School of Electronic Engineering and Computer Science

MSc Computing and Information Systems

Project Report 2018

Real-Time Ambisonic rendering on the Bela Platform

Iuliu Teodor Radu

August 2018

Disclaimer:

This report, with any accompanying documentation and/or implementation, is submitted as part requirement for the degree of MSc in Computing and Information Systems at the Queen Mary University London (University of London)
It is the product of my own labour except where indicated in the text.

N.B.: The full files are to large to fit in the QMPLUS upload form. For a functional snapshot compiled version of the system, please initialise a Bela project with the exact files inside of 'ambisonic_spatialiser' from the following repositories:

Optimised Cube System — https://www.dropbox.com/sh/39v7icukh8wg2h8/AAAE_XJlNOmwRNs4ca9YSsyra?dl=0

Non-Optimised Quad System — https://www.dropbox.com/sh/lia2urhxkiawu38/AAD3qmLSugTPcP0CjN9cyZ5Qa?dl=0

Acknowledgement:

I would like to acknowledge and thank my supervisor, Dr Rebecca Stewart for her helpful guidance and insight throughout this dissertation project.

Table of contents

Abstract

Pioneered in the late 1960s by Michael Gerzon of the Mathematical Institute in Oxford, ambisonics is a superior method for the three dimensional representation and rendering of sound. Why then has it not taken off as the main technology for sound spatialisation used in movie theatre and home entertainment systems? By means of implementing, documenting, and evaluating a real-time ambisonic renderer on the Bela/BeagleBone Black platform, I aim to show why nearly sixty years later it is worth taking another serious look at the applications of ambisonics, reframing it through the lens of virtual, augmented, and mixed reality systems. I outline the main limitations that thus far have impeded its commercial adoption and attempt to show how today's technology nullifies some of these limitations. The system is a proof of concept that a single core processor can run an implementation of a first order ambisonic renderer that can spatialise multiple sources and realistically binauralise them in real-time based on inertial measurement unit orientation inputs. The ARM Cortex A8 processor present in the BeagleBone Black runs at a frequency of 1 gigahertz, representing only a fraction of standard mobile processing power present in most smartphones today. In future, it would be interesting to compare its performance against the implementation of a Vector Base Amplitude Panning (VBAP) spatial renderer on the same platform in terms of ITD, ILD, and spectral accuracy.

Chapter 1 — Introduction

Currently, software developer kits allowing easy 3D sound design for Virtual Reality (VR) are a hot topic with companies like Google developing their own Resonance Audio SDK or Oculus providing a spatialisation feature as part of their Audio SDK. These SDKs are an effort to make the sound design process for VR systems easier and faster, while using ambisonics technology for superior results in environmental modelling. A two sided obstacle seems to influence the mainstream adoption of ambisonics technology in VR and other consumer audio solutions in general. One side of the problem newly being addressed by the aforementioned companies is the tooling available for content creation making it a slow and difficult process for developers to create high quality, well-chiselled audio. Historically, this links into the reason why ambisonics has not been adopted across the board as a main sound spatialisation technology since its inception nearly sixty years ago, a matter I will discuss later. The other difficulty in making VR totally immersive lies in achieving under 15 millisecond latency in the audio reproduction. If there is too much perceived delay during an interaction inside a virtual space, between the point of user input and the point that the resulting sound is heard, the experience becomes unrealistic. Even though there are currently not many VR experiences offering three dimensional audio, real time audio reproduction is still difficult, especially where the extra complexity of network latency or wireless Head Mounted Display has to be dealt with. This is where the question of processing power becomes relevant. VR applications consume notoriously huge amounts of processing power, demanding much higher video refresh rates and resolution than any other gaming technology to date. Although Mobile VR has been democratised through savvy SDKs while doing away with arcane mathematical jargon, current mobile processing power and battery storage is insufficient for rendering realistic

experiences. For cutting edge VR experiences, the applications still need to run on a PC with dedicated graphics cards (GPUs) doing the heavy lifting. I would argue that audio rendering, especially involving ambisonics spatialisation should also be offloaded onto a dedicated audio processor akin to the way that geometric modelling tasks are offloaded to GPUs. The good news is that the processor architecture required for digital signal processing for sound spatialisation comes in a much less expensive form than GPUs. The low cost ARM Cortex A8 processor architecture includes Single Instruction Multiple Data (SIMD) technology, making Fast Fourier Transforms (FFTs) and floating point multiplications extremely efficient. It is worth keeping in mind that this processor model was released in 2005 and has since been deployed on numerous mobile and embedded platforms. It is by no means representative of the latest technology, with current multicore mobile technology hugely outperforming the A8's highest end configuration. Therefore, this project aims to show what can be achieved with just a fraction of today's latest technology in terms of real-time sound spatialisation.

The Bela Platform is a real-time, ultra-low latency embedded system for real-time audio processing featuring the ARM Cortex A8. It is based on the BeagleBone Black (BBB), runs a custom Linux distribution, and optimally integrates audio processing and sensor connectivity into a compact package. Its low latency makes it ideal for use in live performance and art installation scenarios. The purpose of this project is to implement a real-time first order ambisonic renderer and binauraliser on this platform as a proof of concept demonstrating that this can be done inexpensively and is scalable by simply increasing processing resources. The design implications of the renderer are discussed

and the output is evaluated in terms of ITD, ILD, and duration of asynchronous thread exeuction.

The end product features a lot of “moving parts” that have to be carefully managed in order to achieve a real-time experience. Here is a short overview of the sequence of events that process the audio from input to output, which will be discussed in depth in Chapter 3.

- Input sources are loaded into buffers from .wav files and/or real-time microphone input is initialised and continuously collected at the beginning of each call to the render() function.
- Orientation information is collected from an Adafruit BNO055 inertial measurement unit (IMU) and converted into pitch, yaw, and roll variables. These in turn are converted to azimuth and elevation angles to comply with B-Format encoding standards.
- An asynchronous ambisonic rendering function controls the B-Format encoding and decoding process that takes the azimuth and elevation angles and uses them to spatially represent the input audio and to output it as virtual speaker signals.
- The virtual speaker signals get sent to an asynchronous FFT process that translates this data into the frequency domain at specific intervals. This frequency domain data is then convolved with as many Head Related Impulse Responses (HRIRs) (see chapter 2.1.7) as there are virtual speaker outputs and translated back into the time domain.
- The time domain speaker outputs are then added into the Bela platform’s stereo output channels to give the final result.

The end goal is to create a movable soundscape, with the 3D audio sources changing in relation to the user’s head movements. Initial steps in the development of this system

involve adding functionality for playing separate prerecorded samples from particular ‘locations’ inside a two-dimensional field, one at a time. Later steps involve the optimisation of the system and adding three-dimensional functionality, live microphone inputs, live head tracking, and multiple input source capability.

This project could also be framed in the wider context of the progress to commercial success, or rather lack thereof, of ambisonics technology. As briefly mentioned earlier, ambisonics has not made it through to commercial success in the past sixty years due in part to the cumbersome setup and finicky user experience of such a system. In order to make use of ambisonics in a surround sound system be it at home or in a movie theatre, one would need to install an extra piece of expensive hardware, the ambisonic encoder/ decoder. In addition, the speaker array would have to be positioned in a very precise layout corresponding to the ambisonic decoder’s settings. First order ambisonic rendering offers a very small listening area or “sweet spot” while giving a low spatial resolution. This makes it very difficult to allow multiple people to have a good listening experience in a theatre or performance space scenario. Moreover, there has to be a direct sound path between each of the speakers in the array and the listener for a realistic experience. This is especially difficult to achieve when there are obstructing objects or people in the sound’s path that acoustically shadow it. Fortunately, VR technology does away with such limitations as most audio content is to be delivered through headphones. The ambisonic encoder/decoder is still necessary, but now it can take the form of an inexpensive digital processor. The decoded outputs are still in the form of speaker signals, but these can be binauralised by convolution with corresponding HRIRs. Since the HRIRs are recorded in a controlled environment, it means that upon convolution, there is a direct sound path with

no obstructions between each of the virtual speakers and the user. Inevitably, the user is also placed at the centre of this virtual array of speakers. With this comes the problem of stably fixing the virtual speakers to one point in space. Unless the user's head is immobilised, he or she will have three degrees of freedom to turn. Given the headphone setup, the virtual space is obviously rendered the same way irrespective of its relation to the user's head. This naturally gives the user the impression that the sound environment is following their head movements. In order to compensate for this and keep the correct location of sound sources in relation to the user's head, an IMU is placed onto the headphones to track the head's orientation, correctly offsetting the angles fed to the B-Format encoder.

Chapter 2 begins with providing a general overview of sound spatialisation. Some psychoacoustic implications of spatial sound are explored such as the cocktail party effect in section 2.1. Sections 2.2 through 2.5 go into detail about the technical aspects of how ambisonic technology works covering the BFormat, encoding, decoding, and recording techniques. Some insight into the alternative sound spatialisation technique of Vector Base Amplitude Panning is provided in Section 2.6. Finally, Section 2.7 provides some details about HRTFs and how they are used to spatialise sound using convolution.

Chapter 3 provides a detailed account of the system implementation including design decisions, programming strategies, and optimisation.

Chapter 4 concludes the system documentation with an in depth evaluation of the ITD, ILD, and CPU usage of the system.

Chapter 2 — Literature Review

2.0 — Sound Spatialisation

Three dimensional localisation of sound sources by humans is achieved by an interaction between different parts of the auditory system. The human brain is able to tell the azimuth and elevation of a sound source as well as its distance from the head based on many different auditory system cues. It has been shown that humans are better at sound localisation in the horizontal plane than the vertical plane (being able to identify the azimuth better than the elevation of a sound source). The difference in time of the arrival of a sound between the two ears is important for discerning the horizontal direction of a sound source. This is known as the interaural time difference (ITD). A sound will be picked up by the ear closest to it first. The other ear will also pick up the signal after the sound has travelled the extra distance. Naturally, the ITD will vary depending on head dimensions.

The distance to a sound source is extrapolated by the brain based on differences between signal magnitudes at the different ears. The sound signal magnitudes are affected by many variables such as head resonance and diffraction. The head has an acoustic shadowing effect as it obstructs the sound travelling to the ear facing the opposite side of the sound source. This obstruction causes an attenuation of the signal, while the head's resonance also modifies some spectral components of the sound reaching the opposite ear.

It has been shown that spectral characteristics of a signal reaching the eardrums plays a role in the identification of a sound's source in the vertical plane as well as monaural

identification in the horizontal plane. The external ear, including the pinnae and ear canals as well as the head modify the spectral and temporal attributes of sound. For instance, the pinna influences the signal by shadowing, reflection, dispersion, diffraction, interference, and resonance (Blauert, 1996, p. 63). Although these phenomena have been proved to occur, it is impossible to describe them in detail mathematically as the irregular shapes of pinnae vary greatly between individuals. As a result, it is much easier to obtain a 'recording' of the anatomy of an ear by means of defining Head Related Transfer Functions for use in sound spatialisation. Section 2.7 below will detail how Head Related Impulse Responses (HRIRs) are acquired and used to reproduce a sound heard from a specific point in space through binauralisation.

2.1 — The Cocktail Party Effect and directed attention implications

The brain has the ability to focus its auditory attention to one stream of stimuli at a time and marginalise other stimuli of less interest. In an environment of numerous talkers, a partygoer could selectively focus their attention on one particular conversation while blocking out other sounds. This is known as the Cocktail Party Effect, which also scales across other scenarios.

Another example of this effect is well represented by concertgoers, especially those attending orchestral, multi instrumental performances. It is possible, notably for musicians and discerning listeners to focus on just one instrument family or the interaction between specific instruments or 'voices'. In the context of sounds with narrow frequency spectra such as those produced by pitched instruments, visual cues are necessary to aid the localisation of the sound sources in addition to the auditory cues discussed above.

Auditory localisation works much better for sound sources that have a broader frequency spectrum such as the human voice and general percussive environmental noises in the real world which are arbitrary and unorganised. Fortunately, this allows for many applications of sound spatialisation, ranging from artistic to mission critical systems.
A question that springs to mind while exploring the Cocktail Party Effect is 'How many voices could a human focus their attention on simultaneously?'. Anders Ericsson's Deliberate Practice theory (Ericsson et al., 1993) espouses some points of interest that we could take as a guide. In a nutshell, the theory supports the case that both quality and quantity of practice influence an individual's acquisition of an expert level of performance at any particular skill. It becomes relevant to our Cocktail Party Effect insofar as the theory describes the fact that the human mind can keep attention on between three and five elements at any given time for the purpose of improving at a particular task. In the context at hand, it would be worth exploring whether this theory holds true for keeping track of multiple sound sources simultaneously under different conditions. The results would guide design decisions when spatialising sounds, defining desirable functionalities for different use cases and environments. Hypothetically, an optimal implementation of an air traffic communications audio renderer will be different to that of a spatial music renderer for live performance as the attention of the user needs to be directed differently in each scenario.

2.2 — Ambisonics

Ambisonics is a two stage spherical surround sound technique allowing the reproduction of sound sources in a three dimensional field. It differs notably from other surround sound techniques in that it does not transmit speaker signals directly but rather, it encodes a speaker-independent representation of a sound field, which is eventually decoded to a

specific speaker array or rendered binaurally. This representation is known as the B-format.

The first stage of ambisonics involves encoding a sound source based on its horizontal and elevation angles. The second stage involves decoding the B-format into speaker signals for playback based on speaker positioning. This two step process highlights both advantages and disadvantages of the ambisonics technique. The encoding stage ensures a decoupling between the sound data and actual playback, while the decoding stage allows for very accurate rendering topping the quality of traditional panning surround systems (Elen, 1998).

However, the two step process means that commercialising this technology and making it user friendly is difficult, as extra hardware is required on the user's end compared to mainstream stereo or 5.1 surround technology. In order to encode and decode the B-Format sound (Baume et al., 2012), extra physical hardware is required as part of the sound system. Other reasons ambisonics is not widely used in today's surround sound technology are incompatibilities with existing sound systems as well as a lack of user friendly audio production software. Although attempts have been made to ease the burden on end users' systems such as the development of the G-Format (pre-decoded B-Format for standard 5.1 surround systems), ambisonics remains largely unmarketable for wider commercial use due in part to a lot of arcane mathematical jargon and sparse technologies and file formats in want of standardisation.

2.3 — B-Format

The 360 degree sound field information is encoded into the B-Format. The most basic first order ambisonic B-Format carries four channels of information which could be thought of as microphone polar patterns. These channels are conventionally labelled W, X, Y, and Z. The W channel represents an omnidirectional polar pattern, while X, Y, and Z correspond to polar patterns along the three dimensional axes. From an audio engineering perspective, W could be thought of as an omnidirectional microphone which picks up sound from all directions at equal gain and phase; X, Y, and Z could be though of as figure-of-8 bidirectional microphones pointing along the forward-backward, left-right, and up-down axes respectively. It is worth noting that the B-Format assumes the positioning of the above 'virtual microphones' perfectly at the sphere's origin. This presents a number of limitations when it comes to physically recording ambisonically, discussed later in Section 2.1.5.

The B-Format encoder allows us to represent any sound signal ambisonically, thus allowing for a large range of applications. It allows for the deliberate placement of a sound source in any point in the spherical 'soundscape'. The first-order ambisonic encoder takes three parameters, the azimuth θ, the elevation ϕ, and an input signal S. The encoder generates the W channel by reducing the input signal's amplitude by 3dB: $W = \frac{S}{\sqrt{2}}$. The omnidirectional channel's amplitude is reduced in order to evenly distribute the signal's level across the four ambisonic channels (Malham et al., 1995, pp. 58-70). The other channels are calculated as follows:

$$X = S \times \cos\theta \cos\phi \qquad Y = S \times \sin\theta \cos\phi \qquad Z = S \times \sin\phi \tag{2.1}$$

The B-Format decoder allows for the interpretation of sound field data. The decoding algorithm needs information about the target speaker topology in order to compute functions for distributing and weighting the ambisonic channels across all of the speaker signals. For example, in decoding for a simple two dimensional four speaker array, only the W, X, and Y ambisonic channels are needed. If the speakers are equidistant and placed at 45 degree angles from the origin, or 'listening sweet-spot', then the four signals would be computed from the B-Format as follows:

$$LeftFront = W + \frac{X+Y}{\sqrt{2}} \quad RightFront = W + \frac{X-Y}{\sqrt{2}} \quad LeftBack = W + \frac{-X+Y}{\sqrt{2}} \quad RightBack = W + \frac{-X-Y}{\sqrt{2}} \tag{2.2}$$

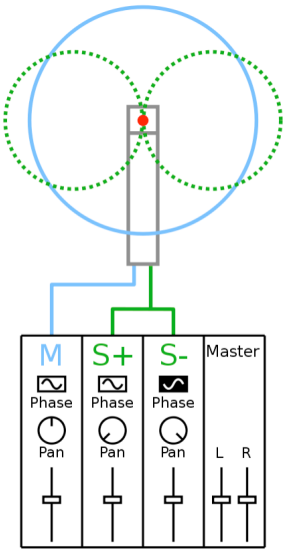
The reduction by 3 dB (division by $\sqrt{2}$) necessary to achieve a cardioid source directional response for each loudspeaker.

As briefly mentioned before, an omnidirectional signal will be present in all of the decoded speaker feeds. This is one of the features which makes the ambisonic technique very different from traditional 5.1 surround. For instance, the rear channels are not merely used for adding ambience or environmental sounds. Rather, each channel is treated equally, thus achieving a more realistic rendering in comparison to other surround systems.

2.4 — Higher Order Ambisonics

A limitation of first order ambisonics is that the spatial resolution of sound is low. It has been shown that sources were difficult to localise and were not stable in native first-order material recorded by a tetrahedral microphone (Nettingsmeier et al., 2011). The B-Format

Figure 2.1 — Mid-side stereo (lainf, 2007)



represents a series of spherical harmonic decompositions up to a predetermined order. Due to computational limitations, these components have to be truncated into floats or doubles. Physically, these decompositions represent the pressure field (our W channel) and its spatial derivatives of successive orders at the origin (Daniel, 2003). In essence, a higher order of spherical harmonic decompositions will yield sound sources that resolve more accurately, while enlarging the 'sweet-spot' listening area. Please refer to Appendix 1 for a table detailing the functions used for encoding up to 3rd order B-Format signals. Increasing the ambisonic order does not come without tradeoffs. With increased orders, the number of speakers increases quadratically for full sphere decoding: $(\ell + 1)^2$ where ℓ is the order. For horizontal only decoding, the number of speakers needed are $2\ell + 1$.

2.5 — Ambisonics Recording Techniques

First order ambisonics could be thought of as the mid-side (MS) stereo recording technique in three dimensions (see Figure 2.1). MS involves setting up a cardioid microphone centred on the sound source, while using a bidirectional microphone to pick

up sound from the sides. The final stereo image could then be tweaked in post production by mixing the centre signal with the side signals for each stereo channel. When scaling this idea to multiple dimensions, we are presented with the difficulty of true coincidence in microphone placement. For example, a two dimensional array would consist of two bidirectional microphones, one along the left to right axis, and one along the front to back axis as well as an omnidirectional capsule. Theoretically to achieve a perfect reproduction, the centre of the bidirectional microphones should be in precisely the same spot. Fortunately, stacking the microphones vertically achieves near perfect coincidence for sources located in the horizontal plane. However, further expanding into the third dimension by stacking another bidirectional microphone along the up to down axis breaks the coincidence along the vertical plane.

Sound field microphones are a compromise that allow the recording of signals that are readily encodable into B-Format. One such design is the tetrahedral array of microphones which record signals for the Left-Front-Up, Left-Back-Down, Right-Back-Up, and Right-Front-Down channels. The initial format recorded by this array is known as A-Format, however, its channels can easily be mixed in order to deduce the W, X, Y, and Z B-Format channels. Some tetrahedral microphone models directly encode their output into B-Format without the need for an intermediate step.

Higher order recording of ambisonic components is not possible using single microphone capsules. However, there are some commercially available microphones such as the 32 channel em32 Eigenmike using equally distributed omnidirectional capsules that perform

signal processing in order to derive higher order ambisonic components (Binelli et al., 2011).

2.6 — Vector Base Amplitude Panning (VBAP)

An alternative for reproducing a three-dimensional placement of sound around a listener is VBAP (Pulkki, 1997). It uses a different approach involving amplitude panning between groups of speakers by extending stereophonic panning to multiple speakers. One of VBAP's noteworthy advantages is that it enables the use of an unlimited amount of speakers, placed in an arbitrary array, the single requirement being that each speaker has to be approximately equidistant from the listener. In a horizontal plane, virtual sources are placed based on the weighting between adjacent speakers. The direction of the physical speakers is represented by vectors. Thus, the virtual source is calculated by adding the speaker vectors, each scaled by its respective channel amplitude factor. The amplitude factors for each channel are normalised to a constant value in order to ensure that the summed and scaled signals add up to the total desired amplitude of the virtual source. This technique is expanded in the case of three dimensions, by grouping the speaker topology into groups of three adjacent speakers, thus forming triangular constellations as seen from the listener's position. It is important to note that placing of a virtual source on the surface of the sound sphere within any of the above triangles, only requires the amplitude panning between the speakers constituting the vertices of that triangle. Therefore, amplitude normalisations have to be calculated between each speaker triplet. The union of all the speaker triplets form the full 'active region' of the listening area. Representing these relationships in terms of vectors instead of spherical trigonometry is advantageous from a computational point of view.

The most obvious limitation of VBAP is its inability to place virtual sources anywhere else other than on the active region. In other words, distance properties of a source cannot be modelled using VBAP. This is also true for the classical B-Format, however there are ways to design ambisonic systems that encode distance information through near field distance coding IIR filters (Daniel, ibid.) or by representing the radius of the 3 dimensional sphere as an angular coordinate on the 4 dimensional hypersphere (Penha, 2008). Details of the implementation of such techniques are beyond the scope of this project, but may be explored as a follow up to the evaluation of existing ambisonic rendering libraries. The premise is that ambisonics offers superior localisation of sound compared to VBAP.

2.7 — Head Related Transfer Functions (HRTF)

In linear-time-invariant system theory, a transfer function is used to describe the output of a system given any input. An HRTF is then a description of how a sound is picked up by the ear from a particular point in space, thus capturing all of the physical cues associated with source localisation. It models the characteristics of a human ear from a systems analysis perspective. HRTFs can be used in pairs (one for each ear) in order to binauralise an input by synthesising sound spatially, giving the illusion that it is reaching the listener from a particular direction. The method for recording this information involves recording a Head Related Impulse Response (HRIR) from the point of view of the ear drum. This is done by placing a microphone inside a subject's ear and playing an impulse signal (usually a sine sweep) from a particular location. There are lots of HRTF databases to choose from, such as the CIPIC, KEMAR, and LISTEN databases. Each of them have different approaches, recording HRIRs at different elevation and azimuth position

increments. Due to the huge amount of data collected, scaling the usage of good quality HRTFs poses a problem as it is infeasible to choose an optimal HRTF from a large database for each individual user. This difficulty has been tackled by research attempting to distill smaller sets of HRTFs based on trials in which subjects had to rate the perceived externalisation quality, elevation, and front/back precision of the binaural synthesis of multiple HRTFs (Katz et al., 2012).

Arbitrary inputs can then be spatialised by convolving them with the HRIR. In practice, especially where computational power is limited and real-time deadlines have to be met, this is done in the frequency domain after taking the Fast Fourier Transform (FFT) of a windowed signal. Keeping in mind that convolution in the time domain is equivalent to point-wise multiplication in the frequency domain, it becomes apparent that it is more costly to implement the convolution in the time domain in real-time scenarios. As a result, I will be using the Neon10 library (discussed in Section 2.2.2) to perform FFTs and Inverse-FFTs as part of the convolution implementation. For the purpose of this paper, I will need to experiment with different sample lengths of HRIRs in order to manage the relatively low computational power of the ARM Cortex A8 processor. The 3dti_AudioToolkit provides a starting point as it contains seven of the IRCAM LISTEN HRIRs in 128, 256, 512, and 1024 sample versions. Most HRIRs are available as SOFA (Spatially Oriented Format for Acoustics) files for use in MATLAB, while some databases also provide raw .wav files of HRIRs. For the sake of simplicity in dealing with only one format for this project, I decided to use a single class for loading and managing .wav data. Chapter 3.1 will go into more detail as to how .wav data is manipulated.

Chapter 3 — System Documentation

3.0 — Overview

The Bela Platform runs an Integrated Developer Environment (IDE) deployed on the NodeJS javascript runtime. Code is organised into projects inside of the IDE, each of which has to contain a render.cpp file. This represents the entry point of each project and is invoked by the Bela system either directly from the IDE or through a terminal. As per the Bela specification, three functions have to be defined inside of render.cpp, the setup(), render(), and cleanup() functions.

setup() runs once at the beginning of the program's execution and is used to allocate memory, initialise any states, and load any files necessary in subsequent processing. This function is run before any audio or sensor processing is done. I take advantage of setup() to do some preprocessing as well as initialise variables before render is run. Firstly, the IMU is initialised and the sound source position vectors are calibrated based on the user's head orientation. The user has to put on the headphones before the program is started so that the IMU's points of reference get set to the initial head orientation. Next, the time domain and frequency domain buffers are allocated via malloc and initialised to 0 via memset and all of the buffer pointers are set (see 3.4). A Hann window is calculated and stored inside a global buffer for later use (see 3.7). Time domain data is then loaded from .wav files. Depending on how many speakers are used for the configuration, a WAVHandler class loads the correct HRIRs into buffers as well as any sound sources to be used for testing (see 3.1). Next, the FFT of each HRIR is taken in order to readily store

them in the frequency domain for efficiency. This saves processing power as whenever the HRIRs are needed for convolution, no extra processing time needs to be taken up. The B-Format encoder and decoder are initialised and the B-Format buffer is initially set to a test sinewave that can be heard at startup for a split second as confirmation that the ambisonic encoder and decoder works. All auxiliary tasks are initialised within setup(), allowing for FFT, IMU, and B-Format encode/decode functions to be called asynchronously (see 3.3).

render() is run by the Bela system in a loop, for the entire duration that the program is run. The Bela platform features an environment data structure called *context* that stores digital, analog, and audio data in buffers representing framed streams. Since input audio data for both stereo channels is held within the same buffer, these inputs have to be interleaved and stored in frames (see 3.1). To extract each point of discrete series data from each stereo channel, a frame by frame loop inside of render() is necessary. Analog inputs are also interleaved, but the Bela features 8 of these, making the aforementioned frame lengths twice as long as the audio frames, thus achieving an analog input sampling rate half of that of the audio sampling rate.

Since the render() function is called in a loop it is necessary to keep state between calls in order to save any processed information from earlier calls. In this implementation, this is generally done with the help of aptly named global variables. Therefore, the contents of render() do not make chronological sense as it contains a mix of past and current buffer processing depending on whether it is sending input to auxiliary functions or pulling information from buffers already having been processed by auxiliary functions. Therefore,

for clarity's sake, in this overview I will explain what happens inside of the render() function chronologically.

Firstly, the IMU auxiliary task is scheduled and throttled, so that it does not run more often than it is necessary. Live audio samples from the input channels are collected and mixed into a mono buffer. For initial purposes, it is not necessary to support stereo inputs on this system. The desired audio is wrapped into a B-Format buffer to be encoded. Once enough samples have been collected inside of the input buffer, an asynchronous B-Format encode and decode task is scheduled. This task pulls information from the IMU and encodes/decodes the sound source with the corresponding angle variables. The decoded output is stored in virtual speaker buffers. Once sure that the speaker buffers are filled, render() schedules the FFT auxiliary task which asynchronously translates each speaker feed into the frequency domain, convolves them with the corresponding HRIRs, and performs an inverse FFT to designated output buffers. The output buffers are then added into the Bela's audioOutChannel buffers thus piping the final processed stream to audio out. Please refer to Appendix 2 for a flowchart of *render*'s execution.

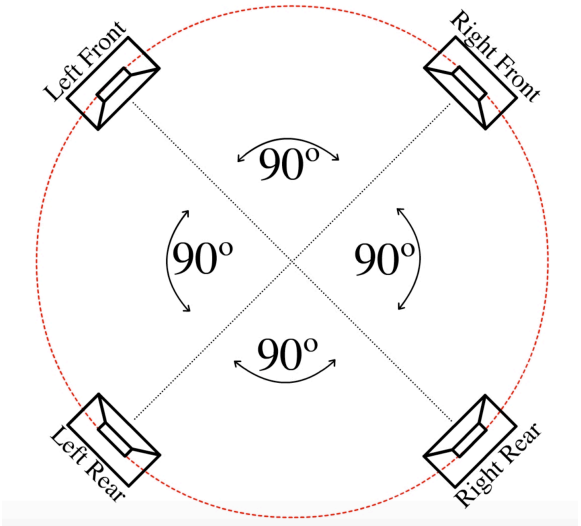
cleanup() runs once at the end of the program execution and should be used to deallocate any resources that have been allocated at setup().

3.1 — Virtual Speaker Layout

The end goal of this system is to spatialise sounds in three dimensions. Naturally, the first step was to implement two dimensional spatialisation. As mentioned before, when decoding the B-Format, the layout of virtual speakers is required by the ambisonic decoder. The ambisonic decoder then generates speaker signals for each of the virtual

speakers located at the specified location. A standard layout that works well is the off centre quad speaker layout requiring four speakers. Figure 3.1 illustrates this layout. This is the layout used in this implementation. Since we are not working with physical speakers, but rather binauralising the speaker output, we require a pair of HRIRs for each virtual speaker location. Generally, most HRTF datasets contain HRIRs for the quad speaker setup, namely, at azimuths of 45°, 135°, 225°, and 315°. In order to binauralise these speaker signals, it is necessary to respectively convolve the speaker outputs with HRIRs at those directions. Since binauralisation requires two convolutions per speaker signal, a total of 8 convolutions are necessary for the quad setup. It is worth noting that the two dimensional B-Format representation of sound sources only requires the W, X, and Y channels to be encoded since there is no information for the Z axis. Relating this back to the mid-side stereo microphone analogy, there is no virtual microphone oriented along the up down axis in this case.

Figure 3.1 — Quad Virtual Speaker Locations (Digenis, 2001-2002)



After correct ambisonic encoding and decoding is achieved for a two dimensional quad virtual speaker setup, the next step was to implement three dimensional spatialisation. A standard speaker layout that works well for three dimensional ambisonic decoding is the off centre cubic layout which we use in this implementation. This time, the layout contains a pair of speakers placed at the same azimuths as in the quad layout, but each speaker of each pair has an elevation of 45° and -45° respectively. The B-Format representation of sound sources now requires the W, X, Y, and Z channels to account for the full sphere. For the purpose of binauralisation, the LISTEN IRCAM HRTF dataset accommodates convolution for each of the virtual speaker directions. This time, the total amount of convolutions necessary is 16, putting twice as much strain on the system as the quad layout.

The library used in this system for the purpose of ambisonic encoding and decoding (discussed in further depth in section 3.6 below) provides convenient naming for the speaker layouts in order to automate the decoding process internally. In our project, the quad speaker and cubic speaker setups are indicated by respectively passing the '`kAmblibQuad`' or '`kAmblibCube`' enumerators to the B-Format decoder configuration.

3.2 — Loading Sound Files

Modularising the system for loading sound files was a design decision that makes it very flexible to experiment with different amounts of input .wav files. For the purpose of this project, it is a requirement to load both HRIRs in the form of .wav files as well as other recordings serving as sound sources. The WAVHandler class abstracts the fussy aspects of loading .wav files and provides uniform functionality for loading multiple files of similar

length into designated buffers. It features local buffers that are dynamically allocated based on .wav file requirements, specified through the constructor by the calling context, allowing for intermediary processing such as deinterleaving. The WAVHandler leverages the *libsndfile* library through a WavLoader class. The *libsndfile* library provides functionality for parsing wav files and extracting information from them into a purposely designed sf_info data structure. sf_info contains information about the number of channels present in the wav file, the sample rate of the audio, the amount of frames, the format, and other legacy metadata. My WavLoader class deals with the logic for file paths and takes care of allocating and deallocating memory for each buffer that is passed to it by reference. It is also responsible for passing file metadata such as wav data length and returns -1 to the WAVHandler class in the event that the file/s only have mono channel and thus deinterleaving is unnecessary.

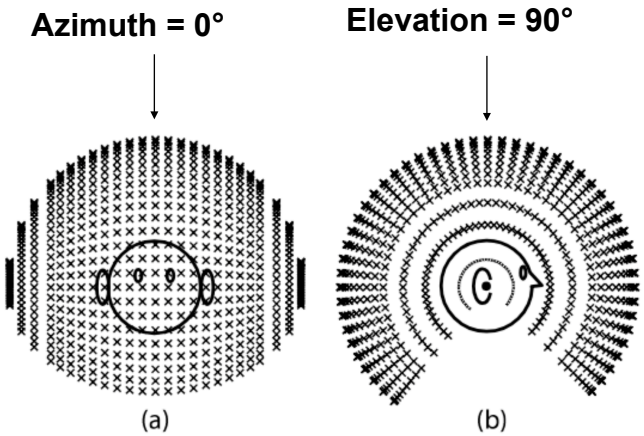
Another rationale behind the WAVHandler class was to allow *render.cpp* to specify the path and number of files to be read from a location inside of the folder. An example execution would look as follows: if 4 HRIRs are necessary to be loaded, then the WAVHandler constructor would be called as `WAVHandler("path/to/hrtf/<filePrefix>", 4);`. The file prefix is then taken into the WavLoader class which runs through a loop 4 times and each time appends ‘0’ through ‘3’ respectively to it, thus loading every file. This assumes that the file names are prepared for this convention. To clarify, four HRIRs might be labelled as ‘hrtf0.wav’ through ‘hrtf3.wav’ respectively, the prefix being ‘hrtf’ and the appended number ‘0’ through ‘3’. For added simplicity, I have written a python script, *renamer.py* that prepares the files present in a folder with the desired prefix and set of sequence numbers.

WAVHandler also provides functionality for deinterleaving stereo buffers in the case that left and right channels need to be stored separately. In the case at hand, the channels have to be separate and thus the WAVHandler constructor automatically deinterleaves the buffers if WavLoader returns 0 (i.e. wav file/s have stereo channels). In addition, WAVHandler exposes two types of methods allowing to make use of loaded wav data. One type feeds one sample at a time upon each call to the method and advances pointers accordingly. This might be used in the event that we would like to feed a certain amount of samples from either or both channels one at a time. This feature is akin to the way that the Scanner object in Java achieves parsing one token at a time. The other type of method spools the entire wav channel buffer into a target buffer passed to it by reference. This latter method is what the render.cpp file at hand uses. In this implementation, the wavs are loaded after a WAVHandler has been instantiated. At this point, the wav files are already loaded in the external class, but need to be 'spooled' into *render.cpp*'s global buffers. For example, `hrtf_handler->spool_into_buffer(hrtf_td_right[i], RIGHT);` loads the ith time domain right-channel HRIR samples into the `hrtf_td_right[i]` global buffer.

3.3 — HRTFs

To clarify once more, to make use of the correct HRTFs in this project, we need the Head Related Impulse Responses for each each ear with impulses being played at very specific points in space. HRIRs are then merely impulse response functions with the human ears acting as the dynamic systems which the impulse is being relatively measured to. As detailed in section 2.7, there are a handful of databases of HRIRs available. To describe

Figure 3.2 — Angle increments (Algazi et al., 2001)



the rationale behind my choice of HRTFs, I will briefly describe the features of two of the most noteworthy databases. A more comprehensive comparison between multiple databases is provided in Appendix 3. The CIPIC database by UC Davis features measurements of 43 human subjects, 27 men and 16 women (Algazi et al. 2001). The HRIR recordings are 200 samples long, with azimuth increments of 5° with a full 360° range and elevation increments of 5.625° with a range between -45° and 90° (directly above the subject). Figure 3.1 shows the collection of HRIRs available. It is generally considered that the point on the azimuth directly in front of the subject is 0°. Points on the azimuth clockwise from the front of the subject are labelled 0° to 180° while points on the azimuth anticlockwise are labelled 0° to -180°. It is worth mentioning that the distance from the sound sources to the subjects was 1 metre in these sessions. 1 metre is generally acoustically considered to be at the boundary between the near and far fields. The ambisonic encoder/decoder at hand assumes that the sound sources are in the far field for reasons beyond the scope of this dissertation. The second IRCAM LISTEN project database differs in that the azimuth and elevation increments are more coarse at 15°, the sound source distance is 1.95 metres, and versions of both raw 8192 and

compensated 512 sample lengths are provided. A total of 187 HRIRs are provided for each of the 51 subjects versus CIPIC's 1250 HRIRs. Chapter 3.6 will describe why for the purpose of this project, both CIPIC and LISTEN contain suitable HRIRs for convolution. This project uses LISTEN's subject #1008's HRIRs among others as this is the first of a group of 7, namely #1008, #1013, #1022, #1031, #1032, #1048, and #1053 that have been identified as an optimal solution as claimed by Katz and Gaëtan (see section 2.7). These 7 HRIRs are also used as part of the 3dti AudioToolkit library implementation for simulation of virtual hearing aid devices. Another justification for using this set of HRIR pairs is that they come in 512 sample versions which allow for easy integer calculations and simple buffer allocations and operations. A last minor consideration is that the source distance is 1.95 metres away from the subject compared to CIPIC's 1 metre, putting the sources well clear beyond the near field.

The design of this project assumes that HRIR buffers are initialised separately from any buffers representing other loaded wav files as described in section 3.1 above. The HRTF_LENGTH macro definition at the top of render.cpp is the central point of control for HRIR buffer lengths. This is an important design aspect as HRIR buffers are initialised (memset) to 0 no matter what their length is. Therefore when the WAVLoader class loads the HRIR wavs, it does not interfere with any extra zeros past the length of the HRIR. This means that the HRIR buffers will be zero padded by default in the event that they are longer than the HRIR itself. As can be seen later in section 3.7, zero padding has a special function when taking the FFT of a windowed signal for the purpose of convolution.

3.4 — Auxiliary Tasks

Multithreading is the process of running multiple pieces of code in parallel. This is especially effective on multicore technology, however it also works well with single core technology. Since the ARM Cortex A8 only features one core, it is up to the Linux operating system to manage multiple threads by setting priorities and keeping a queue. Especially in the case of single core multithreading, it is important to note that the operating system is delegated the task of blocking threads that are not supposed to run and to do some efficient time management based on priority settings in order to prevent the risk of creating a race condition between threads. At the same time, critical threads dealing with real time processing should set a mutex (lock) on variables or buffers holding sensitive data in order to prevent the loss of information through overwriting.

The main thread management system for Bela is an abstraction known as AuxiliaryTask. Auxiliary tasks can be created with a specific priority and scheduled at desired points in the code. By default, Bela runs the audio processing render() as an auxiliary task with the highest priority of 95. Linux threads have a default priority of 0 and can range from 0 to 99. In the case of this project's render.cpp file, the auxiliary tasks are created inside of the setup() function. The `Bela_createAuxiliaryTask(std::function<void()> target_function, int priority, string name)` function returns a desired auxiliary task. A local function is passed as an argument to the above function call to be executed asynchronously upon scheduling of its respective auxiliary task. Auxiliary tasks are scheduled through the `Bela_scheduleAuxiliaryTask(task)` function call. If program control attempts to schedule a task that has not finished yet having already been previously scheduled, nothing happens.

This project makes use of three such auxiliary tasks: the *i2cTask* for collecting and processing IMU data, the *gBformatTask* for processing the ambisonic encoder/decoder, and the *gFFTTask* for processing the FFT, convolution, and IFFT. For reasons involving frantic switching between threads, the `gBFormatTask` priority was set to 89, 1 lower than `gFFTTask` priority (90). The `i2cTask` is set to a low priority of 5 since the IMU is throttled to run every 100 samples, which is more than enough for keeping up with inevitable latency introduced by FFT buffer lengths which are at least 5 times larger. As described in the overview, the `gBFormat` task is only scheduled once enough samples have been stored into the `live_input_buffer` to be able to fill the entire `BFormat_buffer`. To facilitate design reasons outlined in section 4.1, the `BFormat_buffer` is set to the same length as the FFT length. The `gFFTTask` is scheduled every time the correct amount of samples have been accumulated in the `live_input_buffer` corresponding to the FFT length divided by the windowing factor also known as the hop size (`gHopSize`).

3.5 — Circular Buffers and global data passage

As a stream of live input data arrives from the microphone or a sound file, it has to be stored in a buffer sample by sample. The most intuitive way of doing this is to store the most recent sample at index 0 and have past samples at higher indexes. The memory management implications are less elegant for this solution. It would mean that every time a new sample comes in, every older sample needs to be offset by one index to the next memory location i.e. to make space for the most recent sample, memory at location $n-2$ has to be shifted to location $n-1$, $n-3$ to $n-2 \dots 0$ to 1 where n is the buffer length. This does not scale at all for big buffer sizes and is very costly in the context of real-time processing. Circular buffers represent a solution to this problem. No memory shifting has

to be done inside the buffers as long as a write pointer is kept pointing at the most recent sample. This write pointer is incremented every time a new live sample is to be stored. In order to prevent trying to access memory past the end of the buffer array, it is important to do some careful buffer pointer management and make sure that the buffer pointer is reset once it reaches the `buffer_length - 1` index. This way, we can think of the buffer as wrapping around in a circle. As reading from the buffer becomes necessary, it is possible to deduce a read pointer by computing the offset from the write buffer pointer, or alternatively managing a second read pointer tracking a certain amount of samples behind the write pointer. Within asynchronous contexts, it is important to only pass buffer pointers by value and not by reference, as from an auxiliary task's point of view, a global pointer will in all likelihood be arbitrarily incremented throughout its execution, thus more than likely causing unwanted interference with global buffers being manipulated.

Since there are a lot of interacting parts dealing with buffered data, this project implements nearly all buffers as circular buffers. All buffers present in the render.cpp file are global for the easy passage of data between function calls and auxiliary tasks. The end of the render() function contains a section of extensive buffer pointer management. Section 6.1 explores potential design improvements to the code that would for example enable the locking of buffers that are in use to prevent loss of real-time data. However, with some careful pointer management, double buffering of the B-Format task (see 3.6), and suitable auxiliary task priorities, the system at hand performs correctly.

For the sake of readability and efficiency, given multiple buffers of each type are necessary to account for the different virtual speaker signals, buffers are initially declared as two

dimensional arrays or arrays of pointers, avoiding cumbersome repetitious naming conventions. This means that the first array dimension in both cases refers to the nth virtual speaker, while the second dimension refers to the nth sample in the buffer. The only buffers initially declared as two dimensional arrays are the input and output buffers which have been set the arbitrary buffer length of 44100. Almost every other buffer is initially declared as an array of pointers as it is undesirable to prescribe a constant buffer length in the buffers' declarations. This allows some flexibility in dynamically allocating memory based on other variables later in the program's execution without having to keep a buffer pointer for each individual buffer.

The `BFormat_buffer_length` has been set to the length of the FFT as this simplifies the flow of samples between the B-Format task and the FFT task. It means that the FFT task can be fed the exact same chunk of data that has been previously processed by the B-Format task without the need for extra windowing, spooling of samples, and intermediary buffers. If the BFormat_buffer length was a fraction of that of the FFT length, the FFT task would still have to wait for FFT length samples to become processed by the B-Format task. Conversely, if the BFormat_buffer length was a multiple of that of the FFT length, it would add a big amount of latency as FFT tasks would be unnecessarily spaced out further, and when scheduled, would probably cause buffer under-runs while having the processor working in bursts rather than uniformly. Section 3.8 explains the FFT task in more detail.

3.6 — B-Format encode/decode, double buffering

Libspatialaudio provides a comprehensive implementation for up to third order ambisonic encoding and decoding, as well as binauralisation capabilities. Since this project implements binauralisation through convolution and uses the Ne10 library for efficiently performing FFTs, libspatialaudio binauralisation functionality will not be used. Libspatialaudio dates back to 2007 and was originally a fork of ambisonic-lib, developed to support higher order ambisonics. It is very well organised and features plug and play classes that are pulled into the render.cpp of this project. All of the functionality is self contained, with the only requirement that target and source buffers be passed to the encoder and decoder by reference as well as their lengths. The CAmbisonicEncoder and CAmbisonicDecoder objects need to be configured through a Configure() method that specifies the ambisonic order, dimensionality, and B-Format buffer length. Both of these classes feature Process() methods that intuitively transform the input to and from the B-Format. The CBFormat class stores the B-Format content. For added efficiency, it overrides the +=, -= etc. operators to allow the mixing of multiple sound sources into one B-Format buffer. This means that multiple sources could be encoded into the same B-Format buffer, while only one decode operation would be needed to calculate the required virtual speaker feeds. Appendix 4 features a table with the necessary mathematical transformations into B-Format up to third order ambisonics. In this system, support for multiple sources is very flexible, with the first source stream representing the live input from Bela's `audioIn`. The other source streams can be loaded by means of using the WAVLoader class to do the sample management. The `loadBFormatBuffer` performs pointer management for every stream loaded. As this function is embedded into the `process_BFormat` function to be run asynchronously, extra time complexity overhead is

added proportional to the amount of streams being run. That is why keeping the final single stream execution of a cubic setup well under 75% CPU usage is one of the goals of this project.

As one B-Format encode/decode cycle takes about 22 milliseconds or 970 live samples to complete for 4 speaker feeds with a buffer size of 2048 samples as an auxiliary task, it is safe to say that it is not even worth trying to run it synchronously, as this would more than likely cause buffer underruns. However, it runs fast enough that it is worth also loading the B-Format input buffer within the same auxiliary task. This simplifies some of the pointer management and in the case of non live audio input sources, adds the benefit of preprocessing the B-Format encode/decode one FFT length ahead, thus cutting out some of the CPU usage. However, this design requires the implementation of a double buffer for the decoded virtual speaker feeds. As the speaker feeds are always being accessed in each call to render(), if only one buffer were implemented, the B-Format auxiliary task would periodically and suddenly overwrite the stream being fed to the FFT task. This in turn would cause choppy audio. Double buffering ensures that dual virtual speaker feed buffers would be toggled between, with the render() reading in samples from one buffer, while the B-Format decoder would be preparing the other. This way, a continuous stream gets fed to the FFT task, the only sudden change between virtual speaker feed toggles being the resulting speaker weighting. Smoothing the changes in speaker weighting could be an improvement worth considering.

3.7 — IMU and head tracking

Belaonurhead is a library that allows the easy setup and integration of the Adafruit BNO055 9 axis IMU into Bela projects. It can represent orientation in terms of rotation quaternions or Euler angles. For the purpose of this project, it is necessary to perform rotational offset operations in directional vector form. The initial direction of each sound source is represented as azimuth and elevation angles in the *gInitialSourceAzimuth and gInitialSourceElevation* arrays. These are then converted into pitch, yaw, and roll rotational angles before they can be offset by input from the IMU inside a rotateVectors() function. They are then translated back into azimuth and elevation angles and stored in the *gUpdatedSourceAzimuth* and *gUpdatedSourceElevation* arrays. The values in these arrays are used directly as input to the ambisonic encoder's SetPosition() method. The rotateVectors() function is called asynchronously during the B-Format task right before running the B-Format encode/decode.

To achieve accurate head tracking, the IMU is tacked on top of the user's headphones. For optimal results, it is best to have the head in an upright position facing forward before starting the program to allow the IMU setup routine to register a suitable starting orientation.

3.8 — Fast Fourier Transform and convolution via frequency domain

Convolution is key for the purpose of binauralising the decoded output of the ambisonic system. Although it is possible to perform convolution in both time domain and frequency domain, it is much faster to do so in the frequency domain. From a computational perspective, performing convolution in the time domain requires an amount of integrations and signal shifts to be calculated equal to the sum of the length of the signals being

convolved. This is not very scalable with long, continuous signals. On the other hand, in the frequency domain, it is only necessary to perform a much simpler point-wise multiplication of the signals, thus scaling linearly in terms of time complexity.

Simply convolving signals is straightforward when both signals are known in advance and prepared in buffers. The Discrete Fourier Transform (DFT) of both signals is taken, they are then multiplied together in either polar or rectangular form, and finally an Inverse DFT (IDFT) is taken to reconstruct the time domain signal. In our case, the real-time system presents extra challenges. Namely, it is supposed to process live input in real time and convolve on-the-fly. As a result, the signal has to be convolved periodically, by storing samples in buffers before they are convolved. While keeping in mind that any latency introduced that is longer than 15ms becomes noticeable to the user, especially in live performance settings, we are given a window of about 441 samples within which to perform the convolution. Performing a convolution every 512 samples would provide close enough performance for the application at hand. This periodic computation of the DFT is called The Fast Fourier Transform (FFT). For this to work, the signal has to be windowed, or in other words, we have to wait for a certain amount of samples, in our case 512, to fill the FFT buffer before the computation is performed. Inconveniently so, windowing a signal rectangularly (i.e. simply leaving its amplitude unchanged and sending a chunk of samples to the FFT process as they are) presents the issue of spectral leakage. Upon performing a modification to the signal in the frequency domain through convolution, new phases and magnitudes are created in the resulting signal, which could be shifted in relation to the original signal's components. As a result, most phases would most likely be misaligned between FFT windows. The solution is to window the signals using the overlap-add

technique. This method involves applying an envelope to the time domain signals' amplitude before taking the FFT, helping to distribute any spectral leakage more evenly throughout the bins, thus smoothing out the result. As most of the effective envelopes have a somewhat parabolic shape, once the signal is transformed back into the time domain, its amplitude will also vary with the parabolic shape of its envelope. To prevent this while reconstructing the output signal with a constant amplitude, overlapping windows of samples could be sent to the FFT, and added upon taking their IFFT. This way, the processed output will have a constant amplitude after the overlapped addition. In this system, I use the Hann window function to apply to the time domain signal before taking the FFT. A special property of the Hann window function is that when it is overlap-added at 50%, it yields a constant output amplitude, thus reconstructing the expected signal. In this project, the interval between when FFTs are taken is called the *hop size*. The FFT window length is 1024 samples long, while the hop size is half that at 512 samples. To illustrate the FFT's execution, the system waits for 512 samples after which it performs the FFT on a total of 1024 past samples, convolution, IFFT, and adds the result to an output buffer. Continuously doing this, given that this process meets its asynchronous deadline as part of the auxiliary task, a latency of 512 samples is introduced, which amounts to about an acceptable 11.6ms. The Hann window function is given by

where n is the FFT buffer index and N is the FFT buffer length.

$$\omega(n) = 0.5\left[1 - cos\left(\frac{2\pi n}{N-1}\right)\right] \tag{3.1}$$

The system calculates the Hann window function at the setup stage and applies it to each time domain sample inside of the FFT process before transforming it into the frequency domain. Once the windowed input is in the frequency domain, the pre-calculated

frequency domain samples of the HRIRs are used to perform the convolution. Transformation into the frequency domain yields a complex output. In terms of the frequency domain, multiplying one signal by another means multiplying each of the magnitudes and adding the phases. Conversely, dividing signals involves dividing the magnitudes and subtracting the phases. Rather than converting the complex numbers for each frequency bin into polar form it is much easier to perform the multiplication in rectangular form. The following equations illustrate how to compute the new real and imaginary parts of the multiplication in rectangular form:

$$ConvSig.Re[n] = Input.Re[n] \times HRIR.Re[n] - Input.Im[n] \times HRIR.Im[n] \tag{3.2}$$

$$ConvSig.Im[n] = Input.Re[n] \times HRIR.Im[n] + Input.Im[n] \times HRIR.Re[n] \tag{3.3}$$

where the Re and Im dot notation respectively represent the real and imaginary parts of the labelled components, and n is the index of the frequency domain signal window. Once frequency domain convolution is complete, the IFFT of the new signal is taken to transform it back into the time domain. Usually upon overlapping and adding the output Hann windows, the resulting constant amplitude will be a bit higher than the initial input. If the same overall amplitude is required in the audio output, it is important to correct the resulting signal's amplitude by multiplying each sample by a constant scaling factor.

One further important consideration is ensuring that the convolutions performed are linear rather than circular. In a discrete context, when convolving two signals of different lengths, the resulting signal's length in samples will be equal to the sum of the two signals minus 1.

As a result, in our system, it is important to ensure that the FFT buffers are large enough to accommodate the resulting signal. In order to correctly perform the convolution, the frequency domain buffers of both signals have to be equal. This is to ensure that the frequencies of the bins are the same. The way to do this, is to pad the shorter signal with enough zeros that would make it the same number of samples as the other signal. When doing this however and ignoring the former point of accommodating linear convolution, the resulting signal, not fitting in the provided buffer, will circularly wrap and add onto itself from the beginning of the buffer until the sum of both signals' sample lengths minus 1 are accounted for. This causes phase and amplitude misalignment in the output which is very noticeable. Since the ambisonic system's input buffer is 1024 samples long, it was a design decision to ensure linear convolution by means of making the FFT buffer 1536 samples long, thus accommodating the sum of 1024 input samples and 512 HRIR samples. Unfortunately, when the FFT buffers are longer than twice the hop size (in this case greater than 1024 samples), the FFT process contained inside the Bela Auxiliary Task struggles to meet the 512 sample deadline given 16 concurrent FFTs having to be performed for each pair of HRIRs accommodating a cubic speaker layout. As a result, every other chunk of 512 samples does not get scheduled to be processed. The next section details a savvy optimisation involving the combination of ambisonic decoding with the convolution process. This technique saves on the amount of convolutions and FFTs that need to be performed and thus halves the amount of processing necessary to spatialise the audio in our case.

3.9 — Optimising binauralisation and B-Format decoding

The more than double length of the FFT buffer mentioned in the previous section runs well on the quad speaker setup, making the Auxiliary Task deadline every 512 samples. However, for the cubic speaker array, there are too many convolutions taking place which cannot be processed within the 512 sample deadline set by the hop size. An apparently intuitive solution would be to simply double the hop size and give the auxiliary task enough time to perform its task. However, the FFT buffer length also doubles, and the time it takes for an FFT task to complete also scales linearly.

The main issue that scales the CPU usage are the amount of convolutions that have to be done for each speaker. Therefore, for every speaker added, an extra two convolutions have to be done in order to binauralise the output. This does not scale very well in terms of time complexity, especially on an embedded system with limited processing resources. There is however a good solution in which the time complexity does not scale with the speakers, but rather with the ambisonic order of the system, and by extension, with the number of B-Format channels present for each order. Therefore, for first order ambisonic processing requiring four B-Format channels, only eight convolutions would be necessary, irrespective of how many speakers need to be decoded. This is done by combining the decoding and convolution process through the use of preprocessing of the HRIRs. This optimisation essentially skips the computing of virtual speaker signals altogether, removing the need for the B-Format decoding process in addition to halving the amount of convolutions we have to do for the cubic layout.

In order to achieve the above, instead of performing a pair of convolutions for each virtual speaker feed after decoding, a pair of convolutions is performed for each B-Format channel. For this to be possible, it is important to get the B-Format representation of the HRTFs. This is done by essentially encoding all of the HRIRs into the B-Format and then summing across each channel of each HRIR. The correct B-Format HRIR calculations are illustrated below in a channel-wise manner:

$$W(hrir) = 1/\sqrt{2} \times \sum_{k=1}^{N} \left(S_k(hrir) \right) \tag{3.4}$$

$$X(hrir) = \sum_{k=1}^{N} \left(cos(\theta_k)cos(\phi_k) \times S_k(hrir) \right) \tag{3.5}$$

$$Y(hrir) = \sum_{k=1}^{N} \left(sin(\theta_k)cos(\phi_k) \times S_k(hrir) \right) \tag{3.6}$$

$$Z(hrir) = \sum_{k=1}^{N} \left(sin(\phi_k) \times S_k(hrir) \right) \tag{3.7}$$

where N is the number of virtual loudspeakers, θ and ϕ are the corresponding azimuths and elevations respectively, and S is the input signal. These computations are essentially what the B-Format encoder does anyway. Therefore, in the implementation of the system, `setup` is used to recalculate the corresponding B-Format HRIRs, in order to save time during `render`. The FFT process then solely deals with the convolution between the source input's W, X, Y, and Z B-Format channels and the corresponding B-Format channels of the HRIRs. Conveniently, the final left or right channel outputs are simply obtained by summing all of the B-Format channels of the previous convolutions for the left or right sides respectively. As a result, the need for a B-Format decoding process is completely eliminated and the amount of convolutions is reduced from 16 to 8, hence more

than halving the compound processing time of the B-Format and FFT auxiliary tasks. The left and right channel computation is shown below:

$$L/R = \left(W(in) \otimes W(H_{L/R})\right) + \left(X(in) \otimes X(H_{L/R})\right) + \left(Y(in) \otimes Y(H_{L/R})\right) + \left(Z(in) \otimes Z(H_{L/R})\right) \tag{3.8}$$

where H is the HRIR and in is the input signal.

This optimisation not only allows the ambisonic renderer to process a cubic layout of speakers, but also allows the extension of the layout to any number of speakers, given that corresponding HRTFs for desired directions are available. In practice, the optimisation allows the system to run at a stable CPU load of between 65% and 80%, which reaches the goal for single string execution, thus providing enough leeway for potential additional features and multiple audio streams to be processed.

Chapter 4 — Evaluation of System

4.0 — Overview

For the purpose of checking that the ambisonic spatialiser works as expected, a test signal will be passed through the system with different settings and compared against the convolution of the test signal with the HRIRs. Since we could only test the ambisonic system with azimuth and elevation angles for which there is a HRIR present in the LISTEN dataset, it is worth automating this process rather than performing each sample collection by hand. To understand how this testing process is automated it is important to observe how the LISTEN dataset is ordered for each subject. For the purpose of this evaluation, the *1014* and *1013* datasets will be used. Each dataset contains 187 HRIRs in the form of .wav files. The dataset is grouped by azimuth rather than elevation. There are 24 azimuth positions in total. An azimuth increment of 15° separates each HRIR group, accounting for the full circular rotation around the subject. Elevation readings for each azimuth reading are also 15° apart, ranging from -45° (below subject) to 90° (directly above subject). Naturally, the full elevation range is not present for each azimuth group as this would create a few overlapping duplicates. For example, having the 90° elevation reading for each azimuth group would essentially create 23 duplicates of the same HRIR pair. The elevation readings are therefore distributed thus: The 0° azimuth group contains the full elevation range, every second multiple of 15° azimuth groups starting from 15° contain an elevation range of -45° to 45°, every second multiple of 30° azimuth groups starting from 30° contain an elevation range of -45° to 60°, and every multiple of 60° azimuth groups contain an elevation range of -45° to 75°. Knowing this configuration allows us to create an algorithm that correctly renames the HRIRs in such a way that `WAVHandler` could read them sequentially, i.e. from 0 to 186. In addition, it allows us to

write a C++ unit test that exports results as a .csv which could then be loaded into a Python Pandas DataFrame for analysis. The automation is further necessary in the analysis steps.

This evaluation solely comprises signal processing analysis, as the timeframe of this project does not allow for listening tests. The factors being considered in the evaluation are the error between the ambisonic system's response to a test signal and the HRIR convolutions with the test signal, as well as the duration and CPU usage of the system's processes in its non optimised implementation versus its optimised implementation. Two test signals are being used for measuring the ITDs and ILDs of the system: a unit impulse and a 512hz sinewave. For task duration measurements, the C++ `std::chrono` library is used to compute the time difference in microseconds or milliseconds of asynchronous task execution. In measuring ITD and ILD error, we have to consider the HRIR ITDs and ILDs as the reference point. In reality, there will certainly be some disparities between HRIRs in the left and right azimuth hemispheres as the LISTEN HRTF datasets have been acquired from human subjects. Naturally, neither the human ears nor head are perfectly symmetrical, and as a result opposite HRIRs will not feature diametrically opposite phases. In practice, these discrepancies cause certain ramifications involving the ambisonic renderer's behaviour as later discussed alongside the evaluation results. In future tests, it could be possible to mirror HRIRs in order to eradicate these ramifications while also doubling the performance of the system by halving the amount of convolutions necessary.

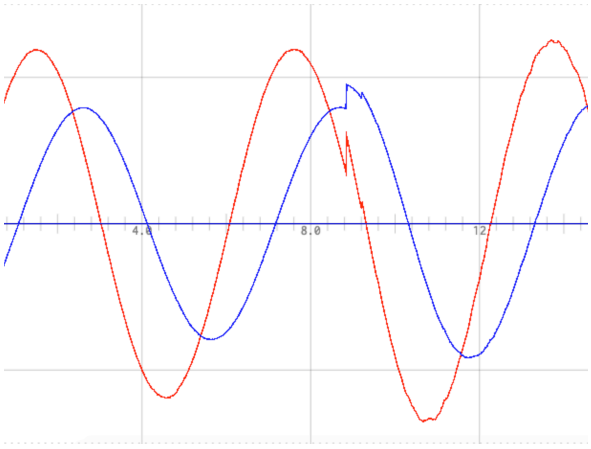
It is further worth noting that detection of ITDs is still an active area of research with many conflicting views. In this case, the ITDs have been detected by cross correlation of left and

right channels by using the SciPy library's Signal module. In a similar fashion, there is scarce information available about ILD detection and thus a definition of the method of its computation is provided in section 4.3 below. The ITD and ILD plots will be shown as a function of azimuth, with information of the HRTFs for which there exist opposing elevations (e.g. 45°/-45°) displayed in the same boxes for comparison. This is to observe the dissimilarities in ITD and ILD between opposing elevation angles. The remaining information for the four elevations which have no opposites i.e. 0°, 60°, 75°, and 90° are plotted on a final fourth pair of axes.

4.1 — Circular versus Linear Convolution comparison and sanity check

As mentioned above, when the convolution FFT buffer is too short to accommodate a linear convolution, the portion of the signal that does not fit in the buffer, wraps and adds onto itself from the beginning of the buffer. For input sine waves that correspond with the frequency bins of the FFT window, this is imperceptible, however, frequencies that are between the bins cause phase discontinuities in the output signal. This is perceptible as periodic clicks. This activity also shows clearly on the oscilloscope for the left and right output channels. Figure 4.1 shows this effect for an input sine wave of 164hz. In this

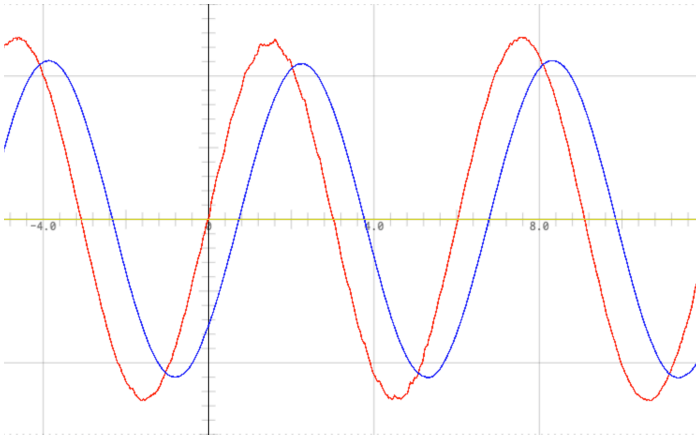
Figure 4.1 — Sinewave Discontinuity



case, the FFT window is 2048 buffers long, taking 2048 samples as input, and performing the convolution with a HRIR of 512 sample length. In this case, the bin frequency interval is approximately 10.77hz. Therefore, this effect would be imperceptible for an input sine wave frequency of 161.55hz, it being a multiple of 10.77hz.

To rectify this and achieve a linear convolution while decreasing latency, the FFT buffer has been resized to 1536 samples. This represents the sum of the length of the HRIR length (512 samples) and input stream window length (1024 samples). For this to work, both signals being convolved need to be the same length. As a result, the HRIR buffer was padded with 1024 samples and the input stream window buffer was padded with 512 samples, thus making both buffers the same length. Once the convolution takes place and the IFFT is performed, the output buffer will no longer be zero padded, as the convolution expands the signal into the full 1536 buffer length. The left and right channel output then rightly looks smooth for whichever frequency is passed as input. Figure 4.2 illustrates the corrected phase of the same 164hz sine wave. Please note that the spiky instance of the right channel shown in red is a result of the Bela's system noise when passing in test signals to `audioIn` from a web based generator. This oscilloscope reading was taken

Figure 4.2 — Correct Sinewave Phase



from a later part of the system's implementation where the input stream window buffer's length was 1024 rather than 2048 samples long. This applies more gain in the final output to artefacts arising from the Bela's ADC/DAC. Although the angle being tested is 90° in both cases, this second system implementation is a 3 dimensional ambisonic renderer and thus features slightly different ILDs as apparent from the figure below.

4.2 — Interaural Time Difference (ITD)

ITD is a significant cue for sound localisation containing low frequencies under 800hz. The method of measuring ITDs is done by cross-correlation between left and right channels (Stewart, 2010). Cross-correlation of HRIR signals has been done in Python with the help of the SciPy library's Signal module. The ITD was calculated by subtracting the positions of maximum correlation from the operation's resulting arrays and expressed in microseconds.

The ITDs have been detected by testing with two separate signals: a unit impulse and a 464.hz sine wave. This frequency ensures that exactly 20 sine cycles are included within the 1024 sample test signal buffer to prevent any potential discrepancies relating to phase detection during the analysis stage. To cross check results, two separate sets of HRTFs have been used, the IRCAM LISTEN 1013 and 1014 datasets. These shall henceforth be referred to as IRC1013 and IRC1014. The azimuth on the following plots is represented as an anticlockwise interval between 0° and 345° with 15° increments for all but three elevation categories, 60°, 75°, and 90°, as these have less azimuth points as previously discussed in section 4.0.

Figure 4.3 — ITD for HRTFs IRC1013 -- Unit Impulse

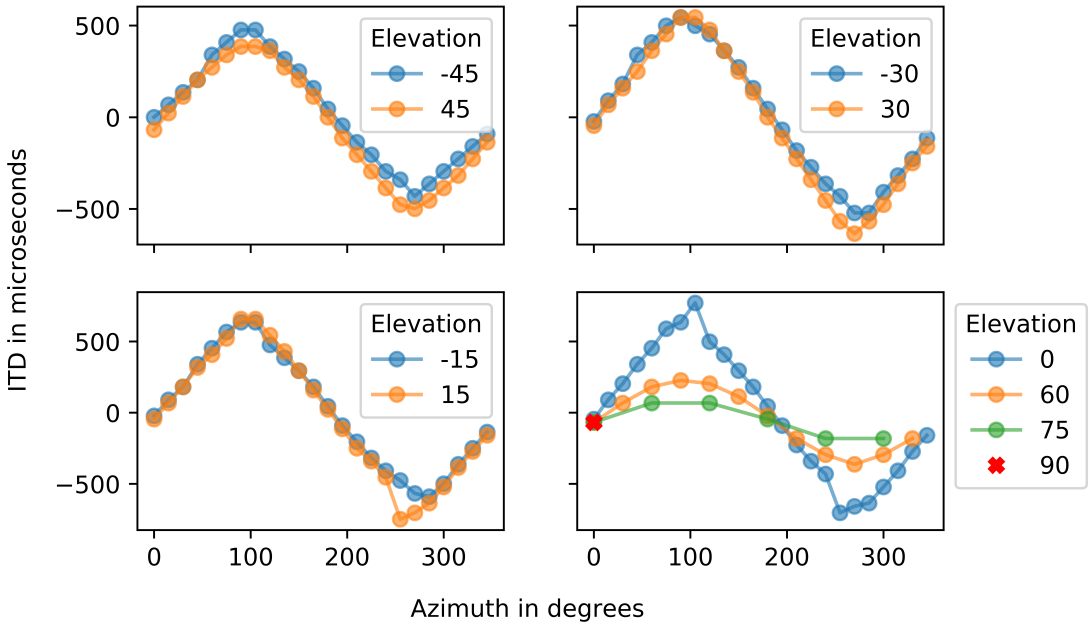
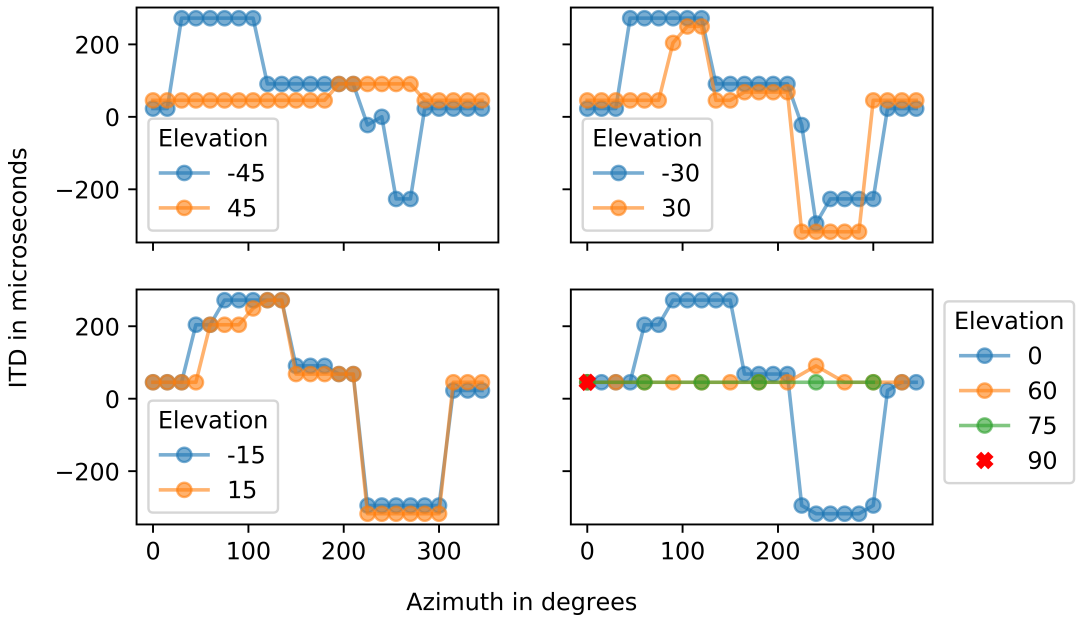
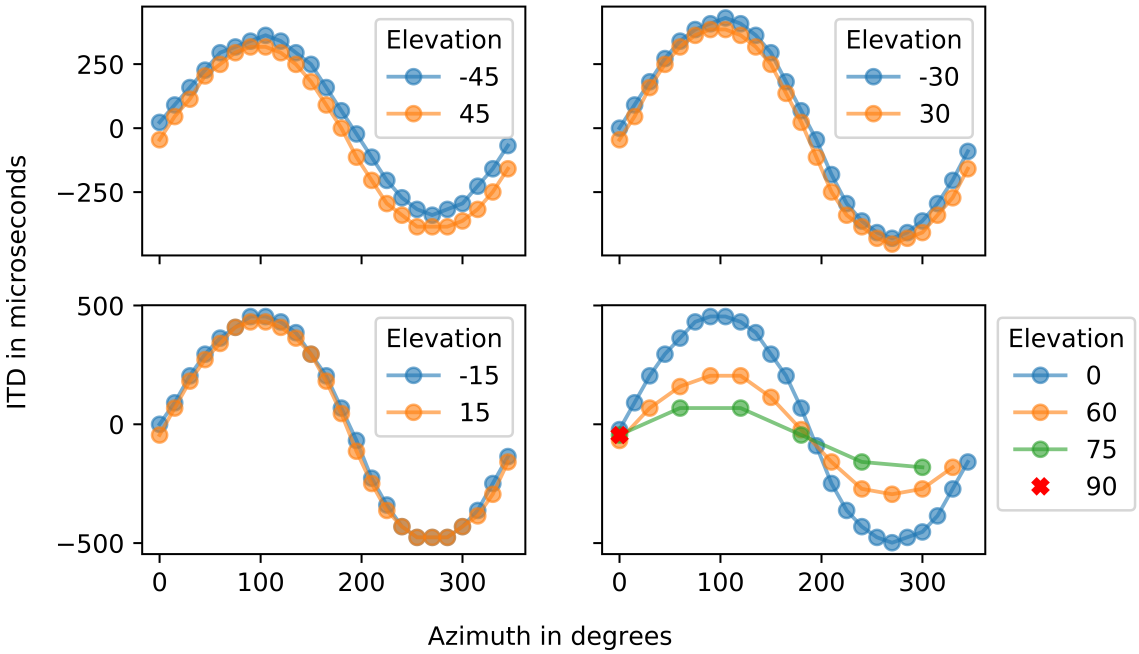


Figure 4.4 — ITD for Ambisonic System IRC1013 -- Unit Impulse



The most prominent feature of the results from the impulse response of the ambisonic system is that the ITD does not change with a smooth curve as we would expect. Instead, the result is very square in shape with the varying azimuth. By contrast, the HRTF impulse

Figure 4.5 — ITD for Ambisonic System IRC1013 -- 464.4hz sinewave



response gives us an expected rounded triangular plot. At first this seems like a significant error, but the ambisonic system reacts completely different to slower changing signals such as the sine wave test signal as can be seen in Figure 4.5. Upon also testing with a white noise signal, the ITD result is similarly square. Therefore, it seems that for quickly varying signals, the ITD is more or less quantised in 90° ranges i.e. 45°-135° for positions at the left, 135°-225° for positions at the back, 225°-315° for positions at the right, and 315°-45° for positions at the front. Please see Appendix X for the IRC1013 ITD unit impulse error. The ITD for the 464.4hz sine wave however follows the expected curve. Another error observed is that the ITD in the ambisonic system is just over half of that of the HRTF. Appendix X illustrates this error. When taking a look at individual ITDs for each of the 8 HRTFs (IRC1014) at each virtual speaker location in the cubic array, it becomes clear that not all of the left side HRTFs are symmetrical to the right. For example, the front, left, bottom HRTF is symmetrical with the front, right, bottom HRTF in terms of ITDs,

with readings at 340μs and -340μs respectively. However, the rear, left, bottom HRTF reads 499μs versus its rear, right, bottom counterpart at -567μs. This may well be the cause of the ITD error of the ambisonic system. Two solutions to this problem would be to either calculate symmetrical HRTFs for the left and right hemispheres, or to add extra speakers in order to smooth out the errors.

4.3 — Interaural Level Difference (ILD)

ILD plays a big role in the localisation of sound containing frequencies above 1600hz. These frequencies have wave lengths which are smaller than the dimensions of the head, thus rendering the phase of the sound less relevant for the detection of sound source direction. Since not much information is available regarding detection of ILDs, it is important to define exactly how this is done in this project. Pourmohammad et al. (2011) describe how the ILDs are detected in a continuous context. Extrapolating this method to the discrete context, ILDs can be calculated in the following way. Firstly, the left and right channel signals have to be shifted according to their sample offset previously determined in the ITD calculation until they are perfectly superimposed. This is necessary to ensure that an equal amount of samples will be considered for each of the signals. Next, the squares of each signal are computed in a sample-wise manner and their sum is taken to give the overall energy. Finally, the difference between the signals is expressed as a ratio in decibels. The below formula illustrates the process.

$$20 \times log_{10}\left(\frac{\sum_{k=0}^{N} L_k^2}{\sum_{k=0}^{N} R_k^2}\right) \tag{4.1}$$

where L and R are the appropriately shifted channel signals and N is the sample length of the channels. The computations in this case are performed with a Python script using the NumPy and Pandas libraries.

For the purpose of observing different behaviours, ILDs have also been detected using the sine wave and unit impulse. First taking a look at the impulse response of the ambisonic system alongside the impulse response of the HRTF, it becomes obvious that the ambisonic system tracks the ILD of the HRTFs, albeit with a flatter curve. Since the ambisonic system uses only eight HRTFs from IRC1013 at 45° angles in each off centre direction, the ILD curve cannot stretch to the 20dB+ point where it should be at a 90° angle. In other words, there is not enough ILD information for the ambisonic system to model the correct output at 90° angles. In the optimised version of the system (see section 3.9), this could simply be rectified by adding extra virtual speakers along the azimuth. In practice, to also support IMU tracking, more vertices should be added in the

Figure 4.6 — ILD for Ambisonic System IRC1013 -- Unit Impulse

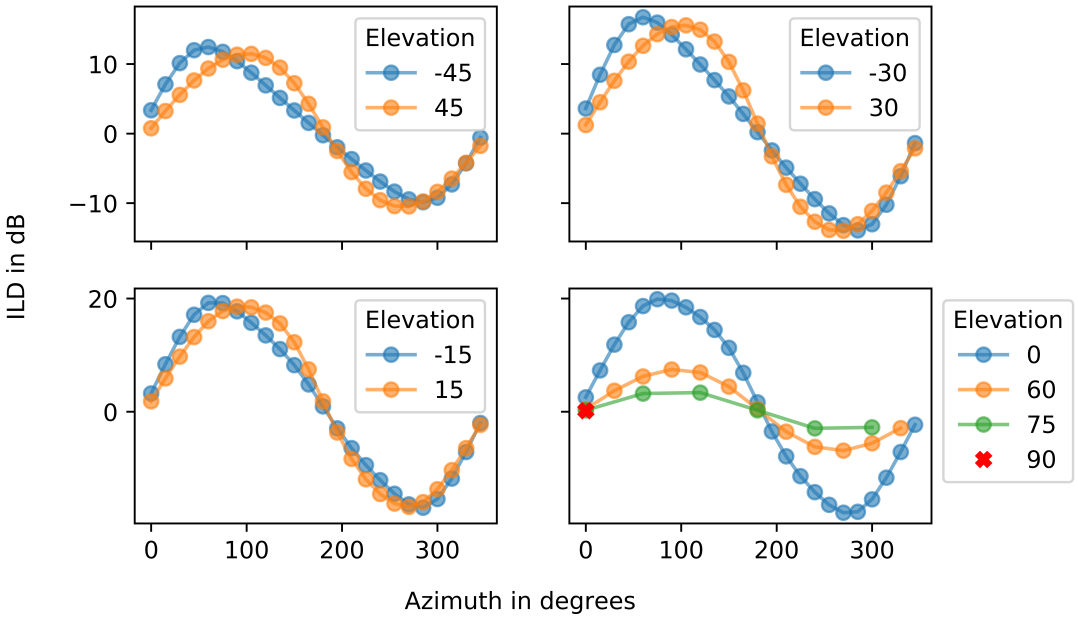
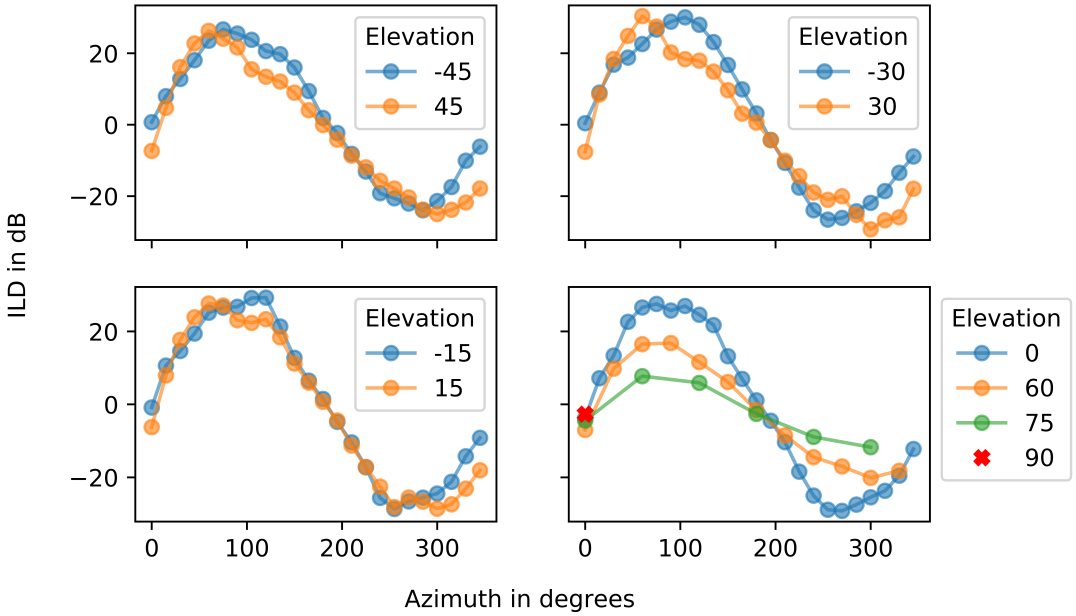


Figure 4.7 — ILD for HRTFs IRC1013 -- Unit Impulse



speaker array. A dodecahedral speaker array would improve both results for the test and the head tracked user experience.

While regarding HRTF datasets as the benchmark, it is important to test the error of the ambisonic system against it. Figure 4.8 illustrates the error between the IRC1013 dataset and ambisonic system. Again, the error has been classified by elevation. It becomes apparent that elevation readings further from the centre (-45°, 45°, 30°, -35°) have nearly double the amount of error compared to central elevation readings (between -15°, and 15°). In the case of a unit impulse test signal, this might have to do with the ambisonic system failing to apply the correct virtual speaker weighting when the source is placed near the location of a virtual speaker. This is confirmed when using the sine wave as the test signal. To approximate the correct ILD for highly varying waveforms, more virtual speakers would have to be added. Alternatively, a more symmetrical HRTF dataset could be used. Differing results have been found for the IRC1014 dataset pertaining to speaker

Figure 4.8 — ILD Error -- HRTF vs Ambisonic System IRC1013 -- Unit Impulse

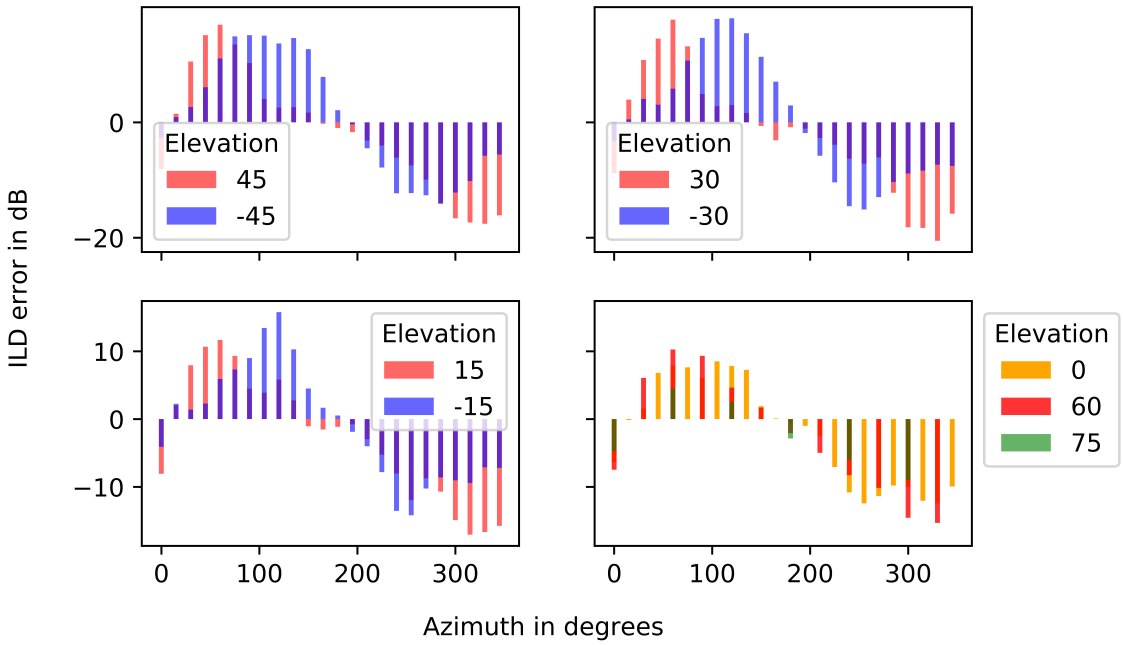
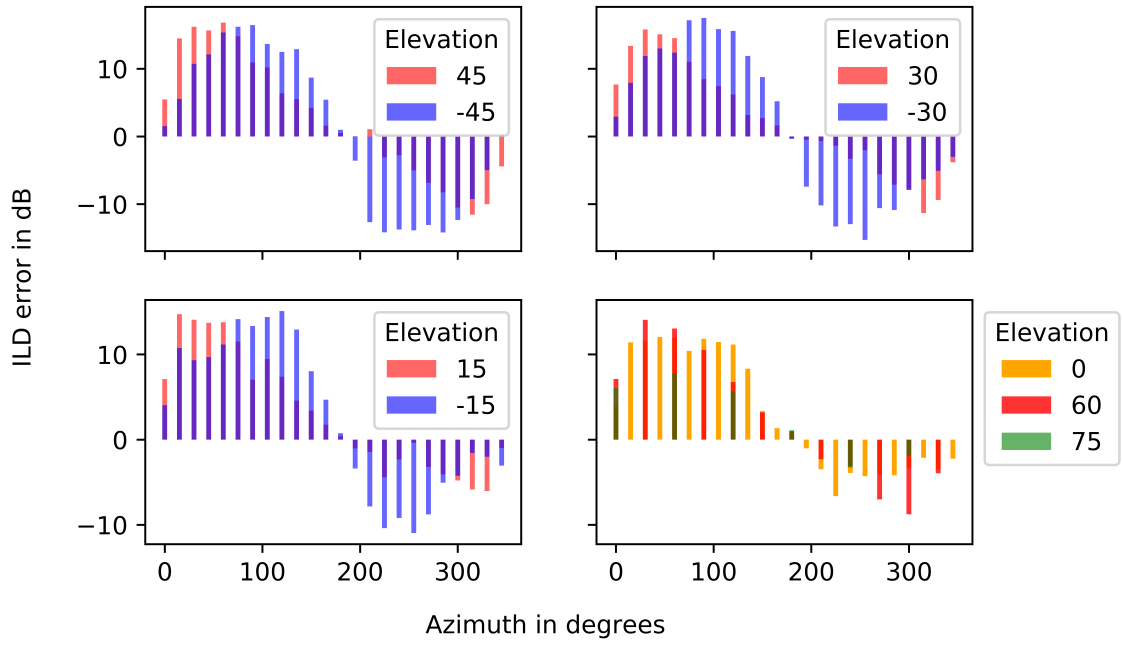


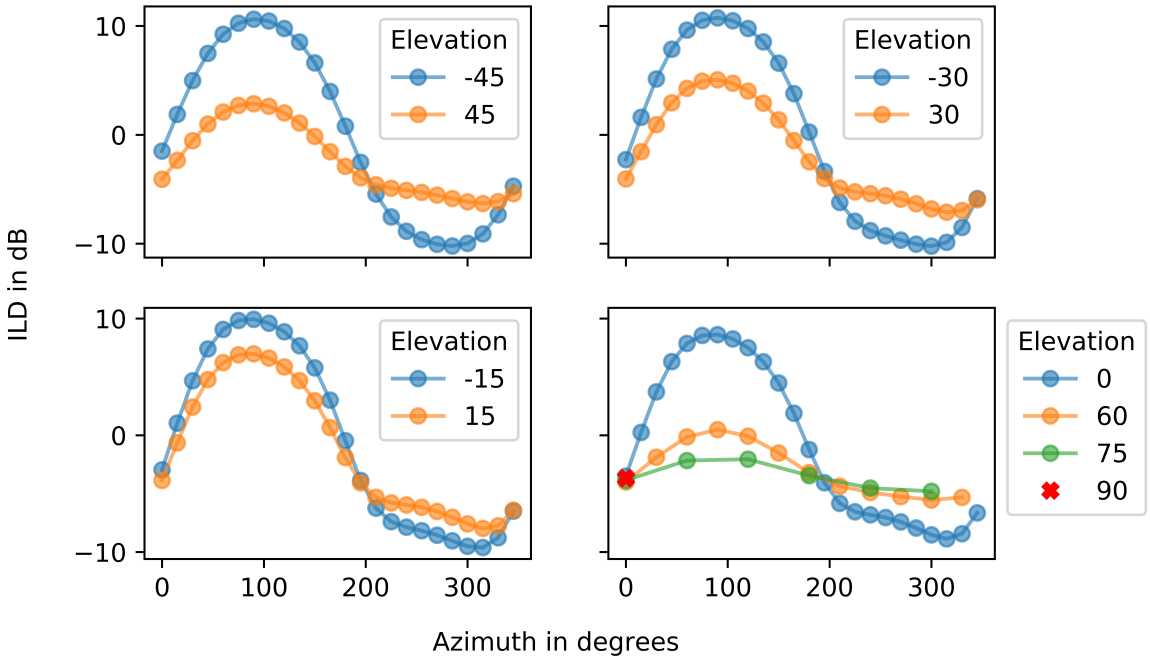
Figure 4.9 — ILD Error -- HRTF vs Ambisonic System IRC1014 -- Unit Impulse



elevation error. As can be seen in Figure 4.9, the error in IRC1014 is much less for the far from centre elevation readings.

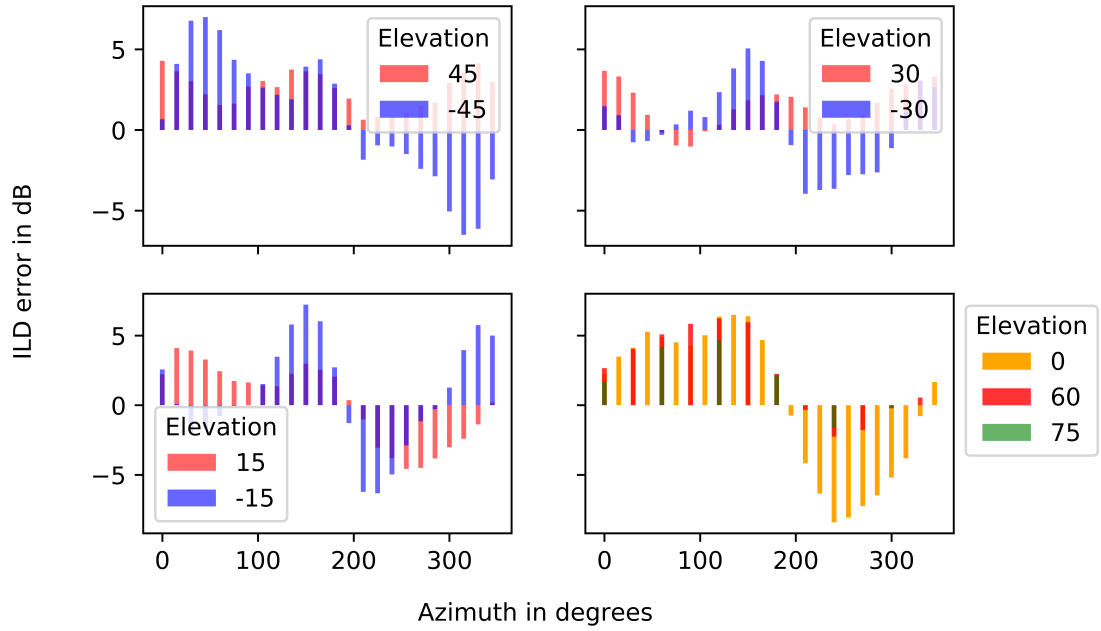
Finally looking at the response of the HRTFs and ambisonic system to the 464.4hz sine wave test signal for the IRC1013 dataset, we immediately become aware of the ramifications of the errors discussed above. The ambisonic system's ILD flattens out for the azimuths in the right hemisphere as seen in Figure 4.10. This indicates that the asymmetry present in HRTFs corresponding to virtual speakers in opposite azimuth hemispheres is increased, thus decreasing the ILD for azimuths in the range of 225° and 330° (most of the right hemisphere). The HRTF response to the sine wave looks a bit more angular as compared to the unit impulse response. This is might be a reflection of the fact that HRTF frequency response is not equal in the 20hz-20khz range. Since this dissertation deals with implementation matters from a computational point of view, it is beyond the scope of this discussion to delve into frequency response comparisons. Counterintuitively so, the error between HRTF and ambisonic system ILDs is less overall

Figure 4.10 — ILD for Ambisonic System IRC1013 -- 464.4hz sinewave



for the sine wave test signal as can be seen in Figure 4.11. The IRC1014 performs slightly better still. See Appendix X for more details. It is then safe to conclude that the IRC1014 dataset performs slightly better specifically in terms of ILD for the purpose of binauarlisation given a cubic virtual speaker array. Therefore, future research could address the extensive comparison across all of the IRCAM LISTEN datasets for different virtual speaker configurations.

Figure 4.11 — ILD Error -- HRTF vs Ambisonic System IRC1013 -- 464.4hz Sinewave



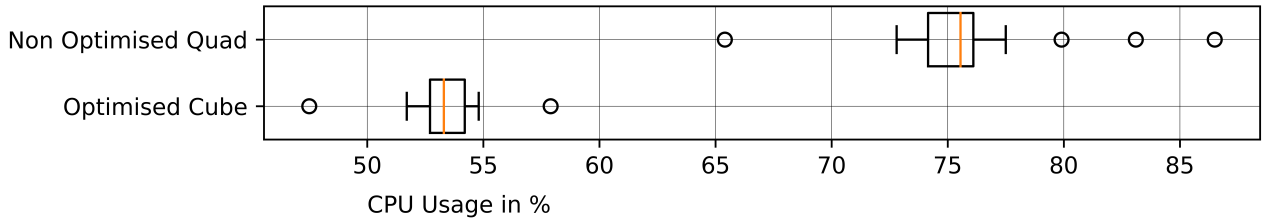
4.4 — CPU Usage and Task Duration

The CPU usage and task duration was measured for both the optimised (see Section 3.9) and non optimised system implementations. Unfortunately, the CPU usage for the non optimised system could not be read for the cubic layout as the console would not respond during execution. Therefore, it is assumed that CPU usage was consistently close to 100% for this scenario. An overall reading was acquired for the non optimised quad setup.

Due to some lack of code refactoring for the optimised version, quad layout information could not be acquired under the time constraints. Both optimised and non optimised implementations were run with the same FFT and target convolution length of 1536 samples and a hop size of 512 samples giving an effective FFT auxiliary task deadline of 11.6ms. Since the BFormat task processes chunks of 1024 samples at a time, its deadline interval is essentially double that of the FFT task. Hence, an FFT, IMU, and half of a BFormat auxiliary task together with any calls to `render` have about 11.6ms to execute for each `gHopSize` amount of input signal.

To begin, having a look at a box plot of 25 CPU readings for the optimised cube layout implementation versus the non optimised quad implementation reveals that the former not only outperforms the latter by approximately 22.4% on average, but also drives double the amount of virtual speakers and uses the CPU more constantly rather than in short bursts. Figure 4.12 indicates this by the lesser number of outliers and shorter candlestick in the former implementation (circles represent outliers in the set of readings) as compared to the latter.

Figure 4.12 — CPU Usage Optimised Cube vs. Non-Optimised Quad



Diving deeper into detail, Table 4.1 illustrates the execution time of individual auxiliary tasks for the different systems and configurations. The non optimised system takes about

Table 4.1 — Optimised vs Non-Optimised System Performance Comparison

	Optimised System	Non Optimised System	
	Cube array	Quad array	Cube array
FFT task mean duration in µs	5609.98	5056.74	21932.48
BFormat task mean duration in µs	217.1	5304.63	13509.49
FFT task mean CPU Usage %	38.172		
BFormat mean CPU Usage %	0.208		
Bela Audio task mean CPU Usage %	14.912		
System Total mean CPU Usage %	53.292	75.690625	

four times the amount of time to complete the FFT task although it is just rendering twice the amount of speakers. The BFormat task starts preparing the virtual speaker signals 1024 samples in advance before the FFT task processes them, allowing us to set the FFT task's thread priority higher. As the table shows, the BFormat task now takes approximately 13.5ms to complete, amounting to about 595 samples. This means that for every second time the FFT process is scheduled, it does not have time to complete. The Bela system tries to switch frantically between threads, making it look as if the FFT task has taken twice the expected amount of time. `std::chrono` also accounts for the time that the auxiliary task has spent idle.

Looking to the optimised system data, the most striking reading is the BFormat task, completing in just ~0.22ms. Previously, the most time consuming aspect of the BFormat task was its ambisonic decoder, which now is completely sidestepped as described in section 3.9. This leaves plenty of time for the FFT tasks to complete and some extra headroom of about 265 samples' duration between each FFT hop.

Finally, total system latency comes to about 1536 samples or 34.83ms plus any negligible latency from Bela's ADC and DAC from input to output. This is more than twice the target amount of time decided at the outset due to design decisions made to favour the quality of output signal rather than low latency. Fortunately, this can be immediately reduced by a third by shortening the input signal window length from 1024 to 512 samples, bringing the FFT and convolution length to a total of 1024 samples. In addition, shorter HRTFs of just 128 samples could be used, thus offering the possibility of theoretically reducing the latency further by a factor of 4.

Chapter 5 — Conclusion

The main end goal of creating movable sound sources as experienced by the user in relation to their head movements has been achieved both in two and three dimensions. The three dimension system required the optimisation discussed in chapter 3.9 to be made. With this came the benefits of extra CPU headroom allowing the addition of extra features such as multiple stream encoding support for spatialising multiple sources simultaneously. The BFormat encoding functionality of the libspatialaudio library has been successfully implemented on the Bela system achieving very fast execution with the use of auxiliary tasks. The 53.3% average system load proves that the cubic ambisonic renderer can leisurely run on the board. The next step would be to implement a second order two dimensional renderer, requiring two extra convolutions in an optimised implementation scenario. Extra virtual speakers could also be added in order to minimise ITD and ILD error while experimenting with symmetrical HRTFs.

Chapter 2 presents a summary of work that has been done thus far on spatialisation technology and its applications. Psychoacoustic elements are briefly considered and an explanation of HRTFs, ambisonic systems, VBAP systems, and traditional spatialisation and recording is provided. Chapter 3 documents the system implementation. This includes a description of the design decisions regarding buffer implementations, head tracking, file I/O, and virtual speaker placement as well as ambisonic optimisation implementation details. Chapter 4 goes on to evaluate the system in terms of ITD, ILD, and CPU usage and task duration, providing graphs and plots of the findings.

This proof of concept also shows that it is worth taking another look at making ambisonics technology more user friendly in the context of virtual and augmented reality. A logical step in the VR direction would be ambisonic binauralisation integration for game engines such as Unity, Unreal, or CryEngine. There is a lot of scope for using this technology in both artistic and functional applications. The processing power needed for rendering low latency audio is now cheaply available in consumer electronics without the need to acquire expensive, finicky hardware. Aside from VR/AR, it would be interesting to develop commercially viable loudspeaker products using ambisonics technology that overcome the obstacles of poor user experience and expensive hardware.

Future research could address the implementation of higher order ambisonic systems on other platforms featuring multicore processors such as the ARM Cortex A-76. This hardware has not yet made a debut into development platforms such as the BeagleBone Black at the time of writing. In future, there is also a need to perform extensive listening trials for this and upcoming implementations of the ambisonic system.

Figure Attributions

Fig 2.1 — By Iainf 23:51, 21 September 2007 (UTC) - self-made; based on Image:MS Stereo.png, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2792296

Fig 3.1 — Aristotel Digenis, *The Implementation of Ambisonics For Restoring Quadraphonic Recordings*, SAE Technology College Sydney & Middlesex University London, 2001-2003

Fig 3.2 — Algazi, Duda, and Thompson, *The CIPIC HRTF Database,* UC Davis, 2001

Bibliography

AKG/IRCAM, *Listen HRTF Database*, http://recherche.ircam.fr/equipes/salles/listen/index.html

Algazi, Duda, Thompson, *The CIPIC HRTF database*, IEEE Workshop on Applications of Signal Processing to Audio and Acoustics 2001

Allen, Baker-Finch, Doust, *Tetrahedral Arrangement Soundfield Microphone*, 2006, http://users.cecs.anu.edu.au/Salman.Durrani/_teaching/TB4.pdf, accessed 15/06/18

Baume, Churnside, *Upping the Auntie: A Broadcaster's Take on Ambisonics*, BBC R&D Publications, 2012

Begot et al., *Applying spatial audio to human interfaces: 25 years of nasa experience,* AES 40th International Conference, Tokyo, Japan, 2010 October 8–10 10

Bela Documentation, https://github.com/BelaPlatform/Bela/wiki, accessed 20/04/18

Binelli, Venturi, Amendola, Alberto, Farina, *Experimental Analysis of Spatial Properties of the Sound Field Inside a Car Employing a Spherical Microphone Array,* 2011

Blauert, Jens, *Spatial Hearing, The Pyschophysics of Human Sound Localization,* The MIT Press, 1996

The CIPIC HRTF Database, https://www.ece.ucdavis.edu/cipic/spatial-sound/hrtf-data/

Daniel, Jérôme, *Spatial Sound Encoding Including Near Field Effect: Introducing Distance Coding Filters and a Viable, New Ambisonic Format,* AES 23rd International Conference, Denmark, 2003

Elen, Richard, *Ambisonics for the New Millennium,* September 1998

Anders Ericsson, K & Krampe, Ralf & Tesch-Roemer, Clemens, *The Role of Deliberate Practice in the Acquisition of Expert Performance,* Psychological Review, 100. 363-406, 1993

Höldrich, Noisternig, & Musil, *A library for realtime 3D binaural sound reproduction in Pure Data (PD),* Institute of Electronic Music and Acoustics, University of Music and Dramatic Arts, Graz, Austria, 2005

Hollerweger, Florian, *An introduction to higher order ambisonic*, http://flo.mur.at/writings/HOA-intro.pdf accessed 20/04/18

Katz & Gaëtan, *Perceptually based [HRTF] database optimization,* Journal of the Acoustical Society of America 131, EL99 (2012)

Kearney & Doyle, *A HRTF Database for Virtual Loudspeaker Rendering,* Audio Engineering Society Convention Paper 9424, 139th Convention, New York USA, 2015.

Keller, Daniel, *Mid-Side Mic Recording Basics,* https://www.uaudio.com/blog/mid-side-mic-recording, accessed 15/06/18

Malham, Myatt, *3-D Sound Spatialisation using Ambisonic Techniques*, Computer Music Journal Vol. 19, No. 4 (1995), The MIT Press

Nettingsmeier, Dohrmann, *Preliminary Studies on Large-Scale Higher-Order Ambisonic Sound Reinforcement Systems,* Ambisonics Symposium, Lexington KY, 2011

Oppenheim & Schafer, *Discrete-Time Signal Processing, Third Edition,* Pearson 2013

Penha, Rui, *Distance Encoding in Ambisonics Using Three Angular Coordinates,* SMC Conference, 2008

Pourmohammad and Ahadi, "TDE-ILD-HRTF-Based 3D entire-space sound source localization using only three microphones and source counting," *Proceedings of the 2011 International Conference on Electrical Engineering and Informatics*, Bandung, 2011

Pulkki, Ville, *Virtual Sound Source Positioning Using Vector Base Amplitude Panning*, Audio Engineering Society, Inc., 1997

Videolabs, *Ambisonic encoding / decoding and binauralisation library*, https://github.com/videolabs/libspatialaudio accessed 20/04/18

https://wiki.xiph.org/Ambisonics, *G-Format*, accessed 15/06/18

3dti_AudioToolkit, University of Malaga and Imperial College London, https://github.com/3DTune-In/3dti_AudioToolkit