ITUPublications
Recommendations

International Telecommunication Union
Radiocommunication Sector

Recommendation ITU-R BS.2127-1

(11/2023)

BS Series: Broadcasting service (sound)

Audio Definition Model renderer for advanced sound systems

Foreword

The role of the Radiocommunication Sector is to ensure the rational, equitable, efficient and economical use of the radio-frequency spectrum by all radiocommunication services, including satellite services, and carry out studies without limit of frequency range on the basis of which Recommendations are adopted.

The regulatory and policy functions of the Radiocommunication Sector are performed by World and Regional Radiocommunication Conferences and Radiocommunication Assemblies supported by Study Groups.

Policy on Intellectual Property Right (IPR)

ITU-R policy on IPR is described in the Common Patent Policy for ITU-T/ITU-R/ISO/IEC referenced in Resolution ITU-R 1. Forms to be used for the submission of patent statements and licensing declarations by patent holders are available from http://www.itu.int/ITU-R/go/patents/en where the Guidelines for Implementation of the Common Patent Policy for ITU-T/ITU-R/ISO/IEC and the ITU-R patent information database can also be found.

Series of ITU-R Recommendations

(Also available online at https://www.itu.int/publ/R-REC/en)

Series	Title
BO	Satellite delivery
BR	Recording for production, archival and play-out; film for television
BS	**Broadcasting service (sound)**
BT	Broadcasting service (television)
F	Fixed service
M	Mobile, radiodetermination, amateur and related satellite services
P	Radiowave propagation
RA	Radio astronomy
RS	Remote sensing systems
S	Fixed-satellite service
SA	Space applications and meteorology
SF	Frequency sharing and coordination between fixed-satellite and fixed service systems
SM	Spectrum management
SNG	Satellite news gathering
TF	Time signals and frequency standards emissions
V	Vocabulary and related subjects

Note: *This ITU-R Recommendation was approved in English under the procedure detailed in Resolution ITU-R 1.*

Electronic Publication
Geneva, 2023

RECOMMENDATION ITU-R BS.2127-1*

Audio Definition Model renderer for advanced sound systems

(Question ITU-R 135-2/6)

(2019-2023)

Scope

This Recommendation specifies the reference renderer for use, including for programme exchange, with the advanced sound systems specified in Recommendation ITU-R BS.2051-2, and the audio-related metadata specified by the Audio Definition Model (ADM) in Recommendation ITU-R BS.2076-1. The audio renderer converts a set of audio signals with associated metadata to a different configuration of audio signals and metadata, based on the provided content metadata and local environmental metadata.

NOTE – Guidelines explaining the usage of the renderer are being developed.

Keywords

ADM, Audio Definition Model, metadata, renderer, AdvSS, advanced sound system, channel-based audio, object-based audio, scene-based audio, multichannel audio

The ITU Radiocommunication Assembly,

considering

a) that Recommendation ITU-R BS.1909 – Performance requirements for an advanced multichannel stereophonic sound system for use with or without accompanying picture, specifies the requirements for an advanced sound system with or without accompanying picture;

b) that Recommendation ITU-R BS.2051 – Advanced sound system for programme production, specifies an advanced sound system which is a system with a reproduction configuration beyond those specified in Recommendation ITU-R BS.775 or a system with any reproduction configuration that can support channel-based, object-based or scene-based input signal or their combination with metadata;

c) that Recommendation ITU-R BS.2076 – Audio Definition Model, specifies the structure of a metadata model that allows the format and content of audio files to be reliably described;

d) that Recommendation ITU-R BS.2094 – Common definitions for the audio definition model, contains a set of common definitions for the Audio Definition Model;

e) that Recommendation ITU-R BS.2125 – A serial representation of the Audio Definition Model, specifies a format of metadata based on the Audio Definition Model, segmented into a time-series of frames;

f) that reproduction of advanced sound systems requires rendering of metadata associated with sound signals in order to present the content to one of the Recommendation ITU-R BS.2051 loudspeaker configurations;

g) that users of advanced sound systems should have freedom in the selection of a rendering method;

---

* This Recommendation should be brought to the attention of ISO, IEC, SMPTE and ETSI.

h) that it is desirable that there is an open specification of a single reference rendering method that may be used for advanced sound system programmes;

i) that the single reference renderer should allow content producers and broadcasters to monitor and perform quality control during content production, verify the use of metadata, and ensure interoperability with other elements of the production chain,

recommends

1 that the rendering methods described in Annex 1 should be the reference for how ADM metadata specified in Recommendation ITU-R BS.2076-1, and accompanying audio signals, are to be interpreted;

2 that Note 1 below be considered part of the Recommendation.

NOTE 1 – Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure e.g. interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words shall in no way be construed to imply partial or total compliance with this Recommendation.

Annex 1

Specifications for ADM renderer for advanced sound systems

TABLE OF CONTENTS

Page

Annex 1 – Specifications for ADM renderer for advanced sound systems 2

1 Introduction 4

1.1 Abbreviations/Glossary 4

2 Conventions 5

2.1 Notations 5

2.2 Coordinate System 5

3 Structure 6

3.1 Target environment behaviour 7

4 ADM-XML Interface 7

4.1 AudioBlockFormat 8

4.2 Position sub-elements 8

4.3 TypeDefinition 8

5 Rendering Items 8
5.1 Metadata Structures 9
5.2 Determination of Rendering Items 11
5.3 Rendering Item Processing 20
6 Shared Renderer Components 21
6.1 Polar Point Source Panner 22
6.2 Determination if angle is inside a range with tolerance 28
6.3 Determine if a channel is an LFE channel from its frequency metadata 29
6.4 Block Processing Channel 30
6.5 Generic Interpretation of Timing Metadata 31
6.6 Interpretation of `TrackSpecs` 32
6.7 Relative Angle 33
6.8 Coordinate Transformations 33
7 Render Items with typeDefinition==Objects 34
7.1 Structure 34
7.2 InterpretObjectMetadata 34
7.3 Gain Calculator 36
7.4 Decorrelation Filters 63
8 Render Items with typeDefinition==DirectSpeakers 63
8.1 Mapping Rules 64
8.2 LFE Determination 64
8.3 Loudspeaker Label Matching 64
8.4 Screen Edge Lock 64
8.5 Bounds Matching 65
9 Render Items with typeDefinition==HOA 65
9.1 Supported HOA formats 65
9.2 Unsupported sub-elements 66
9.3 Rendering of HOA signals over loudspeakers 66
10 Metadata Conversion 68
10.1 *position* Conversion 69
10.2 Extent Conversion 71
10.3 objectDivergence Conversion 73

11 Data Structures and Tables 73
11.1 Internal Metadata Structures 73
11.2 Allocentric Loudspeaker Positions 75
11.3 DirectSpeakers mapping data 80
Bibliography 85
Attachment 1 to Annex 1 (informative) – Guide to corresponding parts of the specification to ADM Metadata 85
A1.1 ADM Metadata across ITU-R ADM Renderer 85
Attachment 2 to Annex 1 (Informative) – An Alternative virtual loudspeaker configuration 87
A2.1 Specification of alternative virtual loudspeaker configuration 87

1 Introduction

This Recommendation describes an audio renderer providing a complete interpretation of the Audio Definition Model (ADM) metadata, specified in Recommendation ITU-R BS.2076-1. Usage of ADM metadata is recommended to describe audio formats used in programme production for Advanced Sound Systems (AdvSS), also known as Next-Generation Audio (NGA) systems. This renderer is capable of rendering audio signals to all loudspeaker configurations specified in Recommendation ITU-R BS.2051-2.

This specification is accompanied by an open source reference implementation, written in Python for file-based ADM processing, available at:

https://www.itu.int/dms_pub/itu-r/oth/0a/07/R0A0700003E0001ZIPE.zip

This specification document is a description of the reference code.

1.1 Abbreviations/Glossary

ADM	Audio definition model
BMF	Broadcast metadata exchange format
BW64	Broadcast wave 64 format
BWF	Broadcast wave format
HOA	Higher-order ambisonics
NGA	Next generation audio
PSP	Point source panner
VBAP	Vector base amplitude panning
XML	Extensible markup language

2 Conventions

2.1 Notations

In this Recommendation the following conventions will be used:

– Text in italic refers to ADM elements, sub-elements, parameters or attributes of Recommendation ITU-R BS.2076-1: *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 `iar.` is omitted.

– Upper case bold is used for matrices: **X**

– Lower case bold is used for vectors: **x**

– Subscripts in the form x_n denotes the n-th element of a vector **x**

– Sections of monospaced text with colour highlighting are used to describe data structures:

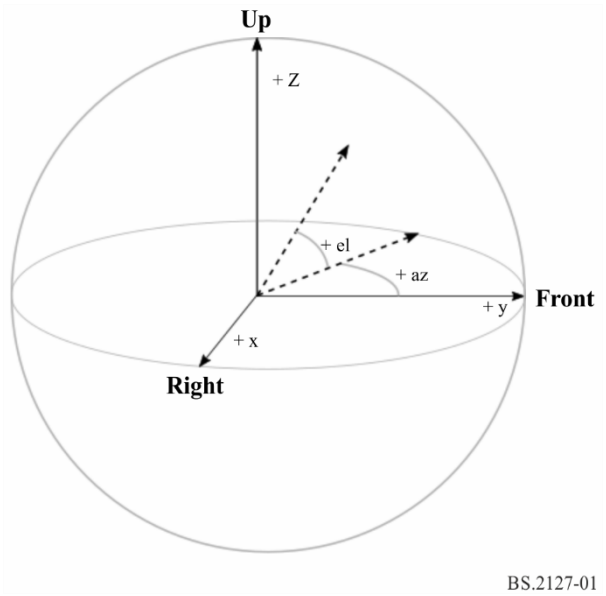
```
struct PolarPosition : Position {
  float azimuth, elevation, distance = 1;
};
```

2.2 Coordinate System

Both Cartesian and Polar Coordinates are used throughout this Recommendation.

FIGURE 1

Coordinate System



The polar coordinates are specified in accordance with Recommendation ITU-R BS.2076-1 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 Recommendation ITU-R BS.2076-1 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 Recommendation ITU-R BS.2076-1, where:

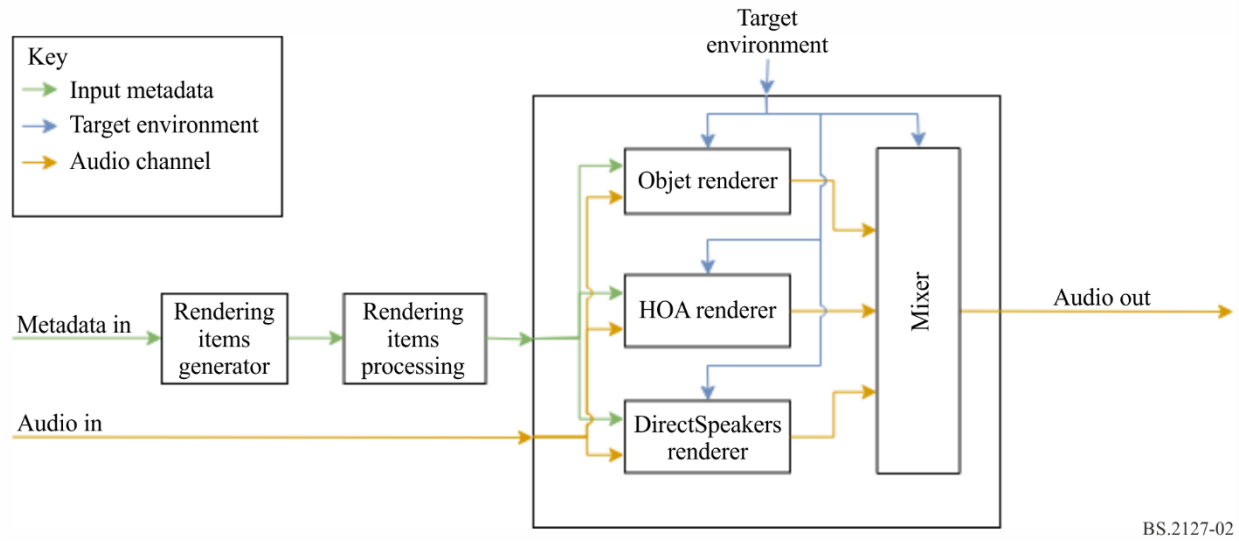
– Elevation, denoted by θ is the angle in radians from the positive Z-Axis.

– 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 §5.2.

– Optional processing to apply importance and conversion emulation is applied to the rendering items as described in § 5.3.

– 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.

Matrix type processing is not shown in the diagram, as this type is handled during the creation of rendering items and as part of the renderers for other types.

3.1 Target environment behaviour

On initialisation, the user may select a loudspeaker layout from those specified in Recommendation ITU-R BS.2051.

The nominal position of each loudspeaker (`polar_nominal_position`) is as specified in Recommendation ITU-R BS.2051. `M+SC` and `M−SC` have nominal azimuths of 15° and −15°.

The real position of each loudspeaker (`polar_position`) may be specified by the user. If this is not given, then the nominal position is used. Given real positions are checked against the ranges given in Recommendation ITU-R BS.2051; if they are not within range, then an error is issued. Additionally, the absolute azimuth of both `M+SC` and `M−SC` loudspeakers must either be between 5° and 25° or between 35° or 60°.

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 Recommendation, 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, the renderer 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 shall be 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 shall be 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 is represented 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 are 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 shall be 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 shall therefore represent 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 (possibly time-varying) parameters needed to render the item;
- `MetadataSource` to hold a series of `TypeMetadata` objects; and
- `RenderingItem` to associate a `MetadataSource` with a source of audio samples and extra information not necessarily required by the renderer.

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;
};
```

Importance data shall be stored in an `ImportanceData` structure:

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

References to input audio samples shall be encapsulated in `TrackSpec` structures, to allow for the specification of silent tracks and Matrix processing. `DirectTrackSpec` specifies that samples shall be read directly from the indicated input track. `SilentTrackSpec` specifies that the samples shall all be zero.

```
struct TrackSpec {};

struct DirectTrackSpec : TrackSpec {
  int track_index;
};

struct SilentTrackSpec : TrackSpec {
};
```

Two `TrackSpec` types are provided to support *typeDefinition==DirectSpeakers*. `MatrixCoefficientTrackSpec` specifies that the parameters specified in `coefficient` (from a Matrix *audioBlockFormat* `coefficient` element) are applied to the samples of `input_track`, while `MixTrackSpec` specifies that the samples from multiple `TrackSpecs` should be mixed together.

```
struct MatrixCoefficientTrackSpec : TrackSpec {
  TrackSpec input_track;
  MatrixCoefficient coefficient;
};

struct MixTrackSpec : TrackSpec {
  vector<TrackSpec> input_tracks;
};
```

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` shall hold the *audioBlockFormat*, the list of *audioPackFormat*s leading to the containing *audioChannelFormat*, plus the common data collected in `ExtraData`.

```
struct DirectSpeakersTypeMetadata : TypeMetadata {
  AudioBlockFormatDirectSpeakers block_format;
  vector<AudioPackFormat> audioPackFormats;
  ExtraData extra_data;
};
```

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

```
struct DirectSpeakersRenderingItem : RenderingItem {
  TrackSpec track_spec;
  MetadataSource metadata_source;
  ImportanceData importance;
};
```

5.1.2 Matrix

typeDefinition==Matrix shall be supported using the `TrackSpec` mechanism in rendering items for other types, so no explicit `MatrixTypeMetadata` or `MatrixRenderingItem` classes are required.

5.1.3 Objects

The `ObjectTypeMetadata` shall hold an *audioBlockFormat* plus the common data collected in `ExtraData`.

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

As each *audioChannelFormat* with *typeDefinition==Objects* can be processed independently, the `RenderingItem` shall contain only a single `TrackSpec`.

```
struct ObjectRenderingItem : RenderingItem {
  TrackSpec track_spec;
  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`.

```
struct HOATypeMetadata : TypeMetadata {
  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 `TrackSpec`, but rather a vector of `TrackSpec`s.

```
struct HOARenderingItem : RenderingItem {
  vector<TrackSpec> track_specs;
  MetadataSource metadata_source;
  vector<ImportanceData> importances;
};
```

5.1.5 Binaural

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

5.2 Determination of Rendering Items

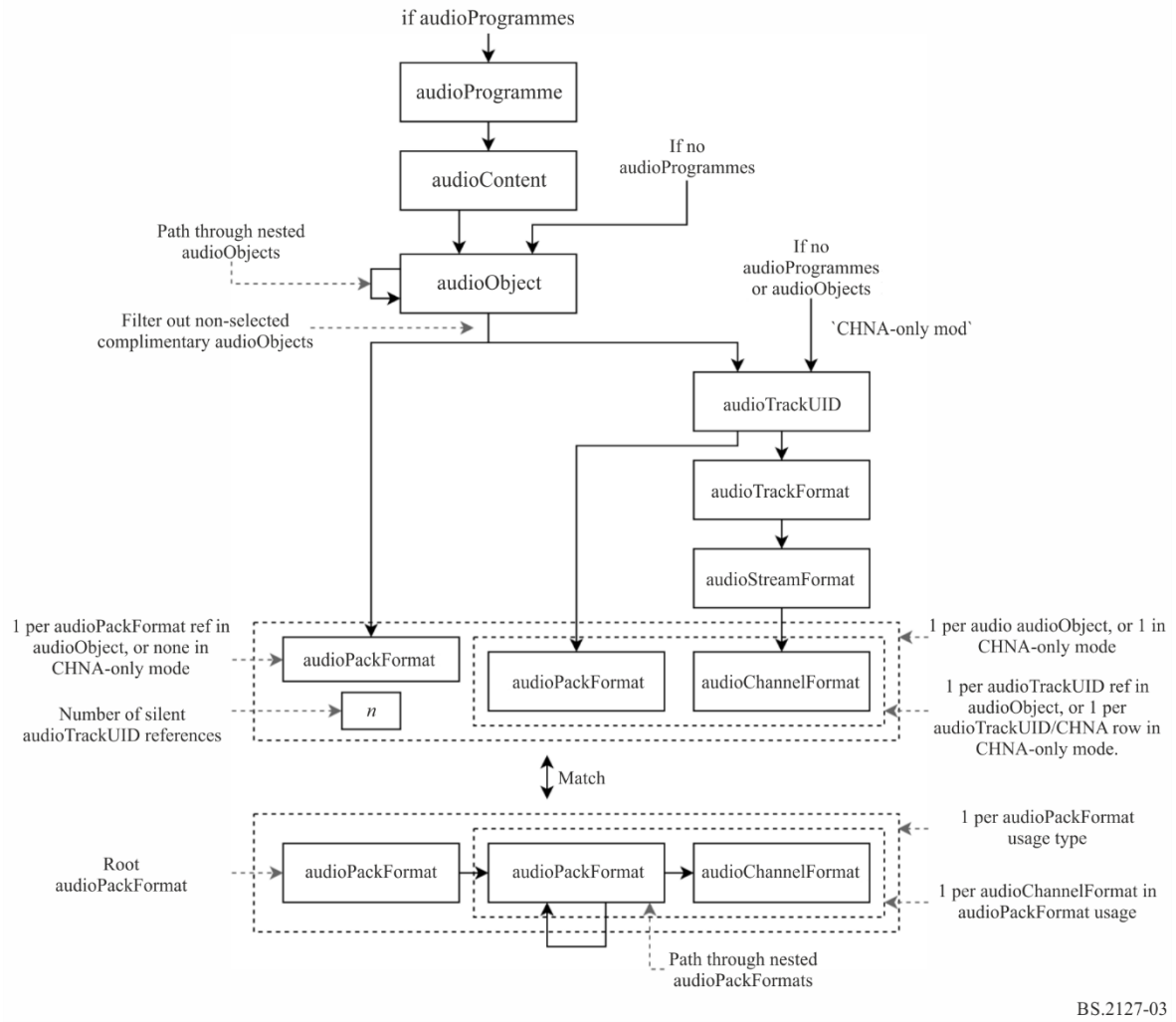
To determine the `RenderingItems`, the ADM structure shall be analysed. Figure 3 illustrates the path that is taken.

The state of the item selection process is carried between the various components in a single object termed the ‘item selection state’, which when completely populated represents all the components that make up a single `RenderingItem`. Each component accepts a single item selection state, and returns copies (zero to many) of it with more entries filled in. These steps are composed together in `select_rendering_items`, a nested loop over the states when modified by each component in turn.

This is implemented in `core.select_items`.

FIGURE 3

Path through ADM structure to determine the `RenderingItems`



5.2.1 Starting point

Rendering item selection can start from multiple points in the ADM structure depending on the elements included in the file.

If there are *audioProgramme* elements, then a single *audioProgramme* is selected; otherwise if there are *audioObject* elements then all *audioObject*s shall be selected; otherwise all *audioTrackUID*s (CHNA rows) are selected (called ‘CHNA-only mode’).

5.2.2 audioProgramme Selection

Only one *audioProgramme* is selected. The programme to use can be selected by the user. If no *audioProgramme* is selected, the one with the numerically lowest ID shall be selected.

5.2.3 audioContent Selection

All *audioContent*s referenced by the selected *audioProgramme* are selected.

5.2.4 audioObject Selection

`audioObjects` shall be set to all possible paths through the *audioObject* hierarchy starting at the selected *audioContent* (following *audioObject* links) in turn.

5.2.5 Complementary *audioObject* Handling

audioComplementaryObject references shall be interpreted as defining groups of *audioObjects*, of which only one *audioObject* will be reproduced.

A group is described by *audioComplementaryObject* references from the default *audioObject* in the group to all non-default *audioObject*s in the group. The user may provide a set of *audioObject*s to select, which overrides the defaults. From this, a set of *audioObject*s to ignore is determined, and states are discarded if any of the *audioObject*s in the *audioObject* path are in this set.

5.2.5.1 Selection of Complementary *audioObjects* to Ignore

First, the set of *audioObject*s selected by the user shall be augmented with the defaults for each group: for each root *audioObject* (an *audioObject* with *audioComplementaryObject* references), if none of the *audioObject*s in the group defined by the root *audioObject* this group are in the set, then the root *audioObject* (the default) shall be added.

The set of *audioObject*s to ignore is then the set of all complementary audioObjects (i.e. *audioObject*s with an *audioComplementaryObject* reference and audioObjects pointed to by an *audioComplementaryObject* reference) minus the augmented set of *audioObject*s selected by the user.

If *audioObject*s not belonging to any complementary group are selected, or multiple *audioObject*s are selected in a single *audioObject* group (either by user error, or as a result of overlapping groups), an error is raised.

5.2.6 *audioPackFormat* Matching

The next step shall be to match the information in an *audioObject* (the list of *audioPackFormat*s, *audioTrackUID*s and number of silent tracks, or simply the list of all *audioTrackUID*s in CHNA-only mode) against the *audioPackFormat* and *audioChannelFormat* structures.

This is specified as a matching/search problem rather than specific paths through the reference structures that have to be resolved, because there are multiple elements on the two sides which have to match and not conflict to form a valid solution.

The match is considered valid only if exactly one solution is found. If no solutions are found, then the metadata is contradictory and an error shall be raised. If multiple solutions are found, then the metadata is ambiguous, and an error shall be raised. For both types of error, diagnostics are run in order to display possible causes of the error to the user.

5.2.6.1 Packs to Match Against

The specification of the *audioPackFormat*s to match against are given as a list of `AllocationPack` structures:

```
struct AllocationChannel {
  AudioChannelFormat channel_format;
  vector<AudioPackFormat> pack_formats;
};

struct AllocationPack {
  AudioPackFormat root_pack;
  vector<AllocationChannel> channels;
};
```

Each one shall specify the root *audioPackFormat* (`root_pack`, the top level *audioPackFormat* which references all channels to be allocated), and a list of the channels to match within that pack. Each channel is a combination of an *audioChannelFormat* reference and a list of possible *audioPackFormat*s which that channel could be associated with.

For each *audioPackFormat* `pack` where *typeDefinition != Matrix*, an `AllocationPack` object is created where:

– `root_pack` is `pack`.

– `channels` has one entry for each *audioChannelFormat* accessible from `pack` (recursively following *audioPackFormat* links), where `pack_formats` contains all the *audioPackFormat*s on the path from `pack` to the *audioChannelFormat* (including `pack`).

While this is a slight simplification of the *audioPackFormat* and *audioChannelFormat* structure, the advantage of this representation is its ability to represent the *audioPackFormat* and *audioChannelFormat* referencing structures used with *Matrix* content, described below.

5.2.6.1.1 Matrix Handling

Matrix *audioPackFormats* can be referenced in multiple ways depending on the intended effect. These reference structures are reflected in the following `AllocationPack`s which are produced for each audioPackFormat `pack` with *typeDefinition==Matrix*:

– If `pack` is a direct or decode matrix, the matrix should be applied if an *audioObject* references both `pack` and a set of *audioTrackUID*s which in turn reference `pack` and channels of the input or encode *audioPackFormat* of `pack`:

  • `root_pack` is pack.

  • `channels` contains one value per *audioChannelFormat* channel in the input *audioPackFormat* of pack (either the *encodePackFormat* or the *inputPackFormat* depending on the type), where `channel_format` is `channel` and `pack_formats` is `[pack]`.

– If `pack` is a direct or decode matrix, the matrix should be treated as having been previously applied to the samples in the file if an *audioObject* references both `pack` and a set of *audioTrackUID*s which in turn reference `pack` (or sub-packs) and channels of `pack`:

  • `root_pack` is `pack`.

  • `channels` contains one value per *audioChannelFormat* `channel` in `pack`, where `channel_format` is `channel` and `pack_formats` contains all *audioPackFormat*s on the path from `pack` to `channel`.

– If `pack` is a decode matrix, its *encodePackFormat* followed by `pack` may be applied if an *audioObject* references `pack` and a set of *audioTrackUID*s which in turn reference *encodePackFormat* and channels of the *inputPackFormat* of *encodePackFormat*:

  • `root_pack` is `pack`.

  • `channels` contains one value per audioChannelFormat `channel` in the inputPackFormat of the encodePackFormat of `pack`, where `channel_format` is `channel`, and `pack_formats` contains all audioPackFormats on the path from the inputPackFormat to `channel`.

The ‘type’ of a matrix *audioPackFormat* is determined using the following rules:

– If it has both an *inputPackFormat* and an *outputPackFormat* reference, it is a direct matrix.

– If it has an *inputPackFormat* reference and no *outputPackFormat* reference, it is an encode matrix.

– If it has an *outputPackFormat* reference and no *inputPackFormat* reference, it is a decode matrix.

– If it has neither an *inputPackFormat* or an *outputPackFormat* reference, an error is raised.

5.2.6.2 Tracks and *audioPackFormat* References to Match

The tracks to match against the `AllocationPack`s shall be specified by three values:

- `tracks`, a list of `AllocationTrack`s, each of which represents an *audioTrackUID* (or CHNA row):

```
class AllocationTrack {
  AudioChannelFormat channel_format;
  AudioPackFormat pack_format;
};
```

  `channel_format` is obtained from an *audioTrackUID* by following the *audioTrackFormat*, *audioStreamFormat* and *audioChannelFormat* references, while `pack_format` is referenced directly by the *audioTrackUID*.

- `pack_refs`, an optional list of *audioPackFormat* references found in an *audioObject*.

  `num_silent_tracks`, the number of 'silent' tracks to allocate, represented in the references from an *audioObject* to `ATU_00000000`.

When determining these structures for an *audioObject*:

- `tracks` contains one entry for each (non-silent) *audioTrackUID* referenced from the *audioObject*.
- `pack_refs` is a list of *audioPackFormat* references contained in the *audioObject*.
- `num_silent_tracks` is the number of silent *audioTrackUID*s referenced (corresponding to references to `ATU_00000000` in the *audioObject*).

while in CHNA-only mode:

- `tracks` contains one entry for each *audioTrackUID* (or CHNA row) in the file.
- `pack_refs` is `None`.
- `num_silent_tracks` is 0.

5.2.6.3 Matching

A match solution is specified as a list of `AllocatedPack` objects:

```
struct AllocatedPack {
  AllocationPack pack;
  vector<tuple<AllocationChannel,
               optional<AllocationTrack>>> allocation;
};
```

Each one associates each *audioChannelFormat* in `pack` with a `track`, or a silent track if the `AllocationTrack` is not specified.

A valid solution has the following properties:

1. For each `AllocatedPack`, each channel in the `AllocationPack` occurs exactly once in `allocation`.
2. Each track in `tracks` occurs exactly once in the output.
3. The number of silent tracks referenced in the output is equal to `num_silent_tracks`.
4. For each associated `AllocationChannel channel` and `AllocationTrack track`, `track.channel_format` is `channel.channel_format`, and `track.pack_format` is in `channel.pack_formats`.

5. If `pack_refs` is not `None`, then there is a one-to-one correspondence between `pack_refs` and the values of `pack.pack.root_pack` for each `AllocatedPack pack`.

Solutions which are the same except for the order of the `AllocationPacks` or the `allocations` within are considered to be equivalent.

Any method which can enumerate all valid and unique (non-equivalent) solutions may be used. In the reference implementation, solutions are found by treating the above properties as a constraint satisfaction problem and enumerating all solutions using a backtracking search.

5.2.6.3.1 Examples

Pack format matching is illustrated in a series of examples below.

First the structures used in the examples are defined. `c1`, `c2`, etc. and `p1`, `p2`, etc. represent references to *audioChannelFormat*s and *audioPackFormat*s (but may be any objects as `allocate_packs` only uses information in the `Allocation...` structures, comparing these references by identity).

A mono pack and a track referencing it:

```
ac1 = AllocationChannel(c1, [p1])
ap1 = AllocationPack(p1, [ac1])
at1 = AllocationTrack(c1, p1)
```

A two channel pack with two pairs of referencing tracks:

```
ac2 = AllocationChannel(c2, [p2])
ac3 = AllocationChannel(c3, [p2])
ap2 = AllocationPack(p2, [ac2, ac3])

at2 = AllocationTrack(c2, p2)
at3 = AllocationTrack(c3, p2)

at4 = AllocationTrack(c2, p2)
at5 = AllocationTrack(c3, p2)
```

Resolving a single mono track in an *audioObject* results in a single solution containing a single allocated pack:

```
assert allocate_packs(
    packs=[ap1, ap2],
    tracks=[at1],
    pack_refs=[p1],
    num_silent_tracks=0,
) == [[AllocatedPack(pack=ap1, allocation=[(ac1, at1)])]]
```

Resolving a single mono track in CHNA-only mode results in the same structure:

```
assert allocate_packs(
    packs=[ap1, ap2],
    tracks=[at1],
    pack_refs=None,
    num_silent_tracks=0,
) == [[AllocatedPack(pack=ap1, allocation=[(ac1, at1)])]]
```

Resolving a single silent track results in the same structure, except that the reference to the track is replaced by `None`:

```
assert allocate_packs(
    packs=[ap1, ap2],
    tracks=[],
```

```
        pack_refs=[p1],
        num_silent_tracks=1,
    ) == [[AllocatedPack(pack=ap1, allocation=[(ac1, None)])]]
```

If there are more tracks than channels available in the pack references then there will be no solutions because rule 2 conflicts with rule 5:

```
    assert allocate_packs(
        packs=[ap1, ap2],
        tracks=[at1],
        pack_refs=[],
        num_silent_tracks=0,
    ) == []
```

If there are more silent tracks than channels available in the pack references then there will be no solutions because rule 2 conflicts with rule 5:

```
    assert allocate_packs(
        packs=[ap1, ap2],
        tracks=[],
        pack_refs=[ap1],
        num_silent_tracks=2,
    ) == []
```

If there is a mismatch between the pack references and the channel/pack information in the tracks there will be no solutions because rules 1, 4 and 5 conflict:

```
    assert allocate_packs(
        packs=[ap1, ap2],
        tracks=[at1, at1],
        pack_refs=[p2],
        num_silent_tracks=0,
    ) == []
```

If there are multiple instances of a multi-channel pack in an *audioObject*, the assignment of tracks to packs is ambiguous so there are multiple solutions:

```
    assert allocate_packs(
        packs=[ap1, ap2],
        tracks=[at2, at3, at4, at5],
        pack_refs=[p2, p2],
        num_silent_tracks=0,
    ) == [
        [AllocatedPack(pack=ap2, allocation=[(ac2, at2), (ac3, at3)]),
         AllocatedPack(pack=ap2, allocation=[(ac2, at4), (ac3, at5)])],
        [AllocatedPack(pack=ap2, allocation=[(ac2, at2), (ac3, at5)]),
         AllocatedPack(pack=ap2, allocation=[(ac2, at4), (ac3, at3)])],
    ]
```

5.2.6.4 Solution Post-Processing

It should be noted that the results of matching are specified in terms of the input structures (`AllocationPack`, `AllocationChannel`, `AllocationTrack`), rather than the underlying references to ADM structures. This is to allow arbitrary mapping between the *audioPackFormat* and *audioChannelFormat* references (in the *audioObject* and *audioTrackUID*) and the information provided to the renderer, as there is no simple correspondence when the *typeDefinition==Matrix* is used.

For a non-matrix `AllocatedPack pack`, the mapping is straightforward. `output_pack` is `pack.pack.root_pack`, and there is a one-to-one mapping between the allocations in `pack.allocation` and the real channel allocation: `AllocationChannel channel` is mapped to `channel.channel_format`, `AllocationTrack track` is mapped to a `DirectTrackSpec` for the track index of the *audioTrackUID* (or CHNA row) associated with `track`, and a missing `AllocationTrack` is mapped to a `SilentTrackSpec`.

For a matrix `AllocatedPack pack`, a more complex mapping is required:

`pack.root_pack` is always a decode or direct pack (see § 5.2.6.1.1), so `output_pack` is `pack.root_pack.outputPackFormat`.

The output channel to track allocation contains one entry per *audioChannelFormat* `matrix_channel` in `root_pack`. These channels have a one-to-one correspondence with the *audioChannelFormats* in `output_pack` established by *outputChannelFormat* references.

The *audioChannelFormat* is matrix_channel.block_formats[0].outputChannelFormat.

The `TrackSpec` is built by recursively following the *inputChannelFormat* references from `matrix_channel` to *audioChannelFormat*s referenced in `pack.allocation`, nesting `MatrixCoefficientTrackSpec`s and `MixTrackSpec`s to apply the processing specified in *coefficient* elements and mix multiple input channels together:

– If `matrix_channel` is referenced in `pack.allocation`, return a `DirectTrackSpec` or `SilentTrackSpec` corresponding with the associated `AllocationTrack` (see above).

– Otherwise, return a `MixTrackSpec` containing one `MatrixCoefficientTrackSpec` for each *coefficient* element `c` in `matrix_channel.block_formats[0].matrix` which applies the processing specified in `c` to the track spec for `c.inputChannelFormat`, determined recursively.

In the reference implementation this is implemented in two sub-classes of `AllocationPack`, which have methods to query the *audioPackFormat* and channel allocation for use by the renderer. The association between `AllocationTrack`s and their corresponding *audioTrackUID*s is likewise maintained using a sub-class of `AllocationTrack`.

5.2.7 Output Rendering Items

Once the root *audioPackFormat* has been determined, and a `TrackSpec` has been assigned to each of its channels, all the information found is translated into one or more `RenderingItem`s.

The process for doing this depends on the type of the root *audioPackFormat*.

5.2.7.1 Shared Components

Some data in rendering items are shared between types, and are therefore derived in the same way too.

5.2.7.1.1 Importance

An `ImportanceData` object should be derived from the item selection state, with the following values:

– `audio_object` is the minimum importance specified in all *audioObject*s in the path.

– `audio_pack_format` is the minimum importance specified in any *audioPackFormat* along the path from the root *audioPackFormat* to the *audioChannelFormat*.

In both cases `None` (importance not specified) is defined as being the highest importance.

5.2.7.1.2 Extra Data

An `ExtraData` object should be derived from the item selection state, with the following values:

- `object_start` is the *start* time of the last *audioObject* in the path (`None` in CHNA-only mode).
- `object_duration` is the *duration* of the last *audioObject* in the path (`None` in CHNA-only mode).
- `reference_screen` is the *audioProgrammeReferenceScreen* of the selected *audioProgramme* (`None` if none is selected).
- `channel_frequency` is the *frequency* element of the selected *audioChannelFormat* (or `None` if one has not been selected, as when creating a HOA rendering item).

5.2.7.2 Output Rendering Items for typeDefinition==Objects or DirectSpeakers

The process for determining rendering items for *Objects* and *DirectSpeakers* is similar – only the types involved and the selection of parameters differ.

One rendering item is produced per *audioChannelFormat* and `track_spec` pair in the channel allocation.

A `MetadataSource` is created which produces one `RenderingItem` (of the appropriate type) per *audioBlockFormat* in the selected *audioChannelFormat*, where the `extra_data` field is determined as above, and the `audioPackFormats` field contains all *audioPackFormat*s on the path between the root *audioPackFormat* and the *audioChannelFormat*. This is wrapped in a `RenderingItem` object (again, of the appropriate type) with the `track_spec` and `importance` determined as above.

5.2.7.3 Output Rendering Items for *typeDefinition==HOA*

One `HOARenderingItem` is produced per pack allocation, containing all the information required to render a group of channels which make up a HOA stream. This information is spread across multiple *audioChannelFormats* and *audioPackFormats* (when nested), which must be consistent.

HOA *audioChannelFormat*s must only contain a single *audioBlockFormat* element; an error is raised otherwise.

A single N`HOATypeMetadata` object is created with parameters derived according to Table 1.

TABLE 1

Properties of `HOATypeMetadata` parameters

`HOATypeMetadata` parameter	*audioBlockFormat* parameter	*audioPackFormat* parameter	count
`rtime`	*rtime*		single
`duration`	*duration*		single
`orders`	*order*		per-channel
`degrees`	*order*		per-channel
`normalization`	*normalization*	*normalization*	single
`nfcRefDist`	*nfcRefDist*	*nfcRefDist*	single
`screenRef`	*screenRef*	*screenRef*	single

All parameters shall be first determined for each *audioChannelFormat* in the root *audioPackFormat*. For parameters which have both *audioBlockFormat* and *audioPackFormat* parameters, the parameter

may be set on the sole *audioBlockFormat* in the *audioChannelFormat*, or any *audioPackFormat* on the path from the root *audioPackFormat* to the *audioChannelFormat*. If multiple copies of a parameter are found for a given *audioChannelFormat* they shall have the same value, otherwise an error shall be raised. If no values for a given parameter and *audioChannelFormat* are found, then the default specified in Recommendation ITU-R BS.2076-1 is applied.

After *nfcRefDist* has been found for a particular *audioChannelFormat*, a value of 0 shall be translated to `None`, which implies that NFC shall not be applied. This is performed at this stage (rather than during XML parsing) so that *nfcRefDist==0.0* is considered to conflict with *nfcRefDist==1.0*, for example.

For parameters which have only a single value (all except `orders` and `degrees`), the parameters determined for all *audioChannelFormats* shall be equal, otherwise an error shall be raised.

`extra_data` is determined as above for the whole *audioPackFormat*.

A `HOARenderingItem` shall be produced with one entry in `track_specs` and `importances` per item in the channel allocation (as described above), and a `MetadataSource` containing only the above `HOATypeMetadata` object.

5.3 Rendering Item Processing

Some renderer functionality is implemented by modifying the list of selected rendering items. Section 5.3.1 describes how content can be removed based on the specified importance level, and § 5.3.3 describes how the effects of downstream metadata conversion may be emulated.

5.3.1 Importance emulation

The *importance* parameters as defined by Recommendation ITU-R BS.2076-1 allows 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 should 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 the renderer 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`.

5.3.1.1 Importance values of `RenderingItems`

Each rendering item can 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 a `RenderingItem` shall 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 a `RenderingItem` shall be used as the *audioPackFormat importance* for this `RenderingItem`.

– An *audioObject* without *importance* value shall not be taken into account when determining the *importance* of a `RenderingItem`.

– An *audioPackFormat* without *importance* value shall not be taken into account when determining the *importance* of a `RenderingItem`.

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

5.3.1.2 Static importance handling

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

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

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

5.3.1.3 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 discarding 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`.

5.3.2 Conversion Emulation

Emulation of metadata conversion may optionally be applied to rendering items. Conversion emulation may be disabled, set to convert metadata to polar form, or set to convert metadata to Cartesian form.

If conversion emulation is enabled, the appropriate function is selected from § 10 and applied to all *audioBlockFormats* with *typeDefinition==Objects* in the selected rendering items.

6 Shared Renderer Components

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

6.1 Polar 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/digital signal 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 *DirectSpeakers* renderer, and as part of the HOA decoder design process.

The point source panner in this renderer is based on the VBAP formulation [2], 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 down-mixing 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 shall be responsible for producing loudspeaker gains over a given spatial extent.

In order to produce gains for a given direction, the point source panner shall query each region in turn, which shall either return a gain vector if it can handle that direction, or a null result if it cannot; 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 shall be initialised with the 3D positions of three loudspeakers:

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

The three output gains **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 shall be 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 loudspeaker 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:

$$\mathbf{g}'' = \frac{\mathbf{g}'}{\|\mathbf{g}'\|_2}$$

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

6.1.2.3 QuadRegion

This represents a spherical quadrilateral region formed by four 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 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) to (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`.

6.1.2.3.1 Formulation

Given the Cartesian position of four 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:

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

$$\mathbf{g} = \frac{\mathbf{g}'}{\|\mathbf{g}'\|_2} \tag{2}$$

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:

$$\mathbf{g} \cdot \mathbf{P} = r\mathbf{d} \tag{3}$$

for some $r > 0$.

6.1.2.3.2 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:

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

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 **P** by **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 **g** can then be calculated using equations 1 and 2. Since the scale of **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

The output signals of a point source for stereo (0+2+0) are provided by a method based on a downmix from 0+5+0 to 0+2+0. The method is separately implemented.

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 five gains, **g′**, in the order M+030, M-030, M+000, M+110, M-110.

– A format conversion matrix from 0+5+0 to 0+2+0 is applied to produce stereo gains **G″** in the order M+030, M-030:

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

– Power normalise **g″** to a value determined by the balance between the front and rear loudspeakers in **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.

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

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

NOTE – **g** from (0+5+0) to (0+2+0) is completely matched with downmix coefficients specified in Recommendation ITU-R BS.775 as follows:

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

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 § 11.1.3), 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 loudspeaker layouts, the following process is used:

1. Update the azimuth of the nominal positions of loudspeakers with label `M+SC` or `M−SC` to ensure correct triangulation with widely-spaced screen loudspeakers. If the real azimuth (`polar_position.azimuth`) is φ, the nominal azimuth φ_n (`polar_nominal_position.azimuth`) is:

$$\varphi_n = \mathrm{sgn}(\varphi) \times \begin{cases} 45 & |\varphi| > 30 \\ 15 & \text{otherwise} \end{cases}$$

2. 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.
3. 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 Recommendation ITU-R BS.2051-2, whereas the real loudspeaker positions are positions which are actually used by the current reproduction system.
4. 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 Recommendation ITU-R BS.2051-2 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 defined in Recommendation ITU-R BS.2051-2, the position may coincide with that of the virtual loudspeaker, creating a step change in the panning function.
5. 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.
6. 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 defined in Recommendation ITU-R BS.2051-2 result in facets with more than four edges.

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

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

6.1.3.1.1 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 shall 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 loudspeakers 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:

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

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

– 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:

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

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

– For each j in S_m:

• Create a virtual upper loudspeaker if $\varphi_{j,n} \geq L_u$, 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

An `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 *DirectSpeakers* components in §§ 7.3.12.1 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:

$$-180 \le \texttt{start} \le \texttt{end} \le 180$$

This function is equivalent to:

$$\texttt{start} - \texttt{tol} \le x' \le \texttt{end} + \texttt{tol}$$

Where $x' = x + 360 \times i$ for some i such that $-180 < x' \le 180$.

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

$$x' \le -90 \vee x' \ge 90$$

Some example ranges and equivalent expressions are shown in Table 2.

TABLE 2

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	$x' \le -90 \vee x' \ge 90$
90	−90	5	$x' \le -85 \vee x' \ge 85$
0	0	0	$x' = 0$
180	180	0	$x' = 180$
−180	−180	0	$x' = 180$
180	180	5	$x' \le -175 \vee x' \ge 175$
−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

$$\texttt{frequency.lowPass} \wedge \neg\texttt{frequency.highPass} \wedge (\texttt{frequency.lowPass} \le 120\ \text{Hz})$$

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`.

NOTE – Recommendation ITU-R BS.2127-0 had the frequency to select the LFE channels set to 200 Hz or less.

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

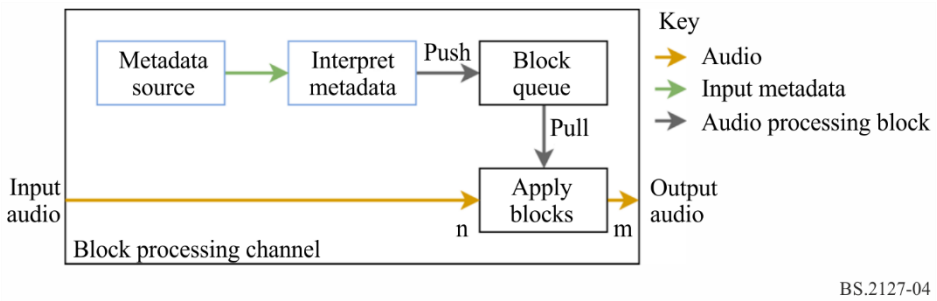


Figure 4 shows the structure used to achieve this. The interface to this component is as follows:

```
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 `input_samples`$[i]$ and `output_samples`$[i]$ represent the global input and output samples `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`:

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

`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 shall not be 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 handing strategies based on their target application environment.

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

6.6 Interpretation of `TrackSpecs`

The audio input to the renderer is through a multi-channel bus directly read from the input file. The input metadata in the form of `RenderingItem`s includes `TrackSpec` objects, which are instructions for extracting channels from this bus, including applying Matrix preprocessing which mixes together multiple channels.

The processing for each `TrackSpec` type is implemented in `core.track_processor`.

Given a `TrackSpec`, a `TrackProcessor` object can be created, which has a single method `process(sample_rate, input_samples)`, which applies the specified processing to `input_samples` and returns the single-channel result (at the given sample rate).

6.6.1 SilentTrackSpec

For n input samples, `process` for a `SilentTrackSpec` returns n zero-valued samples.

6.6.2 DirectTrackSpec

`process` for a `DirectTrackSpec track_spec` returns the input samples in the track specified in `track_spec.track_index` (using zero-based indexing).

6.6.3 MixTrackSpec

`process` for a `MixTrackSpec track_spec` returns the sum of the results of calling `process` on a `TrackProcessor` for each sub-track in `track_spec.input_tracks`.

6.6.4 MatrixCoefficientTrackSpec

`process` for a `MatrixCoefficientTrackSpec track_spec` applies the matrix processing specified in `track_spec.coefficient` (which represents the parameters of a single matrix *coefficient* element) to a single channel specified by `track_spec.input_track`.

If `track_spec.coefficient.gain` is not `None`, the samples are multiplied by `gain`.

If `track_spec.coefficient.delay` is not `None`, the samples are delayed by n samples, `delay` msec, rounded to the nearest sample (with ties broken towards 0):

$$n = \left\lceil \frac{\texttt{sample\_rate} \times \texttt{delay}}{1000} - \frac{1}{2} \right\rceil$$

Some parameters are not supported. If `gainVar`, `delayVar`, `phaseVar` or `phase` are not `None`, or `delay` is negative, an error is raised.

6.7 Relative angle

$\texttt{relative\_angle}(x, y)$ is used to find an equivalent angle to y which is greater than or equal to x. This is used to avoid edge-cases when working with circular arcs.

$\texttt{relative\_angle}(x, y)$ returns $y' = y + 360n$, where n is the smallest integer such that $y' \geq x$

6.8 Coordinate transformations

The `cart` function is defined to translate from polar positions to Cartesian positions according to § 2.2:

$$cart(\varphi, \theta, d) = \{x, y, z\}$$

where:

$$\begin{aligned} x &= \sin\left(-\frac{\pi}{180}\varphi\right)\cos\left(\frac{\pi}{180}\theta\right)d \\ y &= \cos\left(-\frac{\pi}{180}\varphi\right)\cos\left(\frac{\pi}{180}\theta\right)d \\ z &= \sin\left(\frac{\pi}{180}\theta\right)d \end{aligned}$$

The inverse transformations to extract the azimuth and elevation from a Cartesian position are also defined:

$$\begin{aligned} \texttt{azimuth}(\{x, y, z\}) &= -\frac{180}{\pi}\text{atan2}(x, y) \\ \texttt{elevation}(\{x, y, z\}) &= \frac{180}{\pi}\text{atan2}\left(z, \sqrt{x^2 + y^2}\right) \end{aligned}$$

The function `local_coordinate_system` produces a rotation matrix which maps {0,1,0} to a given azimuth and elevation:

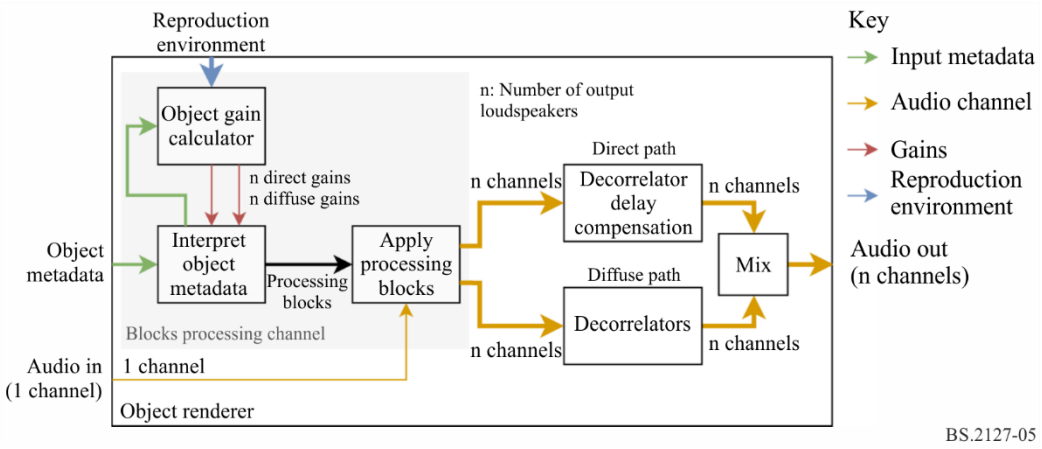
$$\texttt{local\_coordinate\_system}(\varphi, \theta) = \begin{bmatrix} \texttt{cart}(\varphi - 90, 0, 1) \\ \texttt{cart}(\varphi, \theta, 1) \\ \texttt{cart}(\varphi, \theta + 90, 1) \end{bmatrix}$$

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 Fig. 5. This Figure 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.

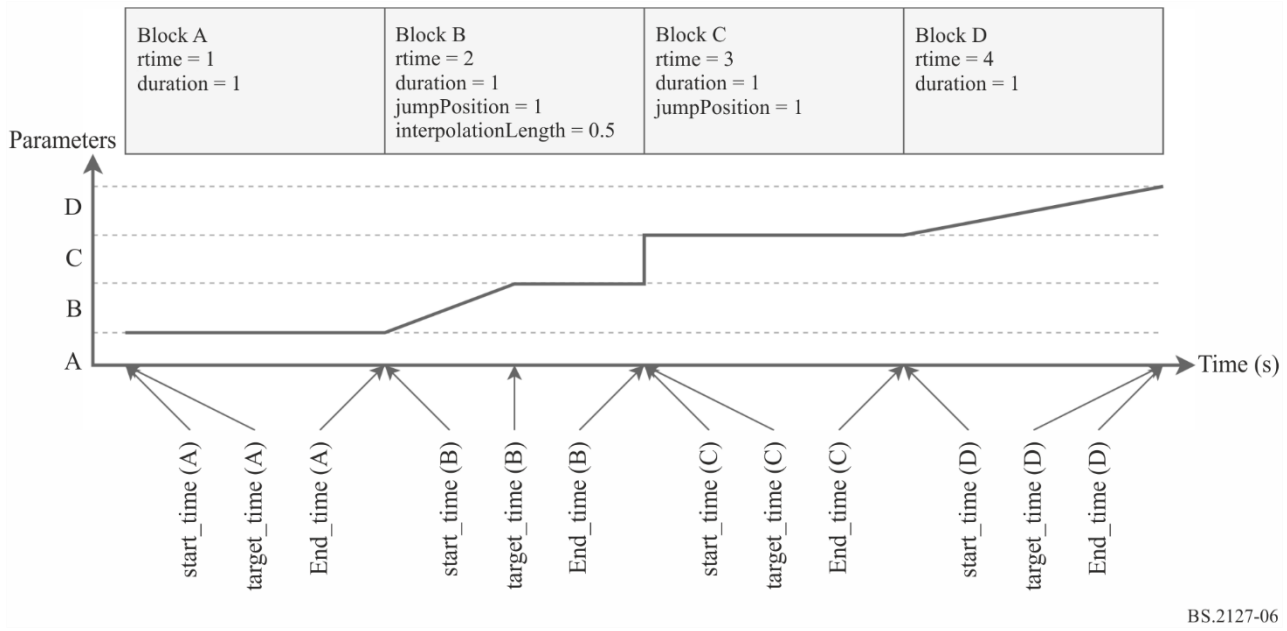
This structure is implemented in `core.objectbased.renderer.ObjectRenderer`.

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 Fig. 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

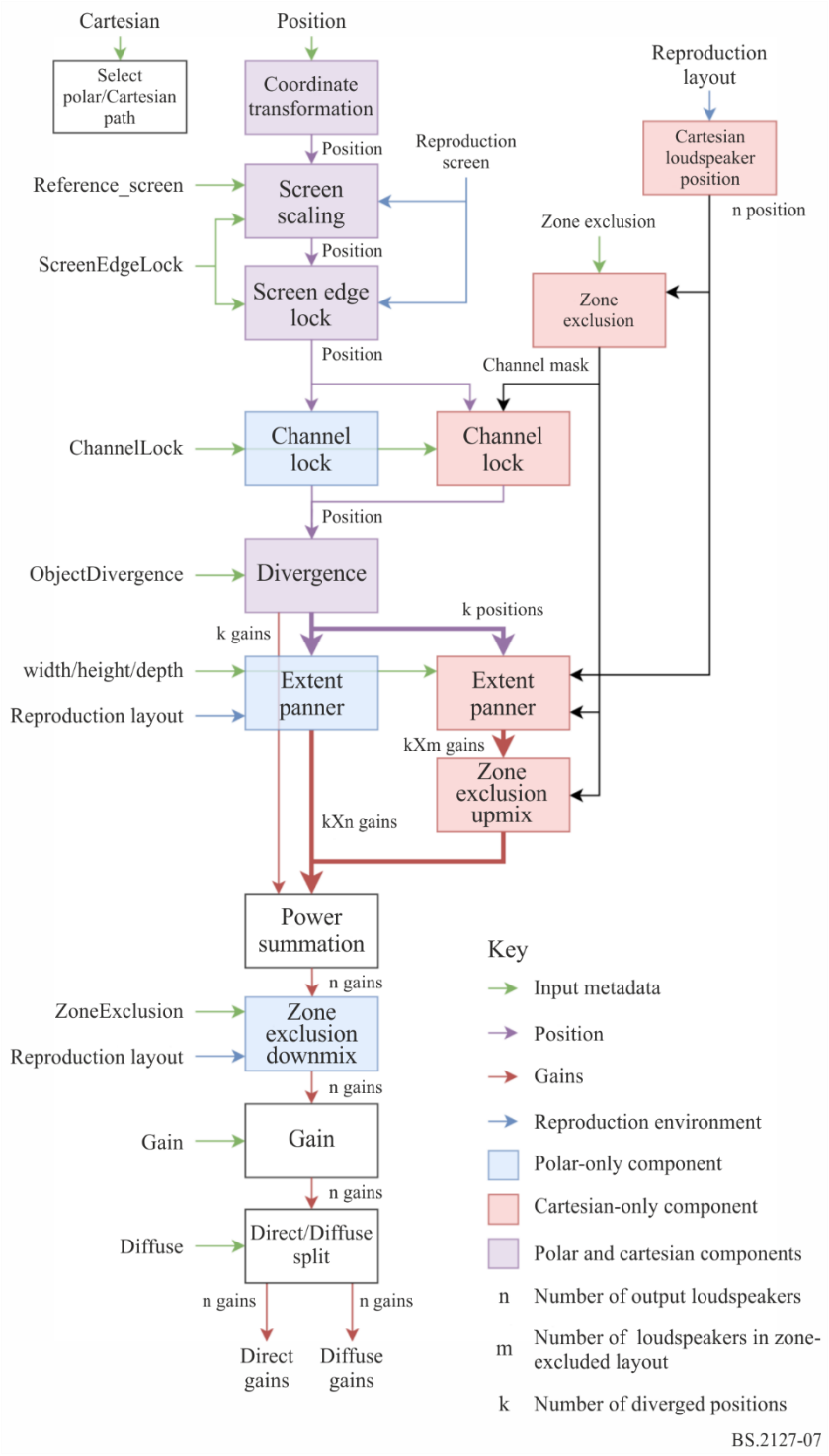
Given an `ObjectTypeMetadata` object, this object calculates a gain for each loudspeaker on the direct and diffuse paths. The interface to this component is:

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

class GainCalc {
    GainCalc(Layout layout);

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

7.3.1 Structure

FIGURE 7

Structure of the gain calculator for *typeDefintion==Objects*



This component is primarily a composite of the sub-components listed in this section. A diagram of the signal flow between these components is shown in Fig. 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` to yield a `CartesianPosition` 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` and `bf.cartesian`, updating `position`. This component is initialised with the reproduction screen (`layout.screen`) and the reproduction layout (`layout`).

– Screen edge lock is applied using the method described in § 7.3.4, with the parameters `position`, `bf.position.screenEdgeLock` and `bf.cartesian`, updating the `position` with the result. This component is initialised with the reproduction screen (`layout.screen`) and the reproduction layout (`layout`).

– If `bf.cartesian`, then:

  • The allocentric position of each loudspeaker in `layout.without_lfe` is determined according to § 7.3.9, yielding the `allo_channel_positions` array.

  • The zone exclusion algorithm described in § 7.3.5 is applied to `allo_channel_positions` and `bf.zone_exclusion`, yielding a boolean mask of loudspeakers to exclude, `excluded`.

  • Channel lock in the allocentric configuration described in § 7.3.6 is applied with the parameters `position`, `bf.channelLock` and `excluded`, updating the `position`.

  otherwise:

  • Channel lock in the egocentric configuration described in § 7.3.6 is applied with the parameters `position` and `bf.channelLock`, updating the `position`.

– Divergence is applied using the method described in § 7.3.7, with the parameters `position`, `bf.objectDivergence` and `bf.cartesian`. This results in up to three extended sources with gains and positions stored in `diverged_gains` and `diverged_positions`.

– If `bf.cartesian`, then:

  • The extent panner described in § 7.3.11 is applied to each `p` in `diverged_positions`, with parameters `channel_positions`, `p`, `bf.width`, `bf.height`, `bf.depth` resulting in gain vectors for the non-excluded loudspeaker array. `channel_positions` is a list of non-excluded channel positions selected from `allo_channel_positions[i]` where `excluded[i]` is `False`.

  • These gain vectors are upmixed according to `excluded`, resulting in a gain for each loudspeaker `i` where `excluded[i]` is `False`, and a zero where `excluded[i]` is `True`, and stored in `gains_for_each_pos`.

  otherwise:

  • The extent panner described in § 7.3.8 is applied to each `p` in `diverged_positions`, with parameters `p`, `bf.width`, `bf.height`, `bf.depth` resulting 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`:

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

– If `bf.cartesian` is not set, zone exclusion as described in § 7.3.12 is applied to `gains` and `bf.zoneExclusion`, resulting in a new `gains` vector. This component is initialised with `layout.without_lfe`.

– `gains` is extended by adding LFE channel gains with value 0 to produce `gains_full`, with one value per loudspeaker in `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 (Informative)

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), it is desirable to preserve the obvious interpretation of those parameters.

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

```
CartesianPosition coord_trans(ObjectPosition position);
```

`position` is first converted to a Cartesian vector **p**.

If `position` is an `ObjectCartesianPosition` then the elements of **p** are clipped to the range $[-1,1]$ before being returned:

$$\texttt{clip}(\mathbf{p}, -1, 1)$$

otherwise **p** is returned unmodified.

`clip` is defined for real numbers as:

$$\texttt{clip}(x, a, b) = \begin{cases} a & x \le a \\ x & a \le x \le b \\ b & b \le x \end{cases}$$

and is trivially applied to each element in a vector:

$$\texttt{clip}(\{x, y, z\}, a, b) = \{\texttt{clip}(x, a, b), \texttt{clip}(y, a, b), \texttt{clip}(z, a, b)\}$$

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,
    bool cartesian
  );
};
```

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;
};
```

A `PolarEdges` object is created from a given `PolarScreen` or `CartesianScreen` object by first transforming the screen into a Cartesian centre position and two vectors (along the x and z directions) which define the surface of the screen, then finding the azimuth and elevation of each of the edges.

For a `PolarScreen screen`, where:

$$\begin{aligned} \varphi &= \texttt{screen.centrePosition.azimuth} \\ \theta &= \texttt{screen.centrePosition.elevation} \\ d &= \texttt{screen.centrePosition.distance} \\ w &= \texttt{screen.widthAzimuth} \\ a &= \texttt{screen.aspectRatio} \end{aligned}$$

the following procedure is used:

– The centre position is a simple Cartesian conversion of the centre position:

$$centre = cart(\varphi, \theta, d)$$

– A Cartesian width and height are calculated:

$$\begin{aligned} \texttt{width} &= d \cdot \tan\left(\frac{\pi}{180}\frac{w}{2}\right) \\ \texttt{height} &= \frac{\texttt{width}}{a} \end{aligned}$$

– `local_coordinate_system` is used to find the screen x and z vectors:

$$\begin{aligned} \begin{bmatrix} l_x \\ l_y \\ l_z \end{bmatrix} &= \texttt{local\_coordinate\_system}(\varphi, \theta) \\ v_x &= \texttt{width} \times l_x \\ v_z &= \texttt{height} \times l_z \end{aligned}$$

For a `CartesianScreen screen`, where:

$$\begin{aligned} w &= \texttt{screen.widthX} \\ a &= \texttt{screen.aspectRatio} \end{aligned}$$

the following procedure is used:

– The centre position is used directly:

$$centre = screen.centrePosition$$

– The width and height are computed:

$$\begin{aligned} \texttt{width} &= \frac{w}{2} \\ \texttt{height} &= \frac{\texttt{width}}{a} \end{aligned}$$

– The screen x and z vectors are defined:

$$\begin{aligned} v_x &= \{\texttt{width}, 0, 0\} \\ v_z &= \{0, 0, \texttt{height}\} \end{aligned}$$

For both screen types, a `PolarEdges` object can then be constructed with:

$$\begin{aligned} \texttt{left\_azimuth} &= \texttt{azimuth}(\texttt{centre} - v_x) \\ \texttt{right\_azimuth} &= \texttt{azimuth}(\texttt{centre} + v_x) \\ \texttt{bottom\_elevation} &= \texttt{elevation}(\texttt{centre} - v_z) \\ \texttt{top\_elevation} &= \texttt{elevation}(\texttt{centre} + v_z) \end{aligned}$$

7.3.3.2 Position Compensation

In some output layouts when `cartesian==true`, vertical panning in front of the listener may be warped. This is compensated for using the `core.screen_common.compensate_position` function:

$$\texttt{compensate\_position}(\varphi, \theta, \texttt{layout}) = \begin{cases} \{\varphi', \theta\} & \text{"U+045"} \in \texttt{layout.channel\_names} \\ \{\varphi, \theta\} & \text{otherwise} \end{cases}$$

where:

– φ_r is formed by piecewise linear interpolation of θ from:

$$\{-90,0,30,90\}$$

to:

$$\left\{30,30,30\frac{30}{45},30\right\}$$

– φ' is formed by piecewise linear interpolation of φ from:

$$\{-180,-30,30,180\}$$

to:

$$\{-180,-\varphi_r,\varphi_r,180\}$$

7.3.3.3 Direction Warping

The warping of positions is defined in `core.screen_scale.PolarScreenScaler.scale_az_el`, which independently warps an azimuth and elevation value. Given the `PolarEdges ref` of the te reference screen and the `PolarEdges rep` of the reproduction screen, this works as follows:

– Piecewise linear interpolation is applied to the azimuth, mapping from the values

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

to

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

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

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

to

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

This is wrapped in `core.screen_scale.PolarScreenScaler.scale_position`, which applies `scale_az_el` to the azimuth and elevation components of a Cartesian vector, leaving the distance unchanged.

7.3.3.4 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.

If `screenRef` is not set or no reproduction screen is provided, the position is returned unmodified. Otherwise the behaviour depends on the `cartesian` flag:

– If `cartesian` is set, polar scaling and compensation is applied by using the conversion described in § 10.1, resulting in a new position $\{x',y',z'\}$:

$$\begin{aligned}\{\varphi,\theta,d\} &= \texttt{point\_cart\_to\_polar}(\texttt{position.x},\texttt{position.y},\texttt{position.z})\\ \{\varphi_s,\theta_s\} &= \texttt{scale\_az\_el}(\varphi,\theta)\\ \{\varphi_{sc},\theta_{sc}\} &= \texttt{compensate\_position}(\varphi_s,\theta_s,\texttt{layout})\\ \{x',y',z'\} &= \texttt{point\_cart\_to\_polar}(\varphi_{sc},\theta_{sc},d)\end{aligned}$$

– Otherwise, `scale_az_el` is applied to the azimuth and elevation components of the position.

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,
    cartesian=False
  );

  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.

`handle_az_el` independently modifies the azimuth and elevation, resulting in a new azimuth and elevation:

– 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.

If `reproduction_screen` is not provided, no position modification occurs.

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.

– If `cartesian` is set, polar scaling and compensation is applied by using the conversion described in § 10.1, resulting in a new position $\{x', y', z'\}$:

$$\begin{aligned} \{\varphi, \theta, d\} &= \texttt{point\_cart\_to\_polar}(\texttt{position.x}, \texttt{position.y}, \texttt{position.z}) \\ \{\varphi_s, \theta_s\} &= \texttt{handle\_az\_el}(\varphi, \theta, \texttt{screen\_edge\_lock}) \\ \{\varphi_{sc}, \theta_{sc}\} &= \texttt{compensate\_position}(\varphi_s, \theta_s, \texttt{layout}) \\ \{x', y', z'\} &= \texttt{point\_cart\_to\_polar}(\varphi_{sc}, \theta_{sc}, d) \end{aligned}$$

– Otherwise, `handle_az_el` is applied to the azimuth and elevation components of the position.

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

7.3.5 Cartesian Zone Exclusion

The Cartesian zone exclusion algorithm begins with the full reproduction layout as `channel_positions` and processes the `ExclusionZone` objects to identify which loudspeakers should be removed – this follows the algorithm set out in § 7.3.12.1. For each

loudspeaker found to be within any of the regions specified by an `ExclusionZone` object, that loudspeaker is removed and if this results in reducing a row of loudspeakers (which share the same y and z coordinates) to a single loudspeaker, then all loudspeakers from the row are removed so that the basic properties required by the point source panner in § 7.3.10 are maintained.

If the process of applying zone exclusion would result in all loudspeakers being removed, then no loudspeakers are removed.

Finally an upmix matrix is created which maps channels in the reduced layout to their original channel in the full layout with unity gain.

This is implemented in `core.allocentric.apply_zone_exclusion`.

7.3.6 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.

The `objectbased._gain_calc.ChannelLockHandlerBase` with the following signature:

```
class ChannelLockHandlerBase {
  ChannelLockHandlerBase(Layout layout);
  CartesianPosition handle(
    CartesianPosition position,
    optional<ChannelLock> channelLock,
    vector<bool> excluded,
  );
};
```

`excluded` is a channel exclusion mask to indicate which loudspeakers should be ignored and is only used in the allocentric path, as only there the channel lock is performed after the zone exclusion.

For the egocentric path the `ChannelLockHandlerBase` is configured in `core.objectbased._gain_calc.EgoChannelLockHandler`.
For the allocentric path the `ChannelLockHandlerBase` is configured in `core.objectbased._gain_calc.AlloChannelLockHandler`.

In egocentric mode, the loudspeaker positions considered are the normalised real loudspeaker positions in `layout`, while in allocentric mode they are the positions according to `core.allocentric.positions_for_layout(layout)`, as described in § 7.3.9.

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

– If `excluded` is not `None`, don't consider the loudspeakers, where `excluded[n] == True` (`n` being the n-th loudspeaker) in the following steps.

– If `channelLock` is `None`, return the original `position`.

– If `channelLock.maxDistance` is not `None` calculate the ℓ_2 distance between each loudspeaker position and `position`, and identify all loudspeakers (within some tolerance), where the distance is smaller than `channelLock.maxDistance` as possible loudspeakers.

– If no possible loudspeaker is identified return `position`.

– In the set of possible loudspeakers identify those loudspeakers closest to `position`. In the egocentric configuration the ℓ_2 distance between `position` and each loudspeaker is used and in the allocentric configuration the weighted distance between `position` and each loudspeaker is used. The weighted distance is calculated as

$$dw_i = \sqrt{w_x \times \left(x_o - x_{spkr_i}\right)^2 + w_y \times \left(y_o - y_{spkr_i}\right)^2 + w_z \times \left(z_o - z_{spkr_i}\right)^2}$$

where:

$$w_x = \frac{1}{16}$$
$$w_y = 4$$
$$w_z = 32$$

– 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.

– The position of the chosen loudspeaker is returned.

7.3.7 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 $\mathbf{p}_c$. 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.

7.3.7.1 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:

– $\forall x,\ g_l + g_r + g_c = 1$

– $x = 0 \Longrightarrow g_l = g_r = 0 \wedge g_c = 1$

– $x = \frac{1}{2} \Longrightarrow g_l = g_r = g_c = \frac{1}{3}$

– $x = 1 \Longrightarrow g_l = g_r = 0.5 \wedge g_c = 0$

7.3.7.2 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.

7.3.7.2.1 Behaviour when `cartesian == true`

For a `position` value $\mathbf{p}$ and `objectDivergence.positionRange` value x, the centre position is simply shifted left and right by x along the x axis, and clipped to $[-1,1]$:

$$\begin{aligned} \mathbf{p}_c &= \mathtt{clip}(\mathbf{p}, -1, 1) \\ \mathbf{p}_l &= \mathtt{clip}(\mathbf{p} - \{x, 0, 0\}, -1, 1) \\ \mathbf{p}_r &= \mathtt{clip}(\mathbf{p} + \{x, 0, 0\}, -1, 1) \end{aligned}$$

`clip` is defined in § 7.3.2.

7.3.7.2.2 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 $\mathbf{p}_c$:

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

7.3.8 Polar extent panner

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

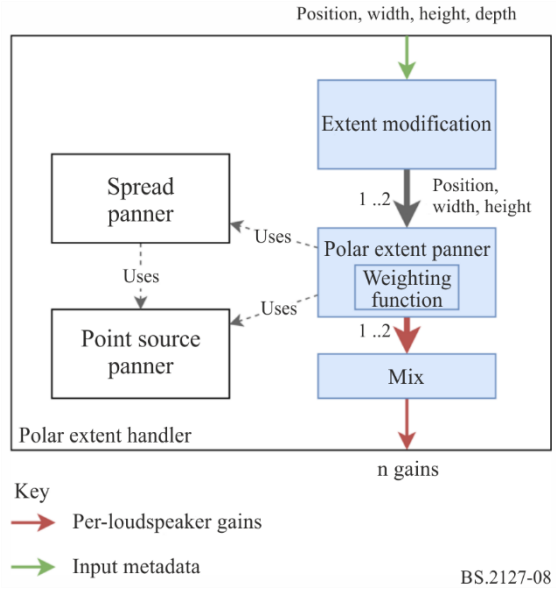
The interface to this class is:

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

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

The structure of the `PolarExtentHandler` class is shown in Fig. 8.

FIGURE 8

Structure of the Extent Handler



Internally, this object holds a reference to a `PolarExtentPanner` as described in § 7.3.8.2, which it uses to calculate the gain vectors.

The `width`, `height` and `position` parameters shall be 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 § 7.3.8.2.

Polar extent rendering modes use the Spreading Panner to generate loudspeaker gains, as described below.

7.3.8.1 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 Fig. 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.

The set of 1652 virtual source positions used in the spreading panner is determined as follows.

For each elevation θ between −90° and 90° inclusive in 5° steps, calculate the number of points n to be spaced evenly around a circle at that elevation, to achieve approximately uniform density on the surface of the unit sphere:

$$n' = \frac{360}{5}\cos\theta$$

$$n = \max(\mathrm{round}(n'),\ 1)$$

Then, for each i in range 0 to $n - 1$ inclusive, calculate the azimuth φ:

$$\varphi = 360\,\frac{i}{n}$$

This results in the point cart (φ, θ, 1).

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 calculated 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`.

7.3.8.2 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:

$$d_1 = \max\left\{0, \| \texttt{position} \|_2 + \frac{\texttt{depth}}{2}\right\}$$
$$d_2 = \max\left\{0, \| \texttt{position} \|_2 - \frac{\texttt{depth}}{2}\right\}$$

– 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:

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

7.3.8.2.1 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 $\texttt{distance} = 0$, the extent is always 360°.

– At $\texttt{distance} = 1$, the original extent is used.

– At $\texttt{distance} > 1$, the extent decreases as the distance increases.

– When $0 < \texttt{distance} < 1$, the extent changes more steeply around $\texttt{distance} = 0$ 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:

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

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

– 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 `distance`:

$$e_1 = 4 \times \frac{180}{\pi} \times \text{atan2}(\texttt{size}, 1)$$
$$e_d = 4 \times \frac{180}{\pi} \times \text{atan2}(\texttt{size}, \texttt{distance})$$

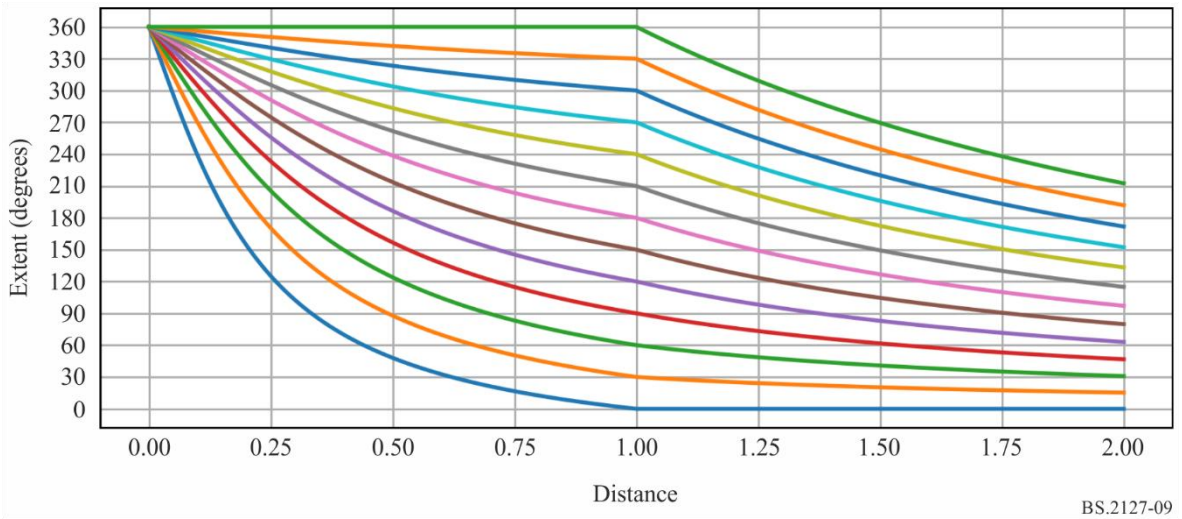
– Piecewise linear interpolation is applied to map e_d back to the original extent when $e_d = e_1$:

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

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

FIGURE 9

Extent modification function for polar extended sources



NOTE – 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.

7.3.8.2.2 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 $max\{width, 5°\}$ and $max\{height, 5°\}$; these are used with the spreading panner described in § 7.3.8.1 and the Polar Weighting Function described below to yield a spread gain vector g_s.

– The position is passed to the point source panner to yield a point source gain vector g_p.

The two vectors are mixed together to produce the vector g such that for zero width and height, the point source gains are used exclusively, while if either the width or height is greater than 5 degrees, the spread gains are used exclusively:

$$g_i = \sqrt{p g_{s,i}^2 + (1-p) g_{p,i}^2}$$

where:

$$p = clip\left(\frac{max(width,height)}{5}, 0,1\right)$$

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`.

7.3.8.2.3 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, Figs 10 and 11, which have the same shape but are rotated 90 degrees (ignoring the warping caused by the projection used).

– The approximate weighting function is now 1 inside a maximally-rounded `width` × `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 `width` = `height`, the weighting function is circular.

  • At `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 Figs 12 and 13.

  • 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 degrees.

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

FIGURE 10

Polar weighting function for `width` = 90° and `height` = 30°

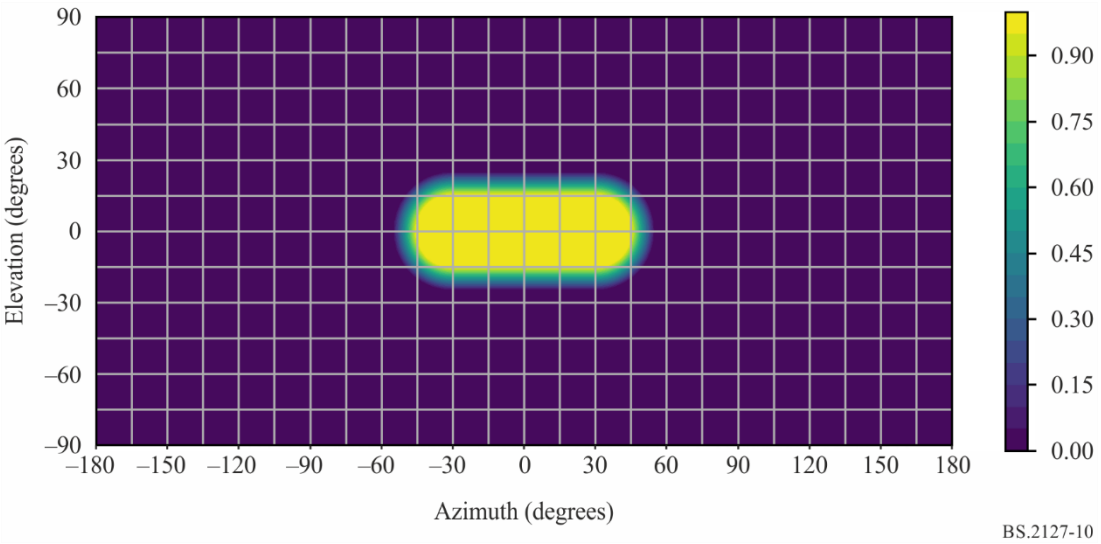


FIGURE 11

Polar weighting function for `width` = 30° and `height` = 90°

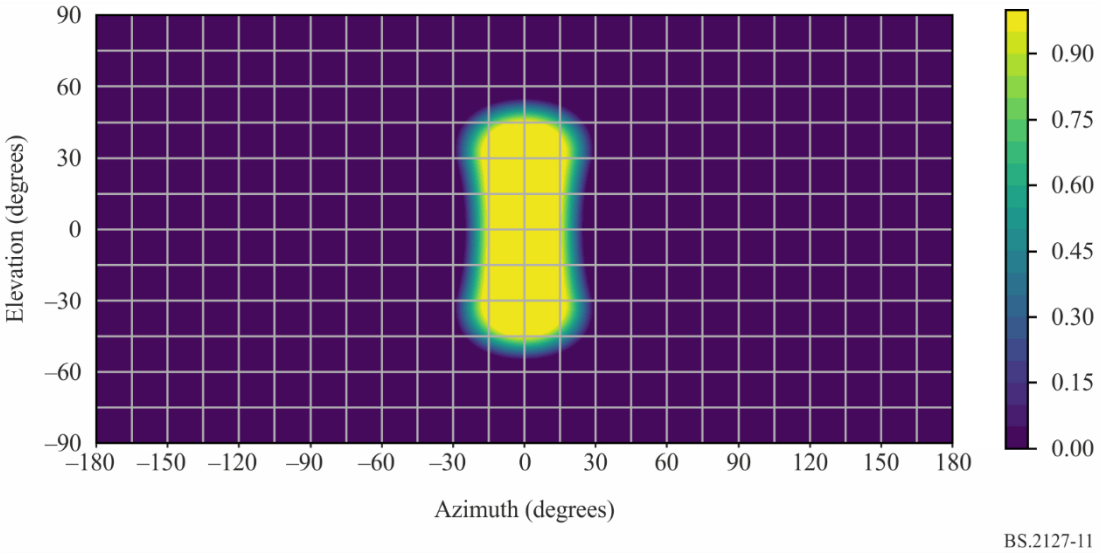


FIGURE 12

Polar weighting function for `width` = 300° and `height` = 30°

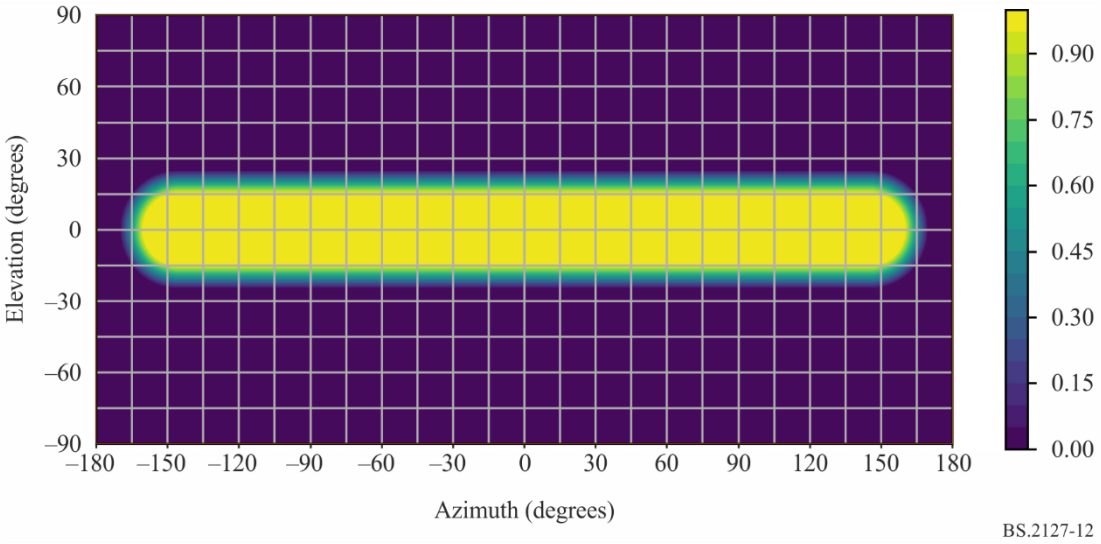
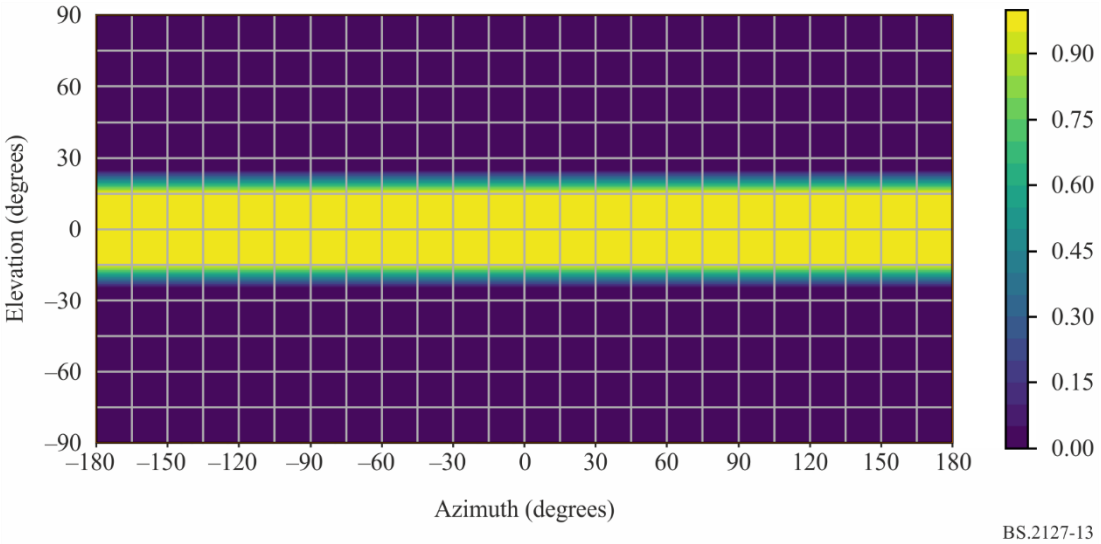


FIGURE 13

Polar weighting function for `width` = 360° and `height` = 30°



7.3.9 Cartesian Loudspeaker Positions

To use the Cartesian point source panner specified in § 7.3.10, a Cartesian position has to be found for each loudspeaker in the layout.

The interface to this component is as follows:

```
vector<CartesianPosition> positions_for_layout(Layout layout)
```

First, the table of positions matching `layout.name` is found in § 11.2.

For each `channel` in `layout.channels`, the `x`, `y`, and `z` parameters of an output `CartesianPosition` are determined as follows:

– If `channel.name` is `M+SC` or `M-SC`, then:

$$\{x, y, z\} = \texttt{point\_polar\_to\_cart}(\texttt{channel.polar\_position.azimuth}, 0, 1)$$

Note that this assumes infinite precision in `point_polar_to_cart`. In practice, the positions must be modified such that:

- $z = 0$
- The y coordinates of both screen loudspeakers must be identical.
- The x coordinates of both screen loudspeakers must be exactly symmetrical about 0.

– Otherwise, the values are given in the table row labelled `channel.name`.

This is implemented in `core.allocentric`.

7.3.10 Cartesian point source panner

The Cartesian point panning algorithm consists of a 3D extension of the 'dual-balance' panner concept that is widely used in 5.1- and 7.1-channel surround sound production.

The inputs to the panner consist of an object's position $[p_{ox}, p_{oy}, p_{oz}]$ and the positions of the N output loudspeakers, all in Cartesian coordinates. Let $[p_{sx}(j), p_{sy}(j), p_{sz}(j)]$ denote the position of the loudspeaker j.

With regards to loudspeaker layout, the point source panner requires that the following conditions are satisfied in order to be able to accurately place a phantom image of the object anywhere in the room:

– The loudspeakers must be grouped into one or more discrete planes in the z-dimension.

– The loudspeakers on each plane must be grouped into one or more discrete rows in the y-dimension.

– On any row where $-1 < y < 1$ (*i.e.*, any row that does not intersect the front or rear walls of the room), there must be loudspeakers at $x = 1$ and $x = -1$.

– Every loudspeaker location must lie on the surface of the room cube, that is, either on the floor, ceiling, or walls.

– Positions that meet these conditions can be found by following the procedure given in § 7.3.9.

Approximately, the loudspeaker gains for a given source position are found by:

– Finding layers of loudspeakers above and below the source, and calculating a z gain for both of these layers based on the z position of the layers and the source.

– In each of the found layers, finding a row of loudspeakers in front and behind the source position, and calculating a y gain for each of these rows based on the y position of the rows and the source.

– In each of the found rows, finding a pair of loudspeakers to the left and the right of the source position, and calculating an x gain for each of these loudspeakers based on the x positions of the loudspeakers and the source.

Up to eight loudspeakers will have been selected; in each of these the gain is $x \times y \times z$; other loudspeakers have zero gain.

The exact specification of the algorithm is given below, calculating a gain $g^{point}(j_x, j_y, j_z)$ for each loudspeaker j. Noting that each axis is separable, it is also useful to observe that $g^{point}(x, y, z) = g^{point_x}(x) \times g^{point_y}(y) \times g^{point_z}(z)$, and that the three independent gains are available as intermediate values in the algorithm.

```
epsilon = 0.001 //small positive constant

//simplification: Use object-centric coordinates, so that object is
//always at the origin.
for (j = 1 to N)
{
  p_sx(j) -= p_ox
  p_sy(j) -= p_oy
  p_sz(j) -= p_oz
}

for (j = 1 to N)
{
  //Z-gain
  z_this = p_sz(j)
  //find loudspeakers in other plane, on other side of object
  if (z_this >= 0) {
    z_other = max({p_sz : p_sz < z_this})
  } else {
    z_other = min({p_sz : p_sz > z_this})
  }
  if (isempty(z_other)) {
    gz = 1.0
  } else if (sign(z_other) == sign(z_this)) {
    gz = 0.0
  } else {
    gz = cos(z_this / (z_other - z_this) * pi /2)
  }

  //Y-gain
  //from among loudspeakers in this plane...
  p_sx_plane = p_sx({i:abs(p_sz(i) - z_this) < epsilon})
  p_sy_plane = p_sy({i:abs(p_sz(i) - z_this) < epsilon})
  y_this = p_sy(j)
  //...find loudspeakers in closest row, on other side of object
  if (y_this >= 0) {
    y_other = max({p_sy_plane : p_sy_plane < y_this})
  } else {
    y_other = min({p_sy_plane : p_sy_plane > y_this})
  }
  if isempty(y_other) {
    gy = 1.0
  } else if (sign(y_other) == sign(y_this)) {
    gy = 0.0
  } else {
    gy = cos(y_this / (y_other - y_this) * pi /2)
  }

  //X-gain
  //Among loudspeakers in this plane and row...
  p_sx_row = p_sx_plane({i:abs(p_sy_plane(i) - y_this) < epsilon})
  x_this = p_sx(j)
  //find loudspeakers in the closest column
  if (x_this >= 0) {
    x_other = max({p_sx_row : p_sx_row < x_this})
  } else {
    x_other = min({p_sx_row : p_sx_row > x_this})
  }
  if (isempty(x_other)) {
```

```
      gx = 1.0
    } else if (sign(x_other) == sign(x_this)) {
      gx = 0.0
    } else {
      gx = cos(x_this / (x_other - x_this) * pi /2)
    }
    g_point(j) = gx * gy * gz
  }
```

Observe that at most eight loudspeakers will have non-zero gains, and that the sum of the squares of the loudspeaker gains will always be 1, so the panning operation is energy-preserving.

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

7.3.11 Cartesian Extent Panner

The purpose of the extent panner is to calculate a gain coefficient for each loudspeaker in the output loudspeaker layout, given an object position and object extents. The intention of extent is to make the object appear larger so that when the extent is at the maximum the object fills the room, while when it is set to zero the object is rendered as a point object.

To achieve this, the extent panner considers a grid of many virtual sources in the room. Each virtual source fires loudspeakers exactly in the same way any object rendered with the point source panner would. The extent panner, when given an object position and object extents, determines which (and how many) of those virtual sources will contribute.

The following steps are necessary to calculate the gains for an object with extent. Each step is explained in more detail in one of the following subsections.

1. Pre-scale the extent parameters.
2. Calculate point gains for all virtual sources.
3. Combine all the gains from virtual sources within the room to produce inside extent gains.
4. Combine all the gains from virtual sources on the boundaries of the room to produce boundary extent gains.
5. Combine the inside and boundary extent gains to produce the final extent gains.
6. Combine the final extent gains with the point gains for the object.

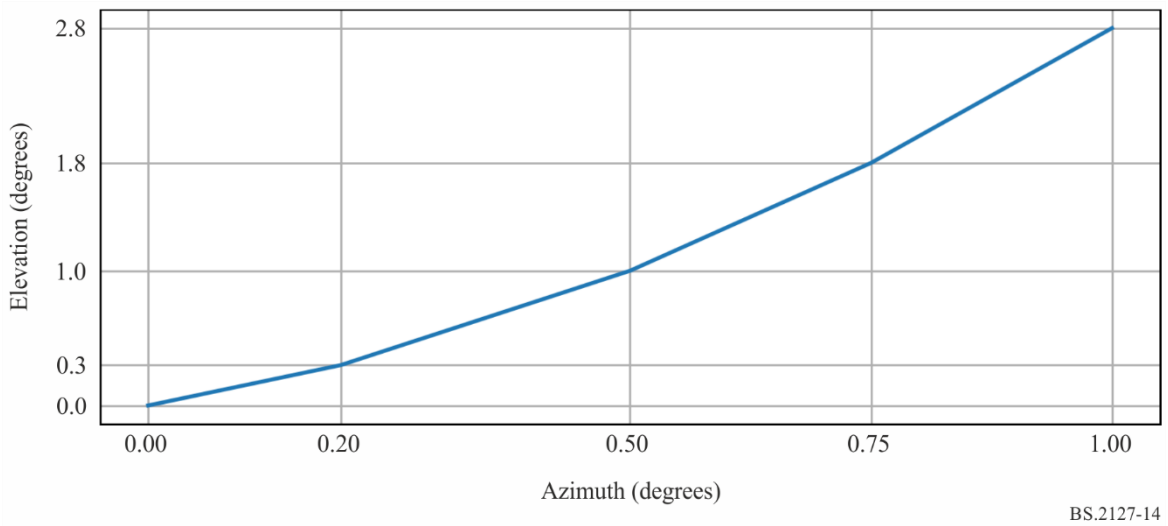
The Cartesian extent panner is implemented is
`core.objectbased.allo_extent.get_gains`.

7.3.11.1 Pre-scaling of extent parameters

Prior to calculating any gains the extent parameter values are scaled up so that the source weighting function behaves more intuitively. The user is exposed to values $s \in [0,1]$, which are mapped into the actual extent used by the algorithm to the range $[0,2.8]$. The mapping is done by a piecewise linear function defined by the value pairs $(0,0), (0.2,0.3), (0.5,1.0), (0.75,1.8), (1,2.8)$ and shown in Fig. 14. The maximum value of 2.8 ensures that when extent is set to maximum (1.0), it truly occupies the whole room. In what follows, the variables $\hat{s}_x, \hat{s}_y, \hat{s}_z$ refer to the input extent values after mapping has been applied.

FIGURE 14

Piecewise-linear mapping between ADM extent parameters and algorithm internal extent values



To maintain desired behaviour under extreme values of extent, minimum values on $\hat{s}_x, \hat{s}_y, \hat{s}_z$ are applied as follows:

$$s_x = max\left(\hat{s}_x, \frac{2}{N_x - 1}\right), s_y = max\left(\hat{s}_y, \frac{2}{N_y - 1}\right), s_z = max\left(\hat{s}_z, \frac{2}{N_z - 1}\right)$$

These restricted values s_x, s_y, s_z are used throughout in the algorithm.

7.3.11.2 Calculating virtual source gains

The grid of virtual sources is defined as a static rectangular uniform grid of $N_x \times N_y \times N_z$ points. The grid spans the range of positions $[-1,1]$ in each dimension. The density needs to be set in a manner that includes a few sources between loudspeakers in a typical layout. Empirical testing showed that $N_x = N_y = N_z = 40$ created an appropriate grid of virtual sources[1]. The notation (x_s, y_s, z_s) will be used to denote the possible coordinates of the virtual sources. Each virtual source creates a set of gains $g_j^{point}(x_s, y_s, z_s)$ to each loudspeaker $j = 1, \ldots, N_j$ of the layout according to the Cartesian point source panner algorithm described in § 7.3.10. Note that if any loudspeakers have been excluded from the layout due to a Zone Exclusion object (see § 7.3.5) the reduced loudspeaker layout is used when calculating the gains.

7.3.11.3 Combining virtual source gains inside the room

The object position and extent $(x_o, y_o, z_o, s_x, s_y, s_z)$ are used to calculate a set of weights that determine how much each virtual source will contribute to the final gains[2]. The weights for each virtual source are denoted $w(x_s, y_s, z_s, x_o, y_o, z_o, s_x, s_y, s_z)$ and are used to scale the point gains for each virtual source. After being weighted, all the virtual source gains are summed together to produce the inside extent gains:

---

[1] For loudspeaker layouts where there are no bottom layer loudspeakers the range of virtual sources in the Z dimension is limited to $[0,1]$, and the recommended value of N_z is 20.

[2] For loudspeaker layouts where there are no bottom layer loudspeakers the extent algorithm uses $z_o = max(p_{oz}, 0)$ as the objects's position in the Z dimension. Otherwise, $z_o = p_{oz}$. For all loudspeaker layouts, the extent algorithm uses the same X- and Y-position as the point source panner (*i.e.*, $y_o = p_{oy}, x_o = p_{ox}$).

$$g_j^{inside}(x_o, y_o, z_o, s_x, s_y, s_z) = \sum_{x_s,y_s,z_s} w\left(x_s, y_s, z_s, x_o, y_o, z_o, s_x, s_y, s_z\right) \times g_j^{point}(x_s, y_s, z_s)$$

However, the extent algorithm combines virtual source gains in a way that varies depending on the extent of the object. In general, this can be described as:

$$g_j^{inside}(x_o, y_o, z_o, s_x, s_y, s_z) = \left[\sum_{x_s,y_s,z_s} \left[w(x_s, y_x, z_s, x_o, y_o, z_o, s_x, s_y, s_y) \times g_j^{point}(x_s, y_s, z_s)\right]^p\right]^{\frac{1}{p}}$$

The extent-dependent exponent p controls the smoothness of the gains across loudspeakers. It ensures homogeneous growth of the object at small s and correct energy distribution across all directions at large s. To calculate p, first sort $\{s_x, s_y, s_z\}$ in descending order, and label the resulting ordered triad: $\{s_1, s_2, s_3\}$. The triad can then be combined to give an effective extent:

$$s_{eff} = \frac{6}{9}s_1 + \frac{2}{9}s_2 + \frac{1}{9}s_3$$

For layouts with a single plane of loudspeakers, such as 0+5+0 or if zone exclusion results in the layout being reduced to a single plane, first sort $\{s_x, s_y\}$ in descending order, and label the resulting ordered duo: $\{s_1, s_2\}$ giving:

$$s_{eff} = \frac{3}{4}s_1 + \frac{1}{4}s_2$$

For 0+2+0 layout (Stereo) or if zone exclusion reduces the set of loudspeakers to a single row, $s_{eff} = s_x$.

The effective extent is them used to calculate a piecewise defined exponent:

$$p = \begin{cases} 6 & s_{eff} \leq 0.5 \\ 6 - 4 \times \frac{s_{eff} - 0.5}{s_{max} - 0.5} & otherwise \end{cases}$$

where $s_{max} = 2.8$, such that when s is at its maximum, $p = 2$.

The weight function can also treat each axis separately and the whole extent computation simplifies if separable weight functions are used:

$$w(x_s, y_s, z_s, x_o, y_o, z_o, s_x, s_y, s_z) = w_x(x_s, x_o, s_x) w_y(y_s, y_o, s_y) w_z(z_s, z_o, s_z)$$

The chosen functions look like something between circles and squares (or spheres and cubes, in 3D):

$$\begin{aligned} w_x(p, o, s) = w_y(p, o, s) &= 10^{-\min\left(\left[\frac{3}{2}\left(\frac{p-o}{2s}\right)\right]^4, 6.5\right)} \\ w_z(p, o, s) &= 10^{-\min\left(\left[\frac{3}{2}\left(\frac{p-o}{s}\right)\right]^4, 6.5\right)} \times \cos\left(s\frac{3\pi}{7}\right) \end{aligned}$$

This means that g_j^{inside} can be simplified to

$$g_j^{inside}(x_o, y_o, z_o, s_x, s_y, s_z) = f_j^x(x_o, s_x) f_j^y(y_o, s_y) f_j^z(z_o, s_z)$$

where:

$$\begin{aligned} f_j^x(x_o, s_x) &= \sum_{x_s}\left[g_j^{point_x}(x_s) w_x(x_s, x_o, s_x)\right]^p \\ f_j^y(y_o, s_y) &= \sum_{y_s}\left[g_j^{point_y}(y_s) w_y(y_s, y_o, s_y)\right]^p \\ f_j^z(z_o, s_z) &= \sum_{z_s}\left[g_j^{point_z}(z_s) w_z(z_s, z_o, s_z)\right]^p \end{aligned}$$

Note that for layouts limited to a single plane of loudspeakers, $f_j^z(z_o, s_z) = 1$, and for a single row of loudspeakers, $f_j^z(z_o, s_z) = f_j^y(y_o, s_y) = 1$.

Additionally, very small values of $f_j(c, s)$ $(10^{-6.5})$ are rounded down to zero to avoid floating point underflow in implementations.

A normalization step is applied to g_j^{inside}:

$$\tilde{g}_j^{inside} = \begin{cases} \dfrac{g_j^{inside}}{\sqrt{\sum_n \left[g_n^{inside}\right]^2}} & \sqrt{\sum_n \left[g_n^{inside}\right]^2} > tol \\ 0 & otherwise \end{cases}$$

where $tol = 10^{-5}$.

7.3.11.4 Combining boundary gains

One further modification is that, for aesthetic reasons, it is important to have a mode where there is no opposite loudspeaker firing. This is accomplished by using virtual sources located only on the boundary. To handle certain loudspeaker layouts as special cases;

- $dim = 1$ for layouts with only a single row of loudspeakers after zone exclusion is applied (e.g. `0+2+0`),
- $dim = 2$ for layouts with only a single plane of loudspeakers after zone exclusion is applied (e.g. `0+5+0`),
- $dim = 4$ for layouts with more than two distinct height planes of loudspeakers after zone exclusion is applied (e.g. `3+7+0` and `9+10+3`), and
- $dim = 3$ otherwise.

The boundary gain is then:

$$\begin{aligned} g_j^{bound}(x_o, y_o, z_o, s_x, s_y, s_z) = {} & b_j^{floor}(z_o, s_z) f_j^x(x_o, s_x) f_j^y(y_o, s_y) \\ & + b_j^{ceil}(z_o, s_z) f_j^x(x_o, s_x) f_j^y(y_o, s_y) \\ & + b_j^{left}(x_o, s_x) f_j^y(y_o, s_y) f_j^z(z_o, s_z) \\ & + b_j^{right}(x_o, s_x) f_j^y(y_o, s_y) f_j^z(z_o, s_z) \\ & + b_j^{front}(y_o, s_y) f_j^x(x_o, s_x) f_j^z(z_o, s_z) \\ & + b_j^{back}(y_o, s_y) f_j^x(x_o, s_x) f_j^z(z_o, s_z) \end{aligned}$$

where:

$$b_j^{floor}(z_o, s_z) = \begin{cases} \left[g_j^{point}(z_s = -1.0)w(-1.0, z_o, s_z)\right]^p & dim = 4 \\ 0 & \text{otherwise} \end{cases}$$

$$b_j^{ceil}(z_o, s_z) = \begin{cases} \left[g_j^{point}(z_s = 1.0)w(1.0, z_o, s_z)\right]^p & dim \geq 3 \\ 0 & \text{otherwise} \end{cases}$$

$$b_j^{left}(x_o, s_x) = \left[g_j^{point}(x_s = -1.0)w(-1.0, x_o, s_x)\right]^p$$

$$b_j^{right}(x_o, s_x) = \left[g_j^{point}(x_s = 1.0)w(1.0, x_o, s_x)\right]^p$$

$$b_j^{front}(y_o, s_y) = \begin{cases} \left[g_j^{point}(y_s = 1.0)w(1.0, y_o, s_y)\right]^p & dim > 1 \\ 0 & \text{otherwise} \end{cases}$$

$$b_j^{back}(y_o, s_y) = \begin{cases} \left[g_j^{point}(y_s = -1.0)w(-1.0, y_o, s_y)\right]^p & dim > 1 \\ 0 & \text{otherwise} \end{cases}$$

7.3.11.5 Combining inside and boundary gains

The boundary gains now need to be combined with the inside gains, so a fade-out factor is introduced for all virtual sources inside the room, with fade-out amount = 'fraction of object outside the room'.

The result is:

$$g_j^{extent} = \left[\tilde{g}_j^{bound} + \left(\mu \times \tilde{g}_j^{inside}\right)\right]^{\frac{1}{p}}$$

where:

$$d_{bound} = \begin{cases} \min(x_o + 1, 1 - x_o) & dim = 1 \\ \min(x_o + 1, 1 - x_o, y_o + 1, 1 - y_o) & dim = 2 \\ \min(x_o + 1, 1 - x_o, y_o + 1, 1 - y_o, z_o + 1, z_o - 1) & \text{otherwise} \end{cases}$$

$$\mu = \begin{cases} h(x_o, s_x)^3 & dim = 1 \\ h(x_o, s_x)h(y_o, s_y)^{\frac{3}{2}} & dim = 2 \\ h(x_o, s_x)h(y_o, s_y)h(z_o, s_z) & \text{otherwise} \end{cases}$$

and $h(c, s)$ is a fade out function for a single dimension.

$$h(c, s) = \begin{cases} \left[\frac{\max(2s, 0.4)^3}{0.16 \times 2s}\right]^{\frac{1}{3}} & d_{bound} \geq s \wedge d_{bound} \geq 0.4 \\ \left[\frac{d_{bound}}{2}\left(\frac{d_{bound}}{0.4}\right)^2\right]^{\frac{1}{3}} & \text{otherwise} \end{cases}$$

As part of the extended object starts moving outside the room, all virtual sources inside the object start fading out, except for those at the boundaries. When an object reaches a boundary, only the boundary gains will be contributing to the extent gains. d_{bound} is the minimum distance to a boundary.

A normalization step is applied to g_j^{extent}

$$\tilde{g}_j^{extent} = \begin{cases} \frac{g_j^{extent}}{\sqrt{\sum_n [g_n^{extent}]^2}} & \sqrt{\sum_n [g_n^{extent}]^2} > tol \\ 0 & \text{otherwise} \end{cases}$$

7.3.11.6 Combining extent gains and point gains

The extent contributions are then combined with the point gains, and a crossfade between them is applied as a function of extent:

$$g_j^{total} = \left(\alpha \times g_j^{point}(x_o, y_o, z_o)\right) + \left(\beta \times \tilde{g}_j^{extent}\right)$$

where:

$$\alpha = \begin{cases} \cos\left(\frac{s_{eff}}{s_{fade}} \times \frac{\pi}{2}\right) & s_{eff} < s_{fade} \\ 0 & \text{otherwise} \end{cases}$$

$$\beta = \begin{cases} \sin\left(\frac{s_{eff}}{s_{fade}} \times \frac{\pi}{2}\right) & s_{eff} < s_{fade} \\ 1 & \text{otherwise} \end{cases}$$

and $s_{fade} = 0.2$.

This ensures smooth panning and smooth growth of the object, providing a nice transition all the way between the smallest and the largest possible extents.

Finally, a last normalization is applied to the gains:

$$G_j^S = \begin{cases} \frac{g_j^{total}}{\sqrt{\sum_n [g_n^{total}]^2}} & \sqrt{\sum_n [g_n^{total}]^2} > tol \\ 0 & \text{otherwise} \end{cases}$$

7.3.12 Polar 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 § 7.3.12.1, and calculating the downmix to route away from the excluded loudspeakers, in § 7.3.12.2.

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.

7.3.12.1 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:

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

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.

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

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

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`.

7.3.12.2 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 3 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`.

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 $\{\mathtt{M+030}, \mathtt{M-030}\}$, unless both of these loudspeakers are excluded in which case it would be routed to the non-excluded loudspeakers in $\{\mathtt{M+110}, \mathtt{M-110}\}$, etc.

TABLE 3

Example loudspeaker association for 4+5+0

Input	Output groups
M+030	{M+030}, {M+000}, {M−030}, {M+110}, {M−110}, {U+030}, {U−030}, {U+110}, {U−110}
M−030	{M−030}, {M+000}, {M+030}, {M−110}, {M+110}, {U−030}, {U+030}, {U−110}, {U+110}
M+000	{M+000}, {M+030 , M−030}, {M+110 , M−110}, {U+030 , U−030}, {U+110 , U−110}
M+110	{M+110}, {M−110}, {M+030}, {M+000}, {M−030}, {U+110}, {U−110}, {U+030}, {U−030}
M−110	{M−110}, {M+110}, {M−030}, {M+000}, {M+030}, {U−110}, {U+110}, {U−030}, {U+030}
U+030	{U+030}, {U−030}, {U+110}, {U−110}, {M+030}, {M+000}, {M−030}, {M+110}, {M−110}
U−030	{U−030}, {U+030}, {U−110}, {U+110}, {M−030}, {M+000}, {M+030}, {M−110}, {M+110}
U+110	{U+110}, {U−110}, {U+030}, {U−030}, {M+110}, {M−110}, {M+030}, {M+000}, {M−030}
U−110	{U−110}, {U+110}, {U−030}, {U+030}, {M−110}, {M+110}, {M−030}, {M+000}, {M+030}

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

7.3.12.2.1 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 four 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 loudspeaker from a higher layer before a lower one. The layer priorities are drawn from Table 4.

– 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 4

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

7.3.12.2.2 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 **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 **p** of length $\frac{N}{2} + 1$ is defined as:

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

– The corresponding frequency vector **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 **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.

8 Render Items with typeDefinition==DirectSpeakers

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

The basic algorithm is as follows:

1. For inputs specified using common definitions *audioPackFormats* describing layouts specified in Recommendation ITU-R BS.2051-2, mapping rules according to § 8.1 are applied.
2. Determine if the metadata refers to an LFE channel (see § 8.2). If it does, then only LFE outputs will be considered, and if it doesn't, only non-LFE outputs will be considered.
3. If any of the *speakerLabels* match a loudspeaker (see § 8.3) the channel is routed to the first loudspeaker that matches. If no *speakerLabel* matches, continue to the next step.
4. 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 § 8.4).
5. If the nominal position of any loudspeaker is within the specified position bounds (see § 8.5), route the channel to the loudspeaker closest to the specified nominal position. The loudspeaker positions used are determined by the type of the position as in § 8.5. If there are no loudspeakers within the bounds, or the closest loudspeaker to the nominal position is not unique), continue to the next step.
6. 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 corresponding to the coordinate type used to define its position 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 Mapping Rules

– If the last *audioPackFormat* listed in `type_metadata.audioPackFormats` is not a common definitions pack format (i.e. it was specified in the input metadata, not read from the common definitions file), do not apply mapping rules.

– Look up the ID of the last *audioPackFormat* listed in `type_metadata.audioPackFormats` in Table 15 to determine `input_layout`. If it is not listed, do not apply mapping rules.

– Try to apply each rule listed in Table 16 in turn. If any rule applies, then the `gains` for the first matching rule listed are used to reproduce this channel. If no rule matches, continue to the next step. A rule matches if all these conditions are met:

  • `rule.speakerLabel` is equal to the first (and only) *speakerLabel* after the normalisation described in § 8.3 is applied.

  • `input_layout` (as determined above) is listed in `rule.input_layouts`, if this is listed.

  • The name of the output loudspeaker layout, `layout.name`, is listed in `rule.output_layouts`, if this is listed.

  • All channel names listed in `rule.gains` exist in `layout.channel_names`.

8.2 LFE Determination

A channel is considered to be an LFE channel if either the frequency element in the audioChannelFormat has a value described in § 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).

NOTE – IAR does not apply any +10 dB offset (see Recommendation ITU-R BS.775) to the LFE channel to address playback calibration, with respect to the main channel, as this is done in the playback device.

8.3 Loudspeaker Label Matching

The matching for *speakerLabels* only works for the labels used in Recommendation. ITU-R BS.2051-2 (e.g. `M+030`) and the URNs used in the file of the common definitions of the ADM specified in Recommendation ITU-R BS.2094-1 (e.g. `urn:itu:bs:2051:0:speaker:M+030`). The labels of LFE1 and LFE2 are specified in Recommendation ITU-R BS.2051-2. When the following *speakerLabels* are used in the ADM file, some substitutions are applied:

– `LFE → LFE1`

– `LFEL → LFE1`

– `LFER → LFE2`

8.4 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.

8.5 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 loudspeaker matches if all coordinates lie within the specified *bounds*. With the exception, that loudspeakers with polar coordinates at the poles (e.g. `T+000`) match any azimuth range, as they have an indeterminate azimuth.

A loudspeaker with nominal polar position `speaker` matches bounds specified in polar coordinates if

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

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` having been converted to Cartesian coordinates using § 7.3.9 matches bounds specified using Cartesian coordinates if

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

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 Recommendation ITU-R BS.2076-1, 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 three 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 this 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.

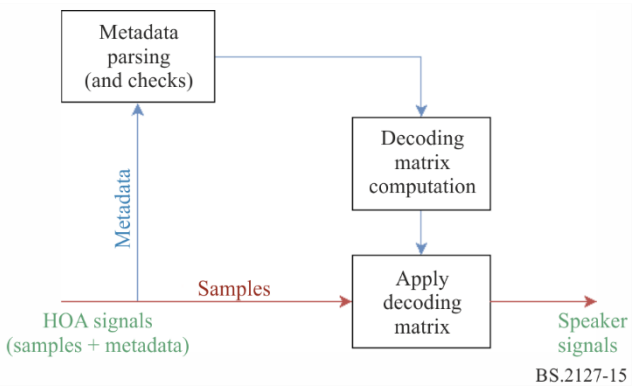
– *equation*, which is meant to be used as a replacement of the *order* and *degree* sub-elements. The current ADM standard does not provide precise rules regarding the format used to specify mathematical formulas. Therefore, this sub-element cannot be supported reliably.

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

FIGURE 15

HOA rendering flow diagram



The process of rendering HOA signals over loudspeakers is summarized in Fig. 15. 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:

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

where:

$\mathbf{S}_{\mathrm{spk}}$: matrix of loudspeaker signals, with dimensions $N_{\mathrm{spk}} \times N_{\mathrm{samp}}$

$\mathbf{S}_{\mathrm{HOA}}$: matrix of HOA signals, with dimensions $N_{\mathrm{HOA}} \times N_{\mathrm{samp}}$

D : real-valued matrix, with dimensions $N_{\mathrm{spk}} \times N_{\mathrm{HOA}}$ and is referred to as the *HOA decoding matrix*

N_{HOA}, N_{spk} and N_{samp} denote the number of HOA signals, loudspeaker 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 renderer implements the AllRAD HOA decoding technique [1]. This method provides robust HOA decoding over irregular loudspeaker layouts such as that described in Recommendation ITU-R BS.2051-2. 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 loudspeakers which are evenly distributed over the sphere, and
2. Panning the virtual loudspeaker signals over the actual loudspeakers.

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 loudspeaker positions

In order to facilitate the calculation of the decoding matrix, the angular positions of the virtual loudspeakers 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 loudspeaker positions than there are HOA signals.

In this renderer, the virtual loudspeaker 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 loudspeaker decoding matrix

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

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

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

where (θ_n, φ_n) denotes the elevation and azimuth angles for the n-th virtual loudspeaker (using the HOA coordinate system and notation as defined in Recommendation ITU-R BS.2076-1) 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 loudspeaker 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

VBAP panning is typically employed for the calculation of the panning gain matrix in the AllRAD HOA decoding method. In this renderer's implementation, the method used to calculate the panning gains is simply that provided for panning point source objects (`core.point_source`).

9.3.1.4 Energy normalisation

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 loudspeaker 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:

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

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

10 Metadata conversion

This section specifies a method for converting between polar and Cartesian parameters within *audioBlockFormat*s with *typeDefinition==Objects*. Metadata conversion cannot by nature be exact; the results of conversion will not exactly match those without. Therefore, the results of conversion should be monitored. Note that extent conversion is not invertible, so conversion back and forth between polar and Cartesian should be avoided.

The interface to the conversion functionality is as follows:

```
AudioBlockFormat to_cartesian(AudioBlockFormat input);
AudioBlockFormat to_polar(AudioBlockFormat input);
```

When `to_cartesian` is called with an `AudioBlockFormat input` where `input.cartesian` is set, `input` is returned as-is. Conversely, `to_polar` is called with an `AudioBlockFormat input` where `input.cartesian` is not, `input` is returned as-is.

Otherwise, in both cases `input.cartesian` is inverted, and the following changes are made to `input` before it is returned:

– `input.position` is converted according to § 10.1.

– `input.width`, `input.height` and `input.depth` are converted according to § 10.2.

– `input.objectDivergence` is converted according to § 10.3.

Conversion is implemented in `core.objectbased.conversion`.

10.1 *position* conversion

Positions are converted such that the polar position of a loudspeaker in the `4+5+0` layout is mapped to the Cartesian coordinate of that loudspeaker used in the Cartesian point source panner, as given in Table 8.

Note that the same conversion, the one based on the 4+5+0 channel configuration, is used regardless of the renderer channel layout. This is done to ensure that the results of the conversion are always consistent, even in the use cases when the renderer reproduction layout being used is not known during the time of the conversion. The 4+5+0 layout was chosen primarily to ensure good conversion for content authored using 0+5+0.

This section describes common definitions used for conversion in both directions; the conversion functions themselves are described in §§ 10.1.1 and 10.1.2.

`map_linear_to_az` and `map_az_to_linear` define an invertible mapping of source positions between azimuths (φ) and linear coordinates (x) between a pair of loudspeakers with azimuths φ_l and $azimuth_r$, considering the panning curves of the point source panners used for polar and Cartesian coordinates.

For example, a polar position φ_o between 0° and −30° has an x position given by:

$$x = \texttt{map\_az\_to\_linear}(0, -30, \varphi_o)$$

The linear to azimuth mapping is defined as:

$$\texttt{map\_linear\_to\_az}(\varphi_l, \varphi_r, x) = \varphi_{\texttt{mid}} + \varphi_{\texttt{rel}}$$

where:

$$\begin{aligned}
\varphi_{\texttt{mid}} &= \frac{\varphi_l + \varphi_r}{2} \\
\varphi_{\texttt{range}} &= \varphi_r - \varphi_{\texttt{mid}} \\
g'_l &= \cos\frac{x\pi}{2} \\
g'_r &= \sin\frac{x\pi}{2} \\
g_r &= \frac{g'_r}{g'_l + g'_r} \\
\varphi_{\texttt{rel}} &= \frac{180}{\pi}\arctan\left(2\left(g_r - \frac{1}{2}\right)\tan\left(\frac{\pi}{180}\varphi_{\texttt{range}}\right)\right)
\end{aligned}$$

The inverse function is defined as:

$$\texttt{map\_az\_to\_linear}(\varphi_l, \varphi_r, \varphi) = \frac{2}{\pi}\text{atan2}(g_r, 1 - g_r)$$

where:

$$\begin{aligned}
\varphi_{\texttt{mid}} &= \frac{\varphi_l + \varphi_r}{2} \\
\varphi_{\texttt{range}} &= \varphi_r - \varphi_{\texttt{mid}} \\
\varphi_{\texttt{rel}} &= \varphi - \varphi_{\texttt{mid}} \\
g_r &= \frac{1}{2} + \frac{\tan\left(\frac{\pi}{180}\varphi_{\texttt{rel}}\right)}{2\tan\left(\frac{\pi}{180}\varphi_{\texttt{range}}\right)}
\end{aligned}$$

This mapping is applied between the mid-layer loudspeaker positions, according to the following rules giving a left and right azimuth, and a left and right x and y position for a given input azimuth:

$$find_{sector}(\varphi) \begin{cases} \{30,0,\{-1,1\},\{0,1\}\} & IAR(\varphi,0,30) \\ \{0,-30,\{0,1\},\{1,1\}\} & IAR(\varphi,-30,0) \\ \{-30,-110,\{1,1\},\{1,-1\}\} & IAR(\varphi,-110,-30) \\ \{-110,110,\{1,-1\},\{-1,-1\}\} & IAR(\varphi,110,-110) \\ \{110,30,\{-1,-1\},\{-1,1\}\} & IAR(\varphi,30,110) \end{cases}$$

$$find\ \ (\varphi) \begin{cases} \{30,0,\{-1,1\},\{0,1\}\} & IAR(\varphi,0,45) \\ \{0,-30,\{0,1\},\{1,1\}\} & IAR(\varphi,-45,0) \\ \{-30,-110,\{1,1\},\{1,-1\}\} & IAR(\varphi,-135,-45) \\ \{-110,110,\{1,-1\},\{-1,-1\}\} & IAR(\varphi,135,-135) \\ \{110,30,\{-1,-1\},\{-1,1\}\} & IAR(\varphi,45,135) \end{cases}$$

where IAR is the function `inside_angle_range` described in § 6.2.

The following parameters are common for both conversion directions:

$$\begin{aligned} \theta_{top} &= 30 \\ \theta'_{top} &= 45 \\ \epsilon &= 1 \times 10^{-10} \end{aligned}$$

10.1.1 Polar to Cartesian

To convert a polar coordinate with azimuth φ, elevation θ and distance d to a Cartesian coordinate the function

$$\texttt{point\_polar\_to\_cart}(\varphi,\theta,d) = x,y,z$$

is used, where if $|\theta| > \theta_{top}$ then:

$$\begin{aligned} \theta' &= \theta'_{top} + (90-\theta'_{top})\frac{|\theta|-\theta_{top}}{90-\theta_{top}} \\ z &= d\,\mathrm{sgn}(\theta) \\ r_{xy} &= d\tan\left(\frac{\pi}{180}(90-\theta')\right) \end{aligned}$$

otherwise:

$$\begin{aligned} \theta' &= \theta'_{top}\frac{\theta}{\theta_{top}} \\ z &= d\tan\left(\frac{\pi}{180}\theta'\right) \\ r_{xy} &= d \end{aligned}$$

finally:

$$\begin{aligned} \{\varphi_l,\varphi_r,\{x_l,y_l\},\{x_r,y_r\}\} &= \texttt{find\_sector}(\varphi) \\ \varphi' &= \texttt{relative\_angle}(\varphi_r,\varphi) \\ \varphi'_l &= \texttt{relative\_angle}(\varphi_r,\varphi_l) \\ p &= \texttt{map\_az\_to\_linear}(\varphi'_l,\varphi_r,\varphi') \\ x &= r_{xy}(x_l + p(x_r - x_l)) \\ y &= r_{xy}(y_l + p(y_r - y_l)) \end{aligned}$$

`relative_angle` is described in § 6.7.

10.1.2 Cartesian to polar

To convert a Cartesian position with coordinates x, y and z to polar, the function

$$\texttt{point\_cart\_to\_polar}(x, y, z) = \varphi, \theta, d$$

is used, where if $|x| < \epsilon$ and $|y| < \epsilon$ then:

$$\{\varphi, \theta, d\} = \begin{cases} \{0,0,0\} & |z| < \epsilon \\ \{0,90\text{sgn}(z), |z|\} & \text{otherwise} \end{cases}$$

otherwise, continue:

$$\begin{aligned} \varphi' &= -\frac{180}{\pi}\text{atan2}(x, y) \\ \{\varphi_l, \varphi_r, \{x_l, y_l\}, \{x_r, y_r\}\} &= \texttt{find\_cart\_sector}(\varphi') \\ [g_l \quad g_r] &= [x \quad y] \cdot \begin{bmatrix} x_l & y_l \\ x_r & y_r \end{bmatrix}^{-1} \\ r_{xy} &= g_l + g_r \\ \varphi'_l &= \texttt{relative\_angle}(\varphi_r, \varphi_l) \\ \varphi_{\texttt{rel}} &= \texttt{map\_linear\_to\_az}\left(\varphi'_l, \varphi_r, \frac{g_r}{r_{xy}}\right) \\ \varphi &= \texttt{relative\_angle}(-180, \varphi_{\texttt{rel}}) \\ \theta' &= \frac{180}{\pi}\arctan\frac{z}{r_{xy}} \end{aligned}$$

If $|\theta'| > \theta'_{\texttt{top}}$, then:

$$\begin{aligned} |\theta| &= \theta_{\texttt{top}} + (90 - \theta_{\texttt{top}})\frac{|\theta'| - \theta'_{\texttt{top}}}{90 - \theta'_{\texttt{top}}} \\ \theta &= |\theta|\text{sgn}\theta' \\ d &= |z| \end{aligned}$$

otherwise:

$$\begin{aligned} \theta &= \theta'\frac{\theta_{\texttt{top}}}{\theta'_{\texttt{top}}} \\ d &= r_{xy} \end{aligned}$$

`local_coordinate_system` is defined in § 6.8.

10.2 Extent conversion

Conversion of extent parameters is implemented in two parts:

– $whd2xyz$ and $xyz2whd$: Functions which convert extent parameters between Cartesian and polar, assuming a source position directly in front of the listener with radius 1.

– `point_polar_to_cart` and `point_cart_to_polar`: functions which handle position and extent conversion. Positions are converted using the methods described in § 10.1. Extent conversion uses $whd2xyz$ and $xyz2whd$, rotating the Cartesian extent to match the position.

Note that extent conversion is not in general invertible.

10.2.1 Polar to Cartesian

`extent_polar_to_cart` takes a polar source position in the form of an azimuth, elevation and distance, and a polar width, height and depth, and returns the Cartesian x, y and z coordinates, and the Cartesian x, y and z sizes:

$$\texttt{extent\_polar\_to\_cart}(\varphi, \theta, d, \texttt{width}, \texttt{height}, \texttt{depth}) = \{x, y, z, s_x, s_y, s_z\}$$

where:

$$\begin{aligned} \{x, y, z\} &= \texttt{point\_polar\_to\_cart}(\varphi, \theta, d) \\ \{s_{x,f}, s_{y,f}, s_{z,f}\} &= \texttt{whd2xyz}(\texttt{width}, \texttt{height}, \texttt{depth}) \\ [\mathbf{M}_x \quad \mathbf{M}_y \quad \mathbf{M}_z] &= \text{diag}([s_{x,f}, s_{y,f}, s_{z,f}]) \cdot \texttt{local\_coordinate\_system}(\varphi, \theta) \\ s_x &= \| \mathbf{M}_x \|_2 \\ s_y &= \| \mathbf{M}_y \|_2 \\ s_z &= \| \mathbf{M}_z \|_2 \end{aligned}$$

and

$$whd2xyz(width, height, depth) = \{s_{x,w}, max(s_{y,w}, s_{y,h}, s_{y,d}), s_{z,h}\}$$

where:

$$\begin{aligned} s_{x,w} &= \begin{cases} \sin\frac{\pi}{180}\frac{\texttt{width}}{2} & \texttt{width} < 180 \\ 1 & \text{otherwise} \end{cases} \\ s_{y,w} &= \frac{1-\cos\frac{\pi \ \texttt{width}}{180 \quad 2}}{2} \\ s_{z,h} &= \begin{cases} \sin\frac{\pi}{180}\frac{\texttt{height}}{2} & \texttt{height} < 180 \\ 1 & \text{otherwise} \end{cases} \\ s_{y,h} &= \frac{1-\cos\frac{\pi \ \texttt{height}}{180 \quad 2}}{2} \\ s_{y,d} &= \texttt{depth} \end{aligned}$$

10.2.2 Cartesian to polar

`extent_cart_to_polar` takes a Cartesian source position in the form of an x, y and z coordinate, a Cartesian extent in the form of an x, y and z size, and returns a polar position and extent as an azimuth, elevation, distance, and a width, height and depth:

$$\texttt{extent\_cart\_to\_polar}(x, y, z, s_x, s_y, s_z) = \{\varphi, \theta, d, \texttt{width}, \texttt{height}, \texttt{depth}\}$$

where:

$$\begin{aligned} \{\varphi, \theta, d\} &= \texttt{point\_cart\_to\_polar}(x, y, z) \\ [\mathbf{M}_x \quad \mathbf{M}_y \quad \mathbf{M}_z] &= \text{diag}([s_x, s_y, s_z]) \cdot \texttt{local\_coordinate\_system}(\varphi, \theta)^T \\ s_{x,f} &= \| \mathbf{M}_x \|_2 \\ s_{y,f} &= \| \mathbf{M}_y \|_2 \\ s_{z,f} &= \| \mathbf{M}_z \|_2 \\ \{\texttt{width}, \texttt{height}, \texttt{depth}\} &= \texttt{xyz2whd}(s_{x,f}, s_{y,f}, s_{z,f}) \end{aligned}$$

and

$$xyz2whd(s_x, s_y, s_z) = \{w, h, d\}$$

where:

$$\begin{aligned} w_{sx} &= 2\frac{180}{\pi}\arcsin s_x \\ w_{sy} &= 2\frac{180}{\pi}\arccos(1-2s_y) \\ w &= w_{sx} + s_x\max(w_{sy}-w_{sx},0) \\ h_{sz} &= 2\frac{180}{\pi}\arcsin s_z \\ h_{sy} &= 2\frac{180}{\pi}\arccos(1-2s_y) \\ h &= h_{sz} + s_z\max(h_{sy}-h_{sz},0) \\ \{s_{x,\mathrm{eq}}, s_{y,\mathrm{eq}}, s_{z,\mathrm{eq}}\} &= \mathrm{whd2xyz}(w,h,0) \\ d &= \max(0, s_y - s_{y,\mathrm{eq}},) \end{aligned}$$

10.3 objectDivergence conversion

`azimuthRange` and `positionRange` are converted according to the following relationship:

$$positionRange = tan\frac{270\times azimuthRange}{\pi}$$

11 Data Structures and Tables

11.1 Internal Metadata Structures

11.1.1 Shared Structures

```
struct Position { };

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

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

struct Screen { };

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

struct CartesianScreen : Screen {
  float aspectRatio;
  CartesianPosition 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;
};
```

11.1.2 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;
};
```

11.1.3 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.2051-2.
  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;
};
```

11.2 Allocentric loudspeaker positions

This data is available in machine readable form in `iar/core/data/allo_positions.yaml`, but is included here for reference.

TABLE 5

Allocentric loudspeaker positions for 0+2+0

Channel	X	Y	Z
M+030	−1	1	0
M-030	1	1	0

TABLE 6

Allocentric loudspeaker positions for 0+5+0

Channel	X	Y	Z
M+030	−1	1	0
M-030	1	1	0
M+000	0	1	0
M+110	−1	−1	0
M-110	1	−1	0
LFE1	-1	1	-1

TABLE 7

Allocentric loudspeaker positions for 2+5+0

Channel	X	Y	Z
M+030	−1	1	0
M-030	1	1	0
M+000	0	1	0
M+110	−1	−1	0
M-110	1	−1	0
U+030	−1	1	1
U-030	1	1	1
LFE1	-1	1	-1

TABLE 8

Allocentric loudspeaker positions for 4+5+0

Channel	X	Y	Z
M+030	−1	1	0
M-030	1	1	0
M+000	0	1	0
M+110	−1	−1	0
M-110	1	−1	0
U+030	−1	1	1
U-030	1	1	1
U+110	−1	−1	1
U-110	1	−1	1
LFE1	-1	1	-1

TABLE 9

Allocentric loudspeaker positions for 4+5+1

Channel	X	Y	Z
M+030	−1	1	0
M-030	1	1	0
M+000	0	1	0
M+110	−1	−1	0
M-110	1	−1	0
U+030	−1	1	1
U-030	1	1	1
U+110	−1	−1	1
U-110	1	−1	1
B+000	0	1	−1
LFE1	-1	1	-1

TABLE 10

Allocentric loudspeaker positions for 3+7+0

Channel	X	Y	Z
M+000	0	1	0
M+030	−1	1	0
M-030	1	1	0
U+045	−1	1	1
U-045	1	1	1
M+090	−1	0	0
M-090	1	0	0
M+135	−1	−1	0
M-135	1	−1	0
UH+180	0	−1	1
LFE1	-1	1	-1
LFE2	1	1	-1

TABLE 11

Allocentric loudspeaker positions for 4+9+0

Channel	X	Y	Z
M+030	−1	1	0
M-030	1	1	0
M+000	0	1	0
M+090	−1	0	0
M-090	1	0	0
M+135	−1	−1	0
M-135	1	−1	0
U+045	−1	1	1
U-045	1	1	1
U+135	−1	−1	1
U-135	1	−1	1
LFE1	-1	1	-1

TABLE 12

Allocentric loudspeaker positions for 9+10+3

Channel	X	Y	Z
M+060	−1	0.414214	0
M-060	1	0.414214	0
M+000	0	1	0
M+135	−1	−1	0
M-135	1	−1	0
M+030	−1	1	0
M-030	1	1	0
M+180	0	−1	0
M+090	−1	0	0
M-090	1	0	0
U+045	−1	1	1
U-045	1	1	1
U+000	0	1	1
T+000	0	0	1
U+135	−1	−1	1
U-135	1	−1	1
U+090	−1	0	1
U-090	1	0	1

TABLE 12 (*end*)

Channel	X	Y	Z
U+180	0	−1	1
B+000	0	1	−1
B+045	−1	1	−1
B-045	1	1	−1
LFE1	-1	1	-1
LFE2	1	1	-1

TABLE 13

Allocentric loudspeaker positions for 0+7+0

Channel	X	Y	Z
M+030	−1	1	0
M-030	1	1	0
M+000	0	1	0
M+090	−1	0	0
M-090	1	0	0
M+135	−1	−1	0
M-135	1	−1	0
LFE1	-1	1	-1

TABLE 14

Allocentric loudspeaker positions for 4+7+0

Channel	X	Y	Z
M+030	−1	1	0
M-030	1	1	0
M+000	0	1	0
M+090	−1	0	0
M-090	1	0	0
M+135	−1	−1	0
M-135	1	−1	0
U+045	−1	1	1
U-045	1	1	1
U+135	−1	−1	1
U-135	1	−1	1
LFE1	-1	1	-1

11.3 DirectSpeakers mapping data

This data is available in machine readable form in `core.direct_speakers.panner.itu_packs` and `core.direct_speakers.panner.rules`, but is included here for reference.

TABLE 15

Mapping from common definitions *audioPackFormatID* to layout name (see § 8.1)

audioPackFormatID	`input_layout`
AP_00010001	0+1+0
AP_00010002	0+2+0
AP_00010003	0+5+0
AP_00010004	2+5+0
AP_00010005	4+5+0
AP_00010007	3+7+0
AP_00010008	4+9+0
AP_00010009	9+10+3
AP_0001000c	0+5+0
AP_0001000f	0+7+0
AP_00010010	4+5+1
AP_00010017	4+7+0

TABLE 16

Mapping rules for DirectSpeakers (see § 8.1)

Input *speakerLabel*	**Output reproduction gains**	`input_layouts`	`output_layouts`
M+000	M+000 = 1		
M+000	M+030 = M-030 = $\sqrt{\frac{1}{2}}$		
M+060	M+060 = 1		
M-060	M-060 = 1		
M+060	M+110 = $\sqrt{\frac{1}{3}}$, M+030 = $\sqrt{\frac{2}{3}}$		
M-060	M-110 = $\sqrt{\frac{1}{3}}$, M-030 = $\sqrt{\frac{2}{3}}$		
M+060	M+030 = M+090 = $\sqrt{\frac{1}{2}}$		
M-060	M-030 = M-090 = $\sqrt{\frac{1}{2}}$		
M+060	M+030 = 1		
M-060	M-030 = 1		

TABLE 16 (*continued*)

Input *speakerLabel*	**Output reproduction gains**	`input_layouts`	`output_layouts`
M+090	M+090 = 1		
M-090	M-090 = 1		
M+090	M+030 = $\sqrt{\frac{1}{3}}$, M+110 = $\sqrt{\frac{2}{3}}$	9+10+3	
M-090	M-030 = $\sqrt{\frac{1}{3}}$, M-110 = $\sqrt{\frac{2}{3}}$	9+10+3	
M+090	M+030 = M+110 = $\sqrt{\frac{1}{2}}$		
M-090	M-030 = M-110 = $\sqrt{\frac{1}{2}}$		
M+090	M+030 = $\sqrt{\frac{1}{2}}$		
M-090	M-030 = $\sqrt{\frac{1}{2}}$		
M+110	M+110 = 1		
M-110	M-110 = 1		
M+110	M+135 = 1		
M-110	M-135 = 1		
M+110	M+030 = $\sqrt{\frac{1}{2}}$		
M-110	M-030 = $\sqrt{\frac{1}{2}}$		
M+135	M+135 = 1		
M-135	M-135 = 1		
M+135	M+110 = 1		
M-135	M-110 = 1		
M+135	M+030 = $\sqrt{\frac{1}{2}}$		
M-135	M-030 = $\sqrt{\frac{1}{2}}$		
M+180	M+180 = 1		
M+180	M+135 = M-135 = $\sqrt{\frac{1}{2}}$		
M+180	M+110 = M-110 = $\sqrt{\frac{1}{2}}$		
M+180	M+030 = M-030 = $\sqrt{\frac{1}{4}}$		
U+000	U+000 = 1		
U+000	U+030 = U-030 = $\sqrt{\frac{1}{2}}$		

TABLE 16 (*continued*)

Input *speakerLabel*	**Output reproduction gains**	`input_layouts`	`output_layouts`
U+000	U+045 = U-045 = $\sqrt{\frac{1}{2}}$		
U+000	M+000 = 1		
U+000	M+030 = M-030 = $\sqrt{\frac{1}{2}}$		
U+030	U+030 = 1		
U-030	U-030 = 1		
U+030	U+045 = 1		
U-030	U-045 = 1		
U+030	M+030 = 1		
U-030	M-030 = 1		
U+045	U+045 = 1		
U-045	U-045 = 1		
U+045	U+030 = 1		
U-045	U-030 = 1		
U+045	M+030 = 1		
U-045	M-030 = 1		
U+090	U+090 = 1		
U-090	U-090 = 1		
U+090	UH+180 = $\sqrt{\frac{1}{3}}$, U+045 = $\sqrt{\frac{2}{3}}$	9+10+3	
U-090	UH+180 = $\sqrt{\frac{1}{3}}$, U-045 = $\sqrt{\frac{2}{3}}$	9+10+3	
U+090	U+030 = U+110 = $\sqrt{\frac{1}{2}}$		
U-090	U-030 = U-110 = $\sqrt{\frac{1}{2}}$		
U+090	U+045 = U+135 = $\sqrt{\frac{1}{2}}$		
U-090	U-045 = U-135 = $\sqrt{\frac{1}{2}}$		
U+090	M+090 = 1		
U-090	M-090 = 1		
U+090	U+030 = M+110 = $\sqrt{\frac{1}{2}}$		
U-090	U-030 = M-110 = $\sqrt{\frac{1}{2}}$		
U+090	M+030 = M+110 = $\sqrt{\frac{1}{2}}$		

TABLE 16 (*continued*)

Input *speakerLabel*	**Output reproduction gains**	`input_layouts`	`output_layouts`
U-090	M-030 = M-110 = $\sqrt{\frac{1}{2}}$		
U+090	M+030 = $\sqrt{\frac{1}{2}}$		
U-090	M-030 = $\sqrt{\frac{1}{2}}$		
U+110	U+110 = 1		
U-110	U-110 = 1		
U+110	U+135 = 1		
U-110	U-135 = 1		
U+110	U+045 = UH+180 = $\sqrt{\frac{1}{2}}$		
U-110	U-045 = UH+180 = $\sqrt{\frac{1}{2}}$		
U+110	M+110 = 1		
U-110	M-110 = 1		
U+110	M+135 = 1		
U-110	M-135 = 1		
U+110	M+030 = $\sqrt{\frac{1}{2}}$		
U-110	M-030 = $\sqrt{\frac{1}{2}}$		
U+135	U+135 = 1		
U-135	U-135 = 1		
U+135	U+110 = 1		
U-135	U-110 = 1		
U+135	U+045 = $\sqrt{\frac{1}{3}}$, UH+180 = $\sqrt{\frac{2}{3}}$	9+10+3	
U-135	U-045 = $\sqrt{\frac{1}{3}}$, UH+180 = $\sqrt{\frac{2}{3}}$	9+10+3	
U+135	U+045 = UH+180 = $\sqrt{\frac{1}{2}}$		
U-135	U-045 = UH+180 = $\sqrt{\frac{1}{2}}$		
U+135	M+135 = 1		
U-135	M-135 = 1		
U+135	M+110 = 1		
U-135	M-110 = 1		
U+135	M+030 = $\sqrt{\frac{1}{2}}$		

TABLE 16 (*continued*)

Input *speakerLabel*	**Output reproduction gains**	`input_layouts`	`output_layouts`
U-135	M-030 = $\sqrt{\frac{1}{2}}$		
U+180	U+180 = 1		
U+180	UH+180 = 1		
U+180	U+135 = U-135 = $\sqrt{\frac{1}{2}}$		
U+180	U+110 = U-110 = $\sqrt{\frac{1}{2}}$		
U+180	M+135 = M-135 = $\sqrt{\frac{1}{2}}$		
U+180	M+110 = M-110 = $\sqrt{\frac{1}{2}}$		
U+180	M+030 = M-030 = $\sqrt{\frac{1}{4}}$		
UH+180	UH+180 = 1		
UH+180	U+180 = 1		
UH+180	U+135 = U-135 = $\sqrt{\frac{1}{2}}$		
UH+180	U+110 = U-110 = $\sqrt{\frac{1}{2}}$		
UH+180	M+135 = M-135 = $\sqrt{\frac{1}{2}}$		
UH+180	M+110 = M-110 = $\sqrt{\frac{1}{2}}$		
UH+180	M+030 = M-030 = $\sqrt{\frac{1}{4}}$		
T+000	T+000 = 1		
T+000	U+045 = U-045 = U+135 = U-135 = $\sqrt{\frac{1}{4}}$		
T+000	U+030 = U-030 = U+110 = U-110 = $\sqrt{\frac{1}{4}}$		
T+000	U+045 = U-045 = UH+180 = $\sqrt{\frac{1}{3}}$		
T+000	U+045 = U-045 = M+135 = M-135 = $\sqrt{\frac{1}{4}}$		
T+000	U+030 = U-030 = M+110 = M-110 = $\sqrt{\frac{1}{4}}$		
T+000	M+030 = M-030 = M+135 = M-135 = $\sqrt{\frac{1}{4}}$		
T+000	M+030 = M-030 = M+110 = M-110 = $\sqrt{\frac{1}{4}}$		

TABLE 16 (*end*)

Input ***speakerLabel***	**Output reproduction gains**	`input_layouts`	`output_layouts`
T+000	M+030 = M-030 = $\sqrt{\frac{1}{4}}$		
B+000	B+000 = 1		
B+000	M+000 = 1		
B+000	M+030 = M-030 = $\sqrt{\frac{1}{2}}$		
B+045	B+045 = 1		
B-045	B-045 = 1		
B+045	M+030 = 1		
B-045	M-030 = 1		
LFE1	LFE1 = 1	9+10+3, 3+7+0	9+10+3, 3+7+0
LFE2	LFE2 = 1	9+10+3, 3+7+0	9+10+3, 3+7+0
LFE1	LFE1 = $\sqrt{\frac{1}{2}}$	9+10+3, 3+7+0	
LFE2	LFE1 = $\sqrt{\frac{1}{2}}$	9+10+3, 3+7+0	
LFE1	LFE1 = 1		

Bibliography

[1] F. Zotter and M. Frank (2012), *All-round ambisonic panning and decoding*, *Journal of the audio engineering society*, vol. 60, no. 10, pp. 807-820.

[2] V. Pulkki, (1997), *Virtual sound source positioning using vector base amplitude panning*, *Journal of the audio engineering society*, vol. 45, no. 6, pp. 456-466.

Attachment 1 to Annex 1 (informative)

Guide to corresponding parts of the specification to ADM Metadata

A1.1 ADM metadata across ITU-R ADM renderer

The purpose of the Table below is to provide a summary list of the key elements of the Renderer together with their locations in the specifications that are given in Annex 1. The specifications should be taken from the listed references.

ADM Metadata ***sub-element (attribute)[coordinate system]***	**Recommendation ITU-R BS.2076-1**	**Annex 1 in this Recommendation**
typeDefinition == "DirectSpeakers"	§ 5.4.3.1 Table 11	§ 8
speakerLabel		§ 8.2
position (azimuth, elevation, distance, screenEdgeLock)		§ 8
typeDefinition == "Matrix"	§ 5.4.3.2	§ 5.2.6.1.1 § 5.2.6.4
outputChannelIDRef	Table 12	§ 5.2.6.1.1
matrix → coefficient (gain, gainVar, phase, phaseVar, delay, delayVar)	Table 13	§ 5.6.4
input / outputPackFormatIDRef	§ 5.5.5.1	§ 5.2.6.1.1
encode / decodePackFormatIDRef		§ 5.2.6.1.1
typeDefinition == "Objects"	§ 5.4.3.3	§ 7
position (azimuth, elevation, distance, screenEdgeLock) [*polar*]	Table 14	§ 6.1 § 7 § 7.3.4
position (X, Y, Z, screenEdgeLock) [*cartesian*]	Table 15	§ 6.1 § 7 § 7.3.10
width, height, depth [*polar*]	Table 14	§ 7.3.8
width, height, depth [*cartesian*]	Table 15	§ 7.3.11
cartesian	Table 16	§ 7.3.1 § 7.3.2
gain		§ 7.3.1
diffuse		§ 7.3.1 § 7.4
channelLock (maxDistance)		§ 7.3.6
objectDivergence (azimuthRange, positionRange) [*polar*]		§ 7.3.7 § 7.3.1
objectDivergence (azimuthRange, positionRange) [*cartesian*]		§ 7.3.7 § 7.3.1
jumpPosition (interpolationLength)		§ 7.2
zoneExclusion → zone (minX, maxX, minY, maxY, minZ, maxZ, minElevation, maxElevation, minAzimuth, maxAzimuth)		§ 7.3.5 § 7.3.12
screenRef		§ 7.3.3
importance		§ 5.3.1 § 5.2.7.1.1
typeDefinition == "HOA"	§ 5.4.3.4	§ 9 § 5.2.7.3
equation	Table 17	§ 9.2
order		§ 9.1.1 § 9.3.1.2

ADM Metadata *sub-element (attribute)[coordinate system]*	Recommendation ITU-R BS.2076-1	Annex 1 in this Recommendation
degree		§ 9.1.1 § 9.3.1.2
normalization	§ 5.4.3.4	§ 9.1.2 § 9.3.1.5
nfcRefDist	Table 17	§ 9.2
screenRef		§ 9.2
typeDefinition == "Binaural"	§ 5.4.3.5	–

Attachment 2 to Annex 1 (informative)

An alternative virtual loudspeaker configuration

A2.1 Specification of alternative virtual loudspeaker configuration

An alternative VBAP virtual loudspeaker configuration to the one specified in § 6.1.3.1 describes the positions of virtual loudspeakers not located on the poles and their fold-down coefficients. The treatment of the ADM metadata remains the same as specified in the main body of this recommendation, with no additional metadata required. The alternative virtual loudspeaker positions and their fold-down coefficients are based on by-ear optimizations. Below is the description of this alternative virtual speaker configuration.

A2.1.1 Configuration Process

The configuration process follows the steps as described in § 6.1.3.1, with the exception of the second step which should be as follows:

2) Virtual speakers are determined by first looking up the tables defined in § A2.1.2. Each subsection under § A2.1.2 define virtual speaker configuration and their fold-downs for specific layout defined in Recommendation ITU-R BS.2051-2.

The other configuration process steps, step (1) and steps (3) through (6), remain as described in § 6.1.3.1.

A2.1.2 Virtual Speakers and Fold-down tables

In the Tables below, the virtual loudspeakers (specified as azimuth and elevation) are in the first row and the physical loudspeakers in the first column. The virtual loudspeaker locations have the same nominal and real positions. The table shows the fold-down coefficients going from virtual loudspeakers to physical loudspeakers.

System A: 0+2+0

For the system A:0+2+0, the method based on a downmix from system B:0+5+0 to system A:0+2+0 as described in § 6.1.2.4 is used. For obtaining the 0+5+0 channels, virtual loudspeakers for the system B:0+5+0 are used as described below.

System B: 0+5+0

	−45, 45	45, 45	−135, 45	135, 45	−45, −45	45, −45	−135, −45	135, −45
M+030		1.0				1.0		
M-030	1.0				1.0			
M+000								
LFE1								
M+110			0.3162	0.9486			0.3162	0.9486
M-110			0.9486	0.3162			0.9486	0.3162

System C: 2+5+0

	−135, 30	135, 30	−45, −45	45, −45	−135, −45	135, −45
M+030				1.0		
M-030			1.0			
M+000						
LFE1						
M+110	0.3162	0.9486			0.3162	0.9486
M-110	0.9486	0.3162			0.9486	0.3162
U+030						
U-030						

System D: 4+5+0

	−45, −45	45, −45	−110, −45	110, −45
M+030		1.0		
M-030	1.0			
M+000				
LFE1				
M+110			0.3162	0.9486
M-110			0.9486	0.3162
U+030				
U-030				
U+110				
U-110				

System E: 4+5+1

This layout has both upper and bottom loudspeakers. No need for virtual loudspeakers as the hull is complete.

System F: 3+7+0

	−135, 30	135, 30	−45, -45	45, −45	−135, −45	135, −45
M+000						
M+030				1.0		
M-030			1.0			
U+045						
U-045						
M+090						
M-090						
M+135		0.7071				1.0
M-135	0.7071				1.0	
UH+180	0.7071	0.7071				
LFE1						
LFE2						

System G: 4+9+0

	−45, −45	45, −45	−135, −45	135, −45
M+030		1.0		
M-030	1.0			
M+000				
LFE1				
M+090				
M-090				
M+135				1.0
M-135			1.0	
U+045				
U-045				
U+135				
U-135				
M+SC				
M-SC				

System H: 9+10+3

Contains loudspeakers in both upper and bottom hemisphere; the hull is complete and so no need for virtual loudspeakers.

System I: 0+7+0

	−45, 45	45, 45	−135, 45	135, 45	−45, −45	45, −45	−135, −45	135, −45
M+030		1.0						
M-030	1.0				1.0	1.0		
M+000								
LFE1								
M+090								
M-090								
M+135				1.0				1.0
M-135			1.0				1.0.	

System J: 4+7+0

	−45, −45	45, −45	−135, −45	135, −45
M+030		1.0		
M-030	1.0			
M+000				
LFE1				
M+090				
M-090				
M+135				1.0
M-135			1.0	
U+045				
U-045				
U+135				
U-135				