FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Spatial sound for immersive video
Student:	Ivan Desiatov
Supervisor:	Ing. Jan Buriánek
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Computer Graphics
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Interactive immersive virtual reality experiences are the realm of game engines, which have been evolving rapidly to match the industry's requirements. In the case of non-interactive experiences, the extensive functionality of more traditional creative software can be used.
However, most consumer-oriented 3D and DAW software, currently doesn't have optimised workflows for creation of VR/AR content, especially it's audio aspect. The aim of this bachelor's thesis is to design and implement a tool to optimise the sound design and mixing workflow of creating three dimensional scene based audio for immersive videoexperiences.
Tasks:

1. Analyse available solutions.
2. Propose a solution providing an improved workflow, while utilising the capabilities of existing creative software.
3. Implement software enabling the proposed workflow.
4. Test and document the resulting software.
5. Evaluate the results and discuss possible extensions/improvements.

Electronically approved by Ing. Radek Richtr, Ph.D. on 11 February 2022 in Prague.

Bachelor's thesis

SPATIAL SOUND FOR IMMERSIVE VIDEO

Ivan Desiatov

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Jan Buriánek
February 16, 2023

Contents

List of Figures

List of Tables

List of code listings

I would like to thank my thesis mentor, Jan Buriánek, for providing advice and valuable industry insight, as well as the people who have supported me throughout the process of putting this together; you know who you are.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for non-profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on February 16, 2023 .....

Abstrakt

Práce představuje nejvýznamnější moderní technologie ve sféře prostorového audia a uvádí výhody, díky kterým ambisonie začíná být standardním formátem zvuku pro 360° video. Práce uvádí čtenáře do teoretických základů ambisonie a popisuje několik důležitých dekódovacích technik. Následně je představeno softwarové řešení umožňující automatické prostorové panorámování zvuku v DAW na základě 3D pozicí objektů (vůči aktivní kameře) ve scéně v Blenderu. Implementace obsahuje dva pluginy - plugin pro Blender, umožňující přístup k současnému stavu 3D scény, a VST plugin, který využívá tyto informace o 3D scéně pro ambisonické panorámování zvuku. Panoramovácí směrové vektory jsou aktualizovaný v reálném čase pomoci meziprocesové komunikace. Umělcům a inženýrům pracujícím na projektu to pak dovoluje mít před sebou náhled animace a současně provádět změny v audiu, a obráceně - slyšet, jak změny v poloze objektů ovlivňují zvuk. Představené řešení je nejužitečnější pro produkci prostorového zvuku pro 360° 3D animace, kde může výrazně snížit počet kanálů vyžadujících ruční panoramování. Najde ale využití i v neanimovaných projektech, kde může sloužit k vizualizaci pohybu zdrojů zvuku, a umožní využití dostupných v Blenderu pokročilých nástrojů pro 3D animaci.

Klíčová slova ambisonie, prostorový zvuk, blender, 360 stupňové video, virtualní realita, digital audio workstation, VST, plugin, JUCE, meziprocesová komunikace

Abstract

This thesis provides an overview of modern immersive audio technologies and outlines why ambisonics is becoming the industry-standard spatial audio format for 360° video. An introduction to ambisonics theory is provided, and several important decoding techniques are described. A software solution is then presented, which implements automatic spatial panning of sounds in a DAW based on the 3D positions of objects (relative to the active camera) in the Blender scene. The implementation consists of two plugins - a Blender plugin to access the 3D scene data, and a VST ambisonic panner plugin that utilises said scene data to calculate the panning direction. Panning direction vectors are updated in real time using interprocess communication, allowing artists to preview the animation while simultaneously making adjustments to the audio, and vice versa. The presented solution is especially useful in producing spatial audio for 360° 3D animations, where it can help reduce the number of audio sources that have to be panned manually. It can however find a place in non-animated projects as well, where it can be used to visualise the movement of sound sources, and would allow to use Blender's advanced 3D animation toolset for animating sound source positions.

Keywords ambisonics, spatial audio, blender, 360 degree video, virtual reality, digital audio workstation, VST, plugin, JUCE, inter-process communication

List of abbreviations

API	Application Programming Interface
AR	Augmented Reality
CBA	Channel-based Audio
DAW	Digital Audio Workstation
FIFO	First In, First Out
FFT	Fast Fourier Transform
FPS	First-Person Shooter
FPS	Frames per Second
GUI	Graphical User Interface
HCI	Human-Computer Interface
HRIR	Head-Related Impulse Response
HRTF	Head-Related Transfer Function
MR	Mixed Reality
M/S Stereo	Mid/Side Stereo
OBA	Object-based Audio
SBA	Scene-based Audio
SDK	Software Development Kit
VR	Virtual Reality
VST	Virtual Studio Technology
UI	User Interface
UX	User Experience

Introduction

Virtual reality and immersive video

Virtual reality, or VR, has become increasingly popular in recent years. VR gaming specifically has been steadily rising to mainstream attention. But other forms of immersive content, such as 360° video are also gaining adoption - YouTube, for example, already provides support for 360° video (and audio) on their platform.

The goal of VR experiences is to fully immerse the user in a virtual environment, striving to erase the gap between the virtual and the real world. This is achieved using advanced human-computer interfaces. Most prominently - VR headsets, that allow the user to see into the virtual world, and various motion tracking systems, enabling the user to interact with it. Like any new form of human-computer interaction, VR has found applications in many industries, but the application most closely related to the goal of this thesis, and, incidentally, the one where most adoption has been seen so far, is entertainment.

While fully interactive experiences, such as games, are prevalent, the applications of VR are not limited to those. Immersive, or 360°, video [1] is a mostly noninteractive form of VR content that benefits from the advantages of VR, but is closer to traditional film and television than other forms of immersive experiences such as games.

Creation of interactive VR experiences shares more similarity with game development than traditional video production. 360° video, on the other hand, is largely linear, allowing to use creative software initially developed without VR content in mind. Compared to specialized immersive content creation programs, such software is often more powerful, costs less, and benefits from higher availability of learning resources regarding it's usage. These factors would make it a great choice for creators, if not for the lacklustre VR content support. Thankfully, many of these tools are extensible via plugins, allowing developers to fill in the gaps.

The role of sound in immersive experiences

Sound is a key ingredient in achieving immersion in a virtual environment. Although it's importance is easily outshined by the importance of sight, hearing is an essential part of human perception and getting audio wrong in a VR context can easily ruin the whole experience. While simulating the perception of touch - another vital sense - is a very difficult task, doing the same for hearing is fortunately already within our reach (although, as with real-time graphics, physically accurate techniques are still too computationally expensive). To do so, as with video, the extension of audio into the third dimension is required. Because of that, creating audio for an immersive experience is quite different to normal audio production. In addition to the usual recording, sound design and mixing work, the spatial position of sounds must be defined and animated.

[1] Throughout this thesis I will prefer using "360° video" instead of "immersive video", because, this being such a new concept, there is some inconsistency in terminology between some sources.

The goal of this thesis

It might seem that to utilise the power of modern DAWs (digital audio workstations) for spatial audio production, the DAW itself has to provide a user interface for spatial panning[2] , be it natively or via plugins, but a different approach may be taken. Blender - a free and open source 3D software - already provides all the required tools and an established workflow for 3D animation. The challenge lies in finding a way to utilise it's capabilities to control the spatial position of sounds while still using the DAW for sound design and mixing. This can be achieved by extending both Blender and the DAW via plugins, allowing the user to use Blender's 3D animation workflow for spatial audio. An existing 3D animation can also be used, providing significant time savings for animated 360° video productions. This thesis aims to design and implement such a solution.

[2]Spatial panning refers to the act of defining the 3D position of a sound source.

Chapter 1

Spatial audio

This chapter serves as an introduction to spatial audio. Different approaches to representing spatial audio information are presented, and their suitability for 360° video applications is discussed. A more in-depth section on ambisonics follows, accompanied by a brief foray into the mechanisms behind human spatial hearing.

1.1 Spatial audio representations

There are three main approaches to representing spatial audio information - channel-based audio, object-based audio and scene-based audio. [1][2][3] Each with their own advantages and disadvantages for specific applications. Let me briefly introduce each paradigm.

1.1.1 Channel-based audio

CBA content consists of a set of audio signals, each of them intended to feed a loudspeaker at a specific position relative to the listener. [2] It is extensively used for 3D sound in broadcasting and film [1][2], but it's core design is flawed in assuming a fixed speaker layout. Reproducing channel based content on a speaker layout with a number of channels that is different to the one it was produced for requires using sophisticated downmixing or upmixing algorithms which may result in loss of quality and spatial resolution. [2][4] A study performed by researchers at the Delft University of Technology shows that Dolby Surround, which employs a downmixing technique [5] to playback surround audio on stereo headphones, showed significantly lower perceived "overall presence" than even traditional stereo (not utilising HRTFs). [6][4] Furthermore, channel-based audio doesn't account for the possibility of speakers being positioned differently than the intended layout (an occurrence common in the real world), which will inevitably alter the directionality of sounds and make the experience differ from the one intended by the content creator(s).

While CBA has been used extensively for delivering surround audio content in film and video games, it has a lot of constraints stemming from the assumption of a specific speaker count and layout made during the production stage. The most important constraint for VR applications is the difficulty in rotating the sound scene as a whole, which is required to keep the sound positions anchored in space while the listener's head may rotate. While it can be done, the results are subpar at best, and result in "*a sound that goes in and out of focus as the channels move into and out of particular speakers*" [7, p. 45].

The other two spatial audio paradigms - OBA and SBA, don't share these flaws of CBA. [1][8] Instead of relying on a specific speaker layout, they describe the audio scene independently of the playback hardware (the end user's speakers or headphones), allowing playback on arbitrary

loudspeaker setups while preserving spatial information and general audio quality better than CBA transformed using downmixing or upmixing techniques. [8][3]

1.1.2 Object-based audio

OBA, which originated in game audio [9], describes audio in a more general way. The content is represented as a virtual sound scene - a collection of sound sources called objects. [10] An audio feed, as well as metadata, describing how the sound should be rendered, is assigned to each object. [10][9] This representation allows OBA content to be rendered on the end user's hardware, accounting for their specific (possibly non-standard [11]) speaker configuration, as well as in the form of binaural audio, which will be described in more detail later. [8][9]

OBA comes with it's own disadvantages - the main of which being the direct correlation between the number of individual sound objects, the bandwidth required to transmit the OBA content, and decoding complexity. [1] There are however proposed solutions to lower the required bandwidth by employing object grouping techniques. [12]

Object-based audio seems to be much better suited for VR applications than CBA. It brings the ability to differentiate between individual audio objects and adjust their position relative to each other which is indispensable for VR games. While the interactivity potential OBA brings is enticing, it is not as beneficial for noninteractive content, and the increased bandwidth requirements are undesirable, especially for 360° video streaming scenarios.

1.1.3 Scene-based audio

While SBA still uses the concept of channels, the channels play a very different role - instead of associating each channel with a specific speaker, as in CBA, or with a specific sound object, as in OBA, the channels in SBA are combined to equally capture the sound coming from all directions. [7][3][9]

Ambisonics is the underlying mathematical framework that makes SBA possible. It is a surprisingly old approach initially described by Michael A. Gerzon in 1972 in his article "Periphony: With-Height Sound Reproduction" [13]. Ambisonic content can easily be transformed to allow playback on any speaker arrangement, it is computationally viable to decode, encode, and apply simple processing, such as rotating the sound scene, in real time, and it's flexible in allowing to lower the required bandwidth at the cost of decreased spatial resolution. [7][14] Ambisonics is in fact so powerful, and such a good fit for 360° video, that it gets a dedicated section.

1.2 Ambisonics

1.2.1 M/S Stereo

To understand how ambisonics make all of this possible, let's start with a simpler concept - stereo audio, specifically it's Mid/Side representation. When the term stereo is mentioned, most readers will probably associate it with X/Y encoding where the audio is encoded using a left and right channel. This representation is natural for humans, because it closely resembles the way our hearing works, but mid/side (commonly shortened to M/S) encoding ([15]) brings significant advantages compared to X/Y. It is very easily downmixed to mono, and gives the ability to widen or narrow the stereo image. [7]

It's easiest to explain how M/S encoding works on a recording example. To record M/S stereo one needs two microphones - one with a cardioid or omnidirectional pattern, and a second microphone with a figure-eight pattern, positioned at a 90° angle relative to the first microphone. The first microphone captures sound from a wide area in front of the microphone or from all

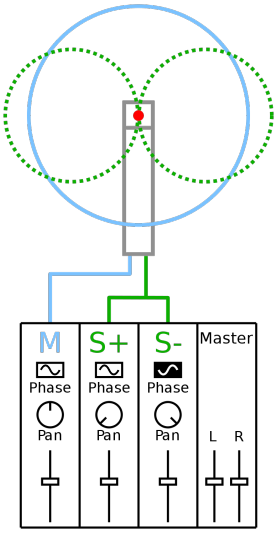

Figure 1.1 M/S stereo recording technique. Image sourced from [18].

directions, this microphone will provide the mid channel signal. A figure-eight pattern microphone captures sound predominantly in front and behind it, so when rotated 90 degrees it will provide the necessary information for our side channel. To get a stereo field, the side channel is duplicated and one of the copies is phase shifted 180°. Let's refer to the phase shifted version as S- and the original side channel as S+. The left channel can then be calculated as $L = M + S+$ and the right as $R = M + S-$. In practice this means hard-panning [1] the $S+$ channel left, and hard-panning the S- channel right. Figure 1.1 visualises the microphone setup and channel routing. To get a mono version, one can just use the mid channel, and to widen or narrow the stereo image, the relative volume of the side channel can be adjusted. [16][7][17]

1.2.2 From M/S stereo to B-format ambisonics

The B-format is the minimal representation of ambisonics [7][p. 42]. It consists of four channels - W, X, Y and Z. The role of the W channel is analogous to the M channel in M/S encoding - it captures sound coming from all directions equally. The other three channels - X, Y and Z - are similar to the S channel in M/S recording, but each one captures sound in a different plane, the Y channel - to the left and right of the listener, the X channel - in front and behind, and the Z channel - above and below. By distributing the source audio between the four channels, a sound can be positioned anywhere on a sphere around the listener. [7][13][3]

1.2.3 Higher Order Ambisonics

After the B-format was initially described, the mathematical framework for ambisonics has been further extended, probably most notably by Jérôme Daniel when he described higher order ambisonics (HOA) in his Ph.D. thesis [19]. HOA extends the B-format by adding additional channels to increase the spatial resolution. After the introduction of HOA, Gerzon's work was referred to as First Order Ambisonics. To represent ambisonics of order N, $(N + 1)^2$ channels

[1] In music production and audio mixing, hard-panning a channel means routing it exclusively to a single speaker. This approach was popular when stereo was first introduced, and can be heard, for example, on many The Beatles records.

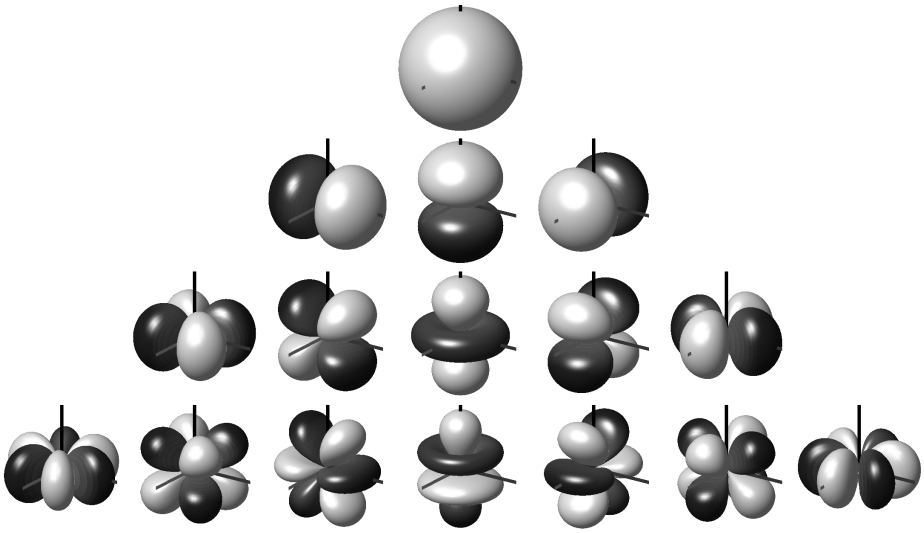

Figure 1.2 Balloon plots of spherical harmonics up to third order. The distance from the origin is defined by the absolute value of the given spherical harmonic. Dark portions represent negative values. (Order increasing towards the bottom of the picture; degree from left to right, with degree 0 in the middle.) Image sourced from [22].

are required. Accuracy of the HOA representation of the original signal as well as localisation accuracy (spatial resolution) increase with increasing HOA order N. [1][7]

The next paragraphs will give a short summary of the mathematical apparatus behind higher order ambisonics.

Functions defined on the surface of a sphere The following equation ([20]) defines a spherical surface of unit radius in the Cartesian coordinate system:

$$S^2 = \{\boldsymbol{x} \in \mathbb{R}^3 : \|\boldsymbol{x}\| = 1\}, \tag{1.1}$$

where $\mathbb{R}^3$ is the three-dimensional space of real numbers, $\boldsymbol{x}$ represents a vector in geometric notation, and $\|\boldsymbol{x}\|$ denotes the Euclidean norm (length) of the vector $\boldsymbol{x}$. Any point on the surface of a sphere can be defined using spherical coordinates (r, θ, ϕ); r denotes the radius of the sphere, θ is the elevation angle, and ϕ is the azimuth angle. Spherical functions are defined on the surface of a unit sphere (with $r = 1$), and accept the elevation and azimuth angles θ and ϕ as inputs. [20] An example spherical function can be seen in the following equation:

$$f(\theta, \phi) = sin(\theta)cos^2(\phi), \ (\theta, \phi) \in S^2. \tag{1.2}$$

Spherical harmonics Spherical harmonics are a special set of spherical functions. Each spherical harmonic function $Y_n^m(\theta, \phi)$ is denoted by it's order $n \in \mathbb{N}$ and it's degree $m \in \mathbb{Z}$. A graphical representation can be seen in figure 1.2. A linear combination of the spherical harmonic functions can be used to approximate any square-integrable spherical function. If the order of the spherical harmonics used is not limited, the resulting infinite series will accurately represent the original function. [21] Of course, an infinite series is not applicable for practical computations. Instead, a linear combination of spherical harmonics up to a specific order N is used. The accuracy of such an approximation increases with the maximum order N.

An ambisonics representation of a sound field is essentially "a set of signals that would be obtained by microphones with specific directionality patterns" ([23]), and the spherical harmonic

functions define the directionality patterns of these theoretical microphones. The values in the channels of an ambisonics-encoded audio signal are the coefficients used in the linear combination of spherical harmonics at a given time, and the ambisonic order N is equal to the maximum order of spherical harmonics used. Returning to the M/S stereo example, it can be seen that the visualisations of first order spherical harmonics in figure 1.2 closely resemble the figure-eight pattern of the microphone used for the side channel. To gain a better intuitive understanding of ambisonic audio, higher order spherical harmonics can be thought of as representing increasingly complex microphone pickup patterns, and increasing the ambisonic order - as adding more microphones, with narrower patterns, thus increasing the spatial resolution of the ambisonics representation.

Hopefully the brief summary presented above will provide the reader with a better intuitive understanding of how ambisonics make SBA a reality. A more in-depth overview can be found in [17] and [20].

1.2.4 Applying ambisonics in practice

Ambisonics is a great theoretical framework for representing a sound scene, but, importantly, it is also applicable in practice and brings many benefits during both the production and delivery stages. Listed below, in no particular order, are some of the practical benefits ambisonics bring to the table:

- Spatial audio content can be produced once, and decoded on the consumer's hardware. Because the decoding is performed on the end user's hardware, information about the playback environment, such as the position and type of speakers, can be provided to the decoder. With this information, a much better listening experience can be achieved than would otherwise be possible. [1][7][14]
- Ambisonics encoding, decoding, as well as rotation of the sound field can be performed very fast[2] on modern computers as most of the computation consists of matrix operations. Furthermore, when hardware resources are limited, computational power can be saved by rendering lower order ambisonics at the cost of fidelity - the extra HOA channels can just be discarded. [1]
- Ambisonics content may sound even better years after it was initially produced. Because ambisonics is such a general way to represent sound, in the future, advanced decoders can provide better sounding and more realistic mixes than what is possible now. [7]
- There are also multiple advantages to using Ambisonics as an intermediate spatial audio format in game engines. Submixing is a very useful audio production technique. Essentially, it comes down to applying processing (audio effects) to multiple sounds at the same time. However, when submixing object based audio, the directionality of individual sounds can't be preserved - the whole submix has to become a single sound object. By submixing multiple sound objects into an ambisonics representation, the directionality of individual sounds can be preserved, while still allowing to process the submix as a whole. Utilising ambisonic submixes or even converting the whole mix to ambisonics before it gets rendered for playback can also bring performance advantages compared to processing each sound object individually. [28][29]

With these key advantages in mind it becomes obvious that ambisonics is a great fit for 360° video. It should be no surprise that it is already being adopted for such applications. One example is the ability to upload 360° video with ambisonics audio to YouTube. [30] A big platform like YouTube supporting ambisonics and 360° video is a milestone in the adoption of

[2]Especially if using SIMD compiler intrinsics ([24][25][26]) and other techniques employed by good mathematical libraries (e.g. [27]).

these technologies. It seems highly probable that user-created immersive content will increase in popularity in the coming years, as the availability of devices capable of 360° video and audio recording increases.

1.3 From data to sound

A detailed description of the inner workings of ambisonic decoding techniques is outside the scope of this thesis, but it wouldn't be complete without a brief overview of the existing approaches and the possibilities they bring in practice. Ambisonics decoding is largely based on how humans perceive sound, so let's take a look at that in a bit more detail.

1.3.1 Spatial hearing

Human hearing is fascinating - with just two points in space at which the sound field is sampled - our ears, we can tell the direction and distance to the sound source, if it's moving or not (and, roughly, at what speed), and many other things about the sound itself, as well as the surrounding environment. In doing that, we rely on numerous acoustical cues that help us in locating the sound source, but also, to a great extent, on our knowledge of the world - making assumptions based on previous experience. [31][32][7]

Imagine you hear the sound of a helicopter flying in the distance, somewhere outside your field of view. Based on the sound alone, you immediately assume numerous things about your surroundings:

- You can tell that there's a helicopter flying because you know what a helicopter sounds like.
- You can roughly estimate how far away it is, because you know how loud a helicopter is, and based on the ratio of environmental reflections (reverberations) to the pure sound - again, relying on previous experience of hearing a helicopter at a known distance.
- If you are in a city, surrounded by tall buildings, depending on the distance to the helicopter, your ability to localise the sound may be severely impaired because the sound reaching your ears has been reflected multiple times by nearby buildings.
- If you are in a field, on the other hand, you should be able to estimate where it is quite precisely.

Hopefully this shows how complex our perception of sound is, and puts to light some of the many factors - external and internal, that can affect it. I must note that the above example is merely illustratory, not a description of a specific experiment. It is however inspired by research in the field of auditory localisation. A good overview of such research can be found in [31] and [33].

Let's now explore the problem of sound localisation from a more theoretical point of view. It has long been suspected that our ability to detect the direction of sounds stems from perceiving differences in the sound reaching each ear. English physicist and baron, Lord Rayleigh, has determined through ingenious experiments, often involving tuning forks, that the perception of direction of sound is affected by differences in amplitude and phase of the sound waves between each ear. The article ([34]) describing his findings is dated April 3, 1876. His findings have later been confirmed with modern experiments. Nowadays we know of three main auditory cues instrumental in horizontal plane sound localisation - interaural loudness difference, interaural time difference, and interaural phase difference. While these are not the only factors at play, and it's impossible to say with any certainty that there aren't any we are still not aware of, they are quite easily simulated, play a major role in auditory spatial perception, and thus are very useful in spatial audio decoding. [31][35][7]

- Interaural intensity difference (IID) or interaural loudness difference (ILD) is the loudness difference between the sound reaching each ear. A sound coming predominantly from one side, will be perceived as louder by the ear closest to it, as the head will partially block the sound before it reaches the other ear.

- Interaural time difference (ITD) is the difference between when a sound reaches each ear. A sound directly in front of the listener would reach each ear at roughly the same time, while a sound to the listener's left would reach the left ear slightly before the right ear. The human brain effortlessly notices these timing differences and interprets them as information about the position of a sound source on a horizontal plane.

- Interaural phase difference (IPD) is affected by the ITD and the frequencies a sound is composed of. A sine wave with a frequency of 1000 Hz that reaches one ear 0.5 ms before the other, will result in an IPD of 180°. Humans can detect IPDs as small as 3 degrees, but only for frequencies below 1000 to 1300 Hz. [7]

Interestingly, the effectiveness of these cues varies depending on the sound's frequency composition. At lower frequencies sound localisation mostly relies on ITDs and IPDs, while IIDs are important at higher frequencies. This can be explained by the fact that shorter (high frequency) waves are easily blocked by the listener's head, while longer ones can reach the other ear with minimal amplitude degradation, thus making IID cues inefficient at lower frequencies. This again has been described by Lord Rayleigh in the 19th century, and has since been termed the "duplex" theory. [31][34]

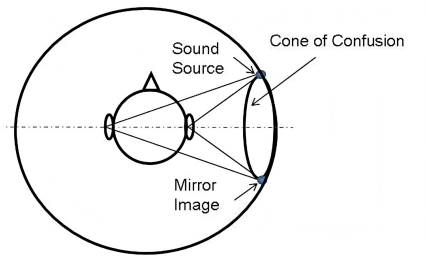

Figure 1.3 Cone of confusion. Image sourced from [36].

The cone of confusion is another important concept in sound localisation research. As illustrated in figure 1.3, sounds located at specific points inside the so-called cone of confusion result in identical interaural timing, phase, and level differences. This makes interaural differences ineffective in discerning sounds in front and behind the listener, as well as above and below. To alleviate this ambiguity, human listeners tend to move their head when trying to locate a sound. Turning the head helps mitigate front/back confusion, and tilting the head aids vertical sound localisation. For the same reason, many animals posses the ability to individually move each ear, removing the need for whole-head movement. [7] Vertical and front/back sound localisation is however still possible without head movement. For sound having a relatively broad and flat spectrum, as is often the case in nature, direction-dependant filtering by the head and pinnae (the flaps on the outer ear) provides vertical and front/back directional cues. [31]

The interaural difference and spectral filtering cues are helpful in detecting the direction of a sound, but there is another dimension to our spatial hearing - distance. Auditory distance perception is far less accurate than our ability to detect the direction to a source of sound. "*The principal acoustic cues for distance perception are intensity (i.e., sound level arriving at the listener's ears), direct-to-reverberant (D/R) energy ratio, and ILD. The relative importance of these cues varies widely across conditions.*" [31] An in-depth discussion of these distance

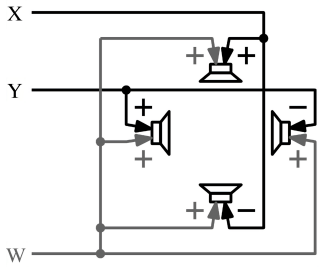

Figure 1.4 The WXY channels routed to 4 speakers for 2D ambisonics playback Image sourced from [17].

cues, and some other aspects of spatial hearing is outside the scope of this thesis, but I highly recommend Chapter 6 of [31] for further reading.

1.3.2 Basic examples of ambisonic decoding

Decoding ambisonics is the process of transitioning from the whole-scene representation back to separate channels that can be played back using individual speakers. For the sake of simplicity let's first take a look at playing back 3 out of 4 B-format channels using 4 speakers. The only processing required for playback is adjusting the phase of the X and Y channels by 180° for the right and back speakers respectively as seen in 1.4. (The front and left speaker receive the unadjusted signal from the X and Y channels, and all speakers receive the W channel equally.) It should be emphasized, that the Z channel is not utilised in this configuration, and thus height information is lost.

The previous example assumes an ideal loudspeaker layout. Unfortunately, that assumption almost never holds true in practice, and thus it is required to playback ambisonic audio with loudspeakers positioned at arbitrary angles towards the listener. To achieve this, it is required to know the angle from each speaker to the listener. Figure 1.5 shows how this can be achieved in practice. $\Theta_1 ... \Theta_L$ are the direction vectors for the loudspeakers; Θ_X, Θ_Y and Θ_Z are the unit vectors for the X, Y and Z channels respectively. The speaker signals ($[S_1 ... S_L]$ are calculated by multiplying the source signal vector $[WXYZ]^T$ with the decoding matrix D. [17]

$$\begin{bmatrix} S_1 \\ \vdots \\ S_L \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & \theta_1^T\theta_X & \theta_1^T\theta_Y & \theta_1^T\theta_Z \\ \vdots & & & \\ 1 & \theta_L^T\theta_X & \theta_L^T\theta_Y & \theta_L^T\theta_Z \end{bmatrix} \begin{bmatrix} W \\ X \\ Y \\ Z \end{bmatrix} = \underbrace{\frac{1}{2} \begin{bmatrix} 1 & \theta_1^T \\ \vdots & \vdots \\ 1 & \theta_L^T \end{bmatrix}}_{\boldsymbol{D}} \begin{bmatrix} W \\ X \\ Y \\ Z \end{bmatrix}.$$

Figure 1.5 Decoding first order ambisonics using a sampling decoder. Image sourced from [17].

The presented decoder is called a "sampling decoder" - ambisonic playback using such a decoder is "comparable to recording each signal with a virtual first-order cardioid microphone aligned with the loudspeaker's direction Θ_l". [17] It is important to note that the speakers need to be uniformly spaced so the loudness of the resulting sound is not affected by the directional composition of the ambisonics signal (so sounds coming from specific directions are not perceived as louder than sounds coming from elsewhere). An example of a 2D speaker array consisting of uniformly spaced speakers can be seen in figure 1.6.

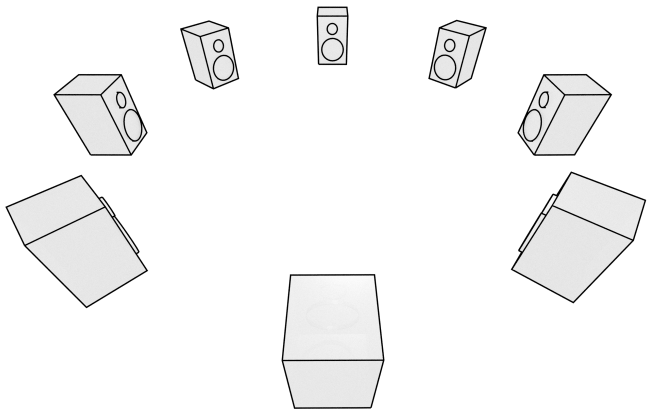

Figure 1.6 A 2D (horizontal plane) surround speaker array. Image courtesy of the author.

1.3.3 Decoding to loudspeakers

The ambisonics panning function $g(\theta)$ that yields the power of the signal for a given direction θ is continuous, but in case of loudspeaker playback, the sound field must be recreated using a discrete number of point-sources. (See [17] for the definition of $g(\theta)$ and a detailed dive into the math behind ambisonics.) Only special arrangements of loudspeakers permit direct sampling without introducing undesired direction-dependant gain variations (as described for first order ambisonics in the previous subsection). These so-called *t-designs* are arrangements of speakers where the distance and direction between all neighbouring speakers are constant. $t \geq 2N + 1$ loudspeakers are required for optimal reproduction of ambisonics of order N. [17][37]

Vector Based Amplitude Panning To describe some of the more advanced ambisonic decoding techniques, let's first introduce Vector Base Amplitude Panning (VBAP). First described by Pulkki in 1997 in [38] and then in [39], VBAP provides a mathematical framework for selecting which loudspeakers from a 2D/3D surround arrangement should be activated and with what gains, to reproduce a virtual sound source at a specific direction.

In the simplest example of stereophonic amplitude panning (two speakers positioned in front of the listener), Pulkki treats the loudspeaker direction vectors as a vector base. A direction vector for a virtual sound source at an arbitrary point on the arc enclosed by the two loudspeakers can be calculated as a linear combination of the two basis vectors. The coordinates of the sound source direction vector in that vector space are then interpreted as the gain values for each loudspeaker. This system is then extended to horizontal arrangements of more than 2 loudspeakers by always activating just 2 loudspeakers - the ones that form the arc on which the virtual sound source lies at that moment. As Pulkki states, utilising just two loudspeakers at a given time might seem wasteful, but this approach increases localisation accuracy.

The 2D case is then generalised to 3D by utilising triplets of loudspeakers instead of pairs. An approach identical to the 2D case is used for arrays of more than 3 speakers. Picking the correct speaker triplet to activate can however pose a problem in cases where the speaker arrangement allows for ambiguous triplet selection and some other situations. Due to this, VBAP does favour specific speaker arrangements, but it is still a flexible approach that can be used in a variety of situations. [39][17]

All-Round Ambisonic Decoding (AllRAD) The AllRAD approach presented in "All-Round Ambisonic Panning and Decoding" by Franz Zotter and Matthias Frank [37] combines VBAP and sampling decoders, as described in previous sections, to achieve a lower direction-dependant loudness variation and good spatial resolution while also providing better results on irregular loudspeaker arrangements.

The idea behind AllRAD is first decoding (sampling) the ambisonic signal to a set of virtual speakers arranged in a t-design, and then using these signals as virtual sources for a slightly modified version of VBAP. Decoding to a regular arrangement of virtual loudspeakers has the advantage of low directional gain variations and good sound localisation, while VBAP makes this approach usable with real-life non-ideal speaker arrangements (higher order t-design speaker arrays are usually not very practical).

For playback on speaker hemispheres (which are more common than full spheres in practice due to several factors), as with normal VBAP, one or more imaginary loudspeakers may be added at the bottom of the arrangement, or at the average direction vector of the real loudspeakers. The signal from the imaginary loudspeaker(s) may either be discarded or equally distributed to the neighbouring loudspeakers. While this approach does not actually reproduce the direction of sounds coming from below correctly, it helps avoid the inability to find a matching speaker triplet (which would make these sounds inaudible), and preserves some of the audio content coming from below (especially for sounds close to the outer ring of the speaker hemisphere). A similar approach can of course be used for other speaker arrangements where there is no speakers in certain parts of the sphere. [40][17]

I should note that while AllRAD is one of the most flexible and practical solutions available today, there are multiple other decoding approaches. Overviews of the other approaches can be found in [17], [41] and [40].

1.3.4 Decoding to headphones

Since human listeners can detect directivity cues with just two ears, it stands to reason, that with the right technology, 360° sound can be reproduced on headphones just as well, if not better, than on loudspeaker arrays. To achieve this, it is required that the main psychoacoustic cues used by humans (as described in section 1.3.1) are correctly reproduced. Ideally, interaural difference cues and changes to the frequency composition of the sound induced by the listener's head and ears need to be accounted for.

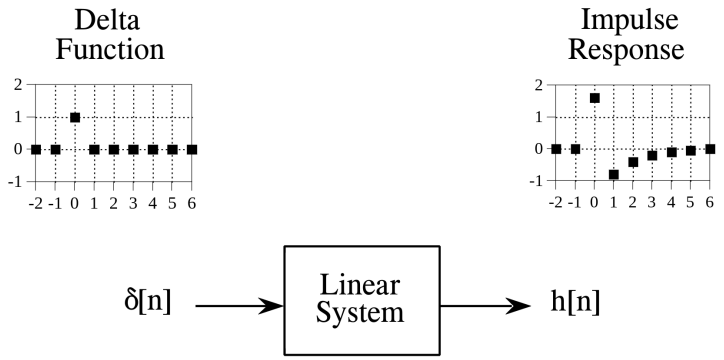

Figure 1.7 Definition of a delta function and an impulse response. $\delta[n]$ is used to identify the delta function. The impulse response of a linear system is usually denoted by $h[n]$. Image sourced from [42].

All of this can be captured using a so-called Head-related Impulse Response (HRIR). First, let's figure out what an impulse response is in general. An impulse response is the output of a linear system when a delta function (unit impulse) is the input. Considering a discrete signal

(N separate samples of a continuous signal), a unit impulse is such a signal where the sample at position 0 has a value of 1, and all other samples have a value of 0. An important property of the delta function is that any signal can be represented as a combination of differently shifted and scaled delta functions.[42] Figure 1.7 provides a graphical definition of the delta function (unit impulse) and it's impulse response. In the case of a HRIR, the impulse response represents how the listener's head and ears affect sound in a so-called free-field - an environment with no reflections or reverberations and thus no colouration of sound.[43] Of course an ideal free-field environment can not be achieved in practice, but anechoic chambers get sufficiently close.

There are multiple methods and variations of methods of measuring a HRIR; a good in-depth overview can be found in [44]. The core idea behind most of them is however the same - microphones are placed into the subject's ears, a specific sound (an excitation signal) is then played back (often multiple times) over a loudspeaker located at a specific point in space, and then the impulse response of the system is calculated from the recorded audio. Multiple measurements with the sound coming from different directions and sometimes distances have to be taken. Often, especially in case of human subjects (as opposed to head models) it is desirable to move the position of the sound source instead of rotating the listener's head.

In the time domain, the relation between the input and the output of the system is given by the following equation ([42]):

$$y(t) = x(t) * h(t) = \int_{-\infty}^{+\infty} x(\tau)h(t-\tau)d\tau, \tag{1.3}$$

where $x(t)$ is the input signal, $y(t)$ the output signal, $h(t)$ is the impulse response of the system, and $*$ denotes convolution. The same equation can be written for discrete signals as follows ([42]):

$$y[n] = x[n] * h[n] = \sum_{j=0}^{N-1} h[j]h[n-j], \tag{1.4}$$

for an N-point impulse response $h[n]$ with indexes spanning from 0 to $N-1$. Another term that the reader would surely encounter when researching ambisonic decoding to headphones, and binaural audio in general is a head-related transfer function. A HRTF is nothing more than a HRIR in the frequency domain. [43] Once a HRIR is calculated, convolution can be utilised to calculate the response of the system to an arbitrary input signal.[42] Using HRTFs measured at different impulse directions, and utilising interpolation techniques, psychoacoustic cues for a sound coming from an arbitrary direction can be reproduced.[44]

Decoding ambisonic audio to headphones using HRTFs can be achieved by utilising virtual loudspeakers. A decoder such as AllRAD is first used to produce a CBA signal set for some favourable arrangement of loudspeakers, such as a t-design, to achieve low loudness variation and good localisation accuracy. Each of the resulting virtual loudspeaker signals is then convolved with a HRTF measured at an appropriate direction. Finally, the signals are added together to produce the final binaural mix. [45]

It is worth mentioning that, as with most topics discussed thus far, there are many possible variations and improvements to the base approach. Some of them are discussed in [17].

1.3.5 A comparison of personalised and generic HRTFs

So-called personalised (or individualised) HRTFs are measured for a specific individual and thus match how that person perceives sound in everyday life as closely as possible. Generic HRTFs on the other hand are synthesized from large datasets of individual measurements with the aim to provide a good enough experience for most people. Unsurprisingly, research shows that individualised HRTFs perform better than generic ones, with the main improvement being a reduction in front-to-back confusion. [46] However, non-personalised HRTFs show good enough performance

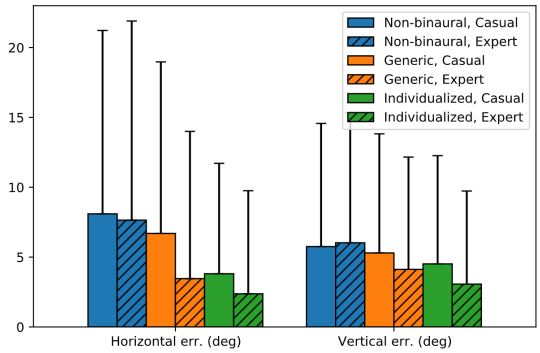

Figure 1.8 Horizontal and vertical localization errors for expert (solid pattern) and casual (hatched pattern) players. Image sourced from [48].

in many cases, and are the more common choice in practice due to the cost and complexity of measuring individual HRTFs. Furthermore, with enough time, a listener's brain is able to adjust to a "foreign" HRTF - experiments performed in [47] show worse localization test performance by inexperienced subjects, than those that went through training (both simultaneously hearing a sound and seeing it's position, and trying to determine a sound's position with post-test feedback). Trained subjects also showed better results when presented with sounds at positions different to the ones used during the training phase of the experiment.

Non-individualised HRTFs are already being widely used in interactive media, especially computer games. Sound localisation is important in first-person shooters and similar titles, and plays an enormous role in increasing immersion in VR. HRTFs are employed in Overwatch, Counter Strike: Global Offensive, Valorant, and many VR titles. There has also been some academic research into the usage of individualised and non-individualised HRTFs in video games. While [48] concludes, that individualised HRTFs expectedly showed better localisation performance, it is of note that the experiment results (Fig. 1.8) showed a relatively similar horizontal localisation performance between (self-reported) experienced players when using non-individualised HRTFs and players using an individualised HRTF. This can realistically be attributed to "expert" players having more experience with other FPS titles with HRTF-based spatial audio, as many of the most popular competitive titles do employ it.

Another study, that focused specifically on VR games [49], showed that sound localisation using generic HRTFs was improved with time, when the players were presented with matching visual and auditory stimuli. The improvements didn't occur when no visual stimulus accompanied the sound, or when the stimuli were not sufficiently synchronised. The experiments also showed that the improvements in localisation didn't transfer to different auditory stimuli (sounds with a significantly different spectral composition). This however should not pose a problem in case of video games and other hand-crafted experiences, as they tend to use a comparatively low number of individual sounds, especially if only sounds whose accurate localisation is important to the overall experience are taken into account.

Chapter 2

An overview of spatial audio production software

This chapter examines some of the spatial audio production tools publicly available at the time of writing and discusses their suitability for 360° video production.

2.1 DAW-based solutions

Most tools for producing ambisonic audio come in the form of DAW plugins, most of them in the VST format[1]. They are usually released in the form of plugin suites, consisting of ambisonics encoders (panners) and decoders.

The encoder plugin takes an input (in most cases mono or stereo), computes the spherical harmonic coefficients, based on the current panning settings, and outputs an ambisonics representation with the sound placed at the desired direction relative to the listener. An instance of the encoding plugin has to be placed on each channel that will be used in the final ambisonics mix, as can be seen in figure 2.1. Recordings from microphone arrays, and other ambisonic audio can also easily be added to the final mix by simply adding the individual channels together.

Using the encoder alone is enough to output an ambisonics mix, but in order for the mixing engineer to hear the results of his actions, or to output the mix in CBA format, the ambisonics representation has to be decoded back to channel-based audio. For these purposes these plugin suites usually include a decoder plugin that converts the ambisonics signals into speaker feeds for a surround setup, or into a stereo binaural representation using a personalized or generic HRTF. The decoding plugin is usually placed on the master bus[2] - figure 2.2.

Because the plugin outputs multiple channels of audio, the digital audio workstation itself has to provide support for multichannel audio. The maximum achievable ambisonics order is then limited by three factors - available computational power (for real time preview), the plugin itself, and the number of channels supported by the DAW. Fortunately, while not prevalent, multichannel support is present is some of the popular digital audio workstations, such as Reaper [51] and Pro Tools [52]; Steinberg's Cubase Pro even provides built-in ambisonics support in the form of encoder and decoder plugins, as well as general multi-channel capabilities [53]. Nuendo - another DAW from Steinberg, geared towards industry professionals, provides ambisonics support with advanced features - their decoder, for example, supports head-tracking to synchronize the

[1] VST is a plugin format for digital audio workstations developed by Steinberg Media Technologies. [50] While competing plugin formats exist, VST is multiplatform, and one of the most widely supported ones.

[2] A master bus is the final stage in audio processing. It combines the signals from all the individual channels (or tracks) in a DAW or a hardware mixing console.

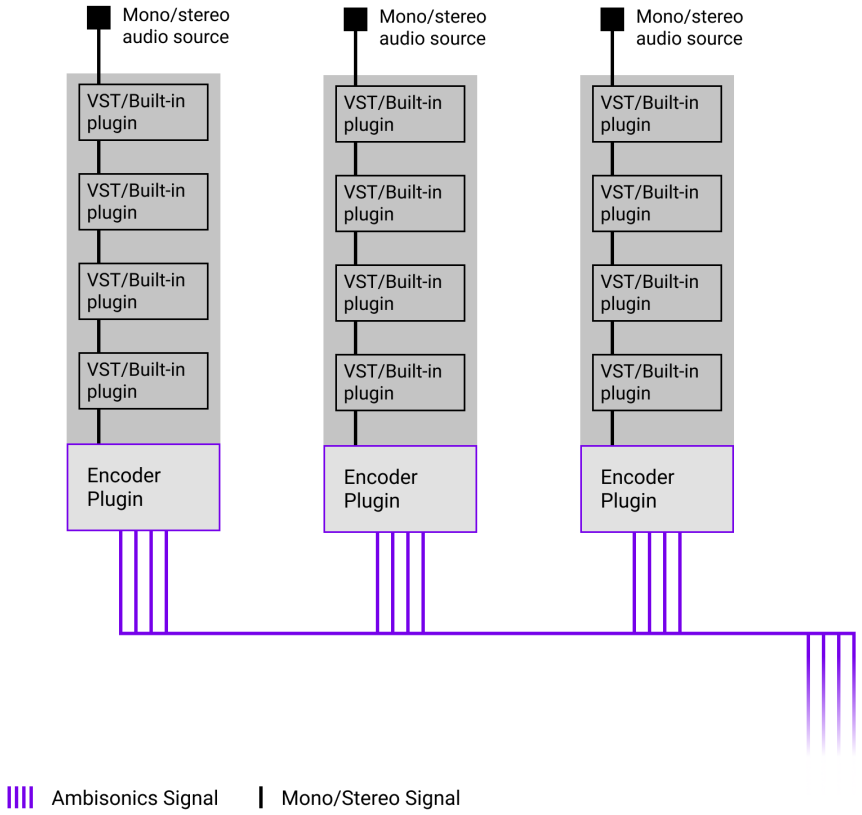

Figure 2.1 Ambisonic encoder plugins placed on individual channels. Image courtesy of the author.

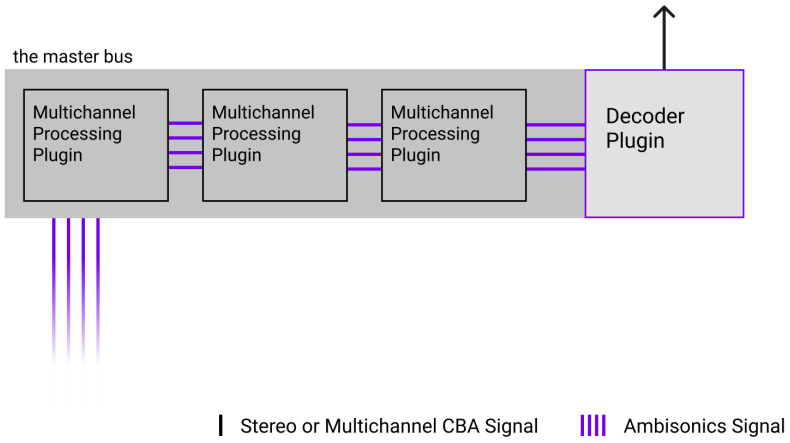

Figure 2.2 A master bus with an ambisonics decoder plugin. Image courtesy of the author.

audio with head movement (Fig. 2.3). Nuendo also has built-in integration with some game audio middleware systems [3] (Wwise and ADX2), as well as features specific to VR audio , including support for dearVR Spatial Connect - a virtual reality environment for spatial audio mixing, which also provides positional data export to the Unity game engine. [54]

Figure 2.3 Head-tracking and spatial panning in Steinberg's Nuendo. Image sourced from [55].

Ambisonic plugin suites usually include multiple types of encoders and decoders. Various encoders can be used for encoding non-ambisonic content with different directivity patterns, e.g. planewave (sound emanating from a single direction) or omnidirectional, as well as transforming recordings from various microphone arrangements into a standardised ambisonics representation. Decoders usually differ in their output format - CBA for various speaker arrangements, binaural stereo, etc.

Listed below are some of the more fully featured ambisonics plugin suites (with a bias towards open source).

- The Ambisonic Toolkit [56]
- ambiX [57]
- IEM Plug-in Suite [58]

The DAW-based approach to spatial audio production provides a lot of flexibility, especially if working in Reaper or another DAW with similar multichannel and routing capabilities. The producer is able to use the in-DAW workflows they have grown accustomed to and can benefit from the vast amount of available plugins. Plugins from different manufacturers, including ambisonic encoders, decoders, and spatial effects, can all be used in a single project. This is approach is popular and flexible, but there is definitely still room for improvement, especially for more specialised workflows such as 360° video production. One area for improvement is integration between the different software used throughout the production process. Such functionality is absent from most DAWs, which is to be expected due to the relatively low number of users that would benefit from it. Fortunately, in many cases, such workflows can be improved by plugin developers.

[3] Middleware systems are software that is integrated into a game engine to handle a specific subset of it's functionality, such as audio playback.

2.2 Game Engines

While digital audio workstations are the obvious choice when it comes to producing any form of audio content, including ambisonics, in some cases, other approaches may be more sensible. The visual fidelity of modern game engines is already on such a high level, that they can be used for 3D productions. Their object-based audio systems also provide extensive functionality including mixing and real time audio processing. [59][60] Theoretically, this could allow them to be used as an all-in-one 360° video production tool. Unfortunately, in reality, using game engines for 360° video production might not be a good choice. Neither of the two most popular commercially available game engines provide reliable solutions for exporting 360° video and audio.

Unreal Engine does come with the ability to export 360° video out of the box, but only as a series of still images, and thus without audio. Exporting ambisonics is not supported in any way at the time of writing, although workarounds, converting a CBA mix from unreal to ambisonics, can be devised (but quality loss would be inevitable in that case). [61] Unity doesn't have built-in support for exporting 360° video or audio, but a third-party solution is available. The "VR Panorama 360 PRO Renderer" tool's Unity Asset Store page [62] does claim to support rendering 360° video, as well as ambisonics audio "for YouTube". But the quality of the implementation and integration with Unity is not guaranteed, this being a third-party product.

Being able to use a single piece of software for both the visual and the audio aspects of production is certainly attractive. Unfortunately, the major commercially available game engines are not currently viable for 360° video production. Their feature-set, however, despite being intended for a different use case, already provides a great foundation that can be built upon by adding some 360° video specific features. With the popularisation of spherical video and VR gaming, it is possible that 360° gameplay demos and game trailers will become more common, and such features will be added as demand increases.

2.3 3D software

Studying the manuals for multiple widely adopted 3D software packages [4] leads to an unsurprising finding - most lack audio features in general, let alone SBA export capabilities.

Both Autodesk's offerings - Maya and 3ds Max provide some degree of audio support, but these features are limited and not usable in a spatial audio context. Maya's audio support is intended purely for synchronising animation to sound. [63] 3ds Max does provide a more fully featured audio editing system, but it's limited to arranging pre-rendered CBA clips, and is incapable of spatial audio encoding in any form. [64]

Blender on the other hand does include a spatial sound system. The user can place speaker objects, representing individual sound sources at arbitrary positions in the 3D scene. The software allows to choose between several distance fall-off models, adjust the direction in which the sound is emitted from each individual speaker object, and even includes doppler effect simulation. [65] The unfortunate part is, that the current version of Blender (at the time of writing - 3.1) only supports exporting stereo. It is not documented anywhere which technique is used to render this stereo output, but, based on my listening test[5], it is not binaural stereo. Interestingly, earlier versions of Blender (at least up until v2.79[6]) did include the ability to export 5.1 and 7.1 surround audio, but it has later been removed. Overall, Blender itself can also be deemed unusable for spatial audio production. A Blender add-on which implements OBA export will be mentioned in the next subsection.

There is however an outlier, both in terms of spatial audio support, and in terms of the software's general purpose - SideFX's Houdini. A software with a target audience intersecting

[4] Blender, Autodesk Maya, Maxon Cinema 4D, Autodesk 3ds Max, and SideFX Houdini.

[5] I placed a speaker object directly above a camera moving on the vertical axis, and listened to the result to check if there was any height information present.

[6] Downloading and launching the aforementioned version of the software confirmed that the feature is present.

those of the previously mentioned programs only to a lesser extent, Houdini takes a completely procedural approach to 3D and includes advanced simulation capabilities. Audio simulation capabilities are also present, and "simulation" is truly the correct word to use here. Houdini's spatial audio system supports distance fall-off, doppler effect, and even obstacle interference, including per-obstacle customisation of transmission and absorption levels for different frequencies. [66][67] As in Blender, sound objects with customisable emission direction can be placed in the 3D scene. [68] The sound can then be "captured" by an arbitrary amount of virtual microphone objects [69], effectively allowing to record audio using microphone setups analogous to those used in the real-world, which makes ambisonics and CBA recordings possible. While I could not find any examples of spatial audio rendered using Houdini, and thus can not assess the quality of the output, the capabilities of this solution seem vast. Houdini, however, is not an audio production environment. Using it's spatial audio system, as with it's other aspects, requires much more specialised knowledge than using other approaches, such as DAWs and game engines.

2.4 Other solutions

This section includes some interesting solutions that didn't fall into any of the above categories and don't have alternatives that would warrant adding a separate subsection.

IRCAM Panoramix Designed and developed at IRCAM (French: Institut de recherche et coordination acoustique/musique, English: Institute for Research and Coordination in Acoustics/Music), Panoramix is a post-production and mixing workstation for 3D-audio content. It aims to provide a "comprehensive environment for mixing, reverberating, and spatializing sound materials from different microphone systems: surround microphone trees, spot microphones, ambient miking, Higher Order Ambisonics capture." ([70]) It is not meant to replace a digital audio workstation, but rather work in conjunction with one, aiding in the mixing stage of the production workflow. The design is inspired by hardware mixing decks with each track represented by a vertical strip. There are various types of input tracks, accommodating different input signal types, e.g. mono, ambisonic, different microphone arrangements. It also includes various other tools useful in spatial mixing and mastering, such as a reverberation engine, as well as the usual mixing instruments - an equaliser and a compressor/expander. A concise feature list can be found in [71]; a conference paper ([70]), that delves deeper into the motivation behind creating the software and the design decisions made in the process might also be of interest to the reader.

The "soundobjects" Blender add-on Created by Jamie Hardt (@luvcapra on GitHub), "soundobjects" is a Blender add-on that allows export of OBA in the ADM Broadcast WAV format. (See [72] for the format specification.) It provides Blender operators that allow to import WAV audio files, add speaker objects to existing objects in the scene, assign audio files to them, and export a file in the ADM Broadcast-WAV format where each audio object corresponds to a Blender speaker object in the scene. The resulting file contains a sound object for each of the speaker objects in the blender scene.

Unfortunately, it seems this project is no longer being developed or maintained. The last commit in the project's GitHub repository ([73]) adds a note about changes in ProTools that result in the inability to import WAV files produced by the add-on. Despite this, the project is still an interesting tool, and, with interest from the open source community, could be revitalised and improved.

Chapter 3

UI/UX Design

This chapter explains the reasoning behind some of the core design decisions, specifies the software requirements, and provides an overview of the GUI layout.

3.1 Real-time approach

Ambilink's[1] main goal lies in allowing to utilise animation data from a Blender scene for ambisonic panning in a DAW. This is achieved with a pair of plugins - a Blender plugin and a VST plugin. The Blender plugin (or "add-on", using Blender's official terminology) provides data about the 3D scene to instances of the VST plugin, that perform the ambisonic panning. Deciding how this data will be passed from the Blender add-on to the VST was the earliest design decision that had to be made, since the chosen approach would dramatically affect the way the software functions, and, naturally, the user experience.

One solution would be to export animation data from Blender to a file. The data could then be imported by the VST plugin. Unfortunately, such a solution, although simple to implement, would be quite suboptimal from a UX perspective. Music production experience has taught me that immediate feedback is immensely helpful when making creative decisions, and that decisions made by sound designers and mixing engineers are undoubtedly of creative nature. Thus, from the very beginning, I've designed the software to include real-time preview functionality - the panning directions are continuously updated based on the current state of the Blender scene, allowing the user to immediately hear how adjustments performed in Blender affect the audio mix.

Of course, the real-time approach also has it's downsides. It increases the overall complexity of the system, and may introduce new types of problems stemming from connection issues. Communication latency may also negatively affect responsiveness. A good implementation should however be able to avoid or mitigate these flaws.

3.2 Requirement definition

Once the decision to make the system real-time was made, I've settled on some functional and non-functional requirements to guide the development.

[1] "Ambilink" is the name given to the software during early stages of development.

Functional requirements

FR1: The system consists of two components - the Blender add-on and the VST plugin.

FR2: The user is able to pick an object from the Blender scene using the VST plugin. The location of that object is then used to calculate the panning direction used by the VST plugin.

FR3: Whenever the location of a Blender object is updated (manually or during animation playback), the panning directions are updated in real time. (Communication between the two components is performed in real time.)

FR4: When a VST instance is bound (subscribed) to a specific Blender object, renaming or deleting the object in Blender is reflected by the VST GUI.

FR5: The VST plugin supports offline rendering[2]. The exported audio is synchronised to the animation.

FR6: The distance from a Blender object to the camera affects the gain of the respective VST distance; the user is able to choose from multiple distance attenuation models.

FR7: The user is able to pick the Ambisonics order and normalisation type (N3D or SN3D) used by the plugin.

Non-functional requirements

It has to be noted that since the software can run on any number of hardware configurations, it is hard to define performance-related requirements unambiguously. Thus, the "scales well" formulation used in the non-functional requirements below should be understood as "scales well for the hardware".

NFR1: Selecting an object from the Blender scene via the VST GUI is quick, even when the scene contains a large number of objects.

NFR2: Real-time direction updates have an acceptable delay, ideally less than 0.5 seconds.

NFR3: The system scales well for Blender scenes containing a large number of objects (e.g. thousands).

3.3 User interaction

With the requirements in place, the next step was designing the way the user will interact with the software. In case of Ambilink, this was a relatively straightforward process, since the system has a single use case, that only requires limited user interaction. A user's workflow might roughly follow these steps:

1. Open the Blender project, and start the Ambilink Blender add-on.
2. Open the DAW, and place the Ambilink VST on the channels containing sounds that should correspond to objects in the Blender scene.
3. For each instance of the Ambilink VST, select an object from the Blender scene.

[2]Offline rendering refers to exporting the audio, as opposed to real-time (online) playback during the production process.

4. Adjust ambisonics-related settings (order, normalisation type) to match what's being used in the project.

5. Make any necessary adjustments to the Blender scene, or the audio mix, including adjusting distance attenuation parameters of the Ambilink VST.

6. Render the resulting audio.

3.3.1 Blender add-on

The Blender plugin needs to support two main actions - launching and stopping the server that will provide data to the VSTs. Most operations in Blender are realised using so-called operators that essentially encapsulate any action that can be performed by the user. Operators can be invoked via buttons in the GUI or using Blender's search functionality. Because the user shouldn't have to start or stop the server often, I've chosen to avoid adding an additional panel to the GUI. Instead both actions are realised using operators that can be invoked from the `File > Export` menu, or using search.

The Blender add-on depends on an external Python module.[3] I did not want to include the module's code as part of the addon, or to automatically install third-party modules without informing the user. Instead I've chosen to be transparent and require the user to explicitly initiate dependency installation. This can be done by pressing a button in Ambilink's add-on preferences panel which can be accessed through Blender's preferences dialogue. If dependencies aren't installed and the user invokes the "Start Server For Ambilink VST" operator, an error message is shown that instructs the user to install dependencies via add-on settings.

3.3.2 VST plugin

The VST plugin should allow the user to perform a much wider range of actions - selecting an object from the Blender scene, changing distance attenuation and ambisonics parameters. Figure 3.1 shows a wireframe of the user interface created before implementation. The GUI consists of a main screen that is shown at launch and two additional screens - the object selection screen and the settings screen.

Main screen The main screen displays the currently active object (if any), and a visual indicator of the current direction from the camera to the object. It also allows to adjust the distance attenuation type and maximum distance to the camera at which the sound gain will be above 0 dB. The distance attenuation parameters are located on the main screen to minimise the number of clicks required to adjust them. This is done because various sounds may require different values, depending on the distance at which the user would want them to become audible. By pressing the button on the right side of the active object indicator the user can transition to the object selection screen. The settings screen can be accessed by pressing the "Settings" button located in the top right corner of the UI.

Object selection screen The object selection screen allows the user to select an object from the Blender scene. Since the Blender scene may include a high number of objects, a search box is present in the top part of the UI, allowing the user to quickly find a specific object. When searching, the first object in the list is highlighted; the user can then select that object by pressing the "Enter" key.

[3] External dependencies will be discussed in Chapter 4.

Settings screen The settings screen is reserved for plugin parameters that are not expected to be adjusted often, and can thus be placed on a separate screen to avoid cluttering the main UI (and save space for adding new parameters further down the line). Right now it only includes the ambisonics order and the normalisation method. The user should only need to adjust these parameters once per project most of the time.

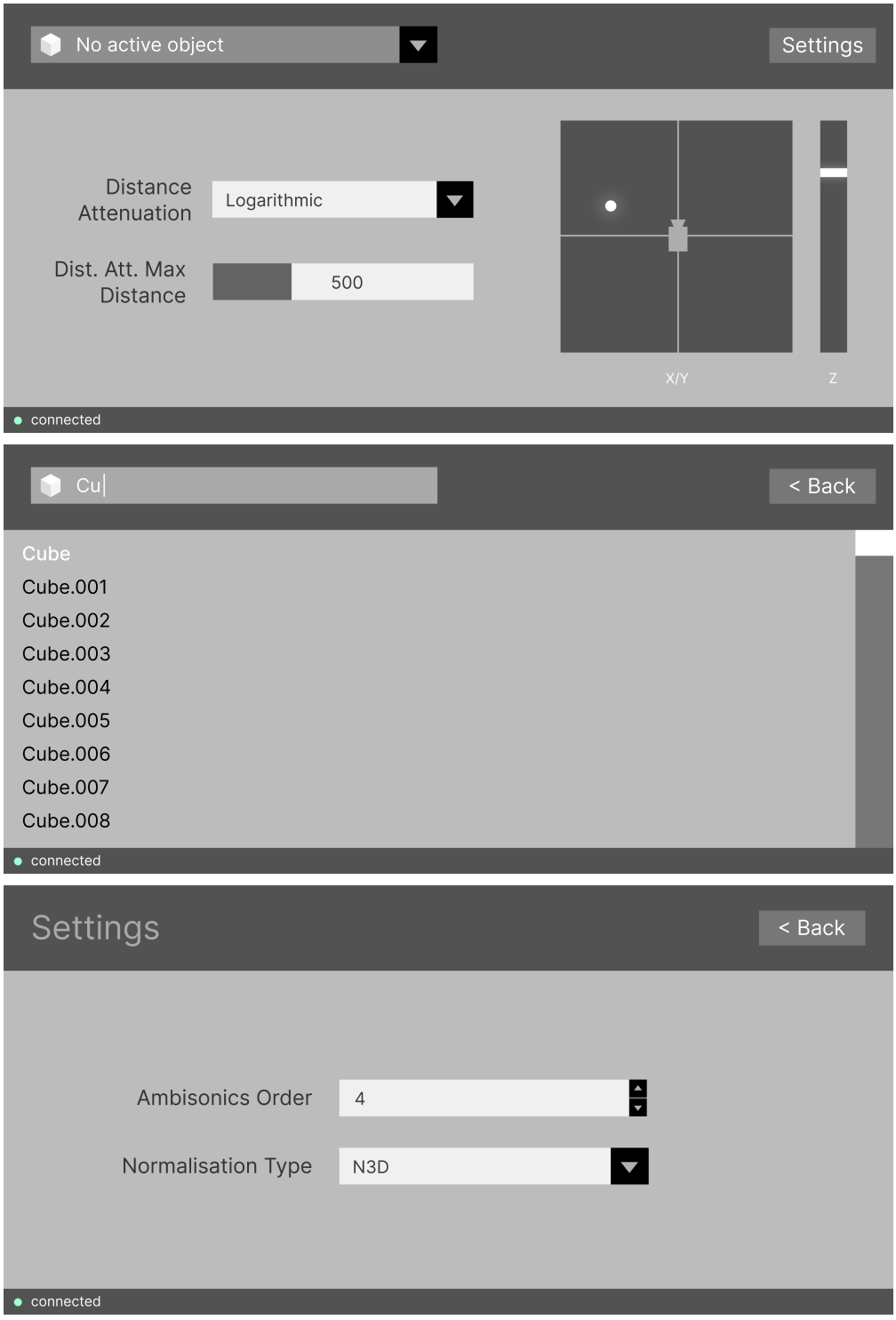

Figure 3.1 Prototype GUI layout for the VST plugin. From top to bottom: main screen, object selection screen, settings screen. Image courtesy of the author.

Chapter 4
Implementation

This chapter provides an overview of the implementation of the Blender add-on and the VST plugin. It present the structure of the IPC protocol, describes the top level architecture of each component, and provides some insight into the challenges that had to be solved.

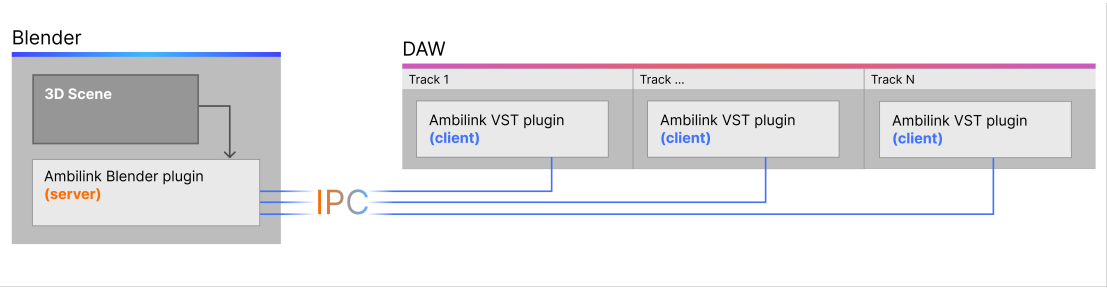

Figure 4.1 Diagram of high level interaction between the VST instances and the Blender plugin. Image courtesy of the author.

4.1 IPC Protocol

To minimise latency in communication between the Blender add-on and the VST plugin, Ambilink utilises IPC [1]. Since Ambilink is developed in a way that should enable eventual support of all three major operating systems, utilising OS-specific IPC mechanisms directly was undesirable. Instead, I've chosen to entrust the low level details of interprocess communication to a third-party library - NNG (nanomsg-next-gen). NNG is multiplatform, provides support for various transports (e.g. IPC, TCP, WebSockets), as well as common communication patterns (e.g. Request/Response, Publisher/Subscriber). While NNG itself is written in C, wrappers exist for many popular programming languages, including Pynng for Python, as well as nngpp for C++. (The latter serves more as a quality of life improvement, as NNG's C interface can of course be used directly from C++ code.) Using the library also highly increases flexibility of the implementation, allowing, for example, to replace IPC with network communication via TCP or WebSockets with minimal effort. All of this made NNG a great choice for Ambilink.

[1] IPC is a generic term - interprocess communication may very well be realised using TCP or other network protocols. In the context of this thesis, however, the term IPC is used to refer to any form of "direct" interprocess communication, such as named pipes or shared memory.

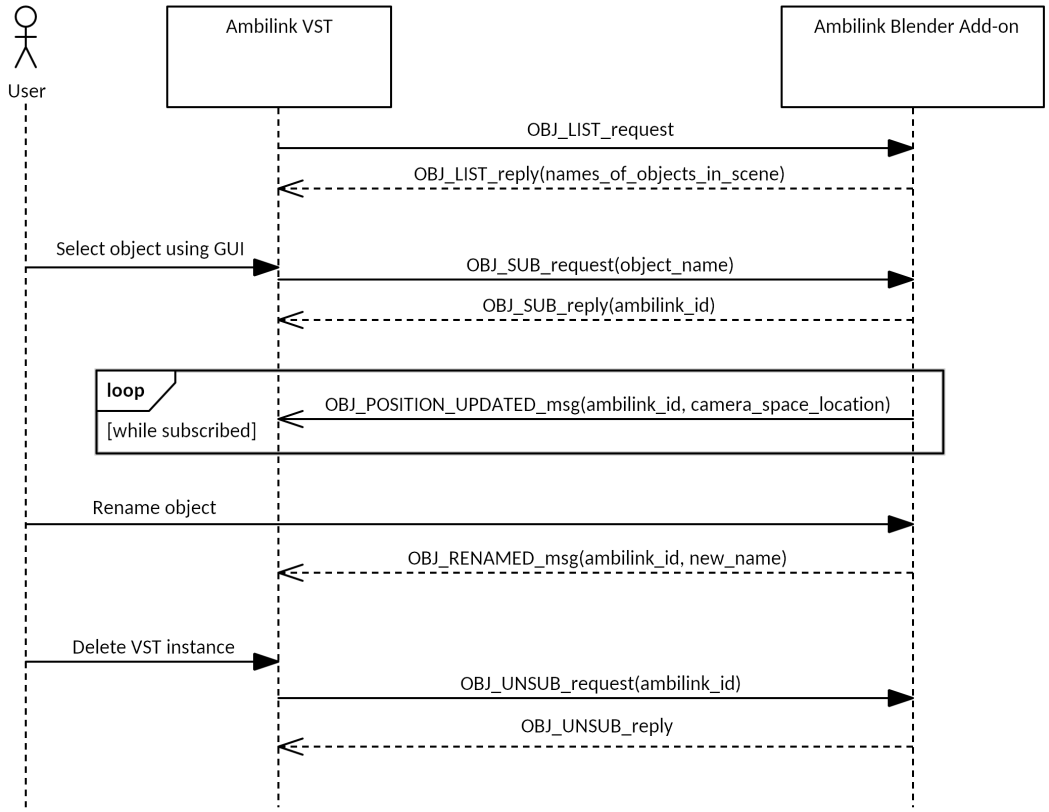

Figure 4.2 Communication sequence between the Blender plugin and an instance of the VST plugin during real-time (online) rendering. Messages with the suffix "request" or "reply" are sent using Req/Rep. Messages with the suffix "msg" are sent using Pub/Sub. Image courtesy of the author.

Figure 4.1 illustrates how the individual components are connected via IPC from a high level perspective. The Blender plugin takes on the role of a server, providing information about the Blender scene to an arbitrary number of clients - instances of the VST plugin.

A sequence diagram illustrating communication between the two plugins in real-time rendering mode is shown in figure 4.2. First, the user picks an object from the Blender scene. The VST instance then subscribes to that object, requesting the Blender add-on to begin informing it about updates to the object's location and other important events. The object is then renamed in Blender, and a rename notification is published by the Blender add-on, allowing the VST GUI to update the object name. Finally, the VST instance is deleted by the user; it's destructor sends a request to unsubscribe from the object. (Object position updates would have continued after the rename message if the instance hadn't been destroyed.) Two different communication patterns implemented by the NNG library were used during that sequence:

Request/Reply (Req/Rep) This pattern is used for all communication initiated by the VST instance - requesting the list of objects in the Blender scene, subscribing to an object, requesting object location data for offline rendering, etc.

Publisher/Subscriber (Pub/Sub) In this pattern, a single publisher produces messages that are consumed by multiple subscribers. Ambilink starts using this pattern once a VST instance is subscribed to an object. The Blender plugin serves as the publisher, and the VST plugin instances as the subscribers. Messages published by the Blender plugin include object location updates (published multiple times per second for all objects with at least one subscribed VST

instance), object name updates, and object deletion notifications.

4.1.1 Protocol messages

```
class ReqRepCommand(IntEnum):
    OBJ_LIST = 0x01
    OBJ_SUB = 0x02
    OBJ_UNSUB = 0x03
    PREPARE_TO_RENDER = 0x04
    INFORM_RENDER_FINISHED = 0x05
    GET_RENDERING_LOCATION_DATA = 0x06
    GET_ANIMATION_INFO = 0x07
    PING = 0xFF

class ReqRepStatusCode(IntEnum):
    SUCCESS = 0x00
    OBJECT_NOT_FOUND = 0x01
    INVALID_REQUEST_DATA = 0x02
    INTERNAL_ERROR = 0xFE
    UNKNOWN_COMMAND = 0xFF
```

Code listing 1 Definition of IPC protocol constants used in Req/Rep messages. (From the source code of the Blender add-on.)

This section provides an overview of the message structure for each communication patter used in the IPC protocol. The `documentation/ipc.md` file included on the attached media defines the structure of each individual message type. Alternatively, documentation generated using Doxygen (`documentation/Doxyfile` on the attached media) will also include the contents of the file.

Request/Reply The Request/Reply part of the protocol consists of a set of commands differentiated by a single-byte command code. A request always starts with the command code, and a reply starts with a status code (also a single byte) indicating if the operation was successful or the reason it failed. In both cases optional command-specific data may follow. Listing 1 shows the list of all commands and status codes and the corresponding constants.

Ambilink IDs During the initial stage of communication (before a subscription is established) Blender objects are referenced by their name. While object names in Blender are unique, using the object's name as the identifier would not be viable as the objects can be renamed after the subscription is established. It would also be inefficient - a whole string would have to be passed for every request referencing a specific object, as well as in every message published by the Blender add-on, increasing the size of most messages. Because of these factors, once a subscription is established, a unique numeric identifier is used to reference Blender objects. This value, called an Ambilink ID, is assigned by the Blender plugin [2], and returned as part of the reply to an `OBJ_SUB` request.

Publisher/Subscriber Messages published by the Blender add-on start with the AmbilinkID of the object they relate to. This allows each VST instance to filter out irrelevant messages based

[2]The process of Ambilink ID assignment will be discussed in more detail in 4.3.3.

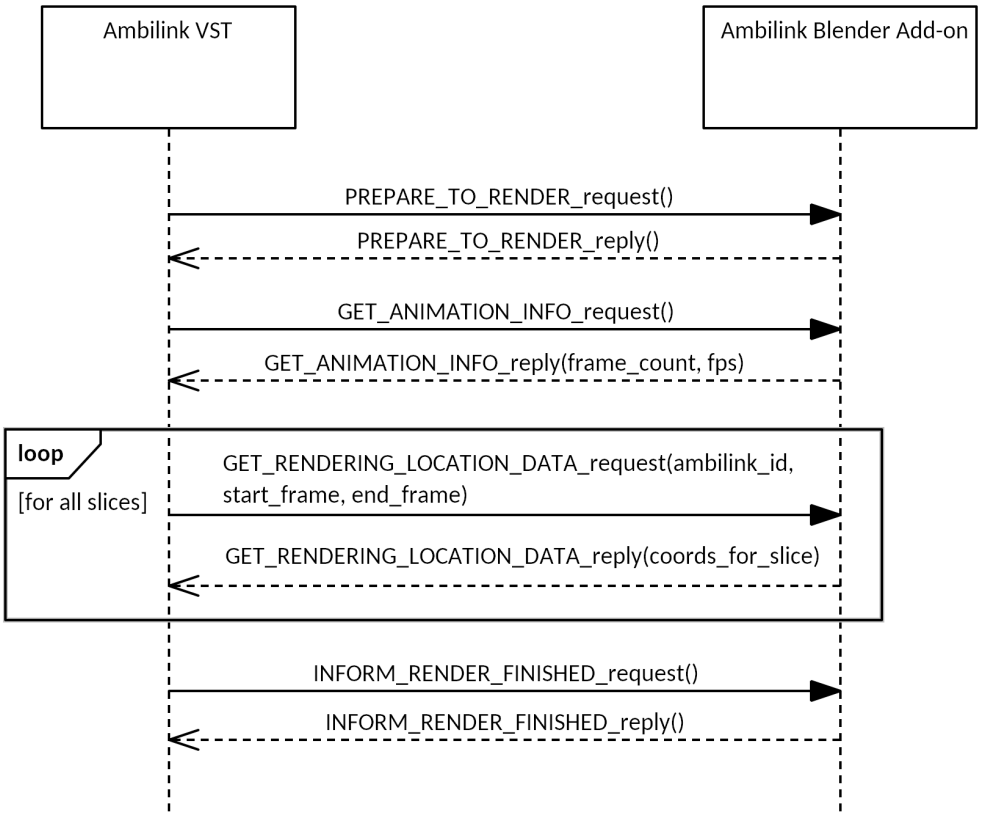

Figure 4.3 Sequence diagram illustrating communication in offline rendering mode. Image courtesy of the author.

on the first two bytes. After the Ambilink ID, a single-byte message type identifier follows. The Blender add-on publishes the following message types: `OBJ_POSITION_UPDATED`, `OBJ_RENAMED`, `OBJ_DELETED`. `OBJ_POSITION_UPDATED` messages include the location of the object in camera space. Using camera space coordinates allows to reduce the bandwidth consumed by these messages since only one set of coordinates needs to be transmitted. The direction and distance from the object to the camera are then calculated by the VST plugin.

4.1.2 Offline rendering

As can be seen from the sequence diagram in figure 4.3, communication during offline rendering (when the audio is being rendered to a file) significantly differs from communication during real-time playback. Pub/Sub is not used, instead, every interaction is initiated by the VST. The `PREPARE_TO_RENDER` request includes no additional data and serves as a notification for the Blender plugin that a render is about to begin. The Blender plugin will receive this message from each VST instance, but only the first one will have an effect. While this may seem wasteful, it greatly simplifies the protocol. Otherwise, all the VST instances would have to communicate with each other, agreeing on a single "master" instance. Once a `PREPARE_TO_RENDER` message is received, the Blender plugins stops publishing `OBJ_POSITION_UPDATED` messages until the render ends, and invalidates the cache used during the previous offline render.

To render the audio, a VST instance needs to know the object's camera space coordinates at each frame of the animation. Instead of requesting all the animation data at once, the VSTs first

request the number of frames and FPS of the animation, and then get the data in slices using multiple `GET_RENDERING_LOCATION_DATA` requests. (Denoted in fig. 4.3 by the "loop" fragment.) This is done because the DAW's GUI thread might be blocked until the first audio sample buffer is processed. [3] Since Ambilink would not have been able to process the first buffer until all the rendering data is received, such an implementation would have made the GUI of some DAW's unresponsive for unacceptable periods when entering rendering mode. Getting rendering data in slices also allows audio processing and IPC to be performed in parallel.

Because the animation may be very long, there is a limit to the number of slices held in memory by each VST instance. If the memory limit is reached, the VST instance waits for the audio thread to consume the previous slice of location data before fetching the next one.

4.2 VST plugin

The Ambilink audio plugin is developed in C++ using the JUCE framework and targets Steinberg's VST3 interface[4]. It outputs ambisonics up to fifth order (The reasons behind this limitation are described in 4.2.6.) with ACN channel ordering and N3D or SN3D normalisation (configurable via the settings screen). The code base is multiplatform, but, at the time of writing this thesis, the build system is only operational on Linux. Documentation comments are used throughout the code base; documentation can be generated using Doxygen (see `documentation/Doxyfile` on attached media).

4.2.1 Third-party libraries

While developing a VST plugin without a specialised framework is possible, it is highly time consuming, and takes a lot of "reinventing the wheel", that's why Ambilink uses JUCE. It provides all of the base functionality required for building audio plugins - a relatively convenient wrapper on top of the base VST interface, a GUI system, audio processing primitives, as well as other features not as pertinent to this specific project. Developing the Ambilink VST with JUCE also opens the opportunity to provide the plugin in other formats later on without requiring major changes to the codebase. Another important factor is that the JUCE framework has a large user base, increasing the number and availability of learning resources, which was of great help, since I had no prior experience developing audio processing plugins. Finally, it is dual-licensed under GPLv3 and a proprietary commercial license, enabling it's use in this project. Besides JUCE, the VST plugin component also uses the Spatial Audio Framework ([74]) to calculate the spherical harmonics coefficients for a given direction of sound, as well as vector data types and some useful trigonometric functions from GLM ([75]).

4.2.2 Software architecture

An audio plugin written with JUCE consists of two main parts. An `AudioProcessor` class deriving from the `juce::AudioProcessor` virtual base class that performs the actual audio processing, receives audio, MIDI, and other data from the host software. The plugin's GUI is implemented in the `AudioProcessorEditor` class that derives from `juce::AudioProcessorEditor`. An instance of the `AudioProcessor` class always exists while the plugin is loaded. `AudioProcessorEditor`, on the other hand, is only instantiated once the plugin's GUI editor is open, and is usually[5]

[3] This is the case for at least one DAW - Reaper.

[4] Since VST3 is the only version of the interface that is currently supported by Steinberg, I've been mostly referring to it as "VST" up to this point and will continue to do so unless differences between versions of the standard are discussed.

[5] Such details may differ between different DAWs.

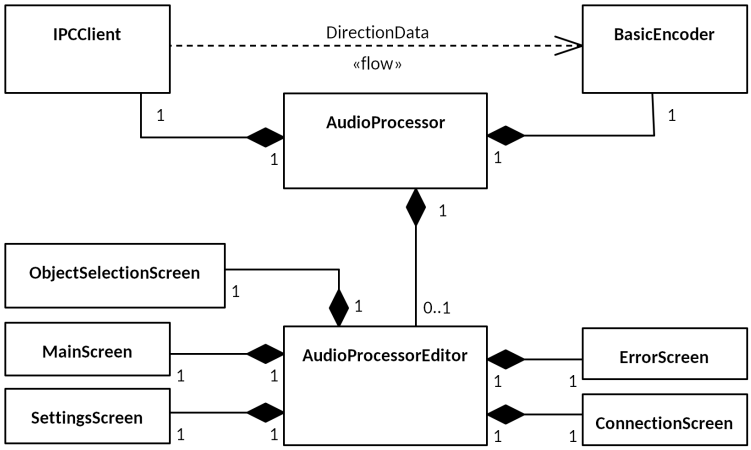

Figure 4.4 Simplified class diagram for the Ambilink VST plugin. (This diagram only includes the most important classes, and is not by any means an exhaustive representation of the whole architecture.) Image courtesy of the author.

destroyed as soon as it is closed.[6]

Figure 4.4 shows the high level structure of the Ambilink VST plugin. The `AudioProcessor` class holds instances of the `IPCClient` and the `BasicEncoder` classes. The `IPCClient` class handles communication with the Blender plugin, and the `BasicEncoder` class performs the ambisonic panning. (The two classes have no knowledge of each other, the direction data is passed through the `AudioProcessor` class.)

4.2.3 IPC

The `IPCClient` class is implemented as a state machine. The states and transitions can be seen in figure 4.5. The "Error" state and corresponding transitions, as well as transitions to the "Disconnected" state are omitted from the diagram to keep it comprehensible. Whenever connection to the Blender plugin is lost (e.g. if Blender is closed, or the server is stopped by the user), the IPC client transitions to the "Disconnected" state, regardless of what state it was in previously. Whenever an exception not caused by connection loss occurs in any of the states, the VST transitions to a special "Error" state. The GUI then displays an error message, and a button allowing to transition back to the "Disconnected" state. (The plugin will then automatically transition to either "Connected" or "Subscribed", depending on what state it was in before the error occurred.) While the users should hopefully never be able to experience the error screen by themselves, having such a fail-safe allows the plugin to recover gracefully from situations that would have otherwise lead to a crash.

`IPCClient` spawns two threads, one for each communication pattern - a requestor thread, and a subscriber thread. Each class implementing a state has specific functions to send requests to the Blender plugin, and to receive messages published using Pub/Sub. These functions are called by the main `IPCClient` class from the respective threads.

To reduce coupling, other parts of the plugin control IPC using a custom event system. For example, the `ObjectSelectionScreen` class can instruct the `IPCClient` to subscribe to an object by publishing an event. The event is passed to `IPCClient` through the `AudioProcessorEditor` and `AudioProcessor` classes. This is done to further reduce coupling and allow some events to be "intercepted" and acted upon by classes other than the `IPCClient`.

[6] A VST plugin does not require a custom GUI, DAW's implement a default GUI for the various audio parameters defined by the VST interface. This was not a viable option for Ambilink.

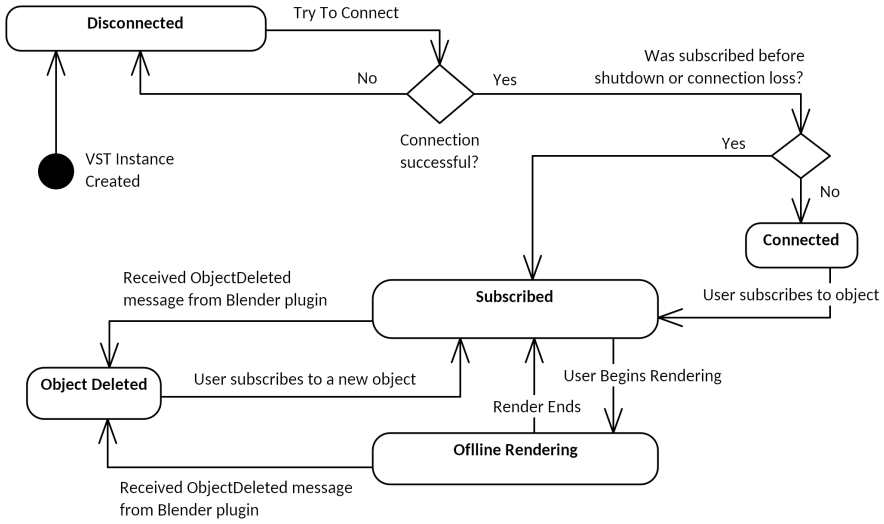

Figure 4.5 State machine diagram describing the way IPC is implemented on the VST side. Image courtesy of the author.

Once an event reaches the `IPCClient`, it is added to a lock-free queue implemented using atomic variables (The implementation used in Ambilink is inspired by Fabian Renn-Giles's farbot library [76].). Thread-safety is required since the events are added and removed by different threads; the lock-free implementation reduces overhead of posting an event. `IPCClient`'s requestor thread consumes events from the queue, and passes them to the currently active state, the state class can then perform any required action, e.g. sending requests or initiating a transition to a different IPC state.

Upon transitioning to the "Subscribed" state, the subscriber thread is spawned and begins consuming messages published by the Blender plugin. When an `OBJ_POSITION_UPDATED` message is received, it calculates the direction and distance to the object from the camera space coordinates, and updates the values of atomic variables, that are then read by the audio thread and passed to the encoder.

4.2.4 Multithreading in a real-time context

Real-time audio processing is quite different from most problems usually encountered by programmers. In case of VST plugins, the host software manages the high-priority threads used for audio processing. The plugin's processing function (`AudioProcessor::processBlock()` in case of plugins using JUCE) is repeatedly called from the audio thread, each time with new audio and/or MIDI data for the plugin to process. [77] The main caveat is that this callback has to always finish in time. Any delay, even if it occurs extremely rarely will result in audible artifacts in the audio. [78][79]

For example, if the sample rate of the host software is $44.1kHz$, and the size of the audio buffers passed to the plugin is 128 samples, then the `AudioProcessor::processBlock()` function must take less than $(1/44100) * 128 = 0.0029s$ to execute, which is just under $3ms$[7]. [79] This limitation rules out any operation that doesn't have a reliable time limit. Examples of such operations include accessing files, any form of network communication (or IPC), and, importantly, using mutex-based thread synchronisation techniques. Practically any call to a function not

[7]The sample rate can be significantly higher (usually in the range of $44.1 - 192kHz$). Buffer sizes can go down to 32, or even 16 samples. (Speaking from personal experience using DAWs and working with audio.)

implemented by the program should be treated as not real-time safe unless the documentation explicitly states otherwise. [78] All of these operations can of course be performed on other threads, but if the data needs to be passed to the audio thread it has to be done without using traditional synchronisation primitives. Thankfully, even when using locks is out of the question, race conditions can be avoided with good design and some tricks. In simpler cases, using an atomic value that is modified by one thread and read by another is sufficient, but various thread-safe lock-free data structures, such as single consumer single producer FIFO queues can also be implemented (mostly by clever use of atomic variables). For further information on real-time audio programming I would recommend the two part talk by Fabian Renn-Giles and Dave Rowland [78][80], which I used as the main source of information on this topic. Fabian Renn-Giles's open source farbot library [76] which implements many real-time safe data structures may also be of interest to the reader.

In case of Ambilink, the data used in the audio processing callback has to be passed from the Pub/Sub thread. Since the data is small[8], this is realised using a pair of atomic variables that are ensured to be lock-free by static assertions (`std::atomic<T>::is_always_lock_free`). The atomic variables are stored by `IPCClient` and read by the audio thread whenever required. In offline rendering mode, however, a lock is utilised so the audio thread can get access to the instance of the `OfflineRendering` state[9]. This does not really pose a problem since the audio is not played back in real time in this case and because locking the mutex will be uncontested unless a communication error occurs and IPC needs to switch to a different state. It is however undebatable that this part of the implementation could benefit from optimisation in the future.

4.2.5 Encoder

As mentioned previously, `BasicEncoder` is the class that handles ambisonic panning of the incoming audio. On each invocation of the `AudioProcessor::processBlock()` function, the encoder's direction and distance parameters are updated. In real-time mode, the latest value processed by the `IPCClient` is used. In offline rendering mode, direction and location at the respective time in the animation are used.

A version of the encoding algorithm used by Ambilink written in pseudocode can be seen in listing 2. (The actual C++ code is too verbose to be included as a listing.) The ambisonic-encoded buffers are produced by matrix multiplication of the input audio sample vector $\boldsymbol{x}$, and the vector of spherical harmonic coefficients $\boldsymbol{s}$ that is calculated using the `getRSH_recur()` function from the Spatial Audio Framework library ([74]) based on the direction received from `IPCClient`. For first-order ambisonics, this would look as follows:

$$\boldsymbol{s}^T\boldsymbol{x} = \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} = \begin{bmatrix} W_1 & W_2 & \cdots & W_n \\ X_1 & X_2 & \cdots & X_n \\ Y_1 & Y_2 & \cdots & Y_n \end{bmatrix} \tag{4.1}$$

The same calculation is performed twice, with the spherical harmonic coefficients and gains from the previous and current method calls, and the resulting buffers are then cross-faded to avoid audible artifacts in the audio when the direction or distance changes rapidly.

[8] A struct of two `float`s for the direction and a single `float` for the distance.

[9] The mutex is only used to prevent the `IPCClient` from switching to a different state, which would result in destruction of the `OfflineRendering` object. Synchronisation between the Req/Rep thread and the audio thread, once the audio thread does gets access to the state, is performed using atomic variables.

```
def encode(input_buffer, ambisonic_order, direction, distance,
           max_distance, distance_attenuation_curve)
{
    curr_sh_coefficients =
        calc_spherical_harmonic_coefficients(ambisonic_order, direction)
    gain = gain_from_distance(distance, max_distance,
                              distance_attenuation_curve)

    encoded_with_prev_direction =
        matmult(_prev_sh_coefficients, input_buffer.with_gain(_prev_gain))
    encoded_with_curr_direction =
        matmult(curr_sh_coefficients, input_buffer.with_gain(gain))

    _prev_sh_coefficients = curr_sh_coefficients
    _prev_gain = gain

    return crossfade(encoded_with_prev_direction,
                     encoded_with_curr_direction)
}
```

Code listing 2 The encoding algorithm used by the `BasicEncoder` class in pseudocode. Member variables are prefixed with underscores.

4.2.6 Limitations

Due to limitations of the VST3 format, Ambilink only supports ambisonics up to fifth order. The VST3 interface requires the plugin's main output bus[10] to conform to one of the output channel sets explicitly specified by Steinberg [11]. The VST3 interface still only officially supports ambisonic channel sets up to third order, but JUCE manages to support fourth and fifth order ambisonics, potentially using some undocumented behaviour of the VST3 interface. Curiously, the previous version of the interface - VST2, supported so-called "discrete" layouts for a plugin's main output, which basically allowed to use any number of output channels without explicitly labeling what CBA or SBA format the plugin's output conforms to. Discussion regarding the issue on Steinberg's forums ([82]) has started as early as April 2018 and continues as of February 2023. Development team members have been intermittently discussing possible solutions and proposals in the thread, but the issue has not been resolved as of February 2023.

4.3 Blender add-on

The Ambilink blender add-on is written in Python (as all Blender add-ons are) and supports Blender version 3.4.1 - the latest one at the time of development. In contrast to the VST, the Blender add-on only uses one external module - pynng, the previously mentioned Python wrapper for the NNG library.

[10] In this context, the word "bus" refers to a set of audio channels.

[11] In the VST3 interface, these are referred to as "speaker arrangements". The current definitions can be found here [81], but the page is likely to be updated as new versions of the SDK are released.

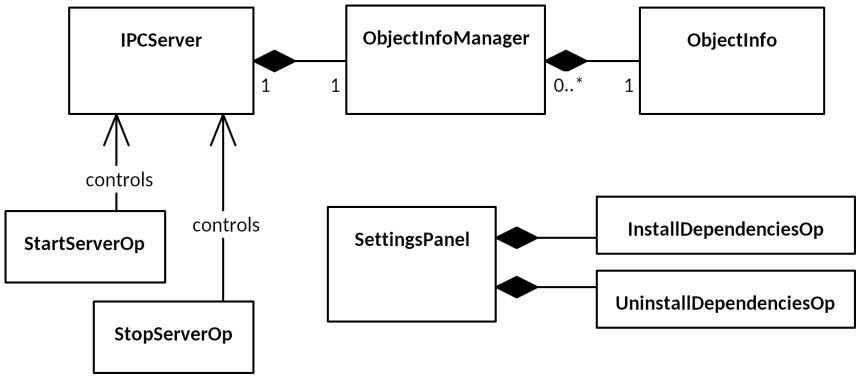

Figure 4.6 Simplified class diagram for the Ambilink Blender add-on. (For the sake of conciseness, the diagram does not include some helper classes, e.g. custom exception types.) Image courtesy of the author.

4.3.1 Software architecture

As can be seen in figure 4.6, the architecture of the Blender add-on is simpler than that of the VST plugin. There are two main classes - `IPCServer` and `ObjectInfoManager`. The `IPCServer` class handles all details of the IPC protocol - decoding incoming requests, correctly structuring and encoding reply data, etc. It holds an instance of `ObjectInfoManager` that takes care of the "business logic" of the add-on, but has no knowledge of the IPC protocol itself. `ObjectInfoManager`'s responsibilities include keeping track of all objects with at least one subscriber, informing the `IPCClient` class (via callbacks) whenever the objects are renamed or deleted, providing object locations and other information about the Blender scene, such as animation length and FPS.

4.3.2 Multithreading considerations

The add-on needs to continuously reply to requests and publish messages for subscribed VST instances. Ideally this would be done in a separate thread, but Blender's Python scripting implementation is not thread safe. Thus, all access to Blender data needs to be performed on the thread the add-on is launched from, which also happens to be the GUI thread[12]. While spawning another thread and passing commands using a thread-safe queue or a similar mechanism is possible, such an approach didn't make much sense for Ambilink, especially for the initial implementation. Because the majority of IPC commands require accessing Blender data, only a limited number of operations, such as encoding and decoding message data, could have been performed in a separate thread, so the performance improvement (if any) provided by a multithreaded implementation wouldn't have been significant enough to justify the added complexity.

Instead of using a separate thread for IPC communication, messages are sent and received by a method that is called on the main thread multiple times per second. This is achieved using the `StartServerOp` ("Start Server For Ambilink VST") modal operator. [13] Once it is invoked by the user, it creates an `IPCServer` instance and schedules a timer that calls it's `serve` method multiple times per second. (The update frequency can be configured via add-on settings, the default is 30 Hz.) On each invocation, the `serve` method replies to all pending requests sent by VST instances and publishes updates for any registered objects[14].

[12] This means that long operations performed by the add-on will result in Blender's GUI being unresponsive.
[13] Modal operators can continue running in the background after being invoked by the user.
[14] I will sometimes refer to objects with at least one subscribing VST instance as "registered objects" for the sake of conciseness.

```
    def get_scene_objects():
        return bpy.context.scene.objects

    def lookup_by_name():
        return get_scene_objects().get(TEST_OBJ_NAME)

    def lookup_by_prop():
        for obj in get_scene_objects():
            if obj.get(TEST_PROP_NAME) == DESIRED_PROP_VALUE:
                return obj
```

Code listing 3 Functions used to compare performance of looking up Blender objects by name and by value of a custom property. Test results are presented in table 4.1.

	A single object has the custom property assigned	All objects have the custom property assigned
`lookup_by_name()`	20.42s	19.76s
`lookup_by_prop()`	96.88s	128.85s
Performance decrease	4.74x	6.51x

Table 4.1 Test results for performance of Blender object lookup by name and by value of a custom property using lookup functions from code listing 3. The times in the table correspond to the total time it took to find a specific Blender object 100000 times. The tests were performed using the `timeit` Python module.

4.3.3 Overcoming Blender API limitations

Blender's Python API is realised via the `bpy` Python module that is accessible to add-ons and scripts. It allows to access and modify most aspects of the 3D scene including the names and locations of objects, so, at first glance, keeping track of this data should be simple. Objects in the scene can easily be accessed by their name, e.g. `bpy.context.scene.objects['Name']`. However, as mentioned previously, the name of the object is not a viable unique identifier for Ambilink's use case since an object can be renamed after a subscription is established, and an Ambilink-specific identifier (Ambilink ID) is used instead.

Unfortunately, as evident from the test results presented in table 4.1, finding an object by it's Ambilink ID is much slower than using the object's name. The times in the table correspond to the total time it took to lookup a specific Blender object 100000 times using the `lookup_by_name()` and `lookup_by_prop()` functions (code listing 3). There was a total of 2380 objects in the Blender scene, and the same object, located at index 1871 in the `bpy.context.scene.objects` collection, was being looked up each time. Two tests have been performed. In the first test, there was only one object in the scene that had the custom property set. In the second test, all objects in the scene had the custom property set, but only one of them had it set to `DESIRED_PROP_VALUE`. The Python script and the Blender project used to perform the tests can be found in the attached media. The results show that lookup using the value of a custom property takes roughly 5 times longer than lookup by name. This can be explained by the fact that getting an object by it's name is handled directly by Blender's C++ implementation, and finding an object by the value of a custom property requires iterating all objects in the scene from Python code. [15]

[15] I have initially expected Blender's implementation to use a hashmap for looking up objects by name, but the

In an ideal case, the Blender add-on would avoid the need to constantly search the list of all objects in the scene by storing a reference[16] to the actual Python object[17] returned by Blender's API. The situation is however complicated by the fact that references to Blender data returned by `bpy` can be invalidated under some circumstances. The most common case when this occurs is whenever the Undo/Redo functionality is used.

To maximise performance, Ambilink uses a combination of storing object references returned by `bpy`, lookup by name, and lookup by Ambilink ID. When a VST instance subscribes to a previously unregistered object, an Ambilink ID is generated and stored in Blender's object data as a custom property. The Python object (reference) returned by Blender, the object's name, and the assigned Ambilink ID are stored in an instance of the `ObjectInfo` class. The name stored by the Ambilink add-on is kept up to date even if the object is renamed by subscribing to updates of it's "name" property using the `bpy.msgbus.subscribe_rna` function. Whenever a property of a Blender object needs to be accessed, the following steps are taken:

1. Try to access the object through the stored reference.
2. If the previous step fails, try to find the object by name.
3. If both previous steps fail, try to find the object by Ambilink ID.

If all of the steps listed above fail, the Blender object is treated as deleted and an `OBJ_DELETED` message is published. Additionally, the add-on registers handlers that are called by Blender on Undo/Redo operations. This allows to keep the stored references valid and avoid the need to lookup objects by name or Ambilink ID most of the time.

While fairly complicated, the described approach minimises the time it takes to access properties of registered objects by using the fastest approach under most circumstances and only falling back to the slower options if needed. The slowest option - lookup by Ambilink ID - should only ever be used in the Undo/Redo handlers, when the operation being undone or redone is renaming an object, and, even in that case, only for one object, because multiple objects can't be renamed as part of one Undo or Redo operation. In all other cases, Blender should invoke the object name update handler registered by each `ObjectInfo` instance, keeping the name stored by the Ambilink add-on up to date, and ensuring lookup by name succeeds.

4.3.4 Limitations

Using custom properties is the only way to attach custom data to Blender objects. For most use cases, this approach works great. The only caveat is, when a Blender object is duplicated it's custom properties are duplicated as well. This is a problem for Ambilink since duplicating an object with at least one subscribing VST instance results in a Blender scene containing two objects with identical Ambilink IDs. This results in what the C++ standard would call undefined behaviour.

The current version of Ambilink does not attempt to solve this in any way, but a potential future workaround would be to override all of Blender's built-in operators that allow the user to duplicate an object resulting in duplication of custom properties. Unfortunately, there are at least four such operators. Two of them are specific to object duplication (`bpy.ops.duplicate_move` and `bpy.ops.duplicate_move_linked`), but the same result can be achieved by copying and pasting an object either from the 3D viewport (`bpy.ops.view3d.pastebuffer`) or the object list (`bpy.ops.outliner.id_paste`).

`RNA_property_collection_lookup_string` function defined in the `rna_access.c` file ([83]) seems to simply iterate over all items in the collection. (Which seems to be more than fast enough.)

[16] Variables in Python "are simply names that refer to objects" ([84]), meaning that assigning an object to a variable always results in storing a reference to that object and never in copying the object itself.

[17] There may be some confusion between objects as in object-oriented programming, and 3D objects in the Blender scene. I will try to avoid the ambiguity by always referring to the first kind as "Python objects".

Chapter 5

Testing and evaluation of the implemented software

This chapter evaluates the implemented software from a technical and user experience perspective. Advantages and disadvantages of the solution are discussed, and possible paths for future improvement are laid out.

5.1 Testing

5.1.1 Ambisonic panning correctness

Since I do not have a surround speaker array at my disposal, I have used a the the BinauralDecoder plugin from the IEM plugin suite [58] to judge the correctness of spatial panning. Subjectively, the software is performing correctly, and the audible direction to the sound source corresponds to the direction from the camera to the object in the Blender scene.

To complement these subjective tests, and to verify that the ambisonic panning is not being performed incorrectly in other aspects, that may not result in obvious audible differences, I've chosen to compare the output of the Ambilink VST to the output of the MultiEncoder plugin from the IEM suite [58].

To achieve this, I've created a simple Blender scene with objects positioned at specific positions relative to the camera (figure 5.1). Switching to Reaper, a simple test project has been used with a single mono track containing an audio file of a pure sine wave (with constant volume) routed to the two multichannel tracks - one with the Ambilink VST, and one with the IEM MultiEncoder (figure 5.2). (The frequency composition of the input signal does not matter for the purposes of this test, but it is of course essential that the input signal is identical for both encoders.)

In comparing the output of the encoders, the IEM EnergyVisualizer [58] plugin has been used to visualise the energy distribution of the ambisonic signal on the surface of a sphere.[1] For each test case, a different object from the Blender scene was selected in Ambilink, and the same direction was set in the IEM MultiEncoder using the plugin's GUI. The SN3D normalisation convention has been used in both encoders, and both visualiser instances. Distance attenuation has been disabled in Ambilink, so the output signal's gain is identical.

As can be seen in figure 5.3 both instances of the IEM EnergyVisualizer produced identical

[1] I have to mention that the EnergyVisualizer plugin (v1.0.3) has proven to be extremely unstable, consistently crashing Reaper whenever it's GUI is closed. The crashes also occurred when no instances of the Ambilink VST or any other plugins were loaded.

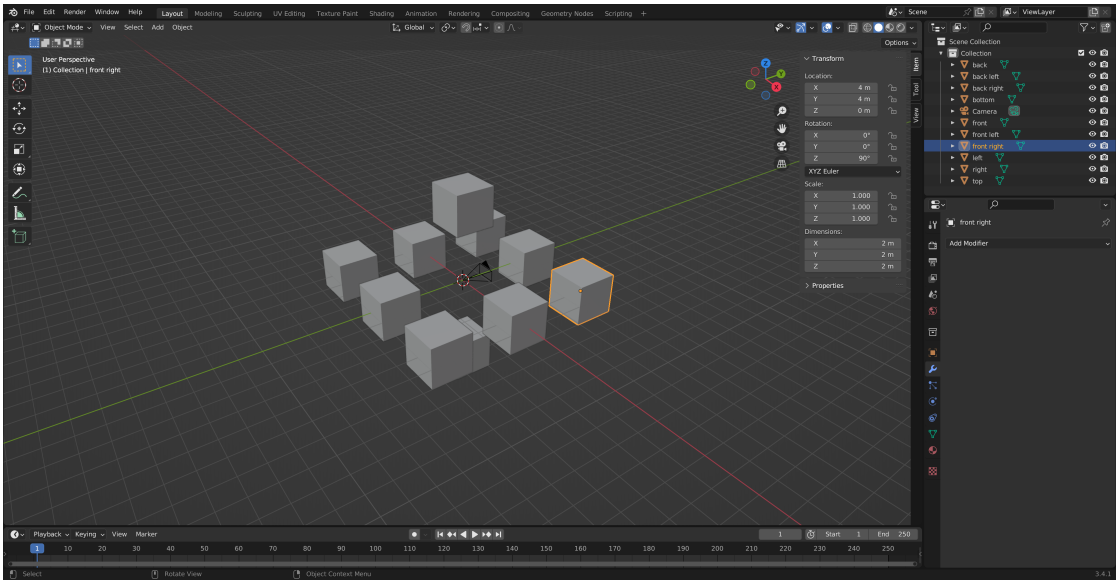

Figure 5.1 The Blender scene used for comparing Ambilink's output to the output of the IEM MultiEncoder plugin. Image courtesy of the author.

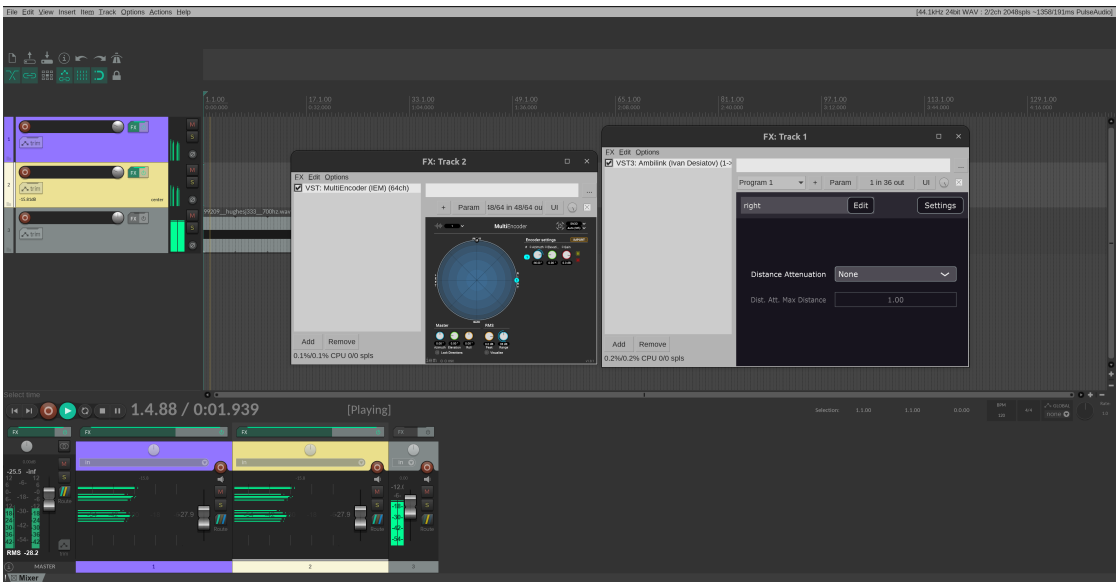

Figure 5.2 Reaper project used for testing with the Ambilink and the IEM encoder plugins open. Image courtesy of the author.

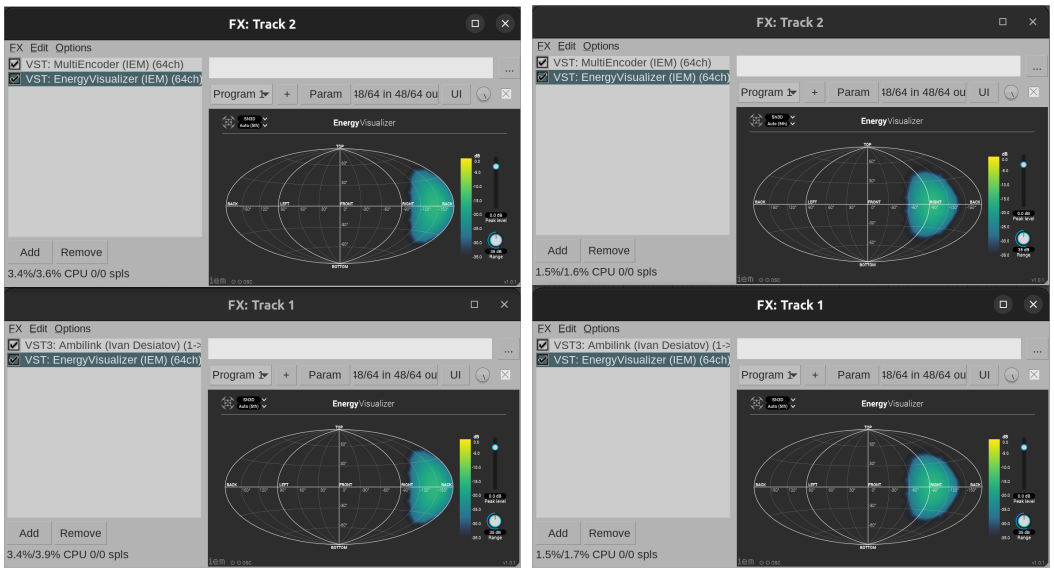

Figure 5.3 The ambisonic output produced by Ambilink (bottom row), and the IEM MultiEncoder (top row), visualised using the IEM EnergyVisualizer plugin. Panning directions are identical for each column. Image courtesy of the author.

visualisations for identical panning directions. Other panning directions have been tested besides the two presented in figure 5.3, and the visualisation of the output of the two encoders has been identical in all cases. A screen capture demonstrating that the movement of an object in Blender matches the visualisation produced by the IEM EnergyVisualizer plugin can be found on the attached media in the `demo_videos_and_renders/` directory under the filename `8_iem_visualiser_realtime_demo.mkv`.

5.1.2 Overall functionality

Automated integration tests of both components would have been the ideal approach for testing interprocess communication and other aspects of the solution. Unfortunately, both plugins require host software to run. Because of that, creation of automated integration tests would have taken up a disproportionate amount of the overall development time. Instead, I have settled on testing the solution manually.

Manual testing has been performed throughout development. A mid-range laptop from 2018 with an AMD Ryzen 5 2500U processor running Ubuntu 22.04.1 has been used for testing. Blender 3.4.1 and Reaper 6.73 have been used to host the plugins. Several screen recordings of performed tests are available on the attached media in the `demo_videos_and_renders/` directory. It should be noted that screen capture was performed locally (without using an external capture card) resulting in additional CPU load. The following paragraphs will summarise the results of the tests. Where applicable, the filenames of screen captures included on the attached media will be referenced by using a monospaced font, e.g. `1_real_time.mkv`.

General VST functionality Picking a Blender object and changing other parameters (Ambisonics order, normalisation type, distance attenuation function and distance attenuation maximum distance) functions correctly. (`0_params_and_object_sub.mkv`) So does saving and loading of the VST plugin's state. Upon reopening the Reaper project, parameter values are consistent to the previous state, and the VST automatically resubscribes to the previous object. I should

note that if the Blender object was renamed while the VST wasn't running, it would show up as deleted on reconnection because Ambilink IDs are only utilised when the VST instance is subscribed. (`4_vst_save_load.mkv`) When a blender object is renamed or deleted, the change is reflected in the VST GUI. (`7_rename_delete.mkv`)

IPC stability The IPC connection has proven to be stable - no spontaneous disconnects have occurred during testing. This was expected since no external factors except system load should be able to affect the interprocess communication. The VST plugin reacts quickly whenever the Blender server is launched, switching to the "Connected" state and updating the GUI practically immediately. The delay before the plugin GUI updates after stopping the Ambilink server or closing Blender is longer, up to multiple seconds, but still acceptable for the use case. (`3_disconnect_reconnect.mkv`)

IPC latency I have performed two latency tests. The first one with a single instance of the Ambilink VST, and the second one with 20 instances all subscribed to different Blender objects[2]. The Blender scene used for the test contained more than 2000 objects overall. Latency has been calculated from screen captures by counting the number of frames between the first frame where the timeline position is updated in the Blender GUI and when a change in panning direction is visible on the IEM EnergyVisualizer instance placed after one of the VSTs. The screen capture has been performed at 30 frames per second.

For the first test, the delay has been around three frames, which is around $100ms$. While this is a relatively significant delay, it is sufficiently low for the use case, and latency has not been an issue during normal use. (`5_latency_test_1.mkv`) Surprisingly, the second test, where the overall load on the system was significantly higher due to the number of active Ambilink instances, all encoding fifth order ambisonics, had the same result of around three frames. (`6_latency_test_2.mkv`) This may be caused by the screen recording software skipping frames, but, even if the real latency was double the measured latency, this would have been a good result considering the increased load. Although these tests don't account for several factors, such as the latency introduced by the IEM EnergyVisualizer plugin, they provide an upper bound that is precise enough to prove that IPC latency is not detrimental to the overall usability of the software.

Blender add-on performance Another factor to consider is how the Ambilink add-on affects Blender performance. In my testing[3], starting the Ambilink server had no noticeable effect on the playback FPS in most cases. To keep the system load as consistent as possible, the Ambilink VST instances were always running in the background (encoding is still performed when disconnected)

In a scene with 70 objects and a very simple animation of some cubes[4], running the Ambilink server at the maximum update frequency of 120Hz[5] with 15 connected VST instances (set to first order to reduce encoding load), resulted in no visible FPS dips. Adding five more VST instances did however result in the overall framerate being lower due to the added CPU load. This made periodic dips down to around 22-23 FPS from the target 24 noticeable. These dips are likely caused by the Ambilink server processing IPC requests and only appear when the CPU is already under significant load. Overall, with a sufficiently performant machine (as opposed to the laptop used for testing), the performance impact of running the Ambilink server should not pose any issues.

[2]Having 20 instances all subscribed to a single object wouldn't have increased the load since only one `OBJECT_POSITION_UPDATED` message would have been published on each update.

[3]These tests have not been recorded, as the screen capture induced too much additional CPU load.

[4]Because of the relatively low-end hardware used for testing, and the Ambilink VSTs running in the background, a more demanding scene was not required to get on the verge of dipping below the target framerate.

[5]This is overkill for normal use, update frequencies in the range of 30-60Hz are sufficient.

Offline rendering During development, many rendering tests have been performed under various conditions - different animation start and end frames (including extremes, such as a single frame animation), different numbers of VST instances, different numbers of objects in the scene. A screen capture of two rendering tests can be found under the filename `2_offline_render.mkv` on the attached media. In all performed tests, offline rendering has functioned correctly - panning directions are synced with the position of objects in the animation.

A short demo animation has been produced and is included on the attached media under the `demo_videos_and_renders/demo_animation/` directory. The 3D animation has been rendered with Blender, the audio has been rendered using Reaper and Ambilink. Only a binaural version could be included on the attached media due to the large size of the full 36-channel fifth order ambisonics file. No other plugins or effects were applied except for the Ambilink encoders on each of the five channels (3 channels are used for objects that can be seen in the video, and 2 more for ambient sounds) and the IEM BinauralDecoder plugin used for decoding.

5.2 Future improvements

The implemented software is fully functional, fulfils the requirements defined in chapter 3 and the overall goal of the thesis. However, while no user experience testing has been performed, it is safe to say there is room for improvement in that regard. The following paragraphs list several unsatisfactory aspects of the user experience in order of descending importance and outline potential paths for future improvement.

Object restoration Several improvements can be made to the Blender add-on in terms of handling operations with objects. Besides the issues with duplicating objects described in 4.3.4, Ambilink also doesn't handle undoing the deletion of an object in a user-friendly way - if the user deletes an object and then undoes the change, the object will still show up as deleted in any previously subscribed VST instances. Both of these aspects are high priority targets for future improvement.

Parameter synchronisation It is likely that all ambisonics encoders used in a project will be using the same settings. Despite that, in the current version of Ambilink, the ambisonics settings (order and normalisation type) have to be manually set for each VST instance. I see two main approaches to solving this problem. One solution would be to initialise new VST instances with the last used settings - for example, if the user creates one instance and adjusts the ambisonics order, the next instance would automatically use the same value.

Another approach would be to add a GUI for managing VST instances. This could be implemented as part of the Blender plugin, or as a standalone application. This would allow the user to easily change the parameters of multiple Ambilink VST instances at once, which could be very useful in some cases. For example, the user could select a group of encoders handling background sounds with lower importance, and lower the ambisonics order, and then increase the order for more important sounds, allowing to easily distribute CPU usage by sound priority. If such a GUI was ever to be implemented, it could, of course, allow to control other parameters besides the ambisonics order. This would open a lot of opportunities for improving usability.

Timeline synchronisation Another useful feature that is missing from the current version of Ambilink is the ability to synchronise the playback position of the animation in Blender with the playback position in the DAW. That would make the real-time preview even more useful since the user would be able to easily preview the animation as a whole without rendering audio out. Unfortunately, this is not a trivial problem to solve, especially since the target playback FPS may not always be reached in Blender. Designing how such a synchronisation system would function from the user's perspective also poses quite a significant UX design challenge.

Panning offsets In some situations it could be desirable to offset the sound source's position from the Blender object's origin without affecting the 3D scene itself. This is not currently possible in Ambilink, but could be a useful future addition. A direction offset can however be achieved even with the current version by utilising an additional child object in Blender, giving it the required offset relative to the parent, and then using the newly created object to position the sound source. The disadvantage of this workaround is that it can result in extra objects cluttering the Blender scene. It also requires the sound designer or mixing engineer to switch their attention away from the digital audio workstation, so a "native" solution would be preferred.

Chapter 6
Conclusion

The first chapter of this thesis provides a broad overview of the spatial audio field, and describes the three main approaches to representing spatial audio information (CBA, OBA, SBA) and their viability for 360° video. Ambisonics in particular is described in more detail from a practical and theoretical perspective, as the approach most suitable for 360° video. The chapter also examines various methods of decoding ambisonics for playback on surround loudspeaker arrays and via headphones (using HRTFs). In the second chapter, various software solutions for spatial audio production are reviewed and their suitability for 360° video production is evaluated. Besides solutions specifically aimed at audio production, the possibility of using other types of software, such as game engines, is explored.

The next two chapters are dedicated to the design and development of Ambilink - a software solution that implements a VST3 ambisonic encoder, the panning direction of which is determined by the position of a 3D object in a Blender scene. The Ambilink Blender add-on provides data about objects in the Blender scene to instances of the VST plugin using interprocess communication. An object from the Blender scene can be selected using the VST plugin's GUI. The direction from the active camera to that object is then used as the panning direction for the ambisonic encoder. When playing audio in-DAW, the panning direction is updated in real time, allowing artists and sound engineers to immediately hear how adjustments made to the Blender scene affect the audio mix. The VST plugin is implemented using JUCE framework; NNG is used for interprocess communication. No analogous solutions exist at the time of writing this thesis.

Finally, the presented software is evaluated from a technical and user experience perspective. To demonstrate the correctness of the ambisonic panning algorithm, the output of the Ambilink VST was compared to the output of the IEM MultiEncoder plugin - a well-regarded solution developed at the Institute of Electronic Music and Acoustics. Several other aspects of the software are evaluated and it is concluded that the software meets the requirements defined in chapter three and fulfils the overall goal of providing an improved workflow for producing spatial audio for 360° video. Potential paths for improving the user experience in the future are also discussed, providing a starting point for future extension of the software.

6.1 Current state

The implemented solution fulfils the goal of providing an improved SBA production workflow for 360° (immersive) video. As intended, it is especially useful for 360° 3D animations - in this case, in addition to allowing the content creators to utilise Blender's 3D animation tools, it should significantly reduce the number of objects that require manual ambisonic panning; for large-scale

projects the time savings could be substantial.

The use cases are not however limited to 360° video. The implemented software can be used in any audio production where ambisonic audio may be desired. In music production, for example, Blender can be used as an improved user interface for spatial panning. Visualising the position of individual sound sources in a full 3D environment could be helpful in creating complex soundscapes. Blender's Python scripting capabilities and animation drivers can also be used to define the position of sounds procedurally, which opens up possibilities for utilising Ambilink in generative music.

6.2 Final words

Overall, I am content with the current state of the project. As stated in the previous section, the current version of Ambilink fulfils the goals of this thesis. The software's architecture is extendable and can serve as a foundation for an even more flexible solution. Because the VST plugin and the Blender add-on are connected via IPC, the resulting software is highly modular, and either of the components can be exchanged to make it serve a slightly different purpose. While I cannot say with certainty that I will continue development of this project myself, it's open-source nature should allow others to improve it beyond what I might have ever envisioned.

Bibliography

1. OLIVIERI, Ferdinando; PETERS, Nils; SEN, Deep. *Scene-Based Audio and Higher Order Ambisonics: A technology overview and application to Next-Generation Audio, VR and 360° Video* [online]. 2019. Tech. rep. European Broadcasting Union. Available also from: `https://tech.ebu.ch/docs/techreview/trev_2019-Q4_SBA_HOA_Technology_Overview.pdf`.
2. HERRE, Jürgen; HILPERT, Johannes; KUNTZ, Achim; PLOGSTIES, Jan. MPEG-H 3D Audio—The New Standard for Coding of Immersive Spatial Audio. *IEEE Journal of Selected Topics in Signal Processing*. 2015, vol. 9, no. 5, pp. 770–779. Available from DOI: `10.1109/JSTSP.2015.2411578`.
3. PETERS, Nils; SEN, Deep; KIM, Moo-Young; WUEBBOLT, Oliver; WEISS, S. Merrill. Scene-Based Audio Implemented with Higher Order Ambisonics (HOA). In: *SMPTE 2015 Annual Technical Conference and Exhibition*. 2015, pp. 1–13. Available from DOI: `10.5594/M001651`.
4. BRINKMAN, Willem-Paul; HOEKSTRA, Allart RD; EGMOND, René van. The effect of 3D audio and other audio techniques on virtual reality experience. *Annual Review of Cybertherapy and Telemedicine 2015*. 2015, pp. 44–48.
5. MELKOTE, Vinay; YEN, Kuan-Chieh; FELLERS, Matt; DAVIDSON, Grant; KUMAR, Vivek. Transform-domain decorrelation in Dolby Digital Plus. In: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2014, pp. 6949–6953. Available from DOI: `10.1109/ICASSP.2014.6854947`.
6. HOEKSTRA, ARD. *3D audio for virtual reality exposure therapy*. 2013. MA thesis. Delft University if Technology.
7. SCHUTZE, Stephan; IRWIN-SCHUTZE, Anna. *New Realities in Audio: A Practical Guide for VR, AR, MR and 360 Video*. 1st ed. Milton: CRC Press, 2018. ISBN 9781138740822.
8. COLEMAN, Philip; FRANCK, Andreas; FRANCOMBE, Jon; LIU, Qingju; CAMPOS, Teofilo de; HUGHES, Richard J.; MENZIES, Dylan; GÁLVEZ, Marcos F. Simón; TANG, Yan; WOODCOCK, James; JACKSON, Philip J. B.; MELCHIOR, Frank; PIKE, Chris; FAZI, Filippo Maria; COX, Trevor J.; HILTON, Adrian. An Audio-Visual System for Object-Based Audio: From Recording to Listening. *IEEE Transactions on Multimedia*. 2018, vol. 20, no. 8, pp. 1919–1931. Available from DOI: `10.1109/TMM.2018.2794780`.
9. SUN, Xuejing. Immersive audio, capture, transport, and rendering: a review. *APSIPA Transactions on Signal and Information Processing*. 2021, vol. 10, e13. Available from DOI: `10.1017/ATSIP.2021.12`.

10. SPORS, Sascha; WIERSTORF, Hagen; RAAKE, Alexander; MELCHIOR, Frank; FRANK, Matthias; ZOTTER, Franz. Spatial Sound With Loudspeakers and Its Perception: A Review of the Current State. *Proceedings of the IEEE*. 2013, vol. 101, no. 9, pp. 1920–1938. Available from DOI: 10.1109/JPROC.2013.2264784.

11. WANG, Jun; CENGARLE, Giulio; TORRES, Juan Felix; ARTEAGA, Daniel. *Adaptive panner of audio objects* [online]. U.S. Patent 10405120, Sep 3, 2019 [visited on 2022-03-26]. Available from: https://patents.justia.com/patent/10405120.

12. BREEBAART, Jeroen; CENGARLE, Giulio; LU, Lie; MATEOS, Toni; PURNHAGEN, Heiko; TSINGOS, Nicolas. Spatial Coding of Complex Object-Based Program Material. *Journal of the Audio Engineering Society*. 2019, vol. 67, no. 7/8, pp. 486–497. Available from DOI: https://doi.org/10.17743/jaes.2018.0067.

13. GERZON, Michael A. Periphony: With-Height Sound Reproduction. *Journal of the Audio Engineering Society*. 1973, vol. 21, no. 1, pp. 2–10.

14. QUALCOMM TECHNOLOGIES, INC. *Scene Based Audio: A Novel Paradigm for Immersive and Interactive Audio User Experience* [online]. 2015 [visited on 2022-03-31]. Tech. rep. Available from: https://www.qualcomm.com/media/documents/files/scene-based-audio-for-mpeg-h-whitepaper.pdf.

15. HERRE, Jürgen et al. From joint stereo to spatial audio coding-recent progress and standardization. In: *Sixth International Conference on Digital Audio Effects (DAFX04)*. Naples, Italy, 2004.

16. KELLER, Daniel. *Mid/Side Mic Recording Basics* [online]. [N.d.] [visited on 2022-03-31]. Available from: https://www.uaudio.com/blog/mid-side-mic-recording/.

17. ZOTTER, Franz; FRANK, Matthias. *Ambisonics: A Practical 3D Audio Theory for Recording, Studio Production, Sound Reinforcement, and Virtual Reality*. 2019. ISBN 978-3-030-17206-0. Available from DOI: 10.1007/978-3-030-17207-7.

18. *Microphone practice* [online]. [N.d.] [visited on 2022-04-01]. Available from: https://en.wikipedia.org/wiki/Microphone_practice.

19. DANIEL, Jérôme. *Représentation de champs acoustiques, application à la transmission et à la reproduction de scènes sonores complexes dans un contexte multimédia* [online]. 2000 [visited on 2022-04-23]. Available from: http://gyronymo.free.fr/audio3D/download_Thesis_PwPt.html#PDFThesis. PhD thesis. University of Paris VI.

20. RAFAELY, B. *Fundamentals of Spherical Array Processing*. Springer International Publishing, 2018. Springer Topics in Signal Processing. ISBN 9783319995601. Available also from: https://books.google.cz/books?id=KZlUuQEACAAJ.

21. ZOTTER, Franz; FRANK, Matthias. Ambisonics: A Practical 3D Audio Theory for Recording, Studio Production, Sound Reinforcement, and Virtual Reality. In: 2019, chap. 4, p. 68. ISBN 978-3-030-17206-0. Available from DOI: 10.1007/978-3-030-17207-7.

22. ZOTTER Franz, Dr. *Spherical Harmonics deg3* [online]. 2013 [visited on 2022-04-23]. Available from: https://commons.wikimedia.org/w/index.php?curid=30239736.

23. DICKINS, Glenn. *Soundfield representation, reconstruction and perception* [online]. 2003 [visited on 2023-02-13]. Available from: https://openresearch-repository.anu.edu.au/bitstream/1885/9728/8/02Whole_Dickins.pdf. PhD thesis. Australian National University.

24. ABERDEEN, Douglas; BAXTER, Jonathan. General matrix-matrix multiplication using SIMD features of the PIII. In: *European Conference on Parallel Processing*. 2000, pp. 980–983.

25. INTEL CORPORATION. *Intel® Intrinsics Guide* [online]. [N.d.] [visited on 2022-04-01]. Available from: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

26. ARM LIMITED. *Optimizing C Code with Neon Intrinsics* [online]. 2022 [visited on 2022-04-01]. Available from: https://developer.arm.com/documentation/102467/0100/?lang=en.

27. IGLBERGER, Klaus; HAGER, Georg; TREIBIG, Jan; RÜDE, Ulrich. High performance smart expression template math libraries. In: *2012 International Conference on High Performance Computing Simulation (HPCS)*. 2012, pp. 367–373. Available from DOI: 10.1109/HPCSim.2012.6266939.

28. BUFFONI, Louis-Xavier. Working with Object-Based Audio. In: [online]. 2016 [visited on 2022-04-05]. Available from: https://blog.audiokinetic.com/working-with-object-based-audio/.

29. BUFFONI, Louis-Xavier. *Ambisonics as an Intermediate Spatial Representation (for VR)* [online]. 2016 [visited on 2022-04-05]. Available from: https://blog.audiokinetic.com/ambisonics-as-an-intermediate-spatial-representation-for-vr/.

30. GOOGLE LLC. *Use spatial audio in 360-degree and VR videos* [online]. [N.d.] [visited on 2022-04-07]. Available from: https://support.google.com/youtube/answer/6395969?hl=en#zippy=%2Cspatial-audio-requirements.

31. MIDDLEBROOKS, John C. Chapter 6 - Sound localization. In: AMINOFF, Michael J.; BOLLER, François; SWAAB, Dick F. (eds.). *The Human Auditory System*. Elsevier, 2015, vol. 129, pp. 99–116. Handbook of Clinical Neurology. ISSN 0072-9752. Available from DOI: https://doi.org/10.1016/B978-0-444-62630-1.00006-8.

32. KOLARIK, Andrew J.; MOORE, Brian C. J.; ZAHORIK, Pavel; CIRSTEA, Silvia; PARDHAN, Shahina. Auditory distance perception in humans: a review of cues, development, neuronal bases, and effects of sensory loss. *Attention, perception & psychophysics*. 2016, vol. 78, no. 2, pp. 373–395. ISSN 1943-393X. Available from DOI: 10.3758/s13414-015-1015-1. PMC4744263[pmcid].

33. MOORE, David R; KING, Andrew J. Auditory perception: The near and far of sound localization. *Current Biology*. 1999, vol. 9, no. 10, R361–R363. ISSN 0960-9822. Available from DOI: https://doi.org/10.1016/S0960-9822(99)80227-9.

34. STRUTT, John William (3rd Baron Rayleigh). On Our Perception of the Direction of a Source of Sound. *Proceedings of the Musical Association* [online]. 1875, vol. 2, pp. 75–84 [visited on 2022-04-12]. ISSN 09588442. Available from: http://www.jstor.org/stable/765209.

35. MIDDLEBROOKS, John; GREEN, David. Sound Localization by Human Listeners. *Annual review of psychology*. 1991, vol. 42, pp. 135–59. Available from DOI: 10.1146/annurev.ps.42.020191.001031.

36. LETOWSKI, Tomasz; LETOWSKI, Szymon. Auditory Spatial Perception: Auditory Localization [online]. 2012 [visited on 2022-04-13]. Available from: https://apps.dtic.mil/sti/pdfs/ADA563540.pdf.

37. ZOTTER, Franz; FRANK, Matthias. All-Round Ambisonic Panning and Decoding. *Journal of The Audio Engineering Society*. 2012, vol. 60, pp. 807–820.

38. PULKKI, V.; HUOPANIEMI, J.; HUOTILAINEN, T. Dsp Tool for 8-Channel Audio Mixing. In: *Nordic Acoustical Meeting NAM 96, Helsinki, Finland, June 12-14, 1996*. Finland: Acoustical Society of Finland, 1996, pp. 307–314.

39. PULKKI, Ville. Virtual Sound Source Positioning Using Vector Base Amplitude Panning. *Journal of the Audio Engineering Society*. 1997, vol. 45, no. 6, pp. 456–466. ISSN 1549-4950.

40. HELLER, Aaron. *Ambisonic Decoders - Music 222, Stanford* [online]. 2021 [visited on 2023-01-05]. Available from: https://ccrma.stanford.edu/courses/222/lectures/14/Ambi-Decoders-Music222-2021.pdf.

41. FRANK, Matthias; ZOTTER, Franz; SONTACCHI, Alois. Producing 3D Audio in Ambisonics. In: *Proceedings of the 57th AES International Conference*. 2015.

42. SMITH, Steven W. *The Scientist and Engineer's Guide to Digital Signal Processing*. USA: California Technical Publishing, 1997. ISBN 0966017633.

43. MØLLER, Henrik. Fundamentals of binaural technology. *Applied Acoustics*. 1992, vol. 36, pp. 171–218. Available from DOI: 10.1016/0003-682X(92)90046-U.

44. LI, Song; PEISSIG, Jürgen. Measurement of Head-Related Transfer Functions: A Review. *Applied Sciences*. 2020, vol. 10, no. 14. ISSN 2076-3417. Available from DOI: 10.3390/app10145014.

45. NOISTERNIG, Markus; SONTACCHI, Alois; MUSIL, Thomas; HÖLDRICH, Robert. A 3D ambisonic based binaural sound reproduction system. *Advances in Engineering Software - AES*. 2012.

46. WENZEL, Elizabeth; ARRUDA, Marianne; KISTLER, Doris; WIGHTMAN, Frederic. Localization using nonindividualized head-related transfer functions. *The Journal of the Acoustical Society of America*. 1993, vol. 94, pp. 111–23. Available from DOI: 10.1121/1.407089.

47. MENDONÇA, Catarina; JA, Santos; CAMPOS, Guilherme; DIAS, Paulo; VIEIRA, Jose. On the adaptation to non-individualised HRTF auralisations: A longitudinal study. In: *Proceedings of the 45th AES International Conference*. 2012.

48. ANDERSEN, Jonas Siim; MICCINI, Riccardo; SERAFIN, Stefania; SPAGNOL, Simone. Evaluation of Individualized HRTFs in a 3D Shooter Game. In: *2021 Immersive and 3D Audio: from Architecture to Automotive (I3DA)*. 2021, pp. 1–10. Available from DOI: 10.1109/I3DA48870.2021.9610934.

49. BERGER, Christopher C.; GONZALEZ-FRANCO, Mar; TAJADURA-JIMÉNEZ, Ana; FLORENCIO, Dinei; ZHANG, Zhengyou. Generic HRTFs May be Good Enough in Virtual Reality. Improving Source Localization through Cross-Modal Plasticity. *Frontiers in Neuroscience*. 2018, vol. 12. ISSN 1662-453X. Available from DOI: 10.3389/fnins.2018.00021.

50. STEINBERG MEDIA TECHNOLOGIES GMBH. *VST - VST 3 Developer Portal* [online]. 2022 [visited on 2023-01-30]. Available from: https://steinbergmedia.github.io/vst3_dev_portal/pages/index.html.

51. *Up and Running: A REAPER User Guide v 6.74* [online]. 2023 [visited on 2023-02-15]. Available from: https://www.reaper.fm/userguide.php.

52. AVID TECHNOLOGY, Inc. *Pro Tools Reference Guide* [online]. 2021 [visited on 2023-02-15]. Available from: https://resources.avid.com/SupportFiles/PT/Pro_Tools_Reference_Guide_2021.3.pdf.

53. STEINBERG MEDIA TECHNOLOGIES GMBH. *3D Mixes for Ambisonics* [online]. 2017 [visited on 2022-04-21]. Available from: https://steinberg.help/cubase_pro_artist/v10/en/cubase_nuendo/topics/surround_sound/surround_sound_ambisonics_c.html.

54. STEINBERG MEDIA TECHNOLOGIES GMBH. *What is Nuendo: Discover All The Features* [online]. 2022 [visited on 2022-04-21]. Available from: https://www.steinberg.net/nuendo/features/.

55. STEINBERG MEDIA TECHNOLOGIES GMBH. *The DAW for VR, AR and Immersive Sound: Nuendo — Steinberg* [online]. [N.d.] [visited on 2023-01-13]. Available from: https://www.steinberg.net/nuendo/virtual-reality.

56. *ATK for Reaper* [online]. 2021 [visited on 2022-04-21]. Available from: https://www.ambisonictoolkit.net/.
57. KRONLACHNER, Matthias. *ambiX v0.2.10 – Ambisonic plugin suite* [online]. 2014 [visited on 2022-04-21]. Available from: http://www.matthiaskronlachner.com/?p=2015.
58. INSTITUTE OF ELECTRONIC MUSIC AND ACOUSTICS. *IEM Plug-in Suite* [online]. [N.d.] [visited on 2022-04-21]. Available from: https://plugins.iem.at/.
59. EPIC GAMES, INC. *Working with Audio — Unreal Engine Documentation* [online]. 2022 [visited on 2022-04-23]. Available from: https://docs.unrealengine.com/4.27/en-US/WorkingWithAudio.
60. UNITY TECHNOLOGIES. *Unity - Manual: Audio* [online]. 2022 [visited on 2022-04-23]. Available from: https://docs.unity3d.com/Manual/Audio.html.
61. KLINT, Renee; CRUZ, John. *A comprehensive guide to creating 360-degree game trailers using Unreal* [online]. 2020 [visited on 2022-04-21]. Available from: https://www.unrealengine.com/en-US/tech-blog/a-comprehensive-guide-to-creating-360-degree-game-trailers-using-unreal.
62. *VR Panorama 360 PRO Renderer — Video — Unity Asset Store* [online]. 2021 [visited on 2022-04-21]. Available from: https://assetstore.unity.com/packages/tools/video/vr-panorama-360-pro-renderer-35102#reviews.
63. AUTODESK INC. *Add Audio To Your Animation — Maya 2022 — Autodesk Knowledge...* [Online]. 2021 [visited on 2022-04-24]. Available from: https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2022/ENU/Maya-Animation/files/GUID-9D69DD33-CAEB-4FB9-9559-67984E9ABC2A-htm.html.
64. AUTODESK INC. *ProSound — 3ds Max 2021 — Autodesk Knowledge...* [Online]. 2020 [visited on 2022-04-24]. Available from: https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2021/ENU/3DSMax-Animation/files/GUID-5881B552-5DB6-433F-B40E-2F2E1A6EA453-htm.html.
65. *Audio Rendering – Blender Manual* [online]. 2022 [visited on 2022-04-24]. Available from: https://docs.blender.org/manual/en/latest/render/output/audio/index.html.
66. SIDE EFFECTS SOFTWARE INC. *Spatial Audio channel node [Houdini Documentation]* [online]. [N.d.] [visited on 2022-04-24]. Available from: https://www.sidefx.com/docs/houdini/nodes/chop/spatial.html.
67. SIDE EFFECTS SOFTWARE INC. *Acoustic channel node [Houdini Documentation]* [online]. [N.d.] [visited on 2022-04-24]. Available from: https://www.sidefx.com/docs/houdini/nodes/chop/acoustic.html.
68. SIDE EFFECTS SOFTWARE INC. *Sound object node [Houdini Documentation]* [online]. [N.d.] [visited on 2022-04-24]. Available from: https://www.sidefx.com/docs/houdini/nodes/obj/sound.html.
69. SIDE EFFECTS SOFTWARE INC. *Microphone object node [Houdini Documentation]* [online]. [N.d.] [visited on 2022-04-24]. Available from: https://www.sidefx.com/docs/houdini/nodes/obj/microphone.html.
70. CARPENTIER, Thibaut. Panoramix: 3D mixing and post-production workstation. In: *42nd International Computer Music Conference (ICMC)*. Utrecht, Netherlands, 2016. Available also from: https://hal.archives-ouvertes.fr/hal-01366547.
71. IRCAM. *Panoramix — Ircam Forum* [online]. 2023 [visited on 2023-01-13]. Available from: https://forum.ircam.fr/projects/detail/panoramix/.
72. DOLBY LABORATORIES, INC. *Dolby Atmos ADM Profile specification* [online]. 2022. Version 1.1 [visited on 2023-01-13]. Available from: https://professionalsupport.dolby.com/s/article/Dolby-Atmos-ADM-Profile-specification?language=en_US.

73. HARDT, James. *soundobjects Blender Add-On* [online]. GitHub, [n.d.] [visited on 2023-01-13]. Available from: https://github.com/iluvcapra/soundobjects_blender_addon.

74. MCCORMACK, Leo et al. *Spatial Audio Framework* [online]. GitHub, [n.d.] [visited on 2023-02-10]. Available from: https://github.com/leomccormack/Spatial_Audio_Framework.

75. RICCIO, Christophe et al. *OpenGL Mathematics (GLM)* [online]. GitHub, [n.d.] [visited on 2023-02-10]. Available from: https://github.com/g-truc/glm.

76. RENN-GILES, Fabian. *farbot* [online]. GitHub, [n.d.] [visited on 2023-02-10]. Available from: https://github.com/hogliux/farbot.

77. RAW MATERIAL SOFTWARE LIMITED. *JUCE: AudioProcessor Class Reference* [online]. [N.d.] [visited on 2023-02-13]. Available from: https://docs.juce.com/master/classAudioProcessor.html.

78. RENN-GILES, Fabian; ROWLAND, Dave. *Audio Developer Conference*. Real-time 101 - part I: Investigating the real-time problem space [online]. 2019 [visited on 2023-02-13]. Available from: https://www.youtube.com/watch?v=Q0vrQFyAdWI.

79. DOUMLER, Timur. *Using locks in real-time audio processing, safely* [online]. 2020 [visited on 2023-02-13]. Available from: https://timur.audio/using-locks-in-real-time-audio-processing-safely.

80. RENN-GILES, Fabian; ROWLAND, Dave. *Audio Developer Conference*. Real-time 101 - part II: The real-time audio developer's toolbox [online]. 2019 [visited on 2023-02-13]. Available from: https://www.youtube.com/watch?v=PoZAo2Vikbo.

81. STEINBERG MEDIA TECHNOLOGIES GMBH. *VST 3 Interfaces: Steinberg::Vst::SpeakerArr Namespace Reference* [online]. [N.d.] [visited on 2023-02-11]. Available from: https://steinbergmedia.github.io/vst3_doc/vstinterfaces/namespaceSteinberg_1_1Vst_1_1SpeakerArr.html.

82. *VST3 HOA Support > 3rd order - Developer / VST 3 SDK - Steinberg Forums* [online]. [N.d.] [visited on 2023-02-11]. Available from: https://forums.steinberg.net/t/vst3-hoa-support-3rd-order/201766/30.

83. BLENDER DEVELOPMENT TEAM. *blender/rna_access.c - blender - Blender Projects* [online]. 2023 [visited on 2023-02-10]. Available from: https://projects.blender.org/blender/blender/src/commit/5d30c3994e6ecf665bfae87d7294216d368f0c59/source/blender/makesrna/intern/rna_access.c.

84. PYTHON SOFTWARE FOUNDATION. *Programming FAQ — Python 3.11.2 documentation* [online]. [N.d.] [visited on 2023-02-14]. Available from: https://docs.python.org/3/faq/programming.html#why-did-changing-list-y-also-change-list-x.

Contents of attached media

- `code` ..... source code of the plugins
- `latex` ..... the LaTeX source code of the thesis
  - `vst` ..... source code of the VST plugin
  - `blender` ..... source code of the Blender add-on
- `documentation` ..... Doxygen documentation
  - `Doxyfile` ..... configuration file for generating documentation
  - `ipc.md` ..... IPC protocol message specification
- `demo_videos_and_renders` ..... screen captures and test renders
  - `demo_animation` ..... demo animation
- `Ambilink.vst3` ..... The VST3 plugin built for Linux
- `Ambilink.zip` ..... The Blender add-on
- `README.md` ..... a short description of the software and build instructions
- `thesis.pdf` ..... The thesis in PDF format