

**DEVELOPMENT OF A 3D AUDIO PANNING AND
REALTIME VISUALISATION TOOLSET USING EMERGING
TECHNOLOGIES**

PAUL FERGUSON

**A thesis submitted in partial fulfilment of the
requirements of Edinburgh Napier University, for the
award of Doctor of Philosophy**

May 2010

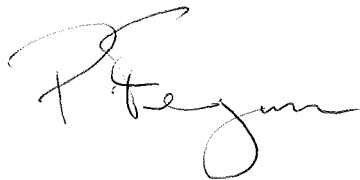
Authorship Declaration

I, Paul Ferguson, confirm that this thesis and the work presented in it are my own achievement.

1. Where I have consulted the published work of others this is always clearly attributed;
2. Where I have quoted from the work of others the source is always given. With the exception of such quotations this thesis is entirely my own work;
3. I have acknowledged all main sources of help;
4. If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;

I have read and understand the penalties associated with plagiarism.

Signed:



Date: 25/5/2010

Abstract

This thesis documents a body of research that links the field of electro-acoustic diffusion and spatialisation with practice in the music and film post-production industries. Three research questions are posed:

“How can the physical user-interfaces used for panning by the music and film post-production industries offer creative alternatives to the fader-based hardware approach commonly used for electro-acoustic performance?”

“How can a Digital Audio Workstation (DAW) be used as an alternative to dedicated panning hardware?”

“How can emerging programming technologies offer creative alternatives to the MAX/MSP or hardware-based tools commonly used for sound spatialisation?”

This practice-based PhD addresses these questions by designing, developing and testing a set of hardware and software tools, the Requirement Specification for this ‘Toolset’ results from literature review and critical analysis of current systems to determine potential research gaps. This analysis is followed by the selection of a suitable methodology for development and testing that allows the research questions to be explored effectively and results in the following Toolset:

OctoPanner: A multi-featured eight-channel 3D touchscreen panner application for Apple’s Mac OS X controlling a DAW-hosted customisable VST 3D panning plug-in with C++ source code.

ShapePanner: A synchronisable shape-based sequencer application for Mac OS X inspired by Experimentalstudio’s Halaphon. The user is able to describe the movement of sounds in a 3D space using shape primitives such as lines and circles and thus extend the capabilities of the Toolset beyond realtime manual manipulation of sounds.

3DMIDI Visualiser: An application to allow the user to work without access to a multi-speaker system by enabling the movement of sounds to be viewed within a virtual room.

Foot Puck: A foot-controlled panning controller enabling a musician to spatialise their instrument using foot movement.

Initial prototyping was achieved using Cycling '74's Max/MSP but the final applications are written using Apple's Cocoa environment and Objective C. This thesis gives close analysis and discussion of the various stages of research carried out; including the use of Apple's **CoreMIDI** and **CoreAudio Clock** OS X Core Services in a Cocoa application.

Table of Contents

Abstract	3
Table of Contents	5
Table of Figures	10
Acknowledgments	12
1 Overview	13
1.1 Background	13
1.2 Research Questions and Context	15
1.3 Thesis Structure.....	16
1.3.1 Part I: Requirements and Methodology	16
1.3.2 Part II: Implementation	16
1.3.3 Part III: Conclusions and contribution to knowledge	17
1.3.4 Appendices	17
Part I: Requirements and Methodology	18
2 Literature review of past diffusion/spatialisation systems with a focus on loudspeaker configuration, control and technology	19
2.1 1951 to 1987: Before MAX.....	19
2.1.1 Four-speaker works and events	19
2.1.2 Six, eight or ten loudspeaker works and events.....	22
2.1.3 Loudspeaker arrays and the Loudspeaker orchestra.....	23
2.2 1988 to present: Post MAX	25
2.2.1 IRCAM and MAX.....	25
2.2.2 CNMAT and Level Control Systems	26
2.3 Chapter summary and research contribution.....	27
3 Currently available institution-based or commercial systems	28
3.1.1 Research Question 1 - Alternative User-Interfaces.....	30
3.1.2 Research Question 2 - DAW Hosting	31
3.1.3 Research Question 3 - Comparing Max/MSP and alternative technologies.....	31
3.1.4 Additional considerations	31
3.1.5 Trajectories, cue-lists and automation	32

3.1.6	Synchronisation.....	32
3.2	Summary of requirements.....	32
3.3	Potential contributions to knowledge.....	35
4	Methodology	36
4.1.1	Waterfall Model	36
4.1.2	Incremental and Iterative Development Model (IID).....	36
4.1.3	Spiral Model	37
4.2	Methodology Conclusion.....	37
4.2.1	Prototype Testing	38
	Part II: Implementation	39
5	Investigating an initial panning solution.....	40
5.1	Introduction	40
5.2	Passive analogue panning.....	41
5.3	Prototype 1: The initial digital panner	41
5.4	Prototype 2 development	43
5.4.1	Background.....	43
5.4.2	Considering Prototype 2 in terms of the IID methodology.....	44
5.5	Joystick development.....	45
5.5.1	Software development for the joystick	46
5.6	Touchscreen development.....	46
5.7	Prototype 2 software	47
5.7.1	Joystick microcontroller software.....	47
5.7.2	Version 2 of the Max/MSP based panner	47
5.7.3	Testing Prototype 2	49
5.8	Chapter Summary and Research Contribution.....	50
6	Development of OctoPanner Version 2	51
6.1	Mac application development.....	51
6.1.1	The Carbon development environment.....	51
6.1.2	The Cocoa development environment.....	51
6.2	OS X Core technologies	52
6.2.1	CoreMIDI.....	53
6.2.2	CoreAudio Clock	53
6.2.3	Core Graphics/OpenGL.....	54

6.3	Cocoa implementation of OctoPanner v2	55
6.3.1	MIDI output from Cocoa	57
6.3.2	Implementing the OctoPanner Version 2 application	58
6.4	Chapter Summary and Research Contribution.....	60
7	The ShapePanner application.....	61
7.1	Introduction	61
7.2	ShapePanner implementation.....	62
7.2.1	Cocoa Bindings	62
7.2.2	The ShapePanner classes	64
7.2.3	Timing/scheduling	64
7.2.4	Representing time	65
7.3	Implementing CoreAudio Clock support.....	66
7.3.1	Creating an accurate scheduler for ShapePanner	67
7.4	Chapter Summary and Research Contribution.....	69
8	3D MIDI Visualiser application	70
8.1	Introduction	70
8.2	Apple's Quartz technology	71
8.3	Chapter Summary and Research Contribution.....	74
9	Foot control of spatialisation.....	75
9.1	Introduction	75
9.2	Foot Puck Prototype	77
9.3	Chapter Research Contribution.....	78
10	Concert use of the version 2 live Toolset.....	79
10.1	Developing a spatialisation technique using Ableton Live	79
10.2	Concert 1 – Haftor Medboe Group, 'The Arches' Glasgow 10-3-2007.....	82
10.2.1	Concert 1 Evaluation	83
10.3	Concert 2 – Haftor Medboe Group, 'The Bongo Club' 17-3-2007	84
10.3.1	Concert 2 Evaluation.....	84
10.4	Chapter research contribution.....	84
10.4.1	Question 1 - Alternative User-Interfaces.....	84
10.4.2	Research Question 2 - DAW Hosting	85
11	OctoPanner Version 3	86

11.1	Saving and loading snapshots	86
11.2	OctoPanner v3 features	87
11.2.1	OctoPanner v3 panels.....	88
11.2.2	Snapshot copying/swapping and initialising.....	89
11.2.3	OctoPanner v3 MIDI setup	89
11.3	Chapter research contribution	90
12	Spatialising <i>Macbenach IV</i>.....	91
12.1	A user trial of Octopanner version 3.....	91
12.2	Pro Tools panner plug-in development using Max/MSP	91
12.3	Reliability and repeatability	94
12.4	Evaluating the Toolset's use for <i>Macbenach IV</i>	94
12.5	Chapter Summary and Research Contribution.....	95
12.5.1	Question 1 - Alternative User Interfaces	95
12.5.2	Research Question 2 - DAW Hosting	95
13	A C++ VST/RTAS panner plug-in	96
13.1	Coding the VST 3D octal panner.....	99
13.2	Guidelines for PFPan plug-in user modification.....	102
13.2.1	Speaker configuration	102
13.2.2	Panning law	103
13.3	Chapter summary and research contribution.....	103
	Part III: Conclusions and contribution to knowledge.....	104
14	Conclusions	105
14.1.1	Research Question 1.....	105
14.1.2	Research Question 2.....	105
14.1.3	Research Question 3.....	106
14.1.4	Summary.....	107
14.2	Contribution to knowledge.....	107
14.3	Limitations and potential for future work.....	108
	Bibliography.....	110
	Appendix A Panning in Pro Tools using MIDI controller emulation.....	121
A.1	JL Cooper CS10.....	121
A.2	CM Labs Motormix	122

A.3 Mackie HUI.....	123
A.3.1 Analysing the HUI protocol.....	123
Appendix B A Cocoa wrapper for Carbon CoreMIDI functions	125
Appendix C Investigating CoreAudio Clock	135
Appendix D PFPan2x4 VST plug-in source code	143

Table of Figures

Fig. 4-1: Boehm Spiral model	37
Fig. 5-1: Passive panner.....	41
Fig. 5-2: Prototype 1 - Craiglockhart panner using Max/Msp.....	42
Fig. 5-3: Pro Tools panner window	43
Fig. 5-4: Prototype 2 multi-channel spatialiser	44
Fig. 5-5: Low-cost joystick viewed from below and above	45
Fig. 5-6: Philips LPC microcontroller-based panner	45
Fig. 5-7: modified Max/MSP simple linear panner.....	47
Fig. 5-8: Six and eight-channel panner	48
Fig. 5-9: Sonic Fusion 2006 Max/MSP panner sub-patch	48
Fig. 5-10: OctoPanner v1 (Sonic Fusion 2006)	49
Fig. 6-1: TestPanner UI and Interface Builder palette	55
Fig. 6-2: Linking controls to corresponding actions	56
Fig. 6-3: OctoPanner v2 main window in Interface Builder.....	59
Fig. 6-4: OctoPanner v2 when running.....	59
Fig. 7-1: ShapePanner prototype interface	61
Fig. 7-2: Bindings test application	63
Fig. 7-3: Text Field and Table Column inspectors showing bindings.....	63
Fig. 7-4: ShapePanner incorporating CoreAudio Clock.....	68
Fig. 8-1: 3D MIDI Visualiser Cocoa application	70
Fig. 8-2: Quartz Composer 3D MIDI Visualiser panel.....	71
Fig. 8-3: Quartz Composer renderer objects in 3D MIDI Visualiser.....	72
Fig. 8-4: 3D Transformation with published inputs	73
Fig. 9-1: Dual-axis foot-controller	75
Fig. 9-2: Ludvig foot controller patent.....	75
Fig. 9-3: Weight distribution based panner evaluation.....	76
Fig. 9-4: 'Puck' based panner evaluation	77
Fig. 9-5: Prototype Foot Puck panner	77
Fig. 10-1: Ableton Live audio track.....	80
Fig. 10-2: OctoPanner v2 modified MIDI Setup panel	81
Fig. 10-3: Ableton return A and return B routing.....	81

Fig. 10-4: Concert layout for 'The Arches' (Glasgow).....	82
Fig. 11-1: OctoPanner version 3 main window.....	87
Fig. 11-2: OctoPanner v3 X,Y,Z mirror controls.....	88
Fig. 11-3: OctoPanner v3 touch keyboard.....	88
Fig. 11-4: OctoPanner v3 snapshot editing panels.....	89
Fig. 11-5: OctoPanner v3 MIDI setup panel.....	90
Fig. 12-1: Pluggo based Pro Tools send and receive plug-ins	92
Fig. 12-2: The send and receive plug-ins in a Pro Tools session	92
Fig. 12-3: Audio section of the Max/MSP (Pluggo) patch for OctoSend plug-in..	93
Fig. 13-1: Senderella V1.08 plug-in in Pro Tools.....	97
Fig. 13-2: PFPan plug-in controls.....	100
Fig. 13-3: PFPan in Ableton Live	101
Fig. 13-4: PFPan in Pro Tools.....	101
Fig. 13-5: PFPan plug-in MIDI mapping	101
Fig. A- 1: CS10 Emulator (Cocoa)	121
Fig. A- 2: HUI Emulator (Cocoa)	124
Fig. B- 1: MIDI port pop-up buttons.....	131
Fig. C- 1: Simple Core Audio Clock test application.....	139
Fig. C- 2: Response by Doug Wyatt to the CAClock anomaly	140
Fig. C- 3: CoreAudio Clock Test App v2	142

Acknowledgments

My sincere thanks go to my supervisor Steve Davismoon and the Postgraduate team at Edinburgh Napier for their guidance and endless encouragement. Grateful thanks to Dave Hook and my other colleagues for their individual contributions and support, especially for taking up the strain when I was at my busiest. Thanks also to Gerard Pape, Haftor Medboe and the other musicians and composers involved in the testing phases. This thesis is dedicated to my family, whose unwavering belief in me made it possible.

1 Overview

1.1 Background

This thesis documents the PhD by Practice-led research that began as a result of a series of electro-acoustic masterclasses and performances in 2004 by staff and visitors to Edinburgh Napier University. The author was technical manager for these events, which included Edinburgh Napier's Steve Davismoon and CCMIX (Centre de Création Musicale Iannis Xenakis, Paris). The technical specification for both of these events requested a symmetrical array of eight full-range loudspeakers driven by the eight bus outputs of an eight bus analogue mixing desk. The system would be used to spatialise live musicians, stereo CD or 8 channel Tascam DA98 digital multitrack. This loudspeaker arrangement differed significantly from the author's experience of surround-sound in 2002 whilst recording and mixing the soundtrack for a Historic Scotland film for the Urquhart Castle Loch Ness Visitors' Centre. The centre's new cinema adopted the ITU BS.775-1 5.1 speaker standard commonly used for DVD, an asymmetrical speaker array consisting of three front speakers, two rear speakers and a sub-woofer (AES 2001 p4).

For spatial positioning of the Urquhart Castle musical elements and sound effects, the author used the joystick-controlled 5.1 panning within the Pro Tools Digital Audio Workstation (DAW). The difference between this and the 8-bus analogue mixing technique used by CCMIX was highlighted when the author and assistant engineer Dave Hook were asked to use the 8-bus mixer to spatialise a live performance for DJ and Tape by Edinburgh Napier student Gavin Fort. Both engineers experienced frustration at their inability to intuitively move a sound in a straight line, square or circular path using the fader-based system, although it must be noted that later research showed that experienced diffusers such as Dennis Smalley were capable of complex manipulation of the eight faders to perform such sound movements (Austin 2001 p23). This leads to an initial research question that will be expanded in 1.2:

"How can the physical user-interfaces and audio software used by the music and film post-production industries offer creative alternatives?"

The author worked from 1983 to 1994 as a software and hardware developer for audio and military applications and had relationships with audio software and hardware manufacturers such as Steinberg and Digidesign, both of whom develop their commercial applications using C and C++. The use of the MAX/MSP graphical programming environment (Cycling '74 2009) that featured repeatedly in the masterclasses contrasted strongly with his personal experience and led to the second research question:

“How can emerging programming technologies offer creative alternatives to the MAX/MSP or hardware-based tools commonly used for sound spatialisation?”

At this point the author saw an opportunity to combine his experience of professional practice in the audio industries with his previous experience as a hardware and software developer. The two research questions suggested a body of research that would investigate alternative user interfaces, DAW hosting and alternative programming technologies and would lead to the development and testing of tools that combined audio industries' practice with electro-acoustic and acousmatic diffusion/spatialisation practice. Although practice-led research leading to the award of PhD through music composition is common and is offered by eleven of the Universities surveyed by Frayling in 1997, Biggs (2000 p2) makes the following statement about the wider application of practice-led PhD research:

“Practice-based projects are those which include as an integral part the production of an original artefact in addition to, or perhaps instead of, the production of a written thesis. They are naturally of great interest to practising artists and designers, but they are not confined to these disciplines. One may find examples in music, in software design, in engineering, in law; in fact in any subject where the result might be an artefact generated in the laboratory or workplace”

In 2005 the Arts and Humanities Research Council (AHRC) commissioned a review of practice-led research that offers the following definition “*Research in which the professional and/or creative practices of art, design or architecture play an instrumental part in an inquiry.*” (Rust et al 2007 p11). The AHRC Funding Guide states that practice-based research leading to a new or improved artefact should integrate with the research questions and research methods to generate

new or enhanced knowledge and understanding in the discipline (AHRC 2009 p15). Biggs (2000 p2) summarises four key points of practice-led research as follows:

- i. Define the research questions to be explored.
- ii. Specify a research context for those research questions.
- iii. Determine a methodology for carrying out the research and answering the research questions.
- iv. State the research's contribution to knowledge and understanding in its area.

1.2 Research Questions and Context

The initial questions can be split and expanded as follows:

RQ1 "How can the physical user-interfaces used for panning by the music and film post-production industries offer creative alternatives to the fader-based hardware approach commonly used for electro-acoustic performance?"

RQ2 "How can a DAW be used as an alternative to dedicated panning hardware?"

RQ3 "How can emerging programming technologies offer creative alternatives to the MAX/MSP or hardware-based tools commonly used for sound spatialisation?"

As stated in 1.1 these questions arise from the limited sampling of spatialisation approaches used by staff and visitors to Edinburgh Napier University and the author's perception of differences between these approaches and those of the music and post-production industries. The initial context for this body of work will be the research and development of a prototype or series of prototypes that will explore the research questions and will be used by staff and visitors participating in the Edinburgh Napier University 'Sonic Fusion' programme of masterclasses and performances. A more detailed research context will result from the Chapter

2 literature review, survey of existing systems and review of working practice by staff and visitors.

1.3 Thesis Structure

This thesis will document the research in three sections:

Part I: Requirements and Methodology

Part II: Implementation

Part III: Conclusions and contribution to knowledge

These sections can be further broken down as follows:

1.3.1 Part I: Requirements and Methodology

The next three chapters of this thesis lead to the generation of the initial Requirement Specification and a methodology for its implementation and evaluation:

Chapter 2: A literature review of past diffusion/spatialisation systems with a focus on technology and loudspeaker configuration.

Chapter 3: A critical review of current systems, leading to a Requirement Specification for a creative hardware and software Toolset that addresses some of their limitations.

Chapter 4: Examination of three commonly used development methodologies and selection of a 'best-fit' methodology that will assist in the implementation of the chapter 3 requirement specification whilst still allowing the research to make a significant and original contribution to knowledge.

1.3.2 Part II: Implementation

Chapters 5 to 13 describe the development of the hardware prototypes and software applications leading to the final Toolset. The initial prototypes are developed using MAX/MSP to allow comparison with later prototypes developed using alternative programming technologies. The use of each prototype in a concert situation is analysed in terms of the research questions.

1.3.3 Part III: Conclusions and contribution to knowledge

Chapter 14 presents a critical evaluation of the research along with its contribution to knowledge and possible future work.

1.3.4 Appendices

Appendices A, B, C and D give more detailed documentation of some of the research findings along with sample code to allow other researchers to utilise the technologies where currently available documentation and example code is minimal or non-existent.

Part I: Requirements and Methodology

2 Literature review of past diffusion/spatialisation systems with a focus on loudspeaker configuration, control and technology.

This chapter examines the evolution of electro-acoustic spatial works in the second half of the 20th Century in relation to their loudspeaker configurations, physical controllers and the advancement of technologies such as the tape recorder, computer and digital audio. Rather than presenting an exhaustive list, this literature review examines trends and key features and thus informs the Requirement Specification for the Toolset in Chapter 3. Since MAX was to have such a significant effect in later years, this chapter is presented in two sections: pre-MAX and post-MAX works and developments.

2.1 1951 to 1987: Before MAX

To help inform the Toolset loudspeaker configuration requirement, the works and events presented in this section will be further categorised by their size of the loudspeaker array into three categories: four speakers, six eight and ten speakers and then large speaker arrays.

2.1.1 Four-speaker works and events

In 1951 the recent availability of the tape recorder allowed Radiodiffusion Television Francais (RTF) engineer Pierre Schaeffer to move from gramophone based composition to a series of pieces for multiple mono tape recorders in conjunction with Pierre Henry. These pieces were projected through four loudspeakers arranged in a tetrahedron – front left, front right, back and overhead (Manning 2004 p26). Sound spatialisation was via the '*potentiomètre d'espace*' also developed that year by Jacques Poullin which consisted of four large receiver coils (exceeding 1 metre in diameter) representing the tetrahedral loudspeaker array placed to the sides, in-front of and above the performer. The coils detected the movements of the performer's hand-held transmitter and controlled the spatialisation via induction in those coils (Manning 2006 p87). Malham and Myatt (1995 p59) state that a variation of this system used a special

5 track tape recorder where 4 tracks each supplied a loudspeaker and the fifth track was spatialised by the performer into those speakers.

In 1956 Karlheinz Stockhausen finished *Gesang der Jünglinge* at Westdeutscher Rundfunk (WDR), this piece had a detailed spatial plan of the loudspeakers and hall, detailing the placement and movement of sounds in the space (Fishman-Johnson 1993 p17). The piece was conceived for four speakers placed around the audience fed from 4-track tape plus a fifth above the audience fed from a manually synchronised mono tape, however, due to practical limitations the premiere took place with a panoramic arrangement of four speakers across the stage. (Zvonar 2000 p3). Following the premiere Stockhausen remixed the piece for quadraphonic playback with the speakers placed equally around the audience as front, left, right and rear (i.e a square rotated by 45 degrees). For his piece *Kontakte* in 1960, Stockhausen used the same 4 speaker diamond loudspeaker configuration as his earlier *Gesang der Jünglinge*, Stockhausen created sounds that orbited the audience by using a highly directional loudspeaker revolving on a turntable whose sound is picked up by four microphones placed around the turntable (Malham & Myatt 1995 p60).

1967 saw the start of Pink Floyd's involvement in four speaker sound with their "Games For May" concert at London's Queen Elizabeth Hall using a rudimentary quadraphonic P.A. system (Cunningham 1997). In 1972 they recorded "Dark side of the Moon" with quadraphonic playback in mind and Engineer Alan Parsons produced a conventional stereo mix and a quadraphonic (quad) mix for the ill-fated quad LP and quad 8-track cassette formats. Quad continued to be a feature of Floyd concerts using a custom mixing desk designed by Allen & Heath that incorporated a joystick panner. After experimenting with different loudspeaker configurations they eventually settled on the front centre, left, right, rear centre diamond favoured by Stockhausen instead of the more conventional left, right and rear left, right pairs used by both of the 1970's 'square-based' quad LP standards. Both of the competing SQ and QS quadraphonic LP formats were, by design, stereo compatible and thus had a front left and right speaker pair supplemented by a rear speaker pair (Crompton 1974 p2).

Also in 1972, Pierre Boulez composed the first of several versions of *Explosante Fixe* for flute, chamber orchestra and electronics as a result of a demonstration of

the capabilities of the Halaphon spatialiser designed by Freiburg Experimentalstudio Director Hans Peter Haller and Engineer Peter Lawo (Warnaby quoting Haller 1996 p32). The Halaphon extended the concept of automated panning demonstrated by Stockhausen with his loudspeaker turntable by providing four channels of automated pattern-based movement (Zolzer 2002 p518).

Until this point the systems described have been analogue, in 1972 John Chowning, together with James A. Moorer, Loren Rush, and John Grey formed a research group at Stanford in psychoacoustics, analysis, digital recording, and digital synthesis design. This group is formally established in 1975 as CCRMA (Center for Computer Research in Music and Acoustics) and the first CCRMA quadraphonic concert took place in 1978 (CCRMA 2009).

Significant to Research Question 1 are two 1960's devices that offered manual panning in contrast to Stockhausen's turntable panning device and the Halaphon described above: the 'Stirrer' designed by Lowell Cross in 1966 for David Tudor (Kuivila 2001 p20), and the 'Azimuth Coordinator' designed by Bernard Speight for Pink Floyd (Cunningham 1997). Both of these devices were potentiometer-based and allowed a sound to be moved between four outputs, Cunningham describes the "Azimuth Coordinator" as follows: *"This elaborate name was given to what was essentially a crude pan pot device made by Bernard Speight, an Abbey Road technical engineer, using four large rheostats which were converted from 270 degree rotation to 90 degree. Along with the shift stick, these elements were housed in a large box and enabled the panning of quadraphonic sound."*

2.1.2 Six, eight or ten loudspeaker works and events

This sub-section examines works and events leading up to 1987 that used more than four loudspeakers but less than the tens and hundreds of loudspeakers to be discussed in 2.1.3. The use of more than four loudspeakers could be expected to be linked to advances in tape recorder track-count, however, around the same time as Schaeffer and Henry's experiments with four speakers, John Cage (1912-1992) and colleagues in the USA formed the "project for music for magnetic tape", an example of which being Cage's "Williams Mix" (1952) for 8 mono tape machines and 8 loudspeakers placed equally around the auditorium (Zvonar 2000 p2).

In 1981 Boulez returns to Freiburg Experimentalstudio to compose *Repons*. The six-speaker piece was commissioned by South-West German Radio using the combined resources of IRCAM and Experimentalstudio as the former did not have an equivalent to the Halaphon at this time (Warnaby quoting Haller 1996 p33).

Also at Experimentalstudio, Luigi Nono begins a fruitful relationship with Haller that continues through the 80's leading to works such as *Prometeo*. *Prometeo* was performed in 1987 in the Hall of the Alte Oper, Frankfurt using a total of 10 loudspeakers, eight around the periphery of the hall and the final two speakers centrally placed firing in opposite directions along the long axis of the hall (Haller 1999 p16 figure 2). The same year, Luciano Berio establishes *Tempo Reale* studio in Florence where Nicola Bernadini and Peter Otto develop TRAILS (Tempo Reale Audio Interaction Location System). TRAILS was capable of 32-channel audio distribution with pre-programmed and real-time spatialisation (Roads quoting Bernadini and Otto 1996 p454).

Just as with four speakers, no standard configuration appears to be emerging and it is becoming clear that the Toolset must be able to support more than one loudspeaker configuration along with pre-determined spatialisation as well as real-time control.

2.1.3 Loudspeaker arrays and the Loudspeaker orchestra

In 1958 Edgard Varèse' *Poème Electronique* was played through allegedly 400+ loudspeakers at the Philips Pavilion at the Brussels World's Fair. The sounds from 15 tape recorders were switched between the various arrays of loudspeakers by telephone relays and the relay routing was controlled by a control tape. (Ernst 1977 p43). The EMF Institute has the following to say: "As Varèse later described it, the sound followed paths through the loudspeaker arrays, and groups of speakers were used to create effects such as reverberation" (EMF 2009)

Expo '70 in Osaka hosted several multi-channel installations, most notably Iannis Xenakis' "Hibiki Hana Ma", a 12 channel piece projected through 800 speakers around, above and under the audience, and Stockhausen's installation in a spherical auditorium consisting of 55 loudspeakers arranged in seven rings from the top to the bottom of the sphere. Zvonar quotes Stockhausen as follows "*To sit inside the sound, to be surrounded by the sound, to be able to follow and experience the movement of the sounds, their speeds and forms in which they move: all this actually creates a completely new situation for musical experience. 'Musical space travel' has finally achieved a three-dimensional spatiality with this auditorium, in contrast to all my previous performances with the one horizontal ring of loudspeakers around the listeners.*"

In 1973 at the Group de Recherches Musicales (GRM), François Bayle created the Acousmonium with eighty loudspeakers placed across a stage, this arrangement has become known as a "loudspeaker orchestra". A different diffusion approach was taken by Christian Clozier for the GMEB Gmebaphone¹, a console and sound processing system with a large number of loudspeakers often

¹ In 1997 version 6 of the Gmebaphone was produced by Christian Clozier, François Giraudon and Jean-Claude Leduc under the banner of Institut International de Musique Electroacoustique de Bourges (IMEB) and was renamed the Cybernephone (IMEB 2009 p2).

varying in size and distance from the audience allowing the diffusion artist to create tonal variations and a near or far sound image (Clozier 2001 p89). A third diffusion approach was taken by Jonty Harrison In 1982 when founding the Birmingham Electro-acoustic Sound Theatre (BEAST). The BEAST sound system is a multi-loudspeaker diffusion system for electro-acoustic music that allows the performer to diffuse a performance via a custom console and Harrison suggests a minimum of MAIN, WIDE, DISTANT and REAR pairs of speakers for reproduction of stereo material (Harrison 1999 p5).

2.2 1988 to present: Post MAX

It will be seen in Chapter 3 that many of the current spatialisation systems have been developed in (or utilise in some way) MAX, therefore its arrival in 1988 can be argued as a turning point alongside the rise of the personal computer. This section looks at the evolution of MAX but also notes the continued use of dedicated hardware systems, for example, in 1990/1991 Stockhausen composed the 8-channel OKTOPHONIE with spatialisation via an Atari-controlled Yamaha DMP7 digital console and a four-channel spatialization unit called the QUEG (Quadrasonic Effects Generator) (Bernardini and Vidolin 2005 p3) developed by Tim Orr and manufactured by EMS in 1975 (Hinton 2001 p10). Pink Floyd also chose a hardware solution for their 1994 tour of the Division Bell album, once again using quad but this time using conventional Yamaha mixing consoles for instruments, vocals and tape replay together with a custom-built 16 channel quadrasonic Midas XL3 with two manually operated joysticks. (Hilton 1994 p3)

2.2.1 IRCAM and MAX

In IRCAM in 1988 Miller Puckette finalised the first version of the graphical programming language MAX named in honour of Max Mathews, author of the RTSKED program. This completed IRCAM's 4X computer digital sound processor. Although MAX was originally conceived as a combined event (MIDI) and signal (DSP) environment, the initial development of the program was MIDI only as this was the only interface available to the 4X system (Puckette 1991 p68). The IRCAM Signal Processing Work Station (ISPW) came into general use for computer music in 1991. It was based on a NeXT computer with a set of DSP cards having multiple inputs and outputs and Max is ported to the NeXT operating system. Opcode Systems release Max as a commercial product under the development of engineer David Zicarelli. In 1996 Puckette develops the MAX-like public domain program Pure Data (pd) to extend the capabilities of MAX (Puckette 1996 p1). The audio DSP aspect of Pd is later used by Zicarelli as a starting point for his "MAX Signal Processing" (MSP) program and in 2000 Zicarelli/Cycling '74 concludes an agreement with Opcode/Gibson and IRCAM and takes over the publication and release of MAX/MSP.

2.2.2 CNMAT and Level Control Systems

In 1989 David Wessel, formerly at IRCAM working alongside Miller Puckette founded the Center for New Music and Audio Technologies (CNMAT) (*pronounced sennmat*) within the University of California, Berkeley. CNMAT is now part of a consortium with Stanford's Center for Computer Research and Acoustics CCRMA and IRCAM. CNMAT has strong links with Level Control Systems (LCS), founded by sound designer Jonathan Deans and software designer Steve Ellison to manufacture automated audio control systems for theatre, sound reinforcement, and location-based entertainment such as theme parks, circuses and cruise ships. Also in 1991, Naut Humon establishes the Sound Traffic Control project using a 12 channel analogue mixer by Level Control Systems and four years later the project is upgraded using a 40 channel digital LCS system consisting of five LD-88 mixers. Part of LCS's supernova system, the LD-88 was an 8 x 8 audio mixer with built-in DSP for equalisation and channel delay. Up to 16 LD-88s could be digitally interconnected to create a 128 input and 128 output system. Control of the LD-88 was via their CueStation software which provides a graphical user interface to the processor's DSP and surround panning functions. The Sound Traffic Control project had an integral 3D viewer window called 'ThreeD' and also included a computer running MAX/MSP (Humon et al. 1998 pp1-5).

In 2006 Level Control Systems became part of MeyerSound and produced their 3rd generation system called **Matrix3** (MeyerSound 2009). This system is the successor to the LD-88 show control system used by the Sound Traffic Control project. Also in 2006, Sound Traffic Control's founder Naut Humon established a theatre venue for Recombinant Media Labs (RCL) in San Francisco, the successor to Sound Traffic Control is named 'Surround Traffic Control' and was a 16 speaker system arranged as two rectangular layers of 8 speakers, one layer at ceiling height and the other below ear height. The system incorporated a Sony DRM100 digital console and custom spatialisation software written using Max MSP, Supercollider, PD and Kyma along with software from Immersive Media Research described in Chapter 3 (Jones 2006 p2). This venue closed in spring 2008 and RCL is currently collaborating with Peter Otto at UCSD which has resulted in a performance in the UCSD experimental theatre in 2009 (Threw 2009).

2.3 Chapter summary and research contribution

The review highlights two separate loudspeaker trends:

- i. Small loudspeaker systems, typically 4 or 8 placed equally around the auditorium, occasionally with an overhead speaker but more commonly in a single two-dimensional layer.
- ii. Large “loudspeaker orchestras” or installations with speaker arrays numbering in the tens or hundreds often weighted with more speakers towards the stage and often with some speakers above the audience.

Looking at the section in terms of positional control, there are two clear types:

- i. Manual positioning devices such as the potentiomètre d'espace, Joystick, Stirrer and Azimuth Coordinator;
- ii. Automated positioning devices such as Halaphon, TRAILS and Stockhausen's loudspeaker turntable.

Considering the chapter in terms of Research Question 1 (user interfaces), It is clear that both manual and automated means of control should be developed as part of the Toolset. It should be noted that although Poulin's potentiomètre d'espace was the only controller capable of panning a sound within a three-dimensional space, several of the works used 3D loudspeaker arrangements therefore the Toolset and its controllers should accommodate 2D and 3D loudspeaker configurations.

Considering the chapter in terms of Research Question 2 (DAW as an alternative to dedicated hardware), although the early spatialisation systems were purely hardware-based, there are now other alternatives in use such as purely software-based solutions using MAX/MSP and hybrid solutions using a hardware matrix mixer controlled by software. As the next chapter will confirm, there still appears to be a research gap to be addressed by Research Question 2: “How can a DAW be used as an alternative to dedicated panning hardware?”

3 Currently available institution-based or commercial systems

Halaphon was the in-house hardware-based spatialisation system developed at Freiburg Experimental Studio in 1971 by Hans Peter Haller and Peter Lawo, initially a 4 channel system using variable speed and waveform envelope oscillators to modulate the amplitude of the loudspeaker outputs and create pattern-based movement (Zolzer 2002 p518). In 1989 the Halaphon functionality was incorporated into the Experimental Studio 48x48 Matrix Mixer. Email discussions in December 2008 with Thomas Hummel at Experimental Studio revealed that the current Halaphon implementation is a MAX/MSP program with pre-programmed movement but manual control of the speed.

TRAILS (Tempo Reale Audio Interactive Location System) by Peter Otto and Nicola Bernardini used a matrix of up to 512 VCAs (Puckette 1991 p5) to create a 24 channel system with eight separate trajectories (Osmond-Smith 1991). The smaller MiniTrails system contained an 8x8 VCA matrix that allowed the user to trigger and level balance 8 channels of movement stored in a playlist (Vidolin 1993). Email discussions in May 2009 with Damiano Meacci at Tempo Reale revealed that the spatialisation system under development is a MAX/MSP program with Tempo Reale externals.

SPAT is a software suite developed by IRCAM that spatially processes source signals with simulation of air absorption, Doppler effect and room acoustic before output using headphones or up to 8 loudspeakers (IRCAM 2008 p1). The suite is comprised of C++ externals and Max patches and in earlier versions of MAX/MSP it was possible to export a MAX patch as a VST plug-in which could then be 'wrapped' into RTAS and Audio Unit DAW plug-in formats. With the release of version 5 however, Max/MSP has discontinued support for VST plug-in generation. Although MAX does not have built-in support for MIDI Time Code, Peter Elsea at University of California, Santa Cruz (UCSC) describes a MAX patch that allows reading of MTC (Elsea 2004 p1). This could be developed to provide basic slave synchronisation but not master.

IMEASY X (Integrated Modular Expandable Audio Spatialisation system) by A&G Soluzioni Digitali is a control application that sends MIDI control messages to their XSPAT standalone DSP hardware that in turn performs spatialisation of eight sounds. The Apple-compliant user interfaces suggest that Apple's software development environment has been used. Spatial input is via a connected pen called "The Bat" that can be moved within a virtual cube to give 3D real-time input into IMEASY (A&G 2009) The application allows master or slave synchronisation with external equipment or software.

Vortex Surround Designer is a product by Immersive Media Research (IM Research 2008) whose Directors and advisors include Peter Otto, the co-designer of the Tempo Reale TRAILS system and Naut Humon, founder of Sound Traffic Control. Input sources are files and the application cannot process real-time live input. Spatial breakpoint-based panning is a strong feature but there is no synchronisation with other software as either master or slave. Distinctive audio setup settings and User Interface items such as menus suggest that this application has been written using MAX/MSP.

ABShowMaker is the control application for the Audiobox AB64 matrix mixer, audio recorder and show controller by Richmond Sound Design (Richmond Sound Design 2009) In addition to fader-based real-time control, the editor allows the user to add show events to a cue list, a diffusion editor allows straightline trajectories to be pre-programmed and added to this list.

Matrix3 by MeyerSound is a hardware-based matrix mixer with hard disk audio playback (MeyerSound 2009). Like its predecessor the LD-88, control of Matrix3 is by CueStation software or faders. A significant feature of the current system is its 'Cue List' approach that allows different settings to be saved as cues to be recalled during a performance either manually or under the control of a master synchronisation source such as MIDI Time Code (MTC). The current version is OS X-based and incorporates 'SpaceMap', a feature of the application that allows trajectories to be entered graphically.

OKTEG was developed in 2007 for Stockhausen's 'Cosmic Pulses' by Gregorio Karman and Joachim Haas from Experimental Studio. OKTEG is a Max/MSP patch containing eight sequencer-driven amplitude-panning modules with

independent fader controlled tempo controls working in conjunction with Pro Tools. Pro Tools was used for sound playback and to record the encoded trajectories produced by OKTEG for subsequent playback. (Nordin quoting Karman 2007 p9).

Table 3-1 below presents these currently available systems and compares key features:

	Platform	language	DAW hosted	Panning Interface	Trajectories & automation	Trajectory view	Sync. Master-slave
SPAT	Mac	MAX	-	User-defined	Requires programming	-	-
Halaphon	Mac+ Matrix mixer	MAX	-	Mouse	2D Shape-based	-	-
OKTEG	Mac	MAX	DAW Partner	Mouse + faders	2D Straightline via Pro Tools	-	Via Pro Tools
TRAILS	Mac	MAX	-	Mouse + faders	2D Straightline	-	-
MATRIX3	Mac + standalone hardware	Undisclosed (C OR C++)	-	User-defined	2D Straightline graphic-based	-	Y
ABShow -Maker	Mac/PC + Audiobox hardware	Undisclosed (C OR C++)	-	Mouse + faders	2D Straightline Table-based	2D non-realtime	Y
IMEASY X	Mac + XSPAT hardware	Undisclosed (C OR C++)	-	Mouse + 'the Bat'	2D Node-based	3D non-realtime	Y
VORTEX	Mac	Undisclosed (MAX)	-	Mouse	2D Breakpoint-based	-	-

Table 3-1: Comparison of current systems

It is useful to consider the table within the context of the three research questions:

faders in the form of the Bat and is therefore the only system that would allow an artist or performer to directly position a sound. The Bat is also the only device that generates 3D positional information.

3.1.2 Research Question 2 - DAW Hosting

None of the systems can currently be DAW-hosted. With the exception of SPAT all these systems are standalone solutions. SPAT is a component and not a complete system therefore SPAT-based solutions are likely to be standalone although prior to MAX/MSP version 5 SPAT could be integrated into a DAW such as Pro Tools as a wrapped VST plug-in (Expansion 2009). Three of the systems use the manufacturer's DSP-based hardware to perform spatialisation, the remaining five systems are standalone systems and can use any Mac-supported multi-channel audio hardware. Okteg is the most recent system and although this MAX/MSP application performs its own spatialisation it is designed to work in conjunction with Pro Tools. Pro Tools automation gives Okteg the ability to record and replay trajectories in relation to a timeline (Nordin 2007 p10).

3.1.3 Research Question 3 - Comparing Max/MSP and alternative technologies

Five of the systems are based on MAX/MSP, the other three use undisclosed programming technology however, since they use Apple user interface elements such as buttons and pop-up menus it is likely that they use the Apple Integrated Development Environment (IDE) and therefore C/C++ or Java languages (Apple 2009 p1). It can be seen from the table that system functions such as synchronisation and trajectory visualisation are missing from the MAX/MSP based systems, hence a contribution to knowledge could be made if this thesis investigates alternative programming technology to provide the synchronisation missing from MAX/MSP solutions. The selection of an appropriate programming technology will be discussed in chapter 6.

3.1.4 Additional considerations

The literature review and the resulting table clearly show that the systems have additional features not considered in the original research questions that could

inform the requirement specification for the Toolset to be developed by this body of research:

3.1.5 Trajectories, cue-lists and automation

Beginning with the original Halaphon, the ability to pre-program a sound's path has become a feature of most of the systems and should therefore be a requirement for the Toolset resulting from this research. The currently available systems generally focus on straightline 'two-dimensional trajectories' therefore there is scope for the research to extend this into three dimensions and include other trajectories such as circular paths. In addition to recording and replaying the user's movement of sound using automation, ABSShowmaker and Matrix3 both feature a cue-list approach allowing the user to numerically and/or graphically pre-describe the movements of sounds and this too should be added to the Toolset requirement specification.

3.1.6 Synchronisation

Only the three commercial systems that use their own DSP hardware can act as timecode masters or slaves and thus be integrated into systems either requiring or generating positional and timing references. As this functionality is missing from MAX/MSP (and therefore from the systems in the table that use it) but is a common feature in commercial audio production, it will be added to the Toolset requirement specification.

3.2 Summary of requirements

At this point the Toolset requirements resulting from the literature review and analysis of current systems can be summarised as follows:

- i. The Toolset is to be DAW hosted
- ii. The Toolset development will include the use of commercially used programming languages and Integrated Development Environments.
- iii. The Toolset will integrate or develop alternative panning interfaces including joysticks and touchscreen controllers.

- iv. The Toolset will provide 3D shape and breakpoint-based automatable trajectories via a cue-list.
- v. The Toolset will allow synchronisation with other applications as a master or slave
- vi. The Toolset will provide real-time 3D trajectory visualisation.

The first two requirements require more detail before development can begin, namely choice of DAW host or hosts and choice of programming language/IDE. Hoffman (1993 p21-23) suggests the combined use of four techniques to obtain an effective requirement specification:

- i. Interviewing prospective users
- ii. Survey of Documents and Existing Systems
- iii. Analysis of Working Practice
- iv. Experimentation with Prototypes

Hoffman's points i and iii should be considered within the context of this research: as stated in section 1.2 this context is the 'Sonic Fusion' series of masterclasses and performances at Edinburgh Napier University. Table 3-2 lists the participating Edinburgh Napier University music staff along with 2004/2005 visitors and the platform and software applications used on that occasion.

User	Institution	Platform	Software
Davismoon, S	Edinburgh Napier University	Mac	MAX/MSP & Pro Tools
Ferguson, P	Edinburgh Napier University	Mac	Pro Tools & Ableton
Giomi, F	Tempo Reale	Mac	MAX/MSP & Pro Tools
Hails, J	Edinburgh Napier University	Mac	MAX/MSP & Pro Tools
Hook, D,	Edinburgh Napier University	Mac	Pro Tools & Ableton
Medboe, H	Edinburgh Napier University	PC/Mac	Cubase
Miranda, E	University of Plymouth	PC	MAX/MSP
Pape, G	CCMIX	Mac	MAX/MSP & Pro Tools
Stillie, B	Edinburgh Napier University	Mac	Logic & Ableton

Table 3-2: Sonic Fusion series staff and 2004/2005 visitors

Although no software application dominates Table 3-2, Pro Tools and Ableton Live (Ableton 2009) stand out as the main DAW alternatives to MAX/MSP. Table 3-2 also clearly shows that the majority of the Sonic Fusion users are Mac-based, as are the University's recording studios and music labs. Although these users were using Mac OS 9 for their performances in 2004 most were experimenting with OS X, therefore to provide maximum contribution to knowledge the research will focus on the emerging OS X. Requirements one and two above can be extended to give the final versions seen in Table 3-3, requirements three to six remain unchanged:

1	The Toolset is to be Mac-hosted using Pro Tools and Ableton Live DAWs.
2	The Toolset development will include the use of emerging programming languages and IDEs for the Mac OS X platform.
3	The Toolset will integrate or develop alternative panning interfaces including joysticks and touchscreen controllers.
4	The Toolset will provide shape and breakpoint-based automatable trajectories via a cue-list.
5	The Toolset will allow synchronisation with other applications as a master or slave
6	The Toolset will provide real-time 3D trajectory visualisation.

Table 3-3: Final Toolset Requirement Specification

Chapter 4 will examine the choice of an appropriate methodology for the implementation and review these requirements.

3.3 Potential contributions to knowledge

This thesis will potentially extend knowledge in its field by:

- i. Pointing towards an alternative perspective on the key developments of spatial audio since 1950 by focussing on loudspeaker topology, control and associated hardware;
- ii. Examining the use of emerging Mac programming technologies and development environments as alternatives to MAX/MSP by implementing a set of requirements followed by critical review;
- iii. Developing alternative physical methods of spatialisation control;
- iv. Developing a novel shape-based approach to spatialisation control via a cue-list;
- v. Expanding the documentation and sample code available to potential application developers in this field.

The final contributions to knowledge resulting from the research, development and evaluation of the Toolset will be reviewed in Chapter 14.

4 Methodology

Having established the primary aims of this body of work in chapter 3, we can now consider three commonly used development methodologies to determine a 'best-fit' strategy for the development and critical evaluation of the Toolset:

4.1.1 Waterfall Model

The author worked as a Defence Industry hardware and software design engineer from 1983-93 on tightly controlled multi-person projects following the Waterfall methodology characterised by the following sequential stages (Oriogun 2002)

- i. Requirements phase
- ii. Design phase
- iii. Implementation
- iv. Systems Testing
- v. Operation and Maintenance

Key to the successful use of this methodology was an emphasis on the Requirements Engineering phase to ensure an accurate and stable Requirements Specification, this view is reinforced by Modarres (Modarres et al 1999 p345). The "client" involvement was critical during this initial phase since that would be their only involvement until the product was delivered. The Waterfall model can be criticised for its inflexibility when requirements are changed during the later project phases, and, since each phase must be completed before the next is carried out, design problems can remain undetected until the System Test phase (Goodliffe 2007 p429).

4.1.2 Incremental and Iterative Development Model (IID)

Incremental Development is a methodology where new features are developed and incrementally added to successive releases of the application or system taking the form of a series of prototypes or software releases. Incremental

Development is usually combined with Iterative Development, defined by Defense Journal contributor Alistair Cockburn as:

“a rework scheduling strategy in which time is set aside to revise and improve parts of the system”. (Cockburn 2008 pp.27–30).

By combining Incremental Development with Iterative Development, the IID methodology allows the developer to benefit from feedback from users, leading to improvements in software quality as well as features.

4.1.3 Spiral Model

The Spiral model was presented by Barry Boehm in 1988 as a risk-driven process rather than document-driven or code-driven (Boehm 1988 p1). In Fig. 4-1 the spiral represents iterations round Boehm’s four-phase process. Goodliffe (2007 p430) states that each 360-degree turn round the spiral represents a single waterfall and that each of these iterations has a typical duration of 6 to 24 months.

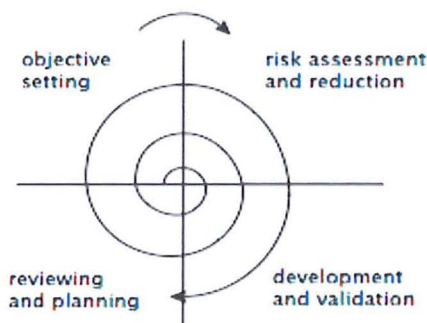


Fig. 4-1: Boehm Spiral model

4.2 Methodology Conclusion

If this research is to fully explore the capabilities of the emerging Mac technologies for OS X, the chosen methodology must accommodate new operating system versions with additional features that could impact upon the

implementation of the Toolset requirement specification. Similarly, client use and real-world testing of the Toolset is also likely to result in modifications and additions to the Toolset requirements. The evolution of the Toolset requirements and the need to periodically revisit the implementation strongly suggest that an Incremental and Iterative methodology or its Spiral derivative would be appropriate and a Waterfall model would not. Although the Spiral methodology's Risk Assessment and Reduction phase may be highly relevant in a cost and/or time-sensitive commercial development, there is potential for conflict with the 'contribution to knowledge' research aim of this work since it involves the use of new OS X technologies with minimal documentation.

4.2.1 Prototype Testing

Although the Edinburgh Napier masterclass context for this research only offers limited scope for user testing, this interaction with users will aid the design process, especially in the areas of user-interface and physical control of spatialisation. If areas of this research are found to offer opportunities for post-PhD research or commercial development then a formal testing and critical evaluation strategy will be required. This engagement with a wider range of users must be intrinsic to the chosen development strategy and at this point a move to the risk-driven Spiral model or its derivatives may be appropriate.

Part II: Implementation

5 Investigating an initial panning solution

5.1 Introduction

This section describes the initial phase of hardware and software development for the first and second prototypes along with their subsequent testing and relates to Research Questions one and three:

RQ1 “How can the physical user-interfaces used for panning by the music and film post-production industries offer creative alternatives to the fader-based hardware approach?”

To allow this question to be answered, physical examples of said user-interfaces must be created. This section first describes joystick, then touchscreen implementations that will be used in the second panning prototype.

RQ3 “How can emerging programming technologies offer creative alternatives to the MAX/MSP or hardware-based tools commonly used for sound spatialisation?”

The first two prototypes will be constructed using MAX/MSP so that a reference can be established to allow comparison between MAX and the alternative programming technologies that will be introduced in section 6.

5.2 Passive analogue panning

As a starting point it is worth considering the passive panning approach used in 1966 by Bernard Speight for Pink Floyd's joystick-based quadraphonic 'Azimuth Co-ordinator' (Cunningham 1997 p2), and later by Neve in consoles such as the VRP series with its 'front-back' and 'left-right' potentiometers (Neve 1991 p 3:13). Fig. 5-1 shows a simple passive panner block diagram:

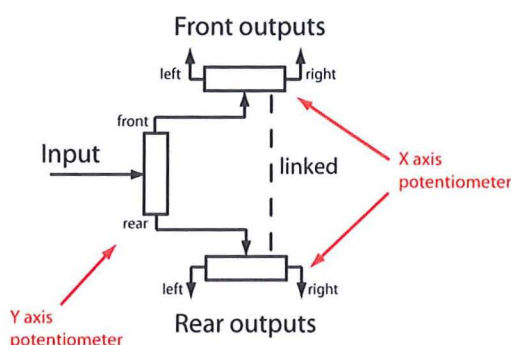


Fig. 5-1: Passive panner

In this diagram the mono input is fed into a potentiometer (pot) controlled by the y-axis of the joystick, thus the y position of the joystick determines the proportion of the signal sent to the front and rear x-axis pots. The x-axis of the joystick controls the dual-ganged front and rear x-axis pots and therefore the left/right panning of the front and rear signals supplied by the y-axis pot. This passive technique for quadraphonic panning will be digitally replicated using Max/MSP to form the basis of the digital panner described in section 5.3 and the DAW plug-in using C++ to be described in section 13.

5.3 Prototype 1: The initial digital panner

Three weeks prior to the opening of Napier University's new Craiglockart Campus mid-September 2004, an interactive soundscape was proposed by Napier University's Ian Tomlin School of Music wherein visitors could trigger sounds and modify them using filtering and other effects. This event offered a useful development deadline for prototype 1 and the proposal was changed from stereo to a quadraphonic speaker layout to allow visitors to spatialise triggered sounds.

The MAX/MSP patch shown in Fig. 5-2 digitally replicates the passive quad output joystick technique examined in Fig. 5-1 by using the 'constant distance xfade' example patch by Cycling '74 as a potentiometer substitute.

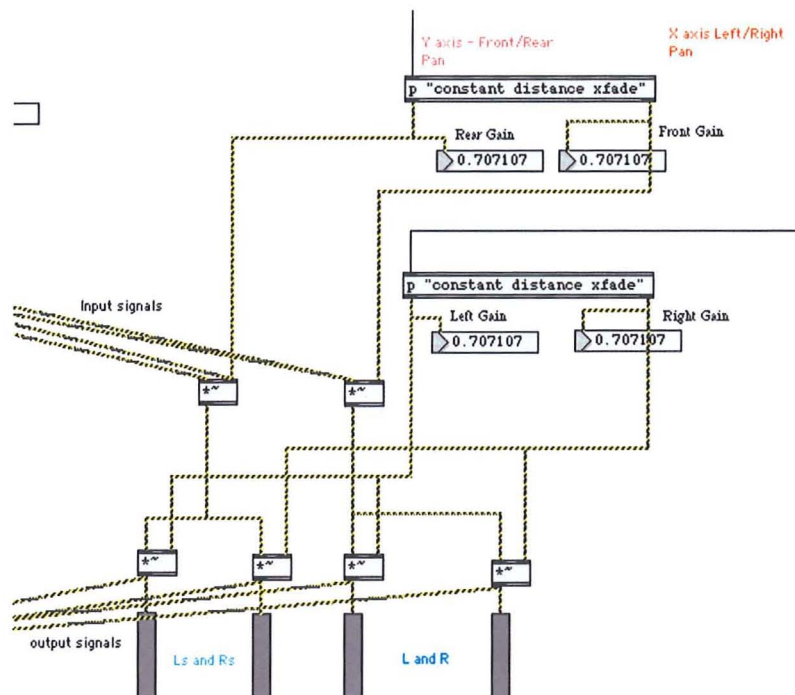


Fig. 5-2: Prototype 1 - Craiglockhart panner using Max/Msp

The y-axis MIDI controller value drives the y-axis constant distance xfade 'pot' whose output is a pair of inversely proportional floating-point front and rear gains ranging from 0 to 1. These gains are used to determine the amount of input signal fed to the front and rear left-right panners controlled by the x-axis MIDI controller value. The host for this patch was a first generation 500MHz Apple G4 Powerbook running Mac OS9 with a Digidesign 002R firewire audio interface. One problem was experienced during testing: Max/MSP would sometimes fail to output any audio, although this could usually be resolved by restarting the computer.

A Wacom Graphire 2 tablet and pen was used as the user input device via the Wacom Max External by Cycling '74's Richard Dudas. The resulting patch worked without problems during the Craiglockhart opening event and was met with enthusiasm by visitors, in particular Simon Gage, Director of the Edinburgh Science Festival. Prototype 1 did not have any visual indication of panning position and it was observed that several users expected the pen to draw

something on the screen and had initial difficulty relating the Wacom tablet to sound movement.

5.4 Prototype 2 development

5.4.1 Background

Before describing the development of joystick and touchscreen controllers for the prototype this section will expand upon the author's use of commercial devices. As stated in 1.1 the initial research questions arose from involvement in 5.1 surround mixing projects for Heritage Scotland and EMI that took place between June and December 2004. The first project used a Digidesign Pro-Control/Edit Pack control surface for the first mix of the Urquhart Castle Visitor Centre soundtrack in 5.1 surround sound. Digidesign's Edit Pack contained two Penny & Giles motorised faders that proved invaluable for simultaneously panning two sound effects that interacted with one another. The motorised faders gave some visual feedback as to the sounds' positions but the standard Pro Tools surround panner window (Fig. 5-3) gave a considerably clearer indication.

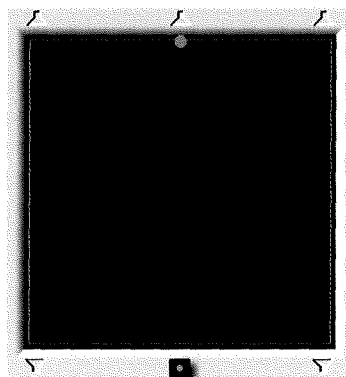


Fig. 5-3: Pro Tools panner window

The second project used the Sony DMX R100 digital console based on the large format Sony Oxford console to surround mix a live recording for DVD. This console has facilities broadly similar to other digital desks such as the Yamaha 02R96 but has a LCD touchscreen as its user-input medium. This touchscreen allows surround sound panning by using a finger to move a dot within a square that represents the room, essentially the same as the Pro Tools panner window but with a touch overlay. This way of manipulating the sound felt intuitive and was immediately much easier to control than a joystick or mouse. The only drawback

was that the touchscreen only allowed one point to be touched at any one time and therefore only one sound could be panned at a time.

A suitable target for the completion of prototype 2 was the opening concert for the 2006 Sonic Fusion Festival in Edinburgh in which it was to feature during a performance by Danish guitarist Haftor Medboe. This required development of the multi-channel spatialiser shown in Fig. 5-4 with the following specifications:

- 8 realtime inputs
- 8 speaker outputs
- 8 graphic panners using a touchscreen
- joystick control of graphic panners via MIDI

5.4.2 Considering Prototype 2 in terms of the IID methodology

Incremental additions: this prototype increment will develop and integrate joystick and touchscreen controllers and this will introduce programming technology other than MAX/MSP.

Iterative changes: this is the first iteration of prototype 1 and will continue to use MAX/MSP for the panning engine with a move to Mac OS X. The MAX/MSP panner will be extended to eight channels. In response to user feedback during the Craiglockhart event, visual representation of the positions of sounds will be added to the user interface.

Fig. 5-4 shows the physical layout of prototype 2:

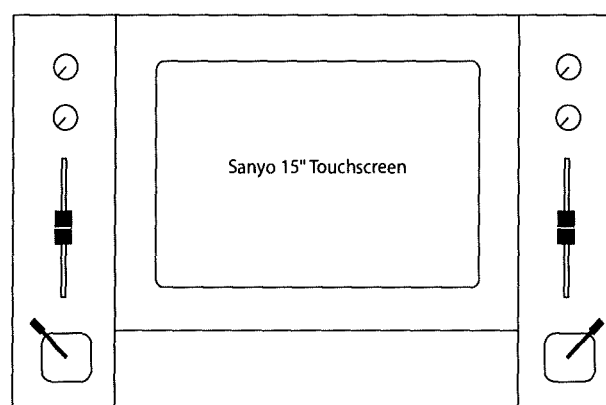


Fig. 5-4: Prototype 2 multi-channel spatialiser

5.5 Joystick development

Two commercial MIDI joysticks were available in October 2004: the Axis Panner by Gallery and the MCS-Panner by JL Cooper. They were both bulky compared to the small joysticks found in Pro-Control and the Yamaha DM2000 and their software protocols were not in the public domain. A small low-cost joystick mechanism was selected that would require additional electronics and software to function as MIDI controller.

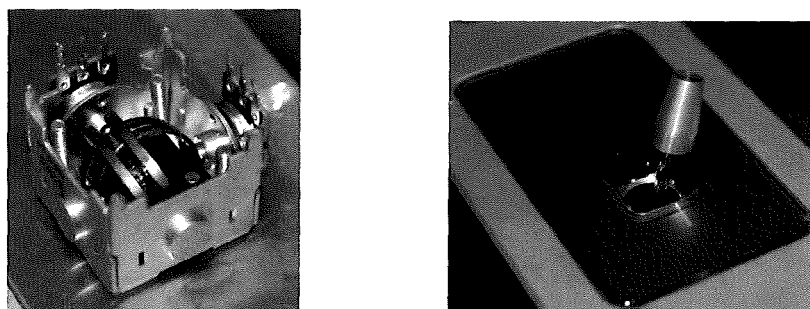


Fig. 5-5: Low-cost joystick viewed from below and above

Due to its high level of on-chip integration of memory and A/D conversion, one of the Philips/NXP LPC family of high integration microcontrollers was selected for the joystick electronics. The LPC935 has an 8031 core and on-board A/D converters and flash memory and allowed the two-chip solution shown in Fig. 5-6:

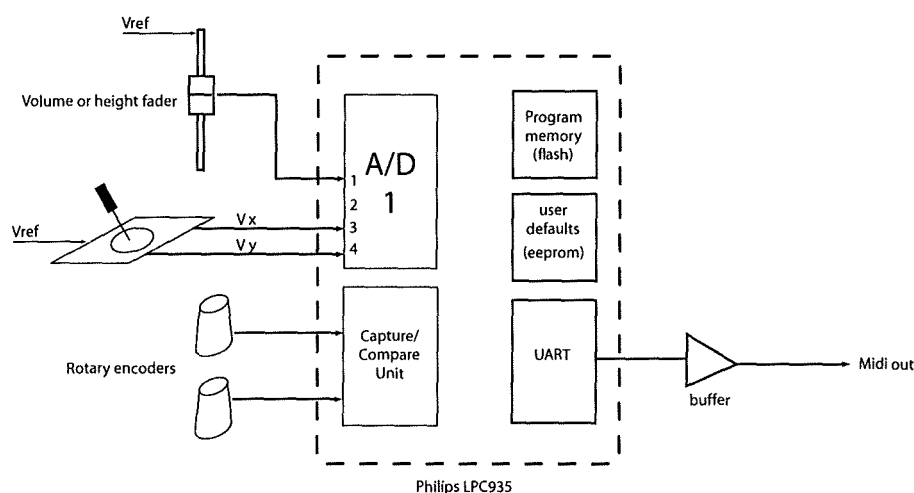


Fig. 5-6: Philips LPC microcontroller-based panner

The Keil MCB900 lpc935 development board was chosen for the prototype development using the included version of the Keil C compiler limited to 4KB program size.

5.5.1 Software development for the joystick

The four-channel A/D converter was set to continuously convert the four channels and to generate an interrupt on completion. The MIDI output via the UART was also interrupt-driven meaning a complete 3 byte MIDI Continuous Controller (CC) message could be loaded into the output buffer for the UART interrupt handler to transmit. The A/D interrupt handler compared each channel with the previously received value and if a change was detected a CC message was loaded into the MIDI output buffer. The CC message numbers were 'hardwired' into the code as CC 12 and 13 for x and y-axes respectively to match the x-y output of Korg's Kaoss Pad (Korg 2007).

The prototype joystick performed satisfactorily although power supply noise affected the Voltage Reference for the joystick causing a random glitching of the A/D converted value. This resulted in random jumps in the position of the panned sound and was corrected by improving the power supply regulation and Vref decoupling.

5.6 Touchscreen development

Following on from the highly positive experience using the touchscreen panning window in the Sony DMX R100, the decision was made to incorporate a small touchscreen into the second prototype to give the user the option of joystick or touchscreen (or both if two sounds were to be panned simultaneously). A search for small LCD touchscreen monitors uncovered a family of devices with USB touchscreens manufactured by the Taiwanese company Lilliput and a seven inch allegedly "Mac compatible" unit was evaluated February 2005. Unfortunately the Mac was unable to detect the presence of the display. After close investigation it was discovered that the Lilliput display was not Display Data Channel (DDC) compliant and did not incorporate the Extended Display Identification Data (EDID) ROM that would allow a computer to query the monitor's parameters. As a result of this lack of DDC support a considerably larger 15" Sanyo LMU-TK15A4T LCD touchscreen monitor salvaged from a multimedia kiosk was evaluated. The touchscreen in this display was a '3M Microtouch' capacitive device and therefore required a much lighter touch compared with the pressure required by the resistive touchscreen on the Lilliput display. The touchscreen controller was a 3M EXII Capacitive interface controller with a RS232 serial

output and worked well with a Keyspan USA28X to RS232 converter and the 3M Microtouch driver.

The 15" Sanyo panel also prompted a re-think of the prototype interface as its larger size and 1024 x 768 pixel resolution allowed either eight small panners or 2 rows of four medium-sized panners to be arranged across the screen.

5.7 Prototype 2 software

5.7.1 Joystick microcontroller software

The software can be split into two parts, firstly the embedded C code in the Philips LPC935 microcontroller written using the Keil C51 compiler and downloaded to the target using Embedded Systems Academy's FlashMagic. Secondly, the Max/MSP based audio manipulation part of the software that would take realtime audio inputs and spatialise them based on the positions obtained from the touchscreen or MIDI joysticks.

5.7.2 Version 2 of the Max/MSP based panner

At this point in the development (January 2006) the new joystick and touchscreen offered the user two ways to manipulate the position of a sound, however the simple Max/MSP based panner developed for the Craiglockhart launch in 2004 only allowed 4 speaker panning and needed to be extended to 8. The Max **simple linear panner** patch was modified to provide a centre speaker output as shown in Fig. 5-7:

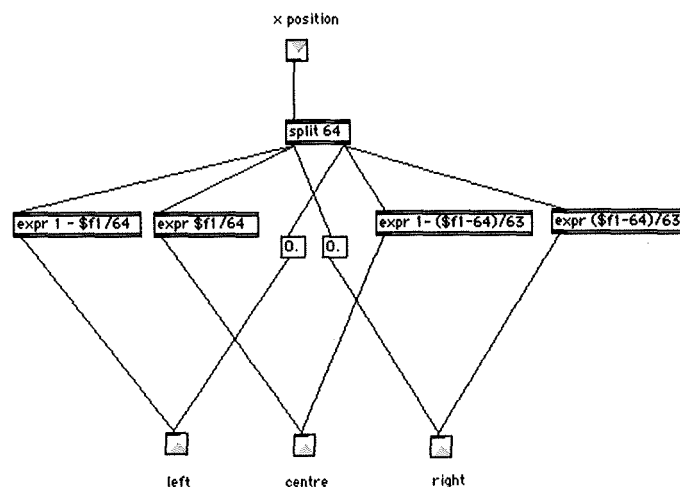


Fig. 5-7: modified Max/MSP simple linear panner

By extending the passive joystick technique shown in Fig. 5-1 and this time using a combination of Left/Right (LR) and Left/Centre/Right (LCR) panners the six and eight speaker configurations in Fig. 5-8 could be obtained. The mono-to-eight-output panner sub-patch using this technique is shown in Fig. 5-9.

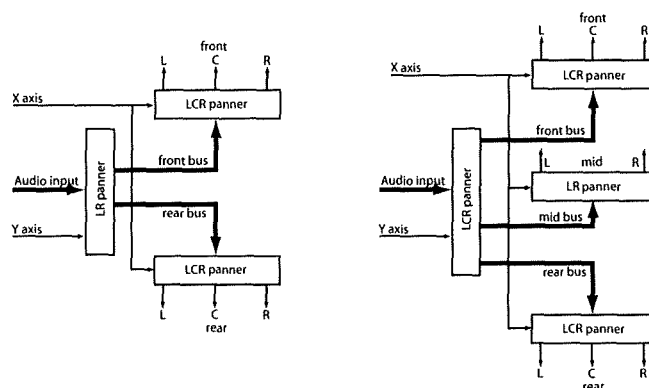


Fig. 5-8: Six and eight-channel panner

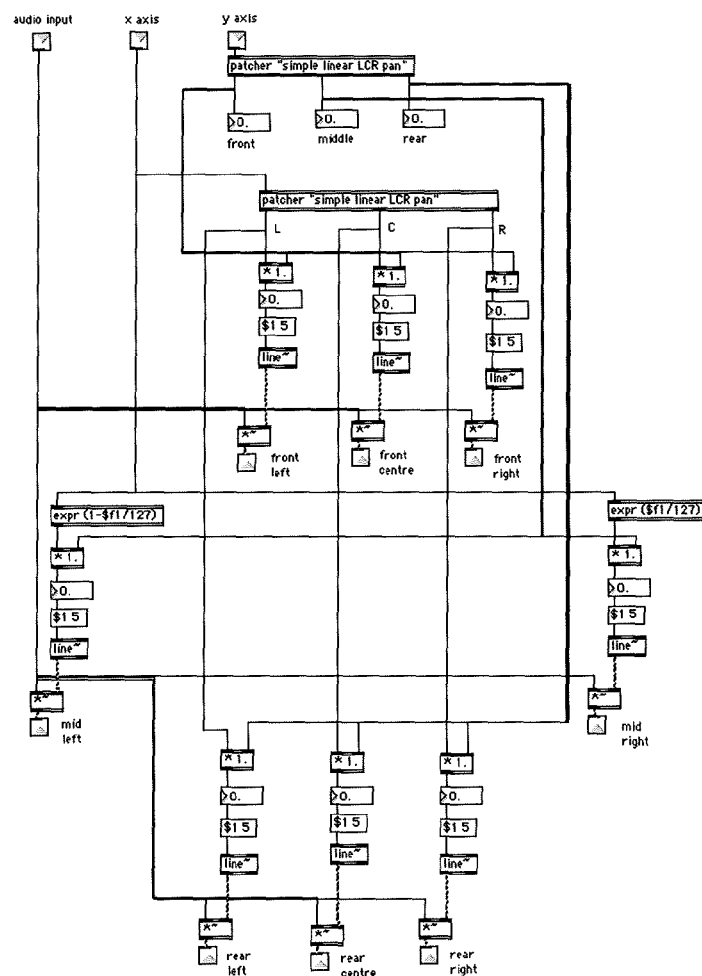


Fig. 5-9: Sonic Fusion 2006 Max/MSP panner sub-patch

The Fig. 5-9 octal sub-patch was replicated eight times to allow eight separate sound inputs to be spatialised to 8 loudspeaker outputs. The Max Icd~ object was used to give a graphical representation of each sound's position based on a simplification of the Pro Tools panner window.

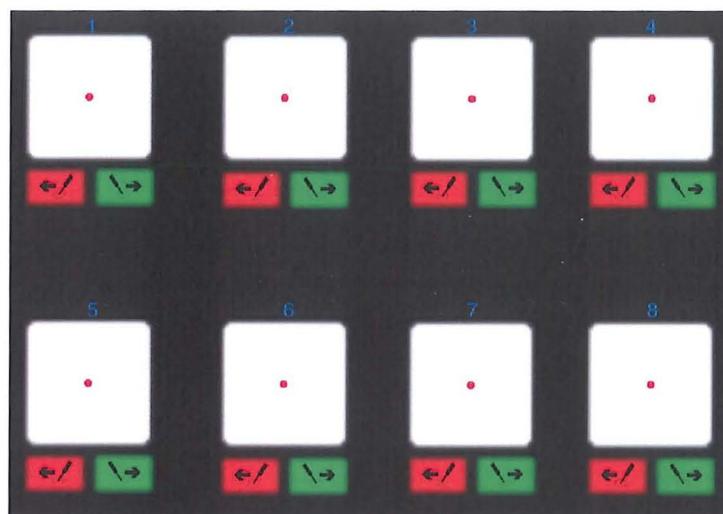


Fig. 5-10: OctoPanner v1 (Sonic Fusion 2006)

5.7.3 Testing Prototype 2

The improvised concert piece by Medboe would use two Line 6 DL4 delay/looper pedals (Line 6 2009) to allow him to build up repeating textures and phrases that would accompany the direct guitar signal. To allow spatialisation of these three sound sources, the two pedals and guitar were connected via DI boxes to a stereo preamp. A Digidesign 192 I/O and Pro Tools HD core card together with its CoreAudio driver provided the Max/MSP runtime environment with 8 inputs and 8 outputs.

The prototype performed flawlessly during the afternoon rehearsals but during the concert the Max/MSP patch failed to output audio. The prototype nature of the system was explained to the audience whilst a reboot was attempted at which point Max/MSP and the audio system successfully connected allowing the piece to be performed as planned.

5.8 Chapter summary and research contribution

Firstly considering Research Question 1, the introduction of the joystick and touchscreen had provided most of the level of positional control anticipated by the author although changing between the eight channels caused positional jumps since the joystick was not motorised. A significant discovery arose from the part-improvised nature of the concert piece. During the performance Medboe had been aware of wanting a specific spatialisation of one particular sound as he played but was unable to communicate this to the engineer. A discussion ensued about enhancements to allow the musician to pan their own sound either by guitar mounted joystick or foot control and this led to the additional research that is discussed in section 9. This was a significant change to the requirement specification and illustrates the flexibility of the IID methodology to incorporate Requirement Specification changes resulting from user testing of each system iteration.

The near-failure of Prototype 2 in concert conditions gave cause for concern, however, the next phase of incremental development would be to introduce the DAW to begin the investigation of Research Question 2. Instead of the integrated prototype 1 and 2 applications, the control and audio functions would be separated and a DAW environment such as Pro Tools or Ableton Live would be used as the panning engine.

As stated in the methodology, this chapter uses MAX/MSP for the first and second prototypes to allow later comparison with alternative programming technologies in line with Research Question 3. The use of Max/MSP allowed a rapidly prototyped digital solution to be developed that successfully meet the tight timescale. In particular, the availability of the MAX external by Dudas allowed straightforward integration of the Wacom tablet.

6 Development of OctoPanner Version 2

By the end of 2005 OS X had successfully and reliably replaced OS 9 for most Mac-based musicians and each major release was adding new 'Core Services' for programmers, however, little information was available to music programmers and example code was sparse. This chapter begins with an overview of Apple's Carbon and Cocoa development environments and the OS X Core Services relevant to this research:

6.1 Mac application development

6.1.1 The Carbon development environment

To ease the software developer's transition from OS 9 to OS X, Apple created the Carbon environment, a collection of C and C++ functions and data structures that allow User Interface (UI) generation, file system and hardware interaction based on the original Macintosh Toolbox API (Inside Carbon: Carbon Porting Guide Apple 2001).

6.1.2 The Cocoa development environment

In 1996 Apple acquired NeXT Computer Inc, the company that was formed by Apple Co-founder Steve Jobs in 1988. Apple continued to develop the NeXTSTEP Unix based operating system which after a further 5 years of development became OS X. Apple renamed the programming environment developed by NeXT to 'Cocoa' and the NS prefix frequently found in Cocoa refers to its NextStep heritage. The programming language developed as part of NeXTSTEP was called Objective-C, this object-oriented superset of C continues to be the main language for most Cocoa programmers although Cocoa also allows the use of C++, Java and Applescript (Duncan-Davidson & Apple 2002 p11).

Although Apple continue to provide support for Carbon, they state that Cocoa based applications are becoming the norm for OS X and that Cocoa allows rapid creation of 'robust, full-featured Mac OS X applications' such as Apple's own GarageBand, iPhoto and Safari. *"Cocoa offers a rich collection of ready-made objects for your application's user interfaceDrawing and imaging, file system interaction, Internationalization, user preferences* (Apple 2006 p1)

6.2 OS X Core technologies

Before discussing some of the the Core Technologies in OS X, it is worth re-examining the final Requirements Specification resulting from literature review and survey of existing systems:

- i. The Toolset is to be Mac-hosted using Pro Tools and Ableton Live DAWs.
- ii. The Toolset development will include the use of emerging programming languages and IDEs for the Mac OS X platform.
- iii. The Toolset will integrate or develop alternative panning interfaces including joysticks and touchscreen controllers.
- iv. The Toolset will provide shape and breakpoint-based automatable trajectories via a cue-list.
- v. The Toolset will allow synchronisation with other applications as a master or slave
- vi. The Toolset will provide real-time 3D trajectory visualisation.

The next prototype of the **OctoPanner** application (version 2) will therefore:

- Work in conjunction with a DAW (Requirement 1)
- Be developed using Cocoa and Objective C (Requirement 2)
- Integrate external MIDI controller devices as per the MAX/MSP version (Requirement 3)

Considering this functionality in terms of Operating System requirements, **OctoPanner** Version 2 will require MIDI input/output services and a graphical User-Interface.

Requirement 4 will be met by the development of an additional application to generate shape-based trajectories. This application will be called **ShapePanner** and its development will be described in chapter 7, it will require an accurate timebase which will tie in with requirement 5 (synchronisation). In terms of Operating System requirements, **ShapePanner** will require MIDI input/output services, a graphical User-Interface and timebase/synchronisation services.

Requirement 6 will be met by the development of an additional application to provide real-time 3D visualisation of the positions of sounds as they are moved by the MIDI outputs of **OctoPanner** or **ShapePanner**. This application will be called **3D MIDI Visualiser** and its development will be described in chapter 8. In terms of Operating System requirements, **3D MIDI Visualiser** will require MIDI input/output services, a graphical User-Interface and 3D graphics services.

Having established the Operating System requirements for the three applications, The OSX Core Services can be evaluated for MIDI input/output, timebase generation, master/slave synchronisation, Graphical UI and 3D graphics generation.

6.2.1 CoreMIDI

Prior to the introduction of OS X there was no agreed standard for accessing MIDI, and audio hardware, sequencer manufacturers such as Opcode, Mark of the Unicorn (MOTU) and Steinberg provided their own low-level software interfaces. Apple's Core Services within OS X provide low-level to system facilities and in May 2000 Doug Wyatt was recruited by Apple to develop MIDI services for OS X to be named CoreMIDI. Wyatt was previously responsible for developing the 'Opcode Music System' (OMS) which later combined with Steinberg's timing engine to create OMS version 2 to create the 'Open Music System'. CoreMIDI is a unified high-performance mechanism allowing programmers to access MIDI devices in OS X. CoreMIDI is specified as a highly-accurate system with time-stamped MIDI messages that are transmitted or received with less than 1 millisecond latency and with jitter less than 200 microseconds.

6.2.2 CoreAudio Clock

One of the most challenging aspects of creating a MIDI or audio application is the creation of a stable and accurate timebase. Manufacturers such as Steinberg and Digidesign have implemented solutions based on different timing references meaning that 120BPM from one application or hardware sequencer/drum machine was not the same as 120BPM on another (Perron 1991 p2). Core Audio Clock is Apple's attempt at providing a unified timebase that would allow programmers to access the same clock. This clock could be internally generated or externally synchronised to an external master clock using MIDI Timecode (MTC) or MIDI Clock.

6.2.3 Core Graphics/OpenGL

As a result of the literature review and analysis of other available systems, Requirement Specification item 6 in Table 3-3 requires a 3D representation of the room and the motion of the sounds in that room. Integrated into OS X is OpenGL, an industry-standard 3D programming language that would allow the creation of the 3D room and sound motion representation. In addition to providing the programmer with Apple's Quartz 2D drawing system, the Core Graphics framework allows OpenGL, Quartz 2D and Quicktime elements to be composited onto the screen. The Quartz Composer application allows the programmer to graphically generate and composite these elements, the resulting 'composition' can then be integrated into a Cocoa application.

6.3 Cocoa implementation of OctoPanner v2

The author's initial evaluation of the Cocoa environment took place in December 2003 using OS 10.2 and Apple's Project Builder application, the implementation of Cocoa-based **OctoPanner** version 2 took place between July and October 2006 using Apple's successor to Project Builder, XCode. What became immediately apparent was the ease with which the graphical UI of an application could be generated using Apple's Interface Builder application and XCode.

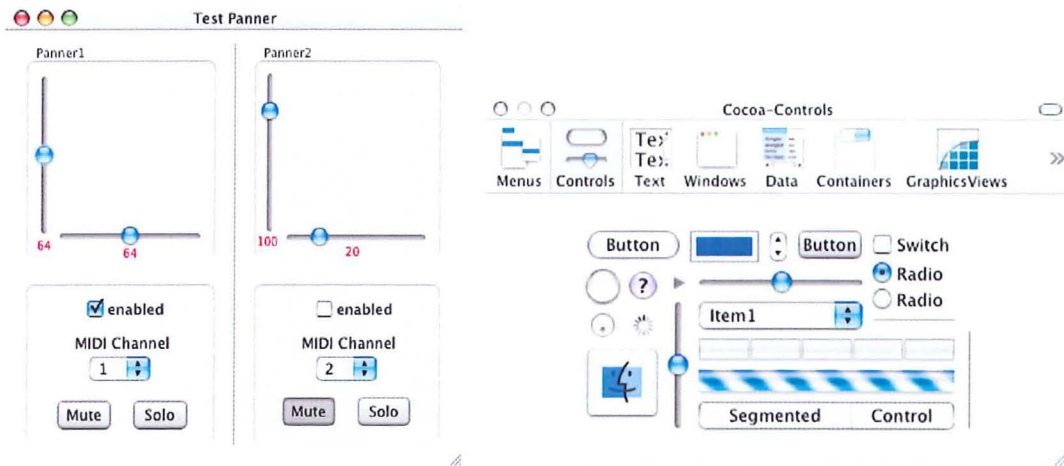


Fig. 6-1: TestPanner UI and Interface Builder palette

Fig. 6-1 shows the UI for a test application called TestPanner created using sliders, buttons, pop-up buttons and check boxes from the Interface Builder palette. The sliders were set to a 0-127 range and the red text fields set to the integer value of their corresponding sliders by control-dragging.

The next stage was to create code to respond to control changes in the UI, a class called **testController** was created based on the NSObject class² and three actions (methods) were added:

xChanged: // this action would be linked to the x sliders

yChanged: // this action would be linked to the y sliders

midiChannelChanged: // this action would be linked to the pop-up buttons

The **testController** class was instantiated in Interface Builder and links from the sliders and pop-ups control-dragged as shown in Fig. 6-2 below:

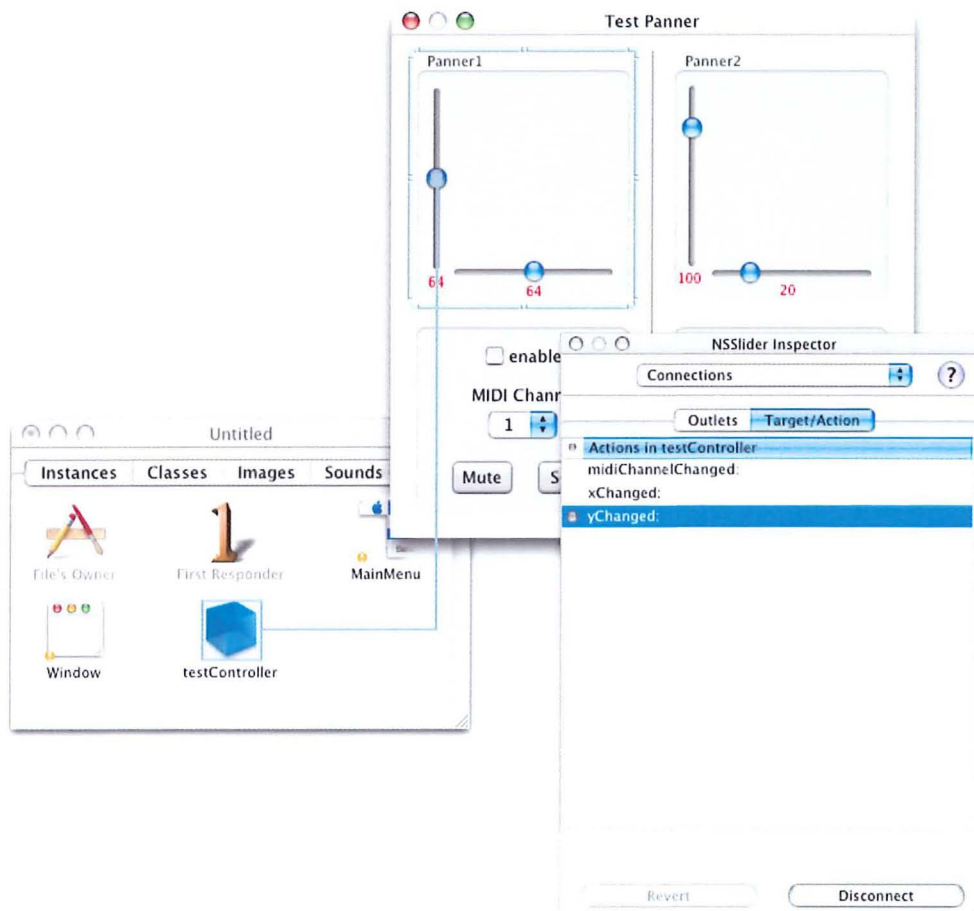


Fig. 6-2: Linking controls to corresponding actions

Next the **testController** header and implementation files were generated and standard C printf text display functions added to the methods called when the x or y sliders were changed:

² This is the main Objective C class, objects sub-classed from NSObject inherit the methods for creating, initialising and destroying themselves

This could then be used in the **TestPanner** application as follows:

```
- (IBAction)xChanged:(id)sender
{
    int newValue = [sender intValue];
    [PFMidi sendCC:2 data:newValue chan:1];

}
```

A CC message could now be sent with no knowledge of the underlying complexities of the Carbon MIDI interface or the actual construction of a CC message. At this point the TestPanner became a functional program with an Apple-compliant UI and MIDI functionality, however, development had taken weeks compared with hours for the same task implemented using Max/MSP.

6.3.2 Implementing the OctoPanner Version 2 application

The second version of **OctoPanner** essentially uses Cocoa to reproduce the eight-touchscreen panner concept developed in Max/MSP for the 2006 Sonic Fusion Festival event. Version 2 would include standard program functions such as menus and preferences to give the look and feel of a standard Mac application.

Although Cocoa provides palettes of the common User Interface items, it does not offer the rich collection of audio and MIDI-related UI objects provided by a music-specific application such as Max/MSP. Instead Interface Builder allows the programmer to add a 'Custom View' to the UI with an assigned custom class allowing the programmer to create custom controls such as the Max **Lcd~** object used in the original panner. Fig. 6-3 shows the .nib file (Next Interface Builder) main window for **OctoPanner V2** with eight Custom Views assigned to a custom **panView** class to be implemented later:

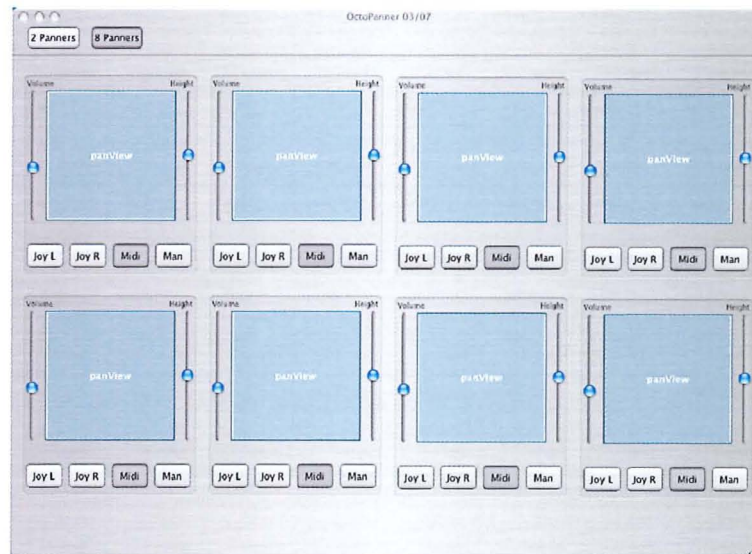


Fig. 6-3: OctoPanner v2 main window in Interface Builder

When the **OctoPanner** application is launched the Cocoa runtime will create eight instances of the **panView** class which will draw the **panView** custom controls into the spaces reserved by the custom views. The **panView** classes will then respond to mouse clicks and drags within the custom controls.

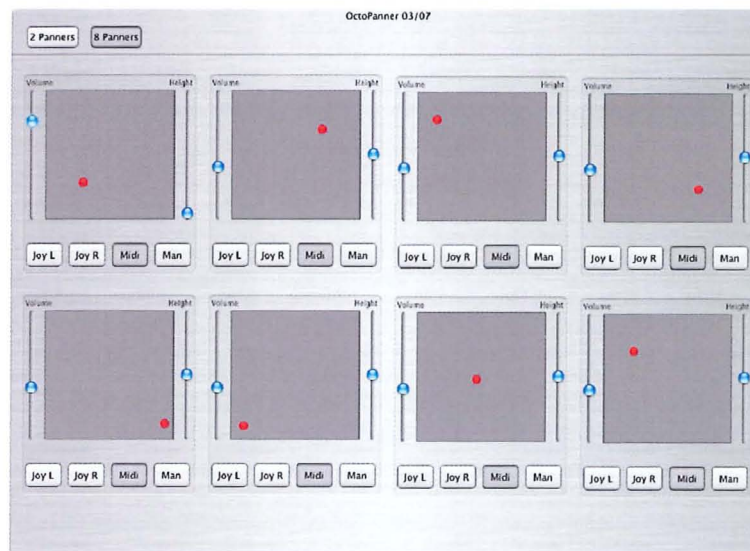


Fig. 6-4: OctoPanner v2 when running

The **panView** custom class is a subclass of the Cocoa **NSView** class and implements a **drawRect** method to draw the panner and **mouseDown** and **mouseDragged** methods to respond to user actions. The code for the **panView** class is based an Apple example

class called `dotView` modified to include scaling features to allow it to be drawn at different sizes.

To provide this application with MIDI output the **PFMidi** class implemented for the **TestPanner** application was reused, this clearly illustrates the advantages of object-orientated programming where each application generated would add to the collection of reusable classes and custom controls in the same way that Max/MSP has evolved. At this point (November 2006) **OctoPanner V2** was essentially a Cocoa equivalent of its Max/MSP predecessor and would remain in this form until feedback was received from performances in Spring 2007.

6.4 Chapter summary and research contribution

This chapter contributes to Research Question 3 and the use of alternative programming technologies to MAX/MSP. At this point MAX/MSP clearly wins in terms of learning curve and development time. This was mainly due to Apple's lack of direct MIDI support in Cocoa, however, the reusable **PFMidi** class would mean that development of subsequent MIDI applications would be much faster. It should be stressed that the functionality of the version 2 **OctoPanner** was no greater than version 1. In terms of IID Methodology this iteration simply took advantage of available technology (OS X + Cocoa) to provide a revised version of **OctoPanner** that now had an Apple-compliant user interface.

7 The ShapePanner application

7.1 Introduction

As stated in 6.2, This application will address the following Requirement Specification items:

4. The Toolset will provide 3D shape and breakpoint-based automatable trajectories via a cue-list.
5. The Toolset will allow synchronisation with other applications as a master or slave

This application will overcome the 2D and circular path restrictions of the Halaphon and the straight-line limitations of TRAILS and Matrix3. Requirement 4 above can be extended into the following design goal:

'A time-ordered list of shape primitives such as lines and circles plus height information used to control the 3D trajectories of sixteen sounds'

The development began with an exercise to investigate how such an application might look if implemented using Interface Builder and the Carbon or Cocoa UI palettes.

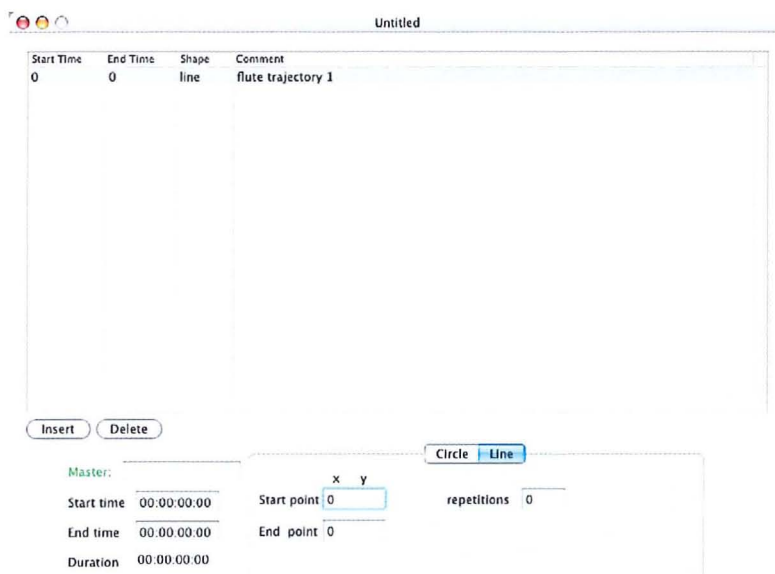


Fig. 7-1: ShapePanner prototype interface

The main element Fig. 7-1 is a 'Table View' configured in Interface Builder to have four columns with headings. This interface follows the Master-Detail database model (Anguish et al 2003 p689) wherein a Master list presents items in abbreviated form and the highlighted item is shown beneath the list in its complete form with all details visible. Each item in the list represents a sound's line or circle trajectory and has a start/stop times and shape parameters. It must be noted that this was simply a User Interface prototype and no classes and methods were written at this time.

7.2 ShapePanner implementation

Code generation for a fully working version began in September 2006, the design followed the commonly used Model-View-Controller (MVC) design model:

Model –an array of data objects representing each shape along with start and stop times. Each object is an instance of a **Shape** class written for this application.

View –the Master-Detail User Interface shown in Fig. 7-1

Controller – code written by the programmer to respond to changes in the View and update the model; conversely any changes to the model would be reflected in the View by the Controller.

7.2.1 Cocoa Bindings

Prior to OS 10.3³ a considerable amount of code was required to move data backwards and forwards between the View and Model, however with 10.3 Apple introduced a very powerful Cocoa feature called 'bindings' (Apple 2005 p1) along with **NSObjectController** and **NSArrayController** classes designed to ease the production of MVC applications. Apple's bindings allow an attribute of an object to be 'bound' to an attribute of another, for example, in Fig. 7-1 the value of the 'start time' text field could be bound to a variable called **startTime** within the currently selected item in the model, changing one would then automatically update the other.

To evaluate this binding mechanism the small Cocoa test application shown in Fig. 7-2 was produced consisting of a Table View with start and stop time columns along with a

³ Of the six commonly available Cocoa programming text books available in May 2008, only Hillegass (2nd ed. 2004) has been updated for OS 10.3. None of them address features introduced in OS 10.4 or 10.5

detail area comprised of two text fields with number formatters set to display times as xx.xx seconds.

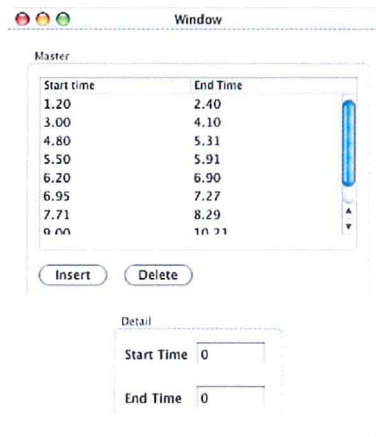


Fig. 7-2: Bindings test application

A class called **Shape** was created with variables called **startTime** and **endTime**. An instance of **NSArrayController** was added to the program and set to reference the **Shape** class just created. Keys **startTime** and **endTime** were then added to the **NSArrayController** instance via the inspector. The Insert and Delete buttons were assigned to the **insert:** and **remove:** actions in **NSArrayController**

To complete the program the detail start/end time text fields and the start/end time Table View columns were bound to the Array Controller as shown in Fig. 7-3:

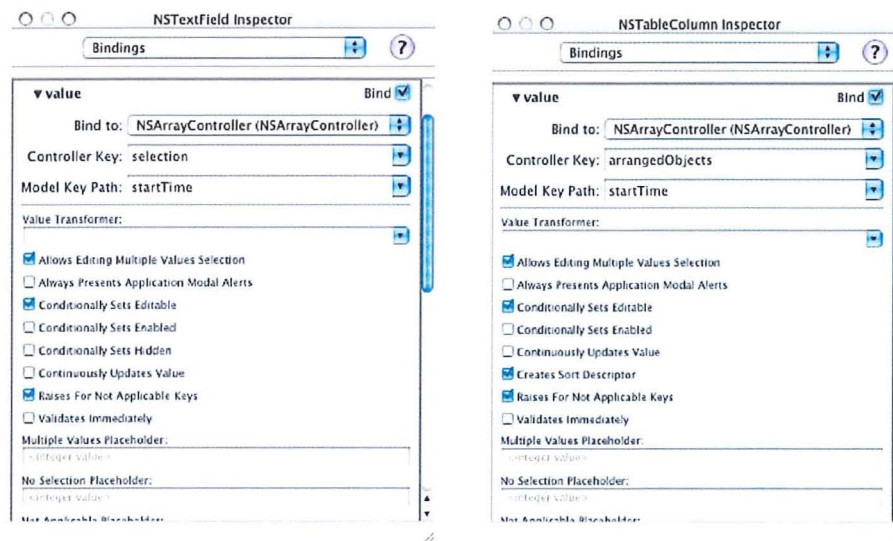


Fig. 7-3: Text Field and Table Column inspectors showing bindings

The only coding required was to implement the **Shape** class that would represent the trajectory shape object in the list. Having established the success of the bindings mechanism, this test application was used as the starting point for the **ShapePanner** program.

7.2.2 The ShapePanner classes

The application implementation can be broken down into the following classes:

shapeController the MVC controller that will respond to incoming time messages and to changes in the UI that require program action

Shape the model will be an array of **Shape** objects, each with a start and end time plus shape parameters. The **Shape** class will have methods for generating coordinates based on a current time supplied by the **shapeController** class

PFMidi this class extends the **PFMidi** class already described and adds MIDI Time Code support to the initialisation and MIDI input and output methods

Additional classes would be required to provide custom UI elements to allow the user to specify the line and circle parameters graphically rather than the numerical method in Fig. 7-2.

7.2.3 Timing/scheduling

Critical to the application is an accurate timebase and a means of scheduling the conversion of a **Shape** object's parameters into MIDI based coordinates that create the trajectory that it specifies. For example, if the user adds a line object to the list that starts at one second and ends at two seconds then the application must output the line's start coordinate when the timebase reaches one second and then continue to output interpolated points at a specified interval; finishing with the line's end coordinate when the timebase reaches two seconds. Any inaccuracies in the timebase and the time interval between points will result in jitter in the sound's positioning. Cocoa provides an **NSTimer** class that allows the creation of single or repeating timer events but Apple

states that the timing accuracy is dependent on CPU load and thus **NSTimer** has a resolution of the order of 50 to 100 milliseconds and is therefore unusable as a timing reference.

Core MIDI on the other hand has a stated latency of less than 1 millisecond so it was decided that external MIDI Time Code (MTC) would be used as the timebase and that the initial version of **ShapePanner** would slave to external timecode generated by Pro Tools. The Pro Tools timecode would be used to schedule the conversion and transmission of trajectory coordinates. The MIDI Specification (MMA 1983) specifies MTC as a series of eight quarter-frame messages representing a complete SMPTE frame every two frames, therefore a regular timing pulse can be obtained from these messages in multiples of 10ms (assuming European 25 FPS frame rate).

50 updates per second was chosen for the output rate of the x,y,z positional information generated by **ShapePanner**. This figure was based on the 20mS sampling frequency chosen by Soundcraft for their DC2000 series console automation after considerable research by the company. (Soundcraft 1997 p29). The 200 microsecond maximum jitter for CoreMIDI thus represents a 1% maximum timing error.

7.2.4 Representing time

As timecode was to be used as the timing reference, standard SMPTE hh:mm:ss:ff representation of time was chosen for the shape start and stop times in the UI. An integer rather than floating point representation of elapsed time was used based on 50Hz timer 'ticks', for example a start time of 00:00:03:01 (3 seconds and 1 frame) at a frame rate of 25FPS (2 ticks per second) would be represented by the integer value $3 \times 50 + 2$.

The standard Cocoa number and date/time formatters available on the Interface Builder palette do not allow timecode display so a custom formatter **PFSimpleFormatter** was sub-classed from **NSFormatter** (Apple 2006 p2). The `stringForObjectValue` and `objectValueForString` methods were overridden to convert the integer 'ticks' representation of timecode into a hh:mm:ss:ff string and to convert a hh:mm:ss:ff input by the user in to the integer representation.

PFFormatter was added to the start/end text fields and table columns programmatically:

```
[[startTimeColumn dataCell] setFormatter:PFSimpleFormatter]; // tell the data cell of the
startTime column to perform the setFormatter method
```

Only MTC quarter-frame decoding was implemented since analysis of the Pro Tools timecode output showed it did not send full-frame messages when its transport was started. Frame rate support was limited to the European 25 FPS as it was anticipated that **CoreAudio Clock** routines would eventually be used once example code became available.

When the **PFMidi** class has decoded a complete SMPTE frame it sends the `updateTimecode:` method to the **shapeController** class who then updates the current time field in the UI. A check is then made to see if the new current timecode value falls within the start and stop time of any shapes in the time-ordered list using the **Shape** class `timeInRange` method:

```
-(bool)timeInRange:(int)currentTime // tests to see if supplied time is between the
shape's start and end times
```

If a shape in the list returns YES then **shapeController** sends that shape the `getPointForTime:` method:

```
-(NSPoint)getPointForTime:(int)currentTime // returns an x,y coordinate corresponding
to supplied time
```

At this point (November 2006) the feasibility of the **ShapePanner** application had been successfully demonstrated although shapes were limited to lines and part of the UI was textual rather than graphical. Work was also needed with regard to timecode support and the generation of point information based on the timecode value.

7.3 Implementing CoreAudio Clock support

One of the most challenging tasks for a programmer producing a MIDI or audio application is the design of a highly accurate timebase. Rather than individual programmers providing their own implementation, Apple's **CoreAudio Clock** offers developers a universal source of timing information for OS X audio/MIDI programs, thus improving compatibility between manufacturers and significantly reducing the amount of code to be written to fully support common application features such as timecode rates, offsets, tempo and time signature maps.

CoreAudio Clock was introduced into OS 10.4 in April 2005 with minimal documentation, by April 2008 a Google search for “CoreAudio Clock” only revealed two Apple documents plus an Apple forum query posted by the author of this thesis. Despite occasional requests by programmers on Apple’s Core Audio forum for more information and example code, to date none has been forthcoming. After proof of concept of the **ShapePanner** program in chapter 7, work began to generate test code to investigate Core Audio Clock based on the available documentation, this work is described in Appendix C.

7.3.1 Creating an accurate scheduler for ShapePanner

The use of an external MIDI timebase such as MTC as the **ShapePanner** positional reference also gives a timing reference since the incoming quarter-frame messages can be used to synchronise the transmission of shape data. However, if the **ShapePanner** is to be able to act as a timecode master, then an internally generated periodic timer event with greater accuracy than the Apple NSTimer class is required. Since a powerful feature of **CoreAudio Clock** is its ability to generate accurate MTC or MIDI clock based on the selected timebase, the author hypothesised that this could be combined with the **CoreMIDI** timestamping mechanism to give the scheduling solution that follows:

If **CoreAudio Clock** could be set to generate MTC, and this MTC was wrapped back into the application via a separate virtual input, the application’s MIDI callback procedure would receive accurate quarter frame pulses based on the selected internal timebase and frame rate. The **ShapePanner** application could use these pulses to calculate timestamps for any shape messages to be sent during the next quarter frame; **CoreMIDI** would then transmit those MIDI messages at the appropriate times based on the timestamps. To investigate this MTC loopback technique, MTC output was added to a test application and the **NSTimer** display update scheduler was successfully replaced by the quarter frame ‘ticks’ synthesised from the internally generated timebase.

Having established that using **CoreAudio Clock** was viable and offered significant advantages to a programmer, **ShapePanner** was rewritten to incorporate this new

technology as its timebase and scheduler, Fig. 7-4 shows the rewritten UI with an added transport bar at the bottom of the screen supporting internal and external clocking.

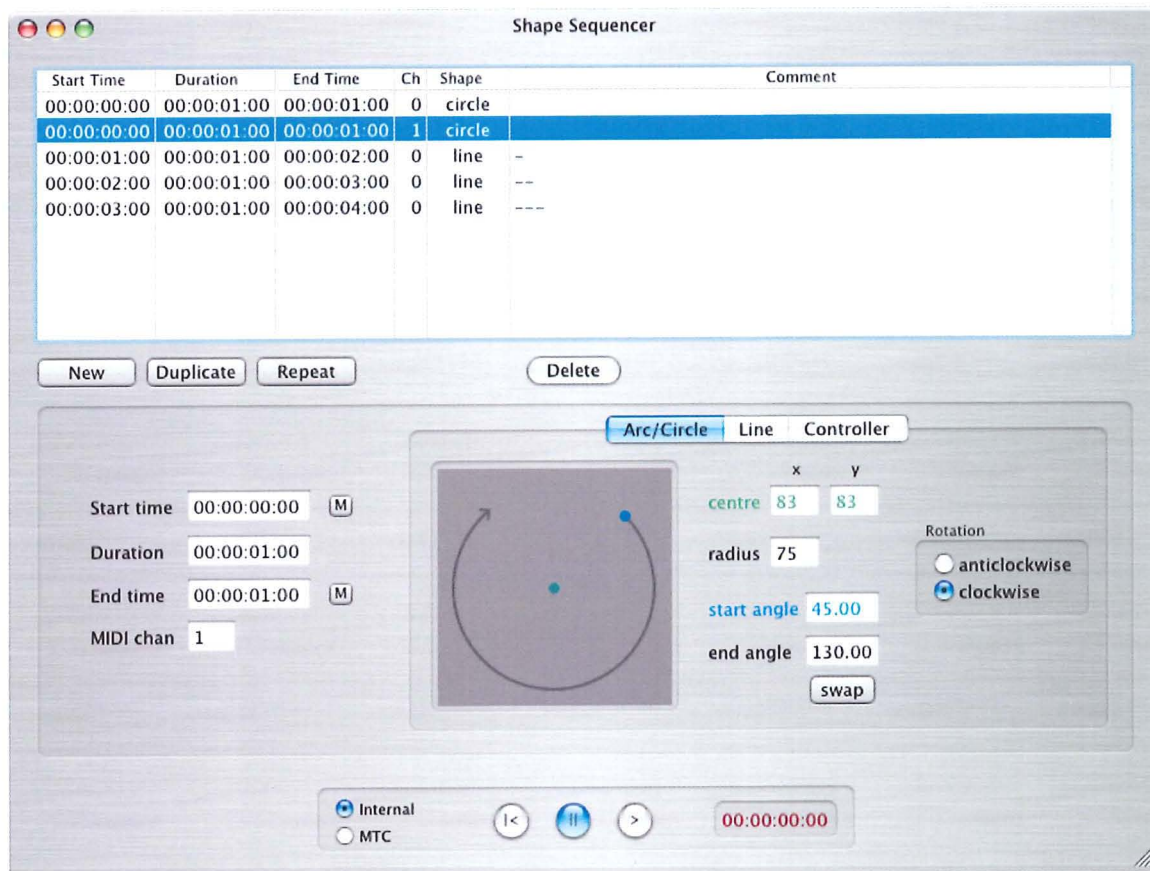


Fig. 7-4: ShapePanner incorporating CoreAudio Clock

7.4 Chapter summary and research contribution

Unlike the implementation of OctoPanner using Cocoa compared with MAX/MSP in the previous chapter there are clear gains here from using Cocoa/Objective C. Much of the development such as the UI and master-detail table was achieved graphically using Interface Builder and Apple's bindings feature rather than lines of code. However, the same cannot be said of CoreAudio Clock, its lack of Cocoa class support means using the low-level Carbon function calls and callbacks. The documentation is extremely minimal and there is no example code available. This is unfortunate since development of ShapePanner shows that CoreAudio clock is a powerful timebase and synchronisation engine hence the inclusion of Appendix C in this thesis to considerably extend the documentation available to other researchers and to provide the first example code.

If we examine this chapter in terms of Research Question 3: "How can emerging programming technologies offer creative alternatives to the MAX/MSP or hardware-based tools commonly used for sound spatialisation?" it can be strongly argued that Cocoa's features such as the master-detail database and MIDI timecode synchronisation support have led to an application whose features would be very difficult to implement in MAX.

8 3D MIDI Visualiser application

8.1 Introduction

As stated in 6.2, the development of this application as part of the Toolset will address the following Requirement Specification item:

6. The Toolset will provide real-time 3D trajectory visualisation

It became evident during the development and testing of **OctoPanner** and **ShapePanner** that such a visual representation of the trajectories of sounds would be useful when access to a performance space and speaker array was not possible. **3D MIDI Visualiser** is a standalone Cocoa application controlled by assignable MIDI messages to allow use with any MIDI panning source.

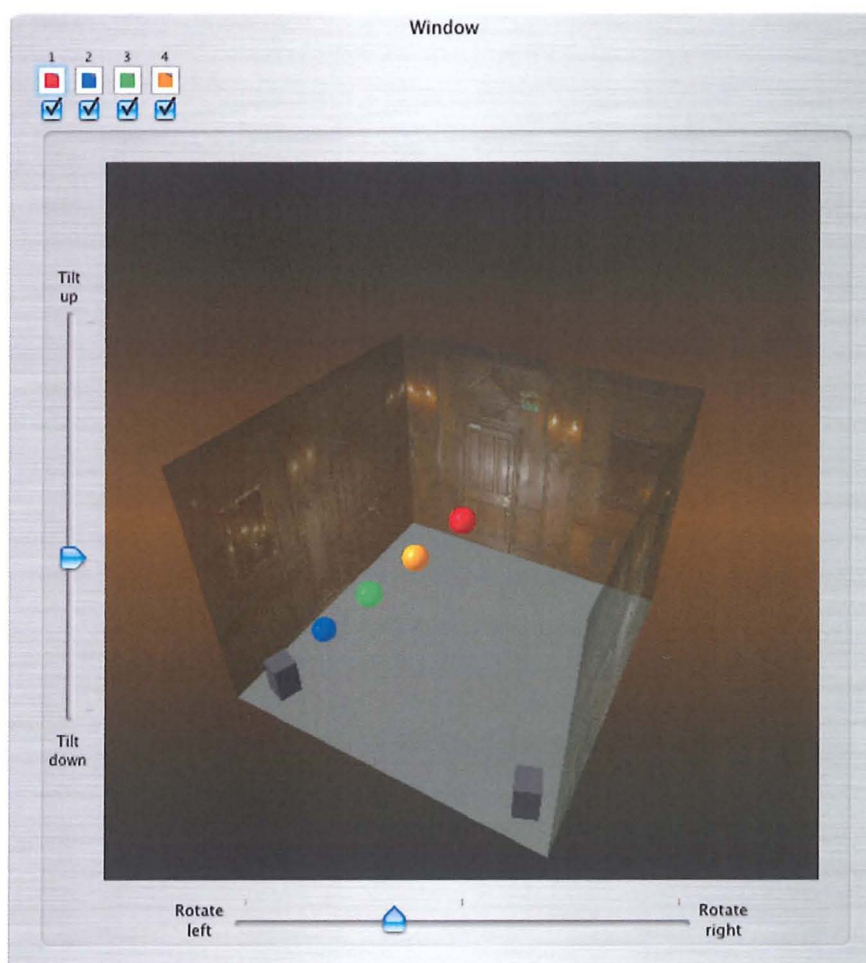


Fig. 8-1: 3D MIDI Visualiser Cocoa application

8.2 Apple's Quartz technology

Key to the **3D MIDI Visualiser** application is the Quartz Composer 'composition' shown in Fig. 8-2. Apple's Quartz Composer tool introduced in OS 10.4 allows the programmer to exploit the Mac's 2D and 3D capabilities without knowledge of the OpenGL graphics open standard. The user can create a composition from elements such as Lighting Environment objects together with sphere and graphics sprite renderers that can be placed, manipulated and externally controlled.

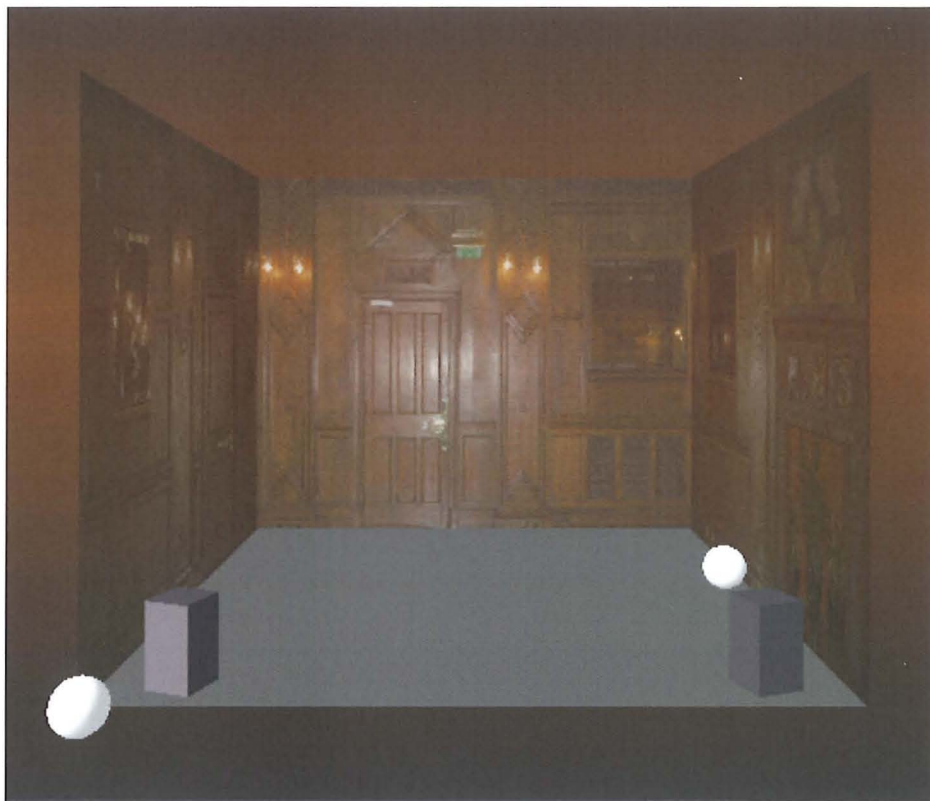


Fig. 8-2: Quartz Composer 3D MIDI Visualiser panel

Fig. 8-3 shows the **3D MIDI Visualiser** quartz composition viewed in Quartz Composer, this composition contains the following elements:

Four sphere renderer objects with external control of their x,y,z coordinates and colour to represent the moving sounds

Two loudspeakers added to investigate the cube rendering capabilities of Quartz Composer

Four graphics sprite objects using photographs of Napier University's ornate Turmeau Hall to represent the three walls and floor

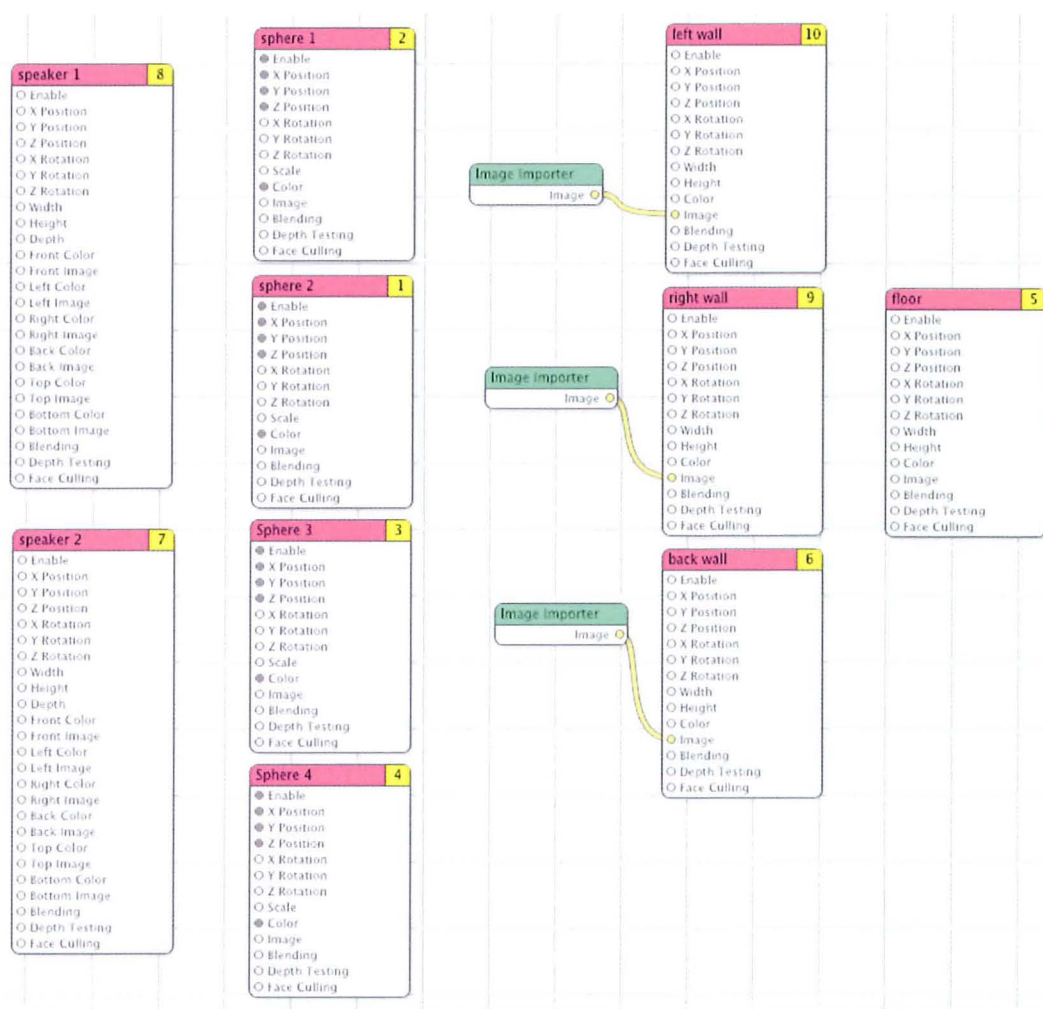


Fig. 8-3: Quartz Composer renderer objects in 3D MIDI Visualiser

The Quartz composition is hierarchical. The renderer objects are assembled inside a **3D Transformation** object to allow the virtual room can be viewed from any angle. This object is itself inside a lighting environment object.

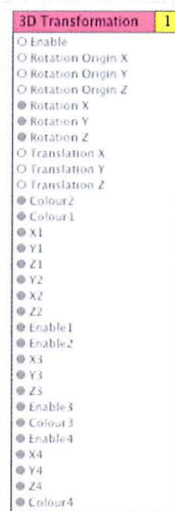


Fig. 8-4: 3D Transformation with published inputs

Fig. 8-4 shows the **3D Transformation** object encapsulating these rendering objects with its own published **x, y** and **z** rotation parameters. Note that the **enable, x,y,z** and **color** parameters of the four sphere renderers are visible indicating that they were 'published' in the layer below.

Although Quartz Composer has a MIDI controller object, incorporation of the QC composition into a Cocoa application allowed a UI with control over the view angle and the number of 'sound' spheres and their colours as shown in Fig. 8-1. To achieve this, a Quartz Composer composition was incorporated into Cocoa using Interface-Builder. The UI elements were then bound to the published inputs in the composition. In Fig. 8-1 the **3D Transformation** object encompassing the room is seen rotated slightly down and to the left via the tilt and rotate sliders and colours have been assigned to the sounds using the OS X inkwell controls towards the top of the window.

8.3 Chapter summary and research contribution

The graphical nature of Quartz Composer allowed rapid prototyping and experimentation leading to the 3D representation of the room. Although the Quartz Composer composition was incorporated into a Cocoa application and thus required Objective C code, a more limited version of 3D MIDI Visualiser could have been generated purely graphically using just Quartz Composer.

It should be noted that a 3D realtime visualiser was not an initial research aim and instead resulted from a research gap shown up by literature review. It nevertheless adds weight to Research Question 3 "How can emerging programming technologies offer creative alternatives to the MAX/MSP or hardware-based tools commonly used for sound spatialisation?"

9 Foot control of spatialisation

9.1 Introduction

As stated in chapter 5, Sonic Fusion 2006 Guitarist Haftor Medboe had requested a device to allow him control of panning as he played. The literature review and survey of current systems in chapters 2 and 3 revealed only one commercially available panning controller, the 'Bat', that offered an alternative means of control to joystick and touchscreen. As none of these devices could be used as the guitarist played, research began in November 2006 into a means of foot control of spatialisation. As a starting point the single-axis volume or wah pedal familiar to guitarists was considered. Although this gave an intuitive representation of front/back sound position, an effective means of left/right control was required. A simple mechanical prototype was assembled in November 2006 with conventional front/back rocking action plus a left/right tilt as per Fig. 9-1:

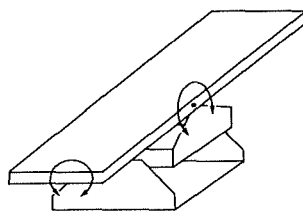


Fig. 9-1: Dual-axis foot-controller

Shortly after this test, literature review uncovered a 2003 patent filed by Lester Ludvig that related to the pedal-steel guitar and describes a “*simultaneous single-foot adjustment of a plurality of continuous range parameters*” using a volume pedal concept with a rotating top element shown in Fig. 9-2:

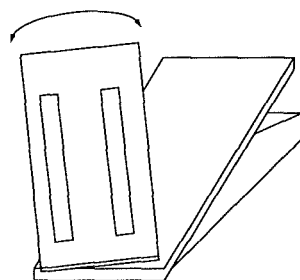


Fig. 9-2: Ludvig foot controller patent

The Fig. 9-1 prototype and a Fig. 9-2 mock-up were tested by four Edinburgh Napier University guitar-playing members of staff and none considered either controller intuitive, in addition, both controllers were criticised for requiring awkward and uncomfortable ankle movements.

To avoid this unnatural ankle movement, the author's third controller proposal was tested by the four guitarists. Fig. 9-3 shows a musician standing on a flexibly mounted foot-plate, the musician would shift his/her weight from the left leg to the right to represent left/right panning, and from ball of foot to heel to represent front/rear panning. Detection of panning position would be via strain gauges or force sensing resistors in each corner. Of the three devices tested, this prototype was preferred by two of the guitarists but the remaining two found it hard to balance, although the concept felt intuitive.

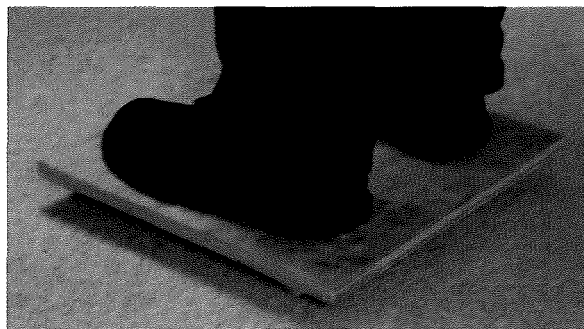


Fig. 9-3: Weight distribution based panner evaluation

The evaluation of the author's fourth and final foot controller design took place in December 2006 following some unrelated experimentation with computer pointing devices. Fig. 9-4 shows a low-friction 'puck' on a highly polished surface and was rated as intuitive and easy to manipulate by the testers. Two observations were made, firstly that the area of the baseboard should be reduced and secondly that the movement of the puck should be mechanically constrained to the board.



Fig. 9-4: 'Puck' based panner evaluation

Having selected the fourth controller type for further development, the resistive, capacitive and infra-red LED based technologies used for touch screens were considered before choosing the electromagnetic induction technique used by Wacom for pen tablet position detection (Wacom 2006 p1).

9.2 Foot Puck Prototype

A Wacom sensing coil and circuit board were mounted inside a plastic puck constructed using 100mm diameter PVC with a low-friction base and a non-slip top surface.

The position-sensing PCB was mounted in a routed recess in the baseboard and covered with high-density foam board. The assembly was completed by a constraining rectangle that allowed the puck to traverse to the edge of the tablet in each direction. A low-friction rubber/felt lamination was later fitted to the edge of the constraining rectangle to reduce the impact noise when the puck hit the edge. Fig. 9-5 shows the completed prototype.

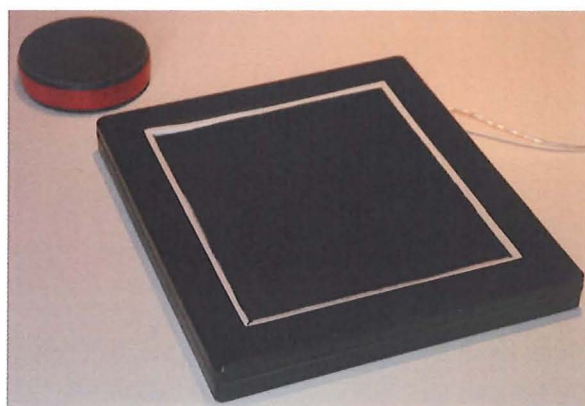


Fig. 9-5: Prototype Foot Puck panner

To decode the USB output from the Wacom board a simple Max/MSP patch was written using the OS X version of the Wacom external used in section 5.6. This patch converted the tablet positional information into MIDI which could then be used to control any of the eight panners in the **OctoPanner v2** application.

The **Foot Puck** was tested in the two March 2007 concerts to be described in chapter 10. To allow concert use of the **Foot Puck** its USB output was extended to the mix position using a USB to RJ45 transceiver pair connected by a 30m cable. Unlike the OS 9 Wacom driver and MAX/MSP Dudas external used in section 5.3, the OS X Wacom driver preferences pane options did not allow a Max patch to run in the background without the tablet movements controlling the screen cursor. As a workaround, the Max patch was hosted on a separate laptop whose MIDI output fed the **OctoPanner v2** application running on its own Mac.

9.3 Chapter Research Contribution

If we first consider the physical user-interface alternatives to be investigated by Research Question 1, the Foot Puck shows that there is scope for development for additional controllers beyond those available in the music and film post-production industries or in existing spatialisation systems. Now considering the use of emerging programming technologies to be investigated by Research Question 3, to implement the Foot Puck in Cocoa would be complex and would require writing low-level code to interface to the Wacom Human Interface Device (HID). Implementing the Foot Puck in MAX/MSP was very straightforward since the low-level code already existed in the form of an 'external' by Richard Dudas (Cycling '74 2007 p1).

10 Concert use of the version 2 live Toolset

In September 2006 a meeting took place to re-examine the May 2006 Sonic Fusion performance using **OctoPanner** v1 and to agree concert dates to trial version two of the Toolset. Two dates in March 2007 were agreed with the Haftor Medboe Group⁴ for surround sound concert performances in Glasgow and Edinburgh. Medboe stressed that a re-occurrence of the almost total failure of the Max/MSP-based **OctoPanner** system during the Sonic Fusion concert would be unacceptable to the concert promoters and a solution avoiding Max/MSP was requested. This offered an opportunity to research the use of a DAW to perform the spatialisation in line with Research Question 2 “How can a DAW be used as an alternative to dedicated panning hardware?” Ableton Live was chosen for the two concerts because it has been developed specifically for live use and also offers strong effects options, it would act as a panning and effects engine controlled by the MIDI panner output from **OctoPanner** v2.

10.1 Developing a spatialisation technique using Ableton Live

A restriction placed on the March 2007 concerts was that both venues were only prepared to re-configure for quadraphonic and not six or eight speaker playback, hence only a quad panning solution needed to be found for Ableton Live. Although this application does not have surround sound support it does feature flexible MIDI mapping of screen controls to external controllers. The output from **OctoPanner** V2 is a series of x,y,z points sent as MIDI CC messages where the MIDI channel indicates which of eight screen panners have been moved; the following scheme was developed to use these CC messages to perform quad panning using Ableton Live:

⁴ The Haftor Medboe Group are a jazz quartet with a strong interest in the incorporation of emerging technology, their second album was released by Linn Records in 2006 in multi-channel SACD format.

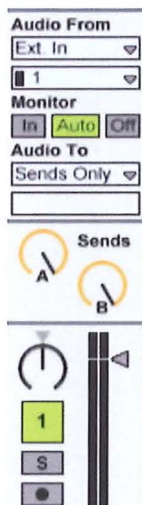


Fig. 10-1: Ableton Live audio track

Fig. 10-1 shows an Ableton Live audio track with a mono external input and its output set to the stereo sends busses A and B. The pan control in this figure has been mapped to the x-axis of the screen panner in **Octopanner** so that the external audio input will be left/right panned identically into the A and B stereo send busses. The implementation of Y-axis control from **OctoPanner** is less straightforward, in the Fig. 5-1 passive panner the x-axis panners were preceded by a y-axis panner feeding mono front and rear busses. As Ableton Live does not have track output busses, the following solution was used:

The **OctoPanner** application was modified to allow it to optionally send out three CC messages to represent a point:

X position: 0 – 127 MIDI CC message

Y position: 0 – 127 MIDI CC message

Reverse Y position: 127 – 0 MIDI CC message

The modified **OctoPanner** MIDI setup panel is shown in Fig. 10-2, this panel allows the user to specify which MIDI CC numbers are to be used for x-axis and y-axis plus reverse y-axis if enabled via the check box.

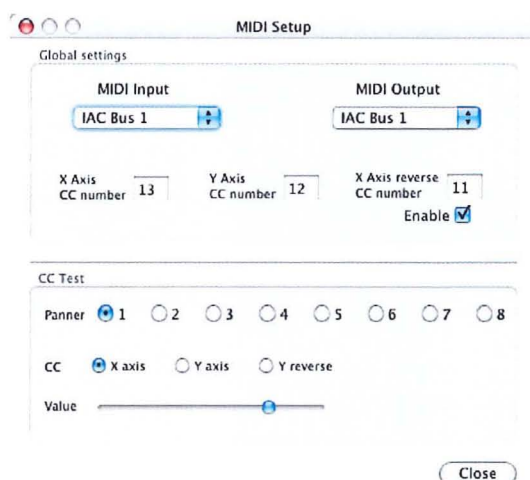


Fig. 10-2: OctoPanner v2 modified MIDI Setup panel

If the y-axis MIDI CC is mapped to Send A in Fig. 10-1 and the reverse y-axis is mapped to Send B then Sends A and B become front and rear busses respectively. When the touch panner is set to full front output ($y = 127$) the front bus send control will be 0dB (fully clockwise) and the rear bus send control will be fully attenuated (anticlockwise). Send A and B will move in opposite directions as the **OctoPanner** y-axis output moves towards the rear ($y=0$).

To map the front and rear busses to physical speakers the send bus return tracks A and B are assigned to output pairs 1-2 and 3-4 as shown in Fig. 10-3

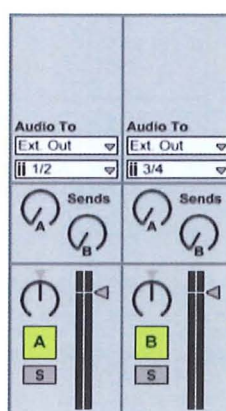


Fig. 10-3: Ableton return A and return B routing

Although the concert requirements for March 2007 were quad, this approach could be extended to include centre speakers between the four corners by modifying **OctoPanner** to subdivide the x axis and/or y axis into 3 or more and output the corresponding number of CC messages. This is explored further in chapter 12.2.

10.2 Concert 1 – Haftor Medboe Group, ‘The Arches’ Glasgow 10-3-2007

The first surround sound concert was to take place in The Arches, a Glasgow arts performance venue. A technical meeting was arranged with the Arches’ senior engineer Phil Zambonini. It was agreed that Zambonini would mix the band as a stereo performance and also manage the floor monitor mixes. The author assisted by Dave Hook from Edinburgh Napier University would take instrument feeds from the Front Of House (FOH) console, add delays and filtering and spatialise the performance into the rear speakers. Although the venue promoter had enthusiastically welcomed the surround sound aspect of the concert and stated that the audience would be seated around a central stage to maximise the effect, no such arrangements were made and the band were obliged to perform in their standard concert layout shown in Fig. 10-4:

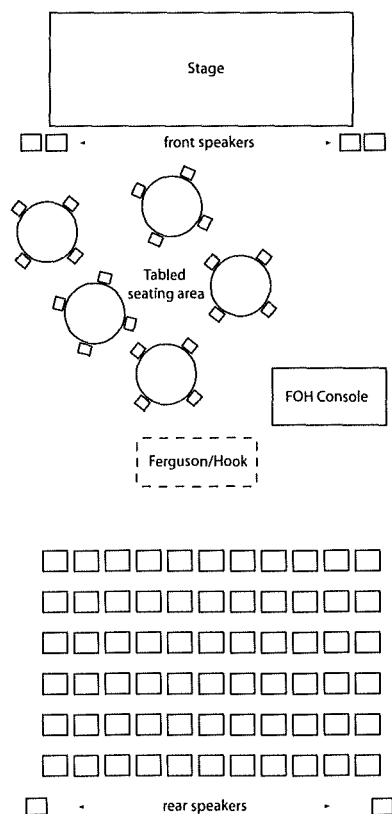


Fig. 10-4: Concert layout for ‘The Arches’ (Glasgow)

It can be seen in Fig. 10-4 that some of the audience are very close to the front speakers and a large distance from the rear speakers. The reverse is true for people choosing to sit towards the rear of the raised seating area. The direct result of this was that any quieter spatialised rear sounds were inaudible to those sitting close to the stage and that the author had to take great care not to overwhelm those nearer the back with loud rear sound. Notwithstanding this, the concert was successful and although the spatialisation was subtle, several members of the audience made favourable, enthusiastic comments.

10.2.1 Concert 1 Evaluation

The combination of **OctoPanner**, the **Foot Puck** and Ableton Live worked very well and was reliable. Zambonini has since made contact regarding follow-up performances geared specifically to surround sound. Concert use of the Toolset also showed that the following modifications/additions to **OctoPanner** were desirable:

Layout

- Under the pressure of the concert the two-by-four panner layout shown in Fig. 6-4 proved to be confusing when moving between the touchscreen and the eight parallel fader Behringer BCF2000 MIDI controller since there was no correlation between them.
- Adding touchscreen mutes and solos would reduce the need to move between the BCF2000 and the touchscreen.
- On-screen labelling equivalent to the white tape 'scribble strip' used by live sound engineers was required to make it clear which panner corresponds to which instrument on stage.

Snapshots

- Possibly the most significant addition to OctoPanner would be a means of storing snapshots of the panner, mute and fader positions determined for each song during the soundcheck/rehearsal.

As there were 7 days between the two concerts, only the simplest of these modifications could be added, namely 'scribble strip' text fields were added to each panner so that the

user could identify them (as 'guitar', 'sax' 'darabuka' etc). The other modifications were later incorporated into **OctoPanner** version 3 described in chapter 11.

10.3 Concert 2 – Haftor Medboe Group, 'The Bongo Club' 17-3-2007

Edinburgh's Bongo Club allowed a more flexible audience arrangement with the audience seated and standing around a central mix position facing the stage. Rear speakers were placed in the corners of the room facing towards the mix position. This centrally placed audience afforded a more adventurous use of surround mixing and the event received a large number of positive comments.

10.3.1 Concert 2 Evaluation

Notably, Medboe found the use of the **Foot Puck** to be intuitive and he particularly enjoyed using it in his improvised sections. Again although incorporation of scribble strips had improved the user experience, the two-by-four panner screen layout was still found to be confusing under pressure and the lack of snapshots was again noted. One additional functional requirement for **OctoPanner** was highlighted by the Edinburgh concert: the design is based on the panning of eight mono sources and although that was largely the case with the Haftor Medboe Group's setup, a stereo microphone pair had been tried on the glockenspiel to good effect. As the touchscreen was a single touch device the left and right glockenspiel faders could not be moved in unison and a requirement was therefore noted to allow two adjacent panners to be linked as a stereo pair.

10.4 Chapter research contribution

The testing allowed by the two concerts gives information relating to two of the Research Questions:

10.4.1 Question 1 - Alternative User-Interfaces

The touchscreen proved highly effective and intuitive and offered clear feedback to the user on the position of the eight sounds. The single-touch nature of the touchscreen was a limitation however. The **Foot Puck** satisfactorily answered the request for a panning device made by Medboe after the 2006 Sonic Fusion performance. This requirement for a musician's live spatialisation controller highlights the fact that the available tools

reviewed in Chapter 3 place most emphasis on control by the spatialising/diffusing engineer.

10.4.2 Research Question 2 - DAW Hosting

The successful use of Ableton Live as the spatialisation engine shows that a case can be made for a DAW as a practical alternative to a hardware matrix mixer for quadraphonic works. It should be noted however that the use of bus sends for eight speakers or more will be considerably more complex.

11 OctoPanner Version 3

Although the layout changes discussed in 10.2 would be relatively straightforward to implement, snapshot save and recall required a major change to the program architecture. For the new version of **OctoPanner** the Model-View-Controller paradigm was used and the panner processing was separated from the view controller code to form a **Panner** class. To incorporate the stereo linking requirement in section 10.3, the **Panner** class was designed to represent a stereo object with left and right x,y and z (height fader), mute, solo and volume Keys⁵. Having created a stereo **Panner** class with these parameters, the eight sets of UI controls could then be bound to one of four stereo **Panner** instances.

11.1 Saving and loading snapshots

Apple provides Cocoa with the **NSCoding** mechanism (Hillegass 2004 pp152-155) for storing the states of objects. To implement this protocol the required `encodeWithCoder:` and `initWithCoder:` methods were written for the stereo **Panner** class. When the user selects save from the File menu each of the four stereo **Panner** instances will be asked by the Cocoa run-time system to encode its Key variables into a stream of bytes. This process is reversed when a file is loaded and each **Panner** object is asked to initialise its Key variables from the supplied byte stream. To allow more than one snapshot per file, an array was added to the main controller class to hold ten versions of the four stereo **Panner** instances. Selecting a snapshot chooses which four of the 40 **Panner** instances are connected to the View (the UI). The tab control at the top of the main window shown in Fig. 11-1 shows that the Snapshot tab is currently selected and thus the 10 snapshot buttons are visible, all 10 snapshots are saved when the user executes the 'save' command.

⁵ In Cocoa a Key refers to the name assigned to a variable, for example `leftVol`, the Key-Value Cocoa methods allow these variable to be accessed from outside a class.

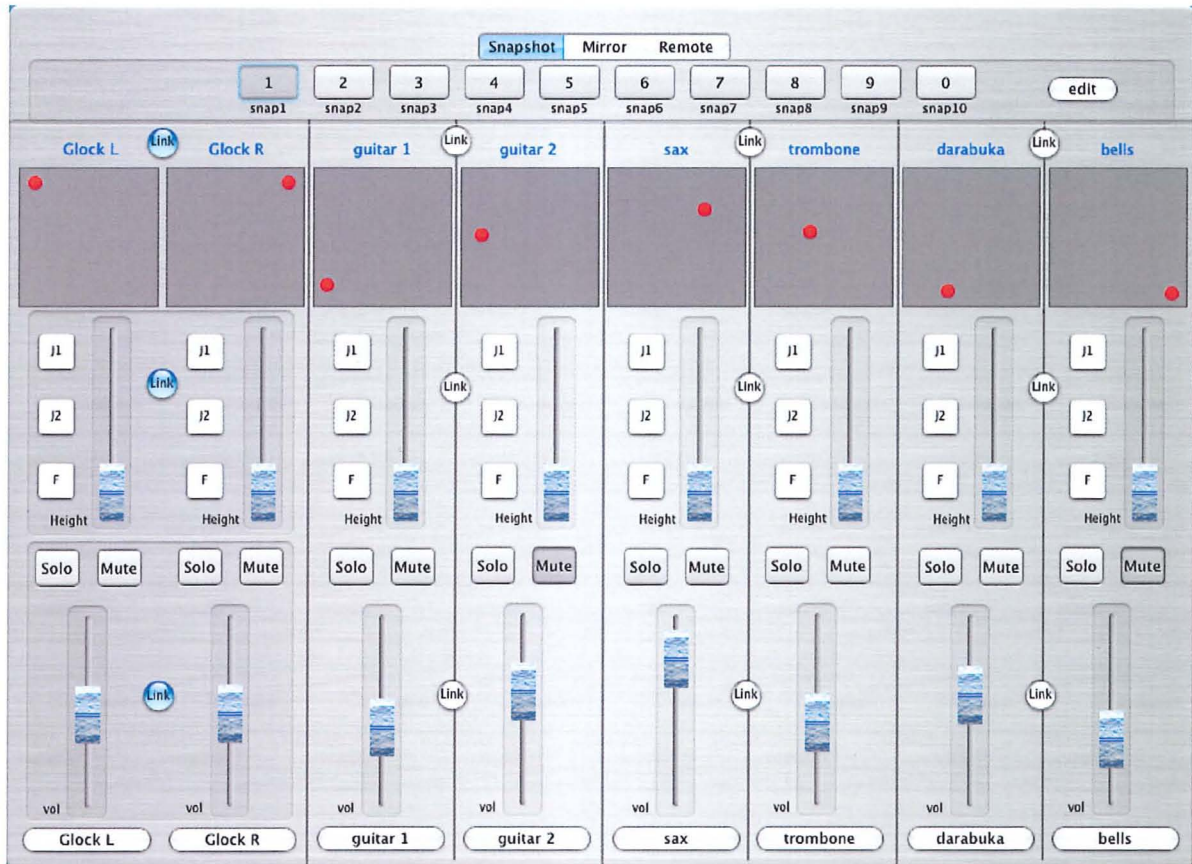


Fig. 11-1: OctoPanner version 3 main window

11.2 OctoPanner v3 features

The two-by-four matrix of panners in versions 1 and 2 (Fig. 6-3) has been replaced by eight vertical 'channel strips': each channel has an x-y panner, height slider, solo, mute and volume fader. Between each pair of channels are 3 link buttons that allow the pan, height and volume sections of adjacent panner pairs to be linked. Note that panners 1 and 2 are linked in Fig. 11-1 which has resulted in the separating line between them being replaced by linking boxes to indicate stereo linking. Next to each height slider are J1, J2 and F buttons that place that channel under remote control by Joystick1 or 2 or by Foot Puck. These are configured as 'radio buttons' so selecting control of a channel automatically de-selects the others.

11.2.1 OctoPanner v3 panels

Although the primary elements in the main window have now been covered, some key features are still be addressed:

In addition to the link buttons on each channel strip, there are three 'mirror' buttons per channel pair that are revealed by selecting the 'mirror' tab at the top of the main window.

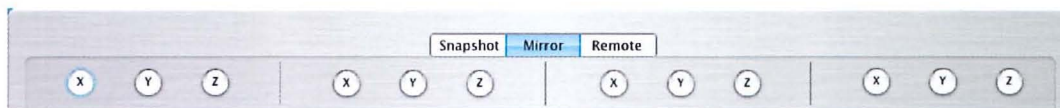


Fig. 11-2: Octopanner v3 X,Y,Z mirror controls

To use mirroring the link buttons must be pressed to link the left and right panners and height fader. If none of the three mirror buttons in Fig. 11-2 are pressed, the panner position and height of the right hand channel in the pair will follow the left hand channel exactly. Pressing an axis mirror button will cause the linked channel to perform the opposite move, for example, if the z mirror is pressed then the right hand sound will move down as the left hand sound moves up. Although originally intended for linking stereo instruments, the mirror buttons allow the movement of one sound to creatively control the movement of another.

Each channel in Fig. 11-1 has a name field above its panner and on a button at the bottom of the strip, as **OctoPanner** is intended for touchscreen as well as mouse control the user must be provided with a means of alphanumeric input – pressing a name button brings up the touch keyboard shown in Fig. 11-3:



Fig. 11-3: OctoPanner v3 touch keyboard

11.2.2 Snapshot copying/swapping and initialising

Pressing the snapshot edit button opens the Snapshot Edit panel with its two tabs:

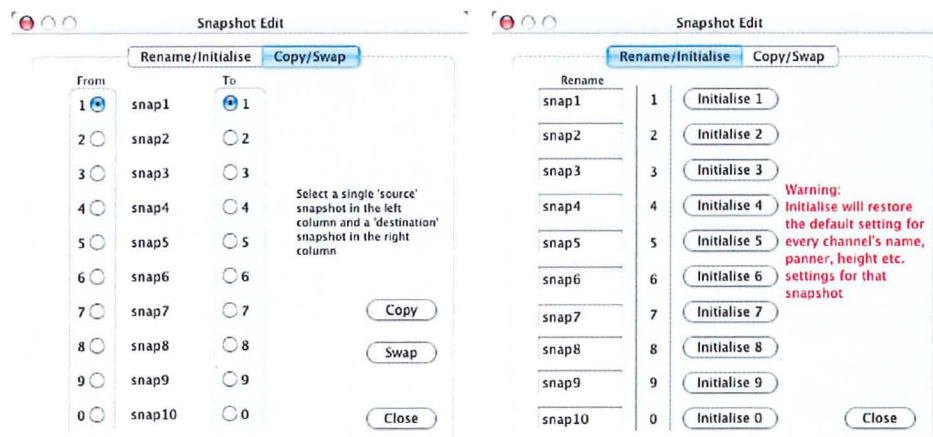


Fig. 11-4: OctoPanner v3 snapshot editing panels

These two tabs allow the user to rename snapshots using the touch keyboard or to initialise them to a blank 'factory setting'. Snapshots can also be copied and swapped, for example a snapshot could be copied and used as a basis for another, and where snapshots are used to represent pieces in a performance their order could be changed using the swap tab.

11.2.3 OctoPanner v3 MIDI setup

The final changes to **OctoPanner** version 3 were to the MIDI setup panel; since each channel has a height slider plus Mute and Solo buttons the user is given a means of specifying a MIDI CC number for those controls. Note that the z-axis can be mirrored to allow control of height in an application such as Ableton Live in the same way as the y-axis was controlled in section 10.1. The send test CC box at the bottom of the MIDI panel has been extended to allow the additional CC messages to be sent to applications that have a MIDI mapping 'learn mode'.

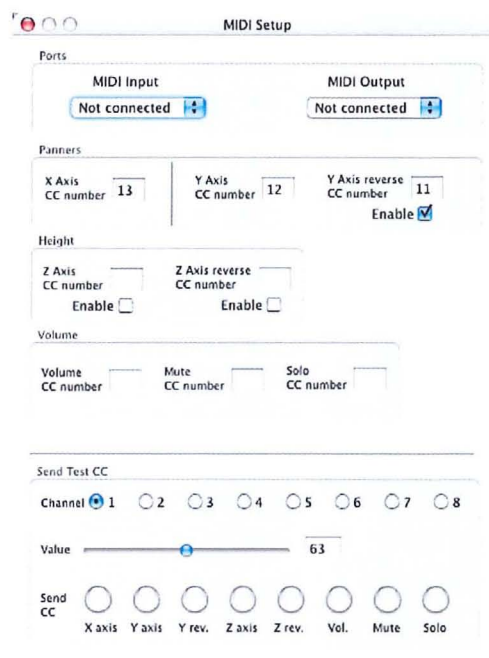


Fig. 11-5: OctoPanner v3 MIDI setup panel

11.3 Chapter research contribution

In terms of the IID methodology this chapter has demonstrated both incremental and iterative development. New features were incorporated but existing features were also reworked as a result of user feedback. This chapter's research informs both Research Questions one and three. Although the touchscreen input device itself has not changed, its functionality has been considerably extended by the addition of the mirroring feature and flexible snapshot save, recall, copy and swap. Because each panner object inherits from a Class with an intrinsic parameter store/retrieve mechanism these features were relatively straightforward to implement and incorporate within an intuitive Apple-compliant user-interface. In this instance Cocoa compares favourably with MAX/MSP's more limited store and recall.

12 Spatialising *Macbenach IV*

12.1 A user trial of Octopanner version 3

The composition of a piece called *Macbenach IV* by Gerard Pape (CCMIX) during the Sonic Fusion 2007 concert series allowed **OctoPanner v3** to receive a real-world test almost 12 months after the first performance using the Max/MSP-based **OctoPanner v1**. Pape had requested a Pro Tools HD system and engineer and had scheduled rehearsals and subsequent recording sessions with trombonist John Kenny. These sessions provided a suitable deadline for the completion of **OctoPanner v3** and a solution for controlling Pro Tools.

Since Pro Tools does not have a MIDI 'learn mode' to allow send controls to be mapped to external controller messages, the control surface emulation research detailed in Appendix A was carried out over four weeks. Although this work provides the reader with a previously unavailable insight into controller emulation for Pro Tools, the result did not perform well enough to be considered for practical use and the alternative means of panning described below was developed.

12.2 Pro Tools panner plug-in development using Max/MSP

Unlike the published VST (Steinberg) and Audio Units (Apple) plug-in specifications, Digidesign only allows RTAS and TDM plug-in development of commercially justified applications and states that education research usage does not generally qualify. There are however, other ways to create an RTAS compatible plug-in, firstly using a VST to RTAS wrapper and secondly using Max/MSP with Cycling '74's Pluggo technology. Max/MSP is able to create a VST compatible plug-in from a Max/MSP patch which then uses the Pluggo runtime environment to generate an RTAS compatible plug-in.

Since Pluggo provides up to eight busses to allow audio transfer between plug-ins, a panning solution was developed wherein a MIDI controlled panner 'Send' plug-in is placed on every audio track and spatialises that audio track input down the eight Pluggo busses representing eight speakers. Eight Pluggo bus 'Receive' plug-ins are placed on

eight Pro Tools aux tracks to receive the spatialised audio and direct it to the eight hardware outputs.

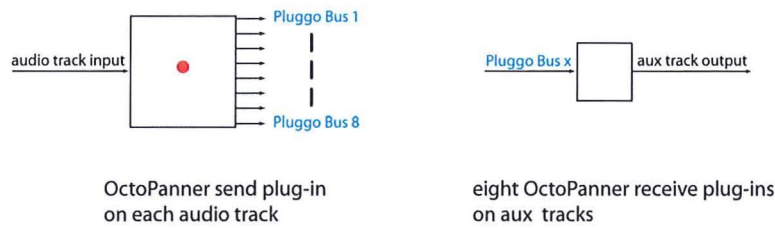


Fig. 12-1: Pluggo based Pro Tools send and receive plug-ins

Fig. 12-2 shows the send and receive plug-ins in a Pro Tools session. Note that next to the 'Audio 1' audio track being spatialised is a Pro Tools MIDI track whose input is connected to **OctoPanner** v3 via a USB MIDI interface and whose output is routed to the panner plug-in on the audio track. This allows the MIDI output from **OctoPanner** to be used to control the panner and also allows **OctoPanner** movements to be recorded and played back, thus **OctoPanner** can be used as a composition tool as well as for live applications.



Fig. 12-2: The send and receive plug-ins in a Pro Tools session

The audio section of the send plug-in Max/MSP patch is shown in Fig. 12-3. It is essentially the eight-speaker Max/MSP patch used in Fig. 5-8 with the `adc~` input object and the `dac~` output objects replaced by `plugin~` and `plugsend~` objects. As the Pluggo environment does not include a MIDI Continuous Controller `ctlin` object, the MIDI control section of the plug-in (not visible) uses the `plugmidiin` and `midiparse` objects to read and check for x or y CC messages.

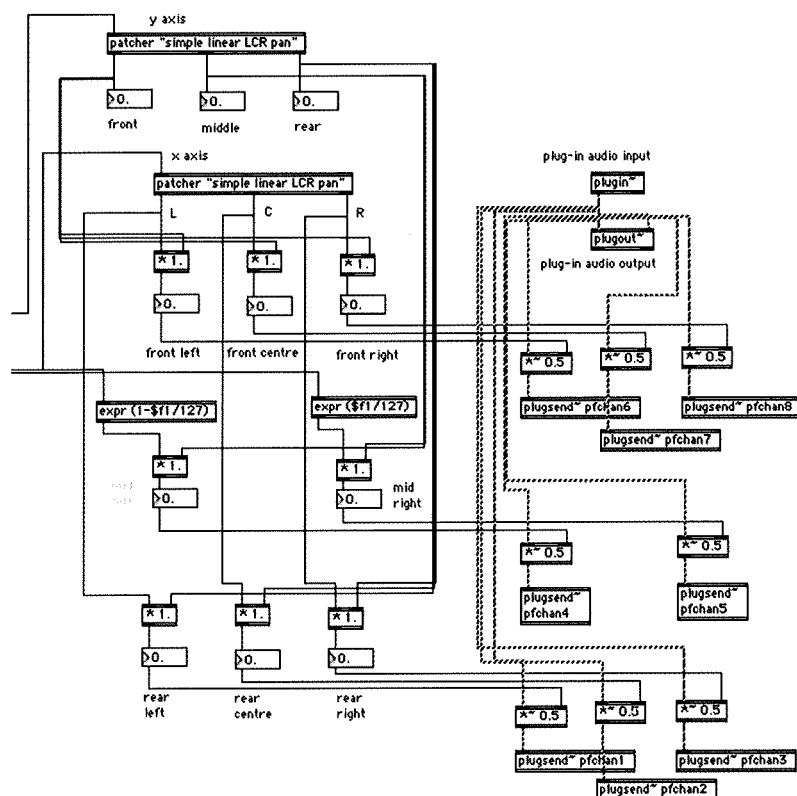


Fig. 12-3: Audio section of the Max/MSP (Pluggo) patch for OctoSend plug-in

This solution to panning in Pro Tools allows the composer or spatialisation engineer to re-configure the room's loudspeaker arrangement by modification of the above patch, thus allowing any permutation of x, y and z array within the limitation of the 8 busses provided by Pluggo.

12.3 Reliability and repeatability

Unfortunately returning to Max/MSP⁶ for the spatialisation appeared to have introduced some audio problems:

When a saved Pro Tools session was re-opened the spatialised audio was often distorted. This distortion could be cured by bypassing and un-bypassing the Pluggo send and receive plug-ins a number of times although this was clearly unsatisfactory. This behaviour appeared to be random in that sometimes the send plug-in was the cause, sometimes one or more of the bus receive plug-ins had distorted outputs.

The more significant problem was that the audio sent down the eight pluggo busses was often delayed by an amount that varied from bus to bus. This random delay appeared to be a random multiple of 512 samples. i.e. multiples of 11.6 milliseconds at 44.1KHz. This could be corrected by adding individual track delays in Pro Tools or by using negative delay compensation offsets but the random nature of the problem meant that the delay for each track had to be examined and set accordingly each time the session was loaded.

12.4 Evaluating the Toolset's use for *Macbenach IV*

Macbenach IV was scored by Pape for eight separately recorded and spatialised trombone parts with microtonal tuning differences. A ninth trombone part would be played live and spatialised using a 2D eight-speaker rectangular configuration. The separate tracks were recorded into Pro Tools during a recording session three days before the premiere of the piece in Craiglockhart Chapel. Spatialisation of the eight Pro Tools tracks was performed one track at a time by Pape using **OctoPanner** version 3 and a resistive touchscreen. The spatialisation information for each track was recorded into Pro Tools to allow editing and playback, with visualisation via the **3D MIDI Visualiser**. Pape found **OctoPanner** to be intuitive and easy to use but found the pressure required by the resistive touchscreen was tiring when large numbers of tracks with vigorous panning were spatialised. As a result of this he asked if it would be possible to copy and mirror one of the recorded automation tracks to automate the final track. An immediate solution was not apparent, but it was later suggested by the author

⁶ Developed using Max 4.6.2 and Pluggo 3.6.1 with OS10.4.8 on G5 Power Mac

that the automation data recorded onto the Pro Tools MIDI track could be sent back to **OctoPanner** to control the left side of a stereo panner pair. The linked and mirrored right side of the stereo panner would then give the panning information that Pape had requested and allow experimentation with x and y axis mirroring. The issue of touchscreen pressure raised by Pape would have been solved by using the capacitive screen in the joystick-based control surface as this only requires a light touch.

12.5 Chapter Summary and Research Contribution

The research described in this chapter can be considered in terms of Research Question 1 (alternative User Interfaces) and Question 2 (DAW hosting).

12.5.1 Question 1 - Alternative User Interfaces

The touchscreen-controlled panning was effective and well received by the user. Since the spatialisation was recorded track by track this technique has a significant advantage over the eight-fader approach. Because only the automation data is recorded rather than the audio from the faders' eight bus outputs, the automation for each track requires considerably less space and is editable. Although Pape did not use any of the ShapePanner output in the final piece he gave a very positive reaction to it and could see potential for use in other projects.

12.5.2 Research Question 2 - DAW Hosting

Since Pro Tools had been specified by Pape for the recording session this had been an ideal opportunity to research this question but the results were mixed. Although the spatialisation was satisfactorily performed, the distortion and timing problems exhibited by the MAX/MSP panner were unacceptable. To potentially address this Chapter offers an alternative approach using C++ that will also allow comparison with MAX/MSP in line with Research Question 3 – Comparing Max/MSP and alternative technologies.

13 A C++ VST/RTAS panner plug-in

The Steinberg Software Development Kit 2.4 (SDK) documentation for VST plug-in development was examined in September 2007 with a view to creating a more reliable panner plug-in than the Max/Pluggo version used for *Macbenach IV*. It was immediately apparent that the VST version 2.4 specification does not include any form of bussing⁷ between plug-ins. It was assumed at that point that the Max/MSP plugSend/Receive bus mechanism was provided by the Pluggo run-time environment, however, in December 2007 the FXpansion VST to RTAS wrapper was used on a Mac without the Pluggo run-time and the FXpansion config tool was able to detect and successfully wrap the Max/MSP PFOctoSend and PFRx1-8 VST plug-ins described in section 12.2. Unfortunately the plug-ins still exhibited the same distortion and delay problems but this did indicate that the inter-plug communication code was within the Max/MSP-generated VST plug-ins and was not provided by the Pluggo environment. This clearly showed that there must be a way to add bus or sidechain functionality to a VST 2.4 plug-in.

In February 2008 an Internet search for VST plug-ins with sidechains led to the “Aux Bus” VST plug-in by Dirk Offringa (Offringa 2005 p1). This plug-in allows 8 channels of communication between its instances. Offringa acknowledges programmer Sean Person:

“...who found the trick to achieve latency-free communication between instances”.

Although the website pointed to by Offringa’s hyperlink no longer exists, a download link for version 1.08 of Person’s Senderella plug-in was eventually located (Person 2005 p1). The FXpansion Wrapper Config tool was used to wrap the Senderella VST plug-in and the resulting RTAS plug-in was tested in Pro Tools 7.3.

⁷ Although busses have been added to the VST version 3.0 specification only Nuendo and Cubase currently support VST 3 plug-ins (April 2008).

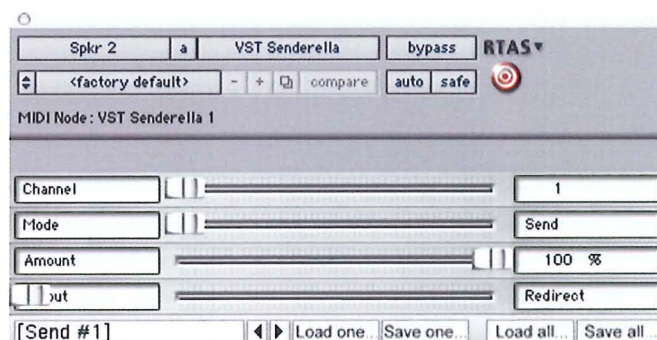


Fig. 13-1: Senderella V1.08 plug-in in Pro Tools

The plug-in shown in Fig. 13-1 was instantiated in SEND mode on an audio track and in RECEIVE mode on an aux track. Although a recognisable signal passed from one instance to the other the signal received by the aux track was badly corrupted and unusable. Various Pro Tools hardware buffer sizes were tried with no improvement but when the number of processors available for RTAS processing was set to one the signal passed cleanly.

To test the integrity of the signal further, two instances of Senderella were used to bus a test signal to two aux tracks, one of which was phase inverted, the result was total signal cancellation indicating that the Senderella busses are sample aligned rather than delayed by multiples of 512 samples as is the case with Max/MSP **plugSend** and **plugReceive**.

A search through all posts on the KVR programmer's forum since 2002 found several posts by ModulR (Sean Person). In March 2005 ModulR posted a reply to a forum question in which he stated his technique for sharing data in Senderella. Later in the thread he presented the C++ code for the simplified Windows version of Senderella.

The Senderella code was examined in detail during March and unsuccessful attempts were made at converting it into a Mac version. Eventually Person's declaration of shared data shown below was extracted and analysed:

```
#pragma data_seg("SHARED")

struct BufferRec
{
    int id;
    int mode;
    float left[MAX_BUFFER_SIZE];
    float right[MAX_BUFFER_SIZE];
    int processCounter;

};

BufferRec buffer[MAX_CHANNELS][MAX_INSTANCES];

int    numSendsOnThisCh[MAX_INSTANCES];
int    numRecvsOnThisCh[MAX_INSTANCES];
int    instanceSlot[MAX_INSTANCES-1];
int    randomCheck;

#pragma data_seg()

#pragma comment(linker, "/section:SHARED,RWS")
```

This code is central to the operation of the Senderella plug-in and this sharing technique is explained by McGahan (Jan 2000 p1). It defines an array of stereo audio buffers that allow a single stereo audio channel to be shared by up to 64 instances of the plug-in. The integer variable `mode` declared in the BufferRec structure indicates whether an instance of the plug-in is functioning as a SEND or as a RECEIVE. By implementing send and receive code in a single plug-in rather than Max/MSP's separate `plugSend` and `plugReceive`, Person's audio buffers can be shared between send/receive instances.

13.1 Coding the VST 3D octal panner

A new Mac 3D panner plug-in called **PFPan** was written by the author in March 2008. Appendix D presents the documented source code to allow user customisation. **PFPan** is based on the gain plug-in example provided by Steinberg in their VST SDK version 2.4. **PFPan** incorporates a modified version of the Person/McGahan data sharing technique with forced initialization of the data to ensure it is placed in the data segment by the compiler (Microsoft 1999 p1). The aGain example does not have a GUI and was chosen so that users of **OctoPanner** or **ShapePanner** wishing to modify the speaker configuration or panning law of the panner plug-in would not have to understand unnecessary code.

Rather than repeat the eight-bus limit in Max/MSP **plugSend/plugReceive**, 16 busses were chosen to provide two layers of eight speakers when using a 3D speaker array. As **ShapePanner** allows up to 16 sounds to be spatialised (each sound is allocated one of the 16 possible MIDI channels) a maximum of 16 send plug-in instances are anticipated therefore the shared array of audio buffers can be specified as follows:

```
const int MAX_BUFFSIZ = 4096;
const int MAX_CHANNELS = 16;
const int MAX_INSTANCES = 16;
#pragma data_seg("PFSHARED")

    struct BufferRec
    {
        int mode;
        float audio[MAX_CHANNELS][MAX_BUFFSIZ];
    };

    BufferRec sneakyBuffer[MAX_INSTANCES] = {1};    // Ferguson - should initialise it
    otherwise it may go in bss_seg not data_seg

#pragma data_seg()

// now instruct the linker that this shared section is Read Write Shared (RWS)
#pragma comment(linker, "/section:PFSHARED, Read Write Shared")
```

Note that although the maximum hardware buffer size in Pro Tools HD is 2048 samples a maximum plug-in buffer size of 4096 was chosen to match the Ableton Live maximum value.

The PFPan plug-in controls are shown in Fig. 13-2:

Control	Function
Mode:	Sets the plug-in instance to either SEND or RECEIVE audio: there can be a maximum of 16 SEND instances and there should be <u>exactly</u> one RECEIVE instance per speaker.
Send/Speaker 1-16:	In SEND mode each send instance of the plug-in must be assigned a unique number from 1 to 16, this number does not correspond to a bus or speaker number. In RECEIVE mode the receive plug-in for each speaker should be set to its appropriate speaker bus number.
X axis:	Slider range from 0 to 1, 0 = max left, 1 = max right
Y axis:	Slider range from 0 to 1, 0 = max rear, 1 = max right
Z axis:	Slider range from 0 to 1, 0 = lowest, 1 = highest
Volume:	Signal attenuation in decibels from -infinity to 0dB

Fig. 13-2: PFPan plug-in controls

Fig. 13-3 shows the **PFPan** plug-in in Ableton Live configured as 'Send' instance no. 1. Note that Ableton's generic interface conveniently allows any two plug-in parameters to be mapped onto the assignable x/y controller. Fig. 13-4 shows the **PFPan** plug-in in Pro Tools via the FXPansion RTAS wrapper, in this case the plug-in has been set to 'Receive' audio sent to speaker bus no. 16.

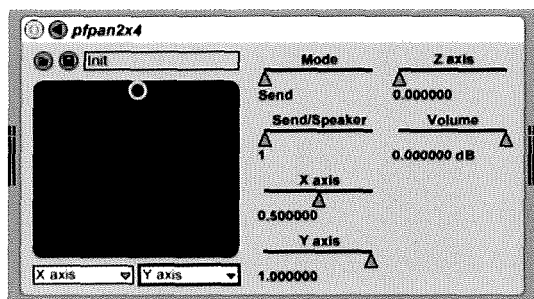


Fig. 13-3: PFPan in Ableton Live

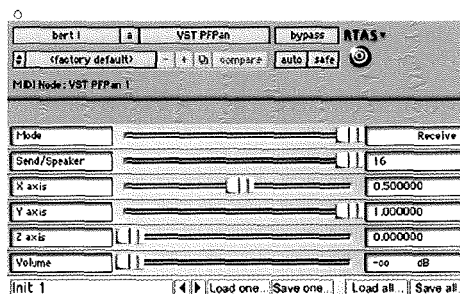


Fig. 13-4: PFPan in Pro Tools

The **PFPan** plug-in was initially coded as a 2D quadrasonic panner and successfully tested in Ableton as a VST plug-in and in Pro Tools via the FXPansion RTAS wrapper. MIDI control of the panner axes and send/receive volumes was added next based on the `canDo` and `processEvent` methods in Steinberg's `vstxsynth` example code in SDK 2.4. Z-axis control was then added and the plug-in tested in a 3D speaker configuration consisting of a lower and upper layer of 4 speakers (2x4). The plug-in sliders are mapped to MIDI Continuous Controllers as shown in Fig. 13-5:

Parameter	MIDI message
Volume	CC 7
X axis	CC 12
Y axis	CC 13
Z axis	CC 14

Fig. 13-5: PFPan plug-in MIDI mapping

The user is able to customise this map by using Xcode to modify the values defined in PFPan.h:

```
const int XAXIS_CC = 12; // Kaoss pad x axis
const int YAXIS_CC = 13; // Kaoss pad y axis
const int ZAXIS_CC = 14;
const int VOLUME_CC = 7;
```

The plug-in must then be rebuilt by selecting Build from the Build menu.

To remove the zipper noise and clicks that occur when large x,y,z or volume changes are made quickly, code was added to the **processReplacing:** method to ramp from the old to new value within the length of one audio buffer, for example if Pro Tools is set to a sample rate of 44.1KHz and a hardware buffer size of 256 samples then **processReplacing:** will gradually implement the x, y, z or volume change within 256 sample periods (5.8mS).

13.2 Guidelines for PFPan plug-in user modification

13.2.1 Speaker configuration

The user may assign any of the 16 speaker busses to a 3D location by modifying the **processReplacing:** method. The x, y and z slider values are floating point numbers ranging from 0 to 1. In the PFPan2x4 example code in Appendix D the origin is defined as the lower rear left corner of the room so the volume of a front left speaker should increase as Y increases but decrease as X increases, i.e proportional to 1 minus X. A simple Quad panner could therefore be implemented as follows:

```
...audio[SPEAKER1][i] = sampleValue * (1-X) * Y; // left front
...audio[SPEAKER2][i] = sampleValue * X * Y; // right front
...audio[SPEAKER3][i] = sampleValue * (1-X) * (1-Y); // left rear
...audio[SPEAKER4][i] = sampleValue * X * (1-Y); // right rear
//...audio[SPEAKER5][i] unused
.
//...audio[SPEAKER16][i] unused
```

13.2.2 Panning law

The panning law could be built in to the mathematics of `processReplacing` or it could be implemented within the `processEvents` method where the MIDI control message is received. To present the simplest example the `PFPan2x4` code is for a linear panner, therefore the MIDI value representing an axis is converted from an integer ranging from 0 to 127 to a float from 0 to 1 simply by dividing by 127.

13.3 Chapter summary and research contribution

The research described in this chapter can be considered in terms of Research Question 2 (DAW hosting) and Question 3 (alternative technologies).

This C++ version of the VST Panner successfully demonstrates the potential for Pro Tools to act as a 16-channel 3D spatialisation engine. When the previous chapter's MAX/MSP-based panner was tested in May 2008 it was assumed that the distortion and audio timing problems believed to be associated with VST/Pluggo and MAX/MSP version 4.6.1 would be addressed by a subsequent software release. It is now known that version 5 of MAX/MSP has removed this functionality, hence this chapter's C++ solution to a VST multi-channel panner and its documented code in Appendix D is significant.

**Part III: Conclusions and contribution to
knowledge**

14 Conclusions

This Chapter begins by evaluating the success of this body of research in answering the original research questions posed.

14.1.1 Research Question 1

“How can the physical user-interfaces used for panning by the music and film post-production industries offer creative alternatives to the fader-based hardware approach commonly used for electro-acoustic performance?”

The positive user feedback suggests that the answer is yes, especially in the case of the final touchscreen version where the physical controller incorporates snapshot recall and a mechanism allowing one sound's movement to mirror the movement of another. The non-motorised joystick was found to be less successful since it could not reflect the true position of a sound when switching between sounds or if automation was used. Although this was not part of the initial question, the literature review highlighted a potential research gap for physical control of spatialisation by musician rather than engineer. University of Sheffield researcher David Moore (2005 p297) suggests that although it will be met with resistance, the time has come to question the appropriateness of the traditional fader as a control paradigm and explore alternate means for a performer to control diffusion such as joysticks, foot-pedals switches and track-balls. The Foot Puck developed during this research appears to offer an original solution to musician control of spatialisation and a patent application is under consideration.

14.1.2 Research Question 2

“How can a DAW be used as an alternative to dedicated panning hardware?”

The qualified success using Ableton Live and Pro Tools as panning engines demonstrates that they have potential although the single-processor limitation of the VST panner developed during this research is a restriction. Recent developments in both DAWs may offer other possibilities; the potential for the Ableton Live DAW to replace a hardware matrix mixer may be increased by the imminent arrival of 'MAX for Live'. Zicarelli (2009 p1) indicates that 'MAX for Live' will include a MAX-based API allowing Ableton Live to be controlled and extended and this is likely to give increased scope for

using this DAW for multi-speaker works. Similarly the release of Pro Tools version 8 may offer new possibilities since plug-in parameters can now be mapped to supported hardware controllers. Although this technique cannot be applied to the panner in the Pro Tools output window, it may offer options when using Paul Neyrinck's Mix51 5.1 panner plug-in (Neyrinck 2009 p1). At present, the Pro Tools 7.1 implementation follows the Sony SDDS standard for cinema release consisting of a planar array of five front loudspeakers plus two rear channels (Sony 2001 p1). It is reasonable to assume that this may be supplemented in a future release by the Blu-ray 7.1 alternative which spaces the seven loudspeakers more evenly round the listener and thus is more in line with electro-acoustic practice (DTS 2006 p10).

14.1.3 Research Question 3

"How can emerging programming technologies offer creative alternatives to the MAX/MSP or hardware-based tools commonly used for sound spatialisation?"

Without doubt, Max/MSP scores highly with its rapid creation of functional programs with a small learning curve provided that the required internal objects or externals are available. This can be at the cost of the user interface since this tends to be driven by what can be easily achieved and often contravenes Apple's 'User Interface Guidelines' (Apple 2007 p1). On the negative side, there have been issues with MAX/MSP audio stability during this research. In contrast, Cocoa has a steeper learning curve and lacks objects (classes) specifically for MIDI and audio meaning that the programmer must include some Carbon code based on the minimal examples provided by Apple and this research contributes significantly in this area. To balance this, the programmer is presented with the full range of services that OS X can provide, for example Quartz 3D graphics and comprehensive clocking and synchronisation via CoreAudio Clock. The Cocoa user is also presented with a drag and drop User Interface building tool with the result that the final result is more likely to present the end user with a familiar user interface and set of functions.

The key phrase in Research Question 3 is 'creative alternatives'. It can be argued that the strengths of Apple's OS X Core technologies allowed development of several features that could be very difficult to duplicate in MAX/MSP, for example:

OctoPanner - Snapshot save, recall, copy and paste.

Shape Panner - Data-driven trajectory generation with a master-detail user interface, master/slave MIDI Time Code and MIDI clock synchronisation.

3D MIDI Visualiser - Realtime 3D display of sound movement within a virtual room using Quartz Composer and Cocoa.

VST Panner plug-in – Not possible using MAX/MSP version 5. Apple's Xcode IDE and C++ were used to create the VST plug-in.

14.1.4 Summary

Common to all three Research Questions is the word 'alternative'. It is clear that the controllers, DAWs and technologies presented here should not necessarily replace what has gone before, instead their use should also be considered as a means of creatively and effectively supplementing older techniques.

14.2 Contribution to knowledge

The author argues that the research presented in this thesis makes a substantial and original contribution to knowledge by:

- Pointing towards an up-to-date, alternative perspective on the key developments of spatial audio since 1950 by focussing on loudspeaker topology, control and associated hardware to identify research gaps;
- Examining the practical use of emerging Mac programming technologies such as Cocoa and Quartz as alternatives to MAX/MSP by implementing a set of requirements followed by critical review;
- Developing alternative physical methods of spatialisation in the form of a foot-controlled 'puck';
- Developing a novel shape-based approach to automated spatialisation control via a synchronisable cue-list;

- Developing techniques for 3D visualisation of MIDI positional data within a virtual room;
- Developing a technique for inter-communication between plug-in instances with subsequent development of a 3D 16-channel DAW plug-in and documented source code to allow configuration by the user.
- Expanding the documentation available on Cocoa and Apple's Core technologies and by providing the first example of sample code using CoreAudio Clock to assist future researchers in this field. It is anticipated that Cocoa will become increasingly significant to developers since iPhone and iPod applications can only be developed using Cocoa and XCODE and this will almost certainly be the case for the anticipated tablet device.

One of the most pleasing aspects of this research has been the response from users and potential users. Discussions are underway with the Royal Academy of Music regarding shape-based panning, and with Aberdeen University about the use of the Pro Tools panning solution. The plug-in solution developed during this research may now be of use to other developers, for example, an email in May 2009 from Damiano Meacci at Tempo Reale indicated that his research into a VST panner had stalled as a result of the removal of pluggo from MAX/MSP version 5.

14.3 Limitations and potential for future work

The Edinburgh Napier performance and masterclass context of this research gave limited scope for interaction with users although the user feedback was useful and informed the design process. Consideration should therefore be given to future engagement with larger numbers and a wider range of users, for example improvised performance, DJs and VDJ's to inform future requirement specifications. If any aspects of this research are then deemed to be candidates for commercial development, consideration should be given to formal testing methods within the chosen development methodology.

The Person/McGahan data sharing mechanism used in the VST panner plug-in only works when a single processor core is assigned to plug-in processing. Although a refinement of their technique could be investigated it is still a workaround for the lack of side-chain busses in the VST 2.4 specification. It is possible that sequencer manufacturers other than Steinberg may implement hosting of VST version 3 plug-ins in the future. This would allow true busses to be used between the send and receive panners. The author's preferred solution would be to develop a true RTAS version of the PFPan plug-in with the support of Digidesign that included a graphical UI to allow users to configure the loudspeaker array to their individual requirements.

A significant limitation of the current OctoPanner is that the single-touch touchscreen only allows use of one panner at a time. Although the system could be reworked using the multi-touch Jazz Mutant Lemur, facilities such as snapshots would be lost. Multi-touch capability is being rolled out across Apple's product line in the form of the iPhone, iPod, Macbook, Magic Mouse and Snow-Leopard. It can be reasonably predicted that the leaked reference to an "Apple Slate" by the New York Times Editor Bill Keller (Gawker 2009 p1) is indeed indicative of its existence. It is likely that such a tablet device will be offer capacitive multi-touch functionality like the iPhone and that its development will be via Cocoa. This could form the basis of a wireless multi-touch version of OctoPanner and thus remove its biggest limitation.

Bibliography

Ableton (2009) *What is Live?* Available at:

<http://www.ableton.com/live> (Accessed: 1-10-2009)

A&G Soluzioni Digitali (2009) *IMEASY - Integrated Modular Expandable Audio Spatialisation system overview* Available at: <http://imeasy.aegweb.it/> (Accessed: 18-02-2009)

AES (2001) *Technical document AESTD1001.0.01-05 Multichannel surround and operations*. New York

Anguish, S., Buck, E. M., Yacktmann D. A. (2003) *Cocoa Programming* SAMS

Apple Computer Inc. (2001) *Learning Carbon* O'Reilly

Apple Inc. (2005) *Cocoa Bindings Reference*. Available at:

<http://developer.apple.com/documentation/Cocoa/Reference/CocoaBindingsRef/>

(Accessed: 12-03-2008)

Apple Inc. (2006) *Cocoa fundamentals Guide*. Available at:

<http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaFundamentals/>

(Accessed: 12-03-2008)

Apple Inc. (2006) *Data Formatting Programming Guide for Cocoa*. Available at:

<http://developer.apple.com/documentation/Cocoa/Conceptual/DataFormatting/>

(Accessed: 12-03-2008)

Apple Inc. (2007) *Inside Mac OS X: Aqua Human Interface Guidelines*. Available at:

<http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/>

(Accessed: 12-03-2008)

Apple (2009) Developer Connection: XCODE Available at:
<http://developer.apple.com/tools/xcode/> (Accessed: 1-10-2009)

Austin, L. (1989) *David Tudor and Larry Austin: A Conversation*, Denton, Texas,
Available at: <http://www.emf.org/tudor/Articles/austin.html> (Accessed: 1-10-2009)

Austin, L. (2001) *Sound Diffusion in Composition and Performance Practice II: An Interview with Ambrose Field* Computer Music Journal, 25:4, pp21-30

Bernardini, N. and Vidolin, A. (2005) *Sustainable live electro-acoustic music*. In Proceedings of the International Sound and Music Computing Conference. Salerno, Italy

Biggs, M.A.R. (2000) *Editorial: the foundations of practice-based research*. Working Papers in Art and Design 1. Available at:
<http://www.herts.ac.uk/artdes/research/papers/wpades/vol1/vol1intro.htm> (Accessed: 18-10-2009)

Boehm, B. (1988) *"A Spiral Model of Software Development and Enhancement"*, "Computer", "IEEE", 21(5): pp 61-72

Bosi, M. (1990) *An interactive real-time system for the control of sound localization*. In S. Arnold and G. Hair, editors, Proceedings of the ICMC, Glasgow

Bruce, M., Cooper, R. (2000) *Creative Product Design: A Practical Guide to Requirements Capture Management* Wiley

CCRMA (2009) *People: John Chowning*
<https://ccrma.stanford.edu/people/john-chowning> (Accessed: 18-10-2009)

Chadabe, J. (1997) *Electric Sound The Past and Promise of Electronic Music* Prentice Hall

Choi, A. (2003) *Fish Creek MIDI (FCM) Framework*. Available at:
<http://members.shaw.ca/akochoi-old/blog/2003/12-07/index.html#2> (Accessed: 12-03-2008)

Chowning, J. M. (1970) *The simulation of moving sound sources*. Audio Engineering Society 39th Convention, pp 692--694

Clozier, C. (2001) *The Gmebaphone Concept and the Cybernéphone Instrument* Computer Music Journal: Vol. 25, no. 4. Cambridge, MA: MIT Press: pp 81-90.

CM Labs (1999) *Motormix Developers Guide version 1.2* CM Labs Inc.

Cockburn, A. (2008) "Using Both Incremental and Iterative Development". STSC CrossTalk: Journal of defense software engineering (USAF Software Technology Support Center) 21 (5)

Crompton, T.W.J. (1974) *The subjective performance of various quadraphonic matrix systems* BBC Research Department Report 1974/29

Cross, L. (2001) *Remembering David Tudor: A 75th Anniversary Memoir* Frankfurter Zeitschrift für Musikwissenschaft, TM1

Cunningham, M. (1997) *Sound On Stage* no. 5, SOS Publications Ltd.

Cycling '74 (2007) *Xoaz's collection of Max and MSP externals and patches* Available at: <http://www.cycling74.com/twiki/bin/view/Share/RichardDudas> (Accessed: 1-10-2009)

Cycling '74 (2009) *Max/MSP/Jitter Product Overview*. Available at: <http://www.cycling74.com/products/mmjoverview> (Accessed: 1-10-2009)

Cutler, M., Robair, G. & Bean (2000). "The Outer Limits - A Survey of Unconventional Musical Input Devices". *Electronic Musician*. Available at: http://emusician.com/mag/emusic_outer_limits/ (Accessed: 1-10-2009)

Doherty, D. (1998). "Sound Diffusion of Stereo Music Over a Multi-Loudspeaker Sound System: From First Principles Onwards to a Successful Experiment". *SAN Journal of Electro-acoustic Music*, 11: 9-11.

Dolby Laboratories (1999). "Surround Sound: Past, Present and Future". Available at:
http://www.dolby.com/assets/pdf/tech_library/2_Surround_Sound_Past.Present.pdf
(Accessed: 18-10-2009)

DTS (2006) *DTS-HD Audio Consumer White Paper for Blu-ray Disc and HD DVD Applications* Available at:
<http://www.dts.com/~media/B962F033C9254AD4B62ECFC6293C9E86.ashx>
(Accessed: 01-10-2009)

Elsa, P. (2004) "MAX and MTC" Available at:
www.pescadoo.net/annexe/max/Max&MTC.pdf (Accessed: 17-02-2009)

EMF (2009) *Electronic Music Foundation – Varese poeme*. Available at:
<http://emfinstitute.emf.org/exhibits/varesepoeme.html> (Accessed: 01-10-2009)

Ernst, D. (1977) *The Evolution of Electronic Music*. New York: Schirmer Books

Fishman-Johnson, E. (1993-94) *The Movement of Sound in Space: An Update - Proceedings of the Bowling Green State University new music & art festivals 14 & 15 paper sessions volumes 5-6*

Frayling, C. et al (eds.) (1997) *Practice-based Doctorates in the Creative and Performing Arts and Design*. N.p. [UK]: UK Council for Graduate Education

Expansion (2009) *VST to RTAS Adapter 2.1 overview*. Available at:
<http://www.fxexpansion.com/index.php?page=15> (Accessed: 17-02-2009)

Gaskell, P.S. and Ratliff, P.A. (1977) *Quadraphony: Developments in Matrix H decoding*
BBC Research Department Report 1977/2

Gawker (2009) Bill Keller: Apple Tablet 'Impending' Available at:
<http://gawker.com/5389636/bill-keller-apple-tablet-impending> (Accessed: 29-10-2009)

Goodliffe, P (2007) *Code Craft: The Practice of Writing Excellent Code*. No Starch Press

Haller, H. P. (1999) *Nono in the studio - Nono in concert - Nono and the interpreters* Contemporary Music Review Vol. 18 Part 2 pp.11-18

Harrison, J. (1998). "Sound, Space, Sculpture - Some Thoughts on the 'What,' 'How' and 'Why' of Sound Diffusion". Organised Sound, 3(2): pp 117-127.

Harrison, J. (1999) *"Diffusion - Theories and Practices, with Particular Reference to the BEAST System"*, EContact!, 2(4)

Hillegass, A. (2004) *Cocoa Programming for Mac OS X 2nd ed.* Addison-Wesley

Hilton, K. (1994) *Floyd in Florida*. Studio Sound no. 8

Hinton, G. (2001) *A Guide to the EMS Product Range 1969 to 1979* Available at: <http://www.ems-synthi.demon.co.uk/emsprods.html#queg> (Accessed: 01-10-2009)

Hoffmann, H.F. (1993) *"Requirements Engineering, a survey of Methods and Tools"* IFI

Humon, Naut et al. (1998) *Sound Traffic Control: An Interactive 3-D Audio System for Live Musical Performance*. ICAD '98 Proceedings, University of Glasgow,

IMEB (2009) *About of the Cybernephone (formerly Gmebaphone)*

Institut International de Musique Electroacoustique de Bourges Available at:

http://www.imeb.net/IMEB_v2/PDF/About-of-the-Cybernephone.doc (Accessed: 01-10-2009)

IM-Research (2008) *Vortex Surround Designer: VSD Features* Available at:

<http://im-research.com/products/designer/> (Accessed: 01-10-2009)

IRCAM (2008) *SPAT Reference Manual – Overview*. Available at:
<http://support.ircam.fr/forum-ol-doc/spat/3.0/spat-3-ref/co/overview.html> (Accessed: 01-10-2009)

Jones, S. (2006) "Recombinant Media Labs" Available at:
http://mixonline.com/design/profiles/audio_recombinant_media_labs/ retrieved 2-11-2008

Kernighan, B., Ritchie, D. (1988) *The C Programming Language* 2nd ed. Prentice Hall

Koftinoff, J (2009) "osc_dev: Oscpak TCP" CREATE forum Available at:
http://lists.create.ucsb.edu/pipermail/osc_dev/2009-August/001735.html (Accessed: 01-10-2009)

Korg (2007) *Korg Kaoss Pad Dynamic Effects/Sampler User Manual* Available at:
http://www.korg.co.uk/downloads/kp3/support/KP3_OM_EFG1.pdf (Accessed: 01-10-2009)

Kuivila, R. (2001) *Open Sources: Words, Circuits, and the Notation/Realization Relation in the Music of David Tudor* Proceedings of the Getty Research Institute Symposium, "The Art of David Tudor," 2001.

KVR Forum (2005) *Topic: Sidechain – sharing data between plugins* Available at:
www.kvraudio.com/forum/viewtopic.php?t=77794 (Accessed: 12-04-2008)

Line 6 (2009) *Line 6 DL4 Stompbox Modeller* Available at:
<http://uk.line6.com/dl4/> (Accessed: 01-10-2009)

Ludvig, LF. (2003) *Extensions and generalizations of the pedal steel guitar*
Patent US 6,852,919 B2

Larman, C Victor R. Basili, V.R. (2003) "Iterative and Incremental Development: A Brief History," *Computer*, vol. 36, no. 6, pp. 47-56

- McGahan, P. (2000) *How to share a data segment in a DLL* Available at:
URL: www.codeproject.com/KB/DLL/data_seg_share.aspx (Accessed: 12-04-2008)
- Manning, P. (2004) *Electronic and Computer Music*. revised ed. Oxford University Press
- Marshall, G. and McCully, B. (1998) *The HUI Reference Guide rev. B* Mackie Inc.
- Malham, D. G. (1990) *Ambisonics - a technique for low-cost, high-precision, three-dimensional sound diffusion*. Proceedings of the International Computer Music Conference 1990, pp 118--120
- Malham, D.G. and Myatt, A. (1994) *3-D Sound Spatialization using Ambisonic Techniques* Computer Music Journal
- Malham, D. G. (1995) *3-d sound spatialization using ambisonic techniques*. Computer Music Journal, 19(4) pp 58--70,
- Malham D.G. (1998) *Composition and diffusion: space in sound in space* Organised Sound 3(2)
- Manning, P. (1993) *Electronic and Computer Music*. 2d ed. Oxford: Clarendon Press
- Manning, P. (2006) "The significance of techné in understanding the art and practice of electro-acoustic composition" Organised Sound 11(1) Cambridge University Press
- Meyer Sound (2007) *Matrix3 Audio Show Control System* Available at:
URL: <http://www.meyersound.com/lcs/matrix3/> (Accessed: 01-10-2009)
- Microsoft (1999) *C/C++ Preprocessor reference – data_seg* Available at:
<http://msdn.microsoft.com/en-us/library/thfhx4st.aspx> (Accessed: 12-04-2008)
- MMA (1983) *The MIDI Specification V1.0* MIDI Manufacturers Association
- Modarres, M. Kaminskiy, M. Kritsov, V. (1999) *"Reliability engineering and risk analysis:*

a practical guide" CRC Press

Mooney, J. (2001). *"Ambipan and Ambidec: Towards a Suite of VST Plugins with Graphical User Interface for Positioning Sound Sources within an Ambisonic Sound-Field in Real Time"*. M.Sc. thesis (Music Technology Research Group; University of York).

Moore, A., Moore, D. & Mooney, J. (2004). *"M2 Diffusion - The Live Diffusion of Sound in Space"*. Proceedings of the International Computer Music Conference (ICMC) 2004 Miami FL.

Moore, D. (2004). *"Real-Time Sound Spatialization: Software Design and Implementation"*. Ph.D. thesis (Department of Music; University of Sheffield)

Nelson, R and Andrews, S (2003) *Practice as Research: Regulations, Protocols and Guidelines* PALATINE, Lancaster University

Neve (1991) *Operator's Handbook for the VR Legend Console* issue 1 Neve Electronics International Ltd, Cambridge

Neyrinck, P. (2009) *Mix 51: 5.1 Surround Panning And Surround Mixing For Pro Tools LE* Available at:

http://www.neyrinck.com/Pages/mix51_more.html (Accessed: 01-10-2009)

Nordin, I. L. (2007) *"Stockhausen Edition no. 91 (Cosmic Pulses)"* Available at:

<http://home.swipnet.se/sonoloco25/stockhausen/91.html> (Accessed: 01-10-2009)

Oriogun P K, (2002) *"Towards Understanding Software Requirements Capture: Experiences of Professional Students using the NIA to Support the Win-Win Spiral Model"*, LTSN-ICS Electronic Journal (ITALICS), ISSN 1473-7507, Volume 1, Issue 2

Osmond-Smith, D. (1991) *Berio* Oxford University Press

Offringa, D. (2005) *Aux Bus Set* Available at:

www.solidsoundstudio.net (Accessed: 12-04-2008)

Parks-Sydow, D. (1995) *More Mac programming techniques* M&T Books, New York

Perron, M. (1991) "How Steady Is Your Click Track?" Preprint 3132 Proceedings of the 91st AES Convention, New York

Person, S. (2005) *Senderella version 1.08* Available at:

www.substructive.com/plugins/senderella-v1.08.zip (Accessed: 12-04-2008)

Place, T (2005) "An Interview with David Wessel" Available at:

<http://www.cycling74.com/story/2005/9/13/19320/0068> (Accessed: 08-11-2008)

Puckette, M. (1991) "Combining Event and Signal Processing in the Max Graphical Programming Environment" *Computer Music Journal* 15(3) MIT

Puckette M (1991) "Something Digital" *Computer Music Journal* 15(4)

Puckette, P (1996) Pure Data: another integrated computer music in Proceedings, International Computer Music Conference Pages 37—41

Pulkki, V. (2000) *Generic panning tools for MAX/MSP*. Proceedings of International Computer Music Conference 2000. pp. 304-307. Berlin, Germany,

Pulkki, V. and Karjalainen, M. (2001) *Localization of amplitude-panned virtual sources I: Stereophonic panning*. *Journal of the Audio Engineering Society*.

Reed, C. (2003) *SynthTest 1.2.1* Available at:

URL: www.manyetas.com/creed/synthtest.html (Accessed: 12-04-2008)

Richmond Audio Design (2001). "The AudioBox Disk Playback Matrix Mixer".

Available at: <http://www.hfi.com/dm16.htm>. (Accessed: 01-10-2009)

Richmond Sound Design (2009) *ABShowMaker User Manual* Available at:

<http://hfi.richmondsounddesign.com/ABSMmanual/ABSMHelp.html> (Accessed: 01-10-2008)

Roads, C (1996) *The Computer Music Tutorial*, MIT Press, Cambridge

Roads, C., Kuchera-Morin, J. & Pope, S. (2001). "Research on Spatial and Surround Sound at CREATE". (Santa Barbara CA.; Berkeley, University of California)

Rolfe, C. (1999). "A Practical Guide to Diffusion". EContact! 2(4). (CEC). Available at: <http://cec.concordia.ca/econtact/Diffusion/pracdiff.htm> (Accessed: 12-04-2008)

Rumsey, F. (2001). *Spatial Audio* Focus Press, Oxford

Rust, C. Mottram, J. Till, J. (2007) "AHRC Research Review: Practice-Led Research in Art, Design and Architecture" Available at: www.ahrc.ac.uk/About/Policy/Documents/Practice-Led_Review_Nov07.pdf (Accessed: 01-10-2009)

Solid State Logic (1995) *SSL 4000G Series Console Operators Manual* 4th edition
Solid State Logic Ltd, Oxford

Sony (2001) *Sony Dynamic Digital Sound: What is SDDS?* Available at: www.sdds.com (Accessed: 01-10-2009)

Soundcraft (1997) *DC2000 Console User Guide*. Harman International Industries, Stamford, CT

Threw, B (2009) "Recombinant Media Labs at UCSD Roundup" Available at: www.barrythrew.com/2009/06/15/recombinant-media-labs-at-ucsd-roundup/ (Accessed: 12-10-2009)

Tutschku, H. (2002). "On the Interpretation of Multi-Channel Electro-acoustic Works on Loudspeaker-Orchestras: Some Thoughts on the GRM-Acoustionium and BEAST". SAN Journal of Electro-acoustic Music, 14: pp 14-16.

Truax, B. (1998). "*Composition and Diffusion - Space in Sound in Space*".
Organised Sound, 3(2): pp 141-146.

Wacom (2006) *EMR (Electro-Magnetic Resonance) Technology* Available at:
www.wacom-components.com/english/technology/emr.html (Accessed: 12-04-2008)

Warnaby, J. (1996) *Review : Das Experimentalstudio der Heinrich Strobel Stiftung des
Sudwestfunks, Freiburg, 1971-1989 by Hans-Peter Haller*. Tempo No. 197

Wiener, R. & Pinson, L. (1988) *An Introduction to Object-oriented Programming and C++*
Addison-Wesley

Worrall, D. (1998). "*Space in Sound - Sound of Space*". Organised Sound, 3(2): 93-99.

Worrall, D. (2002). "*David Worrall Home Page*". Available at:
<http://www.avatar.com.au/worrall/> (Accessed: 01-10-2009)

Wyatt, S. et al. (1999). "*Investigative Studies on Sound Diffusion / Projection*".
EContact! 2(4). (CEC). Available at:
<http://cec.concordia.ca/econtact/Diffusion/Investigative.htm> (Accessed: 12-04-2008)

Vidolin, A (1993) "*Interpretazione musicale e signal processing*"
Centro di Sonologia Computazionale dell'Università di Padova Available at:
www.dei.unipd.it/~musica/Dispense/VidolinMit.pdf (Accessed: 10-10-2009)

Yandell P. (2002) *PYMIDI Framework v1.1* Available at:
<http://pete.yandell.com/software/> (Accessed: 12-04-2008)

Zicarelli, D. (2009) *Tools for Creating Devices in Live* Available at:
<http://www.cycling74.com/story/2009/1/15/114420/967> (Accessed: 01-10-2009)

Zölzer, U (2002): *DAFX Digital Audio Effects*. John Wiley & Sons, Chichester

Zvonar, R. (1999) "*A history of spatial music*", EContact!, 7(4) (CEC)

Appendix A Panning in Pro Tools using MIDI controller emulation

This Appendix investigates a method of surround panning in Pro Tools that does not require a panner plug-in. Unlike Ableton Live, Pro Tools does not allow controls to be mapped to MIDI messages by the user. Instead, the user must use an approved and supported Digidesign controller such as the Command 8 or third-party controllers such as the JLCooper CS10, Mackie HUI or CM Labs Motormix. If **OctoPanner** and **ShapePanner** were to control the pans and sends in Pro Tools, those programs must emulate a supported MIDI Controller.

A.1 JL Cooper CS10

The CS10 was the first controller supported by Pro Tools in 1991, The eight faders are non-motorised and the rotary controllers are 270 degree travel potentiometers rather than endless-travel rotary encoders. The CS10 MIDI implementation is clearly described in the user manual and the CS10 has been emulated by other controller manufacturers such as Roland and Kenton as a result. To investigate the suitability of the CS10 protocol for Pro Tools panning a simple Cocoa application was produced:

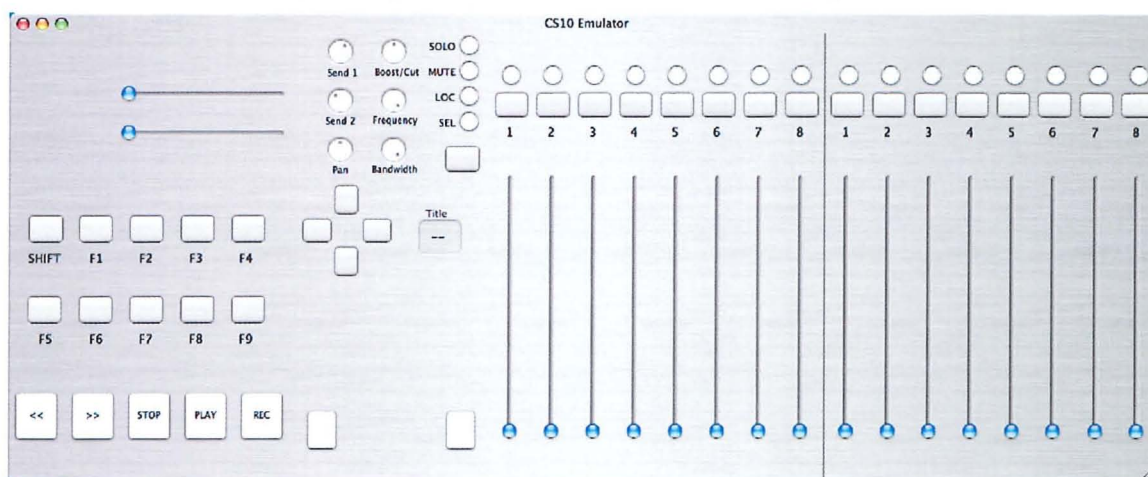


Fig. A- 1: CS10 Emulator (Cocoa)

Pro Tools allows multiple CS10s to be connected so the CS10 emulator emulates 2 devices hence the 16 faders shown in Fig. A- 1. The emulator was recognised by Pro Tools and all controls worked. The non-motorised nature of the faders was immediately

problematic; the incorporation of the original CS10 into Pro Tools added two nulling arrow LEDs similar to those found on a VCA-based console automation system, Pro Tools only 'picks up' the fader when it is moved to the current screen position. Although this can be solved by ramping each fader from min to max during initialisation the same technique cannot be used for the six potentiometers. Since the six pots affect the target track selected by the 'SEL' switches each time a different track is selected the adjusted potentiometers must pass through the current screen control position to pick it up (meaning a controlling application cannot simply update a track's pan/sends then switch to another track). Although this test was unsuccessful it did show that CS10 emulation could be a very simple method for controlling the Pro Tools transport and selecting location markers.

A.2 CM Labs Motormix

The Motormix was investigated next since it is a controller with eight motorised faders and eight endless rotary encoders and its MIDI protocol is also defined in Developer information available from CM Labs. The Cocoa CS10 Emulator was adapted to implement the fader part of the Motormix protocol and to implement the MIDI handshaking interrogation/response used by Pro Tools to detect if a Motormix is connected. The Motormix Emulator was able to successfully control faders in Pro Tools but it then became apparent that the Motormix protocol did not allow the Pro Tools send pans to be controlled⁸. As a result the otherwise successful Motormix emulation was put on hold.

⁸ Unlike Ableton Live where the track pan control also affects the panning of a stereo send, Pro Tools had separate track pan and send pan controls. After this research took place Digidesign introduced a pan linking preference in Pro Tools 7.3

A.3 Mackie HUI

The HUI was a complex MIDI controller designed for Pro Tools with functionality approaching that of Digidesign's now obsolete Pro-Control. The HUI implementation in Pro Tools incorporates a 'fader flip' mode where the currently selected send bank (A to E) is mapped onto the motorised faders with the send level under fader control and the send pan under rotary encoder control. Although this gives the required control of sends and pans the HUI protocol is not published.

A.3.1 Analysing the HUI protocol

The HUI reference guide includes a brief MIDI Implementation Chart that indicate that faders are allocated to two separate blocks of CC messages and that switches are mapped to another block. The chart also indicates a handshake mechanism using MIDI Note On messages. Although this information is minimal it does give enough insight into the protocol to allow further analysis.

A Mackie HUI was connected to a Pro Tools system and Snoize's **MIDI Monitor** used to examine the messages passing between the two, only general results are presented here as the protocol is not in the public domain.

Handshaking - Pro Tools polls the HUI at approximately 0.75 second intervals, unlike the elaborate handshaking required with a Logic Control, the HUI inverts and returns the note on value sent by Pro Tools, this is stated in the HUI implementation chart although unclear.

Switches - The HUI switches are arranged into related banks such as channel strip one to seven, transport, send selects. The switches within each bank are assigned a number from zero to seven. For example, the channel strip number one bank contains Fader touch, select, mute, solo in increasing numerical order. A key press or release is therefore transmitted as two MIDI CC messages: 0F (hex) representing the 'bank number' followed by 2F (hex) representing the 'switch number'.

Faders - Mackie state that the HUI fader resolution is 1024 steps, i.e. 10bit. The Implementation Chart indicates that the channel one fader MSB is transmitted as CC 00 and the LSB as CC 20 (hex). Monitoring the data stream confirms this.

At this point it was observed that not only were there similarities between the HUI and the published Logic Control protocols but also similarities with the published Motormix

meaning sufficient information was available to produce the HUI Emulator shown in Fig. A-2:

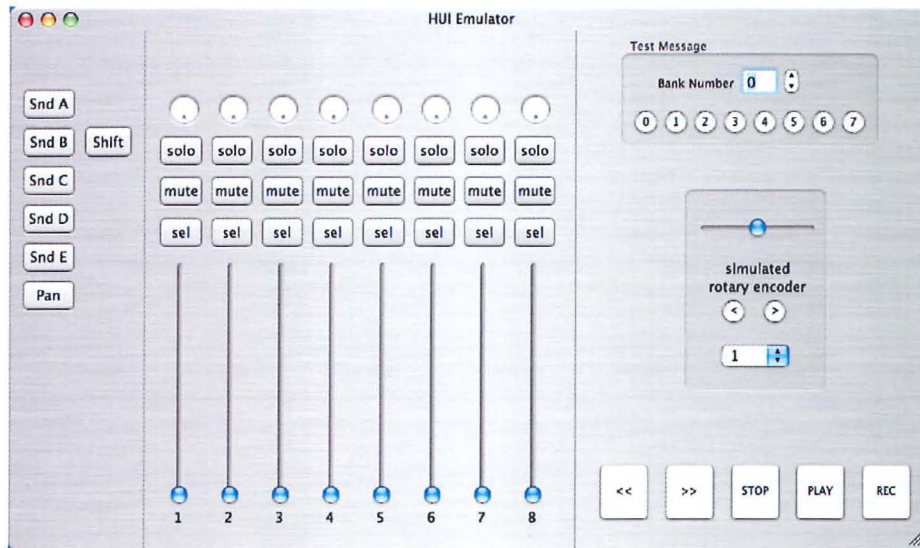


Fig. A- 2: HUI Emulator (Cocoa)

The upper-right corner of Fig. A- 2 shows a test message generator to allow the HUI 'bank + switch number' combinations to be generated to investigate the HUI protocol further. Similarly, below this is a simulation of a rotary encoder, the available documentation showed that the encoders send incremental information to indicate clockwise or anti-clockwise rotation rather than the specific position message sent by the faders.

Having determined enough HUI protocol information to be able to control Pro Tools faders, sends and send pans a HUI class was added to a test version of **OctoPanner** application to convert its x/y coordinate information into Pro Tools control movements. The HUI handshaking response was added to the **PFMidi** class. The result was successful control of Pro Tools however there was a significant issue that was not overcome; large amounts of panner activity caused **OctoPanner** to allocate too much CPU time to the UI thread meaning the delay in responding to the Pro Tools handshaking poll was too great. This resulted in Pro Tools flagging to the user that the HUI was not responding.

Appendix B A Cocoa wrapper for Carbon CoreMIDI functions

The audio section of the Apple Developer site contains minimal information about CoreMIDI. A 2001 Apple document called coreAudio.pdf is the most informative, giving some information on CoreAudio, CoreMIDI and AudioUnits, additional information can be found in the documentation file for midiServices.h that presents the available MIDI routines.

Alongside this sparse documentation is source code for a very simple C program called echo.cpp that echoes MIDI received on any input port to the first output port found. Two things should be noted about this example code, firstly it is a simple command-line application and has no GUI, secondly it is a Carbon application rather than Cocoa. The following analysis of this example is presented to help explain the coreMIDI interface.

Before analysing the program three terms should be explained:

MIDIDevice A piece of MIDI hardware, for example a Midisport 2x2 MIDI interface.

MIDIEntity The MIDI device may sometimes contain separate blocks of functionality, for example the Midisport 2x2 interface contains two separate MIDI ports A and B, each with a MIDI input and output; the Midisport 'MIDIDevice' would be described as containing two 'MIDIEntities' – Ports A and B

MIDIEndpoint This is the lowest level and usually represents a physical MIDI input or output connection, thus the Midisport 2x2's two MIDIEntities A and B each contain two MIDIEndpoints, four in total: MIDI input A, MIDI output A, MIDI input B, MIDI output B.

For a program to send and receive MIDI, the programmer must create a MIDI client and MIDI input and output ports, these input and output ports will then be connected to MIDIEndpoints and references to these will be stored using the following data types defined in midiservices.h:

```
MIDIClientRef      - a reference to the client (the program)
MIDIPortRef       - a reference to a MIDI input or output port
MIDIEndpointRef  - a reference to a physical MIDI connection9
```

⁹ This could also be a virtual endpoint representing a physical cable

Although the sample file is called `echo.cpp` the code is C rather than C++, thus all the MIDI routines are standard C procedures. Since this code is Apple's only example of a MIDI application additional comments have been added to this thesis in blue to explain the example in greater detail.

First declare references to the output port through which MIDI will be sent and to the physical MIDI endpoint that will be attached to this output port

```
MIDIPortRef    gOutPort = NULL;
MIDIEndpointRef gDest = NULL;
.
.
```

Before any Core MIDI functions can be called the application must establish itself as a MIDI client

```
// create client and ports
MIDIClientRef client = NULL;
MIDIClientCreate(CFSTR("MIDI Echo"), NULL, NULL, &client);
```

Now create an input port for the new client and specify the application's `MyReadProc` function as the callback function to be automatically called by Core MIDI when MIDI is received, then create an output port to allow the application to send MIDI

```
MIDIPortRef inPort = NULL;
MIDIInputPortCreate(client, CFSTR("Input port"), MyReadProc, NULL, &inPort);
MIDIOutputPortCreate(client, CFSTR("Output port"), &gOutPort);
```

At this point the application has created a MIDI client with an input and output port but these ports have not been connected to physical MIDI endpoints. In a complete application the user would be given a means of selecting these from a list of available ports – `echo.cpp` does not implement this but [does](#) show how to get information about the available devices

The following code fragment determines how many physical devices are attached and then gets and prints the name, manufacturer and model strings for each device

```
// enumerate devices (not really related to purpose of the echo program
// but shows how to get information about devices)
```

```
int i, n;
CFStringRef pname, pmanuf, pmodel;
char name[64], manuf[64], model[64];

n = MIDIGetNumberOfDevices();
for (i = 0; i < n; ++i) {
    MIDIDeviceRef dev = MIDIGetDevice(i);

    MIDIObjectGetStringProperty(dev, kMIDIPropertyName, &pname);
    MIDIObjectGetStringProperty(dev, kMIDIPropertyManufacturer, &pmanuf);
    MIDIObjectGetStringProperty(dev, kMIDIPropertyModel, &pmodel);

    CFStringGetCString(pname, name, sizeof(name), 0);
    CFStringGetCString(pmanuf, manuf, sizeof(manuf), 0);
    CFStringGetCString(pmodel, model, sizeof(model), 0);
    CFRelease(pname);
    CFRelease(pmanuf);
    CFRelease(pmodel);

    printf("name=%s, manuf=%s, model=%s\n", name, manuf, model);
}
}
```

Rather than allowing the user to choose which MIDI device inputs to receive from, `echo.cpp` connects all available MIDI input endpoints to its input port:

```
// open connections from all sources
n = MIDIGetNumberOfSources();
printf("%d sources\n", n);
for (i = 0; i < n; ++i) {
    MIDIEndpointRef src = MIDIGetSource(i);
    MIDIPortConnectSource(inPort, src, NULL);
}
```

Similarly, the user does not choose which MIDI output to use, instead the MIDI output is echoed to the first available MIDI output endpoint, note that here is no destination equivalent of `MIDIPortConnectSource()`, instead the programmer obtains a reference to the required MIDI output endpoint and passes this to the `MIDISend()` function when sending MIDI

```
// find the first destination
n = MIDIGetNumberOfDestinations();
if (n > 0)
    gDest = MIDIGetDestination(0);
```

Having now created a client and connected MIDI input and output ports, the programme enters a continuous run loop awaiting MIDI messages that will automatically cause the `MyReadProc` callback function specified during initialisation to be called.

To understand the following code it is necessary to know that that the callback function is sent a list of MIDI packets (`MIDIPacketList`) containing one or more complete MIDI messages without running status, each message is supplied as a `MIDIPacket` – a C structure consisting of a length and number of data bytes, for example continuous controller message 1 with a value 127 would be represented by a `MIDIPacket` whose length was 3 and data was B0 01 7F (hex).

If there is an available MIDI output, this function iterates through the supplied packet list and re-sends the packet to the first available output:

```
static void MyReadProc(const MIDIPacketList *pktlist, void *refCon, void
                        *connRefCon)
{
    .
    .
    for (unsigned int j = 0; j < pktlist->numPackets; ++j) {
        for (int i = 0; i < packet->length; ++i) {
            .
            . the incoming MIDI packet is processed here
            .
            packet = MIDIPacketNext(packet);    get the next packet
        }
        MIDISend(gOutPort, gDest, pktlist);    echo the packet
    }
}
```

Having examined the data types and mechanisms for MIDI input and output using the Carbon function calls, the creation of the Cocoa **PFMidi** wrapper can now be considered. The aim of this MIDI class was to provide simple methods to allow a Cocoa application to perform operations such as sending continuous controller messages. Consider first the **PFMidi** method called **sendCC** requiring three parameters: Controller number, data and channel as follows:

```
[PFMidi sendCC:2 data:127 chan:1];    // send a CC#2 message whose value is
                                     127 using MIDI channel 1
```

Moving to the implementation of the **sendCC** method, the method declaration is similar to a C function and is preceded by a minus to indicate it is a method. In this example the **sendCC** method does not return a value hence the **(void)** declaration, the three arguments are integers and this is indicated by the **(int)** prefix

```
- (void) [ PFMidi sendCC:(int) ccNumber data(int) newValue chan:(int)
                                     chanNumber ]
{
```

```

    // method implementation
}

```

The low level transmission of MIDI data must use the Carbon MIDISend function which requires the client output port reference, the MIDI output endpoint reference and a MIDI packet list MIDI data to be sent. The implementation code within this method is essentially the same as Apple's Carbon echo.cpp example:

```

-(void)[ PFMidi sendCC:(int)ccNumber data(int)newValue chan:(int)chanNumber ]
{
    OSStatus status;

    MIDIPacket *packet = &outPacketList.packet[0]; // note: first packet in
                                                    list is the only one that can be
                                                    directly addressed

    outPacketList.numPackets = 1;

    packet->timeStamp = 0; // send immediately
    packet->length = 3; // one controller message
    packet->data[0] = CONTROLLERMSG | (chanNumber & 0xf); // status byte
                                                         for continuous controller message
                                                         with MIDI channel number OR'd into
                                                         the lower nibble

    packet->data[1] = ccNumber;
    packet->data[2] = newValue & 0x7f;

    status = MIDISend(gOutPort, gDest, &outPacketList);
    printf("CC - midiSend returned OSStatus %d\n", status);
}

```

Although writing this method requires knowledge of the CoreMIDI interface and Carbon, the object-orientated nature of Cocoa means its implementation is local to the **PFMidi** class and does not need to be understood by users of the class. Similar methods can be added to the **PFMidi** class to send MIDI note, pitch bend and program change messages. The Fish Creek Framework (Choi 2003 p1) and the PYMIDI Framework (Yandell 2002) are useful examples of Cocoa MIDI wrappers but were not used in this research project since neither provided the timecode and clock support required.

B.2 MIDI System Initialisation and Port selection

As noted earlier, Apple's MIDI echo example does not have a User Interface and does not allow selection of input and output MIDI ports which is unacceptable for a real-world application. The open source SynthTest (Reed 2002 p1) example populates a pop-up button with the enumerated MIDI input endpoints and this idea was used to provide the PFMidi object with MIDI input and output port pop-up buttons (Fig. B- 1)

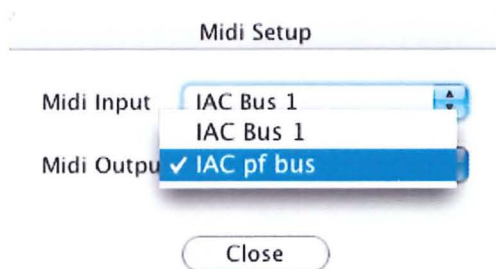


Fig. B- 1: MIDI port pop-up buttons

This is a simplified code fragment that shows the use of the PopUpButton class and its removeAllItems and addItemWithTitle methods to clear a pop-up button and populate it with the list of available MIDI input endpoints

```
// Build list of available ports
[inputPopup removeAllItems];

nDevices = MIDIGetNumberOfSources(); // how many MIDI inputs on this
                                     system?

if (nDevices > 0)
{
    for (i=0; i<nDevices ; ++i)
    {
        MIDIEndpointRef dev = MIDIGetSource(i); // get a reference

        if (dev != NULL)
        {
            MIDIObjectGetStringProperty(dev, kMIDIPropertyName, &pname);
            CFStringGetCString(pname, name, sizeof(name), 0);
            CFRelease(pname);
        }
    }
}
```



```

        [inputPopup addItemWithTitle: [NSString stringWithCString:
                                     name]];
    }
}
}

```

B.3 Calling Cocoa from a Carbon callback function

Decoding of the MTC quarter-note stream was implemented in the **PFMidi** class within the Carbon callback function called **myReadProc**. During Cocoa evaluation in February 2004 this function had proved to be very difficult to modify due to the lack of documentation and example code by Apple and thus it is explained in some detail here.

MyReadProc is based on the callback function described in Apple's `echo.cpp` example. The Cocoa MIDI examples presented so far have only performed MIDI output and this is straightforward even though a Carbon CoreMIDI function is being called from Cocoa. However, the CoreMIDI callback mechanism that calls the user's **myReadProc** function runs on a separate high priority thread¹⁰ and calling a Cocoa method from **myReadProc** caused the program to crash when any MIDI was received. After extensive searches in February 2004 no supporting documentation could be found and a request for help was placed on Apple's CoreAudio developers bulletin board¹¹.

The post received a single response from programmer Patrick Gustovic whose assistance was invaluable:

Paul.

When you call `MIDIInputPortCreate` from an Objective-C context, pass in "self" for the fourth argument (`refCon`). Then when your MIDI read proc is called, that "self" object will pop in as the second argument (`readProcRefCon`). Then you can call back into the context from which you setup the MIDI read proc.

¹⁰ OS X is a multi-threaded language wherein a program can have several totally separate processes (threads) running concurrently.

¹¹ It would appear that this is a common problem experienced by programmers new to Cocoa and MIDI as a search of the Apple CoreAudio List reveals similar requests for assistance dated after this posting.

i.e.

```
void myReadProc(const MIDIPacketList *pktlist, void *readProcRefCon,
void *srcConnRefCon)
{
    [(SomeObjC_Class*)readProcRefCon doSomething];
}
Patrick
```

To illustrate this response consider Apple's sample Carbon echo.cpp code:

First the callback routine **MyReadProc** is established when the input port is created:

```
MIDIInputPortCreate(client, CFSTR("Input port"), MyReadProc, NULL, &inPort);
```

Then the **MyReadProc** function is defined:

```
static void MyReadProc(const MIDIPacketList *pktlist, void *refCon, void
                        *connRefCon)
{
    . // code removed here
}
```

It can be seen here that the fourth argument Gustovic refers to is set to `NULL` when **MyReadProc** is referenced and is not used by the `MYReadProc` example code. To allow the **MyReadProc** callback function to perform Cocoa methods the `NULL` reference currently passed to **MyReadProc** when MIDI is received should be changed to refer to the **PFMidi** class that created it, i.e. `self`

```
MIDIInputPortCreate(client, CFSTR("Input port"), MyReadProc, self, &inPort);
```

Thus the second argument passed to `MyReadProc` is now a reference to the **PFMidi** class:

```
static void MyReadProc(const MIDIPacketList *pktlist, void *refCon, void
                        *connRefCon)
{
    . // code removed here
    .
    [(PFMidi *)refCon doSomething] // ask the PFMidi class to perform
    the required method
```

```
}
```

Note that `refCon` is coerced to be a pointer to a **PFMidi** class which stops Xcode warning that the targeted class might not implement the method required.

Appendix C Investigating CoreAudio Clock

Apple's document entitled "Core Audio Overview" includes a brief description of the technology and refers the programmer to `CoreAudioClock.h` in the `AudioToolbox` framework. This header documents the `CoreAudio` clock system calls in a concise manner aimed at the experienced Carbon Mac programmer. A brief examination of the list below will show that functions are included to create and dispose of clocks and to configure, start and stop them. Additional functions appear to allow conversion between different time representations such as host-time, SMPTE and bars/beats:

`CAClockAddListener` - Adds a callback function to receive notifications of changes to the clock's state.

`CAClockArm` - Allow received sync messages to start the clock.

`CAClockBarBeatTimeToBeats` - Converts a `CABarBeatTime` structure to a number of beats.

`CAClockBeatsToBarBeatTime` - Converts a number of beats to a `CABarBeatTime` structure.

`CAClockDisarm` - Disallow received sync messages from starting the clock.

`CAClockDispose` - Dispose a clock object.

`CAClockGetCurrentTempo` - Obtain the clock's current musical tempo.

`CAClockGetCurrentTime` - Obtain the clock's current position on the media timeline.

`CAClockGetPlayRate` - Obtain the clock's playback rate.

`CAClockGetProperty` Gets the current value of a clock's property.

`CAClockGetPropertyInfo` - Gets information about a clock's property.

`CAClockGetStartTime` - Obtain the position on the media timeline where playback will start, or has already started.

`CAClockNew` - Create a new clock object.

`CAClockRemoveListener` - Removes a listener callback function.

`CAClockSecondsToSMPTETime` - Converts seconds to a SMPTE time representation.

`CAClockSetCurrentTempo` - Manually override the clock's musical tempo during playback.

`CAClockSetCurrentTime` - Sets the clock's current position on the media timeline.

`CAClockSetPlayRate` - Alter the clock's playback rate.

`CAClockSetProperty` - Changes the value of a clock's property.

`CAClockSMPTETimeToSeconds` - Converts a SMPTE time representation to seconds.

`CAClockStart` - Begin advancing the clock on its media timeline.

`CAClockStop` - Stop advancing the clock on its media timeline.

`CAClockTranslateTime` - Convert between time units.

CoreAudioClock.h also presents each of the functions in a slightly expanded form, for example:

CAClockNew

Create a new clock object.

```
extern OSStatus CAClockNew(UInt32 inReservedFlags, CAClockRef *outCAClock);
```

Parameters

`inReservedFlags` - Must be 0.

`outCAClock` - Must be non-null. On successful return, the new clock object.

Return Value

An OSStatus error code.

Availability

Introduced in Mac OS X v10.4.

Again, minimal documentation aimed at providing a reference for an experienced programmer. This format is used for Core Audio in general, however, Core Audio has additional documentation and example code to help introduce the key concepts to programmers.

It was assumed that a clock object was analogous to a CoreMIDI client, in that it must be created and a reference to it used in subsequent function calls. The above description of `CAClockNew` indicates that the clock object will be referenced using a `CAClockRef` suggesting the following test code:

```
CAClockRef pfClockRef;

osErr = CAClockNew(0, &pfClockRef); // create a new clock
printf("clockNew returned %d\n", osErr);
```

Running this code printed 0 to the console indicating successful creation of a clock. This new `CAClockRef` was then passed to `CAClockStart` and `CAClockStop` as follows:

```
osErr = CAClockStart(pfClockRef);
printf("clock start returned %d\n", osErr);
.
.
osErr = CAClockStop(pfClockRef);
printf("clock stop returned %d\n", osErr);
```

The returned 0 OSStatus code indicated that the newly created clock object could be started and stopped successfully. Now calling the `CAClockGetCurrentTime` function would allow verification that a running clock actually existed:

```
CAClockGetCurrentTime
```

Obtain the clock's current position on the media timeline.

```
extern OSStatus CAClockGetCurrentTime(
    CAClockRef inCAClock,
    CAClockTimeFormat inTimeFormat,
    CAClockTime *outTime);
```

Parameters

`inCAClock` - The clock object.

`inTimeFormat` - Specifies the desired format for `outTime`.

`outTime` - On exit, the clock's current time position.

It appeared from the above that to obtain the current clock time, the programmer must specify the required time format which would then be returned as a **CAClockTime** variable

Apple explains the available clock time formats as follows:

`kCAClockTimeFormat_HostTime` - Absolute host time




```
printf("currentTime... returned err %d, time %f\n",osErr,
      pfTimestamp.time.seconds);
```

Having now established that a clock object could be created, started and read, a Cocoa UI was created to allow the above Carbon code to be tested further, The simple UI shown in Fig. C- 1 creates a **CAClock** object on initialisation and has buttons allowing the user to call the **CAClockStart** and **CAClockStop** functions and to check the clock start time and current time using the **CAClockGetCurrentTime** and **CAClockGetStartTime** functions.

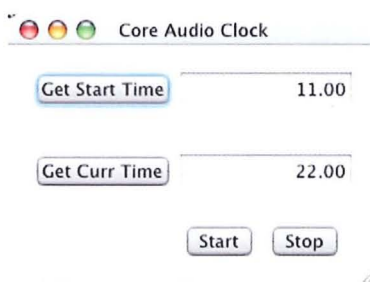


Fig. C- 1: Simple Core Audio Clock test application

Unpredicted behaviour

This test application revealed an apparent anomaly: as expected the newly created clock is stationary, its Start Time is zero and the Current Time is zero; the clock starts when **CAClockStart** is sent and the time updates as expected each time 'Get Current Time' is pressed. However, when **CAClockStop** is sent the clock does not stop advancing as suggested by the **CAClockStop** documentation, pressing 'Get Current Time' shows that the time is still advancing.

It was also observed that when stop is pressed the current time is stored as the new start time but this only happens for the first stop press following a start press. This may indicate a MIDI sequence type of transport behaviour where Stop acts as a timeline pause and Play continues from the position stopped at¹².

¹² This mode corresponds to the "insertion follows playback" option in Pro Tools

After several days of experimentation and literature review this anomaly was flagged on the Apple Developer support forum and received the following response from Apple:

Subject: Re: my CoreAudio Clock won't stop!
 From: Doug Wyatt <email@hidden>
 Date: Wed, 17 Jan 2007 12:30:46 -0800
 Delivered-to: email@hidden
 Delivered-to: email@hidden

That's not something that's turned up in our internal test code.

It'd be best to file a bug with a reproducible test case.

Thanks
 Doug

Fig. C- 2: Response by Doug Wyatt to the CAClock anomaly

It is interesting to note that Doug Wyatt is the programmer responsible for Opcode's OMS and subsequently OS X Core MIDI, as a result of our email exchange a formal developer bug report was filed with Apple. A formal response was received from Apple two weeks later indicating that the test program behaved as expected by Apple, sending **CAClockStop** does NOT stop the clock (as implied by the function documentation), instead it initiates a callback to the user's callback function indicating that the clock has stopped (although the clock continues to run).

In practical terms this means the programmer must implement a Carbon callback function to respond to changes in the clock state and only update time displays when the clock is indicated as running. The reason for this seemingly complex approach became clear when the test code was extended to allow external SMPTE or MIDI Clock control. At this point the transport play/stop state is under external control rather than UI control, the callback mechanism allows an application UI control and external control the same way.

Clues to the implementation of the callback mechanism can be derived from the CAClockAddListener function and the Core MIDI echo.cpp example.

CAClockAddListener – “adds a callback function to receive notifications of changes to the clock's state.”

```
extern OSStatus CAClockAddListener(CAClockRef inCAClock,
```


changes to the clock's state."

```
extern OSStatus CAClockAddListener(CAClockRef inCAClock,
    CAClockListenerProc inListenerProc, void *inUserData);
```

Parameters

`inCAClock` - The clock object.

`inListenerProc` - The callback function.

`inUserData` - This value is passed to the callback function, in the `userData` parameter.

In use this function could immediately follow the creation of a clock object during program initialisation to assign a callback procedure to the new clock. `CoreAudioClock.h` documents `inListenerProc` as follows:

A client-supplied function called when the clock's state changes.

```
typedef void (*CAClockListenerProc)(
    void *userData,
    CAClockMessage message,
    const void *param);
```

Parameters

`userData` - The value passed to `CAClockAddListener` when the callback function was installed.

`Message` - Signifies the kind of event which occurred.

`Param` - This value is specific to the message (currently no messages have values).

Critical to understanding callback use is the message field indicating what type of event has triggered the callback, `CAClockMessage` is documented as follows:

`kCAClockMessage_StartTimeSet` - A new start time was set or received from an external sync source.

`kCAClockMessage_Started` - The clock's time has started moving.

`kCAClockMessage_Stopped` – The clock's time has stopped moving.

`kCAClockMessage_Armed` - The client has called `CAClockArm()`.

`kCAClockMessage_Disarmed` - The client has called `CAClockDisarm()`.

`kCAClockMessage_PropertyChanged` - A clock property has been changed.

`kCAClockMessage_WrongSMPTEFormat` - The clock is receiving SMPTE (MTC) messages in a SMPTE format that does not match the clock's SMPTE format.

Once the significance of the callback procedure to CoreAudio Clock use was understood by the author, a comprehensive version of the test application was written over a one-month period. The UI (Fig. C- 3) allows the user to specify internal, external SMPTE or MIDI clock as the timebase. The `CAClockSetProperty` function has been used to allow the user to set the MIDI source when externally synchronised and the MIDI destination for generated timecode.

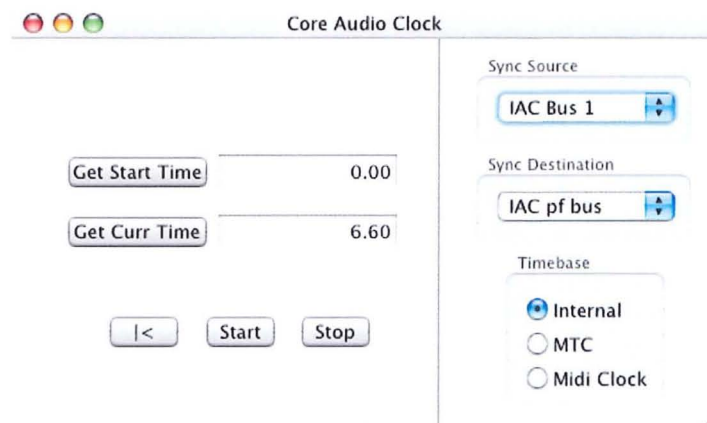


Fig. C- 3: CoreAudio Clock Test App v2

This successful test application demonstrated that Apple's CoreAudio Clock Carbon functions could now be incorporated into the **ShapePanner** application and that it would not be necessary to wait for Apple to publish example code.

Appendix D PFPan2x4 VST plug-in source code

```

//-----
//
// 16/4/2008
// Filename      : pfpan.h
// Created by    : Paul Ferguson copyright 2008 All Rights Reserved
// Description   : Simple 3D panner plugin (Mono->Mono)
// using SDK 2.4 code by Steinberg Media Technologies
//-----
//
//
#ifndef __PFPan__
#define __PFPan__

#include <audioeffectx.h>
#include <math.h>

#define SEND 0
#define RECEIVE 1
#define SPEAKER1 0
#define SPEAKER2 1
#define SPEAKER3 2
#define SPEAKER4 3

#define SPEAKER9 8
#define SPEAKER10 9
#define SPEAKER11 10
#define SPEAKER12 11

const int MAX_BUFFSIZ = 4096;
const int MAX_CHANNELS = 16;
const int MAX_INSTANCES = 16;

const int XAXIS_CC = 12;
const int YAXIS_CC = 13;

```

```
const int ZAXIS_CC = 14;
const int VOLUME_CC = 7;
```

```
enum
{
    // Global
    kNumPrograms = 1,

    // Parameters Tags
    kMode = 0,
    kSpeaker,
    kXAxis,
    kYAxis,
    kZAxis,
    kVolume,

    kNumParams
};
```

```
class PFPan;
```

```
//-----
```

```
class PFPanProgram
{
    friend class PFPan;
public:
    PFPanProgram ();
    ~PFPanProgram () {}
```

```
private:
    float fMode;
    float fSpeaker;
    float fXAxis;
    float fYAxis;
    float fZAxis;
```

```

    float fVolume;
    char name[24];
};

//-----
class PFPan : public AudioEffectX
{
public:
    PFPan (audioMasterCallback audioMaster);
    ~PFPan ();

    //---from AudioEffect-----
    virtual void processReplacing (float** inputs, float** outputs, VstInt32
                                   sampleFrames);
    virtual VstInt32 processEvents (VstEvents* events); // handle midi control
                                                         events

    virtual VstInt32 PFPan::canDo (char* text);
    virtual VstInt32 PFPan::getNumMidiInputChannels ();
    virtual VstInt32 PFPan::getNumMidiOutputChannels ();

    virtual void setProgram (VstInt32 program);
    virtual void setProgramName (char* name);
    virtual void getProgramName (char* name);
    virtual bool getProgramNameIndexed (VstInt32 category, VstInt32 index,
                                         char* text);

    virtual void setParameter (VstInt32 index, float value);
    virtual float getParameter (VstInt32 index);
    virtual void getParameterLabel (VstInt32 index, char* label);
    virtual void getParameterDisplay (VstInt32 index, char* text);
    virtual void getParameterName (VstInt32 index, char* text);

    virtual void resume ();

    virtual bool getEffectName (char* name);

```

```

virtual bool getVendorString (char* text);
virtual bool getProductString (char* text);
virtual VstInt32 getVendorVersion () { return 1000; }

virtual VstPlugCategory getPlugCategory () { return kPlugCategEffect; }

```

protected:

```

PFPanProgram* programs;

float* buffer;
float fMode;
float fSpeaker;
float fXAxis, fYAxis, fZAxis, fVolume;

float oldX, oldY, oldZ, oldVolume;

float mode;
int sendInstanceNum; // each SEND instance must be set to a unique 1-16
                    // number by the user
int receiveSpeakerNum; // for RECEIVE instances - speaker number from 1 to
                    // 16 derived from float 0.0 to 1.0
                    // value supplied by interface

};

#endif

```



```

#pragma data_seg()

// now instruct the linker that this shared section is Read Write Shared
                                (RWS)
#pragma comment(linker, "/section:PFSHARED, RWS")

//-----
                                --
AudioEffect* createEffectInstance (audioMasterCallback audioMaster)
{
    return new PFPan (audioMaster);
}

//-----
                                --

PFPanProgram::PFPanProgram ()
{
    // default Program Values
    fMode = SEND; // SEND is 0.0
    fSpeaker = 0; // speaker 1 of 16
    fXAxis = 0.5; // centred
    fYAxis = 1.0; // front
    fZAxis = 0.0; // bottom
    fVolume = 1.0; // 0 dB

    strcpy (name, "Init");
}

//-----
                                --

PFPan::PFPan (audioMasterCallback audioMaster)
    : AudioEffectX (audioMaster, kNumPrograms, kNumParams)

```



```

//suspend ();

resume ();      // flush buffer
}

//-----
PFPan::~PFPan ()
{
    if (buffer)
        delete[] buffer;
    if (programs)
        delete[] programs;
}

//-----
void PFPan::setProgram (VstInt32 program)
{
    PFPanProgram* ap = &programs[program];

    curProgram = program;
    setParameter (kMode, ap->fMode);
    setParameter (kSpeaker, ap->fSpeaker);
    setParameter (kXAxis, ap->fXAxis);
    setParameter (kYAxis, ap->fYAxis);
    setParameter (kZAxis, ap->fZAxis);
    setParameter (kVolume, ap->fVolume);
}

//-----
void PFPan::setProgramName (char *name)
{
    strcpy (programs[curProgram].name, name);
}

```

```

//-----
void PFPan::getProgramName (char *name)
{
    if (!strcmp (programs[curProgram].name, "Init"))
        sprintf (name, "%s %d", programs[curProgram].name, curProgram + 1);
    else
        strcpy (name, programs[curProgram].name);
}

//-----
-----

bool PFPan::getProgramNameIndexed (VstInt32 category, VstInt32 index, char*
                                   text)
{
    if (index < kNumPrograms)
    {
        strcpy (text, programs[index].name);
        return true;
    }
    return false;
}

//-----

void PFPan::resume ()
{
    // (buffer, 0, size * sizeof (float));
    AudioEffectX::resume ();
}

//-----

void PFPan::setParameter (VstInt32 index, float value)
{
    PFPanProgram* ap = &programs[curProgram];

    switch (index)

```

```

{
    case kMode : fMode = ap->fMode = value;
                break;
    case kSpeaker : fSpeaker = ap->fSpeaker = value;
                  sendInstanceNum = receiveSpeakerNum = (int)
                    ceilf(fSpeaker*15); // the range is
                    0 to 15 to match sneakyBuffer - NOT
                    1 to 16
// clear all the buffers to prevent stale samples being added into the busses
    for (int i = 0; i < MAX_CHANNELS; i++)
    {
        for (int j = 0; j < MAX_INSTANCES; j++)
        {
            for (int k = 0; k < MAX_BUFFSIZ; k++)
            {
                sneakyBuffer[j].audio[i][k] = 0;
            }
        }
    }
    break;
    case kXAxis :
        oldX = fXAxis; // old x,y,z hold the previous
                      value to be used for de-zippering
        fXAxis = ap->fXAxis = value;
        break;
    case kYAxis : oldY = fYAxis;
                  fYAxis = ap->fYAxis = value;
                  break;
    case kZAxis : oldZ = fZAxis;
                  fZAxis = ap->fZAxis = value;
                  break;
    case kVolume : oldVolume = fVolume;
                   fVolume = ap->fVolume = value;
                   break;
}
}

```

```

//-----
float PFPan::getParameter (VstInt32 index)
{
    float v = 0;

    switch (index)
    {
        case kMode :    v = fMode;    break;
        case kSpeaker : v = fSpeaker; break;
        case kXAxis :   v = fXAxis;    break;
        case kYAxis :   v = fYAxis;    break;
        case kZAxis :   v = fZAxis;    break;
        case kVolume :  v = fVolume; break;
    }
    return v;
}

//-----
void PFPan::getParameterName (VstInt32 index, char *label)
{
    switch (index)
    {
        case kMode :    strcpy (label, "Mode");    break;
        case kSpeaker : strcpy (label, "Send/Speaker"); break;
        case kXAxis :   strcpy (label, "X axis");   break;
        case kYAxis :   strcpy (label, "Y axis");   break;
        case kZAxis :   strcpy (label, "Z axis");   break;
        case kVolume :  strcpy (label, "Volume");   break;
    }
}

//-----
void PFPan::getParameterDisplay (VstInt32 index, char *text)
{
    switch (index)

```

```

{
    case kMode :    //int2string ((int)fMode, text, kVstMaxParamStrLen);
                    break;

    case kSpeaker : int2string ((int)(ceilf(fSpeaker*15)+1), text,
                               kVstMaxParamStrLen); break;

    case kXAxis :   float2string (fXAxis, text, kVstMaxParamStrLen); break;
    case kYAxis :   float2string (fYAxis, text, kVstMaxParamStrLen); break;
    case kZAxis :   float2string (fZAxis, text, kVstMaxParamStrLen); break;
    case kVolume :  dB2string (fVolume, text, kVstMaxParamStrLen);
                    break;
}
}

//-----
void PFPan::getParameterLabel (VstInt32 index, char *label)
{
    switch (index)
    {
        case kMode :    if (fMode <0.5) strcpy (label, "Send");
                        else strcpy (label, "Receive"); break;

        case kSpeaker : break;

        case kXAxis :   strcpy (label, "");      break;
        case kYAxis :   strcpy (label, "");      break;
        case kZAxis :   strcpy (label, "");      break;
        case kVolume :  strcpy (label, "dB");    break;
    }
}

//-----
bool PFPan::getEffectName (char* name)
{
    strcpy (name, "PFPan");
    return true;
}

//-----

```



```

bool PFPan::getProductString (char* text)
{
    strcpy (text, "PFPan");
    return true;
}

//-----
bool PFPan::getVendorString (char* text)
{
    strcpy (text, "PFergy");
    return true;
}

//-----MIDI stuff starts here - PF

VstInt32 PFPan::canDo (char* text)
{
    if (!strcmp (text, "receiveVstEvents"))
        return 1;
    if (!strcmp (text, "receiveVstMidiEvent"))
        return 1;

    return -1; // explicitly can't do; 0 => don't know
}

//-----
--

VstInt32 PFPan::getNumMidiInputChannels ()
{
    return 1; // we are monophonic
}

//-----
--

VstInt32 PFPan::getNumMidiOutputChannels ()

```

```

{
    return 0; // no MIDI output back to Host app
}

//-----
--
VstInt32 PFPan::processEvents (VstEvents* ev)
{
    float x;

    for (VstInt32 i = 0; i < ev->numEvents; i++)
    {
        if ((ev->events[i])->type != kVstMidiType)
            continue;

        VstMidiEvent* event = (VstMidiEvent*)ev->events[i];
        char* midiData = event->midiData;
        VstInt32 status = midiData[0] & 0xf0; // ignore the channel nibble

        if (status == 0xb0) // must be a CC message
        {
            x = (float) (midiData[2] & 0x7f);

            if (midiData[1] == XAXIS_CC) // Kaoss pad x-axis
                setParameter (kXAxis, x/127);

            else if (midiData[1] == YAXIS_CC) // Kaoss pad y-axis
                setParameter (kYAxis, x/127);

            else if (midiData[1] == ZAXIS_CC) // Z-axis
                setParameter (kZAxis, x/127);

            else if (midiData[1] == VOLUME_CC) // volume

```

```

        setParameter (kVolume, x/127);
    }
    event++;
}
return 1;
}

//-----
void PFPan::processReplacing (float** inputs, float** outputs, VstInt32
                             sampleFrames)
{
    float* in = inputs[0];
    float* out1 = outputs[0];
    float sampleValue;

    // de-zipper any change in X, Y or Z value by gradually adding in this
    // delta value
    float deltaX = (fXAxis - oldX)/(sampleFrames);
    float deltaY = (fYAxis - oldY)/(sampleFrames);
    float deltaZ = (fZAxis - oldZ)/(sampleFrames);
    float deltaVolume = (fVolume - oldVolume)/(sampleFrames);

    for (int i = 0; i<sampleFrames; i++)
    {
        sampleValue = *in++; // read the input buffer, we'll only use this value
                             // if the plug-in is in SEND mode

        if (fMode < 0.5) // SEND mode so write to shared buffer
        {
            oldX+=deltaX; // each iteration round the loop will increment the old
                          // X,Y,Z and volume values until they
                          // reach the current values

            oldY+=deltaY;
            oldZ+=deltaZ;
            oldVolume +=deltaVolume;
        }
    }
}

```

```

sneakyBuffer[sendInstanceNum].audio[SPEAKER1][i] = sampleValue * (1-oldX) *
    oldY * (1-oldZ) * oldVolume; //
    lower left front
sneakyBuffer[sendInstanceNum].audio[SPEAKER2][i] = sampleValue * oldX * oldY
    * (1-oldZ) * oldVolume; // lower
    right front
sneakyBuffer[sendInstanceNum].audio[SPEAKER3][i] = sampleValue * (1-oldX) *
    (1-oldY) * (1-oldZ) * oldVolume; //
    lower left rear
sneakyBuffer[sendInstanceNum].audio[SPEAKER4][i] = sampleValue * oldX * (1-
    oldY) * (1-oldZ) * oldVolume; //
    lower right rear

// sneakyBuffer[sendInstanceNum].audio[SPEAKER5][i] = sampleValue * //
    speaker not in use
// sneakyBuffer[sendInstanceNum].audio[SPEAKER6][i] = sampleValue * //
    speaker not in use
// sneakyBuffer[sendInstanceNum].audio[SPEAKER7][i] = sampleValue * //
    speaker not in use
// sneakyBuffer[sendInstanceNum].audio[SPEAKER8][i] = sampleValue * //
    speaker not in use

sneakyBuffer[sendInstanceNum].audio[SPEAKER9][i] = sampleValue * (1-oldX) *
    oldY * oldZ * oldVolume; // upper
    left front
sneakyBuffer[sendInstanceNum].audio[SPEAKER10][i] = sampleValue * oldX * oldY
    * oldZ * oldVolume; // upper right
    front
sneakyBuffer[sendInstanceNum].audio[SPEAKER11][i] = sampleValue * (1-oldX) *
    (1-oldY) * oldZ * oldVolume; //
    upper left rear
sneakyBuffer[sendInstanceNum].audio[SPEAKER12][i] = sampleValue * oldX * (1-
    oldY) * oldZ * oldVolume; // upper
    right rear

// sneakyBuffer[sendInstanceNum].audio[SPEAKER13][i] = sampleValue * //
    speaker not in use

```

```
// sneakyBuffer[sendInstanceNum].audio[SPEAKER14][i] = sampleValue * //
// speaker not in use
// sneakyBuffer[sendInstanceNum].audio[SPEAKER15][i] = sampleValue * //
// speaker not in use
// sneakyBuffer[sendInstanceNum].audio[SPEAKER16][i] = sampleValue * //
// speaker not in use

*out1++ = sampleValue; // pass it to this plug-in's output
}

else // RECEIVE mode so read from shared buffer (ignore panner axes)
{
// read all the 16 possible SEND instance buffers and add them together
sampleValue = 0; // clear the sample value,

for (int j = 0; j<MAX_INSTANCES; j++)
{
sampleValue += sneakyBuffer[j].audio[receiveSpeakerNum][i];
}
*out1++ = sampleValue * oldVolume; // scale it then send it to the
// plug-in output
}
}
}
```