EBU

OPERATING EUROVISION AND EURORADIO

TECH 3388

ADM RENDERER FOR USE IN NEXT GENERATION AUDIO BROADCASTING

SOURCE: BTF RENDERER GROUP

SPECIFICATION Version 1.0

Geneva
March 2018

This page and others in the document are intentionally left blank to maintain pagination for two sided printing

Foreword

It has become apparent that the development of a standardised renderer is very important for the health of NGA systems. It is crucial for the content production, quality evaluation and verification of systems developed within the framework for NGA broadcast systems.

This document and the associated reference implementation define an audio renderer, designed by the EBU members, to interpret the ITU-R BS.2076 ADM metadata directly, rather than by conversion to other metadata sets used by commercial systems.

The EBU believes that providing an open reference renderer for ADM metadata interpretation during audio evaluation, production, and monitoring, will be beneficial to the health of the Next Generation Audio ecosystem as a whole.

Another aim is to provide a renderer specification and implementation without ambiguities, to ensure reliable and interoperable implementations of tools used throughout the NGA workflow.

Contents

ADM Renderer for use in Next Generation Audio Broadcasting

EBU Committee	*First Issued*	*Revised*	*Re-issued*
TC	2018		

Keywords: ADM, Audio Definition Model, Metadata, Renderer, NGA, Next Generation Audio.

1. Introduction

This document describes the **EBU ADM Renderer (*EAR*)** - an audio renderer providing a complete interpretation of the **Audio Definition Model (ADM)** metadata, specified in [BS.2076]. Usage of ADM metadata is recommended to describe audio formats used in programme production for **Next-Generation Audio (NGA)** systems, also known as *Advanced Sound Systems*. The EBU ADM Renderer is capable of rendering audio signals to all reproduction systems specified in [BS.2051] ("Advanced sound system for programme production").

This specification is accompanied by an **open source reference implementation**, written in Python for file-based ADM processing, available at https://github.com/ebu/ebu_adm_renderer. This specification document is a description of the reference code.

1.1 Abbreviations and Definitions

This section provides explanations and definitions for certain terms that are used throughout the document.

1.1.1 Abbreviations

ADM	Audio Definition Model
BMF	Broadcast Metadata Exchange Format
BW64	Broadcast Wave 64 Format
BWF	Broadcast Wave Format
EAR	EBU ADM Renderer
HOA	Higher-order Ambisonics
IMF	Interoperable Master Format
MXF	Material Exchange Format
NGA	Next Generation Audio
PSP	Point Source Panner
VBAP	Vector Base Amplitude Panning
XML	Extensible Markup Language

1.1.2 Next Generation Audio

The multimedia world is moving towards a more involving experience for the audience, with enhanced interactivity and immersion. NGA systems aim to provide such enhancements to the listening experience. Immersive audio (also called 3D or spatial audio) implies that sound can appear to come from any direction around the listener, including above and below. To achieve this, NGA systems must support an increased range of reproduction systems (loudspeaker layouts), and so be capable of adapting input signals to the output system. Interactivity can, for example, include the ability for listeners to adjust dialogue levels, change the positions of sounds, or select different languages according to their needs or preferences.

To achieve immersive audio, there are three fundamental approaches in NGA systems: channel-based, scene-based and object-based. With channel-based audio, each channel relates directly to a loudspeaker in a particular location (examples are stereo, 5.1 and 22.2). Scene-based audio represents sound by a combination of dimensional components that combine to make a soundfield, with Ambisonics (and Higher Order Ambisonics - HOA) being the primary technique used to effect this. With scene-based audio the soundfield has to be decoded to a chosen speaker layout or to a binaural signal for use with headphones. Object-based audio represents the sound scene as separate elements (e.g. singer, drums), and adds metadata, e.g. position information, to them, so they can be rendered to be played out from the correct location. Each approach has pros and cons and it is likely that all three will be used to represent programmes, maybe separately, but also they might also be used in combination.

Interactivity can be achieved by using object-based audio. In sending audio objects separately to the end-user, they can easily control how those objects are rendered. Both channel- and scene-based soundfields can be represented as audio objects, it can be considered that all audio can be treated as object-based.

1.1.3 The Audio Definition Model

The ADM [BS.2076] is a general audio format description model, based on XML (but it could be extended to other languages). One of its first applications is an extension to the Broadcast Wave 64 Format (BW64) [BS.2088] file which includes an <axml> chunk to allow the carriage of XML metadata. As the ADM metadata describing the audio format is given in XML, it can readily be inserted into the BW64 file. The ADM provides the necessary metadata for *object-based*, *scene-based* and *channel-based* audio. Moreover, *binaural* signals can be represented in ADM and the *Matrix* type allows the description of parameters for encoding or decoding matrixed audio signals, to be used along with channel-based content, such as mid-side and Lt/Rt representations.

The ADM is designed to describe audio formats as completely as possible. It is not intended to give instructions on how the audio is rendered. For example, a stereo file contains two channels intended for speakers positioned at $-30°$ and $+30°$ azimuth. The ADM metadata can describe this explicitly. What it does not do is to specify how to reproduce these signals in a given reproduction system, such as a standard stereo speaker arrangement, headphones, or a wave field synthesis speaker array. This is the role of the renderer, given the description of the audio input format and the target reproduction system. The ADM metadata should provide enough information for a renderer to fulfil its requirements.

To summarise, the ADM is designed to allow any audio format to be fully specified so it can be processed or rendered correctly.

1.1.4 Renderer

A critically important element of a NGA system is a renderer. A renderer turns input audio signals with accompanying format metadata into output audio signals for a specific reproduction setup (e.g. a stereo loudspeaker pair). A renderer is required to enable reproduction of the above object-based and scene-based (HOA) audio format types, as well as the conversion of a channel-based format from one loudspeaker layout to others. It is similar to a decoder which turns encoded audio into listenable audio, and is after, or integrated into, the decoding process in NGA codec systems. A renderer needs to be used at all points where one wants to listen to the content (e.g. production, monitoring, quality control, archive, consumer devices).

1.2 The EBU ADM Renderer (EAR)

1.2.1 Scope of the EBU ADM Renderer

The initial release of the renderer provides specifications and implementations for all three NGA technologies (object-based, scene-based and channel-based). In terms of ADM metadata, this means the following ADM *typeDefinitions* are supported:

- Objects
- DirectSpeakers
- HOA

All sub-elements and parameters of these *typeDefinitions* are supported by the ADM Renderer unless otherwise explicitly stated.

Specifications and implementations for the other *typeDefinitions* '*Matrix*' and '*Binaural*' will be published in the next major release of the EBU ADM Renderer.

1.2.2 Intended use cases for the EBU ADM Renderer

The use cases for this renderer are:

- Production of NGA programmes
- Archiving of NGA programmes
- Verification of ADM metadata
- Subjective evaluations
- Conversion of metadata from different NGA systems to ADM metadata

1.2.2.1 Production of NGA programmes

The *EAR* can be used for authoring and monitoring of NGA programmes during the entire production process. It allows the sound engineer to make use of all ADM parameters for channel-based, scene-based and object-based NGA technologies. By providing an open and well-defined specification and reference implementation, reliability and interoperability can be achieved in production tools.

1.2.2.2 Archiving of NGA programmes

The *EAR* is suitable for archiving NGA programmes. By providing an open and well-defined specification and reference implementation, with strict version controls, the correct interpretation of ADM metadata in archived programme files can be ensured long-term.

It could be used to provide a reliable interpretation for any media format which can carry the ADM metadata. Besides BW64 for audio-only content, there are currently multiple options available for archival of audio-with-video content (MXF, IMF or BMF).

1.2.2.3 Verification of ADM metadata

The *EAR* directly and explicitly interprets the ADM metadata parameters. It can therefore be used to read, parse, and render audio signals with ADM format metadata for the purpose of verifying the metadata.

1.2.2.4 Subjective evaluations

Being developed by EBU members, without commercial interest, the *EAR* may be used in subjective evaluations. It may serve as a benchmark or anchor renderer during the perceptual evaluation of other NGA systems, for example.

1.2.2.5 Conversion of metadata from different NGA systems to ADM metadata

Multiple NGA renderers are available; often they are associated with vendor-specific NGA codec systems. It is likely different renderers will be used in the production process of different broadcasters. Other renderers may only support a limited subset of ADM parameters or may convert the incoming and outgoing ADM metadata to internal metadata sets. It is currently not clear what the differences between these alternatives are and how conversion should appropriately be applied, but it is crucial for a horizontal market to be able to convert between these alternatives without changing the perceived impression of the programme.

For the conversion of NGA content produced with different renderers or to be distributed with different NGA codecs, the *EAR* may be used as a neutral instance.

2. Conventions

2.1 Notations

In this document the following conventions will be used:

- Text in italic refers to ADM elements, sub-elements, parameters or attributes of [BS.2076]: *audioObject*
- Monospaced text refers to source code (variables, functions, classes) of the reference implementation: `core.point_source.PointSourcePanner`. It should be noted that for readability reasons the prefix `ear.` is omitted.
- Upper case bold is used for matrices: $\mathbf{X}$
- Lower case bold is used for vectors: $\mathbf{x}$
- Subscripts in the form x_n denotes the n-th element of a vector $\mathbf{x}$
- Sections of monospaced text with colour highlighting are used to describe data structures:

```
struct PolarPosition : Position {
  float azimuth, elevation, distance = 1;
};
```

2.2 Coordinate System

Both Cartesian and polar coordinates are used throughout this document.

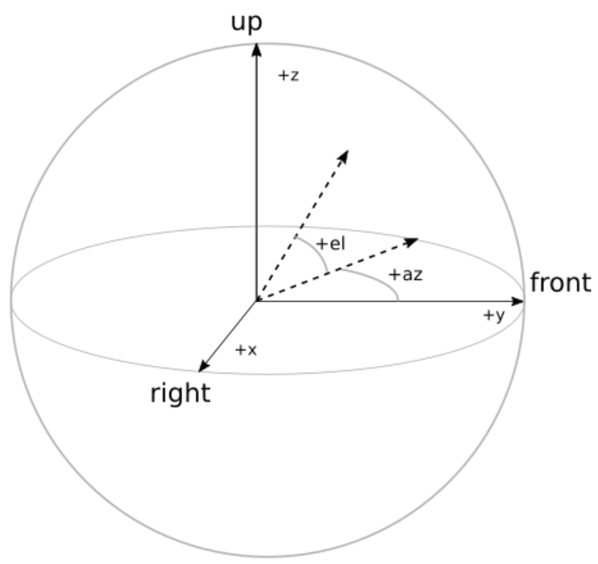


Figure 1: Coordinate System

The polar coordinates are specified in accordance with [BS.2076] as follows:

- Azimuth, denoted by φ, is the angle in the horizontal plane, with 0 degrees in front and positive angles counter-clockwise.
- Elevation, denoted by θ, is the angle above the horizontal plane, with 0 degrees in front and positive angles going up.

The Cartesian coordinates are specified in accordance with [BS.2076] as follows:

- The positive Y-Axis is pointing to the front
- The positive X-Axis is pointing to the right
- The positive Z-Axis is pointing to the top

The HOA decoder specified in § 9 uses the HOA coordinate system and notation as specified in [BS.2076], where:

- Elevation, denoted by θ is the angle in radians from the positive Z-Axis.
- Azimuth, denoted by ϕ, is the angle in the horizontal plane in radians, with 0 in front and positive angles counter-clockwise.

3. Structure

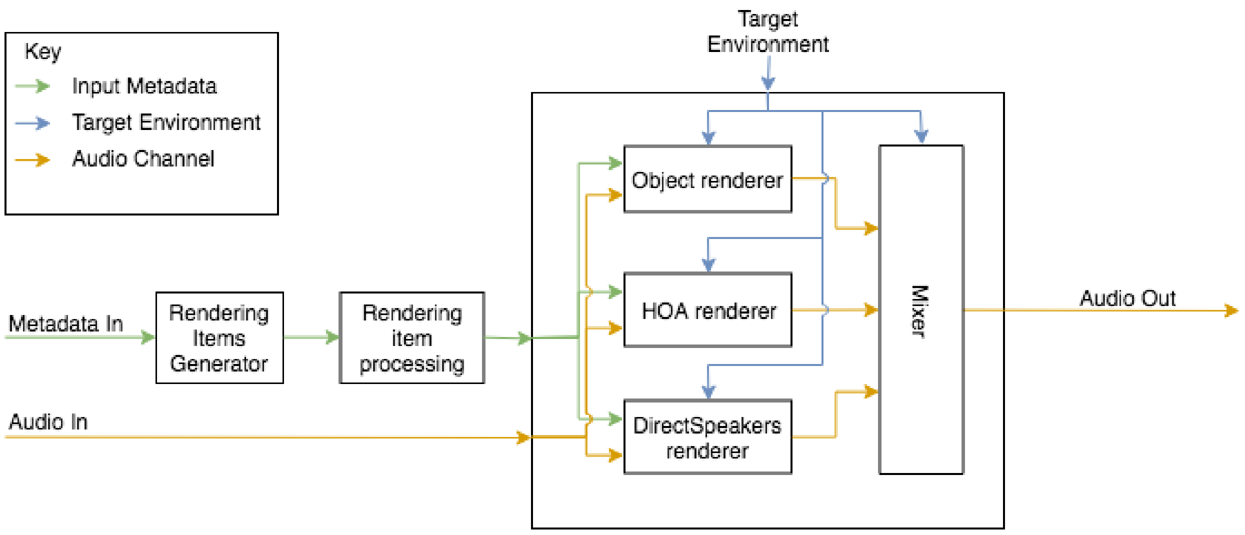


Figure 2: Overall architecture overview

The overall architecture consists of several core components and processing steps, which are described in the following chapters of this document.

- The transformation of ADM data to a set of renderable items is described in § 5
- The rendering itself is split into subcomponents based on the type (*typeDefinition*) of the item:
  - Rendering of object-based content is described in § 7
  - Rendering of direct speaker signals is described in § 8
  - HOA Rendering is described in § 9
  - Shared parts for all components are described in § 6

4. ADM-XML Interface

ADM is a generic metadata model which can be represented naturally as an XML document. The following subsections describe how the ADM is mapped to internal data structures. These are used in the course of this document and are in line with the data structures used by the reference implementation.

It should be noted that despite XML being the typical and common form to represent ADM metadata, *EAR* is not limited to this representation.

The mapping between the ADM and the internal data structures follows a set of simple rules, which are described below. As with all rules, there are some exceptions; these are described in the following subsections.

All the main ADM elements are represented as a subclass derived from ADMElement which has the signature:

```
class ADMElement {
  string id;
  ADM adm_parent;
  bool is_common_definition;
};
```

- Each ADM element class is extended with all the ADM attributes and sub-elements, which are mapped to class attributes.

- If a sub-element contains more than one value it is in itself a class. E.g. the *jumpPosition* sub-element is a class with the signature:

```
class JumpPosition {
  bool flag;
  float interpolationLength;
};
```

- During the parsing of the XML, references to other ADM elements are stored as plain IDs using the sub-element name as attribute name (e.g. `AudioObject.audioPackFormatIDRef`). To simplify the later on access, these references are then resolved in a following step, where resolved elements are added to each data structure directly (`AudioObject.audioPackFormats`).

Following these rules the full signature of the `AudioContent` element looks like this:

```
class AudioContent : ADMElement {
  string audioContentName;
  string audioContentLanguage;
  LoudnessMetaData loudnessMetadata;
  int dialogue;
  vector<AudioObject*> audioObjects;
  vector<string> audioObjectIDRef;
};
```

The main ADM elements and its dedicated classes are implemented in `fileio.adm.elements.main_elements`. The reference resolving is implemented in each class (in ADM and each main ADM element) as the `lazy_lookup_references` method.

The parsing and writing of the ADM is implemented in `fileio.adm.xml`.

4.1 AudioBlockFormat

audioBlockFormat differs from other ADM elements as its sub-elements and attributes are different depending on the *typeDefiniton*. To reflect this, the `AudioBlockFormat` is split into multiple classes, one for each supported *typeDefinition*: `AudioBlockFormatObjects`, `AudioBlockFormatDirectSpeakers` and `AudioBlockFormatHoa`.

These are implemented in `fileio.adm.elements.block_formats`.

4.2 Position sub-elements

Positions may be represented by multiple *position* sub-elements in the ADM. To simplify the internal handling, the values of these sub-elements are combined into a single attribute within the `AudioBlockFormat` representation.

For *typeDefinition==Objects* this is either `ObjectPolarPosition` or `ObjectCartesianPosition`, depending on the coordinate system used. For *typeDefinition==DirectSpeakers* this is `DirectSpeakerPolarPosition` or `DirectSpeakerCartesianPosition`.

4.3 TypeDefinition

The *typeDefinition* and *typeLabel* attributes describe one single property. For that reason, internally only a single entity is used to represent them.

```
enum TypeDefinition {
  DirectSpeakers = 1;
  Matrix = 2;
  Objects = 3;
  HOA = 4;
  Binaural = 5;
};

enum FormatDefinition {
  PCM = 1;
};
```

5. Rendering Items

A `RenderingItem` is a representation of an ADM item to be rendered - holding all the information necessary to do so. An item can thereby be a single *audioChannelFormat* or a group of *audioChannelFormats*. As each *typeDefinition* has different requirements it is necessary to have different metadata structures for each *typeDefinition* to adapt to its specific needs.

The following section describes the used metadata structures in more detail.

5.1 Metadata Structures

The `RenderingItems` are built upon the following base classes:

- `TypeMetadata` to hold all the metadata needed to render the item,
- `ImportanceData` to hold the effective importance values for this item,
- MetadataSource to iterate over the TypeMetadata and
- `RenderingItem` to combine them and hold the `track_indices` corresponding to the channels within the accompanying audio data.

As each *typeDefinition* has different requirements `TypeMetadata` and `RenderingItem` have to be subclassed for each *typeDefinition* to adapt to its specific needs. `MetadataSource` is *typeDefinition* independent. Common data is consolidated in `ExtraData`.

```
struct ExtraData {
  optional<duration> object_start;
  optional<duration> object_duration;
  ReferenceScreen reference_screen;
  Frequency channel_frequency;
};

struct ImportanceData {
  optional<int> audio_object;
  optional<int> audio_pack_format;
};
```

This is implemented in `core.utils.metadata_input`. The following subsections describe the specific implementations for each *typeDefinition* in more detail.

5.1.1 DirectSpeakers

For *typeDefinition==DirectSpeakers* the `TypeMetadata` just holds an `audioBlockFormat` plus the common data collected in `ExtraData`.

```
struct DirectSpeakersTypeMetadata {
  AudioBlockFormatDirectSpeakers block_format;
  ExtraData extra_data;
};
```

As each *audioChannelFormat* with *typeDefinition==DirectSpeakers* can be processed independently, the `RenderingItem` contains only a single `track_index`.

```
struct DirectSpeakersRenderingItem(RenderingItem):{
  int track_index;
  MetadataSource metadata_source;
  ImportanceData importance;
};
```

5.1.2 Matrix

As the *typeDefinition==Matrix* is not supported yet, there are no `MatrixTypeMetadata` and `MatrixRenderingItem` classes.

5.1.3 Objects

The `ObjectTypeMetadata` just holds an `audioBlockFormat` plus the common data collected in `ExtraData`.

```
struct ObjectTypeMetadata {
  AudioBlockFormatObjects block_format;
  ExtraData extra_data;
};
```

As each *audioChannelFormat* with *typeDefinition==Objects* can be processed independently, the `RenderingItem` also contains only a single `track_index`.

```
struct ObjectRenderingItem {
  int track_index;
  MetadataSource metadata_source;
  ImportanceData importance;
};
```

5.1.4 HOA

For *typeDefinition==HOA* the situation is different from *typeDefinition==DirectSpeakers* and *typeDefinition==Objects*, because a pack of *audioChannelFormats* has to be processed together. That is why the `HOATypeMetadata` does not contain an `audioBlockFormat` plus `ExtraData`, but the necessary information is extracted from the *audioBlockFormats* and directly stored in the `HOATypeMetadata`.

```
class HOATypeMetadata {
  vector<int> orders;
  vector<int> degrees;
  optional<string> normalization;
  optional<float> nfcRefDist;
  bool screenRef;
  ExtraData extra_data;
  optional<duration> rtime;
  optional<duration> duration;
};
```

For the same reason the situation for the `HOARenderingItem` is different. Here the `HOARenderingItem` not only contains one `track_index`, but rather a vector of `track_indices`.

```
class HOARenderingItem {
  vector<int> track_indices;
  MetadataSource metadata_source;
  ImportanceData importance;
};
```

5.1.5 Binaural

As the *typeDefinition==Binaural* is not supported yet, there are no `BinauralTypeMetadata` and `BinauralRenderingItem` classes.

5.2 Determination of Rendering Items

To determine the `RenderingItems`, the ADM structure has to be analysed. Figure 3 shows the necessary process path. The starting point is usually an *audioProgramme*. If the file contains multiple *audioProgrammes* the one with the lowest ID will be used by default. Alternatively the *audioProgramme* can be selected manually by its *audioProgrammeID*. If no *audioProgramme* is present the collection of all *audioObjects*, which are not referenced by another *audioObject*, will be the origin.

Crosschecking between the *audioPackFormats* referenced by each *audioObject* and its referenced *audioTrackUID* is performed to verify the consistency. Nested *audioObjects* are supported without a nesting limit, while reference loops are detected. As references from *audioTrackFormats* back to the *audioStreamFormat* are optional, the mapping from an *audioTrackFormat* to the *audioStreamFormat* is done reversely from the *audioStreamFormat*.

This is implemented in `fileio.utils.RenderingItemHandler`.

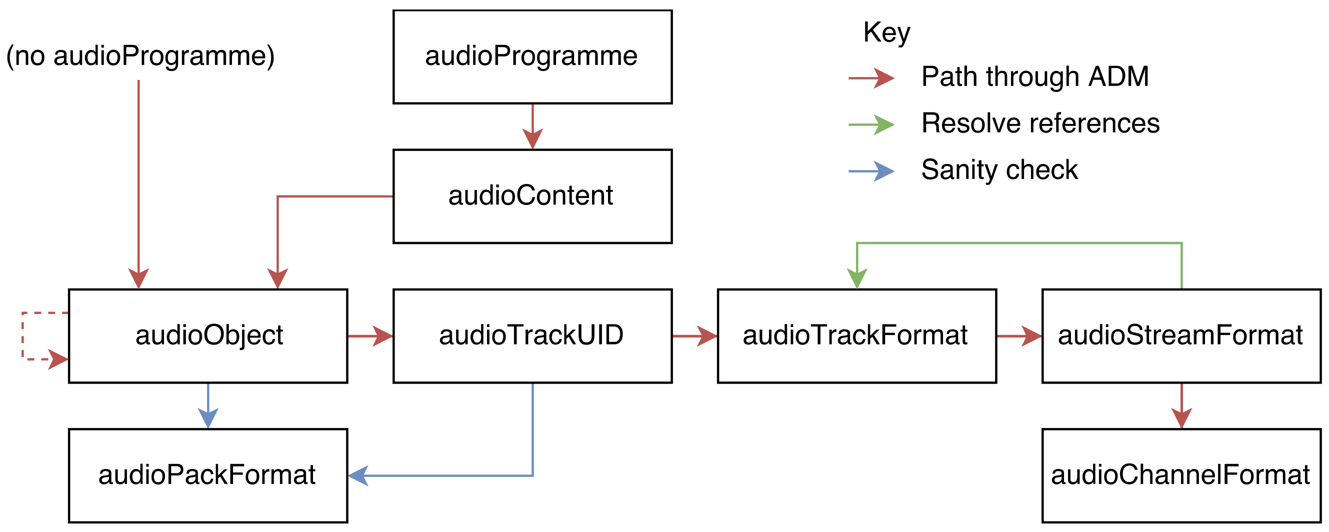


Figure 3: Path through ADM structure to determine the `RenderingItems`

5.3 Importance emulation

The *importance* parameters as defined by [BS.2076] allow a renderer to discard items below a certain level of importance for as yet undetermined, application specific reasons.

The ADM specifies three different *importance* parameters that can be used:

- importance as an audioObject attribute
- importance as an audioPackFormat attribute
- importance as an audioBlockFormat attribute for typeDefinition==Object

The most important difference between those *importance* attributes is that *audioBlockFormat* importance is time-depended, i.e. it may vary over time, while the importance of *audioObject* and *audioPackFormat* is static.

A separate threshold can be used for each *importance* attribute. The determination of desired threshold values is considered as highly application and use case specific and therefore out of scope of a production renderer specification. Instead *EAR* provides means to simulate the effect of applying a given importance threshold to the ADM. This enables content producers to investigate the effects of using *importance* values on the rendering. Therefore, the importance emulation is not part of the actual rendering process, but applied as a post processing step to the `RenderingItems`.

5.3.1 Importance values of `RenderingItems`

Each rendering item might have its own set of effective *importance* values, because *audioObjects* and *audioPackFormats* may be nested. Thus, for each `RenderingItem` all referencing *audioObjects* and *audioPackFormats* involved in the determination of this `RenderingItem` are taken into account.

The following rules are applied:

- If an *audioObject* has an *importance* value below the threshold, all referenced *audioObjects* shall be discarded as well. To achieve this, the lowest *importance* value of all *audioObjects* that lead to an `RenderingItem` will be used as the *audioObject importance* for this `RenderingItem`.
- If an *audioPackFormat* has an *importance* value below the threshold, all referenced *audioPackFormats* shall be discarded as well. To achieve this, the lowest *importance* value of all *audioPackFormats* that lead to an `RenderingItem` will be used as the *audioPackFormat importance* for this `RenderingItem`.
- An *audioObject* without *importance* value will not be taken into account when determining the *importance* of an `RenderingItem`.
- An *audioPackFormat* without *importance* value will not be taken into account when determining the *importance* of an `RenderingItem`

This is implemented in `fileio.utils.RenderingItemHandler`.

5.3.2 Static importance handling

Given a `RenderingItem` with `ImportanceData` , the item will be removed from the list of items to render if either the static importance value (*audioObject*, *audioPackFormat*) is below the respective user-defined threshold:

$$\begin{aligned} & \mathtt{importance.audio\_object} < \mathtt{audio\_object\_threshold} \\ \vee\ & \mathtt{importance.audio\_pack\_format} < \mathtt{audio\_pack\_format\_threshold} \end{aligned}$$

This is implemented in `core.importance.filter_audioObject_by_importance` and `core.importance.filter_audioPackFormat_by_importance`.

5.3.3 Time-varying importance handling

Importance handling on *audioBlockFormat* (*typeDefinition==Object*) level cannot be done by filtering `RenderingItems`, as this item might be below the threshold only for some time. To emulate discaring of rendering items in that particular case, the `RenderingItem` shall be effectively muted for the duration of the *audioBlockFormat*. In this context, "muting an *audioBlockFormat*" is equivalent to assuming `bf.gain` equal to zero for an *audioBlockFormat* `bf`.

This is implemented in `core.importance.MetadataSourceImportanceFilter`.

6. Shared Renderer Components

This section contains descriptions of components that are shared between the sub-renderers for the different *typeDefinitions*.

6.1 Point Source Panner

The point source panner component is the core of the renderer; given information about the loudspeaker layout, and a 3D direction, it produces one gain per loudspeaker which, when applied to a mono waveform and reproduced over loudspeakers, should cause the listener to perceive a sound emanating from the desired direction.

The point source panner is used throughout the renderer — it is used to render point sources specified by object metadata, as well as part of the extent rendering system, as a fall-back for the direct speakers renderer, and as part of the HOA decoder design process.

The point source panner in this renderer is based on the VBAP formulation [Pulkki1997], with several enhancements which make it more suitable for use in broadcast environments:

- In addition to the triplets of loudspeakers as in VBAP, the point source panner supports atomic quadrilaterals of loudspeakers. This solves the same problems as the use of virtual loudspeakers in other systems, but results in a smoother overall panning function.
- Triangulation of the loudspeaker layout is performed on the nominal loudspeaker positions and warped to match the real loudspeaker positions, which ensures that the panning behaviour is always consistent within adaptations of a given layout.
- Virtual loudspeakers and downmixing are used to modify the rendering in some situations in order to correct for observed perceptual effects and produce desirable behaviours in sparse layouts.
- To avoid complicating the design to cater for extremely restricted loudspeaker layouts, 0+2+0 is handled as a special case.

6.1.1 Architecture

The point source panner holds a list of objects with the `RegionHandler` interface; each region object is responsible for producing loudspeaker gains over a given spatial extent.

In order to produce gains for a given direction, the point source panner queries each region in turn, which will either return a gain vector if it can handle that direction, or a null result if it can't; the gain vector from the first region found that can handle the direction is used.

In any valid point source panner, the following two conditions hold:

- At least one region is able to handle any given direction.
- All regions which are able to handle a given direction result in similar gains (within some tolerance).
- Within any region, the produced gains are smooth with respect to the desired direction.

These properties together ensure that gains produced by a point source panner are well defined for all directions, and are always smooth with respect to the direction, within some tolerance.

The available `RegionHandler` types, and the configuration process used to generate the list of regions for a given layout are described in the next sections.

This behaviour is implemented in `core.point_source.PointSourcePanner`.

Additionally, a `PointSourcePannerDownmix` class is implemented with the same interface. When queried with a position, it calls another `PointSourcePanner` to obtain a gain vector, to which it applies a downmix matrix and power normalisation. This is used in § 6.1.3.1 to remap virtual loudspeakers.

6.1.2 Region Types

Most regions produce gains for a subset of the output channels; the mapping from this subset of channels to the full vector of channels is implemented in
`core.point_source.RegionHandler.handle_remap`.

6.1.2.1 Triplet

This represents a spherical triangular region formed by three loudspeakers, implementing basic VBAP.

This region is initialised with the 3D positions of three loudspeakers:

$$\mathbf{P} = [\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3]^T$$

The three output gains $\mathbf{g}$ for a given direction D are such that:

- $\mathbf{g} \cdot \mathbf{P} = s\mathbf{d}$ for some $s > 0$, within a small tolerance.
- $g_i \geq 0 \ \forall \ i \in \{1,2,3\}$
- $\| \mathbf{g} \|_2 = 1$

This `RegionHandler` type is implemented in `core.point_source.Triplet`.

6.1.2.2 VirtualNgon

This represents a region formed by n real loudspeakers, which is split into triangles with the addition of a single virtual loudspeaker. Each triangle is made from two adjacent real loudspeakers and the virtual loudspeaker, which is downmixed to the real loudspeakers by the provided downmix coefficients.

For example, if four real loudspeaker positions $\{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4\}$ and one virtual loudspeaker position $\mathbf{p}_v$ are used, the following triangles would be created:

- $\{\mathbf{p}_v, \mathbf{p}_1, \mathbf{p}_2\}$
- $\{\mathbf{p}_v, \mathbf{p}_2, \mathbf{p}_3\}$
- $\{\mathbf{p}_v, \mathbf{p}_3, \mathbf{p}_4\}$
- $\{\mathbf{p}_v, \mathbf{p}_4, \mathbf{p}_1\}$

When this `RegionHandler` type is queried with a position, each triangle is tried in turn until one returns valid gains, in the same way as the top level point source panner. This produces a vector of n gains for the real loudspeakers, $\mathbf{g} = \{g_1, \ldots, g_n\}$, and the gain for the virtual speaker g_v, which is downmixed to the real loudspeakers by the provided downmix coefficients $\mathbf{w}_{\text{dmx}}$:

$$\mathbf{g}' = \mathbf{g} + \mathbf{W}_{\text{dmx}} \quad g_v$$

Finally, this is power normalised, resulting in the final gains:

$$\mathbf{g}'' = \frac{\mathbf{g}'}{\parallel \mathbf{g}' \parallel_2}$$

This `RegionHandler` type is implemented in `core.point_source.VirtualNgon`.

6.1.2.3 QuadRegion

This represents a spherical quadrilateral region formed by 4 loudspeakers.

The gains are calculated for each loudspeaker by first splitting the position into two components, x and y. x could be considered as the horizontal position within the quadrilateral, being 0 at the left edge and 1 at the right edge, and y the vertical position, being 0 at the bottom edge and 1 at the top edge.

The x and y values are trivially mapped to a gain for each loudspeaker using equations 1 and 2. The x and y value (and therefore the loudspeaker gains) that result in a given velocity vector can be determined by solving equations 1 - 3.

The solution to this problem is of similar complexity to VBAP, and results in the same gain as VBAP at the edges of the quadrilateral, making it possible to use with other `RegionHandler` types in a single point source panner under the rules in § 6.1.1.

The resulting gains are infinitely differentiable with respect to the position within the region, producing results comparable to pair-wise panning between virtual loudspeakers in common situations.

This `RegionHandler` type is implemented in `core.point_source.QuadRegion`.

Formulation

Given the Cartesian position of 4 loudspeakers, $\mathbf{P} = [\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4]$ in anticlockwise order from the perspective of the listener, the gain vector $\mathbf{g}$ is computed as for a source direction $\mathbf{d}$ as:

$$\mathbf{g}' = [(1-x)(1-y), x(1-y), xy, (1-x)y] \qquad (1)$$

$$\mathbf{g} = \frac{\mathbf{g}'}{\parallel \mathbf{g}' \parallel_2} \qquad (2)$$

Where x and y are chosen such that the velocity vector $\mathbf{g} \cdot \mathbf{P}$ has the desired direction $\mathbf{d}$. The magnitude of the velocity vector r is irrelevant, as the gains are power normalised:

$$\mathbf{g} \cdot \mathbf{P} = r\mathbf{d} \qquad (3)$$

for some $r > 0$.

Solution

Given an x value, all velocity vectors $\mathbf{d}$ with this x value are on a plane formed by the origin of the coordinate system and two points some distance along the top and bottom of the quadrilateral:

$$(1-x)\mathbf{p}_1 + x\mathbf{p}_2$$

$$(1-x)\mathbf{p}_4 + x\mathbf{p}_3$$

Therefore:

$$(((1-x)\mathbf{p}_1 + x\mathbf{p}_2) \times ((1-x)\mathbf{p}_4 + x\mathbf{p}_3)) \cdot \mathbf{d} = 0 \qquad (4)$$

This equation can be solved to find x for a given source direction $\mathbf{d}$.

Collect the x terms:

$$[(\mathbf{p}_1 + x(\mathbf{p}_2 - \mathbf{p}_1)) \times (\mathbf{p}_4 + x(\mathbf{p}_3 - \mathbf{p}_4))] \cdot \mathbf{d} = 0$$

Expand the cross product and collect the terms:

$$\begin{aligned}
&[(\mathbf{p}_1 \times \mathbf{p}_4) \\
&+x\ ((\mathbf{p}_1 \times (\mathbf{p}_3 - \mathbf{p}_4)) + ((\mathbf{p}_2 - \mathbf{p}_1) \times \mathbf{p}_4)) \\
&+x^2\ ((\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_4)) \\
&] \cdot \mathbf{d} = 0
\end{aligned}$$

Finally, multiply through $\mathbf{D}$:

$$\begin{aligned}
& [(\mathbf{p}_1 \times \mathbf{p}_4) \cdot \mathbf{d}] \\
+x \quad & [((\mathbf{p}_1 \times (\mathbf{p}_3 - \mathbf{p}_4)) + ((\mathbf{p}_2 - \mathbf{p}_1) \times \mathbf{p}_4)) \cdot \mathbf{d}] \\
+x^2 \quad & [((\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_4)) \cdot \mathbf{d}] \\
& = 0
\end{aligned}$$

The solution for x is therefore the root of a polynomial, which can be solved using standard methods.

By replacing $\mathbf{P}$ by $\mathbf{P}'$ in the above equations, y can be determined too:

$$\mathbf{P}' = [\mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4, \mathbf{p}_1]$$

The gains $\mathbf{g}$ can then be calculated using equations 1 and 2. Since the scale of $\mathbf{d}$ is ignored in equation 4, solutions may be found that produce a velocity vector that is directly opposite to that which was desired. This can be checked by testing that:

$$\mathbf{gP} \cdot \mathbf{d} > 0$$

6.1.2.4 StereoPanDownmix

Stereo output is extremely restricted compared to surround output formats, so a separate method is implemented. This is based on a modified downmix from 0+5+0 to 0+2+0.

The procedure is as follows:

- The input direction is panned using a point source panner configured for 0+5+0 to produce a vector of 5 gains, $\mathbf{g}'$, in the order M+030, M-030, M+000, M+110, M-110.
- A downmix matrix is applied to produce stereo gains $\mathbf{G}''$ in the order M+030, M-030:

$$\mathbf{g}'' = \begin{bmatrix} 1 & 0 & \frac{\sqrt{3}}{3} & \sqrt{\frac{1}{2}} & 0 \\ 0 & 1 & \frac{\sqrt{3}}{3} & 0 & \sqrt{\frac{1}{2}} \end{bmatrix} \cdot \mathbf{g}'$$

  This downmix matrix is derived from the matrix in [BS.775], modified so that the downmix coefficients for M+000 preserve the velocity vector rather than total power.

- Power normalise $\mathbf{g}''$ to a value determined by the balance between the front and rear loudspeakers in $\mathbf{g}'$, such that sources between M+030 and M-030 are not attenuated, while sources between M-110 and M+110 are attenuated by 3 dB.

$$\begin{aligned} a_{\text{front}} &= \max\{g'_1, g'_2, g'_3\} \\ a_{\text{rear}} &= \max\{g'_4, g'_5\} \\ r &= \frac{a_{\text{rear}}}{a_{\text{front}} + a_{\text{rear}}} \\ \mathbf{g} &= \mathbf{g}'' \frac{\frac{1^{\frac{r}{2}}}{2}}{\| \mathbf{g}'' \|_2} \end{aligned}$$

This `RegionHandler` type is implemented in `core.point_source.StereoPanDownmix`.

6.1.3 Configuration Process

The configuration process builds a point source panner containing the above `RegionHandler` types for a given layout. The configuration process takes a `Layout` object (defined in § A.1.1.3), and produces a `PointSourcePanner`.

The configuration process initially selects the behaviour by the `Layout::name` attribute. If the `Layout::name` attribute is `0+2+0` the configuration is handled by the special configuration function for stereo described in § 6.1.3.2. All other cases are handled by a generic function described in § 6.1.3.1.

The configuration process is handled in `core.point_source.configure`.

6.1.3.1 Process for Generic Layouts

To configure a `PointSourcePanner` for generic speaker layouts, the following process is used:

1. Determine the set of remapped virtual loudspeakers as described below. These loudspeakers are added to the set of loudspeakers in the layout, to be treated the same as real loudspeakers.
2. Create two lists of normalised Cartesian loudspeaker positions, which will be used in the next steps; one containing the nominal loudspeaker positions (to triangulate the loudspeaker layout), and one containing the real loudspeaker positions (to use when creating the regions). Nominal loudspeaker positions are the positions specified in [BS.2051], whereas the real loudspeaker positions are positions which are actually used by the current reproduction system.
3. To each list of loudspeaker positions, append one or two virtual loudspeakers, which will become the virtual loudspeaker at the centre of a `VirtualNgon`:
   - $\{0,0,-1\}$ (below the listener) is always added, as no loudspeaker layouts defined in [BS.2051] have a loudspeaker in this position.
   - $\{0,0,1\}$ (above the listener) is added if there is no loudspeaker in the layout with the label `T+000` or `UH+180`. The reason this loudspeaker is not used when `UH+180` exists, is when this is used in the `3+7+0` layout in [BS.2051], the position may coincide with that of the virtual loudspeaker, creating a step change in the panning function.
4. Take the convex hull of the nominal loudspeaker positions. If this algorithm is implemented with floating point arithmetic, errors may cause some facets of the convex hull to be split — facets are merged within a tolerance set such that the result is the same as if the algorithm was implemented with exact arithmetic.
5. Create a `PointSourcePannerDownmix` with the following regions:
   - For each facet of the convex hull which doesn’t contain one of the virtual loudspeakers added in step 3:
     - If the facet has three edges, create a `Triplet` with the real positions of the loudspeakers corresponding to the vertices of the facet.
     - If the facet has four edges, create a `QuadRegion` with the real positions of the loudspeakers corresponding to the vertices of the facet.
   - For each virtual loudspeaker added in step 3, create a `VirtualNgon` with the real positions of the adjacent loudspeakers (all loudspeakers which share a convex hull facet with the virtual loudspeaker) at the edge, the position of the virtual loudspeaker at the centre, and all downmix coefficients set to $\frac{1}{\sqrt{n}}$, where n is the number of adjacent loudspeakers.
   - Note that no layouts in [BS.2051] result in facets with more than 4 edges.

   The downmix coefficients map the virtual loudspeakers to the physical loudspeakers, as described below.

This is implemented in `core.point_source._configure_full`.

Determination of Virtual Loudspeakers with Direct Downmix

For each mid-layer loudspeaker, a virtual loudspeaker is added on the upper and lower layers at the same azimuth as the real loudspeaker if there are no real loudspeakers in the upper or lower layer in that area. These virtual loudspeakers have downmix coefficients that map their output directly to the corresponding mid-level loudspeaker.

As with real loudspeakers, virtual loudspeakers have both a real and a nominal position, the real position being derived from the real positions of the real loudspeakers, and the nominal position being derived from the nominal positions of the real loudspeakers. The inclusion or not of a virtual loudspeaker is based on the nominal positions of the real loudspeakers, so that for a given layout the same set of virtual speakers is always used.

To determine the set of virtual loudspeakers for a given layout, the following procedure is used:

- For each $i \in [1, N]$, where $N = \texttt{len(layouts.channels)}$, the number of channels, define:

$$\begin{aligned}\varphi_{i,r} &= \texttt{layouts.channels[i].polar\_position.azimuth}\\ \varphi_{i,n} &= \texttt{layouts.channels[i].polar\_nominal\_position.azimuth}\\ \theta_{i,r} &= \texttt{layouts.channels[i].polar\_position.elevation}\\ \theta_{i,n} &= \texttt{layouts.channels[i].polar\_nominal\_position.elevation}\end{aligned}$$

- Define three sets of channel indices, identifying channels on the upper, middle and lower layers of the layout:

$$\begin{aligned}S_u &= \{i \mid 30° \le \theta_{i,n} \le 70°\}\\ S_m &= \{i \mid -10° \le \theta_{i,n} \le 10°\}\\ S_l &= \{i \mid -70° \le \theta_{i,n} \le -30°\}\end{aligned}$$

- Virtual loudspeakers have the same nominal and real azimuths as the corresponding real loudspeaker. The real elevation is the mean elevation of the real loudspeakers in the layer if there are any, or $-30°$ or $30°$ for the lower and upper layers otherwise. The nominal elevation is always $-30°$ or $30°$ for the lower and upper layers.

  Define two nominal elevations:

$$\theta'_{u,n} = 30°$$

$$\theta'_{l,n} = -30°$$

  Define two real elevations:

$$\theta'_{u,r} = \begin{cases} 30° & |S_u| = 0 \\ \dfrac{\sum_{j \in S_u} \varphi_{j,r}}{|S_u|} & \text{otherwise} \end{cases}$$

$$\theta'_{l,r} = \begin{cases} 30° & |S_u| = 0 \\ \dfrac{\sum_{j \in S_l} \varphi_{j,r}}{|S_l|} & \text{otherwise} \end{cases}$$

- Loudspeakers are only created on a layer if the absolute nominal azimuth of the corresponding mid-layer loudspeaker is greater or equal to the maximum absolute nominal azimuth of the real loudspeakers on the layer, plus 40°. These azimuth limits are defined as:

$$L_u = \begin{cases} 0 & |S_u| = 0 \\ \max\limits_{j \in S_u} |\varphi_{j,n}| + 40° & \text{otherwise} \end{cases}$$

$$L_l = \begin{cases} 0 & |S_l| = 0 \\ \max\limits_{j \in S_l} |\varphi_{j,n}| + 40° & \text{otherwise} \end{cases}$$

- For each j in S_m:
  - Create a virtual upper loudspeaker if $\varphi_{j,n} \ge L_u$, identified by a `Channel` struct `channel`, with:

$$\begin{aligned}\texttt{channel.polar\_position.azimuth} &= \varphi_{j,r}\\ \texttt{channel.polar\_position.elevation} &= \theta'_{u,r}\\ \texttt{channel.polar\_nominal\_position.azimuth} &= \varphi_{j,n}\\ \texttt{channel.polar\_nominal\_position.elevation} &= \theta'_{u,n}\end{aligned}$$

- Create a virtual lower loudspeaker if $\varphi_{j,n} \geq L_l$, identified by a `Channel` struct `channel`, with:

$$\begin{aligned}\texttt{channel.polar\_position.azimuth} &= \varphi_{j,r}\\ \texttt{channel.polar\_position.elevation} &= \theta'_{l,r}\\ \texttt{channel.polar\_nominal\_position.azimuth} &= \varphi_{j,n}\\ \texttt{channel.polar\_nominal\_position.elevation} &= \theta'_{l,n}\end{aligned}$$

Both have downmix coefficients routing the gains from this loudspeaker to the corresponding mid-layer loudspeaker j.

This is implemented in `core.point_source.extra_pos_vertical_nominal`.

6.1.3.2 Process for 0+2+0

For 0+2+0, a `PointSourcePanner` with a single `StereoPanDownmix` region is returned.

This is implemented in `core.point_source._configure_stereo`.

6.2 *Determination if angle is inside a range with tolerance*

A `inside_angle_range` function is used when comparing angles to given angular ranges, allowing ranges to be specified which include the rear of the coordinate system. This is used in the zone exclusion and direct speaker components in § 7.3.8.1 and § 8.4.

The signature is:

```
bool inside_angle_range(float x, float start, float end, float tol=0.0);
```

This returns true if an angle `x` is within the circular arc which starts at `start` and moves anticlockwise until `end`, expanded by `tol`. All angles are given in degrees.

In the common case where:

$$-180 \leq \mathrm{start} \leq \mathrm{end} \leq 180$$

This function is equivalent to:

$$\mathrm{start} - \mathrm{tol} \leq \mathrm{x}' \leq \mathrm{end} + \mathrm{tol}$$

Where $\mathrm{x}' = \mathrm{x} + 360 \times i$ for some i such that $-180 < \mathrm{x}' \leq 180$.

In other cases, the behaviour is more subtle. For example, if $\mathrm{start} = 90$ and $\mathrm{end} = -90$, this specifies the rear half of the coordinate system:

$$\mathrm{x}' \leq -90 \lor \mathrm{x}' \geq 90$$

Some example ranges and equivalent expressions are shown in Table 1.

Table 1: Expressions equivalent to `inside_angle_range(x, start, end, tol)`

Start	End	tol	Equivalent Expression
-90	90	0	$-90 \le x' \le 90$
-90	90	5	$-95 \le x' \le 95$
90	-90	0	$x' \le -90 \lor x' \ge 90$
90	-90	5	$x' \le -85 \lor x' \ge 85$
0	0	0	$x' = 0$
180	180	0	$x' = 180$
-180	-180	0	$x' = 180$
180	180	5	$x' \le -175 \lor x' \ge 175$
-180	180	0	true

This function is implemented in `core.geom.inside_angle_range`.

6.3 Determine if a channel is an LFE channel from its frequency metadata

Frequency metadata, which may be present as *frequency* sub-elements of *audioChannelFormats*, can be used to determine if a channel is effectively an LFE channel.

The following data structure is used to represent frequency metadata:

```
struct Frequency {
  optional<float> lowPass;
  optional<float> highPass;
};
```

The function with the signature `bool is_lfe(Frequency frequency)` evaluates

$$\texttt{frequency.lowPass} \land \neg\texttt{frequency.highPass} \land (\texttt{frequency.lowPass} \le 200\,\mathrm{Hz})$$

and returns `True` if the channel is assumed to be an LFE channel and `False` otherwise.

This is implemented in `core.renderer_common.is_lfe`.

6.4 Block Processing Channel

When rendering timed ADM metadata, some functionality is required that is the same for all *typeDefinition* values – for a given subset of the input channels, some processing is applied between time bounds, producing loudspeaker channels on the output.

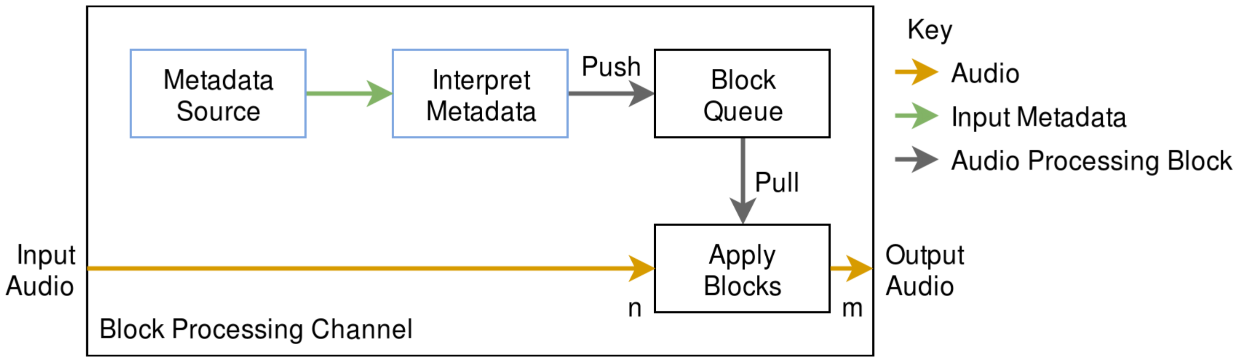


Figure 4: Structure used to process related channels. Components in blue are provided externally.

Figure 4 shows the structure used to achieve this. The interface to this component is as follows:

```
class BlockProcessingChannel {
    BlockProcessingChannel(MetadataSource metadata_source, Callable
interpret_metadata);
    void process(int sample_rate, int start_sample,
        ndarray<float> input_samples, ndarray<float> &output_samples);
};
```

The `MetadataSource` is provided by the system as the mechanism for feeding metadata into the renderer. It has the following interface:

```
class MetadataSource {
    optional<TypeMetadata> get_next_block();
};
```

By repeatedly calling `get_next_block`, the block processing channel receives a sequence of `TypeMetadata` blocks as described in § 5, which correspond to time-bounded blocks of metadata required during rendering.

These metadata blocks are interpreted by the `interpret_metadata` function, which is provided by the renderer for each *typeDefintion*. These functions accept a `TypeMetadata` and return a list of `ProcessingBlock` objects, which encapsulate the time-bounded audio processing required to implement the given `TypeMetadata`. The interpretation for *typeDefinition==Objects* is described in detail in § 7.2. For *typeDefinition==HOA* and *typeDefinition==DirectSpeakers*, a single `ProcessingBlock` is returned.

`ProcessingBlock` objects have the following external interface:

```
class ProcessingBlock {
    Fraction start_sample, end_sample;
    int first_sample, last_sample;

    void process(int in_out_samples_start,
        ndarray<float> input_samples, ndarray<float> &output_samples);
}
```

The samples passed to `process` are assumed to be a subset of the samples in the input/output file, such that $\mathtt{input\_samples}[i]$ and $\mathtt{output\_samples}[i]$ represent the global input and output samples $\mathtt{in\_out\_samples\_start} + i$. The `first_sample` and `last_sample` attributes define the range of global sample numbers s which would be affected by `process`:

$$\mathtt{first\_sample} \leq s \leq \mathtt{last\_sample}$$

`start_sample` and `end_sample` are the fractional start and end sample numbers, which are used to determine the `first_sample` and `last_sample` attributes, and may be used by `ProcessingBlock` subclass implementations.

`BlockProcessingChannel` objects store a queue of `ProcessingBlock`, which is refilled by requesting blocks from the `metadata_source` and passing them through `interpret_metadata`. `BlockProcessingChannel.process` applies processing blocks in this queue to the samples passed to it, using `first_sample` and `last_sample` to determine when to move to the next block.

This structure allows components of the renderer to be decoupled; audio samples may be processed in chunks sizes independent of the metadata block sizes, while retaining sample-accurate metadata processing, and without complicating the renderers with concrete timing concerns.

The decision to allow the renderer to pull metadata blocks in keeps the interpretation of timing metadata within the renderer — if metadata was instead pushed into the renderer, the component doing the pushing would have to know when the next block is required, which depends on the timing information within it.

This functionality is implemented in `core.renderer_common`.

6.4.1 Implemented `ProcessingBlock` Types

Three common processing block types are:

`FixedGains` takes a single input channel and applies n gains, summing the output into n output channels.

`FixedMatrix` takes N input channels and applies a NxM gain matrix to form M output channels.

`InterpGains` takes a single input channel and applies n linearly interpolated gains, summing the output into n output channels. Two gain vectors `gains_start` and `gains_end` are provided, which are the gains to be applied at times `start_sample` and `end_sample`. The gain $g(i, s)$ applied to channel i at sample s is given by:

$$p(s) = \frac{s - \texttt{start\_sample}}{\texttt{end\_sample} - \texttt{start\_sample}}$$

$$g(i, s) = (1 - p(s)) \times \texttt{gains\_start}[i] + p(s) \times \texttt{gains\_end}[i]$$

6.5 Generic Interpretation of Timing Metadata

The determination of block start and end times is shared between renderers for different *typeDefinitions*. For a `TypeMetadata` object `block`, the following process is used:

- The start and end time of the object which contains the block is determined from `block.extra_data.object_start` and `block.extra_data.object_duration`. If `object_start` is `None`, the object is assumed to start at time `0`. If `object_duration` is `None`, it is assumed to extend to infinity.
- The block start and end times are determined from the `rtime` and `duration` attributes:
    - If `rtime` and `duration` are not `None`, then the block start time is assumed to be the object start time plus `rtime`, and the block end time is assumed to be the block start time plus `duration`.
    - If `rtime` and `duration` are `None`, then the block is assumed to extend from the object start time to the object end time.
    - Other `rtime` and `duration` constellations are considered to be an **error**. — for multiple *audioBlockFormat* objects within an *audioChannelFormat*, both *rtime* and *duration* should be provided, while for a single block covering the entire *audioObject*, no *rtime* or *duration* should be provided. Otherwise, the behaviour is undefined.

The times should be checked for consistency. Blocks ending after the object end time or overlapping blocks in a sequence are not allowed and considered to be an **error** An **error condition** means that implementers must consider that something is wrong with the input data. The correct course of action is to fix the system that produced it. In the reference implementation, errors are handled by stopping the rendering process end reporting the error to the user. Other implementations might use different error handling strategies based on their target application environment.

This is implemented in `core.renderer_common.InterpretTimingMetadata`.

7. Render Items with typeDefinition==Objects

7.1 Structure

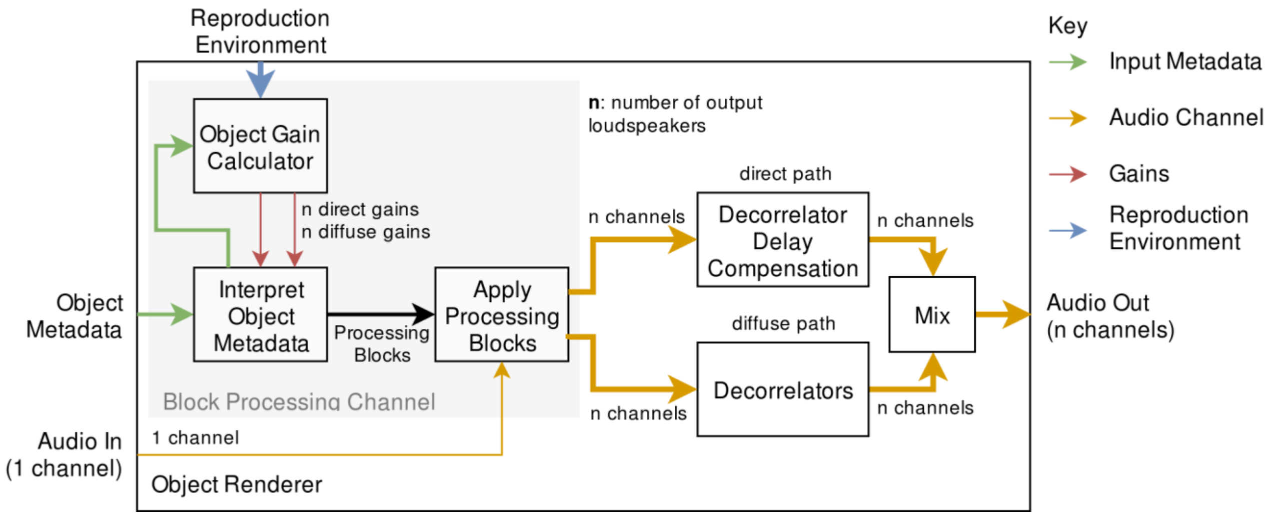


Figure 5: Structure of the Objects renderer

The structure of the renderer for *typeDefinition==Objects* is shown in Figure 5. This diagram shows the processing applied for a single rendering item; rendering multiple items behaves as if this structure is duplicated for each item, with the outputs mixed together.

Metadata enters the renderer in the form of an `ObjectRenderingItem` object, which contains a track index, and a source of `ObjectTypeMetadata` objects representing time-bounded rendering parameters for the identified track.

For each `ObjectTypeMetadata` object, the method described in § 7.2 is applied; this interprets the timing metadata, and calculates gain vectors using the gain calculator described in § 7.3. This produces `ProcessingBlock` objects, which apply time-bounded signal processing operations to the input audio to produce a direct and diffuse bus, each containing one channel per loudspeaker. This approach, and the `BlockProcessingChannel` class which encapsulates it is described in § 6.4.

The diffuse bus is passed through a per-channel decorrelation filter bank, and the direct bus is delayed to match, before being mixed together to form the output. The decorrelation filters and delays are described in § 7.4.

This structure is implemented in `core.objectbased.renderer.ObjectRenderer`.

7.2 InterpretObjectMetadata

Object timing metadata is interpreted in the `InterpretObjectMetadata` class, which fits into the block processing channel structure.

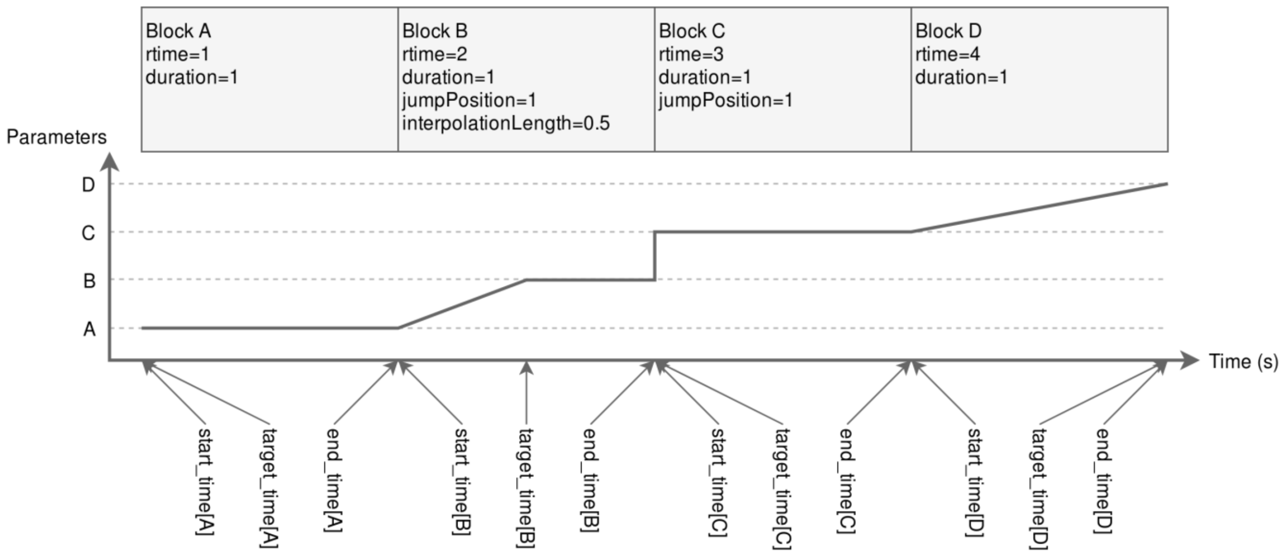


Figure 6: Example audioBlockFormats and the interpreted interpolation curves

For each input `ObjectTypeMetadata`, the following process is used:

- The start and end time `start_time` and `end_time` of the block are determined according to § 6.5.
- The time at which interpolation in this block ends, `target_time` is determined according to the following cases, which are illustrated by corresponding blocks in Figure 6:

A

If this is the first block, or if the `end_time` of the previous block is less than the `start_time` of the current block, then:

$$\texttt{target\_time} = \texttt{start\_time}$$

B

If `bf.jumpPosition.flag` is set, and `bf.jumpPosition.interpolationLength` is not `None`, then:

$$\texttt{target\_time} = \texttt{start\_time} + \texttt{bf.jumpPosition.interpolationLength}$$

C

If `bf.jumpPosition.flag` is set, and `bf.jumpPosition.interpolationLength` is `None`, then:

$$\texttt{target\_time} = \texttt{start\_time}$$

D

If `bf.jumpPosition.flag` is not set, then interpolation occurs over the whole block:

$$\texttt{target\_time} = \texttt{end\_time}$$

- Gain vector `interp_to` is calculated using a `GainCalculator` instance for the current block. `interp_from` is the gain vector calculated for the previous block.
- If $\texttt{start\_time} < \texttt{target\_time}$, an `InterpGains` `ProcessingBlock` is created, which

interpolates from `interp_from` to `interp_to` between `start_time` and `target_time`.

- If `target_time` < `end_time`, an `FixedGains ProcessingBlock` is created, which applies `interp_to` between `start_time` and `target_time`.

This is implemented in `core.objectbased.renderer.InterpretObjectMetadata`.

7.3 Gain Calculator

Given an `ObjectTypeMetadata` object, this object calculates a gain for each loudspeaker on the direct and diffuse paths. The interface to this component is:

```
struct DirectDiffuseGains {
    vector<float> direct;
    vector<float> diffuse;
};

class GainCalc {
    GainCalc(Layout layout);

    DirectDiffuseGains render(ObjectTypeMetadata otm);
};
```

7.3.1 Structure

This component is primarily a composite of the sub-components listed in this section. A diagram of the signal flow between these components for non-LFE content is shown in Figure 7. The behaviour for an `ObjectTypeMetadata` `otm` containing a `block_format` attribute `bf` is as follows:

- The coordinate transform described in § 7.3.2 is applied to bf.position and bf.cartesian to yield a CartPosition object position.
- Screen scaling is applied using the method described in § 7.3.3, with the parameters `position`, `bf.screenRef`, `otm.extra_data.reference_screen`, updating `position`. This component is initialised with `layout.screen`.
- Screen edge lock is applied using the method described in § 7.3.4, with the parameters `position` and `bf.position.screenEdgeLock`, updating the `position` with the result. This component is initialised with the reproduction screen (`layout.screen`).
- Channel lock is applied using the method described in § 7.3.5, with the parameters `position` and `bf.channelLock`, again updating the `position`. This component is initialised with reproduction speaker layout excluding the LFE channels.
- Divergence is applied using the method described in § 7.3.6, with the parameters `position`, `bf.objectDivergence` and `bf.cartesian`. This results in up to 3 extended sources with gains and positions stored in `diverged_gains` and `diverged_positions`.
- The extent panner described in § 7.3.7 is applied to each `p` in `diverged_positions`, with parameters `p`, `bf.width`, `bf.height`, `bf.depth` and `bf.cartesian`. This results in a per-loudspeaker gain vector stored in `gains_for_each_pos`.
- The gains in `gains_for_each_pos` are mixed together with a power determined by `diverged_gains`:

$$\mathtt{gains}[i] = \sqrt{\sum_j \mathtt{diverged\_gains}\,[j] \times \mathtt{gains\_for\_each\_pos}[j,i]^2}$$

- Zone exclusion as described in § 7.3.8 is applied to `gains` and `bf.zoneExclusion`, resulting in a new `gains` vector. This component is initialised with `layout.without_lfe`.

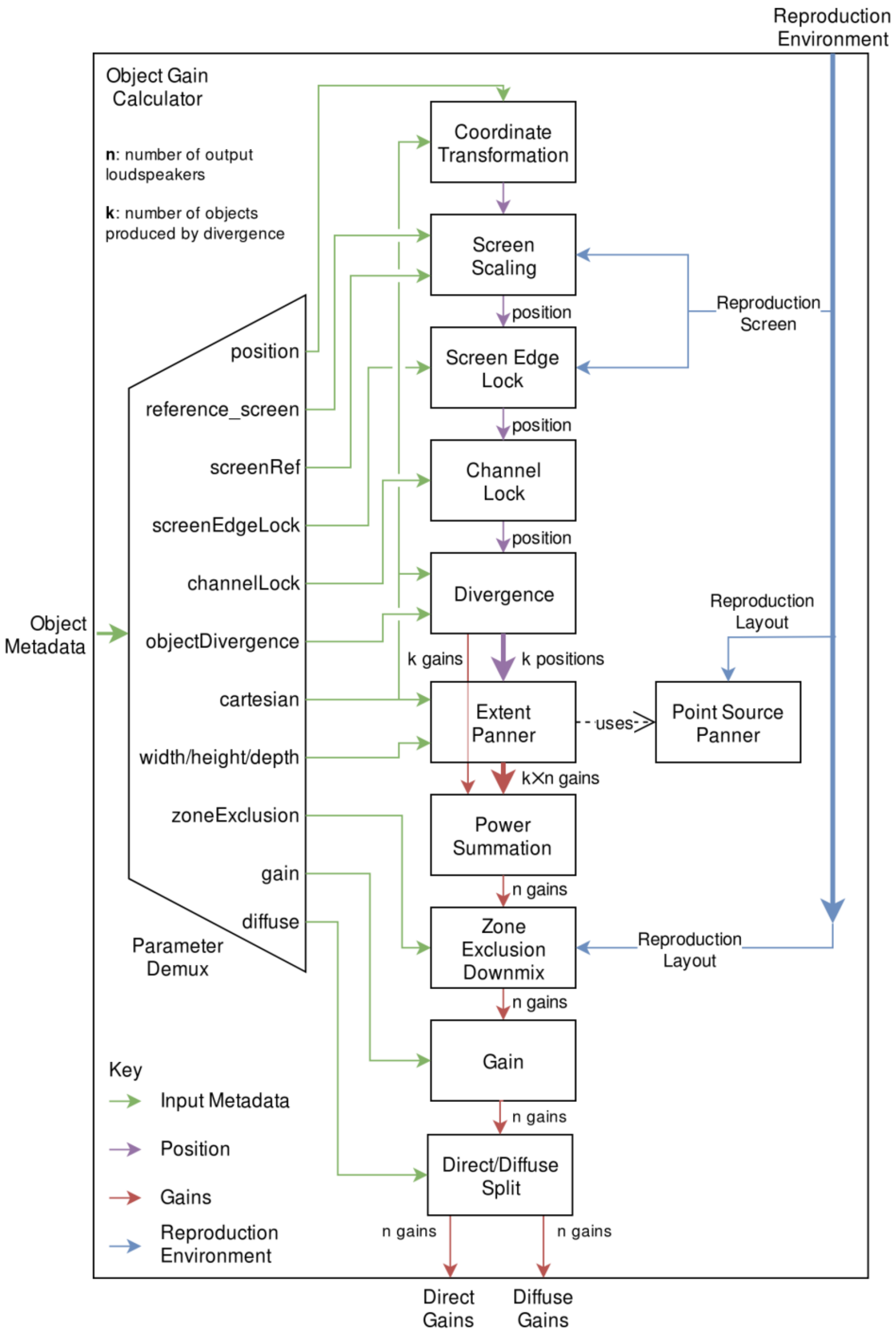


Figure 7: Structure of the gain calculator for typeDefintion==Objects

- If `otm` is an LFE element according to `is_lfe(otm.extra_data.channel_frequency)` as defined in § 6.3, then:
  - An LFE downmix matrix calculated as described in § 7.5 is applied to `gains`, to distribute the loudspeaker gains between the LFE channels, yielding `gains_full`.
  - `gains_full` amplitude normalised by dividing by `sum(gains_full)`.
  - `gains_full` is multiplied by `bf.gain`.

  - A `DirectDiffuseGains` is returned with `direct` set to `gains_full` and `diffuse` set to zeros.
- Otherwise:
  - `gains` is multiplied by `bf.gain`.
  - `gains` is extended by LFE channel gains with value `0` to produce `gains_full` as required by the reproduction layout.
  - `gains_full` is split into a direct and diffuse vector to control the direct and diffuse paths, depending on the `bf.diffuse` parameter. These are returned as a `DirectDiffuseGains` with attributes:

$$\begin{aligned}\texttt{direct} &= \texttt{gains\_full} \times \sqrt{1 - \texttt{bf.diffuse}} \\ \texttt{diffuse} &= \texttt{gains\_full} \times \sqrt{\texttt{bf.diffuse}}\end{aligned}$$

7.3.1.1 Discussion

The structure of the gain calculator is influenced by the following two principles:

- If the parameters are sparse (i.e. only a small number of the possible metadata fields are used), the obvious interpretation of those parameters should be preserved.
- When combinations of parameters are used together, the option that gives the user the most possibilities for different useful behaviours is chosen.

For example:

- Channel lock is implemented as a position modification — if channel lock is used by itself (with appropriate *maxDistance*) then the source will be locked to a channel because of the behaviour of the point source panner, however channel lock can also be used with extent parameters to produce an extended source centred around a particular loudspeaker, for example.
- Diffuseness is not linked to extent — a fully extended diffuse source can be obtained by setting the extent parameters appropriately, but this also allows for use of the decorrelation filtering with less-than-full extents.

7.3.2 Coordinate Transformation

A coordinate transform is implemented in `core.objectbased.gain_calc.coord_trans`, which is used to convert incoming positions into a uniform Cartesian spherical coordinate. It has the following signature:

```
CartesianPosition coord_trans(ObjectPosition position, bool cartesian);
```

`position` is first converted to a Cartesian vector.

If `cartesian` is `true`, the position is warped by `cube_to_sphere`, regardless of whether the position was originally specified using polar or Cartesian coordinates. This function is described below.

7.3.2.1 Cubic and Spherical Spaces

Independent of polar or Cartesian vector representations, two spaces of coordinates are considered in the renderer:

Cubic

Coordinates which are on the surface of a 2-unit cube centred at the origin are considered to be on the convex hull of the loudspeakers, and therefore have no distance effects applied.

Spherical

Coordinates which are on the surface of a unit sphere centred at the origin are considered to be on the convex hull of the loudspeakers, and therefore have no distance effects applied.

Input coordinates are considered to be in cubic space if the `cartesian` block format flag is set, or spherical otherwise.

Two functions `cube_to_sphere` and `sphere_to_cube`, defined in `core.objectbased.gain_calc` are used to convert between positions in these coordinate spaces, by scaling the length of coordinates and preserving the direction:

$$\begin{aligned}\texttt{cube\_to\_sphere}(\mathbf{p}) &= \mathbf{p}\cdot\begin{cases}0 & \|\mathbf{p}\|_2 = 0\\ \dfrac{\|\mathbf{p}\|_\infty}{\|\mathbf{p}\|_2} & \text{otherwise}\end{cases}\\ \texttt{sphere\_to\_cube}(\mathbf{p}) &= \mathbf{p}\cdot\begin{cases}0 & \|\mathbf{p}\|_2 = 0\\ \dfrac{\|\mathbf{p}\|_2}{\|\mathbf{p}\|_\infty} & \text{otherwise}\end{cases}\end{aligned}$$

Where $\|\mathbf{p}\|_\infty$ is the maximum norm:

$$\|\mathbf{p}\|_\infty = \max\{|\mathbf{p}_x|, |\mathbf{p}_y|, |\mathbf{p}_z|\}$$

7.3.3 Screen Scaling

The screen scaling component warps source positions in order to compensate for differences in screen geometry between the production and reproduction environments. The interface to this component is:

```
class ScreenScaleHandler {
  ScreenScaleHandler(Screen reproduction_screen);
  CartesianPosition handle(
    CartesianPosition position,
    bool screenRef,
    Screen reference_screen
  );
};
```

The two screen definitions used are:

Reference Screen

The `audioProgrammeReferenceScreen` listed in the `audioProgramme` element, or the default polar screen size if not provided. This was the screen geometry used during production of the metadata.

Reproduction Screen

Screen geometry in the reproduction environment in which the output of the renderer will be listened to.

Positions within the reference screen are warped so that they appear at corresponding positions in the reproduction screen.

7.3.3.1 Internal Screen Representation

Information about both screens can be provided in either polar or Cartesian coordinates (`PolarScreen` or `CartesianScreen` objects). Unlike object source positions, there is no obvious equivalence between the two, but in order to simplify the implementation a single screen representation is required which can represent both screen types. This is the purpose of the `PolarEdges` structure, which stores the azimuths of the left and right screen edges, and the elevations of the top and bottom screen edges:

```
struct PolarEdges {
    float left_azimuth;
    float right_azimuth;
    float bottom_elevation;
    float top_elevation;
};
```

7.3.3.2 Direction Warping

The warping of positions is defined in `core.screen_scale.PolarScreenScaler.scale_direction`. Given the reference screen `PolarEdges ref` and the reproduction screen `PolarEdges rep`, this works as follows:

- The azimuth, elevation and distance of the input position is determined.
- Piecewise linear interpolation is applied to the azimuth, mapping from the values

  $$\{-180, \mathtt{ref.right\_azimuth}, \mathtt{ref.left\_azimuth}, 180\}$$

  to

  $$\{-180, \mathtt{rep.right\_azimuth}, \mathtt{rep.left\_azimuth}, 180\}$$

- Piecewise linear interpolation is applied to the elevation, mapping from the values

  $$\{-90, \mathtt{ref.bottom\_elevation}, \mathtt{ref.top\_elevation}, 90\}$$

  to

  $$\{-90, \mathtt{rep.bottom\_elevation}, \mathtt{rep.top\_elevation}, 90\}$$

- The modified azimuth and elevation and the original distance are converted back to a Cartesian vector.

7.3.3.3 Metadata Interpretation

If `screenRef` is set and the reproduction screen is provided, the position is passed through `PolarScreenScaler.scale_direction` with the reference and reproduction screen set. Otherwise, the position is returned unmodified.

7.3.4 Screen Edge Lock

The screen edge lock component warps source positions in order to place the source on the indicated edge of the screen. It has the following interface:

```
class ScreenEdgeLockHandler {
  ScreenEdgeLockHandler(Screen reproduction_screen);

  CartesianPosition handle_vector(
    CartesianPosition position,
    ScreenEdgeLock screen_edge_lock
  );

  tuple<float, float> handle_az_el(
    float azimuth,
    float elevation,
    ScreenEdgeLock screen_edge_lock
  );
};
```

On initialisation, this component transforms the `reproduction_screen` into a `PolarEdges` object `polar_edges`, as specified in § 7.3.3.1.

At run time, the azimuth, elevation and distance components of `position` are modified independently, resulting in a new position:

- If `screen_edge_lock.horizontal` is `LEFT`, then the azimuth is set to `polar_edges.left_azimuth`; if it is `RIGHT`, then the azimuth is set to `polar_edges.right_azimuth`; otherwise the azimuth is unchanged.
- If `screen_edge_lock.vertical` is `TOP`, then the elevation is set to `polar_edges.top_elevation`; if it is `BOTTOM`, then the elevation is set to `polar_edges.bottom_elevation`; otherwise the elevation is unchanged.
- The distance is unchanged.

The processing takes place in the polar domain, hence Cartesian positions have to be converted first. The back and forth conversion is applied if the `handle_vector` method is used instead of the `handle_az_el` method.

This component is implemented in `core.screen_edge_lock.ScreenEdgeLockHandler`.

7.3.5 Channel Lock

Channel lock is implemented as a position transformation. If *channelLock* is set and a loudspeaker is within the range specified in *maxDistance*, the position will be transformed to the position of the loudspeaker closest to the original position. In the absence of divergence, extent, zone exclusion and diffuse metadata the source will be reproduced directly by the selected loudspeaker.

Channel lock is implemented in `core.objectbased.gain_calc.ChannelLockHandler` with the following signature:

```
class ChannelLockHandler {
  ChannelLockHandler(Layout layout);
  CartesianPosition handle(
    CartesianPosition position,
    optional<ChannelLock> channelLock
  );
};
```

To apply channel lock metadata, the following procedure is used:

- If `channelLock` is `None`, return the original `position`.
- Otherwise, the loudspeaker whose normalised position is closest to `position` by ℓ_2 distance is identified.
    - If there is no unique closest loudspeaker (within some tolerance), then the loudspeaker from the set of closest loudspeakers with the highest priority is chosen. Priority ordering

of loudspeakers is determined by lexicographic comparison of the tuple:

$$\{|\theta|, \theta, |\varphi|, \varphi\}$$

- ◦ Where φ and θ are the real azimuth and elevation of the loudspeaker. Lower tuples have higher priority — loudspeakers with lower absolute elevations have highest priority, with ties broken by the elevation, then the absolute azimuth, then the azimuth.

- If `channelLock.maxDistance` is `None` or the ℓ_2 distance of this loudspeaker is less than or equal to `channelLock.maxDistance`, the normalised position of the loudspeaker is returned. Otherwise, `position` is returned.

7.3.6 Divergence

Divergence is implemented by adding two additional source positions $\mathbf{p}_l$ and $\mathbf{p}_r$ to the left and the right of the original source position $\mathbf{p}_c$. To ensure the loudness of the item does not change, each source position is associated with a gain value g_l, g_c and g_r.

The Divergence metadata is interpreted in `core.objectbased.gain_calc.diverge`, with the following signature:

```
tuple<vector<float>, vector<CartesianPosition>> diverge(
  CartesianPosition position,
  ObjectDivergence objectDivergence,
  bool cartesian
);
```

This function accepts a 3D position (in this case, the output of the Channel Lock function) and applies the divergence metadata supplied in `objectDivergence`. Three source positions and associated gains are produced, each of which are passed to the extent panner for rendering.

The calculation of these gains and positions is described below.

7.3.6.1 Calculation of Gains

For a given `objectDivergence.value` x, the three gains are calculated as follows:

$$g_c = \frac{1 - x}{x + 1}$$

$$g_l = g_r = \frac{x}{x + 1}$$

This satisfies the following requirements:

- $\forall x,\ g_l + g_r + g_c = 1$
- $x = 0 \Longrightarrow g_l = g_r = 0 \wedge g_c = 1$
- $x = \frac{1}{2} \Longrightarrow g_l = g_r = g_c = \frac{1}{3}$
- $x = 1 \Longrightarrow g_l = g_r = 0.5 \wedge g_c = 0$

7.3.6.2 Calculation of Positions

The positions produced depend on the `cartesian` flag in the block format. A warning is raised if `azimuthRange` and `cartesian` are set, or if `positionRange` is set and `cartesian` is not.

Behaviour when `cartesian == true`

In Cartesian mode, the original cubic space position has been warped into spherical space by the coordinate transform in § 7.3.2. In order to apply divergence in this mode, the position is translated back to cubic space, diverged, and then translated back to spherical space again. This ensures that the diverged positions for a central position with $y = 1$ will appear like other sources with $y = 1$, without distance effects applied.

For a position value $\mathbf{p}$ and objectDivergence.positionRange value x, the three positions are calculated as:

$$\begin{aligned}\mathbf{p'}_c &= \mathtt{sphere\_to\_cube}(\mathbf{p}) \\ \mathbf{p'}_l &= \mathbf{p'}_c - \{x, 0{,}0\} \\ \mathbf{p'}_r &= \mathbf{p'}_c + \{x, 0{,}0\}\end{aligned}$$

These are then translated back to sphere-space:

$$\begin{aligned}\mathbf{p}_c &= \mathtt{cube\_to\_sphere}(\mathbf{p'}_c) \\ \mathbf{p}_l &= \mathtt{cube\_to\_sphere}(\mathbf{p'}_l) \\ \mathbf{p}_r &= \mathtt{cube\_to\_sphere}(\mathbf{p'}_r)\end{aligned}$$

Behaviour when `cartesian == false`

Positions are calculated for a given objectDivergence.azimuthRange a such that from the perspective of the listener the left and right sources are a degrees to the left and right of the centre, and all three sources are in a straight line.

This is achieved by defining three positions centred around the $+y$-axis at a distance $d = \| \mathbf{p}_c \|_2$, where $\mathbf{p}_c$ is the original source position:

$$P'_l = \mathtt{cart}(a, 0, d)$$

$$P'_r = \mathtt{cart}(-a, 0, d)$$

$$P'_c = \mathtt{cart}(0{,}0, d)$$

These are then rotated around the original source direction by the rotation matrix $\mathbf{M}$, which is defined such that $\mathbf{p}_c'$ is mapped onto the original source position $\mathbf{p}_c$:

$$[\mathbf{p}_l, \mathbf{p}_r, \mathbf{p}_c]^T = \mathbf{M} \cdot [\mathbf{p'}_l, \mathbf{p'}_r, \mathbf{p'}_c]^T$$

7.3.7 Extent Panner

The ADM extent parameters are handled in core.objectbased.gain_calc.ExtentHandler; this uses the modules described below to produce a gain vector for given position and extent parameters.

The interface to this class is:

```
class ExtentHandler {
  ExtentHandler(PointSourcePanner psp);

  vector<float> handle(
    CartesianPosition position,
    float width,
    float height,
    float depth,
    bool cartesian);
};
```

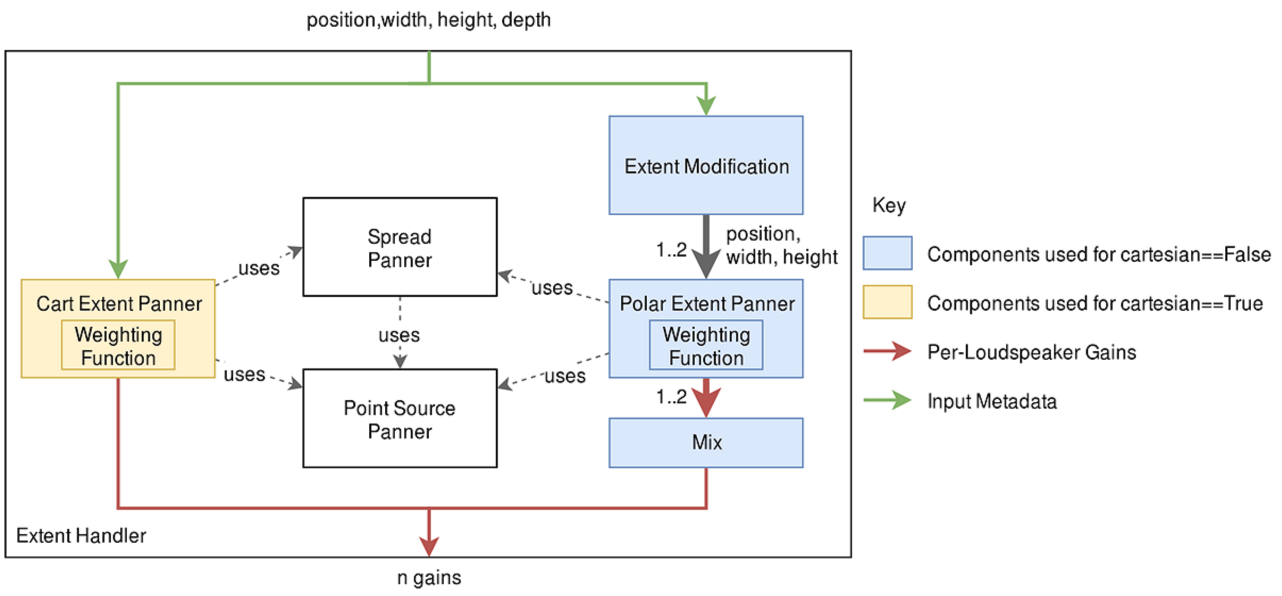


Figure 8: Structure of the Extent Handler

The structure of the `ExtentHandler` class is shown in Figure 8.

Internally, this object holds a reference to a `PolarExtentPanner` as described in § 7.3.7.2 and a `CartExtentPanner` as described in § 7.3.7.3 and uses these to calculate the gain vectors.

If the `cartesian` flag is set, the position and size are passed to the `CartExtentPanner` to generate the returned loudspeaker gain vector, as described in § 7.3.7.3.

If the `cartesian` flag is not set, then the `width`, `height` and `position` parameters are duplicated and modified to handle `depth` parameter and the distance component of `position`; these parameters are passed through the Polar Extent Panner in order to generate a loudspeaker gain vector for each, and finally these gain vectors are mixed together. This procedure is described in § 7.3.7.2.

Both Cartesian and polar extent rendering modes use the Spreading Panner to generate loudspeaker gains, as described below.

7.3.7.1 Spreading Panner

The shape of extended sources in the renderer is defined in terms of a weighting function, which given a 3D direction can calculate a weight for that direction. This weight can be thought of as the amount that a given object should be reproduced in a given direction. For example, for a source in front of the listener which is wider than it is tall, a weighting function like the one represented in Figure 10 may be used.

By producing per-loudspeaker gains which reflect this weighting function, applying these gains to the mono waveform of an object, and applying decorrelation filtering to the resulting channels, an impression of an extended or diffuse sound source with the intended extent parameters can be achieved.

To calculate a gain vector for a given weighting function, the `SpreadingPanner` class is used.

Objects of this type hold a set of virtual source positions, and a loudspeaker gain vector for each of these positions.

During start-up, the point source panner is used the calculate the gain vector for each position.

To calculate the gain vector for a given weighting function, the weighting function is applied to the virtual source positions. The resulting per-virtual-source gain vector is multiplied by the pre-calculated loudspeaker gain vectors to obtain a single per-loudspeaker gain vector. This is then power normalised to obtain the final gain vector.

This is implemented in `core.objectbased.extent.SpreadingPanner`.

7.3.7.2 Rendering Polar Extent

The procedure used to calculate loudspeaker gains for `position`, `width`, `height` and `depth` parameters in polar mode is as follows:

- The `depth` parameter is interpreted as two extended sources with the same direction but different distances. The two distances are:

$$\begin{aligned} d_1 &= \max\left\{0, \| \mathtt{position} \|_2 + \frac{\mathtt{depth}}{2}\right\} \\ d_2 &= \max\left\{0, \| \mathtt{position} \|_2 - \frac{\mathtt{depth}}{2}\right\} \end{aligned}$$

- For each distance, the Polar Extent Panner is used to calculate gain vectors $\mathbf{g}'_1$ and $\mathbf{g}'_2$ from the `position`, and the `width` and `height` modified by the Polar Extent Modification Function, described below.
- The gain vectors are mixed together to produce the output gain vector $\mathbf{g}$, where $\mathbf{g}_i$ is the gain for loudspeaker i:

$$\mathbf{g}_i = \sqrt{\frac{{\mathbf{g}'_{1,i}}^2 + {\mathbf{g}'_{2,i}}^2}{2}}$$

Polar Extent Modification Function

The extent modification function is used to modify the width and height parameters given the distance parameter.

It has the following properties:

- At $\mathtt{distance} = 0$, the extent is always $360°$.
- At $\mathtt{distance} = 1$, the original extent is used.
- At $\mathtt{distance} > 1$, the extent decreases as the distance increases.
- When $0 < \mathtt{distance} < 1$, the extent changes more steeply around $\mathtt{distance} = 0$ for smaller extents.

The extent modification function for `extent` and `distance` is defined as follows:

- The extent in degrees is mapped linearly to an extent along the x axis, with a minimum size of :

$$\mathtt{min\_size} = 0.2$$

$$\mathtt{size} = \mathtt{min\_size} + \frac{(1 - \mathtt{min\_size}) \times \mathtt{extent}}{360°}$$

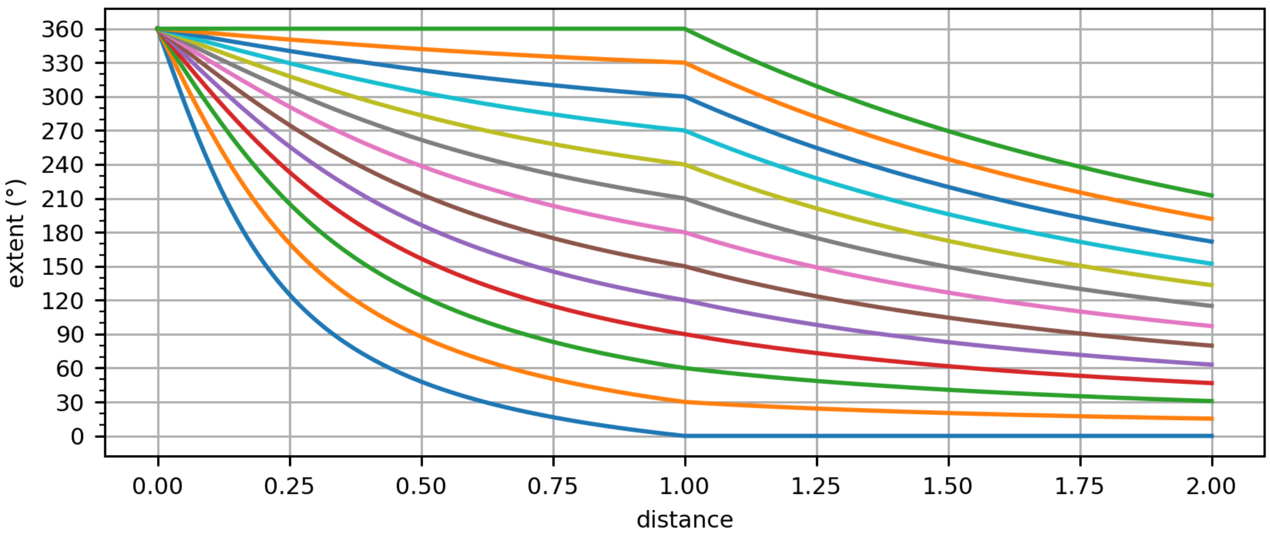
- A right triangle if formed, with the adjacent edge being the distance, and the opposite edge being the distance. The angle formed is then used to determine a new extent; this is calculated for a distance of 1 and $\mathtt{distance}$:

$$
\begin{aligned}
e_1 &= 4 \times \frac{180}{\pi} \times \mathrm{atan2}(\mathtt{size}, 1) \\
e_d &= 4 \times \frac{180}{\pi} \times \mathrm{atan2}(\mathtt{size}, \mathtt{distance})
\end{aligned}
$$

- Piecewise linear interpolation is applied to map e_d back to the original extent when $e_d = e_1$:

$$
\mathtt{extent\_mod} = \begin{cases} \mathtt{extent} \times \dfrac{e_d}{e_1} & e_d < e_1 \\ \mathtt{extent} + (360° - \mathtt{extent}) \times \dfrac{e_d - e_1}{360° - e_1} & e_d \geq e_1 \end{cases}
$$

This is implemented in `core.objectbased.gain_calc.ExtentHandler.extent_mod`. The shape of the extent modification function is shown in Figure 9.



Each line shows how the output extent varies over distance for a given input extent. The extent is not modified where distance=1, so for example the lowermost line shows how the modified extent varies over distance for an input extent of 0.

Figure 9: Extent modification function for polar extended sources

Polar Extent Panner

In order to handle the full range of positions and extents allowed in the ADM, the size must be modified before the Polar Weighting Function can be applied. The following steps are used:

- A modified width and height is computed as $\max\{\mathtt{width}, 5°\}$ and $\max\{\mathtt{height}, 5°\}$; these are used with the spreading panner described in § 7.3.7.1 and the Polar Weighting Function described below to yield a spread gain vector.
- The position is passed to the point source panner to yield a point source gain vector.
- The two vectors are mixed together so that for zero width and height, the point source gains are used exclusively, while if either the width or height is greater than $5°$, the spread gains are used exclusively.

This is required to support small extents — here the non-zero part of the spreading function must be large enough to cover multiple sampling points in order to produce smooth gains, and this imposes a minimum amount of spread, which may be larger than the desired amount.

This is implemented in `core.objectbased.extent.PolarExtentPanner.calc_pv_spread`.

Polar Weighting Function

The weighting function for polar extent rendering is parametrised by a 3D Cartesian vector `position`, and angles `width` and `height` in degrees. Since the distance component of the position is not used, this can be considered as a direction.

The weighting function is as follows:

- A rotation matrix is calculated which maps the position $\{0,1,0\}$ (directly in front of the listener) to the position of the source. This rotation matrix takes the form of a rotation around $\{1,0,0\}$ followed by a rotation around $\{0,0,1\}$. This is implemented in `core.objectbased.extent.calc_basis`.
- If the height is greater than the width, then the coordinate system is flipped to simplify the calculation, as the weighting function for a source with width w and height h should be the same as the weighting function for a source with width h and height w, rotated $90°$ around the source position. This is achieved by swapping the width and height variables, and swapping the x and z rows of the rotation matrix. See, for example, Figure 10 & Figure 11, which have the same shape but are rotated $90°$ (ignoring the warping caused by the projection used).

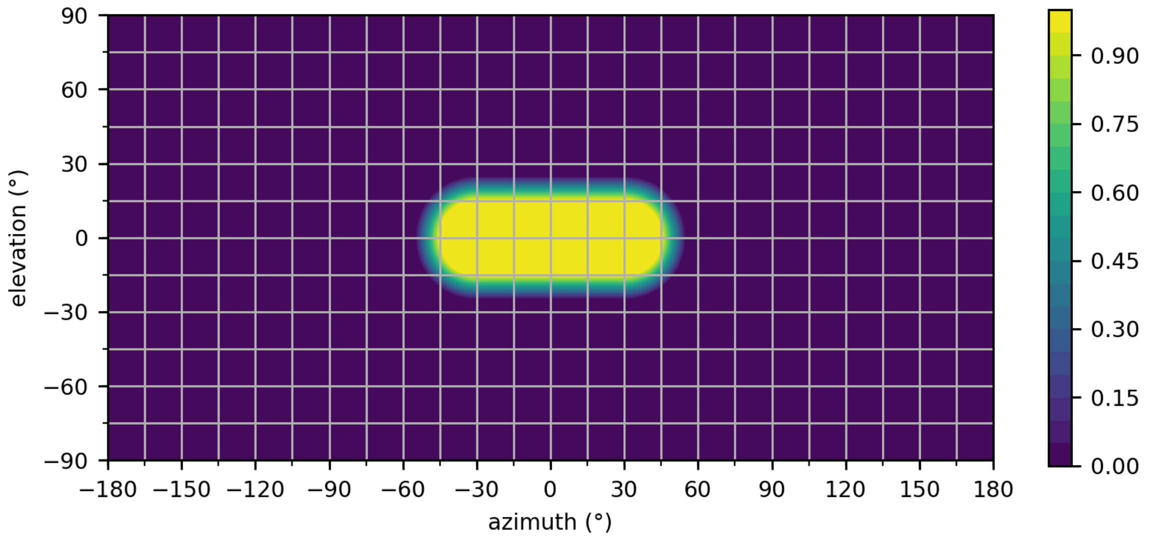


Figure 10: Polar weighting function for $\mathrm{width}{=}90^\circ$ and $\mathrm{height}{=}30^\circ$

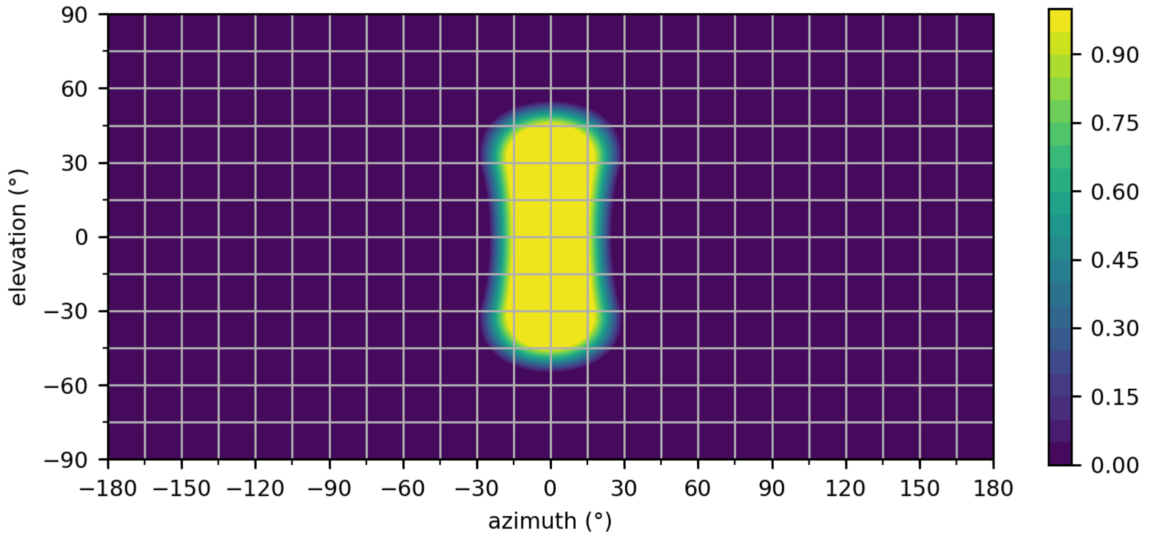


Figure 11: Polar weighting function for $\mathrm{width}{=}30^\circ$ and $\mathrm{height}{=}90^\circ$

- The approximate weighting function is now 1 inside a maximally-rounded $\mathtt{width} \times \mathtt{height}$ rectangle (stadium) in azimuth-elevation space, with a few modifications:
  - The rounded caps are circular in Cartesian space, as the weight is calculated based on the angle from two vectors at their centre. When $\mathtt{width} = \mathtt{height}$, the weighting function is circular.
  - At $\mathtt{width} > 180°$, the width is increased so that when the width reaches $360°$ the rounded parts overlap completely, forming a 'band', where the weighting function has the same value for all positions of the same elevation. See Figure 12 & Figure 13.

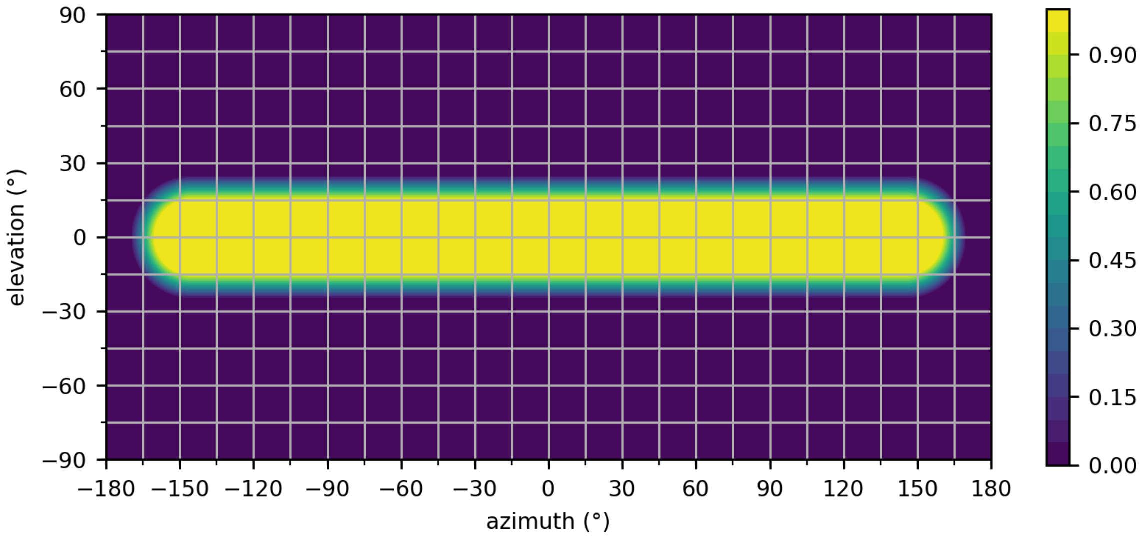


Figure 12: Polar weighting function for width=300° and height=30°

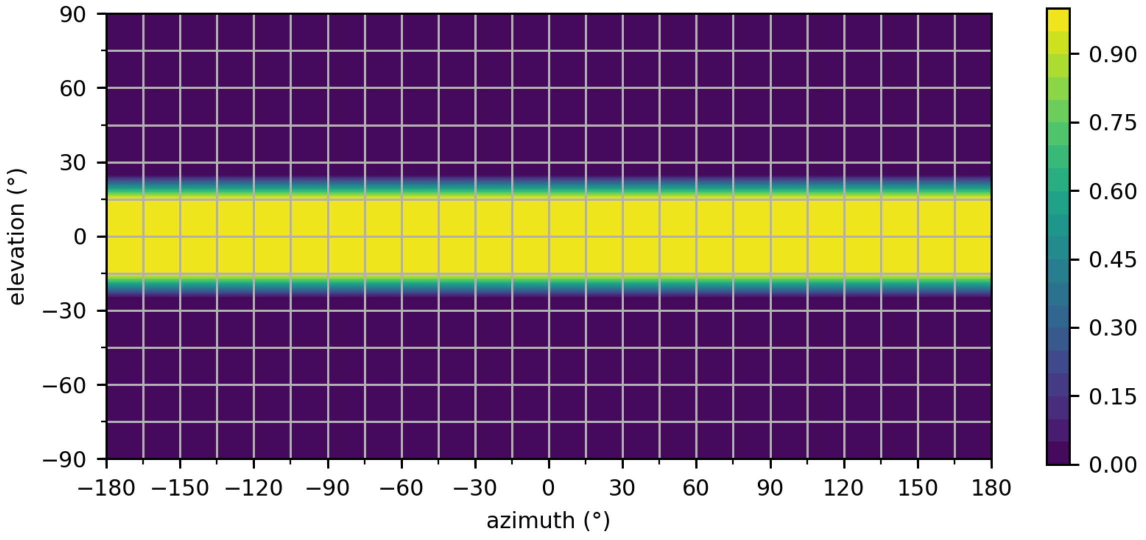


Figure 13: Polar weighting function for width=360° and height=30°

  - A fade is added to the edge of the weighting function; the weight drops from 1 to 0 as the angular distance from the extent reaches $10°$.

This function is implemented in `core.objectbased.extent.PolarExtentPanner.get_weight_func`.

7.3.7.3 Rendering Cartesian Extent

In order to handle the full range of positions and extents allowed in the ADM, the size must be modified before the spread panner can be used with the Cartesian Weighting Function described below to calculate the loudspeaker gains.

The procedure used to calculate loudspeaker gains for `position`, `width`, `height` and `depth` parameters in *cartesian* mode is as follows:

- The ADM *width*, *height* and *depth* parameters are the distance from one edge of the extent to another, so the sizes are halved to match the definition of the ellipsoid used by the Cartesian Weighting Function.
- If the extent is small, then a step in the loudspeaker gains may occur as the position moves through the origin of the coordinate system, so the size along each axis is linearly interpolated between the original size at $\mathtt{radius} \geq 1$ and a minimum size at $\mathtt{radius} = 0$. The minimum size is given by $2\tan 5° \approx 0.175$
- A spread gain vector is produced using the Cartesian Weighting Function and the spreading panner described in § 7.3.7.1, using the modified extent above, with the size along each axis increased to at least the minimum extent. A point source gain vector is produced using the point source panner. The extended and point source gains are mixed together so that for zero extents, only the point source gains are used, and for extents where the size of any axis is over the minimum extent, only the extended gains are used.

This is done to avoid using the spreading panner for small extents, where the discrete sampling of the weighting function may result in gains which do not change smoothly with respect to the parameters.

This is implemented in `core.objectbased.extent.CartExtentPanner.calc_pv_spread`.

Cartesian Weighting Function

The weighting function for Cartesian extent rendering is parametrised by 3D Cartesian vectors `position`, p and `size` s. These vectors define an ellipsoid, with surface points x described by:

$$\left\| \frac{x - p}{s} \right\|_2 = 1 \qquad (5)$$

The weight for a virtual source positioned at v is the square of the distance that the ray vt for $t > 0$ is inside this ellipsoid. The squared distance is used to avoid high rates of change for rays which pass through the edge of the ellipsoid.

$$w(v) = \left(\int_0^\infty H\left(1 - \left\| \frac{vt - p}{s} \right\|_2 \right) dt \right)^2$$

Where H is the Heaviside step function.

This is solved by substituting $x = vt$ into equation 5 to yield a quadratic, and solving for t, to find the distances t_1 and t_2 along the ray where the intersections with the ellipsoid lie. Then:

$$w(v) = |\max\{t_1, 0\} - \max\{t_2, 0\}|^2$$

If there are no real solutions then $w(v) = 0$.

This method is implemented in `core.objectbased.extent.CartExtentPanner.get_weight_func`.

7.3.8 Zone Exclusion

Zone exclusion is applied by downmixing the loudspeaker gain vector produced earlier in the gain calculator in order to avoid sending output to loudspeakers in the excluded zone. This can be split into two parts: deciding which of the loudspeakers are within the excluded zone, in § 7.3.8.1, and calculating the downmix to route away from the excluded loudspeakers, in § 7.3.8.2.

Both the selection of excluded loudspeakers and the calculation of the downmix matrix only consider the nominal position of loudspeakers, so that small changes in the loudspeaker positions do not affect the behaviour of zone exclusion.

7.3.8.1 Selecting Excluded Loudspeakers

The selection of loudspeakers is implemented by processing a list of `ExclusionZone` objects, producing a boolean flag for each loudspeaker which is true if the loudspeaker is within any of the exclusion zones and should therefore be excluded.

For `CartesianZone` objects, the following expression is used to determine if a loudspeaker is within the zone, where $\{x, y, z\}$ is the nominal position of the loudspeaker, converted from polar with a radius of 1:

$$\begin{aligned} \mathtt{minX} - \epsilon <{} & x < \mathtt{maxX} + \epsilon \\ \wedge\, \mathtt{minY} - \epsilon <{} & y < \mathtt{maxY} + \epsilon \\ \wedge\, \mathtt{minZ} - \epsilon <{} & z < \mathtt{maxZ} + \epsilon \end{aligned}$$

Where $\epsilon = 10^{-6}$ is a safety margin to allow for rounding errors when converting between polar and Cartesian coordinates.

For `PolarZone` objects, the following expression is used to determine if a loudspeaker is within the zone, where φ and θ denote the nominal azimuth and elevation of the loudspeaker.

$$\begin{aligned} & \mathtt{minElevation} - \epsilon < \theta < \mathtt{maxElevation} + \epsilon \\ \wedge & \begin{pmatrix} & |\theta| > 90 - \epsilon \\ \vee & \mathrm{IAR}(\varphi, \mathtt{minAzimuth}, \mathtt{maxAzimuth}, \epsilon) \end{pmatrix} \end{aligned}$$

IAR is the function `inside_angle_range`; see § 6.2.

Broadly, the elevation of the loudspeaker must be within the allowed range, while the azimuth only has to be within the allowed range if the absolute elevation is less than 90 degrees.

This is implemented in `core.objectbased.gain_calc.ZoneExclusionHandler.get_excluded`.

7.3.8.2 Downmix for Excluded Loudspeakers

Once the loudspeakers within the zone have been determined, a downmix matrix is designed to route the gains away from these loudspeakers.

The zone exclusion panner object associates with each loudspeaker in the layout a list of groups of output loudspeakers. The downmix matrix is such that the gain from an excluded loudspeaker is routed to all the non-excluded loudspeakers in the first group for which there are non-excluded loudspeakers. This functionality is described in more detail in the next two sections.

As an example, Table 2 shows the groups for loudspeakers in 4+5+0. The first row shows that if `M+030` is excluded, the output for this speaker would be routed to `M+000`, unless this is excluded in which case it would be routed to `M-030`, etc. until `U-110`.

Table 2: Example loudspeaker association for 4+5+0

Input	Output Groups
$\mathrm{M+030}$	$\{\mathrm{M+030}\}$, $\{\mathrm{M+000}\}$, $\{\mathrm{M-030}\}$, $\{\mathrm{M+110}\}$, $\{\mathrm{M-110}\}$, $\{\mathrm{U+030}\}$, $\{\mathrm{U-030}\}$, $\{\mathrm{U+110}\}$, $\{\mathrm{U-110}\}$
$\mathrm{M-030}$	$\{\mathrm{M-030}\}$, $\{\mathrm{M+000}\}$, $\{\mathrm{M+030}\}$, $\{\mathrm{M-110}\}$, $\{\mathrm{M+110}\}$, $\{\mathrm{U-030}\}$, $\{\mathrm{U+030}\}$, $\{\mathrm{U-110}\}$, $\{\mathrm{U+110}\}$
$\mathrm{M+000}$	$\{\mathrm{M+000}\}$, $\{\mathrm{M+030,M-030}\}$, $\{\mathrm{M+110,M-110}\}$, $\{\mathrm{U+030,U-030}\}$, $\{\mathrm{U+110,U-110}\}$
$\mathrm{M+110}$	$\{\mathrm{M+110}\}$, $\{\mathrm{M-110}\}$, $\{\mathrm{M+030}\}$, $\{\mathrm{M+000}\}$, $\{\mathrm{M-030}\}$, $\{\mathrm{U+110}\}$, $\{\mathrm{U-110}\}$, $\{\mathrm{U+030}\}$, $\{\mathrm{U-030}\}$
$\mathrm{M-110}$	$\{\mathrm{M-110}\}$, $\{\mathrm{M+110}\}$, $\{\mathrm{M-030}\}$, $\{\mathrm{M+000}\}$, $\{\mathrm{M+030}\}$, $\{\mathrm{U-110}\}$, $\{\mathrm{U+110}\}$, $\{\mathrm{U-030}\}$, $\{\mathrm{U+030}\}$
$\mathrm{U+030}$	$\{\mathrm{U+030}\}$, $\{\mathrm{U-030}\}$, $\{\mathrm{U+110}\}$, $\{\mathrm{U-110}\}$, $\{\mathrm{M+030}\}$, $\{\mathrm{M+000}\}$, $\{\mathrm{M-030}\}$, $\{\mathrm{M+110}\}$, $\{\mathrm{M-110}\}$
$\mathrm{U-030}$	$\{\mathrm{U-030}\}$, $\{\mathrm{U+030}\}$, $\{\mathrm{U-110}\}$, $\{\mathrm{U+110}\}$, $\{\mathrm{M-030}\}$, $\{\mathrm{M+000}\}$, $\{\mathrm{M+030}\}$, $\{\mathrm{M-110}\}$, $\{\mathrm{M+110}\}$
$\mathrm{U+110}$	$\{\mathrm{U+110}\}$, $\{\mathrm{U-110}\}$, $\{\mathrm{U+030}\}$, $\{\mathrm{U-030}\}$, $\{\mathrm{M+110}\}$, $\{\mathrm{M-110}\}$, $\{\mathrm{M+030}\}$, $\{\mathrm{M+000}\}$, $\{\mathrm{M-030}\}$
$\mathrm{U-110}$	$\{\mathrm{U-110}\}$, $\{\mathrm{U+110}\}$, $\{\mathrm{U-030}\}$, $\{\mathrm{U+030}\}$, $\{\mathrm{M-110}\}$, $\{\mathrm{M+110}\}$, $\{\mathrm{M-030}\}$, $\{\mathrm{M+000}\}$, $\{\mathrm{M+030}\}$

A more complicated example where the grouping has some effect is `M+000`. If this is excluded, then this channel would be split between the non-excluded loudspeakers in $\{\mathrm{M+030,M-030}\}$, unless both of these loudspeakers are excluded in which case it would be routed to the non-excluded loudspeakers in $\{\mathrm{M+110,M-110}\}$, etc.

This functionality is implemented in `core.objectbased.zone.ZoneExclusionDownmix` and `core.objectbased.gain_calc.ZoneExclusionHandler`.

Determination of Loudspeaker Groups

During initialisation, the output loudspeaker groups for each loudspeaker are determined.

For each input loudspeaker, each output loudspeaker is assigned a tuple of floats termed a *key*. The output groups then consist of the output loudspeakers sorted by key, and collected into groups with similar keys. The ordering and grouping is therefore defined mainly by the key function.

The key for an input and output loudspeaker consists of 4 keys:

- An integer layer priority, which is zero if both loudspeakers are on the same layer, and increases as the input and output layers are separated, preferring to select a speaker from a higher layer before a lower one. The layer priorities are drawn from Table 3.
- An integer front/back priority, which is lower if input and output loudspeakers are both in front, to the side of, or behind the listener. Given the y component of the polar nominal position of the input and output loudspeakers after converting to Cartesian, y_i and y_o, this is calculated as:

$$|\mathrm{sgn} y_i - \mathrm{sgn} y_o|$$

- The vector distance between the nominal positions of the two loudspeakers, in order to prefer smaller movements.
- The absolute difference in nominal y coordinates between the two loudspeakers, in order to split groups which are not symmetrical around the yz or xz planes.

Table 3: Layer priority value between two loudspeakers.

Input Layer	Bottom	Mid	Upper	Top
Bottom	0	1	2	3
Mid	3	0	1	2
Upper	3	2	0	1
Top	3	2	1	0

Application of Zone Exclusion

The downmix matrix for a set of excluded loudspeakers E is calculated as follows:

- For N loudspeakers, start with an $N \times N$ downmix matrix $\mathbf{D}$, with each element initialised to 0.
- For each input loudspeaker i, consider each group of candidate loudspeaker indices C in row i of the group table.
    - If all loudspeakers in the group are in the set of ignored loudspeakers, that is $C \subseteq E$, move to the next group.
    - Otherwise, for each j in $C \setminus E$ (the set of loudspeakers in the group that is not excluded), set:

$$D_{i,j} = \frac{1}{|C \setminus E|}$$

  and move to the next loudspeaker.

If all loudspeakers are excluded, $\mathbf{D}$ is set to the identity matrix.

$\mathbf{D}$ is then applied to the incoming gain vector $\mathbf{G}$ to produce $\mathbf{G}'$, by:

$$\mathbf{G}'_j = \sqrt{\sum_i \mathbf{G}_i^2 \, \mathbf{D}_{i,j}}$$

7.4 Decorrelation Filters

When rendering objects where the *diffuse* parameter is greater than 0, the diffuse path of the object renderer is used, which has one decorrelation filter per loudspeaker output.

The filters used are $N = 512$ sample long random-phase allpass FIR filters. The filter for a given output is generated as follows:

- A pseudorandom vector $\mathbf{r}$ with values in the range $[0,1)$ of length $\frac{N}{2} - 1$ is generated using the MT19937 pseudorandom number generator, seeded with the index of the channel name in a sorted list of all channel names in the layout.
- A phase vector $\mathbf{p}$ of length $\frac{N}{2} + 1$ is defined as:

$$\mathbf{p}_n = \begin{cases} 2\pi \mathbf{r}_{n-1} & 1 \le n \le \dfrac{N}{2} - 1 \\ 0 & \text{otherwise} \end{cases}$$

- The corresponding frequency vector $\mathbf{x}$ is defined as $\mathbf{x}_n = \exp(i\mathbf{p}_n)$.
- An inverse real-valued Fourier transform (`irfft` function) is taken of the non-negative-frequency components in $\mathbf{x}$ to obtain the time-domain filter.

This is implemented in `core.objectbased.decorrelate.design_decorrelators`.

The delay introduced by these filters is matched by a $\frac{(N-1)}{2}$ sample delay in the direct path.

7.5 LFE downmix matrix calculation

The downmix matrix $\mathbf{M}_{\mathrm{dmx}}$ mixes each non-LFE channel signal to one of the available LFE channels. It is calculated as follows:

- If the reproduction layout does not have any LFE channels, the downmix matrix is empty.
- If the reproduction layout does have exactly one LFE channel with index j, all gains from each loudspeaker with index i to the LFE channel are set to one:

$$\mathbf{M}_{i,j} = 1$$

- If the reproduction layout has exactly two LFE channels, the matrix coefficients for each loudspeaker with index i to the LFE channels with indices a and b are calculated as follows:

  Let $\mathbf{p}_a$ and $\mathbf{p}_b$ be the position of the two LFE loudspeakers, and $\mathbf{p}_i$ be the position of the ith loudspeaker.

  The calculated gain is the projection of the position onto the vector between $\mathbf{p}_a$ and $\mathbf{p}_b$, scaled to 1 at $\mathbf{p}_b$:

$$\begin{aligned}
\mathbf{v} &= \mathbf{p}_b - \mathbf{p}_a \\
\mathbf{v}' &= \frac{\mathbf{v}}{(\| \mathbf{v} \|_2)^2} \\
g &= \mathbf{v}' \cdot (\mathbf{p}_i - \mathbf{p}_a) \\
g' &= \min\{1, \max\{0, g\}\} \\
\mathbf{M}_{i,a} &= 1 - g' \\
\mathbf{M}_{i,b} &= g'
\end{aligned}$$

  Where $\mathbf{M}_{i,a}$ denotes the downmix coefficient for the i-th loudspeaker to the first LFE channel, $\mathbf{M}_{i,b}$ denotes the downmix coefficient for the i-th loudspeaker to the second LFE channel.

This is implemented in `core.subwoofer.lfe_downmix_matrix`.

8. Render Items with typeDefinition==DirectSpeakers

To render an *audioChannelFormats* with *typeDefintion==DirectSpeakers* it is routed to a matching speaker. If this is not possible the PSP will be used as a fallback.

The basic algorithm is as follows:

1. Determine if the metadata refers to an LFE channel (see § 8.1). If it does, then only LFE outputs will be considered, and if it doesn't, only non-LFE outputs will be considered.
2. If any of the *speakerLabels* match a loudspeaker (see § 8.2) the channel is routed to the first loudspeaker that matches. If no *speakerLabel* matches, continue to the next step.
3. If `screenEdgeLock` is specified the nominal position will be shifted to the horizontal and/or vertical edge of the screen. The minimum and maximum bounds are left untouched (see § 8.3).
4. If the nominal position of any loudspeaker is within the specified position bounds (see § 8.4), route the channel to the loudspeaker closest to the specified nominal position. If there are no loudspeakers within the bounds, or the closest loudspeaker to the nominal is not unique), continue to the next step.
5. If the metadata refers to an LFE route the channel to `LFE1` (if it exists), or discard it. If the metadata refers to a non-LFE channel, use the PSP to render the channel at its nominal position.

The following subsections describe the individual steps in more detail.

This is implemented in `core.direct_speakers.panner.DirectSpeakersPanner`.

8.1 LFE Determination

A channel is considered to be an LFE channel if either the *frequency* element in the *audioChannelFormat* has a *lowPass* of $\leq 200\,\mathrm{Hz}$ (see § 6.3), or if there is a *speakerLabel* which refers to an LFE channel (`LFE1` or `LFE2` after the matching process described below has been applied).

8.2 Speaker Label Matching

The matching for speakerLabels only works for the labels used in [BS.2051] (e.g. M+030) and the URNs used in the common definitions file [BS.2094] (e.g. urn:itu:bs:2051:0:speaker:M+030). To bring the labels of the common definitions in line with the labels in [BS.2051] some substitutions are applied:

- `LFE` → `LFE1`
- `LFEL` → `LFE1`
- `LFER` → `LFE2`

8.3 Screen Edge Lock

The *screenEdgeLock* implementation for *typeDefintion==DirectSpeakers* reuses the `ScreenEdgeLockHandler` used for *typeDefintion==Objects*; described in detail in § 7.3.4. It is used to transform the nominal position only; the minimum and maximum bounds will be left untouched.

This means that if bounds are specified then they are interpreted as absolute bounds irrespective of the screen position; the source will only lock to a channel within the original specified bounds. If bounds are not specified, then the point source panner behaviour will be activated, causing the source to lock to the edge of the screen regardless of if there is a loudspeaker there or not. It is recommended that *screenEdgeLock* and coordinate bounds should not be used together.

8.4 Bounds Matching

A specified minimum or maximum *bound* expands the allowable range away from the nominal position. If the minimum or maximum *bound* is not specified it is set to the nominal coordinate. A speaker matches if all coordinates lie within the specified *bounds*. With the exception, that speakers with polar coordinates at the poles (e.g. `T+000`) match any azimuth range, as they have an indeterminate azimuth.

A loudspeaker with polar position `speaker` matches if

$$\begin{aligned} &\begin{pmatrix} & \mathrm{IAR}(\mathtt{speaker.azimuth},\ \mathtt{azimuth.min},\ \mathtt{azimuth.max}, \epsilon) \\ \vee & |\mathtt{speaker.elevation}| \geq 90° - \epsilon \end{pmatrix} \\ \wedge\quad & \mathtt{elevation.min} - \epsilon \leq \mathtt{speaker.elevation} \leq \mathtt{elevation.max} + \epsilon \\ \wedge\quad & \mathtt{distance.min} - \epsilon \leq \mathtt{speaker.distance} \leq \mathtt{distance.max} + \epsilon \end{aligned}$$

Where IAR is the function `inside_angle_range` (see § 6.2) and $\epsilon = 10^{-5}$ is a safety margin to allow for rounding errors.

A loudspeaker with Cartesian position `speaker` matches if

$$\begin{aligned} & \mathtt{X.min} - \epsilon && \leq \mathtt{speaker.X} \leq \mathtt{X.max} + \epsilon \\ \wedge\ & \mathtt{Y.min} - \epsilon && \leq \mathtt{speaker.Y} \leq \mathtt{Y.max} + \epsilon \\ \wedge\ & \mathtt{Z.min} - \epsilon && \leq \mathtt{speaker.Z} \leq \mathtt{Z.max} + \epsilon \end{aligned}$$

is true.

9. Render Items with typeDefinition==HOA

9.1 Supported HOA formats

9.1.1 HOA order and degree

HOA signals, as defined by the ADM standard, can be rendered up to the order 50 (see details below). In ADM, the HOA channels are signalled individually by their *order* and *degree* via the corresponding HOA type sub-elements. Thus, full-3D HOA scenes (comprised of every order l and degree m up to a given order L), 2D HOA scenes (comprised of every HOA component such that $|m| = l$ up to a given order L), as well as mixed-order HOA scenes can be rendered.

However, in the event where two HOA signals share the same order *and* degree, an exception is raised and the signals are not rendered.

9.1.2 Normalisation

HOA signal normalisation is indicated via the *normalization* HOA type sub-element. All three possible normalisations (N3D, SN3D and FuMa) are supported by this renderer. In ADM, HOA normalisation is specified for each HOA signal individually, thus it is theoretically possible to define HOA scenes whereby the different signals use different normalisations. However this is not supported by this renderer: all HOA channels in an *audioBlockFormat* must share the same normalisation. Lastly, note that the FuMa normalisation is supported up to order 3 only.

9.2 Unsupported sub-elements

The following three sub-elements of the HOA type are currently not interpreted in the rendering:

- *nfcRefDist*, which indicates a reference distance for the loudspeakers. The Near-Field Compensation (NFC) effect, which compensates mismatches between the loudspeaker reference distance and the distance at which loudspeakers are located in the playback layout, is not implemented in the EBU ADM renderer. Implementing this effect in the HOA rendering significantly increases the computational complexity of the renderer, while having a relatively minor impact on the listener's perception of the audio content.
- *screenRef*, which indicates whether the HOA component is screen-related. The expected use of this sub-element is ambiguous in the HOA context; therefore it is not taken into account in the rendering.
- *equation*, which is meant to be used as a replacement of the *order* and *degree* sub-elements. The current ADM standard does not provide precise rules regarding the format used to specify mathematical formulas. Therefore, this sub-element cannot be supported reliably.

Note that, similar to the *normalization* sub-element, all HOA channels in an *audioBlockFormat* must share the same *nfcRefDist* and *screenRef* value to be rendered.

9.3 Rendering of HOA signals over loudspeakers

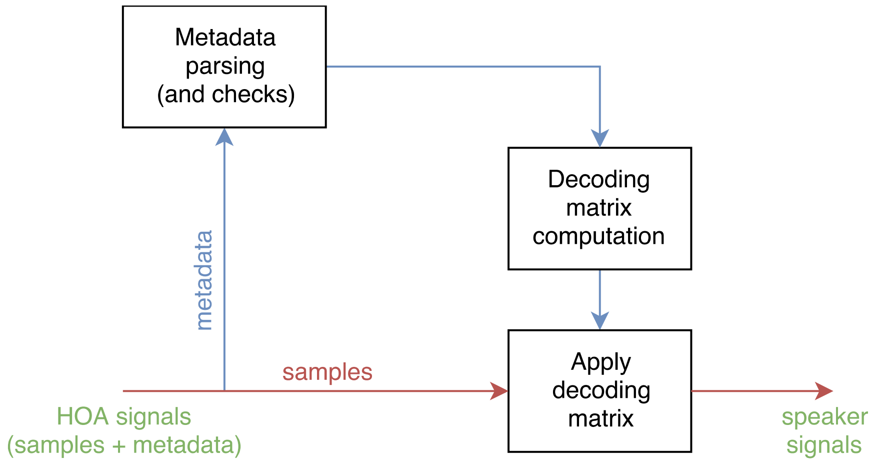

Metadata parsing (and checks)
Decoding matrix computation
metadata
samples
Apply decoding matrix
HOA signals (samples + metadata)
speaker signals

Figure 14: HOA rendering flow diagram

The process of rendering HOA signals over loudspeakers is summarized in Figure 14. First, the ADM metadata is parsed to identify the format of the HOA object and check whether the signals can be rendered unambiguously. Specifically, as stated above, all HOA channels in an *audioBlockFormat* must share the same *normalization*, *nfcRefDist* and *screenRef* sub-element values. Then, a loudspeaker decoding matrix is calculated and applied to the HOA signals. This is expressed by the following equation:

$$\mathbf{S}_{\text{spk}} = \mathbf{D}\,\mathbf{S}_{\text{HOA}}$$

Where:

- $\mathbf{S}_{\text{spk}}$ is a matrix of speaker signals, with dimensions $N_{\text{spk}} \times N_{\text{samp}}$.
- $\mathbf{S}_{\text{HOA}}$ is a matrix of HOA signals, with dimensions $N_{\text{HOA}} \times N_{\text{samp}}$.
- $\mathbf{D}$ is a real-valued matrix, with dimensions $N_{\text{spk}} \times N_{\text{HOA}}$, and is referred to as the *HOA decoding matrix*.
- N_{HOA}, N_{spk} and N_{samp} denote the number of HOA signals, speaker signals and times samples, respectively.

This section expresses the decoding matrix calculation in ACN channel ordering, however the channel allocation used is as specified in the *order* and *degree* parameters in the *audioBlockFormat*.

The decoding matrix is applied through the use of the Block Processing Channel structure described in § 6.4. Specifically, for each incoming `HOATypeMetadata` object, a single `FixedMatrix` processing block is generated, which applies the decoding matrix between times determined in § 6.5.

9.3.1 HOA decoding matrix calculation

The EBU ADM renderer implements the AllRAD HOA decoding technique, as described in [Zotter2012]. This method provides robust HOA decoding over irregular loudspeaker layouts such as that described in [BS.2051]. The calculation of the decoding matrix is done in `core.scenebased.design.HOADecoderDesign`.

Conceptually, the AllRAD decoding method is equivalent to:

1. Decoding the HOA signals to a grid of virtual speakers which are evenly distributed over the sphere, and

2. Panning the virtual speaker signals over the actual speakers.

Mathematically, this can be expressed as:

$$\mathbf{D}' = \nu\, \mathbf{G}\, \mathbf{D}_{\mathrm{virt}}$$

$$\mathbf{D} = \mathbf{D}'\mathrm{diag}(\mathbf{n}^{-1})$$

Where $\mathbf{D}'$ denotes the HOA decoding matrix for *N3D* normalisation, $\mathbf{G}$ is the panning gain matrix, $\mathbf{D}_{\mathrm{virt}}$ is the virtual speaker decoding matrix and ν is an energy normalisation factor. $\mathbf{D}$ is the completed decoding matrix after applying the HOA normalisation vector $\mathbf{n}$ to $\mathbf{D}'$ to apply the desired normalisation.

9.3.1.1 Virtual speaker positions

In order to facilitate the calculation of the decoding matrix, the angular positions of the virtual speakers must be distributed as evenly as possible over the sphere. In addition, as a rule of thumb, there should be about twice as many virtual speaker positions than there are HOA signals.

In the EBU ADM renderer, the virtual speaker positions constitute a 5200-point *spherical-T design*, which makes it well suited for decoding HOA signals up to order 50.

9.3.1.2 Calculation of the virtual speaker decoding matrix

In order to calculate the decoding matrix for the virtual speakers, first the matrix of the HOA coefficients for the virtual speakers, $\mathbf{Y}_{\mathrm{virt}}$, is calculated. This matrix is given by:

$$\mathbf{Y}_{\mathrm{virt}} = \left[\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_{N_{\mathrm{virt}}}\right]$$

$$\mathbf{y}_n = [Y_0^0(\theta_n, \phi_n), Y_1^{-1}(\theta_n, \phi_n), \ldots]^{\mathrm{T}}$$

Where (θ_n, ϕ_n) denotes the elevation and azimuth angles for the n-th virtual speaker (using the HOA coordinate system and notation as defined in [BS.2076]) and Y_l^m denote the real-valued order-l and degree-m spherical harmonic function with *N3D* normalisation. Note that the value of each $Y_l^m(\theta, \phi)$ term depends on the *order* and *degree* sub-elements for each HOA channel.

The virtual speaker HOA decoding matrix is then calculated as the transpose of $\mathbf{Y}_{\mathrm{virt}}$:

$$\mathbf{D}_{\mathrm{virt}} = N_{\mathrm{samp}}^{-1}\mathbf{Y}_{\mathrm{virt}}^{\mathrm{T}}$$

For the choice of virtual loudspeaker positions and *N3D* normalisation this is equivalent to taking the pseudo-inverse of $\mathbf{Y}_{\mathrm{virt}}$.

9.3.1.3 Calculation of the panning gain matrix

The HOA decoding matrix is normalised so that, in the case where the HOA scene consists of a single point source, the total power of the speaker signals is equal to that of the source signal, on average for every possible source location over the sphere.

Mathematically, the normalisation factor ν is calculated as:

$$\nu = \frac{\sqrt{N_{\mathrm{virt}}}}{\|\mathbf{G}\,\mathbf{D}_{\mathrm{virt}}\,\mathbf{Y}_{\mathrm{virt}}\|_{\mathrm{F}}}$$

Where $\|\cdot\|_{\mathrm{F}}$ denotes the Frobenius norm.

9.3.1.5 HOA normalisation

The decoding matrix is divided by the vector $\mathbf{n}$ in order to convert the signal to the *N3D* normalisation for which $\mathbf{D}'$ is designed for. $\mathbf{n}$ is defined for a given *normalization* parameter `norm` as:

$$\mathbf{n}_n^m = \frac{N_{\mathtt{norm}}{}_n^{|m|}}{N_{\mathrm{N3D}}{}_n^{|m|}}$$

$$\mathbf{n} = [\mathbf{n}_0^0, \mathbf{n}_1^{-1}, \dots]$$

10. References

[BS.2076] ITU-R Recommendation BS.2076-1: *Audio Definition Model*.
https://www.itu.int/dms_pubrec/itu-r/rec/bs/R-REC-BS.2076-1-201706-I!!PDF-E.pdf

[BS.2051] ITU-R Recommendation BS.2051-1: *Advanced sound system for programme production*.
https://www.itu.int/dms_pubrec/itu-r/rec/bs/R-REC-BS.2051-1-201706-I!!PDF-E.pdf

[BS.2088] ITU-R Recommendation BS.2088-0: *Long-form file format for the international exchange of audio programme materials with metadata*.
https://www.itu.int/dms_pubrec/itu-r/rec/bs/R-REC-BS.2088-0-201510-I!!PDF-E.pdf

[BS.775] ITU-R Recommendation BS.775-3: *Multichannel stereophonic sound system with and without accompanying picture*.
https://www.itu.int/dms_pubrec/itu-r/rec/bs/R-REC-BS.775-3-201208-I!!PDF-E.pdf

[BS.2094] ITU-R Recommendation BS.2094-1: *Common definitions for the Audio Definition Model*.
https://www.itu.int/dms_pubrec/itu-r/rec/bs/R-REC-BS.2094-1-201706-I!!PDF-E.pdf

[Zotter2012] F. Zotter and M. Frank, "*All-round Ambisonic panning and decoding*," Journal of the audio engineering society, vol. 60, no. 10, pp. 807–820, 2012.

Note: All links are current at time of publishing this document

Annex A: Internal Metadata Structures

A1 *Shared Structures*

```
struct Position { };

struct PolarPosition : Position {
  float azimuth, elevation, distance = 1;
};

struct CartPositon : Position {
  float x, y, z;
};

struct Screen { };

struct PolarScreen : Screen {
  float aspectRatio;
  PolarPosition centrePosition;
  float widthAzimuth;
};

struct CartesianScreen : Screen {
  float aspectRatio;
  CartPositon centrePosition;
  float widthX;
};

struct Frequency {
  optional<float> lowPass;
  optional<float> highPass;
};

struct ExtraData {
  Fraction object_start;
  Fraction object_duration;
  Screen reference_screen;
  Frequency channel_frequency;
};
```

A2 *Input Metadata*

```
struct ChannelLock {
  optional<float> maxDistance;
};

struct ObjectDivergence {
  float value;
  optional<float> azimuthRange;
  optional<float> positionRange;
};
```

```
struct JumpPosition {
  bool flag;
  optional<float> interpolationLength;
};

struct ExclusionZone { };

struct CartesianZone : ExclusionZone {
  float minX;
  float minY;
  float minZ;
  float maxX;
  float maxY;
  float maxZ;
};

struct PolarZone : ExclusionZone {
  float minElevation;
  float maxElevation;
  float minAzimuth;
  float maxAzimuth;
};

struct ScreenEdgeLock {
    enum Horizontal { LEFT; RIGHT; };
    enum Vertical { BOTTOM; TOP; };

    optional<Horizontal> horizontal;
    optional<Vertical> vertical;
};

struct ObjectPosition { };

class PolarObjectPosition : ObjectPosition {
    float azimuth, elevation, distance;
    ScreenEdgeLock screenEdgeLock;
};

class CartesianObjectPosition | ObjectPosition {
    float X, Y, Z;
    ScreenEdgeLock screenEdgeLock;
};

struct AudioBlockFormatObjects {
  ObjectPosition position;
  bool cartesian;
  float width, height, depth;
  float diffuse;
  optional<ChannelLock> channelLock;
  optional<ObjectDivergence> objectDivergence;
  optional<JumpPosition> jumpPosition;
  bool screenRef;
  int importance;
  vector<ExclusionZone> zoneExclusion;
};
```

```
struct ObjectTypeMetadata {
  AudioBlockFormatObjects block_format;
  ExtraData extra_data;
};
```

A3 Reproduction Environment Data

```
struct Channel {
  string name;
  /// The real position of the loudspeaker
  PolarPosition polar_position;
  /// The nominal position of the loudspeaker as in bs.2051.
  PolarPosition polar_nominal_position;
  bool is_lfe;
};

struct Layout {
  /// the ITU-format layout name, e.g. "9+10+3"
  string name;
  vector<Channel> channels;
  Screen screen;
};
```