once.

The analog controllers perform simple functions, and can only control 1 or 2 sources each. More complex synthesis could be achieved using digital processing. This would open the way for simulating room acoustics and radiators more complex than point sources. By adding more complexity to the soundfield, more psychological cues become available to the listener and the overall experience more convincing on a direct perceptual level. Digital technology would also provide much more flexibility for linking the control interface with the audio processing functions.

or 4 channels, (2 using UHJ), 'pre-diffused' music whith rich spatial content can be produced, ready for reproduction on a wide variety of speaker arrays.

[11] York University Music Department has been applying Ambisonic technology in this way since the early 80s.

4.7.8 Binaural Reproduction

The use of headphones with normal stereo recordings creates unnatural sound images ‘in the head’. *Binaural* reproduction aims to create the illusion of a natural soundfield by presenting on headphones, the same signals that the listener would actually receive in the ear centres when listening normally. Binaural recordings can be made with dummy heads containing microphones, but these are of limited use because in reproduction the soundfield cannot be made to rotate relative to the listener, to track head movement.

If a binaural signal is synthesized then head rotation can be easily accommodated by rotating the image prior to binaural conversion. The binaural signals are found by applying the *head related transfer functions, HRTFs* to the incoming sound. HRTFs model the filtering effect of the head, including the ear shape, and so vary from one person to another. In listening to a performance the listener must ideally apply their own approximated HRTF data. The second major inconvenience is that the head motion must be tracked if the correct rotations are to be applied. This is acceptable for single users, but not for a traditional concert audience. One area where binaural technology has promise is on the Internet, as a means for connecting groups of people together. There already exist projects for live music making on the Internet.[12] Binaural listening would provide a common sound space for the participants. Using speaker arrays is a more costly alternative, less practical from the viewpoint of noise pollution and possible less appropriate psychologically given that a communion on the internet is more abstract than physical.[13]

4.7.9 Transaural Stereo

Transaural stereo combines binaural techniques with *cross talk cancellation*. The listener faces two speakers as for normal stereo. The sound received in each ear is the sum of sound from each speaker filtered by the head. For instance the head obscures the signal from the right speaker to the left ear more than the signal from the left speaker. So the transfer from speakers to binaural signals is a 2x2 filter matrix. The matrix is invertible and yields the speaker signals as a function of binaural signals. The binaural signals can be generated by the recording or synthesis processes discussed above. No examples of head-tracking have been found, although this would be possible. Without tracking the image quality deteriorates rapidly as the head rotates and moves away from the ‘sweet spot’.

Transaural stereo is designed for a single listener. Multiple listeners cannot all share the same sweet spot, and they will also obscure each other. There is therefore little use for transaural techniques in the concert hall.

[12]One example can be found at *http://www.sf.resrocket.com/*, 16 Dec 1998.

[13]The Australian company *Lake DSP* has taken the lead in personal binaural systems with the *Dolby Headphone*. This uses B-format to approximate the soundfield, and so the HRTF filters are each reduced to four filter channels. Lake claims to produce binaural sound which is robust to changes of listener. This is most likely due to the use of recorded B-format impulse responses of room acoustics. These create strong additional localization cues which help compensate for HRTF variation. Using B-format also allows natural sound environments to be presented binaurally, with rotation added to track head movement. This is impossible with a simple dummy head recoding. *www.lakedsp.com 16 Dec 1998*

4.7.10 Hypersonic Sound

HSS is a new surround system based on the psychoacoustic phenomena of *difference tones*. When two sine waves of differing frequency are heard together, a quieter tone is experienced with frequency given by the frequency difference. If the sine waves have ultrasonic frequency then the difference tone can be heard alone. By intersecting ultrasonic beams, audible difference tones can be created in local regions of space. HSS inc has not revealed in depth technical information, so it will be a while before this intriguing method can be assessed fully.

4.7.11 Summary

The interests of the audio engineer and the musician, while related, are often not very well aligned. The commercial audio world is largely dominated by recording and editing techniques for broadcasting rather than musically creative tools. Surround sound technology, however, has much to offer electroacoustic music. Ambisonics in particular is well suited to large audiences and can offer realistic perceptual cues not attainable with conventional diffusion. The B-format encoded signal is well suited for recording, manipulation, and synthesis.

4.8 Real-time Signal Processing Tools

4.8.1 Hardware

Until very recently building real-time audio systems was a very specialist activity, involving programming awkward DSP chips in assembler or even designing dedicated DSP hardware. The processing power of widely available general purpose chips has increased to the point where complex audio processing and synthesis can be sustained in real-time. Standard high-level software development tools can be applied to audio applications giving dramatic productivity gains.

4.8.2 C

C was conceived at Bell Laboratories as a language for writing operating systems, so it contains elegant low level syntax for accessing hardware. The same syntax is very appropriate for writing efficient, concise audio DSP code.

C has become very popular amongst the professional software community for a wide range of tasks, and so development tools exist across many platforms. This ensures the portability of C source code, which in turn becomes a reason for using C.

4.8.3 C++

C++ has quickly become the most important general purpose language for commercial uses. It is very nearly a perfect superset of C, and so retains all the low level features such as pointers, which are useful for efficient DSP style code. In addition, many new high-level concepts have been added to aid the construction of large programs. The *Class* structure is particularly suitable for audio processing because it can be used to implement the traditional *unit-generator* structure with ease.

4.8.4 Java

Java is mentioned here not because it is very well suited to efficient audio processing, but because of its portability. From Java access to physical audio ports is wrapped in a standardized interface, like csound, so that an audio program can run immediately on any supporting machine. As Java efficiency and processor speeds continue to increase it may eventually make sense to use Java for serious audio applications.

4.8.5 Csound

Csound was originally derived from the MUSIC 11 and MUSIC 360 programs of Max Mathews, as a program for creating tape compositions containing synthesized and re-synthesized sounds. It is written in C, which has ensured its availability on many platforms.

The user creates two script files, the *orchestra* defining a number of instruments in terms of unit-generator functions and variables, and the *score* containing lists of numbers specifying when and how the instruments should be played.[14] The unit-generators cover a wide range of useful functions, which are constantly being added to by different contributors.

Real-time Csound

Real-time processing was originally available via the *in* command, which can read audio from the external ports. If the output is also to the ports, the system will attempt to process in realtime, under the clock control of the output port. If the processing cannot keep pace with the rate of output samples, the output buffer runs dry and the audio is broken.

In 1991 a set of MIDI unit-generators where added. If a MIDI note on is received on one of the 16 MIDI channels, the instrument with the corresponding ID number is triggered. The MIDI unit-generators can be used to read the MIDI data for processing in the instrument. The data includes velocity, pitch and also variables that change while the note is held, such as pitchbend and control messages. The minimum requirement in the score is to sustain a dummy instrument preventing Csound from halting when no instruments are playing.

[14]An introduction and manual may be found at *The Leeds Csound Site,* [Ver94]

In real-time mode Csound suffers from its original design as a audio batch processor. There are no easy ways for creating rich conditional programming structures found in general purpose languages. One possible approach to this problem is to preprocess control events with a C-like language then pipe the results to Csound.

4.8.6 Graphical Synthesiser Design Systems

Rapid increase in desktop computing power, has led to the appearance of a number of graphical development systems aimed at real-time synthesis design. Typically, processes are represented as boxes connected by lines indicating signal flow. Graphical design aids visualization of the overall design but often at the cost of reducing the level of syntax. One of the first appeared as an audio extension of the *MAX* real-time MIDI environment developed by Miller Puckette at Ircam. The audio version is now known as MAX/FTS on unix systems and MAX/MSP on macintosh systems, and has become a popular commercial product, expecially in America.

Synthbuilder from CCRMA, Stanford University [PJS$^+$98], has been available as freeware for several years on the NeXt platform, and is soon to introduced commercially for Windows. This is a very flexible system, designed as a tool for developing the waveguide synthesis techniques originating from CCRMA.

Reality by Seer Systems[15] is a graphical programmable synthesizer based on CCRMA-style synthesis techniques, and aimed firmly at the general public. Another high quality system, without waveguide synthesis, is *Generator* by Native Instruments[16].

Numerous other graphical-design synthesis packages can be found on the internet, usually integrated with a 'tracker' style sequencer for producing electronic dance music. A good example is *buzz*[17]. This can be used as an introduction to synthesis, but lacks the power and flexibility of the more professional systems.

4.8.7 Computer Aided Design Systems

Within the commercial sector high end general purpose tools exist for the design of signal processing systems, for instance *The Signal Processing Workstation by Cadence Design Systems Inc*. The output can be a hardware specification for programmable logic device, object code for a DSP chip or just C code for a general purpose chip.

While these systems are very user friendly, with professional hierarchal graphical interfaces, they are not designed for rapid testing of real-time audio processes using the host machine. Further more, the included libraries are of limited use in comparison with the built-in Csound library.

[15]*www.reality.com*, 20 Jan 99

[16]*www.native-instruments*, 20 Jan 99

[17]*http://buzz.scene.org* , Dec 1998

Ultimately if special hardware processing is required, professional CAD systems are needed to generate the final design, but it may be worth prototyping the initial designs with tools like Csound.

4.9 Summary

This chapter has reviewed some audio processing and synthesis techniques in relation to live performance. Two emerging technologies are of particular interest. Physical modeling has opened the door to a radically different kind of synthesis, which appeals to the human sensibility for the complexities of natural sound, yet at the same time offers the opportunity to develop instruments that would be physically impractical or impossible. A corresponding vacuum has been created for the development of new performance interfaces which can exploit physical modeling synthesis.

Surround sound technologies have gained importance in the home, cinemas, theatres and museums. Their application to the traditional diffusion techniques of electroacoustic music has been gradual, but possibilities exist for expansion in this area, most notably with Ambisonics. Synthesis with surround sound is increasingly concerned with accurate physical modeling of the environment, as this enhances the perception of direction and distance. A strong link is therefore becoming apparent between physical synthesis and surround technology.

Chapter 5

Alternative Use of Keyboard Controllers for Synthesis

5.1 The MIDI Keyboard as a Controller

The MIDI keyboard, hereafter referred to just as ‘keyboard’, is by far the most common controller. It comes in a wide range of sizes and prices from a 2-octave note-pad to a full 88-note weighted keyboard with hammer action, and these are all compatible thanks to the MIDI interface standard. The usage of keyboards other than for playing notes, samples or chords appears to be quite restricted. Some synthesizers have *arpeggiators* which generate patterns of notes that cycle between held keys.[1]

The purpose of this chapter is to investigate how keyboards might be used in the performance of electroacoustic music while avoiding the pervasive ‘note paradigm’. At first this may seem a contradiction, because keyboards were developed precisely to control patterns of notes. The product of this development consists of a series of ingenious features, some of which can be usefully employed in a more general context:

- *Navigation* - Periodic black and white note patterns help individual keys to be rapidly identified visually. The contour of the keys likewise helps in this respect, as demonstrated by blind pianists. Note that keyboard players, of whom there are many, will have rapid key identification skills already well developed.
- *Precision velocity* and ON / OFF timing - The dynamic range of velocity control is particularly great for *weighted* keyboards or those containing a mimic of real hammer action.
- *Added controllers* - Mod wheels, touch pads, aftertouch are all common items on MIDI keyboards, and offer possibilities for expanding on control by keyboard alone. Polyphonic after-

[1] The word *key* shall refer to the physical keyboard key and should not be confused with key signature. *note* refers to the *sound* generated by playing a key on a conventional keyboard synthesizer.

touch, where each key sends separate pressure information, is of great interest but will not be considered further, as it is rarely found on commercial products.

- *High Level* Techniques - *Chords* are the synchronization of several key on and off events. *Shaking* is a stochastic-like process for repeatedly playing a group of keys, possibly with uneven emphasis. This provides a way to 'perform' a time varying curve, in much the same way as an artist sketches a curve with repeated component strokes.

5.2 New Keyboard Instrument Models

Our aim is to avoid the 'note paradigm', that each *sound* in the final mix is a simple function of only one key. By *sound* we mean that the mix can be composed as a sum of separate independent sounds. [2] The converse of this criteria is that the mix consists of a sum of sounds some of which may be dependent on several keys. This is not as strange as it first seems. The piano sustain pedal is essentially like a key but interacts with the main keys. Even the keys are not truly independent on a real piano, because once a key is being hit and the damper for that key is lifted, it is open to vibrate sympathetically with sound coming from any other vibrating strings. The subtleties of this process contribute significantly to the overall experience of playing a piano.

5.2.1 Instrument classes

There are many possible ways to make key control information interact to produce sound. The most general case is illustrated in **Figure 5.1**. The information from each key consists just of the standard MIDI note information; The note start time and velocity and the note end time. The information from all the keys is processed by a completely general causal filter resulting in audio frequency output.

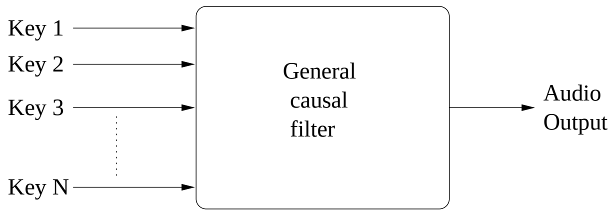


Figure 5.1: Most general keyboard controlled instrument.

A particular subclass of the general case is shown in **Figure 5.2**. Each key passes its signal to a different generalized filter. The filter outputs are combined by simple arithmetic operations to provide the input for another generalized filter, representing a synthesis process. This subclass is a simple example of how several keys may control one sound. The key signals cannot be reconstructed from sound output. Designing subclass structures helps in building an overview of an instrument before considering the finer details.

[2] It is possible that this condition could be relaxed so that we cannot represent the sound mix as a simple sum of independently controlled sounds, and yet the sounds *appear* independent

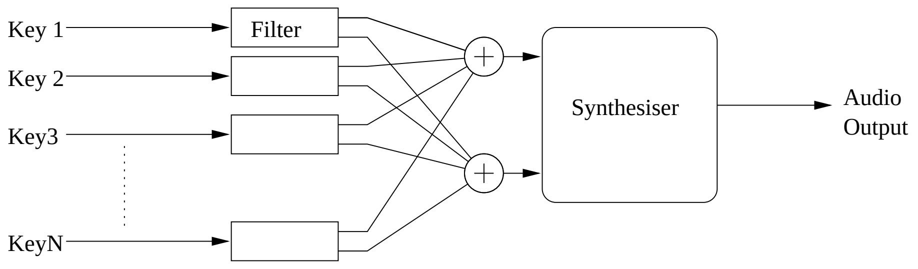


Figure 5.2: A subclass of keyboard instruments

5.3 Spiragraph - a Csound Instrument

Csound was adopted for its convenient features; ready made signal processing functions and a rapid test cycle due to minimal compilation time. This in turn dictated the kind of structures which were most naturally created. The original inspiration for the instrument came from the different kinds of cyclic sounds heard in the natural world. Examples include the chorus of frogs, some types of bird call, locusts, cicadas - the list is large. Common in these sounds are fluctuations in pitch and amplitude occurring predominantly in the range up to around 50Hz. In other words a form of AM and FM is occurring at much lower frequencies than generally used for synthesis, but at higher frequencies than used for vibrato and tremolo effects. [3]

5.3.1 Synthesis

Having established the target sounds, the next step was to design a suitable synthesis system. A way forward was suggested by the *spiragraph* device for drawing patterns such as those found on money notes. Complex cyclic patterns are created by the interlocking motion of different rotating wheels. A similar effect can created sonically by summing several oscillators with frequencies of similar order. At audio frequencies this is uninteresting because we immediately separate the mix into frequency components. However, when using the mix to modulate at the intermediate frequencies the listener can neither resolve separate frequency components nor follow the modulations over time.

5.3.2 Control

How should the oscillating components be controlled? Csound contains built in envelope generators, so a natural possibility is to trigger envelopes using key signals. This falls into the general scheme shown in **Figure 5.2**. *spiragraph1.orc* listed in **Appendix A**, is a Csound orchestra implementing the instrument shown in **Figure 5.3**.

Each key has its own envelope and oscillator wavetable for offsetting the pitch of the main audio oscillator. The envelope parameters and number of the wavetable are selected from a table in

[3]It is in this frequency band that modulation techniques appear effective for communication between many different kinds of animals.

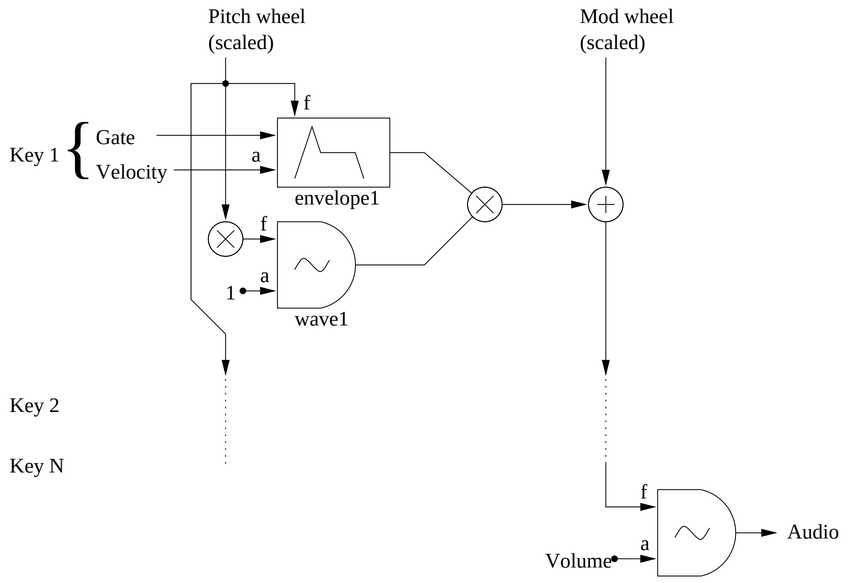


Figure 5.3: Graphical representation of Csound instrument *spiragraph1.orc*

spiragraph.sco. By choosing a constant wavetable, the pitch modulation can consist purely of the envelope part, and vice versa the envelope can be flattened. The example score given has range of different key settings using a few wavetables. Modulation oscillator frequencies are available in a close spaced arrangement, for slowly evolving effects, and in harmonic or near harmonic arrangements. **Track 1** on the CD shows the effect of each key separately, making clear the range of waveforms, envelopes and frequencies of frequency modulation. **Track 2** shows how the depth of frequency modulation increases with key velocity.

In addition to keys, input from the mod and pitch wheels is used to have an global effect on the instrument. The mod wheel controls the centre pitch of the audio oscillator, while the pitch wheel speeds up or slows down the modulating oscillators. In practice the mod wheel is usually left alone, while the pitch wheel is frequently controlled to create *accelerando* and *rallentando* effects. The effect of the wheels is shown in **Tracks 3,4**.

5.3.3 In Performance

The scheme is very simple, but the ear perceives a rich interaction between the keys. The listener is aware of different kinds of pattern in the sound, but at the same time is expectant of new variations. Simple phasing interaction between two keys which differ only by the frequency of modulation is shown in **Track 8**. More complex interactions result from several static keys, **Track 10**, and keys which are performed, **Track 11**. With practice the player learns to predict how different keys will interact when played with different velocities, but the exact sound is unlikely to be repeated, because that would require the phases of all the modulators to be synchronized. **Track 9** gives an example of this using two keys, one held, the other repeated.

5.3.4 Spiragraph2.orc

Spiragraph1.orc was extended by adding amplitude modulators. The envelope component can be used to switch the whole instrument on when any key is depressed. *Spiragraph2.orc* actually uses two amplitude modulators per key, one for each stereo channel, see **Figure 5.4**. This leads to a rich stereo pattern which complements the pitch pattern. Extra amplitude control is provided by adding aftertouch pressure control. Listen to **Tracks 5,6,12,13** for examples of how the amplitude modulation extends the instrument.

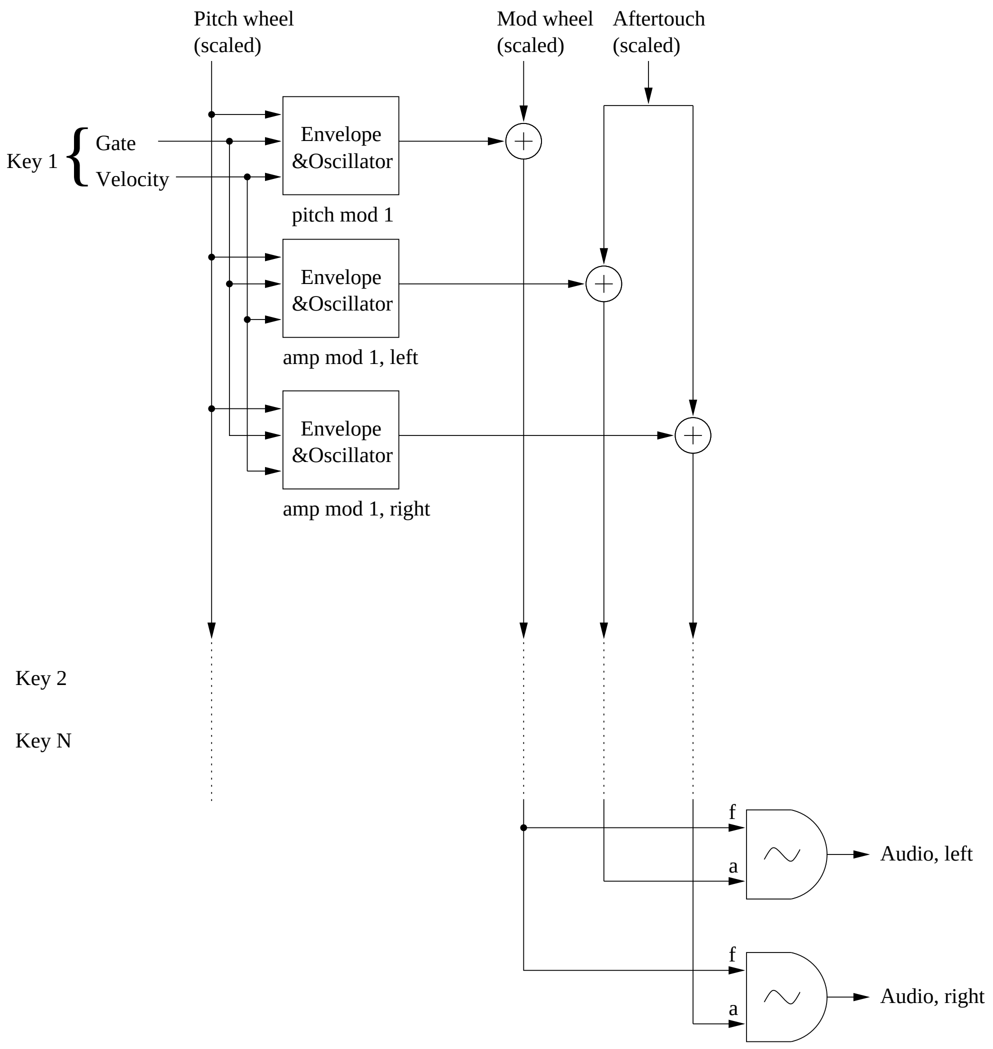


Figure 5.4: Graphical representation of Csound instrument *spiragraph2.orc*

Track 14 is a complete piece, *Lifeforms*, made using Spiragraph2.orc. While this does not demonstrate live performance, it gives an example of what can be achieved in real-time compositional environment. A wide range of cyclic sounds were produced, ranging from low squelching to rapid high pitched warbling reminiscent of the *sewing-machine bird*.

5.3.5 Spiragraph3.orc

The final version of Spiragraph has two audio oscillators driven by separate pitch modulators. The stereo modulation of each oscillator is the same. The extra complexity does not enhance the instrument significantly, and at times the sound becomes confused.

5.4 Summary

This chapter has experimented with the use of alternative architectures for performing sound with a keyboard. Even with the simple system used in the Spiragraph programs, some fascinating effects can be generated. The synthesis core of Spiragraph is interesting in itself, but this interest is only exploited by using a suitable control structure. A more conventional note-based control approach could not do this.

Micro Unpredictability

A feature of the synthesis core which is of general interest is that the player can predict with some accuracy how playing the keys will effect the overall features of the sound, but the intricacies of the sound appear to be constantly changing. This helps to sustain interest in the control of the sound.

Development

Spiragraph was deliberately constructed using a small number of simple components, to emphasize the importance of the overall architecture. There are many possibilities for developing numerous related instruments with increased complexity. Essentially, any synthesis model could be adapted to a structure similar to 5.2.

One important aspect of the general model shown in **Figure 5.1** was not explored. This is the explicit dependence of synthesis input on key events older than the last event. A simple example would be to calculate a moving average of a key's velocity values, to be processed further before passing to the synthesis stage. More complex processing could, for instance, modify future key presses dependent on the frequency of recent key presses.

Chapter 6

LAmb, A System for Live Diffusion Using Ambisonics

6.1 Introduction

Chapter 5 was concerned with developing strategies for performing synthesized sound using a MIDI keyboard. This thread is now continued in application to the diffusion of sound using the Ambisonic methodology introduced in Section 4.7.7. There it was concluded that Ambisonics has unfulfilled potential for live performance. This chapter covers the development of a new diffusion instrument which attempts to tap into some of the unused potential.

First, some new B-format synthesis and processing techniques are described. These are eventually incorporated within a structured performance framework to form the *LAmb*[1] system.

6.1.1 Conventional Ambisonic Diffusion

Ambisonics is a set of methods for recording an approximation of a soundfield about a point and reproducing its impression on a set of speakers surrounding a non-translating listener. Its key design feature is that the impression is holographic. That is, it transforms appropriately to head rotation. The range of positions over which the image is stable makes it practical to enclose an audience within one array. The perceived realism of Ambisonic sound can be contrasted with conventionally diffused sound which has poor inherent image stability unless each independent sound is panned fully to a speaker.

[1]Abbreviation of **L***ive* **Amb***isonics*

Panning Devices

A spin-off from the recording technology has been the development of analog panning devices for converting mono and stereo signals to the soundfield format, *B-format*. The simplest of these is a device which takes a mono signal and creates a B-format signal for which the sound appears to come from the direction indicated by a control. Gerzon's pan-pot devices, [Ger75a], include a simple device equipped with a joystick. While the B-format pan-pot was conceived as a production tool, it has found use at York University in live electroacoustic music, in combination with a mixing desk used for live conventional diffusion.

Distance

One might at first hope that moving the joystick across the central region might give the some perception of change in distance. Unfortunately this is not the case. The central region of these pan-pots is loosely referred to as producing an *interior* sound, meaning the W component is boosted above the normal strength for a B-format source signal. The interior effect is not discussed as a parameter available for careful control, because the analog pan-pot circuitry cannot be easily modified to give particular variations. The interior behaviour of the pan-pots seems to be just a by-product of fixing the perimeter control.

Ambisonics is fundamentally about reproducing the illusion of the sound coming towards the listener from different *directions*. The perception of *distance* comes from a number of cues derived from 'higher level' features of the sound (which are of general importance in any surround sound system). Many of these cues would be totally impractical to simulate in the analog domain.

Distance Cues

There are many cues which are used to perceive distance. It appears that any cue which can mathematically define distance can make a valid contribution to the perception of distance. The following list is compiled from general texts in this area with a few original observations on high-level localisation.

- Air filtering. High frequency attenuation increases with distance.
- Early reflections. The timing and relative magnitude of reflections gives a precise fix on the distance of the source.
- Reverberation mix. A greater reverberation to source mix indicates a more distant source.
- Expected loudness. Natural sounds have typical associated loudnesses at different distances. If the source becomes louder then the timbral quality of the sound probably changes, and the listener adjusts the loudness to distance conversion.

- Object width. If the size of an object radiating sound is known, by recognition of the sound, then its distance can be judged from the angle subtended by the radiated sound at the listener.
- Movement Correlation. Much more information can be gathered by the above methods if the source or listener moves. It is possible correlate sound received at different times.
- Multiple Object Correlation. Sound from several objects provides information on their relative position, but also illuminates the surrounding environment, which in turn helps determine absolute positions.
- Movement Doppler effect. The *Doppler effect* causes changes of pitch which clearly indicate when the source and listener are closest. The distance cannot be directly deduced, but in the presence of other distance indicators a consistent Doppler effect contributes to a convincing perception of the environment, and so a convincing perception of the distances within.
- General consistency. Doppler consistency is only one example of many. We expect to hear objects which obey familiar Newtonian dynamics. Presented with a complex soundfield, we build a picture of objects moving in smooth paths.

Spreading Pan-pots

Gerzon describes, [Ger75b],some devices for spreading mono or stereo sound onto an arc of the Ambisonic sound stage, which helps to give a more natural impression of sound radiating from an object with width. The effect is separately localise each frequency component to a point on an arc. The width control is not linked to the direction joystick, so the width cannot be used to give a direct impression of distance and movement. Again this is a problem of limited analog circuit complexity.

Soundfield Controls

In the pan-pot report, [Ger75a], Gerzon also describes some joystick controlled devices for applying rotation and *dominance* to B-format signals. Recall that dominance is a linear distortion of the field which causes a shift of perceived directions towards one direction, and a relative increase in gain for directions closer to the dominance direction.

6.1.2 Digital Ambisonic Diffusion

Comparing Digital with Analog Techniques

The analog Ambisonic devices are useful, but offer limited scope for creating interesting B-format signals in real-time. B-format recordings of *natural* environments are very complex, containing many directional components. Using software processing, or *native DSP*, in a language such as C has the advantage that the processing complexity is much more manageable, and therefore tractable.

Development is faster due to simpler design methods and faster rebuild times. The end result is also much more portable.

The one serious disadvantage is that the kind of computers which are suitable, fast desktop computers, are not very well suited to the concert environment because of their size, lack of robustness and noise generation. This situation is, of course, rapidly changing as technology advances which provides added impetus for the present developments.

New Controllers

It would be desirable to control several sources at once rather than just one. It is not physically possible for one person to control several joysticks at once, so another method of control is required. The MIDI keyboard was considered for this purpose, at first, simply because of its availability. Then it became clear that keyboard offers some interesting control solutions which are quite different to conventional pan-pot control.

6.2 LAmb Development Introduction

The following sections describe the development of the LAmb system. In practice different parts were developed interactively, rather than in the linear fashion in which they are presented. The processing power available using a Silicon Graphics Indy, although impressive, is obviously limited. It has been important to assess at each stage, the most efficient ways of using the resources.[2]

The system is presented by building upwards from the simplest components, towards an integrated system. The signal processing techniques are presented first, followed by methods of control using a MIDI keyboard and a GUI.

6.3 A New Panning Algorithm

6.3.1 Recreating The Analog Pan-pots in DSP

Before trying to extend the functionality of the analog pan-pots, their simulation in software is first examined. The central region of the joystick control is not immediately well defined, or useful, but the perimeter regions can readily be simulated.

In the following, **bold** type denotes audio signals. If the source signal is $\mathbf{S}$ and the joystick position is given in cartesians by x, y, z with normalised components $\bar{x}, \bar{y}, \bar{z}$, then a B-format signal of constant gain in this direction is, from the discussion in Section 4.7.7:

[2]In the digital domain there is at least the consolation that optimised methods will remain valid for the foreseeable future, since they will transfer essentially unchanged from one computer to the next.

$$\begin{pmatrix} \mathbf{W} \\ \mathbf{X} \\ \mathbf{Y} \\ \mathbf{Z} \end{pmatrix} = \mathbf{S} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \overline{x} \\ \overline{y} \\ \overline{z} \end{pmatrix} \tag{6.1}$$

Because this is the B-format signal of a single source, the gain variables W, X, Y, Z are also defined, for convenience:

$$\begin{pmatrix} W \\ X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \overline{x} \\ \overline{y} \\ \overline{z} \end{pmatrix} \tag{6.2}$$

Using 6.1, the B-format signal remains unchanged when the joystick is moved in a radial direction, until the point when it crosses the centre, and the encoded direction flips by 180 degrees. This behaviour offers an immediate advantage over the original system in that rapid angular changes can be made while maintaining a well defined B-format signal.

6.3.2 Distance Gain Profile

The most obvious improvement to the simulation is to add a gain factor to model the decrease of received signal with distance. There are two factors to consider here; geometry of space and ergonomics of control.

Space Geometry

For a completely free 3-dimensional space, sound energy flux from a point falls as $1/r^2$ with distance r, so the acoustic pressure amplitude falls as $1/r$. For spaces of lower dimension the fall off is less steep, $1/r^k$, $0 < k < 1$. For the case of 1-dimension, for instance a *waveguide* or ideal corridor, there is no fall off in energy flux. In practice some additional exponential component to energy fall off may be introduced to model absorption, by air and walls but this is better implemented using the filtering discussed later.

Using the term $1/r^k$ as a gain factor is impractical because it is unlimited. This arises because the term models a point radiating source. In reality objects have radiation patterns distributed over a non-zero surface area. A means to limit the gain, while retaining a suitable fall off at larger distances, is to use the modified gain factor

$$\beta_r = \frac{a(1+r)}{b(1+r) + r^{1+k}} \qquad k > 0 \tag{6.3}$$

The peak gain, when $r = 0$, is a/b, and the gain at distance is a/r^k. Choose $k = 1$ for 3D space $k = \frac{1}{2}$ for 2D space. The function is smooth about the centre, like the gain factor for a non-point object (unlike simpler functions such as $1/(r^k + 1)$ when $k \leq 1$).

Ergonomics of Control

If the physical position of the joystick is proportionate to the variables x, y, z then the distance of the source is limited by the perimeter position of the joystick. This perimeter distance can be scaled to any value, but this will have the adverse effect of making the central region very small. For ergonomic purposes therefore, it can be desirable to map the joystick to x, y, z in such a way that the rate of change in source distance is much greater for joystick movement near the perimeter. This is a particular case of concentrating human physical control resolution where it is most important. The same issue will arise whatever control scheme is used.

6.3.3 First Attempt to Control the Interior Effect

The gain factor 6.3 was based on the practical need to avoid the side effects of modelling the source as a point. The result gives something of the impression of a non-point source, but the sudden unnatural switch in direction occurring at the origin still remains.

If 6.1 is used with a 2D speaker rig, and z is set fixed , off-centre, the source pans smoothly with no centre switching, although some loss of volume is experienced at the centre. With the distance gain factor added, the source begins to give the impression of a 'cloud' radiating sound. As the source moves through the centre the cloud passes through the listener. Passing through diffuse objects such as fly swarms, is part of our common experience, but it is not *very* often that this happens. Part of the reason for the success of this effect could be due to the lack of visual cues when listening to a surround system. Any effect which provides very strong cues about distance and movement gains more relative importance than if visual cues were present.

The technique can be extended to 3-dimensions by keeping z as before and adding an extra term o behaving like x, y, z but being held fixed and non-zero. The new B-format signal is defined by using 6.1, but with the normals redefined as

$$\bar{x} = x/\sqrt{x^2 + y^2 + z^2 + o^2} \tag{6.4}$$

$$\bar{y} = y/\sqrt{x^2 + y^2 + z^2 + o^2} \tag{6.5}$$

$$\bar{z} = z/\sqrt{x^2 + y^2 + z^2 + o^2} \tag{6.6}$$

6.3.4 Object Synthesis

Combining 6.1 and 6.4 gives the impression of a diffuse object, but in order to control the effect better, more understanding is required. Using the modified normals makes the relative strength of the **W** component dependent on position:

$$X^2 + Y^2 + Z^2 = 2\gamma W^2 \qquad 0 \leq \gamma < 1 \tag{6.7}$$

The case $\gamma = 1$ gives the *optimum* localisation for a directed source[3] As the source moves towards the centre γ falls to zero. For $0 < \gamma < 1$, consistent perception of source direction is guarrenteed by the Ambisonic decoding theorems, [Ger92], but the strength of direction perception weakens as γ falls below 1.

The term *interior* is sometimes used to refer to signals encoded with $\gamma < 1$. This is because when an interior source is static it gives the impression of being inside the listeners head. How, then, can the system using 6.4 give the impression of a diffuse object?

Diffuse Radiating Object Model

Consider a simplified model for a radiating object in 3-dimensions consisting of a sphere radiating the same signal, *S*, perhaps with varying gain, from every part of its volume. **Figure 6.1** shows a one-dimensional section through the centre of such an object.

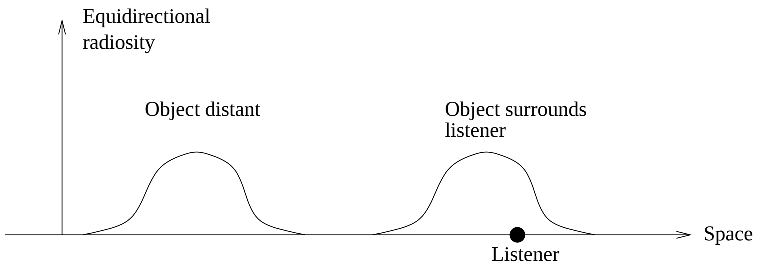


Figure 6.1: A diffuse acoustic object.

Now construct the B-format signal for this model for varying distances from the listener. If we ignore the phase effects, which are minimised by using a spectrally rich signal, then the B-format signal is a simple multiple of the signal *S*. The corresponding γ component defined by 6.7 varies in a similar manner to that resulting from use of 6.4. Likewise, the *loudness* of the image would intuitively appear to obey a similar law to 6.3. Certainly in the limit of increasing separation, the source is seen as a point. In the central region it is less clear. A definition of loudness is required so that the loudness of two B-format signals can be compared. One way is to calculate the physical acoustic energy in the encodings. It is plausible that soundfields with equal energy will sound roughly similar

[3]Strictly speaking this is true for frequencies upto about 700Hz where the velocity cue dominates. Above this energy localisation becomes more important, for which W should be 3dB stronger.

in volume, especially if derived from the same source. Attaching psychoacoustical significance to this measurement is valuable because it then provides a more intuitive basis for performance control.

The total energy is readily computed because $\mathbf{W},\mathbf{X},\mathbf{Y},\mathbf{Z}$ are the magnitudes of orthogonal functions ($\mathbf{W}$ is scaled from its normalized function value). The total energy is $2\mathbf{W^2}+\mathbf{X^2}+\mathbf{Y^2}+\mathbf{Z^2}$. Now the gain relative to the source can be conveniently measured with a new variable, β, defined by:

$$2\mathbf{W^2}+\mathbf{X^2}+\mathbf{Y^2}+\mathbf{Z^2} = \beta^2\mathbf{S^2} \tag{6.8}$$

$$2W^2+X^2+Y^2+Z^2 = \beta^2 \tag{6.9}$$

A plot of β against source-listener distance using 6.4 but without the factor 6.3, shows a dip around 0. This means β and γ variation are mixed when using the modified normals.

Let $\underline{\mathbf{p}}$ represent the source direction in 3-space, then $(\underline{\mathbf{p}},\gamma,\beta)$ form an alternative variable set to (W,X,Y,Z), which is both more psychoacoustically intuitive and more simply related to the object model. It is therefore useful to derive the inverse transformation so that $(\underline{\mathbf{p}},\gamma,\beta)$ can be controlled directly.

Subtracting 6.9 from 6.7,

$$-2W^2 = 2\gamma W^2-\beta^2 \tag{6.10}$$

$$\beta^2 = 2W^2(1+\gamma) \tag{6.11}$$

$$W = \frac{\beta}{\sqrt{2}\sqrt{1+\gamma}} \tag{6.12}$$

From 6.7,

$$\frac{1}{\gamma}(X^2+Y^2+Z^2) = 2W^2 \tag{6.13}$$

Adding 6.9,

$$(X^2+Y^2+Z^2)(1+\frac{1}{\gamma}) = \beta^2 \tag{6.14}$$

$$X^2(1+\frac{y^2}{x^2}+\frac{z^2}{x^2}) = \beta/(1+\frac{1}{\gamma}) \tag{6.15}$$

$y/x, z/x$ and similar terms in expressions for Y and Z are just functions of $\underline{\mathbf{p}}$, which are convenient to evaluate from the cartesian separation of source and listener given by (x,y,z).

The full solution is,

$$X = \begin{cases} 0 & \gamma=0,\, x=0 \\ \dfrac{\beta}{\left[\left(1+\frac{1}{\gamma}\right)\left(1+\frac{y^2}{x^2}+\frac{z^2}{x^2}\right)\right]^{\frac{1}{2}}} & \text{otherwise} \end{cases} \tag{6.16}$$

Similarly for Y and Z. For the diffuse object model, the important features of β and γ distance variation can be deduced without detailed calculation by integration (**Figure 6.2**). β tends to a

$1/r^k$ fall off at distance. γ tends to 1 at distance, and falls to zero at zero separation. To represent consistently an object of the same size, the greatest part of the variation of β and γ must take place over similar distance ranges. The precise shape for the curves near zero will depend on the particular object radiation distribution (shown in **Figure 6.1**).

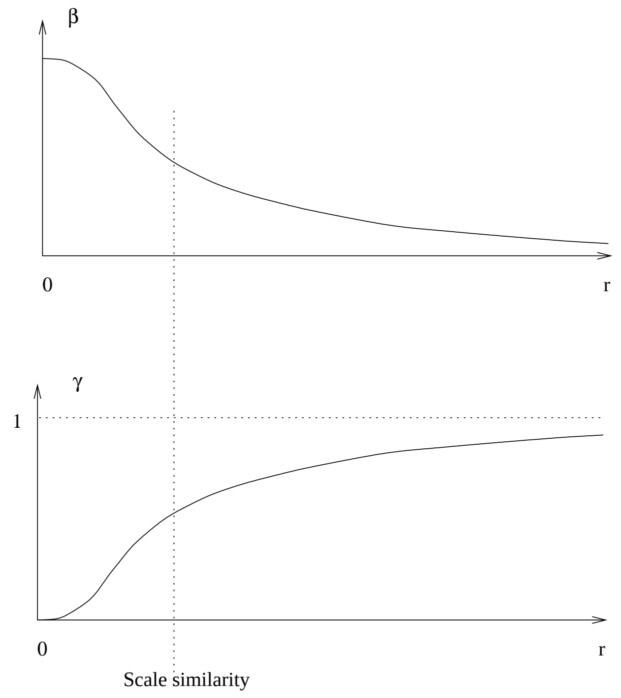


Figure 6.2: Consistent γ and β distance profiles for an object.

A suitable function for β is the factor 6.3. The large distance behaviour of γ is certainly well defined, independent of the object distribution. It does not appear worthwhile calculating an approximation, however, because the effect of γ is only significant near the centre. On the other hand it was important to establish the fall off of β, because this has a significant effect as discussed previously.

For γ a suitable, inexpensive function is,

$$\gamma = \frac{ar^2}{r^2 + b} \tag{6.17}$$

a controls the overall gain, while b controls the scale, or width of the part of the function about $r = 0$.

Approximation and Calculation

The completed panning equations 6.16, 6.3, 6.17, are a little awkward. They may be approximated to help speed up calculation, or simplify code structure. Note however, that the equations do not need to be recalculated at audio rate. The maximum rates of change of W, X, Y, Z are low enough so that they can be approximated by updating at a lower rate and interpolating in between. More details on this will be discussed later.

6.3.5 Acoustics and Movement

It is surprising that simple mixing can produce some sensation of a diffuse object. The main simplifying assumption in the model, was that the object radiates the same source signal everywhere in its volume, giving rise to an interior sound. Real objects display a much more complex variation in radiation distribution, but retain an interior component due to the correlation of sound radiated from different parts of the object.

The added sound of the well-localized reflections from the diffusion room can improve the naturalness of the image, particularly the central high W region, by more closely emulating natural sources. Similarly, movement helps by allowing the listener to correlate the soundfield at successive instants. Exaggerated movement is not required for this effect. Low speeds are often more impressive than high speeds.

6.4 Spreading Effects

The main weakness in the object panning technique is that variation in sound across the object is not encoded. While it is straightforward in principle to construct a complex B-format object by synthesis of many components, in the context of diffusion only a few source signals are available. A method is required for splitting the source signal into components that can be given different directions.

6.4.1 Gerzon's analog spreading pan-pot

One approach, which can be found in [Ger75b], is to effectively split a source signal into separate signals by filtering, then encode these into different directions. In the *spread pan-pot* design, the signal is spread about the direction indicated by a joystick. The analog circuit uses a 6 pole all pass filter to wrap the source spectrum around a line segment in (X,Y) space, as shown in **Figure 6.3**

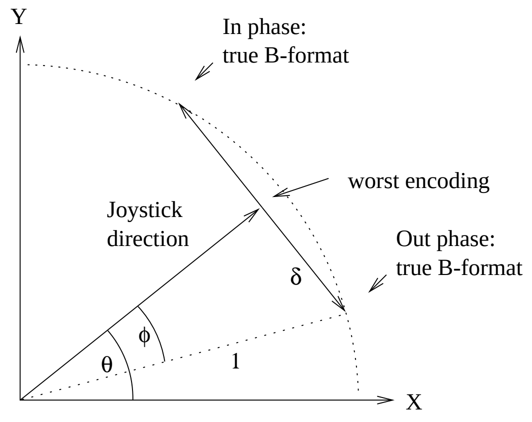


Figure 6.3: Effect of Gerzon's Analog Spread Pan-Pot.

The ends of the line segment are scaled to be encoded correctly, according to $X^2+Y^2=2W^2$, but consequently the centre point of the segment is incorrectly encoded, suffering phase and interior distortions which increase with the spreading angle. These distortions effectively limit the natural-sounding spreading to about 90 degrees. In the limit when the spreading angle is 180 degrees, only two points 180 degrees apart are correctly localized, and these are at 90 degrees to the direction in which the object was moving. Frequencies which show a 90 degree phase shift to the source are completely unlocalized. In the previous section, simple mixing was used to create an object moving through the centre. Although localization is lost completely at the centre, up to that point there remains a consistent direction cue directed towards the object centre. In the spread pan-pot the direction cue splits, leaving a hole.

One compromise which improves the situation is to alter the spread mixing near the centre, so that the (X,Y) components are reduced while the (W) component is boosted to provide the dominant source of energy, as in Section 6.3.4. The result is a signal near the centre which is nearly completely undirected, rather than containing many conflicting directional cues which change in confusing ways as the object moves near the centre.

6.4.2 A Digital Spreading Pan-pot

A similar effect to the analog spread pan-pot can be achieved easily in the digital domain. Delaying the source is in effect a multi-pole all pass filtering operation. Let $\mathbf{S}'$ be the delayed signal and δ the amount of spread, related to the spread arc, ϕ, by $\sin\phi/2 = \delta$. Then the full B-format signal is:

$$\mathbf{X} = (1-\delta^2)^{\frac{1}{2}}\cos\theta\mathbf{S} - \delta\sin\theta\mathbf{S}' \tag{6.18}$$

$$\mathbf{Y} = (1-\delta^2)^{\frac{1}{2}}\sin\theta\mathbf{S} + \delta\cos\theta\mathbf{S}' \tag{6.19}$$

$$\mathbf{W} = \frac{1}{\sqrt{2}}\mathbf{S} \tag{6.20}$$

The spread angle is limited by distortion problems in the same way as the analog version, but has the advantage that the digital all pass filter can wrap the source across the spreading angle many more times. If the delay is chosen to be too big, > 1 ms, then reflection-like artifacts will be heard. If the delay is too small, < 0.1 ms then the comb filtering effect becomes noticeable. Particular delay settings can be chosen to maximize the 'naturalness' of a particular sound.

Controlling Spread for Object Modelling

In the simple object model, δ can be found in terms of the visible object width, w, and the listener-object distance, d. In **Figure 6.4** ϕ has the same meaning as in **Figure 6.3**.

By comparison of the figures,

$$\frac{w}{2d} = \frac{\delta}{(1-\delta^2)^{\frac{1}{2}}} \tag{6.21}$$

$$\Rightarrow \quad \delta = (1+\left(\frac{2d}{w}\right)^2)^{\frac{1}{2}} \tag{6.22}$$

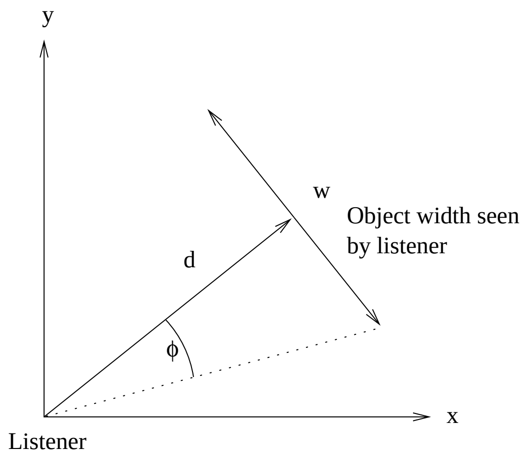


Figure 6.4: Geometry of object relative to listener.

6.4.3 True Controllable Spreading

To simulate more accurately an object which approaches and then possibly envelops the listener, continuous variation in spreading is required from 0 up to 360 degrees, while maintaining good localization. Gerzon describes a *mono spreader*, [Ger75b], which spreads a source signal about the full circle in true B-format. An overview of the spreader is shown in **Figure 6.5**

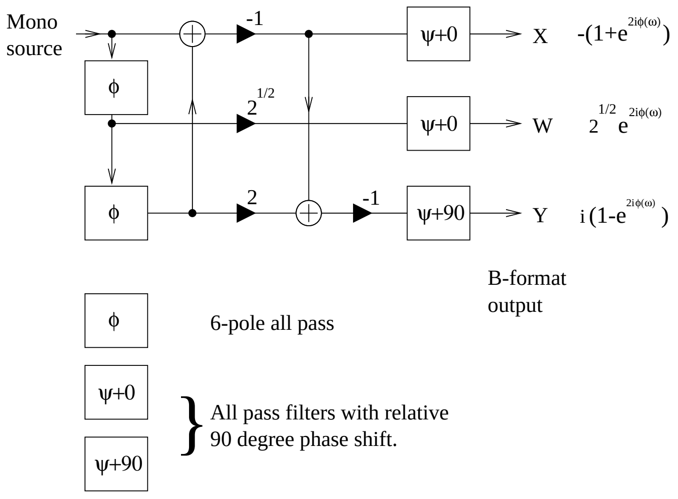

Mono
source
-1
ψ+0
X
$-(1+e^{2i\phi(\omega)})$
ϕ
$2^{1/2}$
ψ+0
W
$2^{1/2} e^{2i\phi(\omega)}$
ϕ
2
-1
ψ+90
Y
$i(1-e^{2i\phi(\omega)})$
B-format
output
ϕ
6-pole all pass
ψ+0
ψ+90
All pass filters with relative
90 degree phase shift.

Figure 6.5: Gerzon's mono spreader.

The phase of the ϕ allpass winds around the circle several times. With the gain matrix the ϕ filters form a phasor encoding of the source. The Ψ filters ensure that output signals are in phase. The figure shows the response of each output to a source with frequency ω, with ϕ(ω) being the phase response of the ϕ filters. It can be quickly verified that $2W^2 = X^2 + Y^2 + Z^2$, and so the output is true B-format.

Controlling ϕ

If the action of ϕ could be restricted to oscillation on a phase arc, rather than the full circle, then the spreader output would consist of true B-format spread over an arc. To design an allpass with a controllable phase arc does not appear straightforward. One possible simplification is to generate a symmetrical arc, then use B-format rotation to locate the centre direction of the B-format spread. B-format rotation does not introduce significant cost. **Figure 6.6** illustrates this. The bold arcs inscribed in each circle indicate the range of directions overwhich the corresponding B-format signal is concentrated.

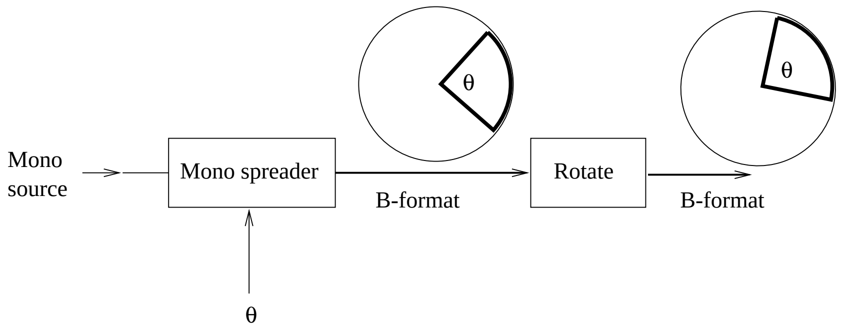


Figure 6.6: True spreader using a controlled allpass filter.

Spreading with Dominance

Another approach, which doesn't require an allpass with controllable phase width, is to apply *dominance* in the direction of the centre of spread. This transforms the full circle so that it becomes concentrated over the required arc (see **Figure 6.7**).

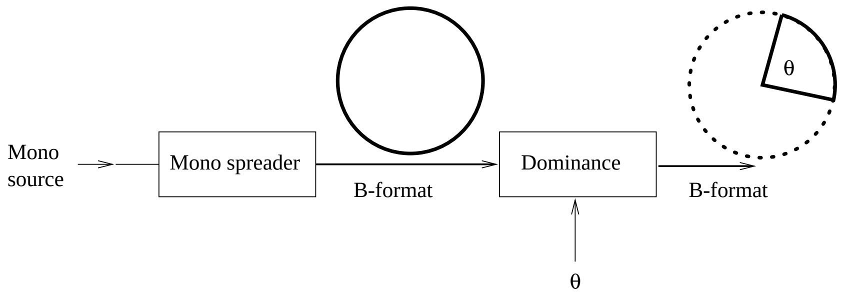


Figure 6.7: True spreader using dominance.

The spread transforms in a suitable fashion for an object passing through the centre, but it is possible for 'rogue' sounds located in the opposite direction to the object to become noticeable. This is because the dominance of a signal directed nearly opposite the direction of dominance does not change direction much.

The dominance spreader is useful although more costly than using the simple delay based spread of Section 6.4.2. Combining the two spreaders would make a better compound spreader, by using dominance in the near range and delay at distance.

6.5 Using B-format Recordings for Object Synthesis

In the preceding spreaders a mono source was processed to spread the component frequencies over an arc, and thus create the impression of an object emitting sound with roughly uniform correlation across its surface or volume. Many real objects have radiation patterns with a much more detailed and interesting structure. These could be synthesized by exhaustive integration, or alternatively we could try and find some means for recording natural radiation patterns so that they can be reproduced.

To simplify the situation initially, consider the radiation pattern about a small, point like object. This cannot be measured directly with a single microphone, however a radiation pattern of comparable complexity can be generated by recording an environment with a soundfield microphone, then inverting the direction of signal travel. In other words a 1st order polar radiation pattern has been formed from a natural source. Let us call this *O-format*. O-format rotates like B-format to give the polar pattern of a rotated object. We are free to compose the environment to achieve a desired polar radiation pattern. Given a collection of point-radiators positioned in space about the listener, see **Figure 6.8**, the B-format signal is composed by encoding the signal received from each object into the direction to that object.

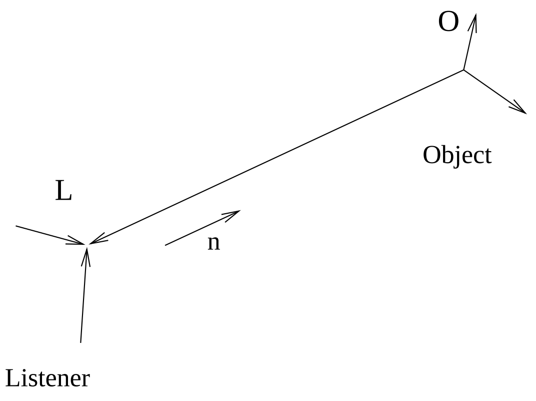


Figure 6.8: Encoding the signal from a point radiator into B-format.

Let the first-order radiation pattern of the object be written **O**, with components written the same as B-format for convenience.

$$\mathbf{O} = \begin{pmatrix} \mathbf{W} \\ \mathbf{X} \\ \mathbf{Y} \\ \mathbf{Z} \end{pmatrix} \tag{6.23}$$

Let the normal vector in the direction from listener to object be **n**.

Define

$$\mathbf{N} \quad = \quad \begin{pmatrix} \sqrt{2} \\ -\mathbf{n} \end{pmatrix} \tag{6.24}$$

$$\overline{\mathbf{N}} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \mathbf{n} \end{pmatrix} \tag{6.25}$$

The 1st order spherical harmonics are cosine functions of direction, which are equal to the components of the normalized direction vector. The received signal, ignoring any distance gain adjustments, is the sum of the harmonics in direction $-\mathbf{n}$. This is just the dot product:

$$\mathbf{O}.\mathbf{N} \tag{6.26}$$

The encoding of the received signal in the received direction is

$$(\mathbf{O}.\mathbf{N})\overline{\mathbf{N}} \tag{6.27}$$

This is a linear transformation from O-format to B-format, and so must consist of rotation, reflection and dominance. Since the B-format signal has a component in only one direction, the transformation must be a maximum-dominance.

6.5.1 Spreading with Dominance

Continuously reducing the dominance factor below maximum level spreads the listener's field to give the impression of a non-point object, in the same way dominance was applied in Section 6.4.3. In terms of reflection, **R**, and dominance, **D**, with the degree of dominance given by λ this transformation is

$$\mathbf{R}_{\mathbf{n}}\mathbf{D}_{-\mathbf{n},\lambda} \tag{6.28}$$

6.5.2 Performance Usage

Performance by diffusion of an O-format signal offers new possibilities as the orientation of objects can be controlled in addition to their position.

The diffusion of an O-format source offers more control possibilities than for a mono source, as the source can be oriented by rotation and pseudo-object width generated using dominance. The reorientation of an object could have a dramatic effect on the content of the final sound, whereas the spread pan-pot designs of Section 6.3.1 cannot give control over the sound content in this way. For example, consider the O-format signal for an object emitting very different sounds in opposite directions. As the performer rotates this object the sound changes back and forth. This direct, yet natural, sound control cannot be produced with models that radiate equally in all directions. The ability to control the sound as well as the localization in a natural way offers many more musical possibilities.

Several point radiators can be grouped to move as one object. The summed image remains true B-format, but with directions encoded over an arc. Surface radiation can be modelled by biasing radiation away from the surface (**Figure 6.9**).

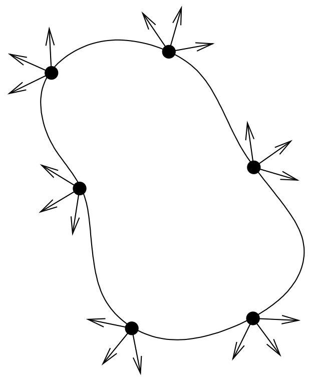


Figure 6.9: Extended surface radiator using composite point radiators.

This scheme provides an efficient method for modelling acoustic objects, without resorting to pseudo physical techniques like dominance. There is enough detail and structure to yield strong high-level cues about the movement and orientation of the object.

6.6 Soundfield Processing

Rotation and dominance have already been applied to the design of spreaders in 2 dimensions, but they are also useful in their own right for manipulating general soundfields. Details of calculation and implementation are presented here.

6.6.1 Rotation

There is 1 degree of rotation freedom in 2 dimensional space and 3 degrees in 3 dimensions. The space of orientations, is not immediately intuitive, but can be aided by parametrisation into Euler angles. These can conveniently be given in terms of the listener's head rotation which would achieve the same transformation on the soundfield: first an *azimuth* rotation about the vertical axis, then *elevation* in the vertical plane, and finally *twist* about the axis of vision.

To clarify the calculation of rotations the following is a precise definition in terms of the azimuth, elevation and twist. In matrix form, rotations about the X,Y and Z axis of angle θ acting on $\begin{pmatrix} \mathbf{W} \\ \mathbf{X} \\ \mathbf{Y} \\ \mathbf{Z} \end{pmatrix}$ are:

$$\mathbf{R_X} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & s \\ 0 & 0 & -s & c \end{pmatrix}, \mathbf{R_Y} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & 0 & s \\ 0 & 0 & 1 & 0 \\ 0 & -s & 0 & c \end{pmatrix}, \mathbf{R_Z} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & s & 0 \\ 0 & -s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.29}$$

Where $c = \cos\theta, s = \sin\theta$.

In the Ambisonic literature, for instance [Ger80, GB92], the usual orientation for the axis relative to a listener facing the 'front' is: X-forward, Y-left, Z-up. One way to control the orientation of the soundfield to minimize operator confusion is to control the orientation of a view which is then mapped to the front. The view can be described by Euler angles, which are more familiarly called *azimuth*, for rotation of the viewing plane about the z-axis, *elevation*, for the angle of the direction of viewing with the horizontal, and *twist* for rotation of view about the direction of viewing. The rotation which maps the view described by azimuth *az*, elevation *el* and twist *tw* to the front is:

$$\mathbf{Y_{-tw} X_{-el} Z_{-az}} \tag{6.30}$$

Note that the precise order of component operations is really only important to someone controlling the rotation, to help them understand what they are doing. The term *front* is used merely to refer to that direction in which the azimuth and elevation are zero, and does not have any special audio significance.

Calculation

The general rotation has the form:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & a & b & c \\ 0 & d & e & f \\ 0 & g & h & i \end{pmatrix} \tag{6.31}$$

Thus the number of multiplications using this matrix to transform a B-format column vector is 9. If the rotation is calculated by applying the component rotations successively the total number of multiplications is $3 \times 4 = 12$. The single matrix operation is therefore preferred if the coefficients do not need recalculating exactly very often. When updating coefficients $a..i$, calculation can be reduced by grouping and reusing terms (See appendix for details).

6.6.2 Dominance

The set of dominance transformations in any direction includes all the rotations, and the general form of the matrix has no fixed 1s or 0s. This significantly increases the overall computational burden. Dominance outside the horizontal plane is of little practical use in most Ambisonic speaker setups, and in any case, control is over complicated. Restricted to the plane, dominance requires two parameters; direction and strength of dominance. Dominance in the X direction can be parametrised by λ in a computationally efficient way described by Gerzon,[GB92]. In matrix form this is:

$$\begin{pmatrix} a & \frac{1}{\sqrt{2}}b & 0 & 0 \\ \sqrt{2}b & a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.32}$$

Where $\quad a = \frac{1}{2}(\lambda + 1/\lambda)\,, \qquad b = \frac{1}{2}(\lambda - 1/\lambda)$

B-format signals directed from the positive X axis transform as follows:

$$\begin{pmatrix} \frac{1}{\sqrt{2}} \\ 1 \\ 0 \\ 0 \end{pmatrix} \quad \rightarrow \quad \lambda \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 1 \\ 0 \\ 0 \end{pmatrix} \tag{6.33}$$

Gains in other directions are less than λ. Maximum dominance occurs in the limit $\lambda \rightarrow \infty$. It is therefore useful to normalize the gain of the transformation by applying an additional factor of $1/\lambda$.

Conjugating X-dominance with rotation in the plane (about the Z axis) gives a useful general expression for dominance in a plane:

$$\mathbf{D}_{\theta,\lambda} = \mathbf{R}_{\mathbf{Z},\theta}\mathbf{D}_{\mathbf{X},\lambda}\mathbf{R}_{\mathbf{Z},-\theta} \tag{6.34}$$

Where θ is the direction of the dominance $\mathbf{D}_{\theta,\lambda}$.

Calculation

The general form of $\mathbf{D}_{\theta,\lambda}$ is

$$\begin{pmatrix} a & b & c & 0 \\ d & e & f & 0 \\ g & h & i & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.35}$$

The situation is the same as for 3D rotation: single matrix calculation is preferred.

6.7 Soundfield Delay Effects

6.7.1 A Flexible Feedback Delay Network

Feedback delay structures are widely used in application to mono signals to create a variety of interesting effects such as echos, reflections, reverberation, chorusing. If the same structures are applied to B-format (4 channel) signals then spatial information is also recirculated. Thus if the input consists of a single encoded source changing direction over time, for example from a pan-pot device, then the output will in general consist of a composition of different directions from different times. The method therefore provides a simple means to enrich the spatial content of simple B-format signals. A flexible and computationally inexpensive recirculating structure is shown in **Figure 6.10**.

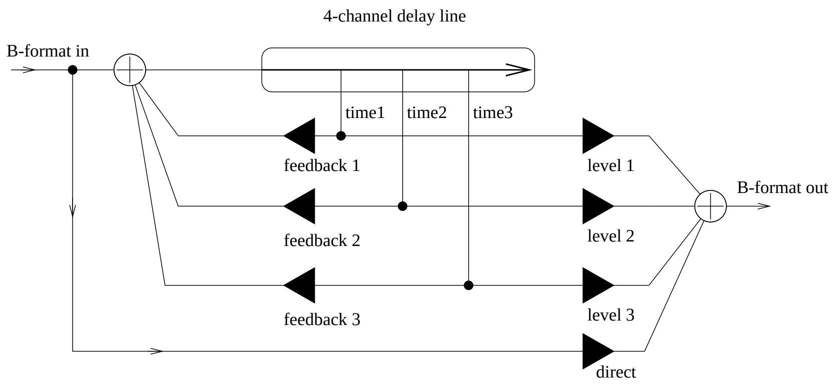


Figure 6.10: A structure for delayed feedback of B-format.

By varying the feedback times and gains a variety of interesting rhythmic effects can be generated.

6.7.2 Early Reflections

The quality of early reflections can greatly aid the localization of a source both in direction and distance. The gain of a reflection is related to its travelled distance and therefore time delay by a $1/r^k$ function where $k = 1$ for open space. Only 2 reflections are needed after hearing the delayed direct signal, to determine the distance of the source, [Ger95]. In practice the perception of distance is strengthened by reception of reflections up to about 40ms after the direct signal.

Reflections with FIR Filtering

Figure 6.11 shows a scheme for creating reflections with the correct gains using an FIR structure.

Creating the direct signal delay separately means that several sources set at different distances can use the same multi-tapped delay line for generating the reflections, increasing efficiency. Adding

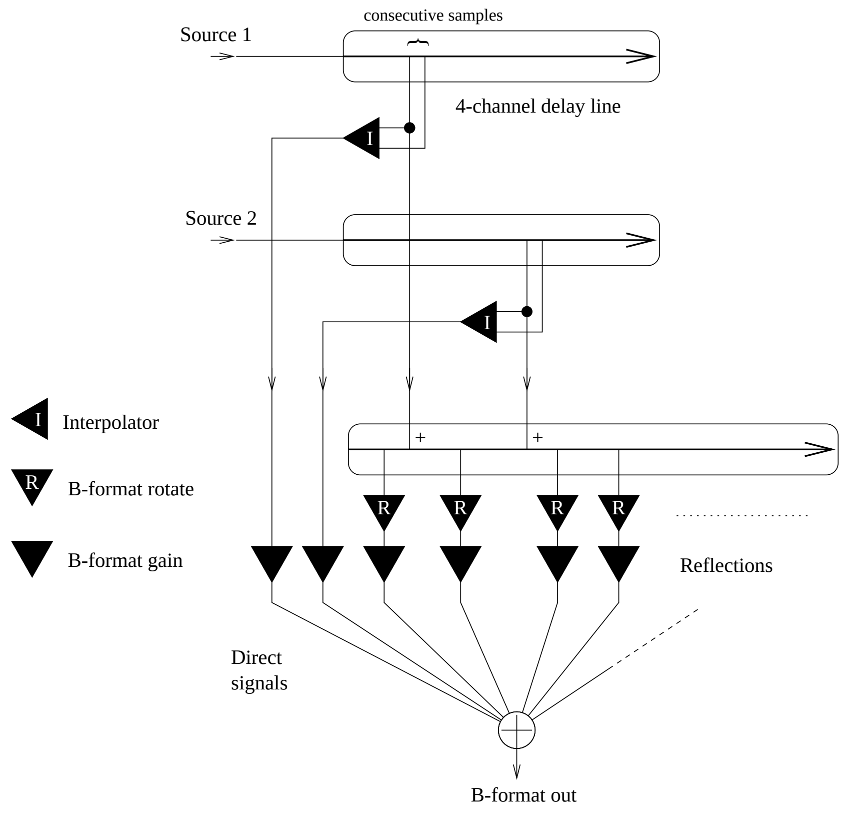


Figure 6.11: Simulating early reflections using an FIR structure.

the delayed direct signal to the corresponding part of the multi-tap delay line ensures that reflections are not produced for times earlier than the direct signal delay. The interpolators are used to produce fractional delays, so that continuous movement of the listener or source can be simulated.[4]

The times of the reflections are fixed. Using soundfield rotations the reflections can be kept at fixed angles relative to the source, which would ideally model a spherical room, but also be suitable for a source close to wall a in a large room. The reverberant tail is completely missing, but this does not matter too much if the playback room has a reverberant tail of its own. The synthesized early reflections override the room's early reflections, then the room's reverberant tail takes over.

Reflections with IIR Filtering

The structure in **Figure 6.10** can be used to simulate the first two early reflections by setting the gains consistently. The following reflections produced by feedback do not obey the $1/r^k$ criteria, but this does not totally invalidate the effect. The loss in accuracy of the IIR technique is made up for by its efficiency in generating reflections. Reflections for several sources can be produced simply by feeding the IIR network with the sum of the B-format encoded sources.

Spatialisation of reflections can be achieved by adding soundfield rotations in the feedback paths. In

[4]This simulation does not reproduce the correct Doppler effect, but it may be useful as an approximation. Full Doppler simulation requires interpolated input from the source, to simulate source movement, and interpolated output from each of the direct and reflected outputs to simulate listener movement.

Figure 6.12 rotations have been added to two of the feedback paths.

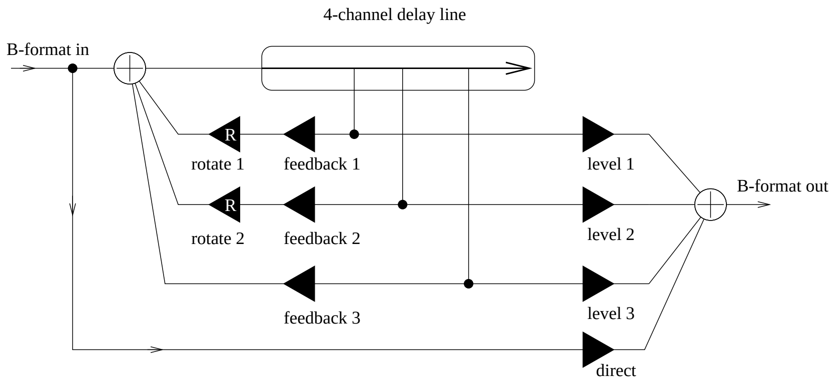


Figure 6.12: Simulation of spatialised reflections.

The effect of using small feedback rotations is to create a 'halo' of reflections directed about the source. Setting the rotation angles to be relatively prime gives a smoother directional spread of reflections. At greater delay times, feedback rotation can be used to create unusual 'creative' effects.

6.8 Distance EQ

Atmospheric frequency attenuation is described by an exponential decrease in response with distance, decay being greatest at high frequencies. The frequency response at any distance can be approximated by a simple 1-pole filter. Huopanierni, [HSK], discusses the use of least square fitting for evaluating filter coefficients. Here a simpler method is presented, which fixes the filter coefficients by ensuring the response is correct at one frequency.

If the filter is defined by

$$y_n = \alpha x_n + \beta y_{n-1} \tag{6.36}$$

Then the frequency response is

$$H_w = \frac{|\alpha|}{|1 - \beta e^{-jw}|} \tag{6.37}$$

To normalize the gain to 1 at $w = 0$ set $\alpha = 1 - \beta$. Let $e^{-\eta_w x}$ be the atmospheric attenuation at frequency w and distance x. Then the filter is fixed by evaluating the filter at one frequency, w_f:

$$e^{-\eta_{w_f} x} \quad = \quad \frac{|\alpha|}{|1 - \beta e^{-jw_f}|} \tag{6.38}$$

Using $w_f = \pi$ ensures attenuation at the highest frequencies is approximated most accurately:

$$e^{-\eta_\pi x} = \frac{1-\beta}{1+\beta} \tag{6.39}$$

$$\beta = \frac{1-e^{-\eta_\pi x}}{1+e^{-\eta_\pi x}} \tag{6.40}$$

6.9 Keyboard Control

6.9.1 Object Movement

The development, in the preceding sections, of practical ways for encoding distance cues into B-format signals naturally demands that *control devices* should allow distance control in a suitable way. The old analog devices used joysticks to pan source signals, but the central region of joystick movement produced a poorly defined output. Joystick control can now be extended by making the radial extension of the joystick control distance cueing. Several MIDI keyboards have joysticks built in. [5]

One joystick can only control one sound at a time, and one pair of hands can only control two joysticks. For live performance we should like to find a means for a performer to control several sources at once. In the spirit of Chapter 5, one way of doing this is to exploit the polyphonic control qualities of the MIDI keyboard keys. Two methods of key control are implemented in LAmb. As in Chapter 5, *key* here refers always to a physical keyboard key rather than a key signature.

- *Position Key Mode* Each source is assigned an octave. When a *position key* is hit the source moves towards a position assigned to that key, with a speed that increases with the strength of the hit. The start and end of travel can be smoothed off by applying a simple low pass filter to position that increases in strength near the start and end points. Re-hitting a key before the destination is reached causes a change in speed, so by repeatedly hitting the key the source speed can be controlled along its path. By frequently switching between several destinations, the source can be moved along paths other than those directly between pairs of destinations.

  Arrangements of positions can be programmed separately for each piece. With a hexagonal speaker rig an arrangement which aids coordination and visualization of positions consists of a position at each point of a hexagon aligned with the rig. LAmb contains such a set of positions as a preset. The positions are assigned to keys in an intelligent way, to make them easier to use.

- *Cursor Key Mode* Each source is assigned an octave and some of the keys within each octave act like cursor keys to move the source up/down, left/right or back/forwards. Hitting the key harder causes a bigger ”jump”. The keys are arranged to make it easy to learn their function (for details see Appendix C). For the movements to be smooth the increments are filtered with

[5] For example the Korg M/T series and the Yamaha SY series. The latter are very good quality. To allow more precise control, the joysticks can be extended with tubes such as empty pen barrels.

a simple 1-pole low pass filter. If a key is re-triggered before an increment is finished, the new increment adds to the previous one, so rapid hitting of a key results in rapid movement of the source.

So that both methods of control can be used in the same piece, a special *control key* is available within each octave for toggling between methods. For example, after using the cursor keys the player can switch to position keys and quickly relocate the source to a known position. This would be difficult to do in cursor mode alone. On the other hand the range of the cursor mode is unlimited, whereas position mode is limited by the range of its set of position keys. Note two sounds could each be controlled by a different method at the same time.

Cursor vs Position Keys

Cursor keys are perhaps the most obvious way of applying keyboard control, but by far the most popular method is position key mode. This may be explained by observing that position key mode does not demand the player to think strictly in terms of fixed axes. Rather the player can link the pattern of spatial sound directly with the pattern of finger movement as if playing a traditional instrument. Cursors require you to know where you are before you can predict the effect of playing, which complicates playing technique.

6.9.2 Soundfield Sample Keys

As well as diffusing recorded and live audio, LAmb allows the player to trigger B-format samples from a range of keys. The playback is basic. The samples can be velocity sensitive if desired. The main intention of the sample player is to break the tape element of a live piece into sections so that it can be synchronized with the main live elements such as acoustic players. Another approach is to use the sample player as an instrument, and take advantage of the rich interactions which can occur between different soundfield recordings.

6.9.3 Soundfield Rotation Keys

Rotation can be controlled with keys operating in cursor mode. More useful is the joystick control described next.

6.9.4 Joystick Control

If a keyboard has a joystick it can be used to control position or rotation. Each position control octave and the rotation section has a control key for ”grabbing” usage of the joystick, making it practical to reassign the joystick over the course of a piece. A third degree of freedom, Z or twist, can be controlled with the modulation wheel.

6.10 Desktop Tools

6.10.1 Evolution

The need for a graphical interface arose initially as a convenient way for setting the behaviour of the MIDI keyboard and configuring the audio and MIDI I/O. Setup file operations were included to facilitate rapid loading of stored configurations.

Interactive Composition

With the graphical interface in place, it was natural to include new controls for directly controlling positions and directions, so that LAmb could be controlled entirely from the desktop using the mouse. This enabled people to spatialise mono sources one at a time and record the output onto tape, for later mixing. This is interactive composition, not performance, but it proved popular, partly because there was much more opportunity to use LAmb in an office rather than a concert environment. To make the process easier a direct-to-disk recording and playback utility was added, including simple means for synchronizing the playback of a reference track with the recorded track.

6.10.2 Object Control Parameters

LAmb implements the processes described in Sections 6.3 and 6.4.2. The controls have been carefully chosen to give flexibility with minimum complexity.

- *Object Size* governs the scale of the γ curve shown in **Figure 6.2**.
- *Bumpiness* effects the scale of the β curve. For a realistic object bumpiness and object size should be opposed, for example low object size with high bumpiness. Other combinations produce a variety of unusual effects. For instance reducing bumpiness and object size creates an effect analogous to audio compression but in space. The listener is aware of the distance relationships due to the γ variation but without the normal amplitude emphasis on close objects.

  The β curve also contains information about the geometry of the environment and how sound spreads through it. For more control this could have been parametrised by another slider, but this was considered too confusing. The name *bumpiness* was deliberately chosen to make the slider more intuitive. Higher settings simulate a space where sound energy spreads out rapidly, lower settings approach the case of a corridor.
- *HF attenuation* controls the scale or equivalently, the rate of attenuation.
- *W adjust* This might be used to tweak the W strength of a channel according to sound content, in order to optimize localization. Alternatively it could be controlled continuously for unusual effects.

- *Z adjust* There is a trade off between clarity of horizontal and vertical direction perception in Ambisonics. Reducing the Z strength favours horizontal perception.

- *Spread Scale* Relates to the size of an object that is modelled by the frequency spreading process described in Section 6.4.2. The spread effect can be used with or without the Object Size setting. The effect is clearest off-centre, but central movement is ok.

- *Spread Max* Controls the spreader delay time. Larger times cause the spectrum to be wrapped across the spreading stage more times.

- *Jumpiness* This is an odd effect which causes the source to jump in steps, but without clicking. The jump can be relative to the current position or fixed to a grid, using the *Jump Snap* option.

- *Key Feel* The player can modify the MIDI keyboard velocities to suit the piece.

6.10.3 Soundfield Delay

The sliders correspond directly to the parameters in **Figure 6.12**. The delay times are multiplied by a master time control, *Time Scale*. This allows the ratio between delay times and hence the echo patterns to be preserved while speeding or slowing the whole pattern.

6.10.4 Dominance

2-Dimensional dominance is controlled from the desktop by moving crosshairs. The direction of the crosshairs relative to the centre gives the direction of dominance, and the distance from the centre determines the strength of dominance. Perimeter regions give maximum dominance.

6.10.5 Locate Listener

Locate Listener allows the player to move their listening point relative to the sources, and so create strong high-level correlated cues about listener movement. The same effect could be achieved by moving the sources simultaneously in a coordinated way, but this would be impractical.

This feature was not part of the original design plan and is implemented only in the graphical interface as a desktop tool. However, it would be well suited to joystick control, possibly indirectly via the control of a virtual craft, such as found in video games.

6.11 Program Structure

LAmb is written in C using the Xforms library for the graphical interface. *lt.c* contains the main audio code. **Figure 6.13** shows the main components. The audio rate processing is arranged in one compact block with no function calls. At intervals various ‘control rate’ functions are called from within the audio loop. Each control function can be given a different priority, to optimize the use of system resources. The control manager decides when function is due for its next call. If several functions are due, the manager makes them wait in turn to limit the erosion of the audio output buffer, and hence the risk of an output glitch.

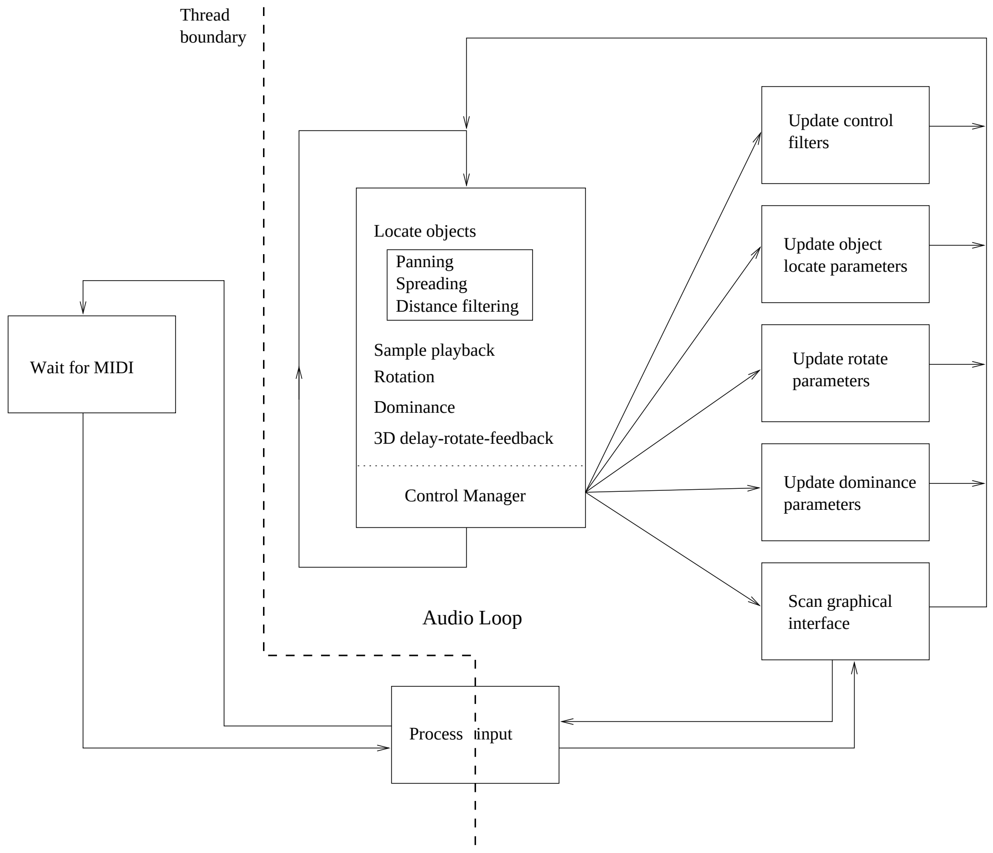


Figure 6.13: LAmb software components.

6.11.1 Control Filter

kfilter() Is a general purpose control filter, designed for upsampling raw input control data to a higher uniform control rate. It acts on an array of variable *target[]* to produce filtered output *filter[]*. The target variables include all the MIDI keyboard input and the slider input from the performance page. Each target has associated filter variables describing the details of its filtering. If a target is storing a coordinate for the last position key pressed for one of the sources, the filter will be set to give a speed of movement dependent on key velocity. Alternatively, if the target holds a joystick coordinate, the filter is set to act rapidly always.

6.11.2 Mixing

The object mixing stage in the audio loop is subject to rapid changes in mixing levels when sources move across the central region. It is therefore important to ensure the mixing is carried out as smoothly as possible. The correct method to ensure this is to upsample the mixing controls to audio rate, with proper band limitation to prevent aliasing. In the case of LAmb this proves awkward and costly. The alternative method implemented uses zero crossing detection to wait for near zero-signals for mixer parameter changes. This method is enhanced by applying a simple DC removal filter to the signal input, which ensures small signals will zero-cross frequently. A spin-off from using zero crossing is that large jumps in position can be made without ‘clicks’. This fact is used to implement the *jumpiness* feature in the performance page.

The mixing for the soundfield rotation and dominance sections is not so critical, and the control rate mixing parameters can be used directly.

6.11.3 The Graphical Interface

The Xforms interface is called regularly by the control manager with the call *fl_check_forms()*. Any user input will trigger a call to a call-back function in *f_cb_c*. Many of the parameters are processed via *control_process and slow_control_process* in lt.c. This arrangement unifies the graphical and MIDI interfaces, allowing better consistency and quicker software updates.

6.11.4 MIDI input

Running along side the main loop, a shared process, *midi_input* waits for MIDI input and sets up the appropriate changes of state in the shared memory. For example, if a position key is hit the corresponding target and its filter parameters must be updated.

Every input on the graphical user interface, filtered or unfiltered, can be set via MIDI. This is so that another program or device can communicate via an internal or external MIDI link.

6.11.5 Efficiency Features

Slider Resolution

The resolution of the sliders is limited to 64. This greatly reduces call-back function overhead. Filtering by kfilter ensures the controls all give a smooth response.

Automatic Resource Control

The different sections in the performance page do not consume any computation time when not being used. In *Locate Object* a channel is off when its volume is at zero.

X-Windows Considerations

It is important that all the performance controls are collected into one X-Window. If they were separated, mouse movement between performance windows would incur a high overhead, increasing the minimum latency which can be maintained.

6.12 The Hexagon Performance Space

Parallel to use of Ambisonics in concerts is the practice and exploration of Ambisonic techniques out of concerts. To this end the author and a graduate music student from York Music Dept, Tim Ward, set about constructing a permanent concert size Ambisonic facility. An excellent opportunity for doing this was provided by a hexagonal rehearsal room approximately 15m across. The ceiling is supported by 6 girders which reach from the central apex to the upper wall corners. See **Figure 6.14**.

6.12.1 Speaker Array

The speakers used were Wharfdale Diamond Vs. These have excellent bass characteristics for their compact size. The overhead speakers were fixed to the girders using specially made clamps. The lower speakers are placed on the floor and are fed from wall sockets. They are removed when not used. Speaker feeds were provided by an unused lighting network, which converges to a box in a small adjoining room. Sockets were arranged on the box in a hexagon pattern to give an easy reference to their destination. Two 8 channel amplifier crates have been used to drive the 12 speakers (The crates are portable and are used in concerts elsewhere).

6.12.2 Decoder Electronics

A programmable Ambisonic decoder is available within the music department, but since this is frequently reconfigured and in use, it was decided to design a simpler decoder dedicated to the Hexagon which would be reconfigurable only via analog controls. The following useful features were included:

- *X,Y,Z controls* For each speaker three rotary potentiometers are available for setting the three decoding matrix coefficients X,Y,Z. The mid point of each pot gives a zero coefficient. The potentiometers are conveniently placed next to the corresponding speaker sends on the front panel.
- *W control* A global W gain control applies the same W component to all the speaker feeds. This is easily adjustable with a dial. In practice different B-format recordings may use different W weightings. A 3D recording played back on a 2D rig may require W adjustment.

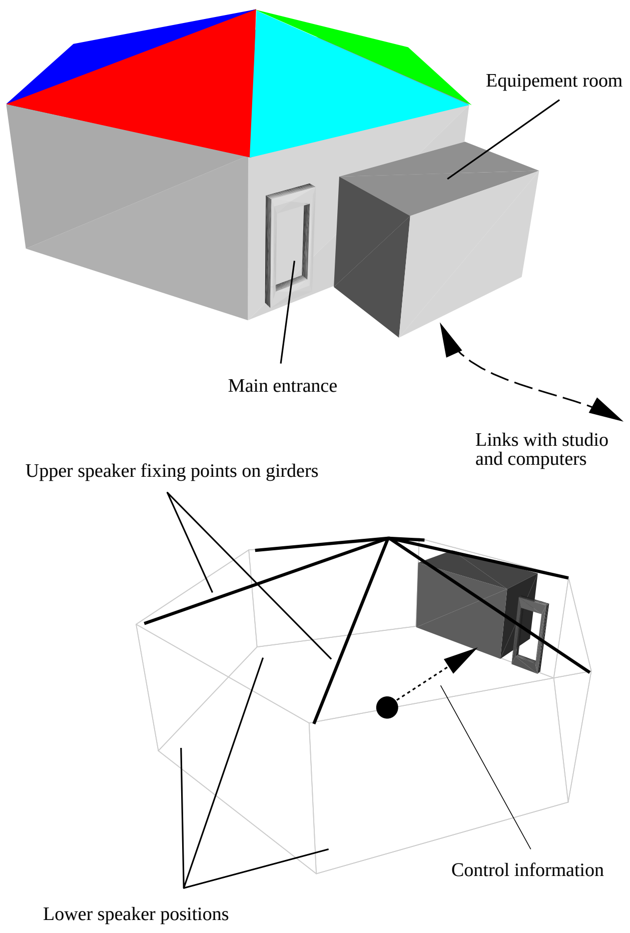


Figure 6.14: The Hexagon performance space

- *Upper speaker gain control* To compensate for the audience being beneath the geometric centre of a typical 3D rig, additional compensating gain is available for all the upper speakers.
- *Test oscillator* To help in setting the X,Y,Z levels and to diagnose problems with the decoder and the speaker rig, an asymmetric oscillator is mounted internally and can be switched onto any of the the W,X,Y,Z input channels. Special meter connection points are available from the front panel, from which the output signals can be measured. Amplitude regulation ensures measurements made do not drift.

6.13 Summary

The techniques presented expand on previous ambisonic diffusion by supplying a wider range of signal processing methods and an alternative polyphonic control interface strategy which only requires readily available components. It has proven a popular instrument and tool at York University Music Department and elsewhere worldwide. In 1997 it was awarded first prize in the real-time category of the *Bourges International Computer Music Competition*.

Practical Considerations

The main barrier to the application of systems such as LAmb in concerts appear from experience to be purely practical. The computer and monitor are not designed to be frequently moved around nor is it easy to do so. Before long, however, it is very likely with current rates of development that convenient ‘lap-top’ computers will offer the same audio facility at low cost.

Live Electronics vs Interactive Composition

A computer-based system which is designed for live performance will inevitably be used primarily for desktop composition. This is partly explained by the lack of mobility of current hardware discussed in the previous paragraph. More important is the ease with which computer systems can be put to work in a desktop environment with rapid access to filing systems and other applications. These same factors are having a massive impact in other areas of art and media, including the recording of traditional performance instruments. It is a cultural phenomenon. However, this should not detract from the value of computers in performance. The designer must consciously avoid thinking in desktop terms.

Realism vs Impressionism

A theme that arose frequently in the development of different synthesis techniques, is the tension between complete realism and impressionism. This is especially acute in spatial audio, because the psychological processes by which we derive spatial information about the environment from sound are complex and deeply rooted. From the music technology stand point strong consideration must be given to both the realistic and abstract qualities when designing a performance instrument.

Developments

Possible future developments in this area include the generation of more complex objects with distributed radiation patterns, expanding on the O-format technique, and means to perform the orientation of such objects. The increase in complexity must, however, be balanced by keeping performance controls inuitive and *musical*. Although realistic cues are powerful tools, ultimately it is the final musical impression that is important.

Chapter 7

SidRat - Live Electronics using Csound

7.1 Introduction

Previous chapters have concentrated on the production of new *instruments* which can be applied in a variety of electroacoustic performance settings. Part of the reason for doing this was to react against the increasingly fractured and impermanent nature of modern *live electronics*. This chapter focuses on the techniques used in an original piece, *SidRat*[1], rather than instrument development. The purpose is to attempt to show how the broader concept of performance embodied in live electronics can be implemented efficiently and flexibly with common and portable tools, in particular using Csound. Here Csound is viewed as a general purpose processor for converting audio and control input from performer(s) into audio output, with possible dependence on elapsed time. SidRat is not an instrument which can be reused but it *is* a piece which can rapidly be recreated elsewhere with minimal worries about what audio processing platform should be used.

Examples of real-time Csound usage in the literature are rare, and tend to concentrate on copying the behaviour of commercial synthesizers. Generally useful characteristics of Csound have already been mentioned in Chapter 4, and are repeated here in reference to live electronics.

- *Portability* The Csound language is script based, and contains no machine dependent syntax. This means that functioning scripts can be run on any other machine running Csound provided only that the machine has enough audio I/O channels, MIDI input and speed, as required. Csound executables are readily available for a wide range of computers, and installations are very common amongst computers in music departments. For live electronics high portability is essential, otherwise considerable work is required to re-engineer the 'electronics' using the

[1]First performed in the Hexagon at York University Music Dept, May 1997

available equipment. [2] The situation is analogous to sheet music read by musicians in live performance. It would be hopeless if music had to be translated for every new performance.

- *Fast Development* The script contains a library of audio processing operations which eliminate much of the need for developing more low level operations. The script compiles rapidly at run time, enabling a rapid development cycle. This is also useful at the rehearsal stage, where changes can be made quickly without wasting performers' time. Slight changes to accommodate available performance controllers can be easily made.
- *Efficiency* Csound is reasonably efficient, although not optimal. Graphics based systems tend to suffer from additional overheads.

7.2 Control Input

Csound offers two methods of real-time input. *in* is used to read from the audio input buffer. When used with the normal *out* command, audio can be processed in real-time.

When MIDI input is enabled with the *-M* option, note-on messages trigger an instance of instrument *n*, where *n* is the channel number of the message. A subsequent note-off message on the same channel terminates the last instrument instance on that channel. This operation is specifically designed for processing keyboard input. Typically, performance in an electroacoustic piece will generate a mixture of switched and continuous input. How should continuous input be conveyed independently to the central Csound processes? An initial note-on message must be sent to switch on a MIDI instrument instance. Thereafter pitchbend or controller messages can be read within the instrument using *pchbend* and *midictrl* commands. The initial note-on could be sent by the performance device itself, or more conveniently introduced by another system process. The latter method is used in SidRat. The extra process is *mdcontrol*. See the appendix for the C source. It receives external MIDI, adding an initial note-on, and transmits the result to the standard output.[3] To make Csound read the mdcontrol output, a simple pipe is used from the command line: *mdcontrol | csound -Mstdin* ... The piece SidRat calls for a footpedal and a footswitch, which transmit MIDI controller information, continuous and switched respectively. The simplest method of implementing the hardware is to use a MIDI keyboard with sustain and volume pedal inputs and route the MIDI output to Csound. The keyboard is hidden from view, and the actual keys are not touched. There are several examples of MIDI keyboards which send controller messages from a continuous voltage input, including the Korg M/T and Yamaha SY series.

[2]A parallel problem is encountered with the performance of old works for live electronics which originally used analog devices. These often specified the use of tape delay units which are no longer available. To recreate such a piece using a computer is time consuming, especially as the subtleties of many analog devices require detailed digital modelling.

[3]An accidental advantage to using *mdcontrol* is that it uses the SGI *md* library for external MIDI input, which is more reliable than the proprietary drivers installed in the standard Irix Csound binary.

7.3 Using a Score in Real-time

An important aspect to many works for live electronics in the 1960's and 1970's were parts in the written score directing human *operators* to control audio processing equipment. A typical example is *Solo* by Stockhausen, in which sound from a solo acoustic performer is processed with a filter and tape-delay network then broadcast on a quadraphonic sound system. Operators control the tape-head positions, feedback gains, output levels and filter settings according to a score.

Csound provides a detailed score system, originally intended for the crafting of electroacoustic music by a repeated cycle of program-compile-listen. However, there is nothing to prevent the score being used in real-time mode.[4] In this case the score is no longer suitable for describing soundevents which take longer than real-time to compute. If a soundevent has no interaction with the player it can be pre-composed and played back as a sample. A more important use for the score is to control real-time processes acting on the performance input, in a manner analogous to human operators reading a score. This can be achieved by switching on instruments which set global variables. The variables are then picked up by the main processes which run within instruments that are permanently on. A useful method for controlling global parameters is illustrated by the following example adapted from *sidrat.orc*:

```
     instr 20
gk1  linseg i(gk1), p3, p4
     endin
```

p4 specifies the new target value of global control variable *gk1*, and *p3* the time to move in a line to the target. The initial value of the variable is read with *i(gk1)* . This ensures that the value of *gk*1 following the instrument varies continuously, without concern needed for the exact synchronisation of a sequence of *instr 20* invocations in the score. Since only a target and a duration are required, and not an initial value as well, the score is very easy to write and alter.

7.4 SidRat

7.4.1 Overview

SidRat is piece for solo percussion and live electronics - Ambisonic rig, Soundfield microphone, footpedal, footswitch, and Csound running on a machine with 4-channel audio I/O and MIDI input. See the Appendix for full performance instructions and code listing and refer to **Figure 7.1** for the equipment layout and connection.

The player sits in the middle of the concert room surrounded by the audience, inside the speaker

[4]However, one limitation to real-time score usage in Csound is that the real-time MIDI instruments, numbers 1-16, cannot be triggered from the score. This appears to be an oversight or bug, as there is no essential reason.

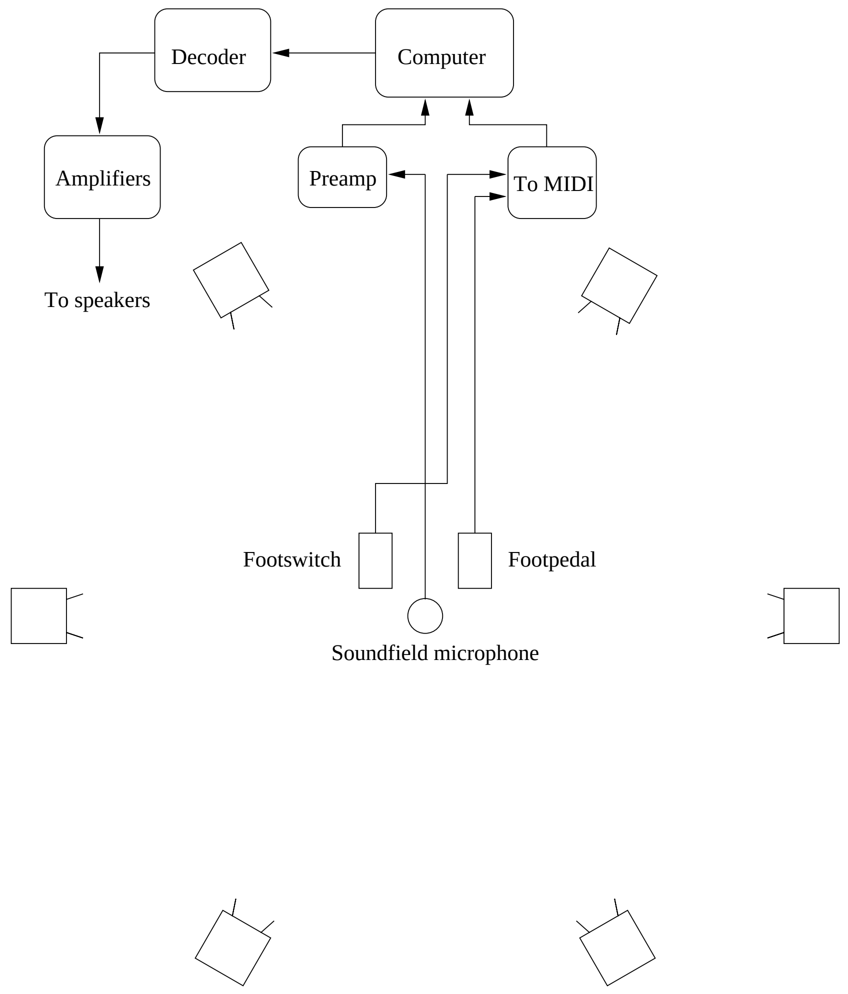


Figure 7.1: Equipment layout for SidRat.

rig. The percussion is varied and must include something capable of a short sharp percussive sound such as a hollow wood block. A Soundfield microphone is hung from a stand in the centre of the percussive set, so that it can receive percussive sound from different directions either from fixed percussion or smaller hand-held percussion that can be moved around the microphone. The output via a preamplifier feeds directly to the computer audio input. Using the Soundfield microphone is a novel means for controlling spatial sound. Rather than using an indirect method of controlling direction, such as a joystick, the percussionist controls direction playing an object in the appropriate position about the microphone.

At the player's feet are the footswitch and footpedal. These send MIDI control messages, via a control unit such as the keyboards discussed above, to the computer. A clock is also placed in front of the player, as a guide in some parts of the score.

The First Movement

The piece is divided into two movements, indicated in the Csound script by a section, s, division. The first movement begins with a pre-composed sample played back from the score. The player gives one hit synchronised with the end of the sample. **Track 15** is recording of the first half of the first movement, omitting the percussion part which follows the initial hit. Stereo recording unfortunately destroys the Ambisonic spatial information. The hit is sampled and stored. Over the remainder of the movement the sample is manipulated and processed to produce a spatialised B-format sound track of varying complexity, which forms the basis for a dialogue with the percussionist. The processed part is similar in each performance, but varies according to the sound of the hit and its exact timing. Thus in rehearsal the player can become accustomed to the processed part, while still being acutely aware how crucial the one initial hit is. The first movement ends with the processed sound being joined with the second pre-composed sample.

The Second Movement

The second movement begins as the first tails away to silence. **Track 16** is a complete recreation of the second movement using synthesised percussion instead of real percussion. Again, the Ambisonic spatial component is lost in the stereo recording. The player now has much more direct control over the speaker sound. Using the footpedals, the input and feedback level in a delay-feedback process are controlled. The player can also control the spatialisation of the sound by playing percussion in the chosen direction relative to the the Soundfield microphone. The original direction of play will be echoed in a delayed and sometimes transformed form on the speaker array. The score still maintains some control over the delay times within the process however. This becomes very apparent towards the end of the second movement around the 130s mark, where the score is producing rapid and exaggerated modulations. The second movement ends with the third pre-composed sample.

7.4.2 Instrument Example I

The main instrument in movement one, *instr 30*, performs the initial task of capturing the first hit, and then the subsequent processing of the sample (See **Figure 7.2**).

The first delay captures the hit while the input gate is briefly open. The feedback is unity, ensuring the hit is recycled unchanged. The output consists of the hit repeated every second. This is encoded into B-format, with the direction changing slowly at a constant rate. The final delay-rotate-feedback network processes the hits to generate patterns of hits. From the score, special instruments are invoked to control global variables that in turn control the network.

7.4.3 Instrument Example II

The central instrument in the second movement, *instr 40*, feeds a B-format signal from the soundfield microphone to a feedback-delay network with one variable tap (see **Figure 7.3**). The tap time is controlled in segments from the score via a filter cascade that upsamples and then low-pass filters.

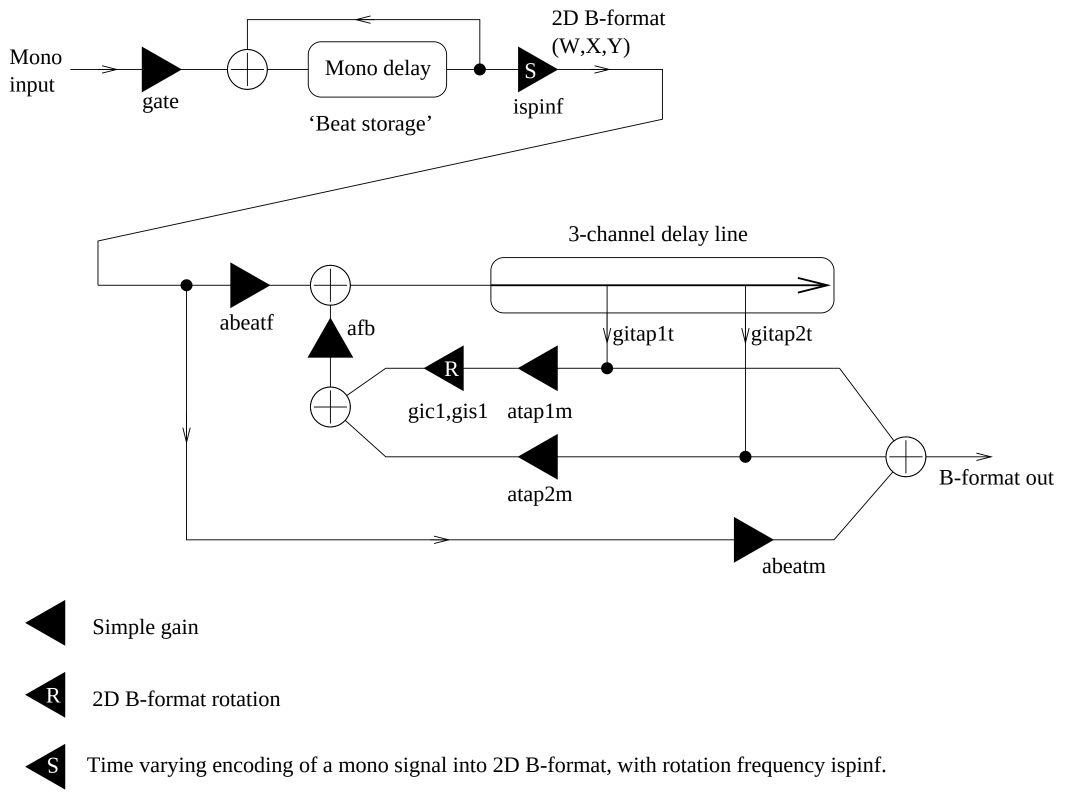


Figure 7.2: instr 30 from SidRat.

[5] The feedback is continously controlled by the footpedal, and the microphone input is gated by the footswitch. Although the gating action is meant to take effect immediately, the control rate variable is upsampled with *upsamp*, creating a small ramp which ensures the gated signal does not click. If the feedback is held near unity and the gate is open, the delay will accumulate material played, forming an ever more elaborate loop. If the gate is shut the loop will continue until the feedback is reduced. The player can choose to build a loop then fade it to a background level, and then build a new loop in the foreground.

An important refinement is shown in the figure which is necessary for helping the percussionist synchronise their playing with the instrument output. *Din* and *Dout* denote the delay times incurred by the audio input and output buffers respectively.[6] Suppose the delay marked *Din+Dout* were absent, and the input gate and feedback are on. If the percussionist made a hit at the same time as an event heard from the speaker, then the next time these events are heard via the delay, the hit will be heard *after* the other event, late by a time *Din+Dout*. This is because the original event was heard *Dout* seconds after it left the delay, and the player response took *Din* seconds to reach the delay input. Synchronisation is restored by using the *Din+Dout* delay indicated. Both events will then

[5]Without the filter, sudden changes in the gradient of the tap control occurring at the end of segments would cause sudden jumps of pitch in the delay tap output. Adding the smoothing filter makes the pitch changes more natural, similar to those produced by manually slowing and speeding up a vinyl record. The dynamics of pitch change are due to the inertia of the record and dynamics of the hand. The smoothing filter, a 1-pole low pass filter with a cutoff of 3 Hz, gives the impression of a mechanical process even though there are none.

[6]Some additional significant delay will act on the output due to the travel time from the speakers to the player. This is approximately 3 ms per metre.

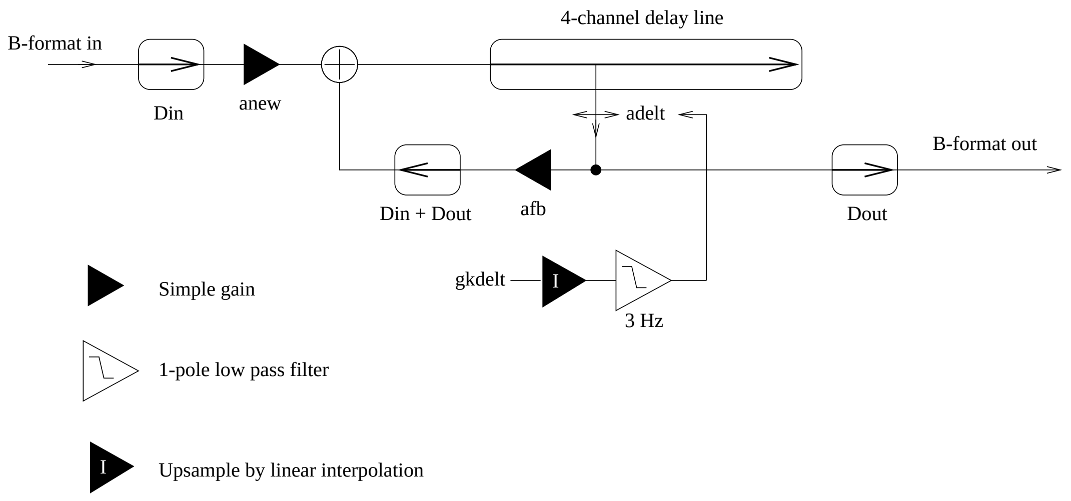


Figure 7.3: instr 40 from SidRat

reoccur at the same time.

7.5 Summary

Techniques

Techniques have been described for dealing with continuous input; using a score to control audio processing; managing the scoring of continuously changing variables, using delay for sample-and-hold; de-clicking gate controls; introducing B-format spatial transformations into delay-feedback structures; compensating for unavoidable system delays in a feedback process; processing control rate variables for driving delay tap times; and using a Soundfield microphone as a performance device for spatialising real objects.

Object-like Robustness

Sidrat shows how a complex piece for live electronics can be made portable. Traditionally such a piece would involve a number of discrete devices that have to be laboriously patched together. Tools such as MAX/MSP are neither free nor globally available or standardized across platforms. Nor are the score facilities as flexible as Csound. Reducing all the processing to a portable script gives SidRat a robust, object-like quality. This same quality is an important attribute of discrete acoustic and electronic instruments.

The Portability of Ambisonics

The Soundfield mic and Ambisonic speaker rig are the least portable components due to their scarcity. The 4-speaker Ambisonic rig format is the most portable since the decoded speaker feeds can be generated in Csound. A 6-speaker rig is a little less portable. Most of the decoding can be done in Csound, but the amplifiers must be networked to drive each other (See the end of *sidrat.orc*). More complex rigs require an external hardware decoder. The Soundfield microphone was used to demonstrate an unusual technique for controlling spatialisation in live performance. Similar, more portable, results could be achieved with a stereo microphone or an array of microphones.

Chapter 8

The CyberWhistle - A New Integrated Instrument

"**Cybernetics** *n.pl.* (usu. treated as sing.) the science of communications and automatic control systems in both machines and living things."

The Concise Oxford Dictionary.

8.1 Introduction

The physical interfaces which have been employed in the developments of previous chapters were readily available, off-the-shelf devices; MIDI keyboards, pedals and switches. Such standardization supports the reconstruction of instruments in different places, and the establishment of a community of practising musicians.[1]. However ingeniously common devices are used, they will always be constraints on the control signals generated, due to the methods of sensing, and mechanical properties of the instrument and the player. It is therefore worth considering the design of new controllers with different constraint properties. If it is desired to encourage the widespread use of the new controller then the construction must be practical, and the parts readily available.

The *CyberWhistle* was designed with such ideas in mind. It consists of a penny whistle containing sensor electronics whose data is relayed to computer via a MIDI link. A natural consequence of using an unusual controller is that conventional synthesis devices such as note-based MIDI synthesisers can no longer be integrated successfully. From the outset low level software synthesis was considered instead. As noted in previous chapters, software synthesis offers flexibility, portability and rapid maintenance. The only major drawback is that it can currently only be performed on cumbersome desktop computers and workstations, although this situation is likely to change.

[1]Software can be easily transported physically, but making it run on different machines or even two different machines of the same type is another matter entirely. See Chapter 7

The name ‘CyberWhistle’ was chosen to reflect at one level the merging of modern technology with an ancient object and on another the interaction between man and technology as a complex two way process. This contrasts with ‘hyperinstruments’ such as the *HyperCello*, [PG97, Lev94] where the original instrument is retained in its entire and form, and modern technology is ‘grafted’ on.

The path taken in designing the CyberWhistle was far from linear. Given the background of aesthetic and practical criteria, the CyberWhistle emerged as a good candidate. In the first part of the chapter elements of the CyberWhistle are presented in the light of these criteria. There follow sections on the hardware and software designs, with reference to recordings on the accompanying CD. Lastly there is a section devoted to enhancements and alternatives to the original design.

8.2 The Concept

8.2.1 The Penny Whistle

The penny whistle represents one of the simplest and oldest forms of wind instrument, consisting only of a cylindrical tube with six fingerholes, attached to a flue mouthpiece. It is thus inexpensive and easily maintained. The bore is fashioned from brass, nickel plated steel or occasionally black lacquered metal. Some players prefer brass because the fingers can then sense the slight surface texture. The player has a clear view of the fingers, which rest only on the upper part of the bore. This provides some measure of visual feedback, that may at least aid the memorisation of finger patterns. The range is nominally two octaves, the upper octave being overblown and much louder. A wide range of sizes are made, each appropriate to a limited range of keys.

In Ireland the whistle is still a central component of the continuing folk music tradition. Its success can largely be ascribed to great economy in construction combined with an expressive array of playing techniques which can be used to generate complex ornaments well suited to the folk music form. (Indeed, one could argue that the form is in part determined by the characteristics of the whistle). Breath and embouchure do not have the same critical role that they have in the clarinet or flute, although good breath control is required for vibrato/tremolo technique and tonguing is essential for shaping fast attacks and decays. Rather, it is finger control to which the whistle player’s attention is focused.

Shading

Shading is the technique of partially covering a finger hole to alter the pitch and to some degree the tone. The shading may be static in order to play chromatic notes not attainable with standard fingering, or dynamic to generate pitch slides. In dynamic shading the finger is moved continuously to or away from fingerhole, usually by pivoting the finger around the side. This can be done gradually in ballad style or rapidly with several finger shadings in combination to generate complex patterns of subtle pitch and tone modulation.

8.2.2 Validating Usage of the Whistle Form

Adopting the form of the whistle is certainly an interesting proposition, but at the outset it is helpful to present some arguments for pursuing this line rather than designing a completely original form.

Mimicking real acoustic instruments such as the penny whistle is a commercially attractive prospect. It may also help in understanding the physical processes in acoustic instruments. However, in the context of the current thesis the main concern is the design of new instruments. The proposal is that by cross-fertilizing the whistle interface with non-mimicking synthesis, a useful new instrument can be created, which retains some interesting general properties of the whistle interface as used in the original whistle. A general argument to support this proposal is that traditional instruments have evolved over considerable lengths of time. Even something as simple as a whistle represents a kind of optimization over many parameters. It seems likely that the control interface may be a local optimum in some sense even if we remove any acoustic design constraints. For example on a piano, the hammer acceleration-release action is in some way an optimal control mechanism which is largely independent of the sound generation mechanism. The particular control properties of the whistle are analyzed under the next heading.

Another important consideration is the familiarity of the interface. At a glance whistle players and, to a degree, listeners will be able to relate to the instrument. They can quickly form ideas of how they might use the instrument. Since many wind musicians have experience of whistles from their early teaching, there already exists a large potential following.

Finally, at the most practical level, the whistle is inexpensive yet attractive, and well suited to being instrumented with internal electronics.

8.2.3 Abstracting Whistle Control Methods

In the remainder of this section the control features of the penny whistle are examined for their general utility independent from the original sound generating mechanism. Considering the finger modulations as purely abstract parameters, shows that the player has control over an unusually large continuous parameter space. This is of interest in electroacoustic music, where the musical emphasis is frequently less directed towards pitched scales or discrete harmony, and more towards the continuous development of timbral characteristics. Furthermore, the whistle finger control is ideally suited to the use of physical modelling synthesis (PMS) for wind instruments, especially where shading is an important technique. PMS is capable of recreating the subtle dynamic characteristics found in natural sound. For the purposes of electroacoustic music a broader view may be taken by transforming realistic models or even constructing synthesis designs using general features extracted from realistic synthesis models[2]. Since electroacoustic music frequently explores the transformation of natural sound, transformation of realistic synthesis provides an interesting variation.

8.2.4 Switch Feedback vs Continuous Feedback

The survey of commercial and non-commercial controllers presented in Chapter 2 finds none that can measure the shading continuum which is being proposed here. It appears that nearly all current windcontroller keys consist only of switches. The Akai EWI has in, addition, pressure sensors under the keys. These sensors do not appear popular, partly because they make it awkward to rest the fingers. Simple switches make it impossible to closely approximate the feedback dynamics which occur when someone plays a traditional keyed wind instrument. Faster key movement in legato playing on an acoustic instrument results in faster tonal transition. Such transitions are not immediately obvious to the ear, but do contribute strongly to the overall authentication of the sound by the listener. Each transition effectively constitutes the attack section of the new note. If we consider feedback has a means for the player to better understand the current state of their instrument, an event occurring only at the end of a prolonged action, e.g. finger movement, is on its own, less helpful than a prolonged feedback event. The piano enjoys a prolonged force feedback event. Drumming on the other hand is reliant upon the regularity of a background rhythm to ensure accurate overall timing, since the prolonged feedback element is much less well defined, consisting of sight and arm / hand internal pressure feedback. The whistle finger control does not obviously offer prolonged force feedback for the fingers, to enforce the prolonged sonic feedback. However, players use the technique of deliberately lifting a finger from a hole gradually, so that the tip breaks contact first, followed by the rest. This provides a strong form of sensory feedback as the fingers have an especially high sensitivity and resolution for touch. The wide edges around the tone holes serve to enhance this. Overall result is precise control of shading, which is not possible simply by moving the finger vertically, or even sliding it horizontally. There are several other wind instruments such as the clarinet and some modern flutes, where shading can be applied, but few are so expressly well suited to the technique as the penny whistle. The traditional Japanese wind instrument, the *Shakuhachi*, provides excellent examples of shading technique. Though it has only four upper holes and one thumb hole, experienced players can generate the chromatic scale and a wide variety of colours. Unsurprisingly, the Japanese developers of the Yamaha VL synthesizer prided themselves on developing an impressive Shakuhachi physical model. The key element missing from this is an interface for shading, as no suitable controllers exist.

8.2.5 Mouth and Breath Control

In the penny whistle, the breath has two main functions; to control volume and to overblow to the first harmonic and possibly higher harmonics. Tonguing enables the player to start and stop breath with great precision. Building up pressure prior to the tongue release, gives some control

[2]The essential features are delay networks and non-linearities.

over fast note attacks: Breath is capable of more detailed and finely balanced control than each finger alone, although its use is coloured by the dynamics of the mouth-mouthpiece system, just as shading is constrained by the dynamics of fingers. Together breath and finger control make a powerful combination; the combinatorial qualities of finger control with the precise dynamics of breath. Unfortunately, a practical breath controller cannot offer the complexity of feedback present in many real wind instruments. In these the acoustic impedance of the mouthpiece changes as the resonant frequency changes, resulting in a wide-ranging resistance to blowing pressure. In addition to this, the player can sense lower frequency vibrations on the lips, and acoustic transients caused by dynamics within the instrument. For the penny whistle these vibrational feedback effects are minimal, and so the form of the whistle breath control can be taken on without much compromise. Conventional commercial wind controllers such as the Yamaha WX5/7/11 incorporate a fake reed. Such emulation is psychologically unsatisfactory because the use of the reed is not accompanied by the kinds of impedance and vibrational effects found in real reed instruments. The saxophone family, for instance, are greatly effected by mouth and throat resonance, which provide good players with a very great degree of additional control. On the other hand, the technique of singing into an instrument, or ‘growling’, may be justly implemented in the new whistle controller, because the main vibrational feedback comes from the player’s own vocal cords and not the instrument itself. The end of the chapter shall address some extensions of this kind.[3]

8.3 Electronic Development

Reasons have been given for constructing a musical interface in the form of a penny whistle, with the capability for measuring breath pressure and the extent to which each fingerhole is covered. Various methods for implementing the measurement will now be considered, while bearing the following desirable characteristics in mind

- *Compactness* The sensing devices must fit into the bore of the CyberWhistle, and preferably the supporting electronics too. External supporting electronics would increase the connection complexity and possibly complicate the design of sensitive analog circuits.
- *Power consumption* To run from small batteries for a convenient amount of time, consumption must be low, $<$20ma.

[3]In a hyperinstrument the player benefits from the same contact feedback present in the original instrument, but the naturally generated sound must be incorporated with the electronic sound. Instruments such as the electric guitar and electric piano radiate very little acoustic energy in comparison with the generated sound. These are cyberinstruments in the sense that acoustic sound sources have been transformed so that electric signals can be extracted directly. A large part of the success of these instruments is due not only to the inherent flexibility for producing a range of sounds, but also to the degree of contact feedback provided via the fingers. In the case of the Rhodes Piano the vibrations of each note pass into the keys. The evolution of a note, once hit, can be ‘heard’ by the fingers. One might consider, then, an ‘electric windinstrument’ in which low frequency vibrations could be sensed by the player, while higher frequencies remain locked in the resonator, and are amplified.

- *Reliability* The circuits must operate satisfactorily in the range of playing conditions. Some variation in behaviour is acceptable, just as it is with acoustic instruments.
- *Cost* If the instrument is to be reproduced it should use low cost, readily available components.

8.3.1 Finger Sensing

Capacitative Distance Transduction

In chapter 1 a great variety of historical methods for transducing position were introduced. The earliest and still highly relevant is capacitative sensing, employed in the Thérémin. If a whistle were fitted with such sensors then the distance range could be no more than the spacing of the holes, which is about 20mm. For greater ranges, there would be significant interference between neighbouring signals. This is not necessarily a problem as the original whistle only responds noticeably over 5mm. The interference might be acceptable, or even musically useful in the same way as Michael Waisvisz 'The Web', [Kre90, And94], transforms input movement into a number of string tensions each of which depends on all the inputs.

Paradiso, [PG97], clarifies the three basic modes by which capacitative sensing may work, and gives details of transmit-receive circuits with good noise rejection. *Transmit mode* is impractical because the players hand must be well connected to the oscillator source, by some kind of metal pad in contact with a thumb. The signal strength varies too much with thumb pressure. Also, the rest of the bore must be insulating, so a simple metal tube can no longer be used. *Conduct mode*, in which capacitance is varied by introducing an isolated conducting object, is not relevant because the finger is essentially earthed by connection to the rest of the body. *Shunt mode* can be used because the body is earthed.

To test this, a capacitative sensor was constructed consisting of transmitter attached to a circular metal ring 1 cm in diameter, and a central circular receiver (See **Figure 8.1**). The transmitter and receiver were insulated with blutak to prevent direct conduction via the finger. The figure shows the connection of a synchronous detector, to give optimum noise rejection. For the test however, the receiver was monitored directly with an oscilloscope. This clearly gives a good enough signal for finger control, without the need to employ the added circuit complexity of synchronous detection.

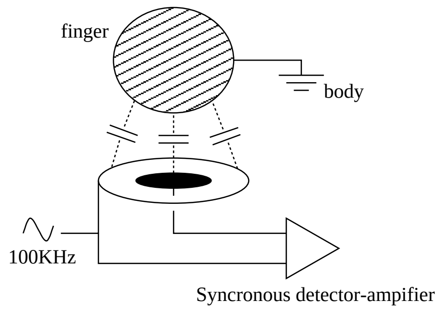


Figure 8.1: Capacitative sensing of finger proximity.

As the finger approaches the capacitor-sensor the conduction through it is reduced. The metal bore helps to isolate each finger hole from other fingers. The output appears sensitive to physical knocks, presumably because vibrations affect the capacitance between the transmitter and receiver. Setting the components in a more solid material should cure this. The drawbacks are that the resolution of the sensing voltage is heavily concentrated in the finger down position, and the circuitry required for generating a voltage suitable for linear sampling is awkward. Since separate detection circuitry is required for each finger, the total circuit complexity becomes unwieldy.

Ultrasonic Distance Transduction

For longer distances the inherent measurement delay of several ms is to be avoided because it adds significantly to the latency of the overall instrument. For short distances audio pulses cannot be made short enough to ensure enough accuracy.

Lightsensor Distance Transduction - LEDs

Infra red LEDs and receivers are sometimes used to determined distance by virtue of the inverse square law of spherical radiation. In the present situation neither device can be directly connected to the finger. An alternative strategy is to place a transmitter and receiver adjacently in a finger hole (**Figure 8.2**).

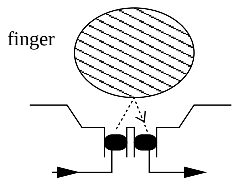


Figure 8.2: Finger proximity sensing using infrared reflection.

As a finger approaches the receiver detects an increasing amount of reflected light. If the light is infrared then it is largely immune to background ambient light (Using a pulsed source greater immunity could be achieved). This system would be subject to variations in the reflectivity of the finger surface, and the effect range is limited by the sharp increase in reflectance near the finger down position.

Yet another approach is to use a light receiver detecting natural light. As the finger approaches the light detected decreases. In this case the light received is subject to ambient light levels but can be compensated by using an ambient light sensor pointing in the same direction (**Figure 8.3**).

Standard photodiode and phototransistor receivers can be configured with a transconductance amplifier to produce a voltage nearly proportional to light intensity. Therefore, dividing the voltages from the compensating and finger sensors gives a measure of the finger coverage. For a given finger position this quotient remains substantially constant to background changes in the overall intensity of light sources present. Exact precision is not required here, only enough to make the instrument robust

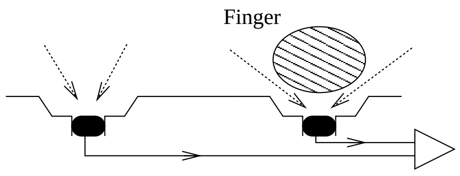


Figure 8.3: Finger proximity sensing using ambient light occlusion.

under different lighting conditions. Dividing voltages is not easy to do, even approximately. Since each finger position voltage must be sampled for digital processing, the most convenient method is to use a dividing ADC. The compensating voltage drives the voltage reference. Unfortunately, general purpose microcontrollers containing ADCs do not offer voltage references which can be taken to the lower voltage rail. The only alternative is to add an external ADC to the microcontroller, which greatly adds to the circuit complexity.

Lightsensor Distance Transduction - LDRs

Light dependent resistors (LDRs), also called *photocells*, closely approximate pure resistances. The low light sensitivity is much better than for semiconductors, but the switching times are relatively poor. The resistance exhibits a ‘cooling-heating’ effect, in which the maximum absolute rate of change of resistance is an increasing function of light level. **Figure 8.4** shows the response of an LDR to a sudden reduction then increase in light level. The precise shape of these changes depends on the light levels and the LDR itself. Smaller LDRs with few tracks give a better response.

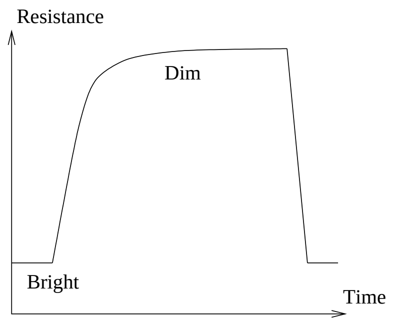


Figure 8.4: LDR response to rapid light decrease then increase.

The behaviour of LDRs does not appear to be well documented. From the behaviour of other semiconductors, for instance the current in photodiodes, we might expect the following relationship for the resistance, R of an LDR which is exposed to light of intensity I (A and B constants):

$$R = AI^B \tag{8.1}$$

To test this hypothesis the light responses of two small cadmium sulphide LDRs, were measured using the arrangement shown in **Figure 8.5**.

Because the intensity relationship must be dependent on light frequency, it was important to make

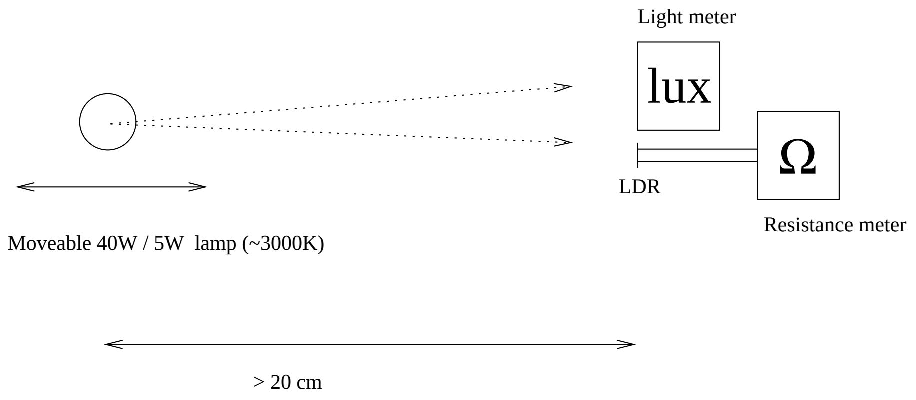


Figure 8.5: Measurement of LDR light response.

measurements using light of a constant spectrum.[4] Varying intensity by changing the lamp supply voltage would not achieve this, so the light was controlled by changing the *distance* between the lamp and the LDR and light meter. For light levels below 300 lux a small 5W DC bulb was used, with the voltage set to give a light of a similar colour.

The results are shown in **Figure 8.6**. The light range goes from dim, 100 lux, to daylight brightness, 1000 lux. Using logarithmic scales, each LDR gives a precise linear plot. This implies that in both cases the LDRs closely satisfy equations of the form 8.1.[5]

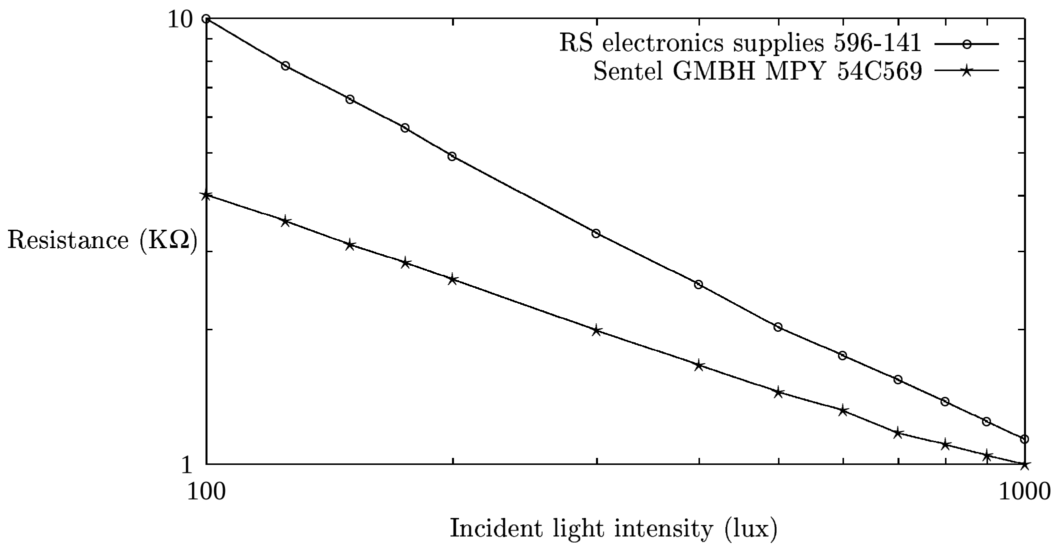


Figure 8.6: The light response of two LDRs.

The LDR light response curve can now be exploited using the ambient light detection method of **Figure 8.3**. If the ambient intensity is I_a and the shaded intensity on the finger hole is I_f, then

[4]Note this is the case when obscuring a finger hole. We assume, reasonably, that variation with the frequency of light will only affect *A* and not *B* in 8.1. To verify this accurately would required more sophistocated light measurement equipment. Given that most performances use ambient lighting of a similar spectral composure, it is not worth pursuing this detail.

[5]It interesting to consider the cause of the differing slopes. The chemical composure is the same, so it must be the geometry of the tracks; their spacing and thickness.

$$\frac{R_f}{R_a} = \left(\frac{I_f}{I_a}\right)^B \tag{8.2}$$

The intensity ratio is constant to ambient light level change, as for the LEDs. So the resistance ratio will also be constant, and can be used to measure finger position.

8.3.2 Practical Sensing Circuit

Dividing resistances to produce a voltage value is achieved very simply by feeding a fixed voltage to an inverting amplifier.[6] In order that one compensating sensor can divide several finger sensors a little more ingenuity is required. **Figure 8.7** shows a practical circuit achieving this. The circuit is arranged to produce voltages rising from zero as each sensor is covered by a finger. Current conservation on the -ve inputs of each opamp gives the following:

$$\frac{V_0}{R_i} = \frac{V_a - V_0}{R_a} \tag{8.3}$$

$$\frac{V_0 - V_a}{R_f} = \frac{V_f - V_0}{R_o} \tag{8.4}$$

Eliminating V_a gives an expression for the compensated finger voltage V_f

$$V_f = V_0\left(1 - \frac{R_a}{R_f}\frac{R_i}{R_o}\right) \tag{8.5}$$

The rapid, and unbounded, change in resistance R_f for small changes of finger position near closure is compensated by taking the reciprocal. The result is a voltage graduation which can be sampled linearly to give a suitable spread in digital resolution over the finger path; concentrated in the closed region, but not overly so. Preset 1 sets the effective distance range of the sensors. Preset 2 sets the upper bound of voltage output, and can also be used to trim out output noise in the finger up state. The resistances R_i and R_o should be chosen to be the resistance of R_f and R_a in dim lighting conditions. The circuit will then fail to operate properly at lower lighting conditions, because V_a will saturate.

Some variation in characteristics is found in the LDRs used. This is most easily compensated for in software and doesn't pose a serious problem. Occasionally, defective LDRs are encountered, and these can usually be picked out by visual inspection of the surface tracks.

To check that the circuit performs as expected, measurements of V_f were made using a similar setup to **Figure 8.5**. The LDR type used was MPY 54C569 by Sentel. Measurements at different ambient intensities were made for a small selection of different finger coverings. Blutak was used to mimic the presence of an obscuring finger being held still.

The results in **Figure 8.8** show that V_f is very stable over the whole light intensity range, for each covering.[7]

[6]Note that had the LDR light response curve been of a different form from 8.1 then it would not be possible to use such a simple method, and ultimately the CyberWhistle might not have been built.

[7]Temperature compensation hasn't been discussed, since its effect is much smaller than light variation. The required

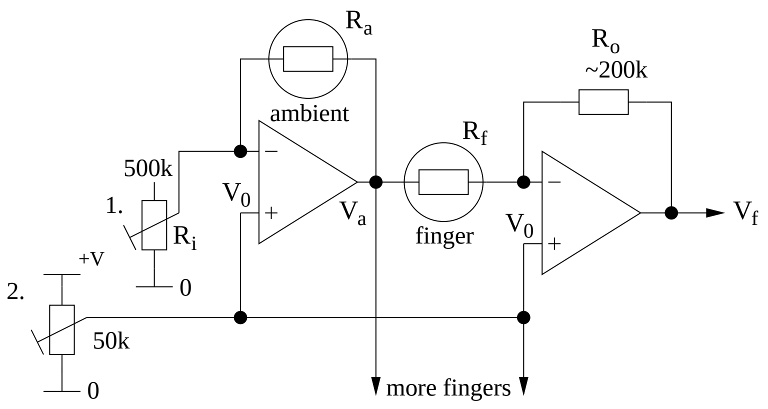


Figure 8.7: LDR finger sensor compensation circuit.

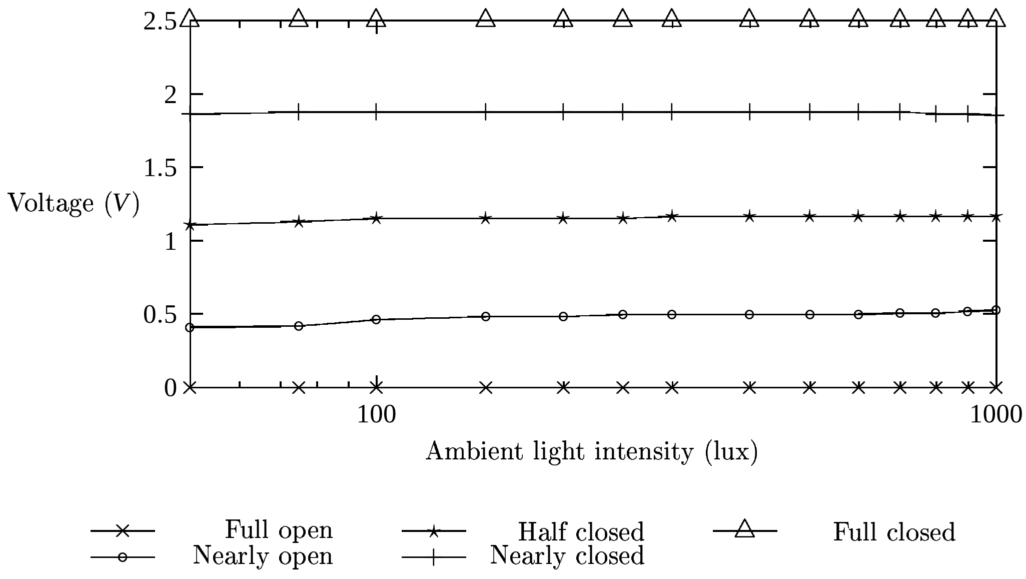


Figure 8.8: Performance test of light compensation circuit.

8.3.3 First Impressions of Using The Ambient Finger Sensor

A simple finger hole set up consists of a piece of card containing holes, with the sensors held in place with blutak. Experimentation with light sources and ways of covering the holes reveals a further advantage of LDRs over photodiodes and transistors. The receptive area is spread over a disc rather than a point. This ensures that the voltage change is gradual even when the source lights cast hard shadows. Different lighting conditions such as diffuse and directed give some control over the finger response. The substantial oscillations of intensity present in fluorescent lighting, about 30% typically, do not cause a fluctuating output, thanks to the ambient compensation. Diffuse response can be achieved with normal stage lighting by shining one light from either side of the performer and one from the front. The heating-cooling process mentioned already is most problematic for low light levels. For the kinds of light levels by which performers normally play the cooling effect effectively

working range cannot reasonably be outside 10 - 30 degrees Celsius, otherwise playing would be too uncomfortable. The LDRs only vary in resistance by less than 5% over this range. Also, the ambient LDR temperature compensates the finger LDR, making the compensated variation even less.

means the very last portion of finger movement between nearly closed and fully closed is lost. In practice this doesn't cause a problem, because the most useful region of finger movement for tactile feedback occurs when the finger is partially in contact with the finger hole.

The intensity of the shadow cast by a finger is in clear view of the player, so providing useful feedback about the current lighting conditions. It is possible to deliberately alter the playing response by changing the orientation of the finger holes to the light sources. The old term *shading* is no longer a metaphor but directly describes the finger sensing process.

8.3.4 MIDI generation

Initially a *Midi Creator* device[8], was used to convert the sensing voltages to Midi streams. Surprisingly, Midi Creator provides no direct way of outputting the quantized voltage inputs so a simple patch was added to the ROM which generates MIDI controller messages for each voltage input. It then quickly became clear that the sampling rate was too slow. Using a dual storage oscilloscope it was found that delay on each sample received is up to 20ms. Such a delay, or *latency*, between control input and audio output in an instrument limits the potential for musical control and expression. In practice an instrument using Midi Creator as a component would have a latency of at least 20ms due to the added delays of other parts. Section 8.4 gives a much more detailed account of latency. To reduce the delay, and the space and power requirements of the voltage to MIDI conversion circuitry, a custom microcontroller was developed using the inexpensive *Microchip 16C71*. This has four channels of analog input, which were extended to ten using an external multiplexer (See **Figure I.1** for details). The power requirements of 5ma at 5V are low enough so that 3 AAA size batteries are sufficient for many hours use. The microcontroller is programmed to scan the inputs and send output if an input has changed. It then pauses before moving on to the next input. The pause limits the rate of MIDI traffic to prevent overloading of the receiving computer. If the inputs are quiet and then suddenly change, the microcontroller can respond very quickly, because the last pause will have long since expired. Pauses of less than 1 ms are possible, but a range of values can be selected by setting configuration pins on the microcontroller. There are also pins for setting the output resolution. The maximum is seven bits, which fit in one MIDI controller message. For lower resolutions the least significant bits in the message are zeroed. This provides another means for limiting MIDI traffic, without sacrificing response time, or changing the value range received by the computer. See Appendix I for the microcontroller assembler code.

8.3.5 Breath Sensing

Commercially methods of breath sensing have varied widely from diaphragm-capacitative devices to optical veins, probably because no suitable off-the-shelf devices existed. When this project began, the only readily available pressure sensing components were bulky devices designed for pressure

[8]Supplied by Dawsons Music, Warrington.

ranges that are too great. Only recently have small convenient devices become available for a wide selection of pressure ranges. So initially, some form of breath sensor had to be designed. **Figure 8.9** shows the mouthpiece assembly for the first form of sensor devised. This is essentially a convenient adaption of the standard whistle mouthpiece, by the addition of a removable plastic insert.

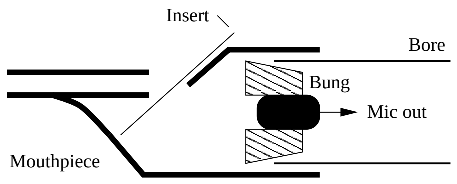


Figure 8.9: Breath sensing by noise processing.

Noise generated by air jet flow over the insert is transmitted via the inner space to the small electret microphone.[9] An amplifier is conveniently included in the package, and only 2 wires are required to provide power and communicate the audio signal at low impedance. The audio signal is conveyed directly to an audio input on the computer. The envelope of the noise waveform is generated by signal processing in software, as will shortly be described. Advantages of this method include high simplicity, economy and low latency. Using MIDI to relay the breath signal creates significant delays both in the sampling process and the software, whereas audio is only subject to delays due to the buffers. This breath control is very light to play, in keeping with the character of the penny whistle. Traditional windcontrollers tend to offer high playing pressures similar to real reed instruments. The audio processing is expensive in computational terms, but is not inappropriate in view of the computation required for the accompanying software synthesis. Feedback from the generated sound to the microphone limits the generated sound level. Shielding by the plastic insert and high pass filtering in the signal processing stage ensures feedback is not intrusive. The microphone is exposed to moisture, although not in the direct path of the air jet. With the microphone used, no problems have been encountered over extended playing periods.

8.3.6 Breath Signal Processing

By listening to the noise sound generated by the arrangement in **Figure 8.9** it appears simple to mentally construct a subjective amplitude envelope. Actually creating an algorithm which will do this with minimal latency is not so easy. The problem is similar to drawing a smooth curve through a cloud of points. This is much easier when all the points are visible rather than just those up to the current point of drawing. Consider the process shown in **Figure 8.10** in which a one pole low pass filter is applied to the square of the audio input.

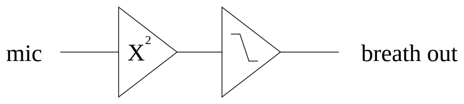


Figure 8.10: Simple envelope extraction.

[9]The part used was *EM10-B* from Maplin Electronics, but many similar miniature electret designs exist.

A cut off frequency of about 3 Hz is required to give a signal smooth enough for making smooth sounding amplitude modulations of other sounds (Higher cutoffs may be sufficient for other types of musical usage of the breath signal). Unfortunately the process smears genuine fast transitions at the start and end of tongued notes. Lower cutoffs may be used to make the response *more* sluggish with a long decay. This behaviour has intrinsic musical value of its own.

A small amount of noise may be desirable in the processed breath signal. This can be achieved by adding some of the input noise signal, which sounds more natural than synthesized noise.

Note Edge Detection

To detect the on and off transitions observe that the input signal then rises from near zero or falls to near zero. These situations are easily recognized with precision by applying a *weak*[10] low pass filter to the absolute value of the input and noting when low level thresholds are crossed. Without the filter the low level signals will be frequently crossed at other times.

Adjustable Smoothing

The smoothing filter in **Figure 8.10** can be controlled using the edge detector so that it responds faster for a short time at the note edges. Providing changes to the filter coefficients are continuous, then the smoothed signal will also be continuous. The result is sharp attacks and stops with smooth output in between. **Figure 8.11** shows a completed practical scheme for generating a useful breath level parameter. The waveforms are drawn in to illustrate how each stage works (It would have been very awkward to record and display *actual* simultaneous waveforms).

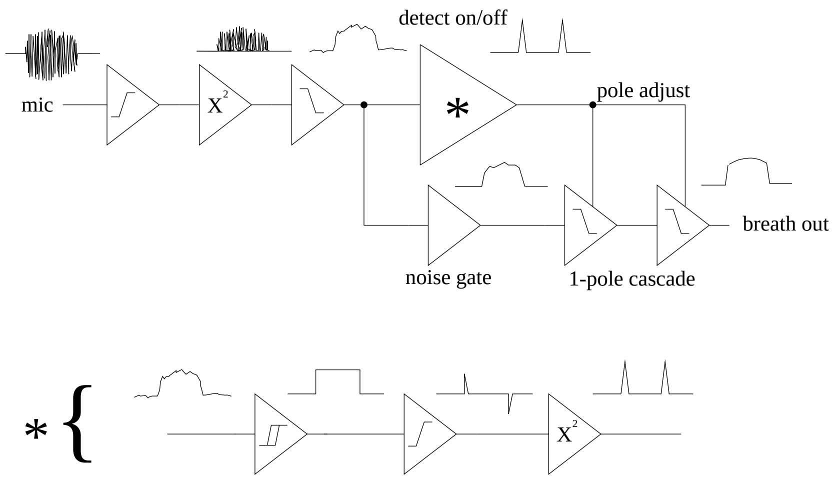


Figure 8.11: Robust envelope extraction with fast note edge response.

The on threshold level is slightly bigger than the off level, creating hysteresis which eliminates fluctuations in output caused by small signal inputs. A further refinement is the chopping of the

[10]This is simply a suggestive term for a low pass filter with a high cut-off frequency or a high pass filter with a low cut off. All filters discussed are 1st order unless stated otherwise.

signal by the noise gate to ensure that small background noises are eliminated: Output begins to rise smoothly from zero only when the on threshold is exceeded. The smoothing section is controlled by driving the pole coefficients directly, as recalculation of pole values from frequency cutoff is too costly, and unnecessary. Driving the two one pole filters in cascade makes a decisive difference by better separating the useful control signal from higher frequencies derived from noise in the microphone input. There is much scope for adjustment. The detect on-off pulses can be adjusted in width by setting the cut off of the high pass filter in *****. A wider pulse causes a longer period of low smoothing, which, if long enough, adds some 'crunch' to the attack by admitting a short period of noisy signal. The height of the pulse determines the fastest rate of change in the output. If too fast the output will sound 'clicky'. The smoothness of the output inside notes is determined by the default cutoff frequency in the smoothing section. A cutoff of 6Hz for each of the one pole filters gives a smooth but agile response which responds well to vibrato breath technique. For higher frequencies noise components become progressively more noticeable, and for lower frequencies the response feels slower.

The process described above captures the important features of breath control; tonguing and slower, smooth variations. There may well be better processing methods, but perhaps with greater computational cost. In particular the main smoothing section could be replaced with optimized higher order filters. Recalculating coefficients for such filters can be costly and also lead to artifacts in the output which disrupt the smoothness that is required.

8.3.7 Overview of Physical Construction

The first version of the CyberWhistle consists of a penny whistle in D with LDRs mounted in a semi cylindrical wooden chassis. See **Figure 8.12**. The chassis is held in place by friction and is readily removed or replaced. The bung prevents moisture from entering the main bore. The mouthpiece is easily removed for cleaning. LDR leads feed through the chassis to a rectangular circuit board which sits on the flat side of the chassis. A ten core cable is used to link the chassis to an external control box, containing the rest of the electronics and batteries. The box provides audio and MIDI output ready for computer processing.

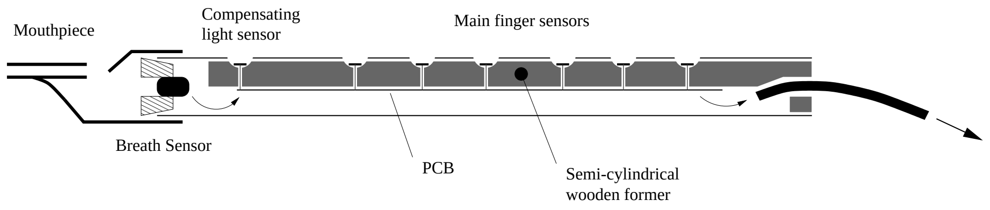


Figure 8.12: CyberWhistle cross-section.

8.4 Tools For Software Synthesis

The purpose of this section is to establish some appropriate working methods for developing synthesis software for the CyberWhistle. This includes the choice of the underlying language and techniques used within the language.

In Section 4.8 various signal processing tools were introduced. At the lowest level are C and C++, which must be compiled with machine specific I/O libraries. Just above this are Csound-like languages, which offer fast development and platform independence at the expense of some efficiency and flexibility. At the highest level are graphical audio DSP design packages with varying degrees of sophistication. SynthBuilder, [PJS+98], would certainly be a useful tool but is currently not available on computers used by this author. In any case, relying on a system with limited portability could hinder future work.

The performance of C, C++ and Csound for audio processing will now be considered in more detail.

8.4.1 Csound and Limitations

Initially real-time Csound was used to experiment with different ideas for software synthesis. Although convenient for implementing simple ideas, several drawbacks quickly emerged. An immediate problem is finding a way to enter MIDI controller data into Csound without passing note events required to open MIDI instruments within the orchestra. This is the same problem encountered in Chapter 7. The work around used before was to pass MIDI via stdin from a MIDI receive process which additionally sends an initial note on. This degrades MIDI transfer performance, and a better solution is to make the microcontroller in the CyberWhistle send a note on event when powered up.

A more central problem is that Csound provides a poor basis for structured programming. Using conditional structures is very awkward and there is little support for object style code. Physical modelling lends itself to object style, as will be shown later. Ultimately the biggest arguments against Csound are the central benchmarks of a real time system; latency and computational efficiency.

8.4.2 Measuring Delays Within The System

Early on in development, while Midi Creator was being used, the timing delays in each part of the system were measured to try and locate trouble areas and improve them. **Figure 8.13** shows the flow of input signals through to the output, and the sources of delay which they encounter.

Sensor Delay

The delay from the finger sensors was measured first. This was expected to be significant because of the heating/cooling property of LDRs discussed earlier. The LDR type used was MPY 54C569

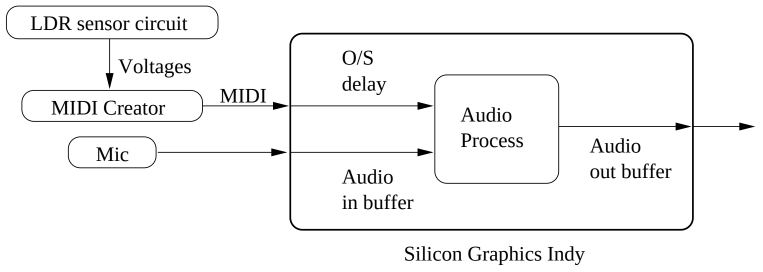


Figure 8.13: Overview of delays acting in the CyberWhistle signal path.

by Sentel. The compensated finger voltage, V_f, for one fingerhole was monitored on a storage oscilloscope set to one-pass trigger mode. Under office lighting conditions, the fingerhole was rapidly covered by sliding some card over. The same was done, with an adjusted trigger level, to capture the change in output as the card is removed rapidly. **Figure 8.14** shows the important features of the resulting traces.

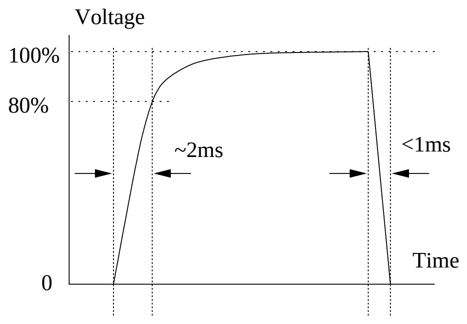


Figure 8.14: Compensated finger voltage for finger-down and finger-up.

The sensor delay cannot be stated simply. When closing the fingerhole the output rises ever more slowly towards the upper voltage limit. To give a delay corresponding to 'finger down', this state must be defined by the output rising above a threshold. The threshold indicated in the figure is a compromise which ensures the delay is acceptable, and also the finger is substantially in contact with the fingerhole, with small gaps letting light in. The delay given of 2ms is not a simple latency however, because the initial response to a finger-down is very rapid. Even in the later stages of closure, the output voltage rate of change is virtually instantaneously affected by the light level. This provides a form of performance feedback, though not as obvious as a direct level change.

The finger up delay is much shorter, because the presence of light immediately increases the absolute rate of change of resistance of the finger LDR.

Midi Creator Delay

MIDI output from the Midi Creator occurs at regular 20ms intervals, so that the information rate is limited by the speed of processing rather than the MIDI standard. Each output burst consists of MIDI control messages for input voltages which have changed since the last burst 20ms previously. This implies the midi messages are delayed by up to 20ms from the input voltage signals, plus some extra delay of a few ms if the burst is long. The incidence of delay is uniformly distributed across each 20ms interval, so the *average* delay is atleast 10ms. The remaining delay all lies within the computer.

Computer Delays

Initially a simple Csound orchestra was written which amplitude modulates a 1000 Hz square wave with the fingerhole signal received via MIDI. This made it easy to recognize the control signal in the computer output using an oscilloscope. Another orchestra was written which copies the audio input from the breath microphone directly to the output. There are two main signal paths to consider. The first passes from the mic through the input audio buffer, the program and the output audio buffer. The other passes via the MIDI buffer to the program and then to the output audio buffer.

The samples are processed in blocks by Csound. The buffers tend to a dynamic state in which a block is read from the input buffer as soon as enough samples are available. The program then processes the block to generate a output block which is written to the output buffer as soon as enough space becomes available. The output buffer is then full. No matter where a sample is in a block it experiences the same overall delay between entering and leaving the buffers, which is equivalent to the size of the output buffer. It was found that, contrary to Silicon Graphics documentation, the smallest audio buffer sizes can be achieved in 4-channel mode, regardless of whether four channels are actually required. The output buffer is then 256 samples in size. This gives a total delay for the audio path of nearly 6 ms. This figure was confirmed by monitoring the input and output of the second orchestra on a dual storage oscilloscope, and measuring the relative offset of the two traces. Latencies for commercial MIDI synthesizers from MIDI input to audio were measured by monitoring MIDI input and audio output, and found to be approximately 3-4ms. The measured delay from Midi Creator voltage input to audio output using the first orchestra was 30 ms. Of this about 20 ms is due to Midi Creator and 6 ms due to the audio output buffer, as previously determined, leaving 4 ms delay in feeding MIDI from the port to the program. This is composed of the delay caused by the operating system and delay resulting from the method by which Csound reads MIDI from the Operating system.

8.4.3 Improving On Csound Latency Figures

Each of the component figures can be improved, the greatest improvement being made by replacing Midi Creator with the custom microcontroller described in Section 8.3.4. This can be adjusted to ensure a maximum 1 ms delay, although in practice larger delays may be used, if for instance the MIDI receiver cannot process high MIDI rates.

Audio Buffer Dynamics

The simplest method of double audio buffer operation, employed by Csound, reads from and writes to the buffers freely without any additional control. The read blocks until sufficient samples are available on the input buffer and the write blocks until room is available at the top of the output buffer. At program start audio channels are initialized with empty buffers, so the audio latency is low. This means that a halt in the program caused by operating system activity or window operations, is very likely to cause build up on the input buffer. The output buffer runs dry resulting in an audible glitch. When the program resumes, the input is swiftly transferred to the output causing a backlog of samples on the output. Further glitches cause a build up in the backlog up to the limit set by the output buffer size. For programs which can process the audio blocks swiftly, the backlog build up can be gradual, causing sporadic glitches at later times. To minimize the chance of glitches from the outset and make the latency predictable, the output buffer should be filled once the audio channel has been opened.

Reducing Audio Buffer Latency

The Silicon Graphics audio library provides a method for waiting until the output buffer has fallen bellow a given level, using *ALsetfillpoint* and the unix command *select*. If we wait for the output buffer to fall in this manner before each block write, then the audio latency can be reduced past the minimum buffer size. This is not the intended use of ALsetfillpoint, but it does work. The main limitations are the overhead in scheduling the select command, and, of course, the ability of the audio program to sustain block transfer and prevent glitches.

Reducing Glitches

Even if the audio latency can be reduced to the desired level, it may not be usable because background processes are causing audio glitches. Increasing the latency reduces the frequency of glitches until there are none. To reduce the chance of glitches at low latencies, the priority of the audio process can be increased. This tends to limit the time of each break from the audio process. In Irix the *schedctl* command can be used to select non-degrading priorities for a process. It is unfortunately necessary to run the computer in stand-alone mode to do this, as super user status is required to enable high priorities. Even then, high priorities pose an increased risk of system failure. Unix was not designed with low latency real-time applications in mind.

Reducing MIDI Input Delay

Csound strobes the MIDI input buffer at boundaries of the audio blocks. This adds delays of up to the block length onto the delay from the MIDI port to the buffer. An alternative strategy is to create a background process that sleeps until MIDI is received. It then writes into the variable space of the main audio process which reads from the shared space once every audio sample. This ensures that the MIDI input buffer is only read when there is something to read.

This technique gives quicker response but for fast MIDI rates the overhead is a little higher than for the Csound method, owing to the increased amount of scheduling disrupting the main audio process. In both cases there is a substantial overhead due to the operating system processing incoming MIDI. This was found to be the case with a variety of drivers including the Silicon Graphics md library, and the driver contained within Csound, suggesting that the ability to process MIDI efficiently is limited by the computer hardware arrangement. A compromise must be struck between MIDI response and the audio processing rate. The MIDI microcontroller facilitates this by means of resolution and rate settings previously discussed in Section 8.3.4.

Latency Improvements Measurement

With the above improvements the latency measurements of Section 8.4.2 were repeated using an dual trace storage oscilloscope again. The Csound scripts were replaced with a C program performing the same functions, but with the MIDI input implemented as above. The audio input was modulated by MIDI note events to give the audio output. The audio to audio latency was 6ms and the MIDI to audio latency 9ms. The total latency for voltage to audio via the microcontroller was 10ms. The new latencies are much better though not as good as typical commercial synthesizer MIDI to audio delay of 3-4 ms.

8.4.4 Computational Efficiency

To implement a given signal processing task a certain minimum number of arithmetic operations are required. In addition, processor time is taken up executing commands such as branches, subroutine calls, operating system I/O routines handling audio and MIDI. To optimize the computational efficiency, the time spent on non arithmetic operations must be minimized subject to overall design constraints in the way the system should be used. Csound consists of a library of routines for the unit generators, and an orchestra and score reader. The score is processed initially to produce a schedule for the usage of unit generators. When the schedule is run, each new instance of a unit generator prompts the creation of workspace containing the internal variable states for that instance including the output audio block, if the output is audio rate. When each instance is executed, the workspace is addressed indirectly with pointers. Block processing of audio samples is carried out with a loop, which may be rolled up in compilation. The overheads for each instance consist of the initial subroutine call and the disruption this has on the cache and the pointer dereferencing. It is true that the overall efficiency can be increased by increasing the audio block size, but this effect only becomes significant for blocks which are comparable to the minimum latency used, set by the minimum audio buffer size. Another problem of processing audio in blocks is that the minimum feedback time for variables in the orchestra is the block length. This limits the design of recursive filters and makes setting delays awkward. Csound was not originally designed for real-time use, so the limitations are not surprising.

8.4.5 Using C++

Using C++ directly offers a practical alternative for creating a modest structured experimental development environment. Unit generators can be defined as classes, and objects defined to represent different instances of a unit generator. The natural core of many audio processing routines, and certainly those considered in this chapter, consists mostly of a few simple operations connected by a hierarchical structure. This can be implemented efficiently in C++ by compiling the objects inline, so there are few function calls. Function calls break the pipelining of multiplications which feature heavily in audio DSP code. Inlining also gives the compilation optimizer more freedom to reorganize register usage. Without function boundaries we are not compelled to process samples in blocks to gain efficiency.

Therefore each object can process one sample at a time efficiently, giving freedom to set delay times in feedback networks with 1-sample resolution.

Nested objects are a natural way of implementing physical models. For instance, a woodwind bore can be decomposed into similar sections each containing one bore hole. Objects also provide a powerful way to build any kind of synthesis model. Csound has only one layer of abstraction, so it is awkward to create new unit generators. Using C++ gives access to the flexibility of C-like control code at every level. In Csound this is available only within the unit generator definitions. The simplicity of programming at ground level, in an orchestra, can make development and experimentation very fast, provided no new features are required. This is mainly because the instantiation of unit generators is carried out automatically. Using C++ without extra layers of programming abstraction implies that instantiation must be made explicitly before the main audio loop. This increases flexibility at the expense of hindering the process of addition and removal of objects during development.

8.4.6 Summary : A Framework For Software Synthesis

Despite reservations about instantiation C++ was adopted as a convenient and efficient tool for implementing audio signal processing. To provide a standardized approach to designing and testing synthesizers for the CyberWhistle, a base class *CWinstr* was defined. This contains a routine for mapping raw MIDI values into control filters and the breath processing algorithm described previously. The mapping process itself is quite flexible. It allows for calibration of the finger holes to allow for slight differences in response. Also, the active range within the calibration region, and the values to which this region maps can be defined. The active range is read at runtime from a file. This facilitates the determination of a standard calibration for the CyberWhistle at hand, with minimum programming outlay. Other CyberWhistles could each have there own calibration file, so that they would perform similarly with the same instrument binary. This makes the binaries portable. The advantage of performing the scaling of the calibrated region at the control stage is to remove this from the audio stage, where the higher rates incur a greater penalty. See *processMidi()* in *CWinstr.c*

Filtering Control Input

Filters are used to interpolate the quantized MIDI information and so approximate the source control voltages sampled at audio rate. This can be important if the control information affects the sound output in a manner such as amplitude modulation. Modulation by an unfiltered MIDI control signal would create the unpleasant ‘zipper’ sound. In some cases it is important that the modulating signal is not only continuous but smooth as well.

Linear point to point interpolation cannot be carried out in real-time without delay because the target point is not available when the start point has been received. In Csound interpolation is carried out on audio blocks by joining the control values at either end, so the MIDI signal is effectively delayed by as much as one audio block.

As an alternative, a simple predictive filter was designed which changes the output by a fixed rate until either the last input is reached or a new input is received. When a new input is received the change rate is recalculated as the rate of change between the last input and the current input. Thus the output is immediately affected by a newly received input. Strictly speaking the latency is zero. This method is implemented as class *AsyncFilt*. Smoothing of the corners can be carried out with a weak[11] low pass filter, but generally the additional expense is not justified by the increased audio quality (The CD audio examples don’t use any additional low pass filtering).

The instruments which inherit CWinstr can modify parameters in the base filters to suit the particular response characteristics required. For instance the in-note smoothness of the breath filter can be altered with *setSmoothBreathFreq*, or fingerhole output ranges can be adjusted. Overloading is used on the input and output of the instrument definitions to easily increase their versatility, for instance by having optional numbers of audio output channels. *run.c* provides an interface to the audio and MIDI device drivers. The audio ports can be configured for buffer size and sampling rate. The MIDI port can be read either by a background process which updates the control filters, or at intervals by the main audio process. The implications of the two methods have already been discussed in Section 8.4.3.

8.4.7 Testing The Framework

For testing purposes the first instrument was created, *CWtest*. This emulates the behaviour of the Csound instrument used in evaluating timing delays. Each finger controls the amplitude of a square wave component in the output. Each finger plays sound to the right or left of the stereo field. A similar instrument, *CWtestSine*, generates sine wave output (using efficient waveguide oscillators by

[11]For instance, a simple 1-pole low pass filter such as Csound’s *tone* with a cut-off of about 40Hz is very useful for transmitting typical human control gestures while suppressing non-smooth points resulting from interpolation.

Smith, [SC92]). Tracks 17-20 on the CD contain short phrases played with the CyberWhistle. The simplicity of the underlying synthesis helps to expose the raw characteristics of the breath and finger control systems. The recording demonstrates the fast on/off breath control response combined with smooth in-note response. The finger responses are even and fast enough to accommodate convincing trills. Listening to the examples cannot give a full appreciation of the control behaviour from the player's perspective, but does at least give some insight.

Finger dynamics enriching the output

Although the test instruments were not designed to be musically useful, they do have some interesting properties. When trilling between two holes the balance and dynamics of the two tones is reflecting the dynamics of the fingers under the constraint of the whistle surface. These dynamics enrich the player interaction and ultimately the sound produced.

Computational performance

By using the unix system monitor utility *os_rview*, the program and system activity can be judged in real-time. For no MIDI input, the C++ code is about 8 times more efficient than the equivalent Csound. As the MIDI input rate is increased, the ratio falls to about 5, as the common MIDI system overhead rises. At maximum time and bit resolution settings on the microcontroller, CWtest functions without glitching. With audio process priority enhanced the audio becomes immune to window operations. Clearly this is a good indication for future instruments, although these are only simple synthesis routines.

8.5 Physical Modelling Synthesizers

The literature on implementing physical models of musical instruments is now very extensive. Unfortunately the computational demands are quite severe even for simple real-time models. The more recent models presented in the literature are well beyond the power of the SGI Indy computer. Attention is focused, instead, on the ways in which the unique control interface of the CyberWhistle may be used to 'bring out' the qualities in the simpler models. It would be appropriate to begin with a model for a penny whistle. Unfortunately, models of flue based instruments are under-represented in the literature, and no practical implementations for recorder like instruments, such as the penny whistle, have been found. This is owing in part to the subtleties of the switched-flow aerodynamics which operate in flues. Implementations of flutes exist, but these really demand a more complex mouthpiece controller so that blowing pressure and jet length can be coordinated together to control pitch.

8.5.1 Hybrid Single-Reed Models

The clarinet is one of the first instruments to be implemented with some success. Smith, [Smi86], describes a simple system which efficiently implements the algorithm presented by McIntryre, Schumacher, Woodhouse, [MSJ83], in the case where the bore is a simple tube.[12] Different pitches are achieved by changing the length of a single delay line representing the bore. For the CyberWhistle the model has been expanded to include the tone holes. Instead of generating a model for a clarinet bore this has been substituted with the much simpler 6 hole whistle bore. Precise 3-port tone hole models have been formulated by Keefe,[Kee82]. Valimaki, Karjalainen and Laakso, [VKL93], have produced efficient implementations of the most essential features in Keefe's models, but the computational load for six such tone holes is still prohibitively heavy for this investigation. Here the tone holes have been modelled in two simple ways. The filter class *WhistleBore* implements the model shown in **Figure 8.15**.

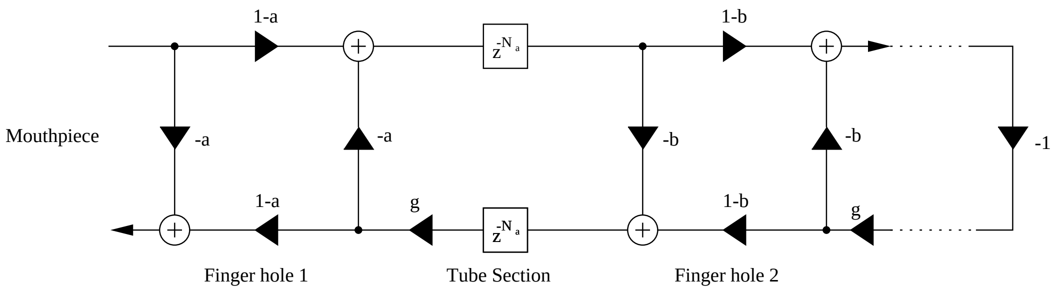


Figure 8.15: Waveguide bore model with 2 port scattering toneholes.

Each tone hole is a 2-port scattering junction parametrised by the corresponding finger control, $0 \leq a, b \leq 1$. When finger 1 is down, $a = 0$, the tone hole acts as perfect transmitter. When the finger is up, $a = 1$ the tone hole acts as a reflector to waves travelling from both sides. Intermediate finger positions, $0 < a < 1$, cause a mixture of transmission and reflection.

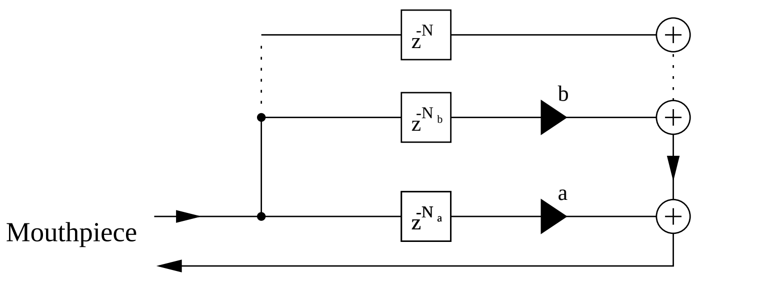
Losses in each section are lumped into one multiplication per section. The losses cannot generally be commuted outside the bore sections because the sections could then contain lossless circulations[13].

The second model, implemented in the class *WhistleBoreP*, is more abstract. Part of the motivation for trying this model is to experiment with the essential features of waveguide models, the delays and non-linearities, without being concerned about the physical correctness. Refer to **Figure 8.16**. The models are equivalent. The lower form uses less memory, but the tuning of fingerholes is not independent and therefore inconvenient.

When a finger hole is opened the corresponding delay is mixed in with the total bore delay. This has a comb filtering effect which suppresses harmonics in a similar manner to a tone hole or a slightly

[12]Perry Cook of Princeton University gives some example C++ code for simple tube instruments at *http://www.cs.princeton.edu/ prc/NewWork.html#STK* (16 October 98)

[13]This might however be exploited as an unusual musical effect. Sections could be selectively 'trapped' and released to create a background of drone style sounds. The effect is evident in some of the CD examples with large bore sections and long decay times.



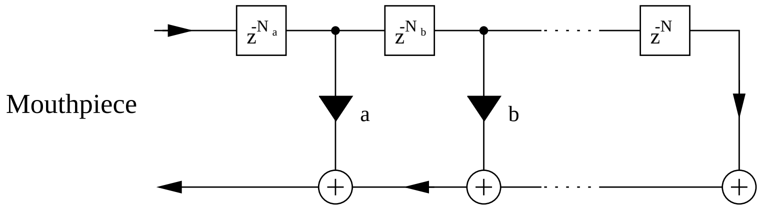


Figure 8.16: Equivalent abstract bore models.

damped string. The two networks in the figure are equivalent. The lower form uses less total delay time and therefore less memory, but the ‘tuning’ of each finger hole cannot be altered independently. Neither model contains discrete filters, let alone separate filtering for each section, as it would be too expensive. However, a filter is applied in lumped form from within the instrument class.

Delay implementation

The WhistleBore classes use a class *Delay* for the delay sections. The implementation of delay is worthy of careful consideration, since it is a key component in waveguide physical models. There is unfortunately no escape from using large amounts of memory in implementing a delay of even a few milliseconds, resulting in regular cache misses. However, the updating of the memory pointers using conditional statements can be improved by using a bitwise AND to wrap the pointers around unconditionally. The instantiation automatically generates the smallest power- of-two memory samples that will contain the delay.

Tuning

The tuning accuracy of the instruments is limited by the sample resolution of Delay. An interpolating delay class, *DelayL*, was created, but this adds significantly to the processing burden. With DelayL, quite general tunings can be theoretically created, but finding suitable bore parameters is difficult. The highest pitches are particularly unreliable because of aliasing caused by the nonlinear part of the instrument. The only solution is to oversample the waveguide system, and filter digitally to produce band-limited output. Oversampling would multiply the computational cost by the oversampling factor, with some additional cost due to the digital band-limiting filter.

CD recordings

The CD contains some example recordings played using CWclariHol and CWclariHolP. The behaviour of the instruments used to make the recordings is now described. The principal aim is to justify the use of the novel features of the CyberWhistle, especially the continuous finger sensing.

Tracks 21,22 : This instrument is tuned to an approximate scale in the upper register. The *Tone* filter class is used for the main lumped filter, with a cutoff of 3000 Hz. This is a first order low pass filter. The sampling rate is 32000 Hz. A little breath noise is used, and each tube section has an attenuation factor of .999. The player is immediately aware that the behaviour of the new instrument is considerably more complex than the test instruments. The sustained part of the tone is greatly affected by the initial profile of the breath attack. A smooth attack leads to a mellower tone, while an abrupt start is likely to create persisting high frequency harmonics. Fast note transitions create a blend from one note to another. When the transition is slowed right down, the first pitch bends in pitch and diminishes in volume, before the next pitch appears and bends to its final value. By carefully reducing the breath pressure over the central part of the transition a convincing unbroken pitch slide can be made from one note to another. This is easier for smaller pitch intervals. Forked fingerings have no effect, because the first fully open hole excludes any signal returning from lower parts of the bore. A real tone hole is never completely reflecting. The simple two port models can be modified to be never complete reflectors by changing the range of the control input. This can be achieved conveniently and efficiently by initial modification of the finger filter output ranges from within the instrument class.

Tracks 23,24 : Increasing the noise injection in the last instrument results in a less refined sound, but the instrument is easier to play. Breath attacks don't need to be so precise, and finger slides are more easily supported.

Tracks 25,26 : The previous tuning of tracks 6 and 7 is transposed down. The result is a very smooth sounding bass clarinet, possibly lacking in some character in the upper frequency range.

Tracks 27,28 : The bore filter of tracks 10 and 11 is weakened to give a much higher cutoff frequency, around 10 KHz, resulting in a brighter sound. The instrument can easily become locked onto high frequency components.

Tracks 29,30 : The tuning is lower still. The high frequency artifacts begin to take on a life of their own, obscuring the underlying bass. The stereo output is composed by mixing signals from different parts of the bore into each channel. This effectively serves as a very simple simulation of the acoustic radiation pattern from a real instrument. In fact, the sound from most natural objects is projected in a complex pattern, and we find it unnatural and sometimes uncomfortable not to hear sound in this way. The projection of sound onto stereo can be extended to multiple speakers. One possible concert configuration is to project parts of the instrument inwards to the audience from surrounding speakers. This would create the impression of being inside the instrument. It would also help to give a true identity to the speakers, which otherwise might seem necessary but inconvenient.

Tracks 31,32 : Sub sonic tunings, with the same bore filter. Broad-band sounds evolve from pulses. Precise shading combinations result in varied set of effects. Good control of breathing is essential to balance the sound and control its evolution. This kind of instrument might be useful in an electroacoustic concert setting.

Tracks 33,34 : These were created using ClariHolP, which uses the alternate simplified form of bore network. This is even harder to tune, but forked fingerings offer more possibilities, since the feedback coefficients are controlled by each finger independently. In this first example it is striking how differently the instrument behaves in comparison with a similar instrument using ClariHol. The nonlinear interaction is unchanged, but changes to the linear bore component are important. An unnatural chaotic sound results when the amplitude is forced sufficiently, which provides an interesting variation on familiar wind instruments.

Tracks 35,36 : The bore filter has been relaxed again, so that the higher frequencies are not so suppressed. Harmonics can be selected more easily than in the case of ClariHol, by closing holes in combination, in the manner of register holes on real instruments.

Tracks 37,38 : The instrument of tracks 18 and 19 has been transposed downwards. The chaotic effect is more prominent and manifests itself as a burbling sound.

Tracks 39,40 : The bore filter is relaxed, resulting in a scream-like quality to the sound, with a powerful broad-band harmonic sound.

8.5.2 Hybrid Flute Models

The flute is more closely related to the penny whistle than the clarinet, but requires several embouchure parameters to control pitch accurately. As pitch is not an important consideration either practically or aesthetically here, some example flute model based instruments are presented. The instrument class *CWfluHol* calls upon a class *Flutemouth* for nonlinear mouth modelling and WhistleBoreP for the bore. Flutemouth is based on an implementation by Cook, [Coo92], which in turn derives from work by Fletcher, Coltman and Isling among others, [MSJ83]. Ideally the player should be able to modulate the jet delay by some means of embouchure control, but here the delay is fixed. It might be possible to control the delay in some intelligent way, as a player would, from the desired pitch, but it seems this approach is unlikely to offer any musical advantages over direct parameter control by an experienced player.

CD recordings

Tracks 41,42 : This example demonstrates how some characteristics of a real flute can be convincingly reproduced. The note transitions in particular are more effective than in a system using simple switched finger states.

Tracks 43,44 : By lowering the tunings and weakening the bore filter, a new abstract instrument is created, similar to the CWclariHol instrument of tracks 16 and 17. In this case the delays corresponding to each bore section are equal, which leads to particularly rich harmonic effects.

8.5.3 Hybrid Bowed Models

It is a simple matter to substitute a model of a mouthpiece excitation mechanism with a model for a bowing interaction, and retain the bore component and associated control by fingerholes. **Figure 8.17** illustrates this scheme. An additional delay is required to model the bridge side of a vibrating string. Alternatively the bridge delay can be replaced with a second bore, and the finger control can be divided between the two bores.

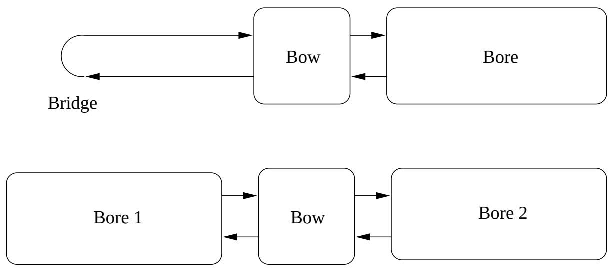


Figure 8.17: Two abstract instrument models using a waveguide bowing interaction.

The motivation for creating such a model is to make use of the interesting bow string interaction in a new context. Finger shadings can be mapped to degrees of finger pressure applied to strings. Above a threshold pressure, the string is clamped and waves are reflected as efficiently as possible. As the pressure falls the damping and transmission increase.

Bow interaction detail

The bow string interaction is the most fiercely nonlinear encountered in real instruments, and is complicated by the presence of hysteresis. It is more complex to implement than the other models. McIntyre et al, [MSJ83], survey previous work on string interaction and present an efficient time domain algorithm for the calculation of the entire instrument. This was simplified by Smith in a DSP realization, [Smi86], containing no hysteresis. The result is the attack of the note, where the hysteresis is most important, is not very convincing, although the sustained section of the note is surprisingly good. In commuted string synthesis, [Smi93], which is reported to be very good overall, the attack is simulated by initially plucking the string at frequent random intervals. Pearson, [Pea96], gives an example of musical synthesis using hysteresis. This is successful in producing interesting starting transients, although the models are constructed of cells and are computationally very expensive. Here two methods are presented for adapting the hysteresis model to waveguide architectures. The class names for the corresponding implementations are *WaveBow* and *MassBow*.[14]

[14]These have only been partially tested due to lack of time. No CD examples are available.

WaveBow

Schumacher, [SW95], presents the solution of the slip-stick bow interaction as the intersection of a sloped line with a curve and a vertical line. In **Figure 8.18** the solution is given more directly as a hysteresis function, in a form suitable for calculation. The arrows on the graph indicate allowed directions of change.

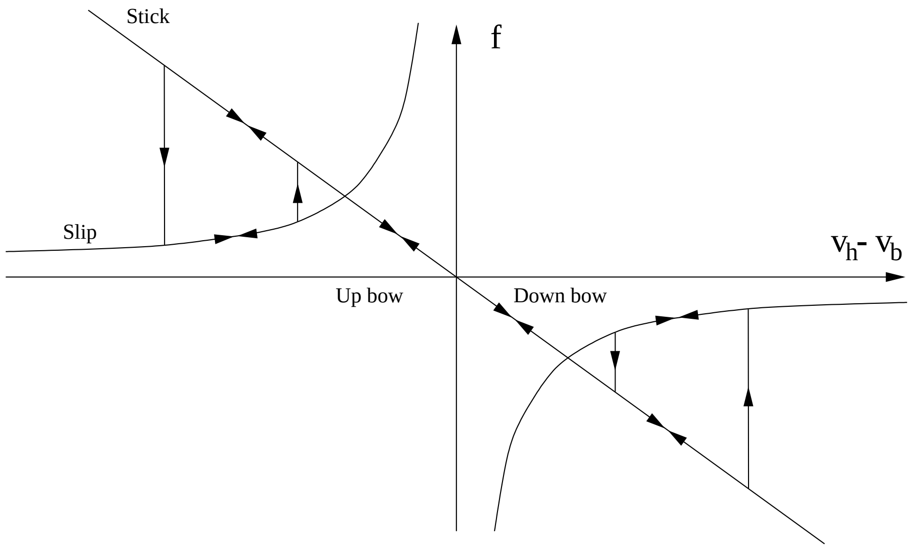


Figure 8.18: Hysteresis function used by WaveBow.

Following the names used by Schumacher; f is the transverse force applied by the bow on the string, V is the string velocity at the bow point, V_b is the bow velocity, V_h is the sum of the *incoming* string waves at the bow point, and Y is the wave admittance of the string. In the waveguide implementation, V_h is just the sum of incoming waveguide signals.

Schumacher relates the velocity of the string to the bowing force as follows:

$$v(t) = \frac{Y}{2} f(t) + v_h(t) \tag{8.6}$$

From this, each outgoing velocity wave from the bow point is the sum of the in-going velocity wave of the same direction and $\frac{Y}{2} f(t)$. **Figure 8.19** expresses this as a waveguide process.

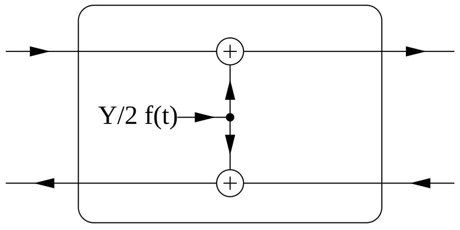


Figure 8.19: Waveguide bow structure. .

To summarize, the calculation procedure is to find v_h from the sum of incoming waves, and then use this to find the bow-force from the hysteresis function. The outgoing waves are just a constant multiple of the bow-force.

MassBow

The main problem with WaveBow are the discontinuities in string velocity generated by the slipping and sticking processes, causing harshness and aliasing in the resulting sound. The slipping process could have a time constant artificially incorporated, but here we try to retain the physical relevance of the model. *MassBow* attempts to overcome the problem by modelling a small section of string around the bow by a mass (see **Figure 8.20**). Since the forces acting on the mass are limited, the mass velocity changes continuously, ensuring outgoing waves are continuous.

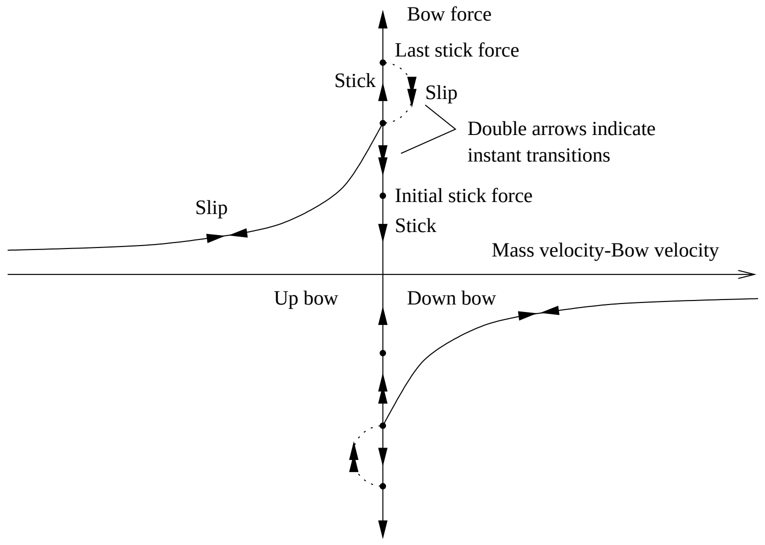


Figure 8.20: The bowforce acting on a bowed mass.

When slipping, the mass accelerates according to the forces applied by the bow and the adjoining string. The bowforce depends on just the relative velocity of the bow and string. Sticking occurs when the relative velocity becomes zero.[15]

The bowforce drops at the moment of sticking, because only enough force is required to balance the returning force in the string caused by string tension. For the brief sticking period the motion of the string and bow is closely described by constant velocity, until the point when the force between bow and string exceeds a threshold. The bow then slips.

The method of calculation is quite different to the case of waveBow. In addition to the velocity wave states, the velocity state, v, of the mass must be stored. Let v_r^+ denote a right going traveling velocity wave leaving the mass, and v_r^- a left going wave approaching from the right as shown in **Figure 8.21**. Similarly denote v_l^+, v_l^-.

The departing waves v_l^- and v_r^+ are found directly from the requirement that the string and mass

[15]In Pearson's model sticking occurs when the string velocity becomes positive, a physically inaccurate condition that none the less yields good results. This suggests how robust the bow model is to modification, and that the underlying hysteresis is the defining characteristic of the instrument rather than the details of how the hysteresis operates.

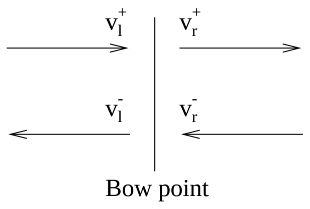


Figure 8.21: Labeling of traveling velocity waves. .

velocities be continuous in the region of the mass:

$$v = v_l^+ + v_l^- = v_r^+ + v_r^- \tag{8.7}$$

$$v_l^- = v - v_l^+ \tag{8.8}$$

$$v_r^+ = v - v_r^- \tag{8.9}$$

Let f be the transverse component of string tension force acting from the left on the right. This can be expressed as the sum of traveling wave components, $f = f^+ + f^-$. Let $f_{string\rightarrow mass}$ be the force on the mass from the string. This is the sum of the transverse components of forces from the string on either side of the mass, which can be given in terms of the traveling velocity waves and the string impedance R, $f^+ = Rv^+$, $f^- = -Rv^-$ (For a detailed explanation of how tension waves can be related to velocity waves refer to [Smi92]). Using 8.7 $f_{string\rightarrow mass}$ can be written in terms of just the incoming velocity waves and the mass velocity:

$$f_{string\rightarrow mass} = f_l^+ + f_l^- - (f_r^+ + f_r^-) \tag{8.10}$$

$$= R(v_l^+ - v_l^- - (v_r^+ - v_r^-)) \tag{8.11}$$

$$= R(v_l^+ - (v - v_l^+) - (v - v_r^- - v_r^-)) \tag{8.12}$$

$$= 2R(v_l^+ + v_r^- - v) \tag{8.13}$$

This force can be seen intuitively to act in the expected sense since if an external force tries to move the mass with enough velocity, the string will oppose with a force in the opposite direction. The only external force acting on the string at the mass is $f_{mass\rightarrow string} = -f_{string\rightarrow mass} = 2R(v - v_l^+ - v_r^-)$. The impedance R is related to the admittance Y by $RY = 1$. Rearranging gives:

$$v = \frac{Y}{2} f_{mass\rightarrow string} - v_l^+ - v_r^- \tag{8.14}$$

This is just 8.6, the result stated in [SW95], with $v_h = v_l^+ + v_r^-$, so now we have an explicit derivation.

Calculation

At each iteration, the acceleration, a, of the mass is determined from the sum of string and bow forces, which is zero if the bow is sticking. The velocity is updated by applying an approximate integrating filter to the acceleration, with a finite response at zero Hz to prevent large DC offsets building up. A simple example of this is the filter defined by $v \leftarrow \beta(v + a\delta t) \quad 0 < \beta < 1$.

The outgoing velocity waves are determined using 8.7. Transition between states occurs when either the stringforce exceeds a threshold or the relative bow-string velocity crosses zero.

General Application

The waveguide-mass structure could be used to realistically model a variety of other musical interfaces - plucks, hits, scrapes and so forth, offering high efficiency combined with more realistic modelling of interface physics. Where mass positions are required these can be found by applying a ‘leaky integrator’ to the mass velocity, as described in [Smi92].

8.6 Abstract Dynamical Synthesis Control

Experimentation with a performance instrument which uses physical models inevitably leads to a comparison with traditional electronic performance instruments; for example keyboards and wind-controllers. Certainly the behaviour of physical modelling instruments is more complex, because the output at any time can depend in a very general way on the entire history of control input up to that point. On the older instruments the output usually depends on inputs at the current time and possibly a few isolated previous times, by use of the keys, pedals and modulation wheels.

Sound Qualities

When talking about the behaviour of a physical modelling instrument, two classes of description consistently arise. The first is to do with the *qualities* of the perceived sound. A sound quality can be viewed as some *perception* of sound which a listener *consistently recognizes*. *Pitch* is a very consistent quality which can be agreed on by different listeners and even ascribed a physical value. *Timbre* is equivalently defined as the collection of qualities not including pitch and loudness. Qualities such as *hollow* are not as well defined physically as pitch, but may still be consistently recognized by different listeners. Each of us recognizes many more qualities that cannot be described by simple analysis.[16]

Quality Control

The second class describes how qualities are varied by the performance control. This is effectively a factoring of the dynamic relationship between the sonic output and control into quality control and quality. The dynamics typically operate on timescales down to about 10ms, for example the quality of a note transition to finger movement near hole closure. The change in a quality actually serves to give definition and perspective to the perception of a quality. Furthermore, a quality comes from an

[16]This is a subjective observation and there appears to be no clear way of testing it. Rather than submerge in a stormy swell of experimental psychology, let the argument be seen as tentative support for the following proposals.

interval of time. If the control is changing the quality on a comparative time scale, then the listener may focus on different qualities that are more cleanly related to the control.

The separation into dynamics and quality can be viewed as a product of the psychoacoustic perception of sound, resulting from our need to understand the physical state of objects from their radiated sound. It suggests a more general approach for constructing synthesis systems for performance than used previously. See **Figure 8.22**.

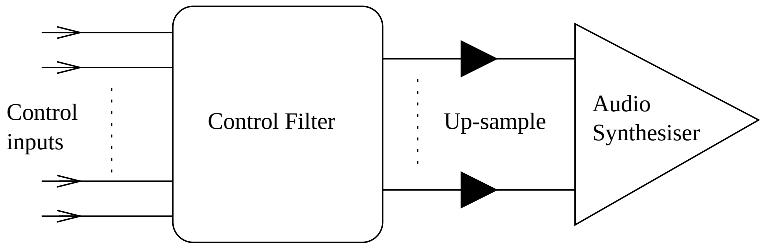


Figure 8.22: A model for instrument synthesis with control processing.

The control signals are processed by a general causal filter at the control rate. This simulates the dynamic aspects of quality control. The filter output is upsampled to audio rate and fed to the audio rate synthesis engine, which takes as inputs instantaneous quality measures. This is a simple model, but it immediately serves to enrich the dynamic behaviour of instruments using the older synthesis techniques such as frequency modulation. In principle the new model allows us to control the dynamic behaviour more easily than is the case for a physical modelling instrument, while giving access to a range of sound qualities which cannot be generated easily by means of classical waveguide networks. The model has the practical advantage that the control filtering consumes little cpu time in comparison with the audio synthesizer, because it is operating at a much lower sample rate. Also, the synthesizer can be of a much cheaper form than a waveguide network, whose efficiency is compromised by the need for memory to implement the delays.

8.7 Enhancements and Modifications

8.7.1 CyberWhistle II

In the second CyberWhistle version, *CWII*, a Bb penny whistle is fitted with a similar wooden chassis. The PCB is large enough to contain all the electronics in surface mount form. The bore is just wide enough to fit a pressure sensor[17] which replaces the microphone of the first version. **Figure 8.23** shows the new mouthpiece arrangement. The rubber stop can be moved a little to adjust the resistance, to the point where the flow can be stopped altogether. In this way the pressure sensor is more flexible than the mic sensor, although the mic is indirectly capable of sensing lower pressures. The pressure signal is converted to MIDI along with the finger sensor output. This is physically more convenient than the extra audio output of the mic sensor, but at the expense of

[17]Honeywell series 24PC

slightly increased latency.

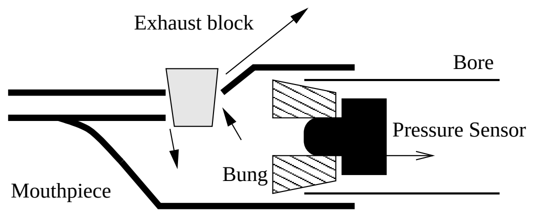


Figure 8.23: Mouthpiece for CyberWhistle II

The output from the chassis consists of a MIDI signal containing breath and finger information. An audio signal filtered from the pressure signal is also transmitted. This may optionally be used for using *singing* techniques to generate sonic modulation effects such as *growling* found in saxophone playing. Unifying the electronics not only simplifies construction, but eliminates oscillation problems associated with having extended leads from sensors to electronics. It is possible to mount a small battery in the bore of CWII. Alternatively power can be delivered remotely from the computer, or a small box carried in a pocket. The total power consumption is still only about 6ma.

8.7.2 Capacitative Mouthpiece

Experimentation with using electric field sensing for detecting finger movement, has led to the following application for measuring the player's embouchure. If the body is earthed via the fingers in contact with the metal case of the whistle, then the lower lip acts as a shunt when pressed against an insulated transmitting and receiving pair. If the lip is pressed harder, or moved closer to the sensor, the shunting action is increased. By measuring the signals from a number of receivers around one transmitter it is possible to deduce to some extent the position and pressure of the lower lip. The technique is robust, and because the lip is in close proximity to the sensor, elaborate noise elimination circuitry, such as synchronous transmit and receive, is not required.

Figure 8.24 shows some possible arrangements from which varying degrees of information can be deduced for different numbers of receivers. *a* simply indicates how much lip is held in close proximity to the sensor. If the pressure is applied to the lip this will increase the output by pressing parts of the lip even nearer. In *b* the outputs could be combined by summation or perhaps a more sophistocated function to give an overall output similar to *a*. An indication of position can then be derived by weighting either output against the summation signal. *c* is a natural extension of *b* to include the ability to measure sideways displacement. The weighting signal is a combination of all three outputs.

There is no particular need to derive exact physical measurements from the sensor outputs. All that matters is that signals can be derived from the outputs which can be controlled independently by the player. Complete independence is not possible, infact some interdependence serves to make the control process more interesting, as can be observed in the embouchure control of real instruments.

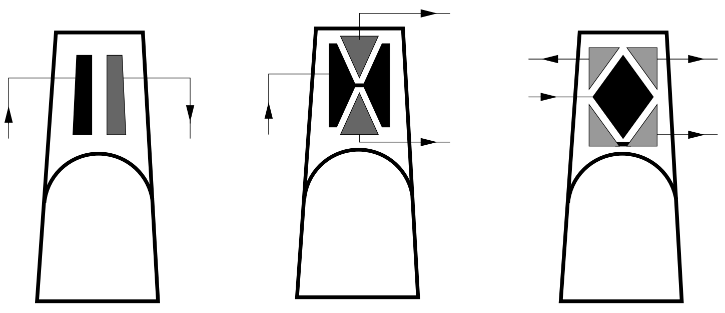


Figure 8.24: Example capacitative mouth sensor geometries.

8.7.3 Wireless control

The output cable of CWI and CWII, is a little inconvenient and may be argued to be aesthetically undesirable. Possible remote communication techniques include radio and infra red. Of these, infra red sensing offers simplest option for transmitting MIDI, but would be no use for the optional audio component. Battery consumption would be increased considerably for IR usage over the original CW circuit[18] but it will still be possible to support the new circuit with a small high performance battery for a useful length of time. The most suitable mounting for the IR transmitter is at the 'bell end' of the bore, using a wide dispersion pattern to allow maximum angular movement.

8.8 Summary

Design

It is hoped from the CD examples that the CyberWhistle can be judged to be successful in applying the finger shading technique to create a variety of familiar and not so familiar instruments. The overall design is inexpensive, robust and works conveniently with a computer capable of software synthesis. In principle a laptop computer could be used and this would greatly improve the portability of the instrument.

Engineering

The compensated finger sensor was shown to have stable operation over a wide range of ambient lighting, and when sampled linearly gives a suitable distribution of resolution for sensing shading technique. Two methods of breath sensing were described, one using the processing of noise from the mouthpiece another measuring pressure directly. Each has advantages, but direct pressure measurement is most practical because no additional audio output is required.

[18]For instance using the HSDL-1001 transceiver by Hewlett Packard with full peak current of 1 amp, 2μs pulse width and assuming 50% full MIDI rate usage, the average LED current consumption is 15ma. This gives a range of a few metres with transmission half-power angle of up to 60 degrees.

The MIDI link used for transmitting control information has sufficient bandwidth for transmitting the variety of gestures indicated on the CD. The main limitation is the rate at which the computer can read MIDI and convey it to the receiving process. The SGI Indy could sustain only about 1/4 of maximum MIDI rate before crashing, and also consumed a disproportionate amount of resources. MIDI performance on other desktop computers is known to be generally poor. It is likely that the situation will improve in the future however[19]. The total control to audio latency could be held within 10ms. Interpolation of control data is necessary to eliminate aliasing effects. To improve latency a zero-latency predictive interpolator was designed.

The software synthesis incorporated two highly simplified bore models with some standard waveguide elements. These formed the interface between the continuous finger control system and the rest of the synthesis system. Despite their simplicity they are responsible for the variety of tone and pitch found in the CD examples. A detailed waveguide model for mass-string interaction was worked out, but as yet not fully implemented. This has the potential for *efficient* simulation of more complex interface problems, for instance involving friction slipping and sticking.

The embouchure sensor described in the enhancements section shows potential for development, although the main problem will be to redesign the mouthpiece to incorporate the new sensors in an practical and ergonomic way. Total power consumption is very low, about 6ma. For 15ma extra an infrared transmitter could be added which would eliminate the cord.

General Observations

Physical modelling is a relatively new and fast developing area of audio synthesis. Many interesting models have been created, some very elaborate, but little has been done to develop appropriate interfaces. The use of physical modelling in keyboard synthesizers adds some ‘life’ to sounds, but the keyboard interface limits the range of performance expression.

The CyberWhistle provides an example of an interface which can make good use of relatively simple physical models, and demonstrates that expensive sensing technology is not required. More important is attention to the details of ergonomic design, and the intelligent distribution of available bandwidth and resolution. By contrast, the use of continuous control devices such as sliders and computer mice is convenient, but too simplistic to be of significant musical value.

In contrast with conventional synthesis systems which have traditionally sat apart from the control interface, the CyberWhistle hardware and synthesis software were developed together resulting in a stronger impression of an integrated instrument. Concentrating on programmable synthesis rather than mainly fixed hardware designs means that future changes can be made without having to compromise the overall design.

The interesting dynamic properties of physical models which become apparent with a good inter-

[19]For instance Microsoft have just announced the latest version of their ‘Direct’ MIDI API, which is claimed to eliminate previous problems.

face, can be considered more abstractly. In Chapter 5 general abstract models were proposed for processing keyboard input. The same models can be applied to fingersensor control, but this time with focus on the processing of continuous signals. Section 8.6 described one approach to this using the idea of *quality* in sound. The potential advantages over using explicit physical modelling are greater flexibility, ease of design and efficiency.

Chapter 9

Conclusion

Music Technology Validation

At first appearance Music and Technology are two very different subjects, the first concerned with aesthetics and the second with rigorous structure. The reason why they make such an intriguing couple appears to be that each subject shares as a secondary concern the primary concern of the other. This thesis is written for an engineering department, so naturally the bias must be towards technology. Unlike many engineering disciplines, however, music technology cannot be validated in pure engineering terms. Musical aesthetics must also play a part.[1] No apology is made for the lack of numeric data attempting to quantify the worth of the designs presented here. Measurements have been made where they are useful to establish the performance of techniques in comparison to techniques already used, but these are not the main results of the thesis. The value of this work lies in the identification of valuable design principles and the creation of methods which embody them. These have been validated in three ways; by supporting arguments, CD recordings and to a lesser degree the reactions of users and listeners.

Below is a summary of each piece of work, followed by a point by point list of claims for valuable original content. The central hypothesis stated in the introduction is particularly well supported - that electronic instruments can benefit from the use of high-level principles abstracted from traditional acoustic instruments. This provides a powerful simplifying guide for instrument designers, which is seldom employed.

Spiragraph

The intention was to use the MIDI keyboard with software synthesis, and develop an open architecture which did not rely on the note-paradigm. This was realized using some novel Csound techniques, including the division of processing between MIDI *instrs* and the main synthesis *instr*. Notes

[1]Many areas of *audio engineering* are validated in pure engineering terms, for instance the performance of lossless compression.

are a feature of many traditional instruments but they exist as a low-level abstraction rather than a physical reality. The note is then a convenient shorthand for a more subtle process. Use of note structures in electronic instruments, exemplified by MIDI, are explicit and describe the process exactly. So moving away from an explicit note model, paradoxically benefits from the principle of added complexity found in acoustic instruments.

The example synthesis routine had the feature of micro-unpredictability; that its overall features could be easily controlled whereas the audio details were much harder. In acoustic instruments this quality is called ‘liveliness’ and adds interest to playing.

LAmb

LAmb combined the idea of alternative keyboard use with specially designed digital control and audio processing for live diffusion. The control method provides a practical alternative to conventional diffusion methods, which avoids being simplistic, yet at the same time is intuitive. This is an important characteristic in acoustic instruments. The new Ambisonic processing methods provide a broader language for the diffusion artist which complements the control method. Traditional Ambisonic synthesis techniques make no allowance for the *size* of a sound object. LAmb addresses this with new techniques which are efficient and effective. More advanced techniques for modeling the orientation of objects are described, although not implemented in LAmb. Other new processes include a flexible and efficient means for generating spatialized delays and reflections, and a digital means for spreading the frequency content of a mono signal. A simplified, but effective, air filtering model is presented.

Spatial synthesis of this kind is based on audio engineering principles, but it has been important to establish a balance with musical considerations. In practice this has meant giving the performer the ability to deviate from pure realism. The method of deviation has been carefully built in to the structure of the control and signal processing.

SidRat

In the past reconstructing a work for live electronics involving several interacting elements has posed serious practical problems. SidRat is a hitherto unique demonstration that Csound can be used to realistically manage the situation by condensing all mechanical processes to portable script. Besides this, techniques are developed to ease the scripting process and handle control data. Several unusual instruments are designed which provide examples of scripted audio processing, including Ambisonic spatialisation. Conceptually, the value of SidRat is to bind the elements of generalised live electronics into an object-like instrument. This is not only very convenient, but promotes an integrated method of compositional thought which would otherwise be difficult. The natural human affinity for objects is a very important principle for traditional instruments.

CyberWhistle

New interfaces face a much more difficult route to acceptance than new software. They benefit by being simple and cheap to construct, reliable and at least partially familiar to the newcomer. The CyberWhistle achieves all this and carefully examines the use of MIDI for control communication, but its chief innovation is the finger detecting mechanism. This enables more expressive and musical control of imitative waveguide synthesis, which provides a natural bridge to the expressive control of more abstract waveguide synthesis.

The design of synthesis was fully integrated with every other aspect of design, so that the term CyberWhistle has come to mean the complete instrument, even though the software is not physically inside the instrument. Established waveguide techniques were augmented with experimental bore designs. Consideration of waveguide bow implementation led to a general technique for integrating masses with waveguides. Finally, a practical model for control-synthesis was presented which concentrates the dynamic qualities of an instrument into an efficient process separated from the audio synthesis. This technique is completely new to the literature.

Of all the projects, the CyberWhistle provides the clearest example of how acoustic instrument design principles can be combined with the flexibility of computer generated sound.

Specific Claims To Valuable Originality

The following list singles out specific claims to originality.

- Spiragraph gives an example of a new general approach to performing sound using a keyboard. One sound object is controlled using several keys. This is of practical as well as aesthetic importance because keyboards are the most abundant form of electronic controller.
- Original Csound program designs are described to smoothly implement the above.
- LAmb contains a variety of novel features. The logical starting point for these is a specific method for using a midi keyboard to control the dynamic localization of several sources simultaneously in performance. The method takes advantage of the natural pianistic and velocity control abilities of keyboard players.
- The diffusion of each source by LAmb is achieved using a novel algorithm based on Ambisonic theory which very efficiently creates the perception of object width both when the listener is inside and outside the object. The parametrisation of the object characteristics is carefully chosen to be intuitive, flexible and yet configurable for effects which depart from the natural acoustic behaviour.
- Another form of spreading is provided which spreads frequency components in the attempt to achieve a more realistic width spreading effect. This is essentially a synthesis of earlier analog

techniques, incorporating a cross-over in spreading algorithms as the object approaches the centre.

- An efficient recursive multichannel delay which can be configured to produce localized reflections for each source simultaneously. This can also be used to generate a variety of interesting abstract spatialized delay effects.
- An air filter model which efficiently produces accurate attenuation at high frequencies without the expense needed to satisfy a broad-band least-squares criterion.
- SidRat demonstrates the little-used power of Csound for the flexible and portable scripting of electronic performances. This is important because having flexibility and portability make it unique.
- In the process, SidRat contains specific examples of novel program architecture of general use in live electronics.
- The CyberWhistle is an embodiment of a successful and original electronic musical instrument. It also contains several important features which mark it apart from other electronic instruments-
- The finger control system continuously senses the position of the fingers, enabling more expressive control than switch based systems which are prevalent.
- The synthesis system based on waveguide techniques, is extended to interface appropriately with the finger control system.
- The physical construction. although simple, is carefully thought out to minimize component count while retaining a traditionally successful ergonomic interface, that of the penny whistle.
- The use of MIDI as a convenient transmission system is optimised beyond the traditional accepted performance.
- Additional synthesis algorithms have been devised which provide a general method for combining waveguide and mass models. These are expected to be useful.
- From consideration of waveguide systems, a generalised synthesis architecture is proposed which offers greater ease of design than waveguide systems yet retains some the interesting dynamic properties, and offers improved efficiency.

The Future

Suggestions have already been made for future work in earlier chapters. The most pressing need is for better, practical interfaces. Once a good interface becomes established, the synthesis component can evolve rapidly. Also, more imagination is required to develop solutions for *combining* controls with synthesis, avoiding simplistic models.

Designing electronic hardware for interfaces becomes easier year by year, and computers become smaller, cheaper and more powerful. A decisive factor in the growth of new instruments could be the inevitable emergence of low power general purpose programmable chips capable of sustaining audio synthesis. These would free designers from reliance on desktop computers, which are expensive, not-portable, unreliable and cannot be integrated into the physical instrument *or* inflexible DSP chips, which currently inhibit a more relaxed approach to synthesis design. As the technology becomes more accessible the community of commercial and academic developers will grow. With less time spent resolving synthesis implementation issues more energy can then hopefully be devoted to the central issues which have been discussed.

Appendix A

Spiragraph Code Listing

The following realtime Csound orchestras map the development of the *Spiragraph* MIDI keyboard instrument. The score *spiragraph.sco* can be used with all three orchestras.

A.1 spiragraph1.orc

One pitch, mono output.

```
;
; spiragraph1.orc
;
; A performance instrument for MIDI keyboard, in Csound.
; While held each key alters the pitch envelope,
; by addition of either a cycling or a one-shot offset.
; The offset amplitude is controlled by the key velocity.
; The mod and pitch wheels are used to control the centre
; pitch and the modulation timebase respectively.
;
; The behaviour of each key is easily changed by modifying the
; tables in spiragraph.sco


sr = 16000
kr = 2000
ksmps = 8
nchnls = 1

gip     = octcps(200)
gkp init gip ; reset pitch accumulator
gkvol   init .5 ; reset overall output volume
gkmod   init 64
gkbend  init 0


    instr 1  ; collect midi data on channel 1

inote   notnum
ivel    veloc
kbend   pchbend 1
kbend   = kbend +1
gkmod   midictrl 1

ipwave  table inote-36, 40, 0,0 ; get wavetable for pitch cycle
ipcps   table inote-36, 41, 0,0 ; get frequency of pitch modification cycle
ipat    table inote-36, 42, 0,0 ; attack time
ipdt    table inote-36, 43, 0,0 ; decay time
ivelf   table inote-36, 44, 0,0 ; vel sensitivity flag
iadj    = 1
if iadj =0 igoto nokeysens
```

```
iadj    = ivelf * ivel/128

nokeysens:
kp  oscili 1, ipcps*kbend, ipwave
kp  linenr  kp, ipat, ipdt, .01 ; attack and decay can be stored params.

gkp = gkp + kp * iadj ; could use aftertouch here..

    endin



    instr 100  ; synthesiser

gkmods  port    gkmod, .04
a1  oscili   4000, cpsoct(gkp+(gkmods-64)/20), 1

out a1
gkp = gip

    endin
```

A.2 spiragraph2.orc

One pitch, stereo output.

```
;
; spiragraph2.orc
;
; A performance instrument for MIDI keyboard, in Csound.
; While held each key alters the pitch envelope,
; by addition of either a cycling or a one-shot offset.
; The offset amplitude is controlled by the key velocity.
; The mod and pitch wheels are used to control the centre
; pitch and the modulation timebase respectively.
; The pattern of stereo output is generated by summing the
; stereo patterns for the keys held.
;
; The behaviour of each key is easily changed by modifying the
; tables in spiragraph.sco


sr = 16000
kr = 1000
ksmps = 16
nchnls = 2

gip  = octcps(200)
gkp init gip ; reset pitch accumulator
gila = 1
gkla init gila
gira = 1
gkra init gira
gkvol init .5 ; reset overall output volume
gkmod init 64
gkbend init 0


instr 1  ; collect midi data on channel 1

inote notnum
ivel veloc
kbend pchbend 1
kbend = kbend +1
gkmod midictrl 1
gkvol chpress 10000

ipwave table inote-36, 40, 0,0 ; get wavetable for pitch cycle
ipcps table inote-36, 41, 0,0 ; get frequency of pitch modification cycle
ipat table inote-36, 42, 0,0 ; attack time
ipdt table inote-36, 43, 0,0 ; decay time
ivelf table inote-36, 44, 0,0 ; vel sensitivity flag
```

```
iadj = 1
if ivelf = 0 igoto nokeysens
iadj = ivelf * ivel/128

nokeysens:

ilwave table inote-36, 50, 0,0 ; get wavetable for left amplitude cycle
ilcps table inote-36, 51, 0,0 ; get frequency of left amplitude modification cycle
ilat table inote-36, 52, 0,0 ; attack time
ildt table inote-36, 53, 0,0 ; decay time

irwave table inote-36, 60, 0,0 ; get wavetable for right amplitude cycle
ircps table inote-36, 61, 0,0 ; get frequency of right amplitude modification cycle
irat table inote-36, 62, 0,0 ; attack time
irdt table inote-36, 63, 0,0 ; decay time

kp oscili 1, ipcps*kbend, ipwave
kla oscili 1, ilcps*kbend, ilwave
kra oscili 1, ircps*kbend, irwave

kp linenr kp, ipat, .01 ; attack and decay can be stored params.
kla linenr kla, ilat, ildt, .01
kra linenr kra, irat, irdt, .01

gkp = gkp + kp * iadj ; Add pitch offset for current key.
gkla  = kla + gkla ; Add left amplitude offset for current key.
gkra  = kra + gkra ; Add right amplitude offset for current key.

endin



instr 100  ; synthesiser

gkmods port gkmod, .2
gkvols  port    gkvol, .2
a1 oscili   gkvols, cpsoct(gkp+(gkmods-64)/20), 1

outs a1*gkla , a1*gkra
gkla = gila ; Reset left amplitude.
gkra = gira  ; Reset right amplitude.
gkp = gip ; Reset to default pitch.

endin
```

A.3 spiragraph3.orc

Two pitches, stereo output.

```
;
; spiragraph3.orc
;
; A performance instrument for MIDI keyboard, in Csound.
; While held each key alters two pitch envelopes,
; by addition of either cycling or one-shot offsets.
; The offset amplitude is controlled by the key velocity.
; The mod and pitch wheels are used to control the centre
; pitch and the modulation timebase respectively.
; The pattern of stereo output is generated by summing the
; stereo patterns for the keys held.
;
; The behaviour of each key is easily changed by modifying the
; tables in spiragraph.sco


sr = 16000
kr = 1600
ksmps = 10
nchnls = 2

gip     = octcps(200)
gkp init gip ; reset pitch accumulator
```

```
gkp2    init gip
gila    = 1
gkla    init gila
gira    = 1
gkra    init gira
gkvol   init .5 ; reset overall output volume
gkmod   init 64
gkbend  init 0


    instr 1  ; collect midi data on channel 1
inote   notnum
ivel    veloc
kbend   pchbend 1
kbend   = kbend +1
gkmod   midictrl 1

ipwave  table inote-36, 40, 0,0 ; get wavetable for pitch cycle
ipcps   table inote-36, 41, 0,0 ; get frequency of pitch modification cycle
ipat    table inote-36, 42, 0,0 ; attack time
ipdt    table inote-36, 43, 0,0 ; decay time
ivelf   table inote-36, 44, 0,0 ; vel sensitivity flag
iadj    = 1
if iadj =0 igoto nokeysens
iadj    = ivelf * ivel/128

ip2wave table inote-36, 70, 0,0 ; get wavetable for pitch cycle
ip2cps  table inote-36, 71, 0,0 ; get frequency of pitch modification cycle
ip2at   table inote-36, 72, 0,0 ; attack time
ip2dt   table inote-36, 73, 0,0 ; decay time

nokeysens:
ilwave  table inote-36, 50, 0,0 ; get wavetable for left amplitude cycle
ilcps   table inote-36, 51, 0,0 ; get frequency of left amplitude modification cycle
ilat    table inote-36, 52, 0,0 ; attack time
ildt    table inote-36, 53, 0,0 ; decay time

irwave  table inote-36, 60, 0,0 ; get wavetable for right amplitude cycle
ircps   table inote-36, 61, 0,0 ; get frequency of right amplitude modification cycle
irat    table inote-36, 62, 0,0 ; attack time
irdt    table inote-36, 63, 0,0 ; decay time

kp  oscil 1, ipcps*kbend, ipwave
kla oscil 1, ilcps*kbend, ilwave
kra oscil 1, ircps*kbend, irwave

kp  linenr  kp, ipat, ipdt, .01 ; attack and decay can be stored params.
kp2 linenr  kp2, ip2at, ip2dt, .01
kla linenr  kla, ilat, ildt, .01
kra linenr  kra, irat, irdt, .01

gkp = gkp + kp * iadj ; could use aftertouch here..
gkp2    = gkp2 + kp2 * iadj
gkla    = gkla + kla
gkra    = gkra + kra

    endin


    instr 100  ; synthesiser

gkmods  port    gkmod, .04
a1  oscili   4000, cpsoct(gkp+(gkmods-64)/20), 1
a2  oscili  4000, cpsoct(gkp2+(gkmods-64)/20), 1
a3  = a1 + a2
outs    a3*gkla , a3*gkra
gkla    = gila
gkra    = gira
gkp = gip
gkp2    = gip

    endin
```

A.4 spiragraph.sco

```
;
; spiragraph.sco
;
; Csound score for spiragraph.orc keyboard instrument.
; The instrument can be quickly reconfigured by changing the
; table entries.


f1 0 8192 10 1

;Pitch offset control: key -> control param index.
f40 0 16 -2   82 82 82 83    83 1 88 88      84 84 84     85 86  87 0 0   ; wavetable nu
f41 0 16 -2   5.1 5.2 20 5.4 190 15 5.7 5.8  5.9 6.0 6.1  0 0 0          ; frequency
f42 0 16 -2   0 1 0 0  2 0 0 3  0 0 0 0  0 0 0 0 0 0                      ; attack times
f43 0 16 -2   .2 .2 1 .2  .2 1 .2 .2  .2 1 .2 .2  .2 .2 .2 .2 .2 .2       ; decay times
f44 0 16 -2   1 1 1 1  1 1 1 1  1 1 1 0  0 0 0 0                          ; velocity sensitivity

;Pitch offset for second oscillator (spiragraph3.orc)
f70 0 16 -2   1 84 84 82    83 1 84 84      84 84 84     85 86  87 0 0   ;wavetable
f71 0 16 -2   3 4 4 5 600 7 8 15 5.7 5.8  5.9 6.0 6.1  0 0 0 ;frequency
f72 0 16 -2   0 1 0 0  2 0 0 3  0 0 0 0  0 0 0 0 0 0 ; attack times
f73 0 16 -2   .2 .2 1 .2  1 1 1 1  1 1 1 1   1 1 1 1 1 1 1 ; decay times

;Left channel amplitude cycle control.
f50 0 16 -2   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1           ; wavetable nu
f51 0 16 -2   .1 1 2 3 .2 5 6 7 8 9 10 0 0 0 0 0        ; frequency
f52 0 16 -2   0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0       ; attack times
f53 0 16 -2   0 0 1 0 0 3 1 0 0 0 0 0 0 0 0 0 0 0       ; decay times

;Right channel amplitude cycle control.
f60 0 16 -2   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
f61 0 16 -2   .2 2 3 4 .3 6 7 8 9 10 11 12 0 0 0 0 0 0
f62 0 16 -2   .5 0 1 0 .5 0 0 0 0 0 0 0 0 0 0 0 0 0
f63 0 16 -2   0 2 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0


;Wavetables for use by the pitch offset and amplitude offset oscillators.
f82 0 8192 -7  -1  96  1 4000  1 96 -1 4000 -1
f83 0 8192 -7  -1 4096 1 4096 -1
f84 0 8192 -7  -1 8000 5 192 -5
f85 0 2    -2   1.1  1.1
f86 0 2    -2   1.2  1.2
f87 0 2    -2   1.3  1.3
f88 0 8    -2   -2 -1.5 -1 -.5 0 .5 1 1.5



i100 0 3600
```

Appendix B

LAmb Tutorial

Appendix C

LAmb Userguide

Appendix D

Screenshots of LAmb

Appendix E

LAmb Code Listing

E.1 Makefile

```
#
# Makefile  for LAmb v1.05
#
# lt.c         : Main audio.
# lt.decode.c  : B-format decoding.
# f.c          : Xforms primary setup.
# f_cb_.c      : Xforms call back functions.

OPT = -O2 -mips2 -s


ltf : lt.o f.o u.o
    cc $(OPT) -o lamb  lt.o f.o u.o      \
    /usr/local/lib/xforms/libforms.so   \
    -lX11 -lmd      \
    -laudio -laudiofile -laudioutil     \
    -lm   -I/usr/local/include/xforms   \
    -rpath /usr/local/lib/xforms/
# realtime loader path.


lt.o    : lt.c f_cb_.c lt.h lt.decode.c f.h
    cc  $(OPT)  -o lt.o -c lt.c \
    -I/usr/local/include/xforms/

f.o : f.c  f.h
    cc $(OPT) -o f.o -c f.c -I/usr/local/include/xforms/
```

E.2 lt.h

```
/*
 * lt.h
 *
 * Header for main audio functions,  LAmb v1.05
 *
 */

#include <dmedia/audio.h>
#include <dmedia/audiofile.h>
#include <dmedia/midi.h>
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#include <limits.h>     /* Process control. */
```

```
#include <sys/types.h>
#include <sys/prctl.h>
#include <sys/schedctl.h>
#define _BSD_SIGNALS
#include <signal.h>

#include <sys/fpu.h>
#include <sys/cachectl.h>
#include <errno.h>



/*DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD*/
/*DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD*/
/*DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD*/

#define OUTPUT_QUAD_DECODED
#define QUAD_FIXED_POSITIONS
#define EXTERNAL_MIDI
#define CHANNEL_MODE AL_4CHANNEL
#define AUDIO_LINES 4

#define AUDIO_INPUT            AL_INPUT_LINE
#define IN_BUFF                3000
#define OUT_BUFF               3000
#define CHANNEL                1       /* MIDI receive channel */
#define SF_PATH                "."
#define RECORD_FILE     "/usr/tmp/record.aiff"
#define AUDITION_FILE       "/usr/tmp/record.aiff"
#define PLAY_FILE       "/usr/tmp/play.aiff"

#define DEL1            800 /* Spread delay sizes. */
#define DEL2            500

#define CONTROL_OFF        0
#define CONTROL_ON             1

/**********************  CONTROL ADDRESSES  *************************/
/**************** Filtered control signal section. ******************/
#define CONTROL_BASE            20      /* Controller number. */
#define CONTROL_TOP             108      /* Top of filtered control signals */

#define SLOW_ADDRESS        18
#define SLOW_DATA       19
#define LOCATE_BASE     20  /* Check c_base consistency. */
#define ROTATE_BASE     60
#define W_MASTER        63
#define Z_MASTER        64

#define W_ADJUST_BASE       67  /* Channel bases */
#define Z_ADJUST_BASE       71
#define VOLUME_BASE             75
#define OBJECT_SIZE_BASE    79
#define BUMPINESS_BASE          83

#define LISTENER_BASE       87  /* Coord base */

#define EQ_DEPTH_BASE           90  /* Channel base */
#define SPREAD_SCALE_BASE   94
#define SPREAD_MAX_BASE     98
#define SPREAD_TIME_BASE    102

#define DOMINANCE_BASE      106
/* Beware CONTROL_TOP */

/************ Unfiltered slow control signal section. **************/
#define DELAY_DIRECT_LEVEL  0
#define DELAY_TAP_LEVEL_BASE    1
#define DELAY_TAP_FEEDBACK_BASE 4
#define DELAY_TAP_TIME_BASE 7
#define POSITION_KEY_BASE   10
#define INPUT_SOURCE        34
#define OUTPUT_FORMAT       35
#define AUDIO_RATE      36
#define PERMUTATION     37
#define X_POLARITY      38
```

```
#define Y_POLARITY      39
#define Z_POLARITY      40
#define MIDI_SOURCE     41
#define MIDI_KEYBOARD_TYPE  42
#define MIDI_CHANNEL        43
#define QUAD_ASPECT     44
#define MASTER_TIME     45
#define BUFFER_IN_SIZE      47
#define BUFFER_OUT_SIZE     48
#define INPUT_LEVEL_BASE    49
#define PANIC           53
#define GUI_OFF         54
#define RESET_DELAY     55
#define RESET_ROTATE        56
#define RESET_SAMPLE        57
#define RESET_LISTENER      58
#define RESET_DOMINANCE     59
#define BLANK1          60
#define BLANK2          61
#define BLANK3          62
#define BLANK4          63
#define JUMPINESS_BASE      64
#define JUMP_MODE_BASE      68
#define SAMPLE_KB       72
#define KEY_FEEL        73  /* ! channel base */
#define RECORD_FORMAT       77
#define DELAY_TAP_ROTATE_BASE 78
#define NEXT 81


#define NULL_BASE              0
#define ROTATE_LINE            10

#define AUDIO_BLOCK_FRAMES  256 /* Size of software buffer. */
#define AUDIO_BLOCK_SAMPLES 1024    /* - reduces OS calls. */
#define CONTROL_PERIOD      15      /* Control period multiplier */
#define FORM_PERIOD         200
#define FILTER_CONTROL_PERIOD   4
#define LOCATE_CONTROL_PERIOD   4
#define ROTATE_CONTROL_PERIOD   8
#define DOMINANCE_CONTROL_PERIOD    9
#define CLIP_PERIOD     200
#define FOLLOW_JUMP             .8f      /* Factor used when reading MIDI controllers */
#define VOLUME_JUMP     .05f
#define KFILTER_FACTOR          .1f
#define FILTER_INCR             .5f
#define FILTER_DIVIDE           100

#define CARTS_UNIT_RADIUS      20
#define POLAR_UNIT_RADIUS      20
#define CYL_UNIT_RADIUS        20
#define CYL_UNIT_HEIGHT        20

#define RADIUS                 1.0     /* Determines spread of sound. */
#define LOUDNESS               1.0    /* Determines max output level. */
#define BUMPINESS          1.0
#define W_FACTOR               0.707   /* See locate() */


/**********  MIDI key control addresses  *****************/

#define MAX_VOICES             10
#define SAMPLE_KEY_BASE        91   /* Where the sample playback starts */
#define ROTATE_KEY_BASE        84   /* Rotate control keys */
#define SPACE_KEY_BASE         36   /* Where space control starts */


#define PI                     3.1412659
#define DELAY_FRAMES           100000
#define DEBUG_SAMPLES       1000

#define MOD8192(a)             ((a<0) ? ((a+32768)%8192) : (a%8192))
                   /* Provides modulo 8192 for +-ve shorts for use in rotate() */
```

```
typedef float ltfloat;        /* For use in lt.c */

/*PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP*/
/*PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP*/
/*PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP*/

void main(int, char *[]);
void setaudio(void);
void setmidi(void);
short user_midi_getc(short);
void loadsf(void);
void set_tables(void);
void set_controls(void);
void set_output_format();
void checkmidi(void);
void control_process(short, short);
void slow_control_process(int, ltfloat);
void kfilter(void);
void locate(void);
void rotate(void);
void sample_playback(short,short);
void space_man(short,short);
void rotate_man(short,short);
void copyright(void);
short joystick_base(short);
void initialize_lamb_form(void);
void finish_form_design(void);
void set_values_in_form(void);
void write_fixed_positions_to_form(void);
char *itos(long);
char *ftos(ltfloat);
void set_channel_stuff(int);
void recall_positions(short);
ltfloat fget_ltfloat(FILE*);
short fget_short(FILE*);
void fput_ltfloat(FILE*, ltfloat);
void fput_short(FILE*, short);
void load_setup(const char*);
void save_setup(const char*);
void input_thread(void*);
void start_thread(void);
void stop_thread(void);
int  at_close(FL_FORM*, void*);
void flush_all_underflows_to_zero(void);
void dominate(void);




/*GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG*/
/*GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG*/
/*GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG*/

ltfloat  control[128];      /* MIDI control received. Used for files and writing to GUI*/
ltfloat  target[128];       /* Transformed control[] value which filter[] aims for. */
ltfloat  filter[128];       /* Filtered controller values, for smoothing and position key effects. */
ltfloat  slow_control[128]; /* 'Slow' MIDI controls sent with 2 MIDI control messages. */
short  status[128];
ALport audio_out;      /* Audio port addresses */
ALport audio_in;
   short in_audio_block[AUDIO_BLOCK_FRAMES][4];     /* For audio input */
   short out_audio_block[AUDIO_BLOCK_FRAMES][4];   /* For audio/record output */
   short bf_audio_block[AUDIO_BLOCK_FRAMES][4];    /* For raw bformat record output */
   short aud_audio_block[AUDIO_BLOCK_FRAMES][4];    /* For audition input */
ltfloat  Sin[8192],Cos[8192];
ltfloat  volume_scale[128];
short  locate_mode[4]     /* Final locate processing defaults to carts */
       ={1,1,1,1};
short  key_locate_mode[4]  /* Indicates keyboard mode: fixed pos(1), cart.s(2), cyls(3) or polars(4) */
       ={1,1,1,1};
short  c_base[4][4]         /* Controller base numbers for locating lines quickly. */
       ={20,20,23,26, 30,30,33,36, 40,40,43,46, 50,50,53,56};

ltfloat     wf[4]={0},xf[4]={0},yf[4]={0},zf[4]={0};     /* Location multipliers */
ltfloat     new_wf[4],new_xf[4],new_yf[4],new_zf[4];   /* -for implementing zero cross mult. */
```

```
short  rotate_mode = 0;     /* Rotate use indicator. */
ltfloat     m1,m2,m3,m4,m5,m6,m7,m8,m9;  /* Rotation matrix X,Y,Z*/
ltfloat     d1, d2, d3, d4, d5, d6, d7, d8, d9; /* Dominance matrix W,X,Y */

char   sample_status = CONTROL_OFF;
short  *sf[20] = {NULL};                /* Pointers to soundfield recordings. */
char   filepath[128];
long   sf_size[20];
short  sample_key_voice[128];
short  voice_state[MAX_VOICES];
short  voice_key[MAX_VOICES];
short  voice_sf[MAX_VOICES];
short  voice_stack[MAX_VOICES];
short  voice_stack_pointer = 0;         /*  Stack is 'full' initially.  */
short  *voice_sample_pointer[MAX_VOICES];
long   voice_sample_counter[MAX_VOICES];
short  voice_volume[MAX_VOICES];



ltfloat  kfilter_mode[128] = {0};             /*  Individual filtering for each controller.  */
ltfloat  kfilter_jump[128] = {FOLLOW_JUMP};

ltfloat  cubic_fixed_positions[24] = {         /*  Fixed position carts */
   1,1,-1,  1,-1,-1,  -1,-1,-1,  -1,1,-1,
   1,1,1,   1,-1,1,   -1,-1,1,   -1,1,1 };

ltfloat  quad_fixed_positions[24] = {
   1,1,0, 1,-1,0, -1,-1,0, -1,1,0,
   .3,.3,0,  .3,-.3,0,  -.3,-.3,0,  -.3,.3,0 };

ltfloat hex_fixed_positions[24] = {
   1,1, 1,    -1,0,0,      -.5,.87,0, -.5,-.87,0,
   0,0,0,     .5,-.87,0,  .5,.87,0,   1, 0, 0     };

ltfloat  vel_jump_map[128];         /* Convert velocity to control filter values. */
ltfloat  vel_push_map[128];

short  joystick_line = 0;    /* ie carts, line 1 */

ltfloat     delay_line[DELAY_FRAMES][4];
long    delay_count[4] = {0,0,0,0};
short   delay_status = CONTROL_OFF;
ltfloat delay_unit = DELAY_FRAMES/128/128;



FD_LAmb *lamb_form;

long    audio_rate_table[8] = { 0, 8000, 11025, 16000, 22050,
                32000, 44100, 48000 };

char    form_status = 1;
short   form_channel = 0;       /* Audio channel being displayed. */
short   form_position_key =1;

long    ticks;      /* Tick count for passing to the user midi function. */

MDport   MidiPort = NULL;

ltfloat delay_direct_level = 1;
ltfloat delay_tap_level1 = 0;
ltfloat delay_tap1 = 0;
ltfloat delay_tap_feedback1 = 0;
ltfloat delay_tap_level2 = 0;
ltfloat delay_tap2 = 0;
ltfloat delay_tap_feedback2 = 0;
ltfloat delay_tap_level3 = 0;
ltfloat delay_tap3 = 0;
ltfloat delay_tap_feedback3 = 0;
ltfloat delay_tap1_cos = 1;
ltfloat delay_tap1_sin = 0;
ltfloat delay_tap2_cos = 1;
ltfloat delay_tap2_sin = 0;
ltfloat delay_tap3_cos = 1;
ltfloat delay_tap3_sin = 0;

ltfloat     squarex=1, squarey=1;
ltfloat     w_master=1;
ltfloat     z_master=1;

ltfloat     gaw,gax,gay,gaz;
ltfloat     *pa1, *pa2, *pa3;
ltfloat     x_polarity, y_polarity, z_polarity;
int     output_format;
short       *record_out_block;

int thread_pid =-1;
short   data[3];            /* Temporary store for a midi message */
short   running_status = 0;   /* ie no initial status */



short   debug_wave[DEBUG_SAMPLES];
long    debug_index=0;

short   plus_clips = 0;
short   old_plus_clips = 0;
short   minus_clips = 0;
short   old_minus_clips = 0;

short   dominance_status = 0;
short   listener_status = 0;

ltfloat follow_jump[4] = { FOLLOW_JUMP };  /* NB it appears this syntax only works ok in C++ */
char    jump_mode[128] = { 0 }; /* Initially no posn snapping. */

AFfilehandle record_fh = AF_NULL_FILEHANDLE;
AFfilehandle play_fh = AF_NULL_FILEHANDLE;
char    record_file[128];
char    play_file[128];
int master_mute = 0;
int record_state = 0, record_format,  play_state = 0;
int in_frame_channels = 4;

ltfloat eqa[4] = {1, 1, 1, 1}, eqb[4] = {0, 0, 0, 0};       /* Distance EQ parameters. */
ltfloat spread[4] = {0, 0, 0, 0};
int spread_time;

int audition_state =0;      /* 0 or record_format */
char    audition_file[128];
AFfilehandle audition_fh = AF_NULL_FILEHANDLE;

/************ Icon Pixelmap, coded in C for user simplicity *****************/
static char *icon_xpm[] = {
/* width height num_colors chars_per_pixel */
"    85    67       2           1",
/* colors */
". c #f8ffff",
"# c #000000",
/* pixels */
".....................................................................................",
".....................................................................................",
".....................................................................................",
".....................................................................................",
".....................................................................................",
".....................................................................................",
".....................................................................................",
".....................................................................................",
".....................................................................................",
".....................................................................................",
".....................................................................................",
"..................................############.......................................",
".................................###..####...#########...............................",
"................................###.............##...###.............................",
"................................##....................####...........................",
".............................####.....#.................###..........................",
"...........................#######....#...................##.........................",
"..........................##...###....#...................####.......................",
".........................##......#...##...#................###.......................",
"........................#####........##...#.......##.........##......................",
"........................###.###.........##.......###..........##.....................",
".......................##.#...#.......#.........###............#.....................",
".......................#......#........##.......#..............##....................",
"......................##.......................#................##...................",
"......................##................##.....#...#.##.........##...................",
"......................###........#..#..............######........#...................",
"......................###.#.#....#.................#...##........##..................",
"......................###.#.......................##....##.......##..................",
".....................###..........................####..##........##.................",
"....................###....###........#..........#####...#........##.................",
"....................##....##.....#...#...........#####...#.........##................",
"....................#....##..........##...........####..##.........###...............",
"...................###..##.....#.##...............####..#............#####...........",
"....................##.##.......###...#.#.........###...#..............#####.........",
"....................####.........##.....#.#........######.................####.......",
"....................###..........#.#....#.#..........##.....................###......",
"...................##...........##..#.#.......................................#......",
"..................##............#....#...................................###..##.....",
".................##............##.#...#...................................###.##.....",
"...............###............##..##.###..................................###.##.....",
".............###............###.......#.......................................##.....",
"...........####...........####..#...........#..............###.................##.....",
".........####...........######......#.#.#................###.................##......",
"........###..........########.####....#...#..............####................##......",
".......##...........#########.............................#..#.#............##.......",
"......##..........######.###########......................#......####....######......",
"......###......######...###.#......#.......#..........................####.##........",
".......##.....#####...####..#....##.......#...............................###........",
"........##...........##..########.##....................................###..........",
".........#####....####....####....#....................................###...........",
"...........########.........###.##..................................###..............",
".............................####..............#...............######................",
"..............................#########...........#......#.######....................",
"..............................####....................###..###.......................",
"..............................###..#..................#...###........................",
"..............................#...................###...###..........................",
"..............................#####.....................##...........................",
"..............................##........................##...........................",
".............................#####.....................#.#...........................",
".............................##.......................#..#...........................",
"............................##......................##.###...........................",
"............................##.......................##.##...........................",
".........................#####.......##...........###..##............................",
"........................#######.......#.......##.....####............................",
".......................####....#...................##..##............................",
"......................#####.....##...................####............................",
"......................##..##.........#..............###.#............................"
};
```

E.3 lt.c

```
/*
 * lt.c
 *
 * Main audio functions for LAmb v1.05
 *
 */


#define VERSION "V1.05"

#define ICON   /* Cute icon. */


/* Deleting this cuts out spreading and audition functions. */
#define FULLCODE

/************************************************************************/


#include "forms.h"      /* Xforms header. */
#include "f.h"          /* lamb form header. */
#include "lt.h"         /* lamb header. */
```

```
#include "f_cb_.c"      /* Callback functions. */

/* ( extra _ is to avoid fdesign overwriting f_cb. ) */



/*MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM*/
/*MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM*/
/*MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM*/

void main(int argc, char *argv[])
{
   short frame;
   short *frame_base, *out_frame_base, *bf_frame_base, *aud_frame_base;
   /* Input and output software buffer pointers. */
   /*  An idea would be to offset initial counts to even out
       processing as much as possible.  */
   int control_count=0;           /* The main control rate counter */
   int filter_control_count=0;
   int locate_control_count=0;
   int rotate_control_count=0;
   int dominance_control_count=0;
   int form_count=0;
   int clip_count=0;

   int n,m,line;
   ltfloat last_aeq[4] = {0, 0, 0, 0}, aeq, adc[4] = {0, 0, 0, 0}, aspread, amono;
   short *sample_pointer, *shp;
   ltfloat *flp;
   short voice,sf_num;
   long  vol;
   ltfloat feedback;

   ltfloat  delay1[1270][4] = {0};  /* For use by the spreader. [frame][channel] */
   ltfloat  *del1p = &(delay1[0][0]);   /* Index and counter */
   int  del1c = 0;

   ltfloat  ga1,ga2,ga3;
   ltfloat  fbw,fbx,fby,fbz, tax, tay;
   ltfloat  aout0, aout1, aout2, aout3;  /* Final output. */

/* Constants for hexagonal decoding. */
   const ltfloat hex1=sqrt(3/2),hex2=1/sqrt(2),hex3=sqrt(2);

   short shift;
   short pg;
   ltfloat* testmem;

/*IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII*/
/*IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII*/
/*IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII*/
/*IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII*/


/* Old hex decode values .. too small.
   hex1 = sqrt(3)/2;
   hex2 = 1/2;
   hex3 = 1;
*/
/*
   printf("Schedctl NDPLOMIN return: %d\n\n", schedctl(NDPRI, 0, NDPLOMIN ));
   printf("Schedctl NDPLOMIN return: %d\n\n", schedctl(NDPRI, 0, NDPLOMIN ));
*/

if (schedctl(NDPRI, 0, NDPHIMIN) == -1 )
    printf("\nPermission for non-degrading priority not granted!\n");

   flush_all_underflows_to_zero();
   make_trig();                         /* Create sin, cos tables */
   set_controls();
   set_tables();
   setaudio();

   n = argc;
   while(n>1) { if (strcmp(argv[--n], "-nogui")==0) form_status=0; }
   if (form_status !=0){
    fl_initialize(&argc, argv, 0, 0, 0);
    initialize_lamb_form();
    }

   load_setup("init.set");
   loadsf();
   copyright();


/* cachectl test: */
/*
    pg = getpagesize();
    testmem = (ltfloat*)memalign(pg, 10000);
    printf("testmem = %x\n", (long)testmem);
    if (cachectl(testmem, (10000/pg)*pg, UNCACHEABLE)==-1) perror (NULL);
*/

/*LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL*/
/*LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL*/
/*LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL*/
/*LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL*/
/*LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL*/
/***************  MAIN AUDIO LOOP *************************************/
   while(1)
   {

    if (play_state == 1) {
        if (AFreadframes(play_fh, AF_DEFAULT_TRACK,in_audio_block, AUDIO_BLOCK_FRAMES) ==0) {
        fl_set_button(lamb_form->record, 0);
        fl_set_button(lamb_form->play, 0);
        play_cb(lamb_form->play, 0);
        record_cb(lamb_form->record, 0);
        }
    }

    else    ALreadsamps(audio_in, in_audio_block, AUDIO_BLOCK_SAMPLES);  /* 4 samps per frame. */

    if (audition_state != 0) {
        if (AFreadframes(audition_fh, AF_DEFAULT_TRACK, aud_audio_block, AUDIO_BLOCK_FRAMES) ==0) {
        fl_set_button(lamb_form->audition, 0);
        audition_cb(lamb_form->audition, 0);
        }
    }

    for(frame_base=&(in_audio_block[0][0]), out_frame_base=&(out_audio_block[0][0]),
    bf_frame_base = &(bf_audio_block[0][0]), aud_frame_base = &(aud_audio_block[0][0]),  frame=0;
    frame< AUDIO_BLOCK_FRAMES;
    frame_base+=in_frame_channels, out_frame_base+=4, bf_frame_base+=4, aud_frame_base+=4, frame++) {

     gaw=gax=gay=gaz=0;


/*********************  Object Pan Mixing  ***********************/

     for(line=0; line< in_frame_channels; line++)       /* in_frame_channels */

            /*   if (locate_mode[line]> -1); maybe */
            /* ie If the new_wf etc are set very low: avoids a nasty jump. */

     if (control[VOLUME_BASE+line] > 0)
     {
           amono =   *(frame_base + line);  /*in_audio_block[frame][line];*/



#ifdef FULLCODE
       /*
        * DC component removal.
        * Note that a main cause of glitching when mixing,  is the signal having a DC component
        * comparable to the amplitude of the signal.
        * NB These do take up quite a bit of processing time..
        */

       amono -= ( adc[line] = .00001 * amono + .99999 * adc[line] );
       /*
        * Distance EQing
        */

       if (eqb[line] > 0)
        amono = eqa[line] * amono  + eqb[line] * last_aeq[line];
#endif


        /* Zero crossing detection.
         * This may not work very well if
         * the signal has any DC.. hence DC removal.
         */

       if (amono < 0 && last_aeq[line] > 0) {
/*
printf("L225 %d\n", aeq);
*/
            wf[line] = new_wf[line];
            xf[line] = new_xf[line];
            yf[line] = new_yf[line];
            zf[line] = new_zf[line];
            }

       last_aeq[line] = amono;  /* This serves zero-crossing and the filtering. */


#ifdef FULLCODE

       /*
        * Spectral spread.
        */
        /*
         *
         *      aspreadx = ( *del1p + amono ) * spread[line]
         *      aspready = ( *del1p - amono ) * spread[line]
         *
         *  Implemented for efficiency by adding in Locate():
         *
         *      yf[line] += spread[line]
         *      xf[line] -= spread[line]
         *      wf[line] needs to be smaller in centre if spread is on.
         */

       aspread = 0;
       if ( spread[line] > 0 ) {
        aspread =  *del1p * spread[line];
        *(del1p++) = amono; /* Delay line access is expensive..Hence we like to avoid it. */
                    /* NB really need separate delay lines for each channel */
       }
#endif


        /* NB rotation to map 'graph paper x-y' */
            /* onto 'B-format x-y' orientation.     */


            gaw += amono * wf[line];
           gax += amono * yf[line]  + aspread; /* NB xf, yf include spread info. */
           gay -= amono * xf[line]  + aspread;
           gaz += amono * zf[line];

/*
printf("L310   %d   %f  %f %f %f %f \n",  line,  amono, gaw,  gax,  gay,  gaz);
*/



       }

    /*
     * Check for delay loop wrap around.
     */

       if ((++del1c) >= spread_time) { del1c = 0; del1p =  &(delay1[0][0]); }
```

```
/*********  Store A Signal For Debugging  **********/
/*  Disrupts Cache??
debug_wave[debug_index++]= amono;
if (debug_index == DEBUG_SAMPLES) debug_index = 0;
*/






/************* Sample Playback *************/
if (sample_status != CONTROL_OFF) {
     for (voice=0; voice<MAX_VOICES; voice++) {
       if (voice_state[voice] == 1) {
        sample_pointer = voice_sample_pointer[voice];
        sf_num = voice_sf[voice];
        vol = voice_volume[voice];
        gaw += *(sample_pointer);
        gax += *(sample_pointer+1);
        gay += *(sample_pointer+2);
        gaz += *(sample_pointer+3);
        voice_sample_pointer[voice] = sample_pointer+4;
        if (--voice_sample_counter[voice] == 0) {
            voice_state[voice] = -1;
            sample_key_voice[voice_key[voice]] = -1;
            voice_stack_pointer--;
            voice_stack[voice_stack_pointer] = voice;
        }
       }
     }
}

#ifdef FULLCODE
if (audition_state == 3) {
    gaw += *aud_frame_base;
    gax += *(aud_frame_base+1);
    gay += *(aud_frame_base+2);
    gaz += *(aud_frame_base+3);
}
#endif

/**********  Rotate  ******************/
     if (rotate_mode != CONTROL_OFF) {
        gax = m1*gax + m2*gay + m3*gaz;
        gay = m4*gax + m5*gay + m6*gaz;
        gaz = m7*gax + m8*gay + m9*gaz;
     }

/**********  Dominance  ******************/
     if (dominance_status != CONTROL_OFF) {
        gaw = d1*gaw + d2*gax + d3*gay;
        gax = d4*gaw + d5*gax + d6*gay;
        gay = d7*gaw + d8*gax + d9*gay;
     }



/**********  3D delay *****************/
     if ( delay_status != CONTROL_OFF ) {
    fbw =  gaw;
    fbx =  gax;
    fby =  gay;
    fbz =  gaz;

    if ( delay_direct_level > 0) {
        gaw *= delay_direct_level;
        gax *= delay_direct_level;
        gay *= delay_direct_level;
        gaz *= delay_direct_level;
    }
    else gaw=gax=gay=gaz=0;


    if ( delay_tap1 > 0) {
    flp = delay_line[ delay_count[1] ];
```

```
        gaw += *flp * delay_tap_level1;
        gax += *(1+flp) * delay_tap_level1;
        gay += *(2+flp) * delay_tap_level1;
        gaz += *(3+flp) * delay_tap_level1;

        fbw += *flp     * delay_tap_feedback1;
        fbx += *(1+flp) * delay_tap_feedback1;
        fby += *(2+flp) * delay_tap_feedback1;
        fbz += *(3+flp) * delay_tap_feedback1;
    }


    if ( delay_tap2 > 0 ) {
    flp = delay_line[ delay_count[2] ];

        tax = *(1+flp) * delay_tap2_cos + *(2+flp) * delay_tap2_sin;
        tay = *(1+flp) * -delay_tap2_sin + *(2+flp) * delay_tap2_cos;
/*
printf("L430 %f %f \n", delay_tap2_cos, delay_tap2_sin);
*/
        gaw += *flp * delay_tap_level2;
        gax += tax * delay_tap_level2;
        gay += tay * delay_tap_level2;
        gaz += *(3+flp) * delay_tap_level2;

        fbw += *flp     * delay_tap_feedback2;
        fbx += tax * delay_tap_feedback2;
        fby += tay * delay_tap_feedback2;
        fbz += *(3+flp) * delay_tap_feedback2;
    }


    if ( delay_tap3 > 0) {
    flp = delay_line[ delay_count[3] ];

        tax = *(1+flp) * delay_tap3_cos + *(2+flp) * delay_tap3_sin;
        tay = *(1+flp) * -delay_tap3_sin + *(2+flp) * delay_tap3_cos;

        gaw += *flp * delay_tap_level3;
        gax += tax * delay_tap_level3;
        gay += tay * delay_tap_level3;
        gaz += *(3+flp) * delay_tap_level3;

        fbw += *flp     * delay_tap_feedback3;
        fbx += tax * delay_tap_feedback3;
        fby += tay * delay_tap_feedback3;
        fbz += *(3+flp) * delay_tap_feedback3;
    }

    flp = delay_line[delay_count[0]];
        *flp = fbw;
        *(1+flp) = fbx;
        *(2+flp) = fby;
        *(3+flp) = fbz;


    if (++delay_count[0] == DELAY_FRAMES) delay_count[0] = 0;
    if (++delay_count[1] == DELAY_FRAMES) delay_count[1] = 0;
    if (++delay_count[2] == DELAY_FRAMES) delay_count[2] = 0;
    if (++delay_count[3] == DELAY_FRAMES) delay_count[3] = 0;

     }


/****************  Permutation, Decoding, Limiting and Block storage  *******/
#include "lt.decode.c"



/*******************  The Control Manager  ***********************/

/***  TODO : may like to create a more ordered and predictable scheduling of control routines!!  ***/

  if (control_count-- == 0) {
     control_count = CONTROL_PERIOD;
/*
```

```
printf("CONTROL\n");
*/

/*
    if ( ALgetfilled(audio_out) > 500 ) {
*/

    if ((form_count--) == 0) {
    if (form_status) fl_check_forms();
    form_count = FORM_PERIOD;
/*
printf("xform\n");
*/
    }

    else if ((filter_control_count--) == 0) {
        kfilter();
        filter_control_count = FILTER_CONTROL_PERIOD;
/*
printf("filter\n");
*/
     }

    else if ((locate_control_count--) == 0) {
        locate();
        locate_control_count = LOCATE_CONTROL_PERIOD;
/*
printf("locate\n");
*/
     }

    else if (rotate_mode !=0 ) {
        if ((rotate_control_count--) == 0) {
        rotate();
        rotate_control_count = ROTATE_CONTROL_PERIOD;
/*
printf("rotate\n");
*/
        }
    }

    else if (dominance_status !=0 ) {
        if ((dominance_control_count--) == 0) {
        dominate();
        dominance_control_count = DOMINANCE_CONTROL_PERIOD;
/*
printf("dominance\n");
*/
        }
    }

    else if ((clip_count--) == 0) {
        clip_count = CLIP_PERIOD;
        if (plus_clips >0 && old_plus_clips ==0) fl_show_object(lamb_form->plus_clip);
        else if (plus_clips ==0 && old_plus_clips >0) fl_hide_object(lamb_form->plus_clip);

        if (minus_clips >0 && old_minus_clips ==0) fl_show_object(lamb_form->minus_clip);
        else if (minus_clips ==0 && old_minus_clips >0) fl_hide_object(lamb_form->minus_clip);

        old_plus_clips = plus_clips;
        old_minus_clips = minus_clips;
        plus_clips = minus_clips = 0;
    }


  }
/********** End of Control Rate Stuff  *****************/



    } /* End of in_audio_block loop */


    if (master_mute == 0) ALwritesamps(audio_out, out_audio_block, AUDIO_BLOCK_SAMPLES);

    if (record_state == 1)
```

```
        if (AFwriteframes(record_fh, AF_DEFAULT_TRACK, record_out_block, AUDIO_BLOCK_FRAMES) ==0) {
            fl_set_button(lamb_form->record, 0);
            record_cb(lamb_form->record, 0);
            fl_show_alert("Recording terminated early.", "Disk space may have been used up.", "", 1);
        };

/*
printf("L370 %d\n", ALgetfilled(audio_out));
if (ALgetfilled(audio_out) == 0) printf("L367!!\n");
*/

   }
}
/************************  End of Main Loop  **************************/
/**********************************************************************/



/**********************************************************************/
void setaudio(void)
{
    ALconfig config_in, config_out;

    long buf[] = {
                AL_INPUT_SOURCE, 0,
                AL_INPUT_RATE, 0,
        AL_CHANNEL_MODE, CHANNEL_MODE,
                AL_OUTPUT_RATE, AL_RATE_INPUTRATE,
                AL_MIC_MODE, AL_STEREO,
                AL_SPEAKER_MUTE_CTL, AL_SPEAKER_MUTE_ON,
                AL_LEFT_INPUT_ATTEN, 255,
                AL_RIGHT_INPUT_ATTEN, 255,
                AL_LEFT2_INPUT_ATTEN, 255,
                AL_RIGHT2_INPUT_ATTEN, 255,
        AL_LEFT_SPEAKER_GAIN, 255,
        AL_RIGHT_SPEAKER_GAIN, 255 };

    ALcloseport(audio_in);
    ALcloseport(audio_out);

    if (slow_control[INPUT_SOURCE] == 1) buf[1] = AL_INPUT_LINE;
    if (slow_control[INPUT_SOURCE] == 2) buf[1] = AL_INPUT_MIC;
    if (slow_control[INPUT_SOURCE] == 3) buf[1] = AL_INPUT_DIGITAL;
    buf[3] = audio_rate_table[(int)slow_control[AUDIO_RATE]];
    buf[13] = 255-slow_control[INPUT_LEVEL_BASE]*2;
    buf[15] = 255-slow_control[INPUT_LEVEL_BASE+1]*2;
    buf[17] = 255-slow_control[INPUT_LEVEL_BASE+2]*2;
    buf[19] = 255-slow_control[INPUT_LEVEL_BASE+3]*2;
    ALsetparams(AL_DEFAULT_DEVICE, buf, 24);  /* 24 = size of buf */

    config_in=ALnewconfig();
    config_out=ALnewconfig();
    ALsetwidth(config_in, AL_SAMPLE_16);
    ALsetwidth(config_out, AL_SAMPLE_16);
    ALsetchannels(config_in, AUDIO_LINES);
    ALsetchannels(config_out, AUDIO_LINES);
    ALsetqueuesize(config_in, slow_control[BUFFER_OUT_SIZE]);
    ALsetqueuesize(config_out, slow_control[BUFFER_OUT_SIZE]);
    audio_in = ALopenport("input","r",config_in);
    audio_out = ALopenport("output","w",config_out);

    ALfreeconfig(config_in);
    ALfreeconfig(config_out);

    while(ALgetfilled(audio_in) < slow_control[BUFFER_OUT_SIZE]);
    /* ( Top up input buffer so that output buffer can fill quickly,
     *   rather than wait for blocks etc. )
     */
}



/*********************************************************************/
void loadsf()
{
   short i;
   long total_size=0;
   void *m_result;
   AFfilehandle fh;
   long *sampwidth, *sampfmt, frames;
   char sfname[50];

    for(i=0; i<20; i++)
    if (sf[i] != NULL) {
        free(sf[i]);
        sf[i] = NULL;
    }

    i = 0;
    while(i<20) {
      sprintf(sfname,"%s%d.aiff", filepath, i+1);  /* standardised filenames. */
          fh = AFopenfile( sfname, "r", 0);
      if (fh==AF_NULL_FILEHANDLE) break;

      if (AFgetchannels(fh, AF_DEFAULT_TRACK) != 4) {
        fl_show_alert("Sample file must be a 4-track B-format file.", "", "", 1);
        fl_check_forms(); }

      else {
        if (AFgetrate(fh, AF_DEFAULT_TRACK) != audio_rate_table[(int)slow_control[AUDIO_RATE]]) {
        fl_show_alert("Warning: audio rate of play file different to current setting.", "", "",1);
        fl_check_forms(); }

        frames  = AFgetframecnt(fh, AF_DEFAULT_TRACK);
        total_size += frames;
        m_result = malloc(sizeof(short)*4*frames);
        if (m_result != NULL) {
                sf_size[i] = frames;
            sf[i] = (short*)m_result;
                printf("Loading .. %s\t%d  \n",sfname,sf_size[i]);
                AFreadframes(fh, AF_DEFAULT_TRACK, sf[i], frames);
        }
        else { printf("%s too big.\n",sfname[i]);
            fl_show_alert("Sample file too big.", "", "", 1);
            sf_size[i] = 0; }
      }
      i++;
      AFclosefile(fh);

    }
    printf("\nSample loading finished.\n\n");
}


/***********************************************************************/
void checkmidi()
{
   short i,param,mod,line,key,vel;

   /************ Controller processing **********/
    if (running_status ==0xb0 | running_status ==0xe0  ) {
            i = data[0];
        param = data[1];
        if (running_status ==0xe0) i=3; /* remap pitchbend */
/*
printf("L498  %d %d \n",  i, param);
*/
    if (joystick_line != -1) {
        switch((short)slow_control[MIDI_KEYBOARD_TYPE]) {
        case 1 :  /* Standard MIDI keyboard. */
            switch (i) {
            case 1 : i = joystick_base(joystick_line)+1;      /* Remap Mod to Y */
                 break;

            case 3 : i = joystick_base(joystick_line);  /* Remap Pitch B to X */
                 break;
            }

        case 2 : /* Korg M/T keyboard */
            switch (i) {
            case 1 :    i=joystick_base(joystick_line)+1;   /* Y-joystick is split between */
                    param=64+param/2;           /* two controllers. */
                    break;

            case 2 :    i=joystick_base(joystick_line)+1;
                    param=64-param/2;
                    break;

            case 3 :    i=joystick_base(joystick_line);
            }

        case 3 : /* Yamaha SY22 keyboard */
            switch (i) {
            case 1 :    i=joystick_base(joystick_line)+2;  /* Remap Mod to Z */
                    break;

            case 16 :   i = joystick_base(joystick_line);  /* Remap joystick X to X */
                    break;

            case 17 :   i=joystick_base(joystick_line)+1;  /* Remap joystick Y to Y */
            }
        }
    }


    control_process(i,param); /* Captures direct low level writes. */
    }



   /************ Note processing************/

         else if (running_status == 0x90 | running_status == 0x80) {
             key = data[0];
             vel = data[1];
/*
printf("L549  %d  %d\n", key, vel);
*/
             if (running_status == 0x80 ) vel = 0;  /* Note off */
             if (key >= slow_control[SAMPLE_KB]) { /* Accept rest of keyboard. */
                sample_playback(key,vel);
                return; }

             else if (key >= ROTATE_KEY_BASE) {
                rotate_man(key,vel);
                return; }

             else if (key >= SPACE_KEY_BASE) {
                space_man(key,vel);
                return; }
         }



}


short   joystick_base(short l)
{
    return( (joystick_line==ROTATE_LINE) ?
        ROTATE_BASE : c_base[l][locate_mode[l]]);
}



/**************************************************************************/
void control_process(short i, short param)
{
short line, mod;
            control[i] = param;
            status[i] = CONTROL_ON;
/*
printf("L633 %d %d\n", i, param);
*/
    /* Slow control bus. */
        if (i==SLOW_DATA) slow_control_process(control[SLOW_ADDRESS],param);

    /*  Fast control defaults. */
```

```
        target[i] = param;      /* Target allows the filter to chase a function of the control input */
        kfilter_jump[i] = FOLLOW_JUMP;
        kfilter_mode[i] = 0;        /* Use linear filtering. */

        /* Locate specific.  */
            if (i>=LOCATE_BASE && i<= LOCATE_BASE+39) {
               line = (i-LOCATE_BASE)/10;  /* Audio line */
               mod = (i-LOCATE_BASE)%10;
           kfilter_jump[i] = follow_jump[line]; /* As controlled by the JUMPINESS slow control */
           jump_mode[i] = slow_control[JUMP_MODE_BASE+line];
/*
printf("L600 %d %f \n",i, kfilter_jump[i]);
*/
            /* Change locate mode if a new mode of controller received.*/
            if (mod<9) locate_mode[line] = (mod / 3)+1;
        }

    /* Object bumpiness  .. saving a few divides in ’locate’.. */
/*
        else if (i>=BUMPINESS_BASE && i<=BUMPINESS_BASE+3) {
        target[i] = param / 64.0f;
        kfilter_jump[i] = 0.01;
        }
*/

    /* Spread time */
        else if (i>=SPREAD_TIME_BASE && i<=SPREAD_TIME_BASE+3) {
        spread_time = param * 10;               /* Realtime crtical variable. */
        control[SPREAD_TIME_BASE] =             /* For display purposes. */
        control[SPREAD_TIME_BASE+1] =
        control[SPREAD_TIME_BASE+2] =
        control[SPREAD_TIME_BASE+3] = param;
        }

    /* Object volume  */
        else if (i>=VOLUME_BASE && i<=VOLUME_BASE+3) {
        line = i-VOLUME_BASE;
        kfilter_mode[i] = 1;  /* Use fractional filtering. */
        kfilter_jump[i] = VOLUME_JUMP;  /* Fractional change */
        target[i] = volume_scale[param]*16;   /* Convert to an exponential scale */
        if (filter[i]< 0.0001) filter[i] = 0.0001;  /* Prevent zero-sticking. */
        }

        /*  Rotate   */
            else if (i>=ROTATE_BASE && i<=ROTATE_BASE+2) rotate_mode=1;

        /*  Dominance   */
            else if (i>=DOMINANCE_BASE && i<=DOMINANCE_BASE+2) dominance_status=1;

    /*  Decode controls */
        else if (i==W_MASTER) w_master= param/64.0f;

        else if (i==Z_MASTER) z_master= param/64.0f;


}

/***************************************************************************/
void slow_control_process(int adr, ltfloat data)
{
short tap, master, n;
ltfloat a, b;

    slow_control[adr] = data;
/*
printf("L693   %d  %f\n",  adr,  data);
*/
    switch(adr) {
    case DELAY_DIRECT_LEVEL : delay_direct_level = data/128.0f; break;

    case DELAY_TAP_LEVEL_BASE :
        delay_tap_level1 = data/128.0f;
        delay_tap1 = delay_tap_level1+delay_tap_feedback1;      /* Update switch for processing tap1 */
        break;
    case DELAY_TAP_LEVEL_BASE+1 :
        delay_tap_level2 = data/128.0f;
        delay_tap2 = delay_tap_level2+delay_tap_feedback2;
        break;
    case DELAY_TAP_LEVEL_BASE+2 :
        delay_tap_level3 = data/128.0f;
        delay_tap3 = delay_tap_level3+delay_tap_feedback3;
        break;

    case DELAY_TAP_FEEDBACK_BASE :
        delay_tap_feedback1 = data/128.0f;
        delay_tap1 = delay_tap_level1+delay_tap_feedback1;
        break;
    case DELAY_TAP_FEEDBACK_BASE+1 :
        delay_tap_feedback2 = data/128.0f;
        delay_tap2 = delay_tap_level2+delay_tap_feedback2;
        break;
    case DELAY_TAP_FEEDBACK_BASE+2 :
        delay_tap_feedback3 = data/128.0f;
        delay_tap3 = delay_tap_level3+delay_tap_feedback3;
        break;

    case DELAY_TAP_TIME_BASE :
    case DELAY_TAP_TIME_BASE+1 :
    case DELAY_TAP_TIME_BASE+2 :

        tap = adr - DELAY_TAP_TIME_BASE +1;
        delay_count[tap] = delay_count[0]-
            delay_unit*(slow_control[MASTER_TIME]+1) * data -1;
        if (delay_count[tap] < 0) delay_count[tap]+=DELAY_FRAMES;
        break;

    case DELAY_TAP_ROTATE_BASE+1 :
        delay_tap2_cos = -Cos[64*(short)data];
        delay_tap2_sin = Sin[64*(short)data];
        break;

    case DELAY_TAP_ROTATE_BASE+2 :
        delay_tap3_cos = -Cos[64*(short)data];
        delay_tap3_sin = Sin[64*(short)data];
        break;


    case MASTER_TIME :
        master = delay_unit * (slow_control[MASTER_TIME]+1);
        for(tap=1; tap<=3; tap++)
        if ((delay_count[tap] = delay_count[0]-
            master * slow_control[DELAY_TAP_TIME_BASE+tap-1] -1) < 0)
             delay_count[tap]+=DELAY_FRAMES;
        break;


    case JUMPINESS_BASE :
    case JUMPINESS_BASE+1 :
    case JUMPINESS_BASE+2 :
    case JUMPINESS_BASE+3 :
        follow_jump[adr - JUMPINESS_BASE] = FOLLOW_JUMP + volume_scale[(short)data]*150;
        break;


    case PERMUTATION :
        switch( (int)data ) {
        case 1: pa1 = &gax;
            pa2 = &gay;
            pa3 = &gaz;
            break;

        case 2: pa1 = &gay;
            pa2 = &gax;
            pa3 = &gaz;
            break;

        case 3: pa1 = &gax;
            pa2 = &gaz;
            pa3 = &gay;
            break;

        case 4: pa1 = &gay;
```

```
        pa2 = &gaz;
        pa3 = &gax;
        break;

    case 5: pa1 = &gaz;
        pa2 = &gax;
        pa3 = &gay;
        break;

    case 6: pa1 = &gaz;
        pa2 = &gay;
        pa3 = &gax;
    } break;

case X_POLARITY : x_polarity = (data==1)?1:-1; break;
case Y_POLARITY : y_polarity = (data==1)?1:-1; break;
case Z_POLARITY : z_polarity = (data==1)?1:-1; break;
case OUTPUT_FORMAT : output_format = data;  break;
case RECORD_FORMAT :
    if (data == 1) record_out_block = &(out_audio_block[0][0]);
    else record_out_block = &(bf_audio_block[0][0]);
    record_format = data;
    break;

case QUAD_ASPECT :
    a = data/64.0f;
    b = sqrt(2/(1+a*a));
    squarex = a*b;
    squarey = b;
    break;

case INPUT_SOURCE :
case AUDIO_RATE :
case BUFFER_IN_SIZE :
case BUFFER_OUT_SIZE :
case INPUT_LEVEL_BASE :
case INPUT_LEVEL_BASE+1 :
case INPUT_LEVEL_BASE+2 :
case INPUT_LEVEL_BASE+3 :

    setaudio(); break;

case MIDI_SOURCE : setmidi(); break;

case PANIC :
    if (data==1) {          /* Testing to 1 makes it harder for random signal error. */
    for(n=0; n<=3; n++) {
        control_process(VOLUME_BASE+n, 0);
        control_process(DELAY_DIRECT_LEVEL+n, 0);
        control_process(DELAY_TAP_FEEDBACK_BASE+n, 0);
    }
    delay_status = CONTROL_OFF;
    rotate_mode = CONTROL_OFF;
    sample_status = CONTROL_OFF;
    slow_control[adr] = 0;      /* Prevent fileload panicing. */
    }
    break;

case GUI_OFF :
    if (data==1) {
    fl_hide_form(lamb_form->LAmb);
    form_status=0;
    slow_control[adr] = 0;      /* Prevent fileload guioffing. */
    }
    break;

case RESET_DELAY :
    delay_status = data;
    break;

case RESET_ROTATE :
    if (data==1) {
    control_process(ROTATE_BASE, 64);
    control_process(ROTATE_BASE+1, 64);
    control_process(ROTATE_BASE+2, 64);
    slow_control[adr] = 0;
```

```
        rotate_mode = CONTROL_OFF;
        }
        break;

    case RESET_SAMPLE :
        if (data==1) {
        slow_control[adr] = 0;
        sample_status = CONTROL_OFF;
        }
        break;

    case RESET_LISTENER :
        if (data==1) {
        control_process(LISTENER_BASE, 64);
        control_process(LISTENER_BASE+1, 64);
        control_process(LISTENER_BASE+2, 64);
        slow_control[adr] = 0;
        listener_status = CONTROL_OFF;
        }
        break;

    case RESET_DOMINANCE :
        if (data==1) {
        control_process(DOMINANCE_BASE, 64);
        control_process(DOMINANCE_BASE+1, 64);
        control_process(DOMINANCE_BASE+2, 64);
        slow_control[adr] = 0;
        dominance_status = CONTROL_OFF;
        }
        break;

    }
}



/**************************************************************************/
void sample_playback(short key, short vel)
{
short voice, sf_num;

                 if (vel > 0) {                      /* Note ON */
                    sf_num = key-slow_control[SAMPLE_KB];
                    if (
/*  Previous version didn't allow multiple notes on the same key:
            sample_key_voice[key] == -1
                    &&
*/
            sf[sf_num] != NULL

            &&  voice_stack_pointer < MAX_VOICES

            &&  sf_size[sf_num] > 0)  {
                        voice = voice_stack[voice_stack_pointer];
                        voice_stack_pointer++;
                        voice_state[ voice ] = 1;
                        voice_key[ voice] = key;
                        voice_sf[ voice ] = sf_num;
                        voice_sample_pointer[voice] = sf[sf_num];
            voice_sample_counter[voice] = sf_size[sf_num];
                        sample_key_voice[key] = voice;
                        voice_volume[voice] = vel;
            sample_status = CONTROL_ON;
/*
printf("Note ON  %d\n",voice);
*/
                    }
                 }
                 else {                              /* Note OFF */
/* Free a voice */
/*
                   if ( (voice = sample_key_voice[key]) >= 0 ) {
                     voice_state[voice] = -1;
                     sample_key_voice[key] = -1;
                     voice_stack_pointer--;
                     voice_stack[voice_stack_pointer] = voice;
                   }
*/
                 }
}

/***************************************************************************/
void space_man(short key, short vel)
{
   short k = key-SPACE_KEY_BASE;
   short line = k /12;
   short cb = c_base[line][1];
   short fkey = k%12;
   short new_key_mode;
   short key_mode = key_locate_mode[line];
   ltfloat r,x,y,z,Cphi,Sphi,push;
   short phi;
   ltfloat *base;
   ltfloat temp;
   int key_feel;


/*****************  Mode Changes  *******************/
   if (vel > 0) {    /* Note ON */
     if (fkey <= 2) {
        new_key_mode = fkey+1;

        if (new_key_mode <= 2 && key_mode == 3) {
            /* convert cyl to carts */
                    phi    = MOD8192((short)(filter[cb+3]*64));
                    r      = filter[cb+4]/CYL_UNIT_RADIUS;
                    z      = (filter[cb+5]-64)/CYL_UNIT_HEIGHT;
                Sphi   = Sin[phi];
                    Cphi   = Cos[phi];
                    x = r * Cphi;
                    y = r * Sphi;
        target[cb] = filter[cb] =  64 + CARTS_UNIT_RADIUS*x;
        target[cb+1] = filter[cb+1] = 64 + CARTS_UNIT_RADIUS*y;
                target[cb+2] = filter[cb+2] = 64 + CARTS_UNIT_RADIUS*z;


                }

        if (new_key_mode == 3 && key_mode <= 2) {
            /* convert carts to cyl */

                    x = (filter[cb]-64)/CARTS_UNIT_RADIUS;
                    y = (filter[cb+1]-64)/CARTS_UNIT_RADIUS;
                    z = (filter[cb+2]-64)/CARTS_UNIT_RADIUS;
                    r = sqrt(x*x + y*y);
            phi = asin(y/r)/PI*64;
            if (x<0) phi = 64 - phi;  /*full circle asin*/
        target[cb+3] = filter[cb+3] = ((r==0)?0:phi);
        target[cb+4] = filter[cb+4] = r * CYL_UNIT_RADIUS;
        target[cb+5] = filter[cb+5] = z * CYL_UNIT_HEIGHT + 64;


            }

                key_locate_mode[line] = new_key_mode;
        locate_mode[line] = (new_key_mode==1)?
                    1 : new_key_mode-1;
     }




     if (fkey == 3) joystick_line = line; /* could be carts or cyls..*/


     switch (key_mode) {


     case 1 :
        if (fkey>=4) {
        /****** Fixed position keys. *******/
        locate_mode[line] = 1;
        base = &slow_control[POSITION_KEY_BASE+3*(fkey-4)];
            target[cb] = *base;     /*  x target  */
            target[cb+1] = *(base+1);   /*  y target  */
            target[cb+2] = *(base+2);   /*  z target  */

        /* Filter jumps must be different to achieve linear motion..*/

        key_feel = slow_control[KEY_FEEL+line];

/*
printf("L1138 %f %f %f \n", target[cb], target[cb+1], target[cb+2] );
*/

        if (vel == 127 || key_feel == 127) {
            kfilter_jump[cb]
            = kfilter_jump[cb+1]
            = kfilter_jump[cb+2]
            = 128;  /* ie causing a single jump to target. Good for special effects. */

        }
        else {
            temp = (vel_jump_map[vel]*key_feel/16.0f)
            / sqrt( (target[cb]-filter[cb])*(target[cb]-filter[cb])
                + (target[cb+1]-filter[cb+1])*(target[cb+1]-filter[cb+1])
                + (target[cb+2]-filter[cb+2])*(target[cb+2]-filter[cb+2]) );
                kfilter_jump[cb] = abs(target[cb]-filter[cb]) * temp;
                kfilter_jump[cb+1] = abs(target[cb+1]-filter[cb+1]) * temp;
                kfilter_jump[cb+2] = abs(target[cb+2]-filter[cb+2]) * temp;
        }

        kfilter_mode[cb]
        = kfilter_mode[cb+1]
        = kfilter_mode[cb+2]
        = 0;

        /* Snap mode is initially off and stays that way. */

        status[cb]=status[cb+1]=status[cb+2]=CONTROL_ON;

        if (joystick_line == line) joystick_line = -1;  /* Kill joystick jitter. */
        } break;



     case 2 :
    if (fkey>=5) {
        /****** Cartesian push keys *******/
        locate_mode[line] = 1;
        push = vel_push_map[vel];
        switch (fkey) {
            case 7 : status[cb]=CONTROL_ON; target[cb] -= push; kfilter_mode[cb] = 1; break;
            case 9 : status[cb]=CONTROL_ON; target[cb] += push; kfilter_mode[cb] = 1; break;
            case 5 : status[cb+1]=CONTROL_ON; target[cb+1] -= push; kfilter_mode[cb+1] = 1; break;
            case 6 : status[cb+1]=CONTROL_ON; target[cb+1] += push; kfilter_mode[cb+1] = 1; break;
            case 11 : status[cb+2]=CONTROL_ON; target[cb+2] -= push; kfilter_mode[cb+2] = 1; break;
            case 10 : status[cb+2]=CONTROL_ON; target[cb+2] += push; kfilter_mode[cb+2] = 1; break;
        }
        if (joystick_line == line) joystick_line = -1;  /* Kill joystick jitter. */
    }; break;

     case 3 :
    if (fkey>=5) {
        /******* Cylindrical push keys *******/
        locate_mode[line] = 2;
        push = vel_push_map[vel];
        switch (fkey) {
            case 9 : status[cb+3]=CONTROL_ON; target[cb+3] -= push; kfilter_mode[cb+3] = 1; break;
            case 7 : status[cb+3]=CONTROL_ON; target[cb+3] += push; kfilter_mode[cb+3] = 1; break;
            case 5 : status[cb+4]=CONTROL_ON; target[cb+4] -= push; kfilter_mode[cb+4] = 1; break;
            case 6 : status[cb+4]=CONTROL_ON; target[cb+4] += push; kfilter_mode[cb+4] = 1; break;
            case 11 : status[cb+5]=CONTROL_ON; target[cb+5] -= push; kfilter_mode[cb+5] = 1; break;
            case 10 : status[cb+5]=CONTROL_ON; target[cb+5] += push; kfilter_mode[cb+5] = 1; break;
        }
        if (joystick_line == line) joystick_line = -1;  /* Kill joystick jitter. */
    }; break;
```

```
      }

   }
}

/***************************************************************************/
void rotate_man(short key, short vel)
{
    ltfloat push;
    rotate_mode = 1;
    push = vel_push_map[vel];
    switch (key-ROTATE_KEY_BASE) {
        case 0 : joystick_line = ROTATE_LINE;
        case 1 : status[ROTATE_BASE]=CONTROL_ON;
                 target[ROTATE_BASE] -= push; kfilter_mode[ROTATE_BASE] = 1; break;
        case 2 : status[ROTATE_BASE]=CONTROL_ON;
                 target[ROTATE_BASE] += push; kfilter_mode[ROTATE_BASE] = 1; break;
        case 3 : status[ROTATE_BASE+1]=CONTROL_ON;
                 target[ROTATE_BASE+1] -= push; kfilter_mode[ROTATE_BASE+1] = 1; break;
        case 4 : status[ROTATE_BASE+1]=CONTROL_ON;
                 target[ROTATE_BASE+1] += push; kfilter_mode[ROTATE_BASE+1] = 1; break;
        case 5 : status[ROTATE_BASE+2]=CONTROL_ON;
                 target[ROTATE_BASE+2] -= push; kfilter_mode[ROTATE_BASE+2] = 1; break;
        case 6 : status[ROTATE_BASE+2]=CONTROL_ON;
                 target[ROTATE_BASE+2] += push; kfilter_mode[ROTATE_BASE+2] = 1; break;
    }
}

/***********************************************************************/
make_trig()
{
  int i;
  for(i=0; i<8192; i++) {
     Sin[i] = sin(i/4096.0f*PI);
     Cos[i] = cos(i/4096.0f*PI);
  }
}

/***********************************************************************/
void kfilter()
{
   int i;

   static ltfloat incr[128];
   static ltfloat time[128];
   ltfloat jump;
/*
printf("L1048 %f  %f  %f\n", target[30], target[31], target[32]);
printf("L1049 %f  %f  %f\n", filter[30], filter[31], filter[32]);
printf("L1050 %f  %f  %f\n", kfilter_mode[30], kfilter_jump[30], kfilter_mode[30]);
*/
   for(i=CONTROL_BASE; i<=CONTROL_TOP; i++) {
      if (status[i] == CONTROL_ON) {
            if (kfilter_mode[i] == 0) {

                /* Simpler follower filter: */
            jump = kfilter_jump[i];   /* Allows position key sensing. */
                if (filter[i]-target[i] > jump) filter[i] -= jump;
                else if (target[i]-filter[i] > jump) filter[i] += jump;
            else if (jump_mode[i]==0) filter[i] = target[i];
            /* mode=1 causes posn snapping for large jump values. */
        }

        else {
            /* Fractional filter */
            jump = filter[i] * kfilter_jump[i];
                if (filter[i]-target[i] > jump) filter[i] -= jump;
                else if (target[i]-filter[i] > jump) filter[i] += jump;
        }

      }

   }

}
```

```
/* More complicated linear path plotting */
/*
      if (last_target[i] != target[i])
         incr[i] = (target[i]-filter[i])/FILTER_DIVIDE;

      filter[i] += incr[i];
      if ( sgn(filter[i]-target[i]) == sgn(incr[i]) ) incr[i]=0;
*/

/* ‘port’ type filter
                jump = (target[i]-filter[i])*KFILTER_FACTOR;
        if (jump > FOLLOW_JUMP) jump = FOLLOW_JUMP;
        else if (jump < -FOLLOW_JUMP) jump = -FOLLOW_JUMP;
            filter[i] += jump;
        }

 */

/*********************************************************************/
void locate()
{
   static char line=0;
   ltfloat x,y,z,xu,yu,zu,su,d,id,d2,dd2,mult, sprd=0;
   ltfloat lx, ly, lz;  /* Listener position */
   ltfloat r,h,Sphi,Cphi,Stheta,Ctheta;
   int b,i,phi,theta;
   ltfloat volume, bumpiness, object_size, w_adjust, z_adjust, eq_depth, spread_scale, spread_max;

   if (++line == 4) line = 0;     /* This spreads locate-work around the clock. */

   volume = filter[VOLUME_BASE+line];  /* Enables overall gain aswell as attenuation. */
   bumpiness = filter[BUMPINESS_BASE+line]*0.005; /* Quicker than 1/64.0f */
   object_size = filter[OBJECT_SIZE_BASE+line]*0.03;
   w_adjust = filter[W_ADJUST_BASE+line]*0.015625;
   z_adjust = filter[Z_ADJUST_BASE+line]*0.015625;
   eq_depth = filter[EQ_DEPTH_BASE+line];
   spread_scale = filter[SPREAD_SCALE_BASE+line]*.002;
   spread_max = filter[SPREAD_MAX_BASE+line]*.01;

   lx = (filter[LISTENER_BASE]-64.011f)/CARTS_UNIT_RADIUS;   /* Ouch - integer divides */
   ly = (filter[LISTENER_BASE+1]-64.011f)/CARTS_UNIT_RADIUS;
   lz = (filter[LISTENER_BASE+2]-64.011f)/CARTS_UNIT_RADIUS;

   b = c_base[line][1];

     switch (locate_mode[line]) {
      case 1 : /*  Cartesians  */

    /* .011f prevents x,y,z being zero: */
            x = (filter[b]-64.011f)/CARTS_UNIT_RADIUS -lx;   /* Posn relative to listener. */
            y = (filter[b+1]-64.011f)/CARTS_UNIT_RADIUS -ly;
            z = (filter[b+2]-64.011f)/CARTS_UNIT_RADIUS -ly;
            d2 = x*x + y*y + z*z;
/*
printf("L1074 %f\n", x);
*/
         mult = volume /(bumpiness*d2 + 1);

             /* id = 1/d */

         id  = 1/ sqrt(d2 + object_size * object_size); /* mips2 hardware sqrt */
             xu = x * id;
             yu = y * id;
             zu = z * id;
         su = object_size * id;   /* su compensates for falling x,y,z. */

#ifdef FULLCODE
    /*
     * Distance eq / HF attenuation.
     */

         eqa[line] = expf(-d2*eq_depth*.002);  /* NB ’d’ is an abstract distance. */
            /* also try : 1/(1+d2*eq_depth); */
         eqb[line] = 1-eqa[line];
        sprd = spread[line] = spread_max  * mult / (1 + d2 * spread_scale );

#endif
             new_wf[line] =  w_adjust * W_FACTOR * ((1 + su)*mult -sprd);
             new_xf[line] =  xu * mult + sprd;
             new_yf[line] =  yu * mult - sprd;
             new_zf[line] =  z_adjust * zu * mult;
/*
printf("L1400 %f %f %f %f\n", new_wf[line], new_xf[line], new_yf[line], new_zf[line]);
*/
             break;

      case 2 : /* Cylindrical cordinates */
        /* Listener posn is awkward to implement for cyls */
            phi    = MOD8192((int)(filter[b+3]*64));
            r      = filter[b+4]/CYL_UNIT_RADIUS;
            z      = (filter[b+5]-64)/CYL_UNIT_HEIGHT;
            Sphi   = Sin[phi];   /*  Table has size 8192.  */
            Cphi   = Cos[phi];

            x = r * Cphi;
            y = r * Sphi;

            d2 = r*r + z*z;
        mult = volume/(bumpiness*d2 + 1);   /* Big bumpiness -> quicker fall off. */

#ifdef FULLCODE
    /*
     * Distance eq / HF attenuation.
     */
         eqa[line] = expf(-d2*eq_depth*.002);  /* NB 'd' is an abstract distance. */
            /* also try : 1/(1+d2*eq_depth); */
         eqb[line] = 1-eqa[line];

        d2 = d2 + object_size * object_size;
            d = sqrt(d2);
            xu = x / d;
            yu = y / d;
            zu = z / d;
        su = object_size /d;

        sprd = spread[line] = spread_max  * mult / (1 + d2 * spread_scale );
#endif

            new_wf[line] =  w_adjust * W_FACTOR * ((1 + su)*mult -sprd);
            new_xf[line] =  xu * mult + sprd;
            new_yf[line] =  yu * mult - sprd;
            new_zf[line] =  z_adjust * zu * mult;
            break;

      case 3 : /* Polar cordinates */
            phi    = MOD8192((int)(filter[b+6]*64));
            theta  = MOD8192((int)(filter[b+7]*64));
            d      = (filter[b+8])/POLAR_UNIT_RADIUS;

            Sphi   = Sin[phi];   /*  Table has size 8192.  */
            Cphi   = Cos[phi];

            Stheta = Sin[theta];
            Ctheta = Cos[theta];

            xu = Ctheta * Cphi;
            yu = Ctheta * Sphi;
            zu = Stheta;
        su = object_size /d;
        d2 = d * d;

         eqa[line] = exp(-d2*eq_depth*.002);  /* NB 'd' is an abstract distance. */
            /* also try : 1/(1+d2*eq_depth); */
         eqb[line] = 1-eqa[line];

        mult = volume/(bumpiness*d2 + 1);   /* Big bumpiness -> quicker fall off. */

            new_wf[line] =  w_adjust * (W_FACTOR + su) *mult;
            new_xf[line] =  xu *mult;
```

```
          new_yf[line] =  yu *mult;
          new_zf[line] =  z_adjust * mult;


     }
/*
printf("L1210  %d %d %f  %f  %f  %f\n", line, locate_mode[line], new_wf[line], new_xf[line],
                                    new_yf[line], new_zf[line]);
*/
}

/*********************************************************************/

void rotate()
{
   int phi, theta, eta;
   ltfloat Sphi, Cphi, Stheta, Ctheta, Seta, Ceta;

 /* Wrap around because of cursor keys... */
 /* It would be better to use target, and do the moding in rotate_man() */
   phi    = MOD8192((int)(filter[ROTATE_BASE]*64.0f));
   theta  = MOD8192((int)(filter[ROTATE_BASE+1]*64.0f));
   eta    = MOD8192((int)(filter[ROTATE_BASE+2]*64.0f));
/*
printf("L1171 %d  %d  %d\n",phi,theta,eta);
*/
   Sphi   = Sin[phi];
   Cphi   = Cos[phi];
   Stheta = Sin[theta];
   Ctheta = Cos[theta];
   Seta   = Sin[eta];
   Ceta   = Cos[eta];

 /* Calculate new matix components */
   m1     = Cphi*Ctheta;
   m2     = Sphi*Ceta-Cphi*Stheta*Seta;
   m3     = Sphi*Seta+Cphi*Stheta*Ceta;

   m4      = -Ctheta*Sphi;
   m5      = Cphi*Ceta+Sphi*Stheta*Seta;
   m6      = Cphi*Seta-Sphi*Stheta*Ceta;

   m7      = -Stheta;
   m8      = -Ctheta*Seta;
   m9      = Ctheta*Ceta;
}

/*********************************************************************/

void dominate()
{
    ltfloat x, y, c, s, d, a, b, l, m, cc, ss;

    y = 64.11 - filter[DOMINANCE_BASE];
    x = filter[DOMINANCE_BASE+1]-64.11;

    d = sqrt(x*x + y*y);
    c = x / d;
    s = y / d;

    l = d * d * 0.006 + 1;  /* "Lambda", the amount of dominance in the preferred direction. range 0.4*/
    a = l+ 1/l;
    b = l- 1/l;

    m = 1/ (2*l);  /* Balancing factor to keep an even volume level in the dominant direction. */

    d1 = a *m;      /* w,w term ..*/
    d2 = b*c*.707 *m;    /* w,x term */
    d3 = b*s*.707 *m;

    cc = c*c;
    ss = s*s;

    d4 = d2*2;
    d5 = (cc*a+ss)*m;
    d6 = (a-1)*s*m;
    d7 = d3*2;
    d8 = d6;
    d9 = (ss*a+cc)*m;
}

/**********************************************************************/

void copyright()
{
printf("\n\nLAmb Live Ambisonic MIDI System  "); printf(VERSION);
printf("\n(c) 1997 Dylan Menzies-Gow, University of York\nrdmg101@unix.york.ac.uk\n\n");
printf("This code may be reproduced for private and academic purposes only.\n\n");
printf("\nREADY\n\n");
}


/***********************************************************************************/
void set_tables()
{
    int n, m;
    ltfloat  a = 0.01;      /* volume_scale start */
    ltfloat  b = 1;         /* volume scale end */
    ltfloat  f = log(b/a)/127.0f;       /* Calculation factor */


    for(n=0; n<128; n++) {
       sample_key_voice[n] = -1;         /* Keys initially unpressed */
                                /* Positive number indicates voice no */

      vel_jump_map[n] = (0.00003f * n * n);
      vel_push_map[n] = (0.001f * n * n );

      volume_scale[n] = a * ( exp(f*n) -1 );  /* -1 takes the scale to zero. */
      jump_mode[n] = 0;     /* No Snap */
   }

   for(n=0; n<MAX_VOICES; n++)  {
      voice_state[n] = 0;
      voice_stack[n] = n;
   }


  for(n=0;n<DELAY_FRAMES;n++) {
    delay_line[n][0] = delay_line[n][1] = delay_line[n][2] = 0; }

    for(n=0;n<4;n++) follow_jump[n] = FOLLOW_JUMP;

}

/***********************************************************************************/
void set_controls()
{
   int n,m;

   for(n=0; n<128; n++) {
      control[n] = 64;
      filter[n] = 64;

      slow_control[n] = 0;  /* Reset all delays and input levels */
   }

    for(n=0;n<4;n++) {
        control[VOLUME_BASE+n] = filter[VOLUME_BASE+n] = 0;
        control[SPREAD_MAX_BASE+n] = filter[SPREAD_MAX_BASE+n] = 0;
        control_process(SPREAD_TIME_BASE+n, 64);
    }

    control_process(W_MASTER, 64); /* Side effects .. w_master,. */
    control_process(Z_MASTER, 64);


   for(n=0; n<128; n++)
      status[n] = CONTROL_OFF;          /* Turn off all filters initially. */

    recall_positions(1);
    slow_control_process(QUAD_ASPECT, 64);  /* All these calls initialise other variables */
    slow_control_process(PERMUTATION, 1);
    slow_control_process(X_POLARITY, 1);
    slow_control_process(Y_POLARITY, 1);
    slow_control_process(Z_POLARITY, 1);
    slow_control_process(DELAY_TAP_TIME_BASE, 0);
    slow_control_process(DELAY_TAP_TIME_BASE+1, 0);
    slow_control_process(DELAY_TAP_TIME_BASE+2, 0);
    slow_control_process(DELAY_TAP_ROTATE_BASE, 64);
    slow_control_process(DELAY_TAP_ROTATE_BASE+1, 64);
    slow_control_process(DELAY_TAP_ROTATE_BASE+2, 64);
    slow_control_process(OUTPUT_FORMAT, 1); /* 1 = B-format */
    slow_control_process(RECORD_FORMAT, 1); /* 1 = Copy output format */

    slow_control[DELAY_DIRECT_LEVEL] = 127;
    slow_control[INPUT_SOURCE] = 1;     /* Line input */
    slow_control[AUDIO_RATE] = 6;       /* 44100 KHz */
    slow_control[BUFFER_IN_SIZE] = IN_BUFF;
    slow_control[BUFFER_OUT_SIZE] = OUT_BUFF;
    slow_control[INPUT_LEVEL_BASE] = 64;
    slow_control[INPUT_LEVEL_BASE+1] = 64;
    slow_control[INPUT_LEVEL_BASE+2] = 64;
    slow_control[INPUT_LEVEL_BASE+3] = 64;
    slow_control[MIDI_SOURCE] = 4;      /* User */
    slow_control[MIDI_KEYBOARD_TYPE] = 1;   /* Auto */
    slow_control[MIDI_CHANNEL] = 1;
    slow_control[KEY_FEEL] = 64;
    slow_control[KEY_FEEL+1] = 64;
    slow_control[KEY_FEEL+2] = 64;
    slow_control[KEY_FEEL+3] = 64;
    slow_control[SAMPLE_KB] = SAMPLE_KEY_BASE;

    strcpy(filepath, SF_PATH);
    strcpy(record_file, RECORD_FILE);
    strcpy(play_file, PLAY_FILE);
    strcpy(audition_file, AUDITION_FILE);

}

/******************************************************************************/
void initialize_lamb_form()
{
    Pixmap icon_pixmap, mask;
    unsigned int w,h;
    void* data;

    lamb_form = create_form_LAmb();
    fl_set_atclose(at_close, data);     /* Kill the thread on exit. */
    finish_form_design();

/*
    icon_pixmap = fl_create_from_bitmapdata(fl_root, icon_bitmap, 85, 67);
    icon_pixmap = fl_read_pixmapfile(fl_root, "lamb.xpm", &w, &h, &mask, 0, 0, 0);
*/

#ifdef ICON
    icon_pixmap = fl_create_from_pixmapdata(fl_root, icon_xpm, &w, &h, &mask, 0, 0, 0);
    fl_set_form_icon(lamb_form->LAmb, icon_pixmap, mask);
#endif
    set_values_in_form();

    fl_show_object(lamb_form->config_gp);
    fl_hide_object(lamb_form->main_gp);

    fl_hide_object(lamb_form->minus_clip);
    fl_hide_object(lamb_form->plus_clip);

    fl_show_form(lamb_form->LAmb,FL_PLACE_CENTER,FL_FULLBORDER,"LAmb");
    fl_check_forms();   /* Look neat! */
    fl_check_forms();

}

int at_close(FL_FORM* form, void* dummy) {
    stop_thread();
    return(FL_OK);
}


void finish_form_design()
{
FD_LAmb *f = lamb_form;

    fl_addto_choice(f->input_source,
       " Line(1,2,3,4) | Line(1,2) / Mic(3,4) | Digital(1,2,3,4) | User function");
    fl_addto_choice(f->output_format,
       " B-format: W,X,Y,Z | Quad: LF,RF,RB,LB | Hex: W,CR,LF,LB | Hex: W,CB,LF,RF");
    fl_addto_choice(f->record_format," Copy output format | Raw B-format ");
    fl_addto_choice(f->audio_rate,
       " 8 kHz | 11.025 kHz | 16 kHz | 22.05 kHz | 32 kHz | 44.1 kHz | 48 kHz");
    fl_addto_choice(f->permutation," X Y Z | Y X Z | X Z Y | Y Z X | Z X Y | Z Y X");
    fl_addto_choice(f->x_polarity," Normal | Inverted");
    fl_addto_choice(f->y_polarity," Normal | Inverted");
    fl_addto_choice(f->z_polarity," Normal | Inverted");
    fl_set_slider_bounds(f->w_master, 0, 127);
    fl_set_slider_step(f->w_master, 1);
    fl_set_slider_bounds(f->z_master, 0, 127);
    fl_set_slider_step(f->z_master, 1);

    fl_set_slider_bounds(f->input_level1, 0, 127);
    fl_set_slider_step(f->input_level1, 1);
    fl_set_slider_bounds(f->input_level2, 0, 127);
    fl_set_slider_step(f->input_level2, 1);
    fl_set_slider_bounds(f->input_level3, 0, 127);
    fl_set_slider_step(f->input_level3, 1);
    fl_set_slider_bounds(f->input_level4, 0, 127);
    fl_set_slider_step(f->input_level4, 1);

    fl_addto_choice(f->midi_source," md, default | md, name ‘lamb’ | Stdin | User ");
    fl_addto_choice(f->midi_keyboard_type," Standard | Korg M/T | Yamaha SY22 ");
    fl_set_input_color(f->midi_channel, 0, 0);

    fl_addto_choice(f->position_key," E | F | F# | G | G# | A | A# | B ");
    fl_set_menu(f->recall, " Quad positions | Cubic positions | Hex positions" );

    fl_set_slider_bounds(f->volume, 0, 127);
    fl_set_slider_step(f->volume, 1);
    fl_set_slider_bounds(f->object_size, 0, 127);
    fl_set_slider_step(f->object_size, 1);
    fl_set_slider_bounds(f->bumpiness, 0, 127);
    fl_set_slider_step(f->bumpiness, 1);
    fl_set_slider_bounds(f->eq_depth, 0, 127);
    fl_set_slider_step(f->eq_depth, 1);
    fl_set_slider_bounds(f->spread_scale, 0, 127);
    fl_set_slider_step(f->spread_scale, 1);
    fl_set_slider_bounds(f->spread_max, 0, 127);
    fl_set_slider_step(f->spread_max, 1);
    fl_set_slider_bounds(f->spread_time, 0, 127);
    fl_set_slider_step(f->spread_time, 1);
    fl_set_slider_bounds(f->w_adjust, 0, 127);
    fl_set_slider_step(f->w_adjust, 1);
    fl_set_slider_bounds(f->z_adjust, 0, 127);
    fl_set_slider_step(f->z_adjust, 1);
    fl_set_slider_bounds(f->jumpiness, 0, 127);
    fl_set_slider_step(f->jumpiness, 1);

    fl_set_positioner_xbounds(f->xy_locate, 0, 127);
    fl_set_positioner_xstep(f->xy_locate, 1);
    fl_set_positioner_ybounds(f->xy_locate, 0, 127);
    fl_set_positioner_xstep(f->xy_locate, 1);
    fl_set_slider_bounds(f->z_locate, 127, 0);
    fl_set_slider_step(f->z_locate, 1);

    fl_set_positioner_xbounds(f->ae_rotate, 0, 127);
    fl_set_positioner_xstep(f->ae_rotate, 1);
    fl_set_positioner_ybounds(f->ae_rotate, 0, 127);
    fl_set_positioner_xstep(f->ae_rotate, 1);
    fl_set_slider_bounds(f->t_rotate, 127, 0);
    fl_set_slider_step(f->t_rotate, 1);

    fl_set_positioner_xbounds(f->xy_listener, 0, 127);
    fl_set_positioner_xstep(f->xy_listener, 1);
    fl_set_positioner_ybounds(f->xy_listener, 0, 127);
    fl_set_positioner_xstep(f->xy_listener, 1);
    fl_set_slider_bounds(f->z_listener, 127, 0);
    fl_set_slider_step(f->z_listener, 1);

    fl_set_positioner_xbounds(f->xy_dominance, 0, 127);
    fl_set_positioner_xstep(f->xy_dominance, 1);
    fl_set_positioner_ybounds(f->xy_dominance, 0, 127);
    fl_set_positioner_xstep(f->xy_dominance, 1);
    fl_set_slider_bounds(f->z_dominance, 127, 0);
    fl_set_slider_step(f->z_dominance, 1);

    fl_set_slider_bounds(f->key_feel, 0, 127);
    fl_set_slider_step(f->key_feel, 1);

    fl_set_slider_bounds(f->direct_level, 0, 127);
    fl_set_slider_step(f->direct_level, 1);
    fl_set_slider_bounds(f->level1, 0, 127);
    fl_set_slider_step(f->level1, 1);
    fl_set_slider_bounds(f->level2, 0, 127);
    fl_set_slider_step(f->level2, 1);
    fl_set_slider_bounds(f->level3, 0, 127);
    fl_set_slider_step(f->level3, 1);
    fl_set_slider_bounds(f->feedback1, 0, 127);
    fl_set_slider_step(f->feedback1, 1);
    fl_set_slider_bounds(f->feedback2, 0, 127);
    fl_set_slider_step(f->feedback2, 1);
    fl_set_slider_bounds(f->feedback3, 0, 127);
    fl_set_slider_step(f->feedback3, 1);
    fl_set_slider_bounds(f->time1, 0, 127);
    fl_set_slider_step(f->time1, 1);
    fl_set_slider_bounds(f->time2, 0, 127);
    fl_set_slider_step(f->time2, 1);
    fl_set_slider_bounds(f->time3, 0, 127);
    fl_set_slider_step(f->time3, 1);
    fl_set_slider_bounds(f->master_time, 0, 127);
    fl_set_slider_step(f->master_time, 1);

}

void set_values_in_form()
{
FD_LAmb *f = lamb_form;
char str[10];

    fl_set_choice(f->input_source, slow_control[INPUT_SOURCE]);
    fl_set_choice(f->output_format, slow_control[OUTPUT_FORMAT]);
    fl_set_choice(f->record_format, slow_control[RECORD_FORMAT]);
    fl_set_choice(f->audio_rate, slow_control[AUDIO_RATE]);
    fl_set_choice(f->permutation, slow_control[PERMUTATION]);
    fl_set_choice(f->x_polarity, slow_control[X_POLARITY]);
    fl_set_choice(f->y_polarity, slow_control[Y_POLARITY]);
    fl_set_choice(f->z_polarity, slow_control[Z_POLARITY]);
    fl_set_input(f->quad_aspect,  ftos(slow_control[QUAD_ASPECT]/64.0f) );
    fl_set_input(f->buffer_out_size,  itos(slow_control[BUFFER_OUT_SIZE]) );
    fl_set_slider_value(f->w_master, control[W_MASTER]);
    fl_set_slider_value(f->z_master, control[Z_MASTER]);
    fl_set_slider_value(f->input_level1, slow_control[INPUT_LEVEL_BASE]);
    fl_set_slider_value(f->input_level2, slow_control[INPUT_LEVEL_BASE+1]);
    fl_set_slider_value(f->input_level3, slow_control[INPUT_LEVEL_BASE+2]);
    fl_set_slider_value(f->input_level4, slow_control[INPUT_LEVEL_BASE+3]);

    fl_set_choice(f->midi_source, slow_control[MIDI_SOURCE]);
    fl_set_choice(f->midi_keyboard_type, slow_control[MIDI_KEYBOARD_TYPE]);
    fl_set_input(f->midi_channel, itos((short)slow_control[MIDI_CHANNEL]));
    fl_set_input(f->sample_base, itos((short)slow_control[SAMPLE_KB]));

    set_channel_stuff(form_channel);

    fl_set_choice(f->position_key, form_position_key);
    write_fixed_positions_to_form();

    fl_set_slider_value(f->direct_level, slow_control[DELAY_DIRECT_LEVEL]);
    fl_set_slider_value(f->level1, slow_control[DELAY_TAP_LEVEL_BASE]);
    fl_set_slider_value(f->time1, slow_control[DELAY_TAP_TIME_BASE]);
    fl_set_slider_value(f->feedback1, slow_control[DELAY_TAP_FEEDBACK_BASE]);
    fl_set_slider_value(f->level2, slow_control[DELAY_TAP_LEVEL_BASE+1]);
    fl_set_slider_value(f->time2, slow_control[DELAY_TAP_TIME_BASE+1]);
    fl_set_slider_value(f->feedback2, slow_control[DELAY_TAP_FEEDBACK_BASE+1]);
    fl_set_slider_value(f->level3, slow_control[DELAY_TAP_LEVEL_BASE+2]);
    fl_set_slider_value(f->time3, slow_control[DELAY_TAP_TIME_BASE+2]);
    fl_set_slider_value(f->feedback3, slow_control[DELAY_TAP_FEEDBACK_BASE+2]);
    fl_set_slider_value(f->master_time, slow_control[MASTER_TIME]);
    fl_set_button(f->delay_reset,  slow_control[RESET_DELAY]);
    fl_set_slider_value(f->rotate2, slow_control[DELAY_TAP_ROTATE_BASE+1]);
    fl_set_slider_value(f->rotate3, slow_control[DELAY_TAP_ROTATE_BASE+2]);


    fl_set_positioner_xvalue(f->ae_rotate, control[ROTATE_BASE]);
    fl_set_positioner_yvalue(f->ae_rotate, control[ROTATE_BASE+1]);
    fl_set_slider_value(f->t_rotate, control[ROTATE_BASE+2]);

    fl_set_positioner_xvalue(f->xy_listener, control[LISTENER_BASE]);
    fl_set_positioner_yvalue(f->xy_listener, control[LISTENER_BASE+1]);
    fl_set_slider_value(f->z_listener, control[LISTENER_BASE+2]);

    fl_set_positioner_xvalue(f->xy_dominance, control[DOMINANCE_BASE]);
    fl_set_positioner_yvalue(f->xy_dominance, control[DOMINANCE_BASE+1]);
    fl_set_slider_value(f->z_dominance, control[DOMINANCE_BASE+2]);

}

void set_channel_stuff(int chan)
{
FD_LAmb *f = lamb_form;

    fl_set_button(f->channel1, (chan == 0));
    fl_set_button(f->channel2, (chan == 1));
    fl_set_button(f->channel3, (chan == 2));
    fl_set_button(f->channel4, (chan == 3));

    fl_set_slider_value(f->volume,control[VOLUME_BASE+chan]);
    fl_set_slider_value(f->object_size,control[OBJECT_SIZE_BASE +chan]);
    fl_set_slider_value(f->bumpiness,control[BUMPINESS_BASE +chan]);
    fl_set_slider_value(f->eq_depth,control[EQ_DEPTH_BASE +chan]);
    fl_set_slider_value(f->spread_scale,control[SPREAD_SCALE_BASE+chan]);
    fl_set_slider_value(f->spread_max,control[SPREAD_MAX_BASE+chan]);
    fl_set_slider_value(f->spread_time,control[SPREAD_TIME_BASE+chan]);
    fl_set_slider_value(f->jumpiness,slow_control[JUMPINESS_BASE+chan]);
    fl_set_button(f->jump_mode, slow_control[JUMP_MODE_BASE+chan]);
    fl_set_slider_value(f->key_feel, slow_control[KEY_FEEL+chan]);

    fl_set_slider_value(f->w_adjust,control[W_ADJUST_BASE +chan]);
    fl_set_slider_value(f->z_adjust,control[Z_ADJUST_BASE +chan]);

    fl_set_positioner_xvalue(f->xy_locate, control[ c_base[chan][1] ]);
    fl_set_positioner_yvalue(f->xy_locate, control[ c_base[chan][1] +1 ]);
    fl_set_slider_value(f->z_locate, control[ c_base[chan][1] +2]);

}


void write_fixed_positions_to_form()
{
FD_LAmb *f = lamb_form;
int n;

    n = POSITION_KEY_BASE + 3*(form_position_key -1);
    fl_set_input(f->x_position, ftos(slow_control[n]/64-1) );
    fl_set_input(f->y_position, ftos(slow_control[++n]/64-1) );
    fl_set_input(f->z_position, ftos(slow_control[++n]/64-1) );
}

char *itos(long i)
{
    static char s[20];
    sprintf(s,"%d",i);
    return(s);
}

char *ftos(ltfloat f)
```

```
{
    static char s[20];
    sprintf(s,"%2.2f",f);
    return(s);
}

void recall_positions(short select) {
    ltfloat* p;
    short n, m;

    switch(select) {
    case 1 : p = quad_fixed_positions; break;
    case 2 : p = cubic_fixed_positions; break;
    case 3 : p = hex_fixed_positions; break;
    }

    for(n=0;n<8;n++)
    for(m=0;m<3;m++)
            slow_control_process(POSITION_KEY_BASE+3*n+m, (p[3*n+m]+1)*64.0f);

}

/*********************************************************************/
void load_setup(const char* filename) {
    FILE* handle;
    char version[10];
    short n;

    if ( (handle = fopen(filename, "r")) == NULL) {
    fprintf(stderr, "Failed to read %s\n", filename);
    return;
    }

    fgets(version, 10, handle);
    version[strlen(version)-1] = NULL;
    if (strcmp(version,  VERSION) != 0)
    fl_show_alert("Old setup version:", version, "", 1);

    fgets(filepath, 128, handle);
    /* Eliminate the /n character. */
    filepath[strlen(filepath)-1] = NULL;

    for(n=0; n<128; n++) {
    control_process(n,  fget_ltfloat(handle) );
    slow_control_process(n,  fget_ltfloat(handle) );
/*
printf("L1908   %d  %f\n", n, control[n]);
*/
    status[n] = 0;  /* No dynamics initially. */
    filter[n] = fget_ltfloat(handle);
    }

    form_channel = fget_short(handle);
    form_position_key = fget_short(handle);
    joystick_line = fget_short(handle);

    for(n=0; n<4; n++) {
    locate_mode[n] = fget_short(handle);
    key_locate_mode[n] = fget_short(handle);
    }
    rotate_mode = fget_short(handle);
    delay_status = fget_short(handle);
    sample_status = fget_short(handle);
    listener_status = fget_short(handle);
    dominance_status = fget_short(handle);

    /* for version 0.97 */

    fgets(record_file, 128, handle);
    /* Eliminate the /n character. */
    record_file[strlen(record_file)-1] = NULL;

    fgets(play_file, 128, handle);
    /* Eliminate the /n character. */
    play_file[strlen(play_file)-1] = NULL;
    fgets(audition_file, 128, handle);
    /* Eliminate the /n character. */
    audition_file[strlen(audition_file)-1] = NULL;

    fclose(handle);
    set_values_in_form();

}

void save_setup(const char* filename) {

    FILE* handle;
    short n;

    if ( (handle = fopen(filename, "w")) == NULL) {
    fprintf(stderr, "Failed to open file for writing.\n");
    return;
    }

    fputs(VERSION, handle);
    fputc('\n',  handle);

    fputs(filepath, handle);
    fputc('\n',  handle);

    for(n=0; n<128; n++) {
    fput_ltfloat(handle, control[n]);
    fput_ltfloat(handle, slow_control[n]);
    fput_ltfloat(handle, filter[n]);
    }


    fput_short(handle, form_channel);
    fput_short(handle, form_position_key);
    fput_short(handle, joystick_line);

    for(n=0; n<4; n++) {
    fput_short(handle, locate_mode[n]);
    fput_short(handle, key_locate_mode[n]);
    }

    fput_short(handle, rotate_mode);
    fput_short(handle, delay_status);
    fput_short(handle, sample_status);
    fput_short(handle, listener_status);
    fput_short(handle, dominance_status);

    fputs(record_file, handle);
    fputc('\n',  handle);
    fputs(play_file, handle);
    fputc('\n',  handle);
    fputs(audition_file, handle);
    fputc('\n',  handle);

    fclose(handle);
}


void fput_ltfloat(FILE* handle, ltfloat data) {
    char* p = (char*)&data;
    char n;

    for(n=1;n<=sizeof(ltfloat);n++)
    fputc(*(p++), handle);
}

ltfloat fget_ltfloat(FILE* handle) {
    ltfloat result;
    char* p = (char*)&result;
    char n;

    for(n=1;n<=sizeof(ltfloat);n++)
    *(p++) = fgetc(handle);

    return(result);
}
```

```
void fput_short(FILE* handle, short data) {
    char* p = (char*)&data;
    char n;

    for(n=1;n<=sizeof(short);n++)
    fputc(*(p++), handle);
}

short fget_short(FILE* handle) {
    short result;
    char* p = (char*)&result;
    char n;

    for(n=1;n<=sizeof(short);n++)
    *(p++) = fgetc(handle);

    return(result);
}



/*TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT*/
/* Read MIDI bytes and indicate to main process when whole message received */
/* Reason for this is because messages arrive has a quick succession of bytes.
 * and we don’t wish main to loose any.
 */


void input_thread(void* dummy) {
   short data_no = 0;
   short channel,  status;
   short byte;
   MDevent *mdbuf = calloc(1, sizeof(MDevent) );

   while(1) {

    switch((short)slow_control[MIDI_SOURCE]) {

        case 1 :
        case 2 :  /* md input */
                mdReceive(MidiPort,  mdbuf, 1);
                status = mdbuf->msg[0];
                channel = (status & 0x0f);
                running_status = status & 0xf0;
/* NB running status is a global var for checkmidi. */
                if ( status < 0xf0 && channel == slow_control[MIDI_CHANNEL]-1 ) {
                    data[0] = mdbuf->msg[1];
                    data[1] = mdbuf->msg[2];
                    checkmidi();
                }
                break;

        case 3 :  /* stdin */
                byte = getchar();
                if (byte >= 0xf0)  running_status = 0;

                else if ((byte>=0x80) && (byte<0xf0)){  /* Status byte */
                    running_status = byte & 0xf0;
                    channel = byte & 0x0f;
                    data_no=0;
                }


                else if (byte<0x80 && running_status != 0) {

                    /************* Channel commands ********************/
                    data[data_no]=byte;

                    if (++data_no == 2) {

                        /*********** Whole command received **********/
                        data_no = 0;
                        if (channel == slow_control[MIDI_CHANNEL]-1) checkmidi();
                    }
                }
                break;

        }
    }
}

/*********************************************************************/
void setmidi()
{
    short err;
    int option = (int)slow_control[MIDI_SOURCE];

    stop_thread();
    if (MidiPort != NULL) mdClosePort(MidiPort);
    if (slow_control[MIDI_SOURCE] <= 3) {  /* For thread using options only. */

        if (option == 3 ) { start_thread(); return; } /* stdin option. */

        printf("Valid md ports = %d\n",mdInit());

        if (option == 1 ) MidiPort = mdOpenInPort(NULL);
        if (option == 2 ) {
            int i;
            MidiPort = NULL;
            for(i=0; i<10; i++) {
                if (strcmp( (const char*)mdGetName(i),  "lamb") == 0) {
                MidiPort = mdOpenInPort("lamb");
                break;
                }
            }

        }

        if (MidiPort==NULL) {
            fl_show_alert("md error.", "Device name may not have been initialised with startmidi.", "", 1);
            fl_set_choice(lamb_form->midi_source, 4); /* user.. option */
            slow_control[MIDI_SOURCE] = 4;
        }

        else {
            mdSetStampMode(MidiPort, MD_NOSTAMP);
            start_thread();
        }
    }


}


/*****  Start / Stop thread for input *****/

void start_thread(void) {

    if ( ((thread_pid)=sproc(input_thread, PR_SALL)) == -1) {
       fprintf(stderr, "Unable to create input thread...aborting.\n");
       exit(-1);
    }
}

void stop_thread(void) {
   if (thread_pid >0 ) kill(thread_pid, SIGKILL);
}



/*****  Fix R4000 FPU exception ‘feature’ *****/
void flush_all_underflows_to_zero()
{
    /* Works on R4K and above */
    union fpc_csr f;
    f.fc_word = get_fpc_csr();
    f.fc_struct.flush = 1;
    set_fpc_csr(f.fc_word);
}
```

E.4 lt.decode.c

```
/*
 * lt.decode.c
 *
 * B-format decoding functions, separated from lt.c for convenience.
 *
 */


#ifdef FULLCODE
if (audition_state == 2) {
    gaw += *aud_frame_base;
    gax += *(aud_frame_base+1);
    gay += *(aud_frame_base+2);
    gaz += *(aud_frame_base+3);
}
#endif

*(bf_frame_base)   = gaw;
*(bf_frame_base+1) = gax;
*(bf_frame_base+2) = gay;
*(bf_frame_base+3) = gaz;


gaz *= z_master;
gaw *= w_master;

/* ga1 etc are addresses pointing to a permutation of gax, gay,gaz. */

ga1 = (*pa1)*x_polarity;
ga2 = (*pa2)*y_polarity;
ga3 = (*pa3)*z_polarity;

    switch( output_format ) {

    case 1: aout0 = gaw;
        aout1 = ga1;
            aout2 = ga2;
            aout3 = ga3;
        break;


    /* Square decoded. Output order is LF, RF, RB, LB. */
    /* The strength of W here is a topic for debate.. */
    case 2: aout0 = gaw+squarex*ga1+squarey*ga2;
            aout1 = gaw+squarex*ga1-squarey*ga2;
        aout2 = gaw-squarex*ga1-squarey*ga2;
        aout3 = gaw-squarex*ga1+squarey*ga2;
        break;

/*
The relative strength of W is a topic for debate,  and will depend on
the room, speaker spacing kind of sound being projected..

hex1 = sqrt(3/2);
hex2 = 1/sqrt(2);
hex3 = sqrt(2);
*/

    /* NB This is HEX1 decoded. Output order is 2W, CR, LF, LB
     * RF, RB and CL are calcuated by matrixing the amps so that:
     * RF = 2W - LB,  RB = 2W - LF,  CL = 2W - CR.
     */
    case 3:
        aout0 = 2*gaw;
        aout1 = gaw - hex3*ga2;
        aout2 = gaw + hex1*ga1 + hex2*ga2;
            aout3 = gaw - hex1*ga1 + hex2*ga2;
        break;

    /* NB This is HEX2 decoded. Output order is 2W, CB, LF, RF
     * RB, LB and CF are calcuated by matrixing the amps so that:
     * RB = 2W - LF,  LB = 2W - RF,  CF = 2W - CB.
     */
    case 4:
        aout0 = 2*gaw;
        aout1 = gaw - hex3*ga1;
            aout2 = gaw + hex2*ga1 + hex1*ga2;
            aout3 = gaw + hex2*ga1 - hex1*ga2;

        break;
    }


#ifdef FULLCODE
/***********   Add audition, already decoded  *****/
if (audition_state == 1) {
    aout0 += *aud_frame_base;
    aout1 += *(aud_frame_base+1);
    aout2 += *(aud_frame_base+2);
    aout3 += *(aud_frame_base+3);
}
#endif

/************  Overflow protection  *************/
    if (aout0 >32767.0f) { aout0 = 32767.0f; plus_clips++; }
    else if (aout0 <-32768.0f) { aout0 = -32768.0f; minus_clips++; }
    if (aout1 >32767.0f) { aout1 = 32767.0f; plus_clips++; }
    else if (aout1 <-32768.0f) { aout1 = -32768.0f; minus_clips++; }
    if (aout2 >32767.0f) { aout2 = 32767.0f; plus_clips++; }
    else if (aout2 <-32768.0f) { aout2 = -32768.0f; minus_clips++; }
    if (aout3 >32767.0f) { aout3 = 32767.0f; plus_clips++; }
    else if (aout3 <-32768.0f) { aout3 = -32768.0f; minus_clips++; }

/************  Frame Store  *********************/
    *(out_frame_base) = aout0;
    *(out_frame_base+1) = aout1;
    *(out_frame_base+2) = aout2;
    *(out_frame_base+3) = aout3;
```

E.5 f.h

```
/*
 * f.h
 *
 * Header file for form definition, generated with fdesign.
 *
 */


#ifndef FD_LAmb_h_
#define FD_LAmb_h_

/**** Callback routines ****/

extern void swap_page_cb(FL_OBJECT *, long);
extern void mute_cb(FL_OBJECT *, long);
extern void debug_cb(FL_OBJECT *, long);
extern void record_cb(FL_OBJECT *, long);
extern void play_cb(FL_OBJECT *, long);
extern void play_record_cb(FL_OBJECT *, long);
extern void audition_cb(FL_OBJECT *, long);
extern void output_format_cb(FL_OBJECT *, long);
extern void permutation_cb(FL_OBJECT *, long);
extern void x_polarity_cb(FL_OBJECT *, long);
extern void y_polarity_cb(FL_OBJECT *, long);
extern void z_polarity_cb(FL_OBJECT *, long);
extern void audio_rate_cb(FL_OBJECT *, long);
extern void midi_source_cb(FL_OBJECT *, long);
extern void midi_channel_cb(FL_OBJECT *, long);
extern void midi_keyboard_type_cb(FL_OBJECT *, long);
extern void load_setup_cb(FL_OBJECT *, long);
extern void save_setup_cb(FL_OBJECT *, long);
extern void load_sample_cb(FL_OBJECT *, long);
extern void input_source_cb(FL_OBJECT *, long);
extern void quad_aspect_cb(FL_OBJECT *, long);
extern void w_master_cb(FL_OBJECT *, long);
extern void z_master_cb(FL_OBJECT *, long);
extern void buffer_out_size_cb(FL_OBJECT *, long);
extern void input_level_cb(FL_OBJECT *, long);
extern void set_record_cb(FL_OBJECT *, long);
extern void set_play_cb(FL_OBJECT *, long);
extern void sample_base_cb(FL_OBJECT *, long);
extern void record_format_cb(FL_OBJECT *, long);
extern void set_audition_cb(FL_OBJECT *, long);
extern void direct_level_cb(FL_OBJECT *, long);
extern void level_cb(FL_OBJECT *, long);
extern void feedback_cb(FL_OBJECT *, long);
extern void time_cb(FL_OBJECT *, long);
extern void master_time_cb(FL_OBJECT *, long);
extern void volume_cb(FL_OBJECT *, long);
extern void object_size_cb(FL_OBJECT *, long);
extern void bumpiness_cb(FL_OBJECT *, long);
extern void eq_depth_cb(FL_OBJECT *, long);
extern void spread_scale_cb(FL_OBJECT *, long);
extern void position_key_cb(FL_OBJECT *, long);
extern void x_position_cb(FL_OBJECT *, long);
extern void y_position_cb(FL_OBJECT *, long);
extern void z_position_cb(FL_OBJECT *, long);
extern void channel_cb(FL_OBJECT *, long);
extern void recall_cb(FL_OBJECT *, long);
extern void w_adjust_cb(FL_OBJECT *, long);
extern void xy_locate_cb(FL_OBJECT *, long);
extern void z_locate_cb(FL_OBJECT *, long);
extern void key_feel_cb(FL_OBJECT *, long);
extern void copy_cb(FL_OBJECT *, long);
extern void ae_rotate_cb(FL_OBJECT *, long);
extern void t_rotate_cb(FL_OBJECT *, long);
extern void rotate_reset_cb(FL_OBJECT *, long);
extern void z_adjust_cb(FL_OBJECT *, long);
extern void delay_reset_cb(FL_OBJECT *, long);
extern void z_dominance_cb(FL_OBJECT *, long);
extern void dominance_reset_cb(FL_OBJECT *, long);
extern void xy_listener_cb(FL_OBJECT *, long);
extern void z_listener_cb(FL_OBJECT *, long);
extern void listener_reset_cb(FL_OBJECT *, long);
extern void xy_dominance_cb(FL_OBJECT *, long);
extern void jumpiness_cb(FL_OBJECT *, long);
extern void jump_mode_cb(FL_OBJECT *, long);
extern void spread_max_cb(FL_OBJECT *, long);
extern void spread_time_cb(FL_OBJECT *, long);
extern void delayRotate_cb(FL_OBJECT *, long);
extern void delayRotateReset_cb(FL_OBJECT *, long);


/**** Forms and Objects ****/

typedef struct {
    FL_FORM *LAmb;
    void *vdata;
    long ldata;
    FL_OBJECT *mute;
    FL_OBJECT *minus_clip;
    FL_OBJECT *plus_clip;
    FL_OBJECT *record;
    FL_OBJECT *play;
    FL_OBJECT *play_record;
    FL_OBJECT *audition;
    FL_OBJECT *auditionfx;
    FL_OBJECT *config_gp;
    FL_OBJECT *output_format;
    FL_OBJECT *permutation;
    FL_OBJECT *x_polarity;
    FL_OBJECT *y_polarity;
    FL_OBJECT *z_polarity;
    FL_OBJECT *audio_rate;
    FL_OBJECT *midi_source;
    FL_OBJECT *midi_channel;
    FL_OBJECT *midi_keyboard_type;
    FL_OBJECT *load_setup;
    FL_OBJECT *save_setup;
    FL_OBJECT *load_sample;
    FL_OBJECT *input_source;
    FL_OBJECT *quad_aspect;
    FL_OBJECT *w_master;
    FL_OBJECT *z_master;
    FL_OBJECT *buffer_out_size;
    FL_OBJECT *input_level1;
    FL_OBJECT *input_level2;
    FL_OBJECT *input_level3;
    FL_OBJECT *input_level4;
    FL_OBJECT *set_record;
    FL_OBJECT *set_play;
    FL_OBJECT *sample_base;
    FL_OBJECT *record_format;
    FL_OBJECT *set_audition;
    FL_OBJECT *main_gp;
    FL_OBJECT *direct_level;
    FL_OBJECT *level1;
    FL_OBJECT *level2;
    FL_OBJECT *level3;
    FL_OBJECT *feedback1;
    FL_OBJECT *feedback2;
    FL_OBJECT *feedback3;
    FL_OBJECT *time2;
    FL_OBJECT *time3;
    FL_OBJECT *master_time;
    FL_OBJECT *volume;
    FL_OBJECT *object_size;
    FL_OBJECT *bumpiness;
    FL_OBJECT *eq_depth;
    FL_OBJECT *spread_scale;
    FL_OBJECT *position_key;
    FL_OBJECT *x_position;
    FL_OBJECT *y_position;
    FL_OBJECT *z_position;
    FL_OBJECT *channel1;
    FL_OBJECT *channel2;
    FL_OBJECT *channel3;
    FL_OBJECT *channel4;
    FL_OBJECT *recall;
    FL_OBJECT *w_adjust;
    FL_OBJECT *xy_locate;
    FL_OBJECT *z_locate;
    FL_OBJECT *key_feel;
    FL_OBJECT *copy;
    FL_OBJECT *ae_rotate;
    FL_OBJECT *t_rotate;
    FL_OBJECT *rotate_reset;
    FL_OBJECT *z_adjust;
    FL_OBJECT *delay_reset;
    FL_OBJECT *z_dominance;
    FL_OBJECT *dominance_reset;
    FL_OBJECT *xy_listener;
    FL_OBJECT *z_listener;
    FL_OBJECT *listener_reset;
    FL_OBJECT *xy_dominance;
    FL_OBJECT *jumpiness;
    FL_OBJECT *jump_mode;
    FL_OBJECT *spread_max;
    FL_OBJECT *spread_time;
    FL_OBJECT *rotate2;
    FL_OBJECT *rotate3;
    FL_OBJECT *delayRotateReset;
    FL_OBJECT *time1;
} FD_LAmb;

extern FD_LAmb * create_form_LAmb(void);

#endif /* FD_LAmb_h_ */
```

E.6 f.c

```
/*
 * f.c
 *
 * Form definition file generated with fdesign.
 *
 */

#include "forms.h"
#include <stdlib.h>
#include "f.h"

FD_LAmb *create_form_LAmb(void)
{
  FL_OBJECT *obj;
  FD_LAmb *fdui = (FD_LAmb *) fl_calloc(1, sizeof(*fdui));

  fdui->LAmb = fl_bgn_form(FL_NO_BOX, 980, 730);
  obj = fl_add_box(FL_UP_BOX,0,0,980,730,"");
    fl_set_object_color(obj,FL_INACTIVE,FL_COL1);
  obj = fl_add_button(FL_HIDDEN_BUTTON,30,20,600,50,"Button");
    fl_set_button_shortcut(obj," ",1);
    fl_set_object_callback(obj,swap_page_cb,0);
  obj = fl_add_text(FL_NORMAL_TEXT,30,20,600,50,"   LAmb 1.05");
    fl_set_object_boxtype(obj,FL_FRAME_BOX);
    fl_set_object_color(obj,FL_WHITE,FL_MCOL);
    fl_set_object_lsize(obj,FL_HUGE_SIZE);
    fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
  obj = fl_add_text(FL_NORMAL_TEXT,211,35,200,30,"Performance Page");
    fl_set_object_color(obj,FL_WHITE,FL_MCOL);
    fl_set_object_lsize(obj,FL_LARGE_SIZE);
    fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
  obj = fl_add_box(FL_UP_BOX,670,20,280,60,"");
  obj = fl_add_text(FL_NORMAL_TEXT,440,41,175,23,"(c)  Dylan Menzies-Gow, University of York");
    fl_set_object_color(obj,FL_WHITE,FL_MCOL);
    fl_set_object_lsize(obj,FL_TINY_SIZE);
    fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
  fdui->mute = obj = fl_add_button(FL_PUSH_BUTTON,895,30,45,40,"Mute");
    fl_set_button_shortcut(obj,"m",1);
    fl_set_object_color(obj,FL_DODGERBLUE,FL_WHITE);
    fl_set_object_lsize(obj,FL_NORMAL_SIZE);
    fl_set_object_lstyle(obj,FL_BOLD_STYLE);
    fl_set_object_callback(obj,mute_cb,0);
  obj = fl_add_button(FL_HIDDEN_BUTTON,920,20,10,10,"");
    fl_set_button_shortcut(obj,"^D",1);
    fl_set_object_callback(obj,debug_cb,0);
  fdui->minus_clip = obj = fl_add_text(FL_NORMAL_TEXT,510,22,60,20,"-Clip!");
    fl_set_object_color(obj,FL_WHITE,FL_WHITE);
    fl_set_object_lsize(obj,FL_NORMAL_SIZE);
    fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
    fl_set_object_lstyle(obj,FL_TIMESBOLDITALIC_STYLE);
  fdui->plus_clip = obj = fl_add_text(FL_NORMAL_TEXT,564,22,60,20,"+Clip!");
    fl_set_object_color(obj,FL_WHITE,FL_WHITE);
    fl_set_object_lsize(obj,FL_NORMAL_SIZE);
    fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
    fl_set_object_lstyle(obj,FL_TIMESBOLDITALIC_STYLE);
  fdui->record = obj = fl_add_button(FL_PUSH_BUTTON,750,30,45,40,"Rec");
    fl_set_button_shortcut(obj,"r",1);
    fl_set_object_color(obj,FL_RED,FL_WHITE);
    fl_set_object_lsize(obj,FL_NORMAL_SIZE);
    fl_set_object_lstyle(obj,FL_BOLD_STYLE);
    fl_set_object_callback(obj,record_cb,0);
  fdui->play = obj = fl_add_button(FL_PUSH_BUTTON,680,30,45,40,"Play");
    fl_set_button_shortcut(obj,"p",1);
    fl_set_object_color(obj,FL_GREEN,FL_WHITE);
    fl_set_object_lsize(obj,FL_NORMAL_SIZE);
    fl_set_object_lstyle(obj,FL_BOLD_STYLE);
    fl_set_object_callback(obj,play_cb,0);
  fdui->play_record = obj = fl_add_button(FL_PUSH_BUTTON,730,30,15,40,"");
    fl_set_button_shortcut(obj,"^R",1);
    fl_set_object_color(obj,FL_RED,FL_WHITE);
    fl_set_object_callback(obj,play_record_cb,0);
  fdui->audition = obj = fl_add_button(FL_PUSH_BUTTON,800,30,40,40,"Aud");
    fl_set_button_shortcut(obj,"a",1);
    fl_set_object_color(obj,FL_DARKVIOLET,FL_WHITE);
    fl_set_object_lsize(obj,FL_NORMAL_SIZE);
    fl_set_object_lstyle(obj,FL_BOLD_STYLE);
    fl_set_object_callback(obj,audition_cb,0);
  fdui->auditionfx = obj = fl_add_button(FL_PUSH_BUTTON,847,30,40,40,"+FX");
```

```
    fl_set_button_shortcut(obj,"^A",1);
    fl_set_object_color(obj,FL_DARKVIOLET,FL_WHITE);
    fl_set_object_lsize(obj,FL_NORMAL_SIZE);
    fl_set_object_lstyle(obj,FL_BOLD_STYLE);
    fl_set_object_callback(obj,audition_cb,1);

  fdui->config_gp = fl_bgn_group();
  obj = fl_add_box(FL_UP_BOX,360,240,260,180,"");
  obj = fl_add_box(FL_UP_BOX,40,90,260,570,"");
  fdui->output_format = obj = fl_add_choice(FL_NORMAL_CHOICE,140,540,130,20,"Output Format");
    fl_set_object_boxtype(obj,FL_DOWN_BOX);
    fl_set_object_color(obj,FL_SLATEBLUE,FL_BLACK);
    fl_set_object_callback(obj,output_format_cb,0);
  fdui->permutation = obj = fl_add_choice(FL_NORMAL_CHOICE,140,420,130,20,"Permutation");
    fl_set_object_boxtype(obj,FL_DOWN_BOX);
    fl_set_object_color(obj,FL_DARKORANGE,FL_BLACK);
    fl_set_object_callback(obj,permutation_cb,0);
  fdui->x_polarity = obj = fl_add_choice(FL_NORMAL_CHOICE,140,450,130,20,"X Polarity");
    fl_set_object_boxtype(obj,FL_DOWN_BOX);
    fl_set_object_color(obj,FL_DARKCYAN,FL_BLACK);
    fl_set_object_callback(obj,x_polarity_cb,0);
  obj = fl_add_text(FL_NORMAL_TEXT,110,670,120,30,"Audio I/O");
    fl_set_object_boxtype(obj,FL_UP_BOX);
    fl_set_object_color(obj,FL_WHITE,FL_MCOL);
    fl_set_object_lalign(obj,FL_ALIGN_CENTER|FL_ALIGN_INSIDE);
    fl_set_object_lstyle(obj,FL_FIXEDBOLD_STYLE);
  fdui->y_polarity = obj = fl_add_choice(FL_NORMAL_CHOICE,140,480,130,20,"Y Polarity");
    fl_set_object_boxtype(obj,FL_DOWN_BOX);
    fl_set_object_color(obj,FL_DARKCYAN,FL_BLACK);
    fl_set_object_callback(obj,y_polarity_cb,0);
  fdui->z_polarity = obj = fl_add_choice(FL_NORMAL_CHOICE,140,510,130,20,"Z Polarity");
    fl_set_object_boxtype(obj,FL_DOWN_BOX);
    fl_set_object_color(obj,FL_DARKCYAN,FL_BLACK);
    fl_set_object_callback(obj,z_polarity_cb,0);
  fdui->audio_rate = obj = fl_add_choice(FL_NORMAL_CHOICE,140,270,130,20,"Audio Rate");
    fl_set_object_boxtype(obj,FL_DOWN_BOX);
    fl_set_object_color(obj,FL_DARKGOLD,FL_BLACK);
    fl_set_object_callback(obj,audio_rate_cb,0);
  fdui->midi_source = obj = fl_add_choice(FL_NORMAL_CHOICE,460,270,130,20,"MIDI Source");
    fl_set_object_boxtype(obj,FL_DOWN_BOX);
    fl_set_object_color(obj,FL_DARKGOLD,FL_BLACK);
    fl_set_object_callback(obj,midi_source_cb,0);
  fdui->midi_channel = obj = fl_add_input(FL_INT_INPUT,495,340,30,25,"MIDI Channel");
    fl_set_object_boxtype(obj,FL_SHADOW_BOX);
    fl_set_object_color(obj,FL_PALEGREEN,FL_PALEGREEN);
    fl_set_object_callback(obj,midi_channel_cb,0);
  fdui->midi_keyboard_type = obj = fl_add_choice(FL_NORMAL_CHOICE,460,300,130,20,"Keyboard Type");
    fl_set_object_boxtype(obj,FL_DOWN_BOX);
    fl_set_object_color(obj,FL_DARKORANGE,FL_BLACK);
    fl_set_object_callback(obj,midi_keyboard_type_cb,0);
  obj = fl_add_text(FL_NORMAL_TEXT,450,430,90,30,"MIDI");
    fl_set_object_boxtype(obj,FL_UP_BOX);
    fl_set_object_color(obj,FL_WHITE,FL_MCOL);
    fl_set_object_lalign(obj,FL_ALIGN_CENTER|FL_ALIGN_INSIDE);
    fl_set_object_lstyle(obj,FL_FIXEDBOLD_STYLE);
  obj = fl_add_box(FL_UP_BOX,360,500,260,160,"");
  obj = fl_add_text(FL_NORMAL_TEXT,445,670,90,30,"Files");
    fl_set_object_boxtype(obj,FL_UP_BOX);
    fl_set_object_color(obj,FL_WHITE,FL_MCOL);
    fl_set_object_lalign(obj,FL_ALIGN_CENTER|FL_ALIGN_INSIDE);
    fl_set_object_lstyle(obj,FL_FIXEDBOLD_STYLE);
  fdui->load_setup = obj = fl_add_button(FL_NORMAL_BUTTON,385,520,100,30,"Load Setup");
    fl_set_button_shortcut(obj,"l",1);
    fl_set_object_color(obj,FL_DARKORANGE,FL_WHITE);
    fl_set_object_callback(obj,load_setup_cb,0);
  fdui->save_setup = obj = fl_add_button(FL_NORMAL_BUTTON,385,565,100,30,"Save Setup");
    fl_set_button_shortcut(obj,"s",1);
    fl_set_object_color(obj,FL_DARKORANGE,FL_WHITE);
    fl_set_object_callback(obj,save_setup_cb,0);
  fdui->load_sample = obj = fl_add_button(FL_NORMAL_BUTTON,385,610,100,30,"Load Samples");
    fl_set_object_color(obj,FL_INDIANRED,FL_WHITE);
    fl_set_object_callback(obj,load_sample_cb,0);
  fdui->input_source = obj = fl_add_choice(FL_NORMAL_CHOICE,140,110,130,20,"Input Source");
    fl_set_object_boxtype(obj,FL_DOWN_BOX);
    fl_set_object_color(obj,FL_INDIANRED,FL_BLACK);
    fl_set_object_callback(obj,input_source_cb,0);
```

```
fdui->quad_aspect = obj = fl_add_input(FL_FLOAT_INPUT,220,610,45,25,"Quad Aspect, front : side?");
  fl_set_object_boxtype(obj,FL_SHADOW_BOX);
  fl_set_object_color(obj,FL_SLATEBLUE,FL_SLATEBLUE);
  fl_set_object_gravity(obj, FL_East, FL_NoGravity);
  fl_set_object_callback(obj,quad_aspect_cb,0);
fdui->w_master = obj = fl_add_slider(FL_HOR_SLIDER,130,370,140,10,"W Master");
  fl_set_object_color(obj,FL_TOMATO,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,w_master_cb,0);
fdui->z_master = obj = fl_add_slider(FL_HOR_SLIDER,130,390,140,10,"Z Master");
  fl_set_object_color(obj,FL_TOMATO,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,z_master_cb,0);
fdui->buffer_out_size = obj = fl_add_input(FL_INT_INPUT,140,305,60,25,"Buffer Size");
  fl_set_object_boxtype(obj,FL_SHADOW_BOX);
  fl_set_object_color(obj,FL_SLATEBLUE,FL_SLATEBLUE);
  fl_set_object_gravity(obj, FL_East, FL_NoGravity);
  fl_set_object_callback(obj,buffer_out_size_cb,0);
obj = fl_add_box(FL_UP_BOX,680,500,260,160,"");
obj = fl_add_text(FL_NORMAL_TEXT,730,670,160,30,"Control Rates");
  fl_set_object_boxtype(obj,FL_UP_BOX);
  fl_set_object_color(obj,FL_WHITE,FL_MCOL);
  fl_set_object_lalign(obj,FL_ALIGN_CENTER|FL_ALIGN_INSIDE);
  fl_set_object_lstyle(obj,FL_FIXEDBOLD_STYLE);
obj = fl_add_slider(FL_HOR_SLIDER,770,520,140,10,"GUI  ");
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
obj = fl_add_slider(FL_HOR_SLIDER,770,540,140,10,"Locate");
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
obj = fl_add_slider(FL_HOR_SLIDER,770,559,140,10,"Rotate");
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
obj = fl_add_slider(FL_HOR_SLIDER,770,579,140,11,"Dominance");
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
fdui->input_level1 = obj = fl_add_slider(FL_HOR_SLIDER,130,150,140,10,"Input level 1");
  fl_set_object_color(obj,FL_WHEAT,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,input_level_cb,0);
fdui->input_level2 = obj = fl_add_slider(FL_HOR_SLIDER,130,170,140,10,"Input level 2");
  fl_set_object_color(obj,FL_WHEAT,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,input_level_cb,1);
fdui->input_level3 = obj = fl_add_slider(FL_HOR_SLIDER,130,190,140,10,"Input level 3");
  fl_set_object_color(obj,FL_WHEAT,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,input_level_cb,2);
fdui->input_level4 = obj = fl_add_slider(FL_HOR_SLIDER,130,210,140,10,"Input level 4");
  fl_set_object_color(obj,FL_WHEAT,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,input_level_cb,3);
fdui->set_record = obj = fl_add_button(FL_NORMAL_BUTTON,495,565,100,30,"Set Record File");
  fl_set_object_color(obj,FL_SLATEBLUE,FL_WHITE);
  fl_set_object_callback(obj,set_record_cb,0);
fdui->set_play = obj = fl_add_button(FL_NORMAL_BUTTON,495,520,100,30,"Set Play File");
  fl_set_object_color(obj,FL_SLATEBLUE,FL_WHITE);
  fl_set_object_callback(obj,set_play_cb,0);
obj = fl_add_box(FL_UP_BOX,680,210,260,210,"");
obj = fl_add_text(FL_NORMAL_TEXT,735,430,160,30,"Shortcut Keys");
  fl_set_object_boxtype(obj,FL_UP_BOX);
  fl_set_object_color(obj,FL_WHITE,FL_MCOL);
  fl_set_object_lalign(obj,FL_ALIGN_CENTER|FL_ALIGN_INSIDE);
  fl_set_object_lstyle(obj,FL_FIXEDBOLD_STYLE);
obj = fl_add_text(FL_NORMAL_TEXT,710,260,210,20," CTRL R    Begin playing and recording");
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
obj = fl_add_text(FL_NORMAL_TEXT,740,320,180,20,"  L     Load setup");
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
obj = fl_add_text(FL_NORMAL_TEXT,740,340,190,20,"  S     Save setup");
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
```

```
obj = fl_add_text(FL_NORMAL_TEXT,741,220,190,20,"  P    Begin playing a soundfile");
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
obj = fl_add_text(FL_NORMAL_TEXT,741,240,190,20,"  R    Begin recording to a soundfile");
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
obj = fl_add_text(FL_NORMAL_TEXT,710,360,220,20,"  SPACE    Switch page");
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
obj = fl_add_text(FL_NORMAL_TEXT,210,35,210,30,"Configuration Page");
  fl_set_object_color(obj,FL_WHITE,FL_MCOL);
  fl_set_object_lsize(obj,FL_LARGE_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
fdui->sample_base = obj = fl_add_input(FL_INT_INPUT,495,375,40,25,"Sample Base Key");
  fl_set_object_boxtype(obj,FL_SHADOW_BOX);
  fl_set_object_color(obj,FL_SLATEBLUE,FL_SLATEBLUE);
  fl_set_object_callback(obj,sample_base_cb,0);
obj = fl_add_text(FL_NORMAL_TEXT,710,380,200,20,"Z,X,C,V     Select channel 1,2,3,4");
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
fdui->record_format = obj = fl_add_choice(FL_NORMAL_CHOICE,140,570,130,20,"Rec/Aud Format");
  fl_set_object_boxtype(obj,FL_DOWN_BOX);
  fl_set_object_color(obj,FL_SLATEBLUE,FL_BLACK);
  fl_set_object_callback(obj,record_format_cb,0);
obj = fl_add_text(FL_NORMAL_TEXT,746,280,155,20,"A    Audition a recoded file");
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
fdui->set_audition = obj = fl_add_button(FL_NORMAL_BUTTON,495,610,100,30,"Set Audition File");
  fl_set_object_color(obj,FL_SLATEBLUE,FL_WHITE);
  fl_set_object_callback(obj,set_audition_cb,0);
obj = fl_add_text(FL_NORMAL_TEXT,712,300,206,20,"CTRL A    Audition through the effects");
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
fl_end_group();


fdui->main_gp = fl_bgn_group();
obj = fl_add_box(FL_UP_BOX,40,530,260,140,"");
obj = fl_add_box(FL_UP_BOX,360,530,260,140,"");
obj = fl_add_text(FL_NORMAL_TEXT,460,640,40,20,"x y");
  fl_set_object_lalign(obj,FL_ALIGN_CENTER|FL_ALIGN_INSIDE);
obj = fl_add_text(FL_NORMAL_TEXT,556,640,20,20,"z");
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
obj = fl_add_text(FL_NORMAL_TEXT,140,640,40,20,"x y");
  fl_set_object_lalign(obj,FL_ALIGN_CENTER|FL_ALIGN_INSIDE);
obj = fl_add_text(FL_NORMAL_TEXT,240,640,20,20,"z");
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
obj = fl_add_box(FL_UP_BOX,40,90,578,360,"");
obj = fl_add_frame(FL_ENGRAVED_FRAME,330,250,250,180,"");
  fl_set_object_color(obj,FL_COL1,FL_COL1);
obj = fl_add_box(FL_UP_BOX,680,110,260,340,"");
obj = fl_add_text(FL_NORMAL_TEXT,720,460,180,30,"Soundfield Delay");
  fl_set_object_boxtype(obj,FL_UP_BOX);
  fl_set_object_color(obj,FL_WHITE,FL_MCOL);
  fl_set_object_lalign(obj,FL_ALIGN_CENTER|FL_ALIGN_INSIDE);
  fl_set_object_lstyle(obj,FL_FIXEDBOLD_STYLE);
fdui->direct_level = obj = fl_add_slider(FL_HOR_NICE_SLIDER,780,125,140,10,"Direct Level");
  fl_set_object_color(obj,FL_SLATEBLUE,FL_BLACK);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,direct_level_cb,0);
fdui->level1 = obj = fl_add_slider(FL_HOR_SLIDER,780,155,140,10,"Level 1");
  fl_set_object_color(obj,FL_DARKGOLD,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,level_cb,0);
fdui->level2 = obj = fl_add_slider(FL_HOR_SLIDER,780,175,140,10,"Level 2");
  fl_set_object_color(obj,FL_DARKGOLD,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,level_cb,1);
fdui->level3 = obj = fl_add_slider(FL_HOR_SLIDER,780,195,140,10,"Level 3");
  fl_set_object_color(obj,FL_DARKGOLD,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,level_cb,2);
fdui->feedback1 = obj = fl_add_slider(FL_HOR_SLIDER,780,225,140,10,"Feedback 1");
  fl_set_object_color(obj,FL_DARKORANGE,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,feedback_cb,0);
fdui->feedback2 = obj = fl_add_slider(FL_HOR_SLIDER,780,245,140,10,"Feedback 2");
```

```
  fl_set_object_color(obj,FL_DARKORANGE,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,feedback_cb,1);
fdui->feedback3 = obj = fl_add_slider(FL_HOR_SLIDER,780,265,140,10,"Feedback 3");
  fl_set_object_color(obj,FL_DARKORANGE,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,feedback_cb,2);
fdui->time2 = obj = fl_add_slider(FL_HOR_SLIDER,780,370,140,10,"Delay 2");
  fl_set_object_color(obj,FL_DARKCYAN,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,time_cb,1);
fdui->time3 = obj = fl_add_slider(FL_HOR_SLIDER,780,390,140,10,"Delay 3");
  fl_set_object_color(obj,FL_DARKCYAN,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,time_cb,2);
fdui->master_time = obj = fl_add_slider(FL_HOR_NICE_SLIDER,780,420,140,10,"Time Scale");
  fl_set_object_color(obj,FL_PALEGREEN,FL_BLACK);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,master_time_cb,0);
obj = fl_add_text(FL_NORMAL_TEXT,250,457,160,30,"Locate Object");
  fl_set_object_boxtype(obj,FL_UP_BOX);
  fl_set_object_color(obj,FL_WHITE,FL_MCOL);
  fl_set_object_lalign(obj,FL_ALIGN_CENTER|FL_ALIGN_INSIDE);
  fl_set_object_lstyle(obj,FL_FIXEDBOLD_STYLE);
fdui->volume = obj = fl_add_slider(FL_HOR_SLIDER,140,150,140,10,"Volume");
  fl_set_object_color(obj,FL_SLATEBLUE,FL_BLACK);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,volume_cb,0);
fdui->object_size = obj = fl_add_slider(FL_HOR_SLIDER,140,170,140,10,"Object Size");
  fl_set_object_color(obj,FL_DARKGOLD,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,object_size_cb,0);
fdui->bumpiness = obj = fl_add_slider(FL_HOR_SLIDER,140,190,140,10,"Bumpiness");
  fl_set_object_color(obj,FL_DARKORANGE,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,bumpiness_cb,0);
fdui->eq_depth = obj = fl_add_slider(FL_HOR_SLIDER,140,250,140,10,"HF attenuation");
  fl_set_object_color(obj,FL_DARKCYAN,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,eq_depth_cb,0);
fdui->spread_scale = obj = fl_add_slider(FL_HOR_SLIDER,140,270,140,10,"Spread Scale");
  fl_set_object_color(obj,FL_YELLOW,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,spread_scale_cb,0);
fdui->position_key = obj = fl_add_choice(FL_NORMAL_CHOICE,420,270,80,20,"Position Key");
  fl_set_object_boxtype(obj,FL_DOWN_BOX);
  fl_set_object_color(obj,FL_INDIANRED,FL_BLACK);
  fl_set_object_callback(obj,position_key_cb,0);
fdui->x_position = obj = fl_add_input(FL_FLOAT_INPUT,420,300,50,25,"X position");
  fl_set_object_boxtype(obj,FL_SHADOW_BOX);
  fl_set_object_color(obj,FL_RED,FL_RED);
  fl_set_object_callback(obj,x_position_cb,0);
fdui->y_position = obj = fl_add_input(FL_FLOAT_INPUT,420,330,50,25,"Y position");
  fl_set_object_boxtype(obj,FL_SHADOW_BOX);
  fl_set_object_color(obj,FL_RED,FL_RED);
  fl_set_object_callback(obj,y_position_cb,0);
fdui->z_position = obj = fl_add_input(FL_FLOAT_INPUT,420,360,50,25,"Z position");
  fl_set_object_boxtype(obj,FL_SHADOW_BOX);
  fl_set_object_color(obj,FL_RED,FL_RED);
  fl_set_object_callback(obj,z_position_cb,0);
obj = fl_add_text(FL_NORMAL_TEXT,70,110,70,20,"Channel");
  fl_set_object_lalign(obj,FL_ALIGN_RIGHT|FL_ALIGN_INSIDE);
fdui->channel1 = obj = fl_add_lightbutton(FL_PUSH_BUTTON,140,110,30,20,"1");
  fl_set_button_shortcut(obj,"z",1);
  fl_set_object_color(obj,FL_BLACK,FL_YELLOW);
  fl_set_object_lsize(obj,FL_TINY_SIZE);
  fl_set_object_callback(obj,channel_cb,0);
fdui->channel2 = obj = fl_add_lightbutton(FL_PUSH_BUTTON,170,110,30,20,"2");
  fl_set_button_shortcut(obj,"x",1);
  fl_set_object_color(obj,FL_BLACK,FL_YELLOW);
  fl_set_object_lsize(obj,FL_TINY_SIZE);
  fl_set_object_callback(obj,channel_cb,1);
fdui->channel3 = obj = fl_add_lightbutton(FL_PUSH_BUTTON,200,110,30,20,"3");
  fl_set_button_shortcut(obj,"c",1);
  fl_set_object_color(obj,FL_BLACK,FL_YELLOW);
  fl_set_object_lsize(obj,FL_TINY_SIZE);
  fl_set_object_callback(obj,channel_cb,2);
fdui->channel4 = obj = fl_add_lightbutton(FL_PUSH_BUTTON,230,110,30,20,"4");
  fl_set_button_shortcut(obj,"v",1);
  fl_set_object_color(obj,FL_BLACK,FL_YELLOW);
  fl_set_object_lsize(obj,FL_TINY_SIZE);
  fl_set_object_callback(obj,channel_cb,3);
fdui->recall = obj = fl_add_menu(FL_PUSH_MENU,489,350,50,30,"Recall");
  fl_set_object_boxtype(obj,FL_UP_BOX);
  fl_set_object_color(obj,FL_DODGERBLUE,FL_WHITE);
  fl_set_object_lstyle(obj,FL_NORMAL_STYLE);
  fl_set_object_callback(obj,recall_cb,0);
fdui->w_adjust = obj = fl_add_slider(FL_HOR_SLIDER,140,210,140,10,"W adjust");
  fl_set_object_color(obj,FL_TOMATO,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,w_adjust_cb,0);
fdui->xy_locate = obj = fl_add_positioner(FL_NORMAL_POSITIONER,390,110,120,110,"");
  fl_set_object_color(obj,FL_DARKORANGE,FL_BLACK);
  fl_set_object_lalign(obj,FL_ALIGN_BOTTOM|FL_ALIGN_INSIDE);
  fl_set_object_callback(obj,xy_locate_cb,0);
fdui->z_locate = obj = fl_add_slider(FL_VERT_SLIDER,540,110,10,110,"");
  fl_set_object_color(obj,FL_DARKGOLD,FL_COL1);
  fl_set_object_callback(obj,z_locate_cb,0);
fdui->key_feel = obj = fl_add_slider(FL_HOR_SLIDER,400,400,140,10,"Key Feel");
  fl_set_object_color(obj,FL_INDIANRED,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,key_feel_cb,0);
obj = fl_add_text(FL_NORMAL_TEXT,442,223,30,20,"x y");
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
obj = fl_add_text(FL_NORMAL_TEXT,560,150,20,25,"z");
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
fdui->copy = obj = fl_add_button(FL_NORMAL_BUTTON,489,310,50,30,"Copy");
  fl_set_object_color(obj,FL_DODGERBLUE,FL_WHITE);
  fl_set_object_callback(obj,copy_cb,0);
obj = fl_add_slider(FL_HOR_SLIDER,140,330,140,10,"Refl. Scale");
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
obj = fl_add_slider(FL_HOR_SLIDER,140,350,140,10,"Refl. Atten.");
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
obj = fl_add_box(FL_UP_BOX,680,530,260,140,"");
obj = fl_add_text(FL_NORMAL_TEXT,720,680,190,30,"Soundfield Rotate");
  fl_set_object_boxtype(obj,FL_UP_BOX);
  fl_set_object_color(obj,FL_WHITE,FL_MCOL);
  fl_set_object_lalign(obj,FL_ALIGN_CENTER|FL_ALIGN_INSIDE);
  fl_set_object_lstyle(obj,FL_FIXEDBOLD_STYLE);
fdui->ae_rotate = obj = fl_add_positioner(FL_NORMAL_POSITIONER,750,550,100,90,"");
  fl_set_object_color(obj,FL_DARKORANGE,FL_BLACK);
  fl_set_object_lalign(obj,FL_ALIGN_BOTTOM|FL_ALIGN_INSIDE);
  fl_set_object_callback(obj,ae_rotate_cb,0);
fdui->t_rotate = obj = fl_add_slider(FL_VERT_SLIDER,880,550,10,90,"");
  fl_set_object_color(obj,FL_DARKGOLD,FL_COL1);
  fl_set_object_callback(obj,t_rotate_cb,0);
obj = fl_add_text(FL_NORMAL_TEXT,745,640,110,20,"Azimuth, Elevation");
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
obj = fl_add_text(FL_NORMAL_TEXT,865,640,40,20,"Twist");
  fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
obj = fl_add_slider(FL_HOR_SLIDER,140,370,140,10,"Bonus Weird");
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
fdui->rotate_reset = obj = fl_add_button(FL_NORMAL_BUTTON,700,580,30,30,"0");
  fl_set_object_color(obj,FL_RED,FL_COL1);
  fl_set_object_callback(obj,rotate_reset_cb,0);
fdui->z_adjust = obj = fl_add_slider(FL_HOR_SLIDER,140,230,140,10,"Z adjust");
  fl_set_object_color(obj,FL_TOMATO,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,z_adjust_cb,0);
fdui->delay_reset = obj = fl_add_button(FL_PUSH_BUTTON,694,165,30,30,"P");
  fl_set_object_color(obj,FL_RED,FL_WHITE);
  fl_set_object_callback(obj,delay_reset_cb,0);
obj = fl_add_button(FL_HIDDEN_BUTTON,640,80,20,10,"");
  fl_set_button_shortcut(obj,"s",1);
  fl_set_object_callback(obj,save_setup_cb,0);
obj = fl_add_text(FL_NORMAL_TEXT,430,680,120,30,"Dominance");
  fl_set_object_boxtype(obj,FL_UP_BOX);
  fl_set_object_color(obj,FL_WHITE,FL_MCOL);
  fl_set_object_lalign(obj,FL_ALIGN_CENTER|FL_ALIGN_INSIDE);
  fl_set_object_lstyle(obj,FL_FIXEDBOLD_STYLE);
fdui->z_dominance = obj = fl_add_slider(FL_VERT_SLIDER,560,550,10,90,"");
  fl_set_object_callback(obj,z_dominance_cb,0);
fdui->dominance_reset = obj = fl_add_button(FL_NORMAL_BUTTON,380,580,30,30,"0");
  fl_set_object_color(obj,FL_RED,FL_COL1);
  fl_set_object_callback(obj,dominance_reset_cb,0);
obj = fl_add_text(FL_NORMAL_TEXT,90,680,170,30,"Locate Listener");
  fl_set_object_boxtype(obj,FL_UP_BOX);
  fl_set_object_color(obj,FL_WHITE,FL_MCOL);
  fl_set_object_lalign(obj,FL_ALIGN_CENTER|FL_ALIGN_INSIDE);
  fl_set_object_lstyle(obj,FL_FIXEDBOLD_STYLE);
fdui->xy_listener = obj = fl_add_positioner(FL_NORMAL_POSITIONER,110,550,100,90,"");
  fl_set_object_color(obj,FL_DARKORANGE,FL_BLACK);
  fl_set_object_lalign(obj,FL_ALIGN_BOTTOM|FL_ALIGN_INSIDE);
  fl_set_object_callback(obj,xy_listener_cb,0);
fdui->z_listener = obj = fl_add_slider(FL_VERT_SLIDER,240,550,10,90,"");
  fl_set_object_color(obj,FL_DARKGOLD,FL_COL1);
  fl_set_object_callback(obj,z_listener_cb,0);
fdui->listener_reset = obj = fl_add_button(FL_NORMAL_BUTTON,60,580,30,30,"0");
  fl_set_object_color(obj,FL_RED,FL_COL1);
  fl_set_object_callback(obj,listener_reset_cb,0);
fdui->xy_dominance = obj = fl_add_positioner(FL_NORMAL_POSITIONER,430,550,100,90,"");
  fl_set_object_color(obj,FL_DARKORANGE,FL_BLACK);
  fl_set_object_lalign(obj,FL_ALIGN_BOTTOM|FL_ALIGN_INSIDE);
  fl_set_object_callback(obj,xy_dominance_cb,0);
fdui->jumpiness = obj = fl_add_slider(FL_HOR_SLIDER,140,390,140,10,"Jumpiness");
  fl_set_object_color(obj,FL_DARKORANGE,FL_BLACK);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,jumpiness_cb,0);
fdui->jump_mode = obj = fl_add_lightbutton(FL_PUSH_BUTTON,140,410,30,20,"Jump Snap");
  fl_set_object_color(obj,FL_BLACK,FL_YELLOW);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,jump_mode_cb,0);
fdui->spread_max = obj = fl_add_slider(FL_HOR_SLIDER,140,290,140,10,"Spread Max");
  fl_set_object_color(obj,FL_YELLOW,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,spread_max_cb,0);
fdui->spread_time = obj = fl_add_slider(FL_HOR_SLIDER,140,310,140,10,"Spread Wraps");
  fl_set_object_color(obj,FL_YELLOW,FL_BLACK);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,spread_time_cb,0);
fdui->rotate2 = obj = fl_add_slider(FL_HOR_SLIDER,780,295,140,10,"Rotate 2");
  fl_set_object_color(obj,FL_RED,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,delayRotate_cb,1);
  fl_set_slider_bounds(obj, 0.00, 127.00);
  fl_set_slider_value(obj, 64.00);
fdui->rotate3 = obj = fl_add_slider(FL_HOR_SLIDER,780,315,140,10,"Rotate 3");
  fl_set_object_color(obj,FL_RED,FL_COL1);
  fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
  fl_set_object_lalign(obj,FL_ALIGN_LEFT);
  fl_set_object_callback(obj,delayRotate_cb,2);
  fl_set_slider_bounds(obj, 0.00, 127.00);
  fl_set_slider_value(obj, 64.00);
fdui->delayRotateReset = obj = fl_add_button(FL_NORMAL_BUTTON,694,295,30,30,"0");
  fl_set_object_color(obj,FL_RED,FL_COL1);
  fl_set_object_callback(obj,delayRotateReset_cb,0);
fdui->time1 = obj = fl_add_slider(FL_HOR_SLIDER,780,350,140,10,"Delay 1");
  fl_set_object_color(obj,FL_DARKCYAN,FL_COL1);
```

```
    fl_set_object_lsize(obj,FL_DEFAULT_SIZE);
    fl_set_object_lalign(obj,FL_ALIGN_LEFT);
    fl_set_object_callback(obj,time_cb,0);
  obj = fl_add_button(FL_HIDDEN_BUTTON,640,90,20,20,"");
    fl_set_button_shortcut(obj,"l",1);
    fl_set_object_callback(obj,load_setup_cb,0);
  fl_end_group();

  fl_end_form();

  fdui->LAmb->fdui = fdui;

  return fdui;
}
/*---------------------------------------*/
```

E.7 f_cb_.c

```
/*
 * f_cb_.c
 *
 * Callbacks for form ‘LAmb’
 *
 */

void input_source_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(INPUT_SOURCE, fl_get_choice(ob));
}

void input_level_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(INPUT_LEVEL_BASE+data,  fl_get_slider_value(ob));
}

void output_format_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(OUTPUT_FORMAT, fl_get_choice(ob));
}

void record_format_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(RECORD_FORMAT, fl_get_choice(ob));
}

void quad_aspect_cb(FL_OBJECT *ob, long data)
{
    float aspect;

    aspect = (atof(fl_get_input(ob)));
    if (aspect>=0 && aspect <=2)
    slow_control_process(QUAD_ASPECT, 64.0*aspect);
}


void permutation_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(PERMUTATION, fl_get_choice(ob));
}

void x_polarity_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(X_POLARITY, fl_get_choice(ob));
}

void y_polarity_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(Y_POLARITY, fl_get_choice(ob));
}

void z_polarity_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(Z_POLARITY, fl_get_choice(ob));
```

```
}

void audio_rate_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(AUDIO_RATE, fl_get_choice(ob));
}

void midi_source_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(MIDI_SOURCE, fl_get_choice(ob));
}

void midi_channel_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(MIDI_CHANNEL, atof(fl_get_input(ob)));
}

void sample_base_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(SAMPLE_KB, atof(fl_get_input(ob)));
}

void midi_keyboard_type_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(MIDI_KEYBOARD_TYPE, fl_get_choice(ob));
}

/*  Probably don’t need this anymore..
void buffer_in_size_cb(FL_OBJECT *ob, long data)
{
    int size;
    size = atof(fl_get_input(ob));
    if (size < 1019 ) size = 1019;
    slow_control_process(BUFFER_IN_SIZE, size);
}
*/

void buffer_out_size_cb(FL_OBJECT *ob, long data)
{
    int size;
    size = atof(fl_get_input(ob));
    if (size < 1020 ) {
        size = 1020;
        fl_set_input(ob, "1020");
        }
    slow_control_process(BUFFER_OUT_SIZE, size);
}


void w_master_cb(FL_OBJECT *ob, long data)
{
    control_process(W_MASTER, fl_get_slider_value(ob));
}


void z_master_cb(FL_OBJECT *ob, long data)
{
    control_process(Z_MASTER, fl_get_slider_value(ob));
}



/******************* Locate Object **********************/

void channel_cb(FL_OBJECT *ob, long chan)
{
FD_LAmb *f = lamb_form;

    form_channel = chan;
    set_channel_stuff(form_channel);
}


void volume_cb(FL_OBJECT *ob, long data)
{
    control_process(VOLUME_BASE+form_channel, fl_get_slider_value(ob));
```

```
}

void object_size_cb(FL_OBJECT *ob, long data)
{
    control_process(OBJECT_SIZE_BASE+form_channel, fl_get_slider_value(ob));
}

void bumpiness_cb(FL_OBJECT *ob, long data)
{
    control_process(BUMPINESS_BASE+form_channel, fl_get_slider_value(ob));
}

void eq_depth_cb(FL_OBJECT *ob, long data)
{
    control_process(EQ_DEPTH_BASE+form_channel, fl_get_slider_value(ob));
}

void spread_scale_cb(FL_OBJECT *ob, long data)
{
    control_process(SPREAD_SCALE_BASE+form_channel, fl_get_slider_value(ob));
}

void spread_max_cb(FL_OBJECT *ob, long data)
{
    control_process(SPREAD_MAX_BASE+form_channel, fl_get_slider_value(ob));
}

void spread_time_cb(FL_OBJECT *ob, long data)
{
    control_process(SPREAD_TIME_BASE+form_channel, fl_get_slider_value(ob));
}

void w_adjust_cb(FL_OBJECT *ob, long data)
{
    control_process(W_ADJUST_BASE +form_channel, fl_get_slider_value(ob));
}

void z_adjust_cb(FL_OBJECT *ob, long data)
{
    control_process(Z_ADJUST_BASE +form_channel, fl_get_slider_value(ob));
}

void xy_locate_cb(FL_OBJECT *ob, long data)
{
    control_process(c_base[form_channel][1], fl_get_positioner_xvalue(ob));
    control_process(c_base[form_channel][1]+1, fl_get_positioner_yvalue(ob));

    /* Prevent external joystick MIDI jitter.*/
    joystick_line = -1;
}

void z_locate_cb(FL_OBJECT *ob, long data)
{
    control_process(c_base[form_channel][1]+2, fl_get_slider_value(ob));
    joystick_line = -1;
}

void jumpiness_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(form_channel+JUMPINESS_BASE, fl_get_slider_value(ob));
}

void jump_mode_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(form_channel+JUMP_MODE_BASE, fl_get_button(ob));
}



/**************** Position Key stuff ****************/

void position_key_cb(FL_OBJECT *ob, long data)
{
    form_position_key = fl_get_choice(ob);
    write_fixed_positions_to_form();
}
```

```
void x_position_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(POSITION_KEY_BASE+3*(form_position_key-1),
                (atof(fl_get_input(ob))+1)*64);

}

void y_position_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(POSITION_KEY_BASE+3*(form_position_key-1)+1,
                (atof(fl_get_input(ob))+1)*64);
}

void z_position_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(POSITION_KEY_BASE+3*(form_position_key-1)+2,
                (atof(fl_get_input(ob))+1)*64);
}

void recall_cb(FL_OBJECT *ob, long data)
{
    recall_positions(fl_get_menu(ob));
    write_fixed_positions_to_form();
}

void copy_cb(FL_OBJECT *ob,  long data)
{
    short n, m;

    slow_control_process(POSITION_KEY_BASE+3*(form_position_key-1),
    fl_get_positioner_xvalue(lamb_form->xy_locate) );

    slow_control_process(POSITION_KEY_BASE+3*(form_position_key-1)+1,
    fl_get_positioner_yvalue(lamb_form->xy_locate) );

    slow_control_process(POSITION_KEY_BASE+3*(form_position_key-1)+2,
    fl_get_slider_value(lamb_form->z_locate) );

    write_fixed_positions_to_form();

}


void key_feel_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(KEY_FEEL+form_channel, fl_get_slider_value(ob));
}

/************  DELAY  ********************/

void direct_level_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(DELAY_DIRECT_LEVEL, fl_get_slider_value(ob));
}

void level_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(DELAY_TAP_LEVEL_BASE+data, fl_get_slider_value(ob));
}

void feedback_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(DELAY_TAP_FEEDBACK_BASE+data, fl_get_slider_value(ob));
}

void time_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(DELAY_TAP_TIME_BASE+data, fl_get_slider_value(ob));
}

void master_time_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(MASTER_TIME, fl_get_slider_value(ob));
}
```

```
void delay_reset_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(RESET_DELAY, fl_get_button(ob));
}


void delayRotate_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(DELAY_TAP_ROTATE_BASE+data, fl_get_slider_value(ob));
}


void delayRotateReset_cb(FL_OBJECT *ob, long data)
{
    int i;
    for(i=0; i<3; i++) {
        slow_control_process(DELAY_TAP_ROTATE_BASE+i, 64);
    }
    fl_set_slider_value(lamb_form->rotate2,64);
    fl_set_slider_value(lamb_form->rotate3,64);
}



/****************** Rotate ********************/
void ae_rotate_cb(FL_OBJECT *ob, long data)
{
    control_process(ROTATE_BASE, fl_get_positioner_xvalue(ob));
    control_process(ROTATE_BASE+1, fl_get_positioner_yvalue(ob));

    /* Prevent external joystick MIDI jitter.*/
    /* Dosn’t bother to check first to see if joystick
     * is being used for a non-rotate task.
     */
    joystick_line = -1;
}

void t_rotate_cb(FL_OBJECT *ob, long data)
{
    control_process(ROTATE_BASE+2, fl_get_slider_value(ob));

    joystick_line = -1;
}

void rotate_reset_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(RESET_ROTATE, 1);

    fl_set_positioner_xvalue(lamb_form->ae_rotate,64);
    fl_set_positioner_yvalue(lamb_form->ae_rotate,64);
    fl_set_slider_value(lamb_form->t_rotate,64);
}



/****************  Files  *************************/

void load_setup_cb(FL_OBJECT *ob, long dummy)
{
    const char* c;
    c = fl_show_fselector("Load Settings", ".", "*.set", "");
    if (c == NULL) return;

    fl_check_forms();
    load_setup(c);
    loadsf();

}


void save_setup_cb(FL_OBJECT *ob, long data)
{
    const char* filename;

    fl_deactivate_form(lamb_form->LAmb);
    filename = fl_show_fselector("Save Settings", ".", "*.set", "");
    fl_activate_form(lamb_form->LAmb);
    if (filename == NULL) return;
    fl_check_forms();
    save_setup(filename);
    fl_refresh_fselector();
}



void load_sample_cb(FL_OBJECT *ob, long data)
{
    const char* c;
    c = fl_show_fselector("Select Sample Directory.", (const char*)filepath, "*.aiff", "");
    if (c == NULL) return;

    strcpy(filepath, fl_get_directory());  /* filepath global - need better name */
    strcat(filepath, "/");
    fl_check_forms();   /* Hide fl_selector so it dosen’t hang while loading sfs */
    fl_check_forms();

    loadsf();
}


void set_record_cb(FL_OBJECT *ob, long dummy)
{
    const char* filename;
    filename = fl_show_fselector("Set Record File", ".", "*.aiff", (const char*)record_file);
    if (filename == NULL) return;

    strcpy(record_file, filename);
}

void set_play_cb(FL_OBJECT *ob, long dummy)
{
    const char* filename;
    filename = fl_show_fselector("Set Play File", ".", "*.aiff", (const char*)play_file);
    if (filename == NULL) return;

    strcpy(play_file, filename);

    play_fh = AFopenfile( play_file, "r", 0);

    if (play_fh == AF_NULL_FILEHANDLE)
        fl_show_alert("Play file not found.", "", "",1);

    else if (AFgetrate(play_fh, AF_DEFAULT_TRACK) != audio_rate_table[(int)slow_control[AUDIO_RATE]])
        fl_show_alert("Warning: audio rate of play file different to current setting.", "", "",1);

    AFclosefile(play_fh);
    return;

}

void set_audition_cb(FL_OBJECT *ob, long dummy)
{
    const char* filename;
    filename = fl_show_fselector("Set Audition File", ".", "*.aiff", (const char*)audition_file);
    if (filename == NULL) return;

    strcpy(audition_file, filename);

    audition_fh = AFopenfile( audition_file, "r", 0);

    if (audition_fh == AF_NULL_FILEHANDLE)
        fl_show_alert("Audition file not found.", "", "",1);

    else if (AFgetrate(audition_fh, AF_DEFAULT_TRACK)
        != audio_rate_table[(int)slow_control[AUDIO_RATE]])
        fl_show_alert("Warning: audio rate of auidition
        file different to current setting.", "", "",1);

    AFclosefile(audition_fh);
    return;

}
```

```
/* This function may not be used.. Mute instead */
void panic_cb(FL_OBJECT *ob, long data)
{

    slow_control_process(PANIC, 1);

    /* Maybe include these in the slow_control_process? */
    fl_set_slider_value(lamb_form->volume, 0);
    fl_set_slider_value(lamb_form->level1, 0);
    fl_set_slider_value(lamb_form->level2, 0);
    fl_set_slider_value(lamb_form->level3, 0);
    fl_set_slider_value(lamb_form->feedback1, 0);
    fl_set_slider_value(lamb_form->feedback2, 0);
    fl_set_slider_value(lamb_form->feedback3, 0);

}

void mute_cb(FL_OBJECT *ob, long data)
{
    master_mute = fl_get_button(ob);
}


void gui_off_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(GUI_OFF, 1);
}


void play_cb(FL_OBJECT *ob, long data)
{
    if (fl_get_button(ob) == 1) {   /* Start / restart playing */
    if (play_fh != AF_NULL_FILEHANDLE) AFclosefile(play_fh);

    if (record_state==1 && strcmp(record_file, play_file) == 0) {
        fl_show_alert("Record and Play files are the same.", "", "",1);
        fl_set_button(ob, 0);
        fl_set_button(lamb_form->record, 0);
        record_cb(lamb_form->record, 0);
        return;
    }


    play_fh = AFopenfile( play_file, "r", 0);
    if (play_fh == AF_NULL_FILEHANDLE) {
        fl_show_alert("Play file not found.", "", "",1);
        fl_set_button(lamb_form->play_record, 0);       /* Kills record if on */
        play_record_cb(lamb_form->play_record, 0);
        return;
    }

    in_frame_channels = AFgetchannels(play_fh, AF_DEFAULT_TRACK);

    play_state = 1;
    }

    else {  /* Stop playing */
    if (play_fh != AF_NULL_FILEHANDLE) AFclosefile(play_fh);
    play_fh = AF_NULL_FILEHANDLE;
    play_state = 0;
    in_frame_channels = 4; /* Back to AL input */
    fl_set_button(lamb_form->play_record, 0);

    }

}


void record_cb(FL_OBJECT *ob, long data)
{
    AFfilesetup fs;

    if (fl_get_button(ob) == 1) {   /* Start / restart recording */
    if (record_fh != AF_NULL_FILEHANDLE) AFclosefile(record_fh);

    if (play_state==1 && strcmp(record_file, play_file) == 0) {
        fl_show_alert("Record and Play files are the same.", "", "",1);
        fl_set_button(ob, 0);
        fl_set_button(lamb_form->play, 0);
        play_cb(lamb_form->play, 0);
        return;
    }


    fs = AFnewfilesetup();
    AFinitfilefmt(fs, AF_FILE_AIFF);
    AFinitchannels(fs, AF_DEFAULT_TRACK, 4);
    AFinitrate(fs, AF_DEFAULT_TRACK, audio_rate_table[(int)slow_control[AUDIO_RATE]]);
    AFinitsampfmt(fs, AF_DEFAULT_TRACK,
                  AF_SAMPFMT_TWOSCOMP, 16); /*in bits */

        record_fh = AFopenfile( record_file, "w", fs);
    AFfreefilesetup(fs);
    if (record_fh == AF_NULL_FILEHANDLE) {
        fl_show_alert("Failed to open record file.", "", "",1);
        fl_set_button(lamb_form->play_record, 0);
        play_record_cb(lamb_form->play_record, 0);
    }
    else record_state = 1;
    }

    else {  /* Stop recording */
    if (record_fh != AF_NULL_FILEHANDLE) AFclosefile(record_fh);
    record_fh = AF_NULL_FILEHANDLE;
    record_state = 0;
    fl_set_button(lamb_form->play_record, 0);
    }
}


void play_record_cb(FL_OBJECT *ob, long data)
{
    if (fl_get_button(ob) == 1) {
    fl_set_button(lamb_form->play, 1);
    fl_set_button(lamb_form->record, 1);
    play_cb(lamb_form->play, 0);
    record_cb(lamb_form->record, 0);
    }

    else {
    fl_set_button(lamb_form->play, 0);
    fl_set_button(lamb_form->record, 0);
    play_cb(lamb_form->play, 0);
    record_cb(lamb_form->record, 0);

    }
}

void audition_cb(FL_OBJECT *ob, long data)
{
    AFfilesetup fs;

    if (fl_get_button(ob) == 1) {   /* Start playing */
    if (audition_fh != AF_NULL_FILEHANDLE) AFclosefile(audition_fh);

    if (record_state==1 && strcmp(record_file, audition_file) == 0) {
        fl_show_alert("Audition and record files are the same.", "", "",1);
        fl_set_button(ob, 0);
        audition_cb(lamb_form->audition, 0);
        return;
    }

    if (record_format == 1 && data == 1) {  /* ie decoded audition file and +FX selected. */
        fl_show_alert("Rec/Aud format must be 'Raw B-format' to add FX.", "", "", 1);
        fl_set_button(ob, 0);
        audition_cb(lamb_form->audition, 0);
        return;
    }
    fs = AFnewfilesetup();
    AFinitfilefmt(fs, AF_FILE_AIFF);
    AFinitchannels(fs, AF_DEFAULT_TRACK, 4);
    AFinitrate(fs, AF_DEFAULT_TRACK, audio_rate_table[(int)slow_control[AUDIO_RATE]]);
    AFinitsampfmt(fs, AF_DEFAULT_TRACK,
                  AF_SAMPFMT_TWOSCOMP, 16); /*in bits */

        audition_fh = AFopenfile( audition_file, "r", fs);
    AFfreefilesetup(fs);
    if (audition_fh == AF_NULL_FILEHANDLE) {
        fl_show_alert("Failed to open audition file.", "", "",1);
        fl_set_button(ob, 0);
        audition_cb(lamb_form->audition, 0);
        return;
    }

    fl_set_button(lamb_form->audition, 1);  /* Looks nice with both Aud and +FX on */
    audition_state = record_format; /* ie audition format = record format */
    if (data==1) audition_state = 3; /* Add audition before FX */
    }

    else {  /* Stop auditioning */
    if (audition_fh != AF_NULL_FILEHANDLE) AFclosefile(audition_fh);
    audition_fh = AF_NULL_FILEHANDLE;
    fl_set_button(lamb_form->audition, 0);
    fl_set_button(lamb_form->auditionfx, 0);
    audition_state = 0;
    }
}


void xy_dominance_cb(FL_OBJECT *ob, long data)
{
    control_process(DOMINANCE_BASE, fl_get_positioner_xvalue(ob));
    control_process(DOMINANCE_BASE+1, fl_get_positioner_yvalue(ob));
}

void z_dominance_cb(FL_OBJECT *ob, long data)
{
    control_process(DOMINANCE_BASE+2, fl_get_slider_value(ob));
}

void dominance_reset_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(RESET_DOMINANCE, 1);

    fl_set_positioner_xvalue(lamb_form->xy_dominance,64);
    fl_set_positioner_yvalue(lamb_form->xy_dominance,64);
    fl_set_slider_value(lamb_form->z_dominance,64);
}


void xy_listener_cb(FL_OBJECT *ob, long data)
{
    control_process(LISTENER_BASE, fl_get_positioner_xvalue(ob));
    control_process(LISTENER_BASE+1, fl_get_positioner_yvalue(ob));
}

void z_listener_cb(FL_OBJECT *ob, long data)
{
    control_process(LISTENER_BASE+2, fl_get_slider_value(ob));
}

void listener_reset_cb(FL_OBJECT *ob, long data)
{
    slow_control_process(RESET_LISTENER, 1);

    fl_set_positioner_xvalue(lamb_form->xy_listener,64);
    fl_set_positioner_yvalue(lamb_form->xy_listener,64);
    fl_set_slider_value(lamb_form->z_listener,64);
}



void swap_page_cb(FL_OBJECT *ob, long data)
{
    static int page = 0;  /* Initially config page */
/*
 * Convenient test of buffer states.
 */
/*
printf("L658 %d %d\n", ALgetfilled(audio_in), ALgetfilled(audio_out));
*/
    page=1-page;

    if (page==0) {
    fl_hide_object(lamb_form->main_gp);
    fl_show_object(lamb_form->config_gp);
    }

    else {
    fl_hide_object(lamb_form->config_gp);
    fl_show_object(lamb_form->main_gp);
     }

    fl_check_forms();
}




void debug_cb(FL_OBJECT *ob, long data)
{
    int n, chars, x;

    printf("\n\n\nStart");
    for(n=0; n<DEBUG_SAMPLES; n++) {

    chars = (debug_wave[n] /512)+64;
    for(x=1; x<=128; x++) {
        printf( "%c", (x<=chars) ? '*' : '.' );
    }
    printf("\n");
    }
    printf("\nEnd\n\n");
}
```

Appendix F

SidRat Performance Instructions and Graphic Score

SidRat, for Solo Percussion and Live Electronics.

Dylan Menzies May '97

Equipment List

- *Percussion, varied* Must include something capable of a sharp percussive hit, such as a wood block. Other instrumentation directions in the score should be followed as closely as possible. These include snare, snare, bongos, deep tom, tabla.
- *Soundfield Microphone* This is the special Ambisonic microphone capable of recording B-format directly. If this cannot be obtained, interesting results can still be had by using a normal stereo microphone. A mic stand is needed for hanging the microphone vertically.
- *Footpedal and Footswitch* The footpedal must send continuous controller information, and the footswitch a 2-state controller. This can be achieved for example by using sustain and volume pedals connected to a MIDI keyboard, and using the MIDI output. *sidrat.orc* can be edited to correct for switch polarity and footpedal scaling.
- *Csound Machine* A computer running Csound with 4 audio input and output channels (2 if a stereo microphone is used), and a MIDI interface. *sidrat.orc* runs comfortably on an SGI Indy with an R4K processor, and so should present no problems for current desktops. The orc variable *icompdel* in *instr 40* should be set according to the instructions given in the orc.
- *Ambisonic Speaker Installation* This can take several forms. The simplest consists of only 4 speakers positioned at the corners of square, pointing in towards the audience. The speaker feeds are obtained by directly amplifying the Csound output. 6 speakers arranged in a hexagon can be driven from 6 networked amplifiers fed by the 4 Csound output channels, but this is only recommended if you are already familiar with this technique. Other speaker arrangements can be driven from an external hardware decoder fed with B-format from Csound. The *sidrat.sco* must be edited to select the correct output instrument. The default is B-format output.
- *Stopwatch Clock* The player uses this to sychronize with the computer. It should be easy to reset at the start of each movement.

Description

The central theme in this piece is the exploration of spatial and temporal relationships in sound. An important part of this process is the shift in control between the soloist and the computer representing fate. The imitation of clock sounds and their transformation is a recurring feature: At times sounds slow down and reverse. The Ambisonic speaker system provides a vivid way for portraying the complex synthesised soundfields.

The player sits in the middle of the concert room with the microphone positioned in the midst of the percussion, so that different percussion can be played at different directions relative to the microphone. To minimise feedback problems, the percussion should be used as close to the microphone as possible. The audience surrounds the player, within the speaker array.

The first movement begins with an opening recorded crescendo immediately followed by the first hit by the player. The hit is recorded and forms the basis of the remaining live electronic sound in the first movement. The electronic part develops gradually from a simple ticking to more erratic rhythms and then complex spatio-temporal entities. Meanwhile the player reacts to the electronics with defiant percussive gestures, although enable to change the course of events. Several skirmishes eventually lead to a more serene state in which the electronics generate a regular spiralling figure, like a cat purring. This is joined by a recorded section of voice based music with a strong spatial-dynamic component. The player has now blended with the electronics to produce light rhythms on the tabla. These fade to silence.

In the second movement the player has gained new powers of control. The footswitch controls the input of microphone sound into a spatialised delay-feedback network, while the footpedal controls the feedback. The player can also control the spatialisation of sound by playing percussion in the appropriate position. The speaker rig acts like a big sonic magnifying glass, blowing the arrangement of percussion upto a scale where the whole audience can hear whats going on. The player has some time to experiment with this new instrument before the electronics make themselves known by interfering with the delay time. The interference becomes gradually more severe, and the player adopts a more aggressive, primitive beat. This leads towards a climax followed immediately by a collapse of the previous sounds falling apart. A new climax begins to develop with a multilayered structure. The player talks and laughs into the microphone, and this becomes trapped into the layers sound. The second climax is not so loud as the first, but is more profound, signalling a new era. The final recorded sample plays out, a collage of alarm bells followed by clocks. The player continues the clock sound by building one up with the delay instrument, now unimpeded by external influence. The clock is held in a state of perfect repetition, as if frozen, and then faded away to nothing.

Performance Notes

The following notes are to be used by the performer in preparation for a concert. The graphical score sheets are intended as a guide during performance. Since the percussion is not precisely written, the performer may find it helpful to edit the score helpful indicators.

Electronics refers to sound generated by the loudspeakers. This includes processed live percussion and some recorded segments.

First Movement

0s	The player starts the stopwatch, and someone else starts sidrat on the computer.
9s	The opening hit must be precisely timed. It should follow as the natural conclusion of the opening recorded phrase. Try experimenting with different sounds to see how it effects the developing part.
31s	Using the muted snare and bongos, at first be *playful* with the electronic part, then become more aggressive. This requires practice in order to anticipate the changes of rhythm.
43s	It is important to stop here, as if suddenly silenced by the newly displayed virtuosity of the electronics.
50s	Press the blunt end of one stick against the tom skin while hitting lightly with the other stick, to create a unsteady, wavering pitch effect. Gradually increase intensity.
64s	Suddenly introduce the other kit percussion. Start with a loud hit and a build a regular rhythm that becomes more intense.
78s	Stop completely, and wait for the electronics to tail away.
100s	Build the intensity again, responding to irregular changes in the electronics rhythm. Use the kit freely especially cymbals.
121s	Stop playing as if too tired to overcome the electronics, which reaches the main climax of this movement.
140s	A new tranquil state has been reached. Introduce the tabla and bongos in sympathy with the regular electronic rhythm. Eventually settle to a steady rhythm. Fade as electronics fades.
240s	Reset the clock.

Second Movement

Feedback is controlled with the footpedal. Pedal down gives full feedback. Pedal up gives no feedback. *Input* is controlled with the footswitch. This is either on, when the pedal is down, or off when the pedal is up.

0s	Begin by making unmusical sounds - arhythmic scrapings, taps.
30s	Introduce feedback, and start building up sounds like the movements inside clocks. Remember that the microphone also records the direction of sound. Experiment with different spatial arrangements of percussion and also moving hand percussion as it is played. After reaching a good loop, fade it away into the background, but not to silence. Restore full feedback and build another clock layer in the foreground. Repeat this process until..
60s	Hold the rhythm frozen by closing the input while keeping the feedback full on.
75s	Kill the rhythm by quickly bringing the feedback to zero. Develop simple tom beat, gradually increasing loudness. The beats must be sparce so that the electronics can be clearly heard in between. Maintain the steady rhythm as the electronics becomes more bizarre. Keep the feedback low.
134s	Be ready to stop otherwise the impact of the immediately following electronic passage will be lost.
142s	The mood here is almost hysterical. The vocals should suggest someone speaking to themselves in an excited state. Develop the talking into sporadic bursts of laughter.
200s	Another clock building section. This time the mood is less experimental, more inevitable. A single layer is built up at medium volume, held and then released.
235	A good way to signal the end of the piece is to switch off the light for the score.

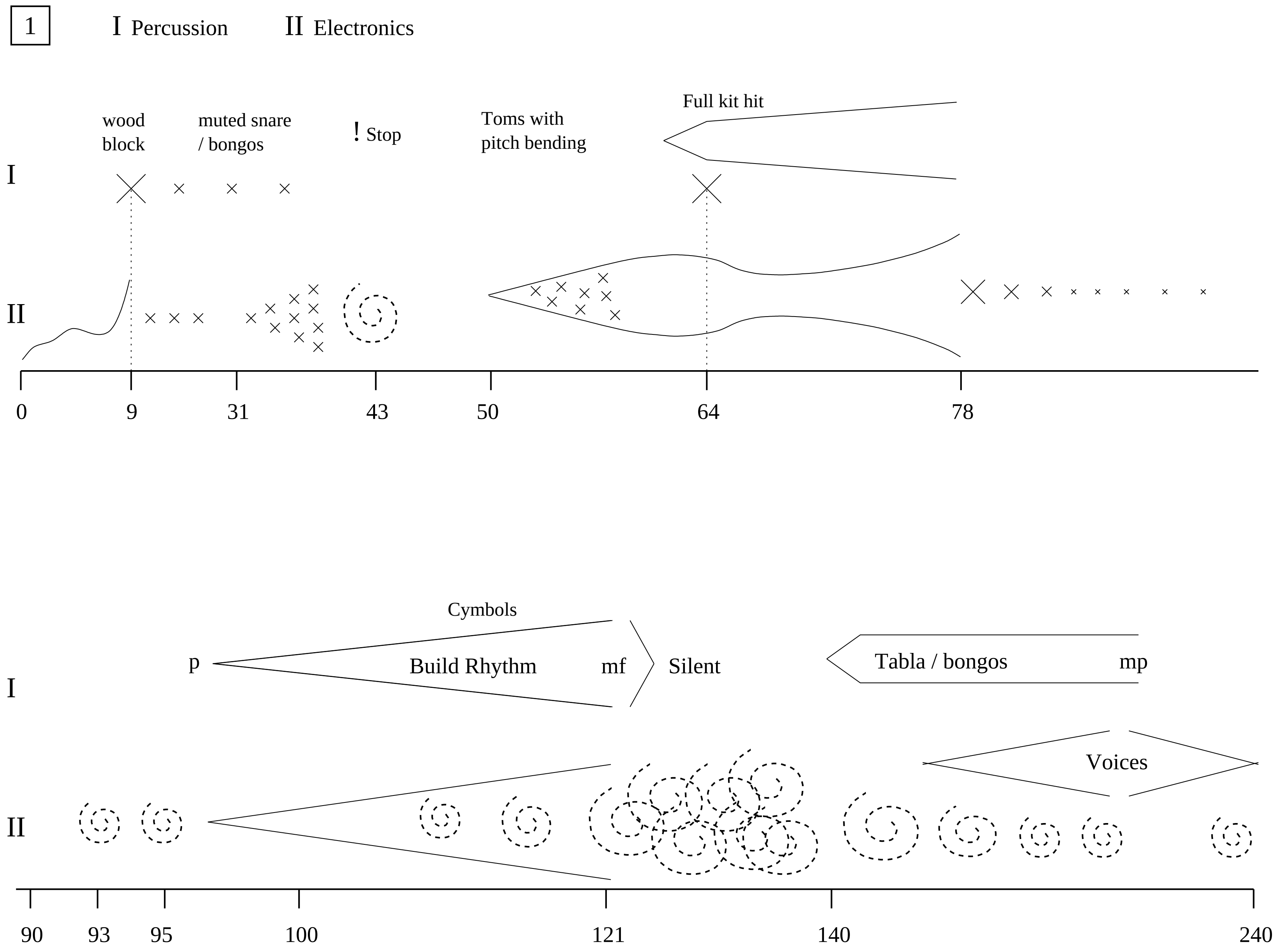


Figure F.1: SidRat Performance Score, First Movement

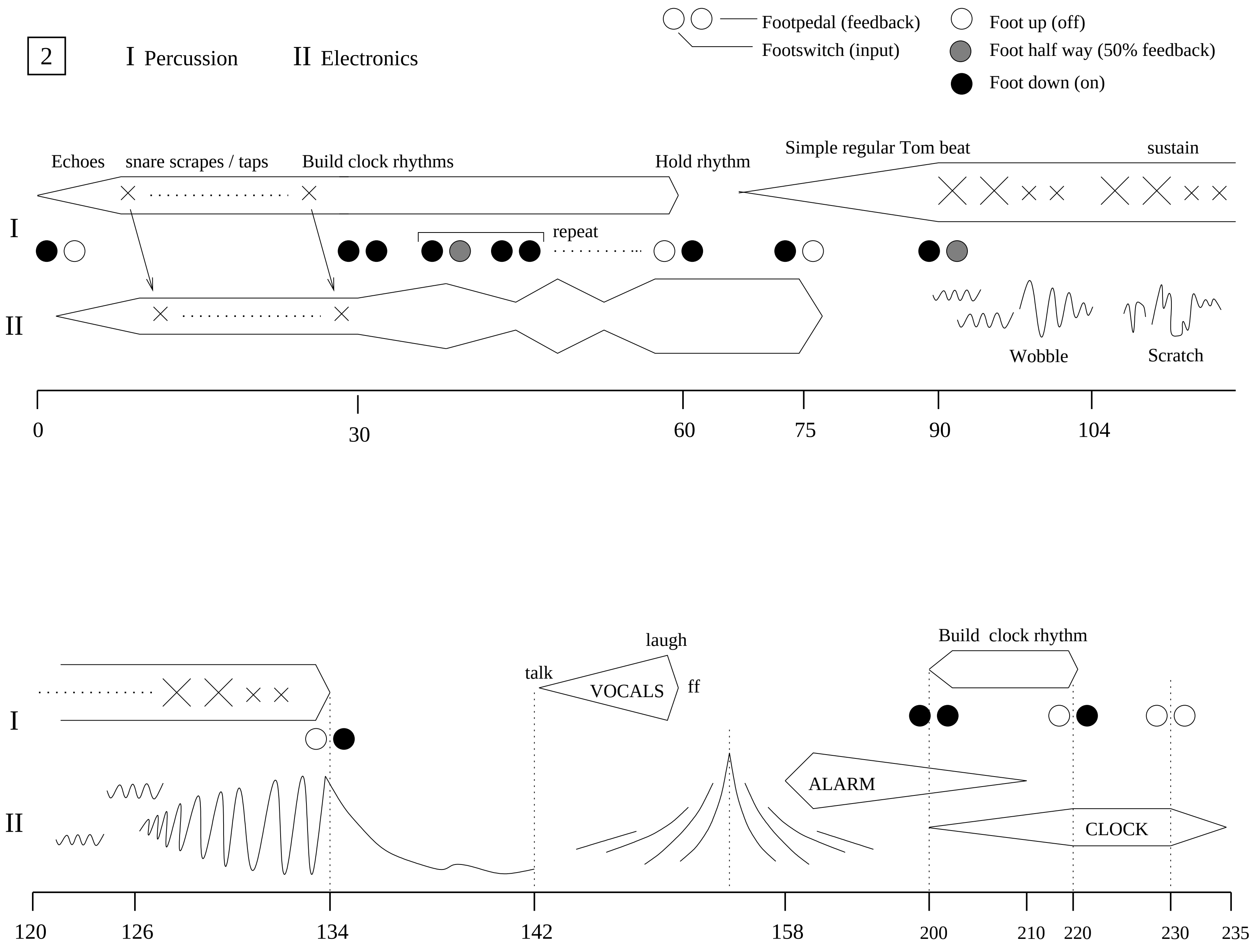


Figure F.2: SidRat Performance Score, Second Movement

Appendix G

SidRat Code Listing

Listings for the Csound script are given, followed by C code for a utility neccessary for handling MIDI input correctly.

G.1 sidrat.orc

```
;;
;; sidrate.orc
;;
;; Interactive performance piece using realtime Csound.
;; Works fine on a Silicon Graphics Indy.
;;
;; Customization
;;
;; 1. Choose the correct output format either by editing the instrument numbers ;; 100,101,102 or selecting the correct
;;
;; 2. Adjust gidelcomp to give the best performance of the feedback instrument
;; in the second movement. Switch on the input and feedback. Make one hit. Make
;; another hit exactly when the echo sounds. The next echo should consist of the
;; first two hits sounding together. The best setting should equal the total
;; audio buffer delay + delay from the speakers to the player @ 3ms per metre.


sr = 32000
kr = 320
ksmps = 100
nchnls = 4

gicompdel = .012 ; Audio buffers delay + delay from speakers to player.

girise  = .05
gidecay = .05
gimin   = .01   ; Min value for expseg envelopes.
gitmin  = .01   ; Smallest rise time for expseg envelopes
ga00    init 0
gkdelt  init 1  ; For the building instr (40)

gaw init 0  ; Pre-initialised workspace
gax init 0
gay init 0
gaz init 0

gkbeatf init 0  ; Private workspace for instr 25-28
gkbeatm init 0
gktap1m init 0
gktap2m init 0
gkfb    init 0

gknew   init 0  ; Instr 40
gkfdb   init 0
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;  Midi channel 1 input. Use with mdcontrol |
;;;;  to provide the initial note on.
    instr 1
gkc7    midictrl 0  ; For keyboard testing
gkc64   midictrl 2

;gkc7   midictrl 7  ; Pedal 1 from Korg M1.  Check This!
;;gkc7  = 127 - gkc7    ; Invert polarity to normally zero. Skip this if using a cts pedal
;gkc64  midictrl 64 ; Sustain pedal from Korg M1


;;; Midi control of feedback (if instr 29 not on)
;gkfb   = gkc64 * 0.007
;gkfb   port gkfb, .3

    endin


;;;;  Audio input
    instr 20
a1,a2,a3,a4 inq
ilfcut  = 30    ; Allows some compensation for room resonance..
ga1 atone a1, ilfcut    ; Remove DC
ga2 atone a2, ilfcut    ; Remove DC
ga3 atone a3, ilfcut    ; Remove DC
ga4 atone a4, ilfcut    ; Remove DC
    endin


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Intro sample
    instr 21
gaw,gax soundin "/usr/bach/rdmg101/SoundFiles/Sid/Final/intro"
gaw = gaw * p4
gax = gax * p4
;gaw,gax    soundin "/usr/electech2/rdmg101/Sid/intro"
    outq    gaw,gax,ga00,ga00
    endin

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Middle sample
    instr 22
;; Perhaps using globals will save glitches by not requiring mem allocs
gaw = gaw * p4
gax = gax * p4
gay = gay * p4
gaz = gaz * p4
gaw,gax,gay,gaz soundin "/usr/bach/rdmg101/SoundFiles/Sid/Final/middle"
    outq gaw,gax,gay,gaz
    endin

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; End sample
    instr 23
gaw = gaw * p4
gax = gax * p4
gay = gay * p4
gaz = gaz * p4
gaw,gax,gay,gaz soundin "/usr/bach/rdmg101/SoundFiles/Sid/Final/end32"
    outq gaw,gax,gay,gaz
    endin

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Drop in a tom sound for the beat - to help developement
    instr 24
;ga1,ga1    soundin "/usr/electech2/rdmg101/Sid/tom"
ga1,ga1 soundin "/usr/bach/rdmg101/SoundFiles/Sid/tom.32k"
    endin



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Change mix and feedback paramters for instr 30
    instr 25    ; Beat feed through to delay
gkbeatf expseg i(gkbeatf)+gimin, p3 , p4+gimin  ; Move on from previous value
gkbeatf = gkbeatf-gimin
    endin

    instr 26    ; Beat mix
gkbeatm expseg i(gkbeatm)+gimin, p3 , p4+gimin  ; Move on from previous value
gkbeatm = gkbeatm-gimin
    endin

    instr 27    ; Feedback 1, with rotate, and tap time.
gktap1m expseg i(gktap1m)+gimin, p3 , p4+gimin  ; Move on from previous value
gktap1m = gktap1m-gimin
gitap1t = p5
gic1    = cos(p6)
gis1    = sin(p6)
    endin

    instr 28    ; Feedback 2, with rotate, and tap time.
gktap2m expseg i(gktap2m)+gimin, p3 , p4+gimin  ; Move on from previous value
gktap2m = gktap2m-gimin
gitap2t = p5
    endin

    instr 29    ; Master tap feeback control
gkfb    expseg i(gkfb)+gimin, p3 , p4+gimin ; Move on from previous value
gkfb    = gkfb-gimin
    endin



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Multi tap bformat rhythm synth with rotated 'beat storage unit'
    instr 30
itpre   = p4    ; Wait before capture
itin    = p5    ; Capture time
iatten  = p6    ; Input attenuation
ispinf  = p7    ; Frequency of x-y rotation of beat.
a00 init 0

idelt   = 2 ; Max delay of delay line.
awtap1  init 0
axtap1  init 0
aytap1  init 0
awtap2  init 0
axtap2  init 0
aytap2  init 0
abeat   init 0

;;; Capture input at a specific time
agate   linseg  0, itpre, 0, girise, iatten, itin, iatten, gidecay, 0 , 1 , 0 ;NB ensure flat


;;; Syncronization test
;;a2    oscil   4000,500,1
;;a2    = a2 * agate
;;  outq a2,a2,a2,a2


;;;;;;  !!!!!! Choose the right channel here ( ga1 on the night. )
a1  = ga1 * agate

;For the purposes of score developement
;a1,a1  soundin "/usr/bach/rdmg101/SoundFiles/Sid/tom.32k"
; /usr/electech2/rdmg101/Sound/tom.out

;;; Beat storage
a1  = a1 + abeat
abeat   delay a1, 1  ;- ksmps/sr    ; For feedback accuracy compensate for block processing delay

;;  outq    abeat,abeat,abeat,abeat

as  oscili 1, ispinf, 1
ac  oscili 1, ispinf, 2

aw  = abeat
ax  = abeat * ac
```

```
ay  = abeat * as
az  = 0


;;; Nice ol' linear interpolation
abeatf  interp gkbeatf
abeatm  interp gkbeatm
atap1m  interp gktap1m
atap2m  interp gktap2m
afb interp gkfb


;Add feedback to main signal route.

afbw    =  atap1m*awtap1            + atap2m*awtap2
afbx    =  atap1m*(axtap1*gic1 - aytap1*gis1)   + atap2m*axtap2
afby    =  atap1m*(axtap1*gis1 + aytap1*gic1)   + atap2m*aytap2

;Output
;gaw    = gaw + aw * abeatm + afbw  ; 'beat mix'
;gax    = gax + ax * abeatm + afbx
;gay    = gay + ay * abeatm + afby

    outq aw * abeatm + afbw, ax * abeatm + afbx, ay * abeatm + afby, 0

aw  = aw * abeatf + afbw * afb
ax  = ax * abeatf + afbx * afb
ay  = ay * abeatf + afby * afb

;Multi tap Delay
adum    delayr idelt
awtap1  deltap gitap1t
awtap2  deltap gitap2t
;awtap3 deltap itap3t
    delayw aw

adum    delayr idelt
axtap1  deltap gitap1t
axtap2  deltap gitap2t
;axtap3 deltap itap3t
    delayw ax

adum    delayr idelt
aytap1  deltap gitap1t
aytap2  deltap gitap2t
;aytap3 deltap itap3t
    delayw ay
    endin



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Mix limit parameter seter for instr 40
    instr 38
gknew   line i(gknew), p3 ,p4/128   ; Don't make p3 < 1 (only control rate)
gkfdb   line i(gkfdb), p3 ,p5/128
    endin

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Delay time glide instrument for instr 40
    instr 39
;;  p4      p5      p6      p7      p8      p9
;;  New tap     wiggle freq const wiggle    wiggle rise t   wiggle decay t  port t
gkdelt  line i(gkdelt), p3, p4

kamp    linseg 0, p7+0.01, p6, p3-p7-p8-0.02, p6, 0.01+p8, 0        ; 0.01 ensures return to zero
koffset oscil kamp, p5, 1

gkdelt  = gkdelt + koffset
;gkdelt port    gkdelt, p9

    endin


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;; The foot control ambisonic loop builder instrument
    instr 40
ifb = 1
idelmix = .3
idirmix = 1
idelt   = 1
ideltmax    = p5
gkdelt  init p4
adelt   init p4
gknew   init p6/128
gkfdb   init p7/128
ifdbt   = p8

adelw   init 0
adelx   init 0
adely   init 0
adelz   init 0


kfb = gkc64 * gkfdb
kfb port kfb, ifdbt
afb interp kfb

knew    = gkc7 * gknew
knew    port knew, .1

display afb, 1

aw      = ga1 * anew + adelw * afb  ; NB adelw is old by idelt + ksmps/sr
ax      = ga2 * anew + adelx * afb  ; due to block processing.
ay      = ga3 * anew + adely * afb
az      = ga4 * anew + adelz * afb


adelt1  interp gkdelt
adelt   tone adelt1, 3      ; Produces smoother, more natural scratch. Limit of 3Hz though.
;; Could try this instead for lower filter freq cut off:
; adelt = adelt1 * .01 + adelt * .99


adum    delayr ideltmax
adelw   deltapi adelt
    delayw aw
adum    delayr ideltmax
adelx   deltapi adelt
    delayw ax
adum    delayr ideltmax
adely   deltapi adelt
    delayw ay
adum    delayr ideltmax
adelz   deltapi adelt
    delayw az


;   outq adelw, adelx, adely, adelz

gaw =  adelw
gax =  adelx
gay =  adely
gaz =  adelz


adelw delay adelw, gicompdel
adelx delay adelx, gicompdel
adely delay adely, gicompdel
adelz delay adelz, gicompdel

    endin



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Rotate processing
    instr 42
;; p4 = whole revolutions
;as oscili 1, p4/p3, 1  ; Very jerky rotate
```

```
;ac oscili 1, p4/p3, 2

;; Super delux smooth rotate
aindx   line 0, p3, .5
acv tablei aindx, 2, 1  ; cos

acv = (1-acv)*.5*p4
as  tablei acv, 1, 1, 0, 1
ac  tablei acv, 2, 1, 0, 1


ax  = ac * gax + as * gay
gay = -as * gax + ac * gay
gax     = ax
    endin



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Hex array output

    instr 100
ih1 = sqrt(3/2)
ih2 = 1/sqrt(2)
ih3 = sqrt(2)

gaw = gaw * 1.5 ; AMS -> hex

aw  = 2 *gaw
acb = gaw - ih3*gay
alf = gaw + ih1*gax + ih2*gay
arf = gaw - ih1*gax + ih2*gay

    outq aw, acb, alf, arf

    endin



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Bformat output

    instr 101
    outq gaw, gax, gay, gaz
    endin


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Quad array output

    instr 102

alf = gaw+gax+gay
arf = gaw-gax+gay
arb = gaw-gax-gay
alb = gaw+gax-gay

    outq    alf, arf, arb, alb
    endin
```

G.2 sidrat.sco

```
;;
;; sidrate.sco
;;
;; Interactive performance piece using realtime Csound.
;; Works fine on a Silicon Graphics Indy.
;;

f1  0   8192    10  1           ; Sin
f2  0   8192    9   1   1   90  ; Cos
```

```
i20     0   10  ; Audio input + regular resets.
i20     0   2   ; Audio input + regular resets.

;;; 1st section Delay process
;;;         predel  capture time    atten   spinfreq
;;i30   0   236 0.001   1       .2  .15
i30     0   236 8.9     1       .2  .15

;i24    0   1   ; Tom hit
;;i24   9.03    1   ; Tom hit

;a0 1   236 ; FF after beat capture.
i21 0.3 9   1   ; Cym intro soundfile (level)


;;; Mix and feedback control for instr 30
;;; 25-Beat feed, 26-Beat Mix, 27-Tap 1 mix, 28-Tap 2 mix, 29-Master feedback
;;; start   dur target  (tapt)  (rotate radians)

i25 9   1   1           ; Feed beat to delay
i26 9   1   1           ; Start steady beat
 i27    18  4   .5  .5  0   ;
i26 22  4   0           ; Beat decays
 i27    26  2   0   .   .   ; Tap1 decays
i26 28  2   1
 i27    31  4   2   .3  3.1 ; Echo at 180, rising to louder
  i28   35  3.5 1   .8      ; Double echo
  i28   38.8    .1  0   .       ; Quick change of time
  i28   38.9    .1  1   .7
  i28   41  1.5 0   .       ; Kill taps
 i27    43  .5  0
  i28   43.9    .1  0
 i27    43.9    .1  1   .1  .2  ; Quick 360 swizzle
   i29  43.9    .1  .9
i26 44.9    .1  0
i25 44.9    .1  0           ; Feed only 1s of beat

;;; Section 2

i26 49  1   0           ; beat off
i25 49  1   0           ; feed off
 i27    49  1   0
  i28   49  1   0
   i29  49  1   0           ; FB off

 i27    50  .1  .2  .7  1   ; Set tap 1
  i28   50  .1  1   .45     ; Set tap 2
   i29  50  .1  .5
i25 50  5   1           ; feed slowly rising
 i27    60  2   0   .   .
i25 63  2   0

 i27    64  .2  1   .8  1   ; Set tap 1 (smooth leading edge)
  i28   64  .1  1   1.1     ; Set tap 2
   i29  64  .1  1
i25 64  2   1           ; feed slowly rising

i25 68  2   0
 i27    68  1   0           ; Leave tap2 to trap the beat

  i28   78  .1  0
 i27    78.1    .2  .95 .3  -.2


;;; Section 3


i25 89  .1  0           ; Reset
i26 89  .1  0
 i27    89  .1  0
  i28   89  .1  0
   i29  89  .1  0

i25 89.9    .1  1
 i27    89.9    .1  1   .1  0   ; Quick 360 swizzle
   i29  89.9    .1  .9
```

```
i25 90.9    .1  0

i25 92.9    .1  1
 i27    92.9    .1  1   .15 -.5 ; 360 swizzle
   i29  92.9    .1  .9
i25 93.9    .1  0

i25 94.9    .1  .3
 i27    94.9    .3  1   .04 .1  ; Fast swizzle
   i29  94.9    .3  .9
i25 95.9    .1  0

i25 96.9    .1  1
 i27    96.9    .1  1   .15 -.5 ; Quick 360 swizzle
   i29  96.9    .1  .9
i25 97.9    .1  0

i25 98.9    .1  .3
 i27    98.9    .3  1   .04 .2  ; Quick 360 swizzle
   i29  98.9    .3  .9
i25 99.9    .1  0

i25 100.9   .1  1
 i27    100.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  100.9   .1  .9
i25 101.9   .1  0

i25 102.9   .1  .3
 i27    102.9   .3  1   .04 .4  ; Quick 360 swizzle
   i29  102.9   .3  .9
i25 103.9   .1  0

i25 104.9   .1  1
 i27    104.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  104.9   .1  .9
i25 105.9   .1  0

i25 106.9   .1  .3
 i27    106.9   .3  1   .02 0   ; Quick 360 swizzle
   i29  106.9   .3  .96
i25 107.9   .1  0

i25 108.9   .1  1
 i27    108.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  108.9   .1  .9
i25 109.9   .1  0

i25 110.9   .1  .3
 i27    110.9   .3  1   .015    -.5 ; Quick 360 swizzle
   i29  110.9   .3  .98
i25 111.9   .1  0

i25 112.9   .1  1
 i27    112.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  112.9   .1  .9
i25 113.9   .1  0

i25 114.9   .1  .3
 i27    114.9   .3  1   .05 .1  ; Quick 360 swizzle
   i29  114.9   .3  .9
i25 115.9   .1  0

i25 116.9   .1  1
 i27    116.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  116.9   .1  .9
i25 117.9   .1  0

i25 118.9   .1  .3
 i27    118.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  118.9   .3  .9
i25 119.9   .1  0

i26 120.9   .1  0
i25 120.9   .1  1
 i27    120.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  120.9   .1  .9
```

```
i25 121.9   .1  0

;i25    122.9   .1  .3
; i27   122.9   .3  1   .05 .1  ; Quick 360 swizzle
;   i29 122.9   .3  .9
;i25    123.9   .1  0

;;; Interrupt in section 3

i25 122.9   .1  0           ; Reset
i26 122.9   .1  0
 i27    122.9   .1  0
  i28   122.9   .1  1   1.8
   i29  122.9   .1  1

  i28   125 .1  0
  i28   125.1   .1  1.5 .7
  i28   127 .3  0
  i28   127.4   .1  1.1 1.5

 i27    129 1   1.5 1   2   ; Mighty rising fb
 i27    133 .2  0
 i28    133 .1  0
 i27    133.2   .1  1   .6  -1

i25 140.9   .1  1
 i27    140.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  140.9   .1  .9
i25 140.9   .1  0

i25 142.9   .1  .3
 i27    142.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  142.9   .3  .9
i25 143.9   .1  0

i25 144.9   .3  1
 i27    144.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  144.9   .1  .9
i25 145.9   .1  0

i25 146.9   .1  .3
 i27    146.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  146.9   .3  .9
i25 147.9   .1  0

i25 148.9   .1  1
 i27    148.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  148.9   .1  .9
i25 149.9   .1  0

i25 150.9   .1  .3
 i27    150.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  150.9   .3  .9
i25 151.9   .1  0

i25 152.9   .1  1
 i27    152.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  152.9   .1  .9
i25 153.9   .1  0

i25 154.9   .1  .3
 i27    154.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  154.9   .3  .9
i25 155.9   .1  0


;;;; Middle Section, 4

i22 158 77.5    1           ; Middle sample

i25 156.9   .1  1
 i27    156.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  156.9   .1  .9
i25 157.9   .1  0

i25 158.9   .1  .3
```

```
 i27    158.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  158.9   .3  .9
i25 159.9   .1  0

i25 160.9   .1  1
 i27    160.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  160.9   .1  .9
i25 161.9   .1  0

i25 162.9   .1  .3
 i27    162.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  162.9   .3  .9
i25 163.9   .1  0

i25 164.9   .1  1
 i27    164.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  164.9   .1  .9
i25 165.9   .1  0

i25 166.9   .1  .3
 i27    166.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  166.9   .3  .9
i25 167.9   .1  0

i25 168.9   .1  1
 i27    168.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  168.9   .1  .9
i25 169.9   .1  0

i25 170.9   .1  .3
 i27    170.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  170.9   .3  .9
i25 171.9   .1  0

;;
i25 172.9   .1  1
 i27    172.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  172.9   .1  .9
i25 173.9   .1  0

i25 174.9   .1  .3
 i27    174.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  177.9   .3  .9
i25 175.9   .1  0

i25 176.9   .1  1
 i27    176.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  176.9   .1  .9
i25 177.9   .1  0

i25 178.9   .1  .3
 i27    178.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  178.9   .3  .9
i25 179.9   .1  0

;;
i25 180.9   .1  1
 i27    180.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  180.9   .1  .9
i25 181.9   .1  0

i25 182.9   .1  .3
 i27    182.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  182.9   .3  .9
i25 183.9   .1  0

i25 184.9   .1  1
 i27    184.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  184.9   .1  .9
i25 185.9   .1  0

i25 186.9   .1  .3
 i27    186.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  186.9   .3  .9
i25 187.9   .1  0
;;
```

```
i25 188.9   .1  1
 i27    188.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  188.9   .1  .9
i25 189.9   .1  0

i25 190.9   .1  .3
 i27    190.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  190.9   .3  .9
i25 191.9   .1  0

i25 192.9   .1  1
 i27    192.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  192.9   .1  .9
i25 193.9   .1  0

i25 194.9   .1  .3
 i27    194.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  194.9   .3  .9
i25 195.9   .1  0


;;
i25 196.9   .1  1
 i27    196.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  196.9   .1  .9
i25 197.9   .1  0

i25 198.9   .1  .3
 i27    198.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  198.9   .3  .9
i25 199.9   .1  0

i25 200.9   .1  1
 i27    200.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  200.9   .1  .9
i25 201.9   .1  0

i25 202.9   .1  .3
 i27    202.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  202.9   .3  .9
i25 203.9   .1  0

;;
i25 204.9   .1  .8
 i27    204.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  204.9   .1  .9
i25 205.9   .1  0

i25 206.9   .1  .3
 i27    206.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  206.9   .3  .9
i25 207.9   .1  0

i25 208.9   .1  .6
 i27    208.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  208.9   .1  .9
i25 209.9   .1  0

i25 210.9   .1  .3
 i27    210.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  210.9   .3  .9
i25 211.9   .1  0

;;
i25 212.9   .1  .5
 i27    212.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  212.9   .1  .9
i25 213.9   .1  0

i25 214.9   .1  .3
 i27    214.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  214.9   .3  .9
i25 215.9   .1  0

i25 216.9   .1  .3
 i27    216.9   .1  1   .15 -.5 ; Quick 360 swizzle
```

```
   i29  216.9   .1  .9
i25 217.9   .1  0

i25 218.9   .1  .2
 i27    218.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  218.9   .3  .9
i25 219.9   .1  0

;;
i25 220.9   .1  .2
 i27    220.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  220.9   .1  .9
i25 221.9   .1  0

i25 222.9   .1  .1
 i27    222.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  222.9   .3  .9
i25 223.9   .1  0

i25 224.9   .1  .2
 i27    224.9   .1  1   .15 -.5 ; Quick 360 swizzle
   i29  224.9   .1  .9
i25 225.9   .1  0

i25 226.9   .1  .1
 i27    226.9   .3  1   .04 -.5 ; Quick 360 swizzle
   i29  226.9   .3  .9
i25 227.9   .1  0

;i25    228.9   .1  .2
; i27   228.9   .1  1   .15 -.5 ; Quick 360 swizzle
;   i29 228.9   .1  .9
;i25    229.9   .1  0

;i25    230.9   .1  .1
; i27   230.9   .3  1   .04 -.5 ; Quick 360 swizzle
;   i29 230.9   .3  .9
;i25    231.9   .1  0

i25 228.9   .1  .2
 i27    228.9   .1  1   .20 -.5 ; Quick 360 swizzle
   i29  228.9   .1  .9
i25 229.9   .1  0

i25 230.9   .1  .1
 i27    230.9   .3  1   .9  -.5 ; Quick 360 swizzle
   i29  230.9   .3  .52
i25 231.9   .1  0

   i29  235 .5  0




;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; section 5. Start new score section. 4 minutes elapsed.
s
;a  0   0   34
f1  0   8192    10  1           ; Sin
f2  0   8192    9   1   1   90  ; Cos

i20     0   240 ; Audio input + filtering.
i101    0   240 ; Bformat output

;; Start loop builder with delay time 1s
;;      dur initial del max del     gknew   gkfdb   ifdbt
i40 0   240 1       8.1     1   1   1




;; Sf rotate
;;      dur Whole rots
i42 40  20  4       ; 1st slow rot
```

```
;; Into heavier section at 60s

;; Wobbles come in at 90s:


;;;;;; Changing delay time linearly
;; Gentle Wobbles leading to an assault course
;;      dur target  w-freq  w   w-rise  w-decay Port time
i39 90  2   1   3.3 .005    1   1
i39 +   2   1   0   0
i39 +   3   1   30  .015    1.5 1.4
i39 +   2   1   0   0   0   0
i39 +   3.5 1   2   .02 1   1


;; Rap
i39 104 .2  1   8   .3  0
i39 +   .8  1   0               ; exact return.
i39 +   1   1   0               ; Blank
i39 +   2   1   4   .2  0   1.5 ; fast
i39 +   1   1   0
i39 +   .1  .3  0
i39 +   .9  1   0
i39 +   2   1   0   0   0
i39 +   .1  .3  0
i39 +   1.4 1   0
i39 +   2   1   6   .2  0   1.5 ; fast
i39 +   .2  1   0   0   0   0
i39 +   .8  1   0   0   0   0


;; Wild wobble
i39 120 2   1   0   0   0   0
i39 +   2   1   10  .01 1.7 0
i39 +   1   1   33  .08 0   .9
i39 +   1   1   0   0   0   0
i39 +   8   1   7.2 .2  7.8 0   ; cresc
i42 126 8   8   ; Rotate - player can use this to sync


;; Loony toons

i39 134 4   8   0   0   0   0    ; Collapse transpose down
i39 +   .1  7   0   0   0   0
i39 +   1.9 8   0   0   0   0


;;          gknew   gkfdb
i38 134 1   1   .95 ; Protect from overloading in the busy bit


i39 +   16  0   0   0   0   0   0
i39 +   32  1   ; Long climb back
i39 +   2   1


;;          gknew   gkfdb
i38 156 1   1   1   ; Restore


;; Final pink bells         ; Play odd hits thru bells, hold
;;          level
i23 158 41  8       ; 44.1k file played at 32k 41s
```

G.3 mdcontrol.c

```
/*
 * mdcontrol.c
 *
 * Tool for priming Csound for receiving MIDI controllers,
 * by first sending a note message.
 *
 * Usage:  mdcontrol | csound ....
 *
 */

#include <dmedia/midi.h>
#include <stdlib.h>
#include <stdio.h>


void main(void)
{
  int i;
  char* byte;
  MDport MIDIport;
  MDevent *evbuf = calloc(1, sizeof(MDevent) );

  fprintf(stderr, "\nTo open a midi device use startmidi -d /dev/ttyd2\n");

  printf("Valid ports = %d\n",mdInit());
    if ((MIDIport = mdOpenInPort(NULL)) == NULL)
    printf("mdOpenInPort error\n");
  mdSetStampMode(MIDIport, MD_NOSTAMP);

  setbuf(stdout, NULL);

/*
 * Send note on message to start csound going.
 */

  putchar(0x90);
  putchar(0x00);
  putchar(0x40);


  while(1)
  {
    mdReceive(MIDIport, evbuf, 1);

    byte = evbuf->msg;
    for(i=0; i < evbuf->msglen; i++) putchar(*(byte++));

    /*
    printf("%x %x %x %x   %d\n", evbuf->msg[0], evbuf->msg[1],
                evbuf->msg[2], evbuf->msg[3], evbuf->msglen);
    */
  }


}
```

Appendix H

CyberWhistle Software Synthesis Code Listing

This appendix contains functions written in C++ for the Silicon Graphics Indy computer to perform software synthesis.

H.1 Makefile

```
#
# Makefile
#
# -for building CyberWhistle instruments.
#
# Extensive inline code means the header dependencies are used.
#


EXEC = run
HOME = /usr/bach/rdmg101/Home

# Optional targets depending on what run is being used for:
# CWtest.o CWclariHol.o  CWfluHolP.o
# CWtestSine.o CWclariHolP.o CWfluHol.o

O_FILES = CWmassBow.o \
    AudioIn.o AudioOut.o MidiIn.o fixFPU.o \
    AsyncFilt.o \
    Filter.o Tone.o Atone.o OnePole.o OneZero.o  DCBlock.o\
    DLineL.o DLineN.o Delay.o DelMix.o Waveguide.o WhistleBore.o \
    WhistleBoreP.o Barrier.o FluteMouth.o \
    WaveMass.o MassBow.o \
    OscWG.o Noise.o \
    CWinstr.o \
    OscTest.o \
    ReedTabl.o JetTabl.o

HEADERS = global.h priority.h AudioIn.h AudioOut.h MidiIn.h fixFPU.h \
    AsyncFilt.h \
    Filter.h Tone.h Atone.h OnePole.h OneZero.h DCBlock.h\
    DLineL.h DLineN.h Delay.h DelMix.h Waveguide.h WhistleBore.h\
    WhistleBoreP.h Barrier.h FluteMouth.h\
    WaveMass.h MassBow.h WaveBow.h \
    OscWG.h Noise.h \
    CWinstr.h CWtest.h CWclarinet.h CWclariHol.h CWclariHolP.h \
    CWfluHol.h CWmassBow.h CWtestSine.h \
    OscTest.h ReedTabl.h JetTabl.h

CC = CC -O2 -mips2

default: $(EXEC)
```

```
$(EXEC):    $(EXEC).c $(HEADERS)  $(O_FILES)
    $(CC)  -o $(EXEC) $(EXEC).c  $(O_FILES) -I$(HOME)/include -L$(HOME)/lib \
                    -lAM -lmd -ldmedia -laudio -lm

# To detect underflows:
# add  -lfpe to compile line above
# Type at prompt: setenv TRAP_FPE "DEBUG ; ALL=COUNT(1)"
# Recompile.


Tone.o:        Tone.h
Atone.o:    Atone.h
Delay.o:    Delay.h
DelMix.o:    DelMix.h
Waveguide.o:    Waveguide.h Delay.h
WhistleBore.o    : WhistleBore.h Waveguide.h Delay.h Barrier.h
CWinstr.o:    CWinstr.h MidiIn.h AsyncFilt.h Atone.h Tone.h
CWtest.o:    CWtest.h CWinstr.h
CWclariHol.o    : CWclariHol.h CWinstr.h ReedTabl.h OneZero.h \
                    Noise.h Delay.h Tone.h WhistleBore.h
CWclariHolP.o    : CWclariHolP.h CWinstr.h ReedTabl.h OneZero.h \
                    Noise.h Delay.h Tone.h WhistleBoreP.h
CWfluHol.o        : CWfluHol.h CWinstr.h JetTabl.h OnePole.h \
                    Noise.h Delay.h DLineL.h WhistleBore.h \
                    FluteMouth.h
CWmassBow.o        : CWmassBow.h CWinstr.h MassBow.h WhistleBoreP.h \
                    Delay.h OnePole.h Tone.h

MassBow.h:    WaveMass.h
OscTest.o:    OscTest.h OscWG.h
JetTabl.o:    JetTabl.h
DCBlock.o:    DCBlock.h
FluteMouth.o:    FluteMouth.h

FluteMouth.h:    JetTabl.h DLineL.h OnePole.h DCBlock.h Noise.h
Delay.h:    Filter.h
Atone.h:    Filter.h
Tone.h:        Filter.h
WhistleBore.h:    Delay.h Barrier.h
WhistleBoreP.h:    Delay.h

fixFPU.o:    fixFPU.h    fixFPU.c
    cc -O2 -mips2 -c fixFPU.c




clean:
    rcsclean
    rm -f *.o
```

H.2 Operating System

The following functions provide the environment within which the CyberWhistle instruments operate.

H.2.1 global.h

```
/*
 * global.h
 *
 * Global definitions.
 *
 */
```

```
#define FLOAT double
#define FLOAT_SIZE sizeof(FLOAT)
#define INLINE inline
#define AUDIO_RATE 22050 //32000
#define PI 3.1415927

#include <stdio.h>
#include <stdlib.h>
```

H.2.2 priority.h

```
#include <ulocks.h>
#include <sys/types.h>
#include <sys/schedctl.h>
#include <stropts.h>
#include <poll.h>
#include <limits.h>
#include <fcntl.h>
```

H.2.3 run.c

```
/*
 * run.c
 * Launchpad for CyberWhistle instruments.
 * Allows audio hardware and MIDI input configuration.
 *
 */

#include <stdio.h>

#include "fixFPU.h"
#include "global.h"
#include "priority.h"

#include "AudioOut.h"
#include "AudioIn.h"
#include "MidiIn.h"



/*
 * Choice of CyberWhistle instruments
 *
#include "CWtest.h"
#include "CWclariHol.h"
#include "CWclariHolP.h"
#include "CWtestSine.h"
#include "CWfluHol.h"
*/

#include "CWmassBow.h"

#define AUDIO_PRIORITY NDPHIMIN        // Omit to use normal priority.

// Create 1 instance of a CyberWhistle instrument.

CWmassBow    instr;


// processMidi is needed as an argument for sproc.
// For calibration, use instr.processMidiC instead of instr.processMidi

void processMidi(MIDI_EVENT* buf){ instr.processMidi(buf); };



void
main(int argc, char **argv)
{
```

```
//    long i=0;
    long pvbuf[16];


printf("At start\n");

//    Direct audio hardware settings.
//
    pvbuf[0] = AL_INPUT_RATE;
    pvbuf[1] = AUDIO_RATE;

    pvbuf[2] = AL_OUTPUT_RATE;
    pvbuf[3] = AL_RATE_INPUTRATE;

    pvbuf[4] = AL_CHANNEL_MODE;
    pvbuf[5] = AL_STEREO;

    pvbuf[6] = AL_INPUT_SOURCE;
    pvbuf[7] = AL_INPUT_MIC;

    pvbuf[8] = AL_LEFT_INPUT_ATTEN;    // Important not to distort mic input.
    pvbuf[9] = 255;

    pvbuf[10] = AL_RIGHT_INPUT_ATTEN;
    pvbuf[11] = 255;

    pvbuf[12] = AL_LEFT_SPEAKER_GAIN;    // Mid gain.
    pvbuf[13] = 20;

    pvbuf[14] = AL_RIGHT_SPEAKER_GAIN;
    pvbuf[15] = 20;

    ALsetparams(AL_DEFAULT_DEVICE, pvbuf, 16);

//    Initialise audio input, and software settings.
//    (A frame is a collection of samples, one for each channel.)
//  Make blockFrames small to reduce control latency,
//  but keep fillPointFrames big enough to provide adequate buffering
//  SIMPLE_BUF causes the buffer to be set to the minimum hardware size.
//  Smaller sizes can be used at the additional cost of regular buffer purging.

printf("Init audioIn..\n");
//                           blockFrames
//                   softChans        bufFrames
//                                   or SIMPLE_BUF
//
    AudioIn audioIn(4,        100,        SIMPLE_BUF);


printf("Init audioOut..\n");
//                           blockFrames
//                   softChans        bufFrames
//                                   or SIMPLE_BUF
//
    AudioOut audioOut(4,    100,        SIMPLE_BUF);


//    Setup the MIDI handling process. Either using a thread (MIDI_FORK)
// or by regular checking from the main audio loop (checkPeriod).
//
//               <Checkperiod>
//               or MIDI_FORK
//

//    MidiIn midiIn(MIDI_FORK,    processMidi);
    MidiIn midiIn(200,    processMidi);



#ifdef AUDIO_PRIORITY
    if ( prctl(PR_RESIDENT) < 0)
        fprintf(stderr, "Unable to insure that parent process is memory-resident.\n");
    else fprintf(stderr, "Parent process is memory resident.\n");

    if (schedctl(NDPRI, 0, AUDIO_PRIORITY) < 0)
        fprintf(stderr, "Audio process failed to use non-degrading priority.\n");
    else fprintf(stderr, "Audio process now using non-degrading priority.\n");
#endif

    fixFPU();  // Eliminates problem with SGI floating point chip.

    printf("\n\nStarting audio loop ..\n\n");

    while(1) {
        FLOAT chan1, chan2, chan3, chan4;


            audioIn.newFrame(&chan1, &chan2, &chan3, &chan4);

            chan1 = instr.newSamp(chan1);
//            chan1 = instr.newSamp();

//            audioOut.newFrame(chan1, chan1, chan1, chan1);
            audioOut.newFrame(instr.lastOutL(), instr.lastOutR(), 0, 0);

            midiIn.regCheck();        // Occasional read in. Not for MIDI_FORK


//    ..For checking the buffer behaviour.
/*
        {
        static int i = 0;

        if (i++ == 100000) {
            int in, out;
            i=0;
            printf("in %d    out %d    sum %d\n\n",
                in =ALgetfilled(audioIn.getPort()),
                out =ALgetfilled(audioOut.getPort()),  in+out );
            }
        }
*/
    }


}

/*----------------------------------------------------------------------
```

H.2.4 fixFPU.h

```
/*
 * fixFPU.h
 *
 * Prevent R4000 calling exception on underflow.
 * This needs to be compiled in C not C++ because of the header.
 *
 */

#ifdef __cplusplus
extern "C" {
#endif

#include <sys/fpu.h>
void fixFPU(void);

#ifdef __cplusplus
}
#endif
```

H.2.5 fixFPU.c

```
/*
 * fixFPU.c
 *
 * Prevent R4000 calling exception on underflow.
 * This needs to be compiled in C not C++ because of the header.
 *
 */

#include "fixFPU.h"


/*****  Fix R4000 FPU exception ‘feature’ *****/
void fixFPU()
{
    /* Works on R4K and above */
    union fpc_csr f;
    f.fc_word = get_fpc_csr();
    f.fc_struct.flush = 1;
    set_fpc_csr(f.fc_word);
}
```

H.2.6 AudioIn.h

```
/*
 * AudioIn.h
 *
 * SGI audio input class, only to be called once per process.
 *
 */

#include <dmedia/audio.h>
#include <stdlib.h>
#include "global.h"
#include <dmedia/audio.h>
#define SAMP_TYPE short
#ifndef SIMPLE_BUF
#define SIMPLE_BUF 0;
#endif

class AudioIn
{
    protected:
        int softChans;
        int blockFrames;
        int bufFrames;
        int    blockSamps;
        int bufSamps;
        int queueSamps;
        int portfd;
        ALport port;
        SAMP_TYPE* block;
        int     blockCounter;
        void readBlock(void);

    public:
        AudioIn(int, int, int);
        ~AudioIn();
        FLOAT newSamp(void);
        void newFrame(FLOAT*);
        void newFrame(FLOAT*, FLOAT*);
        void newFrame(FLOAT*, FLOAT*, FLOAT*, FLOAT*);
        ALport getPort() {return(port);};
};



INLINE void AudioIn :: newFrame(FLOAT*samp1, FLOAT*samp2, FLOAT*samp3, FLOAT*samp4)
{
    if (blockCounter >= blockSamps) {
        blockCounter = 0;

        readBlock();
    }

*samp1 = block[blockCounter];
*samp2 = block[blockCounter+1];
*samp3 = block[blockCounter+2];
*samp4 = block[blockCounter+3];

blockCounter += softChans;
}

INLINE void AudioIn :: newFrame(FLOAT*samp1, FLOAT*samp2)
{
    if (blockCounter >= blockSamps) {
        blockCounter = 0;

        readBlock();
    }

*samp1 = block[blockCounter];
*samp2 = block[blockCounter+1];

blockCounter += softChans;

}

INLINE void AudioIn :: newFrame(FLOAT*samp1)
{
    if (blockCounter >= blockSamps) {
        blockCounter = 0;

        readBlock();
    }

*samp1 = block[blockCounter];

blockCounter += softChans;
}



// Advance and return sample. Use <softChans> times in a group.
// Use this carefully! Its not fool proof, but is convenient.


INLINE FLOAT AudioIn :: newSamp()
{
    if (blockCounter >= blockSamps) {
        blockCounter = 0;

        readBlock();
    }

    return(block[blockCounter++]);
}
```

H.2.7 AudioIn.c

```
/*
 * AudioIn.c
 *
 * SGI audio input class, only to be called once per process.
 *
 */

#include <stdio.h>
#include "AudioIn.h"

AudioIn :: AudioIn(int d1, int d2, int d3)
{
    ALconfig config;

    softChans = d1;
    blockFrames = d2;
    bufFrames = d3;

    blockSamps = blockFrames * softChans;
    bufSamps = bufFrames * softChans;
    queueSamps = blockSamps + bufSamps;
    blockCounter = 0;
// Impose ALlib limits on queue size.
    switch(softChans){
        case 1 : if (queueSamps < 510) queueSamps = 510; break;
        case 2 : // see 4
        case 4 : if (queueSamps < 1020) queueSamps = 1020; break;
        default : fprintf(stderr, "AudioIn: Bad number of channels.\n"); exit(-1);
    }

    block = (SAMP_TYPE*)calloc(queueSamps, sizeof(SAMP_TYPE));    // Allow for queue clearance.


    config = ALnewconfig();

    if(ALsetchannels(config, softChans) < 0)
        exit(-1);

    if(ALsetqueuesize(config, queueSamps) < 0)
        exit(-1);

    if(ALsetwidth(config, AL_SAMPLE_16) < 0)
        exit(-1);

    port = ALopenport("audioInPort", "r", config);
    if(!port) exit(-1);

    portfd = ALgetfd(port);

    ALfreeconfig(config);

}



AudioIn :: ~AudioIn()
{
    if (ALcloseport(port) < 0) exit(-1);
}



void AudioIn :: readBlock()
{
    int excessSamps;



// Discard excess input caused by glitching, when it exceeds limit
if (bufFrames>0) {
    excessSamps = ALgetfilled(port) - blockSamps;
    if ( excessSamps > bufSamps)
    {    ALreadsamps(port, block, (excessSamps)); // & 0xffffffc
//printf("%d %d\n", excessSamps,  bufSamps);
    }
}
    ALreadsamps(port, block, blockSamps);

}
```

H.2.8 AudioOut.h

```
/*
 * AudioOut.h
 *
 * SGI audio output class. (only to be called once per program.)
 */

#include <stdlib.h>
#include "global.h"
#include <dmedia/audio.h>
#include <sys/time.h>  /* timeval for select() */
#define SAMP_TYPE short
#define SIMPLE_BUF 0
```

```
class AudioOut
{
    protected:
        int softChans;
        int blockFrames;
        int    blockSamps;
        int bufFrames;
        int bufSamps;
        int fillPointSamps;
        int queueSamps;
        int simpleBuf;
        int portfd;
        fd_set writefds;
        ALport port;
        SAMP_TYPE* block;
        int     blockCounter;
        void writeBlock(void);


    public:
        AudioOut(int, int, int);
        ~AudioOut();
        void newFrame(FLOAT);
        void newFrame(FLOAT, FLOAT);
        void newFrame(FLOAT, FLOAT, FLOAT, FLOAT);
        ALport getPort() { return(port); };
};



// The following all work with any softChan setting.

INLINE void AudioOut :: newFrame(FLOAT samp)
{
    if (samp > 32767) samp = 32767;            // Maybe better to do this
    else if (samp < -32768) samp = -32768;  // as a block op.

    block[blockCounter] = (SAMP_TYPE)samp;

    blockCounter+= softChans;

    if (blockCounter == blockSamps) {
        blockCounter = 0;
        writeBlock();
    }
}


INLINE void AudioOut :: newFrame(FLOAT samp1, FLOAT samp2)
{
    block[blockCounter] = (SAMP_TYPE)samp1;
    block[blockCounter+1] = (SAMP_TYPE)samp2;

    blockCounter+= softChans;

    if (blockCounter == blockSamps) {
        blockCounter = 0;
        writeBlock();
    }
}


INLINE void AudioOut :: newFrame(FLOAT samp1, FLOAT samp2, FLOAT samp3, FLOAT samp4)
{
    block[blockCounter] = (SAMP_TYPE)samp1;
    block[blockCounter+1] = (SAMP_TYPE)samp2;
    block[blockCounter+2] = (SAMP_TYPE)samp3;
    block[blockCounter+3] = (SAMP_TYPE)samp4;

    blockCounter+= softChans;

    if (blockCounter == blockSamps) {
        blockCounter = 0;
        writeBlock();
    }
}
```

```
// Old style..
/*
INLINE void AudioOut :: newFrame(SAMP_TYPE* frame)
{
    block[blockCounter] = frame[0];
    block[blockCounter+1] = frame[1];    //
    block[blockCounter+2] = frame[2];    // Could be redundant.
    block[blockCounter+3] = frame[3];    //

    blockCounter+= softChans;

    if (blockCounter == blockSamps) {
        blockCounter = 0;
        writeBlock();
    }
}
*/
```

H.2.9 AudioOut.c

```
/*
 * AudioOut.c
 *
 * SGI audio output functions in C. (only to be called once per program.)
 *
 */

//// Todo : make softChans select a suitable function for audio output.

#include <stdio.h>
#include "AudioOut.h"

AudioOut :: AudioOut(int d1, int d2, int d3)
{
ALconfig config;

softChans = d1;
blockFrames = d2;
bufFrames = d3;
simpleBuf = 1; // Don't use select.

blockSamps = blockFrames * softChans;
bufSamps = bufFrames * softChans;
blockCounter = 0;
queueSamps = bufSamps+blockSamps; // Maximum queue size required.

switch(softChans){
case 1 : if (queueSamps < 510) { queueSamps = 510; simpleBuf = 0; } break;
case 2 : // see 4
case 4 : if (queueSamps < 1020) { queueSamps = 1020; simpleBuf = 0; } break;
default : fprintf(stderr, "AudioOut: Bad number of channels.\n"); exit(-1);
}

block = (SAMP_TYPE*)calloc(queueSamps+3, sizeof(SAMP_TYPE));
// +3 allows for 4 samp mode when softChans = 1

if (bufFrames == SIMPLE_BUF) simpleBuf = 1;

fillPointSamps = queueSamps - bufSamps;
// New block can be transfered when
// out-queue has shrunk below bufSamps.


if(ALsetchannels(config, softChans) < 0)
exit(-1);

if(ALsetqueuesize(config, queueSamps) < 0)
exit(-1);

if(ALsetwidth(config, AL_SAMPLE_16) < 0)
exit(-1);
port = ALopenport("audioOutPort", "w", config);
if(!port) exit(-1);

portfd = ALgetfd(port);
    FD_ZERO(&writefds);

ALfreeconfig(config);



/*
 * Fill output queue uoto fill point,
 * to eliminate initial glitches.
 * Block is zeroed initially and has more than buf elements.
 */
ALwritesamps(port, block, bufSamps + blockSamps);

/*
 * The output port should now contain a queue of length that gives
 * maximum normal running latency.
 */

}



AudioOut :: ˜AudioOut()
{
if (ALcloseport(port) < 0) exit(-1);
}


void AudioOut :: writeBlock()
{

// Mainly for use when no audio in buffer is used, to reduce buf size.
// The overhead of select may be counterproductive.
if (!simpleBuf) {
if (ALgetfilled(port)> bufSamps) {
ALsetfillpoint(port, fillPointSamps);
select(portfd+1, (fd_set *)0, &writefds, (fd_set *)0,// Wait for audio buffer to empty
            (struct timeval *)0);
}
}
//printf("*\n");
ALwritesamps(port, block, blockSamps);
}
```

H.2.10 MidiIn.h

```
/*
 * MidiIn.h
 *
 * SGI midi input class.
 * Provides interface to blocking and non-blocking midi reading.
 * Only one object per process.
 *
 */

#if !defined(__MidiIn_h)
#define __MidiIn_h



// Ugly #defs for trying out alternate midi libraries
// No real difference in performance found.

#define MD    // AM or MD determines library for compile
#ifdef AM
    #define MIDI_EVENT AMevent
    #include <AM.h>
#else
    #define MIDI_EVENT MDevent
    #include <dmedia/midi.h>
#endif


#include "global.h"
#include <stdlib.h>
#define MIDI_FORK 0


class MidiIn
{
    protected:
        static void(*processMidiMsg)(MIDI_EVENT*);
        int blockSize;
        int counter;
        MIDI_EVENT *buf;
#ifdef MD
        fd_set fdmdset, fdreadset;
        int fd;
        struct timeval timeout;
#endif

    public:
        ˜MidiIn();
        void readBuf(void);
        void regCheck(void);
        MidiIn(int, void(*)(MIDI_EVENT*));
        friend void midiForkProcess(void*);
};



INLINE void MidiIn :: regCheck()        // Non-blocking, regular check.
{
    if (counter++ < blockSize) return;
    counter = 0;
    readBuf();
}




#endif
```

H.2.11 MidiIn.c

```
/*
 * MidiIn.c
 *
 * SGI midi input class.
 * Provides interface to blocking and non-blocking midi reading.
 * Only one object per process.
 *
 */

// If we used inline c fns in .h the private variables
// could not be protected.. So try inline classes..
// This falls down if we try to make the fork process a member,
// as the members are unbound and cannot be passed as pointers to sproc.
// So we make the forkProcess into a bound function.
// But then midiForkProcess can’t access processMidiMsg..
// So make processMidiMsg static and midiForkProcess a friend fn which
// can then reference processMidiMsg without reference to an object instance.
// This works but it seems pretty ugly.

#include "MidiIn.h"
#include <stdio.h>

#include <unistd.h>        // For select
#include <sys/types.h>
#include <bstring.h>
#include <sys/time.h>

#include <sys/prctl.h>    // For setting up fork process.
```

```
void(*(MidiIn::processMidiMsg))(MIDI_EVENT*); // Static, but only accessible through the class.

#ifdef MD
static MDport inport;
#endif

MidiIn :: MidiIn(int d1, void(*d2)(MIDI_EVENT*)) {

    blockSize = d1;
    processMidiMsg = d2;
    counter = 0;

    buf = (MIDI_EVENT *)calloc(1, sizeof(MIDI_EVENT));


#ifdef AM
if (d1 <= MIDI_FORK) AMopenBlocking("/dev/ttyd2");
        else    AMopen("/dev/ttyd2");
#else // MD
    printf("Number of valid MIDI ports = %d\n", mdInit());

    if ((inport = mdOpenInPort(NULL)) == NULL) {
        fprintf(stderr, "mdOpenInPort error.\n");
        exit(1); }

    mdSetStampMode(inport, MD_NOSTAMP);

    fd = mdGetFd(inport);
    FD_ZERO(&fdmdset);
    FD_SET(fd, &fdmdset);
    timeout.tv_sec = 0;        // Zero wait time for select : non-blocking.
    timeout.tv_usec = 0;
#endif

    if (d1 <= MIDI_FORK) {
        static short int signum=9;
        prctl(PR_SETEXITSIG, (void *)signum);
        sproc(midiForkProcess, PR_SALL);
        system("sleep 1");
    }
}


void MidiIn :: readBuf()        // Non-blocking, regular check.
{
    int result;
    int i=0;
    do{
#ifdef MD
        fdreadset = fdmdset;
        result = select(fd+1, &fdreadset, NULL, NULL, &timeout);
#else
        result = AMgetEvent(buf);
#endif

        if (result>0) {
#ifdef MD
            mdReceive(inport, buf, 1);
#endif
            processMidiMsg(buf);
//            i++;
        }
    }while(result>0);
//    printf("%d\n", i);
}



void midiForkProcess(void*)    // void* because this is the function protype accepted by sproc.
{
    MIDI_EVENT *buf = (MIDI_EVENT *)calloc(1, sizeof(MIDI_EVENT));

    while(1) {
#ifdef AM
        AMgetEvent(buf);    // Blocking
#else
        mdReceive(inport, buf, 1);    // Blocking
#endif
        (MidiIn::processMidiMsg)(buf);
    }
}
```

H.3 Audio Processing Primitives

This is a collection of small audio processing routines, some of which are used to build waveguide operations.

H.3.1 AsyncFilt.h

```
/*
 * AsyncFilt.h
 *
 * Interpolates and upsamples an irregular input.
 * eg : For filtering midi input signals to audio rate.
 *
 * Its predictive,  so has zero latency.
 *
 */

#if !defined(__AsyncFilt_h)
#define __AsyncFilt_h


#include "global.h"

class AsyncFilt {
    protected:
        FLOAT incr;
        long sampsSinceLast;
        int sampGradientCount;
        int maxDeltaSamps;
        int minDeltaSamps;
        FLOAT input;
        FLOAT output;
    public:
        AsyncFilt(int, int);
        AsyncFilt(FLOAT, FLOAT);
        ˜AsyncFilt();
        FLOAT newSamp(void);
        FLOAT getLastOut() { return output; };
        FLOAT getInput() { return input; };
        void setInput(FLOAT);
};


INLINE FLOAT AsyncFilt::newSamp()
{
    if (sampGradientCount > 0) {
        output += incr;
        sampGradientCount--;
    }


    sampsSinceLast++;
    return(output);
}



#endif
```

H.3.2 AsyncFilt.c

```
/*
```

```
 * AsyncFilt.c
 *
 * Interpolates and upsamples an irregular input.
 * eg : For filtering midi input signals to audio rate.
 *
 * Its predictive,  so has zero latency.
 *
 */

#include "AsyncFilt.h"

AsyncFilt::AsyncFilt(FLOAT minDeltaT, FLOAT maxDeltaT)
{
    minDeltaSamps = minDeltaT * AUDIO_RATE;
    maxDeltaSamps = maxDeltaT * AUDIO_RATE;
    output = 0;
    incr = 0;
    sampsSinceLast = 0;
    sampGradientCount = 0;
}

AsyncFilt::AsyncFilt(int d1, int d2)
{
    minDeltaSamps = d1;
    maxDeltaSamps = d2;
    output = 0;
    incr = 0;
    sampsSinceLast = 0;
    sampGradientCount = 0;
}

AsyncFilt::~AsyncFilt() {}


void AsyncFilt::setInput(FLOAT d1)
{
    FLOAT diff;
    input = d1;
    diff = (input-output);
    if (sampsSinceLast > maxDeltaSamps)
        sampsSinceLast = maxDeltaSamps;

    else if (sampsSinceLast < minDeltaSamps)
        sampsSinceLast = minDeltaSamps;

    incr = diff / sampsSinceLast;
    sampGradientCount = sampsSinceLast;

    sampsSinceLast = 0;
}
```

H.3.3 Filter.h

```
/*
 * Filter.h
 *
 * The base class for filters,  in the style of Perry Cook.
 * Not really needed but helps promote logical organisation.
 *
 */

#if !defined(__Filter_h)
#define __Filter_h

#include "global.h"

class Filter
{
  protected:
    FLOAT gain;
    FLOAT *outputs;
    FLOAT *inputs;
    FLOAT lastOutput;
    FLOAT tpidsr;
  public:
    Filter();
    ~Filter();
    FLOAT lastOut() {return lastOutput;};
};

#endif
```

H.3.4 Filter.c

```
/*
 * Filter.c
 *
 * The base class for filters,  in the style of Perry Cook.
 * Not really needed but helps promote logical organisation.
 *
 */

#include "Filter.h"

Filter :: Filter()
{
tpidsr = 2*PI/ AUDIO_RATE;
}

Filter :: ~Filter()
{
}
```

H.3.5 Tone.h

```
/*
 * Tone.h
 *
 * Bare bones one pole low pass filter, to mimic Csound.
 *
 */

#if !defined(__Tone_h)
#define __Tone_h

#include "Filter.h"
#include "math.h"

class Tone : public Filter
{
  protected:
    FLOAT poleCoeff;
    FLOAT sgain;
  public:
    Tone();
    Tone(FLOAT freq);
    ~Tone();
    void clear();
    void setPole(FLOAT pole);
    FLOAT getPole(void) { return(poleCoeff); };
    void setFreq(FLOAT freq);
    FLOAT freq2pole(FLOAT freq);
    FLOAT newSamp(FLOAT sample);
};

INLINE FLOAT Tone :: newSamp(FLOAT sample)
{
    lastOutput = (sgain * sample) + (poleCoeff * lastOutput);
    return lastOutput;
};

INLINE void Tone :: setPole(FLOAT aValue)
{
    poleCoeff = aValue;
    sgain = (1.0 - poleCoeff);
};
```

```
#endif
```

H.3.6 Tone.c

```
/*
 * Tone.c
 *
 * Bare bones one pole low pass filter, to mimic Csound.
 *
 */

#include "Tone.h"

Tone :: Tone() : Filter()
{
    poleCoeff = 0.9;
    sgain = 0.1;
    lastOutput = 0.0;
}

Tone :: Tone(FLOAT freq) : Filter()
{
    setFreq(freq);
    lastOutput = 0.0;
}

Tone :: ~Tone()
{
}

void Tone :: clear()
{
    lastOutput = 0.0;
}

void Tone :: setFreq(FLOAT freq)
{
    double t;

    t = freq * tpidsr;
    if (t > .001) {
        t = 2 - cos(t);
        setPole( t - sqrt(t*t -1) );
    }
    else setPole(1-t);    // Low order approximation for accuracy.

};
```

H.3.7 Atone.h

```
/*
 * Atone.h
 *
 * High Pass filter. 2 zeros,  1 pole.
 * As in Csound, calculation reorganised for efficiency.
 *
 */

#if !defined(__Atone_h)
#define __Atone_h



#include "Filter.h"
#include "math.h"

class Atone : public Filter
{
  protected:
    FLOAT poleCoeff;
  public:
    Atone();
    Atone(FLOAT pole);
    ˜Atone();
    void clear();
    void setPole(FLOAT p) { poleCoeff = p; };
    FLOAT getPole(void) { return(poleCoeff); };
    void setFreq(FLOAT freq);
    FLOAT newSamp(FLOAT sample);
};


INLINE FLOAT Atone :: newSamp(FLOAT sample)
{
/*
 * Using the slightly more efficient Csound algorithm.
 */
    lastOutput = outputs[0] = poleCoeff * (outputs[0] + sample);
    outputs[0] -= sample; // Not really an ’output’ in this case.
    return lastOutput;
}



#endif
```

H.3.8 Atone.c

```
/*
 * Atone.c
 *
 * High Pass filter. 2 zeros,  1 pole .
 * As in Csound, calculation reorganised for efficiency.
 *
 */

#include "Atone.h"

Atone :: Atone() : Filter()
{
    poleCoeff = 0.9;
    outputs = (FLOAT *) malloc(FLOAT_SIZE);
    outputs[0] = 0.0;
}

Atone :: Atone(FLOAT freq) : Filter()
{
    outputs = (FLOAT *) malloc(FLOAT_SIZE);
    outputs[0] = 0.0;
setFreq(freq);
}

Atone :: ˜Atone()
{
    free(outputs);
}

void Atone :: clear()
{
    outputs[0] = 0.0;
    lastOutput = 0.0;
}

void Atone :: setFreq(FLOAT freq)
{
double t;

t = freq * tpidsr;
if (t > .001) {
t = 2 - cos(t);
setPole( t - sqrt(t*t -1) );
}
else setPole(1-t); // Low order approximation for accuracy.
};
```

H.3.9 OnePole.h

```
/*
 * OnePole.h
 *
 * One pole filter class,  after Perry Cook.
 *
 */

#if !defined(__OnePole_h)
#define __OnePole_h

#include "Filter.h"

class OnePole : public Filter
{
  protected:
    FLOAT poleCoeff;
    FLOAT sgain;
  public:
    OnePole();
    ~OnePole();
    void clear();
    void setPole(FLOAT aValue)
    {
    poleCoeff = aValue;
    sgain = gain * (1.0 - poleCoeff);
    };

    void setGain(FLOAT aValue)
    {
    gain = aValue;
    sgain = gain * (1.0 - poleCoeff);  /*  Normalize gain to 1.0 max */
    };

    FLOAT newSamp(FLOAT sample);
};

INLINE FLOAT OnePole :: newSamp(FLOAT
 sample)  /*   Perform Filter Operation */
{
    outputs[0] = (sgain * sample) + (poleCoeff * outputs[0]);
    lastOutput = outputs[0];
    return lastOutput;
}



#endif
```

H.3.10 OnePole.c

```
/*
 * OnePole.c
 *
 * One pole filter class,  after Perry Cook.
 *
 */

#include "OnePole.h"

OnePole :: OnePole() : Filter()
{
    poleCoeff = 0.9;
    gain = 1.0;
    sgain = 0.1;
    outputs = (FLOAT *) malloc(FLOAT_SIZE);
    outputs[0] = 0.0;
}
```

```
OnePole :: ˜OnePole()
{
    free(outputs);
}

void OnePole :: clear()
{
    outputs[0] = 0.0;
    lastOutput = 0.0;
}



/************  Test Main  ************************/
/*
#include <stdio.h>

void main()
{
    long i;
    OnePole test;
    test.setPole(0.99);
    for (i=0;i<150;i++) printf("%lf  ",test.tick(1.0));
    printf("\n\n");

    test.clear();
    test.setPole(0.9);
    test.setGain(2.0);
    for (i=0;i<150;i++) printf("%lf  ",test.tick(0.5));
    printf("\n\n");
}
*/
```

H.3.11 OneZero.h

```
/*
 * OneZero.h
 *
 * One zero filter class,  after Perry Cook
 *
 */

#if !defined(__OneZero_h)
#define __OneZero_h

#include "Filter.h"

class OneZero : public Filter
{
  protected:
    FLOAT zeroCoeff;
    FLOAT sgain;
  public:
    OneZero();
    ˜OneZero();
    void clear();
    void setGain(FLOAT aValue);
    void setCoeff(FLOAT aValue);
    FLOAT newSamp(FLOAT sample);
};

INLINE FLOAT OneZero :: newSamp(FLOAT sample) /*   Perform Filter Operation  */
{
    FLOAT temp;
    temp = sgain * sample;
    lastOutput = (inputs[0] * zeroCoeff) + temp;
    inputs[0] = temp;
    return lastOutput;
}


#endif
```

H.3.12 OneZero.c

```
/*
 * OneZero.h
 *
 * One zero filter class,  after Perry Cook
 *
 */

#include "OneZero.h"

OneZero :: OneZero()
{
    gain = 1.0;
    zeroCoeff = 1.0;
    sgain = 0.5;
    inputs = (FLOAT *) malloc(FLOAT_SIZE);
    this->clear();
}

OneZero :: ˜OneZero()
{
    free(inputs);
}

void OneZero :: clear()
{
    inputs[0] = 0.0;
    lastOutput = 0.0;
}

void OneZero :: setGain(FLOAT aValue)
{
    gain = aValue;
    if (zeroCoeff > 0.0)                  /*  Normalize gain to 1.0 max  */
        sgain = gain / (1.0 + zeroCoeff);
    else
        sgain = gain / (1.0 - zeroCoeff);
}

void OneZero :: setCoeff(FLOAT aValue)
{
    zeroCoeff = aValue;
    if (zeroCoeff > 0.0)                  /*  Normalize gain to 1.0 max  */
        sgain = gain / (1.0 + zeroCoeff);
    else
        sgain = gain / (1.0 - zeroCoeff);
}
```

H.3.13 DCBlock.h

```
/*
 * DCBlock.h
 *
 * Simple high pass filter for blocking DC.
 * Useful for preventing DC build up in a feedback system,
 * for example when using waveguides.
 *
 */

#if !defined(__DCBlock_h)
#define __DCBlock_h

#include "Filter.h"

class DCBlock : public Filter
{
  public:
    DCBlock();
    ˜DCBlock();
    void clear();
    FLOAT newSamp(FLOAT sample);
};
```

```
INLINE FLOAT DCBlock :: newSamp(FLOAT sample)
{
    outputs[0] = sample - inputs[0] + (0.99 * outputs[0]);
    inputs[0] = sample;
    lastOutput = outputs[0];
    return lastOutput;
}


#endif
```

H.3.14 DCBlock.c

```
/*
 * DCBlock.c
 *
 * Simple high pass filter for blocking DC.
 * Useful for preventing DC build up in a feedback system,
 * for example when using waveguides.
 *
 */

#include "DCBlock.h"

DCBlock :: DCBlock()
{
    inputs = (FLOAT *) malloc(FLOAT_SIZE);
    outputs = (FLOAT *) malloc(FLOAT_SIZE);
    this->clear();

}

DCBlock :: ˜DCBlock()
{
    free(inputs);
    free(outputs);
}

void DCBlock :: clear()
{
    outputs[0] = 0.0;
    inputs[0] = 0.0;
    lastOutput = 0.0;
}
```

H.3.15 Delay.h

```
/*
 * Delay.h
 *
 * Simple delay line which uses tables of length 2ˆn to implement
 * efficient wrap around. Any delay size can be generated.
 *
 */

#if !defined(__Delay_h)
#define __Delay_h

#include "Filter.h"

class Delay : public Filter
{
  protected:
    long inPoint;
    long outPoint;
    long length;
    long mask;

  public:
    Delay(long maxSamps);
    Delay(FLOAT maxTime);
    ˜Delay();
    void makeDelay(long samps);
    void clear();
    void setDelayTime(FLOAT time);
    void setDelaySamps(FLOAT samps);
    FLOAT getLastOut(void) { return lastOutput; };
    FLOAT newSamp(FLOAT sample);
};

INLINE FLOAT Delay :: newSamp(FLOAT sample)
{

    inputs[inPoint++] = sample;
    inPoint &= mask;                    // Efficient, no-if, wrap around.

    lastOutput = inputs[outPoint++];
    outPoint &= mask;
    return lastOutput;
}



#endif
```

H.3.16 Delay.c

```
/*
 * Delay.h
 *
 * Simple delay line which uses tables of length 2ˆn to implement
 * efficient wrap around. Any delay size can be generated.
 *
 */


#include "Delay.h"

Delay :: Delay(long samps)
{
    makeDelay(samps);
}


Delay :: Delay(FLOAT time)
{
    makeDelay((long)(AUDIO_RATE * time ));
}

void Delay :: makeDelay(long samps)
{
    length = 1;
    // Need a bit of extra workspace for efficient delay:
    while(length < (samps+1)) length *=2;  // length = 2ˆn
    mask = length-1;
    inputs = (FLOAT *) malloc(length * FLOAT_SIZE);
    this->clear();
    inPoint = 0;
    this->setDelaySamps(samps);    // Can do 0->samps delays

}


Delay :: ˜Delay()
{
    free(inputs);
}

void Delay :: clear()
{
    long i;
    for (i=0;i<length;i++) inputs[i] = 0.0;
    lastOutput = 0;
}
```

```
void Delay :: setDelayTime(FLOAT t)
{
    setDelaySamps(AUDIO_RATE * t);
}


void Delay :: setDelaySamps(FLOAT lag)
{
    outPoint = inPoint - (long) lag;        // read chases write
    while (outPoint<0) outPoint += length;  // modulo maximum length

//    outPoint = (inPoint + length - (long)lag ) & length;    // More efficient.

}
```

H.3.17 OscWG.h

```
/*
 * OscWG.h
 *
 * A waveguide sinewave oscillator after Smith.
 * Amplitude is stable, but freq is only controllable using an offset
 * which changes the amplitude. Can be used for a kind of pseudo FM
 * synthesis.
 *
 */

#include <math.h>
#include "global.h"

class OscWG
{
    private:
        FLOAT x, y;
        FLOAT alpha, alpha0;            // The mult coefficient.
        FLOAT beta;                // Used by setFreqOffset.
        FLOAT tpidsr;            // 2 PI / sample rate

    public:
        OscWG();
        FLOAT lastOut() { return x; };
        void setFreq(FLOAT);
        void setAmp(FLOAT);
        void setAlphaOffset(FLOAT);
        void setFreqOffset(FLOAT);    // An approximation to true offset.
        FLOAT getAlpha() { return alpha; };
        FLOAT getOffsetMult() { return beta; };
        FLOAT newSamp();
};

INLINE void OscWG :: setAlphaOffset(FLOAT o)
{
    alpha = alpha0 + o;
}

INLINE void OscWG :: setFreqOffset(FLOAT f)
{
    alpha = alpha0 + f*beta;
}

INLINE FLOAT OscWG :: newSamp()
{
    FLOAT temp, xTemp;

    temp = alpha*(x+y);
    xTemp = x;
    x = temp-y;
    y = xTemp+temp;

    return x;
}
```

H.3.18 OscWG.c

```
/*
 * OscWG.c
 *
 * A waveguide sinewave oscillator after Smith.
 * Amplitude is stable, but freq is only controllable using an offset
 * which changes the amplitude. Can be used for a kind of pseudo FM
 * synthesis.
 *
 */

#include "OscWG.h"


OscWG :: OscWG()
{
    tpidsr = 2*PI/AUDIO_RATE;
    setFreq(440.0);
    setAmp(1.0);
}

void OscWG :: setFreq(FLOAT f)
{
    alpha0 = alpha = cos(f * tpidsr);
    beta = sin(f * tpidsr)*tpidsr;
}

void OscWG :: setAmp(FLOAT a)
{
    x = a;
    y = 0;
}
```

H.3.19 Noise.h

```
/*
 * Noise.h
 *
 * Noise generator, after Perry Cook.
 *
 */

#if !defined(__Noise_h)
#define __Noise_h

#include "global.h"
#define ONE_OVER_RANDLIMIT 0.00000000093132258

class Noise
{
  protected:
     FLOAT lastOutput;
  public:
    Noise();
    ˜Noise();
    virtual FLOAT newSamp();
    FLOAT lastOut(){ return lastOutput; };
};

INLINE FLOAT Noise :: newSamp()
{
    lastOutput = (FLOAT) random() - 1073741823.0;
    lastOutput *= ONE_OVER_RANDLIMIT;
    return lastOutput;
}


#endif
```

H.3.20 Noise.c

```
/*
 * Noise.h
 *
 * Noise generator, after Perry Cook.
 *
 */

#include "Noise.h"

Noise :: Noise()
{
    lastOutput = 0.0;
}

Noise :: ˜Noise()
{

}
```

H.4 Waveguide Operations

This section includes general waveguide operations, and some specific functions for modelling reed, flute and bow interactions.

H.4.1 Waveguide.h

```
/*
 * Waveguide.h
 *
 * A simple waveguide element consisting of two delay elements.
 *
 */

#include "Delay.h"

class Waveguide
{
    protected:
        Delay *delayInL;
        Delay *delayInR;
        FLOAT inL, inR, outL, outR;

    public:
        Waveguide(FLOAT time);
        Waveguide(long samps);
        ˜Waveguide();
        void newSamp();
        void newSamp(FLOAT inL, FLOAT inR);
        void clear();
        void setInL(FLOAT i) { inL = i; };
        void setInR(FLOAT i) { inR = i; };
        FLOAT getOutL() { return outL; };
        FLOAT getOutR() { return outR; };
};

INLINE void Waveguide :: newSamp()
{
    outL = delayInR->newSamp(inR);
    outR = delayInL->newSamp(inL);
}

INLINE void Waveguide :: newSamp(FLOAT l, FLOAT r)
{
    inL = l;
    inR = r;
    newSamp();
}
```

H.4.2 Waveguide.c

```
/*
 * Waveguide.c
 *
 * A simple waveguide element consisting of two delay elements.
 *
 */



#include "Waveguide.h"

Waveguide :: Waveguide(FLOAT time)
{
    time *= 0.5;    // So that total delay is samps
    delayInL = new Delay(time);
    delayInR = new Delay(time);
}

Waveguide :: Waveguide(long samps)
{
    samps *= 0.5;
    delayInL = new Delay(samps);
    delayInR = new Delay(samps);
}



Waveguide :: ~Waveguide()
{
    delete(delayInL);
    delete(delayInR);
}


void Waveguide :: clear()
{
    delayInL->clear();
    delayInR->clear();
}
```

H.4.3 Barrier.h

```
/*
 * Barrier.h
 *
 * A waveguide junction box which provides an efficient 1 multiplicaion
 * realisation of a lossless 2-port scattering junction.
 * Can be used to model finger holes.
 *
 */

#if !defined(__Barrier_h)
#define __Barrier_h

#include "global.h"

class Barrier
 {
     private:
     FLOAT refl;                    // Reflection coefficient.
     FLOAT inL, inR, outL, outR;    // L,R rep. the two ends.

     public:
     Barrier() { refl = 0; };        // Initially transmitting.
     void setRefl(FLOAT r) { refl = r; };
     void setInL(FLOAT i) { inL = i; };
     void setInR(FLOAT i) { inR = i; };
     void newSamp();
     void newSamp(FLOAT inL, FLOAT inR);
     FLOAT getOutL() { return outL; };
     FLOAT getOutR() { return outR; };
 };


INLINE void Barrier :: newSamp(void)
{
    FLOAT t;
    t = (inL+inR)*-refl;
    outL = inR+t;
    outR = inL+t;
}

INLINE void Barrier :: newSamp(FLOAT l, FLOAT r)
{
    inL = l;
    inR = r;
    newSamp();
}



#endif
```

H.4.4 Barrier.c

```
/*
 * Barrier.c
 *
 * A waveguide junction box which provides an efficient 1 multiplicaion
 * realisation of a lossless 2-port scattering junction.
 * Can be used to model finger holes.
 *
 */

#include "Barrier.h"
```

H.4.5 WhistleBore.h

```
/*
 * WhistleBore.h
 *
 * A simple waveguide model of a penny whistle bore with tone holes
 * modelled with simple 2-port lossless scattering junctions.
 *
 */

 #include "Waveguide.h"
 #include "Delay.h"
 #include "Barrier.h"


 class WhistleBore
 {
    private:
     Waveguide *boreSection1;
    Waveguide *boreSection2;
    Waveguide *boreSection3;
    Waveguide *boreSection4;
    Waveguide *boreSection5;
    Waveguide *boreSection6;
    Delay    *boreEnd;
    Barrier *toneHole1;
    Barrier *toneHole2;
    Barrier *toneHole3;
    Barrier *toneHole4;
    Barrier *toneHole5;
    Barrier *toneHole6;

    FLOAT f0, f1, f2, f3, f4, f5;    // finger positions.
    FLOAT atten;                    // Attenuation per section. Crude.

    FLOAT lastOutput, micOut, micOutL, micOutR;
    public:
    WhistleBore(FLOAT t1, FLOAT t2, FLOAT t3, FLOAT t4, FLOAT t5, FLOAT t6, FLOAT t7);
    WhistleBore(long s1, long s2,  long s3, long s4, long s5, long s6, long s7);
    ~WhistleBore();
    FLOAT newSamp();            // Returns sample out of bore.
    FLOAT newSamp(FLOAT);
    void setIn(FLOAT s) { boreSection1->setInL(s); };
    void setFingers(FLOAT p0, FLOAT p1, FLOAT p2, FLOAT p3, FLOAT p4, FLOAT p5)
        {f0=p0; f1=p1; f2=p2; f3=p3; f4=p4; f5=p5;};
    FLOAT getLastOut() { return lastOutput; };
    FLOAT getMicOutL() { return micOutL; };
    FLOAT getMicOutR() { return micOutR; };
    void setAtten(FLOAT a) { atten = a; };
    void clear();
};



INLINE FLOAT WhistleBore::newSamp()
{

    boreSection1->setInR(toneHole1->getOutL());
    boreSection1->newSamp();


    toneHole1->setRefl(1-f0);        // Full refl when finger up.
    toneHole1->setInL(boreSection1->getOutR());
    toneHole1->setInR(atten* boreSection2->getOutL());
    toneHole1->newSamp();

    boreSection2->newSamp(toneHole1->getOutR(), toneHole2->getOutL());
    toneHole2->setRefl(1-f1);        // Full refl when finger up.
    toneHole2->setInL(boreSection2->getOutR());
    toneHole2->setInR(atten* boreSection3->getOutL());
    toneHole2->newSamp();

    boreSection3->newSamp(toneHole2->getOutR(), toneHole3->getOutL());

    toneHole3->setRefl(1-f2);        // Full refl when finger up.
    toneHole3->setInL(boreSection3->getOutR());
    toneHole3->setInR(atten* boreSection4->getOutL());
    toneHole3->newSamp();

    boreSection4->newSamp(toneHole3->getOutR(), toneHole4->getOutL());

    toneHole4->setRefl(1-f3);        // Full refl when finger up.
    toneHole4->setInL(boreSection4->getOutR());
    toneHole4->setInR(atten* boreSection5->getOutL());
    toneHole4->newSamp();

    boreSection5->newSamp(toneHole4->getOutR(), toneHole5->getOutL());

    toneHole5->setRefl(1-f4);        // Full refl when finger up.
    toneHole5->setInL(boreSection5->getOutR());
    toneHole5->setInR(atten* boreSection6->getOutL());
    toneHole5->newSamp();

    boreSection6->newSamp(toneHole5->getOutR(), toneHole6->getOutL());

    toneHole6->setRefl(1-f5);
    toneHole6->setInL(boreSection6->getOutR());
    toneHole6->setInR(-atten* boreEnd->newSamp(toneHole6->getOutR()) ); // NB - sign
    toneHole6->newSamp();

    micOutL = boreSection1->getOutR()+ boreSection3->getOutR()
            + boreSection5->getOutR();

    micOutR = boreSection2->getOutR() + boreSection4->getOutR()
             + boreSection6->getOutR() + boreEnd->getLastOut();

    micOut = micOutL + micOutR;

    lastOutput = boreSection1->getOutL();
    return(lastOutput);

}
```

```
INLINE FLOAT WhistleBore::newSamp(FLOAT newSamp)
{
    boreSection1->setInL(newSamp);
    return(this->newSamp());
}
```

H.4.6 WhistleBore.c

```
/*
 * WhistleBore.c
 *
 * A simple waveguide model of a penny whistle bore with tone holes
 * modelled with simple 2-port lossless scattering junctions.
 *
 */

#include "WhistleBore.h"

WhistleBore :: WhistleBore(FLOAT t1, FLOAT t2, FLOAT t3, FLOAT t4,
                            FLOAT t5, FLOAT t6, FLOAT t7)
{
    boreSection1 = new Waveguide(t1);    // Upper bore section;
    boreSection2 = new Waveguide(t2);
    boreSection3 = new Waveguide(t3);
    boreSection4 = new Waveguide(t4);    // Upper bore section;
    boreSection5 = new Waveguide(t5);
    boreSection6 = new Waveguide(t6);
    boreEnd = new Delay(t7); // don’t bother with Waveguide here.
    toneHole1 = new Barrier;
    toneHole2 = new Barrier;
    toneHole3 = new Barrier;
    toneHole4 = new Barrier;
    toneHole5 = new Barrier;
    toneHole6 = new Barrier;
    atten = .99;
 }


// Sample length version:
 WhistleBore :: WhistleBore(long t1, long t2, long t3, long t4,
                            long t5, long t6, long t7)
 {
    boreSection1 = new Waveguide(t1);    // Upper bore section;
    boreSection2 = new Waveguide(t2);
    boreSection3 = new Waveguide(t3);
    boreSection4 = new Waveguide(t4);    // Upper bore section;
    boreSection5 = new Waveguide(t5);
    boreSection6 = new Waveguide(t6);
    boreEnd = new Delay(t7); // don’t bother with Waveguide here.
    toneHole1 = new Barrier;
    toneHole2 = new Barrier;
    toneHole3 = new Barrier;
    toneHole4 = new Barrier;
    toneHole5 = new Barrier;
    toneHole6 = new Barrier;
    atten = .99;

 }


 WhistleBore :: ˜WhistleBore()
 {
    delete boreSection1;
    delete boreSection2;
    delete boreSection3;
    delete boreSection4;
    delete boreSection5;
    delete boreSection6;
    delete boreEnd;
    delete toneHole1;
    delete toneHole2;
    delete toneHole3;
    delete toneHole4;
    delete toneHole5;
    delete toneHole6;
 }


void WhistleBore::clear()
{
    boreSection1->clear();
    boreSection2->clear();
    boreSection3->clear();
    boreSection4->clear();
    boreSection5->clear();
    boreSection6->clear();
    boreEnd->clear();

}
```

H.4.7 WhistleBoreP.h

```
/*
 * WhistleBoreP.h
 *
 * A simple waveguide model of a penny whistle bore with tone holes
 * modelled with parallel delay lines.
 *
 */

 #include "Delay.h"


 class WhistleBoreP
 {
    private:
    Delay *bore;        // Closed hole bore delay.
    Delay *toneHole[6];    // Delay from mouthpiece to tonehole.

    FLOAT m0, m1, m2, m3, m4, m5;    // Delay mix values.
    FLOAT atten;                    // Attenuation per section. Crude.
    FLOAT input;

    FLOAT lastOutput, micOut, micOutL, micOutR;

    public:
    WhistleBoreP(FLOAT t, FLOAT t1, FLOAT t2, FLOAT t3, FLOAT t4,
                                FLOAT t5, FLOAT t6);
    WhistleBoreP(long s, long s1, long s2,  long s3, long s4, long s5,
                                long s6);
    ~WhistleBoreP();
    FLOAT newSamp();            // Returns sample out of bore.
    FLOAT newSamp(FLOAT in);
    void setIn(FLOAT i) { input = i; };
    void setFingers(FLOAT p0, FLOAT p1, FLOAT p2, FLOAT p3, FLOAT p4, FLOAT p5)
        {m0=p0; m1=p1; m2=p2; m3=p3; m4=p4; m5=p5;};
    FLOAT getLastOut() { return lastOutput; };
    FLOAT getMicOut() { return micOut; };
    FLOAT getMicOutL() { return micOutL; };
    FLOAT getMicOutR() { return micOutR; };
    void setAtten(FLOAT a) { atten = a; };
    void setDelay(int f, FLOAT t);
    void clear();
};



INLINE FLOAT WhistleBoreP::newSamp()
{
//printf("%f%f%f%f%f%f\n", m0, m1, m2, m3, m4, m5);
    lastOutput = 0;
    micOutL = 0;
    micOutR = 0;

    lastOutput -= bore->newSamp(input);        // Inversion due to end reflection.
    micOutL -= toneHole[0]->newSamp(input)*m0;
    micOutR -= toneHole[1]->newSamp(input)*m1;
    micOutL -= toneHole[2]->newSamp(input)*m2;
    micOutR -= toneHole[3]->newSamp(input)*m3;
    micOutL -= toneHole[4]->newSamp(input)*m4;
    micOutR -= toneHole[5]->newSamp(input)*m5;

    micOut = micOutL + micOutR + lastOutput;
    micOutL += lastOutput;
    micOutR += lastOutput;

    lastOutput = micOut;

    return(lastOutput);

}

INLINE FLOAT WhistleBoreP::newSamp(FLOAT newSamp)
{
    input = newSamp;
    return(this->newSamp());
}
```

H.4.8 WhistleBoreP.c

```
/*
 * WhistleBoreP.c
 *
 * A simple waveguide model of a penny whistle bore with tone holes
 * modelled with parallel delay lines.
 *
 */


#include "WhistleBoreP.h"

WhistleBoreP :: WhistleBoreP(FLOAT t, FLOAT t1, FLOAT t2, FLOAT t3, FLOAT t4,
                            FLOAT t5, FLOAT t6)
{
    bore = new Delay(t);
    toneHole[0] = new Delay(t1);
    toneHole[1] = new Delay(t2);
    toneHole[2] = new Delay(t3);
    toneHole[3] = new Delay(t4);
    toneHole[4] = new Delay(t5);
    toneHole[5] = new Delay(t6);
 }


// Sample length version:
WhistleBoreP :: WhistleBoreP(long s, long s1, long s2, long s3, long s4,
                            long s5, long s6)
{
    bore = new Delay(s);
    toneHole[0] = new Delay(s1);
    toneHole[1] = new Delay(s2);
    toneHole[2] = new Delay(s3);
    toneHole[3] = new Delay(s4);
    toneHole[4] = new Delay(s5);
    toneHole[5] = new Delay(s6);
 }


 WhistleBoreP :: ~WhistleBoreP()
 {
    delete bore;
    delete toneHole[0];
    delete toneHole[1];
    delete toneHole[2];
    delete toneHole[3];
    delete toneHole[4];
    delete toneHole[5];
 }


void WhistleBoreP::clear()
```

```
{
    bore->clear();
    toneHole[0]->clear();
    toneHole[1]->clear();
    toneHole[2]->clear();
    toneHole[3]->clear();
    toneHole[4]->clear();
    toneHole[5]->clear();
}


void WhistleBoreP :: setDelay(int f, FLOAT t)
{
    toneHole[f]->setDelayTime(t);
}
```

H.4.9 ReedTabl.h

```
#if !defined(__CWreedTabl_h)
#define __CWreedTabl_h

/*
 * ReedTabl.h
 *
 * Reed non-linearity approximation,
 * for use with reed waveguide scheme by J Smith.
 *
 */

#include "global.h"

class ReedTabl
{
  protected:
    FLOAT offSet;
    FLOAT slope;
    FLOAT lastOutput;
  public:
    ReedTabl();
    ~ReedTabl();
    void setOffset(double aValue);
    void setSlope(double aValue);
    FLOAT lookup(double deltaP);
    FLOAT lastOut() { return lastOutput; };
};

INLINE FLOAT ReedTabl :: lookup(double deltaP)
    /*   Perform "Table Lookup" by direct clipped  */
    /*   linear function calculation               */
    /*   deltaP is differential reed pressure      */
{
    lastOutput = offSet + (slope * deltaP); /* compute basic non-linearity     */
    if (lastOutput > 1.0) lastOutput = 1.0;     /* if other way, reed slams shut   */
    if (lastOutput < -1.0) lastOutput = -1.0;   /* if all the way open, acts like open end */
    return lastOutput;
}


#endif
```

H.4.10 ReedTabl.c

```
#if !defined(__CWreedTabl_h)
#define __CWreedTabl_h

/*
 * ReedTabl.c
 *
 * Reed non-linearity approximation,
 * for use with reed waveguide scheme by J Smith.
 *
 */
```

```
#include "ReedTabl.h"

ReedTabl :: ReedTabl()
{
    offSet = 0.6;    /* Offset is a bias, related to reed rest position  */
    slope = -0.8;    /* Slope corresponds loosely to reed stiffness      */
}

ReedTabl :: ˜ReedTabl()
{

}

void ReedTabl :: setOffset(double aValue)
{
    offSet = aValue;     /* Offset is a bias, related to reed rest position */
}

void ReedTabl :: setSlope(double aValue)
{
    slope = aValue;      /* Slope corresponds loosely to reed stiffness  */
}
```

H.4.11 JetTabl.h

```
/*
 * JetTabl.h
 *
 * Jet nonlinearity approximation,
 * for use in waveguide implementation by P Cook et al.
 *
 */

#include "global.h"

class JetTabl
{
  protected:
    FLOAT lastOutput;
  public:
    JetTabl();
    ˜JetTabl();
    FLOAT lookup(FLOAT deltaP);
    FLOAT lastOut() { return lastOutput; };
};


INLINE FLOAT JetTabl :: lookup(FLOAT sample) /*   Perform "Table Lookup"    */
{                                        /*   By Polynomial Calculation */


// Alternative sigmoid function ( original seems flawed. )

/*
    if (sample>0) lastOutput = -sample/(sample+3);

    else lastOutput = -sample/(3-sample);
*/

// 'Correct' sigmoid approximation, but dosen't play as easily as
//    the 'incorrect' original.

/*
sample *= .8;
    lastOutput = sample * (sample*sample - 1.0);
    if (lastOutput > .385)
        lastOutput = .385;
    else if (lastOutput < -.385)
        lastOutput = -.385;
lastOutput *= 1.4;
*/
// Original... More complex than a sigmoid.

    lastOutput = sample *
                (sample*sample - 1.0);
    if (lastOutput > 1.0)
        lastOutput = 1.0;
    else if (lastOutput < -1.0)
        lastOutput = -1.0;


    return lastOutput;

}
```

H.4.12 JetTabl.c

```
/*
 * JetTabl.c
 *
 * Jet nonlinearity approximation,
 * for use in waveguide implementation by P Cook et al.
 *
 */

#include "JetTabl.h"

JetTabl :: JetTabl()
{
    lastOutput = 0.0;
}

JetTabl :: ˜JetTabl()
{
}
```

H.4.13 FluteMouth.h

```
/*
 * FluteMouth.h
 *
 * Model of a flute mouthpiece after Cook, Karajalein
 *
 * NB bore filter is not included!
 *
 */


#if !defined(__FluteMouth_h)
#define __FluteMouth_h


#include "JetTabl.h"
#include "DLineL.h"
#include "OnePole.h"
#include "DCBlock.h"
#include "Noise.h"


class FluteMouth
{
  protected:
    DLineL *jetDelay;
    JetTabl *jetTable;
    DCBlock *dcBlock;
    Noise *noise;
    FLOAT jetRefl;
    FLOAT endRefl;
    FLOAT jetLength;
    FLOAT noiseGain;
    FLOAT breathPressure;
    FLOAT fromBore;
    FLOAT lastOutput;    // to bore.

  public:
    FluteMouth(FLOAT time);
    ~FluteMouth();
    void clear();
    void setJetRefl(FLOAT refl){ jetRefl = refl; };
    void setEndRefl(FLOAT refl){ endRefl = refl; };
    void setJetFreq(FLOAT);
    void setJetTime(FLOAT);
    void setJetRatio(FLOAT ratio) { jetDelay->setDelay(jetLength*ratio); };
    void setBreath(FLOAT b) { breathPressure = b; };
    void setBoreIn(FLOAT b) { fromBore = b; };
    void setNoiseGain(FLOAT n) { noiseGain = n; };
    FLOAT newSamp();
    FLOAT newSamp(FLOAT fromBore, FLOAT newSamp);
};

INLINE FLOAT FluteMouth :: newSamp(FLOAT fromBore, FLOAT breathPressure)
{
    FLOAT temp;
    FLOAT pressureDiff;
    FLOAT randPressure;

    randPressure = noiseGain * noise->newSamp();
    randPressure *= breathPressure;
    temp = dcBlock->newSamp(fromBore);                 // Block DC on reflection
    pressureDiff = breathPressure + randPressure    // Breath Pressure
                 - (jetRefl * temp);                //    - reflected
    pressureDiff = jetDelay->newSamp(pressureDiff);    // Jet Delay Line
    lastOutput = jetTable->lookup(pressureDiff)        // Non-Lin Jet + reflected
            + endRefl * temp;
    return lastOutput;

}

INLINE FLOAT FluteMouth :: newSamp()
{
    return newSamp(fromBore, breathPressure);
}


#endif
```

H.4.14 FluteMouth.c

```
/*
 * FluteMouth.c
 *
 * Model of a flute mouthpiece after Cook.
 *
 * NB bore filter is not included!
 *
 */

#include "FluteMouth.h"

FluteMouth :: FluteMouth(FLOAT time)
{
    jetLength =  (AUDIO_RATE * time + 1);
    jetDelay = new DLineL(jetLength);
    jetTable = new JetTabl;
    dcBlock = new DCBlock;
    noise = new Noise;
    this->clear();

    jetDelay->setDelay(jetLength * .5);

    jetRefl = 0.5;
    endRefl = 0.5;
    noiseGain = 0.15;          /* Breath pressure random component   */
}
```

```
FluteMouth :: ˜FluteMouth()
{
    delete jetDelay;
    delete jetTable;
    delete dcBlock;
    delete noise;
}

void FluteMouth :: clear()
{
    jetDelay->clear();
    dcBlock->clear();
}

void FluteMouth :: setJetFreq(FLOAT freq)
{
    FLOAT temp;
    temp = AUDIO_RATE / freq - 2.0;
    if (temp > jetLength) temp = jetLength;
    jetDelay->setDelay(temp);
}

void FluteMouth :: setJetTime(FLOAT time)
{
    FLOAT temp;
    temp = AUDIO_RATE * time - 2.0;
    if (temp > jetLength) temp = jetLength;
    jetDelay->setDelay(temp);
}
```

H.4.15 WaveMass.h

```
/*
 * WaveMass.h
 *
 * WaveMass is a model for a massive particle which is interacting with
 * two *velocity* waveguides attached to opposite ends.
 *
 */

#include "global.h"

class WaveMass
{
    protected:
        FLOAT posn;        // Displacement, scaled for efficiency.
        FLOAT vel;        // 'True' Velocity.
        FLOAT accel;    // Acceleration, scaled for efficiency.
        FLOAT inL, inR, outL, outR;
        FLOAT dt;        // One over sample rate.
        FLOAT zL, zR;

    public:
        WaveMass();
        WaveMass(FLOAT zL,  FLOAT zR);
        WaveMass(FLOAT z);
        void setInL(FLOAT i) { inL = i; };
        void setInR(FLOAT i) { inR = i; };
        void setZL(FLOAT z) { zL = z*dt; };        // Premultiply dt to increase
        void setZR(FLOAT z) { zR = z*dt; };        // efficiency later.
        void setZ(FLOAT z) { zL = zR = z*dt; };
        FLOAT getOutL() { return outL; };
        FLOAT getOutR() { return outR; };
        virtual void newSamp();
        virtual void newSamp(FLOAT inL, FLOAT inR);
};



INLINE void WaveMass :: newSamp()
{
    outL = vel - inL;
    outR = vel - inR;
    // In the following digital filter approx of the dynamics of a mass,
    // dt has been comuted into the impedances zL and zR. Mass is taken
    // be unity.
    // 'posn' is a scaled true posn.

    accel = (inL-outL)*zL  +  (inR-outR)*zR;
    vel += accel;
    posn += vel;
}

INLINE void WaveMass :: newSamp(FLOAT l, FLOAT r){
    inL = l;
    inR = r;

    newSamp();
}
```

H.4.16 WaveMass.c

```
/*
 * WaveMass.c
 *
 * WaveMass is a model for a massive particle which is interacting with
 * two *velocity* waveguides attached to opposite ends.
 *
 */

#include "WaveMass.h"

WaveMass :: WaveMass()
{
    dt = 1 / AUDIO_RATE;
    setZL(1);
    setZR(1);
}

WaveMass :: WaveMass(FLOAT l, FLOAT r)
{
    dt = 1 / AUDIO_RATE;
    setZL(l);
    setZR(r);
}

WaveMass :: WaveMass(FLOAT lr)
{
    dt = 1 / AUDIO_RATE;
    setZL(lr);
    setZR(lr);
}
```

H.4.17 MassBow.h

```
/*
 * MassBow.h
 *
 * Model for a bowed massive particle imbedded in a string.
 * Velocity waves used.
 * This may yield more interesting results than a direct wave model
 * (WaveBow)
 *
 */


 #include "WaveMass.h"


class MassBow : public WaveMass
{
    private:
        enum{slip, stick} state;
        FLOAT bowForce;    // Force applied normal to bow, by player.
        FLOAT bowVel;
        FLOAT Fdiff, Fmin, Fmax, Fscale;    // Friction curve parameters.
        FLOAT velDiff;

    public:
        MassBow();
        void setBowForce(FLOAT f) { bowForce = f; };
        void setBowVel(FLOAT v) { bowVel = v; };
        void setMaxFriction(FLOAT f) { Fmax = f*dt; Fdiff = Fmax-Fmin; };
        void setMinFriction(FLOAT f) { Fmin = f*dt; Fdiff = Fmax-Fmin; };
        void setFrictionScale(FLOAT s) { Fscale = s; };
        void newSamp();

};
```

H.4.18 MassBow.c

```
/*
 * MassBow.c
 *
 * Model for a bowed massive particle imbedded in a string.
 * Velocity waves used.
 * This may yield more interesting results than a direct wave model
 * (WaveBow)
 *
 */


#include"MassBow.h"

MassBow :: MassBow()
{
    accel = vel = posn = 0;
}



void MassBow :: newSamp()
{
    FLOAT newVelDiff, bowTension, friction;

    outL = vel - inL;
    outR = vel - inR;

    // In the following digital filter approx of the dynamics of a mass,
    // dt has been commuted into the impedances zL and zR.
    // Mass is taken to be unity, but can be effected by scaling
    // zL and zR up by desired mass.
    // 'posn' is a scaled from true posn to improve efficiency.


    newVelDiff = vel - bowVel;

    if (state == slip) {

        // Simple friction curve:
        friction = Fmin + Fdiff/(1-newVelDiff*Fscale);

        accel = (inL-outL)*zL  +  (inR-outR)*zR
                + friction;

        vel += accel;

        if (newVelDiff*velDiff <0) {
            state = stick;                // VelDiff crossing zero.
            accel = 0;
        }


    }
    else { // stick : Approx no acceleration -> total force zero

        bowTension = (inL-outL)*zL  +  (inR-outR)*zR;
        if (bowTension > Fmax) state = slip;
    }

    velDiff = newVelDiff;


}
```

H.4.19 WaveBow.h

```
/*
 * WaveBow.h
 *
 * Model for bowing a string implemented entirely with velocity waves,
 * as in McIntyre, Woodhouse, Schumacher.
 * Unlike the implementation by Smith, slip-stick hysteresis is included.
 *
 */

#include "global.h"

class WaveBow
{
    protected:
        FLOAT inL, inR, outL, outR;
        enum{slip, stick} state;
        FLOAT bowForce, bowVel;
        FLOAT Vstick, Vslip;    // Transitions for velDiff.
        FLOAT Fslip;            // Sticking friction
        FLOAT FminRatio, FstickRatio;    // Friction ratio parameters.
        FLOAT Fscale;            // Shape of slipping friction curve.
        FLOAT alpha;            // Slip curve calculation parameter.
        FLOAT beta;                // Stick curve calculation parameter.

    public:
        WaveBow();
        void setInL(FLOAT i) { inL = i; };
        void setInR(FLOAT i) { inR = i; };
        void setBowForce(FLOAT f) { bowForce = Fslip = f; };
        void setBowVel(FLOAT v) { bowVel = v; };

        void bowFunctionInit();
        void setMinFrictionRatio(FLOAT f) { FminRatio = f; bowFunctionInit();};
        void setStickFrictionRatio(FLOAT f) { FstickRatio = f; bowFunctionInit();};
        void setFrictionScale(FLOAT s) { Fscale = s; bowFunctionInit();};
        void setStickVel(FLOAT v) { Vstick = v; };
        void setSlipVel(FLOAT v) { Vslip = v; };

        void newSamp();
        void newSamp(FLOAT inL, FLOAT inR);
};
```

H.4.20 WaveBow.c

```
/*
 * WaveBow.c
 *
 * Model for bowing a string implemented entirely with velocity waves,
 * as in McIntyre, Woodhouse, Schumacher.
 * Unlike the implementation by Smith, slip-stick hysteresis is included.
 *
 */

#include "WaveBow.h"

WaveBow :: WaveBow()
{
    Vscale = 1;
    Vstick = .5;
    Vslip = 1;
    FstickR = .3;    // Make sure fslip <fstick at +-Vstick.
    FminR = .2
}


void WaveBow :: bowFunctionInit()
{
    alpha = -Vscale*Vstick +(1/(FminR+FstickR));
    beta = -1/Vslip;
}


void WaveBow :: newSamp()
{
    FLOAT velDiff, alpha2;

    velDiff = (inL+inR)-velBow;  // 'Vh-Vb' in Woodhouse.


    if (state == slip) {

        if( velDiff < 0 ) alpha2 = alpha; else alpha2 = -alpha;

        friction = FminR - 1/(Vscale*velDiff - alpha2);

        if (velDiff < Vstick && velDiff > -Vstick) state = stick;

    }

    else {

        friction = beta * velDiff;

        if ( velDiff > Vslip || velDiff < -Vslip ) state = slip;
    }


    outL += friction;        // force applied by the bow.
    outR += friction;

}
```

H.5 CyberWhistle Instruments

Atlast, here are the main instruments, including a base class which all instruments inhereit.

H.5.1 CWinstr.h

```
/*
 * CWinstr.h
 *
 * Base class definition for CyberWhistle functions.
 * Includes functions for converting midi to finger positions
 * and audio to breath pressure.
 *
 */

#if !defined(__CWinstr_h)
#define __CWinstr_h


#include "global.h"
#include "MidiIn.h"
#include "AsyncFilt.h"
#include "Atone.h"
#include "Tone.h"


class CWinstr
{
    private:
    Atone *rumbleCut;                    // Mic processing
    Tone *smoothDetect;
    Tone *trigLowPass;
    Atone *trigHighPass;
    Tone *smoothBreath;
    Tone *smoothBreath2;
    Tone *growlLP1;
    Tone *growlLP2;

    FLOAT dip;
    FLOAT smoothPole;            // smoothBreath pole mid-note.
    FLOAT trigger, onLevel, offLevel;

    float fingerUpMidi[6];        // MIDI values giving 'standard' range. Could be an array
    float fingerDownMidi[6];    //
    float fingerUpVal[6];        // Gives the range of the value passed to finger[].
     float fingerDownVal[6];        //
    float fingerUpPos[6];        // Sets the up/down limit poistions within
    float fingerDownPos[6];        // the maximum physical range determined by hardware.

    FLOAT fingerBig[6];            // Calculation parameters.
    FLOAT fingerSmall[6];        //
    FLOAT fingerAdd[6];            //
    FLOAT fingerMult[6];        //

    FLOAT growlGain;

    protected:
    FLOAT lastOutput;
    FLOAT lastOutputL;
    FLOAT lastOutputR;
    FLOAT breath;
    AsyncFilt *finger[6];
    virtual void setBreath(FLOAT b) { breath = b; };
    virtual void setFinger(int f, FLOAT v) { finger[f]->setInput(v); };
    void setFingerPos(FLOAT d, FLOAT u);    // Finger position range..
    void setFingerVals(FLOAT d, FLOAT u);    // .. mapped to value range.
    void setFingerPos(int f, FLOAT d, FLOAT u);
    void setFingerVals(int f, FLOAT d, FLOAT u);

    void calcFingerParams(int f);

    void setSmoothBreathFreq(FLOAT freq);    // External breath tweaks.
    void setBreathEdgeResponseMax(FLOAT d) { dip = d; };
    void setBreathEdgeResponseWindow(FLOAT t) { trigLowPass->setFreq(1/t); };

    void setGrowlLPfreq(FLOAT freq);
    void setGrowlGain(FLOAT gain);
    virtual FLOAT growlNewSamp(FLOAT mic);

    public:
    CWinstr();
    ~CWinstr();
    FLOAT lastOut() { return lastOutput; };
    FLOAT lastOutL() { return lastOutputL; };
    FLOAT lastOutR() { return lastOutputR; };
    virtual void processMidi(MIDI_EVENT* buf);
    virtual void processMidiC(MIDI_EVENT* buf);    // For calibration:
                                                // Produces switch finger
                                                // output.
    virtual FLOAT breathNewSamp(FLOAT samp);
};


INLINE FLOAT CWinstr::breathNewSamp(FLOAT mic)
{
    register FLOAT detect, pulse;

    mic = rumbleCut->newSamp(mic);    // Remove DC at note edges, and finger noise.
    detect = mic*mic*.0000035;        // Square, cheaper than abs, + increases dynamic range.
    detect = smoothDetect->newSamp(detect);

    if (detect > onLevel) trigger = dip;        // Hysteresis against
    else if (detect < offLevel) trigger = 0;    // left over from mic.

    detect -= onLevel;                // Ensure clean switch on.
    if (detect < 0) detect = 0;        //
    pulse = trigLowPass->newSamp(trigger);
    pulse = trigHighPass->newSamp(pulse);
    pulse *= pulse;            // Now have shaped pulses at note edges.

    smoothBreath->setPole(smoothPole-pulse);    // Trigger pulse causes momentary
    smoothBreath2->setPole(smoothPole-pulse);    // increase in low pass cutoff freq
    breath = smoothBreath->newSamp(detect);        // = response improvement.
    breath = smoothBreath2->newSamp(breath);    // Cascaded 1-pole filters.

//    printf("%f\n", breath);

    return(breath);    // Aim to make breath in the range [0,1]
}

INLINE FLOAT CWinstr :: growlNewSamp(FLOAT mic)
{
return( growlGain * growlLP1->newSamp(growlLP2->newSamp(mic)) );
}

#endif
```

H.5.2 CWinstr.c

```
/*
 * CWinstr.c
 *
 * Base class definition for CyberWhistle functions.
 * Includes functions for converting midi to finger positions
 * and audio to breath pressure.
 *
 */


#include "CWinstr.h"

CWinstr :: CWinstr()
{
    int f;
    FILE *calibFile;
    // Calibration data for finger sensors:
    float fDown[6] = { 95.0, 105.0, 105.0, 105.0, 110.0, 105.0 };

    rumbleCut = new Atone(5000);    // Mic processing.
    smoothDetect = new Tone(50);    // Internal stuff.
    trigger = 0;
    onLevel = .001;
    offLevel = .0005;
    trigHighPass = new Atone(200);

    smoothBreath = new Tone();        // Externally settable stuff.
    smoothBreath2 = new Tone();
    trigLowPass = new Tone();

    // Default breath settings: smooth but quick response in-note,
    // + fast on/off response.
    setSmoothBreathFreq(4);
    setBreathEdgeResponseMax(2.5);
    setBreathEdgeResponseWindow(.05);


    for(f=0; f<6; f++) {
        finger[f] = new AsyncFilt(0.01, .1);    // Normal
//        finger[f] = new AsyncFilt(0.01, .01);    // For calibration
        fingerUpMidi[f] = 0.0;
        fingerDownMidi[f] = fDown[f];        // This undercuts the actual maximum
                                // midi values received, to avoid the
                                // region where LDRs become sluggish.
    }

    if ( (calibFile = fopen("calibrate", "r")) != NULL) {
        printf("\nReading calibration file.\n");
        for(f=0; f<6; f++) {
            fscanf(calibFile, "%f%f", &(fingerUpMidi[f]),
                            &(fingerDownMidi[f]));
printf("%f  %f\n", fingerUpMidi[f], fingerDownMidi[f]);
        }
        fclose(calibFile);
    }
    else printf("\nUsing default calibration.\n");


    setFingerPos(0, 1);        // Use full physical range.
    setFingerVals(1, 0);    // Finger down gives 1, up gives 0;

    growlLP1 = new Tone(300);
    growlLP2 = new Tone(300);

    setGrowlGain(.0002);

}

CWinstr::~CWinstr()
{
    int i;
    for(i=0; i<6; i++) delete finger[i];
    delete rumbleCut;
    delete smoothDetect;
    delete trigLowPass;
    delete trigHighPass;
    delete smoothBreath;

    delete growlLP1;
    delete growlLP2;
}

void CWinstr :: processMidi(MIDI_EVENT* buf)
{
    int f = 5-(buf->msg[1]);    // Reordered to make less
                                // confusing with waveguide models:
                                // f = 0 denotes tonehole at mouthend.

    if ((buf->msg[0] & 0xf0) == 0xB0) {
        FLOAT scaled = (buf->msg[2])*fingerMult[f] +fingerAdd[f];
        if (scaled < fingerSmall[f]) scaled = fingerSmall[f];
        else if (scaled > fingerBig[f]) scaled = fingerBig[f];

// printf("%f\n", scaled);
        setFinger(f, scaled);
    }

}

// Test version of above, generates 2 state finger output.

void CWinstr :: processMidiC(MIDI_EVENT* buf)
{
    int f = 5-(buf->msg[1]);    // Reordered to make less
                                // confusing with waveguide models:
                                // f = 0 denotes tonehole at mouthend.

    if ((buf->msg[0] & 0xf0) == 0xB0) {
        FLOAT scaled = (buf->msg[2])*fingerMult[f] +fingerAdd[f];
        if (scaled <= fingerSmall[f]) setFinger(f, fingerSmall[f]);
        else if (scaled >= fingerBig[f]) setFinger(f, fingerBig[f]);

// printf("%f\n", scaled);
    }

}

void CWinstr :: setSmoothBreathFreq(FLOAT freq)
{
    smoothBreath->setFreq(freq);
    smoothBreath2->setFreq(freq);
    smoothPole = smoothBreath->getPole();
}


void CWinstr :: calcFingerParams(int f)
```

```
{
    FLOAT a, b, c, d, g;

    g = fingerDownMidi[f] - fingerUpMidi[f];
    a = fingerDownMidi[f] - g * fingerUpPos[f];
    b = fingerDownMidi[f] - g * fingerDownPos[f];
    c = fingerUpVal[f];
    d = fingerDownVal[f];


    fingerMult[f] = (d-c)/(b-a);
    fingerAdd[f] = c - a*fingerMult[f];

    if (c > d) {
        fingerBig[f] = c;
        fingerSmall[f] = d;
    }
    else {
        fingerBig[f] = d;
        fingerSmall[f] = c;
    }
}

void CWinstr :: setFingerPos(FLOAT d, FLOAT u)
{
    int f;
    for(f=0; f<6; f++)
        setFingerPos(f, d, u);
}

void CWinstr :: setFingerPos(int f, FLOAT d, FLOAT u)
{
    fingerDownPos[f] = d;
    fingerUpPos[f] = u;
    calcFingerParams(f);
}

void CWinstr :: setFingerVals(FLOAT d, FLOAT u)
{
    int f;
    for(f=0; f<6; f++)
        setFingerVals(f, d, u);
}

void CWinstr :: setFingerVals(int f, FLOAT d, FLOAT u)
{
    fingerDownVal[f] = d;
    fingerUpVal[f] = u;
    calcFingerParams(f);
}



void CWinstr :: setGrowlLPfreq(FLOAT f)
{
    growlLP1->setFreq(f);
    growlLP2->setFreq(f);
}

void CWinstr :: setGrowlGain(FLOAT g)
{
    growlGain = g;
}
```

H.5.3 CWtest.h

```
/*
 * CWtest.h
 *
 * Simple CyberWhistle instrument for testing the hardware,
 * and software response times.
 * Each finger controls the amplitude of a square wave oscillator.
 * The square waves generate subharmonic aliased components,
 * so they don’t sound too unpleasant.
 *
 */

#include <stdio.h>
#include "global.h"
#include "CWinstr.h"

class CWtest : public CWinstr
{
    protected:
        int i;
    public:
        CWtest();
        ~CWtest();
        FLOAT newSamp(FLOAT mic);
};


INLINE FLOAT CWtest :: newSamp(FLOAT mic)
{


// Create a harmonically-aliased tone series.
// Max amplitude is 3

#define OCTAVE 8

lastOutput =
      ( ((i/2)&OCTAVE)/OCTAVE -.5)*finger[0]->newSamp()
    + ( ((i/3)&OCTAVE)/OCTAVE -.5)*finger[1]->newSamp()
    + ( ((i/4)&OCTAVE)/OCTAVE -.5)*finger[2]->newSamp()
    + ( ((i/5)&OCTAVE)/OCTAVE -.5)*finger[3]->newSamp()
    + ( ((i/6)&OCTAVE)/OCTAVE -.5)*finger[4]->newSamp()
    + ( ((i/7)&OCTAVE)/OCTAVE -.5)*finger[5]->newSamp();


lastOutput *= breathNewSamp(mic);
i++;

return(lastOutput*5000);


}
```

H.5.4 CWtest.c

```
/*
 * CWtest.h
 *
 * Simple CyberWhistle instrument for testing the hardware,
 * and software response times.
 * Each finger controls the amplitude of a square wave oscillator.
 * The square waves generate subharmonic aliased components,
 * so they don’t sound too unpleasant.
 *
 */

#include "CWtest.h"


CWtest::CWtest() : CWinstr()
{
    i = 0;                // Sample counter for synthesis.

}


CWtest::~CWtest()
{
}
```

H.5.5 CWtestSine.h

```
/*
 * CWtestSine.h
 *
 * Simple Cyber Whistle instrument for testing things generally.
 * Each finger controls the amplitude of a sine wave oscillator.
 *
 */

#include <stdio.h>
#include "global.h"
#include "CWinstr.h"
#include "OscWG.h"

class CWtestSine : public CWinstr
{
    protected:
        OscWG* osc[6];
    public:
        CWtestSine();
        ~CWtestSine();
        FLOAT newSamp(FLOAT mic);
};


INLINE FLOAT CWtestSine :: newSamp(FLOAT mic)
{
FLOAT temp;

lastOutputL =
    +  osc[0]->newSamp()*finger[0]->newSamp()
    +  osc[2]->newSamp()*finger[2]->newSamp()
    +  osc[4]->newSamp()*finger[4]->newSamp();

lastOutputR =
    +  osc[1]->newSamp()*finger[1]->newSamp()
    +  osc[3]->newSamp()*finger[3]->newSamp()
    +  osc[5]->newSamp()*finger[5]->newSamp();

//printf("%f\n", finger[1]->getInput());

temp = breathNewSamp(mic)*5000;

lastOutputL *= temp;
lastOutputR *= temp;

lastOutput = lastOutputL + lastOutputR;

return(lastOutput*5000);


}
```

H.5.6 CWtestSine.c

```
/*
 * CWtestSine.c
 *
 * Simple Cyber Whistle instrument for testing things generally.
 * Each finger controls the amplitude of a sine wave oscillator.
 *
 */

#include "CWtestSine.h"


CWtestSine::CWtestSine() //: CWinstr()
{
    int i;

    for(i=0; i<6; i++) {
        osc[i] = new OscWG();
        osc[i]->setAmp(1);
    }

    osc[0]->setFreq(400.0);
    osc[1]->setFreq(440.0);
    osc[2]->setFreq(500.0);
    osc[3]->setFreq(600.0);
    osc[4]->setFreq(750.0);
    osc[5]->setFreq(800.0);

//    setFingerPos(0, .5);    // Makes finger range work over 1/2 the maximum distance.

}


CWtestSine::~CWtestSine()
{
}
```

H.5.7 CWclariHol.h

```
/*
 * CWclariHol.h
 *
 * Single reed model after Smith.
 * 6-tone hole bore modelled with simple 2-port junctions.
 *
 */


#if !defined(__CWclariHol_h)
#define __CWclariHol_h


#include <stdio.h>
#include "global.h"
#include "CWinstr.h"

#include "ReedTabl.h"
#include "OneZero.h"
#include "Noise.h"

#include "Delay.h"
#include "Tone.h"
#include "WhistleBore.h"

class CWclariHol : public CWinstr
{
  protected:
    ReedTabl *reedTable;
//    OneZero *filter;    // Alternative filters for accumulating
    Tone *filter;        // bore filtering.
    Noise *noise;
    long length;
    FLOAT outputGain;
    FLOAT noiseGain;

    WhistleBore *bore;

    FLOAT tuneDel;
    Delay *tuneBore;
    void start(FLOAT lowestFreq);

  public:
    CWclariHol(FLOAT lowestFreq);
    CWclariHol();
    ~CWclariHol();
    void clear();
    virtual void setFreq(FLOAT frequency);
    virtual FLOAT newSamp(FLOAT mic);

};

#endif
```

H.5.8 CWclariHol.c

```
/*
 * CWclariHol.c
 *
 * Single reed model after Smith.
 * 6-tone hole bore modelled with simple 2-port junctions.
 *
 */

#include "CWclariHol.h"

CWclariHol :: CWclariHol()
{
    start(500);
}

CWclariHol :: CWclariHol(FLOAT f)
{
    start(f);
}

void CWclariHol :: start(FLOAT lowestFreq)
{
    FLOAT boreTime = 1/lowestFreq;

    reedTable = new ReedTabl;
    reedTable->setOffset(.6);
    reedTable->setSlope(-0.08);
//    filter = new OneZero;
//    filter->setCoeff(2);
    noise = new Noise;
    outputGain = 1000.0;
    noiseGain = 0.1;

//  tuneBore = new Delay(1);
//    tuneDel = 100;    // For tuning. In samples.
//    tuneBore->setDelaySamps(tuneDel);

    filter = new Tone(15000);    // Main filter, at mouth end.
    bore = new WhistleBore( .12*boreTime,
                            .01*boreTime,
                            .01*boreTime,
                            .01*boreTime,
                            .01*boreTime,
                            .01*boreTime,
                            .01*boreTime);
    bore->setAtten(.99);

    setFingerPos(0, .5);    // Makes finger range work over 1/2 the maximum distance.

    setSmoothBreathFreq(6);
    setBreathEdgeResponseMax(2.5);
    setBreathEdgeResponseWindow(.05);
}

CWclariHol :: ˜CWclariHol()
{
    delete reedTable;
    delete filter;
    delete noise;
    delete bore;

}


void CWclariHol :: clear()
{
    filter->newSamp(0.0);

    bore->clear();
}

void CWclariHol :: setFreq(FLOAT frequency)
{
```

```
//    delay1->setDelayTime(.5 / frequency);
}



FLOAT CWclariHol :: newSamp(FLOAT mic)
{
    FLOAT pressureDiff;
    FLOAT breathPressure;


//// Simple waveguide reed after Smith,
//// based on model by McIntryre,Schumacher,Woodhouse

    breathPressure = breathNewSamp(mic)*7;
    breathPressure += breathPressure *
            noiseGain * noise->newSamp();
//            + growlNewSamp(mic);            // Maybe useful in some other models.

// For tuning only.
/*
    if (finger[0]->newSamp() > .8) {
        tuneDel *= 1.00001;
        tuneBore->setDelaySamps(tuneDel);
    }
    if (finger[1]->newSamp() > .8) {
        tuneDel *= .99999;
        tuneBore->setDelaySamps(tuneDel);
    }
    if (finger[2]->newSamp() >.8) {
        printf("%f\n", tuneDel);
    }

    pressureDiff = boreFilt1->newSamp(-1 *tuneBore->getLastOut()) - breathPressure;
    tuneBore->newSamp(breathPressure +
        pressureDiff * reedTable->lookup(pressureDiff));
*/

    bore->setFingers(    finger[0]->newSamp(),
                        finger[1]->newSamp(),
                        finger[2]->newSamp(),
                        finger[3]->newSamp(),
                        finger[4]->newSamp(),
                        finger[5]->newSamp());

    pressureDiff = filter->newSamp(bore->newSamp()) - breathPressure;

    bore->setIn(breathPressure +
        pressureDiff * reedTable->lookup(pressureDiff));

    lastOutputL = bore->getMicOutL()*outputGain;
    lastOutputR = bore->getMicOutR()*outputGain;

    return(lastOutputL+lastOutputR);    // Mono output


//printf("%f\n", lastOutput);
//    lastOutput *= breathPressure*outputGain;

}
```

H.5.9 CWclariHolP.h

```
/*
 * CWclariHolP.h
 *
 * Single reed model after Smith.
 * The bore is modelled using parallel delays.
 *
 */


#if !defined(__CWclariHolP_h)
#define __CWclariHolP_h
```

```
#include <stdio.h>
#include "global.h"
#include "CWinstr.h"

#include "ReedTabl.h"
#include "OneZero.h"
#include "Noise.h"

#include "Tone.h"
#include "WhistleBoreP.h"

class CWclariHolP : public CWinstr
{
  protected:
    ReedTabl *reedTable;
//    OneZero *filter;    // Alternative filters for accumulating
    Tone *filter;        // bore filtering.
    Noise *noise;
    long length;
    FLOAT outputGain;
    FLOAT noiseGain;

    WhistleBoreP *bore;

    FLOAT tuneDel;
    Delay *tuneBore;
    void start(FLOAT lowestFreq);

  public:
    CWclariHolP(FLOAT lowestFreq);
    CWclariHolP();
    ~CWclariHolP();
    void clear();
    virtual void setFreq(FLOAT frequency);
    virtual FLOAT newSamp(FLOAT mic);

};


#endif
```

H.5.10 CWclariHolP.c

```
/*
 * CWclariHolP.c
 *
 * Single reed model after Smith.
 * The bore is modelled using parallel delays.
 *
 */


#include "CWclariHolP.h"

CWclariHolP :: CWclariHolP()
{
    start(400);
}

CWclariHolP :: CWclariHolP(FLOAT f)
{
    start(f);
}

void CWclariHolP :: start(FLOAT lowestFreq)
{
    FLOAT boreTime = 1/lowestFreq;

    reedTable = new ReedTabl;
    reedTable->setOffset(.6);
    reedTable->setSlope(-0.08);
//    filter = new OneZero;
//    filter->setCoeff(1.0);
    filter = new Tone(1000);    // Main filter, at mouth end.
    noise = new Noise;
    outputGain = 1000.0;
    noiseGain = 0.02;

//  tuneBore = new Delay(1);
    tuneDel = boreTime;
//    tuneBore->setDelaySamps(tuneDel);

    bore = new WhistleBoreP( boreTime,  // Use with AUDIO_RATE=32000
                            .000846,
                            .001471,
                            .001999,
                            .003509,
                            .003803,
                            .004476
                            );
/*
    bore = new WhistleBoreP( (long)400,
                            (long)200,
                            (long)100,
                            (long)300,
                            (long)50,
                            (long)25,
                            (long)25
                            );
*/
    setFingerPos(0, .5);    // Makes finger range work over 1/2 the maximum distance.
                            // Use -ve val instead of zero to utilize the
                            // 'Slow range' of LDR when finger is fully closed.

    setFingerVals(0, 1);    // Finger down -> zero delay mix in WhistleBoreP.
                            // Finger up -> <1 delay mix.

                            // How to set a single finger:

    setFingerVals(0, 0, .4);
/*
    setFingerVals(1, 0, 0);
    setFingerVals(2, 0, 0);
    setFingerVals(3, 0, 0);
    setFingerVals(4, 0, 0);
    setFingerVals(5, 0, 0);
*/
    setSmoothBreathFreq(6);
    setBreathEdgeResponseMax(2.5);
    setBreathEdgeResponseWindow(.05);
}

CWclariHolP :: ~CWclariHolP()
{
    delete reedTable;
    delete filter;
    delete noise;
    delete bore;

}


void CWclariHolP :: clear()
{
    filter->newSamp(0.0);

    bore->clear();
}

void CWclariHolP :: setFreq(FLOAT frequency)
{
//    delay1->setDelayTime(.5 / frequency);
}



FLOAT CWclariHolP :: newSamp(FLOAT mic)
{
    FLOAT pressureDiff;
    FLOAT breathPressure;


//// Simple waveguide reed after Smith,
//// based on model by McIntryre,Schumacher,Woodhouse

    breathPressure = breathNewSamp(mic)*7;
    breathPressure += breathPressure *
            noiseGain * noise->newSamp();
//            + growlNewSamp(mic);            // Maybe useful in some other models.

// For tuning only.

    if (finger[5]->newSamp() > .8) {
        tuneDel *= 1.00001;
        bore->setDelay(0, tuneDel);
    }
    if (finger[4]->newSamp() > .8) {
        tuneDel *= .99999;
        bore->setDelay(0, tuneDel);
    }
    if (finger[3]->newSamp() >.8) {
        printf("%f\n", tuneDel);
    }

    bore->setMixes(        finger[0]->newSamp(),
                        finger[1]->newSamp(),
                        finger[2]->newSamp(), 0, 0, 0);

/*
    bore->setMixes(        finger[0]->newSamp(),
                        finger[1]->newSamp(),
                        finger[2]->newSamp(),
                        finger[3]->newSamp(),
                        finger[4]->newSamp(),
                        finger[5]->newSamp());
*/
    pressureDiff = .99* filter->newSamp(bore->newSamp()) - breathPressure;

    bore->setIn(breathPressure +
        pressureDiff * reedTable->lookup(pressureDiff));

    lastOutputL = bore->getMicOutL()*outputGain;
    lastOutputR = bore->getMicOutR()*outputGain;
    lastOutput    = lastOutputL + lastOutputR;

//printf("%f\n", bore->getMicOutL());

    return(lastOutput);    // Mono output


//printf("%f\n", lastOutput);
//    lastOutput *= breathPressure*outputGain;

}
```

H.5.11 CWfluHol.h

```
/*
 * CWfluHol.h
 *
 * Flute model after Cook.
 * 6-tonehole bore modelled with 2-port scattering junctions.
 *
 */


#if !defined(__CWfluHol_h)
#define __CWfluHol_h


#include <stdio.h>
#include "global.h"
```

```
#include "CWinstr.h"

#include "FluteMouth.h"
#include "OnePole.h"

#include "Delay.h"
#include "WhistleBore.h"

class CWfluHol : public CWinstr
{
  protected:
    FluteMouth *mouth;
    OnePole *filter;    // Alternative filters for accumulating
//    Tone *filter;        // bore filtering.
    long length;
    FLOAT outputGain;
    FLOAT noiseGain;

    WhistleBore *bore;

    FLOAT tuneDel;
    Delay *tuneBore;
    void start(FLOAT lowestFreq);

  public:
    CWfluHol(FLOAT lowestFreq);
    CWfluHol();
    ~CWfluHol();
    void clear();
    virtual void setFreq(FLOAT frequency);
    virtual FLOAT newSamp(FLOAT mic);

};

#endif
```

H.5.12 CWfluHol.c

```
/*
 * CWfluHol.c
 *
 * Flute model after Cook.
 * 6-tonehole bore modelled with 2-port scattering junctions.
 *
 */


#include "CWfluHol.h"

CWfluHol :: CWfluHol()
{
    start(1000);
}

CWfluHol :: CWfluHol(FLOAT f)
{
    start(f);
}

void CWfluHol :: start(FLOAT lowestFreq)
{
    FLOAT boreTime = 1.5/lowestFreq;

    mouth = new FluteMouth(boreTime);        // Set limit on jet delay.
    mouth->setNoiseGain(.05);
    mouth->setJetRefl(.5);
    mouth->setEndRefl(.5);
    mouth->setJetTime(.2 * boreTime);        // Jet delay should be 0-0.5 * bore delay.

    filter = new OnePole;
    filter->setPole(0.7 - (0.1 * 22050.0 / AUDIO_RATE));
    filter->setGain(1.0);

    outputGain = 4000.0;
```

```
//  tuneBore = new Delay(1);
//    tuneDel = 100;    // For tuning. In samples.
//    tuneBore->setDelaySamps(tuneDel);

//    filter = new Tone(15000);    // Main filter, at mouth end.
    bore = new WhistleBore( .5*boreTime,    // Use with 22050 Hz sample rate
                            .1*boreTime,
                            .1*boreTime,
                            .1*boreTime,
                            .1*boreTime,
                            .1*boreTime,
                            .1*boreTime);
    bore->setAtten(.99);

    setFingerPos(0, .5);    // Makes finger range work over 1/2 the maximum distance.

    setFingerVals(0, .05, .3);    // When using finger0 as jet delay control.

    setSmoothBreathFreq(6);
    setBreathEdgeResponseMax(2.5);
    setBreathEdgeResponseWindow(.05);
}

CWfluHol :: ˜CWfluHol()
{
    delete mouth;
    delete filter;
    delete bore;
}


void CWfluHol :: clear()
{
    filter->newSamp(0.0);

    bore->clear();
}

void CWfluHol :: setFreq(FLOAT frequency)
{
//    delay1->setDelayTime(.5 / frequency);
}



FLOAT CWfluHol :: newSamp(FLOAT mic)
{

// For tuning only.
/*
    if (finger[0]->newSamp() > .8) {
        tuneDel *= 1.00001;
        tuneBore->setDelaySamps(tuneDel);
    }
    if (finger[1]->newSamp() > .8) {
        tuneDel *= .99999;
        tuneBore->setDelaySamps(tuneDel);
    }
    if (finger[2]->newSamp() >.8) {
        printf("%f\n", tuneDel);
    }

*/

    bore->setFingers(    1,    // Always closed
                        finger[1]->newSamp(),
                        finger[2]->newSamp(),
                        finger[3]->newSamp(),
                        finger[4]->newSamp(),
                        finger[5]->newSamp());

    // Do all waveguide stuff:
    // Bore inverts the signal, filter dosen’t

    // Breath increase -> overblowing.
//    mouth->setJetRatio(1/(1+2*breath));    // Assuming breath goes upto 1.

    mouth->setJetRatio(finger[0]->newSamp());

    bore->newSamp( mouth->newSamp(filter->newSamp(bore->getLastOut()),
                breathNewSamp(mic)*.5 ));

    lastOutputL = bore->getMicOutL()*outputGain;
    lastOutputR = bore->getMicOutR()*outputGain;

    return(lastOutputL+lastOutputR);    // Mono output

}
```

H.5.13 CWmassBow.h

```
/*
 * CWmassBow.h
 *
 * Waveguide network with a bowed mass.
 * Fingers control parallel delay feedback on one side of the mass.
 *
 */

#if !defined(__CWmassBow_h)
#define __CWmassBow_h


#include <stdio.h>
#include "global.h"
#include "CWinstr.h"

#include "OnePole.h"
#include "Tone.h"
#include "Delay.h"
#include "WhistleBoreP.h"
#include "MassBow.h"

class CWmassBow : public CWinstr
{
  protected:
    FLOAT outputGain;
    FLOAT tuneDel;
    Delay *tuneBore;
//    Tone *filter;
    OnePole *filter;
    WhistleBoreP *neck;
    Delay *bridge;
    MassBow *bowPoint;
    void start(FLOAT lowestFreq);

  public:
    CWmassBow(FLOAT lowestFreq);
    CWmassBow();
    ~CWmassBow();
    void clear();
    void setFreq(FLOAT frequency);
    FLOAT newSamp(FLOAT mic);

};

#endif
```

H.5.14 CWmassBow.c

```
/*
 * CWmassBow.c
 *
 * Waveguide network with a bowed mass.
 * Fingers control parallel delay feedback on one side of the mass.
 *
 */
```

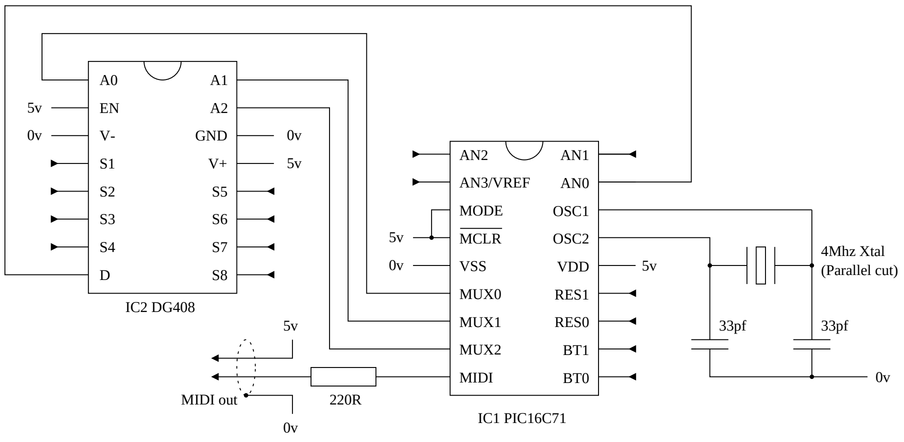
```
#include "CWmassBow.h"

CWmassBow :: CWmassBow()
{
    start(1000);
}

CWmassBow :: CWmassBow(FLOAT f)
{
    start(f);
}

void CWmassBow :: start(FLOAT lowestFreq)
{
    FLOAT boreTime = 1/lowestFreq;


    filter = new OnePole;
    filter->setPole(0.7 - (0.1 * 22050.0 / AUDIO_RATE));
    filter->setGain(1.0);

//    filter = new Tone(15000);    // Main filter, at mouth end.

    outputGain = 4000.0;

    neck = new WhistleBoreP( boreTime,
                            .3*boreTime,
                            .4*boreTime,
                            .5*boreTime,
                            .6*boreTime,
                            .7*boreTime,
                            .8*boreTime);

    bridge = new Delay(boreTime*.3);

    bowPoint = new MassBow();


    setFingerPos(0, .5);    // Makes finger range work over 1/2 the maximum distance.


    setSmoothBreathFreq(6);
    setBreathEdgeResponseMax(2.5);
    setBreathEdgeResponseWindow(.05);
}

CWmassBow :: ~CWmassBow()
{
    delete filter;
}


void CWmassBow :: clear()
{
    filter->newSamp(0.0);
}



FLOAT CWmassBow :: newSamp(FLOAT mic)
{

    neck->setFingers(    finger[0]->newSamp(),
                        finger[1]->newSamp(),
                        finger[2]->newSamp(),
                        finger[3]->newSamp(),
                        finger[4]->newSamp(),
                        finger[5]->newSamp()    );

    bowPoint->setBowVel(breathNewSamp(mic));

    // Do all waveguide stuff:

    bowPoint->setInL(-bridge->newSamp(bowPoint->getOutL()));
    bowPoint->setInR(-filter->newSamp(neck->newSamp(bowPoint->getOutR())));

    bowPoint->newSamp();


    lastOutputL = neck->getMicOutL()*outputGain;
    lastOutputR = neck->getMicOutR()*outputGain;

    return(lastOutputL+lastOutputR);    // Mono output

}
```

Appendix I

Details of the Microcontroller Implementation for the CyberWhistle

I.1 The Schematic

Figure I.1 shows how the Microchip PIC16C71 is used with a 8-1 muliplexer. The box tables show the effect of different pin settings.



RES1	RES0	MIDI Bits
L	L	4
L	H	5
H	L	6
H	H	7

For highest accuracy set RES0=RES1=H

BT1	BT0	Pause (ms)
L	L	0.6
L	H	2.5
H	L	5.1
H	H	10.1

For best response time set BT0 = BT1 = L

Figure I.1: Voltage to MIDI conversion.

I.2 The Software

The following listing is the assembler code required to give the microcontroller the functionality indicated in **Figure I.1**.

```
;;;;
;;;; a2m.asm
;;;;
;;;; Multiple analog channel to MIDI controller converter,
;;;; to be used with an 8-1 multiplexer.
;;;; Includes selectable midi data resolution.
;;;; + MIDI message rate limiting.
;;;;
;;;; Target: Microchip PIC16C71.
;;;;
;;;; Usage: Refer to schematic.
;;;;
;;;;
;;;;
;;;;



;;      User ram file addresses

bitCount        equu    0x0c    ; For counting out midi bits.
midiShiftReg    equ     0x0d    ; For shifting out midi bits to serial output.
pauseCount      equ     0x0e    ; Pause counter
ADsamp          equ     0x0f    ; Last sample read.
ADoldSampBase   equ     0x10    ; Record of old samples on upto 11 channels.
ADchannel       equ     0x20    ; Current input channel, used to set external MUX
midiPauseMult   equ     0x21    ; Multiplier for the long inter-message pauses.
statusCount     equ     0x22    ; For counting messages between status bytes.
resMask         equ     0x23    ; Mask for limiting resolution.
resSelect       equ     0x24    ; Workspace for calculating resMask.
restTime        equ     0x25    ; Limits the midi activity to suit receiver.
temp            equ     0x26    ; Temporaray space.
ANx             equ     0x27    ; PIC analog input count.
ANxMax          equ     0x28    ; " " limit. 3 for mode 0, 2 for mode 1.
AN0Max          equ     0x29    ; Nu of ADchannels for AN0, ie 1 for
                                ; mode 0, 8 for mode 1.

;;      Symbolic constants

midiOutBit              equ     3
controlStatusCode       equ     0xB0    ; Channel 1

        org 0
        goto start

        org 4
        retfie

start
        call    initIO
        call    initAD
        call    initRestTime
        call    initResolution
        call    initMode

initStatus
        clrf    statusCount
        movlw   controlStatusCode       ; Send controller status byte, channel 1
        call    midiByteOut

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;; Main Loop

newSampRound
        clrf    ADchannel
        clrf    ANx
        movlw   b’01000001’     ; Configure AD for ANx = 0;
        movwf   ADCON0
```

```
setMUX
        movf    ADchannel, W
        iorlw   0x08            ; Ensure midiOutBit is kept high between messages.
        movwf   PORTB           ; Set external MUX

aquireMUX
        movlw   .9              ; MUX/AD aquisition time. about 30us.
        call    pause

        call    convertThenMidi

        incf    ADchannel, F
        movf    AN0Max, W
        subwf   ADchannel, W
        btfss   STATUS, Z
        goto    setMUX

;------------------------------------

        incf    ANx, F          ; More for clarity: to make ANxMax
setANx
        movf    ANx, W
        movwf   temp
        rlf     temp, F
        rlf     temp, F
        rlf     temp, W
        iorlw   b'01000001'
        movwf   ADCON0          ; Configure AD for ANx

aquireANx
        movlw   .9              ; MUX/AD aquisition time. about 30us.
        call    pause

        call    convertThenMidi

        incf    ADchannel, F
        incf    ANx, F
        movf    ANx, W
        subwf   ANxMax, W
        btfss   STATUS, Z
        goto    setANx

        goto    newSampRound




;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


convertThenMidi
        bcf     ADCON0, ADIF    ; Start AD conversion
        bsf     ADCON0, GO

ADloop
        btfsc   ADCON0,GO       ; Wait for AD conversion to complete. about 20 us.
        goto    ADloop

ADcomplete
        nop
        movf    ADRES,  W
        bcf     STATUS, C
        rrf     ADsamp          ; Limit ADsamp to 7 bit MIDI resolution.
        movf    resMask, W
        andwf   ADsamp, F

        movf    ADchannel, W
        addlw   ADoldSampBase
        movwf   FSR
        movf    ADsamp, W
        subwf   INDF            ; Subtract ADsamp from old value.
        btfss   STATUS, Z       ; If new samp if different from old then send midi.
        call    sendControlMessage
messageEnd
        movf    ADsamp, W
        movwf   INDF            ; Update old sample value.
        return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


sendControlMessage
        movlw   0x08
        addwf   statusCount, F  ; Send status byte every 32 messages.
        btfss   STATUS, Z
        goto next

sendStatus
        movlw   controlStatusCode       ; Send controller status byte, channel 1
        call    midiByteOut

next    movf    ADchannel, W    ; Controller number.
        call    midiByteOut
        movf    ADsamp, W       ; Value of controller.
        call    midiByteOut

restTest
        movf    restTime, W
        btfsc   STATUS, Z
        return

        movwf   temp
        movlw   .179    ; Unit delays are about 600 us + ADC delays
                        ; = about 640 us = time spent sending a 2 byte midi message.

restLoop
        call pause
        decfsz  temp
        goto    restLoop
        nop
        nop
        return


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

midiByteOut     ;; Each bit lasts exactly 32us.

        movwf   midiShiftReg

        movlw   8
        movwf   bitCount


midiStartBit
        bcf     PORTB, midiOutBit
        movlw   6
        call    pause
        nop
        nop

bitLoop
        rrf     midiShiftReg, F
        clrf    temp
        rlf     temp, F
        rlf     temp, F
        rlf     temp, W                 ; midiOutBit is now set correctly in W.

midiDataBit
        movwf   PORTB

        movlw   5
        call    pause
        nop
        nop
        decfsz  bitCount, F
        goto    bitLoop

        nop
        nop
        nop
        nop
        nop
        nop
        nop

midiStopBit
        bsf     PORTB, midiOutBit
        movlw   6
        call    pause
midiByteOutEnd
        return



pause   ; cycle delay = 3+3*W
        movwf   pauseCount
pauseLoop
        decfsz  pauseCount, F
        goto pauseLoop
pauseEnd
        return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

initIO
        clrf    PORTB
        bsf     STATUS,RP0      ; Select Bank 1
        movlw   b'11110000'     ; (Input : Output)
        movwf   TRISB
        bcf     OPTION, RPBU    ; Weak pull up n inputs
        bcf     STATUS, RP0     ; Select Bank 0
        bsf     PORTB,  midiOutBit      ; Inter message level.
        return

initAD
        bsf     STATUS,RP0      ; Select Bank 1
        movlw   0
        movwf   ADCON1          ; No Vref input on pin RA3
        bcf     STATUS,RP0      ; Select Bank 0

        movlw   b'01000001'     ; Configure AD : XTL clock /8 for low noise.
        movwf   ADCON0
        clrf    ADRES
        return

initRestTime
        movlw   .1
        movwf   restTime
        swapf   PORTB, W
        andlw   0x03
        movwf   temp
        btfsc   STATUS,Z
        return          ; Input = 0 so return restTime = 1

        movlw   .4
        movwf   restTime
        decf    temp
        btfsc   STATUS,Z
        return          ; Input = 1 so return restTime = 4

        movlw   .8
        movwf   restTime
        decf    temp
        btfsc   STATUS,Z
        return          ; Input = 2 so return restTime = 8

        movlw   .16
        movwf   restTime
        return          ; Input = 3 so return restTime = 16


initResolution
        movf    PORTB, W
        andlw   0xc0
        movwf   resSelect
        movlw   0xff
        movwf   resMask
        bsf     STATUS, C       ; First time in loop don't add any zeros.
        movlw   0x40            ; Loop increment

initResLoop
        rlf     resMask
        addwf   resSelect
        bcf     STATUS, C       ; Ready C for next rlf.
        btfss   STATUS, Z       ; Finish when resMask contains right mask.
        goto    initResLoop

; Effect:
;       resSelect       resMask
;       11xxxxxx        11111111
;       10xxxxxx        11111110
;       01xxxxxx        11111100
;       00xxxxxx        11111000

        return



initMode
        ;;; Mode 0 settings:

        bsf     STATUS,RP0
        movlw   0
        movwf   ADCON1          ; No Vref input on pin RA3
        bcf     STATUS,RP0

        movlw   .1              ; AN0 is not multiplexed in this mode.
        movwf   AN0Max
        movlw   .4
        movwf   ANxMax


        btfss   PORTA, 4        ; Test mode pin, RA4
        return

        ;;; Mode 1 settings:

        bsf     STATUS,RP0
        movlw   1
        movwf   ADCON1          ; No Vref input on pin RA3
        bcf     STATUS,RP0

        movlw   .8              ; AN0 is multiplexed in this mode.
        movwf   AN0Max
        movlw   .3              ; AN4 is used as Vref.
        movwf   ANxMax
        return

        end
```

Appendix J

CD Track Listing

Track 1-13 : Spiragraph examples. See Section 5.3.

Single key press **Spiragraph 1**

1. Each key pressed seperately.
2. Increasing key strike velocity for two keys.
3. Increasing Mod Wheel for two keys.
4. Increasing Pitch Wheel for two keys.

Single key press **Spiragraph 2**

5. Key with just amplitude modulation.
6. Amplitude and frequency modulation.

Multiple simultaneous key presses **Spiragraph 1**

7. Simple key, then freq envelope key, then combined.
8. Phasing. Simple key, then similar key, then combined.
9. Unpredicatability. One key held, another repeated.
10. Complex phasing. Several keys held.
11. Performance. Several keys controlled dynamically.

Multiple simultaneous key presses **Spiragraph 2**

12. Complex stereo phasing. Several keys held.
13. Performance. Added stereo complexity.

Track 14 : *Lifeforms*, Spiragraph piece. See Section 5.3.4.

Tracks 15-16 : SidRat excerpts. See Section 7.

15. First Movement, first half, computer part only.
16. Second Movement, simulated live performance.

Tracks 17-44 : The CyberWhistle. See Section 8.4.7.

Test instruments

17. Sine wave instrument, **CWsines**
18. As above, with reverberation added.
19. Square wave instrument, **CWtestSines**
20. As above, with reverberation added.

Using instrument class **CWclariHol**

21. Highly tuned single reed instrument.
22. As above, with reverberation added.
23. High and breathy.
24. As above, with reverberation added.
25. Bass register, mellow.
26. As above, with reverberation added.
27. Bass register, bright.
28. As above, with reverberation added.
29. Lower, bright.
30. As above, with reverberation added.
31. Very low, with high pitched harmonic and aharmonic effects.
32. As above, with reverberation added.

Using instrument class **CWclariHolP**

33. High, breathy, some distortion. Good pitch slides.
34. As above, with reverberation added.
35. Mid pitched, bright. High harmonics can be played easily.
36. As above, with reverberation added.
37. Very low with chaotic burbling sounds.
38. As above, with reverberation added.
39. Very low, bright. Screaming quality.
40. As above, with reverberation added.

Using instrument class **CWfluHol**

41. Recognizable flute.
42. As above, with reverberation added.
43. Very low, harmonic effects, percussive breath start noise.
44. As above, with reverberation added.

Appendix K

Publications and Performances

Publications

Bourges Synthese, June 1997 *LAmb, an Introduction and Tutorial*
(Subsequent invitation to submit a paper for The Computer Music Journal was passed over due to failed communication.)

Colloquium on Musical Informatics XII, Sept 1998
The CyberWhistle, a New Performance Instrument

Awards

Bourges International Software Competition, 1997, winner of real-time / gestural control category with *LAmb*.

UK Patent Application

In name: *University of York*
Title: *Electronic Musical Instruments*
No: 98 20650.1
Filed: 23-Sept-98
Concerns the work relating to the *CyberWhistle*

Relevant Performances

Ambisonix, Vanbrugh Hall, University of York, April 1996.
LAmb used for live diffusion of all music
including taped and live performance.

The Hex, University of York Music Dept, May 97.
Performance of *SidRat* and two pieces which use *LAmb*:
Spherical Construction by John Richards and
What a Difference a Day Makes by Tim Ward.

Bibliography

[And94] Craig Anderton. Steim: In the land of alternate controllers. *Keyboard*, pages 55–62, August 1994.

[App86] J Appleton. The computer and live musical performance. In *Proceedings of the International Computer Music Conference*, page 321, 1986.

[BC93] G Bertini and P Carosi. Light baton system: A system for conducting computer music performance. *Interface*, 22:243–257, 1993.

[Beh96] Gerhard Behles. Directory of /pub/granular. *ftp://vader.kgw.tu-berlin.de/pub/granular*, 11 September 1996.

[Ber73] B Bernfeld. Attempts for better understanding of the directional stereophonic listening mechanism. In *Preprint, Audio Engineering Society 44th Convention, Rotterdam*, 1973.

[CE88] PJ Comerford and BM Eaglestone. Bradford musical instrument simulator and workstation. *Microprocessing and Microprogramming*, 24:555–562, 88.

[Cha90] X Chabot. Gesture interfaces and a software toolkit for performance with electronics. *Computer Music Journal*, 14(2):15–27, 1990.

[Cho73] JM Chowning. The synthesis of complex audio spectra by means of frequency modulation. *J. Audio Eng. Soc.*, 21:526–534, 1973.

[CLF90] Claude Cadoz, Leszek Lisowski, and Jean-Loup Florens. Modular feedback keyboard. In *Proceedings of the International Computer Music Conference*, pages 379–382, 1990.

[CLF93] Claude Cadoz, Annie Luciani, and Jean Loup Florens. Cordis-anima: A modeling system for sound and image synthesis, the general formalism. *Computer Music Journal*, 17(1):19–29, 1993.

[Coo92] Perry R. Cook. A meta wind instrument physical model and a meta controller for real time performance. In *Proceedings of the International Computer Music Conference*, pages 273–276, 1992.

[Dan84] R Dannenberg. An on-lin algorithm for real-time accompianment. In *Proceedings of the International Computer Music Conference*, page 187, 1984.

[DJ85] Charles Dodge and Thomas A. Jerse. *Computer music, synthesis, composition and performance*. New York, Schirmer books, 1985.

[Emm91] Simon Emmerson. Computers and live electronic music - some solutions, many problems. In *Proceedings of the International Computer Music Conference*, pages 135–138, 1991.

[Fav94] Stuart Favilla. The ldr controller. In *Proceedings of the International Computer Music Conference*, pages 177–180, 1994.

[Fle91] NH Fletcher. *The physics of musical instruments*. Springer, 1991.

[Fre89] J Frederickson. Technology and performance in the age of mechanical reproduction. *Some Journal*, pages 193–220, 1989.

[GB92] Michael A. Gerzon and Geoffrey J. Barton. Ambisonic decoders for hdtv. In *Preprint 3345 of the 92nd Audio Engineering Society Convention*, 1992.

[Ger75a] M Gerzon. *NRDC Ambisonic Technology Report 3, Pan Pot and Sound Field Controls*. 1975.

[Ger75b] M Gerzon. *NRDC Ambisonic Technology Report 4, Artificial Reverberation and Spreader Devices*. 1975.

[Ger80] M Gerzon. Practical periphony: The reproduction of full-sphere sound. In *Draft of lecture presented at the Audio Engineering Society 65th Convention*, 1980.

[Ger92] M Gerzon. General metatheory of auditory localisation. In *92nd Audio Engineering Society Convention, Vienna*, 1992.

[Ger95] M Gerzon. Audio signal processor providing simulated source distance control. *British Patent 05555306*, 1995.

[HBH93] A Horner, J Beauchamp, and H Haken. Methods for multiple wavetable synthesis of music instrument tones. *Journal of the Audio Engineering Society*, 41(5):336–356, 1993.

[Hei72] Seppo Heikinheimo. *The Electronic Music of Karlheinz Stockhausen*. Acta Musicologica Fennica 6, 1972.

[HSK] Jyri Huopanierni, Lauri Savioja, and Matti Karjalainen. Modelling of reflections and air absorption.

[Kee82] Douglas H. Keefe. Theory of the single woodwind tonehole. *Acoustical Society of America*, 72(3):676–687, 1982.

[KL90] RB Knapp and HS Lusted. A bioelectric controller for computer music applications. *Computer Music Journal*, 14(1):42–47, 1990.

[Kre90] Volker Krefeld. The hand in the web: An interview with michel waisvisz. *Computer Music Journal*, 14(2):28–33, 1990.

[KWT78] Kaegi, Werner, and Tempelaars. Vosim - a new sound synthesis system. *Journal of the Audio Engineering Society*, 26(6):418–425, 1978.

[Lea59] D.M. Leakey. Some measurements on the effect of interchannel intensity and time differences in two-channel sound systems. *J. Acous. Soc. Am.*, 31:977–987, 1959.

[Lev94] T Levenson. Taming the hypercello. *The Sciences*, page 15, July/August 1994.

[MA93] Joseph Derek Morrison and Jean-Marie Adrien. Mosaic: a framework for modal synthesis. *Computer Music Journal*, 17(1):45–56, 1993.

[Mak62] Y Makita. On the directional localization of sound in the stereophonic sound field. *E.B.U Review*, A(73):102–108, 1962.

[Man85] PD Manning. *Electronics and Computer Music*. Oxford University Press, 1985.

[Mat63] MV Mathews. The digital computer as a musical instrument. *Science*, 142(11):553–557, 1963.

[Mat69] MV Mathews. *The Technology of Computer Music*. MIT Press, 1969.

[Mat91] MV Mathews. The radio baton and conductor program. *Computer Music Journal*, 15(4):37, 1991.

[Men95] D Menzies. An investigation into the design of musical performance instruments. Master's thesis, Dept. of Electronics, University of York, England, 1995.

[MSJ83] M. E. Mcintyre, R.T. Schmacher, and Woodhouse J. On the oscillations of musical instruments. *Journal of the Acoustical Society of America*, 75(5):1325–45, 1983.

[OH84] AWJG Ord-Hume. *Pianola*. George Allen and Unwin, 1984.

[Pea96] M Pearson. *Synthesis of Organic Sounds for Electroacoustic Music*. PhD thesis, Dept. of Electronics, University of York, England, 1996.

[PG97] JA Paradiso and N Gershenfeld. Musical applications of electric field sensing. *Computer Music Journal*, 21(2):69–89, 1997.

[PJS+98] Nick Porcaro, David Jaffe, Pat Scandalis, Julius Smith, Tim Stilson, and Scott Van Duyne. Synthbuilder:a graphical rapid-prototyping tool for the development of music synthesis and effects patches on multiple platforms. *Computer Music Journal*, 22(2):35–43, 1998.

[Pop93] ST Pope. Real-time performance via user interfaces to musical structures. *Interface*, 22:195–212, 1993.

[Puc86] M Puckette. The patcher. In *Proceedings of the International Computer Music Conference*, page 420, 1986.

[Ris90] JC Risset. From piano to computer to piano. In *Proceedings of the International Computer Music Conference*, pages 15–18, 1990.

[Ris94] JC Risset. About the role of computers to bridge gaps in music. In *Proceedings of the 13th World Computer Congress*, volume 2, 1994.

[Rod79] X Rodet. Time-domain formant-wave-functions synthesis. *Actes du NATO-ASI Bonas*, July 1979.

[RSA+95] Curtis Rhodes, John Strawn, Curtis Abbott, John Gordan, and Philip Greenspun. *The Computer Music Tutorial*. MIT Press, 1995.

[SC92] Julius O. Smith and Perry R. Cook. The second order digital waveguide oscillator. In *Proceedings of the International Computer Music Conference*, pages 150–153, 1992.

[SD93] WA Schloss and AJ David. Intelligent musical instruments: The future of musical performance or the demise of the performer? *Interface*, 22:183–193, 1993.

[SG84] Stanley Sadie and George Grove. *The New Grove Dictionary of Music and Musicians*. Macmillan, 1984.

[Smi86] Julius O. Smith. Efficient simulation of the reed bore and bow string mechanisms. In *Proceedings of the International Computer Music Conference*, pages 275–280, 1986.

[Smi91] Julius O. Smith. Viewpoints on the history of digital synthesis. In *Proceedings of the International Computer Music Conference*, pages 1–10, 1991.

[Smi93] Julius O. Smith. Efficient synthesis of stringed musical instruments. In *Proceedings of the International Computer Music Conference*, pages 64–71, 1993.

[Smi92] Julius Smith. Physical modelling using digital waveguides. *Computer Music Journal*, 16(4):74–87, 92.

[SW95] RT Schumacher and J Woodhouse. Computer modelling of violin playing. *Contemporary Physics*, 36(2):79–92, 1995.

[TL96] B Tan and S Lim. Automated parameter optimization for double frequency modulation synthesis using the genetic annealing algorithm. *Journal of the Audio Engineering Society*, 44(1/2):3–15, 1996.

[Ver84] B Vercoe. The synthetic performer in the context of live musical performance. In *Proceedings of the International Computer Music Conference*, 1984.

[Ver92] Barry Vercoe. *CSOUND: a manual for the audio processing system*, 1992.

[Ver94] Barry Vercoe. Csound title page. *http://www.leeds.ac.uk/music/Man/Csound/title.html*, November 1994.

[VKL93] Vesa Valimaki, Matti Karjalainen, and Timo Laakso. Modeling of woodwind bores with finger holes. In *Proceedings of the International Computer Music Conference*, pages 32–39, 1993.

[Wat96] Mary K Watanabe. Synthbuilder. *http://ccrma-www.stanford.edu/CCRMA/Software/SynthBuilder/SynthBuilder.html*, 1996.