Real-time Sound Spatialization, Software Design and Implementation

David Robert Moore

Submitted for the degree of PhD

Department of Music

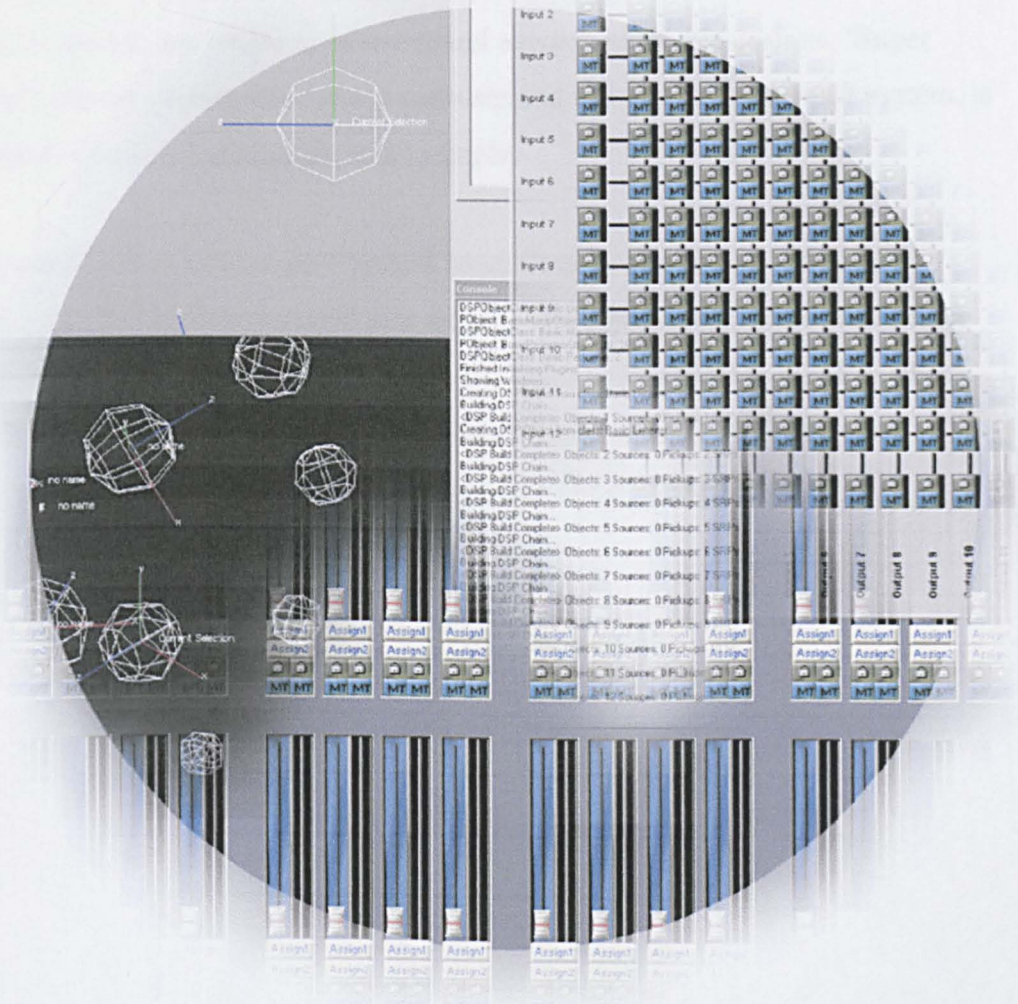
September 2004

THESIS CONTAINS

CD

Real-time Sound Spatialization

Software Design and Implementation



Abstract

'Real-time Sound Spatialization, Software Design and Implementation' explores real-time spatialization signal processing for the sound artist. The thesis is based around the production of two prototype software projects, both of which are examined in design and implementation.

The first project examines a conceptual method for performance based spatialization mixing which aims to expand existing analogue designs. 'Super Diffuse', proven performance grade software and the encompassing M2 system, is submitted, for model evaluation and example.

The second project focuses on Physical Modelling Synthesis and introduces 'Source Ray Pickup Interactions' as a tool for packaging real-time spatialization digital signal processing. Submitted with the theoretical model is the 'Ricochet' software, an implementation of 'Source Ray Pickup Interaction'. 'Ricochet' serves as a model evaluation tool and example of implementation.

Acknowledgments

I would sincerely like to thank the following for their support: Adrian Moore (University of Sheffield), for his academic guidance and personal support throughout my PhD study; Braham Hughes (University of Huddersfield) and Bob Davis, for their support of both my application to Sheffield and post-graduate work as a whole; Jonty Harrison (University of Birmingham), Andrew Lewis (University of Bangor), Robert Dow (University of Edinburgh) and James Mooney (University of Sheffield) for their contributions to development and presentation of the M2 system; Richard Holmes (Digidesign UK), for continued personal and professional guidance; All the staff at the University of Sheffield Music Department, who provided the academic and administrative backup necessary for my study; All my friends and family, who provided me with much appreciated support.

Contents

Preface

In 1998, during the second year of my study for BA in Creative Music Technology, I decided to produce a piece of sound panning software influenced by a workshop hosted in Manchester's 'Green Room'. The workshop was my first experience of both electro-acoustic works and the performance of live stereo sound diffusion and I was struck by the complexity of controlling spatialization of sound. During the homeward journey a discussion of the workshop with my undergraduate supervisor Braham Hughes led to a proposal for a project to link Opcode MAX software control with the Yamaha O1V mixer producing a quadraphonic surround panning system. During 1999 I was introduced to the digital signal processing extension to MAX, MSP. Dissatisfied with the graphical interface possibilities of MAX, I decided to create a dual component octa-phonic panning system with separate GUI software and MAX/MSP processing. 'Deep Pan', produced for the project, was my first large scale attempt at improving both the control of sound source positioning and the quality of its realization.

During rehearsals for the first, and only, demonstration of 'Deep Pan' I was introduced to Dr Adrian Moore of The University of Sheffield and we discussed the possibility of researching software tools for sound spatialization. It was very clear from this meeting that we were both interested in producing fully working tools that would be used by resident composers.

Having accepted the invitation to study for a PhD at Sheffield, I spent the summer investigating tools and technologies available for audio software development. At the time I believed that the forefront of future audio software design would be with host based audio processing, a personal opinion I still hold today. I became interested in Steinberg's VST (Virtual Studio Technology) plugin and ASIO (Audio Streaming Input Output) systems. In order to develop software with these technologies it was simplest to use C++ and I spent the majority of the summer of 1999 learning both the language, Steinberg API's and PC GUI related API's: Win32, MFC, OpenGL and DirectX.

The first software I produced at Sheffield was a suite of plugins to perform 3D panning inside Steinberg's Cubase VST. At the time the VST SDK did not have multi-channel capability for anything more complex than stereo. In order to write this software I designed a background audio bussing system that could sit behind the processing engine of Cubase. Although this method performed correctly, it was essentially a software 'hack' and was always prone to error or incompatibility with future Cubase versions. It became clear from this early work that the VST SDK1 would not be suitable for multi-channel spatialization[1] and I focused my attention on designing stand alone applications for sound spatialization.

My first developments of stand alone software at Sheffield were aimed at producing 3D panning systems and these very experimental attempts culminated in the Source Ray Pickup Interaction concept and the first usable prototype of 'Ricochet' presented in this thesis. At this point my understanding of live sound diffusion was becoming more focused and while setting up stage lighting for the 'Sound Junction 2003' concert I was inspired by the venue's digital lighting console to design a similar parameter mapping system for an audio mix matrix. This became the basis of 'Super Diffuse' and, later, Sheffield's M2 diffusion system.

While researching some of the current techniques in audio software development I have on a number of occasions been faced with a choice of development language. C++ is my language of choice because it is widely documented and the object orientated conceptualization appeals to me. Obviously there are other popular OO languages, JAVA or Smalltalk being first to come to mind. Whilst I appreciate that other languages could certainly have been used, I have found C++ an ideal language for audio development. Furthermore, in learning C++ or any other language, I have personally found code examples to be an efficient method

---

[1] Steinberg's VST SDK2 now supports multi-channel plugins for use in its current flagship software Nuendo. During my initial research I made a number of posts to the VST SDK development mailing list regarding both a necessity for multi-channel capability and the discussion of 'workarounds' that had been made by myself and others.

of understanding the exact workings of a software example. I therefore present real annotated and commented code examples from the actual research software.

Software engineering is a field I have approached from a background in music and creative music technology. This thesis illustrates and complements my research into creative spatial audio production tools.

The real-time tools designed and implemented in this thesis represent my attempts to expand the current creative toolset. I present two tools, each representing one sound spatialization school of thought; live projection of pre-composed works and simulation of physical properties of sound. In both designs I focus on the need for appropriate control structures. Further, I highlight the possibilities of expandable frameworks and show, by example, how such systems can be implemented.

D. Moore, University of Sheffield 2004

1 Introduction

1.1 Why spatialize?

'To deploy space is to choreograph sound: positioning sources and animating movement.' [2]

As humans, an ability to hear the world in three dimensions augments the other senses in making us spatially aware of our surroundings. Primarily this spatial hearing is a survival advantage but it also provides the composer and performer with scope for artistic experimentation.

'The very nature of acoustic transmission of sound within the air as a medium invites composers to manipulate spatial properties in performance. As soon as more than one performer is present there is a spatial element' [3]

Spatialization of sound is commonplace in modern sound art but its use has been long documented prior to the dawn of stereo or even the electronic age.

Stevenson and Zvonor[4] describe the antiphonal[5] music of the medieval church as the first historical example of spatial performance before highlighting Renaissance Venice and specifically Basilica San Marco. Considered the central location for Venetian music during the 16th century, the basilica housed two spatially separated organs and choir lofts[6] prompting the cori spezzati[7] compositional technique. Adrian Willaert, appointed maestro di cappella[8] of Basilica San Marco in 1527, composed the earliest known work in this style,

---

[2] Roads, C: 1997

[3] Stevenson, I: 2000

[4] Zvonor, R: 1999

[5] Antiphonal: (Greek, meaning 'sounding across') A religious chant sung as responses between a single voice and a group of voices, or between two different groups of singers. The effect is described as antiphonal.

[6] Farrell, B: 2001

[7] Cori spezzati: (Italian, meaning 'broken choir')

[8] Maestro di cappella: (Italian, meaning 'director of chapel choir')

Vespers (1550), featuring 'dialog form' and echo effects. The practice of cori spezzati was continued and extended by Andrea Gabrieli and his nephew Giovani Gabrieli, producing works with up to five choirs.

These techniques became more widespread throughout Europe. Zvonor notes *Spem in alium* (1573) by Thomas Tallis featuring 40 separate vocal parts arranged into eight 5-voice choirs and Orazio Benevoli's *Festal Mass* (1628), having 53 parts with two organs and basso continuo. Stevenson further highlights the use of antiphonal choral effects in J.S.Bach's *St. Matthew Passion* and Mozart's *Serenade in D for 4 Orchestras*

Spatial sound became used for theatrical effect during the Romantic period. Examples include Hector Berlioz's *Requiem* (1837), Giuseppe Verdi's Manzoni *Requiem* (1874) and Gustav Mahler's *Symphony No. 2* (1895). In these examples brass ensembles perform either offstage or enter from locations offstage.

During the 20th century similar use of moving, offstage or spatially separated ensembles is employed in Charles Ives' *The Unanswered Question* (1908). Henry Brant's *Antiphony I* (1953), *Voyage Four* (1963) and *Windjammer* (1969) take these ideas further.

Pierre Henry and Pierre Schaeffer created a repertoire of works for tape for playback using a 4 channel tetrahedral configuration of loudspeakers, with Front Left, Front Right, Back, and Overhead. In order to control distribution of the sound, Schaeffer used an interface named the potentiomètre d'espace (1951) which used induction coils to control the signal routing. The photograph below shows Pierre Henry performing with the potentiomètre d'espace during a concert at the Salle de L'Ancien Conservatoire, Paris, 1952[9]. It is perhaps the first example of a real-time spatialization mixing and control system.

[9] Palombini, C: 1999

Stockhausen's *Kontakte* (1960) is an example of a composed work spatialized for quadraphonic playback. Stockhausen made use of rotating loudspeakers surrounded by microphones in order to obtain the effect of rotating sound sources. Essentially Stockhausen is in effect creating a real physical modelling system in order to spatialize, i.e. a simplified system in order to produce the effects of a complicated one.

The Philips Pavilion, an Installation of the Brussels Worlds' Fair, featured the tape composition *Poeme Electronique* (1958) by Edgard Varèse. The sound system used 11 independent channels and projected these channels though 425 loudspeakers located around the pavilion. In order to control sound projection motion effects were recorded and reproduced from a control tape.

Following the Brussels system, Stockhausen continued to employ space in his composition, notably producing works for performance in the German pavilion at Osaka EXPO 70. The pavilion was a geodesic dome containing 55 loudspeakers arranged in 7 rings fully surrounding both performer and listener. Stockhausen commented on the experience of the pavilion:

> *'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.'*

These large scale performances prompted the use of spatialization in multimedia installations with two notable examples, John Cage and Lejaren Hiller's *HPSCHD* and David Tudor's *Rainforest IV (1973)*. Both audio/visual installations made use of multiple loudspeakers but the latter also explored the acoustical characteristics of resonant sculptures hanging in the performance space.

In 1964, John Chowning, a graduate of Stanford University, created a software system on the Music IV computer that allowed a synthesized sound source to be moved along a user defined trajectory[10]. Amplitude, doppler and reverberation localisation cues were generated by the program and written to a four channel quadraphonic tape. This is the first example of a computer being used to control and create spatialized sounds.

1.2 The need for sound spatialization tools.

In the recently published book 'Spatial Sound', Rumsey talks briefly about the increasing consumer interest in spatially encoded sound.

> *'The later part of the twentieth century, particularly the last ten years gave rise to a rapid growth in systems and techniques designed to enhance the spatial quality of reproduced sound, particularly for consumer applications. Larger numbers of loudspeakers became common and systems capable of rendering fully three-dimensional sound images were realised by means of the digital signal processing available in relatively low cost products.'*[11]

Rumsey goes on to comment on new audio media formats acting as the catalyst for this increasing interest:

---

[10] Chowning, J: 1971

[11] Rumsey, F: 2001, p.ix

> *'In recent years the development of new consumer audio formats such as DVD, and digital surround formats for cinema and broadcasting such as Dolby Digital and DTS, have given a new impetus to surround sound. The concept of the home cinema has apparently captured the consumer imagination, leading to widespread installation of surround sound equipment in domestic environments.'* [12]

Other accounts of the increasing interest in surround formats are apparent, Malham, of the University of York appearing to agree, stating the following:

> *'In the dying years of the twentieth century, after more than a hundred years of recorded sound and half a century in which the use of two channel stereo has been widely regarded as synonymous with the high fidelity reproduction of recorded music, multichannel surround sound has finally begun to make real inroads into the audio market.'* [13]

It is conceivable that this interest in consumer surround technologies will continue and future systems will become more sophisticated, perhaps introducing newer surround formats.

Dramatic increases in the computing power of the average home computer and the introduction of high quality and cost effective multi-channel audio I/O cards have enabled the average composer and performer access to sound technologies that were previously confined to the large production studio. This progressive lowering of sound processing latency[14] and increase of sound processing power has brought the possibility of real-time spatial signal processing technology into the forefront of even low budget studio technology. In essence, the sound artist is presented with an audience that has an increasing interest in spatial sound design and composition. Perhaps then, composers and performers should be provided with spatial sound design tools to explore and produce works for the consumer surround sound of today and the spatialization formats of tomorrow.

With this in mind this research sets out to explore techniques that can help to broaden the range of spatial sound design software available to the artist. While

---

[12] ibid., p.17

[13] Malham, D. G: 2000

[14] In audio host based processing, latency refers to the processing time delay between input and output.

researching and expanding spatial techniques the further aim is to develop prototype spatial sound design and performance software.

1.3 Fundamental categories of sound spatialization

Typical techniques for spatialization of sound could be categorised into two very general methodologies:

- Direct spatial positioning of sources, encoding for a known playback configuration, i.e. modelling of source spatialization with some degree of accuracy.
- Projection of pre-composed multi-channel source material into a performance space, i.e. controlling or modifying spatialization appropriately for an audience within a given space.

Positioning audio images within a known speaker configuration such as stereo or 5.1[15] is often achieved using mathematical panning laws. Each panning law governs the appropriate balance of source audio within the given setup, use of stereo panning laws in mixing hardware being the most common spatialization usage. In a similar fashion, the use of ambisonic techniques allows direct positioning and encoding of source material for decoding and playback through any given speaker configuration. Essentially both of these techniques provide usable and often realistic spatially encoded source material. However, both rely heavily on accurate loudspeaker design and placement together with ideal listening conditions, i.e. the listener is positioned at the 'sweet spot' within the configuration and the listening space has sufficient acoustic treatment to limit colouration of the source image.

> *'Even on a good hi-fi system, with the listener in the 'sweet spot', the stability of the stereo image is notoriously fickle – turning or inclining*

[15] 5.1: An increasingly common home and cinema surround sound format consisting of 5 full frequency range speakers and a single sub bass unit.

> *the head, or moving to left or right by just a few inches, can cause all kinds of involuntary shifts in the stereo image.*'[16]

Inherently, the conditions for the ideal listening environment can be difficult to obtain for the large audience in a large space. In this case the only realistic solution capable of retaining an accurate image for the individual is to provide independent headphones for the whole audience. Clearly, this practice would be difficult to achieve logistically and the associated audience discomfort would be undesirable. A multi-channel headphone solution, although possible, suffers from inflexibility in configuration. For example a single headphone driver design may be incapable of playing a stereo piece and an eight channel piece without some elaborate system of reconfiguration, remapping of spatial content being unacceptable in this context.

Contrary to the above concept of pre-defining exact spatial imaging, sound projection or diffusion is often used to present an audience with a vision of the sound that is complemented by the acoustics of the performance space and is tailored for the audience size and location. Here, the idea is to project a composition, commonly stereo or eight channel spatially encoded, through a large number of loudspeakers[17] positioned strategically within the performance space. As Harrison describes, '*...the number and positioning of loudspeakers is primarily a function of the concert space*'[18].

In a typical diffusion concert the 'projectionist'[19], often the composer of the work, strives to present the audio such that pre-composed spatial encoding makes ideal use of the space. It is often argued that this sound diffusion practice is flawed and can be detrimental to a composed piece. Wyatt observes a number of composers with this opinion[20]. Obviously, inherent multi-loudspeaker phase

---

[16] Harrison, J: 1999

[17] Twenty or more loudspeakers being typical, Harrison informs the author that he has used more than eighty in some BEAST concerts.

[18] Harrison, J: 1999

[19] 'Projectionist' sometimes referred to as the ' Diffuser'

[20] Wyatt, S. A: 1999

cancellations and complex performance venue acoustics are clear evidence that a piece will not reach an audience exactly as it did in the studio environment. However, in the absence of a perfect solution, sound diffusion systems provide a method of creatively presenting a work that is acceptable, in fact desirable to many composers. The final decision of whether or not to employ diffusion resides with the composer and therefore it is a creative choice, not a scientific necessity. As Wishart states:

> *'So scientists be forewarned! We may embark on signal processing procedures which will appear bizarre to the scientifically sophisticated.... ...The question we must ask as musicians however is not, are these procedures scientifically valid or even predictable, but rather, do they produce aesthetically useful results on at least some types of sound materials.'*[21]

Later, this thesis presents two software concepts that are designed with the above categorisations in mind. The first software system focuses on diffusion of spatially encoded material. The second is aimed at the design and reproduction of specifically positioned sound sources. However, it should be noted that although the tools have been categorised and their design was focused on particular usage, their use in practice is entirely down to the user. For example Harrison commented[22] that he would find the M2 system suitable for composition as well as live diffusion.

1.4 Existing technology

Many tools already exist for the purposes of spatializing sound. The intention of this research is to produce a number of new tools, not to replace old ones that have already proved their worth. It is hoped that by expanding the basic toolset for the sound artist, the possibilities for creative exploration can be increased. The

---

[21] Wishart, T: 1994, p.5

[22] Harrison, J: verbal discussion during M2 development meeting, University of Sheffield Sound Studios

following sections describe a brief cross section of some tools available, but this is by no means a full discussion of all existing tools.

1.4.1 Plan view spatial panning

Spatial panning is often presented in the form of plan view visualization of the surround format. Simple two dimensional control hardware, often joysticks or direct mouse control, is used to control the source positions within the view. Often this X/Y 2D position forms only part of the panning algorithm with additional controls specifying other algorithm factors.

This plan view basis is the approach taken by the stock surround plugins of Steinberg's 'Nuendo'. Although it is possible within Nuendo to present much more complex and perhaps intuitive interfaces, this approach has the advantage of simplicity in use. Perhaps the most significant disadvantage is the inability to easily express vertical movement from a simple plan view.

The following screenshots show Steinberg's Nuendo version 2 software making use of 2D plan view panning.

The above image shows a single monophonic source panned to the centre of a 7.1 cinema output setup. Blue bars extending outwards show the signal gain for each

output speaker. Other panning methods make different use of the 2D plan view concept. The following display pans audio using a distance based algorithm[23]; circles show amplitude drop in dB for each radial distance:

The image below shows the simple plan view pan controls on 4 monophonic audio channels.

In these examples it should be noted that each source is displayed on a separate plan view. It may make sense in some situations to provide a plan view interface that displays all sources in one viewing area. This way the relative positions of sources can be more easily visualized.

Spain and Polfreman present 'Interpolator',[24] a very hybridized plan view system using a 'light model' for controlling the interpolation of parameters. It would seem feasible that this model could be used for the spatial positioning of sounds.

---

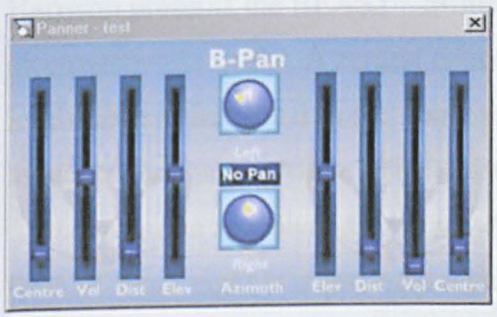
[23] Based on the inverse square law $1/distance^2$

[24] Spain, M; Polfreman, R: 2001

1.4.2 Ambisonics

Ambisonics is an extensively researched subset of general sound spatialization and centres on the concept of mathematically encoding soundfield information such that it can be decoded for almost any playback configuration. The transitional 'encoded' format is known as 'B Format'. A number of methods are available for encoding spatial information including software tools for directly setting spatial characteristics of source material and direct soundfield microphones that can capture the spatial characteristics live.
Malham describes a mathematical method in which single sound sources can be Ambisonically encoded into 'B format' and decoded for arbitrary speaker arrays[25]. He also describes the difficulties of encoding distance information from the basic algorithm. In order to encode distance cues, Malham describes the use of extra digital signal processing techniques.

Mooney[26] presents 'AmbiPan' and 'AmbiDec', VST plugins for direct encoding and decoding into and out of B Format. Malham and Field present a similar system of plugins for Ambisonic special encoding and decoding within host software[27]. The GUI version of 'B-Pan' is shown below:



The following photograph shows the capsule layout of Soundfield's 'B Format' capable microphone the ST250. This microphone, together with its encoding hardware can be used to record the soundfield directly to B Format.

---

[25] Malham, D. G: 1998

[26] Mooney, J: 2000

[27] B-pan and B-dec software hosted at http://www.dmalham.freeserve.co.uk/vst_ambisonics.html

Perhaps the most significant advantage of the Ambisonic system is its ability to decode the same B Format encoded material into different output formats. Effectively only the four B Format channels require storage, an immediate saving on data bandwidth when considering large numbers of output speakers involved in typical multichannel playback systems such as 5.1.

1.4.3 BEAST – DACS 3D

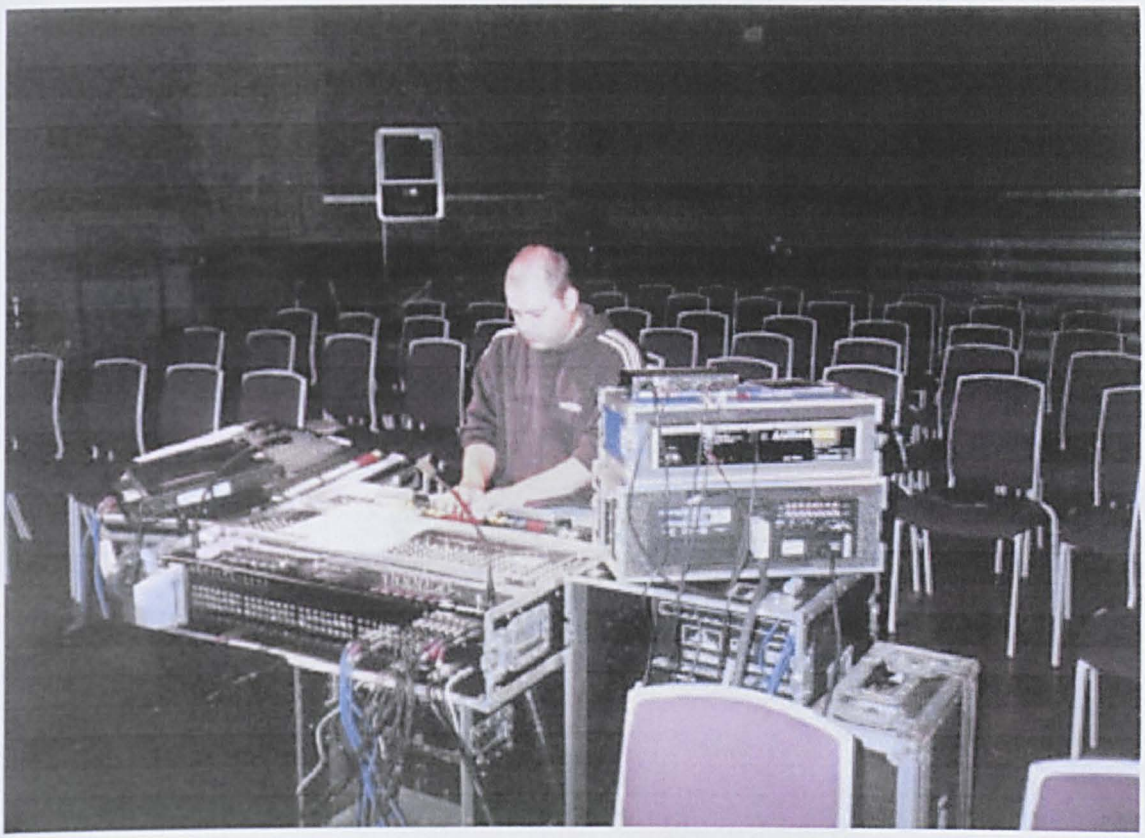
Harrison presents the BEAST[28] diffusion system for the projection of electroacoustic works. A central concept in the BEAST system is the set of loudspeakers described as the *'main eight'*, consisting of four matched pairs of loudspeakers named Main, Wide, Distant and Rear. Harrison regards this set as *'...the absolute minimum for the playback of stereo tapes.'*[29] . In concert, the BEAST main eight is augmented with other matched pairs located to best match the venue; the exact details of these placements are described fully in Harrison's paper on the subject.

In order to disperse audio into the multitude of output loudspeakers the BEAST system uses the DACS 3D console featuring a stereo switch matrix for 24 inputs into 32 outputs. This system allows any input odd/even pair to be routed to any

[28] BEAST - Birmingham ElectroAcoustic Sound Theatre

[29] Harrison, J: 1999

output odd/even pair, the final individual output level being under direct control of the projectionist via a high quality fader.



DACS 3D (Visible in the above photograph) is a purely analogue design and has the advantage of being highly robust whilst perhaps lacking in some of the benefits of digital re-patching or semi-automated control.

1.4.4 Gmebaphone and Cybernéphone

The 'Gmebaphone', later known as the 'Cybernéphone', presents a flexible digital control surface for live diffusion. As with the BEAST system, works are pre-spatialized and projected into the venue at performance time. In the Cybernéphone the control hardware and complex layout of varied loudspeaker designs is treated as an instrument in itself. Loudspeakers have been designed and selected to operate only within limited frequency ranges resulting in varied

colouration in the output. Baudelaire treats the Cybernéphone as an instrument in itself;

> *'...a huge acoustic synthesizer, an interpretation instrument that the composer plays in concert, an instrument that serves to express his composition, to enhance its structure for the benefit of the audience, to bring it to sonic concretization.'*[30]

Loudspeakers are grouped in sets or *'registers'* with audio routed and controlled live via the digital console to present the audience with contrasting perspectives. Below, the photograph shows the 1997 version of the Cybernéphone console.

1.5 Summary

Two general categories of sound spatialization would seem to be in common use; simulation or recording of realistic spatial audio acoustics and projection of pre-spatialized audio into a performance space. It is perhaps interesting to note, that in his paper outlining different spatialization methods, Malham does not reach a conclusion as to an optimum. However, he does conclude the following:

> *'...the optimum system for composition purposes must remain always a decision of the composer, to be made on musical, not technical, grounds.'*[31]

---

[30] Baudelaire: 1997, p.268

[31] Malham, D. G: 1998

In all of the above spatialization methods there is a clear need for appropriate control. With accurate source spatialization systems, (panning and Ambisonics) interfaces that allow good visualization of the simulated 3D space would seem a logical choice. In live sound projection or diffusion, interfaces must allow flexible routing of source sound to outputs.

This research will concentrate on two areas of interest to the sound artist, physical modelling of real and surreal sound environments and tools for live spatial performance.

Beginning with some simple theoretical models of sound spatialization and progressively building them into more generic models, this thesis will attempt to produce flexible theoretical systems for spatial sound design and performance. With these theoretical systems to hand the project will propose and implement real time software tools that will enable the sound artist to experiment and perform. As increasingly complex models are produced the thesis will cover theoretical ground necessary for practical implementation of each software project.

2 Live Diffusion Mixing and Control

Live sound diffusion of composition or performance is essential for many artists and provides an enhanced audience experience. It is theoretically possible to diffuse any number of sound sources into any number of loudspeakers, although in practice typical concerts diffuse stereo recordings into perhaps twenty loudspeakers. The configuration of the speakers is determined by the piece and the performance space. Essentially, the speakers are placed to complement the space or exploit interesting features in it. The pure logistics of setting up a performance space often constrain the diffusion system that is built[32]. As a direct result of this, the performer or 'diffuser' often has to work with different configurations of speakers in each performance space. In collaborative performances of many different composers' works it is often necessary to reach a compromise in the final diffusion setup.

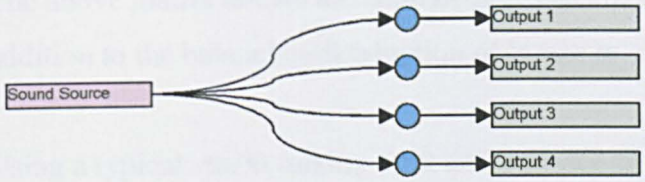
The goal of this project is to provide a method for diffusion mixing that provides advantages in logistical setup and an enhanced control method for the performer. In essence, an attempt is made to step from basic diffusion mixing into a digital solution with potential for future exploration.

2.1 Diffusion mixing through the Mix Matrix

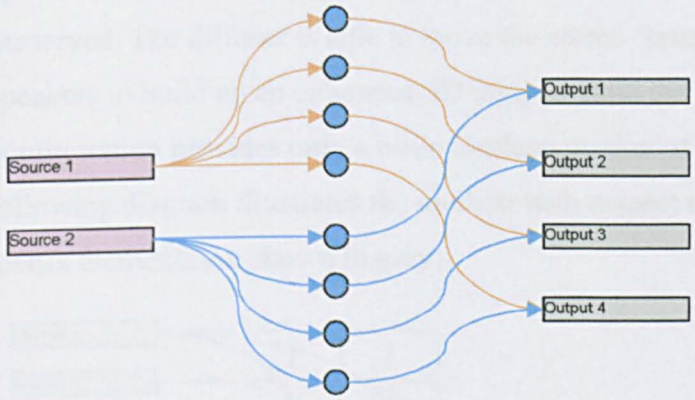
Diffusion of sound from a single source into multiple speakers is commonly achieved through the mix matrix. In a mix matrix[33] a monophonic source is essentially split into a number of mono outputs either by use of switching or by attenuator control. The following diagram illustrates this concept with a four channel output system: (Circles represent switches or attenuators.

[32] Wyatt, S. A: 1999

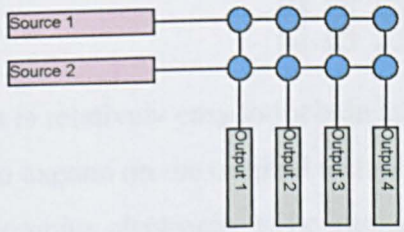
[33] Davis, G; Jones, R: 1990, pp.162-163



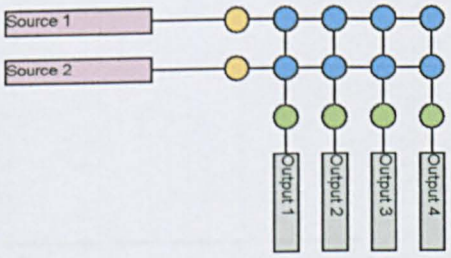
When expanded to include a second input the signal flow becomes more complicated and it is useful to represent it in the form of a grid matrix. The following diagram represents both the first representation expanded for two inputs and the grid representation for the same signal flow:



In matrix form:

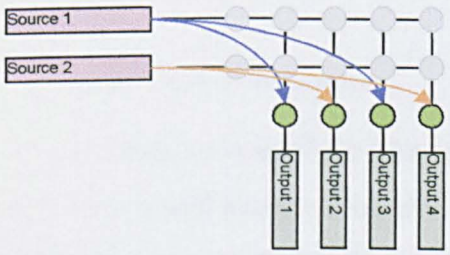


It is clear here that complex input/output systems are more easily represented in this manner. If the matrix is realised with attenuators rather than switches it is possible to exactly adjust the 'quantity' of each input sound going to each output loudspeaker. Adding further attenuators to the above matrix provides additional control of input level and individual output level:



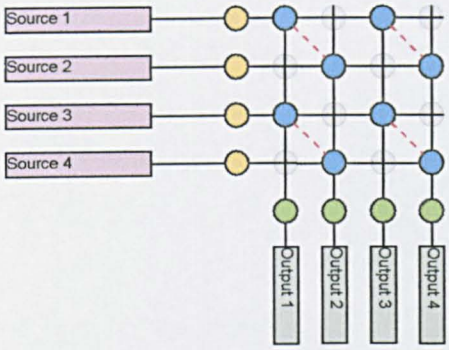
The above matrix allows the additional global control of inputs and outputs in addition to the balancing/distribution of inputs to outputs.

Using a typical studio mixing desk and a device to split the input signal it is possible to realise some of the above mix matrix for live performance. As input a stereo source can be split into odd/even (L/R) pairs of mixer channels and the direct recording outputs can be used to control the output levels of each individual speaker. This configuration would typically be complemented with pairs of speakers assigned such that the stereo left/right spatialization on the recording is preserved. The diffuser is able to move the stereo 'image' between the pairs of speakers to build up an enhanced 3D image within the performance space. This configuration provides only a basic implementation of the full mix matrix. The following diagram illustrates the method with respect to the full matrix. Unused matrix elements are shown in grey:



It is relatively easy to include inline global input attenuators into the splitter box to expand on the original technique but the full power of the matrix requires more complex electronics to be implemented within the mixer and a custom built mixer is necessary. The DACS 3D[34] matrix desk expands on the basic studio technology diffusion desk by providing a switch matrix that allows matrix style routing of input pairs to output pairs. A basic flow diagram of key features is provided here:

[34] See also 1.4.3



Pairs of channels are routable into output pairs via paired switches and this allows the diffuser multiple sources to different speakers all with output control.

It is feasible that an analogue mixing console could be produced to provide the full mix matrix but the ability to control the desk effectively requires some further thought. Consider a similar matrix to the above with 4 inputs and 4 outputs. This would require twenty four attenuators. Attenuator totals are equated as follows:

$$att_{total} = ins * outs + ins + outs$$

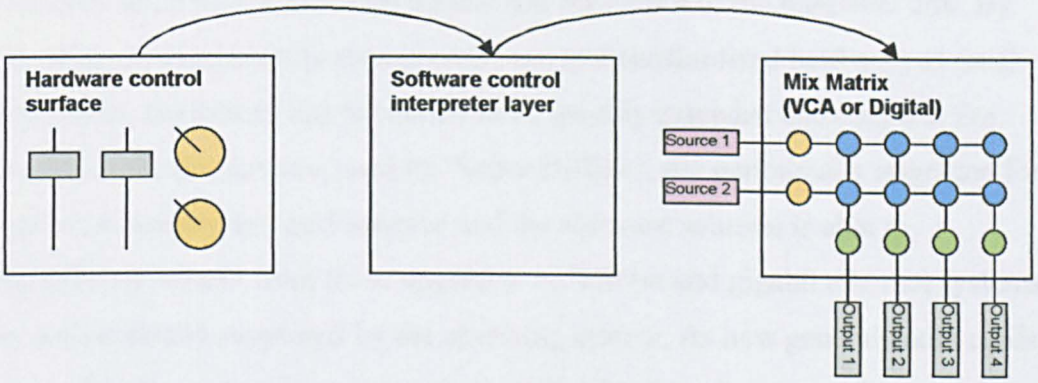
To have hands on control for 24 attenuators is feasible but a more typical 2 in / 24 out system would require controlling 74 attenuators during a performance. A greatly expanded system would require many more attenuators and the feasibility of controlling the matrix mix during a performance becomes more difficult. In addition to control complexity, it is also more difficult to expand an existing system for more inputs and outputs because the matrix requires interconnection of every input to every output. Copeland, Rolfe and Truax specifically highlight the importance of computer control when dealing with large numbers of matrix parameters[35].

Expansion and simplification of full matrix control with additional modularity of I/O requires consideration of digital techniques.

Two methods are considered; control of analogue attenuators via VCA's and full digital implementation of the mix matrix in hardware or software. It is feasible

[35] Copeland, D: 2000; Rolfe, C: 1999; Truax, B: 1997

that an analogue matrix mixer be produced that has full VCA automation of all parameters. A control surface could then be produced to provide hybridized control methods for the mix parameters. Single faders and pots would control multiple mix parameters via some form of mapping software. A similar approach would be taken in a fully digital mixing solution in which mixing is performed by digital attenuators with an external digital control surface. Essentially both of these methods split the system into two major components, matrix mixer and controller. In both cases the expansion of hardware is more difficult than the expansion of software. However, in an all digital system I/O expansion can be achieved simply by adding more ADC/DAC devices and reconfiguring software, whereas the VCA solution requires extensive alterations to matrix and VCA hardware. Hardware control surface expansion would be difficult in both cases but the software layer can be adjusted to cater for new mix parameters without adding new controls. The following diagram illustrates the major concepts for both methods:



The physical layout of the system could be built into a single box but it would be advantageous to separate mix matrix and control systems into separate units for a number of reasons: one control system could control many types of mix matrix; one matrix can be controlled by many different controller systems. Providing a standard control protocol allows many different controller types to be developed independently of the mix matrix. In a performance setting, the ability to control the same matrix using different control devices allows different performers to have the right control device for their particular piece. For the performance space, the ability to have a tailored mix matrix without concern for final control method is beneficial for both cost and setup ease. An added benefit of this modularity is the ability of one control surface to be mapped to multiple mix matrices.

One example of this remote mix matrix is produced by Richmond Sound Design Ltd. The AudioboxAB64[36] is a 64*64 digital mix matrix produced using dedicated DSP chips. Versions of the Audiobox have been used for concert diffusion by 'Sound Travels'[37]. The system is controlled externally by software and control communication is via MIDI. This system provides a good basis for diffusion mixing but its hardware solution is limited in modularity and updateability. For example, 'Super Diffuse', described later, has the ability to work with audio cards from different manufacturers providing a choice in AD/DA converters and I/O channel specifications. A user is thus able to choose the type of connections (Digital or Analogue I/O, Microphone pre-amplified etc.) appropriate to the task at hand while using the same software system. It is also a relatively simple process to upgrade the 'Super Diffuse' software when compared to DSP hardware solutions such as the Richmond design. A simple software installation program can install a new version on the user's system. In a DSP based system it is necessary to provide a firmware update and transmit it to the hardware unit. By providing all functionality through software and standardized hardware as much as possible, the system has the ability to be quickly extended and adapted. For example, ethernet devices, used by 'Super Diffuse', are periodically upgraded for improved stability and performance and the software solution is able to immediately benefit from these upgrades, i.e. kilobit and gigabit ethernet systems are automatically supported by the operating system. As new general audio cards are produced, a software system can immediately take advantage of improvements in AD/DA conversion and higher numbers of channels.

2.1.1 Digital expansion of the basic mix matrix

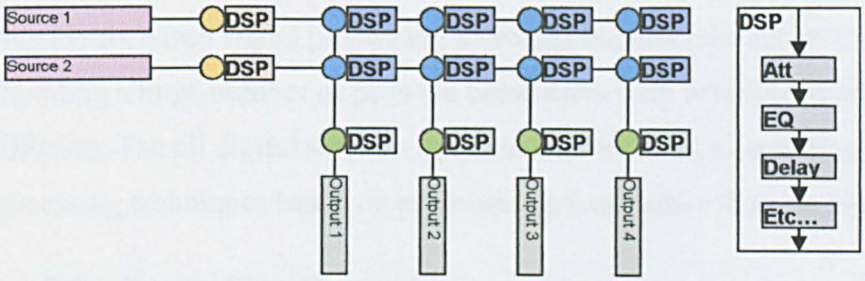
The use of a digital mix matrix allows integration of many digital processing techniques into the basic mix matrix software. As an example the system might consider DSP processing input and output matrix stages. Consider a phase, EQ

---

[36] http://www.richmondsounddesign.com/ab64specs.html

[37] Copeland, D: 2000

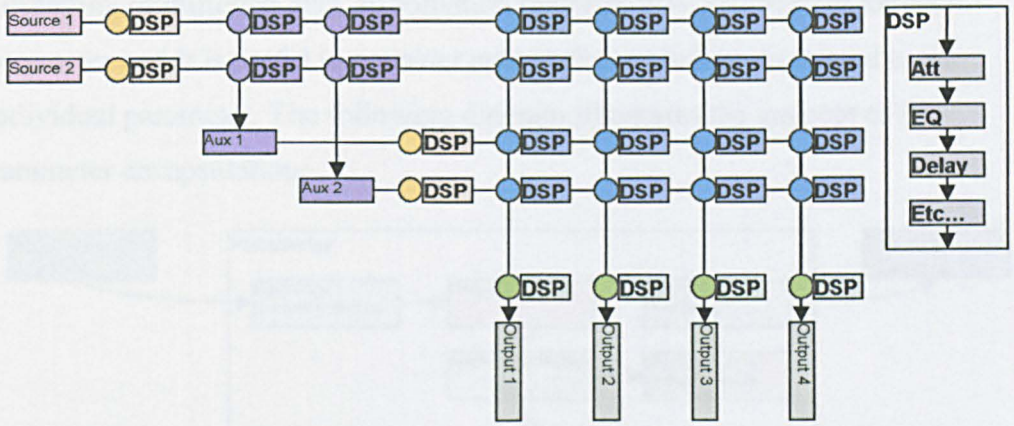
and variable delay stage placed inline with outputs. Frequency response and phase of individual speakers could be tailored for the purposes of correcting room response. The delay stage allows perceived speaker distance to be adjusted for each output. Accurate calculation of delay times would allow correction of speaker phase for differing distances. Extreme use of varying delay times would produce the effect of Doppler shifting. These DSP techniques can be expanded into other areas of the mix matrix and also to include other effect techniques such as reverberation or delays with feedback. In essence the mix matrix becomes a DSP matrix and this is illustrated below:



DSP is inserted into different areas for different requirements: source DSP gives input source effect (processing of global sound input); output DSP provides output effect (processing of global sound output); and individual matrix DSP gives individual input to output effect (processing of individual matrix I/O connections).

2.1.2 Matrix Auxiliaries

It is possible to add channels that act in a similar way to auxiliary channels on a standard mixing desk. These auxiliary channels use a mix matrix to receive signal from sources and then are able to inject the mix back into the main DSP matrix. The following diagram illustrates these aux input channels and the signal flow diagram for them.



This matrix based signal processing increases the potential for creative effects by providing a huge number of possible parameters with which to perform sound diffusion. The all digital software implementation allows expansion of signal processing techniques based on processing power rather than hardware.

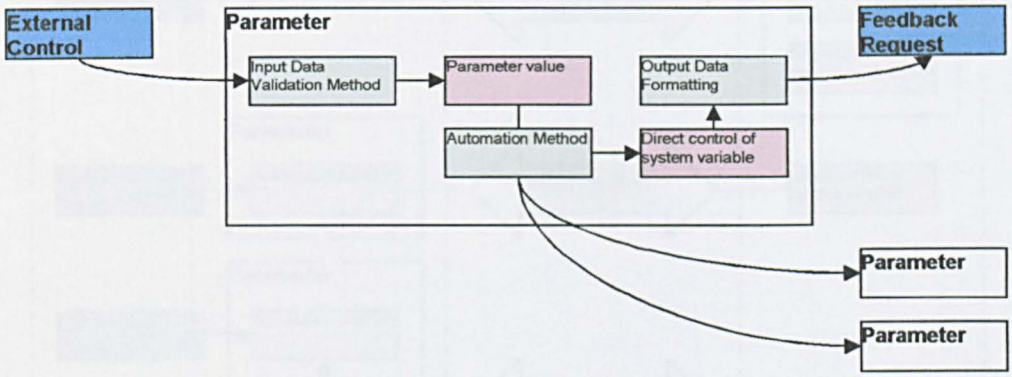
2.2 Controlling the matrix

The control of matrix parameters in any live tool is of paramount importance. It is through control of the diffusion system that the vision of the performer is realized. The ideal performance controller hardware is of course an area of extensive research by both artists and manufacturers. A discussion of the many controllers and interfaces produced for manufacture or in research projects is not a major concern of this thesis. However, as DSP variables will always be the target of any external control, the concept of the parameter is used here to represent a constrained and automated variable of the DSP system that can be manipulated from any external device. This hybrid variable can be defined in such a way as to provide expandability and a layered, tree like control structure.

2.2.1 Parameters (as hybrid variables)

Consider a simple variable representing attenuation amount; it might be defined so that it should have a maximum value of 127 and a minimum value of 0; using an integer representation the variable would have 128 steps. The act of setting and retrieving the variable would require validation of user input and perhaps

formatting of retrieved data. Automation methods may be different for each parameter and it is useful to consider automation technique as related to an individual parameter. The following diagram illustrates the concept of the parameter encapsulation:

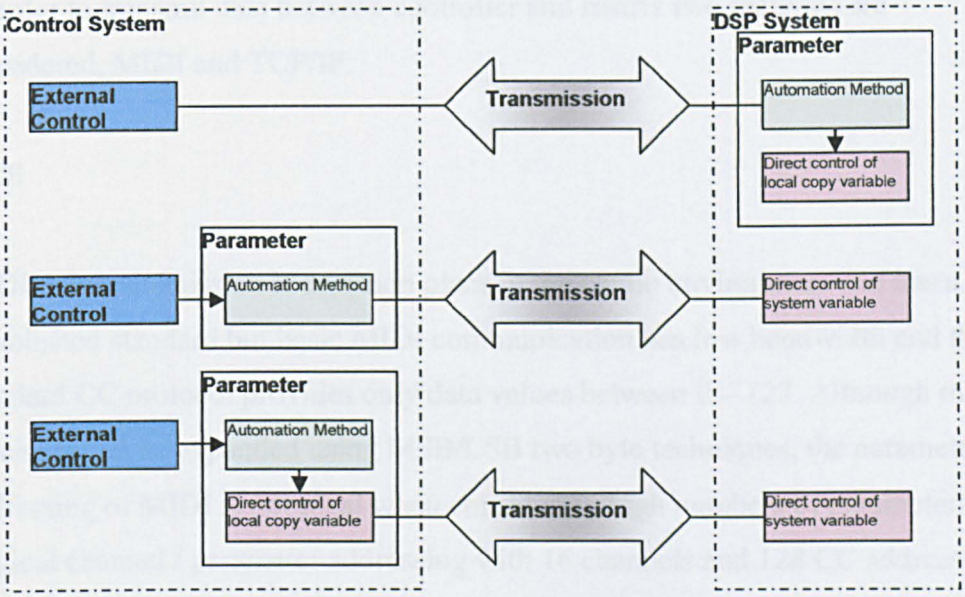


The automation method in the above diagram forms the basis of the parameter tree structure and acts as a placeholder for automation methods. The value contained in the parameter controls the amount of automation to be applied to both the raw DSP variable and any target parameters for automation. In addition, external control is applied via a validation method and feedback of parameter value is given through an output data formatting method. Hierarchical parameter control is provided by the automation algorithm being able to control other system parameters which may in turn control further parameters.

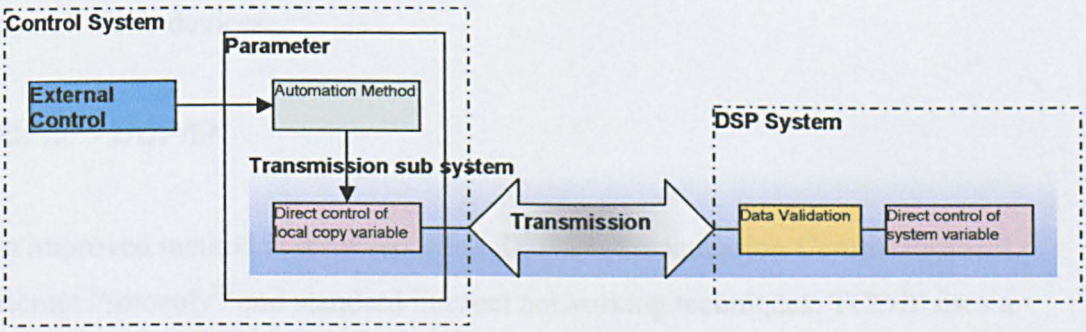
In the simplest of automation algorithms the parameter would provide a direct mapping from internal value to direct system variable, effectively wrapping the system variable in a shell. It is possible to have independent data formats for parameter value and system variable and even multiple system variables controlled from one parameter value.

As discussed above, the system variables may reside in a different unit to the control system. It is necessary to allow the parameter construct to access the system variables in some way. Three methods can be considered: using parameters to encapsulate the actual DSP matrix variables and perform automation within the DSP system; directly transmitting data changes from control system parameters to DSP system variables; maintaining a copy of data

structures in both parts of the system encapsulating control variables with parameters. The following diagram illustrates these three methods:



The first two methods are similar, the only significant differences being location of processing. In the first, parameter processing occurs alongside the DSP calculations and in the second, parameter processing is the responsibility of the control system. At first glance the third method would appear to be inefficient due to redundant data, but the advantage lies in the possibility of varying control and transmission rates. The extra layer of redundancy allows data to be written directly to the local variables without concern for transmission. Copying of the data can then be performed by a separate part of the system. The following diagram shows an extension of this third method to provide a more robust system:



Transmission is handled by a transmission sub system and automation can be continued without concern for the speed of data transmission.

2.2.2 *Data transmission for matrix control*

In order to transmit data between controller and matrix two methods are considered, MIDI and TCP/IP.

MIDI

MIDI data transmission is common place in the studio environment and has a well established standard but basic MIDI communication has low bandwidth and the standard CC protocol provides only data values between 0 – 127. Although this precision can be expanded using MSB/LSB two byte techniques, the parameter addressing of MIDI is not ideal when considering high numbers of parameters. Logical channel / parameter addressing with 16 channels and 128 CC addresses gives 2048 parameter addresses but this is halved in an MSB/LSB solution due to the requirement of two addresses per parameter. A single 32 * 32 mix matrix would require 1088 parameters and this exceeds the number available with MSB/LSB on CC messages. Obviously there are other message types that can be used along with the CC messages but the protocol quickly becomes illogical. MIDI channel cannot be used to represent input or output channel so controllers must be assigned arbitrary CC addresses. In solution to this, a generic messaging protocol utilizing MIDI System Exclusive is possible, but the extra data overhead could start to impede transmission performance and the specific SYSEX protocol takes away some of the advantages of generic MIDI communication between general music devices.

TCP/IP - UDP/IP

An improved method is achieved using TCP/IP (Transmission Control Protocol / Internet Protocol)[38] and standard internet networking techniques. TCP/IP uses a packet based system for data delivery. A given block of data is split into appropriately sized packets containing destination address, ordering and validation checksum information before being transmitted. A destination address

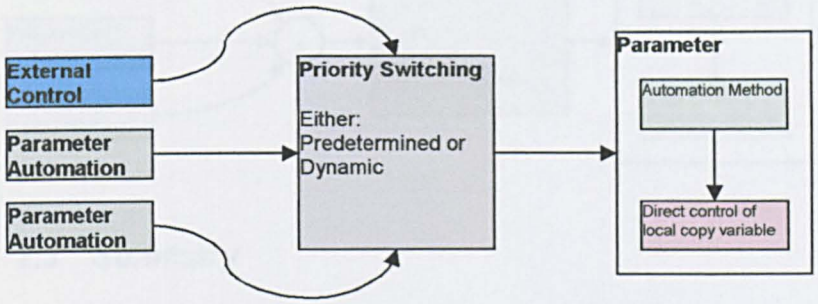
[38] Jones, A; Ohlund, J: 1999, p.136

(IP Address) is given using a code in the form (a.b.c.d) where a,b,c and d are single byte unsigned integers with the range 0-255 e.g. (192.168.3.2). TCP allows posting of data to an address and is thus called a 'transport layer' for communication. TCP/IP provides reliable data transmission though a system of timeouts and delivery replies; a system sending data via the protocol waits for a reply from the target system. If no reply is received within a timeout period the delivery is assumed to have failed and is re-sent. On receiving, the TCP layer attempts to reconstruct data from received packets and will send a request for re-transmission of a packet if corruption has occurred. This implementation of transport reliability puts additional strain on systems so a second protocol can be used if high speed is of greater importance than reliability. UDP/IP (User Datagram Protocol / Internet Protocol) provides unreliable data transmission with less overhead and is most suitable for transmissions of streamed data, i.e. data that will become out of date before it can be re-transmitted. These two transport layer protocols can coexist on the same IP network. It is possible for a single system to transmit or receive data from both protocols simultaneously.

Advantages here are the increased data bandwidth and the advanced message routing, even high bandwidth wireless communication being increasingly commonplace. At the time of writing Digidesign Pro-Control hardware systems use a form of IP communication for transmission of data and Pro-Control technology is well established in both educational and professional studios. The increased use of general network technology means that TCP/UDP is well tested and advances are more frequent than in ageing MIDI hardware systems. This more general use also brings down the market cost of related technology and hardware. Data transmission of huge numbers of parameters is easily possible due to the higher bandwidth and the completely configurable data format allows very logical addressing of the required parameters. For example, it would be very simple to specify 8 bit integers for channel and parameter address fields giving 2^{16} possible addresses (65536). Data for each parameter address could be of arbitrary length allowing any precision or format needed.

2.2.3 Multiple control prioritisation and summing

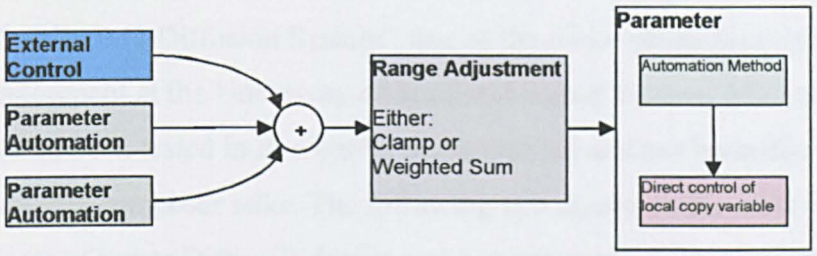
It is possible to have one parameter adjusted from more than one external controller or automated parameter. In order to allow this multiple source control it is necessary to provide some method for combining or selecting source data. Two methods are considered, priority selection of source data and summation of source data.



Source data can be prioritised with predetermined hierarchy or can be dynamically calculated depending on the state of source data and target value. Consider two faders used to control a single parameter: In a predetermined priority system, fader 1 could be given priority over fader 2. If control adjustments came from fader 2 then the system could decide to only allow control if fader 1 was inactive. If instead of predetermining the priority it is calculated constantly from a predefined algorithm then multiple controllers could perhaps be used more fluidly. In this case either fader could be given priority depending on the current state of target variables. Many algorithms could be used in this situation such as use of data from the last adjusted controller; use of data from the most stable controller; use of data from the controller with the closest setting to the target parameter etc. A problem with these prioritised controllers is that low priority data is discarded.

In order to address the problem of discarded data, a method of controller summation could be used. Summing the values received from each data source results in data that is responsive to changes in all sources. However, a problem occurs due to the increased range of the resulting data, i.e. data range [0,127] + [0,127] gives a range [0,254]. This resultant range may not be compatible with the target parameter. There are two obvious strategies: either clamp the data to the correct range, or take the weighted sum of the inputs to obtain the correct range.

In clamping the data range, any data above or below the required range will be discarded. Using a weighted sum or average, the correct range is created from scaled input sources.



2.3 Summary

Having described concepts for a theoretical sound diffusion mixing system this thesis moves on to the implementation techniques used in a real world sound diffusion software tool.

3 'Super Diffuse' Digital Sound Diffusion

'Super Diffuse' was originally conceived to test some of the concepts described in the previous chapter. The 'Super Diffuse' software and related control hardware[39] forms the 'M2 Diffusion System', one of the major projects currently in development at the University of Sheffield Sound Studios. M2 has, at the time of writing, been tested in five public performances and has been discussed in three developer/composer talks. The following text describes the most important aspects of Super Diffuse's design and implementation. This section assumes prior knowledge of software engineering concepts.

3.1 Development Tools

C++ was chosen as a development language for a number of reasons: high speed compiled applications, very low level access to RAM/hardware and support for Object Oriented development. C/C++ is at the time of writing the most widely used language for real-time audio software development.

As with Ricochet, development was achieved with Microsoft's Visual C++ IDE but the overall implementation was intended for stability so modular components were left out of this initial design iteration.

Steinberg's ASIO[40] (Audio Streaming Input Output) is at the time of writing the most widely used Audio I/O technology for professional music software and was chosen over other technologies for its very low latency and wide compatibility. A more detailed report on Steinberg ASIO and the 'ASIO Sub System' dynamic link library developed to access it appears in 6.1.2, 'Steinberg's ASIO for host based audio I/O'. With regard to the design of DSP frameworks the ASIO, VST and

---

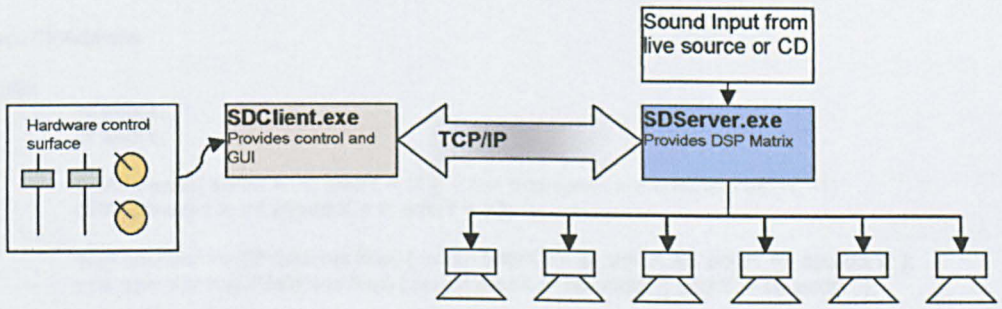
[39] Control hardware designed by Mooney, University of Sheffield.

[40] Steinberg Soft- und Hardware GmbH: 1999

MAX/MSP Externals development documents are valuable references and examples[41].

3.2 Client / Server, Common functionality

Super Diffuse is designed with a client / server architecture in order to provide abstraction between signal processing and control. 'SDServer' uses the ASIOSubSystem.dll developed for this thesis to provide a DSP Matrix. TCP/IP communication is used for transmission of data from the 'SDClient' software. 'SDClient' provides the graphical interface for controller mapping and connection to external control hardware via MIDI. Automation is achieved within the 'SDClient' parameter system. The use of TCP/IP communication allows the two pieces of software to operate on independent computer systems if the need arises. The following diagram illustrates the general software architecture:



The communication between the two pieces of software requires some commonality in design.

3.2.1 Parameter Mapping

Parameter addressing between client and server uses a common base addressing for the DSP parameters. The client software uses an extended address map for the

[41] Steinberg Soft- und Hardware GmbH: 1999; Zicarelli, D: 1998;

purposes of automation effects and parameter grouping. The basic map is defined in the 'parametersystem' header and source files. An enumerated type PARAMETER_ADDRESS_ROW defines the basic address types as follows:

```
enum PARAMETER_ADDRESS_ROW
{
        PAR_INPUT_GAIN = 128,
        PAR_INPUT_MUTE,
        PAR_INPUT_ROUTE,

        PAR_OUTPUT_GAIN,
        PAR_OUTPUT_MUTE,
        PAR_OUTPUT_ROUTE,

        PAR_MASTER_SECTION,

        // client side only parameter address rows
        PAR_GROUP = 256,
        PAR_EFFECT, // the effect fader section
        PAR_EFFECT_PARAM// the start of the effects parameters
};
```

During development, mute and routing were included for possible future revision although the current implementation does not yet make use of them.

Stable address handling within the program is performed with the aid of the CPAddress class defined below:

```
class CPAddress
{
public:
        int addrX;
        int addrY;

        CPAddress(){ addrX = -1; addrY = -1;}; // null addresses are specified as (-1, -1)
        CPAddress(int x, int y){addrX = x; addrY = y;};

        bool operator ==(CPAddress &op) { return addrX == op.addrX && addrY == op.addrY; };
        bool operator !=(CPAddress &op) { return addrX != op.addrX || addrY != op.addrY; };

        void Archive(CArchive &ar);
};
```

This class provides storage for the X and Y portions of an address with overloading of boolean operators greatly simplifying use within conditional statements. The X address refers to the PARAMETER_ADDRESS_ROW type described above and the Y address has differing meanings depending upon the first part of the address. The address fields are both described with 32bit integers giving large scope for future development. This structure is directly used by archiving and the same source files being used in both client/server applications. Common use of source files provides improved consistency during development due to automatic propagation of changes to both applications.

3.2.2 TCP/IP implementation, MFC Sockets

TCP/IP communication in Super Diffuse is performed using MFC Sockets, an encapsulation of Winsock2 which in turn derived from the BSD[42] Sockets implementation used in POSIX[43] compatible systems. MFC Sockets provide simple methods for performing non-blocking asynchronous[44] networking and are fully integrated with the MFC GUI functionality that was used in Super Diffuse.

'ServerComms' source and header files detail the common ground implementation, providing 3 major classes 'SDServerSocket', 'SDListenSocket' and 'SDClientSocket' inheriting from the MFC class 'CSocket'. These inheriting classes provide the client server architecture for the system and define the callbacks for interlinking comms and system code. Received messages are pre-translated and validated before being passed to the application supplied callback.

Socket communication begins with a listening socket (SDListenSocket) waiting for incoming client socket (SDClientSocket) connection requests. Upon reception of a client request, the listening socket passes control to an instance of the server socket (SDServerSocket). Bi-directional communication is possible through this single client/server socket connection although server to client transmission is at present used only for initial 'handshake' server information.

'ServerComms' source also defines some generic structures for data transmission of parameter values, server information and a general message transmission header structure. The use of a generic client/server error numbering system further reduces the code redundancy and reduces potential for bugs.

3.2.3 Heap Array Templates

---

[42] BSD - Berkeley Software Distributions

[43] Love, R: 2003, p.54

[44] Jones, A; Ohlund, J: 1999, p.231

DSPSystem below, uses custom designed Array<> and Array2<> template classes to provide fast and safe access to heap allocated one dimensional and two dimensional arrays. Automatic cleanup of heap allocated ram is achieved in class destructors so the overall memory safety is improved when using these classes over direct use of new and delete operators. These classes are defined in array2.h and are also used in other areas of the client/server software. The basic public interface to these classes is detailed below: (note: implementation has been removed here)

```
template < class T >
class Array
{
public:
        Array();
        Array(int _size);
        Array(Array<T> &a);
        ~Array();

        void Destroy();
        void Create(int _size);
        void Create(int _size, T init);
        T& operator [](int index);
        operator T*();
};
```

The above class supports various constructor methods including a copy constructor[45]. Memory allocation is achieved either via parameterized constructor or directly thought Create() and Destroy() functions. C style array access is provided with an overloaded [] operator and pointer based usage is proved with overloaded T*. Array2<> is the 2 dimensional version of the array class with the following interface:

```
template < class T >
class Array2
{
public:

        int Size();
        int SizeX();
        int SizeY();
        Array2() ;
        Array2(int x, int y);
        Array2(Array2<T> &a);

        void Create(int x, int y);
        void Create(int x, int y, T init);
        ~Array2();
        void Destroy();
        T& Index(int x, int y);
        T& IndexNoBounds(int x, int y);
        T* operator[](int x);
};
```

[45] Stroustrup, B: 2000, pp.245-246

Most functionality is as the Array<> class but element addressing is performed with the function Index(x,y). At the time of writing C++ did not appear to provide a method of overloading [][] (2d array subscript operator).

3.3 SDServer specific

One of the most important goals of the SDServer software was to provide a stable DSP engine that could be run with little human intervention on a remote machine. It would be extremely difficult to eliminate all chances of error but simplification in design helps to reduce the chance of human error in development. For this reason many of the more complex DSP matrix algorithms discussed in theoretical sections have not been implemented in the initial version although inclusion is intended in future revisions. A simplified DSP matrix with I/O attenuation forms the basis of the first SDServer version.

3.3.1 Implementing a real time DSP Matrix

SDServer provides the functionality for signal processing using the ASIOSubSystem.dll described later in this thesis. Signal processing is handled by the DSPSystem class defined in SDDSP.h and SDDSP.cpp. DSPSystem::DSP() forms the basis of the algorithm and its implementation is described below: (simplified version)

```
for(o = 0; o < numOuts; o++) // cycle the outputs
{
        memsetf(out[o],0.0f,bufferSize); // clear the output buffer // optimized to float version

        // only process output if gain is != 0
        // perform interpolate for output gains
        INTERPOLATE(outputGain[o],ioutputGain[o]);
        if(outputGain[o] != 0.0f) // optimize for no processing on output gain 0.0
        {
                for(i = 0; i < numIns; i++) // cycle the inputs
                {
                        INTERPOLATE(inputGain[i],iinputGain[i]);
                        if(inputGain[i] != 0.0) // optimize for no processing on input gain 0.0
                        {
                                INTERPOLATE(matrix.IndexNoBounds(i,o),
imatrix.IndexNoBounds(i,o));
                                float mval = matrix.Index(i,o);
                                if(mval != 0.0f) // optimize for no processing on matrix 0.0
                                {
                                        // now do calculation
                                        gain = mval * inputGain[i] * outputGain[o]; // gain calculation
                                        DSPSumToBuss(in[i],out[o],gain,bufferSize); // matrix
summing
                                }
```

```
                    }
                } // end input cycle
            }
}// end output cycle
```

Essentially the algorithm is simply a nested 'for' loop providing iteration over the I/O matrix. Three conditional statements optimize the (N^2) matrix iteration by discarding DSP calculations when gain factors are zero. Maximum performance increases occur when output gains are zero. INTERPOLATE inline function provides a fast and simple interpolation on gain factors in order to reduce zipper noise. The algorithm is described below:

```
#define IP_FACTOR 0.95f // log interpolation factor
inline void INTERPOLATE(float &param, float &target)
{
        param = (IP_FACTOR * param) + ((1.0f - IP_FACTOR) * target);
}
```

Note that the interpolation factor constant IP_FACTOR is adjustable to produce a satisfactory smoothing curve.

DSPSumToBuss() provides an optimized buffer copy operation that provides optimization when gain factors are one or zero. Copying is performed on 32bit blocks rather than 8bit for improved performance on 32bit processors. When gain factor is 1.0 or -1.0 it is possible to remove a redundant per sample multiplication. 'memsetf()' provides the functionality of the C 'memset()'[46] function but it is optimized for 32 bit floating point data buffers, copying whole 32 bit floats rather than 4 bytes.

Gain factors within DSPSystem are stored twice, first written into a temporary placeholder that can be adjusted by communications code. During DSP() the interpolation routine performs smoothly interpolated transition from the current gain value to the new gain value stored in variables prefixed 'i'. The algorithm used for interpolation is performed at sample buffer resolution rather than per sample. This has the effect of increasing algorithm speed while linking DSP reaction time with buffer size, with high buffer sizes causing much slower

[46] Schildt, H: 1998, p.725

responsiveness. Socket communication interacts with the gain factors via the DSPSystem::SetParameter() function as follows:

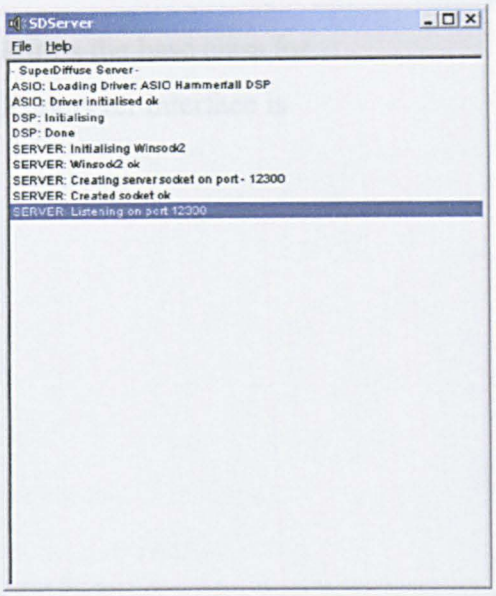
```
void DSPSystem::SetParameter(CPAddress addr,float v)
{
	float value = v  * (1.0f / 128.0f); // conversion giving it a bit of headroom over 1.0f/127.0f

	// value clamping to positive and max gain of 1.0f;
	if(value > 1.0f) value = 1.0f;
	else if(value < 0.0f) value = 0.0f;

	if((addr.addrX >= 0) && (addr.addrX < numIns))
	{
		// this is a matrix parameter
		imatrix.Index(addr.addrX,addr.addrY) = value;
	}
	else
	{
		// something more obvious
		switch(addr.addrX)
		{
		case PAR_INPUT_GAIN:
			iinputGain[addr.addrY] = value;
			break;
		case PAR_OUTPUT_GAIN:
			ioutputGain[addr.addrY] = value;
			break;
		}
	}
}
```

The majority of the above code simply provides mapping from a CPAddress structure into the actual DSP system gain factors. However, in addition to the mapping, the received value is range adjusted and clamped to the range [0, 1] in order to reduce the possibility of digital overdrive. It should be noted that this adjustment occurs in the server, the client produces output data that is not range adjusted. A simple change of clamp range to [-1, 1] allows the server to respond to negative values as inverse phase summing with no changes to the client code.

3.3.2 *Additional functionality*

In addition to DSP handling the server provides a very basic GUI with a console based display of current status. Functionality for displaying of errors and debug reporting was included to use the GUI console. Options for setting up network and ASIO are included and settings are stored between sessions via Windows registry to facilitate the minimum of human intervention in a remote system. SDServer

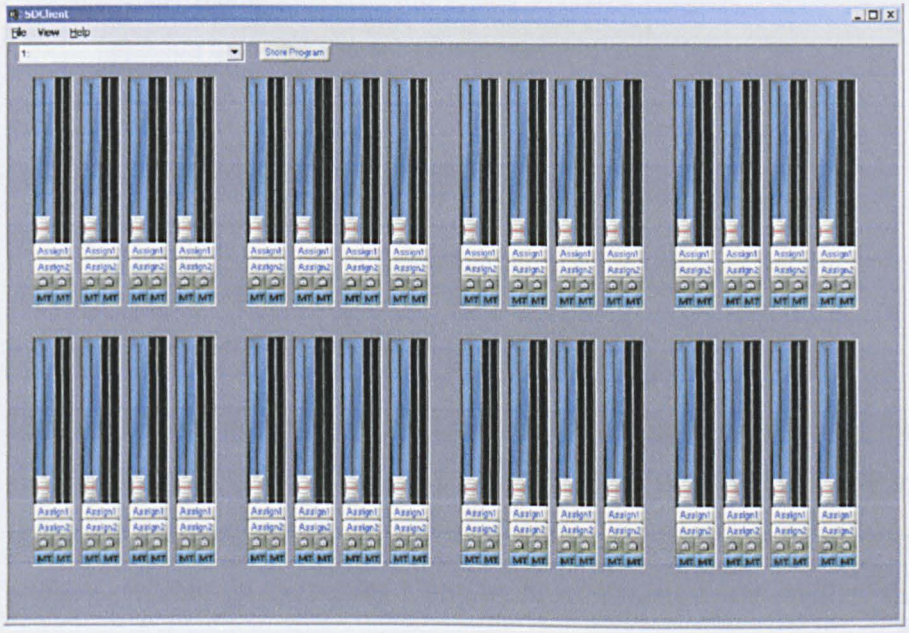
networking supports multiple clients at once and may have clients log in and out many times during a single server session.



This image shows the basic boot-up screen for SDServer, confirmation of initialisation of ASIO card and current network configuration are shown in the console view.

3.4 SDClient specific

SDClient forms the performer's graphical interface into the mix matrix provided by SDServer. External control via MIDI is directly mapped onto the main performance page consisting of 32 virtual faders. The assignment and control of these faders acts as the control entry point into a parameter / automation tree as discussed above. The main page is shown below:



3.4.1 Implementing a parameter base class

In SDClient the parameter is one of the most fundamental concepts of the system architecture. CParameter describes a basic parameter class that builds on the

theoretical parameter concept discussed previously. It also forms the base class for group and effect classes that implement automation. The CParameter interface is as follows:

```
class CParameter
{
        char name[32];
        int value; // the value of the parameter
        int lockedValue;
        bool changed; // flag
        BOOL isLocked;
        BOOL isMuted;
        CPAddress address;
        CParameterSystem* system;
public:
        CParameter();
        virtual ~CParameter() {};
        void SetAddress(CPAddress _address) { address = _address; };
        CPAddress GetAddress() { return address; };
        virtual bool ReferencesAddress(CPAddress _address) { return address == _address; };

        // relating to values
        virtual int GetValue();
        virtual void SetValue(int oldV, int newV);
        virtual void SetValueDirect(int newV) { value = newV; changed = true;};

        // relating to value lock
        virtual void SetLocked(int _value) { isLocked = TRUE; lockedValue = _value; changed = true;};
        int GetLockedValue() { return lockedValue; };
        virtual void Unlock() { isLocked = FALSE; changed = true;};
        BOOL IsLocked() { return isLocked; };

        // relating to mute
        virtual void SetMute() { isMuted = TRUE; changed = true;};
        virtual void UnMute() { isMuted = FALSE; changed = true;};
        BOOL IsMuted() { return isMuted; };

        // other things
        bool HasChanged() { return changed; };
        void ResetChanged() { changed = false; };
        void AttachSystem(CParameterSystem* _system)  { system = _system; };
        CParameterSystem* GetSystem(){return system; };

        void SetName(const char* _name) { strncpy(name,_name,31); name[32] = '\0';};
        const char* GetName() { return name; };

        virtual void Archive(CArchive &ar);
};
```

The basic functionality provides methods for setting and retrieving the stored value of the parameter. Additional methods allow the system to determine if a parameter has recently changed. Much of the interface is defined as virtual for the purposes of inheriting classes. Early in development it was decided that a parameter would have three states relating to value update and retrieval. Normally parameter values can be altered using SetValue() and SetValueDirect(). When in 'Mute' status the parameter is forced to a value of 0 while 'Locked' status allows a fixed value to be locked into the parameter. The purpose of these states is to disable any effects from either automation of external control while allowing the parameter to behave normally to the rest of the system. The implementation

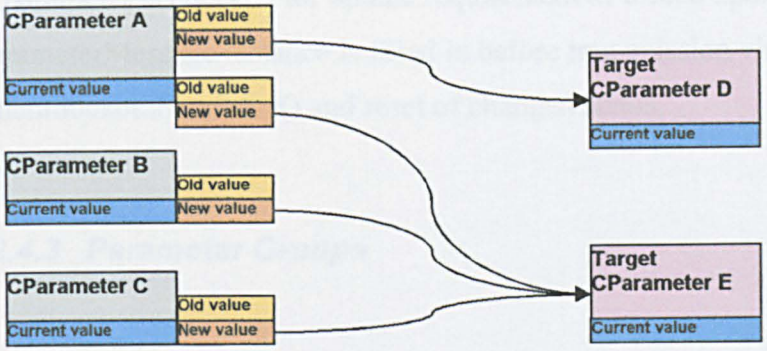
CParameter::GetValue() determines how these states affect the perceived parameter value:

```
int CParameter::GetValue()
{
	if(isMuted == TRUE)
		return 0;
	if(isLocked == TRUE)
		return lockedValue;
	return value;
}
```

Priority is given to the mute status, i.e. any muted parameter always returns zero. Second priority is given to locked status in which case the value returns its current locked value. Only in normal status does CParameter return its actual automatable value. Regardless of status, automation and external control can alter the internal value of the parameter but this value remains internal while any status is in effect. CParameter::SetValue() (below) only allows notification of change if during normal status.

```
void CParameter::SetValue(int oldV,int newV)
{
	int v = newV - oldV;
	if(v != 0)
	{
		value += v;
		if(isMuted == FALSE && isLocked == FALSE) // only set changed when not muted or
locked
		{
			changed = true;
		}
	}
}
```

CParameter::SetValue() and CParameter::SetValueDirect() provide two direct methods of setting parameter values. The SetValue() method requires the calling function to pass both the new and previously sent values for the purposes of correct summing. The actual method of summing is shown in the above code snippet; value delta from the calling function is first calculated and any changes are summed with the current parameter value. Performing the summing in this way allows multiple parameters to sum to a single target without the target storing linkage information. SetValueDirect() sets values without regard for the summing system and is mainly intended for initialization. The following diagram illustrates the variables required for correct summing of parameters. Note that the affected parameter does not store any information regarding the connected parameters.



3.4.2 *Parameter Management*

The management of parameters is implemented with the CParameterSystem class. This has a similar role to the DSPSystem class in SDServer, which is storage of all core system variables (CParameter instances) and correct addressing and updating of these variables. As with DSPSystem, CParameterSystem makes extensive use of Array and Array2 template classes. Methods are provided to directly access CParameter functionality for setting and retrieval of values without the need to extract the required parameter first. These methods allow the interlinking of parameter chains without the need for direct storage of pointers; parameter map address is sufficient to set or retrieve values. Network transmission of parameter values is initiated by SendChangedParameters(); only parameters registering changed values will be transmitted unless the forcedUpdate boolean is set forcing all parameters to be transmitted regardless. The following section of SendChangedParameters() highlights the basic method used: (note condensed version)

```
SDParameterMessage msg;

for(n = 0; n < numInputs; n++)
{       if(inputGain[n].HasChanged() || forcedUpdate)

                msg.addrX = PAR_INPUT_GAIN;
                msg.addrY = n;
                msg.value = inputGain[n].GetValue();

                socket->SendMsg(header,&msg,sizeof(SDParameterMessage));
                // now reset it
                inputGain[n].ResetChanged();
        }
}
```

Each parameter is checked for update requirement or forced update. An SDParameterMessage instance is filled in before transmission via SDClientSocket::SendMsg() and reset of changed status.

3.4.3 Parameter Groups

SDClients allows multiple parameters to be sub grouped together under single parameter control. In order to achieve this the CGroup class inherits and extends the basic CParameter functionality. CGroup stores an STL vector<> [47] of target parameter addresses and overrides SetValue() and SetValueDirect() in order to pass on parameter adjustments to the sub group. CGroup::SetValue() is shown below. Note differences between this and the CParameter::SetValue() above:

```
void CGroup::SetValue(int oldV,int newV)
{
	CParameter::SetValue(oldV,newV);
	GroupUpdate();
}
```

CGroup::GroupUpdate() is required by the above function and is shown below:

```
void CGroup::GroupUpdate()
{
	if(!GetSystem()) return; // groups have to be given a valid pointer to the parameterSystem
	if(HasChanged())
	{
		for(int n = 0; n < paramList.size(); n++)
		{
			int v = (float)GetValue() * (paramList[n].max / 128.0f);
			GetSystem()->SetParameter(paramList[n].addr,paramList[n].oldValue,v);
			paramList[n].oldValue = v;

		}
	}
}
```

CGroup makes use of the data structure 'CGroupParameter' for storage of target CPAddresses and old values. A simple 'for' loop through 'paramList' calling SetParameter causes update of sub parameters. Parameter adjustment is scaled with a maximum sub parameter value producing proportionate group control.

---

[47] Stroustrup, B: 2000, pp.442-458

3.4.4 Eliminating feedback

It should be clear at this point, that the ability to sub group parameters has the potential for feedback and thus stack overflow. A simple method of prevention is to remove the possibility of assigning a self referential loop. The algorithm used traverses the parameter tree checking for a self reference and is performed on assignment of a new target parameter. Due to the modular architecture it is necessary for each CParameter inheriting class to provide its own referential check algorithm. As an example, the following code shows CGroups self referral check algorithm:

```
bool CGroup::ReferencesAddress(CPAddress _address)
{
	if(CParameter::ReferencesAddress(_address)) return true;

	// check all sub addresses
	for(int n = 0; n < paramList.size(); n++)
	{
		if(GetSystem()->GetParameter(paramList[n].addr)->ReferencesAddress(_address))
		{
			return true;
		}
	}
	return false;
}
```

A recursive technique is used to determine self reference against a specified CPAddress; first check against itself using a call to the base class (CParameter::ReferencesAddress()); second, check all contained targets via calls to their overloaded ReferencesAddress() functions. In the event of an address match, the function will return true and the system will be unable to use the specified address as a target within the tested chain. This method of recursive checking through polymorphic functions is future compatible with any new grouping or automation effects. The following code section shows CGroupWnd using the recursive referential check prior to assignment of a parameter to a CGroup (code defined in CGroupWnd::AddParameter()):

```
// check for referencing of this group
if(param->ReferencesAddress(groupAddr))
{
AfxMessageBox("Super diffuse could not add the selected parameter \n to the group due to a circular
reference");
return false;
}
```

A 'Circular reference' warning message is displayed to the user if the reference cannot be added, but the system recovers.

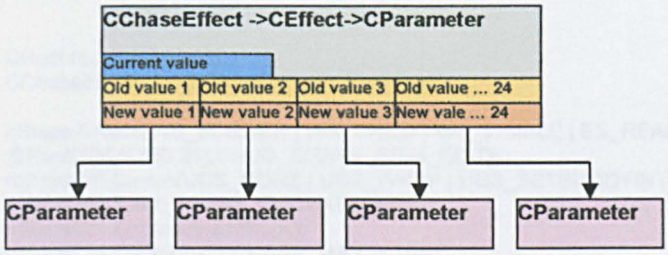
3.4.5 Parameter Automation (Effects)

Updating the CParameterSystem class via its Tick() method causes update of automatable parameters. Automation algorithms are described by inheriting from the CEffect class which extends the CParameter base class and defines the CEffect::Tick(float delta) virtual function. The current version of SDClient introduces three different automation effects but system architecture allows future automation types to be added with only small alterations. Although not yet implemented, future revisions will aim to provide plugin based modularity for automation effects and much of the ground work for this is already in existence. The following code snippet shows the Tick() overloading for the chase effect:

```
void CChaseEffect::Tick(float delta)
{
	float amp = (float)GetValue() / 128.0f;
	float pos = fmodf(t * freq1,numSteps); // get relative position from time using modulus
	int lIndex = (int) pos; // get left index
	int rIndex = lIndex + 1; // get right index
	if (rIndex >= numSteps) rIndex = 0; // check for overlapping index
	int oIndex = lIndex -1;
	if (oIndex < 0) oIndex = numSteps - 1;
	float nullFloat;
	float rad = modff(pos,&nullFloat) * (3.14159f/2.0f); // get fractional part * 90 degrees in rads
	// use the fraction to calculate crossfade between lIndex value and rindexValue

	// set only two values
	int newL = (int)((float)val[lIndex] * cosf(rad) * amp);
	int newR = (int)((float)val[rIndex] * sinf(rad) * amp);

	GetSystem()->SetParameter(addr[oIndex],oldV[oIndex] , 0); // should be 0 when chase has passed it
	GetSystem()->SetParameter(addr[lIndex],oldV[lIndex] , newL);
	GetSystem()->SetParameter(addr[rIndex],oldV[rIndex] , newR);

	oldV[oIndex] = 0; // this one should be 0 by the time the chase has past it
	oldV[lIndex] = newL;
	oldV[rIndex] = newR;

	t += delta; // update the time
}
```

The chase effect stores an array of target addresses (24 in total) and performs cosine based panning laws to crossfade between each. The calls to CParameterSystem::SetParameter() connect the chase effect to its targets with address validity checking. It should be noted that old values (last set values) are stored and retransmitted on subsequent Tick() calls. This storage of old values is a requirement of the parameter summing method (see above) and is used to calculate delta value inside CParameter::SetValue(). The following diagram

illustrates the data storage for the chase effect. Note that CChaseEffect is derived from CEffect and therefore CParameter making it a viable target parameter:



3.4.6 GUI for automated parameters

In order to provide modular user interfaces for the modular parameter system the CFXEditor class was developed in conjunction with the CEffect. The technique used here is similar to Steinberg's VST Plugin editor classes although the GUI code makes extensive use of MFC rather than Steinberg's use of Win32 core libraries. One key factor in this GUI implementation is the lack of window persistence between openings. Rather than store an inactive editor, the system simply discards the last opened window and creates a new one on request. The new window is initialised to the current status of the associated CEffect upon opening and remains linked until a new window is requested. It is the requirement of the CEffect derived class to create its own editor and this action must be performed in overriding the CEffect::GetEditor() virtual function. The following shows how CChaseEffect creates a new editor. Note here the passing of the 'this' pointer into the parameterized constructor of the CChaseEffectEditor derived from CFXEditor.

```
CFXEditor* CChaseEffect::GetEditor()
{
        return new CChaseEffectEditor(this);
}
```

After construction, the new CFXEditor derived instance is returned to the calling function and it is its job to create the appropriate container window and finalize editor construction. After appropriate window construction the editor instance has

its OnCreate() handler called by the framework, CChaseEffect::OnCreate() is shown below:

```
int CChaseEffectEditor::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CFXEditor::OnCreate(lpCreateStruct) == -1)
		return -1;

	CRect rect(5,5,300,30);
	CChaseEffect* f = (CChaseEffect*) GetEffect();

	nSteps.Create(WS_BORDER | WS_CHILD | WS_VISIBLE | ES_READONLY
	,CRect(300,5,350,21),this,ID_CHASE_SPIN_EDIT);
	nStepsCtrl.Create(UDS_HORZ | UDS_WRAP | UDS_SETBUDDYINT | WS_CHILD | WS_VISIBLE,
	CRect(350,5,400,21),this, ID_CHASE_SPIN);
	nStepsCtrl.SetBuddy(&nSteps);
	nStepsCtrl.SetRange(3,CHASE_MAX_STEP);
	nStepsCtrl.SetPos(f->numSteps);

	freq1.Create("Frequency",WS_CHILD | WS_VISIBLE, rect,this,IDK_FREQ1);
	freq1.SetVertical(FALSE);
	freq1.SetMax(CHASE_MAX_STEP);
	freq1.SetMin(0);
	freq1.SetValue(f->freq1);

	rect = CRect(0,50,25,71);
	CString str;
	for(int n = 0; n < CHASE_MAX_STEP; n++)
	{
		str.Format("%d",n+1);
		assign[n].Create(str,WS_VISIBLE,rect,this,ID_FIRST_CHASE_ASSIGN + n);
		rect.OffsetRect(25,0);
	}

	rect = CRect(0,80,25,380);
	for(n = 0; n < CHASE_MAX_STEP; n++)
	{
		str.Format("Value %d",n+1);
		vals[n].Create(str,WS_CHILD | WS_VISIBLE, rect,this,ID_FIRST_CHASE_VALUE + n);
		vals[n].SetMax(127);
		vals[n].SetMin(0);
		vals[n].SetValue(f->val[n]);
		rect.OffsetRect(25,0);
	}
	return 0;
}
```

Upon successful creation a CFXEditor creates all appropriate controls and initialises them to the values stored in the linked CEffect obtained from CFXEditor::GetEffect(). Normal Windows message mapping is used to perform user interaction, with the linked CEffect adjusted accordingly. The above editor uses a combination of basic MFC controls and some specialised ActiveX[48] controls developed for SDClient.

3.4.7 SDClient archiving

[48] Williams, A: 2000, p.115

SDClient makes use of the MFC archiving system[49] for the purpose of data storage and retrieval. Archiving requirements are much simplified over the complex technique used in Ricochet[50] due to the simple software design approach. A basic archiving tree is formed by data and system classes providing Archive() functions, a CArchive MFC instance is passed through the tree and all data is either retrieved from or stored to it. Use of fixed sized object arrays rather than dynamic structures has greatly simplified archiving by reducing the need to reconstruct references and pointers between sessions. The following code snippet shows the CRandomizeEffect::Archive() function: (Note MFC practice of using one archive function for both storage and retrieval in order to maintain accurate file formats)

```
void CRandomizeEffect::Archive(CArchive &ar)
{
	CEffect::Archive(ar); // call base class to store base info

	addr1.Archive(ar);  // only single reference to  CPAddress::Archive
	addr2.Archive(ar); // for both storage and retrieval

	if(ar.IsStoring())
	{
		ar << freq1; // operator << overloading makes variable storage easy
		ar << freq2;
		ar << amp1;
		ar << amp2;
	}
	else
	{
		ar >> freq1;
		ar >> freq2;
		ar >> amp1;
		ar >> amp2;

		oldV1 = 0;
		oldV2 = 0;
	}
}
```

A further advantage of the MFC CArchive class is the overloading of << and >> operators for most built in types, this is also a feature of the C++ STL ofstream[51] class. This method makes single variable storage and retrieval very simple. The Advantage of the single function for storage and retrieval is justified by the CPAddress::Archive() function used to both store and retrieve in one function call.

---

[49] Feuer, A. R: 1997, p.233

[50] See also 5.6

[51] Stroustrup, B: 2000, p.637

3.5 M2 User Manual

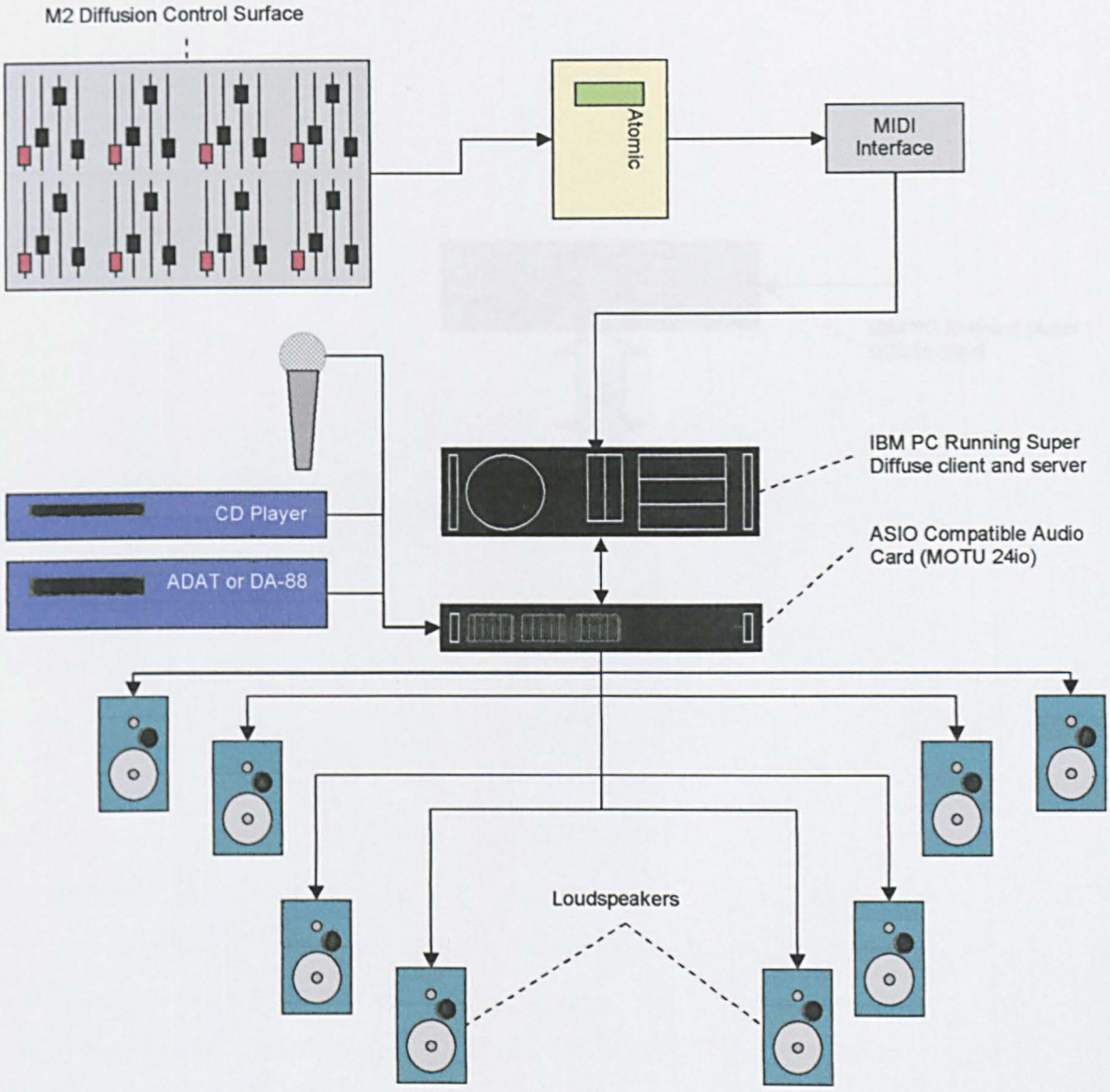
The M2 hardware system comprises the following components:

- M2 Diffusion Control Surface[52]
- CV to MIDI conversion device (Ircam Atomic)
- MIDI interface (MAudio MidiSport 2x2)
- IBM Compatible PC (AMD Athlon 2500 CPU, ASUS Motherboard with onboard LAN and XVGA graphics support)
- Steinberg ASIO compliant audio card (MOTU 24io)

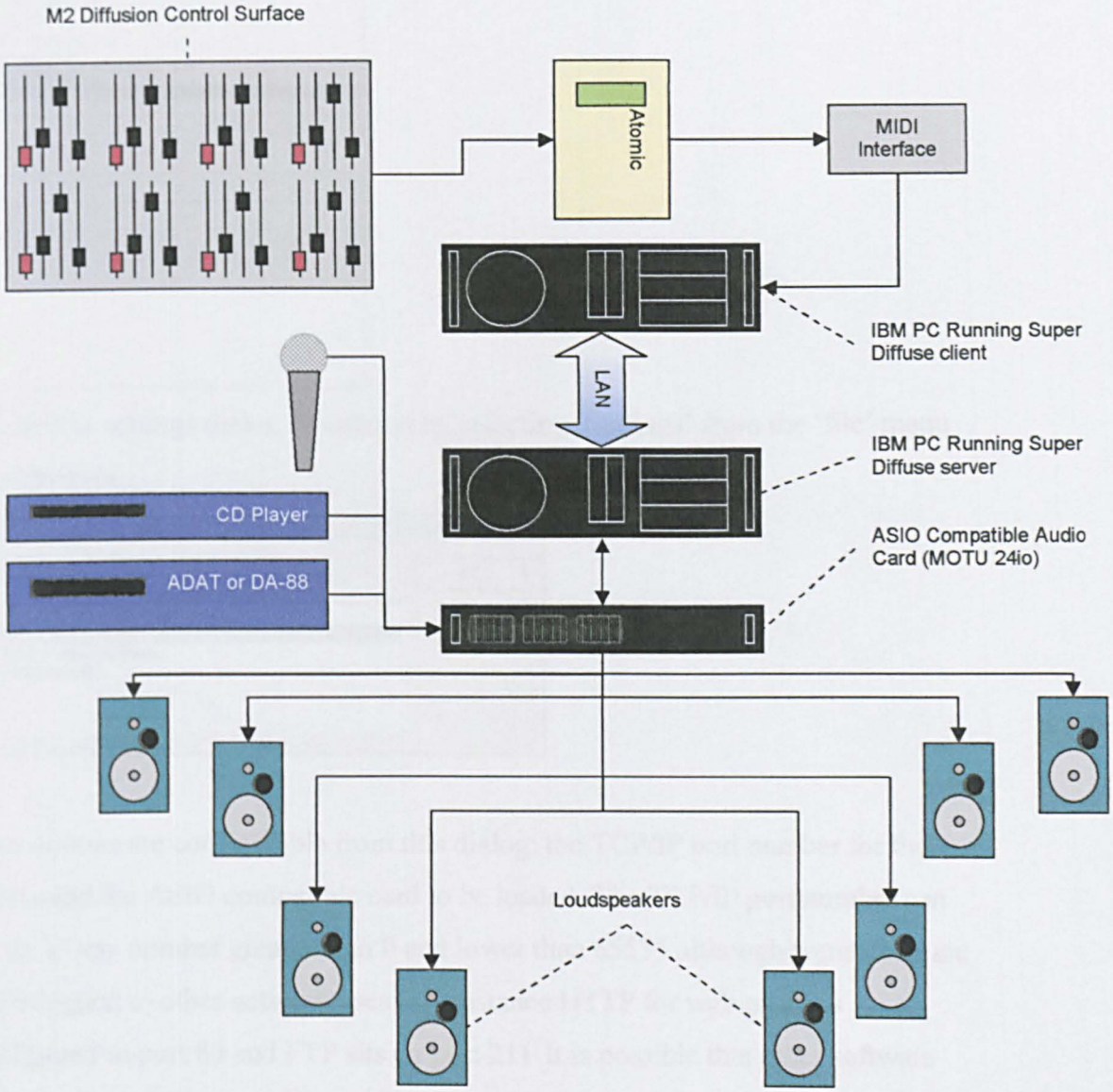
These hardware components form a single computer setup but the client software can be run on a second PC for a remote server setup. These two setups are described below:

[52] Designed by Mooney

3.5.1 Setup method 1 (Single PC system)

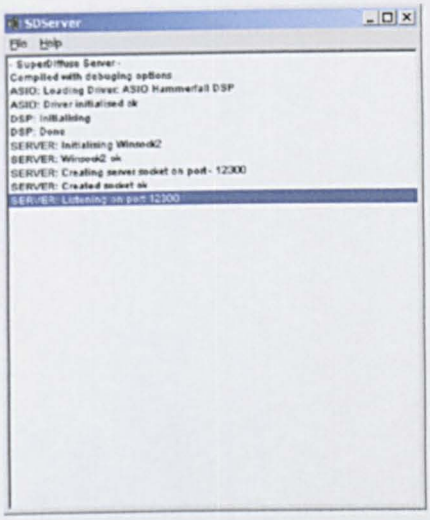


3.5.2 Setup method 2 (Dual PC system)

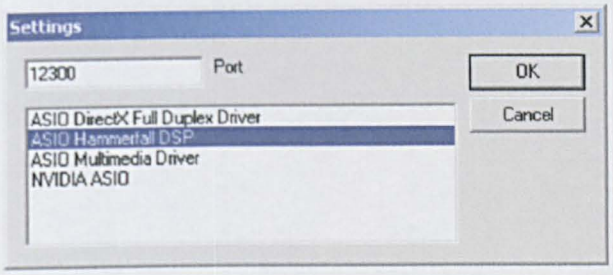


3.5.3 SDServer Configuration

Before using the client software it is necessary to run SDServer software. The client software cannot operate fully without connection to an instance of the SDServer that has been correctly set up.



The server settings dialog is entered by selecting 'Settings' from the 'file' menu of SDServer.

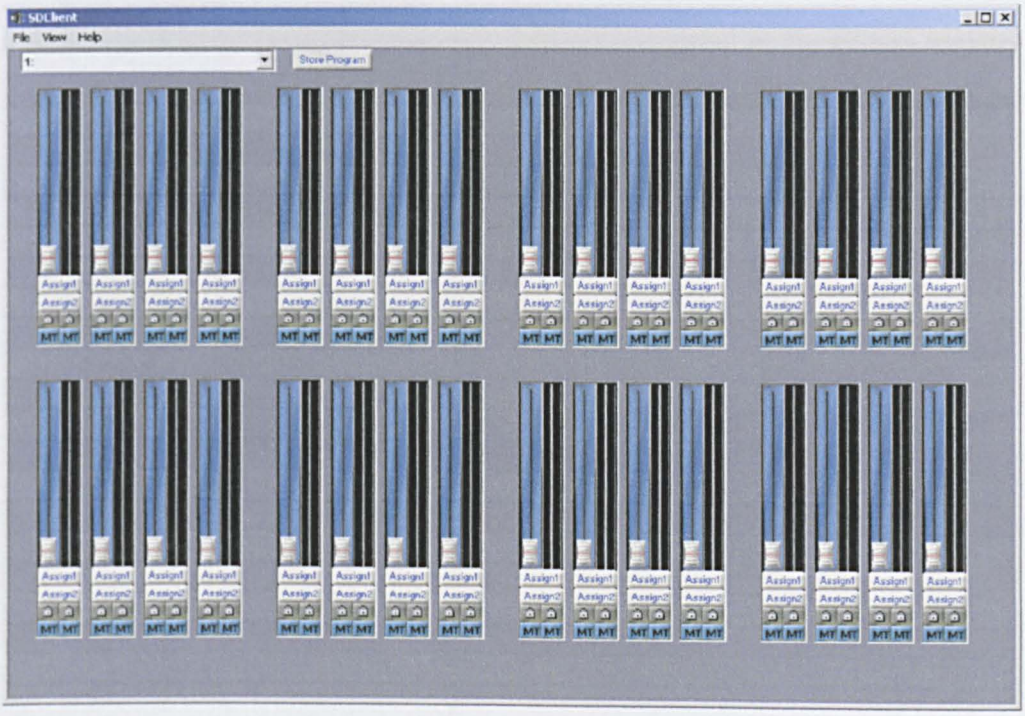


Two options are configurable from this dialog: the TCP/IP port number for the server and the ASIO compatible card to be loaded. The TCP/IP port number can be set to any number greater than 0 and lower than 65535 although some ports are pre-assigned to other network uses (for instance HTTP for web pages is configured to port 80 and FTP sits on port 21). It is possible that other software may need to use the default port of 5000 so this setup functionality allows SDServer to use a different setting. ASIO cards that are available for use will automatically appear in the driver list but this does not necessarily mean that they will work correctly. Although a number of cards have been tested it is possible that SDServer cannot use some manufacturer's cards.

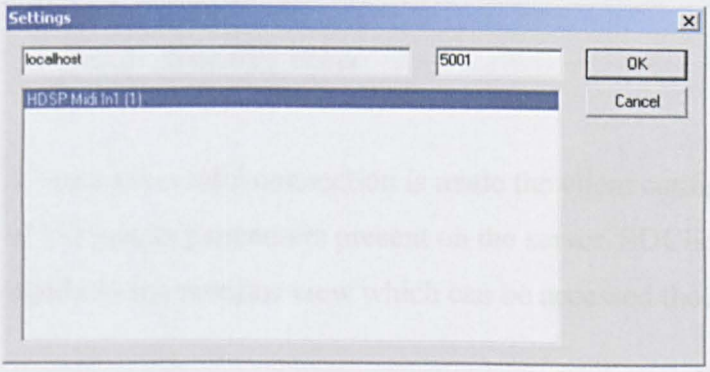
Upon exit of the settings dialog, SDServer will detect changes and attempt TCP/IP listening on the selected port and load the ASIO driver. Settings are automatically saved to the system registry if successful connections are made. Success is reported in the main SDServer dialog with 'SERVER: Create socket ok' and 'ASIO: Driver initialized ok'.

3.5.4 SDClient Configuration

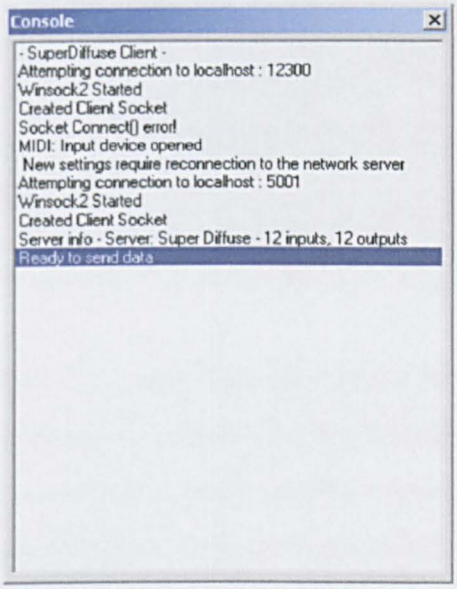
With the server software running in the background, SDClient can be executed and will initially show the following screen:



The opening page is known as the performance view and shows a direct visualization of the 32 assignable master faders. The top 16 faders are controlled externally via MIDI controller 7 (Channel Level) and the lower 16 are assigned to controller 10 (Channel Pan). In order to correctly setup both the external MIDI device and the network connection a user selects the settings option from the File->Settings menu shown below:



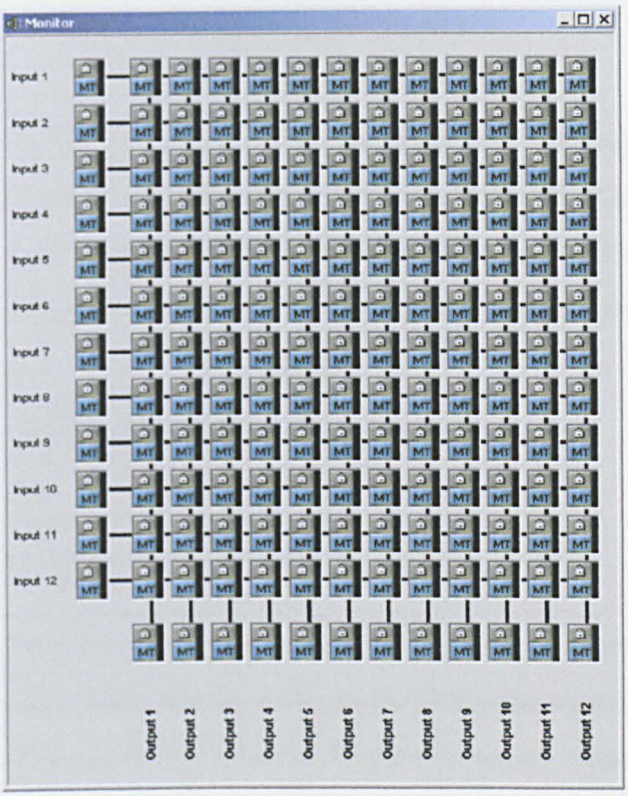
In the above screen shot the client is attempting to connect internally within one machine and 'localhost' has been set as the network address. It is possible to specify any IP address or domain name upon which an SDServer is running. The port '5001' is set to the ip port decided on when setting up the server. In addition to the network settings 'HDSP Midi In (1)' has been selected as the MIDI device for external control. The settings dialog will display all MIDI devices it finds on the local machine. As with the server, settings are stored in the system registry upon exiting the dialog. Status is displayed in a similar console view accessed via the View->Console menu.



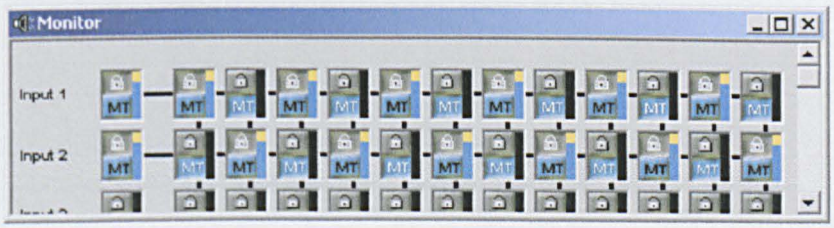
The console view displays the current status of the connection to an SDServer and shows external MIDI device status. In the above screen shot the line 'Server info – Server: Super Diffuse – 12 inputs, 12 outputs' shows a valid connection to an SDServer that has control of a 12*12 mix matrix. This line would display the size of the connected mix matrix on the remote machine.

3.5.5 Monitor view

When a successful connection is made the client configures itself for the number of I/O matrix parameters present on the server. SDClient represents the connected matrix in the monitor view which can be accessed though View->Monitor:



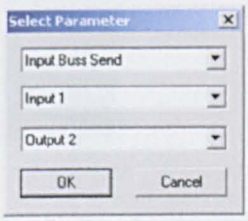
This view shows the attenuation level setting of all parameters in the server. In the screenshot below the lock (padlock icon) and mute (MT) buttons have been used to set direct connections from inputs 1 and 2 to stereo pairs of outputs. A locked parameter is set to zero attenuation and is unaffected by automation or external hardware. A muted parameter is set to full attenuation and is unaffected by automation or external hardware. Muted parameters also override the locked status.



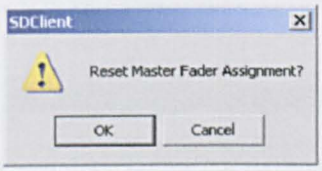
Used in this way the Monitor window can act as a simple direct routing system for all I/O ports on the connected SDServer.

3.5.6 Assigning parameters

On the performance page the moveable fader representations can be assigned to any of the matrix parameters by selecting the 'Assign1' or 'Assign2' buttons. This representation also displays and allows control of the lock/mute status for the assigned parameter. After selecting 'Assign1' or 'Assign2' the user is presented with the following dialog:



Parameters are selected via the drop down combo boxes and any parameter or automation is selectable. In the above the matrix parameter for input 1 attenuated into output 2 is selected. Right clicking on either of the assign buttons removes any assignment after displaying the following confirmation box:



The close up view below shows a single fader assigned to a parameter and with its control level turned up to around 75%. The blue and yellow bar displays the actual value of the assigned parameter which may not correspond to the fader value if automation or grouping has been used. Also note the 'Assign' buttons visible from this screen shot.

3.5.7 Groups

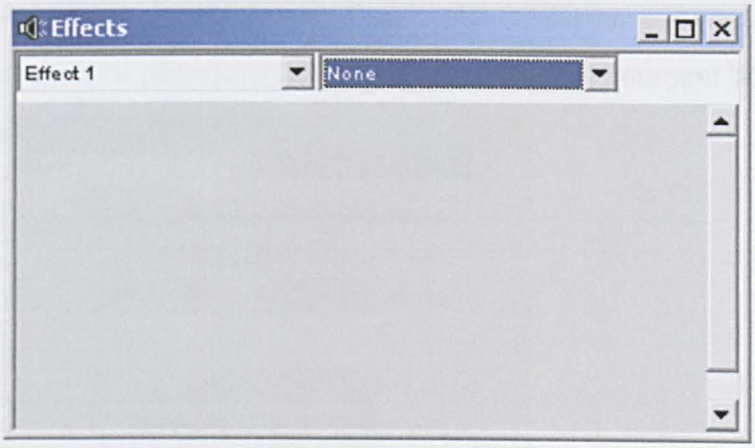
If the user needs to assign a group of parameters to one fader this can be done in 'Group view' accessible via the View->Group menu item.

Above, the user has assigned all 12 available outputs to the group simply by clicking inside a parameter box and dragging up or down to select a relative level. Parameters can also be added via the 'Add Parameter' push button which brings

up the common parameter select dialog discussed previously[53]. Note here that some parameters (Output 2 and 4) have been assigned negative level (-25 and -20) represented in numerical form and by the orange bar graph. Negative values in groups can be used to assign parameters that need to be reduced as the fader is turned up. The 'RM' buttons in each parameter allow a parameter to be removed from a group and the combo box in the upper left allows selection of the group to be edited.

3.5.8 Automation effects

Automation effects are created and edited through the Effect View accessible via the View->Effects menu item:



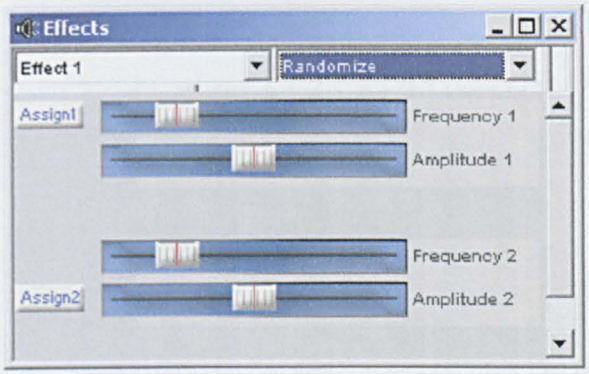
The initial view shows that Effect1 currently contains no effect. An effect is selected via the top left combo box and its type is assigned from the top right combo box. Depending on the selected effect type the appropriate effect interface is shown. Below, the three effect types are shown:

3.5.9 Randomization effect

This effect generates randomized values over time.
Frequency and amplitude of the random value generation is controllable with sliders and the effect may be connected to two independent parameters via the

[53] See also: 3.5.6 *'Assigning parameters'* p.66

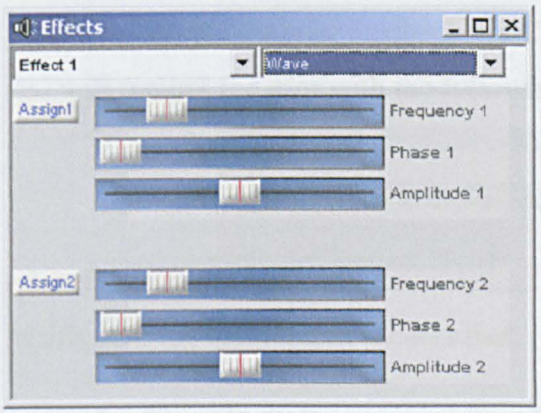
'Assign1' and 'Assign2' buttons. As with other parts of the software, the common parameter select dialog is brought up from these buttons.



3.5.10 Wave automation effect

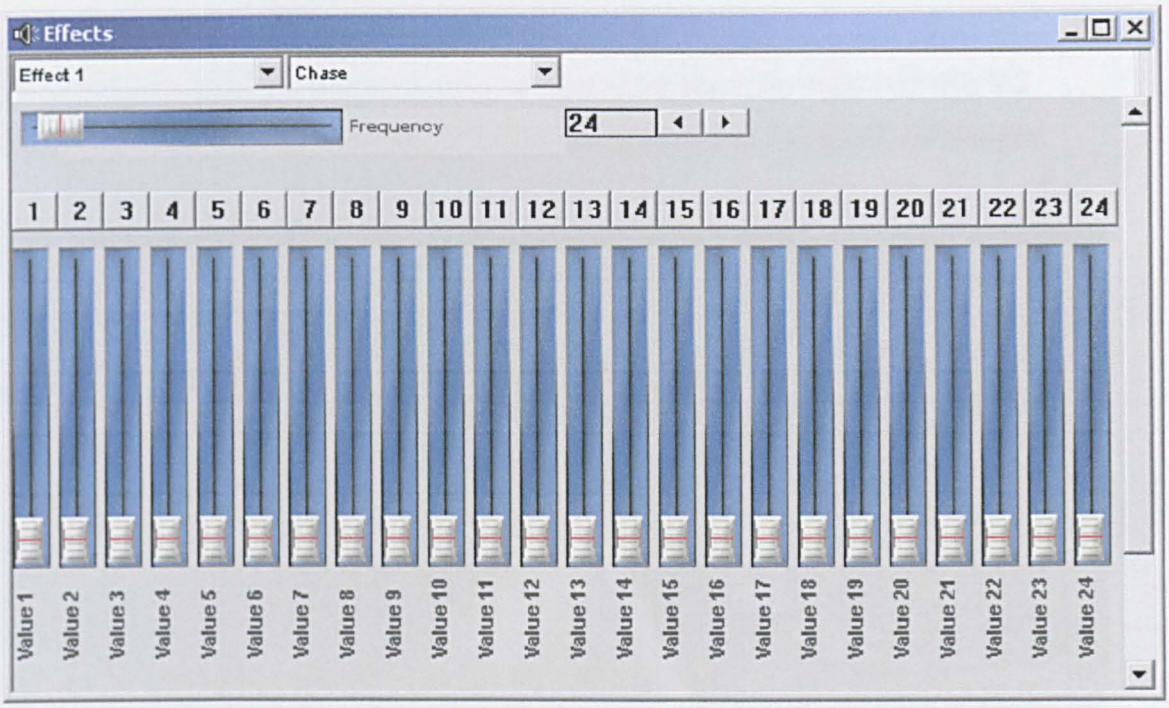
This effect produces a time varying cosine wave.

Frequency amplitude and phase of the cosine wave is controllable from faders and again, two parameters may be assigned via the common buttons.

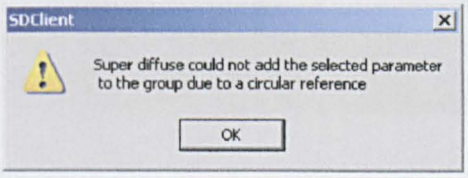


3.5.11 Chase automation effect

This effect acts in a similar manner to a lighting 'Chase', crossfading between a sequence of parameters over time. The numbered buttons allow common parameter selection for any of the 24 assignable slots. Frequency or speed of sequence is controlled from a horizontal fader and the loop step is selected from the left and right arrow buttons. The loop step will always crossfade along the sequence starting from step 1 and moving onto and including the loop step. For each loop step it is possible to set a maximum level using the vertical faders.



It is possible to assign groups of effects and assign groups to effects and this can lead to accidental circular feedback. SDClient prevents feedback automatically and will present the user with the following dialog if an assignment that would cause it occurs:



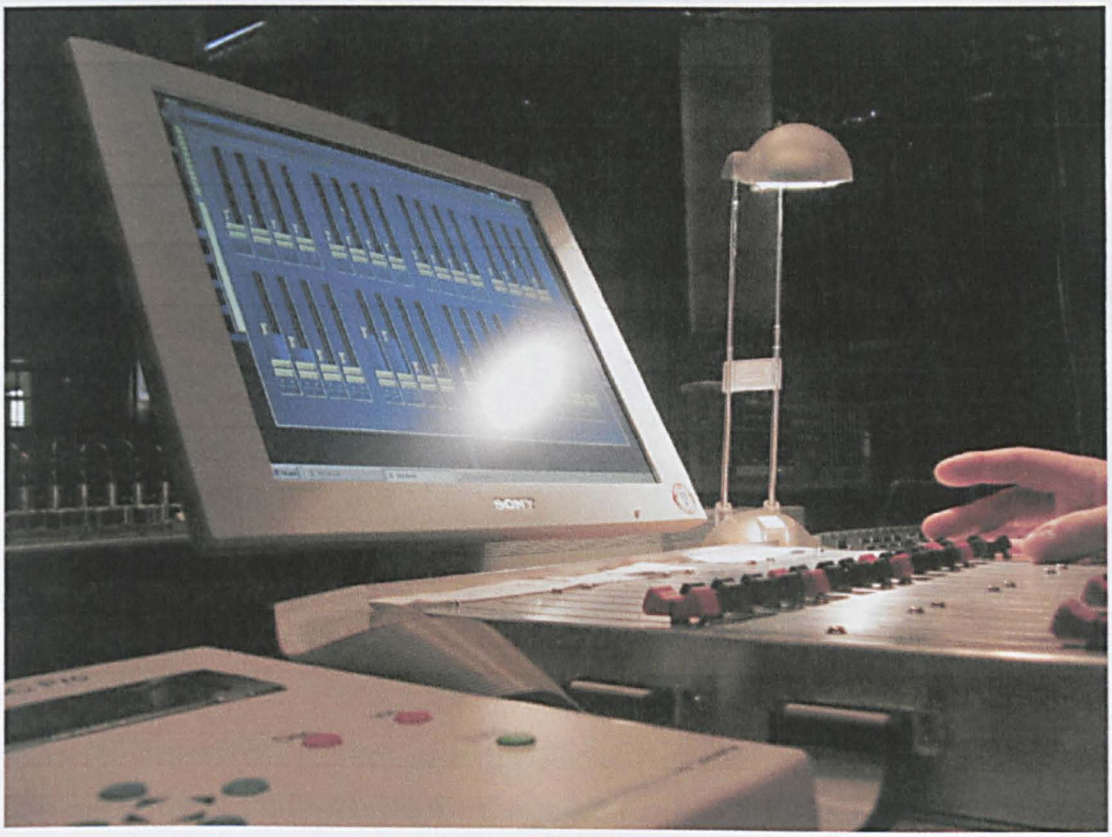
In this case the assignment is cancelled.

3.6 The M2 Diffusion system incorporating Super Diffuse

As described during the introduction to this project, Super Diffuse forms the basis for the University of Sheffield Sound Studio's 'M2' live sound diffusion system. The M2 platform consists of industrial rack mounted computer, custom built fader system and Super Diffuse software components. M2 has at the time of writing managed live sound diffusion for concerts at the universities of Sheffield, Birmingham, Bangor and Edinburgh and is the focus of *'M2 Diffusion – The live diffusion of sound in space'* a paper co-written by A.Moore, D.Moore and J.Mooney presented at the ICMC[54] 2004.

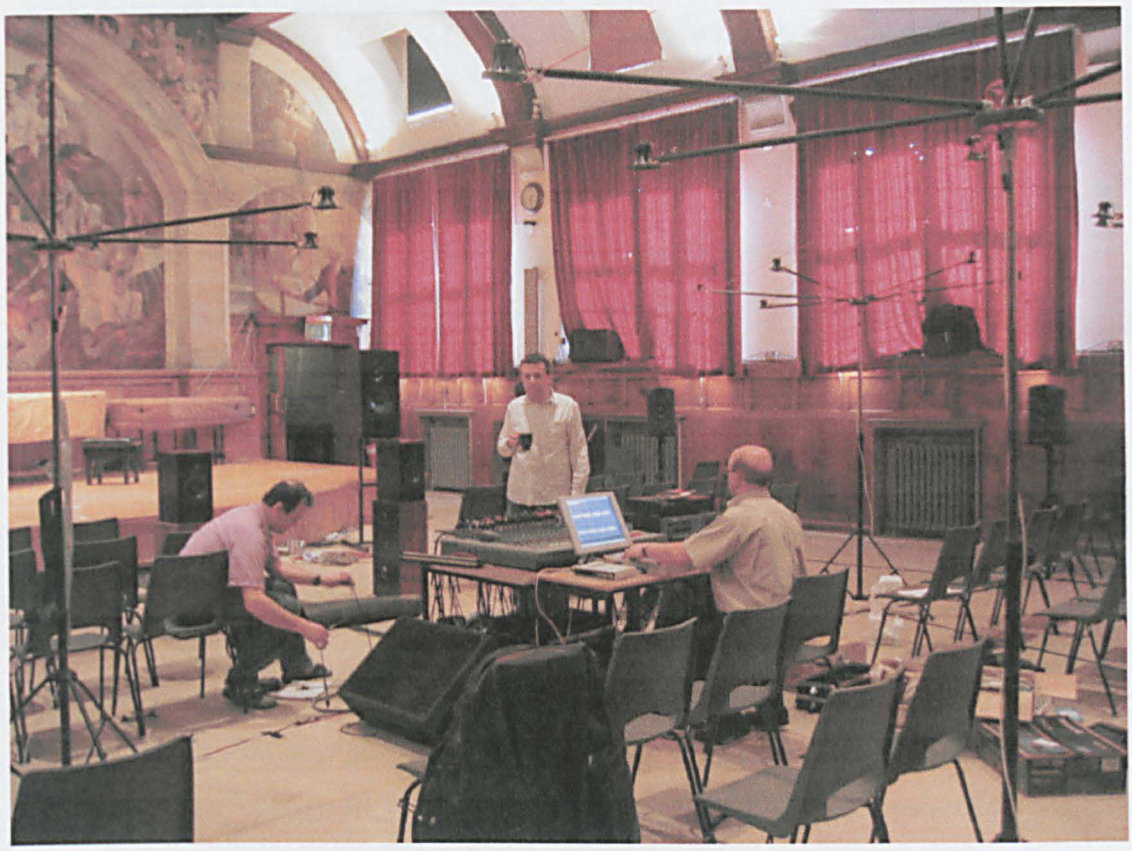
[54] ICMC - International Computer Music Conference

The following photographs were taken during setup and rehearsals with the M2 system for the 2004 Electroacoustic Wales performance at University of Bangor.



Above: Close up photograph of 'Super diffuse' software with the M2 Control hardware visible.

Below: View of the diffusion loudspeaker setup at Bangor, A.Moore in control of the system.



3.6.1 M3 – the future expansion of the M2 system

Super Diffuse and M2 is a prototype system and as such is now the focus of ongoing research. Collaboration with composers and performers has already highlighted many new directions for the project. At the time of writing a number of universities have agreed to a development partnership focusing on the development of M2. It is hoped that this partnership will provide a larger pooling of ideas for both M2 and future sound diffusion tools.

Performance with M2 has highlighted the need for future improvements to setup logistics. Although this is mainly a user interface problem it is clear that improvements need to made to provide a clearer system for classification of I/O ports in audio hardware and a much improved method of selecting different control parameters. Initial development ideas have settled upon the general consensus for a 'Venue Schematic' view. The intention of the GUI element is to construct and present a visual representation of the room layout to the user. This schematic view concept can naturally be extended to cover other areas of the program such as external hardware layout and spatial configuration of input sources. In mapping input schematics against venue schematics it is hoped that

selection will become more intuitive and the possibility of automatic re-mapping for new venues will be possible.

GUI complication implications when using large numbers of I/O channels causes a difficulty in visualising the DSP matrix. In the M2 development system consisting of only 24 inputs and 24 outputs, the user is presented with a very large and complex interface to perform both the assignment of parameters and the viewing of parameter status. In a 96 * 96 channel system, the maximum possible with current hardware[55], parameter presentation would be unusable. Although the use of the 'schematic view' concept will hopefully address this problem to some extent it is felt that functionality should reflect the common usage of low numbers of inputs and high numbers of outputs. As hardware manufacturers typically support equal numbers of I/O channels it has been suggested within development meetings that the system should support low level disabling of hardware I/O ports. If ports are not needed in a particular setup they should be turned off and thus not be included in any interface visualization. DSP performance in the server would also benefit greatly from the ability to turn off channels as conversion from hardware data formats to internal 32 bit float need not occur for unused channels. ASIOSubSystem design is perhaps naïve in assuming that a piece of software would require all of its I/O passing to the host application but this conclusion has only become apparent after using hardware with large I/O numbers.

An ability for the server to take the place of external playback hardware such as CD player, ADAT/DA-88 based tape or other multi-channel playback system again reduces the necessary logistical problem of providing for multiple formats at concert time. In the author's experience, it is common to provide a number of external playback systems during performance and this is essentially a redundant concept if a well featured file format and playback system can be provided. Server side audio streaming directly to the I/O system is proposed in M3 and will be closely tied to the concept of a concert program. Concert programming will provide automatic reconfiguration of the system for the specific needs of a

[55] Hardware: 4 * MOTU 24io on PCI-424 master card.

particular concert item. In order to provide stable file playback the system will use a strictly tested audio format and conversion from performers' presentation formats will occur during setup and rehearsal. Live input from external playback hardware will continue to be supported, but the internal system should remove the need for external hardware in the majority of cases. It is also conceivable that a playback system could support streaming over IP and allow the client to play files directly, although this will not be a feature included in early re-designs.

In order to reduce the need for external hardware and improve logistical setup times it would seem useful to provide an output master section for loudspeaker balancing, EQ and phase compensation. M3 currently proposes output trim controls and a delay unit that can have distance based delay times added. By increasing the delay time of close speakers it is possible to compensate for phase de-correlation in speakers at different distances for the audience. Although this feature is perhaps undesirable for the traditional sound diffusion concept it could still find a use in correcting speaker pairs that would ideally be correlated but cannot be due to the constraints of the venue. Adding the facility for real position measurements to be entered, the 'Venue schematic' concept could be extended to provide the interface for correcting phase de-correlation over sets of loudspeakers. In modern PA systems the use of multi-band EQ is commonplace for correcting the frequency response characteristics of the venue and loudspeakers[56]. It is sensible to assume that corrective EQ might be useful for diffusion systems although its use would again be purely optional. It is possible that this room corrective matrix section could be extended to provide automatic correction from a reference pink noise generator, a system commonly available in digital multi-band EQ units. In the first iterations of M3 development it is certain that output DSP will be designed carefully for the future addition of features discussed above.

In consideration of extending the master output section it is logical that the full DSP matrix concept also be considered and the ability to insert audio plugins, in either VST or DirectX format, has been put forward. Although the addition of this feature would provide a great deal of scope for experimentation, the inherent

---

[56] Stark, S. H: 1996, p.97; Davis, G; Jones, R: 1990, pp.251-252

complication to the DSP section and the possibility of third party plugins causing unforeseen server failure during performance compels a decision not to include this functionality. Obviously, tailor made and well tested algorithms could extend the DSP matrix design with less chance of compatibility problems but the decision has been made not to extend from the attenuators in the first iteration of M3.

In the current version, server to client feedback is very limited with only a small amount of initial hardware configuration information provided upon client connection. In future versions it is intended that the server become more autonomous and provide full control from the client. This is of particular use if the server system is to be locked away, perhaps made inaccessible except for maintenance. Additional feedback of run time information such as audio metering, DSP performance status and error reports would be of benefit to an audio installation. Taken to the extreme this feature could be extended for the purposes of venue health and safety, with supervisor setting of SPL limits and connection to alarms for quick shutdown in a fire. Theatre house lighting systems are often connected to alarms so that lights are forced up for escape.

Many users of M2 have stated that the ability to control sub parameters of automation effects during a performance would be desirable. This feature was included to some extent in the original 'Super Diffuse' but the interfacing functionality and bug testing was not ready for performance and was left out in tested versions. It has been unanimously decided within the development group that this feature will be complete in M3. Addition of this feature is likely to require extensive reviewing of the current parameter system model in order to produce the stability required.

Super Diffuse was originally written for the purposes of performance and one proposed extension is to provide two modes of use for the system. In 'Rehearsal' mode the M3 system will allow editing of performance setup and rehearsal of performance items. The 'Performance' mode is intended to lock settings and provide a degree of stability at performance time. Again this is a concept borrowed from lighting desks which often have similar modes available. It is desirable that performance setup cannot be altered during performance in order to

remove the chance of user error. In simplifying the interface options at performance time the system can be constrained much more efficiently.

In addition to mode setting and multi-channel audio file playback it is proposed that a user should have the facility to record performance cues. These cues would be named and provide storage for notes. During performance mode the system would present the stored cues in sync with audio file playback. With the addition of a stored 'Preroll' time it would be possible to present the cues prior to the related event. In this case a performer would be alerted to significant events during the recording in time to react accordingly. Performance cues have been met with general approval for M3. Future possibilities of this concept include the addition of time locked graphical score display although this will not be featured in the next iteration.

Following discussion regarding the hardware control it has become desirable to provide a hardware abstraction layer similar to that used in ASIO and DirectX for the purposes of generalizing the control method. It is intended that this layer will gather together a number of different hardware control concepts and provide a stable API for use in M3 and other software projects. After further design meetings this system may become linked with ASIOSubSystem[57].

A.Moore (University of Sheffield) has specified a desire for the addition of performance data logging[58]. The goal would be to record sufficient control data for both reproduction and analysis of performance spatialization. This feature could be added to the server system and it is logical that performance data would be logged using the standard MIDI file format. In this format, analysis with 3rd party software would be supported as it is not a goal of M3 to provide local features for statistical analysis.

It should be clear that the basic parameter concept discussed in this thesis is capable of providing control of other performance based systems. In fact, due to

---

[57] See also: 6.1.4

[58] Moore, A; Moore, D; Mooney, J: 2004

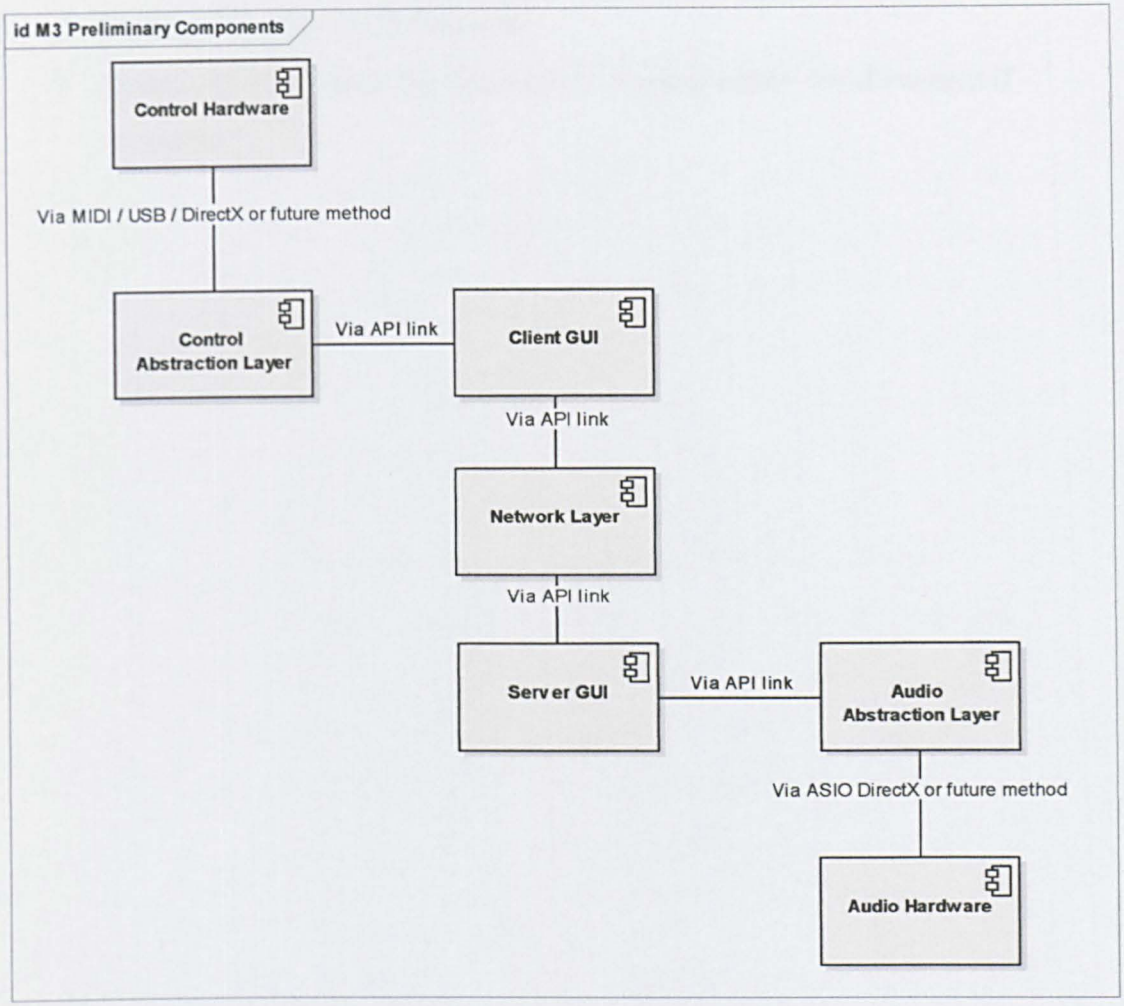
its lighting design concept, it is suited to the control of DMX (Acronym of Digital Multiplex) lighting systems. Very low cost DMX control hardware is available and could easily be addressed from the server software. Although this extension could be useful in some circumstances, M3 is unlikely to provide equal functionality to high quality specialist lighting systems. However, the ability to easily extend the basic parameter system within both the server and the client is perhaps desirable for the addition of control features in later versions. For this reason, M3 design will focus on providing a more extensible parameter system.

At this early stage, much of the design for M3 is not yet fully clarified but some basic guidelines for the development cycle and early software structure diagrams have been put forward as follows:

UML based CASE[59] tools for development have become of interest to the development team and it is hoped that following a well planned software development technique such as 'Rational Unified Process'[60] will promote a stable, well thought out solution.

[59] Computer Aided Software Engineering

[60] Booch, G; Rumbaugh, J; Jacobson, I: 2003, pp.449-453



This layered approach to the components of the system is intended to promote staged development of each component in isolation. For example, providing the interfaces for the Control Abstraction Layer and Network Layer remain unchanged, it will be possible to produce new versions of the Client GUI component. Each component's development will apply the following staged development cycle:

- **Phase 1:** Proposal of ideas and requirements. Address feasibility issues etc.
- **Milestone 1:** Produce requirements 'use case' and feature list documents.
- **Phase 2:** Design and implementation, refinement and finalizing.
- **Milestone 2:** Produce final design document for this iteration.
- **Phase 3:** Construction following design documents.
- **Milestone 3:** Release candidate for testing.

- **Phase 4:** Testing and refinement.
- **Milestone 4:** Produce bug lists and fix obvious errors. Re-document if necessary.

4 The 'Virtual Sound Environment' Model

As highlighted earlier, projection of pre-composed works represents one method of sound spatialization. The M2 / Super Diffuse project represents a proven live sound projection system. It is a logical progression from analogue or hardware based sound diffusion. However, it does not aim to reproduce realistic spatial sound effects and therefore, this thesis now addresses the concept of spatial sound simulation.

The goal of the 'Virtual Sound Environment' project is to create a three dimensional physical modelling system for sound that can simulate many real world situations while still being flexible enough to allow creative experimentation with sound. In order to provide the necessary flexibility a conceptual framework for spatial sound processing is also proposed.

4.1 Requirements of a model

A model of a real system is precisely as stated; a model, not the real thing. In other words the model can never be perfect unless it is the real thing. When attempting to recreate the physical properties of sound it is quickly apparent that it will not be possible to recreate every subtle nuance in any model.

The purpose of a model is to describe a system in sufficient detail for meaningful experimentation to be achieved. With this done the model can be used to produce practical results that would otherwise be difficult or impossible to obtain from experimentation in the real world.

'A model is a simplification of reality. [61]

Obviously, 'results' from a model of sound physics are expected to sound realistic and, regardless of purpose, the closer to achieving realism the better the model. In

[61] Booch, G; Rumbaugh, J; Jacobson, I: 2003, p.6

a laboratory setting it may be necessary to obtain very precise sound measurements and this would place strict requirements on any model used. However, in the case of artistic and creative use, 'realism' may be sacrificed in favour of more personal, qualitative properties of a sound. Therefore, requirements for a creative tool would aim to promote experimentation and composer feedback.

> *'PhM[62] Synthesis methods do not attempt to create a "complete" physical model of an instrument. Rather than accounting for all possible conditions of the instrument's existence, they need only to account for the physics of an instrument in the highly constrained situation of performance.'*[63]

Roads explains that the required accuracy in modelling is related only to the needs of the given situation, in this case modelling an instrument for the purpose of sound creation. It is conceivable that the same is true when modelling sound propagation.

Poli and Rocchesso describe the use of physical sound models as necessary to overcome the *'...slavery to "frozen" sounds.'*[64], preferring PhM's interactivity and direct control.

Artistic process is not the subject of this thesis; however, it is clear and relevant that a creative user of a sound model is concerned with producing works of artistic merit and not scientific accuracy[65]. For this reason, the model created here is carefully designed to be capable of producing desirable results for the artist. Roads puts forward some desirable qualities of physically modelled instruments.

> *'Simulation by physical models can create sounds of fanciful instruments that would otherwise be impossible to build. In this*

---

[62] PhM- Physical Modelling

[63] Roads, C: 1996, p.266

[64] Poli, G; Rocchesso, D: 1998

[65] Wishart, T: 1994, p.5

> *category we can include phantasmagorical instruments whose characteristics and geometry can change over time* [66]

Clearly this creative experimentation with the parameters of physical modelling is not limited to the modelling of instruments and can be applied to spatialization of sound.

When compromises in modelling are inevitably made, due to practical constraints, this project favours creative possibilities rather than precise modelling of physical properties. Poli and Rocchesso describe the inherent computational constraints of real-time spatialization models[67]. Malham highlights computational penalties for precise modelling and describes simplified methods such as ray tracing as being realistic enough for human perception[68]. As an example of computational problems, the decision to make the model calculable in real-time has affected almost every algorithm used in the project. However, the use of real-time processing is perhaps one of the most desirable features to some composers and essential to the live performer. Wishart highlights the usefulness of real-time audio manipulation for the studio composer when attempting to provide elements of performance:

> *'...the success of studio produced sound-art depends on the fusion of the roles of composer and performer in the studio situation. For this to work effectively, real-time processing (wherever this is feasible) is a desirable goal.* [69]

4.2 A simple sound environment model.

Consider a solo violin performance to a small audience. As the soloist plays, the violin's vibrating strings cause very small changes in air pressure. The air

---

[66] Roads, C: 1996, p.266

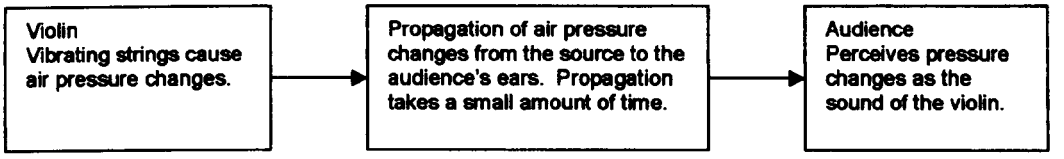
[67] Poli, G; Rocchesso, D: 1998

[68] Malham, D. G: 1998

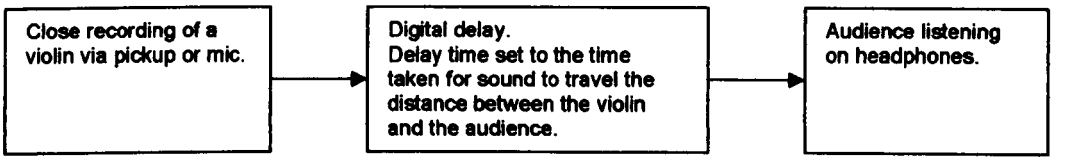
[69] Wishart, T: 1994, p.8

pressure changes, if occurring at a frequency within the human hearing range, are perceived by a nearby audience as the sound of the violin.

Taking the above description and condensing it into a simple flow diagram, the system can be described as follows:



It is possible to take the descriptive diagram above and use it to create a digital process that simulates the propagation of the sound through air. This could be described as a computer based physical model of the sound propagation. It would be possible to capture the direct sound of the violin strings using a pickup and play it back directly into the audience's ears via headphones. Inserting the digital sound propagation simulation between the pickup and the audience would produce a simulated output of the propagation effects. Putting this concept in diagrammatical form produces the following:



The model is not complex enough capture every nuance of the performance space but this simple model does describe one aspect of the real situation quite well. The distance between the violin and the audience is described by the delay and this is quite accurate. The real world delay time between the violin and audience is calculated with the following equation[70]:

$$t_{delay} = \frac{|p_1 - p_2|}{v_s}$$

where:

[70] Smith, J.O: 2002.

v is the speed of sound in air and p1, p2 are vector positions of the violin and audience relative to a single point

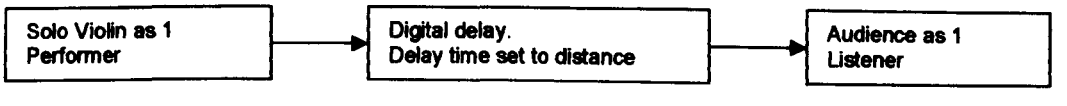
This formula results in a time that can be used in a simple delay line to produce some effects of distance on a sound wave propagating through air.

Of course this is still a long way from a convincing model of the performance space. Many factors have been left out: How does the room affect the sound waves? Where are the audience in relation to the room and the soloist? Is the soloist moving? Is the audience moving or looking in the right direction?

To improve on the model, more generic terms will be used to further simplify the diagram. The soloist and violin can be grouped together and termed a 'Performer', i.e. a 'Performer' defined as a single sound producing entity. For example, an electric guitarist playing through an amplifier could be considered one 'Performer'. As the generic model is built, other terms will be introduced to describe a single point of sound emanating from a performer. In the case of the guitarist it could be said that there are two sound sources, one from the direct sound of the strings and a second from the amplified sound of the amplifier.

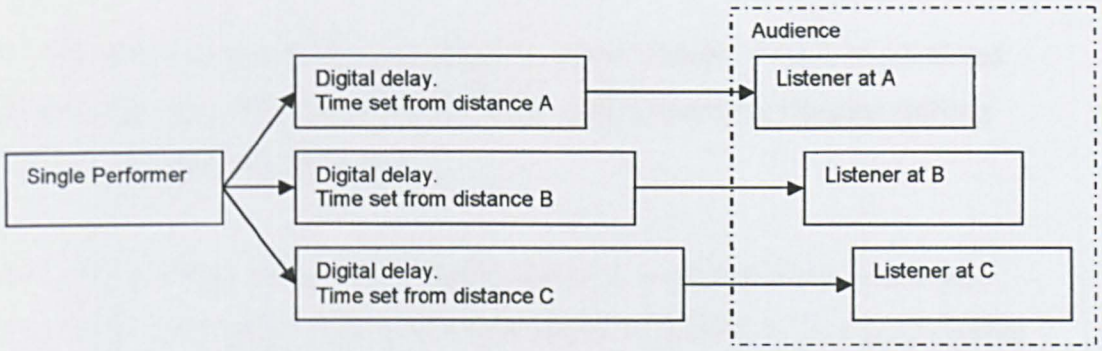
Instead of describing the audience as a group it would be advantageous to consider single listeners to the system, i.e. 'Listener' defined as a single listening entity. For example, a single person listening to a performance would be considered one 'Listener'. A single microphone recording a performance is also one 'Listener'. As with the 'Performer', further terminology will be introduced to describe individual listening areas.

Using the new terms in the basic model produces the following:

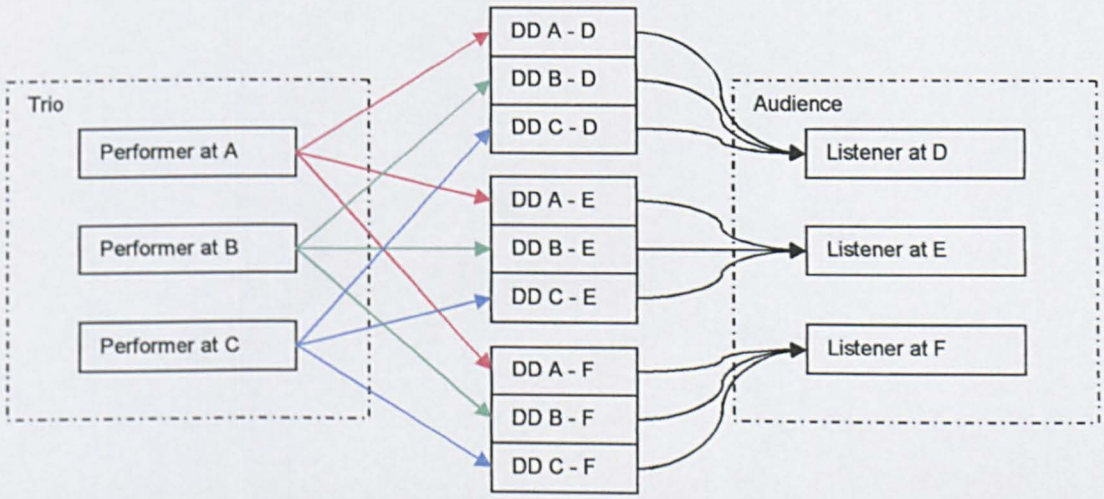


4.3 Improving the model:

In the above model the audience was considered as a single entity, which is of course a gross simplification if the audience is bigger than one person. Every person listening is likely to be located at a different distance from the performer. The model can be adjusted to compensate for this and at the same time use the more generic terminology 'Listener'. Taking a single performer and an audience of three seated at distances A,B and C produces this diagram:



How does the model change when considering more than one 'Performer'? The next diagram expands the above model to include a trio of performers. The digital delay process will be represented by 'DD x - y' where DD is digital delay x and y are start and end locations for the calculated delay time. The letters A,B,C will represent performer locations and D,E,F describe listener locations.

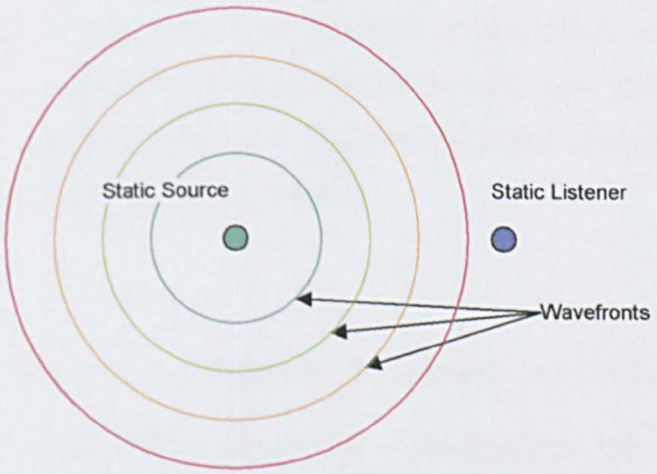


Rapidly, the number of delays needed to describe these basic interactions has become larger. It should also be noted that this is still not a precise model of the real system. However, it is the basis for a much more realistic model that takes

into account the effects of independent performers and listeners. Assuming that the model above calculated the transmission and room effects realistically for each listener then the system could recreate a sound from any point in a theoretical environment with any number of performers. If the system were to be processed in real time it would be possible to move both listener and performer locations within the 'Virtual' Environment and hear the effects immediately.

It is interesting to note at this point that the above 'simple' model, if calculated using digital delay with variable delay time, will produce the Doppler shifting effects associated with velocity.

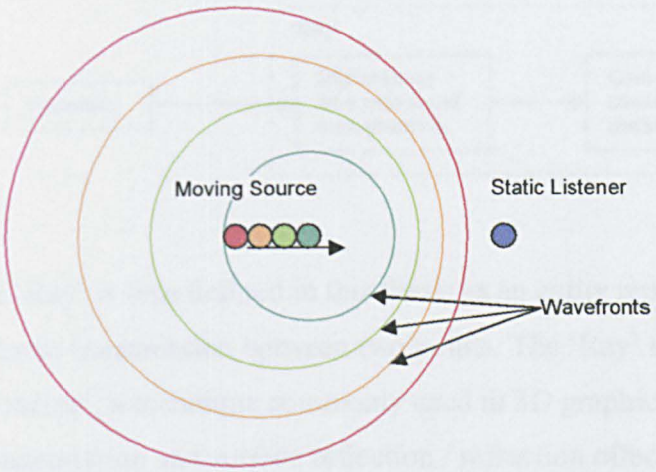
The Doppler effect is perceived in audio waves as a pitch increase or decrease dependent on the radial velocity of sound source to listener. It is an aural cue for speed and therefore adds realism to this 'animated' sound model. The classic example of the Doppler effect is the siren on a police car driving past, with increased pitch as the siren moves towards you then decreased as it moves away[71]. As a sound producing object moves relative to a listener the increasing or decreasing distance causes the wavelength of the sound to be stretched or compressed.



The above diagram shows that the wavefronts emitted from the sound generated source are evenly spaced due to the source remaining statically located. When the

[71] Roads, C: 1996, pp.463-466; Serway, R. A: 1996, pp.487-491

source moves the wavefronts become compressed and thus the wavelength is shorter from the perspective of the listener. See diagram below:



Looking back to the last model, moving a performer location in real time will cause a simulated Doppler shift due to the delay time changing. This works in much the same way as it would in the real world. With the delay line, the same stretching and compressing of wavelength is caused when the delay's read position moves relative to the write position in order to change the delay time.

With the present model of transmission a delay line represents the time taken for a sound to travel between locations. As distance increases sound takes longer to propagate from performer to listener and it also decreases in amplitude. The amplitude loss is due to the spherical nature of sound waves emanating from a point source[72] and is given by:

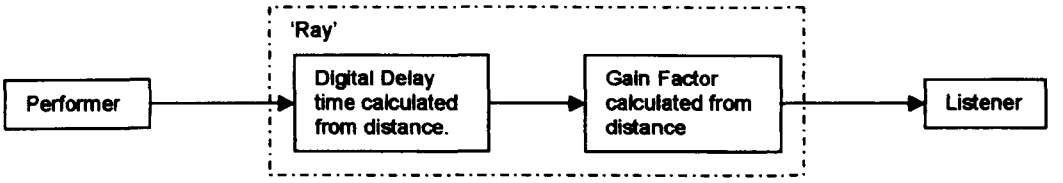
$$i = \frac{1}{d^2}$$

Where: i = sound intensity factor and d = radial distance from the source.

Incorporating the spherical sound propagation concept into the sound transmission model using a gain factor adjusted according to distance produces a more realistic result. As performers are moved away the distance affects both the arrival time of the sound and its amplitude. The term 'Ray' is now used to describe the whole

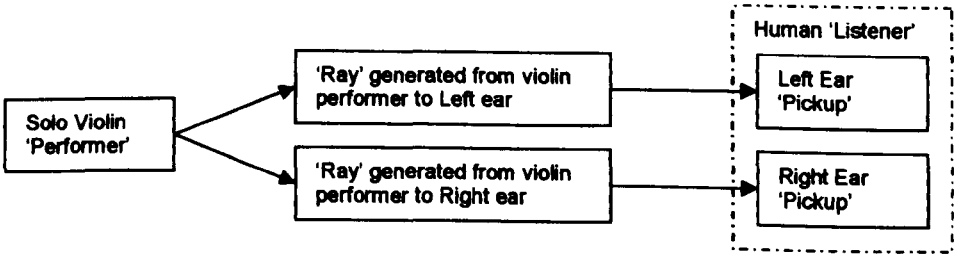
[72] Serway, R. A: pp.484-485; Everest, F. A: 1994, pp.68-71; Smith, J.O: 2002

transmission model used for the system, in this case the combined effects of the delay and the distance gain factor.



A 'Ray' is thus defined in this thesis as an entity representing the effects on sound due to transmission between two points. The 'Ray' term was borrowed from 'Ray Tracing', a technique commonly used in 3D graphics to describe light transmission and surface reflection / refraction effects that also provided some inspiration for this modelling technique[73].

The model of a listener is reasonable but it fails to describe a human being in any great detail as humans have two independent listening organs, the left and right ears. It would be simple to use two listeners to describe one person's ears but this quickly becomes confusing when more than one listener is involved. A better technique is to use the concept of 'Pickups' acting as individual monophonic listening points and to use the 'Listener' term as a convenient logical grouping. A 'Pickup' is defined as a single monophonic omni-directional sound listening point. A 'Listener' is now redefined as a logical container of 'Pickups'. Using these terms in a model describing one human listener and one violin performer produces the following:

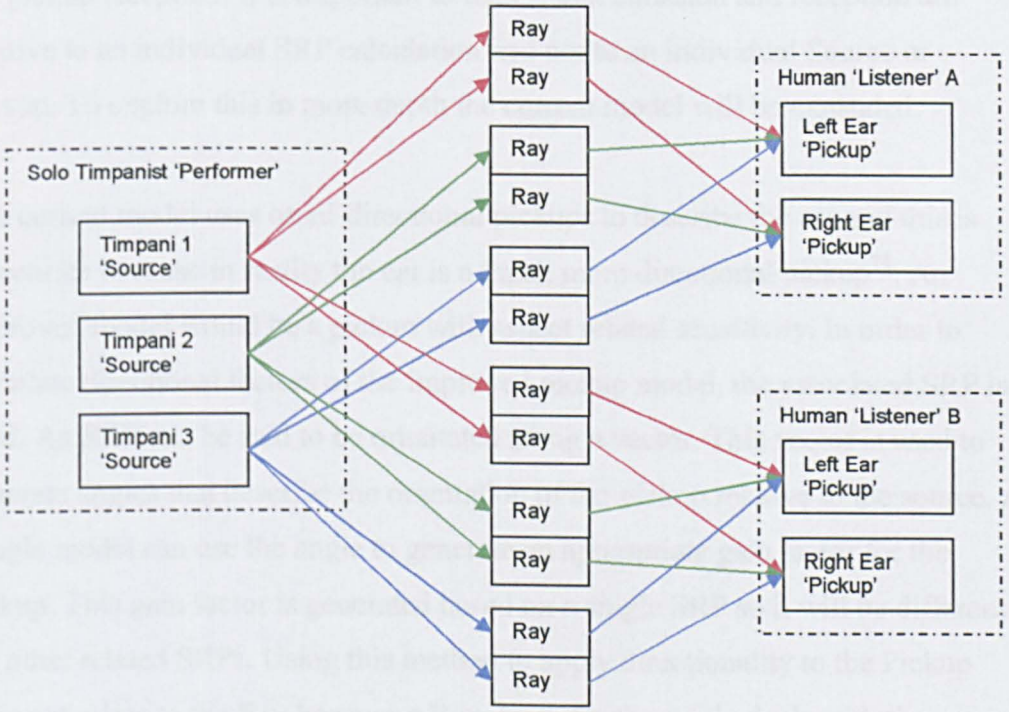


It is important to note here that there are two rays used in this system because each ear is located at a different position. Having two positions means there will

[73] 'Ray' also used in Serway, R. A: 1996, pp.484-485

be two calculated distances and so there must be two rays to perform the processing. In the above case every additional human listener would require an additional two rays to be calculated.

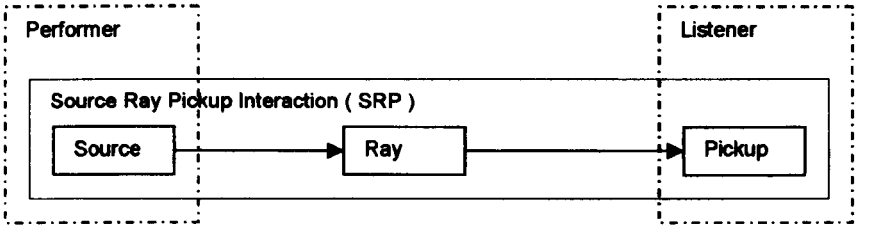
Logically the next step is to consider the 'Performer' as a practical grouping of 'Sources'. This concept becomes crucial when looking at more complex instruments: for example a single percussionist will have many percussion instruments located in different positions. Taking this into account, 'Sources' can be used to describe sound generating points on an instrument or a number of instruments. A guitarist with separate amplification has already been considered as a case for this terminology. A 'Source' is defined as a point from which sound is emitted omni-directionally. The term 'Performer' is redefined as a logical container of 'Sources'. Using this new terminology the following diagram looks at a timpanist's performance. In the example the timpanist uses three timpani, and there are two human listeners A and B.



From this diagram it is clear that every Source is connected to all Pickups via Rays. To describe this connection the term 'Source Ray Pickup Interaction' or 'SRP' is introduced. The number of SRPs in a system is given by the number of Sources multiplied by the number of Pickups.

$$SRP_{total} = pickup_{total} \times source_{total}$$

An SRP can be defined as the process from which to determine sound arriving at a single Pickup from a single Source.



At this point it is necessary to demonstrate the differences between a Ray and an SRP. A Ray describes only the effects of transmission of the sound. An SRP includes the transmission but also adds the properties of the source emission and the pickup reception. It is important to realise that emission and reception are relative to an individual SRP calculation and not to an individual Source or Pickup. To explore this in more depth the current model will be expanded.

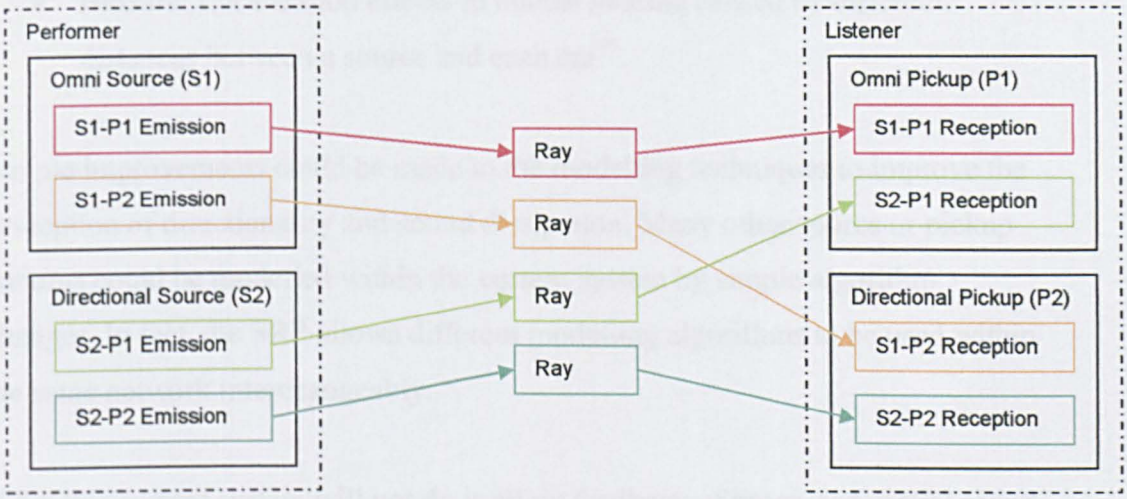
The current model uses omni-directional pickups to describe the ear and this is inaccurate because in reality the ear is a much more directional pickup[74]. An improved model would be a pickup with aspect related sensitivity. In order to calculate directional factors of the improved pickup model, the associated SRP is used. An SRP can be said to be orientated along a vector. This vector is used to generate angles that describe the orientation of the pickup relative to the source. A simple model can use the angle to generate an appropriate gain factor for the pickup. This gain factor is generated based on a single SRP so it will be different for other related SRPs. Using this method to apply directionality to the Pickup does not relate to the Ray because a Ray, by definition, only deals with the transmission of the sound. It is the Pickup which 'receives' the sound 'arriving' at its position. The advantage of using both the Ray model and the SRP is that the

[74] Everest, F.A: 1994, pp.51-53

model describing sound transmission (Ray) is kept independent of the model describing reception of the sound (Pickup).

Further advantages of the SRP model are found when considering directional Sources. A trumpet or speaker are very directional sources and an aspect related gain function similar to that of the pickup can be used to simulate directionality. Again taking the SRP orientation vector and using it to generate a gain factor, an effect is produced that controls the amplitude of the sound passed to the Ray. As with the Pickup, the modelling of orientation is not related to the transmission model so the source modelling function is packaged independently of the Ray.

The following diagram highlights the use of directional Sources and Pickups and demonstrates the concept of the SRP using a Performer that contains both omni and directional Sources with a Listener containing omni and directional Pickups. Individual SRPs are shown with individual colour codes.



The diagram shows how the SRP allows independent modelling methods for Source, Ray and Pickup. These modelling methods are independent from each other i.e. a directional source can be connected to an omni pickup without additional functions being needed.

The independence of modelling functions is important because it allows Source models to be developed without thinking about all possible pickup models. The reverse also applies; Pickup models can be developed without consideration of Source models. Perhaps less obviously, it is also possible to have independent Ray

models used within the same network. One use of this technique would be to allow high accuracy transmission models to be used in critical areas and low accuracy to be used is less critical areas. This possibility has interesting implications for processing optimisation.

4.4 Reviewing the basic SRP model.

The current SRP based network for performers and listeners is able to model the following functions:

- The effect on sound arrival time caused by distance.
- The dissipation of sound energy caused by distance.
- Directional or omni-directional sound sources.
- Directional or omni-directional sound pickups.
- Doppler Shift effects caused by moving sources or listeners.
- Binaural Localization effects in human hearing caused by different distances between a source and each ear[75].

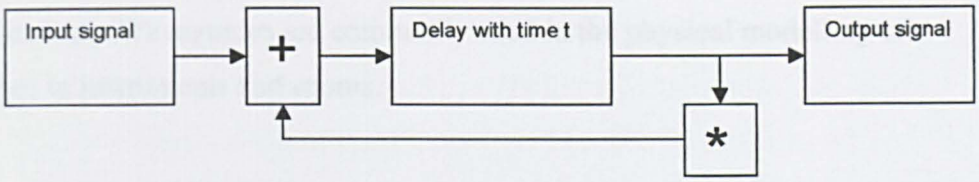
Simple improvements could be made to the modelling techniques to improve the perception of directionality and sound dissipation. Many other source or pickup patterns could be modelled within the current system by simple algorithm changes. In fact, the SRP allows different modelling algorithms to be used within the same network interchangeably.

What this current system will not do is allow feedback of processed spatial sound back into the same system. Allowing for feedback brings the possibility of echo and reverberation. Using small delay values with feedback allows for resonating objects and is the subject of the following sections.

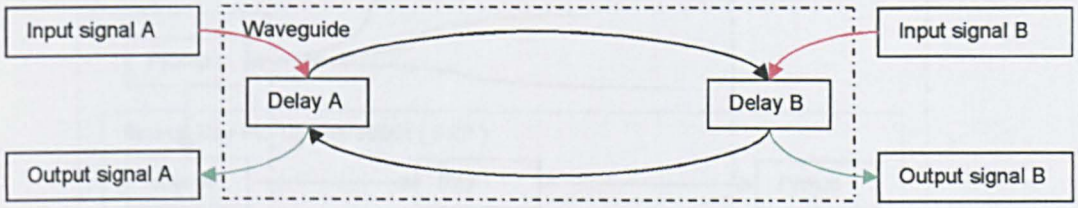
4.5 Adding feedback:

[75] Everest, F. A: 1994, p.54

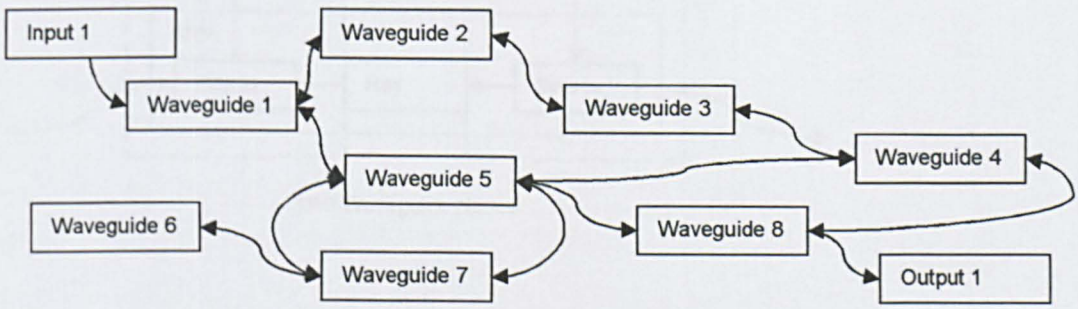
This diagram describes a simple delay line with feedback:



The multiplication symbol represents a gain factor that can be used to control the feedback. The addition symbol represents summation of input and feedback. Positive gain factors < 1 will produce stable feedback with exponential decay in amplitude. Gain factors > 1 will produce unstable feedback with an exponential increase in amplitude. A gain factor of 1 will produce infinite delay feedback. Taking two delay lines and connecting them such that they feedback into each other creates a building block known as a waveguide[76].



Each delay line feeds the other after applying some processing. The processing could be a simple gain but it could also filter the signal in some way. Each delay line can be excited by an incoming signal as well as outputting the signal elsewhere. The delay lines' inputs and outputs form the 'ends' of the waveguide and are termed 'nodes'. Using these waveguide building blocks it is possible to build up a waveguide network that simulates a resonating system by connecting the nodes from many waveguides.

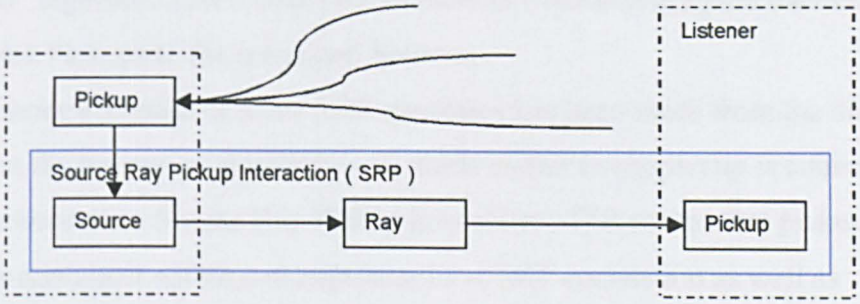


The node connections are bi-directional but with control of feedback gain and/or

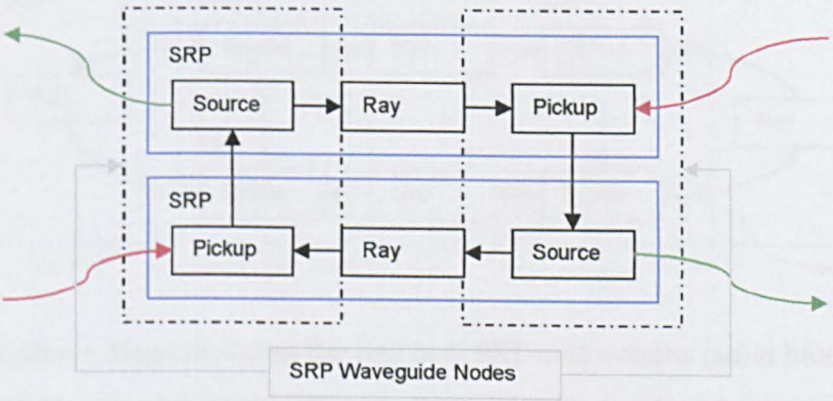
[76] Roads, C: 1996, p.282; See also general references: Smith, J. O: 2004

filtering. A very simple waveguide network can produce a very complex resonating output. Combinations of delay times will produce complex filtering and reflections. Waveguides are commonly used in the physical modelling of resonance in instruments and rooms.

It is possible to build the feedback concept into the SRP model and thus allow for the creation of SRP based waveguide networks. One way for this to be achieved is to allow for a Pickup to feed its 'results' back into a Source. This technique creates a single feedback path that can be considered as half of a single waveguide building block. Adding a second feedback path by performing the same operation twice results in an SRP based waveguide. The following diagrams illustrate this method using the terms defined earlier:



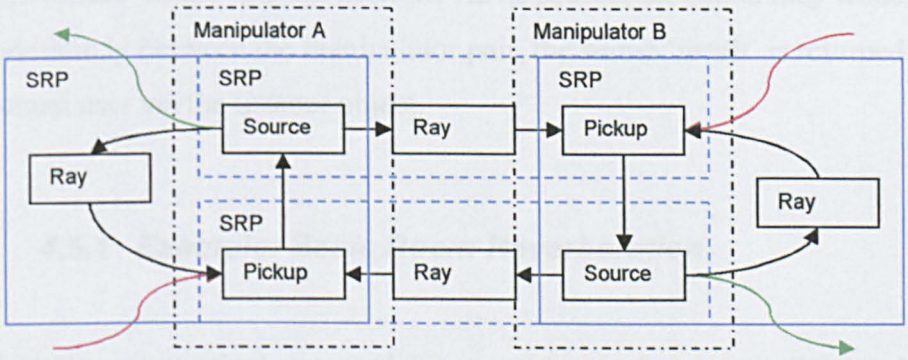
In the above a Pickup's incoming sound is fed forward into a Source. This Source is then considered in the Listener's related SRP.



In the above diagram a waveguide structure has been built using two SRPs. At this point the similarity in structure of the SRP waveguide and the delay based waveguide should be noted. Like the basic waveguide the SRP version also has

two nodes. However, the SRP waveguide is more advanced than the basic structure as it inherently contains the features of pickups and sources. This means that the waveguide nodes can be designed based on the more advanced sources and pickups that feature directionality. In the SRP waveguide model the direct pickup to source transactions form the point at which additional feedback algorithms can be applied. It is possible that a feedback algorithm may simply 'copy' the pickup's sound perspective into the source's outgoing transmission. In this case, feedback is still controlled by both the pickup response and source emission algorithms. Of course filtering could be applied in addition to the effects of pickup and source. This 'copying' algorithm from pickup to source needs a placeholder and for this the term 'manipulator' is put forward. A 'manipulator' is defined as a container of both sources and pickups, it contains a 'copy' or 'transfer' algorithm that dictates the method of transferring sound data from the contained Pickups to the contained Sources.

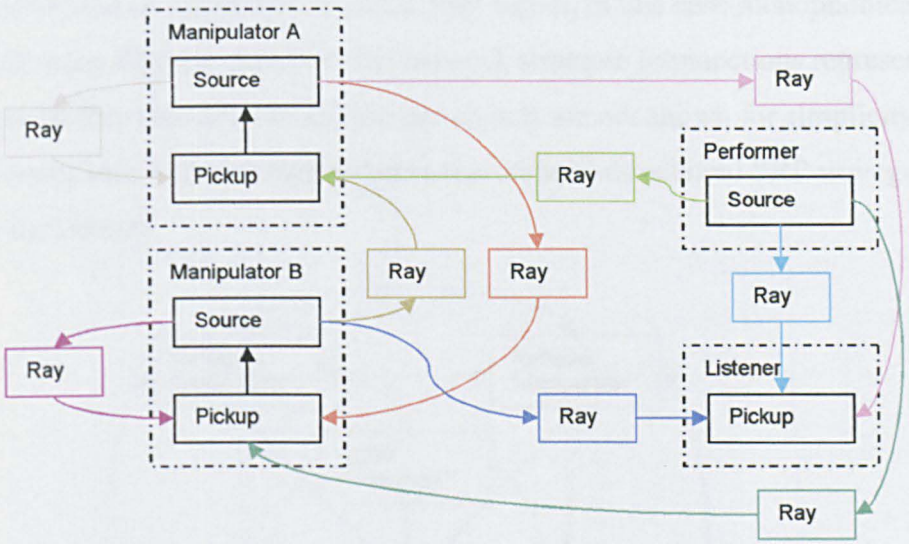
A deliberate omission of some SRP structures has been made from the diagram above in the interest of simplicity. As stated earlier every pickup is connected to every source via a Source Ray Pickup Interaction. This means that pickups and sources contained within a manipulator have SRP connection as well as 'copy/transfer' connection. The following diagram fixes the problem and introduces the new terms:



The above diagram shows the two new SRP connections (solid blue boxes). It should be noted from this diagram that a new feedback path now exists internally within the manipulator. This internal feedback path can be undesirable in certain cases. For example, an omni source and pickup could exist at the same location and the copy/transfer function may be a simple direct copy operation. Due to the zero time delay caused by zero distance between source and pickup the resulting

feedback loop would have no energy loss, and is thus unstable. It is useful to provide possible disconnection of the returning SRP, breaking the feedback loop and allowing special case objects to be considered.

The following diagram illustrates the concepts looked at in the SRP model and describes performer, listener and manipulator objects acting as an SRP waveguide.

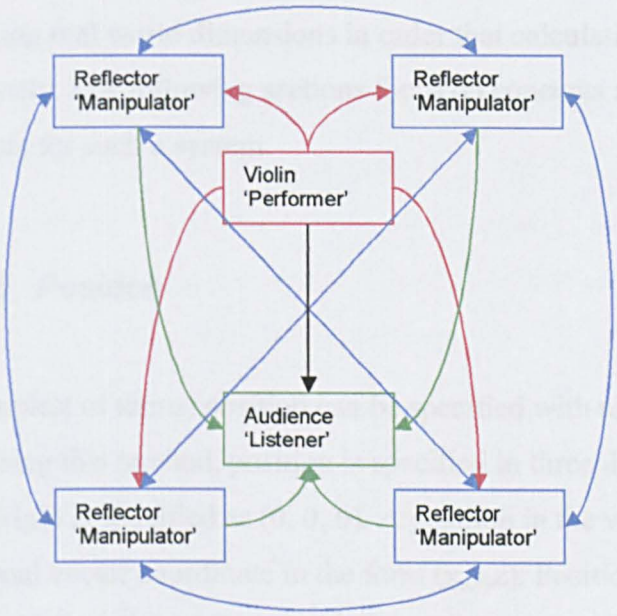


The network above shows the potential of the SRP method. It contains the basic building blocks of a complex spatial model. A single performer object 'introduces' sound into the network via its source, the sound may feedback indefinitely between the manipulator pair, the sound 'result' is returned to the human user via the listener object.

4.5.1 Example: Basic Room Reverberation

Consider a theoretical rectangular room with a performing violinist and an audience of one. A simple model is created easily with the SRP network. Taking a manipulator in which source and pickup exist at the same point, eliminating the self referential feedback (as discussed above), the manipulator can be considered a perfect spherical reflector of sound. This object will now be named a Reflector. Four of these reflectors are placed at the centres of the four walls of the room to be modelled, one reflector per wall. This creates an SRP waveguide network

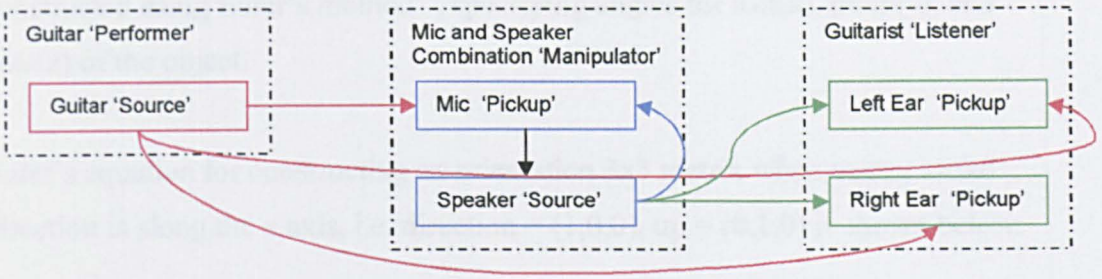
simulating basic room reflections in which the walls are perfect reflectors of sound (note that the ceiling and floor have been intentionally left out to simplify the model). Designing a 'copy/transfer' algorithm that dissipates some of the energy at each 'wall' reflector creates a more precise network. Adjusting the amount of energy reduction simulates different types of surface. Using a performer with one source allows sound energy (from the violin) to be injected into the virtual room and a listener with one pickup allows extraction and conversion of sound into a 'real world' signal, in this case monophonically. The following diagram describes the network structure (connections represent Source Ray Pickup Interactions and the ray objects are not shown for simplicity. Connections between manipulators represent bi-directional SRP waveguide connections).



4.5.2 *Example: Amplified Acoustic Guitar*

Consider an acoustic guitarist rehearsing through a microphone and amplified speaker, with room effects ignored in this case. The guitarist can be thought of as the only listener, so he/she is modelled with a single Listener with two pickups (left and right ears). A microphone and amplifier can be modelled as a single manipulator, one directional pickup for the microphone and a directional source

for the speaker. The body sound of the guitar is a performer with one source. The network produced is as follows:



4.6 Representing the world

Sound processing in SRP networks is performed using positions and orientations of sound objects as direct variables in the processing algorithms. It is vital then that any three dimensional spatialization system uses an appropriate method for representing real world dimensions in order that calculations are performed easily and precisely. The following sections focus on concepts and techniques appropriate for such a system.

4.6.1 Position

In the simplest of terms, position can be specified with reference to a single origin. Using this method, position is specified in three dimensions labelled x, y, z and the origin is specified as (0, 0, 0). A position in the world is given as a three dimensional vector coordinate in the form (x,y,z). Position could also be called a 'translation' from the origin by a vector quantity.

4.6.2 Orientation

For orientation of an entity relative to the world there are a number of useful representations. Three will be discussed.

A 3D normalised vector (a vector with magnitude of 1) can signify the direction an object is pointing, with a second vector required to specify rotation about the first vector. This second vector is sometimes called the 'up vector'.

A second representation makes use of a 3x3 rotation matrix. The matrix is constructed using Euler's method[77], specifying angles for roll(x), pitch(y), and yaw(z) of the object.

Euler's equation for constructing an orientation 3x3 matrix when vector initial direction is along the x axis, i.e. direction = (1,0,0), up = (0,1,0) is shown below:

$$M_{orientation} = \begin{bmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) \\ 0 & 1 & 0 \\ -\sin(\theta_y) & 0 & \cos(\theta_y) \end{bmatrix}_{yaw} \begin{bmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 \\ \sin(\theta_z) & \cos(\theta_z) & 0 \\ 0 & 0 & 1 \end{bmatrix}_{pitch} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) \\ 0 & \sin(\theta_x) & \cos(\theta_x) \end{bmatrix}_{roll}$$

Matrix multiplication is not commutative so the order in which rotation transformations are applied is very important, i.e. $M_a M_b \neq M_b M_a$. A problem associated with rotational matrices occurs when animating a rotation using repeated matrix multiplications, compound floating point accuracy errors can eventually cause the rotation matrix to become combined with unpredictable skewing/scaling/translating transformations.

A less well known orientation method makes use of a mathematical entity called a Quaternion developed by William Hamilton in 1843, during his investigations into complex mathematics[78]. A Quaternion is a 4D entity in the form (n, v_{xyz}) which can be used to represent a complete orientation. Notice here that only 4 terms are required to store a complete orientation as opposed to the 9 used in a 3x3 matrix. A quaternion can be constructed from a specified rotation angle around an arbitrary axis defined by a unit vector:

$$q = \left(\cos\left(\theta/2\right), \quad \sin\left(\theta/2\right)v\right)$$[79]

---

[77] Eberly, D. H: 2001, p.18; Lengyel, E: 2002, p.60

[78] Akenine-Möller, T; Haines, E: 2002, p.44

[79] Bourg, D. M: 2002, p.228

Quaternions can be multiplied by the equation:

$$qp = \left(n_q n_p - v_q.v_p, \quad n_q v_p + n_p v_q + \left(v_q \times v_p\right)\right)^{80}$$

As with matrix multiplication, quaternion multiplication is not commutative[81], i.e. $Q_a Q_b \neq Q_b Q_a$

It is possible to build a quaternion in a similar manner to the rotational matrix Euler method by constructing quaternions aligned to axes and multiplying the resulting quaternion rotations.

Regardless of the representation used, an object's orientation can be called its rotational transform.

4.6.3 Scale

In addition to specifying position and orientation it is useful to specify scaling transforms. Scale can be represented as a uniform scaling factor s_u where s_u scales all vector components equally or by a non-uniform scale vector in the form (s_x, s_y, s_z). A non-uniform scale represents independent scaling factors for each component of a vector.

Other transforms such as skewing or shearing are less relevant to the task of representing the world and so are not discussed here.

4.6.4 SRT Transforms

Three transform types are the most commonly used: translation, rotation and scale. These types can be combined to form a single entity representing the

---

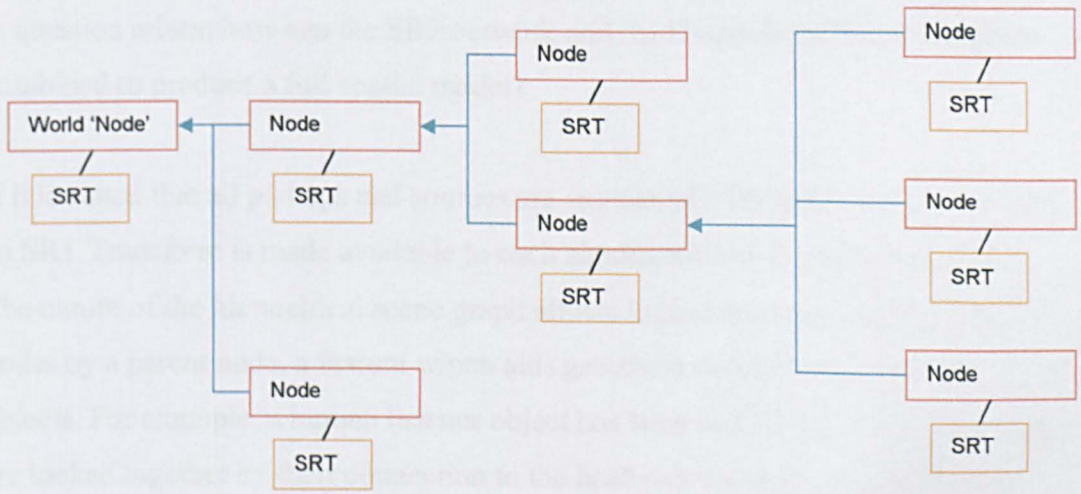
[80] ibid., p.307

[81] Lengyel, E: 2002, p.68

position, orientation and scaling of an object. This entity is commonly referred to as the 'Scale Rotate Translate Transform' or 'SRT'[82]. SRT transformations can be implemented as a 4x4 matrix but it is more efficient in 3D modelling to use a split representation with scale and translation as 3D vectors and a single quaternion to represent rotation. Using this technique allows a considerably simpler calculation of the inverse SRT and further benefits, including reduced effects from compound errors in repeated matrix rotation transformations. An SRT becomes a particularly useful building block in a hierarchical world object model because it can be implemented to behave like a matrix transformation at a fraction of the processing cost.

4.6.5 Hierarchical Scene Graphing

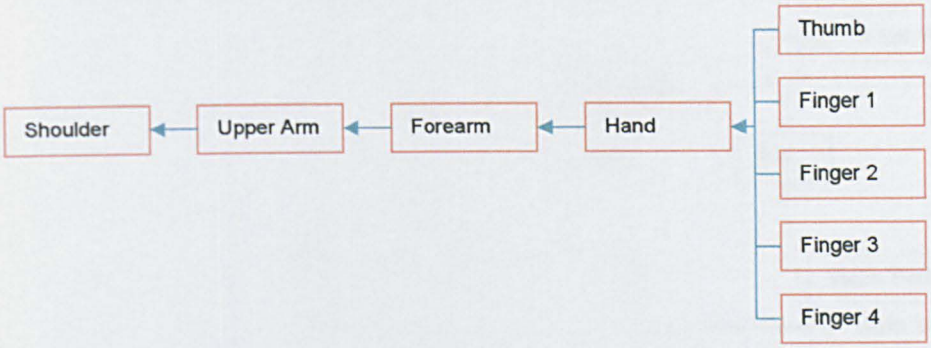
Many 3D modelling systems use a hierarchical tree structure to represent complex transforms involving compound objects. This structure is often called a Hierarchical Scene Graph[83]. The structure is made up of 'nodes' and the first node is the root. Each node can have any number of child nodes. Each node in the tree is paired with an SRT. Instead of using the SRT as a transform relative to the world, the scene graph allows the SRT to transform an object relative to its parent node in the tree. If a parent node's SRT transform is adjusted, the child node's SRTs now represent the transform relative to the new parent node transform.



[82] Eberly, D. H: 2001, p.144

[83] ibid., pp.141-167; Akenine-Möller, T; Haines, E: 2002, pp.346-357

In order to clarify the SRT terminology it is useful to examine a simple model; the following diagram describes possible nodes in a theoretical model of a human arm. All nodes are assumed to be connected with associated SRT structures.



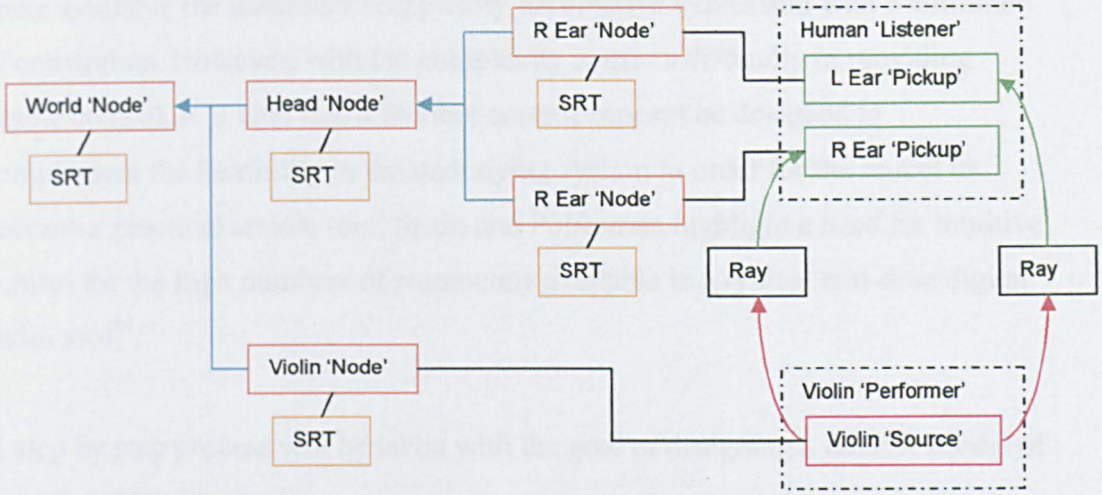
Note that adjustment of the 'shoulder' node's SRT will recursively affect nodes lower in the hierarchy so all nodes are transformed accordingly. Adjusting the hand node has the same recursive effect but only 'thumb' and 'finger' nodes are altered.

The SRT in a scene graph provides a simple method of mapping vectors between coordinate systems. It is possible to traverse the scene graph's tree and use each node's SRT to transform the vector at each stage. Similarly a single SRT that performs direct translation between two coordinate systems on the graph is obtained by traversing the tree and multiplying the SRT Transforms.

A question arises; how can the SRP network and the Hierarchical Scene Graph be combined to produce a full spatial model?

If it is stated that all pickups and sources are associated with individual nodes then an SRT Transform is made available to each significant sound processing object. The nature of the hierarchical scene graph allows logical spatial groupings of nodes by a parent node, a feature which aids greatly in control of compound objects. For example, a human listener object has been considered; a person's ears are locked together by their connection to the head and the ears are modelled by two directional pickups. By grouping nodes associated with each pickup into a 'head' parent node a logical spatial grouping is made which facilitates spatial control. Transform the 'head' node and the pickup nodes are transformed along

with it. This diagram shows the integration SRP Network with that of the scene graph. A human listener and single violin performer are shown.



Association between a pickup and node provides a useful mechanism for storing the transformation data. It is worth noting here that a single Source Ray Pickup interaction can obtain transformation data by traversing the scene graph loop, created between a pickup and source. Conveniently the scene graph traversal can be used to provide positions and orientations of pickups relative to one another. For example, in the above the 'Right Ear Pickup' can obtain the position and orientation of the 'Violin Source' relative to its own current SRT by successive transform of the initial vector (0,0,0), first down the tree to the 'World' node then up through 'Head' and 'R Ear' nodes. Performing the reverse operation through inversion of the transform can provide the 'Violin' source with the position and orientation of the 'Right Ear' pickup relative to itself.

Another advantage of the Scene Graph is its use when rendering a 3D graphical user interface. Its hierarchical implementation works very conveniently with 3D graphical engines. This will be covered in more depth when looking at the implementation of a software solution.

4.7 Providing Flexible Automation and Control

A theoretical 3D system for sound design has been proposed and as it stands there are structures in place for holding and performing suitable DSP algorithms (SRP

Interaction Network). There is also a flexible structure for detailing the spatial layout of the world (SRT based Hierarchical Scene Graph). The two concepts make available the necessary complexity for creative expression with a minimum of constraints. However, with the complexity comes a difficulty in providing usable control. It is vital that a flexible control concept be designed to complement the flexibility in the underlying system in order for the model to become a practical artistic tool. Spain and Polfreman highlight a need for intuitive control for the high numbers of parameters available in a typical real-time digital audio tool[84].

A step by step process will be taken with the goal of designing a flexible model of spatial and DSP control.

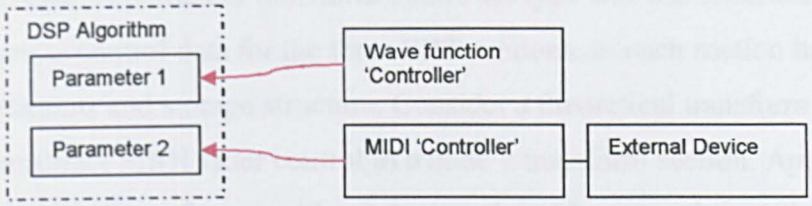
What parameters should be controllable?

If a constraint on the ability to dynamically alter the SRP network and the Scene Graph structure in real-time is assumed, there are only two major controllable parameter areas in the model; control of DSP Parameters that do not relate to the scene graph and control of SRT Transforms within the scene graph. To aid creation of a single spatial model the decision can be made to force all spatially related vector parameters to be nodes in the scene graph. For example, a DSP algorithm requiring a position of reference as a parameter would use a scene graph node to represent that position.

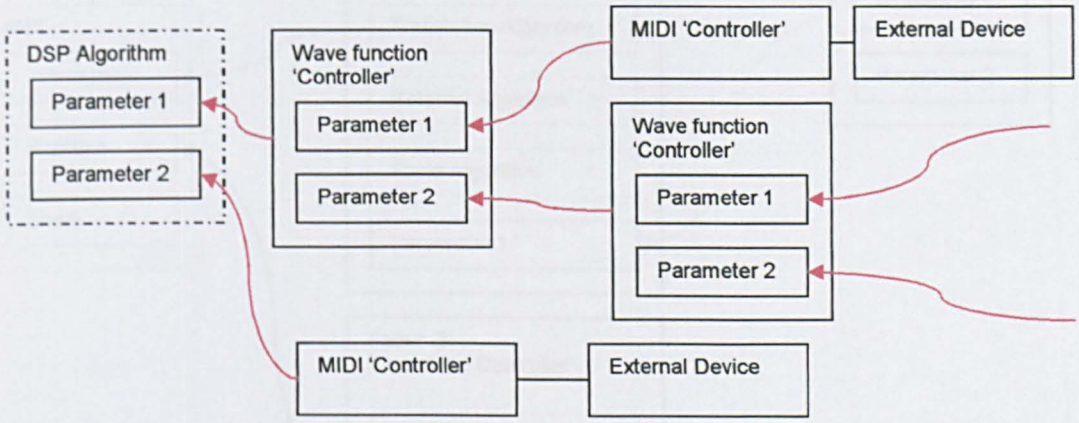
To facilitate control of a parameter the term 'Controller' is introduced. A controller is defined as an entity that can be attached to a parameter in order to provide updated data. The controller concept forms an abstraction layer between a parameter and its method of control. In other words, different classes of controller can be used to control the same parameter. For example, one parameter may be controlled via a wave generating function while another could be controlled by a direct MIDI device.

---

[84] Spain, M; Polfreman, R: 2001



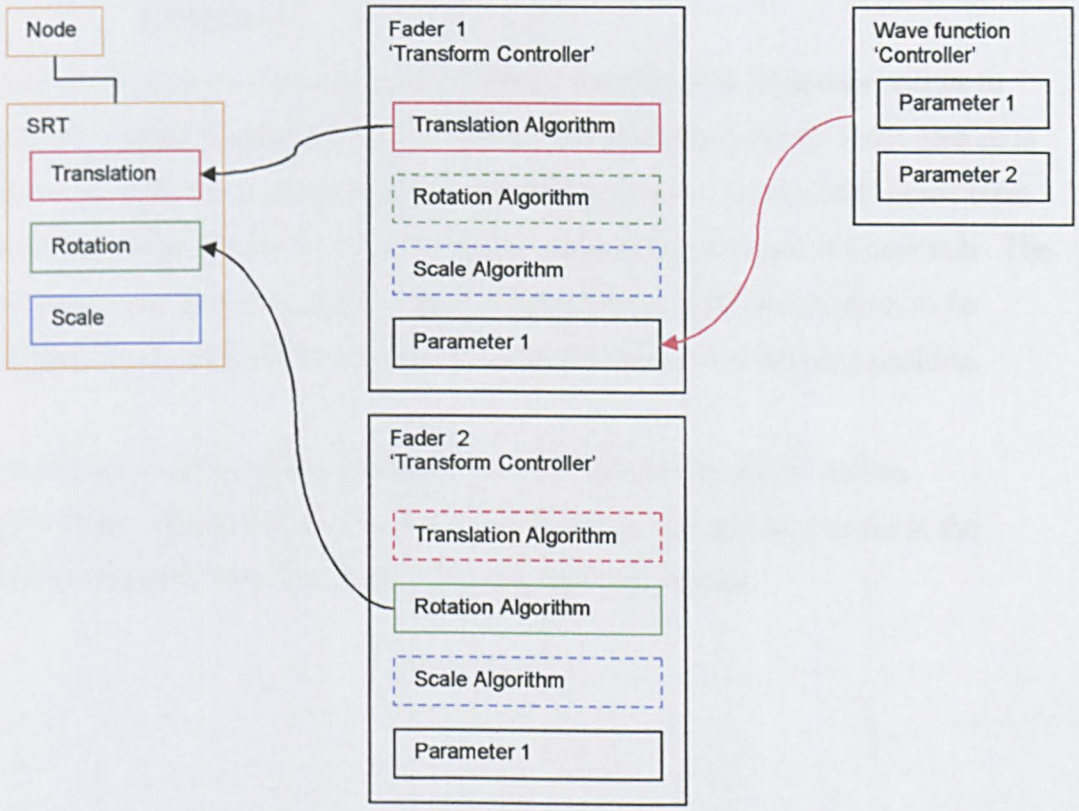
Allowing each parameter in the model to have a single controller attached to it provides a powerful means to control the system. To further the flexibility of control the concept of 'Controller Parameters' are introduced. In the example above, a wave generating controller is used to affect a parameter. By adding parameters to the controller and allowing these parameters to be attached to additional controllers, the control system becomes a hierarchical tree structure. The following diagram shows the hierarchical controller concept by adding two parameters to the 'Wave function' controller type.



It should be noted in the diagram above that the control 'tree' can extend outwards as far as necessary. An implementation of this hierarchical control model would ideally provide utility controllers that allow combining of control data via mathematical or related functions.

Returning to the subject of controlling SRT transforms, the parameters of a single SRT can be broken down into three sections: translation, rotation and scale. Expanding the basic controller entity into a 'Transform controller' entity is a simple way of providing a powerful SRT control method. As with the basic controller the transform controller can optionally contain further parameters, therefore creating a hierarchical control chain. Each SRT can potentially be assigned three separate transform controllers corresponding to position, rotation

and scale. A particular transform controller type will use different algorithms to generate control data for the three SRT sections, as each section has differing constraints and storage structure. Consider a theoretical transform controller that gives direct MIDI fader control to a node's transform section. Applied to position, the fader adjusts the x position between -1 and 1 metre. On rotation it determines the percentage of a full rotation about the y axis. With scale it applies a scaling factor between 0 and 2. The point here is that the controller concept makes sense to a user when attached to any of the sections and the attachment itself determines the appropriate algorithm to use. The following diagram consolidates this concept but it should be noted that the separation of the algorithms would not be necessary in all Transform Controller types.



Describing the transform controller with specific algorithms for each section is less ideal when designing special case controllers that need to control more than one section of the SRT. Consider a theoretical 'Path' transform controller; the controller aims to move a node such that it follows a defined route or path. At all times the node is orientated along the path. The point here is that the controller's algorithm requires it to affect both SRT translation and rotation simultaneously. The most flexible solution is to allow a transform controller to be assigned such

that one controller type may define algorithms for all three sections, but force a single transform section to only have one assigned controller.

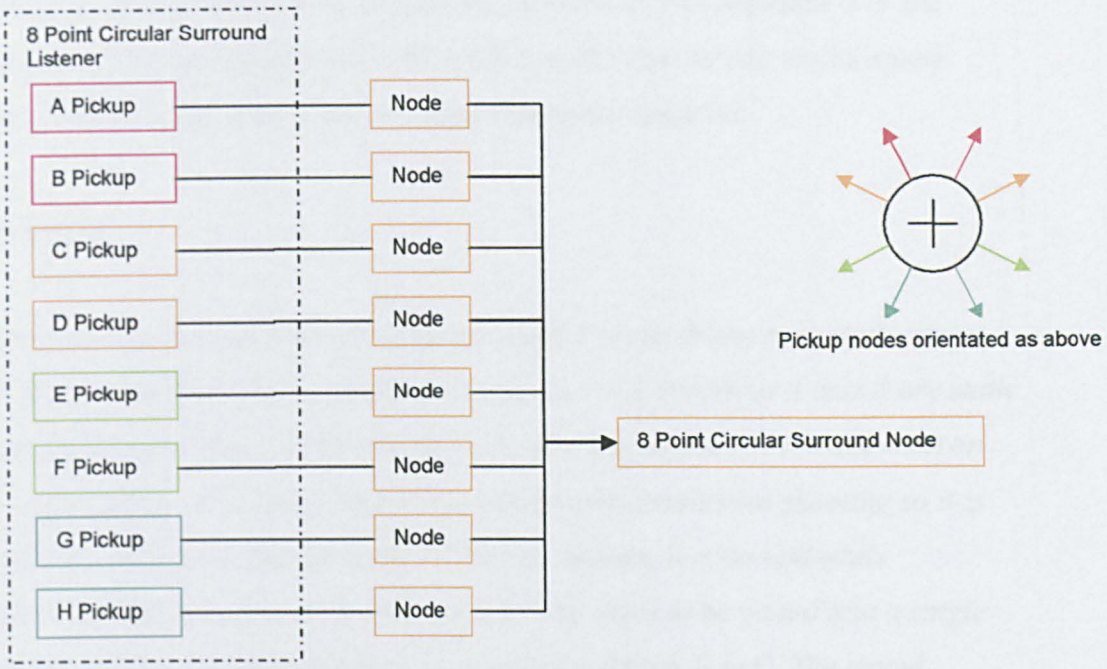
4.8 Consolidating the 'Virtual Sound Environment' model

Two small theoretical case studies will be used to review some of the concepts discussed and illustrate the potential power of a fully implemented system. The case studies are based around a theoretical 'Virtual Environment' model but the principles of use would be similar in a real system.

4.8.1 Case Study: The 'Virtual Sound Environment' model for the composer

Consider a composer of multi-channel works for tape. The composer wishes to generate spatial gestures from monophonic and stereo recordings. Each gesture is to be recorded into a multi-channel sound bite and these 'spatialized' sound bites are to be composited and mixed in a sound sequencing package at a later date. The composer has chosen an eight channel output format with loudspeakers to be located in an evenly distributed circle about the audience's listening position.

The composer has a version of the 'Virtual Sound Environment' and an appropriate 'listener' object has been provided that will produce sound in the desired output format. This object is constructed as follows:



In the above listener, each pickup is assigned to an individual sound output with the channels allocated in the same pairings as coloured above. For each multi-channel sound bite the composer uses file playback performer objects to inject the sound recordings into the virtual environment model. The composer is able to control the positions and aspects of the performers in real-time directly though the interface. Linking a sequencing package via MIDI and using MIDI based transform controllers to move the various nodes in the system provides time-based control. Each sound bite can be recorded to multi-channel tape or even into an audio sequencer directly.

An interesting point here is the ability of the composer to move his or her 'virtual listening point' within the model. The sound bite heard by both the composer and the final audience would of course parallel the virtual listener in the room. Mixing of sound bites can, if so desired, result in the audience hearing a composite of sound perspectives taken from the same modelled environment.

4.8.2 Case Study: The 'Virtual Sound Environment' model in the context of film production.

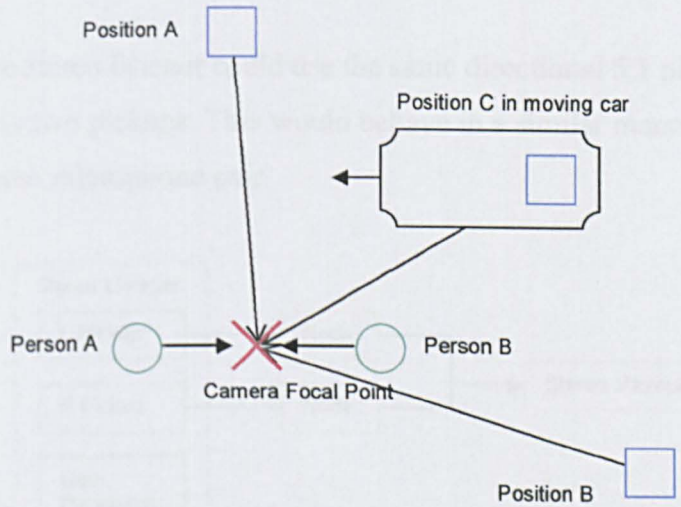
Consider some possible difficulties presented in the sound design for a short film sequence. In this theoretical sequence the sound designer could use traditional

methods to design and mix the spatial components for the sequence. For the purposes of illustrating how the SRP model could relate to real world sound design situations consider the following theoretical situation.

The scene:

During a conversation between subjects A and B a car drives past on the street. The film is shot from three camera positions (A,B,C). Positions A and B are static locations and position C is the moving car. All camera shots focus attention on the conversation. It is impossible to record the conversation at shooting so it is dubbed by the actors and the noise of the car running is recorded while stationary. The film director's concept is for the shots to be edited into a single sequence with a cut from shot A to B, then a blend from B to C. The sound designer decides to complement the film cuts and blend by attempting to spatially mix the sound as if captured from the camera shot locations. Over the blend from B to C it is decided to cross fade the spatial mixes. To further complicate matters the film is to be mixed for multiple formats, 5.1 surround and stereo.

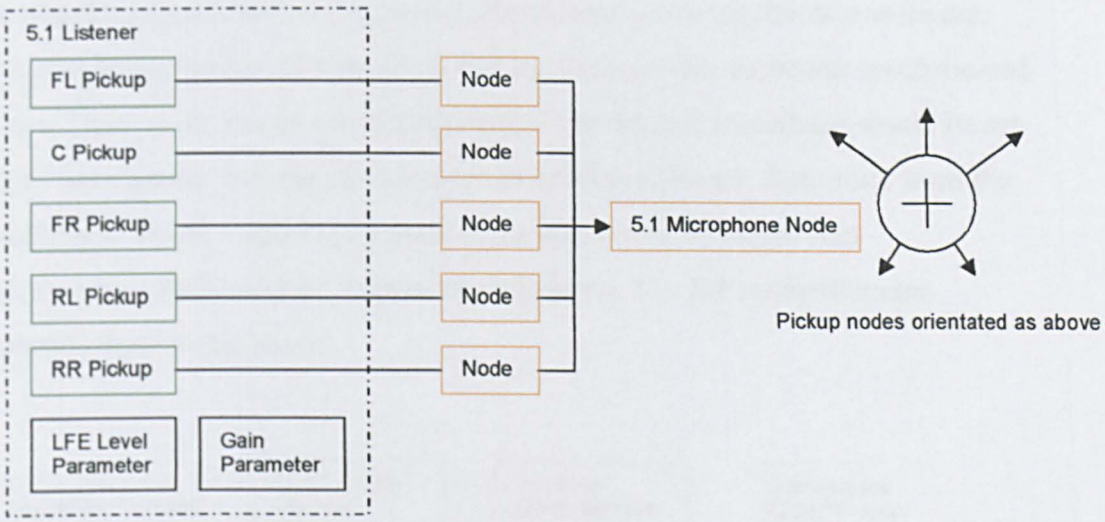
The following diagram provides a simple illustration of the theoretical scene.



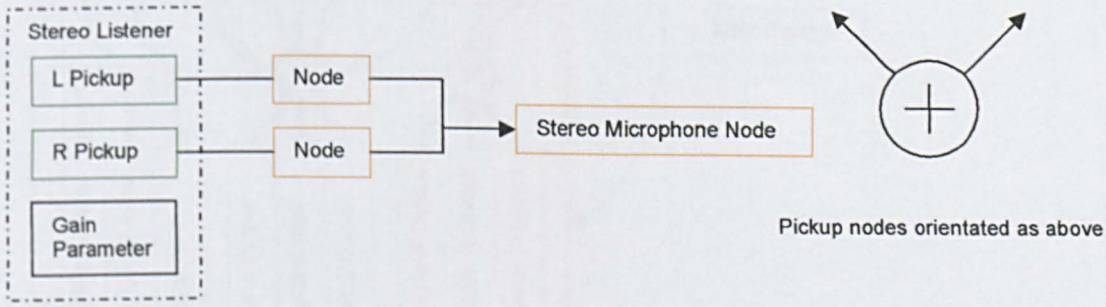
Using SRP to provide a solution:

Obviously there are many conventional compositional methods that could produce appropriate audio for the described situation but the SRP model can represent the

whole scene directly. In a theoretical SRP based implementation the sound designer would be able to work on the project using a simulated visualization of the real world situation. Some theoretical objects could be created and used to simplify the task. Listener objects would be designed to emulate 5.1 and stereo microphones. The 5.1 microphone listener would be created by five directional pickups aimed towards the left, centre, right, rear left and rear right speaker locations in a 5.1 playback system. Control of LFE[85] pickup and listener gain would be provided as parameters of the DSP algorithm. This diagram describes this simple modelling technique.



The stereo listener could use the same directional 5.1 pickup technique but with only two pickups. This would behave in a similar manner to a coincident (X/Y) stereo microphone pair.

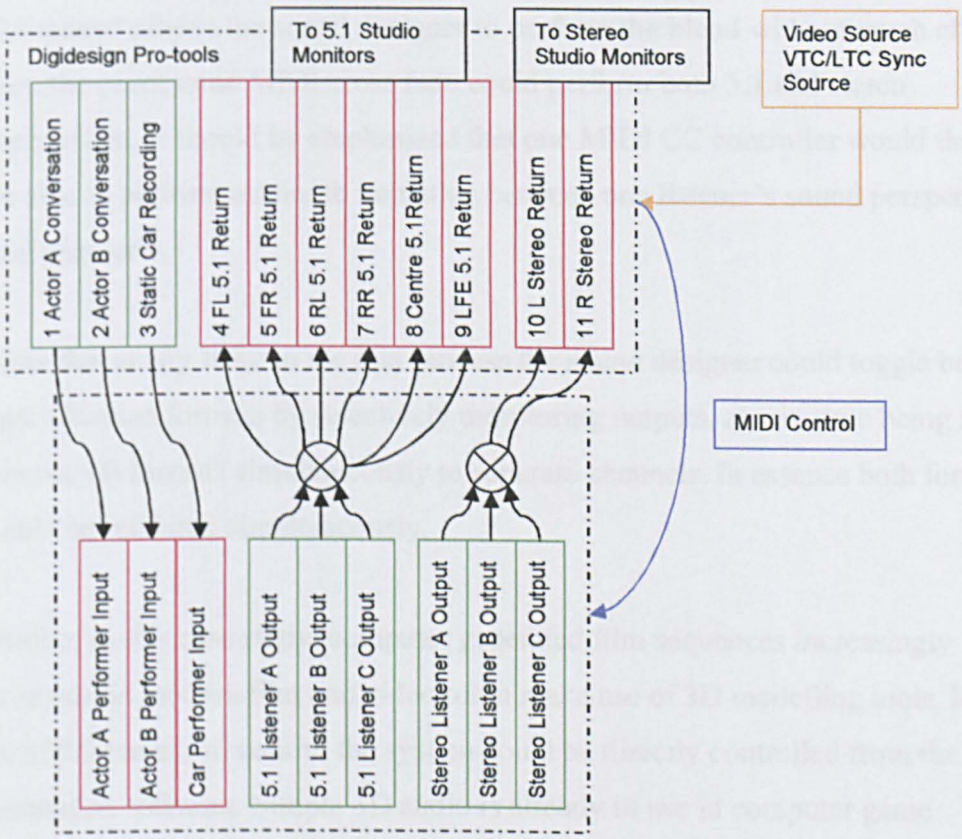


[85] LFE: Low Frequency Effects channel (Sub bass channel of 5.1 surround system)

Assuming that the following theoretical objects are provided with the software implementation, the sound designer could potentially use existing sound design software in conjunction to the SRP model:

- 'MIDI Transform' a transform controller for translation responding to MIDI Continuous Controller data.
- Omni-directional live input source.
- Directional live input source.

The sound designer could use typical multi-channel sound production software, for example Digidesign's Pro-tools software, on a separate computer synchronised to film. Three audio tracks containing edits of the dubbed recordings would be set up and fed digitally into the environment modelling software. Returning from the modelling software, eight digital audio channels would be routed into corresponding tracks and set to monitor their input. The following diagram illustrates these connections:



Using the modelling software, the sound designer could create a basic model of the shooting location and set up three 5.1 listeners and three stereo listeners corresponding to the original camera positions. The outputs of the listeners would be linked back to the recording software's return tracks and directly routed to the studio monitors. Three performers would be created and assigned to the incoming pre-recorded audio tracks. The recording software could have direct control of the position and orientation of listeners and performers via the theoretical 'MIDI Transform' controllers.

While synchronised to the video, the sound designer is in theory able to preview the audio mix for the whole scene. To perform the cuts and cross-fade, MIDI controllers might be assigned to the gain parameters of the three listeners and could again be directly controlled from recording software via MIDI. Using MIDI for control data transfer would allow the sound designer to use familiar sequencing techniques and tools to fully automate the desired parameters. This audio blend between camera perspectives is potentially a very complicated automation using conventional techniques. However, the ability to cross fade the 'Listeners' allows the sound designer to perform the blend without much effort. In fact, the exact same MIDI cross fade could perform both 5.1 and stereo automation. It should be emphasized that one MIDI CC controller would therefore be able to perform a smooth transition between one listener's sound perspective and another's.

Note that at any stage in the mix process the sound designer could toggle between spatialization formats by selectively monitoring outputs, at mix time being able to bounce all formats simultaneously to separate channels. In essence both formats could be rendered simultaneously.

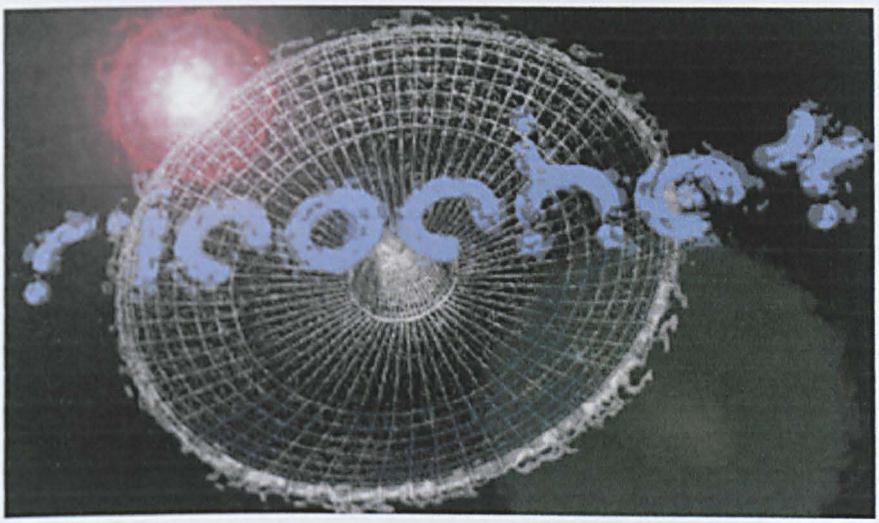
Further to this case study, computer generated film sequences increasingly common in modern film and video often make use of 3D modelling tools. Instead of MIDI transform control the system could be directly controlled from the animation software. Simple 3D audio is already in use in computer game

technologies such as Microsoft's DirectX with real time game animation also linked to audio sound effects.

5 A Real 'Virtual Sound Environment' Modelling Tool

Theoretical case study has provided an indication of the potential for SRP based spatial synthesis. Now this thesis investigates a real world implementation of the concepts presented in the previous sections. 'Ricochet' was developed in parallel with the 'Virtual Sound Environment' conceptualization. It is intended as both a test bed for the model and a tool for the sound artist.

This implementation section of the thesis assumes some knowledge of the C/C++ language, general software development terms and concepts. Many concepts of DSP are related to both project applications and the section entitled 'Real-Time DSP on the host CPU' covers the technical aspects of this.



5.1 Development Tools

'Ricochet' has been developed for the Windows 2000/XP platform using Microsoft's Visual Studio IDE. The version submitted has been written based on two previous prototype versions that highlighted many design implementation problems. These previous versions also helped to consolidate the final model concept. The software makes use of two 3rd party technologies for the purposes of

speeding development and allowing simplified hardware compatibility for audio and 3D graphics.

Silicon Graphic's OpenGL provides a very usable and well supported graphics processing implementation. Its C language API is cross platform compatible across Microsoft Windows and Apple Mac. OpenGL provides access to hardware accelerated 3D graphics functions and so is a useful tool when creating software that requires fast graphical representation of world models. Woo, Neider, Davis, and Shreiner provide a more than adequate description of OpenGL and its use[86]

5.2 Implementing Real-Time SRP Synthesis

Using an object oriented model for development allows a very direct translation of the SRP concepts and entities discussed earlier.

This table below shows the entities discussed in the model and the implementation C++ class names that are used to parallel them. Note that during development the prefix 'R' represented 'Ricochet' and was a prefix used for reducing pollution of the global namespace. The full ricochet source code defines most DSP related functionality in the following files: dspsystem.h and dspsystem.cpp.

Model Entity	Ricochet C++ class
Performer	RDSPObject
Manipulator	RDSPObject
Listener	RDSPObject
Source	RSource
Ray	RRay
Pickup	RPickup
SRP Interaction	RSRPInteraction

RDSPObject provides a base class for specific DSP algorithms to inherit from. This implementation allows a single derived class to act as any of the three container entities. RDSPObject contains key virtual functions for forming part of the DSP framework. These functions are defined here.

```
virtual void PerformerDSP(long bufferSize){};
virtual void ManipulatorDSP(int renderPass,long bufferSize){};
virtual void ListenerDSP(long bufferSize){};
```

[86] Woo, M; Neider, J; Davis, T; Shreiner, D: 1999

It can be seen that these functions are undefined in the base class, they act purely as placeholders for DSP algorithms. Other DSP classes have similar placeholders:

RSource:

```
virtual void AudioPhysics(RSRPInteraction *interaction,long bufferSize){};
```

RRay:

```
virtual void AudioPhysicsTransmit(RSRPInteraction *interaction,DSPFloat *buffer,long bufferSize){};
virtual void AudioPhysicsReceive(RSRPInteraction *interaction,DSPFloat *buffer,long bufferSize){};
```

RPickup: Also contains srpList (explained later)

```
FastDynamicArray<RSRPInteraction*> srpList;
virtual void AudioPhysics(RSRPInteraction* interaction, DSPFloat *buffer,long bufferSize){};
```

The RSRPInteraction structure stores references to all interested parties in a single SRP Interaction. This should become clearer as the algorithm definition moves on. The GLSGVector3 structures, vectorToPickup and vectorToSource contain calculated vectors for each nodes relative position in terms of their local transform[87].

RSRPInteraction: (Condensed definition showing data only)

```
struct RSRPInteraction
{
        RSource *source; // the source involved in the interaction
        RRay *ray; // the ray involved
        RPickup *pickup; // the pickup involved
        GLSGVector3 vectorToPickup; // vector in terms of the source
        GLSGVector3 vectorToSource; // vector in terms of the pickup
};
```

RDSPManager is a singleton object that acts as the central manager for the entire DSP system. Its DSP() function is the call-back function supplied to ASIOSubSystem[88] and GetPickupPerspective() is a function used by Manipulator and Listener algorithms contained in an RDSPObject. The manager contains three RDSPObject lists that hold the currently created Performer, Manipulator and Listener objects for the current SRP network.

RDSPManager: (Condensed definition showing limited data only)

```
FastDynamicArray<RDSPObject*> performerList;
FastDynamicArray<RDSPObject*> manipulatorList;
```

---

[87] See also: 5.3.1 *'Vectors'* p.127

[88] See also: 6.1.3 *'Packaging ASIO in ASIOSubSystem.dll'* p.162

```
        FastDynamicArray<RDSPObject*> listenerList;
        void DSP(DSPFloat **inputs, DSPFloat **outputs, long bufferSize);
        void GetPickupPerspective(RPickup *pickup,DSPFloat *buffer,long bufferSize, RSource *invisible =
0);
```

A rough explanation of the implementation concept follows:

DSP for a single buffer begins from ASIOSubSystem by calling RDSPManager::DSP(). This function performs the following tasks:

1. Update all parameter and variable data ready for processing this buffer.
2. Cycle through each Performer Object and call its RDSPObject::PerformerDSP() function.
3. Cycle through each Manipulator Object and call its RDSPObject::ManipulatorDSP() function
4. Cycle through each Listener Object and call its RDSPObject::ListenerDSP() function

The three DSP functions will in actuality be defined in the classes which inherit from RDSPObject. Exploring the full DSP algorithm involves examples from inherited classes.

BasicPerformerObject::PerformerDSP():

```
void BasicPerformerObject::PerformerDSP(long bufferSize)
{
        float *in = *GetInputChannel(0).buffer;
        float *sBuffer = GetSource(0).buffer;
        memcpy(sBuffer,in,sizeof(float)*bufferSize);
}
```

This performer class contains one physical sound input[89] and one RSource object. The algorithm simply copies from its real audio input buffer to its RSource buffer.

BasicListenerObject::ListenerDSP():

```
void BasicListenerObject::ListenerDSP(long bufferSize)
{
        float *out = *GetOutputChannel(0).buffer;
        RPickup *pickup = &GetPickup(0);

        dspManager.GetPickupPerspective(pickup,out,bufferSize);
}
```

This listener class contains one physical sound output and one RPickup object. After obtaining a pointer to an output buffer and a pointer to its pickup the listener

[89] Physical inputs / outputs: connections to real-world audio I/O – See also: 5.2.1 *'Virtual I/O'* p.124

algorithm makes a single call to RDSPManager::GetPickupPerspective() via the global 'dspManager', passing its pickup and its output buffer. The purpose of GetPickupPerspective() is to obtain an individual pickup's sound 'perspective'[90] and write it into the buffer provided. So in this case, BasicListenerObject requests the DSP manager object to obtain the current perspective for its one pickup and write the sound data directly to its output buffer. Examining a simple Manipulator algorithm further explains the concept.

BasicManipObject::ManipulatorDSP():

```
void BasicManipObject::ManipulatorDSP(int renderPass,long bufferSize)
{
	float *b = source->buffer;
	memset(b,0,sizeof(DSPFloat) * bufferSize);
	dspManager.GetPickupPerspective(pickup,b,bufferSize,source); // source is invisible
}
```

This manipulator class contains one RSource and one RPickup. The algorithm simply obtains its source's write buffer and passes it to the GetPickupPerspective() function along with a pointer to its pickup. In effect the pickup's view of the world is played directly back into the source.

The implementation of SRP synthesis is carried out by the GetPickupPerspective function and this is examined here:

```
void RDSPManager::GetPickupPerspective(RPickup *pickup,DSPFloat *buffer,long bufferSize, RSource
*invisible)
{
	int size = pickup->srpList.Size();
	for(int n = 0; n < size; n++)
	{
		RSRPInteraction *i = pickup->srpList[n]; // get the next interaction

		if(i->source != invisible) // skip if the source needs to be invisible to this pickup
		{
		// update interaction vectors from the SRT transform
		i->vectorToPickup = i->source->GetNode()->GetVectorToTarget(i->pickup->GetNode());
		i->vectorToSource = i->pickup->GetNode()->GetVectorToTarget(i->source->GetNode());

		// perform source dsp
		i->source->AudioPhysics(i,bufferSize);
		// perform pickup dsp
		i->pickup->AudioPhysics(i,buffer,bufferSize);
		}
	}
}
```

[90] Pickup perspective: A rendering of sound from the point of view of an individual pickup.

The first thing to notice is the srpList contained in RPickup. This is another FastDynamicArray and it contains all RSRPInteraction objects associated with this pickup. RPickup::srpList construction is performed by the RDSPManager and is explained later in this text. The concept is fairly simple:

```
Cycle through each SRP interaction skipping an SRP if the source should be invisible.
For each SRP
{
        Update the SRP vectors for use in RSource or RPickup calculations.
        Call the associated source's AudioPhysics() function
        Call the associated pickup's AudioPhysics() function passing in the output target buffer.
}
```

Of note here is the repeated call to RPickup::AudioPhysics(), the associated pickup is always the same for one call of GetPickupPerspective(). Multiple calls are needed because each SRP Interaction will produce different results for each RSource.

Taking a look at an Omni Directional RSource algorithm starts to explain the SRP calculation:

```
void ROmniSource::AudioPhysics(RSRPInteraction *interaction,long bufferSize)
{
        interaction->ray->AudioPhysicsTransmit(interaction,(float*)&buffer,bufferSize);
}
```

Remember here that RDSPObject::PerformerDSP() has already been called for all performer objects, therefore all RSource objects should contain up to date sound data[91]. The source simply obtains the associated RRay from the interaction and calls its AudioPhysicsTransmit() function passing in the pre-filled buffer.

Moving on to the RPickup::AudioPhysics() function:

```
void ROmniPickup::AudioPhysics(RSRPInteraction* interaction,DSPFloat *buffer,long bufferSize)
{
        float *temp = new float[bufferSize]; // creation of temporary buffer
        float *t = temp;
        float *b = buffer;

        interaction->ray->AudioPhysicsReceive(interaction,temp,bufferSize); // this call obtains sound from
the ray

        long size = bufferSize;
        while(size-- > 0)
        {
                *b++ += *t++; // summing calculation
        }
```

[91] Manipulators create a special case and are examined later.

```
        delete [] temp;
}
```

Again this is a simple function, the only complication here is the necessity for 'summing' to the output buffer. This summing (+=) is a requirement of any pickup implementation in order to perform audio mixing of a single pickup's SRPs. In the above code a temporary buffer is passed to the ray to obtain the signal received at the pickup. This temporary buffer is then summed into the output buffer.

RRay transmission:

```
void RBasicRay::AudioPhysicsTransmit(RSRPInteraction *interaction,DSPFloat *buffer,long bufferSize)
{
        // no transmission effects occur so this is just a write onto the delay buffer
        long size = bufferSize;
        float *b = buffer;

        while(--size >= 0)
        {
                writeIndex &= 262143;
                delay[writeIndex] = *b;
                b++;
                writeIndex++;
        }

}
```

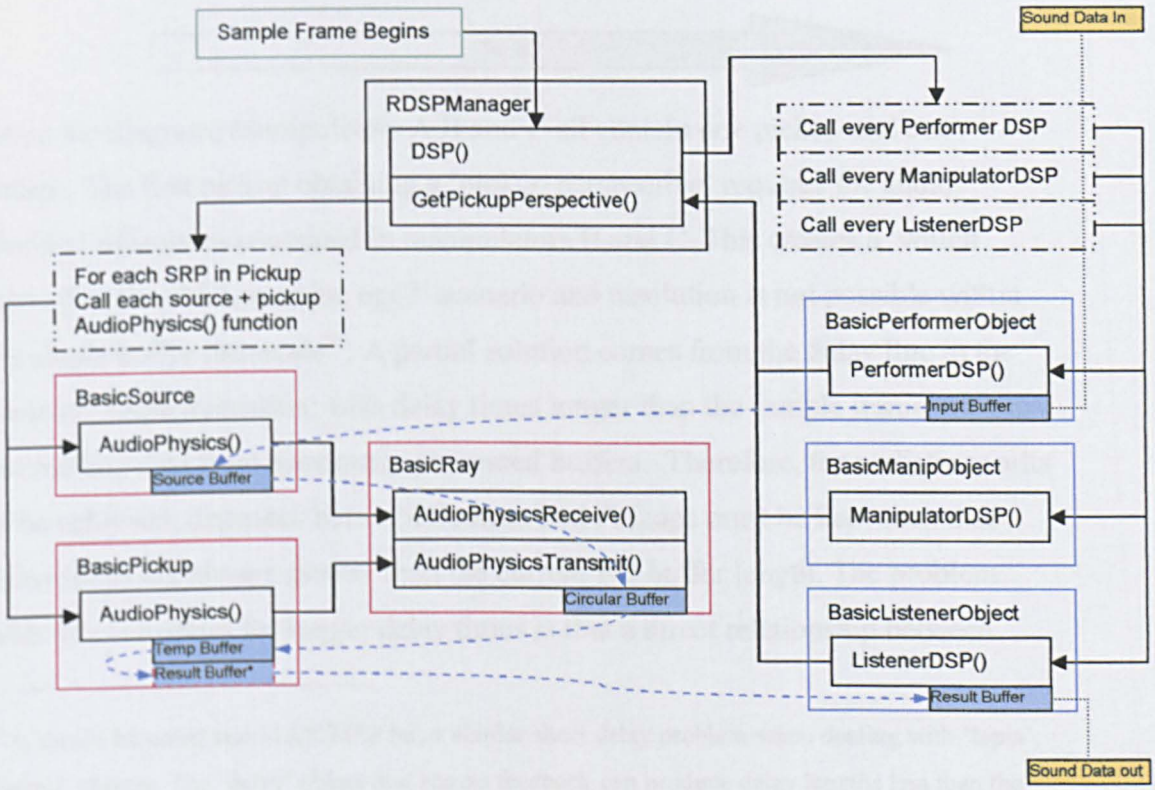
Transmission into the ray is initiated from RSources and the algorithm for this basic ray class is a simple write into a circular delay line. A fast technique is used in this case to perform the circular indexing by using delay buffer sizes in powers of two and a bitwise AND operation to avoid requirement of per sample conditional tests. This basic ray type is a simple delay with distance based gain and most of the calculations are performed in the AudioPhysicsReceive() function.

```
void RBasicRay::AudioPhysicsReceive(RSRPInteraction *interaction,DSPFloat *buffer,long bufferSize)
{
        float ipf = 0.975; // interpolation factor

        float d = interaction->vectorToSource.GetMagnitude(); // distance
        d = ((1.0f - ipf) * d) + (ipf *oldDistance); // log interpolation function
        oldDistance = d;

        float c = 345.0f; // speed of sound
        float t = 44100.0f; // sample rate
        float newdt = (d / c) * t; //delay time is in samples
        float gain = 1.0f / (d * 0.5 + 1.0); // inverse square law with scale factor 0.5: +1 to avoid division by 0
        float dt = lastDT;
        float bs = bufferSize;
        float delayinc = (newdt - dt) / bs;
        float delayedOut;
```

```
        int readIndex; // always calculated

        // proccess
        long size = bufferSize;
        while(size-- > 0)
        {
                dt += delayinc; // increment the float delay time
                // readPos = ((writePos - bufferSize - dt) + size) & loopPoint;
                readIndex = ((writeIndex - size - (int)dt) + 262143) & 262143; // calculate the index
                delayedOut = delay[readIndex]; // get value from the delay line

                *buffer = delayedOut * gain;
                // increment
                buffer++;

        }
        lastDT = dt; // store old dt
}
```

This simple ray algorithm calculates a delay time from the smooth interpolated distance; distance is obtained from the vector to source held in the SRP. The calculated delay time is used to index the circular buffer and obtain audio samples. Delayed samples are written to the output buffer[92] after multiplication by a simple distance based gain factor.

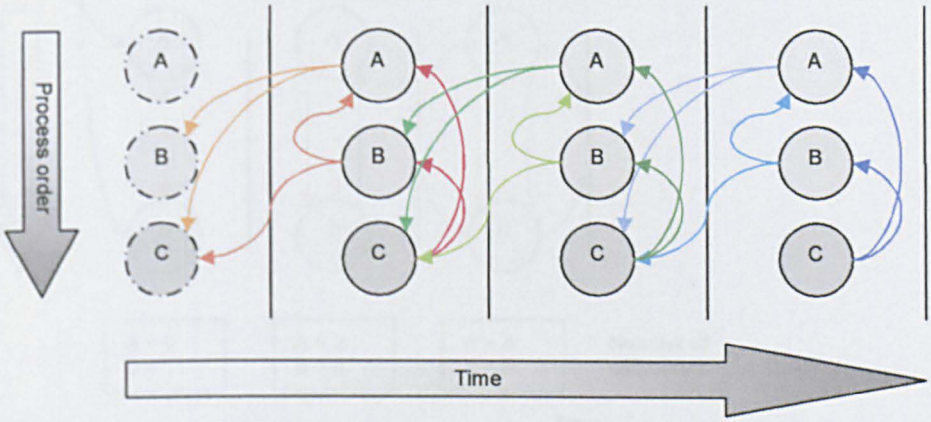
This completes the basic 'Ricochet' SRP implementation and the following diagram illustrates the technique.



[92] The output buffer was passed from the RPickup initiating the call to GetPickupPerspective()

Note that objects represented in blue inherit from RDSPObject; objects in red inherit from the three associated classes RSource, RRay and RPickup; the path of an audio input buffer through one SRP interaction is superimposed to simplify the concept.

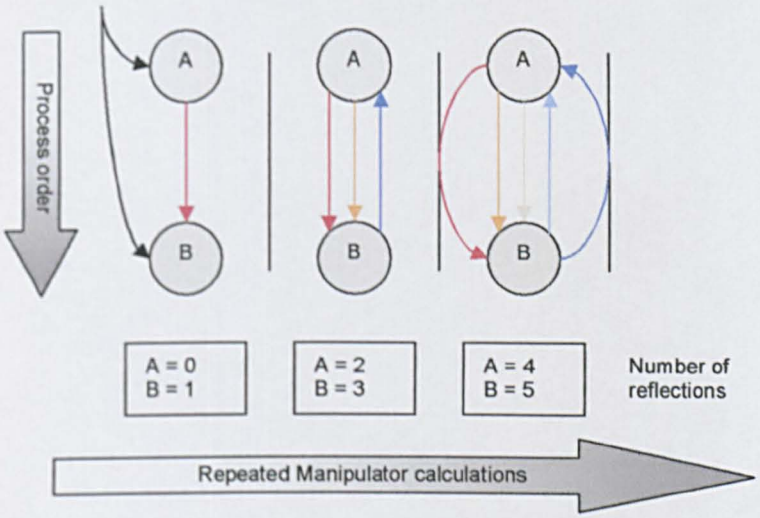
During this explanation, simplifications have been made with respect to Manipulator objects. In a single sample frame each manipulator is considered one at a time. This implies that each pickup contained in manipulators is processed in a sequence and herein lies a problem. To illustrate:



From the diagram; Manipulators A,B and C all contain one pickup and one source. The first pickup obtaining a 'pickup perspective' requires the audio 'results' of sources contained in manipulators B and C. This creates a 'which came first the chicken or the egg?' scenario and resolution is not possible within the single buffer timescale[93]. A partial solution comes from the delay line in the basic ray implementation; with delay times longer than the sample frame, pickups are reading data from previously processed buffers. Therefore, for realistic results to be achieved, distances between Sources and Pickups must be kept such that delay times are always greater than the current I/O buffer length. The problem with this constraint for longer delay times is that a direct relationship between

[93] It should be noted that MAX/MSP has a similar short delay problem when dealing with 'tapin', 'tapout' objects. The 'delay' object that has no feedback can produce delay lengths less than the audio vector size, feedback with 'tapin' requires delays lengths greater than the vector size. Dobrian, C; Zicarelli, D; Puckette, M: 1997, p.198, p.207, pp.280-282. Also determined from experimentation in MAX/MSP.

system latency and simulation detail is created; as latency increases, the ability to model small scale resonance decreases. Note here that Listener Pickups are not affected by delay times smaller than the buffer size so, regardless of latency, accurate distance effects are always achieved. As an attempt to provide small scale resonance effects regardless of latencies, a multiple render pass technique is used. Although not the same principle, this technique takes its inspiration from graphical ray tracing in which a maximum trace depth is used to specify the maximum number of reflections calculated. By repeatedly recalculating the manipulator's DSP function each render pass effectively creates one more reflection in SRPs with delay times less than the buffer size. To illustrate:



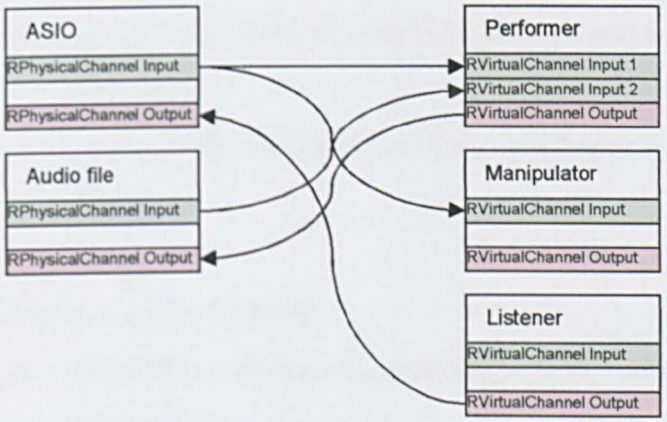
Obviously this technique only produces a finite number of reflections and also produces an imbalance in the number of reflections at each pickup. It does, however, create some illusion of reflection when latencies are high. Examining the implementation of this method, the following code is taken from RDSPManager::DSP()

```
size = manipulatorList.Size();
for(int pass = 0; pass < maxPasses; pass++) // Process a number of render passes
{
        // Process Each Manipulator Object
        for(n = 0; n < size; n++)
        {
                manipulatorList[n]->ManipulatorDSP(pass,bufferSize);
        }
}
```

Note from the above, the multiple render pass method is an (N^2) algorithm[94] and therefore causes a large degree of processor loading.

5.2.1 Virtual I/O

Audio input and output from the physical world to the virtual world is accomplished by the use of two classes RPhysicalChannel and RVirtualChannel. A physical channel can be created as either input or output and essentially forms a buffer containing the current sample frame data. The virtual channel, again creatable as either input or output, is an object that allows connection of performer/ manipulator/ listener objects to a physical channel. The principle is explained here:



RDSPObject contains a list of input virtual channels and a list of output virtual channels. The DSP functions can all access these lists and obtain data from any of the associated virtual inputs. The number of ins and outs is determined by the particular object class that inherits from RDSPObject while the connection of physical inputs and outputs is controlled from the user interface.
RPhysicalChannel provides an abstraction layer between the actual physical audio input and the virtual channel and this allows interchangeable connection between audio I/O methods. For example, audio I/O streaming from files can be accommodated without the need to update virtual channel code.

---

[94] Sedgewick, R: 1999, pp.36-39

At this point the flexibility of the system is hinted at, but to clarify, the following list of DSP features are available in the framework: Note that the Ricochet implementation essentially provides three 'positions' in its DSP process 'chain'; Performer, Manipulator and Listener.

In RDSPObject derived classes:

- Can contain algorithms for use in all three DSP chain positions.
- Can create any number of virtual I/O channels, RVirtualChannels.
- Can create any number of RSource derived objects.
- Can create any number of RPickup derived objects.
- DSP chain position is determined at object creation.
- PerformerDSP() dictates how RVirtualChannels connect to RSource derived objects.
- ManipulatorDSP() dictates how pickups and sources are internally connected.
- ListenerDSP() dictates how RVirtualChannels connect to RPickup derived objects

In RSource derived classes:

- AudioPhysics() dictates how source buffer sound is emitted to an RRay derived class.

In RPickup derived classes:

- AudioPhysics() dictates how sound from an RRay derived class is received.

In RRay derived classes:

- AudioPhysicsTransmit() dictates effects on sound before transmission along a ray.
- AudioPhysicsReceive() dictates effects on sound after transmission along a ray.

With these concepts reviewed, the focus moves on to the task of representing the world.

5.3 Implementing Hierarchical Scene Graphing

In order to implement a scene graph it is first necessary to create a 3D mathematics and utility library to enhance the basic functionality supplied with ANSI C/C++. Making use of C++ operator overloading greatly simplifies the usability of such a library and is well suited to mathematical classes. For reference, the majority of 'Ricochet' maths and scene graph functionality is defined in the scenegraph.h and scenegraph.cpp files. In the 'Ricochet' implementation the prefix GLSG[95] was used, again to reduce namespace pollution.

5.3.1 Vectors

3D Vectors are, for the most, implemented in 'Ricochet' by the structure GLSGVector3. The frequent use of GLSGVector3 and some interesting extension calculations warrants detailed examination.

```
struct RSDK_API GLSGVector3
{
        GLSGFloat x,y,z;

        GLSGVector3(){x = y = z = 0;};
        GLSGVector3(GLSGFloat _x,GLSGFloat _y, GLSGFloat _z){x = _x; y = _y; z = _z;};
        GLSGVector3(GLSGFloat *_v){x = _v[0]; y = _v[1]; z = _v[2];};
        GLSGVector3 operator +(GLSGVector3 &op){return GLSGVector3(x + op.x, y + op.y, z + op.z);};
        GLSGVector3 operator -(GLSGVector3 &op){return GLSGVector3(x - op.x, y - op.y, z - op.z);};
        GLSGVector3 operator -(){return GLSGVector3(-x,-y, -z);};

        GLSGVector3& operator +=(GLSGVector3 &op){x += op.x; y += op.y; z += op.z; return *this;};
        GLSGVector3& operator -=(GLSGVector3 &op){x -= op.x; y -= op.y; z -= op.z; return *this;};
        GLSGVector3& operator *=(float &op){x *= op; y *= op; z *= op; return *this;};
        GLSGVector3& operator /=(float &op){x /= op; y /= op; z /= op; return *this;};

        GLSGVector3 operator *(float &op){ return GLSGVector3(x * op, y * op, z * op); };
        GLSGVector3 operator /(float &op){ return GLSGVector3(x / op, y / op, z / op); };

        GLSGVector3 operator *(GLSGVector3 &op){ return GLSGVector3(x * op.x, y * op.y, z * op.z); };

        GLSGPolar GetPolar();
        GLSGFloat GetMagnitude();
        GLSGFloat GetRadialAngle(int plane = R_ANGLE_NEG_Z); // returns the radial angle

};
```

[95] Representing 'openGL Scene Graph', The Scene Graph implementation also features much of the OpenGL interfacing.

Note that vector multiplication by a scalar provides uniform scaling and component multiplication by a vector causes non uniform scaling; vector addition is equivalent to vector translation. The majority of the functionality is written inline[96] for possible compiler optimisation with the exception of the following:

```
GLSGPolar GLSGVector3::GetPolar()
{
	GLSGPolar p;
	p.rho = sqrt((x*x) + (y*y) + (z*z));
	p.theta = acos(z/p.rho);
	if(y >= 0)
	{
		p.phi = acos(sqrt( z / ((x*x) + (y*y))));
	}
	{
		p.phi = -acos(sqrt( x / ((x*x) + (y*y))));
	}
	return p;
}

GLSGFloat GLSGVector3::GetMagnitude()
{
	return sqrt((x*x) + (y*y) + (z*z));
}

GLSGFloat GLSGVector3::GetRadialAngle(int axis /*=R_ANGLE_NEG_Z*/)
{
	switch(axis)
	{
	case R_ANGLE_NEG_X:
		return fabs(atan2(sqrt((z*z) + (y*y)),-x));
	case R_ANGLE_NEG_Y:
		return fabs(atan2(sqrt((x*x) + (z*z)),-y));
	case R_ANGLE_NEG_Z:
		return fabs(atan2(sqrt((x*x) + (y*y)),-z));
	case R_ANGLE_X:
		return fabs(atan2(sqrt((z*z) + (y*y)),x));
	case R_ANGLE_Y:
		return fabs(atan2(sqrt((x*x) + (z*z)),y));
	case R_ANGLE_Z:
		return fabs(atan2(sqrt((x*x) + (y*y)),z));
	}
	return 0.0f;
}
```

GetPolar() and GetRadialAngle provide useful calculations for use in source and pickup DSP; GetPolar() returns the vector converted to polar coordinates; GetRadialAngle() calculates the angle between the vector and a specific axis vector by the following equation: arranged for radial angle against –x axis

$$\tan\theta = \frac{\sqrt{z^2 + y^2}}{-x}$$

[96] Stroustrup, B: 2000, p.144

This implementation makes use of the 'atan2'[97] function to correctly handle quadrant calculations during the inverse tan operation. The GetRadialAngle() function is used in the RDirectionalSource and RDirectionalPickup classes to determine directional gain factors in DSP.

5.3.2 Quaternions

Quaternions are represented in the Ricochet implementation as GLSGQuaternion's. As with GLSGVector3, standard mathematical operation is achieved with operator overloading. Quaternions can be constructed in a number of ways as shown below:

```
struct RSDK_API GLSGQuaternion // quaternions are used to for quicker and smoother rotational transforms
{
        float w;
        GLSGVector3 v;

        GLSGQuaternion() { w = v.x = v.y = v.z = 0.0f; }; // null constructor
        GLSGQuaternion(float angle, GLSGVector3 axis); // construct from angle around axis

        // construct from manually from know components
        GLSGQuaternion(float _w, float _x, float _y, float _z) { w = _w; v.x = _x; v.y = _y; v.z = _z; } ;
};
```

In practice, quaternions are often constructed with the BuildFromTriAxis() function which makes use of the angle / axis constructor. This constructor takes a directional vector axis and an angle in which to rotate around it. The implementation is shown below:

```
GLSGQuaternion::GLSGQuaternion(float angle,GLSGVector3 axis)
{
        w = cos(DegRad(angle) / 2.0f);
        float s = sin(DegRad(angle) / 2.0f);
        v = axis * s;
}
```

The Quaternion implementation features some useful functions for creation from, and rotation of GLSGVector3s.

```
inline GLSGVector3 QVRotate(GLSGQuaternion q, GLSGVector3 v)
{
        return (q * v * ~q).v;
```

---

[97] Schildt, H: 1998, p.734

```
}
inline GLSGVector3 QVInverseRotate(GLSGQuaternion q,GLSGVector3 v)
{
        return (~q * v * q).v;
}
inline GLSGQuaternion BuildFromTriAxis(GLSGVector3 v)
{
        GLSGQuaternion q(v.x,GLSGVector3(1,0,0));
        q = q * GLSGQuaternion(v.y,GLSGVector3(0,1,0));
        q = q * GLSGQuaternion(v.z,GLSGVector3(0,0,1));
        q.Normalize();
        return q;
}
```

QVRotate() and QVInverseRotate() feature highly in rotation of vectors through the scene graph structure. The quaternion implementation of vector rotation is more efficient than one based on a 3*3 matrix particularly for inversion of the transform; 3*3 matrix inversion requiring significant calculation and the quaternion method requiring a simple rearrangement of the terms.

Note in the above; the advantage of the operator overloading in both vectors and quaternion classes, quaternion vector rotation is calculated using the equation[98]:

$$v' = qv\bar{q}$$

Due to operator overloading the implementation can be coded as `newV = q*v*~q`. It should be clear that the ability to code equations in a manner similar to the hand written format provides a significant aid in translation from conventional notation to implementation[99].

BuildFromTriAxis() allows construction of a quaternion from rotation angles in each axis, the rotations in this case, performed in succession around x - y – z axis.

5.3.3 SRT Transformation

SRT transformation is implemented via the class GLSGTransform: (simplified for clarity).

```
class RSDK_API GLSGTransform // holds an SRT transform, in Quaternion form
```

[98] Eberly, D. H: 2001, p.13

[99] Stroustrup, B: 2000, p.241

```
{
public:
        GLSGVector3 s; // scale factor
        GLSGQuaternion r; // rotation quaternion
        GLSGVector3 t; // translation

        GLSGVector3 LocalToWorld(GLSGVector3 v); // vector transformed from local coord system into
world
        GLSGVector3 WorldToLocal(GLSGVector3 v); // vector transformed from world coord system into
local
        GLSGTransform operator *(GLSGTransform &op); // multiply SRT transforms
        void GLPerformTransform(); // perform this transform on current matrix stack using OpenGL
commands
}
```

From the above note the storage of scale, rotation and translation data in the variables s, r, t. LocalToWorld() provides the functionality to transform a vector from the local (transformed) coordinate system into the world (non-transformed) coordinate system. WorldToLocal() performing the inverse. This functionality forms the basis of the scene graph calculations and is implemented thus:

```
GLSGVector3 GLSGTransform::LocalToWorld(GLSGVector3 v)
{
        return QVRotate(r,v * s) + t;
}
GLSGVector3 GLSGTransform::WorldToLocal(GLSGVector3 v)
{
        return  QVInverseRotate(r,v - t) * GLSGVector3(1/s.x,1/s.y, 1/s.z);
}
```

Note again the use of operator overloading to provide readable code. In LocalToWorld() the vector v, multiplied by the scaling factor s, is rotated via the quaternion r, translation is then performed with a simple addition. To perform inversion the translation t is subtracted from vector v before performing inverse quaternion rotation, finally the vector multiplied by the reciprocal of the scaling factor s.

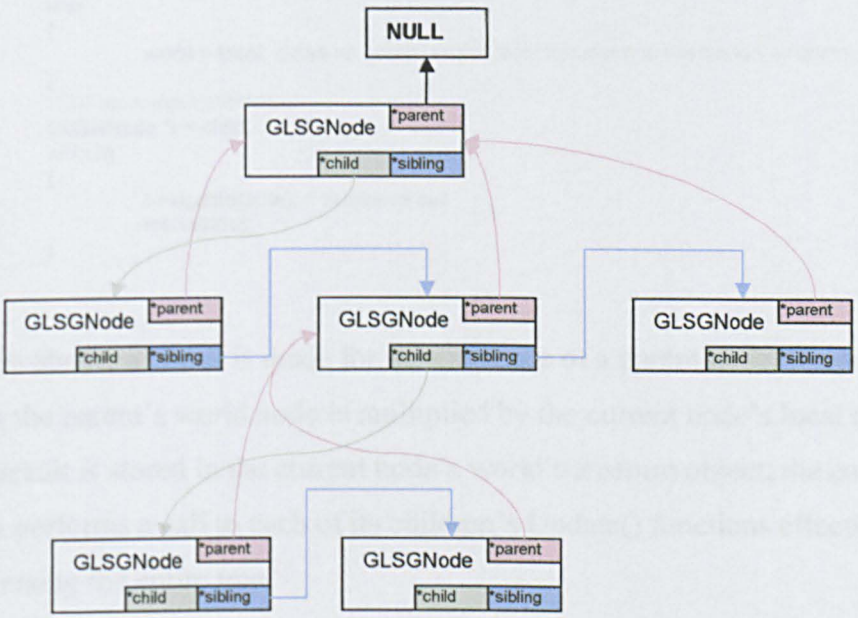
The final function GLPerformTransform() integrates the functionality of the GLSGTransform for use in OpenGL. When called, transformation of the current OpenGL matrix stack occurs, enabling graphics to be rendered relative to the transforms' coordinate system.

5.3.4 Scene Graph Nodes

The Hierarchical Scene Graph is implemented with a tree structure and the basis for the implementation is the GLSGNode class: (highly simplified for clarity).

```
class RSDK_API GLSGNode : public PersistentObject
{
        GLSGNode *parent; // this node's parent
        GLSGNode *child; // points to the first child node
        GLSGNode *sibling; // points to the next sibling node

        // Transforms
        GLSGTransform local; // transform from parent
        GLSGTransform world; // transform to world

        void Update(long time); // recursive update including this node // traversal is top to bottom
}
```

In the above, the pointers *parent, *child and *sibling form the tree structure. The GLSGNode class makes significant use of recursive algorithms to perform the majority of its tasks. Recursion is a typical implementation technique in tree based structures[100]. The following diagram illustrates the tree form used in GLSGNode:



Essentially the sibling pointer forms the connection in a singly linked list[101]; the child pointer linking the first of the node's children. A GLSGNode with a null parent pointer is designated the root of the tree.

---

[100] Sedgewick, R: 1999, p.201

[101] ibid., p.91

Transformation in the Scene Graph is accomplished with the aid of two GLSGTransform objects: local and world. The 'local' transform contains the transformation required at this particular node; the 'world' transform acts as a storage object. A call to the world root node[102] Update() function initiates a recursive traversal of the whole tree. During this traversal each node's world transform object is calculated from its parent's world transform. This storage of successive world transforms provides a pre-calculated single transform from one node's coordinate system to the world system. With this method it is possible to transform from any node in the tree to any other with just two transform calculations. Examining this principle with a simplified Update() function:

```
void GLSGNode::Update(long time)
{
        if(parent)
        {
                world = parent->world * local; // multiply the parents world SRT transform by local transform
        }
        else
        {
                world = local; // has no parent so the local transform is the world transform
        }
        // Do recursive traversal
        GLSGNode *t = child;
        while(t)
        {
                t->Update(time); // recursive call
                t=t->sibling;
        }
}
```

In the above: a check is made for the existence of a parent node; if a parent exists then the parent's world node is multiplied by the current node's local transform; the result is stored in the current node's world transform object; the current node then performs a call to each of its children's Update() functions effectively traversing the entire tree.

The implementation of GLSGNode provides many functions for management of the node tree. These are provided to simplify and validate changes to the structure. Much functionality is provided for the use of transform controllers, a later topic.

---

[102] The world node represents the root node designated as the base of all nodes contained in the world.

5.3.5 Connecting Nodes to SRP objects

Connection of GLSGNodes to RSource and RPickup objects is handled in the base classes by simple methods. Both RSource and RPickup classes are provided with the method AttachToNode() which simply allows a particular pickup to be attached to a particular node. In this implementation it is not necessary for a node to 'know' about objects that are connected to it. It is the responsibility of the source or pickup class to use the transformation information provided in the node.

5.4 Implementing Automation and Control

In the 'Ricochet' example, implementation of automation and control is achieved with one class forming a basis for inherited controller types. RController and GLSGTransformController are parallels to the Controller and Transform Controller entities described in the virtual environment model. To facilitate storage and controller assignment the RParameter class is provided along with extended functionality to the GLSGNode class. As with the 'Virtual Environment Model' the two control sections are treated separately.

5.4.1 Implementing Single Parameter Control

RControllers are assigned to parameters using methods from the parameter class RParameter defined as follows: (simplified for clarity)

```
class RSDK_API RParameter : public PersistentObject
{
        RController *controller;
        float value; // the actual value
public:
        void AssignController(RController *_controller); // assigns a controller to this parameter
        float GetValue(); // returns the parameter value
        void SetValue(float _value);
        void Update(long time); // updates any controllers
};
```

This definition helps visualise the controller-parameter linkage and shows some features of the parameter object class. A parameter's value may be obtained and set via GetValue() and SetValue() methods. AssignController() creates the functionality for connecting a single controller. A call to Update() requests that

the parameter obtain updated data from its controller, if assigned. Moving on to controllers, RController is defined thus: (simplified for clarity).

```
class RSDK_API RController : public PersistentObject, public GUIListener
{
private:
        FastDynamicArray<RParameter*> parameterList;
protected:
        DSPFloat value;
public:
        void CreateParameter(T_RParameterInfo parameterInfo);
        DSPFloat GetValue() { return value; };
        virtual void Update(long time); // overload to determine what happens when the system updates

};
```

The implementations of RParameter and RController act as a pair to form the full hierarchical controller tree. An RParameter may be assigned an RController and a single RController contains a list of sub RParameters. The Update() methods of each class are called in sequence; RParameter::Update() makes a call to its assigned RController::Update(); this in turn calls all RParameter::Update() methods for its list of parameters. This action is essentially a recursive algorithm that performs traversal of a controller/parameter tree. To provide the control algorithm RController is inherited from and the Update() method is overloaded to provide the control algorithm.

5.4.2 Implementing Transform Control

Using the implemented RController as a base class it is possible to add functionality for connection to GLSGTransforms. GLSGTransform controller is therefore defined as follows: (simplified for clarity)

```
class RSDK_API GLSGTransformController : public RController
{
        GLSGTransform *target;
public:
        void SetTarget(GLSGTransform *_target) { target = _target; };

        virtual void UpdateTranslate(long time){}; // overload for translation
        virtual void UpdateRotate(long time){}; // overload for rotation
        virtual void UpdateScale(long time){}; // overload for scale
};
```

The complete functionality for transform controllers to contain sub parameters is inherited directly from RController. An advantage of inheriting is the ability to define a single controller class that can be used for both SRT and single parameter

control; this actually extends the original model slightly. Similarly to RController, control algorithms are defined by overloading methods within inheriting classes, in this case UpdateTranslate(), UpdateRotate() and UpdateScale(). These update functions would typically make adjustments to the target SRT.

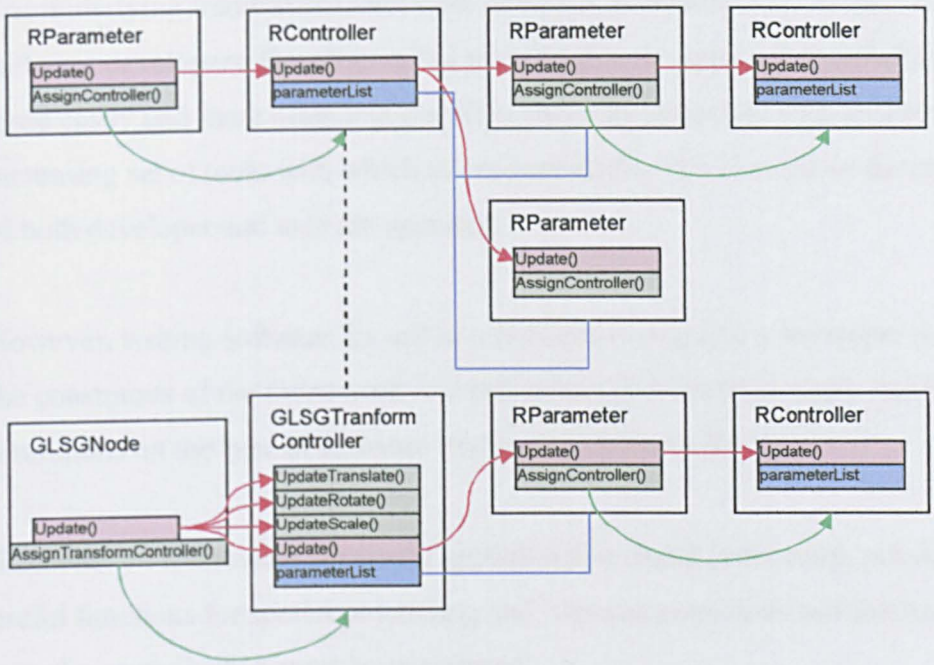
GLSGTransformControllers are connected to GLSGTransforms within the GLSGNode and this connection is performed with the following additional GLSGNode functionality:

```
void GLSGNode::AssignTransformController(GLSGTransformController *controller,int type)
{
	controller->SetTarget(&local);
	RemoveTransformController(type);
	switch(type)
	{
	case T_C_T_TRANSLATE:
		tc_translate = controller;
		break;
	case T_C_T_ROTATE:
		tc_rotate = controller;
		break;
	case T_C_T_SCALE:
		tc_scale = controller;
		break;
	}
}
```

This function allows controllers to be connected to one of three connection points tc_translate, tc_rotate and tc_scale, corresponding to the three components of SRT transformation. Notice the call to GLSGTransformController::SetTarget() passes this node's local transformation SRT as the target for control. The additions to GLSGNode::Update() below, show the different calls made for each type of controller when updating.

```
	if(tc_translate)
	{
		tc_translate->Update(time); // first call updates controller parameters
		tc_translate->UpdateTranslate(time); // this call causes update of the local SRT
	}
	if(tc_rotate)
	{
		tc_rotate->Update(time);// first call updates controller parameters
		tc_rotate->UpdateRotate(time);// this call causes update of the local SRT
	}
	if(tc_scale)
	{
		tc_scale->Update(time);// first call updates controller parameters
		tc_scale->UpdateScale(time);// this call causes update of the local SRT
	}
```

The following diagram shows the entity relationships and update path for control implementation:



5.5 *Implementing Modularity and Expandability*

Audio signal processing software with the facility for extension and experimentation already exists in a number of forms. DSP processes can be described graphically using Cycling 74's Max MSP software. In itself the power and creative flexibility of Max MSP is powerful but an important feature is the ability for the software designer to access the underlying framework of Max and produce 'Externals' for use in the graphical modelling environment. Many researchers in signal processing create Max MSP Externals for experimental purposes and the ability to concentrate solely on an algorithm without regard for interface is of benefit.

Steinberg's VST technology provides another framework for expression of signal processing algorithms. Software developers are able to both design algorithms for use in VST software hosts and produce VST host software that can use other designer's algorithms. A software package that supports the VST framework has immediate access to a large variety of signal processing algorithms in the form of the VST plugin.

The underlying frameworks above are of help to software users as well as software developers. Developers can test and release new experimental algorithms more easily and more often and therefore users are presented with an ever increasing set of tools with which to produce audio. The benefits to the creativity of both developer and user are apparent.

However, writing software for use in a framework requires a developer to follow the constraints of the framework and this means that the framework must impose limitations on the type of software that can be embedded in it.

'Ricochet', a framework for sound spatialization signal processing, provides useful functions for spatial processing and imposes guidelines and limitations on how the spatialization must be performed.

The implementation described in the previous sections serves as a framework for spatial DSP and spatial control. The actual algorithms that perform DSP and control require additional object classes to inherit from the framework base classes. During the discussion of DSP a number of these inherited classes were introduced. In essence, the framework provides the functionality for connection and categorisation of spatial objects using the SRP model. In order to promote creative extension to the SRP model the 'Ricochet' engine provides all of its functionality to the 3rd party developer via a 'Plugin' based Software Development Kit (SDK). This kit can be used by any software developer to create new object classes based on the framework objects. An introduction to the Ricochet SDK (RSDK) is provided by examination of a basic plugin. The following code would be compiled and linked as a dynamic link library that is placed in the ricochet plugin directory:

```
class BasicPerformerObject : public RDSPObject
{
        PERSISTENT_OBJECT_HEADER(BasicPerformerObject,'bper')
public:
        void PerformerDSP(long bufferSize);
        void Create(GLSGVector3 pos);
};
```

Taken directly from baseplugin.h the above class header shows the full class definition for the BasicPerformerObject in an earlier example. The full implementation follows and is taken from baseplugin.cpp:

```
void BasicPerformerObject::Create(GLSGVector3 pos)
{
        CreateChannel("Source Input",true); // create an input channel
        GroupNode* node = new GroupNode; // create a node
        node->SetNode(pos); // set the node to the position set by the interface
        node->SetName("Source 1"); // name the node in the interface
        ROmniSource* source = new ROmniSource; // create a source
        source->AttachToNode(node); // attach the source to the node
        AttachSource(source); // attach the source to the dsp engine
        AttachNode(node); // attach the node to the graphics engine
}

void BasicPerformerObject::PerformerDSP(long bufferSize)
{
        float *in = *GetInputChannel(0).buffer;
        float *sBuffer = GetSource(0).buffer;
        memcpy(sBuffer,in,sizeof(float)*bufferSize);
}
```

The framework components needed by the object i.e. virtual channels, nodes, sources and pickups are obtained within the overloaded Create() function. Using simple function calls, the 3rd party developer is able to build an object that is under full control of the interface. It should be noted here that this is the complete source code for one object and all that remains is for the 3rd party developer to register it within the system. This is performed with a number of simple macros and functions.

```
PERSISTENT_OBJECT_SOURCE(BasicPerformerObject)
int RPluginMain()
{
        SPLASH_MESSAGE("Simple RSDK example plugin.");
        REGISTER_DSP_CLASS(BasicPerformerObject,"Basic Performer",1,0,0);
        return 0;
}
```

The above function name is automatically found and called by Ricochet when it searches its plugin directory. The macro SPLASH_MESSAGE allows an author to have a message displayed by Ricochet when the plugin is loaded. REGISTER_DSP_CLASS simply registers this object class with a name seen by the user and also allows specification of which DSP algorithms exist. PERSISTENT_OBJECT_HEADER and PERSISTENT_OBJECT_SOURCE are macros defined for the purposes of saving and loading, a later topic in this document.

In just 31 lines of code, an object has been developed that behaves as an omni directional performer in a virtual environment. The object's SRT transformation is

hooked into Ricochet's graphical interface providing user control of motion. I/O assignment is immediately provided to the user via the channel object. Details of object positions and parameters are automatically saved and loaded with a Ricochet document. The advantage of 'plugin' modularity is that 3rd party developers can create new tools for the spatial artist without the need to develop a full framework. In particular someone interested in researching the SRP technique can develop new DSP algorithms and test them with other developer's ideas without in depth knowledge of the GUI or SRP framework.

5.6 *Implementing Object Persistence*

In providing a dynamic and expandable system this implementation has created a difficult task for the purposes of saving and loading user files. Consider a typical user of the system who creates an SRP network containing multiple object types, some developed by 3rd party developers, deeply nested groupings of nodes in the scene graph and multiple levels of control. The necessity to store data for dynamically created objects (object persistence) is of particular importance in this project. The particular implementation of the data structures makes significant use of memory pointers to reference data/objects that are dynamically created and destroyed. It is possible to traverse all created objects storing contained data, but a problem arises when storing the pointer references. To store the value of a dynamic memory pointer between user sessions is inherently pointless as recreation of the originally stored dynamic objects will inevitably allocate different memory blocks from the heap. It is feasible that pointer references be discarded in favour of unique ids for each dynamically created object. The disadvantage here is the speed bottleneck in looking up an id and cross referencing it every time object/object interaction is required. This method can be sped up using pointers and id variables; ids provide persistence between sessions; cross referencing to set pointers to accurate object memory locations is performed on file load. Of course the problem here is the implied doubling in data storage required per pointer and the increased complexity of dynamic creation algorithms. In this case a simple tree structure becomes significantly more complex to design.

The solution makes use of the inherent uniqueness of pointer values. Two objects cannot occupy the same memory location so object pointers always take a unique value. By using pointer values as unique ids it is possible to provide an efficient object persistence solution that does not suffer in speed or complexity of data structure. Object persistence is achieved by storing information about an object's class and storing its current memory location as its unique id. Each object's individual data is stored along with any pointers to persistent objects. Upon loading, objects are recreated from the stored class information. The stored memory location is used to create a lookup table of old and new memory locations for the object. Loading of data is as normal but any stored pointers are first cross referenced in the lookup table and the new memory location is stored. With this method the entire data structure of the system, regardless of complexity in referencing, is stored and retrieved successfully.

5.6.1 Object Persistence in 'Ricochet'

- Objects that require persistent storage inherit from PersistentObject
- Persistent object classes provide a 'factory'[103] function via use of macros.
- Persistent object classes provide Run Time Type Information (RTTI)[104] data via same macros.
- If an object contains other references/pointers to other persistent objects it overloads the virtual function PrepSave(). Within this function any child objects PrepSave() function is called, providing data structure traversal.

When Saving:

- The system sets up the file stream object ready for saving. A new file is created etc.
- The system then calls PrepSave() for all of the parent objects that require storage.

---

[103] Stroustrup, B: 2000, p.323 and Eberly, D. H: 2001, p.460

[104] Stroustrup, B: 2000, pp.407-418 and Eberly, D. H: 2001, pp.444-451

- The PrepSave() call propagates through the data structures to all objects that need saving.
- With the PrepSave() calls done, the file stream now has a list of all objects that require saving.
- The file stream traverses the list and performs the following for each object:
    - Object info data header is stored first and contains the RTTI info and the current memory address of the object to be saved.
- The stream then reiterates the list performing the following:
    - The object's Save() function is called.
    - The individual save functions will store static data for themselves.
    - Any object references must be saved using the SaveObject() of the file stream.
- The base system can now save any of its own raw data. Any objects that must be pointed at can be saved with SaveObject().

The file structure at the end of this is as follows:

{[RTTI and instance info]}... {[ObjectDataSize][Object data]}... {[System raw data]}

RTTI and instance info is stored in the same order as the object data.

When Loading:

- The system sets up the file stream for loading of the data.
- The system traverses the file and for each object block performs the following:
    - The object header is loaded.
    - RTTI data is used to create a new object.
    - The new object's address and its old memory address are stored in a lookup table. If the object failed to create then a NULL value is stored.

- Traverse the list of created objects stored in the table. If an object could not be created then a memory NULL pointer will have been stored and the system will skip the data. For each object the following is performed:
    - Read object data header info – basically just the size of the stored data
    - Call the object's own Load() function
    - Any object pointers will be read via LoadObject() which cross references the old pointer value with the new one. It is possible that this function will return a NULL pointer and in this case the object should be coded to act accordingly.
- The final system data is loaded and if any pointers exist they can be loaded as above.

The full source for this technique is supplied in the files: persistence.h and persistence.cpp. These files contain many classes that essentially perform the above technique. This thesis will not provide an in depth discussion of this code but a review of the steps necessary to create a persistent object class is relevant for the software development kit.

A new class taking advantage of the 'Ricochet' object persistence model can do so very simply. Two macros PERSISTENT_OBJECT_HEADER and PERSISTENT_OBJECT_SOURCE are used to insert most persistence functionality to a new class. It is also a requirement that a base class inherits from PersistentObject. Overloading of three virtual functions PrepSave(), Save() and Load() allows an object to provide information about child objects and store and retrieve class specific data. The FStream class provides access to the actual file and a saving or loading object will be presented with the FStream instance that is in use. To examine this concept the following is a condensed version of RDSPObject's persistence methods:

```
class RSDK_API RDSPObject : public PersistentObject, public GUIListener
{
        PERSISTENT_OBJECT_HEADER(RDSPObject,'rsdp')
        virtual void PrepSave(FStream &stream);
        virtual void Save(FStream &stream);
        virtual void Load(FStream &stream);
}
```

The above is from the header file and the following should be noted: PersistentObject is inherited from; the PERSISTENT_OBJECT_HEADER() macro is used first in the class definition passing the name of this class and a unique 4 character code (developer defined); all three virtual functions are overloaded. The source implementation follows (simplified for clarity):

```
PERSISTENT_OBJECT_SOURCE(RDSPObject)

void RDSPObject::PrepSave(FStream &stream)
{
        PersistentObject::PrepSave(stream);
        int n;
        for(n = 0; n < info.numSources;n++)
        {
                sourceList[n]->PrepSave(stream); // register source instances
        }
}
 void RDSPObject::Save(FStream &stream)
{
        // save global object data
        stream.WriteData(&info,sizeof(RDSPObjectInfo),1);
        int n;
        for(n = 0; n < info.numSources;n++)
        {
                stream.SaveObject(sourceList[n]); // save source instances
        }

}
void RDSPObject::Load(FStream &stream)
{
        // load global object info
        stream.ReadData(&info,sizeof(RDSPObjectInfo),1);

        int n;
        for(n = 0; n < info.numSources;n++)
        {
                sourceList.Add((RSource*)stream.LoadObject());// load source instances
        }
}
```

Firstly note use of the PERSISTENT_OBJECT_SOURCE macro, again passed the class name. _SOURCE in this case represents 'source code' because this macro must be used in a C++ source file. The implementation of PrepSave() first calls the base class PrepSave() with the current stream causing registration and correct storage of this object instance. BaseDSPObject then goes on to register its child objects, in this case all of the sources. In an object that has no child persistent objects the overloading of PrepSave() is not necessary as the base class provides the necessary functionality. Save() and Load() are similar using FStream::WriteData() and FStream::ReadData() to write and read blocks of non dynamic class data. Storage of child objects is accomplished with

FStream::SaveObject() called in this case for all sources. FStream::LoadObject() is used to load each source and perform the pointer cross reference before adding to the source list.

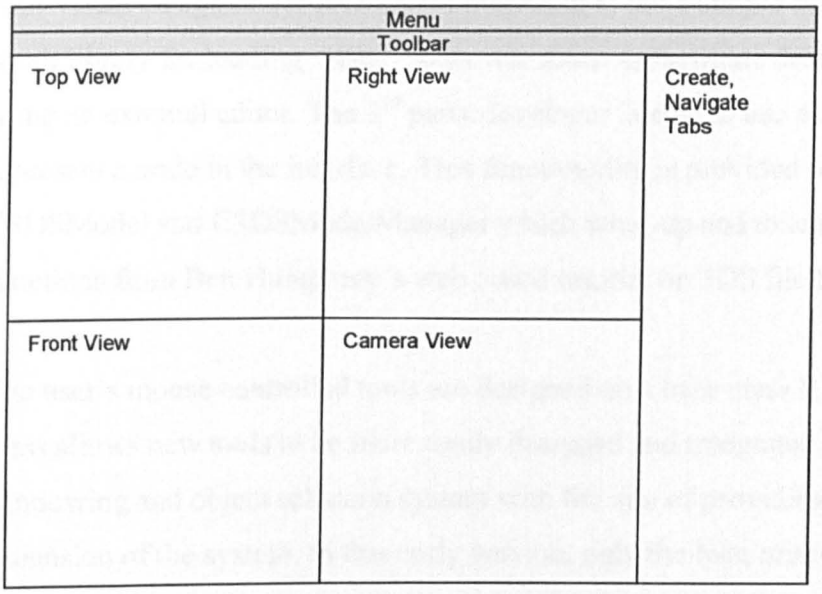
5.7 *'Ricochet' General Implementation Methods*

A complete line by line explanation of Ricochet's implementation is of little relevance to the core of this thesis. In lieu of this the following section provides only an overview of the general implementation methods and ongoing development areas. It should be noted however, that although the methods are less relevant to the SRP technique, many are still vital to production of a complete software application.

The Graphical User Interface (GUI) for 'Ricochet' was developed using a combination of the Microsoft Foundation Classes (MFC) and the Win32 API. Comprehensive coverage of these APIs is given in Petzold, Brown, Feuer, Gregory and Jones[105]. The Win32 API was used as a base for the ricochet plugins' own GUI interfaces. It was intended that the ricochet SDK should not force the use of MFC.

The design of the GUI was loosely based on Discrete's '3D Studio Max' software, a tool for 3D graphics modelling and rendering. The basic principle of the design is four windows displaying views from different 'cameras' in the 3D world. These windows provide a CAD like design space where action in one view is immediately shown in all the others. A menu and toolbar provide access to the commands in the interface. Creation of objects and direct navigation is provided in a tabbed window. A tabbed properties window shows the properties of the currently selected object. This diagram describes the basic layout:

[105] Petzold, C: 1999; Brown, S: 2000; Feuer, A. R: 1997; Gregory, K: 1997; Jones, R. M: 2000



As discussed earlier, the implementation of 'Ricochet' makes use of OpenGL to provide the 3D graphics rendering required by the user interface. Integration of MFC and OpenGL was performed by creating an extension to the MFC CWnd class called GLViewWnd. Each view is created with the GLViewWnd class with all other GUI features created by extensions to other MFC classes. The OpenGL implementation makes use of many extensions to the GLSGNode class. These extensions provide for both rendering of 3D meshes and selection of objects. Node selection is performed with OpenGL's pick matrix and off screen rendering. Selection of groups of objects is achieved with a special GLSGNode which is inserted into the Scene Graph.

In order to provide user interface features to plugin developers without forcing every developer to use MFC, it was decided to create a simplified set of GUI interface classes and functions that would integrate with the main interface automatically. This method provided the best solution for a researcher wishing to produce a simple experimental SRP object. It should be noted however, that the 3rd party developer is not required to use the simple GUI library and may use either Win32 or MFC if so desired. This simple GUI was created using only the raw Win32 command set. All classes and functions relating to the simple GUI are contained in the files PluginGUI.h and PluginGUI.cpp. These are compiled and linked into the dynamic link library PluginGUI.dll.

Non-visual design of 3D meshes for rendering is not a simple task and therefore a small library for loading .3DS[106] files was made to facilitate design of meshes using an external editor. The 3rd party developer is able to use a .3DS file mesh to represent a node in the interface. This functionality is provided by the classes C3DSModel and C3DSModelManager which wrap-up and extend classes and functions from Ben Humphrey's web based tutorial on 3DS file loading[107].

The user's mouse controlled tools are designed on a base class RTool. This base class allows new tools to be more easily designed and integrated within the windowing and object selection system with the aim of providing for future expansion of the system. In this early version, only the bare minimum tools for creating, selecting, moving, rotating and scaling are provided.

In order to implement the modular plugin architecture the system makes use of Microsoft Windows Dynamic Link Libraries (DLLs)[108]. The Windows API provides functions that can obtain a pointer to a function contained in a DLL. 'Ricochet' searches through all DLLs contained in its plugins directory and attempts to obtain a pointer to a function called RPluginMain(). If the function exists then Ricochet runs it and the plugin is then registered together with any additional functionality. The RPluginManager class is designed to perform all the necessary registration of plugins. This technique is used in all of the plugin architectures looked at during research and is well discussed in Steinberg's VST SDK. It would have been simpler to provide an architecture that forced one plugin to provide only one additional object class to the system but it was felt more useful to provide the option of creating a whole library of additional classes in one DLL. A single plugin may register many classes with the active RPluginManager.

Internal reporting macros and functions were designed initially to aid run time debugging of the main system but this functionality has been purposely left in the system to aid the 3rd party developer. These features can produce run time reports

---

[106] Discrete's 3DS Max export format supported on a number of other 3D applications

[107] Humphreys, B: 2002

[108] Williams, A: 2000, p.15

to a debug window and will automatically be skipped when compiling a release version.

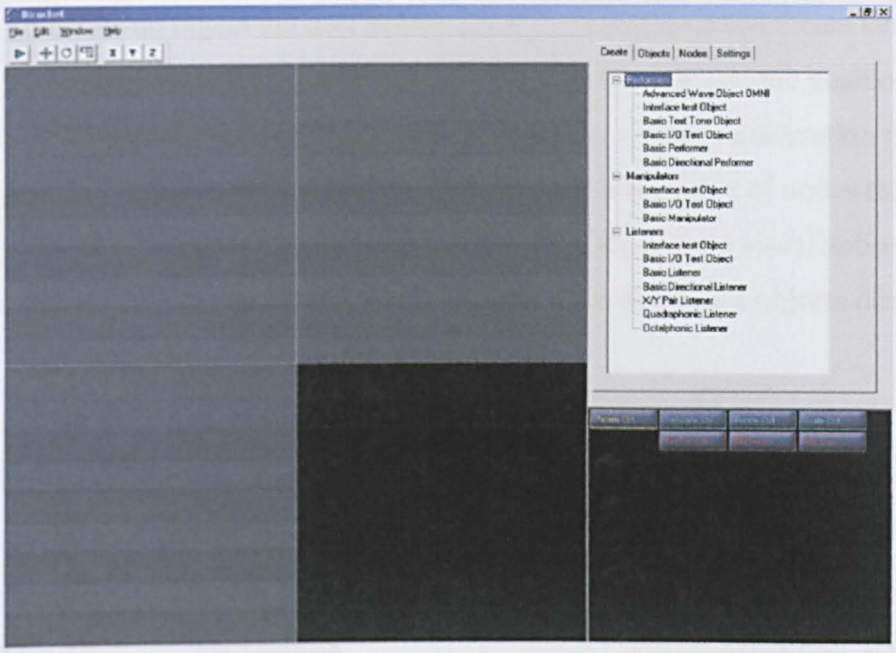
A future extension to 'Ricochet' could be the integration of Steinberg VST plugins and the beginnings of an appropriate VST plugin host are provided in the as yet incomplete CVSTPlugin class.

5.8 Ricochet user manual

5.8.1 Initial boot up

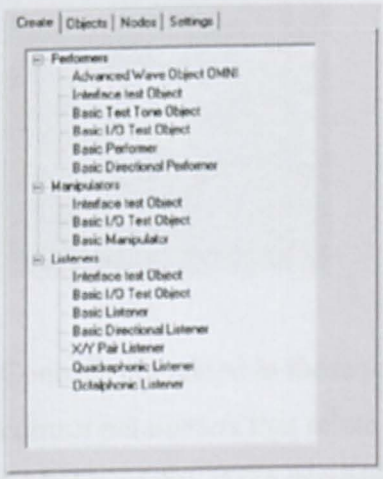
After executing the Ricochet application a small splash screen is displayed before reaching the main page. The splash screen displays the available RSDK plugins as it detects them on the lower display line. Below, the screenshot shows the splash screen after successfully loading plugins.

The main page of Ricochet is presented immediately after the splash screen loading and initially displays as follows:

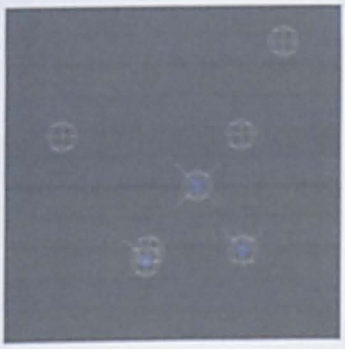


The four main OpenGL editing panes form the majority of the interface and provide the user with Top, Left, Front and Perspective views in order to visualize the 3d sound space. In the upper right of the screen the 'Create' tab shows a tree view displaying a categorized list of the available objects for creation. The 'Objects' tab shows objects that have been created and the 'Nodes' tab displays the complete world node hierarchy. Settings can be adjusted from the 'Settings' tab.

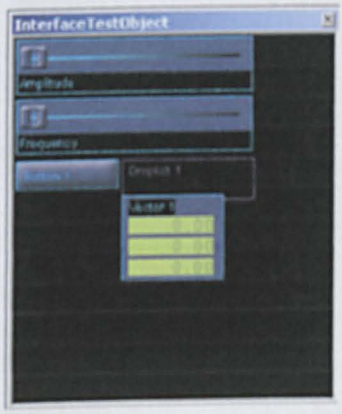
5.8.2 *Creating objects: Create Tab*



To create an object the user selects an object class from the create tab and clicks within any of the edit panes. All 3d object classes will use the position of the click to establish a creation location and will display a node visualization graphic in the edit panes. Visualization graphics are related to the types of nodes that are created. If an object creates 8 nodes then the user will see 8 node visualizations in the edit pane. The following diagram shows a selection of created objects displayed in the 'Top' edit pane.



Once created, the object will open its parameter window and this window may be moved to a desired location. An object's parameter window displays the editable parameters that relate to the whole object such as pickup gain or directionality. The screenshot below shows a test interface object class purely for development purposes but it highlights a typical interface GUI

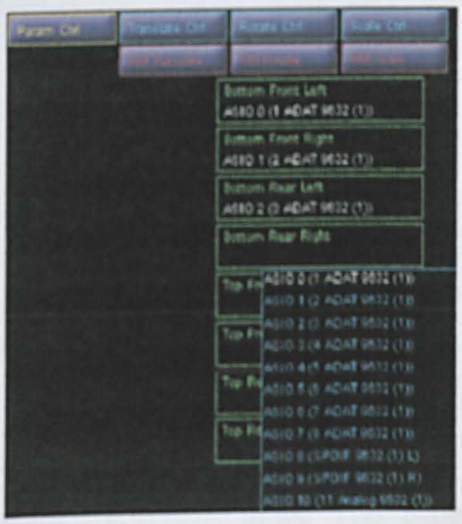


Controls contained in these popup interfaces operate in real time and typically control parameters that relate to DSP or other aspects not encompassed by the node hierarchy. If the parameter GUI disappears from view then the 'Param Ctrl' button will bring it back into focus.

5.8.3 Selecting DSP Objects: Objects Tab

Created objects are displayed in the objects tab under the processing categories Performers, Manipulators and Listeners. Selecting an object in this view causes it to become the object of focus for the routing window described below.

5.8.4 Routing DSP Objects: Routing Window

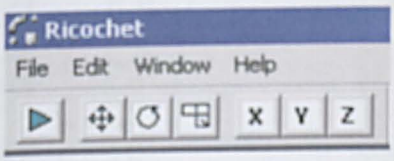


The above screenshot shows the routing window for an octaphonic listening object. The blue context menu appears after clicking on any of the virtual output controls and allows routing of the virtual output to a physical output. In the

screenshot below virtual input routing is shown. Note: Some object classes may have both inputs and outputs.

5.8.5 *Processing, selecting and manipulating nodes*

A 3d node can be selected by clicking on its representation in the edit pane. Multiple nodes may be selected by clicking and dragging to form an encompassing rectangle. Once a selection is made the user is able to manipulate it using the basic tools in the toolbar (shown below).

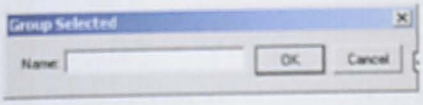


The 'Play' button is the processing on/off toggle and this determines whether or not audio and controller processing is running. When processing is off no sound is heard and the animated controllers remain static.

The 'Translate' tool allows the selection to be moved within the plane of the current edit pane, in the 'Top' pane this represents a movement in the XZ plane of the modelled environment. Rotate and scale tools make use of the axis lock buttons X, Y and Z. When using the rotate tool the axis of rotation is selected using the axis lock buttons, clicking and moving the mouse up and down directly

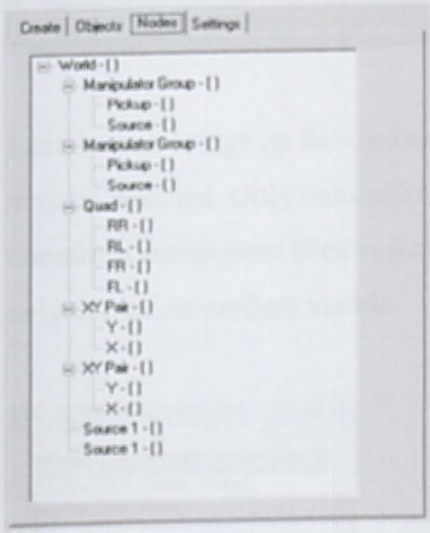
rotates the selection group around the selected axis. The scale tool works in a similar way but the axis lock selects the axis along which scaling is applied.

Any selection of nodes can be grouped at any time by selecting Edit->Group. A dialog is displayed for the entry of a group name (shown below). Note: grouped nodes cannot be ungrouped in the current version of the interface.

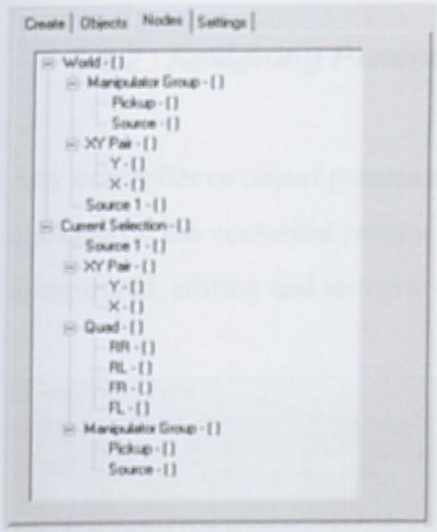


Groups can be selected in the same manner as any other node. If a node is selected its highest level encompassing group is selected. It is not currently possible to reopen a grouped node.

5.8.6 Viewing and Controlling Nodes: Node Tab



The node tab (above) displays the complete node/group hierarchy based on the world node. When a node selection is made the tab also displays the selection node which forms the mechanism for node manipulation. Selected nodes will be removed temporarily from the world node and added to the selection node. Upon de-selection, the transform contained in the temporary selection node is applied directly to all selection child nodes. The original world node hierarchy is then restored.



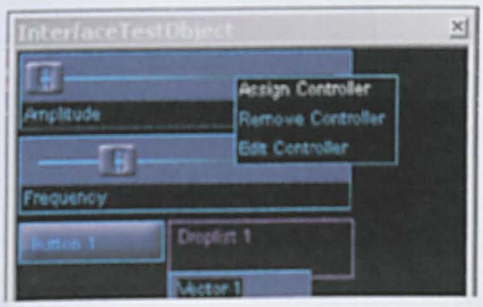
Nodes can be highlighted in the node tab view. A highlighted node can have controllers applied to it via the controller assign buttons.

Each button brings up the controller assign dialog and a vector controller class may be selected. Only controllers that have the capability of acting on the selected transform component (Scale, Rotate or Translate) are visible in the dialog. Note: only vector controllers visible.

Upon selecting a controller class a new controller instance is created and its editing GUI is shown in a popup window.

5.8.7 Assigning Parameter Controllers

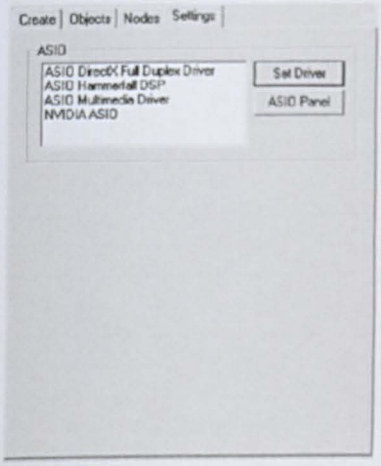
Any controller or object parameter dialog can assign sub-controllers by right clicking on the contained parameter controls. The context menu shown allows assignment, editing and removal of a controller.



Upon assignment a controller selection dialog is shown displaying a list of the available scalar controller classes. Selection of a class results in the creation of a controller instance and display of the controller's popup GUI. A hierarchical tree of controllers can be built up be adding sub-controllers to each GUI. The screenshot below shows the controller selection dialog. Note: only scalar controllers visible.

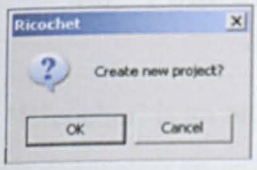
5.8.8 Changing Settings: Settings Tab

An ASIO driver can be selected from the settings tab. The driver's own settings panel can be opened with the Panel button.

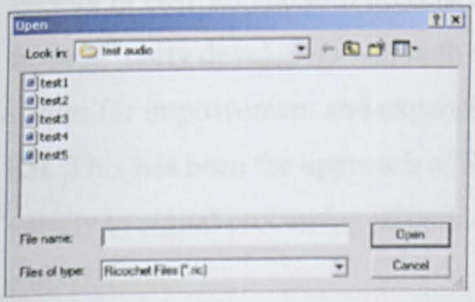


5.8.9 Project Management

At any time, a new blank project can be created by selecting 'File->Project->New'. Selecting 'New' will present the following confirmation dialog:

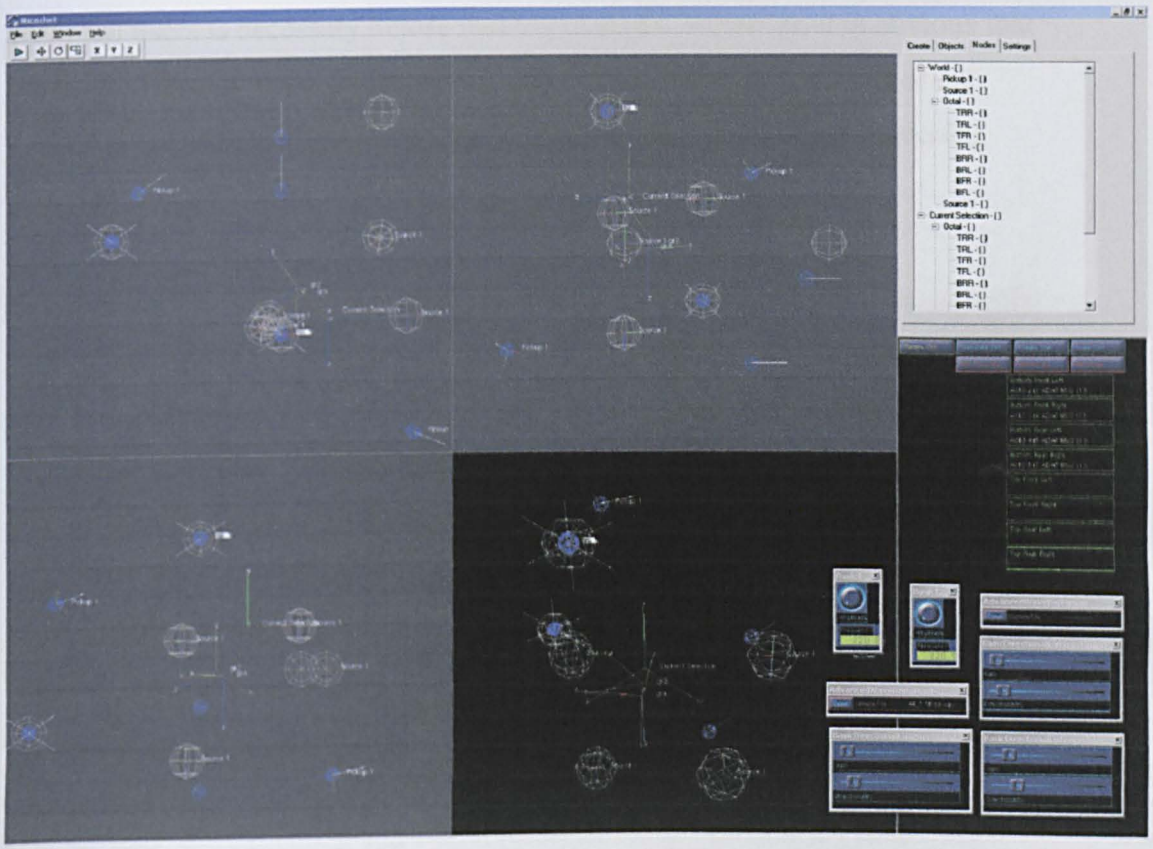


Saving and loading is also achieved from the 'File->Project' menu. Saving or loading a project presents the user with the standard windows file selection dialog in order to choose a file.



A single '.ric' file is created for each saved scene. Upon loading a '.ric' file the system will destroy the currently active project.

Ricochet screenshot showing many created objects multiple selections and a number of open parameter windows.



5.9 *Future expansion of 'Ricochet'*

'Ricochet' is an ongoing project and it has much potential for future development in numerous areas. The software was designed from the start to be modular and the first expansions of the system will likely be in the form of advanced plugin objects. There are many possibilities but it is a reasonable assumption that more variety in DSP and control objects would be the first goal. It is hoped that interest from 3rd party developers will both increase the variety in objects and provide a forum for improvement and expansion of the Ricochet Software Development Kit. This has been the approach of Steinberg in its VST plugin framework and the variety in signal processing of both commercial and non-commercial products is considerable.

Future development of the user interface will strive to improve the usability of the software with respect to the sound artist. At present the graphical interface does not have the benefit of solid user testing and feedback. Serious consultation with potential users is necessary before making changes. However, it is obvious that particular interest will come from the expansion of hardware control of the system, for example, motion tracking of performance and live instrumental control.

A major future goal is improved integration of the software into the sound artist's typical tool box of 3rd party software. This would provide: increased file format support; support for the multitude of DSP plugin formats, VST, DirectX and RTAS; support for inter-process / network audio transmission; direct control by 3D graphics software such as 3DS Max. Much of this work would focus on an improved version of the audio sub system that can provide both simpler use in experimental software and greater functionality.

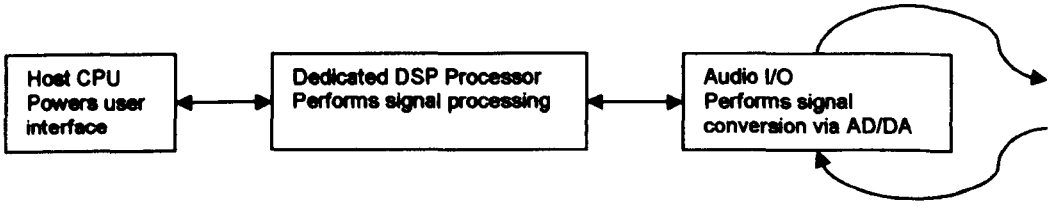
Perhaps though, any future developments should not ask the question 'What software should be designed?' but instead ask 'What do artists want to create and how do they want to create it?'

6 Audio and MIDI libraries

This section describes some of the libraries that were developed and used during the development of Ricochet and Super Diffuse.

6.1 Real-time DSP on the host CPU

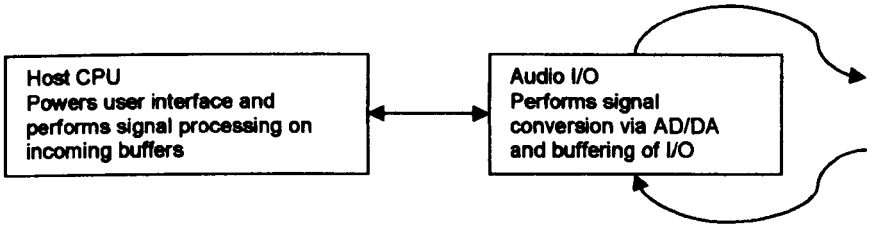
In order to perform real-time signal processing on a modern personal computer there are two techniques and associated hardware. One method assists the 'host' CPU (ie. Intel Pentium or equivalent Motorola), performing all DSP calculations inside an expansion board. This dedicated DSP processing board, sometimes integrated with Audio I/O, is under user control via the host CPU. In a typical system all signal processing occurs on the DSP board and all user interface features are performed on the host.



The technique allows signal processing to be performed with very low latencies (1 sample latency is often claimed) and at very high sample rates (192 kHz on Pro-tools HD at the time of writing)[109]. Of use to the software developer is the ability to perform DSP at a single sample level without concern for other system tasks. If the required latency is 1 sample, the DSP algorithm for a single sample of input must be computed within a single sample rate clock tick. However, since no other processing needs to be performed, using the full power of the chip to complete the DSP algorithm is acceptable. A disadvantage of dedicated DSP is its high cost for both initial purchase and future expansion or upgrade.

[109] Single sample latency at 192 kHz produces a potential 'reaction' time of 1/192000th of a second.

In the 'host based' solution, both DSP and user interface processing are processed simultaneously by the host CPU, AD/DA conversion is performed by an expansion circuit.



Obviously, compared to the dedicated hardware solution, the host CPU is now required to perform significantly extra data processing. More importantly however, the host must also perform two tasks at once. Signal processing in dedicated hardware can be performed continuously but in host based DSP there are many other tasks to be performed simultaneously. These tasks are often time critical and require the CPU to be interrupted. In order to perform DSP on the host CPU it is necessary to buffer the input signal and perform DSP in a burst. Identical performance to dedicated DSP can only be achieved by buffering in single sample blocks but this is not feasible with current CPU speeds and complex DSP algorithms. If, however, buffering is used to split the I/O stream into sample blocks, the host CPU is able to take advantage of various techniques which reduce the processing load. At this point it should be noted that the size of I/O buffers directly affects system latency and the potential for host processing optimisation. It is important to realise that many optimisations, ignoring the I/O latency, are achieved with absolutely identical outcome to a per sample algorithm. However, some have the disadvantage of losing control reaction speed and sample level accuracy. The design of a real-time host DSP algorithm has added complexity because of the need to optimise speed. Although host based DSP is not as accurate and arguably more algorithmically complex than a dedicated system, it is highly cost effective due to the reduced expense on hardware. This reduced cost makes host based processing more accessible to the average user.

6.1.1 Notes on host based optimisation

Host based systems using buffering can take advantage of many optimisations[110] but some common ones are explained as follows. In this case 'cost' refers to 'processing cost' in CPU cycles. Some prior knowledge of software engineering terms is assumed.

- Performing simple calculations on large blocks of data will sometimes take advantage of CPU optimisations such as caching and pipelining, an optimising compiler can also make use of these features.
- Performing calculations once per buffer is less costly than performing them once per sample so it is common to pre-calculate a value and use it for the whole buffer.
- Performing function calls is a costly CPU operation due to pushing of registers into a call stack and program jump overhead. For this reason calling of functions tends to be kept to a minimum on per sample operations. Often, 'macro' functions and the C++ 'inline' concept is used to simulate function based code for the developer but remove it from the compiled algorithm.

6.1.2 Steinberg's ASIO for host based audio I/O

Audio Streaming Input and Output (ASIO) is a library that allows low latency connection to current audio I/O cards. ASIO forms an abstraction layer between audio hardware and software by requiring hardware developers to provide a driver compliant with the ASIO specification and by providing functions to the host software that allow generic control of any driver. This hardware abstraction

[110] Coulter, D: 2000, p.209

technique is common for software connection to other optional card types, for example graphics cards.

In order to develop music software the developer includes the relevant header files and source from Steinberg's ASIO SDK and performs a lengthy but simple initialisation creating audio buffer memory blocks and making a number of function calls to access the selected ASIO Driver. ASIO makes use of a 'call-back' function for the purpose of providing the hardware manufacturers ASIO driver with a function to perform when an audio buffer is ready for processing by the host. In other words the host provides a pointer to a function that is designed to process an audio buffer using the desired algorithm. The host supplied call-back function is required to deal with a number of different audio buffer formats (16 bit integer, 24 bit integer contained in 32 bits, 64 bit floating point etc.).

6.1.3 Packaging ASIO in ASIOSubSystem.dll

In order to facilitate faster development of common audio programs an intermediate library was created to package the ASIO library and a number of conversion routines into a dynamic link library for creating audio programs processing with 32bit floating point. The ASIOSubSystem.dll was one of the earliest pieces of software developed for this research project and was subsequently used and expanded during development of Ricochet and Super Diffuse. The version used with software described in this thesis was tested with a range of audio hardware including the following:

- MOTU: 2408 and 24io
- RME: Hammerfall DSP 96/32 and Multiface
- Creamware: Scope fusion platform
- Creative: Soundblaster 64 and Audigy

In pseudo code, use of the ASIOSubSystem.dll is as follows:

```
Initialise asio and create 32bit float buffers for all inputs and outputs in the system
ASIOBufferCallbackRoutine()
For each input channel buffer
{
```

```
        call conversion routine on this channel from driver format to float
}
call ASIOSubSystem host supplied function with converted float buffers
for each output channel buffer
{
        call conversion routine on this channel from float to driver format
}
```

A typical conversion routine is as follows. Notice the use of 'inline' optimisation to replace function call. This function provides conversion from 32bit integer in most significant bit format to 32bit float (DSPFloat) format used by the host.

```
inline void int32MSB16ToFloat(int *input,DSPFloat *output, int size)
{
        float ratio = 1.0 / 32768.0;
        size++;
        while(--size > 0)
        {
                *output = ratio * (float)*input;
                input++;
                output++;
        }
}
```

Firstly the function pre-calculates (for the whole buffer) a ratio between the maximum integer value 32768 and the DSPFloat maximum value of 1.0 (The compiler actually pre-calculates this ratio at compile time). Each sample in the buffer is multiplied by the ratio to provide the conversion. The '`while(--size > 0)`' iteration is used quite frequently in the DSP code of both projects in this thesis and provides a loop connected to buffer size. By incrementing the size value once the iteration condition is able to use the pre-decrement (--n) method which is faster than the post-decrement (n--).

A typical application produced call-back has the following format.

```
void DSPMain(DSPFloat **inputs,DSPFloat **outputs,long bufferSize)
{
        // Perform application specified per buffer DSP here....
}
```

Setting up an ASIO driver and specification of a DSP algorithm using ASIOSubSystem is as follows. This code section opens a driver, runs the null algorithm defined above for 10 seconds then shuts down gracefully.

```
SetDSPCallback(DSPMain); // set the dsp callback passing a function pointer to the application DSP call-back
InitASIO("ASIO Multimedia Driver",44100.0); // Open the driver called ASIO Multimedia Driver S/R 44.1kHz
StartASIO(); // Inform the driver to begin processing audio
Sleep(10000); // Windows command to wait for 10 seconds
```

```
StopASIO(); // Inform the driver to stop processing audio
CloseASIO(); // Unload the driver and cleanup memory etc.
```

It should be clear from the above that using the ASIO Sub System DLL provides a very quick and simple method for accessing ASIO Drivers.

6.1.4 Future expansion of ASIOSubSystem

Developed during the first few months of research at USSS, ASIOSubSystem has begun to show signs of age despite periodic bug fixes and upgrades. A new version is planned for future research and expansion of other software and a small proposed feature list follows:

- Complete redesign and rewrite of the system;
- Full C++ class implementation dropping current function based system; Inheritance to replace function callback mechanism;
- Full and tested audio card format conversion with emphasis on speed optimization. Optimized DSP functions to be provided to the client;
- Improved browsing of available drivers including a default ASIO setup and selection dialog that can be overridden by client software if necessary;
- Improved browsing of audio card capabilities and the addition of named I/O channels again using a default dialog;
- Per I/O channel default options: allow channels to be disabled, muted and balanced;
- Integral channel routing similar to Ricochet's virtual I/O system;
- Improved client ASIO information system;
- Integral safety features, improved exception handling and automatic driver cleanup in the event of software failure. This is intended to safely catch serious machine shutdown errors that can occur with incorrect use of the current library;
- Integral thread synchronisation features, DSP Mutex / Critical section;
- Built in support for file and network audio streaming, transparent connection with I/O. WAV and OGG Vorbis file support;

- Direct file streaming for all output streams to enable offline rendering direct from the library. This would be performed without the need to re-route channels;
- Integration with MIDIManager system to create a singular audio media I/O system;
- MTC timecode support functionality with disk stream synchronisation;
- Integrated support for 3rd party VST plugins;
- Integrated DSP algorithms such as forward and inverse FFT, FIR and IIR filters, other basic building blocks such as oscillators;

It should be noted that some of the functionality proposed has been researched and even featured in test applications but has not reached a state of sufficient usability to be fully documented.

6.2 *Accessing and Distributing MIDI Information*

The software projects discussed in this thesis make use of MIDI for control and automation. In both, the standard MIDI I/O system provided by Microsoft Windows is used to access hardware. Messick and Penfold detail both the windows MIDI API and the MIDI specification[111]. In order to simplify the setup of drivers and provide a more robust implementation, a pair of classes was created, wrapping the basic Windows Functionality. The MIDIManager and MIDIListener classes defined directly in both applications allow simple addition of MIDI I/O to individual object classes.

The current implementation of the MIDIManager class only allows for single input and output devices to be concurrently open. However, the input may be on a separate device to the output.

The MIDIManager class is intended as a singleton object that forms a central point of reference from which to control the initialisation and shutdown of MIDI

[111] Messick, P: 1998; Penfold, R. A: 1995

devices. Examination of the class header shows the interface methods: (simplified for clarity)

```
class MIDIManager
{
public:
        MIDIManager();
        ~MIDIManager();
        unsigned long GetMessageCount() { return msgCounter; };
        int GetNumInputDevices();
        const char* GetInputDeviceName(int id);
        int OpenInputDevice(int id);
        int GetNumOutputDevices();
        const char* GetOutputDeviceName(int id);
        int OpenOutputDevice(int id);
        int GetControllerValue(int channel, int controller);
        float GetMappedControllerValue(int channel, int controller, float min, float max);
        void MidiInMessage(BYTE status, BYTE dataA, BYTE dataB, DWORD timeStamp);
        void MidiOutMessage(BYTE status, BYTE dataA, BYTE dataB, DWORD timeStamp);
        void CloseDevices();
        void Start();
        void Stop();
        void Halt() { if(isRunning) { Stop(); wasRunning = true; } };
        void Resume() { if(wasRunning) { Start(); } };
        void RegisterMIDIListener(MIDIListener *listener);
        void UnregisterMIDIListener(MIDIListener *listener);
};
extern MIDIManager midiManager; // the one singleton instance of the midi manager
```

Sending MIDI data requires a call to OpenOutputDevice() specifying an id number for the desired device. Devices are assigned a sequential id number, the first being device(0). Manufacturer device names are obtained by calling GetOutputDeviceName() again specifying the id of the desired device. Once a device has been successfully opened, calls can be made to MidiOutMessage() to send short MIDI messages (SYSEX is not supported in this version).

MIDI input devices are initialised in the same manner as MIDI output devices but from then on the task is slightly more complicated and requires the use of the second class MIDIListener. Forwarding incoming MIDI messages to all areas of the program involves a callback mechanism provided though object class inheritance. An object class that needs to be informed of any received messages must inherit from MIDIListener and overload the MidiMessage() function. It is not necessary for any other function calls to be made because the MIDIListener makes the correct references at construction. The following shows the definition of MIDIListener:

```
class MIDIListener
{
public:
        MIDIListener();
        virtual ~MIDIListener();
        virtual void MidiMessage(BYTE status, BYTE dataA, BYTE dataB, DWORD timeStamp) {} ;
};
```

Any class that inherits and overloads correctly has its version of the MidiMessage function called automatically by the MIDIManager whenever a MIDI message is received. At this point 'status', 'dataA' and 'dataB' contain the message and 'timestamp' contains the arrival time of the message specified in milliseconds from system boot up. As an example of use the following code is the overloaded MidiMessage() function from Super Diffuse's CMidiControllerInput class: (Note MIDI status byte conversion functions also written into the MIDI management module)

```
void CMidiControllerInput::MidiMessage(BYTE status,BYTE dataA,BYTE dataB,DWORD timeStamp)
{
        if((MIDIStatusToChannel(status) == midiChannel) &&
                (MIDIStatusToType(status) == MIDI_CONTROL_CHANGE) &&
                (dataA == midiControllerNum))
        {
                if(target)
                        target->SetValueExternal(dataB);
        }
}
```

In the above example it is clear how obtaining and translating a simple Control Change message is not as simple as it might be. It should be realised that reception of MIDI messages in this manner is not particularly efficient as a complete function call is used for every MIDI listener, for every message; in addition, the inherent callback mechanism effectively means that MIDI message reception runs in a separate thread of execution and thread synchronisation needs to be considered. If Control Change messages are the only desired input and no direct notification of message reception is required the MIDIManager class provides an alternative.

The MIDIManger object automatically tracks value changes to all Control Change addresses across all 16 MIDI channels on the current input device. These values are stored in a 16 * 128 array for later access. This method does not suffer from thread synchronisation problems. A call to GetControllerValue() specifying midi channel and CC id returns the last sent value of the specified Control Change address. Additionally the manager provides GetMappedControllerValue() which automatically maps from the MIDI CC range (0..127) to the range specified. As

an example the following code is the Update() function from Ricochet's MIDIScalarController class:

```
void MIDIScalarController::Update(long time)
{
        RController::Update(time);
        value = midiManager.GetMappedControllerValue((int)GetParameter(0).GetValue(),
                                                      (int)GetParameter(1).GetValue(),
                                                      GetParameter(2).GetValue(),
                                                      GetParameter(3).GetValue());
}
```

In the above the Update() function 'polls' GetMappedControllerValue to obtain a pre mapped value; RParameters are used to set range, channel and CC address.

7 Conclusions

7.1 Super Diffuse and M2

M2 and its control software 'Super Diffuse' has successfully diffused during real concert performances and at the time of writing it, or its successor M3, is scheduled to 'perform' in the upcoming USSS Sound Junction 2005. Comments from users have been both favourable and constructive. M2 has resulted in the formation of a diffusion software development partnership between the Universities of Bangor, Belfast, Birmingham, Edinburgh and Sheffield and this in itself is a very exciting future prospect. The presentation of *'M2 Diffusion – The live diffusion of sound in space'*[112] at ICMC 2004 is expected to promote further development interest.

7.2 SRP and 'Ricochet'

The 'Source Ray Pickup Interaction' concept for modelling the 'Virtual Sound Environment' has provided a promising expansion of the basic waveguide network. By classifying and implementing distinct entities for spatial signal processing a modular spatial waveguide network framework has been conceived.

The 'Ricochet' project creates a practical test bed for an SRP interaction model. The testing has proved that the SRP technique can be implemented successfully in a real-time host based system. Modularity of the Ricochet Software Development Kit has enabled 3rd party exploration of the SRP model without the need to redevelop an entire framework.

---

[112] Moore, A; Moore, D; Mooney, J: 2004

'A model is a simplification of reality.'[113] However, this model can be improved and refined; it is hoped that 'Ricochet' will act as a basis for future expansion of the SRP model and as an example of general implementation techniques for the 3D sound spatialization models of the future.

7.3 In summary

From its inception in 1999, this project has set out to produce new tools for spatialization of sound. Two tools have been successfully implemented and offer sound artists new ways of working. In creating new tools the possibilities for creativity have been expanded but it is the products of such tools that will determine their usefulness. It is hoped that the products of this research will further future creative possibilities.

7.4 Personal Reflection

As I write this thesis and consequently reinvestigate deep-rooted aspects of the software projects, I find myself wishing to redesign these more elegantly or for increased features. This is clearly a factor of my increasing programming experience, but also reflects my ongoing interest in this field. If time were limitless I would explore many more of the ideas described in the futures section of each project. However, I expect to find that all explored avenues will create yet more future possibilities. Time is a precious commodity and I have found it difficult to stop modifying or tweaking projects in order to present this 'snapshot' of perhaps limitless research scope. I am hopeful that the requirements of working in a team will force focused and well constrained development plans that will further the research in steady stages.

The M2 system specifically has been a highly successful project for the University of Sheffield Sound Studios. Interest from third parties in both industry and

[113] Booch, G; Rumbaugh, J; Jacobson, I: 2003, p.6

academia has been exceptional, much greater than I had expected. At the time of writing, the studios have had development interest from Richmond Audio and have entered a collaboration agreement with DACS Ltd to co-create a specialized control surface for diffusion. Links created from this collaborative venture provide a basis for future research in both Ricochet and Super Diffuse. I hope that this forum will eventually lead to a concept for a spatialization tool that unifies and extends the projects presented here.

Perhaps most pleasing are the comments I have had from users. At Bangor, Andrew Lewis made the decision to use the M2 system in place of the system that had been previously set-up. I felt that this made a strong case for the significance of the project. Jonty Harrison of Birmingham was asked about his interest in the M2 system by James Mooney during a recorded interview and had these complimentary words on M2 relating to his visit to Sheffield *'...I think it's fascinating and the control is fantastic – and I want one...'*. Later he enquires about a Macintosh version and talks about some of the things he would explore with the M2 system in its current state. Harrison seems very keen to explore sound diffusion using the M2 system and provide very constructive feedback for future versions. I think that this is an extremely positive outcome from the project and I expect it to further potential research at Sheffield for some time.

References

Akenine-Möller, T; Haines, E. (2002). *Real-Time Rendering Second Edition.* Natick, Massachusetts: A K Peters.

Baudelaire. (1997). 'Presentation of the Gmebaphone concept and the Cybernephone instrument.' in *Composition Diffusion en Musique Electroacoustic Volume III.* Academie Bourges: Editions MNEMOSYNE. pp. 266 - 281

Booch, G; Rumbaugh, J; Jacobson, I. (2003). *The Unified Modeling Language.* Boston: Addison Wesley.

Bourg, D. M. (2002). *Physics for Game Developers.* Sebastopol: O'Reilly.

Brown, S. (2000). *Visual Basic Developers Guide to the Win32 API.* San Francisco: Sybex.

Chowning, J. (1971). *'The Simulation of Moving Sound Sources.'* http://ccrma.stanford.edu/courses/220a-fall-2001/chowning.pdf

Copeland, D. (2000). 'The Audience At The Center: Diffusion Practice at Sound Travels.' Toronto. http.//www.soundtravels.ca/soundtravels/2003/audcen.html

Coulter, D. (2000). *Digital Audio Processing.* Lawrence, Kansas: R & D Books.

Davis, G; Jones, R. (1990). *Sound Reinforcement Handbook Second Edition.* Milwaukee: Hal Leonard.

Dobrian, C; Zicarelli, D; Puckette, M. (1997). *MSP The Documentation.* San Francisco: Cycling '74.

Eberly, D. H. (2001). *3D Game Engine Design. A Practical approach to Real-Time Computer Graphics.* San Francisco: Morgan Kauffman.

Everest, F. A. (1994). *The Master Handbook of Acoustics 3rd Edition.* New York: TAB Books McGraw Hill.

Farrell, B. (2001). 'Adrian Willaert (1490-1562).' http://www.ptloma.edu/music/MUH/composers/willaert/willaert.htm

Feuer, A. R. (1997). *MFC Programming.* Reading, Massachusetts: Addison Wesley Longman Inc.

Gregory, K. (1997). *Special Edition Using Visual C++ 5.* Indianapolis: Que Corporation.

Harrison, J. (1999). 'Diffusion: theories and practices, with particular reference to the BEAST system.' http.//cec.concordia.ca/econtact/Diffusion/Beast.htm

Humphreys, B. (2002). '3DS File Loader Tutorial.' http.//www.gametutorials.com/Tutorials/opengl/OpenGL_Pg4.htm

Jones, A; Ohlund, J. (1999). *Network Programming for Microsoft Windows.* Redmond, Washington: Microsoft Press.

Jones, R. M. (2000). *Introduction to MFC Programming with Visual C++ .* New Jersey: Prentice Hall PTR.

Lengyel, E. (2002). *Mathematics for 3D Game Programming and Computer Graphics.* Hingham, Massachusetts: Charles River Media.

Love, R. (2003). *Linux Kernel Development.* Indianapolis: Sams Publishing.

Malham, D.G. (1998). 'Approaches to spatialisation.' *Organised Sound.* 3(2). pp.167-177

Malham, D.G. (2000). 'Homogeneous and Nonhomogeneous Surround Sound Systems.'
http://www.york.ac.uk/inst/mustech/3d_audio/homogeneous.htm

Messick, P. (1998). *Maximum MIDI Music Application in C++* . Greenwich, Connecticut: Manning Publications.

Mooney, J. (2000). 'Ambipan and Ambidec - Towards a Suite of VST Plugins with GUI for Positioning Sound Sources Within an Ambisonic Soundfield in Real-time.' York.
http://www.york.ac.uk/inst/mustech/gsp/mustech_projects.html

Moore, A; Moore, D; Mooney, J. (2004*). 'M2 Diffusion – The live diffusion of sound in space.'* Miami: ICMC 2004.

Palombini, C. (1999). 'Musique Concrète Revisited.'
http://www.rem.ufpr.br/REMv4/vol4/arti-palombini.htm

Penfold, R. A. (1995). *Advanced MIDI User's Guide Second Edition.* Kent: PC Publishing.

Petzold, C. (1999). *Programming Windows Fifth Edition.*
Redmond, Washington: Microsoft Press.

Poli, G; Rocchesso, D. (1998). 'Physically based sound modelling.' *Organised Sound.* 3(1). pp. 61-76

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

Roads, C. (1997). 'Musical Space: the virtual and the physical.' *Composition Diffusion en Musique Electroacoustic, Volume III*. Academie Bourges: Editions MNEMOSYNE. pp.158-160

Rolfe, C. (1999). 'A Practical Guide To Diffusion.' http.//www.soundtravels.ca/soundtravels/2003/difpract.html

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

Schildt, H. (1998). *C++: The Complete Reference Third Edition.* Berkeley: Osborne McGraw Hill.

Sedgewick, R. (1999). *Algorithms Third Edition in C++* . Reading, Massachusetts: Addison Wesley.

Serway, R. A. (1996). *Physics for Scientists and Engineers with Modern Physics Fourth Edition.* Philadelphia: Saunders College Publishing.

Smith, J. O. (2002). 'An Acoustic Echo Simulator.', 'Converting Propagation Distance to Delay Length.', 'Spherical Waves from a Point Source.' *Physical Audio Signal Processing.* http.//www.ccrma.stanford.edu/~jos/waveguide/...

Spain, M; Polfreman, R. (2001). 'Interpolator: a two-dimensional graphical interpolation system for the simultaneous control of digital signal processing parameters.' *Organised Sound.* 6(2). pp.147-151

Stark, S. H. (1996). *Live Sound Reinforcemen: A Comprehensive Guide to P.A and Music Reinforcement Systems and Technology.* Emeryville, California: Mix Books.

Steinberg Soft- und Hardware GmbH. (1999). 'Visual Studio Technology Plug-In Specification 2.0 Software Development Kit Documentation Release #1.' Germany: Steinberg.
http.//www.steinberg.de (By application only)

Steinberg Soft- und Hardware GmbH. (1999), 'Audio Streaming Input Output Specification, Development Kit 2.0 Document Release #1.' Germany: Steinberg.
http.//www.steinberg.de (By application only)

Stevenson, I. (2000), 'Diffusion: Realisation, Analysis and Evaluation', ACMC
http://www.mikropol.net/volume6/stevenson_i/stevenson_i.html

Stroustrup, B. (2000). *The C++ Programming Language Third Edition.* Reading, Massachusetts: Addison Wesley.

Truax, B. (1997). 'Composition & Diffusion. Space In Sound In Space.' Bourges.
http.//www.sfu.ca/~truax/bourges.html

Williams, A. (2000). *Windows 2000 Systems Programming Black Book.* Scottsdale, Arizona: Corolis.

Wishart, T. (1994). *Audible Design.* UK: Orpheus the Pantomime Ltd.

Woo, M; Neider, J; Davis, T; Shreiner, D. (1999). *Open GL Programming Guide Third Edition.* Reading, Massachusetts: Addison Wesley.

Wyatt, S. A. (1999). 'Investigative Studies on Sound Diffusion/Projection.' Illinois.
http.//cec.concordia.ca/contact/Diffusion/Investigative.htm

Zicarelli, D. (1998). 'How to Write MSP Externals Revision 3 of 1.'
http.//www.cycling74.com

Zvonar, R. (1999), *'A HISTORY OF SPATIAL MUSIC'*
http://www.zvonar.com/writing/spatial_music/History.html