SMPTE REGISTERED DISCLOSURE DOCUMENT

MDA Program Specification

The attached document is a Registered Disclosure Document prepared by the proponent identified below. It has been examined by the appropriate SMPTE Technology Committee and is believed to contain adequate information to satisfy the objectives defined in the Scope, and to be technically consistent.

This document is NOT a Standard, Recommended Practice or Engineering Guideline, and does NOT imply a finding or representation of the Society.

Every attempt has been made to ensure that the information contained in this document is accurate. Errors in this document should be reported to the proponent identified below, with a copy to eng@smpte.org.

All other inquiries in respect of this document, including inquiries as to intellectual property requirements that may be attached to use of the disclosed technology, should be addressed to the proponent identified below.

Proponent contact information:

Scott Smyers
DTS, Inc.
130 Knowles Drive, Suite B
Los Gatos, CA 95032

Email: scott.smyers@dts.com

Approved
August 23, 2016

Table of Contents

Conventions

All sections are normative, unless otherwise indicated.

Pseudo-code and property names use font style courier new.

The expressions MAY, NEED NOT, SHALL, SHALL NOT, SHOULD, and SHOULD NOT indicate normative behavior ISO/EIC Directives, Part 2.

VERBAL FORM	SEMANTICS
MAY	It is allowed
NEED NOT	It is not required that
SHALL	Is required that
SHALL NOT	Is required to be not
SHOULD	It is recommended that
SHOULD NOT	It is not recommended that

Introduction (Informative)

The MDA Program, or simply Program hereafter, is a self-contained object-based audio program. As such it consists of a collection of audio objects, each combining an audio waveform with metadata. The metadata indicates, for instance, when the object occurs on the Program timeline or where it is positioned within the soundfield. It is used to control the mapping of the audio object waveform to output loudspeakers at playback.

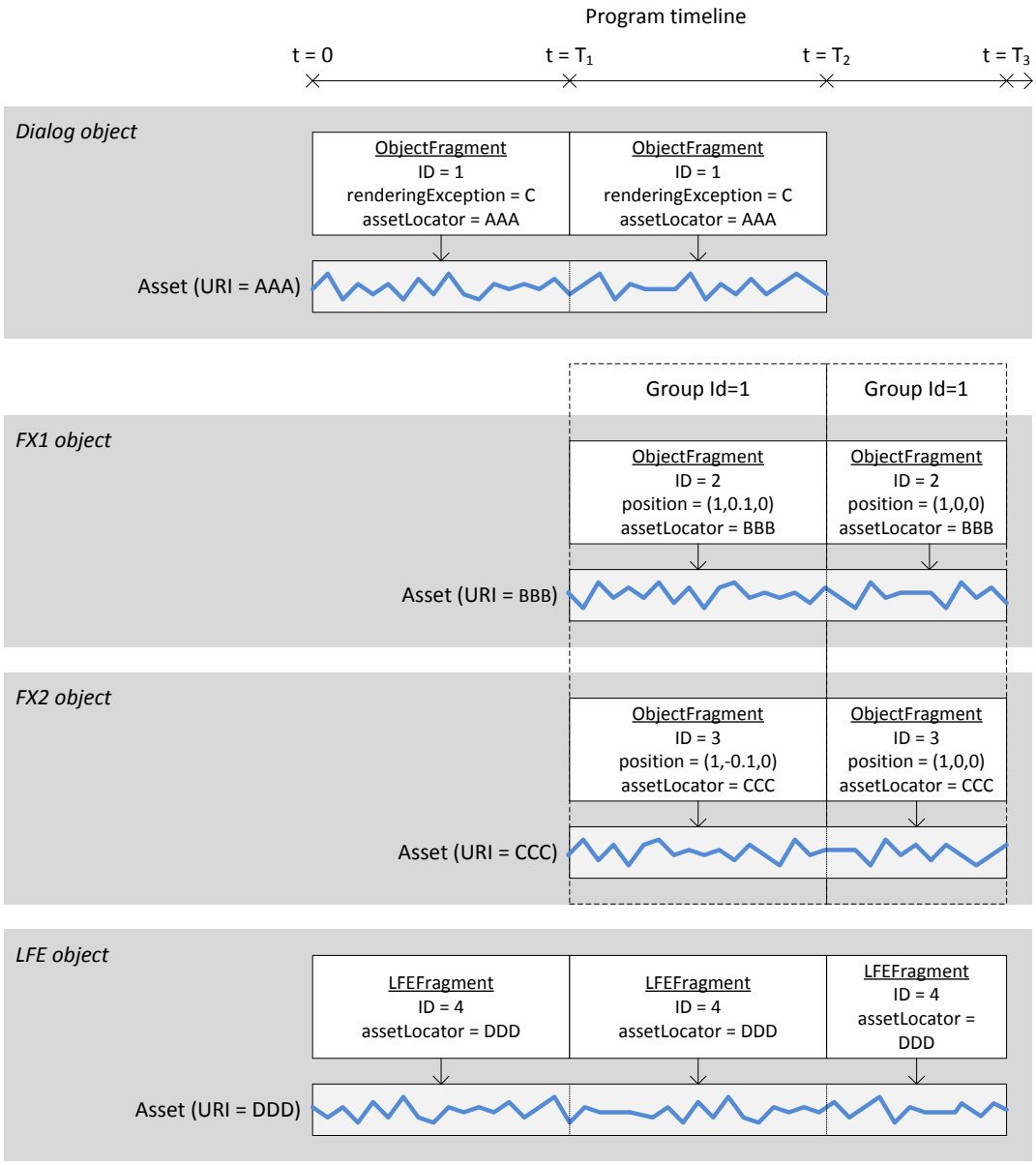

Figure 1 – Sample Program. Only a subset of all Fragment properties are shown

Figure 1 depicts a Program that consists of 4 audio objects: Dialog, FX1, FX2 and LFE. While the LFE object exists from t = 0 to t = T_3, the dialog object exists only from t = 0 to t = T_2, and the FX2 and FX2 objects from t = T_1 to t = T_3. The Program object model allows for any number of audio objects to overlap at any point in time, and an audio object can be as short as a sample or as long as the program. Each audio object is assigned an identifier that is unique within the scope of the Program.

The metadata associated with each object is divided into Fragments, each corresponding to a period of time during which the metadata is static. To simplify the Program structure, Fragment boundaries are aligned.

Two kinds of Fragments, and hence audio objects, are defined:

- An ObjectFragment corresponds to an object associated with a spatial locus. For instance, the position of the FX1 and FX2 objects in Figure 1 changes from t = T_1 to t = T_3. This spatial locus is used to determine the loudspeakers that will output the waveform associated with the object. It is also possible to instruct that an object waveform be routed through a specific loudspeaker, if present.

- An LFEFragment corresponds to an object whose waveform is intended for routing to a Low Frequency Effect (LFE) channel, and is therefore not associated with a spatial location.

Each Fragment references a sequence of audio samples, i.e. the object waveform, within an underlying asset identified using a Uniform Resource Identifier (URI). Depending on applications, the asset can be carried alongside the Fragment metadata or be remote. Multiple Fragments can reference the same audio samples within a single asset.

As illustrated by the FX1 and FX2 objects, Fragments can be combined into a Group, which logically groups the two ObjectFragments and contains metadata common to them . The object model also allows Fragments to be combined into a Switch, which indicates that only one of the Fragments is rendered at any given time. Groups and Switches are recursive entities that can themselves contain Groups and Switches. From the perspective of the object model, Fragments, Groups and Switches are all subclasses of the Entity class, which represents arbitrary entities of the Program timeline.

In order to specify unambiguously how object metadata is used to map object waveforms to loudspeaker outputs, i.e. rendered, a Reference Renderer is fully specified in Section 9. The Reference Renderer uses the Vector Base Amplitude Panning formalism (VBAP), which was introduced by Pulkki et al. and has since been extensively studied. VBAP is an extension of the familiar tangent law for pair-wise panning to three-dimensional speaker configurations. Specifically, given a loudspeaker triplet on the unit sphere and a point source object located within the spherical triangle defined by the loudspeakers, the contribution of the object waveform to each of the loudspeaker is determined by the coordinates of the object within the linear basis formed by the three loudspeakers (see Figure 2). Objects with a finite extent can be rendered as a collection of point sources. More complex loudspeaker configurations can be decomposed into multiple speaker triplets.

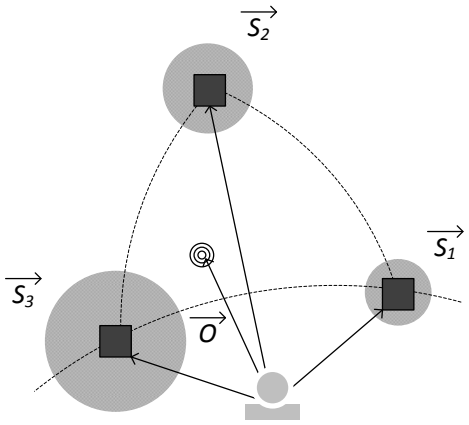

Figure 2 – Rendering audio objects using VBAP. The shaded areas show the relative output power at each of the speakers and are determined by expressing the object vector in the basis formed by the three loudspeakers.

To support a range of applications within its stated scope, the Program object model is designed to be flexible, e.g. the number of simultaneous Fragments is not limited, and offers multiple extension points. Applications are therefore expected to constrain or extend the object model to suit their specific requirements. Similarly, this specification does not define a concrete representation of the Program, and mappings to bitstream structures and transmission mechanisms are left to other documents.

1 Scope

This document specifies the object model and reference renderer for the MDA Program. The MDA Program is a self-contained representation of an object-based soundfield designed for linear content. It is specified independently of transport mechanisms.

2 Normative References

Internet Engineering Task Force (IETF) (January 2005). RFC 3986 – Uniform Resource Identifier (URI): Generic Syntax

ISO/EIC Directives, Part 2. Edition 6.0, 2011-04

OMG, Object Constraint Language (OCL), Version 2.3.1, http://www.omg.org/spec/OCL/2.3.1/PDF

OMG, Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF

3 Timeline

A Program defines a sample-accurate timeline onto which Entity instances (see Section 6.6) are placed.

Positions on the timeline SHALL be expressed as integer multiples of the inverse of the Program audio sample rate (see Section 6.5.2), i.e. as an integer number of audio samples.

The origin of the timeline (t=0) is arbitrary.

4 Audio Object

An Audio Object is the sequence of all Fragment instances (see Section 6.6) with the same `id`, ordered as they appear on the timeline.

Two Fragment instances belong to the same Audio Object if and only if they have the same `id` value.

5 Coordinate System

This specification uses the Cartesian coordinate system illustrated in Figure 3 and specified as follows:

- The listener is located at the origin O=(0,0,0), facing the front of the room;
- The positive z-axis is perpendicular to the floor of the room, and directed to the ceiling;
- The positive y-axis is directed towards the front of the room;
- The positive x-axis is directed to the right of the listener;
- Loudspeakers lie on the unit sphere S; and
- The unit circle in the x-y plane is the locus of traditional horizontal two-dimensional loudspeaker configurations.

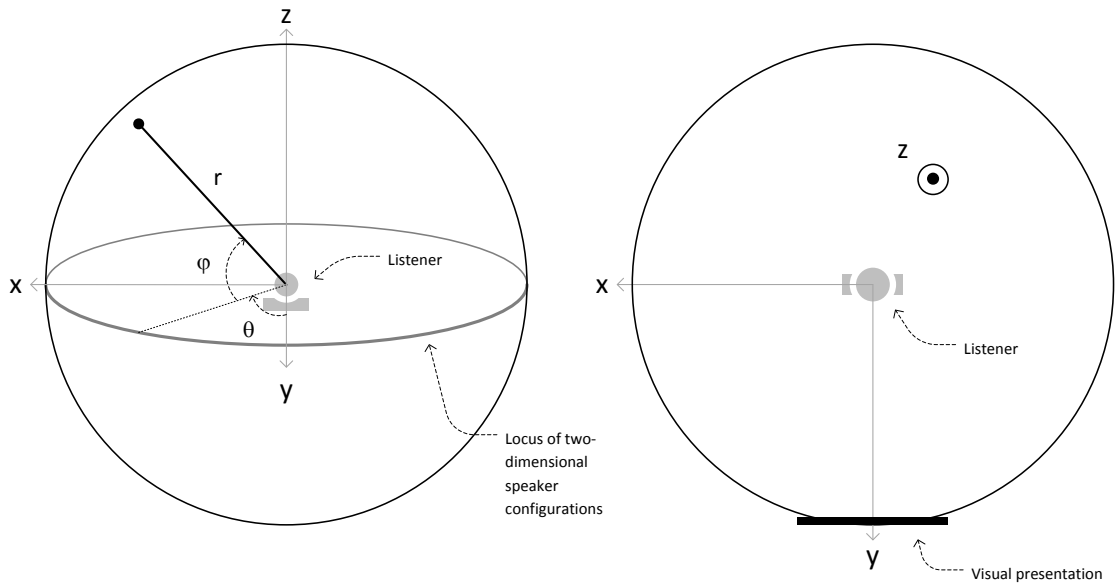

Figure 3 – Program Coordinate System

For convenience, the following modified spherical coordinate system is also defined.

$$x = \rho \sin\theta \cos\varphi$$
$$y = \rho \cos\theta \cos\varphi$$
$$z = \rho \sin\varphi$$

The symbols ρ (*rho*), θ (*theta*) and φ (*phi*) denote the radius, azimuth and elevation of the object, respectively.

6 Object Model

6.1 General

The Program object model is specified using a combination of prose, UML as specified in OMG Unified Modeling Language (UML), and OCL as specified in OMG Object Constraint Language (OCL). The prose shall take precedence over the UML and OCL notations in case of conflict.

If an optional property is absent, its value shall be unspecified unless a default value is provided, in which case its value shall be the default value.

Values that are identified as reserved SHALL NOT be used in this version of the specification and, if present in a Program, SHALL be ignored by implementation conforming to this version of the specification.

The notation `#SymbolName` refers to the URI constant with symbol `SymbolName`.

6.2 Namespace

UML elements defined herein SHALL be members of the MDA Package with the namespace specified in Table 1.

Table 1 – MDA Object Model Namespace

Symbol	URI
mdaroot	http://mdaif.org
mdacore	<mdaroot>/core/1.0/

6.3 Versioning

The namespace specified in Section 6.2 SHALL only be associated with Program instances that conform to this specification.

Program instances using specifications that modify the latter, including future versions of this specification, SHALL use a different namespace.

6.4 Program

6.4.1 General

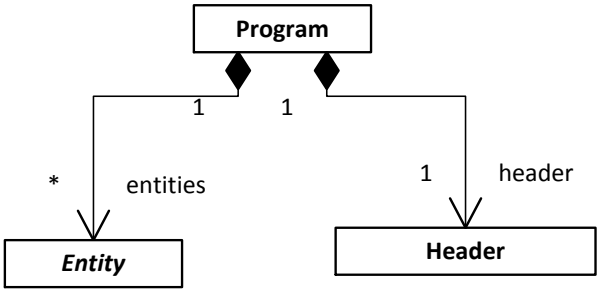

Figure 4 – Program Model

A Program instance is a single complete Program, which contains all information necessary for reproduction.

6.4.2 header

The `header` property SHALL contain information applicable to the Program as a whole.

Two Program instances SHALL NOT have identical `header.programURI` values unless the two instances are identical.

6.4.3 entities

The `entities` property contains all entities associated with the Program.

No two Entity instances with the same `id` property value SHALL overlap on the timeline.

All Entity instances with identical `id` property values SHALL be of the same concrete subclass.

The start or end offset of an Entity instance SHALL NOT belong to the open interval bounded by the start and end offsets of another Entity.

6.5 Header

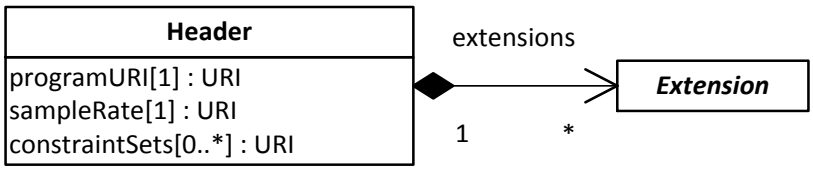

Figure 5 – Header Model

6.5.1 programURI

The `programURI` property uniquely identifies the Program instance.

The `programURI` property shall consist of no more than 64 characters, with the meaning of character specified in IETF RFC 3986.

6.5.2 sampleRate

The `sampleRate` property indicates the audio sampling rate of the Program.

Note: Section 6 defines common values for audio sampling rates.

6.5.3 constraintSets

The `Program` object model MAY be constrained and extended by multiple applications, each potentially defining additional metadata properties and applying a set of constraints beyond those specified herein. Implementations can use the `constraintSets` to rapidly determine whether they are capable of processing a `Program`.

Each item of the `constraintSets` property SHALL be unambiguously associated with a collection of normative provisions (beyond those specified herein) to which the `Program` conforms.

No two items of the `constraintSets` property SHALL be equal.

6.5.4 extensions

The `extensions` property allows application-specific metadata (contained in a concrete subclass of the Extension class) to be associated with the `Program`.

6.6 Entity

6.6.1 General

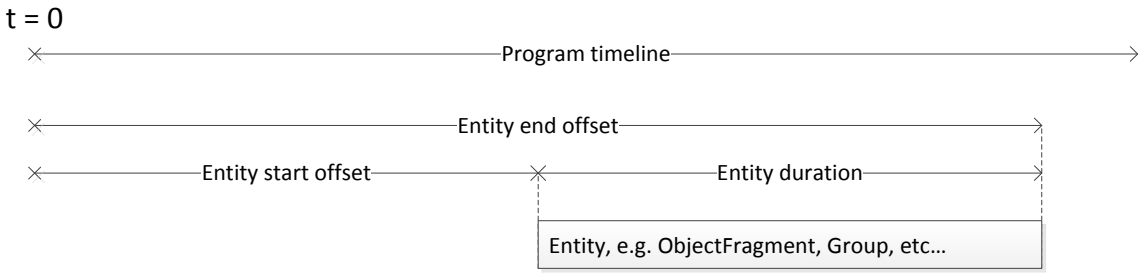

Figure 6 – Positioning an Entity instance on the Program Timeline

As illustrated in Figure 6, each `Entity` instance SHALL be associated with a start and end offset on the timeline, relative to the origin of the timeline.

The end offset SHALL be larger than the start offset.

The duration of the `Entity` instance is the difference between the end and start offsets.

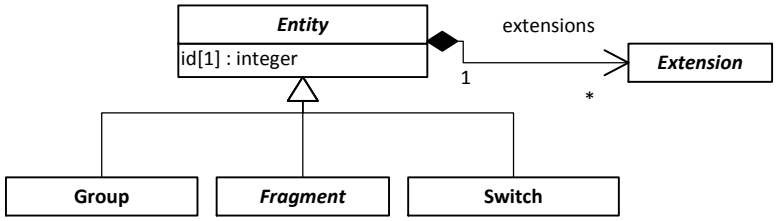

Figure 7 – Entity Model

This specification defines a number of concrete subclasses of the `Entity` class, and future revisions MAY define additional ones.

6.6.2 id

The `id` property allows multiple related `Entity` instances to be uniquely linked within the scope of the Program.

The value of the `id` property SHALL belong to the range [0, 2^{32}-1].

6.6.3 extensions

The `extensions` property allows application-specific metadata (contained in concrete subclasses of the Extension class defined by the application) to be associated with an `Entity`.

6.7 Group

6.7.1 General

A `Group` instance is a logical group of `Entity` instances, all of which are intended to be rendered.

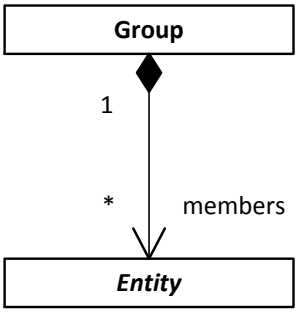

Figure 8 – Group Model

The start offset of a `Group` instance SHALL be the smallest start offset of all `Entity` instances it contains.

The end offset of a Group instance SHALL be the largest end offset of all `Entity` instances it contains.

Note: Section 9.3.1 specifies that all `Entity` instances within a `Group` instance are rendered.

6.8 Switch

6.8.1 General

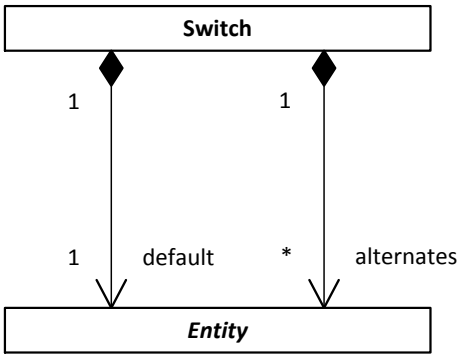

Figure 9 – Switch Model

A Switch instance is a logical group of `Entity` instances, only one of which is intended to be rendered.

The start offset of a `Switch` instance SHALL be the smallest start offset of all `Entity` instances it contains.

The end offset of a `Switch` instance SHALL be the largest end offset of all Entity instances it contains.

Note: Section 9.3.1 specifies that only one Entity instance within a `Switch` instance is rendered. This single instance is that referenced by the `default` property unless otherwise specified by the rendering context.

6.9 Fragment

6.9.1 General

A `Fragment` represents an `Entity` that spans a specified interval of the Program timeline.

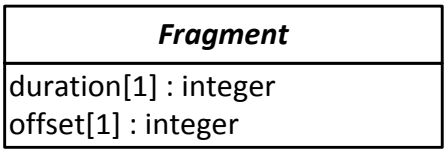

Figure 10 – Fragment Model

The start offset of a `Fragment` instance shall be the value of its `offset` property.

The end offset of a `Fragment` instance shall be the sum of the values of its `offset` and `duration` properties.

6.9.2 offset property

The value of the `offset` property SHALL be in the range [0, 2^{64}-1].

6.9.3 duration property

The value of the `duration` property SHALL be in the range [0, 2^{16}-1].

6.10 MonoSourceFragment

6.10.1 General

A `MonoSourceFragment` represents a sound source that emits a sequence of monaural audio samples, and whose characteristics are static over the duration of the `MonoSourceFragment`.

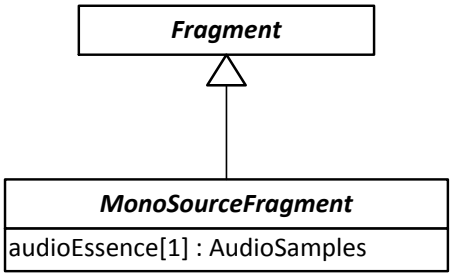

Figure 11 – MonoSourceFragment Model

6.10.2 audioEssence

The `audioEssence` property references the audio samples associated with the sound source.

The `audioEssence.assetURI` property shall reference a sequence of monaural audio samples.

6.11 ObjectFragment

6.11.1 General

An `ObjectFragment` instance corresponds to a collection of extended sound sources emitting the same sequence of monaural audio samples.

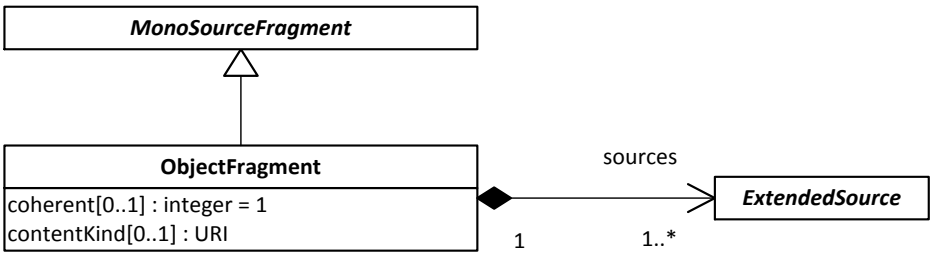

Figure 12 – ObjectFragment Model

6.11.2 coherent

The `coherent` property indicates whether the source is rendered coherently (`coherent` equal to 1) or diffusively (`coherent` equal to 0) across its extent.

The value of `coherent` SHALL be in the set {0, 1}.

6.11.3 contentKind

The `contentKind` property indicates the audio content of the `ObjectFragment`.

Table 2 defines a number of values that MAY be present, but other specifications MAY define additional values.

Implementations MAY ignore values they do not recognize.

6.11.4 sources

The `sources` property contains the extended sound sources associated with the `ObjectFragment` instance.

6.12 ExtendedSource

6.12.1 General

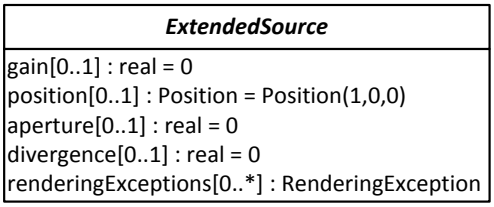

Figure 13 – ExtendedSource Model

An `ExtendedSource` instance represents a sound source positioned within the soundfield. The extent of the sound source is parameterized by aperture and divergence values.

6.12.2 gain

The `gain` property specifies a gain value that allows both the relative and absolute loudness of extended sources to be adjusted.

The value of `gain` SHALL be in the set {-∞, -410, … , 100}/4 and expressed in dB.

6.12.3 position

The `position` property is the position of the sound source.

6.12.4 aperture

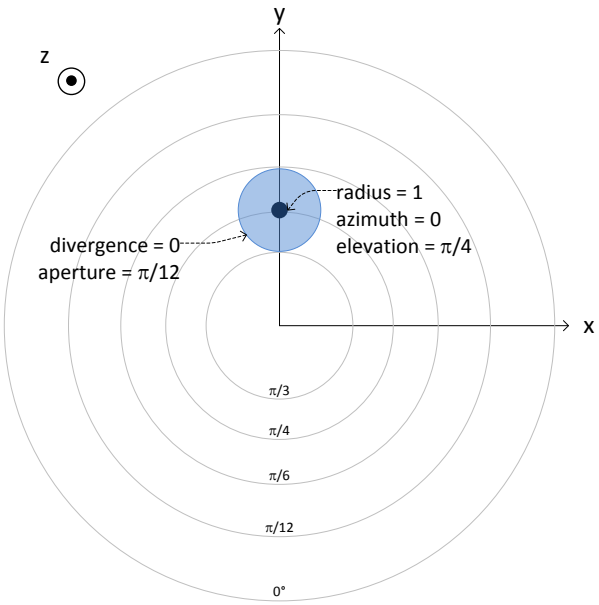

Figure 14 – π/4 aperture shown using stereographic projection.

The `aperture` property is the nominal extent of the sound source before the application of divergence (see Section 6.12.5). The nominal extent is the spherical cap defined by the intersection of (i) the sphere whose radius is the segment between the position of the sound source and the origin and (ii) the infinite right circular solid cone whose apex is at the origin and its aperture equal to twice the `aperture` property value. Figure 14 illustrates a π/4 rad aperture using stereographic projection[1].

The `aperture` property SHALL be in radians and in the set π/255 · {0, 1... 255}.

Note: An aperture value of π indicates that the source extent covers the entire sphere.

[1] http://en.wikipedia.org/wiki/Stereographic_projection

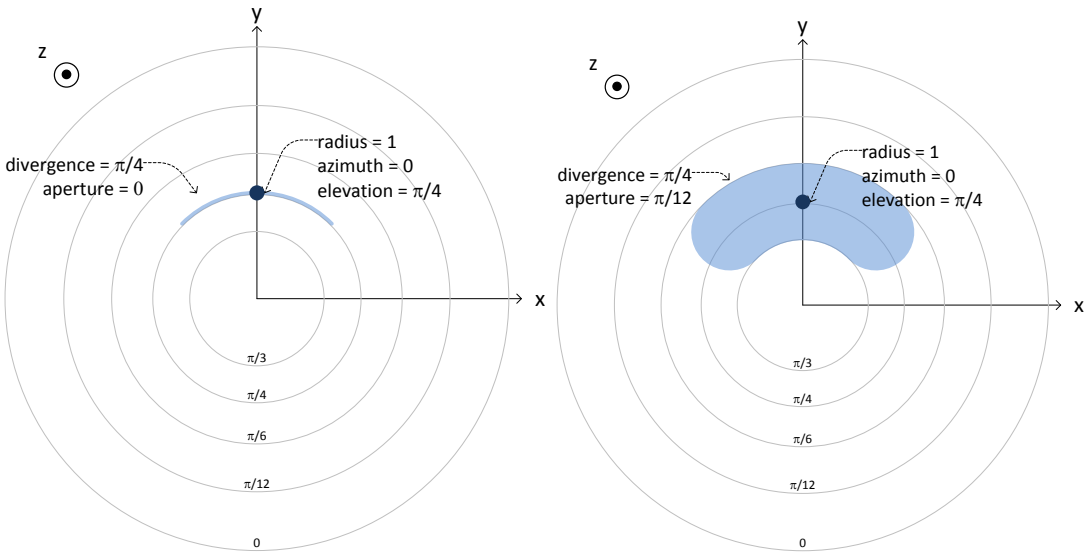

Figure 15 – Combining Aperture and Divergence (stereographic projection)

6.12.5 divergence

The `divergence` property is the half angle of the horizontal arc (latitude) centered at the sound source location over which the nominal extent is spread to yield the source extent.

The `divergence` property SHALL be in radians and in the set $\pi/255 \cdot \{0, 1... 255\}$.

Figure 15 illustrates a $\pi/4$ rad divergence applied to a $\pi/12$ rad and 0 rad aperture.

6.12.6 renderingExceptions

The `renderingExceptions` property allows the default renderer behavior to be altered when rendering into specified soundfields, i.e. target rendering configurations. `RenderingExceptions` are defined in Section 6.15 and their semantics in the MDA reference renderer is specified in Section 9.3.2.

A given value of the `renderingException.targetConfiguration` property, including the absence of such property, SHALL occur at most once in the `renderingExceptions` property.

6.13 LFEFragment

6.13.1 General

An `LFEFragment` instance represents a low-frequency effects sound source.

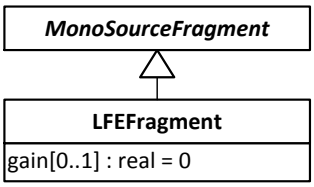

Figure 16 – LFEFragment Model

6.13.2 gain

The `gain` property specifies a gain that is applied to the source audio samples. It allows the relative gain of sources to be adjusted without modifying the latter, e.g. to alter the mix creatively.

The value of `gain` SHALL be in the set {-∞, -410, … , 100}/4 and expressed in dB.

6.14 AudioSamples

6.14.1 General

An `AudioSamples` instance selects a sequence of audio samples from an underlying asset.

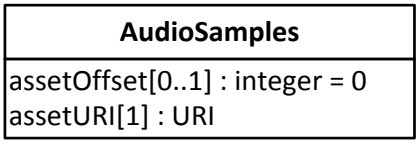

Figure 17 – AudioSamples Model

6.14.2 assetOffset

The `assetOffset` property shall indicate the offset of the first audio sample selected within the asset referenced by the `assetURI` property.

The value of the `assetOffset` property SHALL be in the range [0, 2^{64}-1] in units of audio samples.

6.14.3 assetURI

The `assetURI` property shall reference a sequence of audio samples. The nature of the reference is left to specifications mapping the Program to concrete representations.

No two sequences of audio samples SHALL have identical `assetURI` values unless they are identical.

6.15 RenderingException

6.15.1 General

The `RenderingException` class allows the default rendering behavior to be overridden in the presence of the target rendering configurations listed in the `targetConfiguration` property.

Each concrete subclasses of the `RenderingException` class SHALL specify a specific rendering behavior.

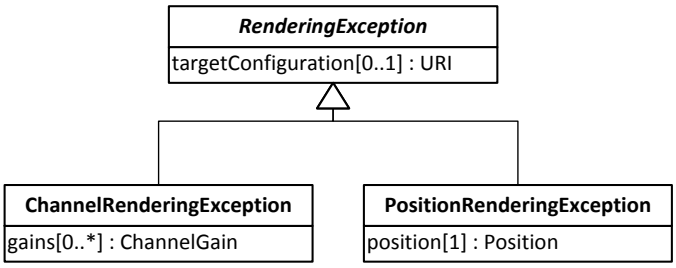

Figure 18 – RenderingException Model

6.15.2 targetConfiguration property

The `targetConfigurations` property indicates the target rendering configuration to which the `RenderingException` instance shall apply (see Section 9.3.2).

The absence of a `targetConfiguration` value property indicates that the `RenderingException` instance applies to any target rendering configuration.

6.16 PositionRenderingException

The `PositionRenderingException` allows the author to indicate an alternate position for the object, as specified by its `position` property, when rendering to specified target rendering configurations.

6.17 ChannelRenderingException

6.17.1 General

The `ChannelRenderingException` allows the author to explicitly specify channels to which the object waveform is routed, when rendering to specified target rendering configuration.

6.17.2 gains

Each item of the `gains` property specifies an output channel (specified by its `channel` property) to which the audio samples of the object are routed after applying a gain (specified by its `gain` property).

If the `gains` property contains no elements, the rendering of the `ObjectFragment` instance is suppressed.

Two different members of the `gains` property SHALL NOT have identical `gains.channel` values.

6.18 ChannelGain

6.18.1 General

A `ChannelGain` instance associates a gain with an audio channel.

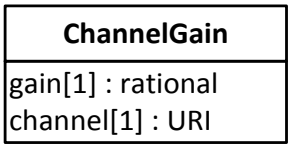

Figure 19 – ChannelGain Model

6.18.2 gain property

The value of the `gain` property SHALL belong to the set {-255/4, -254/4, … 0} and SHALL be in units of dB.

6.18.3 channel property

The `channel` property indicates the audio channel to which the gain property applies.

6.19 Position

6.19.1 General

A `Position` instance represents a position in the coordinate system specified in Section 5.

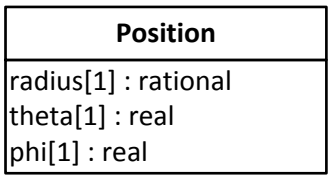

Figure 20 – Position Model

6.19.2 radius

The value of the `radius` property corresponds to the ρ coordinate of the position, i.e. the radial distance.

The value of the `radius` property SHALL be in the set {0, 1, 2... 2046, 2048, 2049,… 4094}/2047.

6.19.3 Theta property

The value of the `theta` property corresponds to the θ coordinate of the position, i.e. the azimuth.

The value of the `theta` property SHALL be in the set $\pi/2048 \cdot \{-2048, -2047... 2047\}$.

6.19.4 Phi property

The value of the `phi` property corresponds to the φ coordinate of the position, i.e. the elevation.

The value of the `phi` property SHALL be in the set $\pi/2046 \cdot \{-1023, -1022... 1023\}$.

7 URI Constants

Table 2 SHALL define URI constants used by this specification.

Table 2 – URI Constants

Symbol	URI	Definition
Content Kind		
ContentKindDialog	<mdacore>/labels/content-kind/dialog	Any spoken words or narration, including unidentifiable human vocalizations e.g. crowd walla or cheers.
ContentKindEffects	<mdacore>/labels/content-kind/effects	Sound effects of any kind, including ambience and Foley.
ContentKindMusic	<mdacore>/labels/content-kind/music	Includes all underlying music elements, e.g. score and songs, as well as diegetic music from sources integral to the story, e.g. a concert or a radio.
Sampling Rate		
48000Sampling	<mdacore>/labels/sample-rate/48000Hz	Audio sampled at 48,000 Hz
96000Sampling	<mdacore>/labels/sample-rate/96000Hz	Audio sampled at 96,000 Hz

8 Basic Data Types

8.1 General

This specification makes use of the following data types. No assumption is made on their encoding.

8.1.1 Real

The real data type shall consist of all real numbers.

8.1.2 Rational

The rational data type shall consist of all rational numbers.

8.1.3 Integer

The integer data type shall consist of all integer numbers.

8.1.4 URI

The URI data type shall be a URI as specified in IETF RFC 3986.

9 Reference Renderer

9.1 Overview

The reference renderer, illustrated in Figure 21, operates on successive offsets in the Program timeline. With the exception of two functions whose definition is outside of the scope of this specification, i.e. `ApplyDecorrelation` and `ApplySmoothing`, the reference renderer is stateless.

The reference renderer uses the rendering configuration information contained in a `Configuration` structure (see Section 9.2) to render to speaker signals the `MonoSourceFragment` instances that exists at each offset.

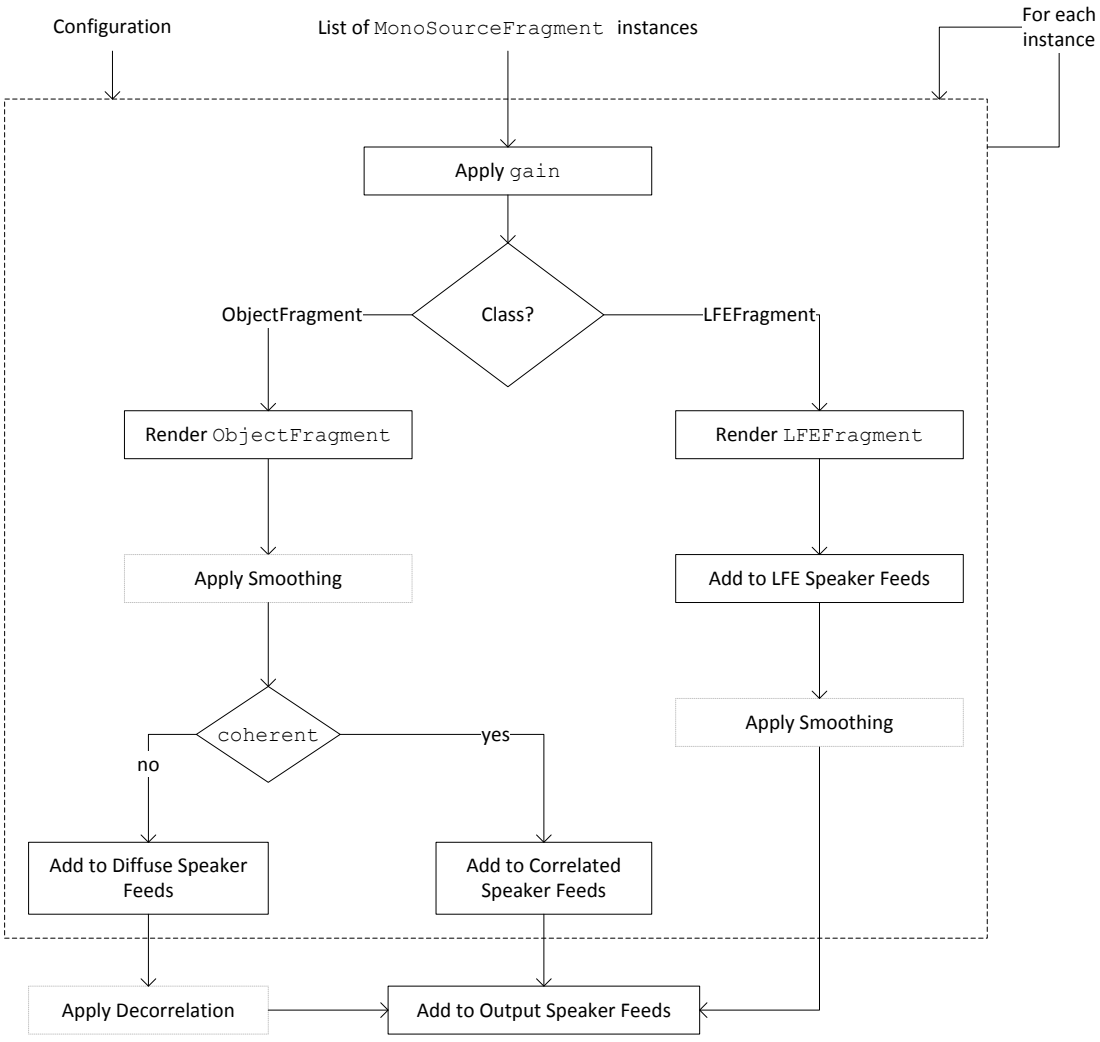

Figure 21 – Reference rendering process overview.
The dotted line and gray-filled processes are not specified and left to applications.

9.2 Configuration

9.2.1 General

The `Configuration` structure holds the information necessary for the reference renderer to render `MonoSourceFragment` instances to loudspeakers. With the exception of `virtualSources`, which is generated by the rendering process, all other properties are provided as input to the rendering process.

```
struct Configuration {

 URI                 soundfieldName;
 NormalSpeaker   speakers[];
 LFESpeaker                      lfe;
 Patch                    patches[];
 VirtualSource   virtualSources[];

};
```

9.2.2 soundfieldName

If not `NULL`, `Configuration.soundfieldName` SHALL identify the speaker configuration embodied by the combination of `Configuration.speakers` and `Configuration.lfe` (see Section 9.2.3).

As specified in Section 6.12.6 and Section 9.3.2, the `Configuration.soundfieldName` is used to determine whether or not a `RenderingException` instance applies.

9.2.3 speakers

Two kinds of loudspeakers can be specified, normal and Low Frequency Effects (LFE), corresponding to `Configuration.speakers` and `Configuration.lfe`, respectively.

```
struct Speaker {

 URI name;

};

Struct NormalSpeaker : Speaker {

 Position       position;
 float     mixcoefs[];

}

Struct LFESpeaker : Speaker {
}
```

The combination of `Configuration.speakers` and `Configuration.lfe` is the set of loudspeakers to which `MonoSourceFragment` instances are rendered.

`NormalSpeaker.position.radius` SHALL be equal to 1.

No two elements of `speakers` SHALL have identical `position` properties.

No two elements of `speakers` SHALL have identical `name` properties.

For a given index m in the `mixcoefs` array:

- if `speakers[m].mixcoefs[m]` is equal to 1, all other `speakers[m].mixcoefs[n ≠ m]` SHALL be equal to 0, and the speaker is a physical speaker; and
- otherwise `speakers[m].mixcoefs[m]` shall be equal to 0, and the speaker is a virtual speaker and all `n ≠ m` for which `speakers[m].mixcoefs[n]` is not equal to zero shall correspond to a physical speaker.

9.2.3.1 name

If present, the Speaker.name property identifies the speaker within the speaker configuration indicated by the soundfieldName property.

Note: This name is used when processing ChannelRenderingExceptions (see Section 9.3.2).

9.2.3.2 position

The NormalSpeaker.position is the position of the speaker expressed in the coordinate system of Section 6.19.

9.2.3.3 mixcoefs

The mixcoefs property is an array of gain values, each associated with the Speaker instance in Configuration.speakers with the same index.

9.2.4 patches

```
typedef int   Patch[3];
```

A `Patch` is a triplet of distinct integers, each an index of a `Speaker` instance in `Configuration.speakers`.

Each patch in `Configuration.patches` shall satisfy the following conditions:

- the position of the three `Speaker` instances shall define a plane, i.e. the speaker positions are linearly independent; and
- the half-space defined by this plane and the origin shall contain all speakers.

Note: Not all members of `Configuration.speakers` are referenced from `Configuration.patches`.

9.2.5 virtual Sources

```
struct VirtualSource {
 Position       position;
 float          gains[];
};
```

To render extended sources, the renderer uses a collection of point sources uniformly sampled over the unit sphere (the virtual sources stored in `Configuration.virtualSources`).

`Configuration.virtualSources` shall be computed according to `ComputeVirtualSources()`.

```
ComputeVirtualSources(Configuration config) {

 int kPhiHalfDivs = 32;
 int kThetaDivs = kPhiHalfDivs * 4;

 for(int i = -kPhiHalfDivs; i <= kPhiHalfDivs; i++) {

  float phi = i * PI/2.0f/kPhiHalfDivs;

  int n = (i == -kPhiHalfDivs  || i == kPhiHalfDivs) ? 1 :
         floor(kThetaDivs * cos(phi));

  for(int j = 0; j < n ; j++) {

   float theta = 2 * PI * j / n;

   VirtualSource vs;

   vs.position.radius = 1;
   vs.position.theta = theta;
   vs.position.phi = phi;

   RenderPointSource(config, vs.position, vs.gains);

   config.virtualSources.append(vs);
  }
 }
}
```

The `kPhiHalfDivs` and `kThetaDivs` constants determine the number of virtual sources present on a given meridian and on the equator of the unit sphere, respectively.

9.3 Rendering Process

9.3.1 ProcessOffset

9.3.1.1 General

For every successive offset T within Program timeline, the reference renderer shall execute `ProcessOffset(config, fragments, speakerOutput, lfeOutput)`, with:

- `config` as specified in Section 9.2.
- `fragments` consisting of all `MonoSourceFragment` instances for which `MonoSourceFragment.start` ≤ T < `MonoSourceFragment.start` + `MonoSourceFragment.duration`, with the exception of `MonoSourceFragment` instances referenced by `Switch.alternates` unless specified otherwise by the application.
- Each element of `speakerOutput` corresponding to the signal to be output by the element in `config.speakers` with the same index
- `lfeOutput` corresponding to the signal to be output by the LFE loudspeaker, if any.

```
ProcessOffset(  Configuration config,
                     MonoSourceFragment fragments[],
                     float &speakerOutput[],
                     float &lfeOutput ) {

 speakerOutput.fill(0);
 lfeOutput = 0;

 float coherentOutput[speakerOutput.size()];
 float diffuseOutput[speakerOutput.size()];

 foreach(fragment in fragments) {

  float sample = GetCurrentAudioSample(fragment.audioSamples);

  // Render

  if (fragment intstanceOf ObjectFragment) {

   float gains[speakerOutput.size()];

   gains = RenderObjectFragment(config, fragment);

   // compute the absolute gain to be applied

   float totalGain = 0;

   foreach(source in fragment.sources) {

    totalGain += pow(10, source.gain / 20);

   }

   gains *= totalGain;

   ApplySmoothing(config, fragment.id, gains);

   if (fragment.coherent) {
    coherentOutput += sample * gains;
   } else {
    diffuseOutput += sample * gains;
```

```
    }
   } else if (fragment intstanceOf LFEFragment){
    if (config.lfe != NULL) {
     float gain = pow(10, fragment.gain / 20);
     ApplySmoothing(config, fragment.id, gain);
     lfeOutput += sample * gain;
    }
   }
  }
 ApplyDecorrelation(config, diffuseOutput);
 speakerOutput = coherentOutput + diffuseOutput;
}
```

`GetCurrentAudioSample(AudioSamples audioSample)` SHALL return the value of the audio sample of `audioSample` associated at the current timeline position.

9.3.1.2 ApplySmoothing

`ApplySmoothing(Configuration config, integer id, float &gains[])`:

- SHOULD minimize transients in elements of `gains` for the `MonoSourceFragment` identified by `id` across calls to `ApplySmoothing()`;
- SHOULD preserve normalization of the `gains` vector; and
- MAY preserve state across calls.

9.3.1.3 ApplyDecorrelation

`ApplyDecorrelation(config, diffuseOutput)`, which is not specified here, SHOULD minimize correlation across elements of `diffuseOutput`.

`ApplyDecorrelation()` MAY preserve state across calls.

9.3.2 Render Object Fragment

The `RenderObjectFragment` function renders an object `fragment`, returning speaker gains normalized to unit power.

```
float[] RenderObjectFragment(Configuration config, ObjectFragment
          fragment) {
 float gains[config.speakers.size()];
```

```
gains.fill(0);

// render each extended source individually

foreach(source in fragment.sources) {

 float sgains[config.speakers.size()];

 // is there an exact channel rendering exception match?

 RenderingException ex =
  fragment.renderingExceptions.first(r |
       r.targetConfiguration.equals(config.soundfieldName) and
       r instanceOf ChannelRenderingException and
       r.gains.forAll(g |
       config.speakers.collect(name).exists(g.channel)
  );

// if none found, look for a position exception

 if (ex == NULL) {
  ex =
   fragment.renderingExceptions.first(r |
       (r.targetConfiguration.equals(config.soundfieldName) or
        r.targetConfiguration.equals(NULL)) and
       r instanceOf PositionRenderingException
  );
 }

 // if none found, look for a wildcard channel rendering exception

 if (ex == NULL) {
  ex =
   fragment.renderingExceptions.first(r |
       r.targetConfiguration.equals(NULL) and
       r instanceOf ChannelRenderingException and
       r.gains.forAll(g |
       config.speakers.collect(name).exists(g.channel)
  );
 }

 if (ex instanceOf ChannelRenderingException) {

  // process channel rendering exception

  foreach (gain in ex.gains) {

   int i = config.speakers.index(s | s.name == gain.channel);

   sgains[i] = gain.coef;

  }

 } else {
```

```
    if (ex instanceOf PositionRenderingException) {

     // process position rendering exception

     source.position = selectedEx.position;
    }

    // render as point source if fewer then two virtual sources
    // belong to the extent of the source

    if (RenderExtendedSource(config, source, sgains) < 2) {
     sgains = RenderPointSource(config, source.position);
    }

   }

   // apply speaker mixmap

   for (int i = 0; i < gains.size(); i++) {

    for (int j = 0; j < gains.size(); j++) {

       gains[j] += sgains[i] * config.speakers[i].mixcoefs[j];

    }

   }

   // apply gain

   for(int i = 0; i < gains.size(); i++) {
    gains[i] *= pow(10, source.gain / 20);
   }

 }

 // normalize to unit power

 return gains / gains.norm();

}
```

9.3.3 RenderPatch

`RenderPatch` returns the contribution of a source located at `cartesianPosition` to each element of a triplet `speakerPatch` of speakers according to the Vector Base Amplitude Panning (VBAP [0]) formalism.

```
float[] RenderPatch(
     NormalSpeaker[3] speakerPatch,
     float[3] cartesianPosition
     ) {
```

```
 Matrix m = Matrix(
      speakerPatch[0].position.toCartesian(),
      speakerPatch[1].position.toCartesian(),
      speakerPatch[2].position.toCartesian()
 );

 return m.invert() * cartesianPosition;
}
```

The method `Position::toCartesian()` returns a `float[3]` array containing the (x, y, z) coordinates of the Position instance.

9.3.4 Point Source Rendering

A point source at position `pos` is rendered by applying `RenderPatch` to all patches in `config.patches` that intersect the source and averaging the resulting gains.

```
float[] RenderPointSource(
    Configuration config,
    Position pos) {

 float coefs[3];
 float gains[config.speakers.size()];

 int count = 0;

 foreach(patch in config.patches) {

  coefs = RenderPatch(
   {
    config.speakers[patch[0]],
    config.speakers[patch[1]],
    config.speakers[patch[2]]
   },
   pos.toCartesian()
  );

  // did the source fall within the patch?

  if (coefs[0] < 0 || coefs[1] < 0 || coefs[2] < 0) continue;

  int nsg = (coefs[0] > 0 ? 1 : 0)
                    + (coefs[1] > 0 ? 1 : 0)
                    + (coefs[2] > 0 ? 1 : 0);

  if (nsg == 2) {

   // two patches are expected if the source is on a patch edge

   gains[patch[0]] += coefs[0] / 2;
   gains[patch[1]] += coefs[1] / 2;
   gains[patch[2]] += coefs[2] / 2;

   count += 1/2;
```

```
   } else {

    gains[patch[0]] += coefs[0];
    gains[patch[1]] += coefs[1];
    gains[patch[2]] += coefs[2];

    count++;

    // no further contributions are expected if the source coincides with
         a patch vertex

    if (nsg == 1) break;

   };

 }

 if (count > 0) {

  foreach (gain in gains) {
   gain /= count;
  }

 }

 return gains;

}
```

9.4 RenderExtendedSource

The `RenderExtendedSource` function sums the gains of the virtual sources of `config.virtualSources` that fall within the extent defined by `source` and returns the number of virtual sources selected.

```
int RenderExtendedSource(
        Configuration config,
        ExtendedSource source,
        float gains[]) {

 VirtualSource vss[] =
  SelectSources(config.virtualSources, source);

 foreach(vs in vss) {
  gains += vs.gains;
 }

 return vss.size();
}
```

The function `SelectSources(virtualSources, fragment)` selects all elements of `virtualSources` that are within the extent of `fragment`, as defined in Section 6.12.5.

Bibliography (Informative)

Pulkki, Ville, Virtual Sound Source Positioning Using Vector Base Amplitude Panning, JAES Volume 45 Issue 6 pp. 456-466; June 1997