`libspatialaudio`: Spatial Audio Library for Ambisonic, Object, Speaker and Binaural Rendering

Peter Stitt

October 2025

Contents

Chapter 1

Getting Started

1.1 Introduction

`libspatialaudio`[1] is an open source library for rendering spatial audio. Originally it is a fork of ambisonic-lib from Aristotel Digenis. It was adopted and extended for spatial audio rendering in VLC Media Player 3.0[2] in 2017 and further extended in 2020 to incorporate a renderer compatible with Audio Definition Model (ADM) streams [1], based on the ADM renderer specification ITU-R BS.2127 [2]. This extension added DirectSpeaker and Object rendering to the library.

The library can be used to encode mono signals to Ambisonics, to decode Ambisonics to loudspeakers or to binaural. It can also perform sound field rotations, which allows it to be used with data from a head-tracker. The main spatialisation processors are outlined in chapter 2, along with the theoretical background on which they are based. `libspatialaudio` also includes a renderer that allows it to use metadata to spatialise an audio stream. The renderer can accept Object, HOA and DirectSpeaker streams and can render to loudspeakers or binaural. It can be used for rendering ADM or IAMF signals. Chapter 5 outlines the `libspatialaudio` Renderer (compatible with ADM ITU-R BS.2127 and IAMF). Code examples are given for those interested primarily in implementation of `libspatialaudio`.

The library allows you to encode, decode, rotate, zoom HOA audio streams up to the 3rd order. It can output to standard and custom loudspeakers arrays. To playback with headphones, the binauralizer applies an HRTF (either a SOFA file or the included MIT HRTF) to provide a spatial binaural rendering effect. The binauralization can also be used to render multichannel streams (5.1, 7.1...).

---

[1] https://github.com/videolabs/libspatialaudio

[2] https://www.videolan.org/vlc/releases/3.0.20.html Last accessed 20/11/2023

1.2 Features

The library has four main uses:

- Ambisonics: encoding, rotation, zooming, psychoacoustic optimisation and decoding of HOA signals is provided. These function up to 3rd order Ambisonics. For more details on the Ambisonics capabilities of `libspatialaudio` please see chapter 2.
- Loudspeaker Binauralization: convert loudspeaker signals to binaural, with the option to use custom HRTFs loaded via a SOFA file. Read more on how to binauralise loudspeaker signals chapter 3.
- Object Spatialisation: convert mono signals to loudspeaker signals. The sound source is panned as a point source using amplitude panning. For a guide on how to spatialise objects please see chapter 4.
- Metadata-based Renderering: render Object, DirectSpeaker, HOA and Binaural streams using their metadata based on Rec. ITU-R BS.2127-1. For a guide on how to use the Renderer see chapter 5.

1.3 Learning About `libspatialaudio`

The documentation includes

- background theory (mathematical, psychoacoustic, DSP).
  - All background theory sections have been written such that it should be accessible to anyone with some mathematical background. However, even without a mathematical background you should still be able to follow what is going on.
- implementation details (layout out the choices made while writing the library)
- code examples (overviews of the main classes and code examples).

To learn more about the Ambisonics processors start in chapter 2. To learn more about the Renderer start in chapter 5. These two chapters provide full examples of both signal flows, as well as links to more details for those who want to learn more or get more specific details.

For those who do not need to know the background theory, most sections contain details of the code, as well as an example of how to implement it separately from the theory.

1.4 Building `libspatialaudio`

To build `libspatialaudio`, follow these steps:

1. 1. Clone the repository (if you haven't already):

```
    git clone https://github.com/videolabs/libspatialaudio.git
    cd libspatialaudio
```

2. Create a 'build' directory and navigate into it:

```
    mkdir build
    cd build
```

3. Run CMake to configure the project:

```
    cmake ..
```

4. Build the project using Make:

```
    make
```

1.5 License

libspatialaudio is released under LGPLv2.1 (or later) and is also available under a commercial license. For full details see the LICENSE file.

Chapter 2

Ambisonic Processing

2.1 Overview of Ambisonic Processors

2.1.1 Ambisonic Processors

The main Ambisonic processing classes are:

- `AmbisonicEncoder`: Converts a mono signal to an Ambisonics signal of order 1 to 3. Read more in section 2.2.
- `AmbisonicRotator`: Applies a user-specified rotation to the sound field. This is useful for head tracking or keeping the audio aligned with a 360 video. Read more in section 2.3.
- `AmbisonicZoomer`: An acoustical "zoom" that focuses on the front of the sound scene. Read more in section 2.4.
- `AmbisonicDecoder`: A decoder that converts an Ambisonics signal to a set of loudspeaker signals. It contains optimised presets for 5.1 and 7.1 loudspeaker layouts. Custom layouts can also have their decoding matrix defined by the user, for additional flexibility. Read more in section 2.6.2.
- `AmbisonicBinauralizer`: A decoder that converts an Ambisonics signal to a binaural signal. This allows for immersive audio to be experienced over headphones. Custom HRTFs can be loaded using the .SOFA format, allowing for a customised listening experience. Read more in section 2.7.
- `AmbisonicAllRAD`: A decoder that converts an Ambisonics signal to a set of loudspeaker signals corresponding to the ITU layouts or IAMF specificiation layouts. It uses the AllRAD method to calculate the decoding matrix. Read more in section 2.6.4.

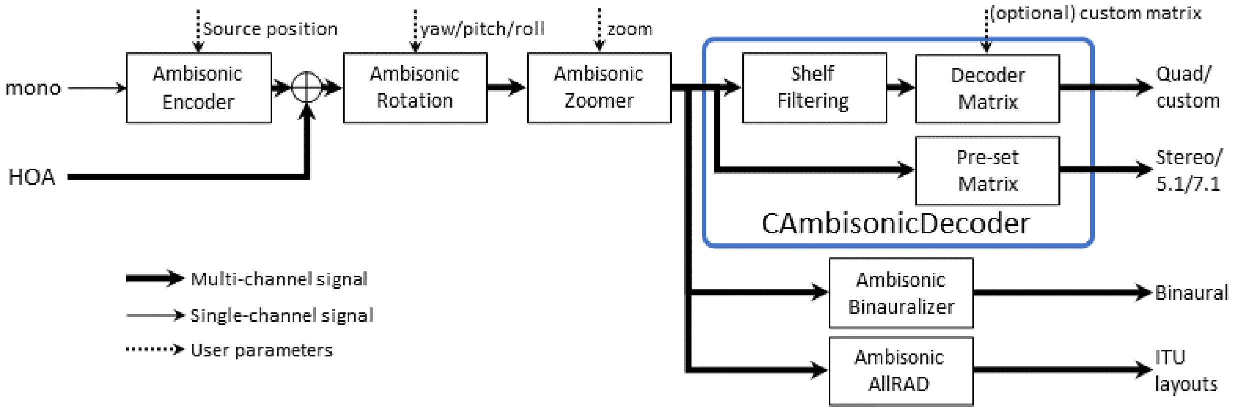


Figure 2.1: The signal flow to encode a mono signal to HOA, adding to any pre-encoded HOA streams, with rotation and zooming processing before decoding to different formats.

2.1.2 Ambisonics Signal Flow

In general, unless your signal is already in Ambisonics format, the signal chain will begin with `AmbisonicEncoder` to bring the audio in to the Ambisonics domain. The rotation and zooming processing elements are optional and will be context dependent. However, they should generally be applied to a summation of all the sound sources making up the scene, rather than applying to each one individually.

Finally, the signal needs to be converted from a Ambisonics to the listener's particular layout. The choice of which decoder should be used is generally decided by the target output layout. ITU layouts could use `AmbisonicAllRAD` or, if a custom decoder is available, `AmbisonicDecoder`. Binauralisation over headphones uses `AmbisonicBinauralization`.

The above figure shows the signal flow used to encode a mono input signal to HOA, perform rotation of the sound field (using `AmbisonicRotator`) and then to acoustically zoom on the front of the sound field (using `AmbisonicZoomer`). Finally, it is decoded to the listener using one of the decoding methods.

2.1.3 Code Example

This example shows how to apply the entire signal chain described above and to convert the output to binaural.

See [here](AmbisonicDecoding.md) for specific examples on how to decode using ‘AmbisonicDecoder‘ or ‘AmbisonicAllRAD‘.

```cpp
#include "Ambisonics.h"

using namespace spaudio;

```

```
const unsigned int sampleRate = 48000;
const int nBlockLength = 512;

// Higher ambisonic order means higher spatial resolution and more channels required
const unsigned int nOrder = 1;
// Set the fade time to the length of one block
const float fadeTimeInMilliSec = 1000.f * (float)nBlockLength / (float)sampleRate;

std::vector<float> sinewave(nBlockLength);
// Fill the vector with a sine wave
for (int i = 0; i < nBlockLength; ++i)
    sinewave[i] = (float)std::sin((float)M_PI * 2.f * 440.f * (float)i / (float)sampleRate);

// B-format buffer
BFormat myBFormat;
myBFormat.Configure(nOrder, true, nBlockLength);
myBFormat.Reset();

// Encode the signal to Ambisonics
AmbisonicEncoder myEncoder;
myEncoder.Configure(nOrder, true, sampleRate, 0);
PolarPosition<float> position;
position.azimuth = 0;
position.elevation = 0;
position.distance = 1.f;
myEncoder.SetPosition(position);
myEncoder.Reset();

// Set up the rotator
AmbisonicRotator myRotator;
myRotator.Configure(nOrder, true, nBlockLength, sampleRate, fadeTimeInMilliSec);
RotationOrientation rotOri;
rotOri.yaw = 0.5 * M_PI;
myRotator.SetOrientation(rotOri);

// Set up the zoomer processor
AmbisonicZoomer myZoomer;
myZoomer.Configure(nOrder, true, nBlockLength, 0);
myZoomer.SetZoom(0.5f);

// Set up the binaural decoder
AmbisonicBinauralizer myDecoder;
unsigned int tailLength = 0;
myDecoder.Configure(nOrder, true, sampleRate, nBlockLength, tailLength);
// Configure buffers to hold the decoded signal
const unsigned int nEar = 2;
float** earOut = new float* [nEar];
for (int iEar = 0; iEar < nEar; ++iEar)
    earOut[iEar] = new float[nBlockLength];

```

```
// Process the audio ==============================
// Encode the signal to Ambisonics
myEncoder.Process(sinewave.data(), nBlockLength, &myBFormat);

// Rotate the Ambisonics signal
myRotator.Process(&myBFormat, nBlockLength);

// Apply zooming to the Ambisonics signal
myZoomer.Process(&myBFormat, nBlockLength);

// Decode the signal to binaural
myDecoder.Process(&myBFormat, earOut, nBlockLength);

// Cleanup
for (unsigned iEar = 0; iEar < nEar; ++iEar)
    delete[] earOut[iEar];
delete[] earOut;
```

2.2 Ambisonic Encoding

2.2.1 Theory and Implementation Details

Ambisonic encoding is the conversion of a mono signal to Ambisonics of a specified order. Ambisonic order governs the spatial resolution of the sound scene. Higher orders lead to higher resolution at the expense of increased processing requirements. The number of encoded channels is $(N+1)^2$ meaning 1st, 2nd and 3rd order require 4, 9 and 16 channels respectively.

A mono input signal is converted to Ambisonics through multiplication with a series of encoding gains. `libspatialaudio` uses the AmbiX specification [3], which orders the channels in the ACN format and uses SN3D normalisation. The encoded signal is thus calculated as

$$\mathbf{b}_N(t) = \mathbf{y}_N(\theta, \phi)s(t) \tag{2.1}$$

where $s(t)$ is the mono input signal at time t and $\mathbf{y}_N(\theta, \phi)$ is a column vector of length $(N+1)^2$ containing the SN3D-weighted spherical harmonic gains $[Y_0^0(\theta, \phi), Y_1^{-1}(\theta, \phi), Y_1^0(\theta, \phi), \ldots, Y_N^N(\theta, \phi)]^{\mathrm{T}}$. The angles θ and ϕ are the direction of the sound source and are expected in degrees. The azimuth θ is positive in the anti-clockwise direction, so +90° is to the left of the listener. The elevation ϕ ranges from -90° below the listener to +90° above.

Encoding Gain Interpolation

In order to allow to real-time changes in source direction with minimal audio artefacts ("zipper" sounds) as the direction changes, `AmbisonicEncoder` uses an instance of `GainInterp` internally. Every time the direction of the source is changed the gain vector $\mathbf{y}_N(\theta, \phi)$ is updated and a linear interpolation is applied going from the current to new values. This length of the interpolation can be specified by the user. Its ideal length will depend on the signals but 10 ms will reduce most "zipper" sounds.

2.2.2 `AmbisonicEncoder`

The `AmbisonicEncoder` class is used to convert a mono signal to Ambisonics. When the encoding position is modified it internally smooths the encoding gains $\mathbf{y}_N(\theta,\phi)$ to avoid unwanted clicks in the output.

Configuration

Before calling any other functions the object must first be configured by calling `Configure()` with the appropriate values. If the values are supported then the it will return ‘true‘ and the object can now be used.

The configuration parameters are:

- nOrder: The ambisonic order from 1 to 3.
- b3D: A bool to indicate if the signal is to be encoded to 2D (azimuth only) or 3D (azimuth and elevation). 3D should be preferred.
- sampleRate: The sample rate of the audio being used e.g. 44100 Hz, 48000 Hz etc. This must be an integer value greater than zero.
- fadeTimeMilliSec: The time in milliseconds to fade from an old set of encoding gains to another. Lower values will lead to lower latency at the expense of possible audio artefacts. Higher values will lead to increased latency before the source reaches the new encoded position. A value of 10 ms is usually a good starting point.

Set Encoding Direction

The encoding direction is set as a polar direction in radians using the `SetPosition()` function. It takes a `PolarPosition<float>` as an input.

Note: the distance is ignored. Only the encoding direction is set.

Encoding a Signal

An array of floats can be encoded using either the `Process()` or `ProcessAccumul()` functions. These two functions process the input signal in the same way. The only difference is that `ProcessAccumul()` will add the newly encoded signal to the output with an optional gain, whereas `Process()` will replace the destination signal with the encoded signal.

The inputs are:

- pfSrc: A pointer to the mono input signal.
- nSamples: The length of the input signal in samples.
- pBFDst: A pointer to the destination B-format signal.
- nOffset: Optional offset position when writing to the output. When set to zero this will write the signal to the start of `pBFDst`. Any non-zero value will write to the output with a delay of the specified number of samples, leaving any preceding samples unchanged. The offset and input signal length must not be such that the encoded signal would be written beyond the end of `pBFDst` i.e. `nSamples + nOffset <= pfDst->GetSampleCount()`.
- (`ProcessAccumul()` only) fGain: Optional gain to apply to the output before it is added to the signal in ‘pBFDst‘.

2.2.3 Code Example

This example shows how to convert a mono sine wave to an Ambisonics signal that rotates around the listener from the front and then to the left, back, right and back to the front.

```
#include "Ambisonics.h"

using namespace spaudio;

const unsigned int sampleRate = 48000;
const int nBlockLength = 512;
const int nBlocks = 94;
const int nSigSamples = nBlocks * nBlockLength; // Approximately 1 second @ 48 kHz

// Higher ambisonic order means higher spatial resolution and more channels required
const unsigned int nOrder = 1;
// Set the fade time to the length of one block
const float fadeTimeInMilliSec = 1000.f * (float)nBlockLength / (float)sampleRate;

std::vector<float> sinewave(nSigSamples);
// Fill the vector with a sine wave
for (int i = 0; i < nSigSamples; ++i)
    sinewave[i] = (float)std::sin((float)M_PI * 2.f * 440.f * (float)i / (float)sampleRate);

// Destination B-format buffer
BFormat myBFormat;
myBFormat.Configure(nOrder, true, nSigSamples);
myBFormat.Reset();

// Set up and configure the Ambisonics encoder
AmbisonicEncoder myEncoder;
```

```
myEncoder.Configure(nOrder, true, sampleRate, fadeTimeInMilliSec);

// Set test signal's initial direction in the sound field
PolarPosition<float> position;
position.azimuth = 0;
position.elevation = 0;
position.distance = 1.f;
myEncoder.SetPosition(position);
myEncoder.Reset();

for (int iBlock = 0; iBlock < nBlocks; ++iBlock)
{
    // Update the encoding position to reach by the end of the block
    position.azimuth = (float)(iBlock + 1) / (float)nBlocks * 2.f * (float)M_PI;
    myEncoder.SetPosition(position);

    // Encode the first block, writing to the appropriate point of the destination buffer
    const unsigned int iSamp = iBlock * nBlockLength;
    myEncoder.Process(&sinewave[iSamp], nBlockLength, &myBFormat, iSamp);
}
```

2.3 Ambisonic Rotation

2.3.1 Theory and Implementation Details

Any signal in the ambisonic domain can be rotated by application of a rotation matrix. This allows (for example) headtracking data to be used to rotate the sound field in the opposite direction to how the listener is moving their head, which will keep the sound sources in the correct position in space, rather than them moving with the listeners head.

Rotation is performed by multiplying the Ambisonic signal $\mathbf{b}_N(t)$ by a rotation matrix:

$$\mathbf{b}'_N(t) = \mathbf{R}_N(\alpha, \beta, \gamma)\mathbf{b}_N(t). \tag{2.2}$$

The rotation angles are yaw, pitch and roll around the z-axis, y-axis and x-axis respectively.

A positive yaw angle will cause a sound source to rotated in a clockwise direction with respect to the origin when viewed from along the z-axis (above). From the point-of-view of the listener this is as if they turn to their left.

A positive pitch angle will cause a sound source to move above the listener. From the point-of-view of the listener this is as if they tilt their head forward.

A positive roll angle will rotate a sound from the left to the right underneath the listener. From the point-of-view of the listener this is as if they roll their head to the right.

Rotations can be applied in any order but changing the order of the rotation will not result in the same output, so care must be taken. The yaw-pitch-roll order, where rotations around the z-axis are performed first, is common in Ambisonics, but `AmbisonicRotator` gives the option for any combination to be used.

Rotation of the sound field is particularly beneficial when working with a binaural renderer because it fixes the sound sources in space. This can help to reduce front-to-back confusions or with sound source externalisation.

Note that `AmbisonicRotator` uses hard-coded equations to generate the individual yaw, pitch and roll rotation matrices. However, for higher orders a recurrence relation based method can be used to compute $\mathbf{R}_N(\alpha, \beta, \gamma)$ [4].

2.3.2 `AmbisonicRotator`

The `AmbisonicRotator` class is used to change the orientation of a supplied Ambisonic signal in real-time. In order to avoid clicks while rotating the rotation matrices are interpolated over a specified length of time set during configuration of the object.

Configuration

Before calling any other functions the object must first be configured by calling `Configure()` with the appropriate values. If the values are supported then it will return `true` and the object can now be used.

The configuration parameters are:

- **nOrder**: The ambisonic order from 1 to 3. - **b3D**: `AmbisonicRotator` only supports rotation of 3D sound scenes. This should be set to `true` or else configuration will fail. - **nBlockSize**: The maximum block size `Process()` is expected to handle. - **sampleRate**: The sample rate of the audio being used e.g. 44100 Hz, 48000 Hz etc. This must be an integer value greater than zero. - **fadeTimeMilliSec**: The time in milliseconds to fade from an old matrix to another. Lower values will lead to lower latency at the expense of possible audio artefacts. Higher values will lead to increased latency before the source reaches the new orientation. A value of 10 ms is usually a good starting point.

Set Rotation Order and Orientation

The default rotation ordering is yaw-pitch-roll. However, any desired order can be specified using `SetRotationOrder()`. Changing the rotation ordering will cause the rotation matrix to be updated using the new ordering.

The yaw, pitch, and roll values are set by passing a `RotationOrientation` to `SetOrientation()`. The rotation angles should be supplied in radians.

Rotating an Ambisonic Signal

A B-format signal can be rotated using the `Process()` function. The input signal is replaced by the rotated signal. The inputs are:

- **pBFSrcDst**: A pointer to the source B-format signal that is replaced with the processed signal. - **nSamples**: The length of the input signal in samples.

2.3.3 Code Example

This example shows how to rotate an Ambisonics signal by 90° ($\pi/2$ radians) so that it is heard as coming from the right of the listener when decoded.

```cpp
#include "Ambisonics.h"

using namespace spaudio;

const unsigned int sampleRate = 48000;
const int nBlockLength = 512;

// Higher ambisonic order means higher spatial resolution and more channels required
const unsigned int nOrder = 1;
// Set the fade time to the length of one block
const float fadeTimeInMilliSec = 1000.f * (float)nBlockLength / (float)sampleRate;

std::vector<float> sinewave(nBlockLength);
// Fill the vector with a sine wave
for (int i = 0; i < nBlockLength; ++i)
    sinewave[i] = (float)std::sin((float)M_PI * 2.f * 440.f * (float)i / (float)sampleRate);

// B-format buffer
BFormat myBFormat;
myBFormat.Configure(nOrder, true, nBlockLength);
myBFormat.Reset();

// Encode the signal to Ambisonics
AmbisonicEncoder myEncoder;
myEncoder.Configure(nOrder, true, sampleRate, 0);
PolarPosition<float> position;
position.azimuth = 0;
position.elevation = 0;
position.distance = 1.f;
myEncoder.SetPosition(position);
myEncoder.Reset();
myEncoder.Process(sinewave.data(), nBlockLength, &myBFormat);

// Set up the rotator
AmbisonicRotator myRotator;
myRotator.Configure(nOrder, true, nBlockLength, sampleRate, fadeTimeInMilliSec);
```

```
RotationOrientation rotOri;
rotOri.yaw = 0.5 * M_PI; // pi/2 radians = 90 degrees
myRotator.SetOrientation(rotOri);

// Rotate the Ambisonics signal
myRotator.Process(&myBFormat, nBlockLength);
```

2.4 Ambisonic Zoomer

2.4.1 Theory and Implementation Details

The `AmbisonicZoomer` class allows for a kind of acoustic zoom to the front of the sound field. It was implemented for use with the zoom function with 360-videos in VLC Media Player to place emphasis on sounds in the view direction.

It samples the sound field to the front with a virtual cardioid microphone with order matching the order of the input signal. This mono virtual microphone signal is then re-encoded back to a signal at the front of the sound field and blended with the full sound field based on the zoom value set in `AmbisonicZoomer::SetZoom`.

A zoom value of 0 will perform no transformation to the ambisonic signal while a zoom of 1 will lead to an extreme zoom, which results in mono signal encoded to the front of the sound field.

Another method for achieving a similar effect would be to decode the Ambisonics signal to a set of points regularly spaced on the sphere, applying a directional weight to these points, then re-encoding the points to Ambisonics. However, this requires a transformation from and to the Ambisonics domain over a large number of points. This makes it more CPU intensive than the simple method used here, and so it is not implemented in `libspatialaudio`.

2.4.2 `AmbisonicZoomer`

Configuration

Before calling any other functions the object must first be configured by calling `Configure()` with the appropriate values. If the values are supported then it will return `true` and the object can now be used.

The configuration parameters are:

- **nOrder**: The ambisonic order from 1 to 3.
- **b3D**: A bool to indicate if the signal to be decoded is 2D (azimuth only) or 3D (azimuth and elevation).
- **nBlockSize**: The maximum block size `Process()` is expected to handle.
- **sampleRate**: The sample rate of the signal expected by ‘Process()‘.

Setting the Zoom Amount

The amount of zoom can be set from 0 to 1 using `SetZoom()`. A value of 0 corresponds to no zoom and the signal will be returned unchanged after processing. A value of 1 means that the maximum amount of zooming will be applied and almost all of the signal will be from the re-encoded forward-facing virtual microphone.

Zooming a Signal

A B-format signal can have zooming applied using the `Process()` function. The input signal is replaced by the processed signal.

The inputs are:

- **pBFSrcDst**: A pointer to the source B-format signal.
- **nSamples**: The number of samples to process.

Code Example

This example shows how to apply zooming to an Ambisonics signal.

```
#include "Ambisonics.h"

using namespace spaudio;

const unsigned int sampleRate = 48000;
const int nBlockLength = 512;

// Higher ambisonic order means higher spatial resolution and more channels required
const unsigned int nOrder = 1;

std::vector<float> sinewave(nBlockLength);
// Fill the vector with a sine wave
for (int i = 0; i < nBlockLength; ++i)
    sinewave[i] = (float)std::sin((float)M_PI * 2.f * 440.f * (float)i / (float)sampleRate);

// B-format buffer
BFormat myBFormat;
myBFormat.Configure(nOrder, true, nBlockLength);
myBFormat.Reset();

// Encode the signal to Ambisonics
AmbisonicEncoder myEncoder;
myEncoder.Configure(nOrder, true, sampleRate, 0);
PolarPosition<float> position;
position.azimuth = 0;
position.elevation = 0;
position.distance = 1.f;
myEncoder.SetPosition(position);
myEncoder.Reset();
myEncoder.Process(sinewave.data(), nBlockLength, &myBFormat);

// Set up the zoomer processor
AmbisonicZoomer myZoomer;
myZoomer.Configure(nOrder, true, nBlockLength, 0);
myZoomer.SetZoom(0.5f);

// Apply zooming to the Ambisonics signal
myZoomer.Process(&myBFormat, nBlockLength);
```

2.5 Ambisonic Psychoacoustic Optimisation

When decoding ambisonic signals to loudspeakers (or binaural using the virtual loudspeaker approach) there is a limit frequency f_{lim} above which the sound field is no longer considered well-reproduced. This limit frequency increases with the ambisonic order as can be approximated using [5]

$$f_{\text{lim}} = \frac{cN}{4R(N+1)\sin(\pi/(2N+2))} \tag{2.3}$$

where c is the speed of sound in m/s, and R is the radius of the reproduction area in metres. For a central listener, and assuming $R = 0.09$ m (approximate radius of a human head), the limit frequencies for first- to third-order are approximately 674 Hz, 1270 Hz, and 1867 Hz.

Ambisonic theory optimises for the velocity vector at low frequencies and the energy vector at higher ones. This optimision is performed simply by applying a gain by-order to the input signal. At low frequencies the gain is unity and this is known as a basic decoder. At higher frequencies the gains are chosen to psychoacoustically optimised the signal. This is done such that the energy in the decoded loudspeaker array is concentrated in the source direction. This is known as max $\mathbf{r}_\text{E}$ decoding.

The weights applied to the channels of each set of channels of degree n are

$$a_n^{2D} = \cos\left(\frac{n\pi}{2N+2}\right) \tag{2.4}$$

for 2D decoding [6]. For 3D decoding they can be approximated using [7]

$$a_n^{3D} = P_n\left(\cos\left(\frac{137.9^\circ}{N+1.51}\right)\right) \tag{2.5}$$

where P_n is a Legendre polynomial. The decoding equation eq. (2.7) becomes

$$\mathbf{x}(t) = \mathbf{D}_N^{\text{SN3D}}\text{diag}(\mathbf{a}_N)\mathbf{b}_N(t) \tag{2.6}$$

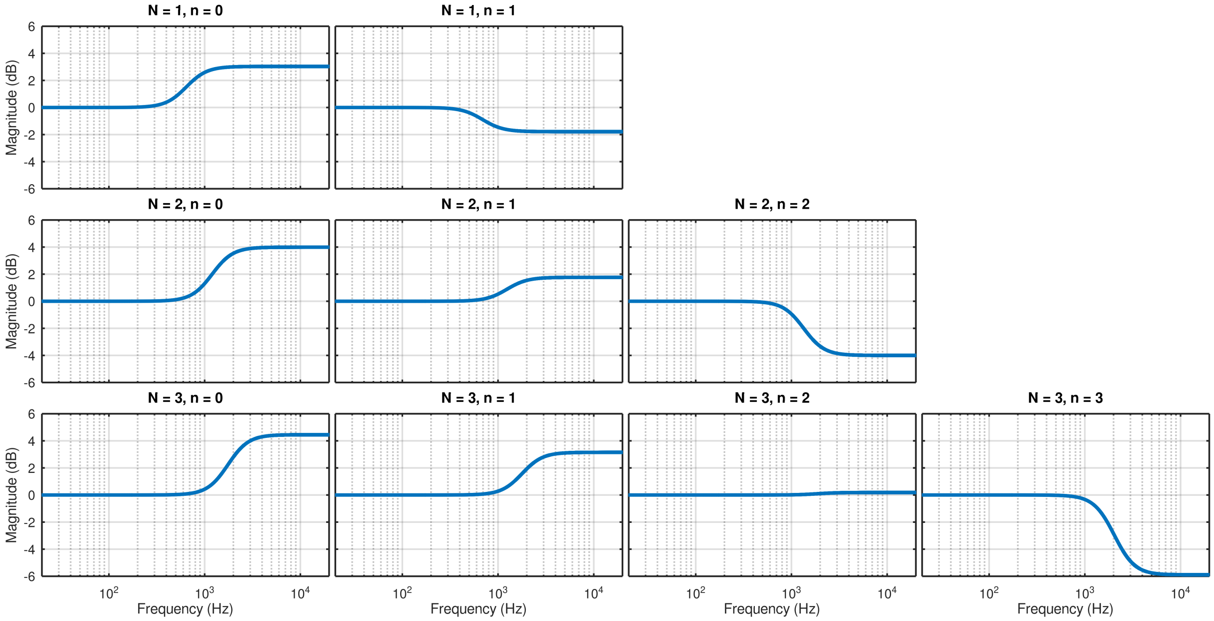


Figure 2.2: The magnitudes of the psychoacoustic optimisation filters each group of channel for first- to third-order. The $n = 0$ filter is applied to the first ambisonic channel, $n = 1$ to the three first-order order channels, etc.

where $\mathbf{a}_N$ is a vector of length $(N + 1)^2$ containing the max r_E gains with an energy compensation gain applied. The gains applied to frequencies below the limit frequency are all unity so $\mathrm{diag}(\mathbf{a}_N)$ is the identity matrix, meaning eq. (2.6) collapses back to eq. (2.7).

The frequency-dependent application of these gains leads to a set of shelf filters that are applied to channels of the same order. The filtered signal can then be decoded using a frequency-independent decoder.

Shelf Filter Implementation

Psychoacoustic optimisation is implemented in `AmbisonicOptimFilters`. In order to apply the optimisation gains in a frequency-dependent manner $\mathbf{b}_N$ is filtered by shelf filters with a transition frequency set to the limit frequency, unity gain below and a_n gain above.

The filters must be phase-matched to ensure correct decoding. `AmbisonicOptimFilters` uses 4th-order Linkwitz-Riley filters for this purpose. These are implemented as IIR filters, meaning that the optimisation filtering will have low latency suitable for real-time applications. Linkwitz-Riley filters have the advantageous property that when a low-passed and high-passed signal are summed the magnitude response is flat. In practice, `AmbisonicOptimFilters` applies a low- and high-pass filter to the input, multiplies the low-passed signal by a_n and sums it with the high-passed signal.

The magnitude responses of each of the optimisation filters is shown in the fig. 2.2 for orders 1 to 3.

Note: `AmbisonicDecoder` and `AmbisonicAllRAD` already use `AmbisonicOptimFilters` internally. Therefore, psychoacoustic optimisation should not be applied before using these decoders.

2.5.1 `AmbisonicOptimFilters`

Configuration

Before calling any other functions the object must first be configured by calling `Configure()` with the appropriate values. If the values are supported then it will return `true` and the object can now be used.

The configuration parameters are:

- **nOrder**: The ambisonic order from 1 to 3.
- **b3D**: A bool to indicate if the signal is to be filtered is 2D (azimuth only) or 3D (azimuth and elevation).
- **nBlockSize**: The maximum number of samples the object is expected to process at a time.
- **sampleRate**: The sample rate of the audio being used e.g. 44100 Hz, 48000 Hz etc. This must be an integer value greater than zero.

Set Custom Optimisation Gains

`AmbisonicOptimFilters` defaults to using max r_E optimisation. However, the user can supply their own optimisation gains to be applied to the high frequency shelf using `SetHighFrequencyGains()`. The number of gains should be equal to $N + 1$.

Apply Optimisation Filters

A B-format signal can be optimised using the `Process()` function. The input signal is replaced by the optimised signal.

The inputs are:

- **pBFSrcDst**: A pointer to the source B-format signal that is replaced with the processed signal.
- **nSamples**: The length of the input signal in samples.

2.5.2 Code Example

```cpp
#include "Ambisonics.h"

using namespace spaudio;

const unsigned int sampleRate = 48000;
const int nBlockLength = 512;

// Higher ambisonic order means higher spatial resolution and more channels required
const unsigned int nOrder = 1;
// Set the fade time to the length of one block
const float fadeTimeInMilliSec = 1000.f * (float)nBlockLength / (float)sampleRate;

std::vector<float> sinewave(nBlockLength);
// Fill the vector with a sine wave
for (int i = 0; i < nBlockLength; ++i)
    sinewave[i] = (float)std::sin((float)M_PI * 2.f * 440.f * (float)i / (float)sampleRate);

// B-format buffer
BFormat myBFormat;
myBFormat.Configure(nOrder, true, nBlockLength);
myBFormat.Reset();

// Encode the signal to Ambisonics
AmbisonicEncoder myEncoder;
myEncoder.Configure(nOrder, true, sampleRate, 0);
PolarPosition<float> position;
position.azimuth = 0;
position.elevation = 0;
position.distance = 1.f;
myEncoder.SetPosition(position);
myEncoder.Reset();
myEncoder.Process(sinewave.data(), nBlockLength, &myBFormat);

// Set up the optimisation filters
AmbisonicOptimFilters myOptim;
myOptim.Configure(nOrder, true, nBlockLength, sampleRate);

// Filter the Ambisonics signal
myOptim.Process(&myBFormat, nBlockLength);
```

2.6 Ambisonic Decoding

2.6.1 Theory and Implementation Details

An ambisonic signal cannot be played back directly. A decoder is required to convert the signal from the spherical harmonic representation to either loudspeaker signals (implemented in `AmbisonicDecoder` and `AmbisonicAllRAD`) or headphone signals (implemented in `AmbisonicBinauralizer`).

An ambisonic signal of order N can be decoded to a set of loudspeakers using a decoding matrix $\mathbf{D}_N^{\mathrm{SN3D}}$ which has size $M \times (N+1)^2$ where M is the total number of loudspeakers. The output loudspeaker signal $\mathbf{x}(t)$ is given by

$$\mathbf{x}(t) = \mathbf{D}_N^{\mathrm{SN3D}} \mathbf{b}_N(t). \tag{2.7}$$

The decoding matrix can be calculated using a number of different methods [8, 7, 9]. If the loudspeaker layout is regularly distributed on a sphere then decoding can be performed simply by sampling the ambisonic signal at each of the directions of the loudspeakers, known as a sampling ambisonic decoder (SAD) [10]. The sampling decoding matrix $\mathbf{D}_N$ is

$$\mathbf{D}_N^{\mathrm{SN3D}} = \frac{1}{\sqrt{M}} \left[\tilde{\mathbf{y}}_N(\theta_1, \phi_1), \tilde{\mathbf{y}}_N(\theta_2, \phi_2), \ldots, \tilde{\mathbf{y}}_N(\theta_M, \phi_M)\right]^{\mathrm{T}} \mathrm{diag}(\mathbf{q}_N) = \frac{1}{\sqrt{M}} \tilde{\mathbf{Y}}_N^{\mathrm{T}} \mathrm{diag}(\mathbf{q}_N) \tag{2.8}$$

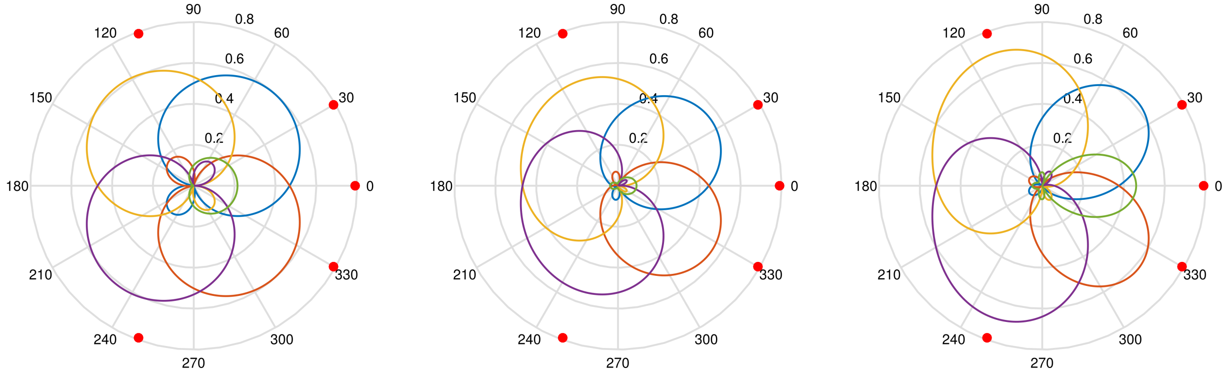
where $\tilde{\mathbf{y}}_{\mathbb{N}}(\theta_m, \phi_m)$ is the vector of spherical harmonics of the m-th loudspeaker direction **with N3D normalisation** and $\mathbf{q}_{\mathbb{N}}$ is a vector of gains to convert $\mathbf{b}_N(t)$ from SN3D to N3D normalisation.

Decoding to Irregular Loudspeaker Layouts Using Non-Linear Optimisation

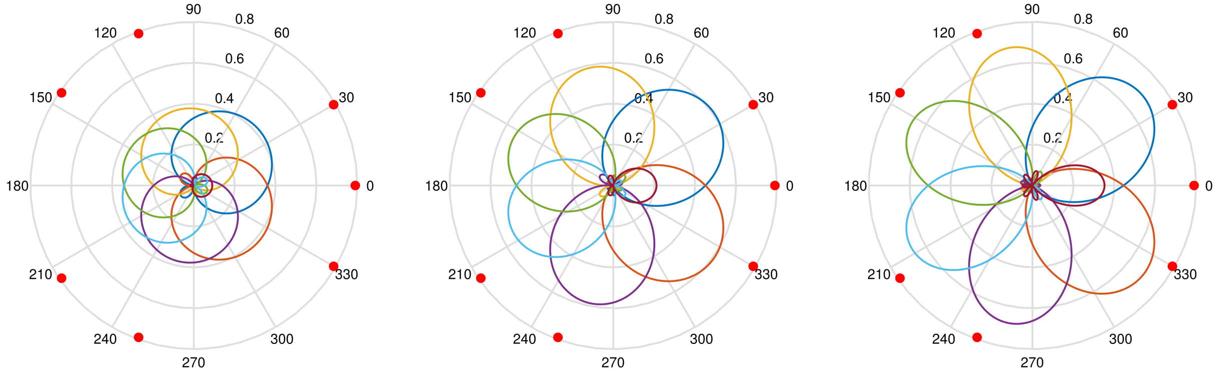
If the layout is irregular, such as the ITU 5.1 and 7.1 layouts, then finding a suitable decoding method becomes more complex. A significant limitation is that most decoder calculation methods require at least as many loudspeakers as there are ambisonic channels. This limits 5.1 to first-order. Wiggins [11] used an optimisation technique that uses the Gerzon velocity and energy vectors [12], along with the total pressure and energy sum of the loudspeaker gains, to calculate an optimised decoder for the 5.1 layout. Due to the optimisation method the higher order spherical harmonics are able to contribute, meaning decoders can be obtained for first- to third-order.

The 5.1 decoders in `libspatialaudio` were kindly provided by Bruce Wiggins. The decoders for the 7.1 layout were generated using a method inspired by the non-linear optimisation work of Wiggins. A decoder derived in the manner should not have psychoacoustic optimisation gains or filters (see section 2.5) applied since the desired optimisations were factored into the decoder optimisation.

Figure 2.3 shows the gains of each of the loudspeakers for the 5.1 and 7.1 arrays for the orders 1 to 3 with the decoders in `libspatialaudio`. The use of higher-order harmonics is particularly evident in the asymmetric shape of the two rear-surround speakers of the 5.1 layout for third-order.

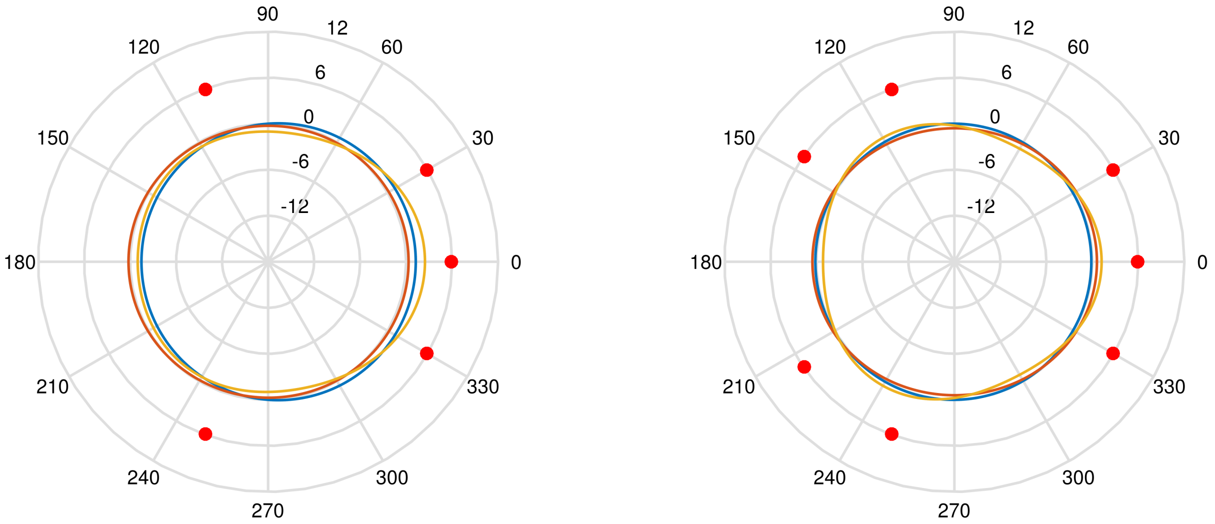


(a) First-order 5.1 gains (b) Second-order 5.1 gains (c) Third-order 5.1 gains



(d) First-order 7.1 gains (e) Second-order 7.1 gains (f) Third-order 7.1 gains

Figure 2.3: Loudspeaker gains for 5.1 and 7.1 arrays for first- to third-order Ambisonics. The directions of the loudspeakers are shown as red circles.



(a) 5.1 array

(b) 7.1 array

Figure 2.4: The sum of the loudspeaker gains for 5.1 and 7.1 arrays for source positions on the horizontal. The first-order levels are shown in blue, second-order in red, and third-order in yellow. The loudspeaker directions are shown as red circles.

Figure 2.4 shows the mean level of the decoders for sources around the horizontal. It shows that there is in a small level variation with source position around the horizontal at most 3.5 dB for the third-order 5.1 decoder. Consistent level across source directions is important in maintaining the balance of the sound scene when several sources have been panned to different directions.

Decoding to Irregular Loudspeaker Layouts Using AllRAD

The AllRAD method [7] first decodes the signal to a spherically regular t-design virtual loudspeaker layout and then uses VBAP [13] to pan the virtual loudspeakers to the real layout. The aim is to allow for optimal decoding to the virtual loudspeaker layout and then to spatialise this with a robust panning algorithm.

Additional imaginary loudspeakers can also be included in the main real loudspeaker layout to help when decoding to, for example, a dome.

The AllRAD method is the preferred decoding method of the EBU Audio Definition Model renderer (Recommendation ITU-R BS.2127-1) and Alliance For Open Media's Immersive Audio Model and Formats (IAMF) specification (https://aomediacodec.github.io/iamf/).

2.6.2 `AmbisonicDecoder`

`AmbisonicDecoder` decodes an Ambisonics signal to either predefined or custom layouts using one of several methods:

- the SAD method: sampling at each of the specified loudspeaker directions. This is suboptimal for any layout that is not regular on the sphere.
- optimised decoder: the 5.1 and 7.1 preset layouts use the non-linearly optimised decoder matrices described above, since the SAD method is particularly unsuited to these layouts. The preset decoder matrices are loaded automatically if either of those layouts are selected or if a custom array that matches the directions is defined.
- user decoder definition: the user can define the decoder matrix coefficients directly for the specified layout. This means the class can be used with any decoding method that the user requires. See here for more details on how to set a custom decoder matrix.

When a decoder matrix other than one of the presets is used `AmbisonicDecoder` applies shelf filtering to psychoacoustically optimise the decoded signal. Read more about psychoacoustic optimisation here.

Configuration

Before calling any other functions the object must first be configured by calling `Configure()` with the appropriate values. If the values are supported then it will return `true` and the object can now be used.

The configuration parameters are:

- **nOrder**: The ambisonic order from 1 to 3.
- **b3D**: A bool to indicate if the signal to be decoded is 2D (azimuth only) or 3D (azimuth and elevation).
- **nBlockSize**: The maximum number of samples the decoder is expected to process at a time.
- **sampleRate**: The sample rate of the audio being used e.g. 44100 Hz, 48000 Hz etc. This must be an integer value greater than zero.
- **nSpeakerSetup**: Choice of loudspeaker layout from one of the predefined options or to indicate use of a custom layout. Note that the 5.1 and 7.1 layouts **do not** use the ITU ordering. They use L/R/Ls/Rs/C/LFE and L/R/Ls/Rs/Lr/Rr/C/LFE respectively. This matches the channel ordering in VLC media player.
- **nSpeakers**: The total number of loudspeakers if a custom layout is used. This is ignored if one of the predefined layouts is selected.

Set Custom Decoder Coefficients

The `SetCoefficient()` function can be used to define a custom decoder for the currently selected layout.

The parameters are:

- **nSpeaker**: The index of the loudspeaker for which the coefficient is to be set.
- **nChannel**: The index of the corresponding Ambisonic channel for which the coefficient is to be set.
- **fCoeff**: The decoder matrix coefficient.

The following code will set the decoder matrix to hold the values stored in `myDecMat`.

```
for (int iSpeaker = 0; iSpeaker < nLdspk; ++iSpeaker)
    for (int iCoeff = 0; iCoeff < nAmbiComponents; ++iCoeff)
    {
        myDecoder.SetCoefficients(iSpeaker, iCoeff, myDecMat[iSpeaker][
   iCoeff]);
    }
```

Note: Calling `Refresh()` will overwrite any coefficients set in this way and replace them with the default values for the selected layout (either a SAD matrix or a preset).

Decoding a Signal

A B-format signal can be decoded to the loudspeaker signals using the `Process()` function. The processing is non-replacing, so the original B-format signal is unchanged and the decoded signal is contained in the output array.

The inputs are:

- **pBFSrc**: A pointer to the source B-format signal.
- **nSamples**: The length of the input signal in samples.
- **ppDst**: Array of pointers of size nLdspk x nSamples containing the decoded signal.

2.6.3 `AmbisonicDecoder` Code Example

This example shows how to decode an Ambisonics signal to a 5.1 loudspeaker layout. It loads the optimised preset decoder for this layout automatically.

```
#include "Ambisonics.h"

using namespace spaudio;
```

```
const unsigned int sampleRate = 48000;
const int nBlockLength = 512;

// Higher ambisonic order means higher spatial resolution and more channels required
const unsigned int nOrder = 1;

std::vector<float> sinewave(nBlockLength);
// Fill the vector with a sine wave
for (int i = 0; i < nBlockLength; ++i)
    sinewave[i] = (float)std::sin((float)M_PI * 2.f * 440.f * (float)i / (float)sampleRate);

// B-format buffer
BFormat myBFormat;
myBFormat.Configure(nOrder, true, nBlockLength);
myBFormat.Reset();

// Encode the signal to Ambisonics
AmbisonicEncoder myEncoder;
myEncoder.Configure(nOrder, true, sampleRate, 0);
PolarPosition<float> position;
position.azimuth = 0;
position.elevation = 0;
position.distance = 1.f;
myEncoder.SetPosition(position);
myEncoder.Reset();
myEncoder.Process(sinewave.data(), nBlockLength, &myBFormat);

// Set up the decoder for a 5.1 layout
AmbisonicDecoder myDecoder;
myDecoder.Configure(nOrder, true, nBlockLength, sampleRate, Amblib_SpeakerSetUps::kAmblib_51);

// Configure buffers to hold the decoded signal
const unsigned int nLdspk = myDecoder.GetSpeakerCount();
float** ldspkOut = new float* [nLdspk];
for (int iLdspk = 0; iLdspk < nLdspk; ++iLdspk)
    ldspkOut[iLdspk] = new float[nBlockLength];

// Decode the Ambisonics signal
myDecoder.Process(&myBFormat, nBlockLength, ldspkOut);

// Cleanup
for (unsigned iLdspk = 0; iLdspk < nLdspk; ++iLdspk)
    delete[] ldspkOut[iLdspk];
delete[] ldspkOut;
```

2.6.4 `AmbisonicAllRAD`

`AmbisonicAllRAD` decodes an Ambisonics signal to a layout defined in the ADM renderer specification (Recommendation ITU-R BS.2127-1) and AOM's IAMF specification (https://aomediacodec.github.io/iamf/). It does not currently support decoding to arbitrary loudspeaker layouts.

`AmbisonicAllRAD` optionally applies shelf filtering to psychoacoustically optimise the decoded signal. Read more about psychoacoustic optimisation here.

Configuration

Before calling any other functions the object must first be configured by calling `Configure()` with the appropriate values. If the values are supported then it will return `true` and the object can now be used.

The configuration parameters are:

- **nOrder**: The ambisonic order from 1 to 3.
- **nBlockSize**: The maximum number of samples the decoder is expected to process at a time.
- **sampleRate**: The sample rate of the audio being used e.g. 44100 Hz, 48000 Hz etc. This must be an integer value greater than zero.
- **layoutName**: The name of the loudspeaker layout. This must be one of those specified in the ADM or IAMF specifications. The valid names are:
  - `"0+2+0"`: BS.2051-3 System A (Stereo)
  - `"0+4+0"`: Quad
  - `"0+5+0"`: BS.2051-3 System B (5.1)
  - `"2+5+0"`: BS.2051-3 System C (5.1.2)
  - `"4+5+0"`: BS.2051-3 System D (5.1.4)
  - `"4+5+1"`: BS.2051-3 System E
  - `"3+7+0"`: BS.2051-3 System F
  - `"4+9+0"`: BS.2051-3 System G
  - `"9+10+3"`: BS.2051-3 System H
  - `"0+7+0"`: BS.2051-3 System I (7.1)
  - `"4+7+0"`: BS.2051-3 System J (7.1.4)
  - `"2+7+0"`: 7.1.2 (IAMF v1.0.0-errata)
  - `"2+3+0"`: 3.1.2 (IAMF v1.0.0-errata)

  - `"9+10+5"`: EBU Tech 3369 (BEAR) 9+10+5 - 9+10+3 with LFE1 & LFE2 removed and B+135 & B-135 added

- **useLFE**: If `true` then the decoded signal will include the LFE channels. If `false` then the LFE channels will be excluded i.e. a 0+5+0 (5.1) signal will be output as a 5-channel signal instead of a 6-channel one.
- **useOptimFilts**: (Optional) `false` by default. If true, then psychoacoustic optimisation filters will be applied before decoding the signal.

Decoding a Signal

A B-format signal can be decoded to the loudspeaker signals using the `Process()` function. The processing is non-replacing, so the original B-format signal is unchanged, and the decoded signal is contained in the output array.

The inputs are:

- **pBFSrc**: A pointer to the source B-format signal.
- **nSamples**: The length of the input signal in samples.
- **ppDst**: Array of pointers of size nLdspk x nSamples containing the decoded signal.

2.6.5 `AmbisonicAllRAD` Code Example

This example shows how to decode an Ambisonics signal to an ITU 5.1 loudspeaker layout using the AllRAD method.

```cpp
#include "Ambisonics.h"

using namespace spaudio;

const unsigned int sampleRate = 48000;
const int nBlockLength = 512;

// Higher ambisonic order means higher spatial resolution and more channels required
const unsigned int nOrder = 1;

std::vector<float> sinewave(nBlockLength);
// Fill the vector with a sine wave
for (int i = 0; i < nBlockLength; ++i)
    sinewave[i] = (float)std::sin((float)M_PI * 2.f * 440.f * (float)i / (float)sampleRate);

// B-format buffer
BFormat myBFormat;
myBFormat.Configure(nOrder, true, nBlockLength);
myBFormat.Reset();

```

```
// Encode the signal to Ambisonics
AmbisonicEncoder myEncoder;
myEncoder.Configure(nOrder, true, sampleRate, 0);
PolarPosition<float> position;
position.azimuth = 0;
position.elevation = 0;
position.distance = 1.f;
myEncoder.SetPosition(position);
myEncoder.Reset();
myEncoder.Process(sinewave.data(), nBlockLength, &myBFormat);

// Set up the decoder for a 5.1 layout
AmbisonicAllRAD myDecoder;
myDecoder.Configure(nOrder, nBlockLength, sampleRate, "0+5+0");

// Configure buffers to hold the decoded signal
const unsigned int nLdspk = myDecoder.GetSpeakerCount();
float** ldspkOut = new float* [nLdspk];
for (int iLdspk = 0; iLdspk < nLdspk; ++iLdspk)
    ldspkOut[iLdspk] = new float[nBlockLength];

// Decode the Ambisonics signal
myDecoder.Process(&myBFormat, nBlockLength, ldspkOut);

// Cleanup
for (unsigned iLdspk = 0; iLdspk < nLdspk; ++iLdspk)
    delete[] ldspkOut[iLdspk];
delete[] ldspkOut;
```

2.7 Ambisonic-to-Binaural Decoding

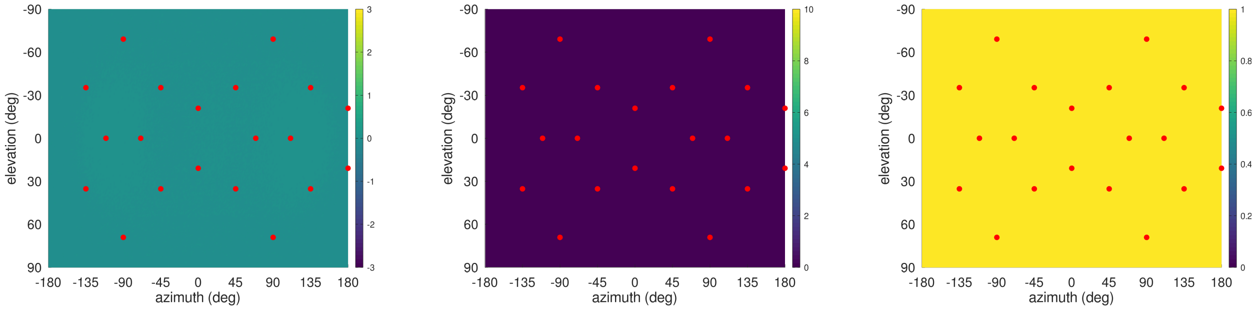
2.7.1 Theory and Implementation Details

There are several methods for decoding from Ambisonics to binaural [14, 15, 16]. In general, decoding is performed by convolution of the ambisonic signal with a matrix of filters. `AmbisonicBinauralizer` uses a virtual-speaker approach [14].

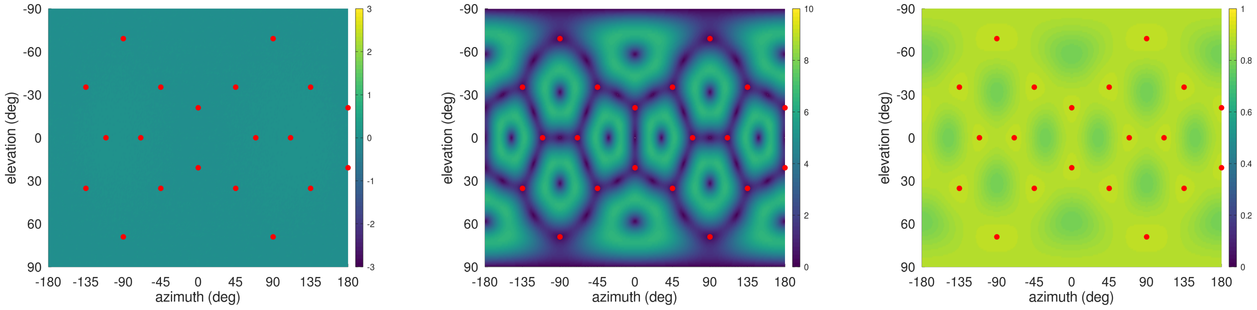
The virtual loudspeaker approach essentially decodes the signal to known loudspeaker array and then applies an HRTF to each of the loudspeakers, which are summed for 2-channel output. This requires $2 \times M$ convolutions, where M is the number of virtual loudspeakers.

In order to reduce the number of convolutions the HRTF of each virtual speaker is scaled by the corresponding ambisonic gains and summed to create a filter for each spherical harmonic. For an HRIR of length k this is calculated using

$$\mathbf{Z}^{\mathrm{bin}}_{N,ear} = \frac{1}{\sqrt{M}} \left[\tilde{\mathbf{y}}_N(\theta_1,\phi_1), \tilde{\mathbf{y}}_N(\theta_2,\phi_2), \ldots, \tilde{\mathbf{y}}_N(\theta_M,\phi_M)\right] \mathbf{H}_{\mathrm{ear}} \tag{2.9}$$



(a) Total amplitude (b) Velocity vector error (c) Velocity vector mag.



(d) Total energy (e) Energy vector error (f) Energy vector mag.

Figure 2.5: The quality measures for a basic decoder (top row) and max r_{E} decoder (bottom row) as a function of the source direction across the whole sphere for a third-order decoding to a dodecahedral loudspeaker array. The directions of the loudspeakers are shown as red circles.

where $\mathbf{Z}^{\mathrm{bin}}_{N,ear}$ is a matrix of size $(N+1)^2 \times k$ containing the spherical harmonic decomposition of the loudspeaker array HRIRs, $\mathbf{H}_{\mathrm{ear}}$ is an $M \times k$ matrix containing the HRIRs corresponding to each of the loudspeaker directions for the left or right ear. This leads to $2 \times (N+1)^2$ convolutions which can be calculated using

$$x^{ear}_{\mathrm{bin}}(t) = \sum_{i=0}^{(N+1)^2} q_i z^{ear}_i(t) \circledast b_i(t) \tag{2.10}$$

where $z^{ear}_i(t)$ is the filter in the i-th row of $\mathbf{Z}^{\mathrm{bin}}_{\mathrm{N,ear}}$, $b_i(t)$ is the i-th term in the ambisonic signal $\mathbf{b}_N(t)$, and q_i is the i-th element of $\mathbf{q}_N$ which contains gains that convert the input signal from SN3D to N3D.

The number of convolutions can be reduced further to only $(N+1)^2$ convolutions if the head is assumed to be symmetric [17]. This is done by calculating the convolutions in eq. (2.10) for one ear and storing them. They are summed for the first ear directly. For the opposite ear the spherical harmonics that are left-right symmtertic have their polarity inverted and before being summed.

Virtual Loudspeaker Layout Choices

For first-order signals `AmbisonicBinauralizer` uses a virtual layout on the vertices of a cube (8 virtual loudspeakers). For second- and third-order a virtual layout on the vertices of a dodecahedron (20 virtual loudspeakers) is used. Therefore, in both cases, preprocessing the HRTFs reduces the number of convolutions required (from 8 to 4 for first-order and 20 to 9 or 16 for second- and third-order respectively). This meets a requirement of `libspatialaudio` to minimise its CPU use.

A cube was chosen for first-order because it is a mathematically ideal layout for this order. The dodocahedron was chosen because it provides good results for second- and third-order order and 8 of its 20 loudspeaker directions coincide directly with the cube layout. This allows for the number of HRTFs stored for both layouts to be only 20, meeting a requirement that `libspatialaudio` be lightweight.

Figure 2.5 shows the total loudspeaker gain and energy for the dodecahedron for third-order (first- and second-order are now shown because they have equal performance for all source directions). It also shows the difference between the panning direction and velocity and energy vectors [12], representing a prediction of perceived direction for low and high frequencies respectively. Finally, it also shows the velocity and energy vector magnitudes, for which a value of 1 is ideal (although an energy vector magnitude of 1 is not possible for a sound source reproduced by more than one loudspeaker).

The figure shows that both amplitude and energy are constant for all source directions. The velocity vector has perfect localisation and magnitude, indicating that at low frequencies the decoder is also performing ideally. The max r_{E} decoder for high frequencies shows small deviations between the source direction and the energy vector prediction. However, the maximum error is only 6.5° . The energy vector magnitude also exhibits some variation with source position. This may result in small variations in the perceived source width, but this effect will be small.

Psychoacoustic Optimisation

As outlined in section 2.5 shelf filtering can be applied to Ambisonics signal so that they are decoded in a psychoacoustically optimised manner.

2.7.2 `AmbisonicBinauralizer`

Configuration

Before calling any other functions the object must first be configured by calling `Configure()` with the appropriate values. If the values are supported then it will return `true` and the object can now be used.

The configuration parameters are:

- **nOrder**: The ambisonic order from 1 to 3.
- **b3D**: A bool to indicate if the signal is to be decoded is 2D (azimuth only) or 3D (azimuth and elevation).
- **nBlockSize**: The maximum number of samples the decoder is expected to process at a time.
- **sampleRate**: The sample rate of the audio being used e.g. 44100 Hz, 48000 Hz etc. This must be an integer value greater than zero.
- **tailLength**: The value is replaced with the length of the HRIRs used for the binauralisation.
- **HRTFPath**: An optional path to the .SOFA file containing the HRTF. If no path is supplied and the `HAVE_MIT_HRTF` compiler flag is used, then the MIT HRTF will be used.
- **lowCpuMode**: (Optional) If this is set to true (its default value), then the symmetric head assumption is used to reduce CPU usage [17].

Decoding a Signal

A B-format signal can be decoded to the binaural signals using the `Process()` function. The processing is non-replacing, so the original B-format signal is unchanged, and the decoded signal is contained in the output array.

The inputs are:

- **pBFSrc**: A pointer to the source B-format signal.
- **ppDst**: Array of pointers of size 2 x nSamples containing the binaurally decoded signal.
- **nSamples**: (Optional) The length of the input signal in samples. If this is not supplied, then `nBlockSize` samples are assumed.

2.7.3 Code Example

This example shows how to decode an Ambisonics signal to binaural.

```
#include "Ambisonics.h"

using namespace spaudio;

const unsigned int sampleRate = 48000;
const int nBlockLength = 512;

// Higher ambisonic order means higher spatial resolution and more
    channels required
```

```cpp
const unsigned int nOrder = 1;

std::vector<float> sinewave(nBlockLength);
// Fill the vector with a sine wave
for (int i = 0; i < nBlockLength; ++i)
    sinewave[i] = (float)std::sin((float)M_PI * 2.f * 440.f * (float)i / (
   float)sampleRate);

// B-format buffer
BFormat myBFormat;
myBFormat.Configure(nOrder, true, nBlockLength);
myBFormat.Reset();

// Encode the signal to Ambisonics
AmbisonicEncoder myEncoder;
myEncoder.Configure(nOrder, true, sampleRate, 0);
PolarPosition<float> position;
position.azimuth = (float)M_PI * 0.5f;
position.elevation = 0;
position.distance = 1.f;
myEncoder.SetPosition(position);
myEncoder.Reset();
myEncoder.Process(sinewave.data(), nBlockLength, &myBFormat);

// Set up the binaural decoder
AmbisonicBinauralizer myDecoder;
unsigned int tailLength = 0;
myDecoder.Configure(nOrder, true, sampleRate, nBlockLength, tailLength);

// Configure buffers to hold the decoded signal
const unsigned int nEar = 2;
float** earOut = new float* [nEar];
for (int iEar = 0; iEar < nEar; ++iEar)
    earOut[iEar] = new float[nBlockLength];

// Decode the Ambisonics signal
myDecoder.Process(&myBFormat, earOut, nBlockLength);

// Cleanup
for (unsigned iEar = 0; iEar < nEar; ++iEar)
    delete[] earOut[iEar];
delete[] earOut;
```

Chapter 3

Speaker Binauralization

3.1 Theory and Implementation Details

Binauralization of a loudspeaker array uses a simple method. First, a pair of head-related impulse responses (HRIRs) for each of the loudspeaker directions is convolved with the signal of that loudspeaker. The convolved signals are summed to give a final 2-channel binaural signal for listening over headphones.

$$z_{ear}(t) = \sum_{m=0}^{M-1} h_{m,ear}(t) \circledast x_m(t) \tag{3.1}$$

where $z_{ear}(t)$ is the signal at the left or right ear, h_{ear} is the corresponding HRIR for the left/right ear, $x_m(t)$ is the signal of the m-th loudspeaker and M is the total number of loudspeakers.

`SpeakersBinauralizer` provides a simple convolution of a set of supplied loudspeaker signals with a pair of HRTFs (one for each ear). The convolution is implemented in the frequency domain using an overlap-add algorithm.

If `libmysofa` is activated at compile time then the HRTF can be loaded from a .SOFA file, allowing for the use of custom HRTFs. Custom HRTFs allow for users to potentially load their own HRTF, giving highly personalised rendering.

3.2 `SpeakersBinauralizer`

3.2.1 Configuration

Before calling any other functions the object must first be configured by calling `Configure()` with the appropriate values. If the values are supported then it will return `true` and the object can now be used.

The configuration parameters are:

- **sampleRate**: The sample rate of the audio being used e.g. 44100 Hz, 48000 Hz etc. This must be an integer value greater than zero.
- **nBlockSize**: The maximum number of samples the decoder is expected to process at a time.
- **speakers**: A pointer to an array of `AmbisonicSpeaker` holding the speaker directions.
- **nSpeakers**: The number of speakers to be binauralized.
- **tailLength**: The value is replaced with the length of the HRIRs used for the binauralisation.
- **HRTFPath**: An optional path to the .SOFA file containing the HRTF. If no path is supplied and the `HAVE_MIT_HRTF` compiler flag is used, then the MIT HRTF will be used.

3.2.2 Binauralizing a Signal

A set of loudspeaker signals can be decoded to the binaural signals using the `Process()` function. The processing is non-replacing, so the original loudspeaker signals are unchanged, and the decoded signal is contained in the output array.

The inputs are:

- **pBFSrc**: A pointer to the source B-format signal.
- **ppDst**: Array of pointers of size 2 x nBlockSize containing the binaurally decoded signal.

Code Example

This example shows how to convert a set of loudspeaker signals to a 2-channel binaural signal.

```
#include "SpeakersBinauralizer.h"

using namespace spaudio;

const unsigned int sampleRate = 48000;
const int nBlockLength = 512;

// Configure the speaker layout
const unsigned int nSpeakers = 5;
AmbisonicSpeaker speakers[nSpeakers];
speakers[0].SetPosition({ 30.f / 180.f * (float)M_PI, 0.f, 1.f }); // L
speakers[1].SetPosition({ -30.f / 180.f * (float)M_PI, 0.f, 1.f }); // R
speakers[2].SetPosition({ 0.f / 180.f * (float)M_PI, 0.f, 1.f }); // C
speakers[3].SetPosition({ 110.f / 180.f * (float)M_PI, 0.f, 1.f }); // Ls
speakers[4].SetPosition({ -110.f / 180.f * (float)M_PI, 0.f, 1.f }); // Rs
```

```

// Configure buffers to hold the 5.0 signal
float** ldspkInput = new float* [5];
for (int iLdspk = 0; iLdspk < 5; ++iLdspk)
{
    ldspkInput[iLdspk] = new float[nBlockLength];
    // Fill L and C channels, panning the signal to half way between both speakers
    for (int i = 0; i < nBlockLength; ++i)
        ldspkInput[iLdspk][i] = iLdspk == 0 || iLdspk == 2 ? (float)std::sin((float)M_PI * 2.f * 440.f * (float)i / (float)sampleRate) : 0.f;
}

// Set up the binauralizer
SpeakersBinauralizer myBinaural;
unsigned int tailLength = 0;
myBinaural.Configure(sampleRate, nBlockLength, speakers, nSpeakers, tailLength);

// Configure buffers to hold the decoded signal
const unsigned int nEar = 2;
float** earOut = new float* [nEar];
for (int iEar = 0; iEar < nEar; ++iEar)
    earOut[iEar] = new float[nBlockLength];

// Convert the loudspeaker signals to binaural
myBinaural.Process(&ldspkInput[0], earOut);

// Cleanup
for (unsigned iEar = 0; iEar < nEar; ++iEar)
    delete[] earOut[iEar];
delete[] earOut;
for (unsigned iLdspk = 0; iLdspk < nSpeakers; ++iLdspk)
    delete[] ldspkInput[iLdspk];
delete[] ldspkInput;
```

Chapter 4

Object Spatialisation

4.1 Theory and Implementation Details

Object-based panning is the process of placing a mono signal at an arbitrary position in a loudspeaker layout by distributing its energy across the available loudspeakers. `ObjectPanner` implements Vector Base Amplitude Panning (VBAP) following the algorithm described in the Audio Definition Model (ADM) renderer specification [13, 2].

`ObjectPanner` can be used to pan a mono sound source as a point source on a given loudspeaker array. More advanced Object rendering is possible with the `Renderer` class that allows for metadata to specify additional parameters, such as object extent.

4.1.1 VBAP Algorithm

Classic VBAP works by selecting a subset of loudspeakers (a base) that encloses the desired source direction and calculating gain factors such that the vector sum of the loudspeaker positions matches the target direction. The gains are normalized to preserve energy.

For a source at direction vector d and a loudspeaker base matrix L containing unit vectors in the loudspeaker directions, the gains g are obtained by solving:

$$\mathrm{Lg} = \mathrm{d} \tag{4.1}$$

with the constraint that all elements of g are non-negative. The resulting gains g' are then normalized:

$$\mathrm{g}' = \frac{\mathrm{g}}{\|\mathrm{g}\|} \tag{4.2}$$

4.1.2 Gain Interpolation

To avoid audible artefacts when the source position changes (clicks or “zipper noise”), `ObjectPanner` uses an internal `GainInterp` object. When the position is updated, the loudspeaker gain vector is interpolated over a user-specified fade time (typically 10 ms). This ensures smooth transitions.

4.2 `ObjectPanner`

The `ObjectPanner` class is used to distribute a mono signal across a loudspeaker layout using VBAP. It supports smooth position updates to reduce real-time “zipper” effects for moving sources.

4.2.1 Configuration

Before calling any other functions the object must first be configured by calling `Configure()` with the appropriate values. If the values are supported then it will return `true` and the object can now be used.

The configuration parameters are:

- **layout**: An `OutputLayout` to select a supported output layout.
- **sampleRate**: The sample rate of the audio being used (e.g. 44100 Hz, 48000 Hz). Must be an integer > 0.
- **fadeTimeMilliSec**: The time in milliseconds to fade from an old set of panning gains to another. Lower values reduce latency but may introduce artefacts; higher values increase smoothness at the cost of responsiveness.

4.2.2 Set Panning Position

The panning position is set as a polar direction in radians using the `SetPosition()` function. It takes a `PolarPosition<double>` as input.

4.2.3 Panning a Signal

An array of floats can be panned using either the `Process()` or `ProcessAccumul()` functions. These two functions process the input signal in the same way. The only difference is that `ProcessAccumul()` will add the newly panned signal to the output with an optional gain, whereas `Process()` will replace the destination signal with the panned signal.

The inputs are:

- **pfSrc**: A pointer to the mono input signal.

- **nSamples**: The length of the input signal in samples.
- **ppDst**: A pointer to the destination buffer containing one channel per loudspeaker.
- `nSamplesOut`: The number of samples in the output buffer. Must be at least `nSamplesIn + nOffset`.
- `nOffset`: Optional offset position when writing to the output.
- (`ProcessAccumul()` only) `fGain`: Optional gain to apply before accumulation.

4.2.4 Code Example

This example shows how to pan a mono sine wave into a 5.1 loudspeaker layout using `ObjectPanner`.

```
#include "ObjectPanner.h"

using namespace spaudio;

const unsigned int sampleRate = 48000;
const int nBlockLength = 512;

auto layout = OutputLayout::FivePointOne;

ObjectPanner objPanner;
objPanner.Configure(layout, sampleRate, 1000.f * (float)nBlockLength / (float)sampleRate);
unsigned nLdspk = (unsigned)objPanner.GetNumSpeakers();

// Generate a block of sine wave samples
std::vector<float> sinewave(nBlockLength);
for (int i = 0; i < nBlockLength; ++i)
    sinewave[i] = (float)std::sin((float)M_PI * 2.f * 440.f * (float)i / (float)sampleRate);

// Prepare the output stream (5.1 layout)
float** renderStream = new float* [nLdspk];
for (unsigned int i = 0; i < nLdspk; ++i)
{
    renderStream[i] = new float[nBlockLength];
    std::fill(renderStream[i], renderStream[i] + nBlockLength, 0.f);
}

for (float az = 0.f; az < 360.f; az += 1.f)
{
    auto position = PolarPosition<double>{ az, 0.f, 1.f };
    objPanner.SetPosition(position);
    objPanner.Process(sinewave.data(), nBlockLength, renderStream, nBlockLength);
}

for (unsigned iLdspk = 0; iLdspk < nLdspk; ++iLdspk)
    delete[] renderStream[iLdspk];
delete[] renderStream;
```

Chapter 5

Renderer

5.1 Overview

`libspatialaudio` includes a renderer for the Audio Definition Model (ADM) renderer detailed in Rec. ITU-R BS.2127-1 [2]. It also aims to support elements of the AOM's Immersive Audio Model and Formats (IAMF) specification[1].

The Renderer specifies rendering methods for Objects, HOA, and DirectSpeaker streams:

- Object streams consist of a mono audio signal and metadata which specify how they are spatialised. More details can be read in section 5.2.
- HOA streams are summed and decoded to the target playback format (loudspeakers or binaural). More details and an example can be found in section 5.3.
- DirectSpeaker streams are intended to be sent directly to a specific loudspeaker in the target layout, if it is present. If it is not present then the DirectSpeaker stream will be spatialised based on the target layout and its associated metadata. More details can be read in section 5.4.

Rec. ITU-R BS.2127-1 [2] should be consulted for full details about the rendering algorithms.

Supported Output Formats

`Renderer` supports all of the loudspeaker layouts specified in [2] as well as additional ones specified in the IAMF.

The supported ITU layouts are:

- `"0+2+0"`: BS.2051-3 System A (Stereo)

---

[1] https://aomediacodec.github.io/iamf/

- `"0+5+0"`: BS.2051-3 System B (5.1)
- `"2+5+0"`: BS.2051-3 System C (5.1.2)
- `"4+5+0"`: BS.2051-3 System D (5.1.4)
- `"4+5+1"`: BS.2051-3 System E
- `"3+7+0"`: BS.2051-3 System F
- `"4+9+0"`: BS.2051-3 System G
- `"9+10+3"`: BS.2051-3 System H
- `"0+7+0"`: BS.2051-3 System I (7.1)
- `"4+7+0"`: BS.2051-3 System J (7.1.4)

The additional IAMF layouts are:

- `"2+7+0"`: 7.1.2 (IAMF v1.1.0)
- `"2+3+0"`: 3.1.2 (IAMF v1.1.0)

A quad layout is also supported, along with the virtual layout specified in [18]:

- `"0+4+0"`: Quad
- `"9+10+5"`: EBU Tech 3369 (BEAR) 9+10+5 - 9+10+3 with LFE1 and LFE2 removed and B+135 and B-135 added. This is used internally for the binaural rendering.

In addition to these loudspeaker layouts, `Renderer` implements a custom binaural renderer that is inspired by the BEAR renderer specified in [18]. Full details on the binaural rendering processing can be read in section 5.5.

5.1.1 `Renderer`

The `Renderer` is the only class you will need to interact with to render an a mixture of Object, DirectSpeaker and/or HOA feeds, such as ADM or IAMF. It accepts audio streams and metadata to produce a signal rendered to the specified output format.

Use the class as follows:

1. Configure the class by informing it of the different streams it should expect.
2. Add each stream using the appropriate function i.e. `AddHOA()` for an Ambisonics signal.
3. Once all streams have been added use `GetRenderedAudio()`. This clears any internal buffers containing waiting audio.

4. Repeat steps 2 and 3.

Configuration

Before calling any other functions the object must first be configured by calling `Configure()` with the appropriate values. If the values are supported then the it will return ‘true‘ and the object can now be used.

The configuration parameters are:

- **outputTarget**: The target output layout selected from `OutputLayout`.
- **hoaOrder**: The ambisonic order of the signal to be rendered. This also control the order of the HOA processing used for binaural output, so higher should be preferred.
- **nSampleRate**: Sample rate of the streams to be processed.
- **nSamples**: The maximum number of samples in an audio frame to be added to the stream.
- **channelInfo**: Information about the expected stream formats.
- **HRTFPath**: Path to an HRTF SOFA-file to be used when the output layout is binaural.
- **reproductionScreen**: (Optional) Reproduction screen details used for screen scaling/locking.
- **layoutPositions**: (Optional) Real polar positions for each of the loudspeaker in the layout. This is used if they do not exactly match the ITU specification. Note that they must be within the range allowed by the specification.

AddObject

`AddObject()` should be called for every Object stream to be rendered. Its corresponding `ObjectMetadata` should be supplied with it.

This function can be called multiple times for each Object each time `GetRenderedAudio()` is called by using the offset parameter. However, care must be taken to ensure that each new call would not lead to the total number of samples added that frame going beyond the value nSamples set in `Configure()`.

The inputs are:

- **pIn** Pointer to the mono object buffer to be rendered.
- **nSamples** Number of samples in the stream.
- **metadata** Metadata for the object stream.
- **nOffset** (Optional) Number of samples of delay to applied to the signal.

AddHOA

`AddHOA()` should be called for every HOA stream to be rendered. Its corresponding `HoaMetadata` should be supplied with it.

This function can be called multiple times for each Object each time `GetRenderedAudio()` is called by using the offset parameter. However, care must be taken to ensure that each new call would not lead to the total number of samples added that frame going beyond the value nSamples set in `Configure()`.

The inputs are:

- **pHoaIn** The HOA audio buffers to be rendered of size nAmbiCh x nSamples
- **nSamples** Number of samples in the stream.
- **metadata** Metadata for the HOA stream.
- **nOffset** (Optional) Number of samples of delay to applied to the signal.

AddDirectSpeaker

`AddDirectSpeaker()` should be called for every DirectSpeaker stream to be rendered. Its corresponding `DirectSpeakerMetadata` should be supplied with it.

This function can be called multiple times for each Object each time `GetRenderedAudio()` is called by using the offset parameter. However, care must be taken to ensure that each new call would not lead to the total number of samples added that frame going beyond the value nSamples set in `Configure()`.

The inputs are:

- **pIn** Pointer to the mono DirectSpeaker buffer to be rendered.
- **nSamples** Number of samples in the stream.
- **metadata** Metadata for the DirectSpeaker stream.
- **nOffset** (Optional) Number of samples of delay to applied to the signal.

AddBinaural

`AddBinaural()` should be called for every Binaural stream to be rendered. If the output format is not set to binaural then any audio added here is discarded.

This function can be called multiple times for each Object each time `GetRenderedAudio()` is called by using the offset parameter. However, care must be taken to ensure that each new call would not lead to the total number of samples added that frame going beyond the value nSamples set in `Configure()`.

The inputs are:

- **pHoaIn** The binaural audio buffers to be rendered of size 2 x nSamples

- **nSamples** Number of samples in the stream.
- **nOffset** (Optional) Number of samples of delay to applied to the signal.

GetRenderedAudio

Get the rendered audio using `GetRenderedAudio()`. After it has been called all internal buffers are reset. Therefore, it is very important that this function only be called after all streams have been added.

- **pRender** Rendered audio output.
- **nSamples** The number of samples to get.

5.1.2 Code Example

This example renders an Object, HOA and DirectSpeaker stream to binaural. For each of the stream types the metadata is generated along with an audio stream. The `StreamInformation` used to configure `Renderer` is generated when the metadata is "read". `StreamInformation` keeps track of the total number of tracks as well as what type they have. Note that when the metadata is being generated the track indices must match the ordering with which the tracks are added to the `StreamInformation`.

In this example an Object rotates around the listener (left, back, right and again to front), the HOA stream is encoded with a source at 90° and the DirectSpeaker corresponds to the right-wide loudspeaker placed at -60° . Replace the streams with your own mono audio to hear the motion clearer.

```
const unsigned int sampleRate = 48000;
const int nBlockLength = 512;
const int nBlocks = 470;
const int nSigSamples = nBlocks * nBlockLength; // Approximately 5 seconds
    @ 48 kHz

// Ambisonic order
const unsigned int nOrder = 3;

// Prepare the stream to hold the rendered audio (binaural)
const unsigned int nLdspk = 2;
float** renderStream = new float* [nLdspk];
for (unsigned int i = 0; i < nLdspk; ++i)
{
    renderStream[i] = new float[nSigSamples];
}

// Track index for each channel in the streams
unsigned int trackInd = 0;
// The stream information used to configure Renderer
StreamInformation streamInfo;

```

```cpp
// Add an object stream ==================================
// Prepare the metadata for the stream
ObjectMetadata objectMetadata;
objectMetadata.trackInd = trackInd++;
objectMetadata.blockLength = nBlockLength;
objectMetadata.cartesian = false;

// Channel lock (off by default)
objectMetadata.channelLock = ChannelLock();
objectMetadata.channelLock->maxDistance = 0.f; // <- Increase this to see the signal snap to fully in the left loudspeaker

// Object divergence (off by default)
objectMetadata.objectDivergence = ObjectDivergence();
objectMetadata.objectDivergence->azimuthRange = 30.; // <- Controls the width of the divergence
objectMetadata.objectDivergence->value = 0.; // <- Increase this to apply object divergence

// Object extent (off by default)
objectMetadata.width = 0.; // <- Increase this to increase the width and spread the Object over more adjacent loudspeakers

// Configure the stream information. In this case there is only a single channel stream
streamInfo.nChannels++;
streamInfo.typeDefinition.push_back(TypeDefinition::Objects);

std::vector<float> objectStream(nSigSamples);
// Fill the vector with a sine wave
for (int i = 0; i < nSigSamples; ++i)
    objectStream[i] = (float)std::sin((float)M_PI * 2.f * 440.f * (float)i / (float)sampleRate);

// Add an HOA stream =================================
const unsigned int nHoaCh = OrderToComponents(nOrder, true);
// Set the fade time to the length of one block
const float fadeTimeInMilliSec = 0.f;

std::vector<float> hoaSinewave(nSigSamples);
// Fill the vector with a sine wave
for (int i = 0; i < nSigSamples; ++i)
    hoaSinewave[i] = (float)std::sin((float)M_PI * 2.f * 700.f * (float)i / (float)sampleRate);

// Destination B-format buffer
BFormat myBFormat;
myBFormat.Configure(nOrder, true, nSigSamples);
myBFormat.Reset();

// Set up and configure the Ambisonics encoder
AmbisonicEncoder myEncoder;
myEncoder.Configure(nOrder, true, sampleRate, fadeTimeInMilliSec);

```

```
// Set test signal's initial direction in the sound field
PolarPosition<float> position;
position.azimuth = 0.5f * (float)M_PI;
position.elevation = 0;
position.distance = 1.f;
myEncoder.SetPosition(position);
myEncoder.Reset();
myEncoder.Process(hoaSinewave.data(), nSigSamples, &myBFormat);

// Prepare the stream to be processed by the renderer
float** hoaStream = new float* [nHoaCh];
for (unsigned int i = 0; i < nHoaCh; ++i)
{
    hoaStream[i] = new float[nSigSamples];
    myBFormat.ExtractStream(&hoaStream[i][0], i, nSigSamples);
}

// Prepare the metadata for the stream
HoaMetadata hoaMetadata;
hoaMetadata.normalization = "SN3D";

// Configure the stream information. In this case there is only a single HOA stream of (nOrder + 1)^2 channels
for (unsigned int i = 0; i < nHoaCh; ++i)
{
    streamInfo.nChannels++;
    streamInfo.typeDefinition.push_back(TypeDefinition::HOA);

    // Set the metadata for each channel
    int order, degree;
    ComponentToOrderAndDegree(i, true, order, degree);
    hoaMetadata.degrees.push_back(degree);
    hoaMetadata.orders.push_back(order);
    hoaMetadata.trackInds.push_back(trackInd++);
}

// Add a DirectSpeaker stream ==========================
std::vector<float> speakerSinewave(nSigSamples);
// Fill the vector with a sine wave
for (int i = 0; i < nSigSamples; ++i)
    speakerSinewave[i] = (float)std::sin((float)M_PI * 2.f * 185.f * (float)i / (float)sampleRate);

// Prepare the metadata for the stream
DirectSpeakerMetadata directSpeakerMetadata;
directSpeakerMetadata.trackInd = trackInd++;
directSpeakerMetadata.speakerLabel = "M-060"; // Wide-right speaker
directSpeakerMetadata.audioPackFormatID = std::string("AP_00010009"); // 9+10+3 layout

// Configure the stream information. In this case there is only a single channel stream
streamInfo.nChannels++;
streamInfo.typeDefinition.push_back(TypeDefinition::DirectSpeakers);
```

```
// Render the stream =================================
// Set up the Renderer
Renderer renderer;
renderer.Configure(OutputLayout::Binaural, nOrder, sampleRate, nBlockLength, streamInfo);

float** renderBlock = new float* [2];
float** hoaBlock = new float* [nHoaCh];

for (int iBlock = 0; iBlock < nBlocks; ++iBlock)
{
    const unsigned int iSamp = iBlock * nBlockLength;

    // Get "new" Object metadata
    objectMetadata.position.polarPosition().azimuth = (double)iBlock / (double)nBlocks * 360.;
    // Add the Object stream to be rendered
    renderer.AddObject(&objectStream[iSamp], nBlockLength, objectMetadata);

    // Add the HOA stream to be rendered
    for (unsigned int i = 0; i < nHoaCh; ++i)
        hoaBlock[i] = &hoaStream[i][iSamp];
    renderer.AddHoa(hoaBlock, nBlockLength, hoaMetadata);

    // Add the DirectSpeaker stream to be rendered
    renderer.AddDirectSpeaker(&speakerSinewave[iSamp], nBlockLength, directSpeakerMetadata);

    // Render the stream
    renderBlock[0] = &renderStream[0][iSamp];
    renderBlock[1] = &renderStream[1][iSamp];
    renderer.GetRenderedAudio(renderBlock, nBlockLength);
}

// Cleanup
for (unsigned i = 0; i < nHoaCh; ++i)
    delete[] hoaStream[i];
delete[] hoaStream;
for (unsigned i = 0; i < nLdspk; ++i)
    delete[] renderStream[i];
delete[] renderStream;
delete[] renderBlock;
delete[] hoaBlock;
```

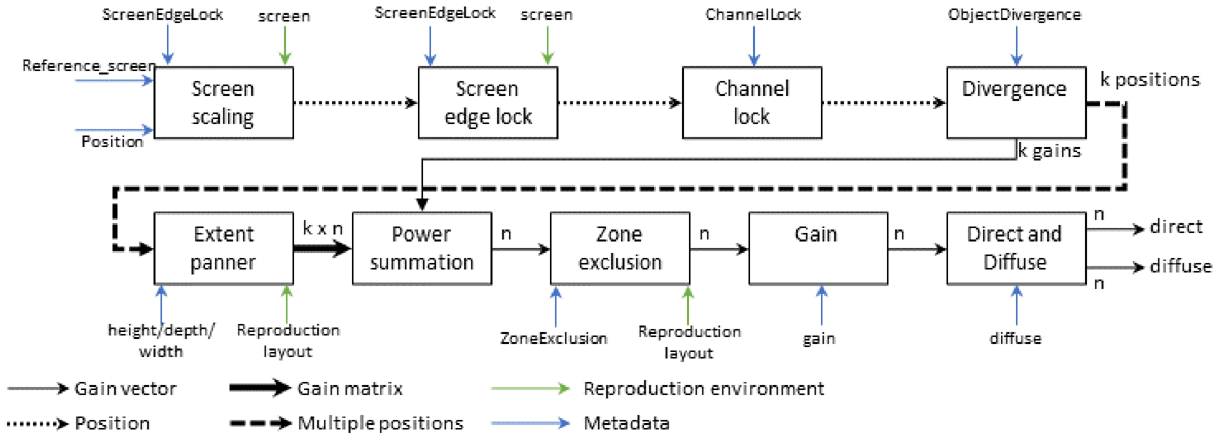


Figure 5.1: The processing logic for calculating direct and diffuse gains for the Polar path

5.2 ADM Object Rendering

5.2.1 Overview

When rendering an Object a set of loudspeaker gains are calculated that will pan the object to the direction specified by the metadata. The metadata controls properties such as the width of the sound source, as well as whether or not its position is to be modified based on the reproduction system. The ratio of direct to diffuse signal is also specified in the metadata. The Object direct and diffuse panning gains are computed by `GainCalculator`. Depending on whether the metadata Cartesian flag is true or false the gains will be calculated in one of two ways, described in the following sections.

Polar Processing Path

When the Cartesian flag is set to false the polar processing path is used to render the stream. Figure 5.1 shows the processing logic for the calculation of the direct and diffuse gains in `GainCalculator`.

The first processing stage relates to whether or not the position of the Object is to be modified. If a reproduction screen has been specified then the Object metadata can request the position of the Object be modified based on the screen properties, potentially modifying the position of the Object. Following that, it is also possible to ensure that the object is sent only to a single loudspeaker, if one is close enough. This is known as Channel Locking. If no loudspeaker is within range then the Object stream is panned using its position.

Once the position of the Object has been determined, ObjectDivergence can be applied. If the metadata specifies that Divergence be used then two additional positions are added to the Object position. These positions are to the left and right of the Object position and their distance is specified in the metadata. Each of these positions also has a specified gain based on the amount of divergence. For zero divergence the original Object direction has a gain of unity and the others have zero gain. For full divergence the original Object direction has zero gain and the left and right positions have gains of 0.5.

Once the divergence processing is complete the up-to-three divergence positions are each used to calculate a corresponding vector of panning gains based on their positions. The gains are calculated using `SpreadPanner`, which allows for an azimuthal width and elevational height to be specified in the metadata. The spread panner essentially pans copies of the Object to a set of grid positions that cover the full sphere. A weight between 0 and 1 is applied to each of these grid positions based on the metadata width and height parameters before normalisation of the total weight by the sum of the grid weights. The larger the width and height of the specified spread, the more of the grid points have non-zero gains, leading to a larger source width. Each grid point then has panning gains calculated using vector base amplitude panning (VBAP) [13] based on their position. These gains are multiplied by their corresponding weights and summed to get the final panning gains for each of the three divergence positions. Each of these three vectors of gains are summed using the divergence gains and they are power-normalised.

If ZoneExclusion is specified in the metadata then any loudspeaker in the excluded zone will have their gains shared out among other groups of loudspeakers. The full method for calculating the downmixing matrix is given in section 7.3.12.2 of [2].

A global gain is applied to the spatialisation gains before calculation of direct and diffuse gain vectors. The direct and diffuse gain vectors are generated by multiplying the spatialisation gains by $\sqrt{1-d}$ and $\sqrt{d}$ respectively, where d is the diffuseness parameter specified in the metadata. The Object signal is then multiplied by the direct and diffuse gain vectors. The diffuse signal has a set of decorrelation filters of length N applied to each channel, creating a less well localisable signal. After appropriate compensation delay of $(N-1)/2$ samples is applied to the direct signal the direct and diffuse signals are summed.

Cartesian Processing Path

If the Cartesian flag is set to true then the cartesian processing path is used to render the stream. Figure 5.2 shows the processing logic for the calculation of the direct and diffuse gains in `GainCalculator`.

First the zone exclusion metadata is used to create a vector of masks for the loudspeakers in the reproduction layout. These are used when calculating channel locking and extent panning later in the path.

Like the Polar path, the position of the Object is modified using screen scaling,

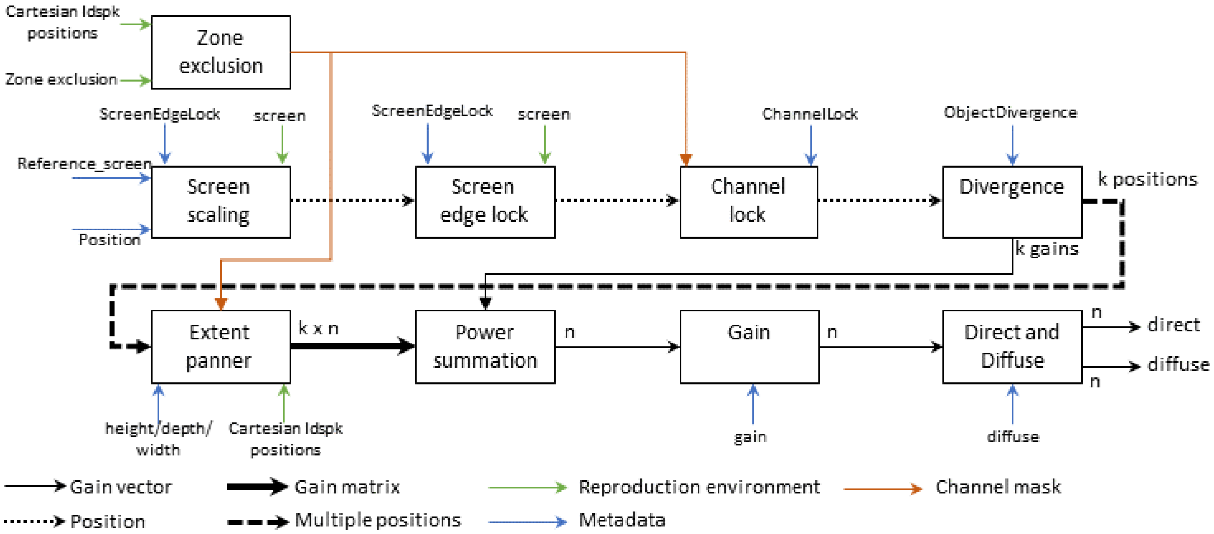


Figure 5.2: The processing logic for calculating direct and diffuse gains for the Polar path

screen edge lock and channel locking. The channel mask from the zone exclusion is used to ensure that the object is not locked to a channel that has been excluded.

Once the position of the Object has been modified ObjectDivergence is applied. This can create additional copies of the object panned to the left and right of its position.

For each of the divergence positions a set of loudspeaker gains is calculated using the `AllocentricExtent`, which uses the height, width and depth metadata parameters. The panning uses the Cartesian/Allocentric panner rather than the PointSourcePanner of the Polar path.

These gain vectors are multiplied by their corresponding divergence gains and summed to get the final panning gains for the Object. They are then power normalised and a gain is applied based on the metadata parameter.

Finally, the direct and diffuse gains are calculated in the same manner as for the Polar path.

5.2.2 Code Example

This example simulated receiving a mono Object stream and its meta data. As written here, the object is rendered on a 5.1 layout at an angle of 20° . Try adjusting the channel locking, divergence and width to see the effect on the rendering. For example, increasing the channel lock distance will cause the signal to snap to the left speaker completely.

```
#include "AdmRenderer.h"
```

```cpp

using namespace spaudio;

const unsigned int sampleRate = 48000;
const int nBlockLength = 512;

// Ambisonic order (not used in this example but expected in Configure())
const unsigned int nOrder = 1;

std::vector<float> sinewave(nBlockLength);
// Fill the vector with a sine wave
for (int i = 0; i < nBlockLength; ++i)
    sinewave[i] = (float)std::sin((float)M_PI * 2.f * 440.f * (float)i / (float)sampleRate);

// Prepare the stream to hold the rendered audio (5.1)
const unsigned int nLdspk = 6;
float** renderStream = new float* [nLdspk];
for (unsigned int i = 0; i < nLdspk; ++i)
{
    renderStream[i] = new float[nBlockLength];
}

// Prepare the metadata for the stream
ObjectMetadata objectMetadata;
objectMetadata.trackInd = 0;
objectMetadata.blockLength = nBlockLength;
objectMetadata.cartesian = false;
objectMetadata.position.polarPosition().azimuth = 20.;
objectMetadata.position.polarPosition().elevation = 0.;
objectMetadata.position.polarPosition().distance = 1.;

// Channel lock (off by default)
objectMetadata.channelLock = ChannelLock();
objectMetadata.channelLock->maxDistance = 0.f; // <- Increase this to see the signal snap to fully in the left loudspeaker

// Object divergence (off by default)
objectMetadata.objectDivergence = ObjectDivergence();
objectMetadata.objectDivergence->azimuthRange = 30.; // <- Controls the width of the divergence
objectMetadata.objectDivergence->value = 0.; // <- Increase this to apply object divergence

// Object extent (off by default)
objectMetadata.width = 0.; // <- Increase this to increase the width and spread the Object over more adjacent loudspeakers

// Configure the stream information. In this case there is only a single channel stream
StreamInformation streamInfo;
streamInfo.nChannels = 1;
streamInfo.typeDefinition.push_back(TypeDefinition::Objects);

```

```
// Set up the Renderer
Renderer renderer;
renderer.Configure(OutputLayout::FivePointOne, nOrder, sampleRate,
    nBlockLength, streamInfo);

// Add the Object stream to be rendered
renderer.AddObject(sinewave.data(), nBlockLength, objectMetadata);

// Render the stream
renderer.GetRenderedAudio(renderStream, nBlockLength);

// Cleanup
for (unsigned i = 0; i < nLdspk; ++i)
    delete[] renderStream[i];
delete[] renderStream;
```

5.3 ADM HOA Rendering

5.3.1 Overview

`Renderer` allows for Higher Order Ambisonics (HOA) streams to be decoded to any of the specified loudspeaker layouts. It supports decoding up to a maximum of order 3. Decoding is performed using the AllRAD method, using the `AmbisonicAllRAD` class internally. See section 2.6 for more details about the AllRAD method.

The HOA stream has the "order" and "degree" of each channel defined in its metadata which is used to sort the channels of the stream. `Renderer` uses the ACN channel sorting internally.

The normalisation of the orders can be SN3D, N3D or FuMa. The signals are converted to SN3D, if they are not already.

Converting to ACN channel sorting and SN3D normalisation ensures the signal is in the AmbiX format, ensuring the signals are compatible with the `libspatialaudio` Ambisonics processors.

Full details can be found in section 9 of Rec. ITU-R BS.2127-1 [2].

5.3.2 Code Example

In this example an HOA signal is decoded to binaural. The AmbiX (SN3D/ACN) HOA stream and its metadata are generated as part of the example but would normally be received from the stream to be rendered.

The HOA stream is then added to the renderer using `AddHOA()`, which adds it to an internal buffer. `AddHOA()` can be called multiple times if there are multiple HOA streams to be rendered. When `GetRenderedAudio()` is called the accumulated HOA buffer is decoded to the specified output format. The internal HOA buffer is then cleared so that new streams can be added to the next frame to be rendered.

For a more complete example with multiple different stream types being rendered see section 5.1.2.

```cpp
#include "AdmRenderer.h"

using namespace spaudio;

const unsigned int sampleRate = 48000;
const int nBlockLength = 512;

// Higher ambisonic order means higher spatial resolution and more channels required
const unsigned int nOrder = 1;
const unsigned int nHoaCh = OrderToComponents(nOrder, true);
// Set the fade time to the length of one block
const float fadeTimeInMilliSec = 1000.f * (float)nBlockLength / (float)sampleRate;

std::vector<float> sinewave(nBlockLength);
// Fill the vector with a sine wave
for (int i = 0; i < nBlockLength; ++i)
    sinewave[i] = (float)std::sin((float)M_PI * 2.f * 440.f * (float)i / (float)sampleRate);

// Destination B-format buffer
BFormat myBFormat;
myBFormat.Configure(nOrder, true, nBlockLength);
myBFormat.Reset();

// Set up and configure the Ambisonics encoder
AmbisonicEncoder myEncoder;
myEncoder.Configure(nOrder, true, sampleRate, fadeTimeInMilliSec);

// Set test signal's initial direction in the sound field
PolarPosition<float> position;
position.azimuth = 0.5f * (float)M_PI;
position.elevation = 0;
position.distance = 1.f;
myEncoder.SetPosition(position);
myEncoder.Reset();
myEncoder.Process(sinewave.data(), nBlockLength, &myBFormat);

// Prepare the stream to be processed by the renderer
float** hoaStream = new float* [nHoaCh];
for (unsigned int i = 0; i < nHoaCh; ++i)
{
    hoaStream[i] = new float[nBlockLength];
    myBFormat.ExtractStream(hoaStream[i], i, nBlockLength);
}

// Prepare the stream to hold the rendered audio (binaural)
```

```cpp
float** renderStream = new float* [2];
for (unsigned int i = 0; i < 2; ++i)
{
    renderStream[i] = new float[nBlockLength];
}

// Prepare the metadata for the stream
HoaMetadata hoaMetadata;
hoaMetadata.normalization = "SN3D";

// Configure the stream information. In this case there is only a single HOA stream of (nOrder + 1)^2 channels
StreamInformation streamInfo;
for (unsigned int i = 0; i < nHoaCh; ++i)
{
    streamInfo.nChannels++;
    streamInfo.typeDefinition.push_back(TypeDefinition::HOA);

    // Set the metadata for each channel
    int order, degree;
    ComponentToOrderAndDegree(i, true, order, degree);
    hoaMetadata.degrees.push_back(degree);
    hoaMetadata.orders.push_back(order);
    hoaMetadata.trackInds.push_back(i);
}

// Set up the Renderer
Renderer renderer;
renderer.Configure(OutputLayout::Binaural, nOrder, sampleRate, nBlockLength, streamInfo);

// Add the HOA stream to be rendered
renderer.AddHoa(hoaStream, nBlockLength, hoaMetadata);

// Render the stream
renderer.GetRenderedAudio(renderStream, nBlockLength);

// Cleanup
for (unsigned i = 0; i < nHoaCh; ++i)
    delete[] hoaStream[i];
delete[] hoaStream;
for (unsigned i = 0; i < 2; ++i)
    delete[] renderStream[i];
delete[] renderStream;
```

5.4 ADM DirectSpeaker Rendering

5.4.1 Overview

A mono audio stream with DirectSpeaker metadata is spatialised by sending it directly to the specified loudspeaker i.e. a DirectSpeaker stream labelled as belonging to the left loudspeaker will be sent to it directly. If the specified loudspeaker is not present in the reproduction layout then it checks a set of MappingRules. In the case a suitable mapping rule exists, it will be used to calculate the gains to be applied to the stream.

As a fallback, or if no label is supplied, the position of the DirectSpeaker stream can be specified. After screen edge locking, the position of the stream is checked and if there is a sufficiently close loudspeaker then the signal will be sent directly to it. If no loudspeaker is close enough then the DirectSpeaker stream is panned as a point source to its specified direction.

Full details can be found in section 8 of Rec. ITU-R BS.2127-1 [2].

5.4.2 Code Example

In this example a DirectSpeaker signal for the wide-left (M+060) loudspeaker from a 9+10+3 layout is rendered to a 5.1 layout. The stream and its metadata are generated as part of the example but would normally be received from the stream to be rendered.

The stream is first added to be rendered using `AddDirectSpeaker()`. When `GetRenderedAudio()` is called the stream is converted (in this case) to binaural and the DirectSpeaker buffer internally is cleared, ready for the next `AddDirectSpeaker()` call.

```cpp
#include "AdmRenderer.h"

using namespace spaudio;

const unsigned int sampleRate = 48000;
const int nBlockLength = 512;

// Ambisonic order (not used in this example but expected in Configure())
const unsigned int nOrder = 1;

std::vector<float> sinewave(nBlockLength);
// Fill the vector with a sine wave
for (int i = 0; i < nBlockLength; ++i)
    sinewave[i] = (float)std::sin((float)M_PI * 2.f * 440.f * (float)i / (float)sampleRate);

// Prepare the stream to hold the rendered audio (5.1)
const unsigned int nLdspk = 6;
float** renderStream = new float* [nLdspk];
for (unsigned int i = 0; i < nLdspk; ++i)
{
    renderStream[i] = new float[nBlockLength];
}

// Prepare the metadata for the stream
DirectSpeakerMetadata directSpeakerMetadata;
```

```
directSpeakerMetadata.trackInd = 0;
directSpeakerMetadata.speakerLabel = "M+060"; // Wide-left speaker
directSpeakerMetadata.audioPackFormatID = std::string("AP_00010009"); // 9+10+3 layout

// Configure the stream information. In this case there is only a single channel stream
StreamInformation streamInfo;
streamInfo.nChannels = 1;
streamInfo.typeDefinition.push_back(TypeDefinition::DirectSpeakers);

// Set up the Renderer
Renderer renderer;
renderer.Configure(OutputLayout::FivePointOne, nOrder, sampleRate, nBlockLength, streamInfo);

// Add the DirectSpeaker stream to be rendered
renderer.AddDirectSpeaker(sinewave.data(), nBlockLength, directSpeakerMetadata);

// Render the stream
renderer.GetRenderedAudio(renderStream, nBlockLength);

// Cleanup
for (unsigned i = 0; i < nLdspk; ++i)
    delete[] renderStream[i];
delete[] renderStream;
```

5.5 Binaural Rendering

5.5.1 Overview

The ADM specification does not have a specific implementation for rendering to binaural. EBU's BEAR [18] provides a method for rendering an ADM stream to binaural. However, BEAR requires a binaural room impulse responses (BRIRs) that would increase the size of `libspatialaudio` by too large a degree.

Therefore, the binaural rendering of `Renderer` is inspired by BEAR in its use of a virtual loudspeaker layout but the final binauralisation is performed differently, in order to remove the requirement for including more HRTFs than those used for `AmbisonicBinaurlization`.

BEAR uses a virtual 9+10+5 loudspeaker layout and then binauralises the loudspeaker signals using BRIRs in the corresponding directions based on the loudspeaker position and the listener head orientation.

`Renderer` uses the same virtual loudspeaker layout for rendering Object and DirectSpeaker signals. These loudspeaker signals are then converted to Ambisonics by encoding them based on the loudspeaker directions (ignoring listener head rotation). This encoded signal is then summed with any HOA streams that have

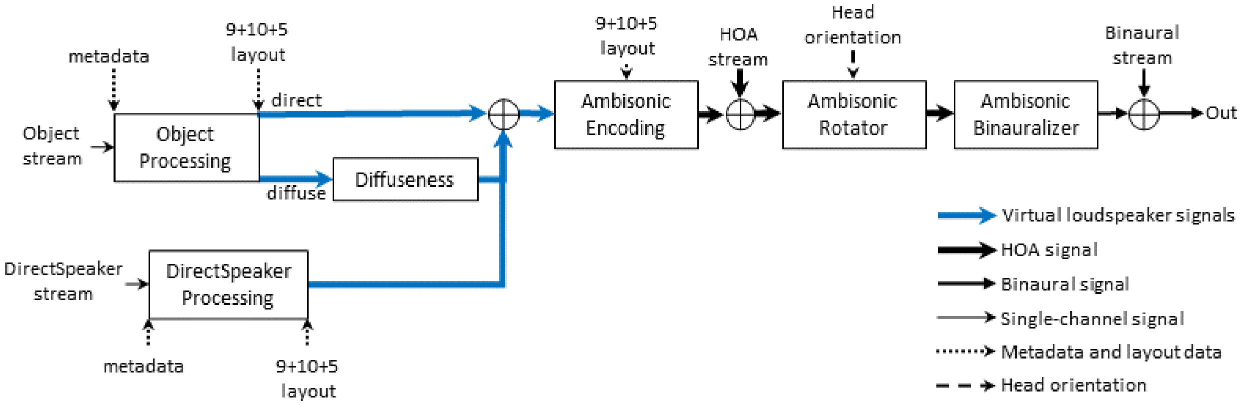


Figure 5.3: The signal flow for the `libspatialaudio` binaural ADM renderer.

been added. The composite stream is then rotated using 'AmbisonicRotator' to apply the listener head rotation. Finally, any direct binaural signal are added to the output. The signal flow is shown in the fig. 5.3.

Advantages and Disadvantages Compared to BEAR

The advantages of this method, besides the fact that no new HRTFs need to be added to the library, are that:

- the number of expensive convolution is kept low. BEAR needs 48 convolutions (24 loudspeakers, 2 ears) to be performed, compared to 32 using the described method (16 channels, 2 ear). Furthermore, since BEAR uses BRIR the CPU load of the convolutions is increased compared to using anechoic HRIRs.
- custom HRTFs can be easily loaded using the .SOFA file format.
- in case very low CPU use is required the binaural rendering can use 1st order Ambisonics internally, reducing the number of convolutions to only 8 (or 4 if the symmetric head assumption is used).

The main disadvantage of this method is that the lack of the room in rendered binaural could reduce externalisation when compared to the BRIR method. It has been shown that using a BRIR instead of an anechoic HRIR can improve externalisation.

5.5.2 Code Example

See section 5.1.2 in the main ADM overview for a full code example that renderers multiple different stream types to binaural.

Bibliography

[1] International Telecommunication Union. *Audio Definition Model*. International Telecommunication Union, Geneva, Switzerland, Recommendation ITU-R BS.2076-2 edition, 2019. URL `https://www.itu.int/rec/R-REC-BS.2076-2-201910-I/en`.

[2] International Telecommunication Union. *Audio Definition Model renderer for advanced sound systems*. International Telecommunication Union, Geneva, Switzerland, Recommendation ITU-R BS.2127-0 edition, 2019. URL `https://www.itu.int/rec/r-rec-bs.2127/en`.

[3] Christian Nachbar, Franz Zotter, Etienne Deleflie, and Alois Sontacchi. Ambix - A Suggested Ambisonics Format. In *Ambisonics Symposium*, volume 3, pages 1–11, Lexington, KY, 2011.

[4] Joseph Ivanic and Klaus Ruedenberg. Rotation matrices for real spherical harmonics. direct determination by recursion. *The Journal of Physical Chemistry*, 100(15):6342–6347, 1996.

[5] Stéphanie Bertet, Jérôme Daniel, Etienne Parizet, and Olivier Warusfel. Investigation on localisation accuracy for first and higher order Ambisonics reproduced sound sources. *Acta Acustica united with Acustica*, 99(4):642–657, 2013. doi: http://dx.doi.org/10.3813/AAA.918643.

[6] Jérôme Daniel. *Représentation de champs acoustiques, application à la transmission et à la reproduction de scènes sonores complexes dans un contexte multimédia*. PhD thesis, University of Paris 6, 2000.

[7] Franz Zotter and Matthias Frank. All-round ambisonic panning and decoding. *Journal of the Audio Engineering Society*, 60(10):807–820, 2012.

[8] Franz Zotter, Hannes Pomberger, and Markus Noisternig. Energy-Preserving Ambisonic Decoding. *Acta Acustica united with Acustica*, 98(1):37–47, January 2012. doi: http://dx.doi.org/10.3813/AAA.918490.

[9] MA Poletti. Three-Dimensional Surround Sound Systems Based on Spherical Harmonics. *J. Audio Eng. Soc*, 53(11):1004 –1025, 2005.

[10] Franz Zotter and Matthias Frank. *Ambisonics: A practical 3D audio theory for recording, studio production, sound reinforcement, and virtual reality*. Springer Nature, 2019.

[11] Bruce Wiggins. The generation of panning laws for irregular speaker arrays using heuristic methods. In *AES 31st International Conference*, London, 2007.

[12] Michael Gerzon. General metatheory of auditory localisation. In *92nd Convention of the Audio Engineering Society*, Vienna, March 1992.

[13] Ville Pulkki. Virtual sound source positioning using vector base amplitude panning. *Journal of the Audio Engineering Society*, 45(6):456–466, 1997.

[14] Markus Noisternig, Alois Sontacchi, Thomas Musil, and Robert Höldrich. A 3D Ambisonic Based Binaural Sound Reproduction System. In *24th International Conference of the Audio Engineering Society*, pages 1–5, June 2003.

[15] Christian Schörkhuber, Markus Zaunschirm, and Robert Höldrich. Binaural rendering of ambisonic signals via magnitude least squares. In *Proceedings of the DAGA*, volume 44, pages 339–342, 2018.

[16] Markus Zaunschirm, Christian Schörkhuber, and Robert Höldrich. Binaural rendering of Ambisonic signals by head-related impulse response time alignment and a diffuseness constraint. *The Journal of the Acoustical Society of America*, 143(6):3616–3627, 2018. ISSN 0001-4966. doi: 10.1121/1.5040489.

[17] Archontis Politis and David Poirier-Quinot. Jsambisonics: A web audio library for interactive spatial sound processing on the web. In *Interactive Audio Systems Symposium*, 2016.

[18] International Telecommunication Union. *Binaural EBU ADM Renderer for (BEAR) for object-based sound over headphones*. EBU, Geneva, Switzerland, EBU Tech 3369 edition, 2023. URL `https://tech.ebu.ch/docs/tech/tech3396.pdf`.