AcousticAVE library (libaave)

Generated by Doxygen 1.8.4

Fri Mar 14 2014 16:47:21

Contents

Chapter 1

Introduction

The AcousticAVE library (libaave) is an auralisation library. It is the equivalent of a 3D graphics visualisation library, but for audio: given a model of a room, the positions of the sound sources, the position and head orientation of the listener, and the anechoic audio stream of each sound source, libaave produces the 3D binaural soundfield that would be heard by that listener in that virtual environment.

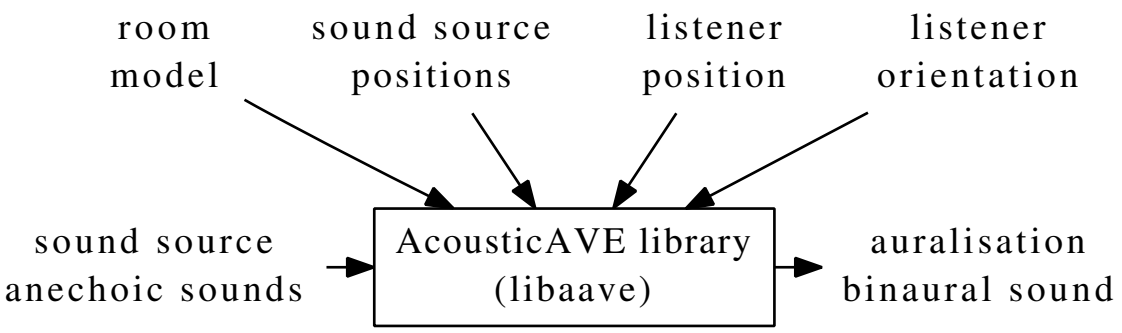


Figure 1.1: Role of the AcousticAVE library

libaave supports moving sound sources and listener, therefore it can be used for auralisation of interactive virtual reality environments, usually in combination with a 3D graphics visualisation library.

libaave performs in real-time for virtual rooms of some complexity (number of surfaces) and some order of sound reflections (configured by the user), more specifically the total number of sounds, that depend on the processor used. Benchmarks:

- Intel Atom N2600 1.6GHz: 66 sounds
- Intel Xeon 2GHz: 89 sounds
- AMD Opteron 248 2.2GHz: 193 sounds

The following diagrams illustrate typical usages of the AcousticAVE library for developing auralisation programs, one single-threaded and one multi-threaded.

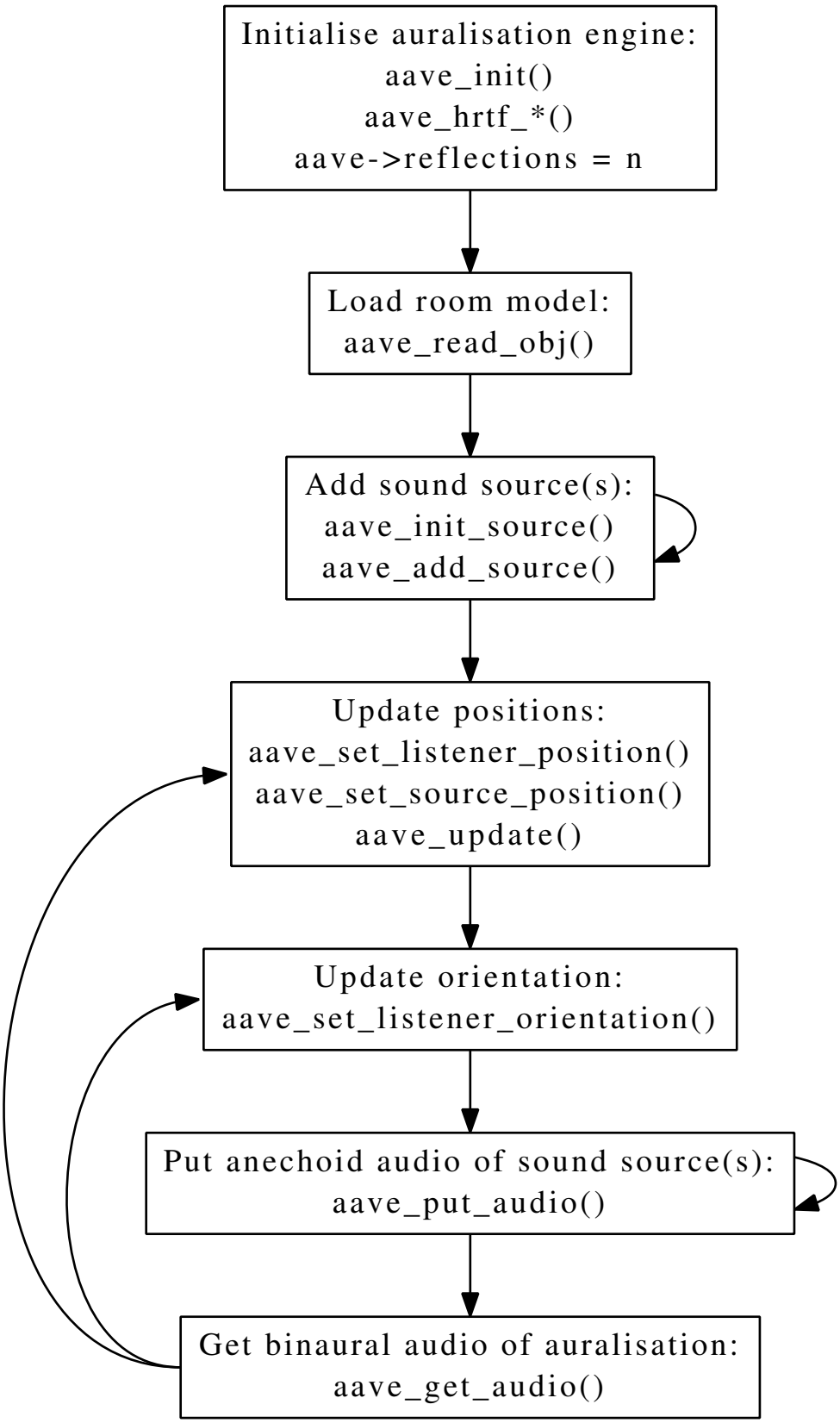


Figure 1.2: Single-thread usage example

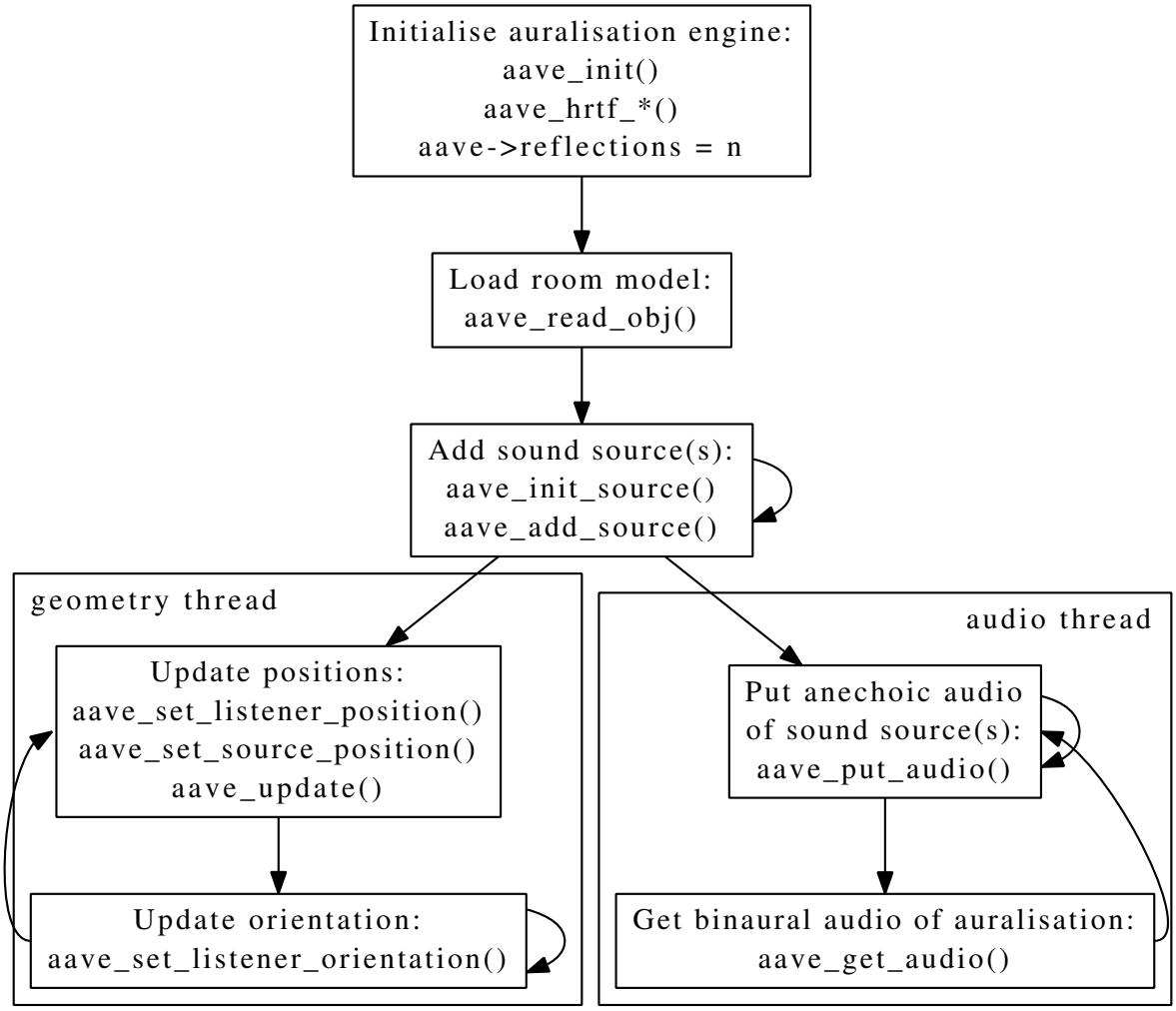


Figure 1.3: Multi-thread usage example

1.1 Installation

libaave is implemented in ANSI C and does not depend on any external library, so it is fairly portable (it is known to work on GNU/Linux and Microsoft Windows systems, but should also work on any other platform with an ANSI C compiler).

1. Download the source code:

```
git clone https://code.ua.pt/git/acousticave
```

2. Build libaave:

```
cd acousticave/libaave
make
```

For a quick start on using libaave in your programs, see the examples in acousticave/libaave/examples/.

1.2 Coordinates

For all 3D points (positions and room model vertices), libaave uses the coordinate system commonly used by CAD architecture programs: x=North, y=West, z=up, in metres.

For the listener head orientation angles, libaave uses the coordinate system commonly used by inertial sensors: x=North, y=East, z=down, in radians. This means:

- roll $>$ 0 is tilt right, roll $<$ 0 is tilt left
- pitch $>$ 0 is tilt up, pitch $<$ 0 is tilt down
- yaw $>$ 0 is rotate right, yaw $<$ 0 is rotate left, yaw = 0 is North

1.3 Overview

The file aave.h is the public header file of libaave. Include it from your program source files to use its structures and functions.

The file geometry.c implements all functions related to geometry processing: construction of the 3D room model, coordinate conversions, determining the image-source positions, the sound reflection paths, the audible and non-audible sound paths, and passing all this information to the audio processing functions.

The file obj.c contains a convenience function that reads a 3D model from a Wavefront .OBJ file and calls the appropriate functions in geometry.c to construct the room to be auralised in just one step.

The file audio.c implements the core of the audio processing: receiving anechoic audio data from the sound sources, head-related transfer function (HRTF) processing, and generating the corresponding auralised audio data in binaural format.

The files hrtf_cipic.c, hrtf_listen.c, hrtf_mit.c, and hrtf_tub.c implement the interface functions for using the CIPIC, LISTEN, MIT, and TU-Berlin HRTF sets, respectively.

The files dft.h and idft.h implement the Discrete Fourier Transform and Inverse Discrete Fourier Transform algorithms, respectively, used mainly for the HRTF processing.

The file material.c implements the functions related to the sound absorption caused by the different surface materials: the table of material reflection coefficients by frequency band, the table lookup, and the design of the audio filters.

The file reverb.c implements a simple artificial reverberation algorithm that adds a tail of late reflections to the auralisation output. The file reverb_dattorro.c implements the Dattorro reverberator.

The directory tools contains the programs used to automatically generate the hrtf_$*$_set$*$.c source files from the respective HRTF data sets, the dftsincos.c source file with the sin() and cos() lookup table for the DFT and IDFT algorithms, and miscellaneous utility programs to handle or generate audio files.

The directory tests/ contains programs to verify the correctness of the functions implemented in libaave.

The directory examples/ contains programs to show how libaave can be used for different auralisation applications.

The directory doc/ contains the files to generate this document from the documentation written in the source files.

1.4 Acknowledgements

The development of libaave was funded by the Portuguese Government through FCT (Fundação para a Ciência e a Tecnologia) as part of the project AcousticAVE: Auralisation Models and Applications for Virtual Reality Environments (PTDC/EEA-ELC/112137/2009).

Chapter 2

Todo List

Global aave_create_sounds_recursively (struct aave *aave, struct aave_source *source, unsigned order, unsigned o, struct aave_surface *surfaces[], float image_sources[][3])

Implement the iterative version of this recursive algorithm.

Global AAVE_FS

To support different audio sampling frequencies without having to recompile the library, this value would be set in a member of the aave structure at runtime instead. Of course, that would incur in performance penalties, most importantly in the audio processing (one floating-point division per audio sample per auralised sound). Furthermore, the HRTF data sets would have to be resampled to the desired sampling frequency.

Global aave_hrtf_cipic_get (const float *hrtf[2], int elevation, int azimuth)

Use all elevation measures available, not just 0 degrees.

Global aave_hrtf_listen_get (const float *hrtf[2], int elevation, int azimuth)

Elevations 60, 75 and 90.

Global AAVE_MAX_HRTF

When using HRTFs with less frames (MIT only has 128) there is a considerable waste of memory throughout the library. However, this way the code is much simpler, and slightly faster. Nevertheless, it would be nice if this value could be changed at runtime when the user selects the HRTF set to use.

Global AAVE_MAX_REFLECTIONS

To support different maximum orders of reflections per instance, this value would be set in a member of the aave structure at runtime and the sounds hash table allocated accordingly. However, I think this is not worth the trouble. Just change this value and recompile, if you want more orders of reflections (and your computer can handle them). The waste is only 4 or 8 bytes per reflection order that is not used, for 32-bit or 64-bit processors respectively.

Global aave_surface::points [32][3+2]

Remove hardcoded maximum number of points per surface.

Global allpass (struct allpass *ap, float x, float g, unsigned delay)

Check if the tap is really x1 or x2.

Global allpass::buffer [2656]

Set maximum delay from the delays of all all-pass blocks.

File audio.c

Here, it might be more efficient to use the overlap-save method instead of the overlap-add method.

Class dc_block_filter

Improve bandwidth definition.

Global delay::buffer [16384]

Set maximum delay from the delays of all delay blocks.

Global dft_index_table []

If the order of the material absorption filter designed in material.c increases to N $>$ 128, increase this table accordingly.

Global idft (IDFT_TYPE *x, float *X, unsigned n)

round instead of truncate

Global MAX_VERTICES

Use dynamic memory allocation for the array of vertices to support "unlimited" number of vertices.

File reverb_dattorro.c

Make the code reentrant (move the static structures to aave).

File reverb_jot.c

A `dc_block_filter` was introduced to aproximate low frequency damping. Improve this filter (or introduce another) for flexible bandwidth selection.

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

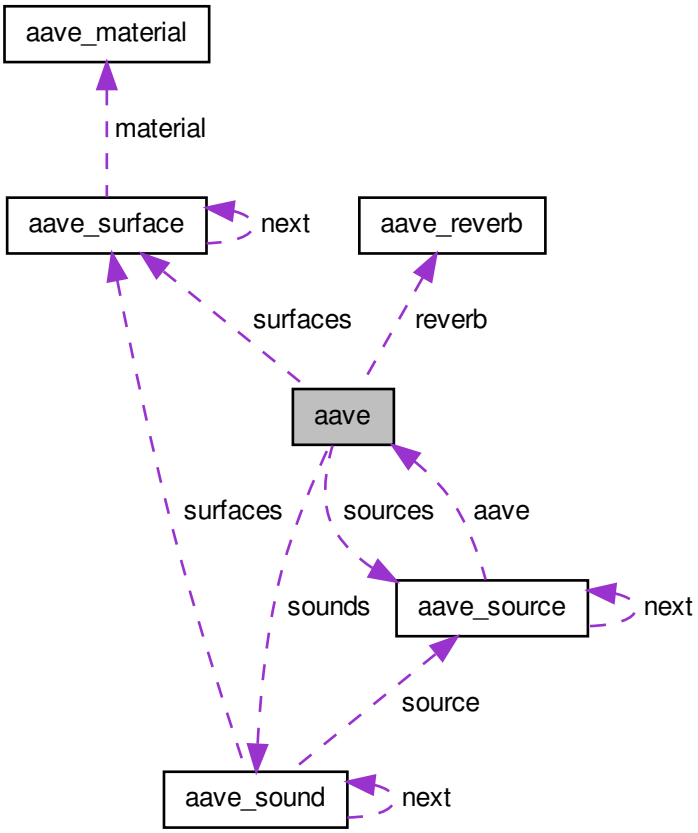
Chapter 5

Data Structure Documentation

5.1 aave Struct Reference

`#include <aave.h>`

Collaboration diagram for aave:



Data Fields

- float position [3]
- float orientation [3][3]
- struct aave_surface * surfaces
- unsigned nsurfaces
- float room_material_absorption
- unsigned volume
- unsigned area
- struct aave_source * sources
- struct aave_sound * sounds [AAVE_MAX_REFLECTIONS]
- unsigned reflections
- struct aave_reverb * reverb
- float gain
- unsigned hrtf_frames
- void(* hrtf_get)(const float *hrtf[2], int elevation, int azimuth)
- unsigned hrtf_output_buffer_index
- short hrtf_output_buffer [AAVE_MAX_HRTF *4]
- int hrtf_overlap_add_buffer [2][AAVE_MAX_HRTF *2]

5.1.1 Detailed Description

The AcousticAVE main data structure. It contains all the information that defines one acoustic world and its present auralisation state.

Examples:

examples/circle.c, examples/elevation.c, examples/line.c, and examples/stream.c.

5.1.2 Field Documentation

5.1.2.1 unsigned aave::area

Sum of all room surface areas.

5.1.2.2 float aave::gain

Gain to apply to the output sound.

5.1.2.3 unsigned aave::hrtf_frames

The number of frames of the HRTFs currently in use (power of 2).

5.1.2.4 void(* aave::hrtf_get)(const float *hrtf[2], int elevation, int azimuth)

Function to get the HRTF pair for some elevation and azimuth.

5.1.2.5 short aave::hrtf_output_buffer[AAVE_MAX_HRTF *4]

HRTF audio block output buffer (2 16-bit channels interleaved).

5.1.2.6 unsigned aave::hrtf_output_buffer_index

Index of the next frame of the HRTF output buffer to be consumed.

5.1.2.7 int aave::hrtf_overlap_add_buffer[2][AAVE_MAX_HRTF ∗2]

HRTF overlap-add buffer (2 32-bit channels).

5.1.2.8 unsigned aave::nsurfaces

Number of surfaces in the list of surfaces.

5.1.2.9 float aave::orientation[3][3]

Listener head orientation (inertial-to-body rotation matrix).

5.1.2.10 float aave::position[3]

Position of the listener in the auralization world [x,y,z] (m).

5.1.2.11 unsigned aave::reflections

Maximum number of reflections to calculate for each source.

Examples:

examples/stream.c.

5.1.2.12 struct aave_reverb∗ aave::reverb

Late reverberation parameters.

5.1.2.13 float aave::room_material_absorption

Average of all room surface absorption coeficients.

5.1.2.14 struct aave_sound∗ aave::sounds[AAVE_MAX_REFLECTIONS]

Hash table of sounds to auralise, indexed by reflection order.

5.1.2.15 struct aave_source∗ aave::sources

Singly-linked list of sound sources in the auralisation world.

5.1.2.16 struct aave_surface∗ aave::surfaces

Singly-linked list of surfaces in the auralisation world.

5.1.2.17 unsigned aave::volume

Room volume (m3).

The documentation for this struct was generated from the following file:

- aave.h

5.2 aave_material Struct Reference

```
#include <aave.h>
```

Data Fields

- const char * name
- const unsigned char reflection_factors [AAVE_MATERIAL_REFLECTION_FACTORS]

5.2.1 Detailed Description

Acoustic properties of a material.

5.2.2 Field Documentation

5.2.2.1 const char* aave_material::name

Name used in the usemtl directives of the .obj files.

5.2.2.2 const unsigned char aave_material::reflection_factors[AAVE_MATERIAL_REFLECTION_FACTORS]

Reflection factors (multiplied by 100), by frequency band.

The documentation for this struct was generated from the following file:

- aave.h

5.3 aave_reverb Struct Reference

Data Fields

- unsigned RT60
- float Tmixing
- float rc
- float pre_delay
- float alpha
- short decorrelation_coefs [FDN_ORDER][2]
- float beta
- float fdn_output_taps [FDN_ORDER]
- float absorption_gain [FDN_ORDER]
- float absorption_bandwidth [FDN_ORDER]
- float mix
- float level
- short active

5.3.1 Field Documentation

5.3.1.1 float aave_reverb::absorption_bandwidth[FDN_ORDER]

Bandwidth for the absorption (lowpass) filters.

5.3.1.2 float aave_reverb::absorption_gain[FDN_ORDER]

Amplitude attenuation for the absorption (lowpass) filters.

5.3.1.3 short aave_reverb::active

Flag to activate/deactivate late reverberation.

5.3.1.4 float aave_reverb::alpha

alpha = Tr(pi) / Tr(0). Ratio of the RT at the Nyquist frequency and the DC frequency.

5.3.1.5 float aave_reverb::beta

beta = 1 - sqrt(alpha) / 1 + sqrt(alpha). Bandwidth coeficient for the tone correction (highpass) filter.

5.3.1.6 short aave_reverb::decorrelation_coefs[FDN_ORDER][2]

Decorrelation coeficients for producing a decorrelated stereo output.

5.3.1.7 float aave_reverb::fdn_output_taps[FDN_ORDER]

Circulating matrix outputs storage after every multiplication.

5.3.1.8 float aave_reverb::level

Gain/attenuation for managing the level of the late reverberation.

5.3.1.9 float aave_reverb::mix

Constant attenuation to the stereo output of the reverberator.

5.3.1.10 float aave_reverb::pre_delay

Reverb predelay (samples). Sum of perceptual delay (Tmixing) and latency due to HRTF buffering.

5.3.1.11 float aave_reverb::rc

Crital distance at which the direct sound preassure is equal to the reverberation sound pressure.

5.3.1.12 unsigned aave_reverb::RT60

Reverberation time (miliseconds).

5.3.1.13 float aave_reverb::Tmixing

Tmixing = sqrt(Volume) (miliseconds). Predelay of the late reverberation.

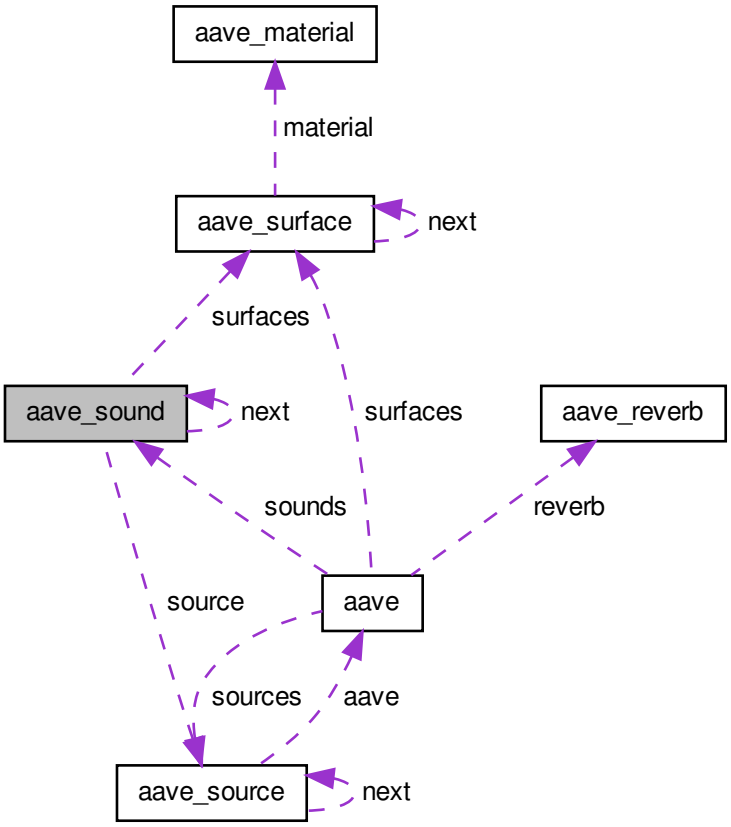
The documentation for this struct was generated from the following file:

- aave.h

5.4 aave_sound Struct Reference

`#include <aave.h>`

Collaboration diagram for aave_sound:



Data Fields

- struct aave_sound * next
- struct aave_source * source
- float * position
- int audible
- unsigned fade_samples
- float distance

- float distance_smooth
- const float * hrtf [2]
- struct aave_surface * surfaces [AAVE_MAX_REFLECTIONS]
- float image_sources [AAVE_MAX_REFLECTIONS][3]
- float reflection_points [AAVE_MAX_REFLECTIONS][3]
- float dft [AAVE_MAX_HRTF *4]
- float filter [AAVE_MAX_HRTF *4]

5.4.1 Detailed Description

Data for each sound to auralise.

5.4.2 Field Documentation

5.4.2.1 int aave_sound::audible

Flag that indicates if the sound is audible (1) or not (0).

5.4.2.2 float aave_sound::dft[AAVE_MAX_HRTF *4]

The DFT of the previous audio block.

5.4.2.3 float aave_sound::distance

The previous distance value used (for the crossfading).

5.4.2.4 float aave_sound::distance_smooth

Smooth (low-pass filtered) distance value (for the resampling).

5.4.2.5 unsigned aave_sound::fade_samples

The previous fade-in/out sample count value used (for the fade-in/out of appearing/disappearing sounds).

5.4.2.6 float aave_sound::filter[AAVE_MAX_HRTF *4]

The material absorption filter DFT.

5.4.2.7 const float* aave_sound::hrtf[2]

The previous HRTF pair used (for the crossfading).

5.4.2.8 float aave_sound::image_sources[AAVE_MAX_REFLECTIONS][3]

The image-source positions calculated for each reflection.

5.4.2.9 struct aave_sound* aave_sound::next

Pointer to the next sound (singly-linked list).

5.4.2.10 float* aave_sound::position

Position of the source or image-source of this sound [x,y,z] (m). If this is a direct sound, this points to source->position. If it is a reflection, this points to image_sources[order].

5.4.2.11 float aave_sound::reflection_points[AAVE_MAX_REFLECTIONS][3]

The points [x,y,z] where this sound reflects.

5.4.2.12 struct aave_source* aave_sound::source

The sound source that is producing this sound.

5.4.2.13 struct aave_surface* aave_sound::surfaces[AAVE_MAX_REFLECTIONS]

The surfaces where this sound reflects.

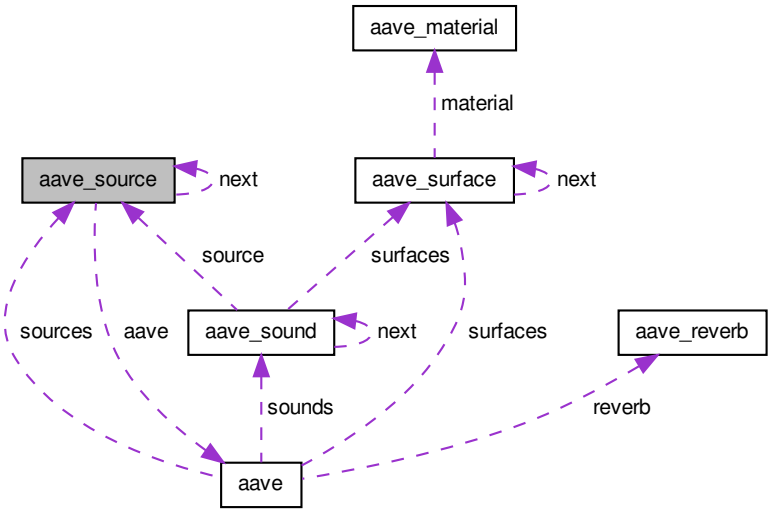
The documentation for this struct was generated from the following file:

- aave.h

5.5 aave_source Struct Reference

`#include <aave.h>`

Collaboration diagram for aave_source:



Data Fields

- struct aave_source * next
- struct aave * aave

- float position [3]
- unsigned buffer_index
- short buffer [AAVE_SOURCE_BUFSIZE]

5.5.1 Detailed Description

Data for each sound source in the auralisation world.

Examples:

examples/circle.c, examples/elevation.c, examples/line.c, and examples/stream.c.

5.5.2 Field Documentation

5.5.2.1 struct aave* aave_source::aave

The AcousticAVE engine this sound source is associated with.

5.5.2.2 short aave_source::buffer[AAVE_SOURCE_BUFSIZE]

Ring buffer to store the recent past anechoic samples.

Examples:

examples/stream.c.

5.5.2.3 unsigned aave_source::buffer_index

Index of the most recently inserted sample.

5.5.2.4 struct aave_source* aave_source::next

Pointer to the next source (singly-linked list).

5.5.2.5 float aave_source::position[3]

Position of the source in the auralisation world [x,y,z] (m).

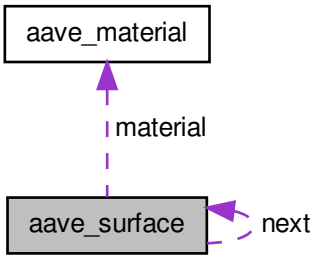
The documentation for this struct was generated from the following file:

- aave.h

5.6 aave_surface Struct Reference

```
#include <aave.h>
```

Collaboration diagram for aave_surface:



Data Fields

- struct aave_surface * next
- struct aave_material * material
- float avg_absorption_coef
- float normal [3]
- float **distance**
- float versors [2][3]
- unsigned npoints
- float points [32][3+2]

5.6.1 Detailed Description

Data for each surface that makes the auralisation world. A surface is defined as an n-point planar polygon. The points are specified in anti-clockwise order.

5.6.2 Field Documentation

5.6.2.1 float aave_surface::avg_absorption_coef

Average absorption coeficient.

5.6.2.2 struct aave_material* aave_surface::material

Material of the surface.

5.6.2.3 struct aave_surface* aave_surface::next

Pointer to the next surface (singly-linked list).

5.6.2.4 float aave_surface::normal[3]

Specification of the plane where this surface lays, in Hessian normal form: unit normal vector and distance from origin. (http://mathworld.wolfram.com/HessianNormalForm.html) The distance is used to calculate the position of the image sources. The unit normal vector is used just about everywhere.

5.6.2.5 unsigned aave_surface::npoints

Number of points of the polygon (minimum 3).

5.6.2.6 float aave_surface::points[32][3+2]

Coordinates of each point, in counter-clockwise order (counter-clockwise normal).

World coordinates:

- x = points[i][0]
- y = points[i][1]
- z = points[i][2]

Local coordinates (internally used by the polygon-line intersection algorithm):

- ex = points[i][3]
- ex = points[i][4]

Todo Remove hardcoded maximum number of points per surface.

5.6.2.7 float aave_surface::versors[2][3]

Alternate specification of the plane where this surface lays using 2 versors and a point (points[0]). This is used to perform calculations in local coordinates, namely the non-convex polygon - line intersection algorithm.

The documentation for this struct was generated from the following file:

- aave.h

5.7 absorption_filter Struct Reference

Data Fields

- float y

5.7.1 Detailed Description

Data of a low-pass filter for frequency-dependent reverberation time.

5.7.2 Field Documentation

5.7.2.1 float absorption_filter::y

Previous output of the filter (y[n-1]).

The documentation for this struct was generated from the following file:

- reverb_jot.c

5.8 allpass Struct Reference

Data Fields

- float tap
- unsigned index
- float buffer [2656]

5.8.1 Detailed Description

Data of an all-pass filter.

5.8.2 Field Documentation

5.8.2.1 float allpass::buffer[2656]

Buffer to store the inserted values.

Todo Set maximum delay from the delays of all all-pass blocks.

5.8.2.2 unsigned allpass::index

Index to the latest value inserted in the buffer.

5.8.2.3 float allpass::tap

Value captured inside the feedback network.

The documentation for this struct was generated from the following file:

- reverb_dattorro.c

5.9 dc_block_filter Struct Reference

Data Fields

- float y
- float x
- float b

5.9.1 Detailed Description

Data of a dc block filter for a rough reverb highpass.

Todo Improve bandwidth definition.

5.9.2 Field Documentation

5.9.2.1 float dc_block_filter::b

bandwidth.

5.9.2.2 float dc_block_filter::x

Previous input of the filter (x[n-1]).

5.9.2.3 float dc_block_filter::y

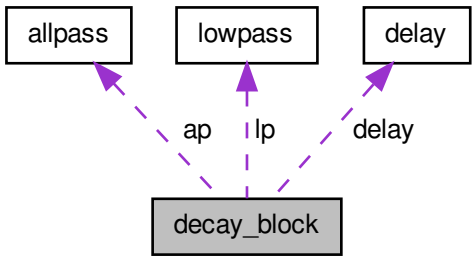
Previous output of the filter (y[n-1]).

The documentation for this struct was generated from the following file:

- reverb_jot.c

5.10 decay_block Struct Reference

Collaboration diagram for decay_block:



Data Fields

- float out
- struct lowpass lp
- struct allpass ap [2]
- struct delay delay [2]

5.10.1 Detailed Description

Data of a decay block.

5.10.2 Field Documentation

5.10.2.1 struct allpass decay_block::ap[2]

The decay all-pass filters.

5.10.2.2 struct delay decay_block::delay[2]

The delay lines.

5.10.2.3 struct lowpass decay_block::lp

The dampping low-pass filter.

5.10.2.4 float decay_block::out

The previous output value.

The documentation for this struct was generated from the following file:

- reverb_dattorro.c

5.11 delay Struct Reference

Data Fields

- unsigned index
- float buffer [16384]

5.11.1 Detailed Description

Data of a delay block.

5.11.2 Field Documentation

5.11.2.1 float delay::buffer[16384]

Buffer to store the inserted values.

Todo Set maximum delay from the delays of all delay blocks.

5.11.2.2 unsigned delay::index

Index to the latest value inserted in the buffer.

The documentation for this struct was generated from the following file:

- reverb_dattorro.c

5.12 delay_filter Struct Reference

Data Fields

- unsigned index
- float buffer [MAX_DELAY_TIME]

5.12.1 Detailed Description

Data of a delay filter.

5.12.2 Field Documentation

5.12.2.1 float delay_filter::buffer[MAX_DELAY_TIME]

Buffer to store the inserted values.

5.12.2.2 unsigned delay_filter::index

Index to the latest value inserted in the buffer.

The documentation for this struct was generated from the following file:

- reverb_jot.c

5.13 lowpass Struct Reference

Data Fields

- float y

5.13.1 Detailed Description

Data of a low-pass filter.

5.13.2 Field Documentation

5.13.2.1 float lowpass::y

Previous output of the filter (y[n-1]).

The documentation for this struct was generated from the following file:

- reverb_dattorro.c

5.14 tone_correction_filter Struct Reference

Data Fields

- float y

5.14.1 Detailed Description

Data of a tone correction (high pass) filter.

5.14.2 Field Documentation

5.14.2.1 float tone_correction_filter::y

Previous output of the filter (y[n-1]).

The documentation for this struct was generated from the following file:

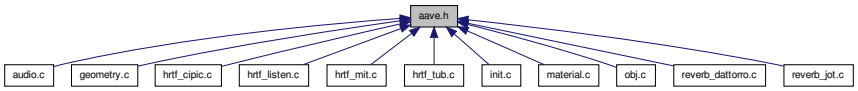
- reverb_jot.c

Chapter 6

File Documentation

6.1 aave.h File Reference

This graph shows which files directly or indirectly include this file:



Data Structures

- struct aave
- struct aave_source
- struct aave_surface
- struct aave_sound
- struct aave_material
- struct aave_reverb

Macros

- #define AAVE_FS 44100
- #define AAVE_MAX_REFLECTIONS 16
- #define AAVE_MAX_HRTF 2048
- #define AAVE_SOURCE_BUFSIZE 131072
- #define AAVE_MATERIAL_REFLECTION_FACTORS 7
- #define AAVE_SOUND_SPEED 343.2
- #define FDN_ORDER 64

Functions

- void aave_get_audio (struct aave ∗, short ∗, unsigned)
- void aave_put_audio (struct aave_source ∗, const short ∗, unsigned)
- unsigned dft_index (unsigned, unsigned)
- void aave_add_source (struct aave ∗, struct aave_source ∗)
- void aave_add_surface (struct aave ∗, struct aave_surface ∗)

- void aave_get_coordinates (const struct aave *, const float *, float *, float *, float *)
- void aave_set_listener_orientation (struct aave *, float, float, float)
- void aave_set_listener_position (struct aave *, float, float, float)
- void aave_set_source_position (struct aave_source *, float, float, float)
- void aave_update (struct aave *)
- void aave_hrtf_cipic (struct aave *)
- void aave_hrtf_listen (struct aave *)
- void aave_hrtf_mit (struct aave *)
- void aave_hrtf_tub (struct aave *)
- void aave_init (struct aave *, unsigned)
- void aave_init_source (struct aave *, struct aave_source *)
- struct aave_material * aave_get_material (const char *)
- void aave_get_material_filter (struct aave *, struct aave_surface **, unsigned, float *)
- void aave_read_obj (struct aave *, const char *, unsigned, unsigned)
- void aave_reverb_dattorro (struct aave *, short *, unsigned)
- void aave_reverb_jot (struct aave *, short *, unsigned)
- void init_reverb (struct aave_reverb *, float, float, float)
- void **print_reverb_parameters** (struct aave *, struct aave_reverb *)

Variables

- struct aave_material aave_material_none

6.1.1 Detailed Description

The aave.h file is the public header file of the AcousticAAVE library (libaave). It contains the declarations of the structures and functions used and implemented by the library. Include it from your program source files to use them.

The following diagram provides an overview of the data structures in the library and their associations.

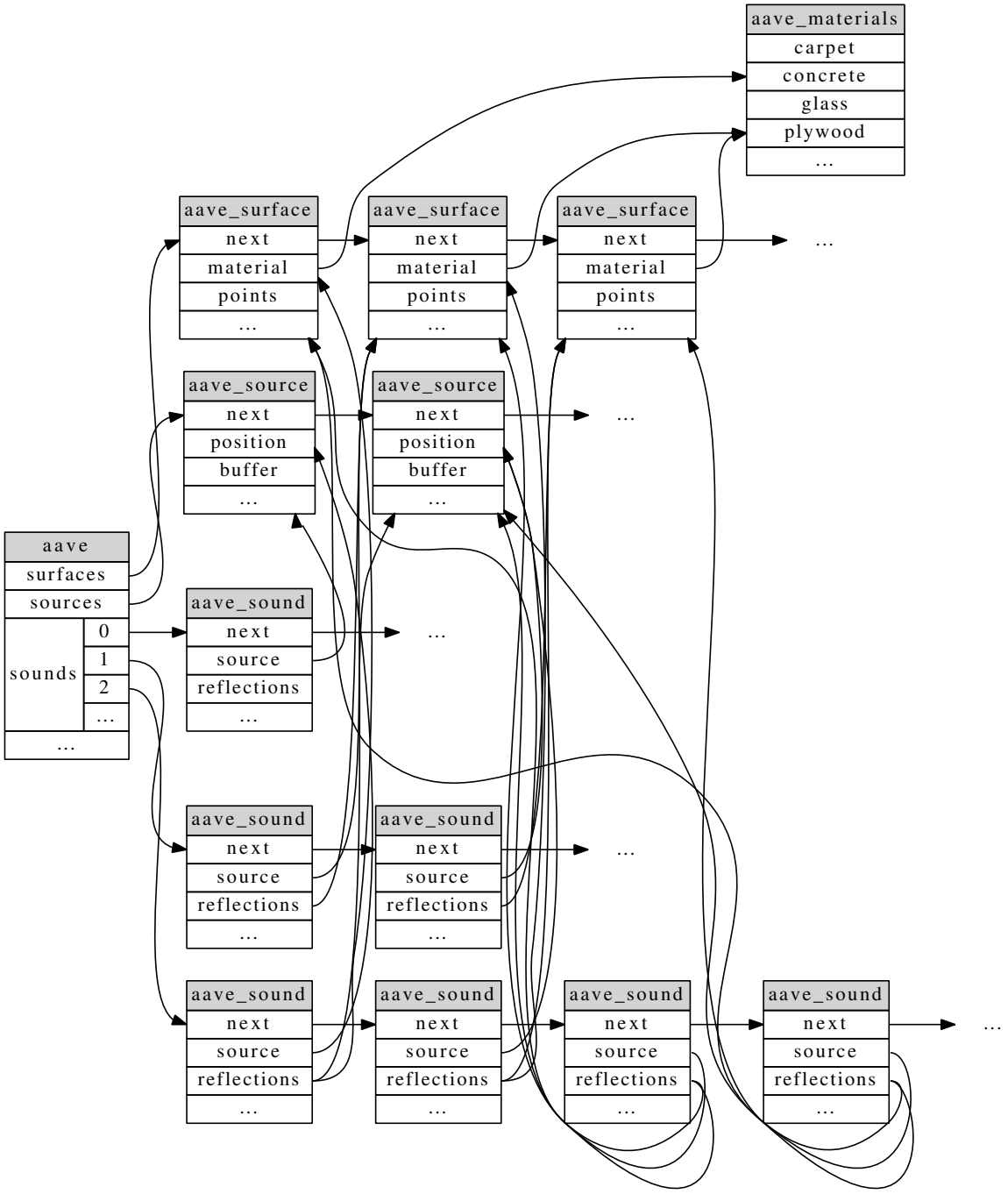


Figure 6.1: Data structure diagram

The aave structure is the main data structure of the AcousticAVE library. It contains all necessary information to completely define an auralisation world and its present auralisation state.

The aave structure contains a list of aave_surface structures. These surfaces define the geometry of the auralisation world. The material of each surface is indicated by pointing to the corresponding element in the aave_materials table.

The aave structure contains a list of aave_source structures, one for each sound source present in the auralisation world.

The sound sources and surfaces in the auralisation world generate a number of sounds that reach the listener, and that are to be auralised by the library. These sounds are stored in the aave structure in a hash table indexed by the reflection order: direct sounds are put in the list of aave_sound structures pointed by sounds[0], 1st reflection sounds

are put in sounds[1], 2nd reflections in sounds[2], etc... The aave_update() function is responsible for maintaining the hash table up to date, and the aave_get_audio() function is responsible for auralising all sounds currently in it.

Each aave_sound structure contains a reference to its sound source and references to each surface where that sound reflects.

6.1.2 Macro Definition Documentation

6.1.2.1 #define AAVE_FS 44100

The audio sampling frequency, in Hz, used throughout the library. In particular, libaave expects the anechoic input data of the sound sources to be delivered in this sampling frequency. The auralisation binaural output data is also in this sampling frequency.

Todo To support different audio sampling frequencies without having to recompile the library, this value would be set in a member of the aave structure at runtime instead. Of course, that would incur in performance penalties, most importantly in the audio processing (one floating-point division per audio sample per auralised sound). Furthermore, the HRTF data sets would have to be resampled to the desired sampling frequency.

Examples:

examples/circle.c, examples/elevation.c, and examples/line.c.

6.1.2.2 #define AAVE_MATERIAL_REFLECTION_FACTORS 7

The number of reflection factors that specify each material. The corresponding frequencies are: 125, 250, 500, 1000, 2000, 4000, 8000 Hz.

6.1.2.3 #define AAVE_MAX_HRTF 2048

The maximum number of frames of an HRTF. (The longest HRTFs are TU-Berlin's: 2048).

Todo When using HRTFs with less frames (MIT only has 128) there is a considerable waste of memory throughout the library. However, this way the code is much simpler, and slightly faster. Nevertheless, it would be nice if this value could be changed at runtime when the user selects the HRTF set to use.

6.1.2.4 #define AAVE_MAX_REFLECTIONS 16

The maximum order of reflections supported. This defines the maximum value the user can select for the order of reflections to calculate in the auralisation process.

Todo To support different maximum orders of reflections per instance, this value would be set in a member of the aave structure at runtime and the sounds hash table allocated accordingly. However, I think this is not worth the trouble. Just change this value and recompile, if you want more orders of reflections (and your computer can handle them). The waste is only 4 or 8 bytes per reflection order that is not used, for 32-bit or 64-bit processors respectively.

6.1.2.5 #define AAVE_SOUND_SPEED 343.2

Speed of sound in dry air at 20 degrees Celsius (m/s).

Reference: https://en.wikipedia.org/wiki/Speed_of_sound

6.1.2.6 #define AAVE_SOURCE_BUFSIZE 131072

The number of past anechoic samples to hold for each sound source. This effectively defines the maximum distance that can be auralised:

distance [m] = AAVE_SOURCE_BUFSIZE ∗ AAVE_SOUND_SPEED [m/s] / AAVE_FS [Hz]

Must be a power of 2 to allow for the most efficient implementation. Some possible values (and corresponding maximum distances for fs=44100Hz):

32768 (255m), 65536 (510m), 131072 (1020m), 262144 (2040m), 524288 (4080m)

6.1.2.7 #define FDN_ORDER 64

The order of the circulation matrix for the FDN late reverberator.

6.1.3 Function Documentation

6.1.3.1 void aave_add_source (struct aave ∗ *aave,* struct aave_source ∗ *source*)

Add a sound source to the auralisation world.

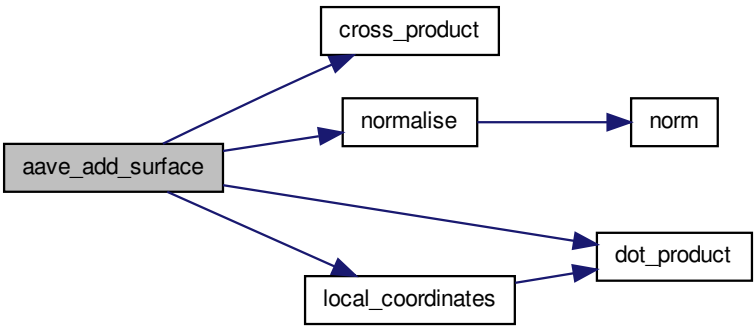
Examples:

examples/circle.c, examples/elevation.c, examples/line.c, and examples/stream.c.

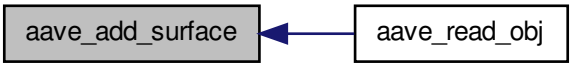
6.1.3.2 void aave_add_surface (struct aave ∗ *aave,* struct aave_surface ∗ *surface*)

Add a surface to the auralisation world.

Here is the call graph for this function:



Here is the caller graph for this function:



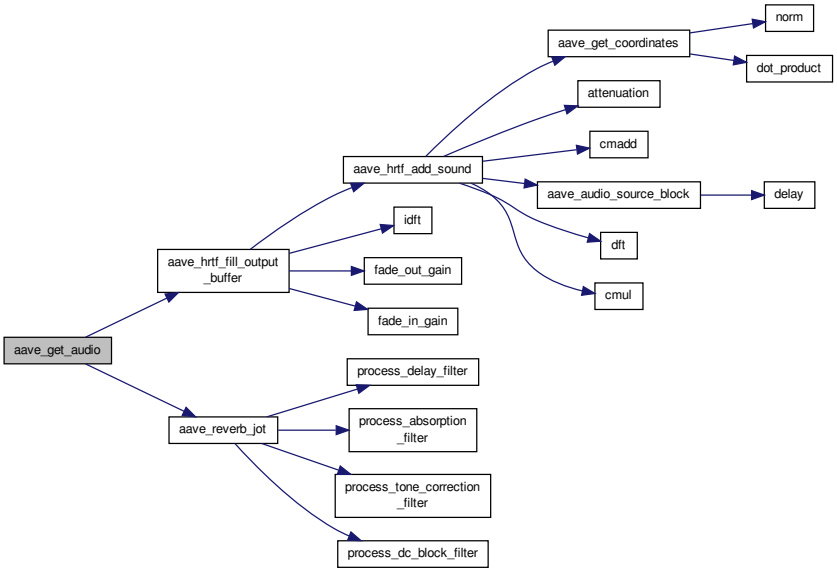
6.1.3.3 void aave_get_audio (struct aave * *aave*, short * *buf*, unsigned *n*)

Generate `n` 16-bit 2-channel frames of the auralisation world `aave` and put them in the memory location pointed by `buf`.

Examples:

examples/circle.c, examples/elevation.c, examples/line.c, and examples/stream.c.

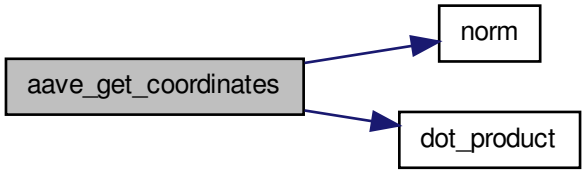
Here is the call graph for this function:



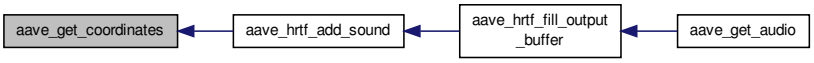
6.1.3.4 void aave_get_coordinates (const struct aave * *aave*, const float * *source_position*, float * *distance*, float * *elevation*, float * *azimuth*)

Get the `distance` (m), `azimuth` (rad) and `elevation` (rad) coordinates of the position `source_position` of a source relative to the listener.

Here is the call graph for this function:



Here is the caller graph for this function:

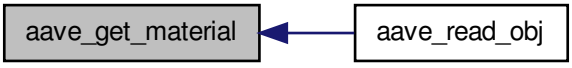


6.1.3.5 struct aave_material* aave_get_material (const char * *name*)

Return the material with the specified `name`. If no material is found with such name, return aave_material_none.

The search is performed using the binary search algorithm (http://en.wikipedia.org/wiki/Binary_search_algorithm), that's why the table of materials must be ordered by name.

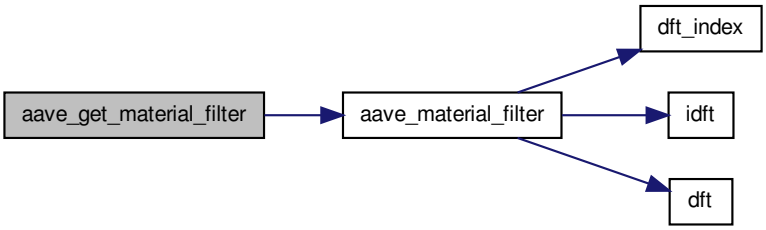
Here is the caller graph for this function:



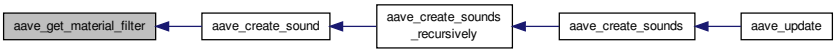
6.1.3.6 void aave_get_material_filter (struct aave * *aave,* struct aave_surface ** *surfaces,* unsigned *reflections,* float * *filter*)

Design the material absorption filter for the specified sequence of `surfaces` and reflection order `reflections`. The calculated DFT coefficients of the filter are stored in `filter`, which must have 4 times the elements of the HRIRs of the HRTF set currently in use.

Here is the call graph for this function:



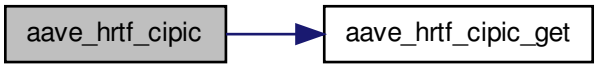
Here is the caller graph for this function:



6.1.3.7 void aave_hrtf_cipic (struct aave * a)

Select the CIPIC HRTF set for the auralisation process.

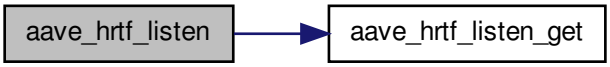
Here is the call graph for this function:



6.1.3.8 void aave_hrtf_listen (struct aave * a)

Select the LISTEN HRTF set for the auralisation process.

Here is the call graph for this function:



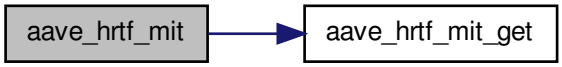
6.1.3.9 void aave_hrtf_mit (struct aave * a)

Select the MIT KEMAR HRTF compact set for the auralisation process.

Examples:

examples/circle.c, examples/elevation.c, examples/line.c, and examples/stream.c.

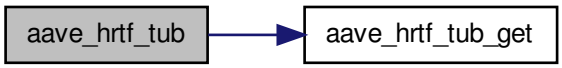
Here is the call graph for this function:



6.1.3.10 void aave_hrtf_tub (struct aave * a)

Select the TU-Berlin HRTF set for the auralisation process.

Here is the call graph for this function:



6.1.3.11 void aave_init (struct aave * aave, unsigned rt60)

Initialise the auralisation data structure.

The listener's initial position is (0, 0, 0).

The listener's head initial orientation is invalid! You must call aave_set_listener_orientation()!

The initial output gain is 1 (0dB).

The artificial reverberation tail is initially enabled.

Examples:

examples/circle.c, examples/elevation.c, examples/line.c, and examples/stream.c.

Here is the call graph for this function:

6.1.3.12 void aave_init_source (struct aave * *aave*, struct aave_source * *source*)

Initialise a sound source data structure to be used by the aave engine.

Examples:

examples/circle.c, examples/elevation.c, examples/line.c, and examples/stream.c.

6.1.3.13 void aave_put_audio (struct aave_source * *source*, const short * *audio*, unsigned *n*)

Put the `n` frames pointed by `audio` in the ring buffer of `source`.

Examples:

examples/circle.c, examples/elevation.c, examples/line.c, and examples/stream.c.

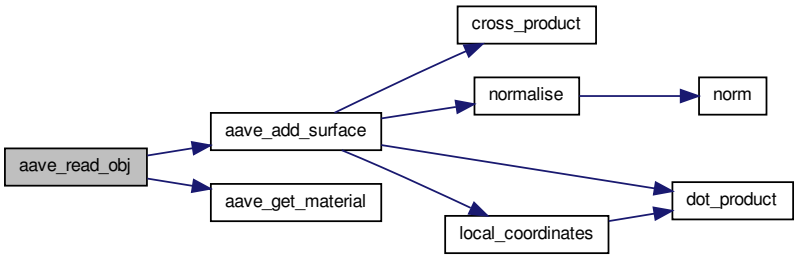
6.1.3.14 void aave_read_obj (struct aave * *aave*, const char * *filename*, unsigned *volume*, unsigned *area*)

Read the .obj file `filename` and add its contents to the auralisation engine `aave`.

Examples:

examples/line.c, and examples/stream.c.

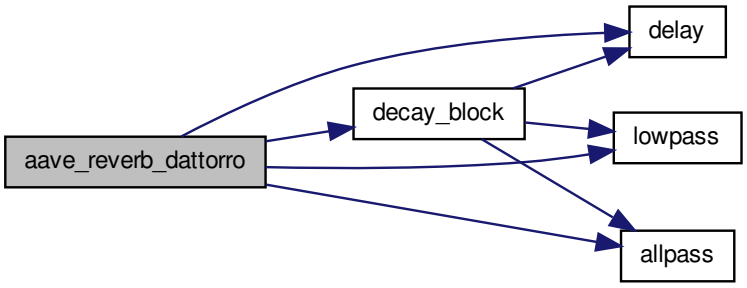
Here is the call graph for this function:



6.1.3.15 void aave_reverb_dattorro (struct aave * *aave*, short * *audio*, unsigned *n*)

Run a Dattorro reverberator to add an artificial reverberation tail to the `n` binaural frames (2 * `n` samples) pointed by `audio`.

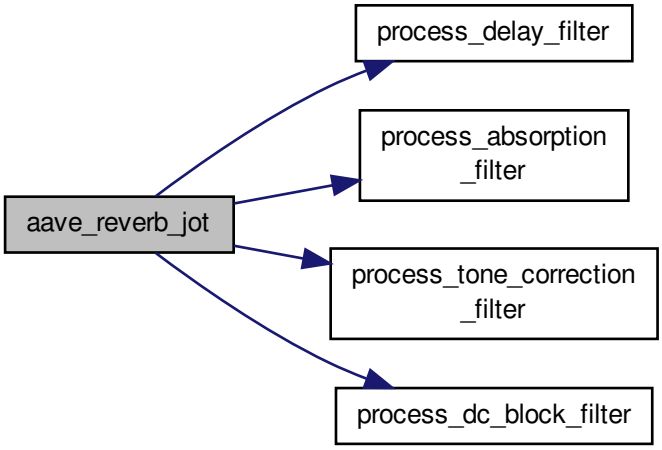
Here is the call graph for this function:



6.1.3.16 void aave_reverb_jot (struct aave * *aave*, short * *audio*, unsigned *n*)

Run a Jot FDN reverberator to add an artificial reverberation tail to the `n` single channel frames (`n` samples) of each anechoic sound source pointed by `aave->sources`. Store output in `audio`.

Here is the call graph for this function:



Here is the caller graph for this function:

6.1.3.17 void aave_set_listener_orientation (struct aave * *aave,* float *roll,* float *pitch,* float *yaw*)

Set the orientation of the listener's head.

Examples:

examples/circle.c, examples/elevation.c, examples/line.c, and examples/stream.c.

6.1.3.18 void aave_set_listener_position (struct aave * *aave,* float *x,* float *y,* float *z*)

Set the position of the listener.

The aave_update() function should be called afterwards to update the state of the auralisation engine to reflect the new position.

Examples:

examples/circle.c, examples/elevation.c, examples/line.c, and examples/stream.c.

6.1.3.19 void aave_set_source_position (struct aave_source * *source*, float *x*, float *y*, float *z*)

Set the position of a sound source.

The aave_update() function should be called afterwards to update the state of the auralisation engine to reflect the new position.

Examples:

examples/circle.c, examples/elevation.c, examples/line.c, and examples/stream.c.

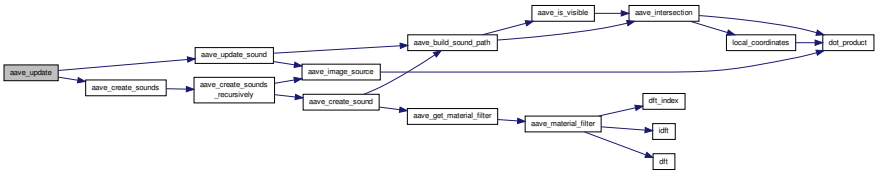
6.1.3.20 void aave_update (struct aave * *aave*)

Update the whole state of the auralisation world. Runs the visibility checks for all sounds from all sources.

Examples:

examples/circle.c, examples/elevation.c, examples/line.c, and examples/stream.c.

Here is the call graph for this function:



6.1.3.21 unsigned dft_index (unsigned *i*, unsigned *n*)

This function returns the index into the DFT data calculated by dft() that contains the Fourier coefficient `i` for a DFT of size `n`. `n` is a power of 2, up to the maximum supported by dft_index_table (currently 128). `i` is a value from 0 up to `n` / 2 - 1, since the input data is real:

- X[0] = X[0].real; X[0].imag = 0;
- X[1] = X[N/2].real; X[N/2].imag = 0;
- X[N/2+i].real = X[N/2-i].real;
- X[N/2+i].imag = - X[N/2-i].imag;

Here is the caller graph for this function:

6.1.3.22 void init_reverb (struct aave_reverb * *reverb*, float *volume*, float *area*, float *abs*)

Initialize reverb parameters.

Here is the caller graph for this function:

6.1.4 Variable Documentation

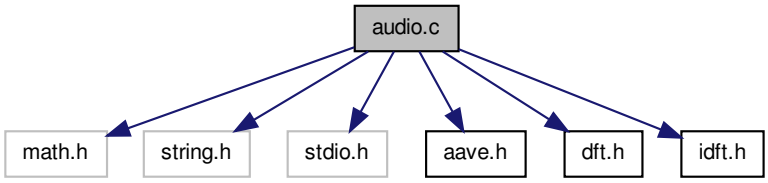
6.1.4.1 struct aave_material aave_material_none

Full reflective material to use when no material is specified for a surface, or when the specified material is not found in aave_materials.

6.2 audio.c File Reference

```
#include <math.h>
#include <string.h>
#include <stdio.h>
#include "aave.h"
#include "dft.h"
#include "idft.h"
```

Include dependency graph for audio.c:



Macros

- #define DFT_TYPE short
- #define IDFT_TYPE int
- #define AAVE_FADE_SAMPLES 4096
- #define AAVE_DISTANCE_B1 0.99977

Functions

- static float attenuation (float distance)
- static float fade_in_gain (unsigned i, unsigned frames)

- static float fade_out_gain (unsigned i, unsigned frames)
- static void cmul (float *a, const float *b, unsigned n)
- static void cmadd (float *y, const float *a, const float *b, unsigned n, float g)
- static void aave_audio_source_block (struct aave_sound *sound, float distance, short *x, unsigned frames, unsigned delay)
- static void aave_hrtf_add_sound (struct aave *aave, struct aave_sound *sound, float ydft[3][2][AAVE_MAX_HRTF *4], unsigned delay, unsigned frames)
- static void aave_hrtf_fill_output_buffer (struct aave *aave, unsigned delay, unsigned frames)
- void aave_get_audio (struct aave *aave, short *buf, unsigned n)
- void aave_put_audio (struct aave_source *source, const short *audio, unsigned n)

6.2.1 Detailed Description

The audio.c file contains the functions that implement the core of the auralisation audio processing of the AcousticAVE library (libaave). The following image illustrates the audio processing model of the library, as seen by the user.

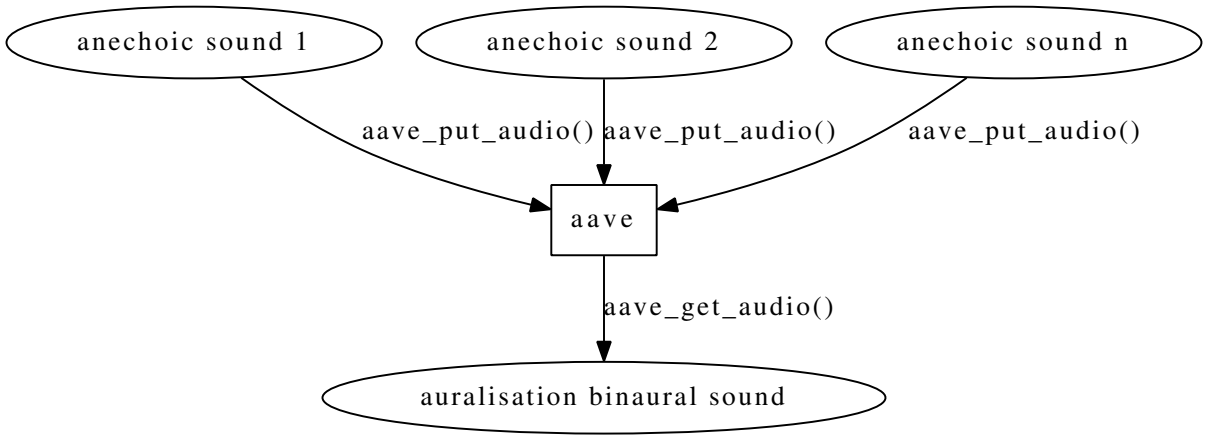


Figure 6.2: Audio processing model

The user passes the anechoic audio data of each sound source in the auralisation world to the library using aave_put_audio() and then retrieves the calculated auralisation binaural audio data using aave_get_audio().

Although the library supports processing one audio frame at a time, users would typically work with blocks of audio frames at a time, for efficiency reasons of the underlying operating system. Each time, the number of anechoic audio frames (samples) in the blocks passed to each source using aave_put_audio() must be the same for all sources, and the same number of binaural audio frames (samples times 2) must be retrieved using aave_get_audio().

The current implementation of the library runs most efficiently when the number of audio frames per block is a multiple of the number of samples of the selected head-related transfer functions (HRTF), times 2, as that is the size of the discrete Fourier transforms (DFT) performed internally.

The HRTF set to use for the auralisation process can be selected by calling one of aave_hrtf_cipic(), aave_hrtf_listen(), aave_hrtf_mit(), or aave_hrtf_tub(), for the CIPIC, LISTEN, MIT, or TU-Berlin HRTF sets, respectively, and it must be done before ever calling aave_put_audio().

The following describes the internal implementation details of the audio processing performed by the library to produce the auralisation. Mere users of the library may want to skip the rest of this section.

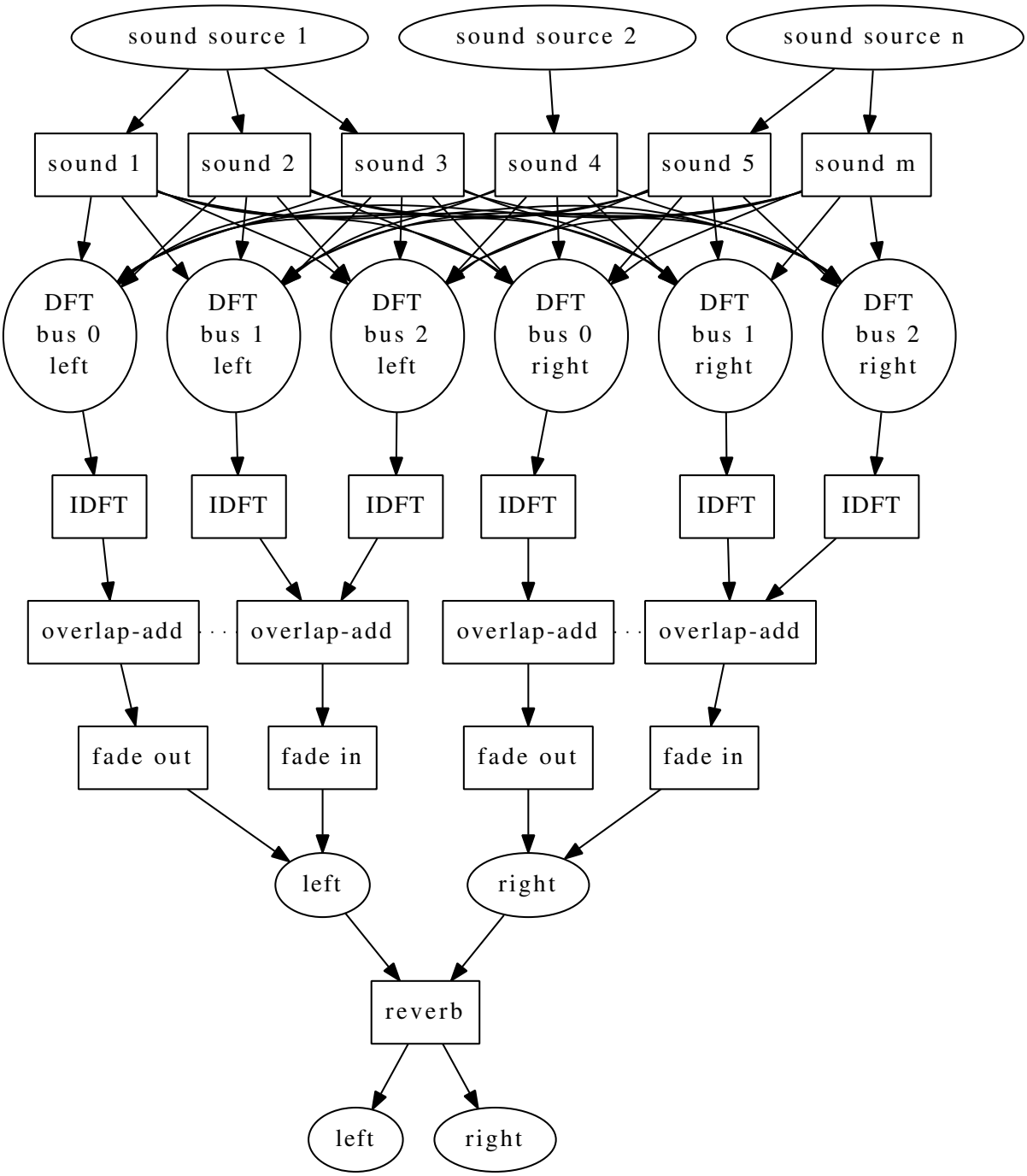


Figure 6.3: Overall audio processing block diagram

The above image illustrates the entire auralisation audio processing, performed by aave_get_audio(), from the anechoic audio data of all sound sources present, to the resulting auralisation left and right (binaural) audio data.

Each sound source originates a number of sounds that reach the listener (direct sound and reflection sounds, as many as the geometry.c part of the auralisation process is able to find in a given amount of time). Each sound is processed differently according to the travelled distance, direction of arrival, and materials of the reflection surfaces, mainly. This audio processing is represented by the sound-i boxes in the image above and is further expanded in the image below.

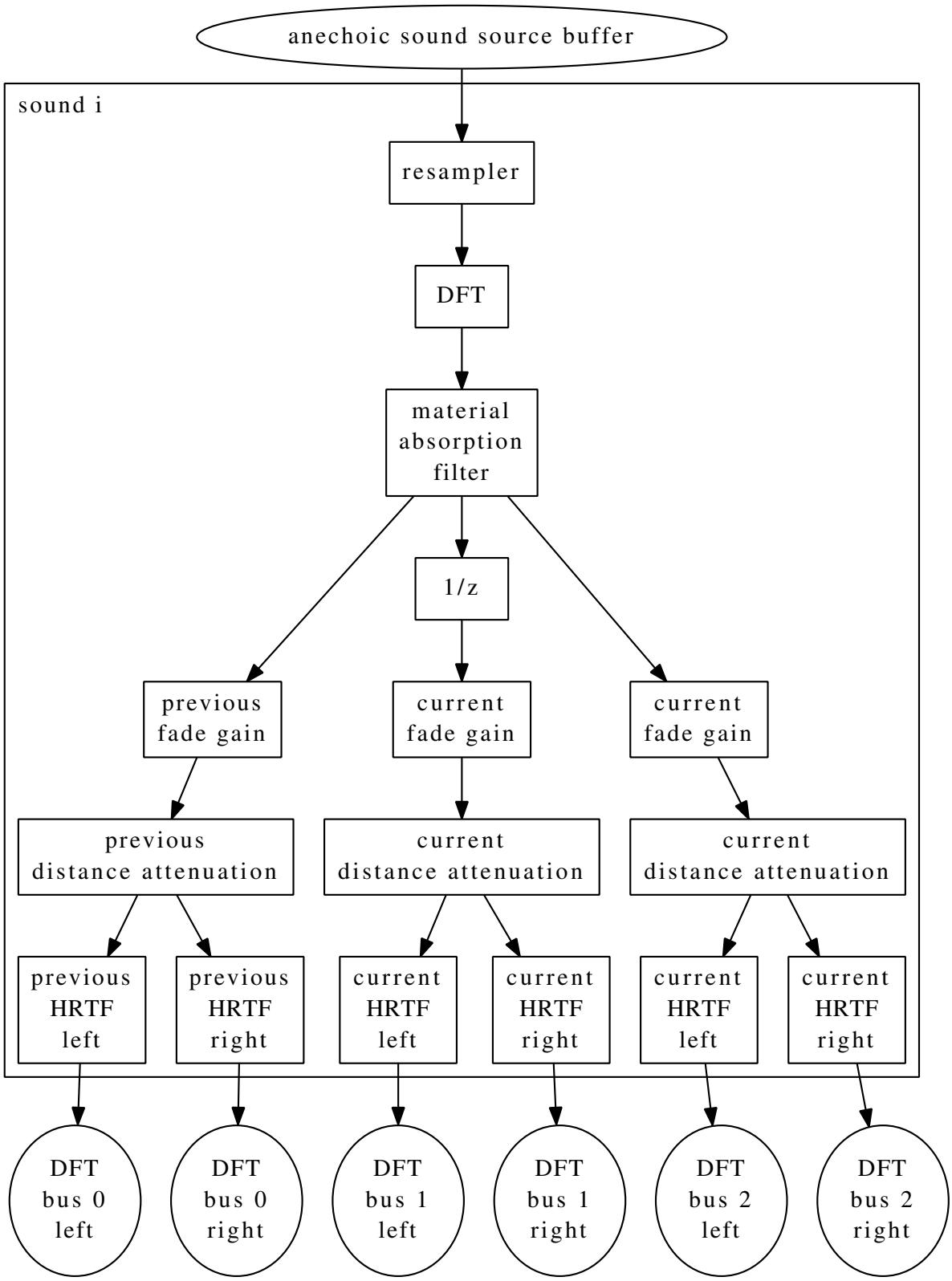


Figure 6.4: Individual sound processing block diagram

The anechoic sound source buffer is a ring buffer that contains the anechoic audio data of the sound source, supplied by aave_put_audio(). Each sound originated from this sound source gets its input audio data from this ring buffer, from the delayed audio sample position corresponding to the travelled distance of that particular sound.

When the listener or sound sources move, the travelled distances change, in a discontinuous manner, since the positions' update rate is usually much lower than the audio sampling rate. The resampler block is responsible for producing a stream of audio data without discontinuities, from the discontinuous distance values. It does this by upsampling and low-pass filtering the distance values, and then interpolating (zero-order) the audio samples in the ring buffer with the corresponding fractional delay, as described in: Peter Brinkman and Michael Gogins, "Doppler effects without equations", Proc. of the 16th Int. Conf. on Digital Audio Effects (DAFx-13).

The HRTF processing is performed in the frequency domain, using the fast convolution method mentioned in: Udo Zolzer, "Digital Audio Signal Processing", 2nd Edition, Section 5.3 Nonrecursive Audio Filters. The DFT block thus zero-pads and converts the audio data from time domain to frequency domain, in dft.h. Note that, without the resampler block, three DFT blocks would be needed instead of just one.

The material absorption filter block implements the sound attenuation by frequency band of the materials of the surfaces where the sound reflects. This filtering is efficiently performed in the frequency domain simply by calculating N complex multiplications, in cmul(). The design of the filter is implemented in material.c.

The anechoic sound is turned into binaural by applying the HRTF pair that corresponds to the direction (elevation and azimuth) of arrival of the sound relative to the listener's head. When the listener moves her head, the HRTF pair changes. This causes discontinuities in the output binaural sound, mainly due to the phase differences between the previous and current HRTF pair. The greater the movement, the more audible these discontinuities are. To mask these discontinuities, it is implemented a method that applies both HRTF pairs, corresponding to the previous and current directions, and then fades-out the previous and fades-in the current. It is a multiple-sound and binaural version of the algorithm described in: Tom Barker et al, "Real-time auralisation system for virtual microphone positioning", Proc. of the 15th Int. Conf. on Digital Audio Effects (DAFx12).

The fade-in/out gains of appearing/disappearing sounds and the amplitude attenuations of sounds with distance are also performed in the frequency domain, taking advantage that they can be applied at the same time the HRTF filter is, simply by multiplying the magnitude of the frequency response by the appropriate combined gain value, in cmadd().

The result of each sound processing block is therefore 3 binaural audio signals (6 total), still in the frequency domain: DFT bus 0 with the current anechoic audio block processed with the previous distance and direction parametres, DFT bus 1 with the previous block processed with the current parametres, and DFT bus 2 with the current block processed with the current parametres.

The binaural audio signals now need to be converted back to the time domain. But instead of performing 6 inverse discrete Fourier transforms (IDFT) per sound and then summing them in the time domain, the different sounds are summed still in the frequency domain, into the DFT busses pictured. This way, only 6 IDFT are performed in total, independently of the number of sounds.

To complete the fast convolution method, the sounds in the 6 busses, now in the time domain, are overlap-added, as described in: Udo Zolzer, "Digital Audio Signal Processing", 2nd Edition, Section 5.3.2 Fast Convolution of Long Sequences.

Todo Here, it might be more efficient to use the overlap-save method instead of the overlap-add method.

Linear fade-outs and fade-ins are then applied to the previous and current parametre busses, respectively, to complete the crossfade method described in Tom Barker et al (see above).

At this point, the left and right signals contain the binaural auralisation of the direct sounds and reflection sounds up to a given reflection order. An artificial reverberation tail, implemented in reverb.c, is finally added to simulate the late reflections that the geometry.c part could not calculate.

6.2.2 Macro Definition Documentation

6.2.2.1 #define AAVE_DISTANCE_B1 0.99977

The b1 coefficient of the single-pole, low-pass, recursive filter implemented in the resampler block to smooth the distance values:

y[n] = b1 ∗ y[n-1] + (1 - b1) ∗ x[n]

Design steps:

d = number of samples for the filter to decay to 36.8%

If the distance is updated at about 10Hz (0.1s), then we could make:

d = 0.1 ∗ fs = 0.1 ∗ 44100 = 4410 samples

Then obtain the b1 coefficient:

b1 = exp(-1/d) = exp(-1/4410) ∼= 0.99977

Reference: The Scientist and Engineer's Guide to DSP, chapter 19.

6.2.2.2 #define AAVE_FADE_SAMPLES 4096

Duration of the fade-in/out for appearing/disappering sounds, in number of audio samples (must be a multiple of AAVE_MAX_HRTF): 4096 samples / 44100 Hz ∼= 93ms.

6.2.2.3 #define DFT_TYPE short

Create a dft() function to convert 16-bit audio samples to frequency.

6.2.2.4 #define IDFT_TYPE int

Create an idft() function to convert frequency to 32-bit audio samples.

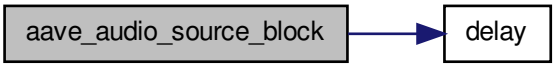
6.2.3 Function Documentation

6.2.3.1 static void aave_audio_source_block (struct aave_sound ∗ *sound,* float *distance,* short ∗ *x,* unsigned *frames,* unsigned *delay*) [static]

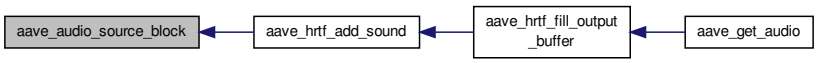
Generate one audio source block. `sound` is the sound whose source to get the anechoic audio data from, `distance` is the distance from the (image) source to the listener, `x` is the buffer to store the generated audio data, `frames` is the number of frames (anechoic samples) to generate, and `delay` is the number of frames of pre-delay to apply to the sound to account for audio user blocks larger than the size of the HRTFs.

The distance value changes abruptly between audio blocks every time the listener or the sound source move. To generate audio blocks without discontinuities, the anechoic samples of the sound source are resampled, using first-order interpolation (linear interpolation), a good compromise between audio quality and processing time, to match an upsampled and low- pass filtered version of the discontinuous distance value, as described in Peter Brinkman and Michael Gogins, "Doppler effects without equations", Proc. of the 16th Int. Conf. on Digital Audio Effects (DAFx-13).

Here is the call graph for this function:



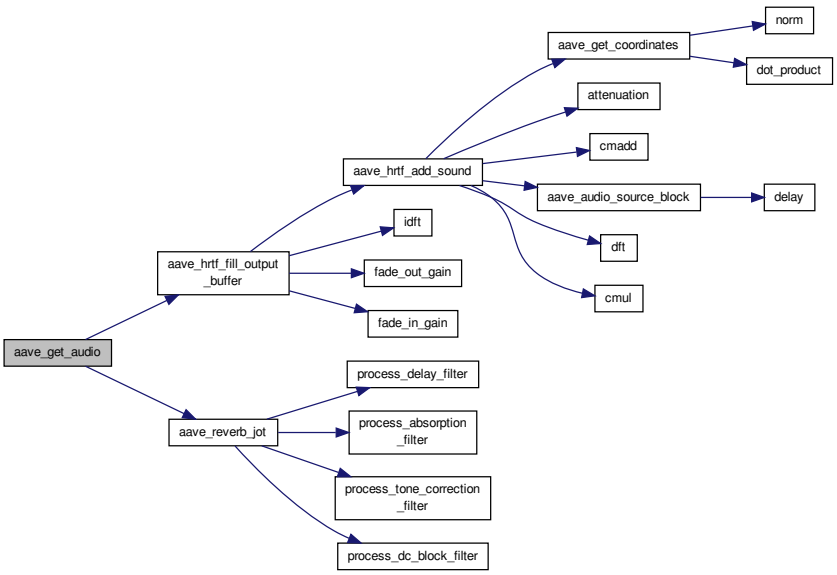
Here is the caller graph for this function:



6.2.3.2 void aave_get_audio (struct aave * *aave*, short * *buf*, unsigned *n*)

Generate `n` 16-bit 2-channel frames of the auralisation world `aave` and put them in the memory location pointed by `buf`.

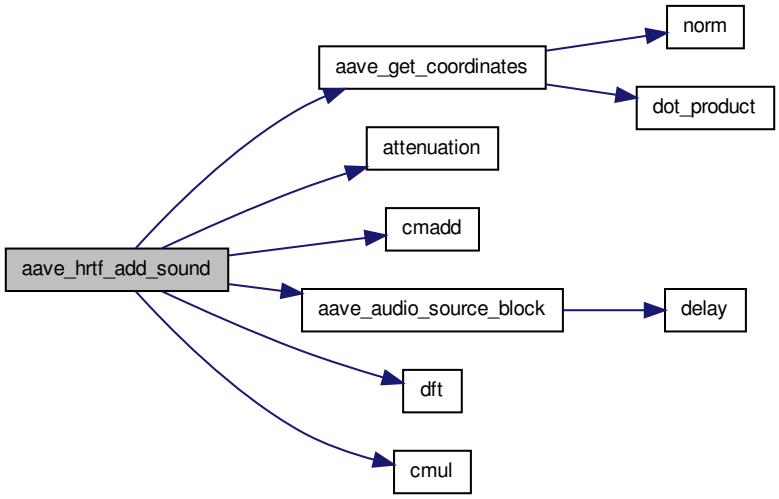
Here is the call graph for this function:



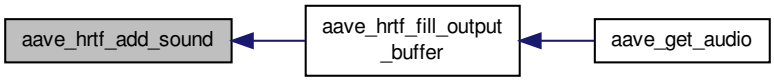
6.2.3.3 static void aave_hrtf_add_sound (struct aave * *aave*, struct aave_sound * *sound*, float *ydft[3][2][AAVE_MAX_HRTF *4]*, unsigned *delay*, unsigned *frames*) [static]

Process one `sound` and add it to the DFT busses `ydft`. `frames` is the number of frames to process. `delay` is the number of frames of pre-delay to apply to the sound to account for audio user blocks larger than the size of the HRTFs.

Here is the call graph for this function:



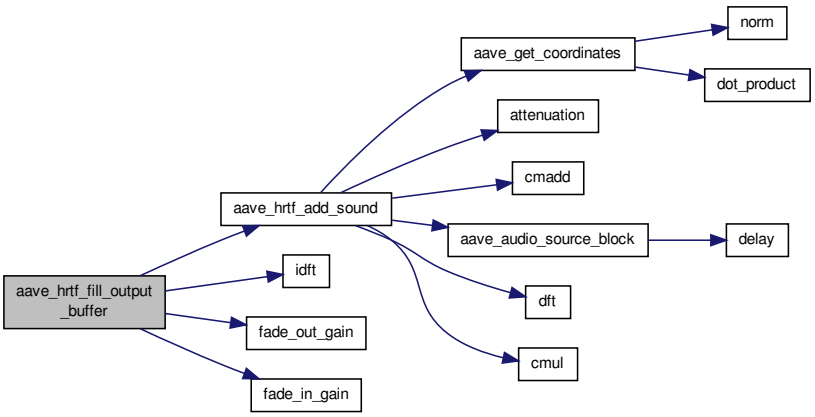
Here is the caller graph for this function:



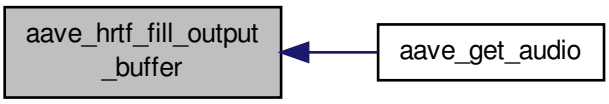
6.2.3.4 static void aave_hrtf_fill_output_buffer (struct aave * *aave*, unsigned *delay*, unsigned *frames*) [static]

Generate one audio buffer of binaural data for the auralisation world `aave` with all sounds in it. `frames` is the number of frames to generate. `delay` is the number of frames of pre-delay to apply to all sounds to account for audio user blocks larger than the size of the HRTFs.

Here is the call graph for this function:



Here is the caller graph for this function:



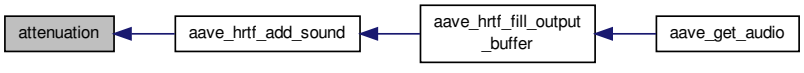
6.2.3.5 void aave_put_audio (struct aave_source * *source*, const short * *audio*, unsigned *n*)

Put the `n` frames pointed by `audio` in the ring buffer of `source`.

6.2.3.6 static float attenuation (float *distance*) [static]

Return the gain corresponding to the amplitude attenuation of a sound at the specified `distance` (m).

Here is the caller graph for this function:



6.2.3.7 static void cmadd (float * *y*, const float * *a*, const float * *b*, unsigned *n*, float *g*) [static]

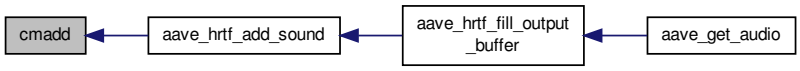
Calculate the Complex Multiplication and ADDition `y += g * a * b` of size `n`.

Y += g ∗ A ∗ B

Y += g ∗ (ar + j ai) ∗ (br + j br)

Y += g ∗ (ar ∗ br - ai ∗ bi) + j g ∗ (ar ∗ bi + ai ∗ br)

Here is the caller graph for this function:



6.2.3.8 static void cmul (float ∗ *a*, const float ∗ *b*, unsigned *n*) [static]

Calculate the Complex MULtiplication `a = a ∗ b` of size `n`.

A = A ∗ B

A = (ar + j ai) ∗ (br + j br)

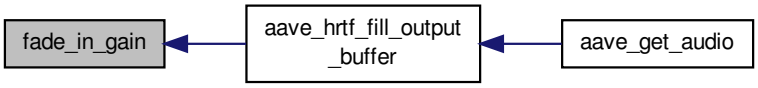
A = (ar ∗ br - ai ∗ bi) + j (ar ∗ bi + ai ∗ br)

Here is the caller graph for this function:

6.2.3.9 static float fade_in_gain (unsigned *i*, unsigned *frames*) [static]

Return the fade-in gain at index `i` for a window of the specified `frames`.

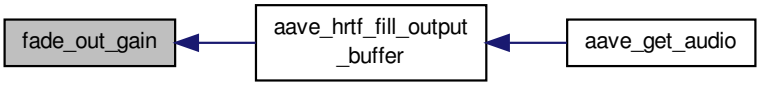
Here is the caller graph for this function:



6.2.3.10 static float fade_out_gain (unsigned *i*, unsigned *frames*) [static]

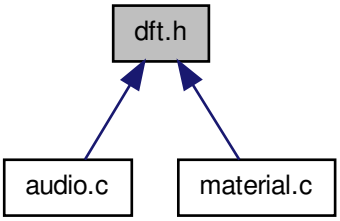
Return the fade-out gain at index `i` for a window of the specified `frames`.

Here is the caller graph for this function:



6.3 dft.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- static void dft (float ∗X, const DFT_TYPE ∗x, unsigned n)

Variables

- const float dftsincos [ ][2]

6.3.1 Detailed Description

The dft.h file implements the discrete Fourier transform (DFT) of real-input data of power-of-2 sizes, using the Cooley-Tukey FFT algorithm (http://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm).

It is about 3 times faster than the equivalent real input to complex-Hermitian output plan fftw_plan_dft_r2c_1d of the fftw library (http://www.fftw.org/). Furthermore, it implicitly type-converts and zero-pads the input data to twice the size, making the comparison even more favorable for this implementation.

The drawback is that the ouput Fourier coefficients end up unordered. However, this is irrevelant for our main purpose of performing fast convolutions, because the corresponding inverse discrete Fourier transform (IDFT) implemented in idft.h also uses the same order. For the case where it is necessary to know the order of the Fourier coefficients, namely the design of the material absorption filters in material.c, the function dft_index() can be used to retrieve the Fourier coefficients in any desired order.

This dft.h file is implemented as a "template". To create a dft() function to use in your source code to transform input data of some type, for example type short (16-bit audio samples), include the following in your source code file:

```
#define DFT_TYPE short
#include "dft.h"
```

This will insert a static dft() function in your source code file that calculates the Fourier coefficients of an array of short integers.

6.3.2 Function Documentation

6.3.2.1 static void dft (float * *X*, const DFT_TYPE * *x*, unsigned *n*) [static]

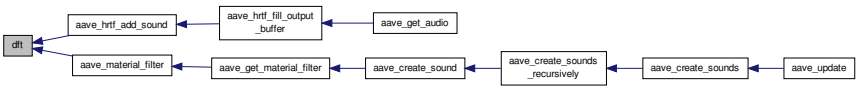
This dft function calculates the `n` point discrete Fourier transform of the zero-padded real-input data values pointed by `x` and stores the Fourier coefficients in `X`. `x` points to `n` / 2 elements (elements `n` / 2 to `n` - 1 are implicitly zero-padded). `X` points to `n` elements, which correspond to the Fourier coefficients 0 to `n` / 2, (un)ordered as follows:

- X[0] = X[0].real;
- X[1] = X[N/2].real;
- X[2] = X[N/4].real;
- X[3] = X[N/4].imag;
- etc... (see dft_index() for the complete ordering)

Remember that when the input data is real:

- X[0].imag = 0;
- X[N/2].imag = 0;
- X[N/2+i].real = X[N/2-i].real;
- X[N/2+i].imag = - X[N/2-i].imag;

Here is the caller graph for this function:



6.3.3 Variable Documentation

6.3.3.1 const float dftsincos[ ][2]

Table with the pre-calculated sin() and cos() values.

6.4 dftindex.c File Reference

Functions

- unsigned dft_index (unsigned i, unsigned n)

Variables

- static const unsigned char dft_index_table [ ]

6.4.1 Detailed Description

The discrete Fourier transform (DFT) implementation in dft.h stores the Fourier coefficients in non-sequential order. This dftindex.c file implements a dft_index() function that returns the index into the calculated DFT data that corresponds to each Fourier coefficient.

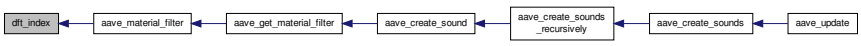
6.4.2 Function Documentation

6.4.2.1 unsigned dft_index (unsigned *i*, unsigned *n*)

This function returns the index into the DFT data calculated by dft() that contains the Fourier coefficient `i` for a DFT of size `n`. `n` is a power of 2, up to the maximum supported by dft_index_table (currently 128). `i` is a value from 0 up to `n` / 2 - 1, since the input data is real:

- X[0] = X[0].real; X[0].imag = 0;
- X[1] = X[N/2].real; X[N/2].imag = 0;
- X[N/2+i].real = X[N/2-i].real;
- X[N/2+i].imag = - X[N/2-i].imag;

Here is the caller graph for this function:



6.4.3 Variable Documentation

6.4.3.1 const unsigned char dft_index_table[ ] [static]

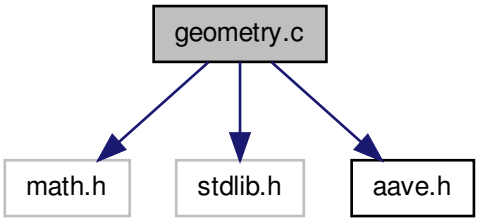
The DFT index lookup table, for N = 128. This means the dft_index() function therefore only supports N <= 128.

Todo If the order of the material absorption filter designed in material.c increases to N > 128, increase this table accordingly.

6.5 geometry.c File Reference

```
#include <math.h>
#include <stdlib.h>
#include "aave.h"
```

Include dependency graph for geometry.c:



Functions

- static float dot_product (const float a[3], const float b[3])
- static void cross_product (float n[3], const float a[3], const float b[3])
- static float norm (const float x[3])
- static void normalise (float y[3], const float x[3])
- static void aave_image_source (const struct aave_surface *surface, const float source[3], float image[3])
- static void local_coordinates (float y[2], const float x[3], const struct aave_surface *surface)
- static int aave_intersection (const struct aave_surface *surface, const float a[3], const float b[3], const float v[3], float xyz[3])
- static int aave_is_visible (const struct aave *aave, const float a[3], const float b[3])
- static int aave_build_sound_path (struct aave *aave, struct aave_source *source, unsigned order, struct aave_surface *surfaces[], float image_sources[][3], float x[][3])
- static void aave_create_sound (struct aave *aave, struct aave_source *source, unsigned order, struct aave_surface *surfaces[], float image_sources[][3])
- static void aave_create_sounds_recursively (struct aave *aave, struct aave_source *source, unsigned order, unsigned o, struct aave_surface *surfaces[], float image_sources[][3])
- static void aave_create_sounds (struct aave *aave, struct aave_source *source, unsigned order)
- static void aave_update_sound (struct aave *aave, struct aave_sound *sound, unsigned order)
- void aave_get_coordinates (const struct aave *aave, const float *source_position, float *distance, float *elevation, float *azimuth)
- void aave_add_source (struct aave *aave, struct aave_source *source)
- void aave_add_surface (struct aave *aave, struct aave_surface *surface)
- void aave_set_listener_orientation (struct aave *aave, float roll, float pitch, float yaw)
- void aave_set_listener_position (struct aave *aave, float x, float y, float z)
- void aave_set_source_position (struct aave_source *source, float x, float y, float z)
- void aave_update (struct aave *aave)

6.5.1 Detailed Description

The geometry.c file contains the functions that implement the geometry part of the auralisation process, based on the image source model shown in: "Auralization: Fundamentals of Acoustics, Modelling, Simulation, Algorithms and Acoustic Virtual Reality", Michael Vorlander, 2008, Section 11.3 Image source model.

At startup, the aave_add_surface() function is called for each surface, and the aave_add_source() function is called for each sound source, to build the auralisation world.

At runtime, the aave_set_listener_position() or aave_set_source_position() functions are called when the listener or sound sources move, followed by aave_update() to perform all geometric calculations to discover the audible

sounds for the new positions, and the aave_set_listener_orientation() function is called when the listener moves her head.

6.5.2 Function Documentation

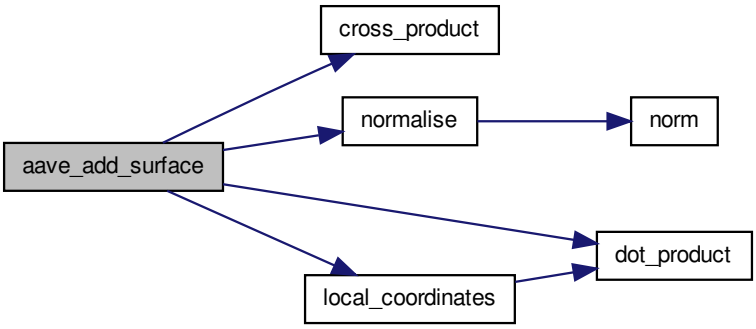
6.5.2.1 void aave_add_source (struct aave * *aave*, struct aave_source * *source*)

Add a sound source to the auralisation world.

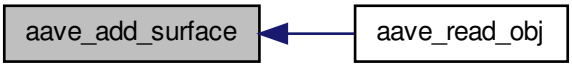
6.5.2.2 void aave_add_surface (struct aave * *aave*, struct aave_surface * *surface*)

Add a surface to the auralisation world.

Here is the call graph for this function:



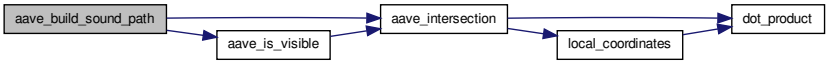
Here is the caller graph for this function:



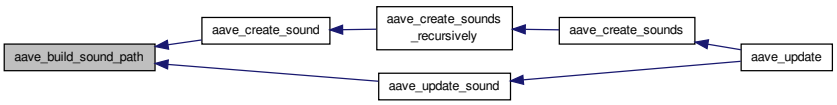
6.5.2.3 static int aave_build_sound_path (struct aave * *aave*, struct aave_source * *source*, unsigned *order*, struct aave_surface * *surfaces[]*, float *image_sources[][3]*, float *x[][3]*) [static]

Create the sound path from the source pointed by to the listerner, for reflection order `order`, that reflects on the specified sequence of `surfaces`, with corresponding image source positions `image_sources`. The calculated reflection points are stored in `x`. Returns 1 if the sound path is audible, or 0 otherwise.

Here is the call graph for this function:



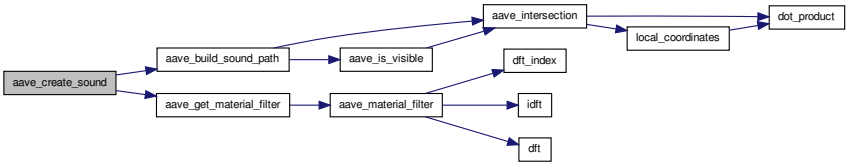
Here is the caller graph for this function:



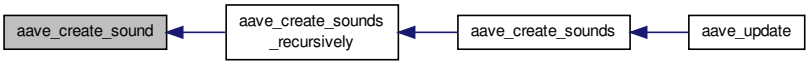
6.5.2.4 static void aave_create_sound (struct aave * *aave,* struct aave_source * *source,* unsigned *order,* struct aave_surface * *surfaces[],* float *image_sources[][3]*) [static]

Create a sound to be auralised by the audio processing. `source` is the sound source that originates the sound, `order` is the reflection order of the sound, `surfaces` is the sequence of surfaces where the sound reflects, and `image_sources` are the positions of the corresponding image-sources.

Here is the call graph for this function:



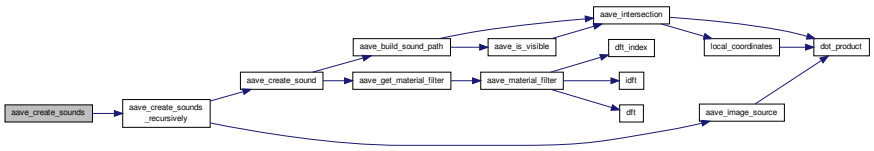
Here is the caller graph for this function:



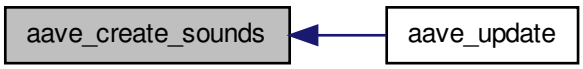
6.5.2.5 static void aave_create_sounds (struct aave * *aave,* struct aave_source * *source,* unsigned *order*) [static]

Create all audible sounds originated from the specified sound source up to, and including, the specified reflection order.

Here is the call graph for this function:



Here is the caller graph for this function:

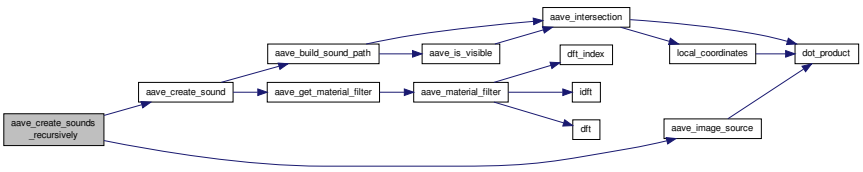


6.5.2.6 static void aave_create_sounds_recursively (struct aave * *aave*, struct aave_source * *source*, unsigned *order*, unsigned *o*, struct aave_surface * *surfaces[]*, float *image_sources[][3]*) [static]

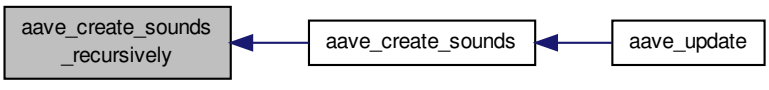
Recursively create all audible sounds of a given reflection order that originate from a sound source. `source` is the sound source, `order` is the reflection order, `o` is the current reflection order in the recursive process, `surfaces` is the stack of surfaces where the current sound reflects, `image_sources` is the stack of corresponding image source positions.

Todo Implement the iterative version of this recursive algorithm.

Here is the call graph for this function:



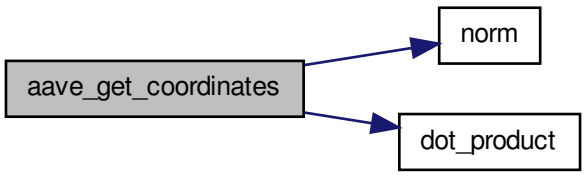
Here is the caller graph for this function:



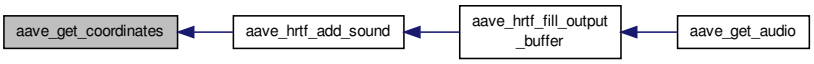
6.5.2.7 void aave_get_coordinates (const struct aave * *aave,* const float * *source_position,* float * *distance,* float * *elevation,* float * *azimuth*)

Get the `distance` (m), `azimuth` (rad) and `elevation` (rad) coordinates of the position `source_position` of a source relative to the listener.

Here is the call graph for this function:



Here is the caller graph for this function:

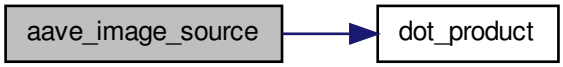


6.5.2.8 static void aave_image_source (const struct aave_surface * *surface,* const float *source[3],* float *image[3]*) [static]

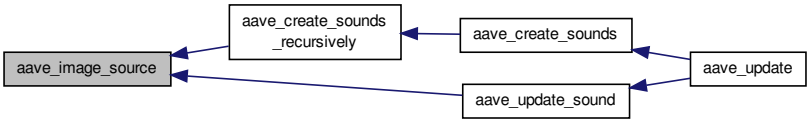
Calculate the position [x,y,z] of the image-source of the sound source at position `source` created by the surface pointed by `surface`. The position of the image-source is returned in `image`.

Reference: "Auralization: Fundamentals of Acoustics, Modelling, Simulation, Algorithms and Acoustic Virtual Reality", Michael Vorlander, 2008, Section 11.3.1 Classical model.

Here is the call graph for this function:



Here is the caller graph for this function:

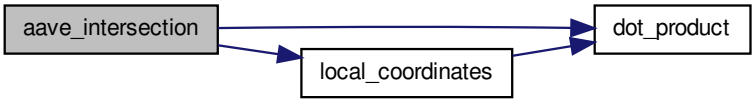


6.5.2.9 static int aave_intersection (const struct aave_surface * *surface*, const float *a[3]*, const float *b[3]*, const float *v[3]*, float *xyz[3]*) [static]

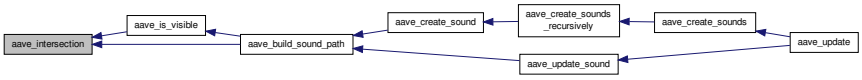
Check if the vector `v` (a line segment from point `a` to point `b`) intersects the surface pointed by `surface`. Returns 0 if false or 1 if true, and the intersection point in `xyz`.

Reference: PNPOLY - Point Inclusion in Polygon Test, W. Randolph Franklin, http://www.ecse.rpi.edu/~wrf/Research/Short_Notes/pnpoly.html

Here is the call graph for this function:



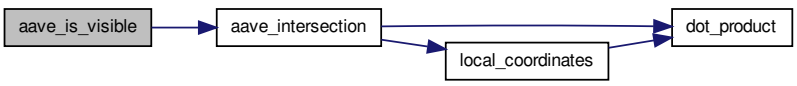
Here is the caller graph for this function:



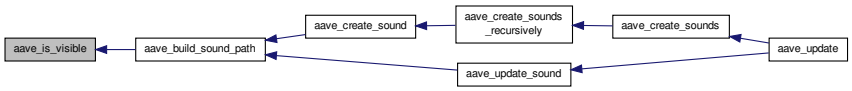
6.5.2.10 static int aave_is_visible (const struct aave * *aave*, const float *a[3]*, const float *b[3]*) [static]

Check if the sound path from point `a` to point `b` is visible. Returns 0 if the line segment b-a is intersected by any surface, or 1 otherwise.

Here is the call graph for this function:



Here is the caller graph for this function:



6.5.2.11 void aave_set_listener_orientation (struct aave * *aave*, float *roll*, float *pitch*, float *yaw*)

Set the orientation of the listener's head.

6.5.2.12 void aave_set_listener_position (struct aave * *aave*, float *x*, float *y*, float *z*)

Set the position of the listener.

The aave_update() function should be called afterwards to update the state of the auralisation engine to reflect the new position.

6.5.2.13 void aave_set_source_position (struct aave_source * *source*, float *x*, float *y*, float *z*)

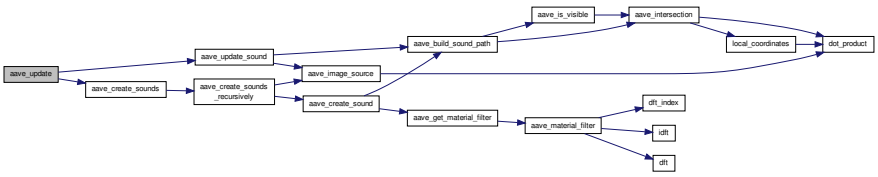
Set the position of a sound source.

The aave_update() function should be called afterwards to update the state of the auralisation engine to reflect the new position.

6.5.2.14 void aave_update (struct aave * *aave*)

Update the whole state of the auralisation world. Runs the visibility checks for all sounds from all sources.

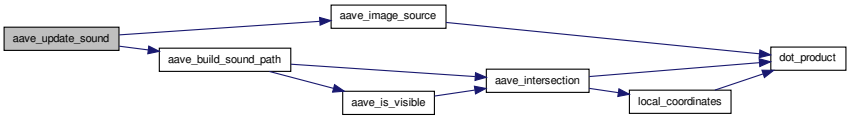
Here is the call graph for this function:



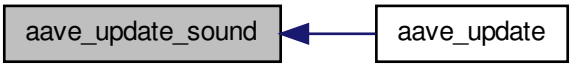
6.5.2.15 static void aave_update_sound (struct aave * *aave*, struct aave_sound * *sound*, unsigned *order*) [static]

Update the distance and azimuth of the listener relative to a sound source. Calculate the (distance, elevation, azimuth) vector from the listener-to-source vector (x, y, z) with listener orientation (roll, pitch, yaw). Angles in radians.

Here is the call graph for this function:



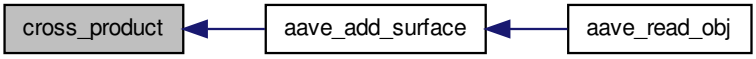
Here is the caller graph for this function:



6.5.2.16 static void cross_product (float *n[3]*, const float *a[3]*, const float *b[3]*) [static]

Calculate the cross product n = a x b.

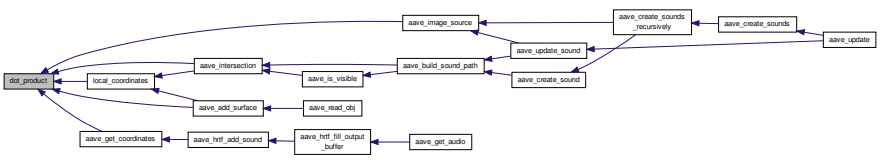
Here is the caller graph for this function:



6.5.2.17 static float dot_product (const float *a[3]*, const float *b[3]*) [static]

Calculate the dot product a . b

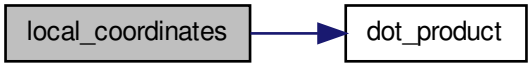
Here is the caller graph for this function:



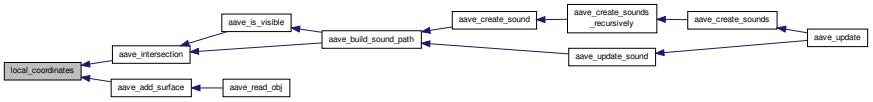
6.5.2.18 static void local_coordinates (float *y[2],* const float *x[3],* const struct aave_surface * *surface*) [static]

Calculate the coordinates `y` [x,y] of the point `x` [x,y,z] in the local coordinates of the surface pointed by `surface`.

Here is the call graph for this function:



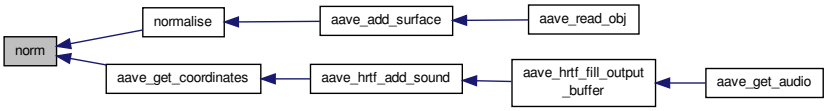
Here is the caller graph for this function:



6.5.2.19 static float norm (const float *x[3]*) [static]

Calculate the norm of a vector.

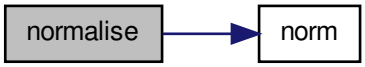
Here is the caller graph for this function:



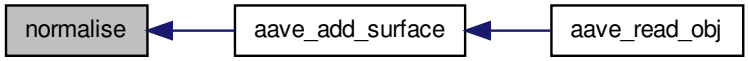
6.5.2.20 static void normalise (float *y[3],* const float *x[3]*) [static]

"Normalise" a vector (divide it by its norm): y = x / norm(x)

Here is the call graph for this function:

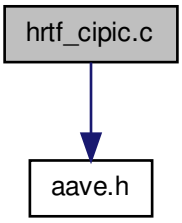


Here is the caller graph for this function:



6.6 hrtf_cipic.c File Reference

`#include "aave.h"`
Include dependency graph for hrtf_cipic.c:



Macros

- #define hrtf_cipic_set hrtf_cipic_set_008

Functions

- static void aave_hrtf_cipic_get (const float ∗hrtf[2], int elevation, int azimuth)
- void aave_hrtf_cipic (struct aave ∗a)

Variables

- const float hrtf_cipic_set_008 [][1024]

6.6.1 Detailed Description

The hrtf_cipic.c file implements the interface to use the CIPIC HRTF set. To select this set for the auralisation process, call aave_hrtf_cipic() after initialising the aave structure and before calling aave_put_audio().

The CIPIC HRTF set consists of head-related impulse responses (HRIR) of 45 subjects. This interface admits using one of them at a time, chosen at compile time.

For each subject, HRIR measurements are available for 64 elevations in 5.625 degree steps (360/64) and the following azimuths: -80, -65, -55, -45 to 45 in 5 degree steps, 55, 65 and 80 degrees. This interface supports all azimuths, but currently only elevation 0.

Each HRIR is 200 samples long at 44100Hz (∼4.5ms). Because the discrete Fourier transform (DFT) implemented in dft.h is optimized for powers of 2, each HRIR is zero-padded to 256 samples for use in this interface.

References: V. R. Algazi, R. O. Duda, D. M. Thompson and C. Avendano, "The CIPIC HRTF Database", Proc. 2001 IEEE Workshop on Applications of Signal Processing to Audio and Electroacoustics, pp. 99-102, Mohonk Mountain House, New Paltz, NY, Oct. 21-24, 2001.

6.6.2 Macro Definition Documentation

6.6.2.1 #define hrtf_cipic_set hrtf_cipic_set_008

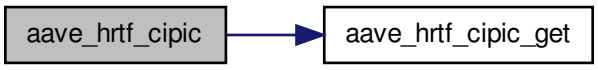
The subject of the CIPIC HRTF set to use (subject 008).

6.6.3 Function Documentation

6.6.3.1 void aave_hrtf_cipic (struct aave * a)

Select the CIPIC HRTF set for the auralisation process.

Here is the call graph for this function:



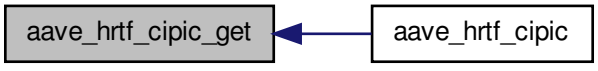
6.6.3.2 static void aave_hrtf_cipic_get (const float * hrtf[2], int elevation, int azimuth) [static]

Get the closest HRTF pair for the specified coordinates. `elevation` is [-90;90] degrees. `azimuth` is [-180;180] degrees. `hrtf`[0] will be the left HRTF, `hrtf`[1] the right HRTF.

Currently, all elevations map to elevation 0.

Todo Use all elevation measures available, not just 0 degrees.

Here is the caller graph for this function:



6.6.4 Variable Documentation

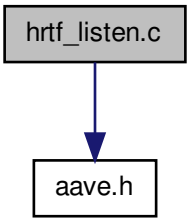
6.6.4.1 const float hrtf_cipic_set_008[ ][1024]

The HRTF set, generated by tools/hrtf_cipic_set.c (subject 008).

6.7 hrtf_listen.c File Reference

```
#include "aave.h"
```

Include dependency graph for hrtf_listen.c:



Functions

- static void aave_hrtf_listen_get (const float *hrtf[2], int elevation, int azimuth)
- void aave_hrtf_listen (struct aave *a)

Variables

- const float hrtf_listen_set_1040 [ ][2][2048]

6.7.1 Detailed Description

The hrtf_listen.c file implements the interface to use the LISTEN (IRCAM/AKG) HRTF set. To select this set for the auralisation process, call aave_hrtf_listen() after initialising the aave structure and before calling aave_put_audio().

The LISTEN HRTF set consists of head-related impulse responses (HRIR) of 51 subjects. This interface admits using one of them at a time, chosen at compile time. Each HRIR is 512 samples long at 44100Hz (~11.6ms).

For each subject, HRIR measurements are available for the following elevations: -45, -30, -15, 0, 15, 30, 45, 60, 75 and 90 degrees. Currently, this interface does not support elevations 60, 75 and 90. The azimuths available are from -180 to 180 degrees, in 15 degree steps.

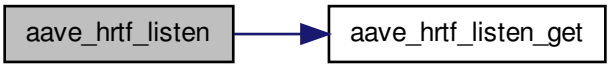
References: http://recherche.ircam.fr/equipes/salles/listen/

6.7.2 Function Documentation

6.7.2.1 void aave_hrtf_listen (struct aave * *a*)

Select the LISTEN HRTF set for the auralisation process.

Here is the call graph for this function:

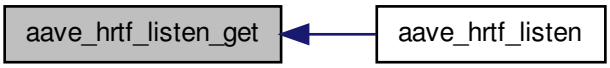


6.7.2.2 static void aave_hrtf_listen_get (const float * *hrtf[2],* int *elevation,* int *azimuth*) [static]

Get the closest HRTF pair for the specified coordinates. `elevation` is [-90;90] degrees. `azimuth` is [-180;180] degrees. `hrtf`[0] will be the left HRTF, `hrtf`[1] the right HRTF.

Todo Elevations 60, 75 and 90.

Here is the caller graph for this function:



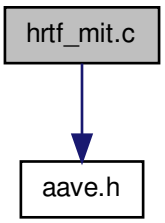
6.7.3 Variable Documentation

6.7.3.1 const float hrtf_listen_set_1040[ ][2][2048]

The HRTF set, generated by tools/hrtf_listen_set.c (subject 1040).

6.8 hrtf_mit.c File Reference

`#include "aave.h"`
Include dependency graph for hrtf_mit.c:



Functions

- static void aave_hrtf_mit_get (const float *hrtf[2], int elevation, int azimuth)
- void aave_hrtf_mit (struct aave *a)

Variables

- const float hrtf_mit_set [][2][512]

6.8.1 Detailed Description

The hrtf_mit.c file implements the interface to use the MIT KEMAR HRTF compact set. To select this set for the auralisation process, call aave_hrtf_mit() after initialising the aave structure and before calling aave_put_audio().

The MIT KEMAR HRTF compact set consists of head-related impulse responses (HRIR) of a KEMAR dummy at 1.4m distance, data-reduced by post-processing to be made more compact. Each HRIR is 128 samples long at 44100Hz (~2.9ms) and only the left ear is stored (the right ear is assumed to be symmetric).

HRIR measurements are available for elevations from -40 to 90 degrees in 10 degree steps. Depending on the elevation, the available azimuths are:

- elevation 90: 1 azimuth measure
- elevation 80: azimuths in 30 degree steps
- elevation 70: azimuths in 15 degree steps
- elevation 60: azimuths in 10 degree steps
- elevation 50: azimuths in 8 degree steps
- elevation 40 and -40: azimuths in 6.5 degree steps
- elevation 30 and -30: azimuths in 6 degree steps
- elevation 20 to -20: azimuths in 5 degree steps

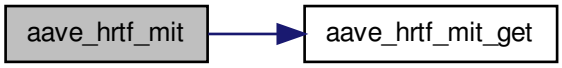
References: http://sound.media.mit.edu/resources/KEMAR.html

6.8.2 Function Documentation

6.8.2.1 void aave_hrtf_mit (struct aave * a)

Select the MIT KEMAR HRTF compact set for the auralisation process.

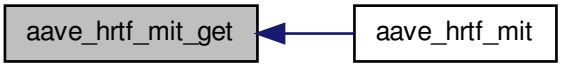
Here is the call graph for this function:



6.8.2.2 static void aave_hrtf_mit_get (const float * *hrtf[2],* int *elevation,* int *azimuth*) [static]

Get the closest HRTF pair for the specified coordinates. `elevation` is [-90;90] degrees. `azimuth` is [-180;180] degrees. `hrtf`[0] will be the left HRTF, `hrtf`[1] the right HRTF.

Here is the caller graph for this function:



6.8.3 Variable Documentation

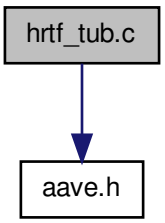
6.8.3.1 const float hrtf_mit_set[ ][2][512]

The HRTF set, generated by tools/hrtf_mit_set.c.

6.9 hrtf_tub.c File Reference

```
#include "aave.h"
```

Include dependency graph for hrtf_tub.c:



Functions

- static void aave_hrtf_tub_get (const float *hrtf[2], int elevation, int azimuth)
- void aave_hrtf_tub (struct aave *a)

Variables

- const float hrtf_tub_set [720][4096]

6.9.1 Detailed Description

The hrtf_tub.c file implements the interface to use the TU-Berlin HRTF set. To select this set for the auralisation process, call aave_hrtf_tub() after initialising the aave structure and before calling aave_put_audio().

The TU-Berlin HRTF set consists of head-related impulse responses (HRIR) of a KEMAR manikin at 3m, 2m, 1m and 0.5m distances, in the horizontal plane (elevation 0 only), all 360 degrees in azimuth in steps of 1 degree. This interface admits using one of the distance sets at a time, chosen at compile time.

Each HRIR is 2048 samples long at 44100Hz (~46.4ms). For computational efficiency (memory and processing), this interface only uses the first 1024 samples.

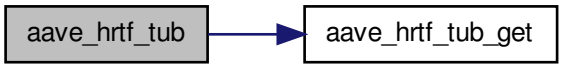
References: Hagen Wierstorf, Matthias Geier, Alexander Raake and Sascha Spors, "A Free Database of Head--Related Impulse Response Measurements in the Horizontal Plane with Multiple Distances", 130th Convention of the Audio Engineering Society, May 2011.

6.9.2 Function Documentation

6.9.2.1 void aave_hrtf_tub (struct aave * *a*)

Select the TU-Berlin HRTF set for the auralisation process.

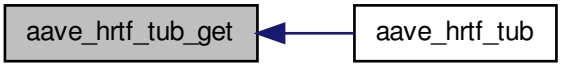
Here is the call graph for this function:



6.9.2.2 static void aave_hrtf_tub_get (const float * *hrtf[2],* int *elevation,* int *azimuth*) [static]

Get the closest HRTF pair for the specified coordinates. `elevation` is [-90;90] degrees. `azimuth` is [-180;180] degrees. `hrtf`[0] will be the left HRTF, `hrtf`[1] the right HRTF.

Here is the caller graph for this function:



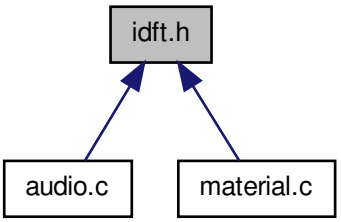
6.9.3 Variable Documentation

6.9.3.1 const float hrtf_tub_set[720][4096]

The HRTF set, generated by tools/hrtf_mit_tub.c (distance 2m).

6.10 idft.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- static void idft (IDFT_TYPE *x, float *X, unsigned n)

Variables

- const float dftsincos [][2]

6.10.1 Detailed Description

The idft.h file implements the inverse discrete Fourier transform (IDFT) of Fourier coefficients of real-input data of power-of-2 sizes, using the Cooley-Tukey FFT algorithm (http://en.wikipedia.org/wiki/Cooley%-E2%80%93Tukey_FFT_algorithm).

It is about 2 times faster than the equivalent complex-Hermitian input to real output plan fftw_plan_dft_c2r_1d of the fftw library (http://www.fftw.org/).

The input Fourier coefficients use the same order as returned by the discrete Fourier transform implemented in dft.h. However, the ouput real data values are correctly ordered.

This idft.h file is implemented as a "template". To create an idft() function to use in your source code to transform Fourier coefficients back into real data of some type, for example type int (32-bit audio samples), include the following in your source code:

```
#define IDFT_TYPE int
#include "idft.h"
```

This will insert a static idft() function in your source code file that transforms Fourier coefficients back into the corresponding integers.

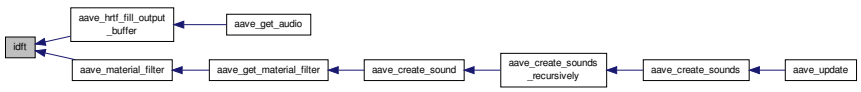
6.10.2 Function Documentation

6.10.2.1 static void idft (IDFT_TYPE * x, float * X, unsigned n) [static]

This idft function calculates the `n` point inverse discrete Fourier transform of the Fourier coefficients pointed by `X` and stores the real output data in `x`. `X` points to `n` elements, which correspond to the Fourier coefficients 0 to `n` / 2, in the order described in dft.h. `x` points to `n` elements correctly ordered.

Todo round instead of truncate

Here is the caller graph for this function:



6.10.3 Variable Documentation

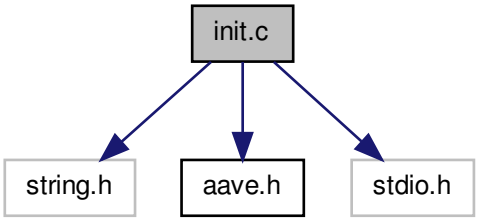
6.10.3.1 const float dftsincos[][2]

Table with the pre-calculated sin() and cos() values.

6.11 init.c File Reference

```
#include <string.h>
#include "aave.h"
#include "stdio.h"
```

Include dependency graph for init.c:



Functions

- void aave_init (struct aave *aave, unsigned rt60)
- void aave_init_source (struct aave *aave, struct aave_source *source)

6.11.1 Detailed Description

The init.c file contains the functions to initialise the data structures used in the AcousticAVE library.

6.11.2 Function Documentation

6.11.2.1 void aave_init (struct aave * *aave,* unsigned *rt60*)

Initialise the auralisation data structure.

The listener's initial position is (0, 0, 0).

The listener's head initial orientation is invalid! You must call aave_set_listener_orientation()!

The initial output gain is 1 (0dB).

The artificial reverberation tail is initially enabled.

Here is the call graph for this function:

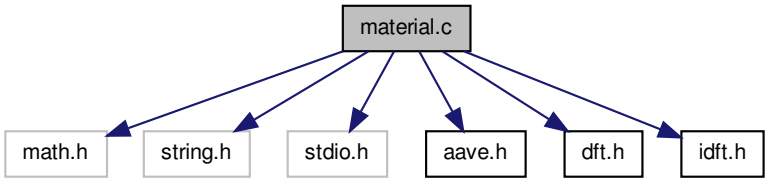
6.11.2.2 void aave_init_source (struct aave * *aave*, struct aave_source * *source*)

Initialise a sound source data structure to be used by the aave engine.

6.12 material.c File Reference

```
#include <math.h>
#include <string.h>
#include <stdio.h>
#include "aave.h"
#include "dft.h"
#include "idft.h"
```

Include dependency graph for material.c:



Macros

- #define DFT_TYPE float
- #define IDFT_TYPE float
- #define N 128
- #define **print_dft**(x, n)
- #define **print_vec**(h, n)

Functions

- struct aave_material * aave_get_material (const char *name)
- static void aave_material_filter (const float *k, float *x, unsigned n)
- void aave_get_material_filter (struct aave *aave, struct aave_surface **surfaces, unsigned reflections, float *filter)

Variables

- static struct aave_material aave_materials []
- struct aave_material aave_material_none

6.12.1 Detailed Description

The material.c file contains the table of materials, with their acoustic reflection factors by frequency band, and the functions to design the material absorption audio filters to apply those reflection factors.

The filtering, implemented in audio.c, is performed in the frequency domain, taking advantage of the fact that the sounds are already converted to the frequency domain anyway to perform the HRTF filtering. This allows for the

use of material absorption filters with linear phase (best audio quality) and long impulse response (best frequency resolution). The fixed delay induced by the length of the filter can be easily compensated upstream in the auralisation process, if needed.

Calculating the Fourier coefficients of the filters is thus implemented using the technique of filter design by frequency sampling, shown in: Udo Zolzer, "Digital Audio Signal Processing", 2nd Edition, Wiley, Section 5.3.3 Filter Design by Frequency Sampling.

6.12.2 Macro Definition Documentation

6.12.2.1 #define DFT_TYPE float

Create the dft() function to convert filter coefficients to Fourier coefficients.

6.12.2.2 #define IDFT_TYPE float

Create the idft() function to convert Fourier coefficients to filter coefficients.

6.12.2.3 #define N 128

The legth of the filter to design. Should be the same length as the smallest HRIR (MIT = 128).

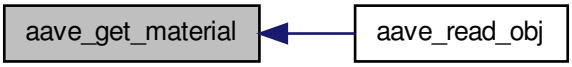
6.12.3 Function Documentation

6.12.3.1 struct aave_material* aave_get_material (const char * *name*)

Return the material with the specified `name`. If no material is found with such name, return aave_material_none.

The search is performed using the binary search algorithm (`http://en.wikipedia.org/wiki/Binary_search_algorithm`), that's why the table of materials must be ordered by name.

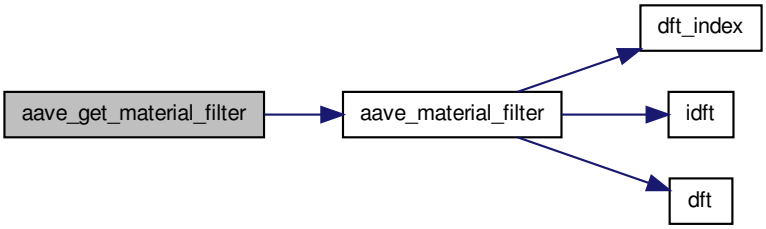
Here is the caller graph for this function:



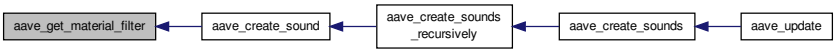
6.12.3.2 void aave_get_material_filter (struct aave * *aave,* struct aave_surface ** *surfaces,* unsigned *reflections,* float * *filter*)

Design the material absorption filter for the specified sequence of `surfaces` and reflection order `reflections`. The calculated DFT coefficients of the filter are stored in `filter`, which must have 4 times the elements of the HRIRs of the HRTF set currently in use.

Here is the call graph for this function:



Here is the caller graph for this function:

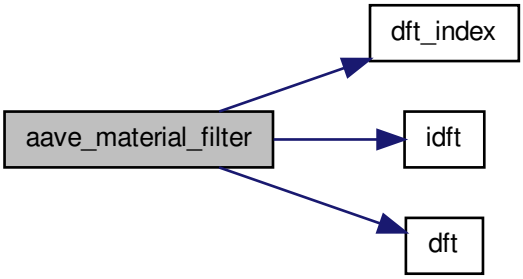


6.12.3.3 static void aave_material_filter (const float * *k*, float * *x*, unsigned *n*) [static]

Design the material absorption filter for the reflection factors `k`. The calculated DFT coefficients of the filter are stored in `x`, which must have at least 4 * `n` elements to account for the zero-padding, and where `n` is the size of the HRIRs of the HRTF currently in use.

Reference: Udo Zolzer, "Digital Audio Signal Processing", 2nd Edition, Wiley, Section 5.3.3 Filter Design by Frequency Sampling.

Here is the call graph for this function:



Here is the caller graph for this function:

6.12.4 Variable Documentation

6.12.4.1 struct aave_material aave_material_none

Initial value:

```
= {
    0, { 100, 100, 100, 100, 100, 100, 100 }
}
```

Full reflective material to use when no material is specified for a surface, or when the specified material is not found in aave_materials.

6.12.4.2 struct aave_material aave_materials[] [static]

Initial value:

```
= {
    { "carpet", { 96, 96, 84, 63, 50, 45, 45 } },
    { "concrete", { 99, 98, 97, 99, 99, 99, 99 } },
    { "cotton_curtains", { 84, 74, 59, 66, 64, 54, 54 } },
    { "glass_window", { 95, 97, 98, 98, 98, 98, 98 } },
    { "thin_plywood", { 76, 89, 95, 96, 97, 97, 97 } },
}
```

Table of materials, ordered by name.

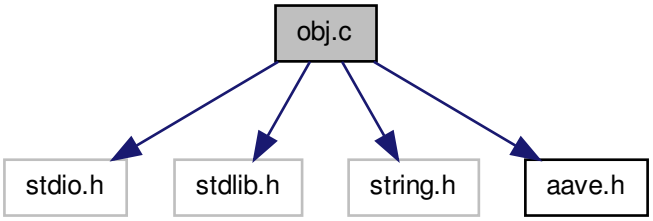
The reflection factors are calculated from random-incidence absorption coefficient alpha values using the equation: r = sqrt(1 - alpha).

Reference: "Auralization: Fundamentals of Acoustics, Modelling, Simulation, Algorithms and Acoustic Virtual Reality", Michael Vorlander, 2008, Annex - Tables of random-incidence absorption coefficients, alpha, and Section 11.3.2 Audability test.

6.13 obj.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "aave.h"
```

Include dependency graph for obj.c:



Macros

- #define MAX_VERTICES 1024

Functions

- void aave_read_obj (struct aave *aave, const char *filename, unsigned volume, unsigned area)

6.13.1 Detailed Description

Read room model from Wavefront .obj file.

The following elements of the .obj specification are supported: v (vertex), f (face), usemtl (material name). All other elements are ignored, as they are irrelevant for auralisation.

The material name in the usemtl element specify the material of the aave_materials table to use for the succeeding face.

References: Wavefront .obj file: http://en.wikipedia.org/wiki/Wavefront_.obj_file

6.13.2 Macro Definition Documentation

6.13.2.1 #define MAX_VERTICES 1024

Maximum number of vertices in the .obj file.

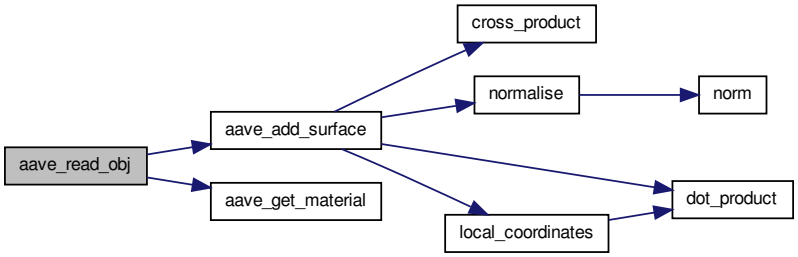
Todo Use dynamic memory allocation for the array of vertices to support "unlimited" number of vertices.

6.13.3 Function Documentation

6.13.3.1 void aave_read_obj (struct aave * *aave*, const char * *filename*, unsigned *volume*, unsigned *area*)

Read the .obj file `filename` and add its contents to the auralisation engine `aave`.

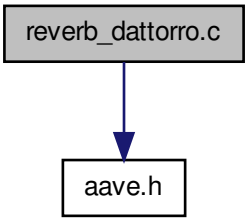
Here is the call graph for this function:



6.14 reverb_dattorro.c File Reference

```
#include "aave.h"
```

Include dependency graph for reverb_dattorro.c:



Data Structures

- struct delay
- struct lowpass
- struct allpass
- struct decay_block

Macros

- #define PREDELAY (0.15 ∗ AAVE_FS)
- #define BANDWIDTH 0.7
- #define INPUT_DIFFUSION_1 0.65
- #define **INPUT_DIFFUSION_2** 0.6
- #define DECAY_DIFFUSION_1 0.625
- #define **DECAY_DIFFUSION_2** 0.7
- #define DECAY 0.8
- #define DAMPING 0.7
- #define WET 0.3

Functions

- static float delay (struct delay *d, float x, unsigned k)
- static float lowpass (struct lowpass *lp, float x, float b)
- static float allpass (struct allpass *ap, float x, float g, unsigned delay)
- static void decay_block (struct decay_block *b, float x, unsigned i, float *out1, float *out2, float *out3)
- void aave_reverb_dattorro (struct aave *aave, short *audio, unsigned n)

6.14.1 Detailed Description

The reverb_dattorro.c file implements a Dattorro reverberator to add an artificial reverberation tail to the output of the auralisation generated by the audio.c part of the auralisation process, to simulate the late reflections that the geometry.c part of the auralisation process could not determine in time.

Todo Make the code reentrant (move the static structures to aave).

Reference: Jon Dattorro, "Effect Design, Part 1: Reverberator and Other Filters", J. Audio Eng. Soc (AES), Vol. 45, No 9, Sep 1997.

6.14.2 Macro Definition Documentation

6.14.2.1 #define BANDWIDTH 0.7

Bandwidth of the early low-pass filter.

6.14.2.2 #define DAMPING 0.7

Damping of the late low-pass filters.

6.14.2.3 #define DECAY 0.8

Decay parameter.

6.14.2.4 #define DECAY_DIFFUSION_1 0.625

Decay diffusion gain of the late all-pass filters.

6.14.2.5 #define INPUT_DIFFUSION_1 0.65

Input diffusion gains of the early all-pass filters.

6.14.2.6 #define PREDELAY (0.15 * AAVE_FS)

Pre-delay, in number of samples (150ms).

6.14.2.7 #define WET 0.3

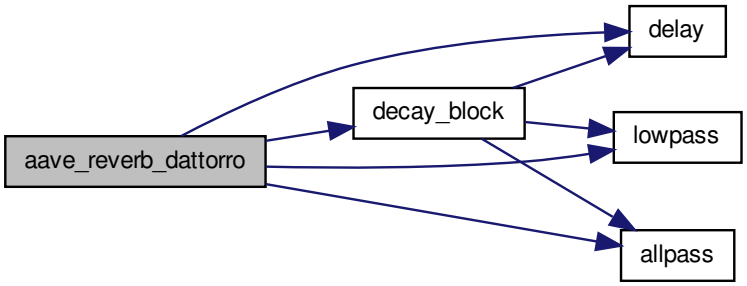
Gain of the wet path of the reverberator.

6.14.3 Function Documentation

6.14.3.1 void aave_reverb_dattorro (struct aave * *aave*, short * *audio*, unsigned *n*)

Run a Dattorro reverberator to add an artificial reverberation tail to the `n` binaural frames (2 * `n` samples) pointed by `audio`.

Here is the call graph for this function:

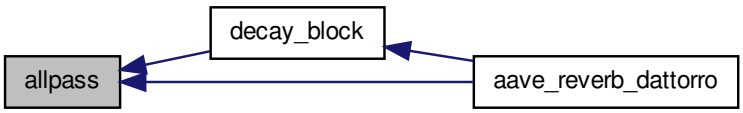


6.14.3.2 static float allpass (struct allpass * *ap*, float *x*, float *g*, unsigned *delay*) [static]

Execute an all-pass filter: H(z) = (g + z^-k) / (1 + g z^-k). `ap` is the all-pass filter data, `x` is the input value, `g` is the gain coefficient, and `delay` is the number of samples of delay. Returns the output value of the all-pass filter.

Todo Check if the tap is really x1 or x2.

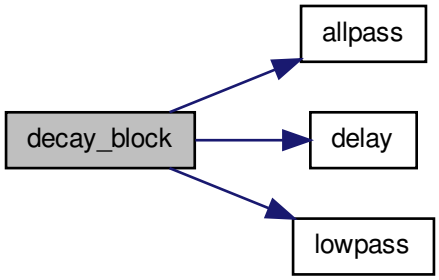
Here is the caller graph for this function:



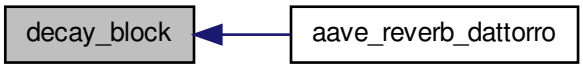
6.14.3.3 static void decay_block (struct decay_block * *b*, float *x*, unsigned *i*, float * *out1*, float * *out2*, float * *out3*) [static]

Execute a decay block. `b` is the decay block data, `x` is the input value, `i` is the index of the decay block parameters to use (0 or 1), and `out1`, `out2`, `out3` are the output taps.

Here is the call graph for this function:



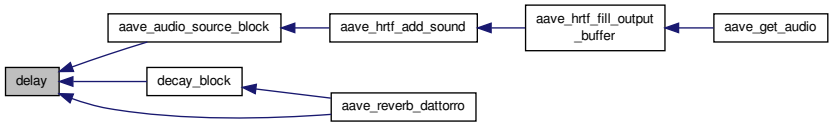
Here is the caller graph for this function:



6.14.3.4 static float delay (struct delay * *d*, float *x*, unsigned *k*) [static]

Execute a delay block: y[n] = x[n - k]. `d` is the delay block data, `x` is the input value, and `k` is the number of samples of delay. Returns the output value y[n].

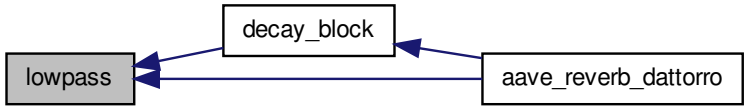
Here is the caller graph for this function:



6.14.3.5 static float lowpass (struct lowpass * *lp*, float *x*, float *b*) [static]

Execute a low-pass filter: y[n] = b * x[n] + (1 - b) * y[n-1]. `lp` is the low-pass filter data, `x` is the input value, `b` is the bandwidth/damping coefficient. Returns the output value y[n].

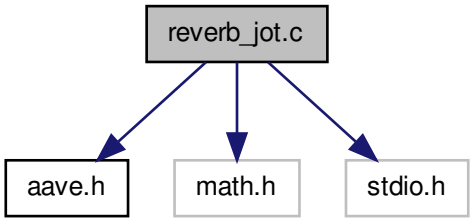
Here is the caller graph for this function:



6.15 reverb_jot.c File Reference

```
#include "aave.h"
#include "math.h"
#include "stdio.h"
```

Include dependency graph for reverb_jot.c:



Data Structures

- struct delay_filter
- struct absorption_filter
- struct tone_correction_filter
- struct dc_block_filter

Macros

- #define MAX_DELAY_TIME 8820

Functions

- static float process_delay_filter (struct delay_filter ∗d, float x, unsigned k)
- static float process_absorption_filter (struct absorption_filter ∗af, float x, float g, float b)
- static float process_tone_correction_filter (struct tone_correction_filter ∗tcf, float x, float b)
- static float process_dc_block_filter (struct dc_block_filter ∗dcbf, float x)
- void **print_reverb_parameters** (struct aave ∗aave, struct aave_reverb ∗rev)

- void init_reverb (struct aave_reverb *reverb, float volume, float area, float abs)
- void aave_reverb_jot (struct aave *aave, short *audio, unsigned n)

Variables

- static const short feedback_delays [FDN_ORDER] = {29,53,79,101,127,149,173,197,223,251,277,307,331,353,379,401,431,45
- struct dc_block_filter **dcbf** [] = {{0,0,0.01},{0,0,0.01}}

6.15.1 Detailed Description

The reverb_jot.c file implements a classic Jot FDN reverberator to add an artificial reverberation tail to each anechoic sound source pointed by `aave->sources`, to simulate the late reflections that the geometry.c part of the auralisation process could not determine in time.

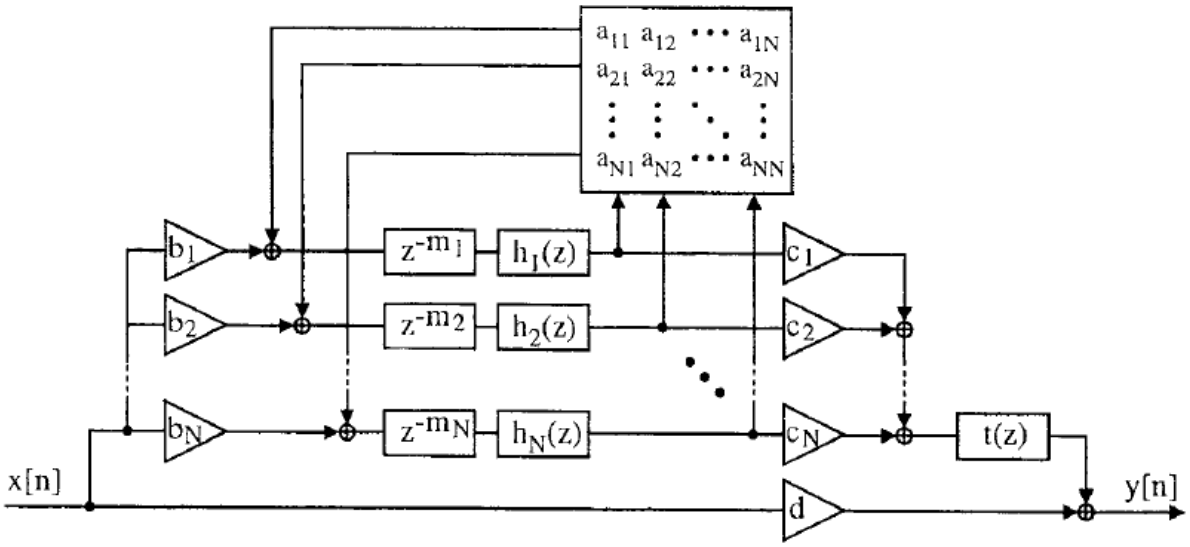


Figure 6.5: Jot's Feedback Delay Network diagram

N = 64 is the currently adopted circulation matrix order, b[N] is a unitary vector (identity, not implemented) and c[N,2] is a two collumn matrix for decorrelation of stereo output. The circulating matrix a[N,N] is of Householder type, represented by the equation a[N,N] = j[N,N] - 2/N * u[N] * u[N]^T where j[N,N] is a permutation matrix and u[N] a unitary collumn vector. Amplitude and high frequency damping are performed by `absorption_filter` h(z) and high pass tone correction is performed by `tone_correction_filter` t(z), for left and right outputs. `delay_filter` z sizes `feedback_delays` are chosen from a set of prime numbers within the range 0-1800.

Reverb predelay `Tmixing` is calculated acording to Jot's aproximation Tmixing = sqrt(Volume). Reverb constant amplitude atenuation is calculated as 1/rc, where rc = pow((area * abs)/(16 * PI),0.5) is the critical distance at which direct sound energy is equal to reverberation energy, area is the total surface area for the room and abs is the average absorption coeficient of the room. The `pre_delay` value used is the Tmixing predelay value summed with hrtf buffer size to compensate the latency introduced by HRTF processing.

Todo A `dc_block_filter` was introduced to aproximate low frequency damping. Improve this filter (or introduce another) for flexible bandwidth selection.

Reference: Jasmin Frenette, "REDUCING ARTIFICIAL REVERBERATION ALGORITHM REQUIREMENTS USING TIME-VARIANT FEEDBACK DELAY NETWORKS", Master Thesis, Dec 2000.

6.15.2 Macro Definition Documentation

6.15.2.1 #define MAX_DELAY_TIME 8820

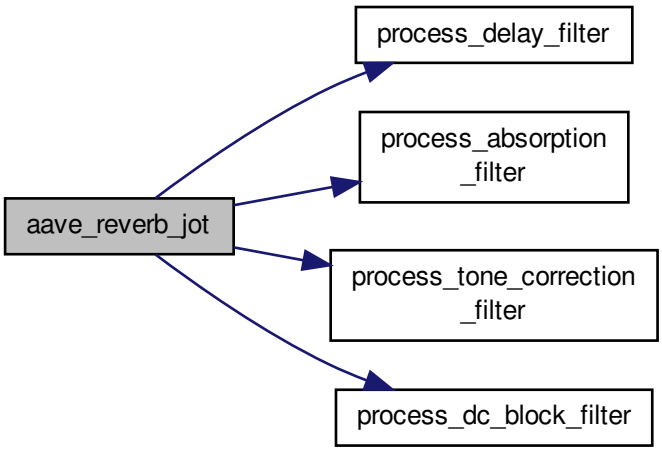
Define a maximum delay time of 200 ms

6.15.3 Function Documentation

6.15.3.1 void aave_reverb_jot (struct aave * *aave*, short * *audio*, unsigned *n*)

Run a Jot FDN reverberator to add an artificial reverberation tail to the `n` single channel frames (`n` samples) of each anechoic sound source pointed by `aave->sources`. Store output in `audio`.

Here is the call graph for this function:



Here is the caller graph for this function:

6.15.3.2 void init_reverb (struct aave_reverb * *reverb*, float *volume*, float *area*, float *abs*)

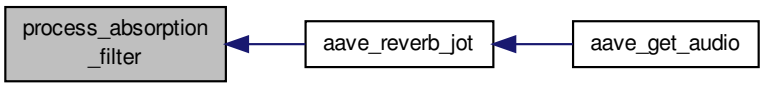
Initialize reverb parameters.

Here is the caller graph for this function:

6.15.3.3 static float process_absorption_filter (struct absorption_filter * *af,* float *x,* float *g,* float *b*) [static]

Execute a low-pass filter with gain atenuation: y[n] = g * (1-b) * x[n] + b * y[n-1]. `lp` is the low-pass filter data, `x` is the input value, `b` is the bandwidth/damping coefficient. `b` is the gain atenuation. Returns the output value y[n].

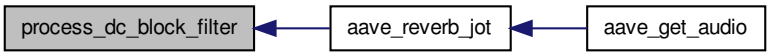
Here is the caller graph for this function:



6.15.3.4 static float process_dc_block_filter (struct dc_block_filter * *dcbf,* float *x*) [static]

Execute a dc block filter: y[n] = g * (x[n] - x[n-1]) + (1-b) * y[n-1]. `dcbf` is the dc block filter data, `x` is the input value, Returns the output value dcbf->y.

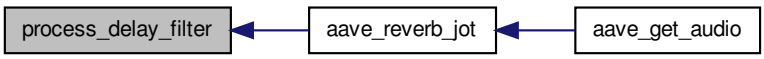
Here is the caller graph for this function:



6.15.3.5 static float process_delay_filter (struct delay_filter * *d,* float *x,* unsigned *k*) [static]

Execute a delay block: y[n] = x[n - k]. `d` is the delay block data, `x` is the input value, and `k` is the number of samples of delay. Returns the output value y[n].

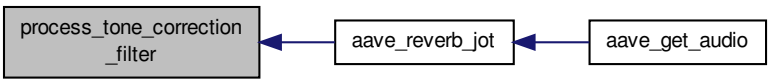
Here is the caller graph for this function:



6.15.3.6 static float process_tone_correction_filter (struct tone_correction_filter * *tcf,* float *x,* float *b*) [static]

Execute a tone correction (high pass) filter: y[n] = g * (x[n] - b * x[n-1]) / 1-b. `tcf` is the high-pass filter data, `x` is the input value, `b` is the bandwidth/damping coefficient. Returns the output value y[n].

Here is the caller graph for this function:



6.15.4 Variable Documentation

6.15.4.1 const short feedback_delays[FDN_ORDER] = {29,53,79,101,127,149,173,197,223,251,277,307,331,353,379,401,431,457,479,503,541,563,587,613,641,673,701,727,751,773,797,821,853,877,907 [static]

N = 64 delay sizes for each fdn delay line. Prime numbers within the range 0 - 1800.

Chapter 7

Example Documentation

7.1 examples/circle.c

An example of auralisation of a moving sound source describing a circle around the listener.

```
/*
 * libaave/examples/circle.c: sinusoid sound circling around listener
 *
 * Copyright 2013 Universidade de Aveiro
 *
 * Funded by FCT project AcousticAVE (PTDC/EEA-ELC/112137/2009)
 *
 * Written by Andre B. Oliveira <abo@ua.pt>
 */

/*
 * Usage: ./circle > output.raw
 */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "aave.h"

/* The frequency of the sinusoid (Hz). */
#define F 1000

/* The distance the sound source is from the listener (m). */
#define D 1

/* The angular velocity that the sound source is moving at (rad/s). */
#if 0
#define W (2 * M_PI / 10) /* do one full circle in 10 seconds */
#else
#define W (2 * M_PI) /* do one full circle per second */
#endif

int main()
{
    struct aave *aave;
    struct aave_source *source;
    double angle;
    float x, y;
    unsigned n;
    short in, out[2];

    /* Initialise auralisation engine. */
    aave = malloc(sizeof *aave);
    aave_init(aave);

    /* Select the HRTF set to use. */
    /* aave_hrtf_cipic(aave); */
    /* aave_hrtf_listen(aave); */
    aave_hrtf_mit(aave);
    /* aave_hrtf_tub(aave); */

    /* Set position and orientation of the listener. */
    aave_set_listener_position(aave, 0, 0, 0);
    aave_set_listener_orientation(aave, 0, 0, 0);

    /* Add the sound source to the auralisation world. */
```

```
    source = malloc(sizeof *source);
    aave_init_source(aave, source);
    aave_add_source(aave, source);

    /* Number of samples processed so far. */
    n = 0;

    /* Process one sample at a time until we do 10 full circles. */
    for (angle = 0; angle <= 10 * 2 * M_PI; angle += W / AAVE_FS) {

        /* Set the position of the source for this sample. */
        x = D * cos(angle);
        y = D * sin(angle);
        aave_set_source_position(source, x, y, 0);

        /* Update the geometry state of the auralisation engine. */
        aave_update(aave);

        /* Generate one sample of audio for the sound source. */
        in = sin(2 * M_PI * F / AAVE_FS * n++) * 32767;
        aave_put_audio(source, &in, 1);

        /* Collect the two samples of auralised binaural output. */
        aave_get_audio(aave, out, 1);

        /* Write the output to stdout in raw format. */
        if (fwrite(out, sizeof(out[0]), 2, stdout) != 2)
            return 1;
    }

    return 0;
}
```

7.2 examples/elevation.c

An example of auralisation of multiple sound source at different heights with the listener passing by.

```
/*
 * libaave/examples/elevation.c: multiple sound source heights
 *
 * Copyright 2013 Universidade de Aveiro
 *
 * Funded by FCT project AcousticAVE (PTDC/EEA-ELC/112137/2009)
 *
 * Written by Andre B. Oliveira <abo@ua.pt>
 */

/*
 * Usage: ./elevation sound1.raw sound2.raw ... > binaural.raw
 */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "aave.h"

/* The height distance between each sound source (m). */
#define H 10

/* The horizontal distance between listener and sound sources (m). */
#define D 2

/* The velocity that the listener is moving up (m/s). */
#define V 1.5

int main(int argc, char **argv)
{
    struct aave *aave;
    struct aave_source *sources;
    FILE **sounds;
    short in[1], out[2];
    unsigned i, k, n;

    /* Initialise the auralisation engine. */
    aave = malloc(sizeof *aave);
    aave_init(aave);

    /* Select the HRTF set to use. */
    /* aave_hrtf_cipic(aave); */
    /* aave_hrtf_listen(aave); */
    aave_hrtf_mit(aave);
    /* aave_hrtf_tub(aave); */
```

```
    /* Set the orientation of the listener. */
    aave_set_listener_orientation(aave, 0, 0, 0);

    /* Number of sounds specified in the arguments. */
    n = argc - 1;

    /* Initialise the sound sources. */
    sources = malloc(n * sizeof *sources);
    sounds = malloc(n * sizeof *sounds);
    for (i = 0; i < n; i++) {
        aave_init_source(aave, &sources[i]);
        aave_add_source(aave, &sources[i]);
        aave_set_source_position(&sources[i], D, 0, i * H);
        sounds[i] = fopen(argv[i+1], "rb");
        if (!sounds[i]) {
            fprintf(stderr, "error opening file %s\n", argv[i+1]);
            return 1;
        }
    }

    /* Process one sample at a time until we get to the highest source. */
    for (k = 0; k <= n * (AAVE_FS * H / (float)V); k++) {

        /* Update the position of the listener. */
        aave_set_listener_position(aave, 0, 0, k * (V/(float)
      AAVE_FS));

        /* Update the geometry state of the auralisation engine. */
        aave_update(aave);

        /*
         * Read one sample from each sound file and give it to
         * its sound source. If the file ends, give it a 0 sample.
         */
        for (i = 0; i < n; i++) {
            if (fread(in, sizeof(in[0]), 1, sounds[i]) != 1)
                in[0] = 0;
            aave_put_audio(&sources[i], in, 1);
        }

        /* Run the engine to get the corresponding binaural frame. */
        aave_get_audio(aave, out, 1);

        /* Write the binaural frame (2 samples) to stdout. */
        if (fwrite(out, sizeof(out[0]), 2, stdout) != 2)
            return 1;
    }

    return 0;
}
```

7.3 examples/line.c

An example of auralisation of a sound source moving on a straight line passing by the listener.

```
/*
 * libaave/examples/line.c: sound source moving on a straight line
 *
 * Copyright 2013 Universidade de Aveiro
 *
 * Funded by FCT project AcousticAVE (PTDC/EEA-ELC/112137/2009)
 *
 * Written by Andre B. Oliveira <abo@ua.pt>
 */

/*
 * Usage: ./line < mono.raw > binaural.raw
 */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "aave.h"

/* The velocity that the sound source is moving at (m/s). */
#define V (40 /* km/h */ * 1000 / 3600.)

/* The initial Y coordinate position of the sound source (m). */
#define Y0 -60

int main(int argc, char **argv)
```

```
{
    struct aave *aave;
    struct aave_source *source;
    short in[1], out[2];
    float y;

    /* Initialise the auralisation engine. */
    aave = malloc(sizeof *aave);
    aave_init(aave);

    /* Select the HRTF set to use. */
    /* aave_hrtf_cipic(aave); */
    /* aave_hrtf_listen(aave); */
    aave_hrtf_mit(aave);
    /* aave_hrtf_tub(aave); */

    /* Set the position and orientation of the listener. */
    aave_set_listener_position(aave, 0, 0, 0);
    aave_set_listener_orientation(aave, 0, 0, 0);

    /* Load a room model file, if specified on the arguments. */
    if (argc == 2)
        aave_read_obj(aave, argv[1]);

    /* Add a sound source to the auralisation world. */
    source = malloc(sizeof *source);
    aave_init_source(aave, source);
    aave_add_source(aave, source);

    /* Initial position of the sound source. */
    y = Y0;

    /* Read and process one sample at a time from the sound file. */
    while (fread(in, sizeof(in[0]), 1, stdin) == 1) {

        /* Set the position of the source for this sound sample. */
        aave_set_source_position(source, 2, y, 0);

        /* Update the geometry state of the auralisation engine. */
        aave_update(aave);

        /* Pass this sound sample to the auralization engine. */
        aave_put_audio(source, in, 1);

        /* Run the engine to get the corresponding binaural frame. */
        aave_get_audio(aave, out, 1);

        /* Write the binaural frame (2 samples) to stdout. */
        if (fwrite(out, sizeof(out[0]), 2, stdout) != 2)
            return 1;

        /* Update the position of the source for the next sample. */
        y += V / AAVE_FS;
    }

    return 0;
}
```

7.4 examples/stream.c

An example of auralisation of a streaming input sound, in real-time, with a user-specified room model and reflection order, using the Advanced Linux Sound Architecture.

```
/*
 * libaave/examples/stream.c: auralise a streaming input sound
 *
 * Copyright 2013 Universidade de Aveiro
 *
 * Funded by FCT project AcousticAVE (PTDC/EEA-ELC/112137/2009)
 *
 * Written by Andre B. Oliveira <abo@ua.pt>
 */

#include <stdio.h>
#include <alsa/asoundlib.h>
#include "aave.h"

/* Number of frames to capture/playback per loop. */
#define FRAMES 2048

int main(int argc, char **argv)
```

```
{
    struct aave *aave;
    struct aave_source *source;
    snd_pcm_t *capture, *playback;
    int n;
    short buffer[FRAMES * 2];

    if (argc != 3) {
        fprintf(stderr,
            "Usage: %s MODEL.OBJ REFLECTION_ORDER\n", argv[0]);
        return 1;
    }

    /* Initialise auralisation engine. */
    aave = malloc(sizeof *aave);
    aave_init(aave);

    /* Select the HRTF set to use. */
    /* aave_hrtf_cipic(aave); */
    /* aave_hrtf_listen(aave); */
    aave_hrtf_mit(aave);
    /* aave_hrtf_tub(aave); */

    /* Read the room model file. */
    aave_read_obj(aave, argv[1]);

    /* Set the highest order of reflections to generate. */
    aave->reflections = atoi(argv[2]);

    /* Set the position and orientation of the listener. */
    aave_set_listener_position(aave, 0, 0, 0);
    aave_set_listener_orientation(aave, 0, 0, 0);

    /* Add the sound source to the auralisation world. */
    source = malloc(sizeof *source);
    aave_init_source(aave, source);
    aave_add_source(aave, source);
    aave_set_source_position(source, 3, 0, 0);

    /* Perform the geometry calculations. */
    aave_update(aave);

    /* Open ALSA capture device. */
    n = snd_pcm_open(&capture, "default", SND_PCM_STREAM_CAPTURE, 0);
    if (n < 0) {
        fprintf(stderr, "snd_pcm_open: %s\n", snd_strerror(n));
        return 1;
    }
    n = snd_pcm_set_params(capture, SND_PCM_FORMAT_S16,
                SND_PCM_ACCESS_RW_INTERLEAVED,
                1, 44100, 0, 100000); /* 100ms latency */
    if (n < 0) {
        fprintf(stderr, "snd_pcm_set_params: %s\n", snd_strerror(n));
        return 1;
    }

    /* Open ALSA playback device. */
    n = snd_pcm_open(&playback, "hw", SND_PCM_STREAM_PLAYBACK, 0);
    if (n < 0) {
        fprintf(stderr, "snd_pcm_open: %s\n", snd_strerror(n));
        return 1;
    }
    n = snd_pcm_set_params(playback, SND_PCM_FORMAT_S16,
                SND_PCM_ACCESS_RW_INTERLEAVED,
                2, 44100, 0, 100000); /* 100ms latency */
    if (n < 0) {
        fprintf(stderr, "snd_pcm_set_params: %s\n", snd_strerror(n));
        return 1;
    }

    for (;;) {
        /* Read the mono frames from the capture device. */
        n = snd_pcm_readi(capture, buffer, FRAMES);
        if (n < 0) {
            fprintf(stderr, "snd_pcm_readi: %s\n",
                            snd_strerror(n));
            return 1;
        }

        /* Feed the mono frames to the sound source. */
        aave_put_audio(source, buffer, n);

        /* Get the binaural frames of the auralisation result. */
        aave_get_audio(aave, buffer, n);

        /* Write the binaural frames to the playback device. */
        n = snd_pcm_writei(playback, buffer, n);
```

```
        if (n < 0) {
            fprintf(stderr, "snd_pcm_writei: %s\n",
                 snd_strerror(n));
            return 1;
        }
    }

    return 0;
}
```

Index