Ambipan and Ambidec

Towards a suite of VST plugins with graphical user interface for positioning sound sources within an ambisonic sound-field in real time

James Mooney

MSc Music Technology

Supervisors: Dr Damian Murphy and Dr Ambrose Field

31st August 2001
Preface

As the volume of this report testifies, the task of producing VST plugins for ambisonic panning and decoding and the design and implementation of a graphical user interface for the former, has been an unpredictably enormous undertaking. With hindsight it is clear that the two key areas of development – VST plugins, and GUIs – are really two quite distinctly separate areas, each fully deserving of a dedicated project and report of their own. Nonetheless, the final outcome of the project has been very productive, and although the GUI has not yet been integrated with the plugin, good progress has been made in all areas of research and development.

The first five chapters of this report summarise the key areas of research from which this project has been derived. These are intended as relevant background reading, but nonetheless, a basic understanding of the subject matter is assumed in subsequent chapters. Chapters 6 to 9 explain the design and implementation of the plugins and user interface. The basic concepts of the VST paradigm are explained in fairly abstract terms and then with more specific reference to the implementation of the plugins. Similarly, OpenGL is described first in its own right, and then in its capacity as the graphics engine with which the Ambipan interface was realised. Chapters 10 and 11 detail the testing procedure under which the correct functioning of the plugins and interface was verified. Some proposed future developments are also cited.

Readers may find it useful first to refer to the summary conference paper (given in Appendix IV), which briefly summarises the main aspects of this research. Source code for the plugins and graphical user interface is given in Appendices I, II, and III.

The attached 3.5” floppy disk contains Win32 dynamic link libraries (DLLs) for each of the plugins. These should be copied into the VST-plugins folder accessed by Nuendo. Note that Steinberg’s Nuendo is the only VST host capable of more-than-two-channel output, and so represents the only application to date suitable for running the Ambipan and Ambidec plugins. The GUI for Ambipan is supplied on the floppy disk as an executable console application, again for Win32.
Acknowledgements

I would like to thank those members of the VST-Plugins and Sursound mailing lists whose consistently helpful advice has been very much appreciated, and likewise members of the OpenGL and Nuendo public forums. Specific thanks to the Dr Math team for their prompt and concise advice on mathematical problems that I (perhaps naively) believed would never trouble me again beyond high school. Thanks to my supervisors Damian Murphy and Ambrose Field for their support and enthusiasm, and also to Rob Fletcher of Computing Service and Dave Malham of the Music Department for their help with problems related to OpenGL and ambisonics, respectively. Obviously a mention is due, last but not least, to fellow York University VST developers Sa, Rich, Thibaut, Chris and Bruce for providing strength in numbers!
Contents

1.1 Interaural Time and Intensity Differences ............................................................... 9
1.2 Head Related Transfer Functions............................................................................... 11
1.3 Ratio of Direct Sound to Reverberant Sound ........................................................... 12
1.4 High-Frequency Attenuation ..................................................................................... 13
2.1 Stereophonic.............................................................................................................. 14
2.2 Quadraphonic........................................................................................................... 16
2.3 Dolby 5.1 .................................................................................................................. 17
2.4 Binaural and Transaural........................................................................................... 19
3.1 The Four Stages of Ambisonic Encoding and Decoding ........................................... 21
3.1.1 Ambisonic A-Format ............................................................................................. 21
3.1.2 Ambisonic B-Format............................................................................................. 22
3.1.2.1 Artificial Synthesis of B-Format Signals ............................................................. 23
3.1.3 Ambisonic D-Format ............................................................................................. 23
3.1.4 Ambisonic C-Format ............................................................................................. 25
3.2 Sound-field Transformations .................................................................................... 26
3.3 Some Limitations ...................................................................................................... 26
3.4 Ambisonics ‘vs’ Dolby 5.1 ....................................................................................... 27
4.1 Offline Applications ................................................................................................... 34
4.2 Real-Time Applications ............................................................................................ 38
5.1 Issues of 3D Control ................................................................................................. 42
5.2 APIs for Creating Graphical User Interfaces ........................................................... 44
5.2.1 VSTGUI ................................................................................................................ 44
5.2.2 XForms ................................................................................................................. 45
5.2.3 FOX GUI ............................................................................................................... 45
5.2.4 GLUI ..................................................................................................................... 46
6.1 Ambipan Specification .............................................................................................. 47
6.1.1 Ambisonic Panning of Stereo Input Signals ......................................................... 47
9.3.3 Determining the Sound-Source Position using Cartesian Coordinates .............................................. 99
9.3.4 Determining the Sound-Source Position using Polar Coordinates ...................................................... 101
9.3.5 Deriving Sound-Source Coordinates in Code .......................................................................................... 104
9.3.6 Placing the Sound-Source on each Sound-Field Elevation .................................................................... 105
9.3.7 Control of the Stereo Height and Stereo Width Sliders ....................................................................... 107
10.1 Calibrating Nuendo .................................................................................................................................. 109
10.1.1 Calibrating the Input Gain .................................................................................................................... 109
10.1.2 Calibrating the Effect Return Gain ......................................................................................................... 111
10.2 Testing Ambipan ...................................................................................................................................... 113
Test 1: Azimuth = 90°, Elevation = 0° .................................................................................................................. 114
Test 2: Azimuth = 225°, Elevation = 0° .............................................................................................................. 115
Test 3: Azimuth = 0°, Elevation = 90° ................................................................................................................ 116
Test 4: Azimuth = 0°, Elevation = 135° .............................................................................................................. 117
Test 5: Azimuth = 217°, Elevation = 311° ........................................................................................................... 118
Test 6: Azimuth = 128°, Elevation = 37° ............................................................................................................. 119
10.2.1 Summary of Tests 1 – 6 ......................................................................................................................... 120
10.3 Testing Ambidec ........................................................................................................................................ 120
Test 7: Mono Decode ....................................................................................................................................... 121
Test 8: Elevation Increased Linearly from 0° to 360° over 10 Seconds – Stereo Decode ................................. 121
Test 9: Azimuth Increased Linearly from 0° to 360° over 10 Seconds – Stereo Decode .............................. 122
Test 10: Azimuth Increased Linearly from 0° to 360° over 10 Seconds – Square Decode ............................. 122
Test 11: Azimuth Increased Linearly from 0° to 360° over 10 Seconds – 3/2 Decode ................................. 123
Test 12: Azimuth Increased Linearly from 0° to 360° over 10 Seconds – Cube Decode ............................... 123
Test 13: Elevation Increased Linearly from 0° to 360° over 10 Seconds – Square Decode .......................... 124
Test 14: Elevation Increased Linearly from 0° to 360° over 10 Seconds – 3/2 Decode .............................. 124
Test 15: Elevation Increased Linearly from 0° to 360° over 10 Seconds – Cube Decode ............................. 125
10.4 Testing the Ambipan GUI .......................................................................................................................... 125
Test 16: 0° Azimuth, 0° Elevation .................................................................................................................... 126
Test 17: 90° Azimuth, 0° Elevation ................................................................................................................... 127
<table>
<thead>
<tr>
<th>Test</th>
<th>Azimuth (°)</th>
<th>Elevation (°)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>180</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>270</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>90</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>180</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td>270</td>
</tr>
</tbody>
</table>

11.1 Implementation of Ambipan as a Spatialiser Plug-In with GUI.........................131
11.2 More Creative use of the Preset Programs in Ambipan.....................................132
11.3 More Flexible Decoding Options ........................................................................133
11.4 Collation of all Spatialised Sources in a Single Display Module............................133
11.5 Implementation of Distancing Laws.......................................................................134
11.6 Second Order Support...........................................................................................135
Introduction

As a composer interested in the creative possibilities of sound in space, I have noticed that surround sound technologies have been directed almost entirely towards the consumer market. This essentially precludes any kind of creative implementation of surround sound systems. Further, of those applications and technologies that do exist, few – if any – are genuinely conducive to the creative process. Many computer applications, for example, exist only as stand-alone programs, forcing the composer to leave their chosen mixing environment and carry out spatial processing externally. Further, a large proportion of the software available operates on an offline (non-real-time), command-prompt basis, effectively removing any elements of spontaneity and experimentation from the exercise.

It is for these reasons that I have decided to develop software for panning sound sources in real-time in an ambisonic sound field, with the explicit objective of ease-of-use. An additional focus of my research will be the development of an intuitive graphical user interface (GUI) for the movement of source sounds in three dimensions. Further, it is intended that the software will be fully functional with minimal hardware requirements: composers should be able to compose spatially without the need for specialised or novel input devices.

This report covers the background research, design, and implementation of two VST plugins – Ambipan and Ambidec – and a graphical user interface for the former. This software has been developed as a compositional aid for the positioning of sound sources within a three-dimensional sound-field in real time. The chosen paradigm for the sound spatialisation is the ambisonic system, which is described in Chapter 3. The Ambipan plugin and its associated interface are designed for positioning sound sources within a sound-field and performing ambisonic encoding in real-time. The Ambidec plugin is designed for carrying out ambisonic decoding to a variety of preset speaker configurations, again in real-time.
Fundamental to the whole field of sound spatialisation are the physical and psychoacoustical phenomena which explain the way humans perceive sound. It is important to have at least a basic understanding of these principles before going on to examine the other areas which will be encountered throughout the course of this research. A concise, but thorough, introduction to the subject is given by Kendall (1995), while a more in-depth accounts are given by Blauert (1983), and Howard and Angus (1996). Gerzon (1974) is also a good reference, although perhaps slightly dated.

Broadly speaking, there are five main acoustical phenomena that inform the way humans are able to understand the location of a sound source without visual cues. These are as follows:

1. interaural time difference (ITD) (and interaural phase difference);
2. interaural intensity difference (IID);
3. head-related transfer functions (HRTFs);
4. ratio of direct sound to reverberant sound;
5. acuity of high-frequency attenuation;

Items 1 through 3 are concerned with the directionality of the sound source, while 4 and 5 are related to the distance between the source and the listener. This chapter will briefly explain these principles, in order that their impact on the (re)creation of surround-sound environments may be understood.
1.1 Interaural Time and Intensity Differences

Due to the distance between the ears and the shading effect of the head, sound waves are likely to arrive at each ear at slightly different times, and with slightly different intensities. This is because sound travels at a finite speed ($344 \text{ ms}^{-1}$), and sound energy is naturally absorbed by air and (more so) by the head. These effects are known as the interaural time difference (ITD) and the interaural intensity difference (IID). Sound sources in different positions with respect to the listener will accordingly result in different ITDs and IIDs, and this plays an important role in the way humans are able to perceive the directionality of sound.

It has been proved that the extent to which ITD and IID contribute to one’s perception of the directionality of a sound source is frequency-dependent. At frequencies below around 700 Hz, the IID is negligible. This is because the wavelength of a 700 Hz frequency is roughly twice as long as the average distance between the ears, so the head does not absorb a significant amount of energy at frequencies lower than this (Gerzon 1974: 184). At such frequencies it is primarily the ITD and resulting interaural phase difference that informs us of the direction of the sound source. Above around 700 Hz, however, phase difference does not contribute usefully to our perception of sound direction. But because of the shorter wavelength of these frequencies, the head has a shading effect, absorbing energy and resulting in more acute IIDs at frequencies of above 700 Hz.

A full explanation of exactly why these phenomena occur would be too lengthy to discuss here, but can be found in Howard and Angus (1996: 96-104). A good rule of thumb is that low frequencies (below 700 Hz) are localised according to interaural time difference (and phase difference) while with higher frequencies (above 700 Hz), this depends on the interaural intensity difference. But IID and ITD alone do not explain our ability to localise sounds:
In fact, IID and ITD only affect the extent of the lateralization [sic.] of the sound source, that is, its perceived position along the interaural axis, a left/right axis between the ears. With only IID and ITD, a listener cannot determine whether an acoustic event is in front, above, behind, or below. (Kendall 1995: 31)

This is perhaps not strictly accurate in terms of front/back localisation, where IID and ITD can, in fact, inform localisation. If a sound is directly in front of, or directly behind the listener, there will be no IID or ITD. Similarly, sounds arriving at angles of $\theta$ (in front of the listener) and $\theta+180^\circ$ (the same angle, but behind the listener) will exhibit the same IID and ITD characteristics. The front/back position of the sound is therefore ambiguous. However, if the listener turns his/her head to the right, a sound source in front of the listener will arrive at the left ear earlier, and with greater intensity, than the right, vice versa for a sound behind the listener. By rotating the head in this way, the listener is shifting his/her interaural axis, and the resultant changes in IID and ITD help to disambiguate the front/back orientation of the sound source.

Nevertheless, IID and ITD certainly do not represent the only means of human sound localisation. For example, we have not mentioned how sounds are localised in terms of their height. This is achieved by means of Head Related Transfer Functions, a more sophisticated and nuanced characteristic of the human hearing system.

1.2 Head Related Transfer Functions

Direction perception is affected by the way in which the listener’s body (specifically the torso, head, outer ears, and ear canals) affect the sound waves reaching each ear (Kendall 1995: 26). We can imagine that each ear has a different ‘filter response’ (Howard and Angus 1996: 341) which varies with the ITD, IID, the acoustics of the listener’s body, and the directional
characteristics of the sound source. As a result, the perceived sounds at each ear may differ in both time and frequency domains. This variable ‘filter response’ is known as a Head Related Transfer Function (HRTF). As Kendall explains, ‘the sound arriving at each ear is spectrally modified by the HRTF, each ear has a different transformation, and the transformation changes as the head and/or the source moves’ (Kendall 1995: 31). So while IID and ITD relate to the amplitude and temporal characteristics of sound, HRTFs are concerned with the spectral characteristics, and how these aid localisation.

Being dependant on the physical characteristics of the listener, HRTFs can vary considerably from person to person. This, in combination with the complexity and potentially dynamic nature of the interactions involved, can cause serious difficulties in the artificial recreation of such effects, for example, in binaural techniques, which will be discussed later.

1.3 Ratio of Direct Sound to Reverberant Sound

As mentioned earlier, directionality is not the only aspect of a sound’s position with respect to the listener. Distance perception is another important consideration and this, to a greater extent that the principles mentioned so far, is informed by the environment in which the sound is propagated. This is an important point which raises both technical and aesthetic issues, which will be discussed later.

When a sound propagates in an enclosed environment (or, more accurately, in an environment ‘with boundaries’, not necessarily enclosed (Howard and Angus 1996: 233)), the first sound a listener in that environment will hear is the direct sound: the sound which has travelled the shortest route between source and listener. As Kendall explains, ‘this initial direct sound provides the least compromised information about the direction of the sound
event’ (Kendall 1995: 25). It is important to note that ‘the direct sound level is dependent on the distance between the listener and the sound source’ (Howard and Angus 1996: 244).

Soon after hearing the direct sound, the listener will experience sound which has been reflected from various surfaces within the environment. The first few reflections perceived are known as early reflections and ‘are separated in both time and direction from the direct sound’ (Howard and Angus 1996: 236).

Eventually the rate of arrival of reflections at the listener becomes so high that individual instances can no longer be perceived as such. This is known as reverberant sound. In contrast with the direct sound, the perceived level of reverberant sound does not vary depending on the distance between source and listener (Howard and Angus 1996: 243-4), therefore the ratio of direct sound to reverberant sound gives a psychoacoustic indication of this distance.

1.4 High-Frequency Attenuation

Another factor which contributes to distance perception is the effect of air absorption. High frequencies in particular are susceptible to these effects, especially in humid or smoky environments. It follows that the greater the distance a sound has travelled though air, the greater the perceived effects of high frequency loss. ‘This is one of the reasons that people sound duller when they are speaking at a distance’ (Howard and Angus 1996: 273-4).

Therefore, a sound demonstrating a low ratio of direct to reverberant sound, with apparent loss in the high frequency area, will be perceived as relatively distant from the listener. Conversely, a sound with a high direct to reverberant ratio, which sounds bright, would be more likely to be perceived as relatively close to the listener.
Having examined the ways in which humans can localise sound in space, we will now go on to examine the various technical means by which illusions of sound sources can be created in a surround sound environment, and which physical and psychoacoustic principles (if any) these systems take into consideration. Ambisonics, being the method chosen for the proposed application, will be discussed in greater detail in the next chapter.

2.1 Stereophonic

Stereo, which is normally understood to mean two-channel stereo, has been the prevailing standard for recorded sound since the introduction of the stereo LP in 1958. For an optimum stereo image, the two loudspeakers should be separated by an angle of sixty degrees or less with respect to the listening position. This is illustrated in the diagram below.
Typically, stereo systems rely only on difference in amplitude (intensity) between the two speakers to give the impression of a sound source anywhere on the ‘active arc’ between them (Pulkki 1998: 130). If a sound source appears to exist between two speakers in this way, it is referred to as a ‘phantom source’ (Thiele and Plenge 1977). This is the basis on which traditional panning works, and is also the principle behind recordings made using coincident pairs of directional microphones. It is possible, although less common, to produce the stereo image by way of time differences between the two loudspeakers. This is the supporting theory behind recordings made using spaced pairs of omni-directional microphones, and is very rarely used in the synthetic generation of stereo images (Malham 1998: 173).

Such stereo setups work well provided the listener is standing directly in the ‘sweet spot’ between the two speakers, and provided that the angle between the speakers is sixty degrees (or less). If the listener is positioned ‘side-on’ (i.e. at an angle of ninety degrees with respect to the optimum listening position), localisation is virtually impossible. With angles of more than sixty degrees between loudspeakers, ‘the virtual source gets spectrally more spread and more colourations occur’ (Pulkki 1998: 130). An over-wide angle between speakers can also result in what is known as the ‘hole in the middle’ effect, whereby phantom sound sources in the middle of the active arc between two speakers suffer from amplitude loss. This has been identified as a characteristic problem of amplitude panning between speakers (Gerzon 1977: 401-2). There are however, measures which can be taken to reduce these effects. Harrison (1998) utilises additional stereo pairs of loudspeakers positioned variously around the performance venue to provide a more homogeneous stereo image for auditioners of the BEAST diffusion system. Harrison specifically cites a centre-stage stereo pair as a remedy for the ‘hole in the middle’ effect in larger venues. A similar approach is taken in 5.1 systems which will be described later.
Pulkki (1998) describes a system for extending the pair-wise stereo paradigm to a triplet-wise equivalent using groups of three speakers, in order to achieve more dimensions of directionality. For further details of the frequency-dependent imaging capabilities of two-channel stereo systems, and how these can be improved by ‘stereo shuffling’ techniques, see (Malham 1998).

### 2.2 Quadraphonic

A number of techniques have tried to expand on the pair-wise amplitude-panning techniques used in traditional stereo, to provide a more accurate reproduction of the entire sound field. Quadraphonic systems were introduced, unsuccessfully, in the early 1970s as a proposed solution for pantophonic (horizontal only) surround sound. Four loudspeakers are positioned in a square layout, each pointing towards a central listener. The angle between adjacent speakers, therefore, is ninety degrees. This is illustrated below.

![QUADRAPHONIC](image)

Gerzon (1977) provides a good analysis of the reasons behind the poor performance of quad systems. Seemingly most of the problems are rooted in the over-wide ninety-degree angle between the speakers. As mentioned earlier, this causes the ‘hole in the middle’ effect
which, in turn, can cause sources to be pulled towards the speaker to which they are closest, especially if the listener turns his or her head to face that particular speaker. Further, in the same way localisation is not possible at ninety degrees off-axis in a standard two-channel stereo setup, so localisation along the sides of a quadraphonic setup is equally poor.

In summary, quadraphonic systems suffered from essentially the same problems as stereophonic systems, but it seems that augmenting the number of speakers from two to four resulted in a parallel augmentation of these localisation problems. It is for these reasons that quadraphonic systems failed, at least in their proposed capacity to provide a more realistic representation of a pantophonic sound field. However, it is argued (Vennonen 1994) that the failure of quadraphonic systems prompted more detailed research into the principles of sound localisation, ultimately resulting in the ambisonic and binaural techniques which will be explained below. Additionally, the matrixing techniques used in quadraphonic systems in order to encode the four channels into two for easy transmission (on two-channel LPs for example) were to become the basis for cinema surround sound systems such as Dolby 5.1 (Dolby 1999), which will be discussed presently.

2.3 Dolby 5.1

Dolby 5.1 is another extension of pair-wise stereo panning techniques. The front-left and – right loudspeakers are reinforced with a centre speaker, providing stable central localisation even for off-centre audience members. Additionally, two surround channels are added – left surround and right surround – which feed speakers positioned to the rear of the audience. These five channels account for the ‘5’ in 5.1. The ‘0.1’ refers to the low frequency effects (LFE) channel which is used for enhancing film soundtracks with frequencies below 40 Hz.
The subwoofer to which the LFE channel is sent is often positioned between the front-left and centre speakers, resting on (perhaps faulty) assumptions that it is impossible to resolve the directionality of very low frequencies. The ITU standard speaker layout for 5.1 systems is shown in the diagram below.

![DOLBY 5.1](image)

Notice the potential for good sound localisation towards the front of the rig, that gets increasingly worse towards the back. The 80° angle between the front speakers and the surround speakers means that localisation along the sides will be difficult. Further, most audience members will be ‘side on’ with respect to this pair of speakers, so will not be able to localise sounds between front and surround speakers at all. Localisation between the surround speakers themselves is made very difficult by the

I use the collective term ‘audience’ (as opposed to the singular ‘listener’) quite deliberately, as it should be acknowledged in all stages of its evaluation that 5.1 was developed specifically for the reproduction of film soundtracks. Precedence is given to front-oriented sounds, while surround channels are often delayed so that they do not arrive at the ears of audience members before the sounds ‘from on screen’. Gerzon (1977: 404) condemns surround sound systems which are ‘too critically dependent on any highly specific assumptions about […] the position or number of […] speakers’. It is for this reason that systems such as Dolby 5.1 may not be quite so suitable for creative purposes as some of the more portable methods. Nonetheless, 5.1 is more or less established as a standard for
surround sound reproduction, and systems are already widespread in both industry and consumer sectors. With increasing interest in art for multiple media, the importance of cinema surround sound systems cannot be ignored, and this is certainly reflected in the audio and electroacoustic community. This will be discussed further in the next chapter, in the section entitled ‘Ambisonics vs. Dolby 5.1’.

### 2.4 Binaural and Transaural

In comparison with the techniques described so far, binaural is substantially different insofar as it is designed solely for reproduction over headphones. Binaural audio makes use of ITD and IID characteristics, along with HRTF information, to create the illusion of a three-dimensional audio environment around a single listener. It is for this reason that binaural audio has found its most frequent application in immersive virtual reality (VR) environments (Begault 1994).

Binaural recordings can be obtained by placing microphones in the ears of a dummy head (or a real head, in fact), or by applying average HRTF filters artificially by computer. However, as mentioned earlier, HRTFs, ITDs, and IIDs can vary considerably from person to person, so there is no guarantee that recordings made using either of these methods will result in the desired spatial effects being perceived by any given listener. In particular, listeners may have trouble in determining whether a source is behind or in front of them (Kendall 1995: 38). Additionally, owing to the complex nature of HRTFs (discussed earlier), their artificial simulation is very processor intensive.

Another problem associated with binaural audio results directly from its being reproduced over headphones. If the listener turns his or her head, the positions of virtual sound sources
will move as well. This can be overcome by head tracking in real time and compensating appropriately for movements of the head. However, this does add complexity to a process which is already computationally expensive. Notwithstanding these problems, binaural audio can be astonishingly convincing, particularly if source sounds were recorded using the listener’s own HRTFs.

Binaural recordings will not be reproduced correctly over loudspeakers because of cross-talk, that is, signal from the left speaker reaching the right ear, and vice versa. Binaural recordings can, however, be reproduced over loudspeakers providing some measures are taken to ensure that cross-talk cancellation occurs. Very basically speaking, this is achieved by issuing a phase-reversed copy of the output of the right-side speaker from the left-side speaker, and vice versa. Obviously, it is not quite as simple as that, as these cancellation signals must also be optimised to the listener’s own particular HRTFs if best results are to be achieved, and this imposes obvious practical limitations on the system. For a more in depth introduction to cross-talk cancellation, HRTFs, and binaural and transaural recording see (Kendall 1995).
3: The Ambisonic System

Vennonen describes ambisonics very concisely as ‘a psychoacoustically optimised way of encoding an infinite number of sound directions into a limited number of channels, and then decoding them to a given loudspeaker layout on playback’ (Vennonen 1994). The means of psychoacoustic optimisation will not be discussed in detail here: for a brief explanation please refer to (Vennonen 1994). An important aspect of ambisonics that distinguishes it from the other techniques discussed so far, is that the processes of recording (encoding) and reproduction (decoding) are clearly separated from one another. In this capacity, it is an example of a ‘kernel’ system (Gerzon 1977: 400). An infinite number of directional components can be encoded into four channels, and then decoded – using either software or hardware decoding means – for reproduction over a particular setup of loudspeakers.

3.1 The Four Stages of Ambisonic Encoding and Decoding

3.1.1 Ambisonic A-Format

It is easiest to begin an explanation of the theories of ambisonics with a description of how it is possible to record a three-dimensional sound field. As left-right spatial information can be captured using a pair of coincident directional microphones, it follows that spatial information for the full three-dimensional sound field can be captured if further coincident microphones are added at different angles to each other. The Soundfield microphone

---

1 In first-order ambisonics the characteristics of the sound-field are encoded into four channels. For clarity (and because it is still the most commonly used format for ambisonics), my summary of the principles involved will refer only to the first order. Higher orders, which basically involve capturing directional information for a higher number of axes at the B-format stage, will not be discussed here.
forestalls the need to do this with separate microphones (which would, in fact, be impossible in practice due to, amongst other things, the shadowing effects of the microphone casings).

The Soundfield microphone contains a near-coincident tetrahedral arrangement of four sub-cardioid microphone capsules, which are angled in the directions necessary to capture spatial information in three dimensions. Recording with this array of capsules results, obviously, in four discreet signals: left-back (L_B); left-front (L_F); right-back (R_B); and right-front (R_F). These four signals derived directly from the capsules of the Soundfield microphone are known collectively as ambisonic A-format.

3.1.2 Ambisonic B-Format

The A-format signals are then processed by a control unit in order to derive the directional velocities for each of the three axes of the sound-field – X, Y, and Z, and the omni-directional pressure component, W. This is achieved with the following equations:

\[
\begin{align*}
W &= L_B + L_F + R_B + R_F \\
X &= L_F - R_B + R_F - L_B \\
Y &= L_F - R_B - R_F + L_B \\
Z &= L_F + R_B - R_F - L_B
\end{align*}
\]

Some additional calculation is also required to account for the fact that the four microphone capsules are not exactly coincident. It is also possible to make a number of other adjustments to the signal at this stage, such as gain compensation, low frequency attenuation, and so on.

The final W, X, Y, and Z signals derived from this sequence of processes are collectively known as ambisonic B-format. This format is the one most commonly used to deal with
ambisonic sound-field recordings. Artificial sound-fields are normally synthesised in this form, and existing sound-field recordings (natural or artificial) can be most easily manipulated at this stage. Derivation of B-format often – but not always – represents the end of the ambisonic encoding process.²

3.1.2.1 Artificial Synthesis of B-Format Signals

In order to calculate the necessary B-format channels to artificially place a sound source within a sound-field, its position must be described by two angles referenced to centre-front. These angles must indicate the horizontal rotation, or azimuth (angle A) and the vertical rotation, or elevation (angle B).

By convention, 0° azimuth, 0° elevation is directly in front of the forward-facing listener, at the centre-front of the sound-field. Positive angle of rotation for azimuth is anticlockwise from this position, while positive rotation for elevation is upwards.

The required B-format channels are then given by the following equations:

\[
W = \text{source signal} \times 0.707 \\
X = \text{source signal} \times \cos A \times \cos B \\
Y = \text{source signal} \times \sin A \times \cos B \\
Z = \text{source signal} \times \sin B
\]

3.1.3 Ambisonic D-Format

²Ambisonic C-format, as it is not always a necessary stage in the signal chain, will be discussed later, so as not to confuse matters.
The final stage of the ambisonic process is decoding, which concerns the reproduction of the encoded sound-field over loudspeakers. This is carried out by an ambisonic decoder – which can be implemented either as hardware or software – designed for decoding to a particular arrangement of loudspeakers. This **D-format** stage consists of unique signals to be fed to each loudspeaker in the rig.

In its most basic form, ambisonic decoding, like encoding, is a matter of simple mathematics. Each individual speaker, depending on its position relative to the centre of the sound-field, is fed a mathematically derived combination of the B-format signals. The decoding equations for simple square (pantophonic) and cube (periphonic) loudspeaker arrays are given as follows (Malham and Myatt 1995: 64):

**Square – left-front (LF), right-front (RF), left-back (LB), and right-back (RB):**

\[
\begin{align*}
LF &= W + 0.707(X + Y) \\
RF &= W + 0.707(X - Y) \\
LB &= W + 0.707(-X + Y) \\
RB &= W + 0.707(-X - Y)
\end{align*}
\]

**Cube – same as square, but in both ‘up’ (U) and ‘down’ (D) planes:**

\[
\begin{align*}
LFU &= W + 0.707(X + Y + Z) \\
RFU &= W + 0.707(X - Y + Z) \\
LBU &= W + 0.707(-X + Y + Z) \\
RBU &= W + 0.707(-X - Y + Z) \\
LFD &= W + 0.707(X + Y - Z) \\
RFD &= W + 0.707(X - Y - Z) \\
LBD &= W + 0.707(-X + Y - Z) \\
RBD &= W + 0.707(-X - Y - Z)
\end{align*}
\]
These are ‘optimum for listening positions close to the loudspeakers or outside the speaker array’ and ‘where the listening area is significantly smaller than the speaker array, a more hypercardioid shape can be employed by increasing the directivity factor, which results in improved imaging for centrally located listeners’ (Malham and Myatt 1995: 64). Decoding equations for a variety of commonly used speaker configurations are given in Furse (2000). It is these decoding equations that are implemented in the Ambidec plugin, which will be fully described in subsequent chapters. Furse provides equations which deliver ‘a strict idealised response that satisfies the Ambisonic matching equations’, resulting in a stable spatial image over a relatively small area in the central area of the sound-field. These have a high directivity factor, as described by Malham and Myatt. ‘Controlled opposites’ decoding equations are also given which will result in an ‘in-phase’ response, providing ‘a larger listening area at the expense of some directional information’ (Furse 2000). These equations rely more heavily on the omni-directional W component, and therefore have a lower directivity factor.

3.1.4 Ambisonic C-Format

It was mentioned earlier that B-format is often, but not always, the end of the ambisonic encoding process. It is possible, with use of phase manipulation, along with matrixing techniques similar to those used in quadraphonic systems, to further encode B-format signals into only two channels. Known as ambisonic C-format, or UHJ, this technique is particularly useful when only two channels are available, for example, for CD production or broadcasting. Use of a UHJ decoder will provide 360-degree surround sound in the horizontal plane (full periphonic encoding is not possible in C-format) from a two-channel encoded recording,
albeit with slight loss of directional fidelity. For a more in-depth discussion of ambisonic UHJ see (Gaskell 1979).

3.2 Sound-field Transformations

As decoding for a particular loudspeaker setup is effectively performing a transformation of the B-format representation of the sound-field, so it is possible to perform different kinds of sound-field manipulation at the B-format stage. These transformations of the entire sound-field are typically carried out by applying specific matrix calculations. Processes include: tilt, tumble, and rotation – rotations around the x, y, and z axes, respectively; and dominance (or zoom) – to give a particular directional precedence to the sound-field. There are others, but their existence is, in the main, theoretical. Simple equations for rotation, tilt, and tumble are given in (Malham and Myatt 1995: 63).

3.3 Some Limitations

Like any other surround sound reproduction system, ambisonics is not without its drawbacks. One notable restriction imposed by the ambisonic system is that it is impossible to have a sound emanate from one speaker only. This situation would supposedly become less of a problem with increasing orders of ambisonics, but as these are mainly theoretical in conception the problem remains a real one. Pulkki’s VBAP system, mentioned earlier, has the following specification:

If the virtual source is panned in the same direction as any of the loudspeakers, the signal emanates only from that particular loudspeaker. If the virtual source is panned to a line connecting two loudspeakers, the sound is applied only to that pair […]. If the virtual source is
located at the centre of the active triangle, the gain factors of the loudspeakers are equal (Pulkki 1998: 131).

Further, while reasonably adaptable between speaker configurations, ambisonics does still require purpose-designed decoding algorithms and/or circuitry to produce the correct speaker feed signals. The VBAP system allows the use to enter Cartesian coordinates relating to the positions of loudspeakers, and calculates the necessary speaker feeds from this information. A similar system for the automatic decoding of ambisonics has been conceived at the University of York, but not yet implemented (Glinos 2000).

3.4 Ambisonics ‘vs’ Dolby 5.1

Since its commercial release in 1992, Dolby Digital (5.1) has more or less become established as the standard for multi-channel audio. Subsequently, availability of laser discs and DVDs to the consumer market has extended this standard to the home, and the recent introduction of the DVD-Audio disc, and Sony’s rival Super Audio Compact Disc (SACD), has prompted an influx of audio-only releases in 5.1 surround sound. Additionally, there are a number of popular computer sound cards (such as the Soundblaster Live! Platinum) which support 5.1 output, and these are complimented by an increasing range of software, from games to audio applications, with 5.1 support. Accordingly, there is much discussion among bodies such as the Audio Engineering Society on the subject of 5.1 surround sound.

Given the widespread dominance of 5.1 systems, one would be justified in questioning the viability of applications which support alternative spatialisation technologies (such as ambisonics), which do not necessarily have as much commercial and consumer backing. The

3 The 19th Conference of the Audio Engineering Society, which took place in June 2001, was on the subject of surround sound and featured several papers and seminars concerning issues of 5.1 recording techniques, mixing, etc. For further details see – http://www.aes.org/events/19.
question, however, is not entirely one of commercial viability, but is more concerned with the suitability of a particular system with respect to the requirements of the end user. In other words, how appropriate (or otherwise) is the 5.1 system for the (re)production of music? This question raises a number of issues, ranging from the aesthetic requirements of composers, to the practicalities of producing audio material within the confines of any specific surround sound system, to the potential need for a unified performance practice for electroacoustic music. These issues will be discussed presently.

Malham notes that ‘although, for commercial reasons, Cinema Style systems are increasingly being pressed into use for musical recording and composition, they are not really designed for the purpose’ (Malham 1998: 174). This is an important factor, emphasising that the Dolby Digital specification was designed specifically in response to issues relating to cinema broadcast. For example, Bamford observes that ‘the [Dolby] surround response has been tailored such that it is extremely difficult to localise sounds from the rear of the listening area’ (Bamford 1995). In other words, the system has been designed to draw the listener’s (viewer’s) attention towards the front of the sound field; sounds to the rear are difficult to localise and therefore are treated as ‘background noise’, essentially. For a cinema sound system this is probably appropriate: the audience’s attention is drawn towards the screen. Equally, there are many musical occasions where it is necessary, or at least conventional, to draw the listener’s attention towards the front. In most concerts, for example, the performers are situated in front of the audience, and this is where one would expect the listener’s attention to be focused. This conventional ‘spectacle’ approach is iterated by Dolby themselves when describing how best to utilise the surround channels of a 5.1 system: ‘maybe the principle that has served so well for the film industry can also work for music mixing: don’t use the surrounds to distract the listener’s attention away from the story’ (Dolby 1999a).
The problem with the above statement is that it assumes that ‘the story’ is always located in front of the audience which, although true enough in most cinematic experiences, and those instances of musical performance mentioned earlier, is certainly not the case in all musical situations. Cinema productions and most live musical performances all involve some kind of visual element (whether it is the action on screen, or the spectacle of the performers themselves), and this is usually oriented at the front of the auditorium (indeed, one could say it defines where the front is). In the case of electroacoustic compositions, there is often no performer as such, and thus no visual indication of where the front of the performance space is. Therefore, there is no actual need for any plane of the sound field to be treated preferentially.

Malham (1999) describes a surround sound system which does not treat any direction preferentially (i.e. one which considers all directions equally) as ‘homogeneous’. Malham’s other main concern is the quality of localisation offered by a surround sound system, that is, how stable the sound field is over how large an area, and to what extent audience members perceive the directionality of sounds correctly regardless of their position in the auditorium, rotation of the head, and so on. A surround sound system that does present (as far as reasonably possible) the same uniform sound field under all these circumstances is described as ‘coherent’. Certainly this is an important issue, and one which is related to the problem of accurately portraying the intentions of the composer despite the variable nature of performance venues, audio equipment, and audience. We will return to this subject later. In any case, in terms of the system itself, 5.1 is neither a homogeneous, nor a coherent system for surround sound. This means that composers working within a 5.1 environment are restricted in the way in which they can use the sound field.

That composers are restricted in their creative use of, for example, the rear of the sound field when working with 5.1 is directly symptomatic of the design of the system itself.
Another restriction which results directly from the design is the way in which sounds are positioned within a 5.1 sound field. Composers are forced to think in terms of five (six if we are including the LFE channel) channels during the mixing process, where it would be much easier to consider sounds in terms of their position in the sound field. At the moment there is no agreed standard on how this should be done, and mixing in 5.1 remains a relatively complex process. Various hardware and software solutions to this problem exist and others are being developed. Steinberg’s *Nuendo* features a surround sound panner which can be configured to various speaker arrangements, and amplitude panning is carried out using a simple GUI. In terms of hardware, there exist numerous commercially available mixing desks which are designed for panning in 5.1 surround sound. However, feeding directly to five speakers in order to achieve spatialisation by mechanical means is a complicated and often problematic process. Holman concedes that ‘the mechanics are a little more difficult [compared to two channel stereo panning] because of the number of channels’ (Holman 1999: 117). Generally 5.1 mixing desks facilitate multi-channel panning in one of two ways: three-knob panners, consisting of panning controls for L/C/R, F/S (front/surround), and LS/RS controls; and joystick controls. Holman gives the following example of panning a sound using the three-knob interface:

Let’s say I want to pan a sound from left-front to right-surround. I would preset the L/C/R knob to left, and the LS/RS knob to RS, and perform the pan on the F/S knob when it is needed in the mix.

(Holman 2000: 119)

Clearly this process is not very intuitive, requiring a considerable degree of forethought and, inevitably, less straightforward sound trajectories would require increasingly complex manipulations of the panning controls. This impedes the creative process of positioning sounds in space. Another limitation of mixing in this way is that the end result is a 5.1-
specific mix, which cannot be played with the correct results over any other arrangement of
speakers without a complete remix.

In addition to these mechanical impositions, spatialisation under the 5.1 system is subject
to the various problems associated with the localisation capabilities of amplitude panning.
While the joystick interface obviously offers a more intuitive method of positioning sound
sources than does the three-knob method, problems stemming from the amplitude-panning
remain unaccounted-for. For instance, the position of the joystick does not necessarily
correspond with the perceived position of the sound source. When the joystick is centred, the
implication is a point source at the centre of the sound field, while in actuality ‘a huge variety
of directions will be heard depending exactly on where one is sitting, and a listener seated
precisely at the center [sic.] hears a mess, with each loudspeaker’s sound affected by the
associated Head Related Transfer Functions’ (Holman 2000: 120).

The ambisonic system offers solutions to these problems, from both creative and
perceptual stand-points. It allows composers to place sound sources spatially by providing
angles of azimuth and elevation. The means by which these parameters are given (that is, the
interface) is another issue which will be discussed later. The point is that the positions and
trajectories of sounds can be specified without any reference to the playback environment,
thus bypassing the mechanical implications of thinking in terms of five discreet output
channels. This allows the possibility of an interface which deals only with the positioning
and movement of sounds. Without any other issues to contend with, the interface can be
designed and optimised specifically for that purpose. In terms of the homogeneity of the
sound field, it has been proved (Bamford 1995; MacCabe and Furlong 1994; Malham 1999)
that ambisonic systems deliver more stable localisation than methods based only on amplitude
panning. In fact, Bamford states that it would be possible to incorporate ambisonic methods
into a 5.1 system and achieve better localisation characteristics than were offered by the original 5.1 system (Bamford 1995).

Returning to the subject of the homogeneous electroacoustic performance, there is another sense in which use of the 5.1 system can be seen as a solution. Most obviously, the extensive implementation of the 5.1 system in cinemas and homes (whether ideally suited to the particular application or otherwise), gives composers the opportunity to create 5.1 mixes with the assurance that there will be widespread access to appropriate – and standardised – playback facilities in both public and private listening environments. But further, the calibration of the acoustics of many cinema halls under the LucasFilm THX specification (Robinson 2000) ensures that these cinemas do not differ significantly from one another in terms of their acoustic response. Increasing interest in audio-visual and multiple media composition further suggests the cinema hall as an appropriate performance venue, and the high degree of standardisation among cinemas (at least in comparison with other performance venues) offers as plausible and practical a solution to the problem of performance homogeneity as could be imagined. The use of cinemas as venues for the performance of visual and audio-visual compositions can thus be seen as an attempt to achieve a standardised performance practice for electroacoustic music via the medium of standardising the venue itself. Such an approach has also been taken by the Australian Centre for the Arts and Technology (ACAT), whose Geodesic Dome was built as a portable performance venue, with surround sound and video projection facilities, for which pieces could be specifically written and performed in different locations. A different approach is taken by Harrison (1998), who suggests that measures should be taken – both in terms of chosen equipment, and the way it is utilised – to ensure that the acoustical qualities of a performance space do not obscure the effects intended by the composer. He describes the Birmingham Electroacoustic Sound Theatre (BEAST) as a variety of possible speaker configurations which can be employed as
acoustical conditions dictate necessary to this end. While these systems obviously offer more flexibility in terms of their surround sound capabilities, the use of cinemas as performance venues is clearly a more convenient solution. For further discussion of this see (Mooney 2000).

It has been stated that the ambisonic system allows the positions of sound sources to be specified in an intuitive way (given the right interface, of course), without reference to any specific speaker layout, and that this is more compositionally intuitive than traditional 5.1 mixing methods. Equally, it has been established that the use of cinemas (which in the vast majority of cases employ 5.1 surround sound systems) as standard performance venues addresses a number of other important issues in the field of electroacoustic composition. As it is fully possible (although not necessarily ideal) to decode B-format signals to a 5.1-style speaker rig, it seems that ambisonics is not, in fact, completely divorced from more widespread 5.1 technology, and could be incorporated into this system, as Bamford (1995) acknowledges. The advantages of this would be numerous. Composers would be able to realise fully periphonic pieces in the most intuitive way, free from the technical difficulties that can arise from mixing for a discreet number of fixed output channels. Nonetheless, by decoding for a 5.1 speaker layout, composers are able to take advantage of the many 5.1 systems that exist in cinemas and homes knowing that these performance environments are (cinemas to a greater extent, homes to a lesser extent) essentially acoustically similar. And, of course, should performance over a different speaker rig be necessary, this would not require any additional mixing, only another set of decoding parameters. Obviously, the compromise is the loss of height information.
4: Existing Software for Sound Spatialisation

The research carried out so far indicates two main considerations in the implementation of software for carrying out manipulations in 3D: graphical metaphors (in the form of GUls); and specialised input devices. Obviously these factors almost always interact by necessity. For instance, it would be unusual for an input device to make manipulations directly without some kind of intermediate graphical metaphor. Having said this, the proportions in which GUls and input devices operate within different environments can be tremendously variable, from straight-forward offline parameter entry to immersive virtual reality with fully three-dimensional input, audio, and visual devices. The following sections will discuss the various approaches to 3D control for sound spatialisation, with reference to relevant sources and existing applications.

4.1 Offline Applications

One of the biggest problems in dealing with fully periphonic sound spatialisation in software is that of how to represent three-dimensional constructs (four if we are to include time) on (normally) two-dimensional screens, and how to manipulate them using (normally) two-dimensional input devices. Some applications get around these problems by representing positions in and trajectories though space as parameters. The Csound implementation proposed by Malham and Myatt (1995) falls into this category, allowing monophonic source sounds to be positioned statically in an ambisonic sound-field by the declaration (in the Csound score file) of angles, subtended to centre-front, for the horizontal and vertical planes.
Movements around the circumference of the sound-field can be achieved by giving start and end angles.

This software only represents the simplest aspects of ambisonic sound-field synthesis and is only really intended as an introduction to the area. Only two positions, with linear interpolation between them, can be allocated to a single sound file, and being based on Csound, compilation and decoding is obviously required before processes can be auditioned (although Malham and Myatt also offer a Csound orchestra file for horizontal-only decoding). The software does not allow sources to be moved through the sound-field, nor does it allow sources to be placed ‘distantly’. The latter can only be achieved by building distance characteristics into the source sounds themselves, either at the recording stage or by external post-processing. This does, however, mean that the composer has full control over the distance processing applied to sounds, which would not necessarily be the case if the function was built into the software. Pulkki’s VBAP offers much the same creative scope, but is based on amplitude panning rather than ambisonic principles.

The MGP Toolkit, developed at the University of York, works in conjunction with an audio sequencing application called Mix for the Irix platform. A Cubase-style mix is created in stereo, then ambisonic positions and trajectories can be allocated to the individual sound files within that mix. Points in space are defined parametrically by the user as Cartesian coordinates (x, y, z) or, alternatively, can be selected from a set of pre-defined positions as a short-cut, if high degrees of precision are not necessary. There are functions with distance laws incorporated, which accordingly apply reverb for sounds ‘behind’ the speakers, and functions which carry out ambisonic panning without distance laws. Circular motion, with optional linear variation of distance, is also possible.

This system allows the composer to create a stereo mix (i.e. arrange sonic material temporally) and then work on ambisonic panning from within the framework of the entire
piece. In this respect it offers a more holistic solution, perhaps easier than panning individual sounds ‘on the fly’ in the Csound environment, as in Pulkki (1998) and Malham and Myatt (1995). However, sounds must still be panned individually with no audio or visual reference point for the spatialisation, and compilation and decoding are still necessary before audition is possible. Again, trajectories may only have a start and end point with linear interpolation being applied in between them.

Malham’s (2001) BF (presumably short for B-format) package consists of four parts: BFPan, for panning mono sources in an ambisonic sound-field; BFProc, for applying transformations (rotations, dominance, etc.) to existing B-format recordings; BFMix, for mixing B-format sound-fields; and BFDec, for decoding. These are command-line applications, and all parameters must be supplied in the form of TCL (Tool Command Language) scripts. This software operates on an offline basis and, being driven by a scripting language, complex processes, or manipulations involving long sound files, can be very slow. It also requires the user to have some knowledge of TCL, knowledge that a composer does not necessarily have. Further, each component of BF must be invoked separately, and multiple processes often require multiple renderings: to assemble an entire ambisonic composition using these tools would be a rather convoluted process. This said, the BF package is work in progress and does offer considerably more control over the sound-field than any of the software reviewed so far. It is probably the only software package with support for second order ambisonics. Bringing the elements of BF together into a single user-environment, with an intuitive GUI would be worthwhile.

SndSpace (Starks and Linton 1994) is an example of a Csound-driven sound spatialisation application which addresses the problem of manual parameter entry:
Clearly entering this data manually is tedious, especially if parameters are to be varied (e.g. if the sound moves across the sound stage). To solve this problem, a graphical interface has been implemented to allow the user to enter all relevant data in a more intuitive manner. (Starks and Linton 1994: 468)

*SndSpace* combines acoustic physical modelling (whereby the exact acoustical properties of a room are simulated by computer processing) and binaural techniques to create three-dimensional audio illusion over headphones. Such an approach stresses the almost unavoidable crossover between sound spatialisation and (virtual) room acoustics. The user can specify: the dimensions of the room; the position of the listener (user) in the room, and the direction his or her head is pointing in; the position of the source sound; and the absorption co-efficients of the walls. All of these parameters can be dynamically altered via the GUI.

Three different views (windows) are used to represent the trajectory of the sound source through space: standard elevation; end elevation; and plan view. The user may draw a line in one of these windows using the mouse pointer while the sound is playing. In this way, position and time parameters are dealt with simultaneously, however, the sound is **not** spatialised in real time. Further, the fragmented graphical representation of the three dimensional axes means that x, y, and z coordinates cannot be entered simultaneously, and this does not result in an easy visualisation (or auralisation) of the trajectory. This problem is compounded by the fact that, running under Csound, spatialisations must be rendered before they can be played, therefore experimentation is time consuming. As the computer implementation of HRTFs is so computationally expensive, it is likely that rendering times will be high.

Starks and Linton do cite real-time operation as a future objective, but state that ‘this will require extensive software development and more powerful hardware’ (Starks and Linton 1994: 470). They conclude that:
The development of graphical desktop VR systems appears to be sufficient to implement realistic 3-D graphics representations. It follows that acoustic manipulation can be approached in a similar manner, although ultimately an immersive application may be required to realise its full potential. (Starks and Linton 1994: 469).

4.2 Real-Time Applications

The applications examined so far (with the exception of SndSpace) are all dependant on command-line, or otherwise text based interfacing. This parametric approach can be an advantage, giving composers extremely accurate control over the spatial manipulations they make. On the other hand, there is no obvious way that these applications could work in real time (without almost comprehensive redesign), and the often lengthy rendering time required by these programs could be seen as further impeding the creative process. Freed cites real-time processing as one of the key areas for development in computer music software (Freed 1995: 4). There are, however, a number of applications already in existence that deal with sound spatialisation in real time.

Ballan, Mozzoni, and Rocchesso (1994) present a system fairly similar to SndSpace that operates in real time. A GUI allows the user to move the sound source as well as change the dimensions of the virtual and listening rooms. The GUI is not described in great detail but it can be assumed that it is fairly simple, as horizontal-only output is created which does not present the same interfacing problems as full periphonic control. The spatialisation can also be controlled by MIDI, although details of exactly how this is achieved are not disclosed. This program has obvious limitations, namely that it can handle only one monophonic audio input at a time. Further, being based on horizontal-only spatialisation, it does not directly address the issues of controlling fully three-dimensional metaphors. Quad Pan (Pérez et al
1996) is a similar system which does allow multiple sound sources to be panned in real time, as well as adding a performance element to the spatialisation. The same will be noted of Menzies-Gow’s LAmb. Quad Pan, again, only supports pantaphonic output. Nonetheless, the use of MIDI represents an alternative approach to the control of sounds in space, and one which has been taken in several more sophisticated projects.

The ACAT dome (Vennonen 1994) uses mouse movements to control real-time sound source positioning on the horizontal plane, while a variable height foot-pedal provides the vertical information via MIDI control data. This seems fairly intuitive and represents good use of different control methods for different tasks. However, it does only allow the movement of one sound source at a time. If multiple sources are required, these must be spatialised separately and then mixed later in the studio. As mentioned earlier, this ‘one at a time’ approach to the spatialisation of sounds is not, perhaps, the easiest method from a composer’s point of view.

Menzies-Gow’s (1999) LAmb (short for Live Ambisonics) uses the MIDI protocol extensively as a method of controlling the positioning and movement of multiple sound sources in real time in an ambisonic sound-field. Mono sources can be moved around using a five-octave MIDI keyboard in any of three different ‘modes’: position mode; cartesian cursor mode; and cylindrical cursor mode. In position mode, positions within the sound-field are allocated to particular keys. When a key is hit, the sound source moves towards the position indicated by that key, at a speed given by the velocity of the key-press. The ability to express two quantities (note number and velocity) in one gesture is seen here to be an advantage. In cartesian cursor mode, six ‘cursor keys’ on the MIDI keyboard are used to nudge the sound source in a particular direction. The size of the movement varies with the MIDI velocity. Cylindrical cursor mode operates in exactly the same way as cartesian cursor mode, only with cylindrical coordinates. Multiple sources can be controlled simultaneously in these ways by
assigning each source to an octave of the keyboard. Clearly this limits the maximum number of sources to five, but it is questionable whether simultaneous control of more than five objects by one person in real time is even possible. Additionally, single sound sources can be controlled by joystick when using a suitably equipped MIDI keyboard. Sound-field rotations can be carried out using rotation cursor keys on the in a similar way to the cartesian cursor keys.

*Lamb* can also be controlled completely from the desktop using the more traditional (computer) keyboard and mouse interfaces. Here, only one sound source can be moved at a time, but on the other hand, no MIDI keyboard is required, making the system available to those without MIDI keyboards and those whose (lack of) keyboard skill does not permit them operate the system in this way. Additionally, there are feature which can only be controlled from the desktop, such as sound-field delay and dominance.

*Lamb* demonstrates the use of several different control paradigms for different tasks. The use of MIDI keyboard for the simultaneous ‘playing’ of up to five different sound sources attempts to achieve sound spatialisation in a musically intuitive way, offering scope for a truly virtuosic technique to be developed with practice. This highlights a potential compromise between ease and immediacy of use, and flexibility or potential for skilled implementation. This potential trade-off is recognised by Kirk and Hunt:

Most people would expect an ideal computer system to be easy to use and easy to learn. However, nobody expects learning to drive a car to be easy. No musician would ever say that a violin or a clarinet was easy to use. […] By making the assumption that interfaces should be easy we are in danger of undervaluing the adaptation capabilities of human operators and thus limiting the potential human-computer interaction to a lowest common denominator of ‘easy to use’ commands.
(Kirk and Hunt 1999: 311-2)

While these applications demonstrate the flexibility of the MIDI protocol, there are some potential disadvantages. MIDI control data offers 128 discreet values for continuous change –
a resolution roughly equivalent to 7-bit computer processing – and operates at a data rate of approximately 31 kHz. While the data rate may not be such a problem in controlling sound spatialisation, the low resolution of control data can cause audible ‘jumps’ in sound trajectories which are supposed to be smooth. Given that home computers with 32-bit processing at clock speeds of up to 1.7 GHz (no doubt this figure will be even higher by the time this document is completed) are more or less common, it seems ridiculous that software should be held back by the shortcomings of MIDI.
5: Graphical User Interfaces

This chapter will very briefly discuss some of the techniques used to carry out three-dimensional manipulations with graphical user interfaces. Some application programming interfaces (APIs) for the creation of GUIs will also be described.

5.1 Issues of 3D Control

If only standard (computer) keyboard and mouse input devices are to be used, the greatest challenge in the development of the proposed application will be the control of three-dimensions simultaneously using input devices with fewer dimensions. It has already been mentioned that the input device and GUI are fundamentally linked. With only two dimensions physically represented in the control device, it becomes the task of an innovative GUI to make ‘six-degrees-of-freedom’ (6-DoF) manipulations possible.

Chen et al (1986) carried out some research into the various possible solutions to this problem. Four ‘traditional’ display and interfacing techniques for the control of three-dimensional rotations are described thus:

1. Sliders: Typically the user adjusts the x, y, and z sliders graphically displayed on the screen to indicate the amount of rotation in each axis independently.

2. Menu selection: The user first selects the axis from a text menu and then holds down the mouse button while moving the mouse in one dimension to indicate the amount of rotation.

3. Button press: The user holds down one of three buttons on the mouse or keyboard, and moves the mouse in one dimension to indicate the amount of rotation.

4. Two-axes valuator: The user moves the mouse in two dimensions to control rotation of two of the three axes.

(Chen et al 1988: 121)
Clearly options three and four lend themselves more readily to effective real-time operation.

Three ‘new’ methods of performing 3D rotations are then described: overlapping sliders; continuous XY plus Z; and virtual sphere techniques. The overlapping sliders controller works on exactly the same principle as the traditional sliders approach, whereby each slider controls rotation about a particular axis. Here, however, the x, y, and z sliders are represented by horizontal, vertical, and circular sliders, respectively, thus improving stimulus-response and making the interface more intuitive.

In the continuous XY plus Z method, rotation around the X and Y axes is simultaneously controlled, as in the two-axes valuator method. Rotation about the Z axis is invoked by holding down the mouse button, thus constraining the X and Y axes. This constraint approach seems particularly useful in controlling 3D manipulations, especially where more accurate control is required. For example, if a rotation about the Y axis is required, the other axes could be constrained. The possibilities of constraining are further described in (Pachet and Delarue 1999).

The virtual sphere technique is described thus:

The virtual sphere controller simulates the mechanics of a physical 3-D trackball that can freely rotate about any arbitrary axis in 3-space. On the display screen, the user can imagine viewing an object encased in a glass sphere. Rotation is then a matter of rolling the sphere and therefore the object with the mouse cursor. Up-and-down and left-and-right movement at the centre of the circle is equivalent to ‘rolling’ the imaginary sphere at its apex and produces rotation about the x-axis and y-axis respectively. Movement along (or completely outside) the edge of the circle is equivalent to rolling the sphere at the edge and produces rotation about z. (Chen et al 1988: 123)

Experiments revealed that users could perform rotations faster, and more intuitively, with the virtual sphere. For a description of similar methods, refer to (Hultquits 1990) and (Shoemake
1992). Simple rotations (i.e. about one axis only) were carried out more accurately using the more traditional sliders method, due to the other axes being constrained.

5.2 APIs for Creating Graphical User Interfaces

There exist many application programming interfaces (APIs) designed specifically for the creation of GUIs for software applications. These APIs typically provide the programmer with a library of predefined interfacing objects (called ‘widgets’) such as buttons, sliders, scroll-bars, text-entry boxes, and so on. A number of these APIs were examined in order to assess their suitability for developing an interface for controlling the movement of a sound source in three-dimensional space.

5.2.1 VSTGUI

Virtual Studio Technology Graphical User Interface (VSTGUI) (Steinberg 2001) is an API developed by Steinberg specifically for the implementation of interfaces for VST plugins. It defines a number of widgets such as on-off buttons, horizontal and vertical sliders, knobs, option menus, and text-entry boxes which can be easily integrated with the main code of a VST plugin. In this respect it has an advantage over the other APIs considered, where communication with the plugin itself may not be so straightforward. Additionally, support is readily available from Steinberg’s VST mailing list. However, VSTGUI does not offer any widgets that are particularly suitable for multi-dimensional control, nor does it support any method for rendering graphics for visualisation purposes. It would not, for example, be possible using only the VSTGUI library to render a three-dimensional graphical visualisation of a sound source in a sound-field. Rather, VSTGUI seems to be tailored for creating
graphical widgets for the control of parameters in the most general sense. For the majority of
plugins, many of which emulate physical DSP hardware, such conventional means of control
are perfectly adequate. But, as has already been concluded, more advanced interfacing
techniques are required for the real-time control of objects in three-dimensions.

5.2.2 XForms

XForms (Zhao and Overmars 1995) is a free, open source graphical user interface API written
originally for the X-Window operating system, but since ported to Windows and Linux. Like
VSTGUI it offers a variety of widgets for the graphical control of parameters, but as a more
general API it also features interface elements which are less specific in their applications. Of
particular interest in the area of three-dimensional control are positioner and dial widgets.
The former allows positions and movements to be specified in a two-dimensional plane, while
the latter consists of a dial which can be rotated fully through 360°, a functionality not
included in the dial widgets defined by the VSTGUI library. XForms also supports canvases,
whereby an area of the window can be specified for graphics rendering under a different
graphics API. OpenGL canvases have been defined whereby ‘the object class takes care of all
the initialisation thus freeing the application programmer from worrying about the [operating
system] specifics so he [sic.] can concentrate on OpenGL rendering’ (Zhao and Overmars
1995). OpenGL is a flexible graphics API capable of rendering visual objects in three
dimensions – it will be discussed more thoroughly in Chapter 8.

For these reasons, XForms was seriously considered as the API under which the Ambipan
interface would be coded. Unfortunately, it turns out that while ports of XForms have been
made for Win32 and Linux platforms, OpenGL support remains X-Windows (SGI) only,
although hopefully this will change in the near future.
5.2.3 FOX GUI

FOX (FOX GUI 2001) is a C++ based API for the development of graphical user interfaces. In terms of its features, it is similar to XForms – OpenGL canvases are also possible using FOX – but is fully implemented for Windows, Linux, and SGI platforms.

5.2.4 GLUI

The Graphics Library User Interface (GLUI) is a GUI API designed specifically for use with OpenGL running under GLUT (both of these APIs and their functionalities will be described in Chapter 8). Most common GUI widgets, such as check boxes, radio buttons, spinners, scroll-bars, and so on are included in this library. Additionnaly, it features widgets which are custom-designed for viewing graphical objects in three dimensions, such as rotation and translation (movement along a straight line) controls. These features may be useful for the control of sound objects in three dimensional space.
6: Descriptions

This chapter consists of conceptual descriptions of the plugins and GUI. Guidelines on how to correctly route signals in Nuendo are also given.

6.1 Ambipan Specification

Ambipan, takes a mono or stereo input signal and outputs four channels of first-order ambisonic B-format. Each Ambipan plugin is capable of panning the audio from one track of a VST mixing desk. In this capacity it works in a similar way to a channel insert effect. By installing an Ambipan plugin on each active track on the mixing desk, an ambisonic mix can be created.

Internally, Ambipan positions the sound source based on two parameters: azimuth and elevation. When the GUI is fully integrated, these parameters will be expressed simultaneously by means which will be described later. Using the default interface provided in Steinberg’s Nuendo\(^4\) sequencer, a horizontal slider is created for each parameter.

6.1.1 Ambisonic Panning of Stereo Input Signals

Carrying out ambisonic panning with a monophonic input signal is a simple process: the source signal is simply substituted into the B-format encoding equations along with the appropriate angles of azimuth and elevation. When the input signal is a two-channel stereo file, however, the question arises of how to handle each of the input channels. Perhaps the

\(^4\) At present, Steinberg’s Nuendo is the only VST host application with sufficient multi-channel capabilities to support ambisonic encoding and decoding in real time.
most obvious solution is to mix the left and right channels of the input signal together and divide by two, thus creating a mono sound file. This can then be panned as described earlier. While this may be a perfectly adequate solution for some purposes, it does essentially defeat the purpose of having a stereo input file in the first place.

The question is therefore how to preserve the stereo separation of a stereo input file when producing the B-format channels, and how to place the left and right channels within the sound-field without compromising the stereo image. Ambipan provides two controls designed for this purpose: Stereo Width, and Stereo Height. These parameters control the separation, in degrees, between the left and right channels in terms of angles of azimuth and elevation, respectively. This is illustrated in the diagram below.

Here, L and R represent the left and right channels of the stereo input signal, respectively. Stereo width is represented by θ, and stereo height by Φ. The absolute position of the sound source – that is, the position of the stereo source, considered as a single point in space – is taken as the point exactly in the middle of the active arc which runs between the left and right channels.
Both width and height are limited to a maximum of 180°. This is because values greater than 180° would result in the sound source appearing on the opposite side of the sound-field in with respect to its absolute position. In other words, a source positioned at 0° azimuth, and with a stereo width of 270°, would be the same as a source positioned at 180° azimuth with a stereo width of 90° (except in the latter the left and right channels would be reversed). This would be confusing, therefore width and height have been limited to a maximum of 180° separation.

6.1.2 User Definable Presets

Ambipan allows the user to specify values of azimuth, elevation, width, and height, and store these values as a preset. At present up to sixteen presets can be stored, but this number has been arbitrarily chosen and could be very easily increased. When the user switches from one preset to another, the parameter values of the second preset will instantaneously be restored.

6.2 Ambidec Specification

As mentioned earlier, ambisonic decoding is carried out by a separate plugin, Ambidec, which should be installed in the master effects rack of the VST mixing desk. Ambidec takes four B-format signals as its input and outputs a feed signal for each speaker in the chosen speaker setup. Supported decoding options are: mono; stereo; square; 3/2; and cube. All apart from the cube array support only horizontal surround sound. Each speaker feed signal is output to a discreet channel on master section of the VST mixing desk. The number of the master channel to which each speaker feed signal is sent for each setup is illustrated in the diagrams.
below. Horizontal-only rigs are shown in plan elevation. In each diagram, a dotted line runs from the centre of the rig to 0° azimuth (i.e. pointing towards the front of the rig).

**MONO**

1

**STEREO**

1

2

**SQUARE**

1

2

3

4

**3/2**

1

5

2

3

4
Ambidec has no custom GUI, and therefore uses the default interface provided by the host application. In Nuendo the default interface consists of a horizontal slider, which can be moved from left to right to select the desired speaker rig to decode to.

**6.2.1 Correct Routing of the Ambidec Speaker Feed Outputs in Steinberg’s Nuendo**

The numbering of the Ambidec speaker feed outputs as given above relates to which input channel of the master section of the VST mixing desk that particular speaker feed signal will be sent to. Because host applications may have different ways of handling this, it is ultimately the responsibility of the user to ensure that each speaker signal gets routed to a sound-card output which is, in turn, connected to the correctly positioned speaker. However, because at present Steinberg’s Nuendo is the only VST host with more-than-two-channel
output, an explanation of how to route speaker feed signals correctly, using specifically this software, follows.

As stated, each speaker feed signal output from Ambidec is assigned to an input channel on the master section of the VST mixing desk. The number of the master section input channel to which each speaker feed signal will be sent is given in the diagrams above. It is up to the user to ensure that there are enough input channels in the master section to receive the speaker feed signals for each speaker in the chosen decoding setup. This is done by setting ‘Number of Channels’ in the VST Master Setup dialog accordingly. It does not matter if the user specifies too many output channels here – extra channels will simply not be used – but if there are too few output channels, speaker feed signals being sent to channels beyond those indicated in the VST Master Setup will be discarded. Note also that, although master section channels can be given names, azimuths, radii (relating to supposed speaker positions), and so on, these have no effect on the audio output. In other words, the audio output is unaffected by what Nuendo ‘thinks’ the speaker setup is.\(^5\)

What is important, however, is that Nuendo automatically routes each pair of master section channels (i.e. 1 and 2, 3 and 4, etc.) to a stereo output bus. The user must activate as many output buses as are required to receive the number of speaker feed signals produced by the chosen decoding configuration. For example, two output buses are required for decoding to a square rig, four for decoding to a cubic rig, and so on. Output buses can be activated from within the VST Outputs dialog, which will display a stereo output bus for every pair of physical analogue outputs on the sound-card. Nuendo will automatically route the first two channels of the master section to the left and right channels of Bus 1 (this is always active, and may not be deactivated). The next two master section channels will be routed to the left

\(^5\) This said, users will probably find it useful to at least label each of the master section channels with a name that reflects the position of the speaker to which that channel will eventually be sent.
and right channels next active output bus, and so on. Exactly which master channels are being routed to which output buses can be verified (but not altered) in the VST Master Setup dialog.

Each output bus must then be routed, again via the VST Outputs dialog, to a pair of physical analogue sound-card outputs. The signal from each sound card output must then be amplified and sent to the corresponding speaker in the chosen speaker rig. The signal path for a square rig decode is illustrated in the diagram overleaf.
Although technically it does not matter which output buses and sound-card outputs the speaker feed signals are routed to, it makes sense to ensure that the numbering of the master section input channels reflects the numbering of the sound-card outputs if possible (this is illustrated in the diagram above). In this way, signals taken from the sound-card outputs can
be connected to their corresponding speakers simply by following the numbering of the sound-card outputs.

### 6.3 Ambipan GUI Specification

It has been stated that interfacing methods such as text entry and three-slider axis control are inadequate and non-intuitive for controlling the movements of objects in three-dimensional space. Equally, it has been noted that accurate and intuitive control of such movements using standard two-dimensional input devices, and two-dimensional means of displaying the information, is difficult (perhaps impossible) to achieve. Further, it was noted in the previous chapter that none of the available GUI APIs are well suited to interfacing in real time in three-dimensions. Therefore, control over the position of a sound source in three simultaneous dimensions (four if we were to include time) in Ambipan is achieved through a custom-designed GUI. The sound-field is represented metaphorically as a sphere. The interface consists of three different elevations (views, that is) of this sphere: back elevation; side elevation; and plan elevation. These represent the sound-field viewed directly from behind, directly towards the right-most semi-sphere, and directly from above, respectively. There is also a quasi-three-dimensional representation of the sound-field which comprises a sphere surrounded by a wire-frame cube. This view is rotated slightly about the X and Z axes so that movement in all directions can be visualised. Screenshots of the Ambipan GUI in its current state of development can be found in tests 16 to 22 in Chapter 10.

The user can place the sound source – represented by the smaller sphere – anywhere on the surface of the sphere by clicking at the desired position on one of the three different elevations of the sound-field. Consider, however, that for any given point on these elevations of the sound-field, there are two possible positions on the surface of the complete sphere that
could be implied. For example, if the user clicks in the centre of the back elevation of the sound-field, this could indicate a sound source at either 0° azimuth, 0° elevation, or it could represent a sound source at 180° azimuth, 0° elevation. This problem is overcome by making use of the two mouse buttons. In the case of the back elevation, the left mouse button indicates that the sound source is in the front half of the sound-field, while the right button indicates that it is in the rear half. Similarly, a left click in the plan elevation will place a source in the bottom half of the sound-field, while a right click will place the source in the top half. With the side view, perhaps obviously, the left button is used for moving sounds around the left-most semi-sphere of the sound-field, while the right button is used for movements in the right semi-sphere. Any movement of the sound source in one view, will result in the repositioning of the source in all the other views. Note that the quasi-three-dimensional view is purely for visualisation purposes, and does not include any element of user control.

Stereo-width and –height parameters can be adjusted with the two sliders provided. At present the GUI does not produce a separate visual object for each of the left and right channels of a stereo input file. Instead, stereo sound sources are represented as one object, which is located at the absolute position of the sound source.
7: Virtual Studio Technology (VST)

This chapter will explain the principles behind Virtual Studio Technology (VST), which is the chosen paradigm for the implementation of the Ambipan and Ambidec plugins. It will also outline some of the key VST functions and how they should be implemented within a VST plugin. Some extracts from the Ambipan source code will be used, but the purpose will be to demonstrate general plugin development techniques rather than to highlight the issues specific to the development of Ambipan, which will be discussed in Chapter 9.

7.1 Description and Advantages of the VST Paradigm

Virtual Studio Technology (VST) is a standard for digital signal processing software designed by Steinberg to promote compatibility and extendibility between VST-implemented software. There are two types of VST software: the VST host; and the VST plugin. The host is the ‘main’ audio application, the environment that provides the framework and key functionalities for audio processing, as well as dealing with tasks such as file-handling. VST hosts are standalone applications, and include sequencers such as Steinberg’s own Cubase VST and Nuendo, and Emagic’s Logic Audio; wave editors such as Wavelab and Sound Forge; and other audio applications including Audiomulch and Fruityloops. VST plugins are not standalone applications in themselves, but can be used from within any VST host application to extend the functionality of the software. For example, an audio sequencer may not itself include any facility to apply distortion to an audio track; a very likely scenario in fact. In the days before VST, this would mean that users would have to apply distortion to their sound files using dedicated external DSP hardware (or software), before importing the files back into the sequencing software. Nowadays, numerous distortion effects exist as VST plugins,
which can be opened from within host applications, thus curbing the need for extra standalone applications or external hardware.

That VST plugins can be opened from within a host application, making the process of adding effects to sound files much easier, is a major advantage of the system, and there are several others. As VST is a growing technology, increasingly being implemented by software companies, so the range of VST-compatible software is ever increasing. Further, the fact that VST plugins can be used in any VST host application allows users to work within their chosen environment, thus reducing the need to learn to use new software for certain tasks. From a development point of view, VST makes the task of programming simple audio applications easier. In creating VST plugins programmers do not have to worry about things like file- and window-handling, as these are all managed by the host application. Further, most VST host applications provide their own means of presenting parameters to the user therefore, assuming no highly specific user-interface characteristics are required, developers need not concern themselves with the process of writing a GUI for their plugin.

All of the VST base-classes have been coded such that they will compile for Windows, Apple Macintosh, BeOS, and SGI platforms with no necessary changes to the source code. This makes the task of writing cross-platform software, which can be a very time-consuming process, much simpler. Finally, the VST specification is, more or less, open source, at least insofar as the source code is available to download from the Steinberg website. This means that VST development is open to anyone, and this has resulted in a busy mailing list from which support may be sought.

http://www.steinberg.net
7.2 How Does VST Work?

More technically speaking, VST plugins receive blocks of audio in input buffers from the host application. Output buffers are also provided by the host, and there is no way a plugin can influence the way that the host application deals with these buffers. Within the plugin code, DSP algorithms can be applied to the audio received at the input buffers before it is passed back to the host application via the output buffers. As the plugin simply receives audio from the host without knowing anything about the application itself, so the host similarly receives audio back from the plugin with only very little knowledge of its internal workings:

From the host application’s point of view, a VST Plug-In is a black box with an arbitrary number of inputs, outputs, and associated parameters. The Host needs no knowledge of the plug-in process to be able to use it.
(Steinberg 2000: 3)

On the simplest level, this is the way that host and plugins communicate. VST plugins can also provide various parameters which can be changed in real time either by the user – via a graphical user interface – or automatically by the host, using parameter information recorded from user input.

So how is this communiqué between hosts and plugins actually facilitated? The answer is through the use of standard C++ classes. Steinberg have provided template structures and classes for the ‘generic audio effect’, which contain all the necessary functions for hosts and plugs to communicate. This ‘template’ takes the form of a group of C++ files which are supplied with the VST Software Developers Kit (VSTSDK). These files are named as follows: AEffect.h; AudioEffect.hpp; AudioEffect.cpp; aeffectx.h; auidoeffectx.h; and auidoeffectx.cpp. The first three files represent the VST 1.0 specification which was the first implementation of VST to be commercially released. Those files ending with an ‘x’ represent the VST 2.0 specification which was introduced by
Steinberg in 1999. These are extensions to the original VST specification. Specifically, `audioeffectx` inherits from `AudioEffect`, and therefore incorporates all of its properties.

In developing a plugin, a further C++ class is derived from this template, thus immediately inheriting all of its properties. Developers can then override certain base-class functions to suit the particular needs of their plugin, or create new functions which are internal to the plugin and completely independent of the host application. The important fact is, however, that all the methods of communication between hosts and plugins are standardised, as they are inherited from a generic base-class. It is advisable to be familiar with the VST base-classes and header files when undertaking plugin development. A full explanation of all the methods which make up the base-classes is beyond the scope of this report, but a short summary of how some of the most important functions operate follows presently.

### 7.2.1 The ‘AudioEffect’ and ‘audioeffectx’ Base Classes

As mentioned earlier, it is important to note from the outset that the classes `AudioEffect` and `audioeffectx` are effectively one entity. To all intents and purposes it is reasonable to consider `audioeffectx` as the containing the complete VST specification, as it includes the earlier `AudioEffect`. From this point onwards, reference to `audioeffectx` should be read as ‘AudioEffect and audioeffectx’, unless specific reference to a single one of these base classes is explicitly cited.

`audioeffectx` contains the prototypes of a set of virtual C++ functions which are called by the host application to carry out interactions with plugins. For example, when a host application wants to send audio to a plugin for processing, it will call either `process()` or `processReplacing()`, depending on exactly how the plugin is being used (how Nuendo
calls these functions will be discussed later). In this capacity, the `process()` and `processReplacing()` functions are the most important members of the `AudioEffect` class, as they contain the algorithm that performs the audio transformation. These functions will be described more fully later.

### 7.2.2 Initialisation of Plugins

When a host application opens a plugin, it calls the constructor function of that plugin. This should contain calls to certain `AudioEffect` functions which are necessary to initialise the plugin. The most important of these functions are:

```cpp
class AudioEffect {
public:
    virtual void setNumInputs (long inputs) = 0;
    virtual void setNumOutputs (long outputs) = 0;
    virtual void setUniqueID (long iD) = 0;

private:
    // Additional methods...
};
```

Extract from `AudioEffect.hpp` (Steinberg 2000)

The first two of these functions are fairly self-explanatory: they inform the host how many input and output buffers are required by the plugin. Exactly how this information is acted upon depends on the host application. In Nuendo (and probably others) the number of inputs and outputs specified are used to determine the number of elements in two arrays of pointers. These, in turn, point to the audio input and output buffers. A plugin that declares two inputs and two outputs will cause two input buffers and two output buffers to be allocated. There is no major problem with setting the number of inputs or outputs to be larger than your plugin actually requires, except that plugins may not install under certain circumstances depending on the number of inputs and outputs. In Nuendo for example, a plugin with more than two outputs may not be installed as a channel insert effect. This raises issues which will be discussed further in Chapter 8. Declaring one output and then using two, however, will
almost certainly cause the host application to crash, as audio will be written to an undeclared buffer.

The `setUniqueID` function is used to associate a four-character ID with your plugin which will be used by host applications to identify that plugin. Plugins which do not call this function within their constructor will not be detected by host applications as plugins, therefore it is absolutely necessary to call this function in every VST plugin. Further, every plugin must have a *unique* identity, otherwise it may not be recognised by host applications. It is therefore advisable to use a combination of four characters that are not likely to have been used before. Steinberg keep a record of registered unique IDs, which can be consulted to see if a particular ID has been registered before and is therefore being used by another plugin. This does not mean to say, however, that all VST plugins have their unique IDs registered with Steinberg.

Although the three functions described above are virtual, it should not normally be necessary to override them, as their functionality is so simple. The only circumstance under which these functions are likely to be overridden is if a plugin has to do something else at the same time it sets the number of inputs, outputs, or the unique ID, which seems unlikely.

Further important initialisation data is obtained from the definition of the plugin’s constructor function:

```cpp
Ambipan::Ambipan (audioMasterCallback audioMaster)
    : AudioEffectX (audioMaster, 16, kNumParams)
{ etc... }
```

The type `audioMasterCallback` is passed to the plugin class by the hosting application to handle interaction with the plugin. This is simply passed on to the base-class (`AudioEffectX`) and can be forgotten about. The other parameters passed to `AudioEffectX` — in this case 16, and `kNumParams` — are important as they inform the
base-class and host application how many programs, and how many parameters, respectively, that the plugin has. A program the name for a ‘snapshot’ of all the parameters of a plugin. The value passed to indicate the number of programs (in this case, 16) is assigned to the base-class protected long named numPrograms, which is typically used (among other things) to dynamically allocate memory for an array of plugin-program classes. This process will be described more fully in the section entitled ‘Dealing with Programs’. Plugins often define a few preset programs that the user might find useful. The constant kNumParams can be used to inform the base-class of how many parameters the plugin has, provided that the parameters are enumerated as recommended by Steinberg. In Ambipan, this is done as follows:

```c
enum
{
    kAzimuth,
    kElevation,
    kStereoWidth,
    kStereoHeight,
    kNumParams
};
```

Enumerating parameters like this means that they can be referred to by name rather than by number. Why this is particularly useful will be described in the section entitled ‘Dealing with Parameters’.

### 7.2.3 The `process()` and `processReplacing()` Functions

The `process` and `processReplacing` functions are declared thus:

```c
virtual void process (float **inputs, float **outputs, long sampleFrames)
virtual void processReplacing (float **inputs, float **outputs, long sampleFrames)
```

Extract from AudioEffect.hpp (Steinberg 2000)
Inputs and outputs are pointers to the audio input and output buffers, respectively (actually they are pointers to arrays of pointers which, in turn, point to each of the input and output buffers). Audio is sent from the host application to the plugin through the input buffers, and the plugin returns processed audio to the host via the output buffers. SampleFrames is used to keep track of the current sample being processed. When the host application calls process or processReplacing, it sends a block of a certain number of audio samples to the input buffers. The number of samples in this block will be stored in sampleFrames, which is typically decremented on each cycle of a process loop. When sampleFrames reaches 0 (i.e. when we have processed the last sample from the block of audio sent from the host), the process or processReplacing function returns and (assuming there is some audio left) the host calls it again, sending the next block of audio samples.

As described so far, both process() and processReplacing() work in exactly the same way, but there is one crucial difference. In processReplacing(), the output from the DSP algorithm (that is, the transformed audio) overwrites the plugin output buffers which are sent back to the host application, whereas in process(), the output is added onto the contents of the plugin output buffers. It is important to observe this difference because it is a characteristic that the developers of VST hosting software may have ‘taken as read’ when designing the way in which their application handles audio. For example, in Nuendo, VST plugins which are installed as send effects will result in process() being called. The audio returned from all active send effects in Nuendo is contained in one audio stream. This is possible because it is assumed that the algorithm contained in process() will add its output to this audio stream. If process() overwrites the output buffers, therefore, the output from other audio channels may be overwritten as well. Channel and master insert
effects, however, will cause Nuendo to call `processReplacing()`, where the output from the plugin overwrites the output buffers.

### 7.2.4 Dealing With Parameters

In the vast majority of plugins, the audio process just described will depend on a number of user-definable parameters. For example, a delay plugin will typically allow the user to adjust the length of the delay, the amount of feedback of the delayed signal, and the wet-to-dry ratio of the final output signal. *In VST, all parameters are handled as floating point numbers between 0.0 and 1.0.* The dynamic control of parameters is achieved through a group of functions, again predefined in the `AudioEffect` base-class. The functions which are responsible for the handling of plugin parameters are declared as follows:

```cpp
virtual void setParameter (long index, float value)
virtual float getParameter (long index)
virtual void getParameterName (long index, char *text)
virtual void getParameterLabel (long index, char *label)
virtual void getParameterDisplay (long index, char *text)
```

Extract from AudioEffect.hpp (Steinberg 2000)

Again, note that each of these functions is virtual, allowing developers to override them as necessary to suit their plugin. The first two of these functions deal with the actual changing of parameter values (there is also another function – `setParameterAutomated` – which deals with the automatic updating of parameters; this will not be described here). When the host wants to set a parameter, either from direct user input or from automation, it calls `setParameter`, with `index` identifying which parameter is to be set and `value` indicating what value to assign to that parameter. If parameters have been enumerated, as described earlier, then these functions can refer to them by name rather than by number,
making the process of writing easily understandable code much easier. Therefore, a simple `setParameter` function may look something like this:
void Ambipan::setParameter (long index, float value)
{
    switch (index)
    {
        case kAzimuth :    fAzimuth = value; break;
        case kElevation :  fElevation = value; break;
        case kStereoWidth : fStereoWidth = value; break;
        case kStereoHeight: fStereoHeight = value; break;
    }
}

Here we can see the enum parameter names being used as a switch index to assign the value setParameter was called with to the correct parameter variable. Note that this is a simplified version of the Ambipan setParameter function. The getParameter function works almost identically except that rather than the host assigning a value to a parameter, here the plugin returns a parameter value to the host.

It was mentioned earlier that many VST hosts have default methods of presenting plugin parameters to the user, if the plugin does not define a unique GUI of its own. In these cases, the latter three of the above named functions are called to inform the host application about how to identify the parameters to the user. Specifically, getParameterName will be called by the host to inform it what name to display to identify each parameter (‘Gain’ for example), while getParameterLabel will be called to obtain the name of the unit under which that parameter operates (for example ‘decibels’ or ’dB’). The getParameterName function for Ambipan looks like this:

void Ambipan::getParameterName (long index, char *label)
{
    switch (index)
    {
        case kAzimuth :        strcpy (label, "Azimuth "); break;
        case kElevation :      strcpy (label, "Elevation"); break;
        case kStereoWidth :    strcpy (label, " Width "); break;
        case kStereoHeight :   strcpy (label, " Height "); break;
    }
}
The `getParameterLabel` function is practically identical. Text is copied into a string variable, a pointer to which is passed by the host when it calls the function.

`GetParameterDisplay` is used to perform any conversions necessary to translate between the value of the parameter as used inside the plugin (which, as stated earlier, will be a floating point value between 0.0 and 1.0), and the value that should be presented to the user. In the most simple instance this could merely involve a float to string conversion (for which `AudioEffect` supplies a predefined function, `float2string`), and the user would see a value between 0.0 and 1.0 which is a direct representation of the value of that parameter within the plugin code at that particular time. Normally, however, this function will incorporate some kind of scaling in order that the parameter value be described in terms which are useful to the user. A gain value between 0.0 and 1.0 is not meaningful, for example, so conversion to a decibel scale (using the predefined `dB2string` function) would be included here. The `GetParameterDisplay` function will normally call Steinberg’s `float2string` function, either directly or indirectly, to convert from floating point values (i.e. parameters or scaled representations of them) into text which can be copied into the character array, a pointer to which is passed by the host when it calls the function. Obviously, there is nothing wrong with developers performing their own float to string conversion.

### 7.2.5 Dealing with Programs

As well as allowing plugins to handle multiple parameters, the VST specification also allows plugins to manage multiple programs, that is, ‘snapshots’ of the plugin’s full set of parameters. These can be initialised to various different settings when the plugin is constructed, thus providing a set of different presets for the plugin. In order to incorporate multiple programs into a plugin, it is necessary to define a plugin program class (this process
will be described more fully in Chapter 9). This class will typically have the plugin class itself as a friend class, thus allowing it to interact directly with the plugin’s private variables. This is necessary because when a particular program (i.e. set of parameters) is selected by the user, the chosen program will have to assign its parameter values to the parameters currently being used by the plugin.

As with parameters, there are several functions declared in the AudioEffect base-class that facilitate dealing with programs. These are:

```c
virtual void setProgram (long program) {curProgram = program;}  
virtual long getProgram() {return curProgram;}  
virtual void setProgramName (char *name) {*name = 0;}  
virtual void getProgramName (char *name) {*name = 0;}
```

We can already see that these functions work in a similar way to those provided to deal with parameters. When the host calls setProgram, either when a plugin is installed or when the user changes the program, it passes the function a long containing the number of the chosen program. In the inline definition, all that happens is that the program number passed gets assigned to the variable curProgram. CurProgram is a variable defined in the base-class that is used to identify the program currently in use by a plugin, that is, the program whose parameters are being affected by changes made to parameters by the user. Changes to parameters will only affect those of the current program, and developers must always use the variable curProgram to identify the current program. Developers will usually need to override the setProgram function, because by default all it does is set the current program variable. That is to say, it does not actually change any of the plugin’s parameters to reflect those stored in the program that has just been selected. To this end, Ambipan overrides setProgram as follows:
void Ambipan::setProgram (long program)
{
    AmbipanProgram * ap = &programs[program];
    curProgram = program;
    setParameter (kAzimuth, ap->fAzimuth);
    setParameter (kElevation, ap->fElevation);
    setParameter (kStereoWidth, ap->fStereoWidth);
    setParameter (kStereoHeight, ap->fStereoHeight);
}

In Ambipan’s constructor function, memory for all the programs was allocated as follows:

    programs = new AmbipanProgram[numPrograms];

Recall that numPrograms is a variable declared in the base-class that is assigned the value passed to it by the constructor function of the plugin. AmbipanProgram is the name of the plugin’s program class. We therefore have an array of AmbipanPrograms that is equal in size to the number of programs requested by the constructor function. Returning to our setProgram function, we see that a pointer is created to the program\textsuperscript{th} element of this array – in other words, a pointer to the program that has just been selected by the user. We then use that pointer to assign the parameter values held in that particular program object to the parameter variables that are being used by the plugin. This is done by calling the setParameter function. Notice also that we still assign the program number passed by the host application to the variable that denotes the current program.

The remaining functions used to handle programs are much simpler. GetProgram, as we can see from its inline definition, simply returns the value of curProgram, which as we know reflects the number of the program currently in use by the plugin. SetProgramName and getProgramName operate in a manner identical to setParameterName and getParameterName, which were described earlier. In each case a pointer to a character array is passed by the host application when it calls the function. In the case of
setProgramName, we copy the contents of that array into the name variable of the current program. For getProgramName, we copy the name of the current program into the array passed by the host.
8: OpenGL and the GL Utility Toolkit (GLUT)

All of the graphics rendering which takes place in the Ambipan GUI has been carried out using OpenGL. Most of the other APIs considered (which, unlike OpenGL, are designed specifically for the implementation of GUIs) consist of predefined widgets (sliders, check-boxes, etc.) which are generally not suitable for the real-time manipulation of objects in three dimensions. For this reason it was decided that a custom-designed widget for the positioning of an object on the surface of a sphere was required. OpenGL was chosen primarily because of its suitability for dealing relatively easily with objects in simulated three-dimensional spaces. Also, as a widely accepted standard for computer graphics it is well documented and support is easily and readily available. This chapter will explain the process of rendering graphics with OpenGL using the GL Utility Library (GLUT) to perform tasks such as window handling and user input.

8.1 Description of the OpenGL API

OpenGL is a versatile C-based API capable of rendering two- and three-dimensional graphics. The OpenGL library includes a selection of graphics ‘primitives’ which can be combined to create graphical objects. Primitives include points, lines, and polygons (triangles, quadrilaterals, etc.), and are created by specifying vertices (points) in space. A single point needs only one vertex to fully describe it, whereas a quadrilateral requires four. All OpenGL graphical objects, no matter how complex, are made up from these simple primitives. There are also a number of predefined objects available in external APIs. The GL Utility Toolkit (GLUT), for example, includes predefined objects such as spheres, cubes, and even teapots, to save programmers the trouble of assembling such objects from scratch. The various
functionalities of GLUT will be discussed later. Using these primitives, or external higher level functions which are based on them, programmers can create three-dimensional graphical objects and manipulate them within a scene.

As well as positioning three-dimensional graphical objects, OpenGL allows the programmer to specify and position light sources. These light sources may be of any colour and intensity, and may point in any direction. Programmers may also specify various properties of the material from which graphical objects are made, for example, how much light they reflect. This obviously has a bearing on how the object appears in various different lighting circumstances. Numerous other effects are available, such as depth cueing (distant objects appear dimmer), shadowing, motion-blurring, and so on.

OpenGL has been designed to be hardware-independent, that is, compatible with many different hardware specifications and operating systems. For this reason, OpenGL does not include any window- or user-input-handling facilities, as these tend to be system-specific. However, as with common three-dimensional objects, there are higher-level APIs which can deal with these tasks, some even maintaining a certain degree of platform independence.

8.2 How OpenGL Works

OpenGL is what is known as a ‘state machine’. In other words, the user can put OpenGL into a certain state, and it will remain in that state until it is changed. For example, the current colour is a state variable which can be set by the programmer. Once the current colour has been set, all objects will be drawn with that colour, until it is changed again. The same applies for the many other state variables used in OpenGL.

In terms of rendering graphics, OpenGL uses what is known as a pipeline, that is, ‘a process that can take two or more distinct stages or steps’ (Wright and Sweet 2000: 34). In
basic terms, the pipeline for every frame (‘snapshot’) of graphics that needs to be calculated is as follows:

1. Calls are made to OpenGL API functions – these are placed in the OpenGL command buffer;
2. Vertex information is analysed, and graphical objects are created;
3. Transformation: how the objects should be positioned relative to each other is calculated, based on the positions given for the objects and the chosen vantage point (‘camera angle’);
4. Lighting: the colour of each object is calculated – this is basically based on the material of the object and the specified lighting conditions.
5. Rasterisation: the mathematical description of objects and colours is converted to pixels on the screen.
6. The complete graphical information is passed on to the frame buffer. When this buffer is ‘flushed’, the graphics which have just been calculated are displayed on the screen, assuming that all system-specific windowing tasks and so on have been handled properly.

8.2.1 Initialising OpenGL

Initialising OpenGL involves setting the values of various state variables that will not change at any time during the graphics rendering process. Because of the enormous number of different ways in which OpenGL could be initialised, no attempt will be made to describe the initialisation process in abstract terms. Rather, this section will give a brief explanation of how OpenGL was initialised for the purposes of the Ambipan GUI.
The most important initialisation procedure is the specification of the clipping area and setting of the viewport. Both of these procedures are to do with the way coordinates are understood by OpenGL, and how they are mapped onto the screen. The clipping area can be defined using (among others) the `glOrtho` function:

```c
void glOrtho (Gldouble left, Gldouble right, Gldouble bottom,
              Gldouble top, Gldouble near, Gldouble far);
```

The value assigned to the variable `left` informs OpenGL of what coordinate value should be assigned to the left-most extremity of the x-axis of the screen (window). Similarly, the values given for `right`, `bottom`, and `top` inform OpenGL of the coordinate values to be assigned right, bottom, and top extremities of the window, respectively. The arguments `near` and `far` refer to the coordinates to be given to the Z-axis, or depth-axis, which effectively goes straight ‘into’ the computer screen. Basically, setting the clipping area lets OpenGL know how the x, y, and z axes are numbered. Note that these coordinate mappings remain the same no matter what shape the window is. If the user stretches the window so that it is twice as long as it was originally, the left and right extremities of the elongated x-axis will still be the same as specified when `glOrtho` was last called.

If the rendering window can be freely resized without affecting the coordinate values assigned to its axes, this will cause the image to be distorted if nothing is done to prevent this from happening. For this reason, OpenGL allows the programmer to specify a viewport, which is the area of the window into which OpenGL graphics are rendered. This is carried out using the `glViewport` function. If we code this function so that the aspect ratio (that is, the ratio of height to width) of the viewport is always the same, then the rendered graphics will always have the same aspect ratio, and will not be distorted if the window is resized. Alternatively, if we constrain the rendering window so that it cannot be resized in the first
place, then we can simply configure the viewport to be equal to the entire window, knowing that the lengths of its axes will always remain the same. However, as OpenGL does not deal directly with windowing tasks, it is not possible to do this using OpenGL function calls. Rather it must be done with whatever API is being used for window handling. How to constrain the size of the window using GLUT will be described later.

Other initialisation procedures are mainly concerned with setting various OpenGL state variables that will remain constant throughout the rendering process. These could include the clear colour (the colour that the ‘blank’ screen will be), the shading model (how OpenGL interpolates between different colours), and so on. The various attributes of static light sources can also be specified at the initialisation stage.

### 8.2.2 Positioning Graphical Objects in OpenGL

Primitives are positioned by specifying Cartesian coordinates for each of their vertices. This is done by calling the `glBegin` function with the type of primitive to be drawn as its parameter, then specifying each of the vertices using (for example) the `glVertex3f` function, and then calling the `glEnd` function. This piece of code will render a line between points (0,0,0) and (100,100,100):

```c
  glBegin (GL_LINES);
  glVertex3f (0.0, 0.0, 0.0);
  glVertex3f (100.0, 100.0, 100.0);
  glEnd();
```

Obviously, how this line appears on the screen will depend on how the coordinate system was set up.
This method is fine for objects that will always be in the same place, and at the same angle of rotation. But suppose we want to specify an object which can be rotated, or moved around in the scene. To specify all of the vertices with variable modifiers would be cumbersome if the object is moving around, and near-impossible if a three-dimensional object is to be rotated. For this reason, we have the `glTranslatef` and `glRotatef` functions, which are used for moving and rotating the coordinate system, respectively. The `glTranslatef` function is used for moving the coordinate system along a straight line, so that the origin (0,0,0) is moved to the newly specified position. The `glRotatef` function is used to rotate the coordinate system by a specified angle about a specified arbitrary axis. By shifting the coordinate system in such a way we are able to position, for example, predefined GLUT objects, which are automatically positioned at the origin. We can also rotate the coordinate system, say, to display a solid cube at a certain angle of rotation without having to manually calculate the positions of each of the vertices.

8.3 Description of the GLUT API

As mentioned earlier, OpenGL does not include any functions for dealing with window handling, input and output, or any of the other platform-specific interactions that a user interface may have to call upon. It deals only with the rendering of graphics and is neither concerned with the means by which these graphics are to be displayed by the operating system, nor the way in which the user may interact with the graphics rendering process. Essentially, it is not possible to write a useful piece of software using only OpenGL: another API is required to provide the means by which OpenGL graphics can actually be displayed on a computer monitor. There are numerous APIs under which OpenGL can be implemented to add these functionalities. The chosen API for the Ambipan interface is the GL Utility Toolkit
GLUT). It has been chosen because it is well documented and support is available within York University (Fletcher 2001). GLUT is also recommended and supported in many of the leading publications on OpenGL programming, including Angel (2000), the second edition of Neider, Davis, and Woo (1993), and Wright and Sweet (2000). Additionally, GLUT has been written in such a way that code will compile under X-Window, Windows, and OS2 platforms. This would not be possible, for example, if the Windows-specific Microsoft Foundation Classes (MFC) had been used.

Essentially, GLUT is the framework (or, rather, one possible framework) within which OpenGL rendering can take place. GLUT processes events using callback functions, that is, functions which the programmer has specified should be called in response to certain events. For example, the following call:

```c
glutDisplayFunc(DrawGLScene);
```

will register a function named `DrawGLScene` as the function to be called whenever GLUT requires the OpenGL rendering to actually take place. It is this function that will contain the OpenGL drawing routine and the majority of the calls to GL functions. Callback functions are registered (specified) when GLUT is initialised, for instance, in the `main` function of a piece of C code. Some of the more important callback functions will be described shortly.

### 8.3.1 Initialising GLUT

There are a number of standard GLUT function calls associated with initialisation. The `glutInit` function is used to initialise the GLUT library in the first instance. Other than the fact that this must be the first GLUT function to be called, programmers need not be
concerned with exactly what it does, which in any case is not made particularly clear in the
documentation. Some of the other functions called to initialise GLUT and create windows are
declared as follows:

```c
void glutInitWindowSize (int width, int height);
void glutInitWindowPosition (int x, int y);
void glutInitDisplayMode (unsigned int mode);
```

The first two of these function calls are self-explanatory. The integers width and height
which are taken as parameters by the glutInitWindowSize function refer to the
dimensions, in pixels, of the window, while x and y of glutInitWindowPosition are
the coordinates, again in pixels, of the origin (0,0) of the window. The functions used to
actually create and destroy windows will be discussed in the next section.

The last of the three functions given above is used to set a few properties of the way in
which OpenGL graphics are rendered into the window(s). This is done by OR’ing various
GLUT macros which describe the properties of the resulting display mode. A typical call to
glutInitDisplayMode would be as follows:

```c
glutInitDisplayMode (GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
```

The first and last macros respectively indicate that colours will be treated as containing red,
green, blue, and alpha (transparency) components, and that a depth buffer will be created for
positioning of objects further ‘into’ the screen. The GLUT_DOUBLE macro means that two
buffers will be used for rendering graphics. This will result in much smoother animation,
because the frame stored in one buffer will be displayed while the next frame is being
rendered into the other buffer. Buffers are then swapped every time the display needs to be
updated, using the glutSwapBuffers function.
8.3.2 Window Handling in GLUT

Dealing with windows in GLUT is facilitated by a few simple functions. The most important of these functions are as follows:

```c
int glutCreateWindow (char *name);
int glutDestroyWindow (int win);
void glutReshapeWindow (int width, int height);
```

The first of these functions will create a window with the characteristics specified with calls to the initialisation functions (see previous section). The variable `name` specifies the title that will appear along the title bar of the window. As this function returns an integer value, so it must be assigned to a variable which is then used as the identifier for that window. For example:

```c
window = glutCreateWindow("AmbiPan");
```

where `window` is a variable of type `int`. How the `glutDestroyWindow` function works should now be obvious. The integer `win` is the identifier of the window to be destroyed.

The last of these functions is, again, self-explanatory. A call to this function will resize the current window to the dimensions given in `width` and `height`, as usual in pixels. If we make a call to this function whenever the user tries to resize the window, and that call always specifies the same values for `width` and `height`, then we can constrain the window so that it cannot be resized. The reasons why this might be useful were explained in the previous section entitled ‘Initialising OpenGL’.
As the use of the expression ‘current window’ in the previous paragraph suggests, there are GLUT functions for the creation of sub-windows, and for selecting which window is ‘current’ (active). There are also functions which reposition windows, functions which stack windows, functions which iconify windows, and several others. These, however, will not be described here, as they are not implemented in the Ambipan GUI.

### 8.3.3 The `glutMouseFunc` and `glutMotionFunc` Callbacks

Earlier it was mentioned that GLUT handles events by calling various registered functions when certain types of events occur. We have stated that the function given as the parameter to glutDisplayFunc will be registered as the callback for OpenGL graphics rendering. Further callback functions add mouse functionality to GLUT applications. The callback registration functions are declared as follows:

```c
void glutMouseFunc (void (*func) (int button, int state, int x, int y));
void glutMotionFunc (void (*func) (int x, int y));
```

These are used to register the functions which GLUT will call to handle information received by the operating system from the mouse. In this way, programmers can incorporate mouse interaction into OpenGL applications. While the glutDisplayFunc function takes as its parameter a pointer to a function which itself receives no further parameters, we can see that the above functions take as their parameters pointers to functions which do themselves require further parameters. These parameters are passed by GLUT to the appropriate callback function whenever that particular function is called. It is important to ensure that the various callback functions registered with GLUT are designed to receive the parameters that GLUT
will pass to them. For example, if we register a function called Mouse as the GLUT mouse callback, then GLUT will expect that function to take four integers as its parameters.

The integer parameters x and y that GLUT will pass to the callbacks registered with each of the above functions, will contain the x and y coordinates of the mouse pointer at the time the callback function was called. These coordinates will be in pixels relative to the current window (GLUT applications may run more than one window simultaneously but only one can be active at any given time). In GLUT, the top left-hand corner of the current window is considered to be (0,0), therefore X coordinates increase to the right, while Y coordinates increase downwards. This contradicts the OpenGL coordinate scheme, which usually considers the lower-left corner of a window to be (0,0). However, converting the coordinates returned by GLUT to correspond with those used in OpenGL is a straightforward task.

The integer parameters button and state will be passed GLUT macros which reflect the mouse button in question, and whether it has just been depressed or released, respectively. The values which can be passed to button are \texttt{#defined} in the GLUT header as \texttt{GLUT\_LEFT\_BUTTON}, \texttt{GLUT\_MIDDLE\_BUTTON}, and \texttt{GLUT\_RIGHT\_BUTTON}, the meanings of which are self-explanatory. The variable state will indicate whether that button has just been pressed (\texttt{GLUT\_DOWN}) or released (\texttt{GLUT\_UP}).

Given this information, programmers can write routines which act in various different ways depending on the type of mouse input received. These routines could include using mouse input to change the values of variables which are, in turn, used as parameters in the OpenGL rendering process. In this way, mouse input can be used as a means to interact directly with the graphics on screen, which is one of the most fundamental principles of many GUIs.
9: Implementation

All coding was carried out using Microsoft’s Visual Studio 6.0 C++ compiler. The following sections will describe the Ambipan and Ambidec plugins and the GUI for Ambipan from a more technical standpoint, as well as discussing some of the issues encountered throughout the implementation process. It is hoped that the broader contexts of Chapters 7 and 8 will facilitate an easier understanding of the processes involved in implementing these programs. To this end, the description of the implementation will be limited, where possible, to a simple demonstration – using code extracts – of how the principles and procedures outlined in Chapters 7 and 8 have been applied to these particular programs.

9.1 Implementation of Ambipan Plugin

9.1.1 Initialisation

Firstly, we enumerate the parameters that will be used in the operation of the Ambipan plugin. This simple process was described in Chapter 7 but there is no harm in reiterating:

```cpp
enum {
    kAzimuth,
    kElevation,
    kStereoWidth,
    kStereoHeight,
    kNumParams
};
```

Next, we declare the AmbipanProgram class, which defines an object used to store parameter values in a ‘preset’. Its declaration is simple, and as follows:
class Ambipan;

class AmbipanProgram
{
friend class Ambipan;
public:
AmbipanProgram();
~AmbipanProgram() {}
private:
float fAzimuth, fElevation, fStereoWidth, fStereoHeight;
char name[24];
};

We must make a forward-declaration of the class Ambipan, because it is declared as a friend class of AmbipanProgram before it is itself prototyped. We know already that each AmbipanProgram object will have its own private set of variables into which parameter values will be stored. These follow the same nomenclature as is used in the main Ambipan class. Each program will also have its own name, stored in a character array. Notice that the destructor function, ~AmbipanProgram, does not do anything: destruction of AmbipanProgram objects is taken care of by the destructor function of the Ambipan class.

We will now examine the Ambipan plugin class itself:

class Ambipan : public AudioEffectX
{
public:
Ambipan(audioMasterCallback audioMaster);
~Ambipan();

virtual void process(float **inputs, float **outputs, long sampleframes);
virtual void processReplacing(float **inputs, float **outputs, long sampleFrames);
virtual void setProgram(long program);
virtual void setProgramName(char *name);
virtual void getProgramName(char *name);
virtual void setParameter(long index, float value);
virtual float getParameter(long index);
virtual void getParameterLabel(long index, char *label);
virtual void getParameterDisplay(long index, char *text);
virtual void getParameterName(long index, char *text);
private:
    AmbipanProgram *programs;
    float fAzimuth, fElevation, fStereoWidth, fStereoHeight
    oldAzimuth, oldElevation, oldWidth, oldHeight,
    newAzimuth, newElevation, newWidth, newHeight,
    i; // interpolation factor: 0.999 seems to be good
}

Here we can see declarations of all the AudioEffect functions that Ambipan will override (all declared virtual). All of these were described in Chapter 7, and so should be familiar already.

We know that when a plugin is installed, the hosting application calls the constructor function for that plugin. Ambipan’s constructor function looks like this:

```cpp
Ambipan::Ambipan (audioMasterCallback audioMaster)
    : AudioEffectX (audioMaster, 16, kNumParams) // 16 progs (4 params)
{
    programs = new AmbipanProgram[numPrograms];

    // INITIALSE PARAMETERS TO 0 ON CONSTRUCTION OF Ambipan OBJECT //
    fAzimuth = fElevation = fStereoWidth = fStereoHeight
    = oldAzimuth = oldElevation = oldWidth = oldHeight = 0;
    i = 0.999; // interpolation factor for parameters - to stop clicking

    if (programs)
        setProgram (0);
    setNumInputs (2);
    setNumOutputs (4);

    canProcessReplacing ();
    canMono(); // means the plugin CAN be used as a send effect
    setUniqueID ('HZST'); // = horizontal stereo compatible
}
```

The arguments taken by this function and its base-class AudioEffectX were described in Chapter 7 and will not be covered again here. Next, we see the creation of an array of AmbipanProgram objects, one for each of the sixteen user-definable programs. All of the variables are initialised to zero on the construction of an Ambipan object, with the exception of i, which is set to 0.999. This variable is a multiplier which is used to interpolate between
changed parameter values – this process will be described in due course. The default program is set to 0, and then we declare two inputs (so we can have mono or stereo input) and four outputs (for the four B-format channels). The call to `canProcessReplacing` informs the host application that Ambipan allows its output to overwrite the output buffers, that is, there will be none of the original input signal mixed into the output. Calling `canMono` informs the host that monophonic input is also acceptable. The Ambipan header file also defines a macro named `TO_RADIANS`, which has the value 6.2831853, (approximately) equal to $2\pi$. This is used to scale parameters from $0.0 – 1.0$ to $0.0 – 2\pi$, thus giving the full range of $0^\circ$ to $360^\circ$ (in radians). These new values can then be used to express angles of azimuth and elevation.

### 9.1.2 A Closer Look at Parameter Handling using Programs

We will now take a closer look at the way parameters are set in Ambipan. A simplified version of the code fragment that Ambipan uses to do this was given in the section entitled ‘Dealing with Parameters’. It is simplified insofar as it doesn’t take into consideration the fact that Ambipan has sixteen user-configurable programs. If this simplified code was used in the final plugin, it would work in the sense that changes made to the parameters would be reflected in the audio output. However, these changes would *not* be applied to the parameters stored in the current program. Therefore, if the user was to set some parameters on program 1, change to program 2, and then return to program 1, the changes he/she made to the first program would not have been stored. On changing the program, parameters would simply return to their default values as initialised in the constructor function of the `AmbipanProgram` class (i.e. all parameters $= 0.0$). The actual code used to set parameters in Ambipan is as follows:
void Ambipan::setParameter (long index, float value)
{
AmbipanProgram * ap = &programs[curProgram];

switch (index)
{
    case kAzimuth :
        fAzimuth = ap->fAzimuth = value;
        newAzimuth = fAzimuth * TO_RADIANS; break;

    case kElevation :
        fElevation = ap->fElevation = value;
        newElevation = fElevation * TO_RADIANS; break;

    case kStereoWidth :
        fStereoWidth = ap->fStereoWidth = value;
        newWidth = fStereoWidth * (TO_RADIANS/4); break;

    case kStereoHeight:
        fStereoHeight = ap->fStereoHeight = value;
        newHeight = fStereoHeight * (TO_RADIANS/4); break;
}

if (editor)
    editor->postUpdate();
}

Firstly, a pointer to the current program is created – recall that programs is an array of AmbipanProgram objects, and that curProgram, used here to index the array, refers to the program currently in use. We then switch between parameters exactly as described in the earlier simplified version of this function. But as well as assigning the parameter value passed by the host to the parameter variable being used by the plugin (e.g. fAzimuth =), we also assign it to the copy of that parameter held in the current program (ap->fAzimuth =). In this way, changes made to the plugin’s parameters are also made to the parameters of the current program. To return to our previous scenario, now if the user makes changes to program 1, then switches to program 2 and back again, the changes he/she made will have been stored to the parameter variables held in program 1, and will therefore still be there on returning to this program setting.
9.1.3 Interpolation Between Changed Parameter Values to avoid Clicking

From the code extract given above, we can also see that four variables named newAzimuth, newElevation, newWidth, and newHeight, are assigned versions of the corresponding parameters which have been scaled to angles in radians. This is part of the process that interpolates between old and new parameter settings when parameters are changed, a routine which is necessary to prevent unacceptable clicks in the output audio stream resulting from sudden changes in parameter values. The code that actually carries out the interpolation is incorporated into the audio process loop, but is more easily demonstrated if we miss out the actual audio process:

```c
while(--sampleframes >= 0)
{
    azimuth = (i*oldAzimuth) + ((1-i)*newAzimuth);
    elevation = (i*oldElevation) + ((1-i)*newElevation);
    width = (i*oldWidth) + ((1-i)*newWidth);
    height = (i*oldHeight) + ((1-i)*newHeight);

    //***********************
    /* AUDIO PROCESS GOES IN HERE! */
    //**************************/

    oldAzimuth = azimuth;
    oldElevation = elevation;
    oldWidth = width;
    oldHeight = height;
}
```

Here, azimuth, elevation, width, and height are the variables which actually directly affect the audio process, that is, it is the values held in these variables that are substituted into the ambisonic encoding equations. NewAzimuth, newElevation, newWidth, and newHeight, as we know, are set in the function setParameter, in other words whenever that particular parameter is changed. If we consider the first cycle around the loop after parameter $N$ is changed, the variable old$N$ will contain the value held by $n$ (the variable that actually affects the audio processing) before it was changed. Then $n$, will
become \((0.999 \times \text{old}N) + (0.001 \times \text{new}N)\) (recall that we initialised \(i\) to 0.999 in the plugin’s constructor function), before \(\text{old}N\) becomes the value we have just assigned to \(n\). In this way, when a parameter changes, the corresponding variable which is used to control the audio processing will not change instantaneously, but rather will interpolate exponentially between the old parameter value and the new. The decision to set the scaling factor, \(i\), to 0.999 was reached via a process of experimentation. Values much below 0.999, it was found, did not totally eliminate the clicking, while values higher than 0.999 would be imposing unnecessary demands on the CPU considering that 0.999 proved itself to be adequate.

9.1.4 Handling of Stereophonic Input Signals

The way in which Ambipan handles stereophonic input signals using the stereo width and stereo height parameters was described earlier in Chapter 6. The next few paragraphs will examine how this specification has been implemented. Stereo width and stereo height parameters are ultimately controlled by the variables named \(f\text{StereoWidth}\) and \(f\text{StereoHeight}\), respectively, which range from 0.0 to 1.0. We have seen from Ambipan’s \texttt{setParameter}\ function that the values stored in these variables are multiplied by \((\text{TO\_RADIANS}/4)\), resulting in a range of 0.0 to \(\pi/2\) (or 0° to 90° if we think in terms of angles in degrees). These scaled values are then assigned to \(\text{newWidth}\) and \(\text{newHeight}\), respectively, where they become involved in the interpolation process described earlier. So for the control of stereo height and stereo width we effectively have two parameters, each with a range of 0 to \(\pi/2\) radians. If we consider the absolute azimuth of the sound source (i.e. where a monophonic source would be positioned at that angle of azimuth) to be 0, and width to be the value of the scaled stereo width parameter (i.e. in the range 0 to \(\pi/2\)) then:
L = 0+width
and
R = 0-width

This means that the angle of azimuth of the ‘left’ channel of the stereo input signal will be increased in the positive direction (i.e. anticlockwise, by ambisonic convention) by an angle of width radians. The azimuth of the ‘right’ channel will be decreased (rotated in the negative direction) also by an angle of width radians. The angle between the ‘left’ and ‘right’ channels is therefore equal to 2width, and it is this value that the user will be presented with as the stereo width. Similarly with stereo height, if we take the absolute angle of elevation as Φ, and height to be the value of the scaled stereo height parameter (0 to π/2), then:

L = Φ+height
and
R = Φ-height

The elevation of the ‘left’ channel will be modified in the positive direction, and that of the ‘right’, in the negative, both by an angle of height radians. The value presented to the user as the ‘stereo height’ will therefore be equal to 2height.

In code this is achieved by taking the left and right channels of the stereo input file and carrying out ambisonic panning on each of them individually, thus creating two sets of B-format signals, one for the ‘left’ channel and one for the ‘right’. These two sets of B-format signals are then mixed by adding them together and dividing by two, resulting in the final B-format output channels. This process is as follows:

```c
// B-format encoding for 'left' channel of stereo input file
w1 = *in1 * 0.7071;
x1 = *in1 * cos(azimuth + width) * cos(elevation + height);
y1 = *in1 * sin(azimuth + width) * cos(elevation + height);
z1 = *in1 * sin(elevation + height);
```
Here *in1 and *in2 are pointers to the two plugin audio input buffers supplied by the host application, referring respectively to the left and right channels of the stereo input file. We can see standard B-format encoding equations being applied to derive w1, x1, y1, and z1 for the ‘left’ channel, and w2, x2, y2, and z2 for the ‘right’ channel. The absolute angles of azimuth and elevation (represented here by azimuth and elevation) are modified by adding (for the ‘left’ channel) or subtracting (for the ‘right’) the values stored in width and height, respectively. The two separate sets of B-format signals are then mixed and assigned to w, x, y, and z, which will then each be sent to a separate plugin output buffer.

9.1.5 Temporary Implementation as a Send Effect

At present, Ambipan will only install in Nuendo as a channel send effect, and it is for this reason that all of the ambisonic encoding takes place in the process function rather than in processReplacing, where we would expect it to happen. Because the plugin output buffers are overwritten, only one Ambipan plugin can be installed at any one time, because installing further plugins would overwrite the output of the first one. This is, obviously, not an ideal scenario but is, for various reasons, unavoidable at this stage in the plugin’s development. The plugin may not be installed as a channel insert effect as would be desirable, because Nuendo does not allow insert effects to have more outputs than exist on the channel in which
the insert is installed. Because Nuendo channels can only be mono or stereo, so a plugin with four outputs is not permitted as a channel insert. Channel inserts are permitted to have more than two output if they declare themselves in the ‘spatialiser’ category (how to do this is explained in Chapter 11). However, spatialiser plugins cannot use Nuendo’s default parameter interface, but must define their own GUI for dealing with user input. Because the GUI for Ambipan is not yet in such an advanced state of development as to be integrated with the plugin, so implementation of Ambipan as a spatialiser plugin has, thus far at least, not been possible.

9.2 Implementation of Ambidec Plugin

The implementation of the Ambidec plugin is much simpler than that of Ambipan. Firstly, it only contains one program, thus preventing the need for a separate plugin-program class. This also simplifies the content of the overridden functions for parameter and program handling. Additionally, Ambidec only has one parameter – the chosen speaker rig to decode to – and this further simplifies the parameter-handling functions.

The constructor of the Ambidec class makes a call to setNumOutputs declaring eight audio output buffers. This means that no matter which decoding option is selected, there are still eight audio output buffers in existence. Exactly how many of these buffers will be used depends on the chosen decoding option.

The declaration of the Ambidec class contains the following private variables:

```c
float fSpeakerRig;
int speakerRig;
char programName[32];
```
The character array `programName` is simply used to give the name “default” to Ambidec’s single program preset, and does not have any major bearing on the way that the code works. The value of this parameter used to select the rig to decode to (which, like all VST parameters, is a floating point number between 0.0 and 1.0) is stored in `fSpeakerRig`. Ambidec also `#define` five macros which represent the present five decode rig options: `MONO`; `STEREO`; `SQUARE`; `THREE_TWO`; and `CUBE`. One of these macros is assigned to the integer `speakerRig` depending on which rig the user has selected. The user selects which rig to decode to using the interface provided by the host application, which in Nuendo is a single slider. When the user moves the slider, the host application calls Ambidec’s `setParameter` function as follows:

```cpp
void Ambidec::setParameter (long index, float value) {
  fSpeakerRig = value;

  // set speakerRig flag for parameter display and
  // to use as switch index in processReplacing()
  if (fSpeakerRig > 0.8) speakerRig = CUBE;
  else if (fSpeakerRig > 0.6) speakerRig = THREE_TWO;
  else if (fSpeakerRig > 0.4) speakerRig = SQUARE;
  else if (fSpeakerRig > 0.2) speakerRig = STEREO;
  else speakerRig = MONO;
}
```

The value passed by the host application (which, recall, is a floating point number between 0.0 and 1.0) is assigned to the variable in which the parameter is stored. The way in which the value of `fSpeakerRig` then determines which macro is assigned to `speakerRig`, as viewed on the Nuendo default interface, is as follows:
This is achieved in code by the sequence of if and else statements.

The integer `speakerRig` is also used in Ambidec’s `processReplacing` function, as the index of a switch which outputs the correct speaker feed signals to the number of output buffers required by the rig in question. The decoding equations used are the ‘in-phase’ response equations as given in Furse (2000). These give better localisation characteristics over a larger listening area (i.e. not just in the very centre of the rig) at the expense of a little directional fidelity. As Ambidec will always be used as an insert effect in the master section of the VST mixing desk, so there will never be any need to mix the output of the plugin with the original input signal (indeed, it would simply not make sense to do so). For this reason, the process function, which would normally mix the input and output, does not do anything at all.

As we know that the four inputs which will be received by the Ambidec plugin are ambisonic B-format channels in the order W, X, Y, Z, we can simply assign each of the four plugin input buffers to variables with these names. Having done this, the remainder of the `processReplacing` function is just to assign the result of each speaker-feed equation to the corresponding output buffer as per the speaker orders given in Chapter 6.

**9.3 Implementation of Ambipan GUI**
The graphical user interface for the Ambipan plugin is probably the most complicated entity covered in this report. Its present stage of implementation is described in the following paragraphs.

### 9.3.1 Setting up and Initialising the Window

The window into which all OpenGL graphics are rendered in the Ambipan GUI was created with the following GLUT function calls:

```c
    glutInitWindowSize(Xsize,Ysize);
    glutInitWindowPosition(0,0);
    window = glutCreateWindow("AmbiPan");
```

Window is the integer variable declared to hold the identifier for our window, while global variables Xsize and Ysize, the dimensions of the window in pixels, have been initialised to values of 450 and 473 respectively. The call to glutInitWindowPosition dictates that the top-left corner of the window will be positioned in the top-left corner of the screen. We register the function ReSizeGLScene as the GLUT reshape function, which is called every time the user resizes the rendering window. This function is as follows:

```c
    GLvoid ReSizeGLScene(GLint Width, GLint Height)
    {
        glutReshapeWindow (Xsize, Ysize);
    }
```

If the user tries to resize the window, glutReshapeWindow will be called to maintain the dimensions of the window as Xsize by Ysize. Thus, it is effectively impossible to resize the window.
In Chapter 8 we made the distinction between the coordinate system used by GLUT, and that understood by OpenGL. For the purposes of the Ambipan GUI it is desirable to initialise these coordinate systems so that they are effectively the same. In this way one can reference the other without the need to perform any conversion between the two. The coordinate system used by OpenGL is set up as illustrated in the diagram below.

As we can see, the dimensions of the x and y axes match the dimensions of the GLUT window. This means that the mouse coordinates returned by the GLUT mouse input callback functions can be inserted directly into the OpenGL code, thus making the process of mouse interaction much easier. However, recall that in GLUT, the y axis extends from zero at the top of the window, to its maximum value at the bottom. We can see from the diagram above that in our coordinate system, it is the other way round. This problem is overcome where necessary by reassigning the mouse coordinates passed by GLUT to our mouse callback functions (these are \( mx \) and \( my \), respectively) to two new global variables named \( \text{pointerX} \) and \( \text{pointerY} \) as follows:
Within this window are three solid spheres which represent the three different elevations of the sound-field. Nine global variables are initialised to describe the centre positions of these spheres. These are backX, backY, backZ, sideX, sideY, sideZ, planX, planY, and planZ, and are initialised so that the coordinates of the centre positions of the back, side, and plan-view spheres are (75,75,225), (375,225,225), and (225,375,225), respectively. A further two global variables, largeRadius and smallRadius, are used to carry the values for the radii of the sound-field and sound-source spheres, respectively. These are initialised as 50 and 6 respectively.

### 9.3.2 Interfacing with Graphical Elevations of the Sound-field

In Chapter 6 it was stated that the user can position a sound source anywhere on the surface of the sphere (sound-field) by clicking at a position on one of the three different elevations of the sound-field provided by the GUI. It is therefore necessary to create a routine which determines which of these three elevations the has chosen to interface with. For this purpose, a function named determineControlArea was defined:

```c
int determineControlArea (int left, int bottom, int right, int top)
{
    if (pointerX >= left)
    {
        if (pointerX < right)
        {
            if (pointerY >= bottom)
            {
                if (pointerY < top)
                {
                    return 1;
                }
            }
            else return 0;
        }
    }
    else return 0;
}
```
The four integers passed to this function are two sets of coordinates which represent the bottom-left and top-right corners of a rectangle within the interface window, which will be referred to as a ‘control area’. The function will return 1 if the mouse pointer is inside this control area, otherwise it will return 0. By calling this function with three control areas which surround each of the solid spheres, it is therefore possible to determine which one the user is trying to control. The control areas can be seen as squares drawn around each elevation sphere in the diagram below.

We can now go on to describe how we use mouse coordinate values passed by GLUT to position the sound source – represented by a small red sphere – at the correct position on the surface of each of the different sound-field elevation spheres. This is done by calculating x, y, and z coordinates for the centre of the sound-source sphere, based on the mouse coordinates passed when the user clicks somewhere inside one of the active positioning squares defined by our calls to detemineControlArea.

Our three elevations of the sound-field represent one entity viewed from three different angles. The two dimensions of freedom offered by the mouse correspond with two simultaneous dimensions of control possible within each elevation, while the remaining third dimension must be calculated from the other two. In the case of the back elevation, the two dimensions of mouse control correspond with movements around the xy plane: the z position must be calculated. In the plan elevation, the mouse directly controls movements in the xz plane, while the y position must be calculated. In the side elevation, the user can directly specify the yz position, but the x value must be calculated. This is similar to the ‘two-axes
valuator' paradigm (Chen et al. 1986), which was discussed in Chapter 5, and is better explained diagramatically:

We have now explained how mouse coordinate values are passed to our mouse control callback function, how we use `determineControlArea` to ascertain which sound-field elevation the user is interacting with, and which axes each elevation allows the user direct control over. How we derive cartesian coordinates for the position of the sound-source on the surface of the sound-field from this information will now be explained, using the back elevation as an example.
9.3.3 Determining the Sound-Source Position using Cartesian Coordinates

We can deduce that, in the case of the back elevation, the x mouse coordinate (pointerX) reflects the position of the sound source along the x axis of the sound-field, offset by the x value of the centre of the back-elevation sphere (backX). Similarly, pointerY reflects the position of the sound source along the y axis of the sound-field, offset by the value of backY. Cartesian coordinates are related to the radius of a sphere according to the following equation:

\[ x^2 + y^2 + z^2 = r^2 \]

where x, y, and z are cartesian coordinates relative to an origin at the centre of the sphere, and r is the radius of the sphere. For our spheres, where the radius = 50, the x, y, and z coordinates will correspond with the axes given in the diagram below.

![Diagram of coordinates relative to the centre of each sound-field elevation sphere](image)
For the back elevation sphere, we know the values of \( x \) and \( y \) and \( r \), and can work out the value of \( z \) by rearranging the formula as follows:

\[
z = \sqrt{r^2 - (x^2 + y^2)}
\]

Substituting in the appropriate variables from the program code gives:

\[
sourceZ = \sqrt{(largeRadius^2 - ((pointerX-backX)^2 + (pointerY-frontY)^2))}
\]

Here we see the centre sphere coordinates ((backX, backY)) being subtracted from the mouse coordinates ((pointerX, pointerY)). This is because the mouse coordinates are relative to the entire window, whereas we want coordinates relative to the centre of the back elevation sphere.

But consider the fact that every pair of \( xy \) values, there are two possible values of \( z \): one negative and one positive. We resolve this ambiguity, in this case, with the following line of code:

\[
if \ (\text{rightButton}) \ sourceZ = -sourceZ;
\]

Quite straight forward: if the user is controlling the position of the sound-source on the back elevation by clicking with the right mouse button, then the resulting \( z \) value calculated will always be negative (i.e. on the ‘screen side’ of the sphere), otherwise it will be positive. The rightButton flag is set to 1 in our glutMouseFunc callback function if the right-hand mouse button is pressed, otherwise it is set to 0. Similarly, a flag named leftButton exists to describe the state of the left-hand mouse button.
9.3.4 Determining the Sound-Source Position using Polar Coordinates

The routine described above works fine as long as the mouse pointer is actually ‘on’ the surface of the sphere. However, recall that the control areas defined for each of the spheres are actually wider and taller than the spheres themselves (by 25 pixels on each side, to be precise). This means that it is possible for the user to click outside the surface area of the sphere but still be within the control area for that sphere. In the case of the back-view sphere, this could cause values of x and y to be specified that are actually beyond the surface of the sphere, which would make the right-hand side of the equation used to calculate z, negative. As it is impossible to calculate the square root of a negative number, this will cause an error. Therefore, we need to work out if the point the user is clicking on is on the surface of the sphere and if not, position the sound source on the point of a straight line drawn between the centre of the sphere to the mouse pointer that intersects with the edge of the sphere. To do this, we need to work out the length of that straight line: if it is longer than the radius of the sphere, then the mouse pointer is outside the surface area of the sphere. This line then becomes the hypotenuse of a right-angled triangle, whereby we can work out the angle of rotation about the z-axis (remember, we are still using the back-elevation sphere as an example) subtended to the positive y-axis, a positive rotation being in the anti-clockwise direction. If we know this angle, we can place the sound source on the surface of the sphere at that angle of rotation about the z-axis, and on the plane z = 0. The lines and angles (θs) which we may have to calculate are illustrated in the diagram below, where each white point represents a place outside the surface of the sphere where the user could potentially try to place the sound source.
From the diagram above we can see how the x and y coordinates of the mouse pointer, and the x and y coordinates of the centre of the back-elevation sphere, are used to calculate the lengths of the sides of the triangle opposite and adjacent to the angle $\theta$. We can therefore work out the size of the angle according to the following trigonometric formula:

$$\theta = \arctan ()$$

We can also see that the way in which coordinate variables are used to calculate the lengths of the required sides of the triangle are different depending on which quadrant the mouse click occurred in (quadrants are numbered 1 to 4). It is therefore also necessary to ascertain which quadrant of the control area a mouse click occurred in, and this is achieved by the following function:

```c
int determineQuadrant (int oX, int oY)
{
    if (pointerX < oX)
    {
        if (pointerY > oY)
        {
            return 1;
        }
    }
    else
```
The arguments \( oX \) and \( oY \) passed to this function are the x and y coordinates of the centre (origin) of the sphere in question, so in the case of the back-elevation sphere we pass the values \( \text{backX} \) and \( \text{backY} \). The function will then return 1, 2, 3, or 4, depending on which quadrant the mouse-click occurred in. Therefore, if we call \text{determineQuadrant} as the index of a switch, we can calculate the opposite and adjacent sides of the triangle accordingly in each case, and then calculate the size of the angle from these. It can be seen from the diagram above which angle we are calculating in each quadrant – this is represented in each case by \( \theta \). Only in the first quadrant, however, does the calculated angle actually represent the full angle of rotation subtended to the positive y-axis. Therefore, in the second quadrant we must add 90° \((\pi/2)\) to the calculated angle, in the third quadrant we must add 180° \((\pi)\), and in the fourth quadrant we must add 270° \((1.5\pi)\).

Having worked out the appropriate angle, x and y coordinates for the sound source are calculated using the following formulae:

\[
x = r \times \sin(\text{angle}) \\
\text{and} \\
y = r \times \cos(\text{angle})
\]
where $r$ is the radius of the sphere and angle is the final angle calculated from the procedures given above (i.e. $0$ plus whatever proportion of $\pi$ is required by the quadrant). In the case of the back-elevation sphere, we know that $z$ will always be equal to zero under these circumstances, because the $x$ and $y$ coordinates will always be placing the sound source on the very edge of the semi-sphere, where $z = 0$ is the only possibility.

9.3.5 Deriving Sound-Source Coordinates in Code

We have now defined two different ways in which cartesian coordinates are obtained for the position of the sound source on the ‘surface’ of the sound-field. Which method is actually used is determined by whether the mouse click occurred on or off the surface of the sphere, in other words, whether the hypotenuse of the triangle(s) given in the diagram above is less-than-or-equal in length to the radius of the sphere, or greater in length. As we know what arguments define the lengths of the opposite and adjacent sides of the triangle for each quadrant, we can define the length of the hypotenuse using the following trigonometry:

$$\text{hypotenuse} = \sqrt{\text{opposite}^2 + \text{adjacent}^2}$$

As we are squaring the numbers, so it does not matter if the lengths of the opposite and adjacent sides are calculated as negative in this case. Therefore we do not need to call `determineQuadrant` to ensure that positive lengths are always derived. The section of the Ambipan GUI code that creates $x$, $y$, and $z$ coordinates for the position of the sound source when the user is interfacing with the back-elevation sphere follows:
// calculate hypotenuse of triangle
hyp = sqrt(pow((pointerY-backY), 2) + pow((pointerX-backX), 2));

if (hyp > largeRadius) // if pointer *not* on surface of sphere
{
    switch (determineQuadrant(backX, backY))
    {
        case 1: angle=atan((backX-pointerX)/(pointerY-backY)); break;
        case 2: angle=atan((backY-pointerY)/(backX-pointerX))+(PI/2); break;
        case 3: angle=atan((pointerX-backX)/(backY-pointerY))+PI; break;
        case 4: angle=atan((pointerY-backY)/(pointerX-backX))+(1.5*PI); break;
    }
    sourceX = largeRadius*sin(-angle);
    sourceY = largeRadius*cos(-angle);
    sourceZ = 0.0;
}
else // if the mouse pointer *is* on the surface of the sphere
{
    // CALCULATE POSITION OF THE SOUND SOURCE RELATIVE TO THE
    // CENTRE OF THE SPHERE - x AND y ARE CALCULATED FROM MOUSE
    // POSITION, WHILE z IS CALCULATED FROM x AND y
    sourceX = pointerX-backX;
    sourceY = pointerY-backY;
    sourceZ = sqrt((largeRadius*largeRadius)-(sourceX*sourceX) - (sourceY*sourceY));
    if (rightButton) sourceZ = -sourceZ;
}

9.3.6 Placing the Sound-Source on each Sound-Field Elevation

The previous few paragraphs described how mouse interaction with any of the three sound-field elevation spheres is used to produce cartesian coordinates relative to the centre of the sound-field which describe the position of the sound-source on its surface. The small sphere that represents the sound source is then placed on the surface of each sound-field elevation sphere by translating to its centre, and then further translating to the position specified by the sound-source coordinates. In the case of the back-elevation, this process is all that is required,
because this elevation represents the sound-field viewed directly from behind. The other elevations, however, represent the sound-field viewed from two different angles of rotation. It is therefore necessary, before translating to the position of the sound source, to rotate the coordinate system by 270° about the x-axis in the case of the plan-elevation, and by 90° about the y-axis for the side-elevation. All of these actions are carried out in the OpenGL rendering callback function, which was declared with a call to glutDisplayFunc as the function DrawGLScene. The part of this function that draws the side-elevation sphere and positions the sound-source sphere at the correct position on its surface, for example, is as follows:

```c
/////////////////////////////////////////////////////////
/// POSITION THE SMALL SPHERE ON THE SIDE VIEW SPHERE ///
/////////////////////////////////////////////////////////
glLoadIdentity();
glTranslatef(sideX, sideY, sideZ); // move to centre of side sphere
glRotatef(90.0f, 0.0f, 10.0f, 0.0f); // rotate 90degrees around y-axis
glTranslatef(-sourceX, sourceY, sourceZ); // move to position of sound source
glColor4f(1.0, 0.0, 0.0, 1.0); // select red
glutSolidSphere(smallRadius, 10, 10); // create sound-source sphere

/////////////////////////////////////////////////////////
/// DRAW THE SIDE VIEW SPHERE ///
/////////////////////////////////////////////////////////
glLoadIdentity(); // reset modelview
glTranslatef(sideX, sideY, sideZ); // move to centre of sphere
glColor4f(0.0, 1.0, 0.0, 0.4); // select green
glutSolidSphere(largeRadius, 30, 30); // draw side-elevation sphere
```

Notice that the sound-source sphere is created before the sound-field sphere. This is because the Ambipan GUI allows for the possibility that the sound-source sphere may be positioned ‘behind’ the sound-source sphere. As we have enabled depth testing (with an initial call to glEnable(GL_DEPTH_TEST)) so if we create the large sphere first, and then create the small sphere at a position that is ‘behind’ the large (i.e. further down the positive z-axis) then the depth test will determine that the small sphere is behind the large, is therefore not visible, and so should not be drawn at all. We get around this by creating the small sphere first, then if we create the large sphere ‘in front’ of it, we can still see the small
sphere because of the transparency of the large sphere (recall that 
\[ \text{glColor4f}(0.0,1.0,0.0,0.4) \] selects the colour green with an alpha (transparency) value of 0.4).

### 9.3.7 Control of the Stereo Height and Stereo Width Sliders

Control of the stereo-height and stereo-width sliders is a much simpler process than that of the positioning of the sound source on the surface of the sound field. A control area slightly wider and taller than each slider is defined (again, using calls to determineControlArea) so that the program code can tell if the user is trying to adjust the stereo-width or stereo-height settings. In the case of the stereo-height slider, when the user clicks (with either mouse button) in the corresponding control area, the y mouse coordinate is mapped onto the position of the slider, which is stored in a variable named \text{heightPos} in the OpenGL drawing function. If the value of \text{heightPos} is such that the slider would be positioned outside of the maximum and minimum extremities, then it is reassigned to be equal to the maximum or minimum permitted setting. The code that calculates the desired position of the stereo-height slider, again contained in the mouse motion callback function \text{Motion}, is as follows:

```c
// //////////////////////////////////////////////////////////////////////////
//***************************************************************************
//*** CONTROLLING THE STEREO HEIGHT SLIDER ***
//***************************************************************************
// //////////////////////////////////////////////////////////////////////////
if (determineControlArea(43,200,107,400))
{
    heightPos = pointerY;

    if (heightPos > (300+slideLength-buttonSize))
    {
        heightPos = (300+slideLength-buttonSize);
    }

    if (heightPos < (300-slideLength+buttonSize))
```
The value of `heightPos` is then incorporated into the function `DrawGLScene` as follows:

```c
    /// PLACE BUTTON ON HEIGHT SLIDER ///
    glColor3ub(0,0,255); // blue
    glBegin(GL_QUADS); // draw rectangular button
    glVertex3f(75-slideWidth,heightPos+buttonSize,450.0f); // top-left
    glVertex3f(75-slideWidth,heightPos-buttonSize,450.0f); // bottom-left
    glVertex3f(75+slideWidth,heightPos-buttonSize,450.0f); // bottom-right
    glVertex3f(75+slideWidth,heightPos+buttonSize,450.0f); // top-right
```

Calculating the position of the stereo-width slider is carried out by two very similar pieces of code. The main differences are the control area defined, and that it is the x mouse coordinate that gets mapped onto the position of the slider button, because the stereo-width slider is horizontal rather than vertical.
10: Testing

The Ambipan and Ambidec plugins have been thoroughly tested using Steinberg’s Nuendo 1.5 as the host application. As mentioned earlier, at the time of writing this application represented the only VST host with the capability of outputting to more than two discreet sound-card outputs, therefore it has not been possible to perform tests using any other host applications. The purpose of these tests is to ensure that the Ambipan and Ambidec plugins are producing the expected output given any set of parameters. Due to unforeseen last-minute circumstances it has not been possible to provide tests for the stereo width and stereo height parameters. This chapter thoroughly documents the test procedure and results.

10.1 Calibrating Nuendo

10.1.1 Calibrating the Input Gain

Before any testing could take place it was important to see that Nuendo was calibrated properly. As the output from Nuendo’s VU-meters is to be used as the information on which the plugins’ correct functioning be evaluated, it is important to ensure that this information is not erroneous. Nuendo was calibrated using a single monophonic sine wave of 440 Hz frequency, with a peak amplitude of 15000 and a sampling rate of 48 kHz. When used as the input signal to a Nuendo VST channel, this sine wave resulted in a reading of −9.3 dB on the channel input VU-meter. As we know all the necessary characteristics of the source signal, we can calculate its true amplitude. The root mean square of a sine wave of peak amplitude (a) is given by the following equation:
RMS(a) = \sqrt{2} \times a

So for our sine wave, where a = 15000:

RMS(a) = \sqrt{2} \times 15000
      = 10606.6

We also know that we can calculate gain in dBu, using the following equation:

gain in dBu = 20\log_{10}()

As we are working in 16-bit precision, so the maximum possible amplitude value is 32767.
Substituting our values gives:

gain in dBu = 20\log_{10}()
           = 20\log_{10}(0.2289)
           = -9.8 \text{ dBu}

Compared to the VU-meter reading of –9.3 dB this indicates an offset of 0.5 dB. This was corrected by applying a gain of –0.48 dB (as the VU-meter displays are only accurate to one decimal place a little experimentation was necessary to fine-tune the adjustment) to the input channel on the mixing desk. The results are shown below.
The left-most panel is channel 1 of the VST mixing desk. Below the left-most VU-meter we can see the –0.48 dB adjustment made to the gain of the input signal, resulting in the expected input amplitude of –9.8 dB. The right-most pair of VU-meters indicate the levels being received by output bus 1, demonstrating that both left and right channels are receiving the same signal at a level of –9.8 dB. This confirms that audio is passing through Nuendo without any unwanted gain adjustments, and therefore that any gain adjustments observed in the following tests will have been performed by Ambipan and not by the host application.

### 10.1.2 Calibrating the Effect Return Gain

For reasons described earlier it was not possible at the time the tests were carried out to implement Ambipan as a channel insert effect, therefore it was necessary to carry out the tests using Ambipan as a send effect. For this reason it was necessary to calibrate the gain of the effect return, to ensure that the signal would not be amplified or attenuated before being
returned to the channel. With the default Ambipan settings (where azimuth, elevation, stereo height, and stereo width are all set to zero) the source signal was passed through the plugin. Using standard ambisonic encoding equations (see Chapter 3), the expected output results for the W, X, Y, and Z channels with azimuth and elevation values of 0° (as shown by the left and right channels of output buses 1 and 2, respectively) were calculated as follows:

\[
\begin{align*}
W &= 0.7071 \times 15000 \\
&= 10606.6 \\
RMS(W) &= \sqrt{0.5 \times 10606.6} \\
&= 7500 \\
&= -12.8 \text{ dB}
\end{align*}
\]

\[
\begin{align*}
X &= \cos(0) \times \cos(0) \times 15000 \\
&= 15000 \\
RMS(X) &= \sqrt{0.5 \times 15000} \\
&= 10606.6 \\
&= -9.8 \text{ dB}
\end{align*}
\]

\[
\begin{align*}
Y &= \sin(0) \times \cos(0) \times 15000 \\
&= 0 \\
&= -\infty \text{ dB}
\end{align*}
\]

\[
\begin{align*}
Z &= \sin(0) \\
&= 0 \\
&= -\infty \text{ dB}
\end{align*}
\]

Passing the source signal through Ambipan, the return gain was adjusted until the output bus VU-meters demonstrated the values given above. This occurred with the return gain slider in the centre position, as illustrated in the diagram below.
The tests which follow were carried out with the return gain set to this position. It can therefore be concluded that if the levels recorded on the VU-meters of the output buses correspond with the pre-calculated expected output levels for each channel, then Ambipan is performing ambisonic B-format encoding correctly.

10.2 Testing Ambipan

The function of the Ambipan plugin is to take a monophonic or stereophonic sound source and, based on the given parameters of azimuth and elevation (and stereo width and stereo height in the case of two-channel source sounds), position that source within an ambisonic sound-field, outputting four corresponding channels of B-format. As the positioning of stereophonic sources is simply an extension of the procedure used to position monophonic sources, the ability of Ambipan to position mono sound sources correctly was the first functionality to be assessed. Accordingly, the stereo width and stereo height controls were both set to zero throughout these tests.

The same monophonic sine wave described earlier was sent to channel 1 of the VST mixing desk, then passed through Ambipan with various different azimuth and elevation settings. The amplitudes shown on the output bus VU-meters were then recorded and compared with the pre-calculated expected results.
Test 1: Azimuth = 90°, Elevation = 0°

Calculated values of W, X, Y, and Z:

- W = -12.8 dB
- X = \( \cos(90) \times \cos(0) \times 15000 \)
  = 0
  = -\infty dB
- Y = \( \sin(90) \times \cos(0) \times 15000 \)
  = 15000
  \( \text{RMS}(Y) = 10606.6 \)
  = -9.8 dB
- Z = \( \sin(0) \times 15000 \)
  = 0
  = -\infty dB

Observed results – W = bus 1L; X = bus 1R; Y = bus 2L; Z = bus 2R:
Test 2: Azimuth = 225°, Elevation = 0°

Calculated values of W, X, Y, and Z:

\[
\begin{align*}
W &= -12.8 \text{ dB} \\
X &= \cos(225) \times \cos(0) \times 15000 \\
&= -10606.6 \\
\text{RMS}(X) &= -7500 \\
&= -12.8 \text{ dB} \\
Y &= \sin(225) \times \cos(0) \times 15000 \\
&= -10606.6 \\
\text{RMS}(Y) &= -7500 \\
&= -12.8 \text{ dB} \\
Z &= \sin(0) \times 15000 \\
&= 0 \\
&= -\infty \text{ dB}
\end{align*}
\]

Observed results – W = bus 1L; X = bus 1R; Y = bus 2L; Z = bus 2R:
Test 3: Azimuth = 0°, Elevation = 90°

Calculated values of W, X, Y, and Z:

\[
W = -12.8 \text{ dB}
\]
\[
X = \cos(0) \times \cos(90) \times 15000
= 0
= -\infty \text{ dB}
\]
\[
Y = \sin(0) \times \cos(90) \times 15000
= 0
= -\infty \text{ dB}
\]
\[
Z = \sin(90) \times 15000
= 15000
\text{RMS}(Z) = 10606.6
= -9.8 \text{ dB}
\]

Observed results – W = bus 1L; X = bus 1R; Y = bus 2L; Z = bus 2R:
Test 4: Azimuth = 0°, Elevation = 135°

Calculated values of W, X, Y, and Z:

\[
\begin{align*}
W & = -12.8 \text{ dB} \\
X & = \cos(0) \cdot \cos(135) \cdot 15000 \\
& = -10606.6 \\
\text{RMS}(X) & = -7500 \\
& = -12.8 \text{ dB} \\
Y & = \sin(0) \cdot \cos(135) \cdot 15000 \\
& = 0 \\
& = -\infty \text{ dB} \\
Z & = \sin(135) \cdot 15000 \\
& = 10606.6 \\
\text{RMS}(Z) & = 7500 \\
& = -12.8 \text{ dB}
\end{align*}
\]

Observed results – W = bus 1L; X = bus 1R; Y = bus 2L; Z = bus 2R:
Test 5: Azimuth = 217°, Elevation = 311°

Calculated values of W, X, Y, and Z:

\[
W = -12.8 \text{ dB}
\]
\[
X = \cos(217) \times \cos(311) \times 15000 = -7859.285
\]
\[
\text{RMS}(X) = -5557.354 = -15.4 \text{ dB}
\]
\[
Y = \sin(217) \times \cos(311) \times 15000 = -5922.392
\]
\[
\text{RMS}(Y) = -4187.764 = -17.9 \text{ dB}
\]
\[
Z = \sin(311) \times 15000 = -11320.644
\]
\[
\text{RMS}(Z) = -8004.904 = -12.2 \text{ dB}
\]

Observed results – W = bus 1L; X = bus 1R; Y = bus 2L; Z = bus 2R:
**Test 6: Azimuth = 128°, Elevation = 37°**

Calculated values of W, X, Y, and Z:

- **W** = -12.8 dB

  \[
  X = \cos(128) \cdot \cos(37) \cdot 15000 \\
  = 7375.311
  \\
  \text{RMS}(X) = 5215.132 \\
  = -16.0 \text{ dB}
  \]

- **Y** = sin(128) * cos(37) * 15000 \\
  = 9440.006 \\
  \text{RMS}(Y) = 6675.092 \\
  = -13.8 \text{ dB}

- **Z** = sin(37) * 15000 \\
  = 9027.225 \\
  \text{RMS}(Z) = 6383.212 \\
  = -14.2 \text{ dB}

Observed results – W = bus 1L; X = bus 1R; Y = bus 2L; Z = bus 2R:
10.2.1 Summary of Tests 1 – 6

In these six tests there were a total of two discrepancies between projected results and observed results. These were in test 5 – where Z was calculated as $-12.2$ dB but the observed result was $-12.3$ dB – and in test 6, where X was calculated to be $-13.8$ dB but observed to be $-13.9$ dB. Apart from being very small (and almost unquestionably insignificant) errors, these can be accounted for by two factors. Firstly, the calibration of the Ambipan effect return gain had to be carried out using a VU-meter with precision to only one decimal place, as described earlier. This results in a potential error of $\pm 0.05$ dB right at the beginning of the signal chain. Secondly, the number of calculations involved in solving the encoding equations, calculating RMS values, converting to dB, and so on, results in further loss of precision, introducing a potential error into the projected output values themselves. From the results of tests 1 through 6 it can be concluded that Ambipan is outputting the correct B-format signals based on the values assigned by the user to the parameters of azimuth and elevation.

10.3 Testing Ambidec

Having ascertained that Ambipan is functioning correctly and as expected, a further series of tests were conducted to evaluate the accuracy of the decoding carried out by the Ambidec plugin. Using Nuendo, a plugin automation track was created to pan the same sine wave used in tests 1 to 6 from 0° azimuth to 360° azimuth over a period of ten seconds, with the elevation set to 0°. Audio was exported for each of the decoding options using this automation track. Similarly, a second automation track was created to pan the sine wave from 0° elevation to 360° elevation, with the azimuth set to 0°. Again, audio was exported for each of the decoding options. The following time-domain analyses of the audio exported
demonstrate that Ambidec is decoding to each of the speaker rigs correctly, and as expected. Numbers next to each speaker feed time-domain analysis correspond with the speaker numbers for each rig, as enumerated in Chapter 6. It can be seen from these analyses that the azimuthal and elevational rotations each pan the signal smoothly between the expected speakers. Note that in the analyses of the 3/2 decode, the rear speakers exhibit higher amplitude level that the front speakers. This is to compensate for the asymmetry of the rig (Furse 2000). Note also that the increasingly ‘stepped’ appearance of the analyses as the number of speaker feed signals increases is due to magnification carried out in the analysis software, and not because the amplitudes of the signals themselves were actually stepped. That amplitude is smoothly adjusted for all decoding options will become very clear when the system is auditioned.

**Test 7: Mono Decode**

It was found, as expected, that neither the azimuth nor the elevation setting had any influence on the amplitude of the monophonic output. Time-domain analyses have not been included as they would illustrate nothing more than a sine wave of constant amplitude from beginning to end.

**Test 8: Elevation Increased Linearly from 0° to 360° over 10 Seconds – Stereo Decode**

Again, as expected, the state of the elevation control was found to have no effect on the two speaker feed outputs of a stereo decode. Time-domain analyses have not been included for reasons explained in Test 7.
Test 9: Azimuth Increased Linearly from 0° to 360° over 10 Seconds – Stereo Decode

Test 10: Azimuth Increased Linearly from 0° to 360° over 10 Seconds – Square Decode
Test 11: Azimuth Increased Linearly from 0° to 360° over 10 Seconds – 3/2 Decode

Test 12: Azimuth Increased Linearly from 0° to 360° over 10 Seconds – Cube Decode
Test 13: Elevation Increased Linearly from 0° to 360° over 10 Seconds – Square Decode

Test 14: Elevation Increased Linearly from 0° to 360° over 10 Seconds – 3/2 Decode
10.4 Testing the Ambipan GUI

As integration with the Ambipan plugin has not yet been achieved, so the most important functionality of the Ambipan GUI is that when the sound source is moved to a particular position on one sound-field elevation, the sound source is correctly represented on the other two sound-field elevations, as well as on the central (non-controllable) elevation. This is best tested by moving the sound source to a position which will be easily recognisable on all elevations of the sound-field, and verifying visually that the sound source is in the correct place in each elevation. The following tests confirm that the Ambipan GUI is correctly positioning the sound source on all four sound-field elevations based on user input from any of the three controllable elevations.
**Test 16: 0° Azimuth, 0° Elevation**

This test illustrates the front of the sound-field and can be used as a reference point to verify the correct positionings in the subsequent tests. The final interface will indicate the front of the sound-field in each elevation, for clarity. In the diagrams that follow if the sound source (represented by the smaller sphere) is ‘behind’ the sound-field sphere, it appears to be a slightly lighter shade of grey than if it is ‘in front’. The front/back positioning is visually much more obvious in full colour, and when using the real interface with a moving sound source.
**Test 17: 90° Azimuth, 0° Elevation**

![Diagram of Test 17](image1)

**Test 18: 180° Azimuth, 0° Elevation**

![Diagram of Test 18](image2)
**Test 19: 270° Azimuth, 0° Elevation**

**Test 20: 0° Azimuth, 90° Elevation**
Test 21: 0° Azimuth, 180° Elevation

Test 22: 0° Azimuth, 270° Elevation
11: Conclusions and Future Work

The purpose of this project has not necessarily been to reach any kind of conclusion, but rather to engage in the development of a suite of software implementations with a view to providing functionality that does not exist in previous sound-spatialisation software. Specifically, the main aims of the project have been to produce a VST plugin implementations of ambisonic panning and decoding utilities, and to look into the possibility of developing more intuitive interfaces for controlling such software.

The relatively recent possibility of real-time audio processing has brought with it a new set of interfacing challenges, because parameters must be expressed at the same time as they are acted on. We are therefore to a much greater extent affected by the limitations imposed upon us by the interface, whether physical or graphical. Most commonly-used GUI APIs do not offer interfacing paradigms suitable for the real-time manipulation of objects in three-dimensional space, but rather, are designed for creating GUIs for more day-to-day applications which do not require such real-time capabilities. For this reason it has been necessary to develop a custom GUI specifically aimed at the task of positioning sounds in space. The Ambipan GUI is a first attempt to satisfy this criterion, and a successful one, insofar as the user can place the sound source anywhere on the ‘surface’ of the sound-field using any of three different elevations thereof. Such use of three elevations provides the user both with multiple interfaces for realising different trajectories, and greater means of visualising the position of the sound in space.

It is clear that, however much has been achieved so far, the Ambipan and Ambidec plugins, and the user interface for Ambipan, are works in progress. The author hopes to continue their development as part of the programme of study for a PhD, therefore a ‘wish-list’ of improvements and refinements – some merely desirable, some absolutely necessary –
has been drawn up. Some proposed future developments are outlined, roughly in order of priority in the following paragraphs.

### 11.1 Implementation of Ambipan as a Spatialiser Plug-In with GUI

At present the implementation of the Ambipan plugin is such that, in Nuendo, it may only be installed as a channel send effect. In this state the plugin functions correctly if installed on a single VST mixing desk channel, but because its `process` function overwrites the plugin output buffers (the only way to avoid the original input signal causing erroneous B-format output), the channel on which Ambipan is installed will overwrite any other VST channels that may exist. Obviously this situation is unsatisfactory, and therefore one of the most pressing aspects of the future development of Ambipan will be its implementation as an insert effect.

Presently, Nuendo will not accept Ambipan as a ‘normal’ channel insert effect because it has more than the permitted maximum of two audio output streams. However, Nuendo, under the new VST 2.0 specification, does offer support for multi-channel-output insert effects if plugins are declared to be of the ‘spatialiser’ category. This can be done by overriding the following `audioeffectx` function:

```cpp
virtual VstPlugCategory getPlugCategory() {if (cEffect.flags & effFlagIsSynth) return kPlugCategSynth; return kPlugCategUnknown;}
```

As we can see, by default (as per the inline definition) the function returns the ‘unknown’ category, unless the plugin in question has set the `effFlagIsSynth` flag via a call to the `audioeffectx` function `isSynth()`. If we override this function to return the enumerated `VstPlugCategory` named `kPlugCategSpatializer`, then Nuendo will
install the plugin in the panner section of the corresponding VST channel, which allows insert effects to have more than two audio outputs.

However, this override has not yet been implemented. This is due to the fact that in Nuendo, plugins of the kPlugCategSpacializer category do not have access to Nuendo’s default method for displaying plugin parameters. Enquiries as to the reasons behind this have proved unsuccessful. Evidently, the only way to interface properly with a spatialiser plugin is for the plugin to define its own custom GUI. As the Ambipan GUI is not yet at a sufficiently advanced state of development to be integrated with the plugin, so implementation as a spatialiser has thus far been impossible. It is for these reasons that the implementation of Ambipan as a channel send effect, although not ideal, has been the only direction in which work could progress thus far.

11.2 More Creative use of the Preset Programs in Ambipan

Ambipan allows the user to store up to sixteen sets parameters as programs. At present the user can select a pre-set program, and the parameter values will instantaneously change. This could be useful if we want to return a sound source to the same position repeatedly, for example. This functionality could be further developed by the addition of an adjustable timer, which would determine the time taken to interpolate between the parameter settings of two different programs. Such a feature exists in Steinberg’s GRM Tools plugin suite. The possibility of non-linear interpolation could also be considered, allowing sound sources to accelerate or decelerate from one preset position to another. This feature was suggested by Malham (2001) in the original specification for the BF package.
11.3 More Flexible Decoding Options

Ambidec allows the user to decode to monophonic, stereophonic, quadraphonic, 3/2, and cube speaker rigs. This covers a large portion of the speaker setups which most people are likely to have. Decoding to rigs containing more than eight speakers is not, at present, a possibility, because Nuendo allows a maximum of only eight discreet channels on the master section of the VST mixing desk. However, more flexible decoding should be considered as a future possibility. For example, the implementation of a slider which interpolates between a mathematically strict idealised directional response, and an ‘in-phase’ response (Furse 2000) would be useful for optimising decoding for the size of listening area required. A single listener could use the strict response, while the ‘in-phase’ response would be more appropriate for an audience spread over a larger listening area.

User-definable sound-field manipulations such as rotation, dominance, and so on (explained in Chapter 3) could also be carried out at the decoding stage. This would obviously introduce quite different interfacing challenges and it is for this reason that these functionalities have been avoided thus far.

Additionally, psychoacoustic optimisation has been little discussed in this report, and the decoding carried out by Ambidec does not incorporate any. The additional option of psychoacoustic optimisation could result in a marked improvement in sound-field stability if carefully implemented.

11.4 Collation of all Spatialised Sources in a Single Display Module

With an Ambipan plugin installed on each channel of the VST mixing desk, each GUI (when fully integrated with the plugin itself) will provide visual information as to the position of the audio on that particular channel. For this reason, there will be no way in which two sound
sources, from two different VST channels, can be visualised on the same interface. This would, however, be a very useful feature, and has been implemented in SpinAudio’s (2001a) 3D Panner Studio, which is also a VST plugin. Therefore it must be possible, somehow, to collate information from a number of plugin objects (assuming they are all instances of the same plugin, of course). Research into how this is achieved should be considered as a future development.

11.5 Implementation of Distancing Laws

This would obviously be the shared responsibility of the Ambipan and Ambidec plugins. At present, Ambipan only allows the user to position a sound source on the surface of the sphere that metaphorically represents the sound-field. The main reason behind this is that, as soon as distance from the listener is to be implied, changes to the frequency content of sound sources must be imposed. Further, our perception of how distant a sound source is, is to a considerable extent dependent on the characteristics of the listening environment, as described in Chapter 1. Therefore any algorithm designed to simulate sound source distance would either have to make sweeping assumptions about the acoustics of the listening environment, or allow the user to configure these characteristics. Like psychoacoustic optimisation, this is a process that involves changing the spectral content of the source material, something that developers should implement with care, especially when dealing with applications designed for use by composers. Measures must be taken to ensure that no frequency-domain alterations should be imposed if this is not the composer’s intention. For this reason it is stressed that any future developments which perform such transformations of source material, such as distancing laws and psychoacoustic optimisation, should be fully bypassable by the user. This was thought to be too large an undertaking for a first implementation, but remains a consideration for future developments.
11.6 Second Order Support

Ambipan takes a monophonic or stereophonic sound source and outputs four channels of first-order ambisonic B-format. Similarly, Ambidec receives these first-order signals, and creates speaker feeds for the chosen speaker rig. Implementation of higher-order support has not been considered to date. Indeed, it is not known if Nuendo can deal with plugins with as many as nine outputs, the number of channels necessary to carry second-order B-format. The maximum number of channels permitted in the master section of the Nuendo VST mixing desk is eight, which perhaps suggests that more-than-eight channel support is not part of the Nuendo 1.5 specification. However, should this change, second- and higher-order support should certainly be a consideration.
References and Bibliography


Appendix I – Ambipan Source Code

////////////////////////////////////
///// AMBIPAN.HPP /////
////////////////////////////////////

///////////////////////////////////////////////////////////
///// VST PLUGINS FOR AMBISONIC ENCODING AND DECODING /////
///// BY JAMES MOONEY, 2001 /////
///////////////////////////////////////////////////////////

//------------------------------
VST Plug-In Technology by Steinberg
VST is a trademark of Steinberg Soft und Hardware GmbH

------------------------------------------------------------
VST Plug-In Technology by Steinberg
VST is a trademark of Steinberg Soft und Hardware GmbH
------------------------------------------------------------

#include "audioeffectx.h"
#include <string.h>

#define TO_RADIANS 6.2831853 // for converting parameters to degrees in radians

// enumerate parameters for easy indexing
enum {
    kAzimuth,
    kElevation,
    kStereoWidth,
    kStereoHeight,
    kNumParams
};

class Ambipan;

// define class for program (preset) objects
class AmbipanProgram
{
    friend class Ambipan;
    public:
    AmbipanProgram();
    ~AmbipanProgram();

    private:
    float fAzimuth, fElevation, fStereoWidth, fStereoHeight;
    char name[24];

    // define plugin class itself
    class Ambipan : public AudioEffectX
    {
    public:
        Ambipan(audioMasterCallback audioMaster);
        ~Ambipan();

        virtual void process(float **inputs, float **outputs, long sampleframes);
        virtual void processReplacing(float **inputs, float **outputs, long sampleFrames);
        virtual void setProgram(long program);
        virtual void setProgramName(char *name);
        virtual void getProgramName(char *name);
        virtual void setParameter(long index, float value);
        virtual float getParameter(long index);
        virtual void getParameterLabel(long index, char *label);
        virtual void getParameterDisplay(long index, char *text);
        virtual void getParameterName(long index, char *text);
    }
private:
AmbipanProgram *programs;
float fAzimuth, fElevation, fStereoWidth, fStereoHeight, // parameters
oldAzimuth, oldElevation, oldWidth, oldHeight, // for interpolation
newAzimuth, newElevation, newWidth, newHeight, // for interpolation factor: 0.999 seems
i; // to be good to avoid clicking

AmbipanProgram::AmbipanProgram ()
{
    fAzimuth = 0.0;
fElevation = 0.0;
fStereoWidth = 0.0;
fStereoHeight = 0.0;
strcpy (name, "Init");
}

Ambipan::Ambipan (audioMasterCallback audioMaster)
    : AudioEffectX (audioMaster, 16, kNumParams) // 16 programs (4 parameters)
{
    programs = new AmbipanProgram[numPrograms];
/// INITIALISE PARAMETERS TO 0 ON CONSTRUCTION OF Ambipan OBJECT ///
fAzimuth = fElevation = fStereoWidth = fStereoHeight = oldAzimuth = oldElevation = oldWidth = oldHeight = 0;

i = 0.999;  // interpolation factor for parameters - to stop clicking: 0.999 is good

if (programs) // set default program to 0
    setProgram (0);

setNumInputs (2);  // 2 audio input buffers
setNumOutputs (4);  // 4 audio output buffers - for B-format channels

canProcessReplacing ();
canMono();  // canMono() means the plugin CAN be used as a send effect
setUniqueID ('HZST');  // = horizontal stereo compatible

Ambipan::~Ambipan ()
{
    if (programs) // deallocate memory set aside for program objects
        delete[] programs;
}

void Ambipan::setProgram (long program)
{
    AmbipanProgram * ap = &programs[program];  // create pointer to selected program

    curProgram = program;  // set current program variable

    // change parameters to those stored in the selected program //
    setParameter (kAzimuth, ap->fAzimuth);
    setParameter (kElevation, ap->fElevation);
    setParameter (kStereoWidth, ap->fStereoWidth);
    setParameter (kStereoHeight, ap->fStereoHeight);
}

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

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

void Ambipan::setParameter (long index, float value)
{
    AmbipanProgram * ap = &programs[curProgram];

    switch (index)
    {
    // change current parameter and also the copy of
    // the parameter stored in the current program
    case kAzimuth :
        fAzimuth = ap->fAzimuth = value;
        newAzimuth = fAzimuth * TO_RADIANS;break;

    case kElevation :
        fElevation = ap->fElevation = value;
        newElevation = fElevation * TO_RADIANS; break;
    }
case kStereoWidth : fStereoWidth = ap->fStereoWidth = value; newWidth = fStereoWidth * (TO_RADIANS/4); break; // so maximum stereo separation is 180 degrees

case kStereoHeight: fStereoHeight = ap->fStereoHeight = value; newHeight = fStereoHeight * (TO_RADIANS/4); break;

if (editor) // for future use - update parameters of GUI also
    editor->postUpdate();

float Ambipan::getParameter (long index)
{
    float v = 0;
    switch (index)
    {
        case kAzimuth : v = fAzimuth; break;
        case kElevation : v = fElevation; break;
        case kStereoWidth : v = fStereoWidth; break;
        case kStereoHeight: v = fStereoHeight; break;
    }
    return v; // return requested parameter to the host application
}

void Ambipan::getParameterName (long index, char *label)
{
    switch (index)
    {
        case kAzimuth : strcpy (label, "Azimuth "); break;
        case kElevation : strcpy (label, "Elevation"); break;
        case kStereoWidth : strcpy (label, " Width "); break;
        case kStereoHeight: strcpy (label, " Height "); break;
    }
}

void Ambipan::getParameterDisplay (long index, char *text)
{
    switch (index)
    {
        // convert az & ele parameters to values between 0 and 360
        case kAzimuth : float2string ((fAzimuth*360), text); break;
        case kElevation : float2string ((fElevation*360), text); break;

        // convert width & height parameters to values between 0 and 180
        case kStereoWidth : float2string ((fStereoWidth*180), text); break;
        case kStereoHeight: float2string ((fStereoHeight*180), text); break;
    }
}

void Ambipan::getParameterLabel (long index, char *label)
{
    strcpy (label, "degrees "); // because *all* parameters are in degrees
}

void Ambipan::process (float **inputs, float **outputs, long sampleframes)
{
    // create pointers to input and output buffers
    float *in1 = inputs[0];
    float *in2 = inputs[1];
    float *out1 = outputs[0];
    float *out2 = outputs[1];
    float *out3 = outputs[2];
    float *out4 = outputs[3];
double w, x, y, z, w1, x1, y1, z1, w2, x2, y2, z2,
    azimuth,
    elevation,
    width,
    height;

while(--sampleframes >= 0) {
    // interpolate between parameter values to prevent clicking
    azimuth = (i*oldAzimuth) + ((1-i)*newAzimuth);
    elevation = (i*oldElevation) + ((1-i)*newElevation);
    width = (i*oldWidth) + ((1-i)*newWidth);
    height = (i*oldHeight) + ((1-i)*newHeight);

    // B-format encoding for 'left' channel of stereo input file
    w1 = *in1 * 0.7071;
    x1 = *in1 * cos(azimuth + width) * cos(elevation + height);
    y1 = *in1 * sin(azimuth + width) * cos(elevation + height);
    z1 = *in1 * sin(elevation + height);

    // B-format encoding for 'right' channel of stereo input file
    w2 = *in2 * 0.7071;
    x2 = *in2 * cos(azimuth - width) * cos(elevation - height);
    y2 = *in2 * sin(azimuth - width) * cos(elevation - height);
    z2 = *in2 * sin(elevation - height);

    // Add 'L' and 'R' B-formats together to create a single sound-field
    w = (w1 + w2)/2;
    x = (x1 + x2)/2;
    y = (y1 + y2)/2;
    z = (z1 + z2)/2;

    // output B-format to output buffers for external decoding
    (*out1++) = w; // note: even in process(), replacing process is
    (*out2++) = x; // forced there is no accumulation as it wouldn't
    (*out3++) = y; // make sense.
    (*out4++) = z;

    // increment input pointers
    *in1++;
    *in2++;

    // update oldX values for interpolation process
    oldAzimuth = azimuth;
    oldElevation = elevation;
    oldWidth = width;
    oldHeight = height;
}

// void Ambipan::processReplacing (float **inputs, float **outputs, long sampleframes)
// nothing to do here at the moment
{}
Appendix II – Ambidec Source Code

///////////////////////////////////////////////////////////////////////////////////////////////////
///// AMBIDECE.HPP /////
///////////////////////////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////////////////////////
///// VST PLUGINS FOR AMBISONIC ENCODING AND DECODING /////
///// BY JAMES MOONEY, 2001 /////
///////////////////////////////////////////////////////////////////////////////////////////////////

/*----------------------------------------------
VST Plug-In Technology by Steinberg
VST is a trademark of Steinberg Soft und Hardware GmbH
----------------------------------------------*/

#include "audioeffectx.h"
#include <string.h>

class Ambidec: public AudioEffectX
{
public:
    Ambidec(audioMasterCallback audioMaster);
    ~Ambidec();
    virtual void process(float **inputs, float **outputs, long sampleframes);
    virtual void processReplacing(float **inputs, float **outputs, long sampleFrames);
    virtual void setProgramName(char *name);
    virtual void getProgramName(char *name);
    virtual void setParameter(long index, float value);
    virtual float getParameter(long index);
    virtual void getParameterLabel(long index, char *label);
    virtual void getParameterDisplay(long index, char *text);
    virtual void getParameterName(long index, char *text);

private:
    float fSpeakerRig;
    int speakerRig;
    char programName[32];
};
VST Plug-In Technology by Steinberg
VST is a trademark of Steinberg Soft und Hardware GmbH

#include <AEffect.h>
#include <aeffectx.h>
#include <AEffEditor.hpp>
#include <AudioEffect.hpp>
#include <audioeffectx.h>
#include <AudioEffect.cpp>
#include <audioeffectx.cpp>
#include "AEffEditor.hpp"
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "Ambidec.hpp"

#define MONO 0
#define STEREO 1
#define SQUARE 2
#define THREE_TWO 3
#define CUBE 4

Ambidec::Ambidec (audioMasterCallback audioMaster)
: AudioEffectX (audioMaster, 1, 1) // 1 program and 1 parameter
{
    fSpeakerRig = 0.3f; // so default decode option is stereo (see setParameter)
speakerRig = STEREO;

    setNumInputs (4); // four inputs for B-format channels
    setNumOutputs (8); // eight outputs for cube decode
    canProcessReplacing ();
    setUniqueID ('AMDC'); // ambisonic decoder
    strcpy(programName, "Default"); // name the program default
}

Ambidec::~Ambidec ()
{} // nothing to do here as no program classes are created

void Ambidec::setProgramName (char *name)
{
    strcpy (programName, name);
}

void Ambidec::getProgramName (char *name)
{
    strcpy (name, programName);
}
void Ambidec::setParameter (long index, float value)
{
    fSpeakerRig = value;

    // set speakerRig flag for parameter display and
    // to use as switch index in processReplacing()
    if (fSpeakerRig > 0.8) speakerRig = CUBE;
    else if (fSpeakerRig > 0.6) speakerRig = THREE_TWO;
    else if (fSpeakerRig > 0.4) speakerRig = SQUARE;
    else if (fSpeakerRig > 0.2) speakerRig = STEREO;
    else speakerRig = MONO;
}

float Ambidec::getParameter (long index)
{
    return fSpeakerRig;
}

void Ambidec::getParameterName (long index, char *label)
{
    strcpy (label, "Speakers");
}

void Ambidec::getParameterDisplay (long index, char *text)
{
    switch (speakerRig)
    {
    case MONO : strcpy(text, "mono"); break;
    case STEREO : strcpy(text, "stereo"); break;
    case SQUARE : strcpy(text, "square"); break;
    case THREE_TWO : strcpy(text, "3/2"); break;
    case CUBE : strcpy(text, "cube"); break;
    }
}

void Ambidec::getParameterLabel (long index, char *label)
{
    strcpy (label, " rig ");
}

void Ambidec::process (float **inputs, float **outputs, long sampleframes)
{} // nothing to do here as plugin should always be used as a master insert

void Ambidec::processReplacing (float **inputs, float **outputs, long sampleframes)
{
    // declare pointers for input and output buffers
    float *in1 = inputs[0];
    float *in2 = inputs[1];
    float *in3 = inputs[2];
    float *in4 = inputs[3];
    float *out1 = outputs[0];
    float *out2 = outputs[1];
    float *out3 = outputs[2];
    float *out4 = outputs[3];
    float *out5 = outputs[4];
    float *out6 = outputs[5];
    float *out7 = outputs[6];
    float *out8 = outputs[7];
double w, x, y, z;
while(--sampleframes >= 0)
{
    w = *in1; // store B-format input into corresponding variables
    x = *in2;
    y = *in3;
    z = *in4;

    switch (speakerRig) { // switch between decode options
        case MONO :
            // decode to single speaker
            (*out1++) = (1.4142 * w);
            break;

        case STEREO :
            // decode to stereo pair
            (*out1++) = (0.7071*w) + (0.5 * y); // left
            (*out2++) = (0.7071*w) - (0.5 * y); // right
            break;

        case SQUARE :
            // decode to four-speaker square (horizontal only)
            (*out1++) = (0.3536*w) + (0.1768 * (x + y)); // front-left
            (*out2++) = (0.3536*w) + (0.1768 * (x - y)); // front-right
            (*out3++) = (0.3536*w) + (0.1768 * (-x + y)); // rear-left
            (*out4++) = (0.3536*w) + (0.1768 * (-x - y)); // rear-right
            break;

        case THREE_TWO :
            // decode to five-speaker 5.1-style rig (horizontal only)
            (*out1++) = (0.169 * w) + (0.1518 * x) + (0.1696 * y); // front-left
            (*out2++) = (0.169 * w) + (0.1757 * x) - (0.1696 * y); // front-right
            (*out3++) = (0.4563 * w) - (0.2396 * x) + (0.2937 * y); // rear-left
            (*out4++) = (0.4563 * w) - (0.2396 * x) - (0.2937 * y); // rear-right
            (*out5++) = (0.1635 * w) + (0.1757 * x); // centre;
            break;

        case CUBE:
            // decode to cube rig (periphonic)
            (*out1++) = (0.1768*w)+(0.0722*x)+(0.0722*y)-(0.0722*z); // fLd
            (*out2++) = (0.1768*w)+(0.0722*x)-(0.0722*y)+(0.0722*z); // fRd
            (*out3++) = (0.1768*w)-(0.0722*x)+(0.0722*y)-(0.0722*z); // rLd
            (*out4++) = (0.1768*w)-(0.0722*x)-(0.0722*y)+(0.0722*z); // rRd
            (*out5++) = (0.1768*w)+(0.0722*x)+(0.0722*y)+(0.0722*z); // fLu
            (*out6++) = (0.1768*w)+(0.0722*x)-(0.0722*y)+(0.0722*z); // fRu
            (*out7++) = (0.1768*w)-(0.0722*x)+(0.0722*y)+(0.0722*z); // rLu
            (*out8++) = (0.1768*w)-(0.0722*x)-(0.0722*y)+(0.0722*z); // rRu
            break;

        default : // this should never really be needed!
            break;
    }
}
/ increment input pointers
 *in1++;
 *in2++;
 *in3++;
 *in4++;
Appendix III – Ambipan GUI source code

////////////////////////////////////////////////
// Graphical Interface for Ambipan using OpenGL and GLUT //
// by James Mooney //
////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h> // needed for exit() prototype [see normalKey() function]
#include <math.h>
#include <GL/glut.h> // GLUT header also includes necessary GL headers

#define ESCAPE 27 // 27 is the ASCII code for the <escape> key
#define PI 3.14159

GLint window; // the identifier of our GLUT window

///// SET MAXIMUM COORDINATE VALUES FOR X Y AND Z AXES /////
///// THESE ARE USED FOR BOTH THE WINDOW SIZE IN PIXELS
///// AND THE OPENGL COORDINATE SYSTEM
GLint Xsize=450;
GLint Ysize=474; // to allow room for text above the plan view
GLint nearPlane=-450; // -ve because OpenGL assumes Z axis is -ve

///// INITIALISE MOUSE-BUTTON AND WINDOW-ENTRY FLAGS /////
GLint leftButton = 0;
GLint rightButton = 0;
GLint pointerInWindow = 0;

///// INITIALISE VARIABLES WHERE MOUSE POINTER COORDS WILL BE STORED /////
GLint pointerX = 0;
GLint pointerY = 0;

GLint slideLength = 75; // *half* of the slider length
GLint slideWidth = 12; // *half* of the slider height
GLint buttonSize = 6; // *half* of the button size

GLfloat heightPos = 231; // for positioning buttons on the height and width sliders
GLfloat widthPos = 231;

///// DEFINE RADIUS AND CENTRE POSITION VALUES FOR ALL SPHERES /////
GLfloat largeRadius = 50.0f; // radius of large spheres (sound-fields)
GLfloat smallRadius = 6.0f; // radius of small spheres (sound sources)

GLfloat centreX = 225; // centre sphere position in centre of window
GLfloat centreY = 225; // this sphere is for visualisation only!
GLfloat centreZ = 225;

GLfloat backX = 75; // position of back view sphere
GLfloat backY = 75;
GLfloat backZ = 225;

GLfloat sideX = 375; // position of side view sphere
GLfloat sideY = 225;
GLfloat sideZ = 225;

GLfloat planX = 225; // position of plan view sphere
GLfloat planY = 375;
GLfloat planZ = 225;

GLfloat sourceX = 0; // for the position of the sound source
GLfloat sourceY = 0; // - initial position is centre-front therefore.
GLfloat sourceZ = 50;

///// THESE ARE THE PARAMETERS THAT WILL BE SENT BACK TO THE PLUGIN /////
GLfloat azimuth = 0;
GLfloat elevation = 0;
GLfloat width = 0;
GLfloat height = 0;

/*/-----------------------------------*/

------ FOR FUTURE USE.

GLfloat azimuth = 0;
GLfloat elevation = 0;
GLfloat width = 0;
GLfloat height = 0;

/*------------------------------------------------*/

){// INITIALISE OPENGL - ALL THE STUFF THAT ONLY NEEDS TO BE DONE ONCE }/

GLvoid InitGL(GLfloat Width, GLfloat Height)
{
    glClearColor(0.0, 0.0, 0.0, 0.0); // Set clear color to black
    glShadeModel(GL_SMOOTH); // smooth shading between colours, please!
    glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );
    glEnable(GL_DEPTH_TEST); // Enable depth testing, so objects 'behind' other
    // objects are hidden. Otherwise the visibility of
    // objects depends only on the order in which they
    // are rendered.
    glEnable(GL_BLEND); // Enable transparency.
    glLineWidth(1.0); // set the line width
    // calibrate coordinate system to be the same as
    // the screen coord system
    glViewport(0, 0, Width, Height); // Set the viewport
    glMatrixMode(GL_PROJECTION); // Select the projection matrix
    glLoadIdentity(); // Reset The Projection Matrix
    glOrtho(0, Width, 0, Height, nearPlane, 0); // calibrate the coordinate system
    glMatrixMode(GL_MODELVIEW); // switch back to the modelview matrix
}

/*------------------------------------------------*/

){// THIS FUNCTION IS CALLED TO DRAW TEXT INTO THE SCREEN }/

void DrawText(GLint x, GLint y, char* s, GLfloat r, GLfloat g, GLfloat b)
{
    int lines;
    char* p;
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    glOrtho(0.0, glutGet(GLUT_WINDOW_WIDTH),
            0.0, glutGet(GLUT_WINDOW_HEIGHT), -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glColor3f(r,g,b);
    glRasterPos2i(x, y);
    for(p = s, lines = 0; *p; p++) {
        if (*p == '\n') {
            lines++;
            glRasterPos2i(x, y-(lines*12));
        }
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, *p); 
    }
    glPopMatrix();
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
}

/*------------------------------------------------*/

}
/***** THIS IS CALLED BY Mouse() TO DETERMINE WHICH ******/
/***** AREA OF THE SCREEN THE USER IS CONTROLLING. ******/
/***** THE VALUES PASSED ARE THE COORDINATE PAIRS OF******/
/***** THE BOTTOM-LEFT AND TOP-RIGHT POINTS OF THE ******/
/***** AREA OF THE SCREEN YOU ARE CHECKING FOR ******/
int determineControlArea (int left, int bottom, int right, int top)
{
    if (pointerX >= left)
    {
        if (pointerX < right)
        {
            if (pointerY >= bottom)
            {
                if (pointerY < top)
                {
                    return 1;
                }
                else return 0;
            }
            else return 0;
        }
        else return 0;
    }
    else return 0;
}

int determineQuadrant (int oX, int oY)
{
    if (pointerX < oX)
    {
        if (pointerY > oY)
        {
            return 1;
        }
        else
        {
            return 2;
        }
    }
    else
    {
        if (pointerY < oY)
        {
            return 3;
        }
        else
        {
            return 4;
        }
    }
}
/*-------------------------------------------------------------------*/

///////////////////////////////////////////////////////////////// 
///************
/// *** GLUT REGISTERED CALLBACK FUNCTIONS (SEE main) ***///
///*****************************************************/// 
/////////////////////////////////////////////////////////////////

/*----------------------------
---------------------------------------*/

///// WINDOW RESIZE CALLBACK FUNCTION ---
///// IF USER TRIES TO RESIZE THE WINDOW THIS FUNCTION WILL BE CALLED /////
GLvoid ReSizeGLScene(GLint Width, GLint Height)
{
    glutReshapeWindow (Xsize, Ysize); // keep window same size always!
}

/*-------------------------------------------------------------------*/

///// KEYBOARD CALLBACK FUNCTION--
///// THIS FUNCTION IS CALLED WHENEVER A KEY WITH AN ASCII CODE IS PRESSED /////
void NormalKey(GLubyte key, GLint x, GLint y)
{
    switch ( key ) {
        case ESCAPE :
            glutDestroyWindow(window); // Kill our window
            exit(0); // exit
            break; // Do we need this???
        }
}

/*-------------------------------------------------------------------*/

///// MOUSE BUTTON CALLBACK FUNCTION--
///// THIS IS CALLED WHENEVER A MOUSE BUTTON IS PRESSED /////
void Mouse ( int which_button, int state, int mx, int my )
{
    switch( which_button )
    {
        case GLUT_LEFT_BUTTON:
            if( state == GLUT_DOWN ) { // if left button is pressed....
                leftButton = 1; // set leftButton flag
                pointerX = mx;
                pointerY = Ysize-my; // the Y axis coords returned by GLUT
                                // are reversed!
                glutPostRedisplay(); // call the rendering function again
                                // to update display
            }
            else {
                leftButton = 0; // reset flag
            }
            break;

        case GLUT_RIGHT_BUTTON:
            if( state == GLUT_DOWN ) { // if right button is pressed...
                rightButton = 1; // set rightButton flag
                pointerX = mx;
                pointerY = Ysize-my;
            }
            else {
                rightButton = 0; // reset flag
            }
            break;

        default:
            break;
    }
}

/*-------------------------------------------------------------------*/
MOUSE MOTION CALLBACK FUNCTION --
THIS FUNCTION RECEIVES POINTER COORDINATES FROM THE MOUSE
-- THIS IS WHERE THE MOUSE COORDS ARE CONVERTED INTO COORDS
FOR THE SOUND SOURCE, AND WHERE THE SLIDERS ARE CONTROLLED
void Motion (int mx, int my)
{
    // variables used for constraining source
    // movement to the surface of the sphere
    GLfloat hyp;
    GLfloat angle;

    if (leftButton || rightButton) // engage this routine only if
    {
        // a mouse button is pressed
        if (!(leftButton && rightButton)) // but NOT if both are pressed!
        {
            // update global mouse pointer variables
            pointerX = mx;
            pointerY = Ysize – my
            glutPostRedisplay();

            //*** CONTROLLING THE PLAN VIEW ***/
            //*********************************************************/
            //*********************************************************/
            //*** CONTROLLING THE PLAN VIEW ***/
            //*********************************************************/
            //*********************************************************/
            if (determineControlArea(150,300,300,450))
            {
                hyp = sqrt(pow((pointerY-planY),2) +
                           pow((pointerX-planX),2));

                if (hyp > largeRadius) // if the pointer is NOT on
                {
                    // the surface of sphere
                    switch (determineQuadrant(planX, planY))
                    {
                        case 1: angle=atan((planX-pointerX)/(pointerY-planY)); break;
                        case 2: angle=atan((planY-pointerY)/(planX-pointerX)+(PI/2)); break;
                        case 3: angle=atan((pointerX-planX)/(planY-pointerY)+(1.5*PI)); break;
                        case 4: angle=atan((pointerY-planY)/(pointerX-planX)); break;
                    }
                    sourceX = largeRadius*sin(-angle);
                    sourceZ = largeRadius*cos(-angle);
                    sourceY = 0.0;
                }
                else // if the mouse pointer IS on the surface of the sphere
                {
                    // CALCULATE POSITION OF THE SOUND SOURCE RELATIVE TO THE
                    // CENTRE OF THE SPHERE - x AND y ARE CALCULATED FROM MOUSE
                    // POSITION, WHILE z IS CALCULATED FROM x AND y
                    sourceX = pointerX-planX;
                    sourceZ = pointerY-planY;// because the y-axis of plan view is
                    // the z-axis of the sound-field
                    sourceY = sqrt((largeRadius*largeRadius)-(sourceX*sourceX)+(sourceZ*sourceZ));
                    if (leftButton) sourceY = -sourceY;
                }
            }
        }
    }
}
if (determineControlArea(300,150,450,300))
{
    hyp = sqrt(pow((pointerY-sideY),2) + pow((pointerX-sideX),2));
    if (hyp > largeRadius) // if pointer is NOT on surface of sphere
    {
        switch (determineQuadrant(sideX, sideY))
        {
            case 1: angle=atan((sideX-pointerX)/(pointerY-sideY)); break;
            case 2: angle=atan((sideY-pointerY)/(sideX-pointerX))+(PI/2); break;
            case 3: angle=atan((pointerX-sideX)/(sideY-pointerY))+(PI); break;
            case 4: angle=atan((pointerY-sideY)/(pointerX-sideX))+(1.5*PI); break;
        }
        sourceZ = largeRadius*sin(-angle);
        sourceY = largeRadius*cos(-angle);
        sourceX = 0.0;
    }
    else // if the mouse pointer IS on the surface of the sphere
    {
        // CALCULATE POSITION OF THE SOUND SOURCE RELATIVE TO THE
        // CENTRE OF THE SPHERE - Y AND Z ARE CALCULATED FROM MOUSE
        // POSITION, WHILE X IS CALCULATED FROM Y AND Z
        sourceY = pointerY-sideY; // because the x-axis of the side view
        // is the z axis of the sound-field
        sourceZ = pointerX-sideX; // because the x-axis of the side view
        // is the z axis of the sound-field
        sourceX = sqrt((largeRadius*largeRadius)-(sourceY*sourceY)-(sourceZ*sourceZ));
        // if left button click, then sound source is "behind"
        if (leftButton) sourceX = -sourceX;
    }
}

if (determineControlArea(0,0,150,150))
{
    hyp = sqrt(pow((pointerY-backY),2) + pow((pointerX-backX),2));
    if (hyp > largeRadius) // if pointer is NOT on surface of sphere
    {
        switch (determineQuadrant(backX, backY))
        {
            case 1: angle=atan((backX-pointerX)/(pointerY-backY)); break;
            case 2: angle=atan((backY-pointerY)/(backX-pointerX))+(PI/2); break;
            case 3: angle=atan((pointerX-backX)/(backY-pointerY))+(PI); break;
            case 4: angle=atan((pointerY-backY)/(pointerX-backX))+(1.5*PI); break;
        }
        sourceX = largeRadius*sin(-angle);
        sourceY = largeRadius*cos(-angle);
        sourceZ = 0.0;
    }
else // if the mouse pointer IS on the surface of the sphere
{
    sourceX = pointerX-backX;
    sourceY = pointerY-backY;
    sourceZ = sqrt((largeRadius*largeRadius)-(sourceX*sourceX)-(sourceY*sourceY));
    if (rightButton) sourceZ = -sourceZ;
}

///////////////////////////////////////////////////////////////
//*****************************************************************************
//*** CONTROLLING THE STEREO HEIGHT SLIDER ***
//*****************************************************************************
///////////////////////////////////////////////////////////////
if (determineControlArea(43,200,107,400))
{
    heightPos = pointerY;
    if (heightPos > (300+slideLength-buttonSize))
    {
        heightPos = (300+slideLength-buttonSize);
    }
    if (heightPos < (300-slideLength+buttonSize))
    {
        heightPos = (300-slideLength+buttonSize);
    }
    height = (heightPos-231)/138;
    glutPostRedisplay();
}

///////////////////////////////////////////////////////////////
//*****************************************************************************
//*** CONTROLLING THE STEREO WIDTH SLIDER ***
//*****************************************************************************
///////////////////////////////////////////////////////////////
if (determineControlArea(200,43,400,107))
{
    widthPos = pointerX;
    if (widthPos > (300+slideLength-buttonSize))
    {
        widthPos = (300+slideLength-buttonSize);
    }
    if (widthPos < (300-slideLength+buttonSize))
    {
        widthPos = (300-slideLength+buttonSize);
    }
    width = (widthPos-231)/138;
    glutPostRedisplay();
}

glutPostRedisplay(); // update the display

/*********************************************************************************

///// OPENGL DRAWING CALLBACK FUNCTION
---
///// THIS IS WHERE ALL THE GRAPHICS RENDERING GOES ON ... /////
GLvoid DrawGLScene()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // clear screen & depth buffer
    glPushMatrix();
    glMatrixMode(GL_MODELVIEW); // select modelview

 Sử dụng các đường kẻ xung quanh vùng kiểm soát bằng cách vẽ các đường thẳng.

```
// COLOR 3UB(255,255,1.0); // yellow
glColor3ub(255,255,1.0);
glBegin(GL_LINES);
glVertex3f(150.0f,450.0f,450.0f);
glVertex3f(300.0f,450.0f,450.0f);
glVertex3f(150.0f,450.0f,450.0f);
glVertex3f(150.0f,300.0f,450.0f);
glVertex3f(300.0f,450.0f,450.0f);
glVertex3f(300.0f,150.0f,450.0f);
glVertex3f(150.0f,300.0f,450.0f);
glVertex3f(450.0f,300.0f,450.0f);
glVertex3f(300.0f,150.0f,450.0f);
glVertex3f(300.0f,150.0f,450.0f);
glVertex3f(450.0f,150.0f,450.0f);
glVertex3f(450.0f,300.0f,450.0f);
glVertex3f(0.0f,0.0f,450.0f);
glVertex3f(0.0f,150.0f,450.0f);
glVertex3f(0.0f,150.0f,450.0f);
glVertex3f(150.0f,150.0f,450.0f);
glVertex3f(150.0f,150.0f,450.0f);
glVertex3f(150.0f,0.0f,450.0f);
glVertex3f(150.0f,0.0f,450.0f);
glVertex3f(0.0f,0.0f,450.0f);
glEnd();
```
Appendix III

เหมาะสมของ red ที่เล็กที่สุด:

// POSITION THE SMALL SPHERE ON THE BACK VIEW SPHERE \\
glColor4f(1.0,0.0,0.0,1.0); // select red โมเดลสร้างรูปทรงต่าง ๆ
glLoadIdentity();
glTranslatef(backX, backY, backZ); // move to centre of side view sphere
glTranslatef(sourceX, sourceY, -sourceZ); // move to position of sound source
gColor4f(1.0,0.0,0.0,1.0); // select red
// DRAW THE BACK VIEW SPHERE \\
glutSolidSphere (smallRadius,10,10);

// POSITION THE SMALL SPHERE ON THE SIDE VIEW SPHERE \\
GLfloat matrix[16];
glLoadIdentity(); // reset modelview
glTranslatef(sideX, sideY, sideZ); // move to centre of side view sphere
glRotatef(90.0f, 0.0f, 10.0f, 0.0f); // rotate 90degrees around y-axis
glTranslatef(-sourceX, sourceY, sourceZ); // move to position of sound source
gColor4f(1.0,0.0,0.0,1.0); // select red
// DRAW THE SIDE VIEW SPHERE \\
glutSolidSphere (largeRadius, 30, 30); // draw sphere

// POSITION THE SMALL SPHERE ON THE PLAN VIEW SPHERE \\
GLfloat matrix[16];
glLoadIdentity(); // reset modelview
glTranslatef(planX, planY, planZ); // move to centre of side view sphere
glRotatef(270.0f, 10.0f, 0.0f, 0.0f); // rotate 270degrees around X-axis
glTranslatef(sourceX, -sourceY, sourceZ); // move to position of sound source
gColor4f(1.0,0.0,0.0,1.0); // select red
// DRAW THE PLAN VIEW SPHERE \\
glutSolidSphere (smallRadius,10,10);

// POSITION THE SMALL SPHERE ON THE CENTRE VIEW SPHERE \\
GLfloat matrix[16];
glLoadIdentity(); // reset modelview
glTranslatef(centreX, centreY, centreZ); // move to centre of side view sphere
glRotatef(202.5f, 0.0f, 1.0f, 0.0f); // rotate 202.5degrees around y-axis
glTranslatef(-12.5f, 1.0f, 0.0f, 0.0f); // rotate -12.5degrees about x-axis
glTranslatef(sourceX, sourceY, sourceZ); // move to position of sound source
gColor4f(1.0,0.0,0.0,1.0); // select red
// DRAW THE CENTRE VIEW SPHERE \\
glutSolidSphere (smallRadius,10,10);
glLoadIdentity(); // reset modelview
glTranslatef(centreX, centreY, centreZ); // move to centre of sphere

glColor4f(0.0,1.0,0.0,0.4); // select green

glutSolidSphere (largeRadius, 30, 30); // draw sphere

DrawText(214,452,"Plan",1.0,1.0,1.0); // label plan-elevation

/* Initialisation and window creation */

glutInit(&argc, argv); /* Initialize GLUT state. */

glutInitDisplayMode(GLUT_RGBA | /* RGB and Alpha */
GLUT_DOUBLE| /* double buffer */
GLUT_DEPTH); /* Z buffer (depth)*/

glutInitWindowSize(Xsize,Ysize); /* set initial window size. */

window = glutCreateWindow("AmbiPan"); /* Open a window with a title. */

InitGL(Xsize,Ysize); /* Initialize our window. */

/* Now register the various callback functions */

glutDisplayFunc(DrawGLScene); /* Function to do all our OpenGL drawing. */

glutReshapeFunc(ReSizeGLScene);

glutKeyboardFunc(NormalKey); /*Normal key is pressed */

glutMouseFunc(Mouse);

glutMotionFunc(Motion);

/* Now drop into the event loop from which we never return */

glutMainLoop(); /* Start Event Processing Engine. */
return 1;
}
Towards a suite of VST plugins with graphical user interface for positioning sound sources within an ambisonic sound-field in real time.

JAMES MOONEY
Music Technology Group, University of York, Heslington, York, YO10 5DD, United Kingdom
Email: James@Mooney.tc

This paper details the research, design, and implementation of two VST plugins – Ambipan and Ambidec – for carrying out fully 3D ambisonic surround-sound panning and decoding, respectively, in real-time. The initial design and implementation towards an intuitive custom-designed graphical user interface (GUI) for the former is also detailed, although at present the GUI has not been integrated with the plugin. VST is briefly explained from a technical standpoint, before specific details of the plugin implementations are covered. Similarly OpenGL, the graphics API chosen for the realisation of the GUI, is briefly explained before details of the implementation are given. The paper concludes with some suggested directions in which this research could be further developed.

1. INTRODUCTION

As a composer interested in the creative possibilities of sound in space, I have noticed that surround sound technologies have been directed almost entirely towards the consumer market. This essentially precludes any kind of creative implementation of surround sound systems. Further, of those applications and technologies that do exist, few – if any – are genuinely conducive to the creative process. Many computer applications exist only as stand-alone programs, forcing the composer to leave their chosen mixing environment and carry out spatial processing externally. Further, a large proportion of the software available operates on an offline (non-real-time), command-prompt basis, effectively removing any elements of spontaneity and experimentation from the exercise.

It is for these reasons that I have decided to develop software for panning sound sources in real-time in an ambisonic sound field, with the explicit objective of ease-of-use. An additional focus of my research is the development of an intuitive graphical user interface (GUI) specifically for this purpose. Further, it is intended that the software will be fully functional with minimal hardware requirements: composers should be able to compose spatially without the need for specialised or novel input devices.

2. THE AMBISONIC SYSTEM

Vennonen describes ambisonics very concisely as ‘a psychoacoustically optimised way of encoding an infinite number of sound directions into a limited number of channels, and then decoding them to a given loudspeaker layout on playback’ (Vennonen 1994). The means of psychoacoustic optimisation, as they have not to date been implemented in the Ambipan and Ambidec plugins, will not be discussed.

An important aspect of ambisonics that distinguishes it from the other surround-sound paradigms is that the processes of recording (encoding) and reproduction (decoding) are clearly separated from one another. In this capacity, it is an example of a ‘kernel’ system (Gerzon 1977: 400). An infinite number of directional components can be encoded into four channels – collectively known as ambisonic B-format – and then decoded for reproduction over a particular setup of loudspeakers.

In order to calculate the necessary B-format channels to artificially place a sound source within a sound-field, its position must be described by two angles. These angles must indicate the horizontal rotation, or azimuth (angle A) and the vertical rotation, or elevation (angle B). By convention, 0° azimuth, 0° elevation is directly in front of the forward-facing listener, at the centre-front of the sound-field. Positive angle of rotation for azimuth is anticlockwise from this position, while positive rotation for elevation is upwards. The required B-format channels are then given by the following equations:

\[ W = \text{source signal} \times 0.707 \]
\[ X = \text{source signal} \times \cos A \times \cos B \]

This is the case in first-order ambisonics. With higher orders, which involve capturing directional information at a higher resolution, more channels are required. At present, Ambipan and Ambidec support only first-order ambisonics.
The final stage of the ambisonic process is decoding, which concerns the reproduction of the encoded sound-field over loudspeakers. This is carried out by an ambisonic decoder – which can be implemented either as hardware or software – designed for decoding to a particular arrangement of loudspeakers. Decoding involves calculating a unique signal to be fed to each loudspeaker in the rig: each speaker, depending on its position relative to the centre of the sound-field, is fed a mathematically derived combination of the B-format signals.

Purse (2000) provides equations which deliver ‘a strict idealised response that satisfies the Ambisonic matching equations’, resulting in a stable spatial image over a relatively small area in the central area of the sound-field. These have a high directivity factor (i.e. a high directional XYZ to omni-directional W ratio). ‘Controlled opposites’ decoding equations are also given, which provide an ‘in-phase’ response, delivering ‘a larger listening area at the expense of some directional information’. These equations rely more heavily on the W component, and therefore have a lower directivity factor. It is these ‘in-phase’ equations which have been implemented in the Ambidec plugin.

3. AMBIPAN SPECIFICATION

Ambipan takes a mono or stereo input signal and outputs four channels of first-order ambisonic B-format. Each Ambipan plugin is capable of panning the audio from one track of a VST mixing desk. In this capacity it works in a similar way to a channel insert effect. By installing an Ambipan plugin on each active track on the mixing desk, an ambisonic mix can be created.

Internally, Ambipan positions the sound source based on two parameters: azimuth and elevation. When the GUI is fully integrated, these parameters will be expressed simultaneously by means which will be described later. Using the default interface provided in Steinberg’s Nuendo8 sequencer, a horizontal slider is created for each parameter.

3.1 AMBISONIC PANNING OF STEREOPHONIC INPUT SIGNALS

Ambipan provides two controls for defining the separation of the two channels of a stereo input file: Stereo Width, and Stereo Height. These parameters control the separation, in degrees, between the left and right channels in terms of angles of azimuth (θ) and elevation (Φ), respectively. This is illustrated in the diagram below.

Both stereo-width and -height are limited to maximums of 180˚. When working with stereo input files, the absolute position of the sound source (that is, the position inferred by the azimuth and elevation values) is taken as the point exactly in the middle of the active arc which runs between the ‘left’ and ‘right’ channels of the input file.

3.2 USER-DEFINABLE PRESETS

Ambipan allows the user to specify values of azimuth, elevation, width, and height, and store these values as a preset. At present up to sixteen presets can be stored, but this number has been arbitrarily chosen and could be very easily increased. When the user switches from one preset to another, the parameter values of the second preset will instantaneously be restored.

4. AMBIDECC SPECIFICATION

8 At present, Steinberg’s Nuendo is the only VST host application with sufficient multi-channel capabilities to support ambisonic encoding and decoding in real time.
Ambisonic decoding is carried out by a separate plugin, Ambidec, which should be installed in the master section of the VST mixing desk. Ambidec takes four B-format signals as its input and outputs a feed signal for each speaker in the chosen speaker setup. Supported decoding options are mono, stereo, square, 3/2, and cube. All apart from the cube array support only horizontal surround sound. Each speaker feed signal is output to a discreet channel on master section of the VST mixing desk. The number of the master channel to which each speaker feed signal is sent for each setup is illustrated in the diagrams below. Horizontal-only rigs are shown in plan elevation. In each diagram, a dotted line runs from the centre of the rig to 0˚ azimuth (i.e. pointing towards the front of the rig).

Ambidec has no custom GUI, and therefore uses the default interface provided by the host application. In Nuendo 1.0 the default interface consists of a horizontal slider, which can be moved from left to right to select the desired speaker rig to decode to.

5. AMBIPAN GUI SPECIFICATION

Control over the position of a sound source in three simultaneous dimensions (four if we were to include time) for the Ambipan plugin is achieved through a custom-designed GUI (although the plugin and interface have not yet been integrated). The sound-field is represented metaphorically as a sphere. The interface consists of three different views (elevations) of this sphere: back elevation; side elevation; and plan elevation. These represent the sound-field viewed directly from behind, directly towards the right-most semi-sphere, and directly from above, respectively. There is also a quasi-three-dimensional representation of the sound-field which comprises a sphere surrounded by a wire-frame cube. This view is rotated slightly about the X and Z axes so that movement in all directions can be visualised. An screenshot of the GUI at its present state of development is given in the diagram below. It shows a sound source positioned at 0˚ azimuth, 0˚ elevation.
The user can place the sound source – represented by the smaller sphere – anywhere on the surface of the sphere by clicking at the desired position on one of the three different elevations of the sound-field. But for any given point on these elevations, there are two possible positions on the surface of the complete sphere that could be implied. For example, if the user clicks in the centre of the back elevation of the sound-field, this could indicate a sound source at either 0° azimuth, 0° elevation, or it could represent a sound source at 180° azimuth, 0° elevation. This problem is overcome by making use of the two mouse buttons. In the case of the back elevation, the left mouse button indicates that the sound source is in the front half of the sound-field, while the right button indicates that it is in the rear half. Similarly, a left click in the plan elevation will place a source in the bottom half of the sound-field, while a right click will place the source in the top half. With the side view, perhaps obviously, the left button is used for moving sounds around the left-most semi-sphere of the sound-field, while the right button is used for movements in the right semi-sphere. Any movement of the sound source in one view, will result in the repositioning of the source in all the other views. Note that the quasi-three-dimensional view is purely for visualisation purposes, and does not include any element of user control.

Stereo-width and –height parameters can be adjusted with the two sliders provided. At present the GUI does not produce a separate visual object for each of the left and right channels of a stereo input file. Instead, stereo sound sources are represented as one object, which is located at the absolute position of the sound source.

6. VIRTUAL STUDIO TECHNOLOGY

Virtual Studio Technology (VST) is a standard for digital signal processing software designed by Steinberg to promote compatibility and extendibility between VST-implemented applications. There are two types of VST software: the VST host; and the VST plugin. The host is the ‘main’ audio application, the environment that provides the framework and key functionalities for audio processing, as well as dealing with tasks such as file-handling. VST hosts are standalone applications, and include sequencers such as Steinberg’s own Cubase VST and Nuendo, and Emagic’s Logic Audio; wave editors such as Wavelab and Sound Forge; and other audio applications including Audiomulch and Fruityloops. VST plugins are not standalone applications in themselves, but can be used from within any VST host application to extend the functionality of the software.

That VST plugins can be opened from within a host application, making, for example, the process of adding effects to sound files much easier, is a major advantage of the system, and there are several others. As VST is a growing technology, increasingly implemented by DSP software companies, so the range of VST-compatible software is ever increasing. Further, the fact that VST plugins can be used in any VST host application allows users to work within their chosen environment, thus reducing the need to learn to use new software. From a development point of view, VST makes the task of programming simple audio applications easier. In creating VST plugins programmers do not have to worry about such things as file- and window-handling, as these are all managed by the host application. Further, most VST host applications provide their own means of presenting parameters to the user therefore, assuming no highly specific user-interface characteristics are required, developers need not concern themselves with the process of writing a GUI for their plugin, at least during the development stage.

All of the VST base-classes have been coded such that they will compile for Windows, Apple Macintosh, BeOS, and SGI platforms without changes to the source code. This makes the task of writing cross-platform software, which can be a very time-consuming process, much simpler. Finally, the VST specification is, more or less, open source, at least insofar as the source code and documentation is available to download (Steinberg 2000). This means that VST development is open to anyone, and this has resulted in a busy mailing list from which support may be sought.

6.1 COMMUNICATION BETWEEN HOSTS AND PLUGINS
The communiqué between hosts and plugins is facilitated through the use of standard C++ classes. Steinberg have provided template structures and classes for the 'generic audio effect', which contain all the necessary functions for hosts and plugs to communicate. This 'template' takes the form of a C++ class named audioeffectx, which in turn inherits from the class AudioEffect. The latter defines the functionalities of the VST 2.0 specification, while the former defines the earlier VST 1.0 methods. All the relevant files are supplied with the VST Software Developers Kit (VSTSDK) (Steinberg 2000). In developing a plugin, a further C++ class is derived from audioeffectx, thus immediately inheriting all of its properties. Developers then override (redefine) certain base-class functions as necessary to suit the particular needs of their plugin.

6.2 DEALING WITH PARAMETERS AND PROGRAMS

VST plugins can also provide various parameters which can be changed in real time either by the user – via a graphical user interface – or automatically by the host, using parameter information recorded from user input. These parameters are used to change the characteristics of the audio processing carried out by the plugin. In VST, all parameters are handled as floating point numbers between 0.0 and 1.0.

The dynamic control of parameters is achieved through a group of functions, which are predefined in the AudioEffect base-class. These functions are variously called by the host whenever it needs to set (or query) the value of a parameter, or obtain information about a parameter for displaying to the user.

Plugins may also support multiple ‘programs’. A program is a ‘snapshot’ of all the plugin’s parameter settings; a preset, effectively. Typically, a program class is defined whereby each program object has its own private set of plugin parameters.

6.3 AUDIO PROCESSING

When the host wants to send audio to a plugin for processing, it calls the AudioEffect function process, or processReplacing, depending on how the plugin has been installed. In doing so it passes two pointers, one to an array of input buffers and one to an array of output buffers. Exactly how many buffers are contained within each array depends on how many inputs and outputs the plugin has declared in its constructor function. The plugin can then access the audio through the input buffers, process it, and return it to the host via the output buffers. This is achieved by cycling around an audio processing loop that lasts for as long as there are samples contained in the input buffers.

7. IMPLEMENTATION OF THE AMBIPAN PLUGIN

The Ambipan header file defines a macro named TO_RADIANS with the value 6.2831853, (approximately) equal to 2\pi. This is used to scale parameters from 0.0 – 1.0 to 0.0 – 2\pi, thus giving the full range of 0˚ to 360˚. These new values can then be used to express angles of azimuth and elevation. These converted parameter values are then substituted into the ambisonic encoding equations in the process function.

7.1 INTERPOLATION BETWEEN PARAMETERS

In an early prototype of Ambipan, it was found that rapid changes to parameter settings resulted in unacceptable clicks in the output audio. This problem was overcome by interpolating between the old parameter setting and the new whenever a parameter is changed. When the azimuth parameter (for example) is changed the value assigned to the parameter is converted to an angle in radians and assigned to a variable named newAzimuth. The interpolation is then carried out in code as follows:

\[
\text{azimuth} = (i \times \text{oldAzimuth}) + ((1-i) \times \text{newAzimuth});
\]

where azimuth is the value that is actually substituted into the encoding equations, and oldAzimuth is (initially) the value of the azimuth parameter before it was changed. The constant i is an interpolation factor initialised to the value 0.999. Each cycle around the processing loop, the value of azimuth is assigned to oldAzimuth. In this way, when a parameter changes, the corresponding variable which is used to control the audio processing will not change instantaneously, but rather will interpolate exponentially between the old parameter value and the new.

7.2 STEREO-WIDTH AND –HEIGHT
The variables which control stereo-width and -height are multiplied by \((\text{TO}_\text{RADIANS}/4)\) and reassigned, resulting in a range of \(0.0\) to \(\pi/2\). If we consider the absolute azimuth of the stereo sound source to be \(\theta\), and width to be the value of the scaled stereo width parameter, then:

\[
L = \theta + \text{width} \\
\text{and} \\
R = \theta - \text{width}
\]

This means that the angle of azimuth of the ‘left’ channel of the stereo input signal is increased in the positive direction by an angle of \(\text{width}\) radians. The azimuth of the ‘right’ channel is decreased, also by an angle of \(\text{width}\) radians. Similarly with stereo height, the elevation of the ‘left’ channel will be modified in the positive direction, and that of the ‘right’, in the negative, both by an angle of \(\text{height}\) radians.

In the `process` function this is achieved firstly by creating two sets of B-format signals, one for the ‘left’ channel and one for the ‘right’. These signals are then added together and the results divided by two, to derive the final B-format signals.

### 7.3 TEMPORARY IMPLEMENTATION AS A SEND EFFECT

At present, Ambipan will only install in Nuendo as a channel send effect. Because the plugin output buffers are overwritten, (this is what happens when plugins are installed as send effects) only one Ambipan plugin can be installed at any time, because installing further plugins would overwrite all other output. This is, obviously, not an ideal scenario but is for various reasons unavoidable at this stage in the plugin’s development. The plugin may not be installed as a channel insert effect as would be desirable, because Nuendo does not allow insert effects to have more than two outputs. There are ways around this which the author is aware of, but their implementation relies on the Ambipan GUI being fully integrated with the plugin, which has not been possible thus far.

### 8. IMPLEMENTATION OF THE AMBIDECEC PLUGIN

Ambidec has only one parameter, which is used to determine which speaker rig the B-format input will be decoded to. Five macros, which represent the five decode rig options, are `#define`: MONO; STEREO; SQUARE; THREE_TWO; and CUBE. Depending on the value of the parameter, one of these macros is assigned to an index variable, which is then used to switch between the various different decoding equations in the `processReplacing` function. Note that it is the responsibility of the user to ensure that there are enough master section channels to receive the speaker feeds for the chosen decode option. It is also up to the user to ensure that these channels are correctly routed to output buses and, in turn, to physical sound card outputs.

### 9. IMPLEMENTATION OF THE AMBIPAN GUI

The graphical user interface for the Ambipan plugin has been implemented using OpenGL running under the GL Utility Toolkit (GLUT). OpenGL is a versatile C-based API capable of rendering two- and three-dimensional graphics. It was chosen primarily because of its suitability for dealing relatively easily with objects in simulated three-dimensional spaces. Also, as a widely accepted standard for computer graphics it is well documented and support is easily and readily available. OpenGL does not include any functions for dealing with window handling, input and output, or any of the other platform-specific interactions that a user interface may have to call upon. It deals only with the rendering of graphics.

GLUT is the chosen framework within which OpenGL rendering can take place. It processes events using callback functions, that is, functions which the programmer has specified should be called in response to certain events, such as mouse and keyboard input. It also interfaces with the operating system to provide the window handling functions that make OpenGL rendering possible.

#### 9.1 INITIALISING THE DISPLAY

Firstly, the OpenGL coordinate system was calibrated to be exactly the same as the screen coordinates (in pixels) of the rendering window. This is illustrated in the diagram below.
The coordinate system was fixed by constraining the window so that it cannot be resized by the user. Were this not the case, resizing the window would result in the OpenGL coordinates remaining the same while the number of pixels along the screen coordinate axes would change.

Three solid spheres, which represent the three different elevations of the sound-field, are then created using GLUT’s predefined sphere object. Nine global variables are initialised to describe the centre positions of these spheres. These are backX, backY, backZ, sideX, sideY, sideZ, planX, planY, and planZ, and are initialised so that the coordinates of the centre positions of the back, side, and plan-view spheres are (75,75,225), (375,225,225), and (225,375,225), respectively. A further two global variables, largeRadius and smallRadius, are used to carry the values for the radii of the sound-field and sound-source spheres, respectively. These are initialised to 50 and 6.

9.2 INTERFACING WITH GRAPHICAL ELEVATIONS OF THE SOUND-FIELD

It has been stated that the user can position a sound source anywhere on the surface of the sphere (sound-field) by clicking at a position on one of the three different elevations of the sound-field provided by the GUI. It is therefore necessary to create a routine which determines which of these three elevations the has chosen to interface with. For this purpose, a function named determineControlArea was defined. This function is passed two sets of coordinates (four integers) which represent the bottom-left and top-right corners of a rectangle (‘control area’) within the window. The function will return 1 if the mouse pointer is inside this control area, otherwise it will return 0. By calling this function with three control areas which surround each of the solid spheres, it is therefore possible to determine which one the user is trying to control.

Our three elevations of the sound-field represent one entity viewed from three different angles. The two dimensions of freedom offered by the mouse correspond with two simultaneous dimensions of control possible within each elevation, while the remaining third dimension must be calculated from the other two. In the case of the back elevation, the two dimensions of mouse control correspond with movements around the xy plane: the z position must be calculated. In the plan elevation, the mouse directly controls movements in the xz plane (y must be calculated), while with the side elevation, the user can directly specify the yz position (x must be calculated). This is illustrated below.

9.3 CALCULATING THE POSITION OF THE SOUND SOURCE
This is done by calculating cartesian coordinates for the sound-source relative to the centre of the sound-field, based on the mouse coordinates passed when the user clicks somewhere inside one of the control areas. Exactly how this is achieved will be explained, using the back elevation as an example.

We can deduce that, in the case of the back elevation, the x mouse coordinate reflects the position of the sound source along the x axis of the sound-field, offset by the x value of the centre of the back-elevation sphere. Similarly, the y mouse coordinate reflects the position of the sound source along the y axis of the sound-field, offset by the value of the y coordinate of the centre of the sphere. Cartesian coordinates relative to the centre of a sphere are related to the radius of that sphere according to the following equation:

\[ x^2 + y^2 + z^2 = r^2 \]

We know the values of x and y and r, and can therefore work out the value of z by rearranging the formula as follows:

\[ z = \sqrt{(r^2 - (x^2 + y^2))} \]

But consider the fact that every pair of xy values, there are two possible values of z: one negative and one positive. How we resolve this ambiguity is quite straightforward. If the user is controlling the position of the sound-source on the back elevation by clicking with the right mouse button, then the resulting z value calculated will always be negative (i.e. on the ‘screen side’ of the sphere), otherwise it will be positive.

9.4 CONSTRAINTING SOUND-SOURCE POSITION TO THE SURFACE OF THE SPHERE

The routine described above works fine as long as the mouse pointer is actually ‘on’ the surface of the sphere. However, recall that the control areas defined for each of the spheres are actually wider and taller than the spheres themselves. This means that it is possible for the user to click outside the surface area of the sphere but still be within the control area for that sphere. This could cause values of x and y (recall that we are using the back-elevation as an example) to be specified that are actually beyond the surface of the sphere, which would make the right-hand side of the equation used to calculate z, negative. As it is impossible to calculate the square root of a negative number, this will cause an error.

Therefore, we need to work out if the point the user is clicking on is on the surface of the sphere and if not, position the sound source on the point of a straight line drawn between the centre of the sphere to the mouse pointer that intersects with the edge of the sphere. To do this, we need to work out the length of that straight line: if it is longer than the radius of the sphere, then the mouse pointer is outside the surface area of the sphere. This line then becomes the hypotenuse of a right-angled triangle, whereby we can work out the angle of rotation about the z-axis. If we know this angle, we can place the sound source on the surface of the sphere at that angle of rotation about the z-axis, and on the plane \( z = 0 \). The lines and angles (lbs) which we may have to calculate are illustrated in the diagram below, where each white point represents a place outside the surface of the sphere where the user could potentially try to place the sound source. Recall also that backX and backY are the xy centre coordinates of the back-elevation sphere. PointerX and pointerY are the mouse coordinates, expressed in pixels.

From the diagram above we can see how the x and y coordinates of the mouse pointer, and the x and y coordinates of the centre of the back-elevation sphere, are used to calculate the lengths of the sides of the triangle opposite and adjacent to the angle \( \theta \). We can therefore work out the size of the angle according to the following trigonometric formula:

\[ \theta = \arctan(\text{?}) \]

We can also see that the way in which coordinate variables are used to calculate the lengths of the required sides of the triangle are different depending on which quadrant the mouse click occurred in (quadrants are numbered 1
It is therefore also necessary to ascertain which quadrant of the control area a mouse click has occurred in. For this, a function is defined which takes the coordinates of the centre of the control area as parameters and returns the number of the quadrant.

It can be seen from the diagram which angle (θ) we are calculating in each quadrant. Only in the first quadrant does the calculated angle actually represent the full angle of rotation subtended to the positive y-axis. Therefore, in the second quadrant we must add 90˚ (or π/2) to the calculated angle, in the third quadrant we must add 180˚ (π), and in the fourth quadrant we must add 270˚ (1.5π). Having worked out the appropriate angle, x and y coordinates for the sound source are calculated using the following formulae:

\[
\begin{align*}
x &= r\sin(\text{angle}) \\
y &= r\cos(\text{angle})
\end{align*}
\]

where r is the radius of the sphere. We know that z will always be equal to zero under these circumstances, because the x and y coordinates will always be placing the sound source on the very edge of the semi-sphere, where z = 0 is the only possibility.

9.5 PLACING THE SOUND SOURCE ON THE SURFACE OF EACH SPHERE

The small sphere that represents the sound source is placed on the surface of each sound-field elevation sphere by translating to its centre, and then further translating to the position specified by the sound-source coordinates. In the case of the back-elevation, this process is all that is required, because this elevation represents the sound-field viewed directly from behind. The other elevations, however, represent the sound-field viewed from two different angles of rotation. It is therefore necessary, before translating to the position of the sound source, to rotate the coordinate system by 270˚ about the x-axis in the case of the plan-elevation, and by 90˚ about the y-axis for the side-elevation.

9.6 CONTROL OF THE STEREO-HEIGHT AND –WIDTH SLIDERS

Control of the stereo-height and stereo-width sliders is a much simpler process than that of the positioning of the sound source on the surface of the sound field. A control area slightly wider and taller than each slider is defined so that the program code can tell if the user is trying to adjust the stereo-width or stereo-height settings. In the case of the stereo-height slider, when the user clicks in the corresponding control area, the y mouse coordinate is mapped onto the position of the slider in the OpenGL drawing function. If the value of the mouse coordinate is such that the slider would be positioned outside of the maximum and minimum extremities, then it is reassigned to be equal to the maximum or minimum permitted setting. Calculating the position of the stereo-width slider is a very similar procedure. The main differences are the control area defined, and that it is the x mouse coordinate that gets mapped onto the position of the slider button, because the stereo-width slider is horizontal rather than vertical.

10. CONCLUSIONS AND FUTURE WORK

The purpose of this project has not necessarily been to reach any kind of conclusion, but rather to engage in the development of a suite of software implementations with a view to providing functionality that does not exist in previous sound-spatialisation software. Specifically, the main aims of the project have been to produce a VST plugin implementations of ambisonic panning and decoding utilities, and to look into the possibility of developing more intuitive interfaces for controlling such software.

It is clear that, however much has been achieved so far, the Ambipan and Ambidec plugins, and the user interface for Ambipan, are works in progress. The author hopes to continue their development as part of the programme of study for a PhD, therefore a ‘wish-list’ of improvements and refinements – some merely desirable, some absolutely necessary – has been drawn up. Some proposed future developments are outlined, roughly in order of priority in the following sections.

10.1 IMPLEMENTATION OF AMBIPAN AS A SPATIALISER PLUG-IN WITH GUI

At present the implementation of the Ambipan plugin is such that, in Nuendo, it may only be installed as a channel send effect. In this state the plugin functions correctly if installed on a single VST mixing desk channel, but because its process function overwrites the plugin output buffers (the only way to avoid the original input signal causing erroneous B-format output), the channel on which Ambipan is installed will overwrite any other
VST channels that may exist. Obviously this situation is unsatisfactory, and therefore one of the most pressing aspects of the future development of Ambipan will be its implementation as an insert effect. Presently, Nuendo will not accept Ambipan as a ‘normal’ channel insert effect because it has more than the permitted maximum of two audio output streams. However, Nuendo, under the new VST 2.0 specification, does offer support for multi-channel-output insert effects if plugins are declared to be of the ‘spatialiser’ category. If this is the case then Nuendo will install the plugin in the panner section of the corresponding VST channel, which allows effects to have more than two audio outputs.

However, this has not yet been implemented due to the fact that plugins of this category do not have access to Nuendo’s default method for displaying plugin parameters. Enquiries as to the reasons behind this have proved unsuccessful, but evidently, the only way to interface properly with a spatialiser plugin is for the plugin to define its own custom GUI. As the Ambipan GUI is not yet at a sufficiently advanced state of development to be integrated with the plugin, so implementation as a spatialiser has thus far been impossible.

10.2 MORE CREATIVE USE OF THE PRESET PROGRAMS IN AMBIPAN

Ambipan allows the user to store up to sixteen sets of parameters as programs. At present, the user can select a preset program, and the parameter values will instantaneously change. This could be useful if we want to return a sound source to the same position repeatedly, for example. This functionality could be further developed by the addition of an adjustable timer, which would determine the time taken to interpolate between the parameter settings of two different programs. Such a feature exists in Steinberg’s GRM Tools plugin suite.

10.3 MORE FLEXIBLE DECODING OPTIONS

Ambidec allows the user to decode to monophonic, stereophonic, quadraphonic, 3/2, and cube speaker rigs. Decoding to rigs containing more than eight speakers is not, at present, a possibility, because Nuendo allows a maximum of only eight discreet channels on the master section of the VST mixing desk. However, more flexible decoding should be considered as a future possibility. For example, the implementation of a slider which interpolates between a mathematically strict idealised directional response, and an ‘in-phase’ response (Furse 2000) would be useful for optimising decoding for the size of listening area required.

User-definable sound-field manipulations such as rotation, dominance, and so on could also be carried out at the decoding stage. This would obviously introduce quite different interfacing challenges and it is for this reason that these functionalities have been avoided thus far.

Additionally, no psychoacoustic optimisation is included in the decoding carried out by Ambidec, and this could result in a marked improvement in sound-field stability if carefully implemented.

10.4 COLLATION OF ALL SPATIALISED SOURCES IN A SINGLE DISPLAY MODULE

With an Ambipan plugin installed on each channel of the VST mixing desk, each GUI (when fully integrated with the plugin itself) will provide visual information as to the position of the audio on that particular channel. For this reason, there will be no way in which two sound sources, from two different VST channels, can be visualised on the same interface. This would, however, be a very useful feature, and has been implemented in SpinAudio’s (2001) 3D Panner Studio, which is also a VST plugin. Research into how this is achieved should be considered as a future development.

10.5 IMPLEMENTATION OF DISTANCING LAWS

At present, Ambipan only allows the user to position a sound source on the surface of the sphere that metaphorically represents the sound-field. The main reason behind this is that, as soon as distance from the listener is to be implied, changes to the frequency content of sound sources must be imposed. Further, our perception of how distant a sound source is, is to a considerable extent dependent on the characteristics of the listening environment. Therefore any algorithm designed to simulate sound source distance would either have to make sweeping assumptions about the acoustics of the listening environment, or allow the user to configure these characteristics. This was thought to be too large an undertaking for a first implementation, but remains a consideration for future developments.

10.6 SECOND ORDER SUPPORT

Ambipan takes a monophonic or stereophonic sound source and outputs four channels of first-order ambisonic B-format. Similarly, Ambidec receives these first-order signals, and creates speaker feeds for the chosen speaker
Implementation of higher-order support has not been considered to date. Indeed, it is not known if Nuendo can deal with plugins with as many as nine outputs, the number of channels necessary to carry second-order B-format. The maximum number of channels permitted in the master section of the Nuendo VST mixing desk is eight, which perhaps suggests that more-than-eight channel support is not part of the Nuendo specification. However, should this change, second- and higher-order support should certainly be a consideration.

REFERENCES


