Binaural Rendering Toolbox (BRT)

Technical documentation

Diana Research Group - University of Malaga

Table of contents

1. Welcome to Binaural Rendering Toolbox

The Binaural Rendering Toolbox (BRT) is a set of software libraries, applications, and definitions aimed as a virtual laboratory for psychoacoustic experimentation. The BRT is developed in the framework of the SONICOM project and include the algorithms developed in the 3D Tune-In Toolkit in a new open, extensible architecture.

1.1 Quick Start Guide

You can use BRT in three different ways depending on the level of integration which you need with your applications:

- If you are a C++ programmer building audio applications, you can use the BRT Library, as a C++ library integrated in your own audio application, which handles audio inputs and outputs.
- If you are using any other platform (Max MSP, PureData, Matlab, Unity, Python, etc.) and want to use BRT, you can use BeRTA Renderer controlled via OSC commands. BeRTA Renderer is a standalone application which manages audio interfaces and is able to read audio files. It is controllable via OSC commands and is able to perform all functionalities available in the BRT Library.
- If you just want to explore the main BRT features, or test or showcase some custom audio scene using a specific configuration of BRT modlels using BeRTA Renderer, you can use BeRTA GUI as a graphical interface to control BeRTA Renderer.

1.2 Components

- BRT Library. Core component that implements the algorithms for binaural rendering, providing a flexible and extensible framework for simulating complex auditory environments in psychoacoustic research.
- BRT Applications.
- BeRTA Renderer. Audio renderer that integrates the BRT library.
- BeRTA GUI. Graphical interface to contro BeRTA Renderer with OSC messages.
- OSC commands. Full definition of commands to control BeRTA Renderer.
- SOFA files. BRT extensively uses files in SOFA format (AES69-2022) to manage HRTFs, BRIRs, source directivities, Headphone compensation and other binaural filters, near field compensation filters and annotated audio.

1.3 Credits

This software is being developed by a team coordinated by

- Arcadio Reyes-Lecuona (Diana Research Group, University of Malaga). Contact: areyes@uma.es
- Lorenzo Picinali (Audio Experience Design Team, Imperial College London). Contact: l.picinali@imperial.ac.uk

The current members of the core development team are (in alphabetical order):

- Daniel Gonzalez-Toledo (University of Malaga)
- Maria Cuevas-Rodriguez (University of Malaga)
- Luis Molina-Tanco (University of Malaga)

Other contributors:

- Francisco Morales Benítez (Former contributor at the University of Malaga mainly in BeRTA GUI and BeRTA renderer)
- Marco Fontana (Developer of SDN Environment Model at the Laboratorio di Informatica Musicale, Università degli Studi di Milano)

1.4 Copyright and License

This software is distributed under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Copyright (c) for each module belongs to its respective authors, who may vary as there are contributions from different institutions in this software. This copyright information is specified in the headers of the corresponding files

This library includes pieces of code from the 3DTI AudioToolkit, shared under GPLv3 license and copyright (c) by University of Málaga (contact: areyes@uma.es) and Imperial College London (contact: l.picinali@imperial.ac.uk). See headers in the source code files.

1.5 Acknowledgements

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement no.101017743

2. BRT Library

2.1 BRT Library: General Overview

The **Binaural Rendering Toolbox (BRT)** is an advanced C++ library designed to support psychoacoustic research and the development of immersive audio applications. As a core component of the BRT Toolbox, the library provides a modular, open-architecture framework that facilitates the creation of dynamic and reproducible virtual acoustic scenarios. This section introduces the main features, purpose, and usage of the BRT Library.

2.1.1 Introduction

The BRT Library was developed as part of the SONICOM project to address the increasing need for flexible, extensible, and reproducible tools in psychoacoustic research. It builds upon the experience gained from the 3D Tune-In Toolkit while introducing significant architectural improvements, including modularity and ease of extension. The library is intended for researchers and developers aiming to simulate complex binaural scenarios, such as real-time dynamic environments with multiple listeners and sound sources.

2.1.2 Key Features

1. **Modularity and Extensibility**:

- The library adopts a header-only structure, simplifying integration into various projects.
- It is organized into three layers:
- **Top Layer**: This layer comprises high-level modules, referred to as "models," which include listener, source, and environment models, as well as binaural filters. These modules are interconnected to create diverse acoustic scenarios. *Developers using the BRT Library* to build audio applications will primarily interact with this layer, as it provides a straightforward interface for simulating complex binaural environments.
- **Middle Layer**: This layer contains a collection of signal processing and service modules that serve as the foundation for the high-level models. Signal processing modules include components like convolvers and filters, while service modules manage data such as impulse responses. *Developers contributing new algorithms or models* to the library will need to understand this layer, as it provides the building blocks for extending the library's functionality.
- **Bottom Layer**: At the core of the library, this layer consists of foundational classes and templates that define the modular architecture. It implements mechanisms for interconnecting modules and ensures the flexibility and scalability of the library. This layer is primarily relevant for developers seeking to modify or enhance the library's architecture itself.

2. **Dynamic Virtual Acoustics**:

- Supports anechoic and reverberant environments.
- Enables simulation of near-field effects and dynamic HRTF swapping.
- Supports 6DoF for both Listeners and sources

3. **Applications and Portability**:

- The library, written in standard C++, is designed for deployment on various platforms, including standalone executables, mobile, and wearable devices. Its modular and multilevel structure enables flexible configurations, making deployment adaptable to diverse systems.

2.1.3 Purpose and Applications

The BRT Library provides a flexible and extensible framework tailored to the needs of researchers and developers in psychoacoustics, auditory modeling, and immersive audio rendering. Its design supports a variety of use cases, including:

- **Custom Binaural Rendering Pipelines**: Configure and implement tailored audio rendering workflows for specific experimental or application needs.
- **Dynamic Auditory Modeling**: Test and validate auditory models with dynamic scenarios, supporting 6DoF of both listeners and sources.
- **Rapid Testing of Simulated Characteristics**: Use integrated applications for quick testing of BRIRs, HRTFs, and other auditory properties.
- **Full Process Chain Characterization**: Characterize complete models, including source, environment, and listener chains, as directional impulse responses or log binaural stimuli along with the position and orientation data used for synthesis.

Developer-Friendly Design

The library minimizes the complexity of implementing novel algorithms and adapting to unique research requirements. Its modular structure offers well-defined internal interfaces and reusable building blocks, enabling efficient customization. Developers can:

- Use **predefined modules** to simulate specific acoustic scenarios, such as free-field or room acoustics.
- Extend the library by creating custom **low-level processing** or **high-level modeling** modules to support new features or research needs.

Active Development and Future Capabilities

The BRT Library is actively maintained and continuously evolving. Planned enhancements include:

- Support for **multi-listener scenarios**, enabling collaborative and interactive auditory experiments.
- Integration of **advanced reverberation models**, such as hybrid methods combining Image Source Methods (ISM) with other techniques for the reverberation tail.
- Development of **hearing loss and hearing aid simulations**, broadening the scope of auditory applications.

2.1.4 Description

The library is organized into three layers, as previously discussed. From the perspective of a C++ programmer[1], using the library mainly involves working with high-level and service modules.

High-Level Modules

High-level modules are responsible for audio rendering. Each module models specific physical and/or psychoacoustic phenomena, depending on the use case. The following types of models have been implemented, grouped into four categories:

- **Source Models**: Each monaural sound source to be rendered requires the instantiation of a source model. These models serve as the main entry points to the library for applications during rendering. At a minimum, applications must provide the audio samples of each source for every frame and, if applicable, update their position and orientation.
- Omnidirectional Source Model
- Directional Source Model
- **Listener Models**: Listener models generate binaural audio by receiving a single audio channel per source and producing two output channels (left and right ears). They can actually include Listener+Environment models using Binaural Room Impulse Responses (BRIR). They are categorized as follows:
- HRTF Convolution Models: These models simulate sound perception using Head-Related Transfer Functions (HRTF). Signals from sound sources are convolved with binaural impulse responses stored in HRTF Service modules. Whose data is usually read from a SOFA file. Currently two such models have been implemented: **Direct HRTF Convolution Model** and **Ambisonic BRIR Convolution Model**.
- BRIR Convolution Models: These models use Binaural Room Impulse Responses (BRIR) to simulate acoustic spaces, convolving source signals with room-specific responses stored in HRBRIR service modules. The RIR data will typically be read from a SOFA file and may contain impulse responses with the transmitter and receiver located at various locations in the room. Currently two models have been implemented: **Direct BRIR Convolution Model** and **Ambisonic BRIR Convolution Model**.
- **Environment Models**: These models simulate various acoustic environments:
- Free Field: Simulates propagation in a free-field environment, including propagation delay, attenuation, and filtering.
- SDN: Simulates room reverberation using the Scattering Delay Networks method.
- *ISM*: Simulates room reverberation using the Image Source Method *(under development)*.
- *Hybrid: ISM + Convolution*: Simulates room reverberation where early reflections are modeled using the Image Source Method, and the reverberant tail is simulated through convolution with a BRIR *(under development)*.
- **Bilateral Filters**: They perform filtering on binaural signals.
- SOS Filters: Generic module to perform binaural filtering based on second-order sections, enabling the simulation of devices such as ear protection devices of headphone compensations.
- *Non-linear filters: support advanced models such as **hearing loss simulation** (under development).*
- *FIR filters: support applications such as **headphone compensation** (under development).*
- **Listener**: Each listener instantiated in the BRT library represents a “real” listener for which you want to render binaural audio. The application must keep its position/orientation updated and must at the end of each audio frame collect the output samples.

Service Modules

Service modules store the essential data required for rendering. This data typically comes from SOFA files, although other formats could be used. Key service modules include:

- HRTF: Stores head-related impulse responses indexed by azimuth and elevation.
- BRIR: Stores room-related impulse responses indexed by azimuth and elevation.
- DirectivityTF: Stores transfer functions of a sound source based on the position of the listener and the sources.
- SOSFilters: Stores coefficients for second-order sections of a filter, which can be fixed or vary based on distance, azimuth, and/or elevation.
- AmbisonicBIR: Stores the impulse responses of the virtual loudspeakers in the ambisonic domains, in order to achieve a process with simultaneous impulse responses convolution and ambisonic decoding.

2.1.5 Usage

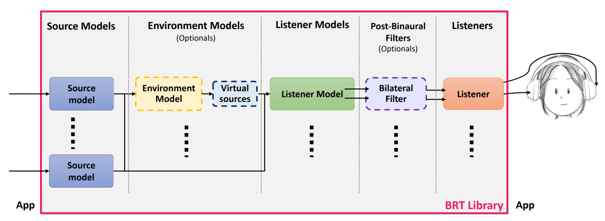
To render audio using the library, the required modules must be instantiated and interconnected based on the desired simulation (configuration). This is managed through the `brtManager` class [URL]. For detailed instructions, refer to the **Setup/Examples** section.

General Workflow

The diagram below outlines the modular interconnections in the BRT Library and the typical workflow for rendering audio:

1. **Source Models**: Instantiate a source model for each monaural sound source. The application is responsible for feeding these models with audio samples for each frame and updating their position and orientation as needed.
2. **Listener Models**: Connect source models to listener models to generate binaural audio. If an environment simulation is required, source models should first be connected to environment models.
3. **Environment Models (Optional)**: These models simulate acoustic spaces and produce virtual sources, which are then connected to the listener models. Applications only need to connect the modules; the library handles virtual source generation.
4. **Bilateral Filters (Optional)**: To simulate auditory devices, such as headphones, connect a bilateral filter between listener models and listener outputs.
5. **Listeners**: Instantiate one listener module for each real listener. The application must collect the output samples from each listener at the end of every audio frame.

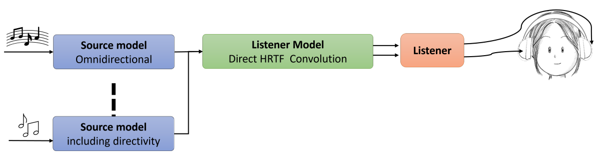
This modular approach provides flexibility while ensuring that all components interact seamlessly. The application is tasked with managing input and output data flow, while the library takes care of the processing.



Generic connection diagram of the BRT models

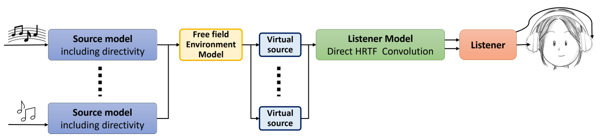
Below are examples of configurations that can be created:

Example 1 - Basic Anechoic Simulation: Combine a listener model with source models for anechoic rendering.



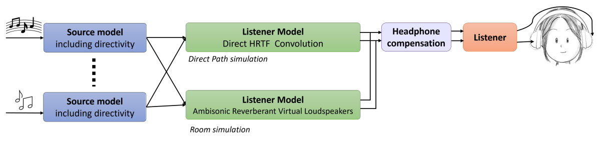
Configuration for direct path simulation (anechoic simulation)

Example 2 - Propagation simulation: Add an environment model to simulate propagation.



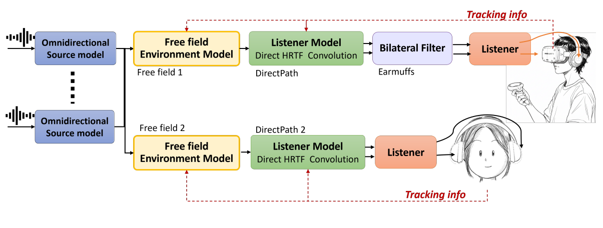
Configuration for direct path simulation adding propagation in a free environment (anechoic simulation + propagation)

Example 3 - Hybrid simulation and custom filters: Independent simulation for direct sound and room reverberation, using a hybrid approach. In addition, a bilateral filter is added to simulate hearing protectors or to compensate for headphones.



Configuration with direct path (anechoic) and reverberation (room) simulation, plus filtering.

Example 4 - Simultaneous rendering of two listeners: Simultaneous and independent rendering of two listeners. Each can have their own position in space, HRTF, etc. In addition, filtering is added to one of the listeners to simulate ear protective devices.



Simultaneous rendering of two listeners

For further technical details and code examples, visit the official repository: BRT Library GitHub.

1. If you prefer a higher-level approach, you can use our applications and control them via OSC commands. For more details, see Applications. ↩

2.2 Source Models

2.2.1 Source Models

Each monaural sound source to be rendered requires the instantiation of a source model. Each sound source must be connected to a source model that can transform the sound depending on the source position (location and orientation). These models therefore serve as the main library entry points for applications during rendering. At a minimum, applications must provide the audio samples of each source for each frame and, if applicable, update their position and orientation.

There are currently two fundamental source models and one virtual source model:

- Omnidirectional Model
- Directivity Model
- Virtual Sources Model

2.2.2 Omnidirectional Source Model

The **Omnidirectional Source Model** serves as a straightforward interface for applications to interact with the BRT Library during the rendering process. This model is characterized by its simplicity, as it primarily passes the source's position and the audio samples provided by the application to the connected modules. These modules can include listener models and environment models. Applications must supply monaural audio samples for each frame and update the source's position and orientation when necessary.

Functional Overview

To use the omnidirectional source model, it must first be instantiated and connected to the desired modules, such as listener or environment models. Once connected, the application is responsible for managing the source's position and orientation, which can be updated dynamically as needed using the method `` `SetSourceTransform ``. During each audio frame, the application must also provide the corresponding monaural audio samples, usind the `SetBuffer` method.

Connections

Modules to which it connects:

```
- Environment models
- Listener models
```

r C++ developer

- **File**: /include/SourceModels/SourceOmnidirectionalModel.hpp
- **Class name**: CSourceOmnidirectionalModel
- **Inheritance**: CSourceModelBase
- **Namespace**: BRTSourceModel

Class inheritance diagram

Omnidirectional Source Model Class diagram

Omnidirectional Source Model Class diagram.

How to instantiate

```cpp
// Assuming that the ID of this environment model is contained in
_environmentID.
brtManager.BeginSetup();
std::shared_ptr<BRTSourceModel::CSourceOmnidirectionalModel>
soundSource = brtManager-
>CreateSoundSource<BRTSourceModel::CSourceOmnidirectionalModel>(soundSour
brtManager.EndSetup();
if (soundSource == nullptr) {
    // error
}
```

How to connect

Connect it to a listener model.

```cpp
// Assuming that the soundSource could be a ID(string) or a
std::shared_ptr<BRTSourceModel::CSourceModelBase>;
std::shared_ptr<BRTListenerModel::CListenerModelBase> listenerModel =
brtManager-
>GetListenerModel<BRTListenerModel::CListenerModelBase>(_listenerModelID)
if (listenerModel != nullptr) {
    bool control = listenerModel->ConnectSoundSource(soundSource);
}
```

Connect it to a environment model.

```cpp
// Assuming that the ID of this source model is contained in _sourceID
and
// that the ID of this environment is contained in
_environmentModelID.
std::shared_ptr<BRTEnvironmentModel::CEnviromentModelBase>
environmentModel = brtManager-
>GetEnvironmentModel<BRTEnvironmentModel::CEnviromentModelBase>(_environm
if (environmentModel != nullptr) {
    bool control = environmentModel->ConnectSoundSource(_sourceID);
}
```

Public methods

```cpp
void SetBuffer(const CMonoBuffer<float>& _buffer)
CMonoBuffer<float> GetBuffer()

void SetSourceTransform(Common::CTransform _transform)
const Common::CTransform& GetSourceTransform() const

std::string GetID()

TSourceType GetSourceType()
```

2.2.3 Directivity Source Model

The **Directivity Source Model** provides an interface for applications to interact with the BRT Library while implementing frequency-dependent directivity for sound sources. This model takes into account the direction of the listener relative to the source and applies a corresponding filter to transform the audio signal as it would be radiated in that direction. Once processed, the model transmits the filtered audio samples to the connected modules, such as listener or environment models, while also providing real-time updates on the source's position and orientation. Applications must supply monaural audio samples for each frame and dynamically update the source's position and orientation as needed.

Functional Overview

After instantiating the Directivity Source Model and connecting it to the desired modules (listener and/or environment models), the application manages the source’s position and orientation using the `SetSourceTransform` method. This allows for dynamic updates to the source's state. For each audio frame, the application must also provide the corresponding monaural audio samples via the `SetBuffer` method.

When the model receives the audio samples, it performs a convolution with the directivity data to account for the directional characteristics of the source. This data is supplied by the DirectivityTF service module, which provides frequency responses based on the source's orientation and the relative position between the source and the listener. The convolutions are executed in the frequency domain using the uniformly partitioned convolution algorithm, ensuring efficient and accurate processing.

Architecture

The internal block diagram of this class is as follows:

Directivity Sound Source Model Internal diagram

Directivity Sound Source Model Internal diagram.

Configuration Options

This model allows configuration by calling its methods or by BRT internal commands:

- **Directivity (on/off)**: Omnidirectional sound source when off.
- **DirectivityTF to be used**: The DirectivityTF service module to be used for rendering. The system supports dynamic, hot-swapping of the service module being used.

Connections

Modules to which it connects:

```
- Environment models
- Listener models
```

r C++ developer

- **File**: /include/SourceModels/SourceDirectivityModel.hpp
- **Class name**: CSourceDirectivityModel
- **Inheritance**: CSourceModelBase
- **Namespace**: BRTSourceModel

Class inheritance diagram

Directivity Source Model Class diagram

Directivity Source Model Class diagram.

How to instantiate

```
// Assuming that the ID of this environment model is contained in
_environmentID.
brtManager.BeginSetup();
std::shared_ptr<BRTSourceModel::CSourceDirectivityModel> soundSource
= brtManager-
>CreateSoundSource<BRTSourceModel::CSourceDirectivityModel>(soundSourceID
brtManager.EndSetup();
if (soundSource == nullptr) {
    // error
}
```

How to connect

Connect it to a listener model.

```
// Assuming that the soundSource could be a ID(string) or a
std::shared_ptr<BRTSourceModel::CSourceModelBase>;
std::shared_ptr<BRTListenerModel::CListenerModelBase> listenerModel =
brtManager-
>GetListenerModel<BRTListenerModel::CListenerModelBase>(_listenerModelID)
if (listenerModel != nullptr) {
    bool control = listenerModel->ConnectSoundSource(soundSource);
}
```

Connect it to a environment model.

```
// Assuming that the ID of this source model is contained in _sourceID
and
// that the ID of this environment is contained in
_environmentModelID.
std::shared_ptr<BRTEnvironmentModel::CEnviromentModelBase>
environmentModel = brtManager-
>GetEnvironmentModel<BRTEnvironmentModel::CEnviromentModelBase>(_environm
if (environmentModel != nullptr) {
    bool control = environmentModel->ConnectSoundSource(_sourceID);
}
```

Public methods

```
void SetBuffer(const CMonoBuffer<float>& _buffer)
CMonoBuffer<float> GetBuffer()

void SetSourceTransform(Common::CTransform _transform)
const Common::CTransform& GetSourceTransform() const

std::string GetID()

TSourceType GetSourceType()

bool SetDirectivityTF(std::shared_ptr< BRTServices::CDirectivityTF >
_sourceDirectivityTF) override
std::shared_ptr<BRTServices::CDirectivityTF> GetDirectivityTF()
override
void RemoveDirectivityTF() override

void SetDirectivityEnable(bool _enabled) override
```

2.2.4 Virtual Source Model

The **Virtual Source Model** is an internal component of the BRT Library, used primarily by certain environment models to simulate complex acoustic scenarios. For each original source, the model generates multiple virtual sources that may differ in position, orientation, and audio samples.

These virtual sources act as independent omnidirectional sources, enabling accurate representation of reflections, reverberation, and other spatial effects.

This model is not intended for direct use by applications but instead functions as a foundational tool for the library's environment models to perform advanced acoustic simulations.

2.3 Listener Models

2.3.1 Listener Acoustic Models

Listener models are high-level modules of the BRT library that handle audio rendering. More specifically these modules try to simulate the binaural sound perceived by the subjects. These models can simulate only the direct sound, or only the reverberation, or all of them together. Currently, four models have been implemented, grouped into two categories:

Listener Models based on HRTFs

The BRT library includes listening models that use head impulse responses (HRTF) to simulate the sound perceived by a user in a realistic way. These models work by convolving the signals from the sound sources with the binaural head impulse response (HRTF). This HRTF will typically be read from a SOFA file. Currently two such models have been implemented:

- Based on direct convolution with HRTF: Simulates the direct path using convolution with HRTFs.
- Based on convolution with HRTF in the Ambisonics domains: Simulates the direct path using convolution with HRTFs in the ambisonic domain.

Listener & Environment Models based on BRIRs

The BRT library includes environment models that leverage room impulse responses (RIRs) to simulate realistic acoustic spaces. These models function by convolving the signals from sound sources with the binaural room impulse response (BRIR) of the room. The RIR will be read from a SOFA file and can contain impulse responses with both emmitter and receiver located in several locations in the room. Two models are currently implemented:

- Based on direct convolution with BRIR: Simulates both the direct path and reverberation through convolution with a BRIR.
- Based on convolution with BRIR in the Ambisonics domain: Simulates the direct path and reverberation using convolution with a BRIR in the ambisonic domain.

Others

- *Listener Auditory Model of Hearing Loss*: *(Under development)*.
- *Listener Model of Hearing Aid*: *(Under development)*.

2.3.2 HRTF Models

Listener Models Based on HRTFs

The BRT Library includes listener models that utilize **Head-Related Transfer Functions (HRTFs)** to realistically simulate the sound perceived by a listener. An HRTF is a mathematical representation of how an individual's head, ears, and torso affect the sound arriving from a specific direction before reaching the eardrums. This representation captures how the human auditory system localizes sound in a 3D space, including cues such as interaural time differences (ITD) and interaural level differences (ILD).

By convolving the signals from the sound sources with the binaural HRTFs, these models replicate the way sound interacts with the anatomy of the listener, offering a highly realistic auditory experience. This convolution process enables the simulation of the **direct sound path**, meaning the sound traveling directly from the source to the listener without reflections or reverberation.

Data Source for HRTFs

These HRTF rendering models require HRTF data to be preloaded and preprocessed in the HRTF service module. For more details, see the documentation on the HRTF service module.

The HRTFs used by these models are typically read from files formatted in the SOFA (Spatially Oriented Format for Acoustics). This standard is widely adopted in the audio research community for storing spatially oriented acoustic data, including head-related impulse responses.

The BRT Library provides an integrated HRTF loader, capable of reading SOFA files and extracting the necessary data for simulation. For more details, see the documentation on the Readers.

Implemented Listener Models

Currently, the following listener models based on HRTFs have been implemented:

- Direct HRTF Convolution Model: Simulates the direct sound path using convolution with HRTFs.
- HRTF Convolution model in the Ambisonics Domain: Simulates the direct sound path using convolution with HRTFs in the ambisonic domain.

These models are designed to provide flexibility and precision for a variety of psychoacoustic experiments and immersive audio applications.

Listener Model based on HRTF direct convolution

The **Listener Direct HRTF Convolution Model** module of the library allows the rendering of spatial audio from multiple sound sources. It simulates the direct path between source and listener by taking advantage of *direct convolution* with head impulse responses (HRTF). This module independently processes each of the input sources, taking into account their positions and the position and orientation of the listener.

Subsequently, this model performs a *near-field correction*, independent for each of the sources. This correction is made by filtering the signals corresponding to each ear. The coefficients of these filters are dependent on the distance between source and listener and the interaural azimuth.

Finally, the model mixes all channels, left and right separately, to give a single output per ear.

The convolutions are performed in the frequency domain using the uniformly partitioned convolution algorithm. Filtering is performed in the time domain. The HRTF service module is responsible for providing the impulse responses in each frame. While the SOS filter module is in charge of providing the filters coefficients. Both classes must have been previously configured, read more in the services modules section.

ARCHITECTURE

The internal block diagram of this class is as follows:

Listener Direct HRTF Convolution Model Internal diagram

Listener Direct HRTF Convolution Model Internal Block diagram.

CONFIGURATION OPTIONS

This model allows configuration by calling its methods or by BRT internal commands:

- **Model (on/off)**: Silent when off.
- **Gain (float)**: Extra gain to be applied to the model output.
- **Spatialization (on/off)**: Transparent when off.
- **Interpolation (on/off)**: When switched on, HRIRs and delays are calculated at the exact position (relative source-listener position). For this purpose, barycentric interpolation is performed, starting from the three closest points. When it is switched off, the HRIR and delay with the closest position are chosen.
- **Near Field Compensation (on/off)**: The near field correction is applied when on.
- **ITD Simulation (on/off)**: When activated, a separate delay is added to each ear to simulate the interaural time difference[1]. This delay is provided by the HRTF service module and may be provided in the SOFA structure or will be calculated from the head size, for more information see HRTF Service Module. When off, it does not simulate interaural time difference.
- **Parallax Correction (on/off)**: When it is switched on, a cross-ear parallax correction is apllied. This correction is based on calculating the projection of the vector from the ear to the source on the HRTF sphere (i.e. the sphere on the surface of which the HRTF was measured), giving a more accurate rendering, especially for near-field and far-field sound sources. When deactivated, the calculation is based on the centre of the listener's head.
- **HRTF to be used**: The HRTF service module to be used for rendering. The system supports dynamic, hot-swapping of the service module being used.
- **Nearfield filter (SOS filter) to be used**: The SOS filter service module to be used for rendering. The system supports dynamic, hot-swapping of the service module being used.

CONNECTIONS

Modules to which it supports connections:

```
- Source models
- Environment models
```

Modules to which it connects:

```
- Listener
- Billateral Filter
```

r C++ developer

- **File**: /include/ListenerModels/ListenerDirectHRTFConvolution.hpp
- **Class name**: CListenerDirectHRTFConvolutionModel
- **Inheritance**: CListenerModelBase
- **Namespace**: BRTListenerModel
- **Classes that instance**:
- BRTProcessing::CHRTFConvolverProcessor
- BRTProcessing::CNearFieldEffectProcessor

CLASS INHERITANCE DIAGRAM

Listener HRTF Model Internal diagram

Listener HRTF Model Internal diagram.

HOW TO INSTANTIATE

```
// Assuming that the ID of this listener model is contained in
_listenerModelID.
brtManager.BeginSetup();
std::shared_ptr<BRTListenerModel::CListenerDirectHRTFConvolutionModel>lis
=
brtManager.CreateListenerModel<BRTListenerModel::CListenerDirectHRTFConvo
brtManager.EndSetup();
if (listenerModel == nullptr) {
    // ERROR
}
```

HOW TO CONNECT

Connect it to a listener.

```
// Assuming that the ID of this listener is contained in _listenerID
and
// that the ID of this listener model is contained in
_listenerModelID.
std::shared_ptr<BRTBase::CListener> listener =
brtManager.GetListener(_listenerID);
if (listener != nullptr) {
    brtManager.BeginSetup();
    bool control = listener->ConnectListenerModel(_listenerModelID);
    brtManager.EndSetup();
}
```

Connect an environment model to it.

```
// Assuming that the ID of this listener model is contained in
_listenerModelID.
// that the ID of this environment is contained in
_environmentModelID.
std::shared_ptr<BRTListenerModel::CListenerModelBase> listenerModel =
brtManager.GetListenerModel<BRTListenerModel::CListenerModelBase>(_listen
if (listenerModel != nullptr) {
    brtManager.BeginSetup();
    bool control = listenerModel-
>ConnectEnvironmentModel(_environmentModelID);
    brtManager.EndSetup();
}
```

Connect a source model to it.

```
// Assuming that the soundSource could be a ID(string) or a
std::shared_ptr<BRTSourceModel::CSourceModelBase>;
std::shared_ptr<BRTListenerModel::CListenerModelBase> listenerModel =
brtManager-
>GetListenerModel<BRTListenerModel::CListenerModelBase>(_listenerModelID)
if (listenerModel != nullptr) {
    bool control = listenerModel->ConnectSoundSource(soundSource);
}
```

PUBLIC METHODS

```
void EnableModel() override
void DisableModel() override
void EnableSpatialization() override
void DisableSpatialization() override
bool IsSpatializationEnabled() override

void EnableInterpolation() override
void DisableInterpolation() override
bool IsInterpolationEnabled() override

void EnableNearFieldEffect() override
void DisableNearFieldEffect() override
bool IsNearFieldEffectEnabled() override

void EnableITDSimulation() override
void DisableITDSimulation() override
bool IsITDSimulationEnabled() override

void EnableParallaxCorrection() override
void DisableParallaxCorrection() override
bool IsParallaxCorrectionEnabled() override

bool SetHRTF(std::shared_ptr< BRTServices::CHRTF > _listenerHRTF)
override
std::shared_ptr<BRTServices::CHRTF> GetHRTF() const override
void RemoveHRTF() override

bool
SetNearFieldCompensationFilters(std::shared_ptr<BRTServices::CSOSFilters>
_listenerILD) override
std::shared_ptr<BRTServices::CSOSFilters>
GetNearFieldCompensationFilters() const override
void RemoveNearFierldCompensationFilters() override

bool
ConnectSoundSource(std::shared_ptr<BRTSourceModel::CSourceModelBase>
_source) override
bool ConnectSoundSource(const std::string & _sourceID) override
bool
DisconnectSoundSource(std::shared_ptr<BRTSourceModel::CSourceModelBase>
_source) override
bool DisconnectSoundSource(const std::string & _sourceID) override

bool ConnectEnvironmentModel(const std::string & _environmentModelID)
override
bool DisconnectEnvironmentModel(const std::string &
_environmentModelID) override

void ResetProcessorBuffers()
void UpdateCommand() override
```

1. To perform the convolution task correctly, avoiding comb filters, it is necessary that the delays of the impulse responses have been removed, for more information see HRTF Service Module. ↩

Listener Model based on HRTF convolution in the Ambisonics domain

The **Listener Ambisonic Virtual Loudspeakers Model** module enables spatial audio rendering from multiple sound sources by simulating the direct path between the source and the listener through convolution in the ambisonic domain. This is based on the impulse responses stored in an HRTF and the selected ambisonic order (currently implemented up to order 3). The process involves two main stages: a bilateral ambisonic encoding and an ambisonic convolution/decoding.

In the first stage, a bilateral ambisonic encoding is performed for each input sound source. This encoding generates (N) ambisonic channels per ear, where (N) depends on the selected ambisonic order. The encoding process begins by introducing independent delays for each ear to simulate interaural time differences (ITD). Subsequently, a near-field correction is applied through independent filtering of the signals for each ear, with filter coefficients determined by the distance between the source and the listener as well as the interaural azimuth. Finally, ambisonic encoding is carried out separately for each ear, resulting in (N) ambisonic channels for each ear per sound source.

In the second stage, two tasks are performed simultaneously: convolution with impulse responses and ambisonic decoding (for more details, see the AmbisonicBIR section). There are two convolution/decoding blocks, one for each ear, allowing independent processing for the left and right channels. Each block begins by separately mixing the ambisonic channels generated during the encoding stage. It then performs convolution in the frequency domain for each channel using precomputed impulse responses stored in the AmbisonicBIR service module. These impulse responses represent the ambisonic mixture of the virtual loudspeaker responses, enabling both convolution and ambisonic decoding to be executed simultaneously. Finally, the output is mixed and transformed back into the time domain, resulting in the final signal for the corresponding ear.

This modular approach ensures efficient and precise spatial audio rendering. The linearity of the operations is leveraged to reduce computational overhead by minimizing the number of convolutions required. The result is a highly realistic simulation of the direct sound path in the ambisonic domain, offering support for scalable ambisonic orders up to the third order.

For further details on the functionality of the Bilateral Ambisonic Encoder and the Ambisonic Domain Convolver, refer to their respective sections in the documentation.

ARCHITECTURE

The internal block diagram of this class is as follows:

Listener Ambisonic Virtual LoudSpeakers Model - Internal diagram

Listener Ambisonic Virtual LoudSpeakers Model - Internal diagram

CONFIGURATION OPTIONS

This model allows configuration by calling its methods or by BRT internal commands:

- **Model (on/off)**: Silent when off.
- **Near Field Compensation (on/off)**: The near field correction is applied when on.
- **ITD Simulation (on/off)**: When activated, a separate delay is added to each ear to simulate the interaural time difference[1]. This delay is provided by the HRTF service module and may be provided in the SOFA structure or will be calculated from the head size. When off, it does not simulate interaural time difference.
- **Parallax Correction (on/off)**: When it is switched on, a cross-ear parallax correction is apllied. This correction is based on calculating the projection of the vector from the ear to the source on the HRTF sphere (i.e. the sphere on the surface of which the HRTF was measured), giving a more accurate rendering, especially for near-field and far-field sound sources. When deactivated, the calculation is based on the centre of the listener's head.
- **HRTF to be used**: The HRTF service module to be used for rendering. The system supports dynamic, hot-swapping of the service module being used.
- **Nearfield filter (SOS filter) to be used**: The SOS filter service module to be used for rendering. The system supports dynamic, hot-swapping of the service module being used.
- **Ambisonic Order**: The order of the ambisonic coding to be used. Currently only orders between 1 (default) and 3 are valid.
- **Ambisonic Normalization**: The ambisonic normalization to be used. The available options are: N3D (default), SN3D, maxN

CONNECTIONS

Modules to which it supports connections:

```
- Source models
- Environment models
```

Modules to which it connects:

```
- Listener
- Bilateral Filter
```

r C++ developer

- **File**: \include\ListenerModels\ListenerAmbisonicVirtualLoudspeakersModel.hpp
- **Class name**: CListenerAmbisonicVirtualLoudspeakersModel
- **Inheritance**: CListenerModelBase
- **Namespace**: BRTListenerModel
- **Classes that instance**:
- BRTProcessing::CAmbisonicDomainConvolverProcessor
- BRTProcessing::CBilateralAmbisonicEncoderProcessor

CLASS INHERITANCE DIAGRAM

Listener Ambisonic Virtual LoudSpeakers class diagram

Listener Ambisonic Virtual LoudSpeakers class diagram

HOW TO INSTANTIATE

```
// Assuming that the ID of this listener model is contained in
_listenerModelID.
brtManager.BeginSetup();
std::shared_ptr<BRTListenerModel::CListenerAmbisonicVirtualLoudspeakersMo
=
brtManager.CreateListenerModel<BRTListenerModel::CListenerAmbisonicVirtua
brtManager.EndSetup();
if (listenerModel == nullptr) {
    // ERROR
}
```

HOW TO CONNECT

Connect it to a listener.

```
// Assuming that the ID of this listener is contained in _listenerID
and
// that the ID of this listener model is contained in
_listenerModelID.
std::shared_ptr<BRTBase::CListener> listener =
brtManager.GetListener(_listenerID);
if (listener != nullptr) {
    brtManager.BeginSetup();
    bool control = listener->ConnectListenerModel(_listenerModelID);
    brtManager.EndSetup();
}
```

Connect an environment model to it.

```
// Assuming that the ID of this listener model is contained in
_listenerModelID.
// that the ID of this environment is contained in
_environmentModelID.
std::shared_ptr<BRTListenerModel::CListenerModelBase> listenerModel =
brtManager.GetListenerModel<BRTListenerModel::CListenerModelBase>(_listen
if (listenerModel != nullptr) {
    brtManager.BeginSetup();
    bool control = listenerModel-
>ConnectEnvironmentModel(_environmentModelID);
    brtManager.EndSetup();
}
```

Connect a source model to it.

```
// Assuming that the soundSource could be a ID(string) or a
std::shared_ptr<BRTSourceModel::CSourceModelBase>;
std::shared_ptr<BRTListenerModel::CListenerModelBase> listenerModel =
brtManager-
>GetListenerModel<BRTListenerModel::CListenerModelBase>(_listenerModelID)
if (listenerModel != nullptr) {
    bool control = listenerModel->ConnectSoundSource(soundSource);
}
```

PUBLIC METHODS

```
void EnableModel() override
void DisableModel() override

void EnableNearFieldEffect() override
void DisableNearFieldEffect() override
bool IsNearFieldEffectEnabled() override

void EnableITDSimulation() override
void DisableITDSimulation() override
bool IsITDSimulationEnabled() override

void EnableParallaxCorrection() override
void DisableParallaxCorrection() override
bool IsParallaxCorrectionEnabled() override

bool SetAmbisonicOrder(int _ambisonicOrder) override
int GetAmbisonicOrder() override

bool SetAmbisonicNormalization(Common::TAmbisonicNormalization
_ambisonicNormalization) override
bool SetAmbisonicNormalization(std::string _ambisonicNormalization)
override
Common::TAmbisonicNormalization GetAmbisonicNormalization() override

bool SetHRTF(std::shared_ptr< BRTServices::CHRTF > _listenerHRTF)
override
std::shared_ptr<BRTServices::CHRTF> GetHRTF() const override
void RemoveHRTF() override

bool
SetNearFieldCompensationFilters(std::shared_ptr<BRTServices::CSOSFilters>
_listenerILD) override
std::shared_ptr<BRTServices::CSOSFilters>
GetNearFieldCompensationFilters() const override
void RemoveNearFierldCompensationFilters() override

bool
ConnectSoundSource(std::shared_ptr<BRTSourceModel::CSourceModelBase>
_source) override
bool ConnectSoundSource(const std::string & _sourceID) override
bool
DisconnectSoundSource(std::shared_ptr<BRTSourceModel::CSourceModelBase>
_source) override
bool DisconnectSoundSource(const std::string & _sourceID) override

bool ConnectEnvironmentModel(const std::string & _environmentModelID)
override
bool DisconnectEnvironmentModel(const std::string &
_environmentModelID) override

void ResetProcessorBuffers()
void UpdateCommand() override
```

1. To perform the convolution task correctly, avoiding comb filters, it is necessary that the delays of the impulse responses have been removed. ↩

2.3.3 RIR models

Listener & Environment Models Based on BRIRs

The BRT Library includes environment models that use **Room Impulse Responses (RIRs)** to simulate realistic acoustic spaces. These models function by convolving the signals from sound sources with the **Binaural Room Impulse Response (BRIR)** of a specific room. A BRIR is an extension of the RIR that captures not only the acoustic characteristics of the room but also the binaural filtering effects caused by the listener's anatomy. This allows for a highly immersive audio experience, as the BRIR models how sound interacts with both the environment and the listener's physiology.

The BRIR typically includes both the direct sound path and the reverberation components of the sound field. It is recorded using microphones placed in or near the listener’s ears while a sound source emits test signals from different positions in the room. If the BRIR does not include information about the direct sound path, the rendering performed by the library will simulate only the acoustic characteristics of the environment, such as reflections and reverberation. This ensures flexibility in adapting to different BRIR datasets while maintaining accurate spatial rendering.

Data Source for BRIRs

The BRIR rendering models require BRIRsF data to be preloaded and preprocessed in the BRIR service module. For more details, see the documentation on the BRIR service module.

The BRIRs used by these models are typically stored in files formatted according to the SOFA (Spatially Oriented Format for Acoustics). This format is widely adopted in acoustic research for storing spatially oriented impulse responses. The BRT Library includes a dedicated module for reading and managing these files. This module is capable of loading BRIR data from SOFA files and extracting the necessary data for simulation. For more details, see the documentation on the Readers.

Implemented Models

Two BRIR-based environment models are currently available in the BRT Library:

- **Direct BRIR Convolution Model**: In this model, each sound source signal is convolved independently with the corresponding BRIR. This approach ensures the spatial resolution defined in the BRIR dataset, as each source has its own dedicated convolution process.
- **Ambisonic Reverberant Virtual Loudspeakers (RVL) Model**: This model encodes all sound sources into an ambisonic format of order 1, 2, or 3 before performing convolution in the ambisonic domain. By convolving the ambisonic signals with the BRIR, this model achieves more efficient processing when rendering multiple sources, as it requires fewer BRIR datasets for the directions of arrival.

Refining the Impulse Response

To optimize the impulse response, the BRT Library allows for windowing techniques that refine the BRIR. A rectangular window with smoothed edges can be applied using raised cosine fade-in and fade-out transitions, with a configurable slope. This process supports:

- **Silencing Initial Impulse Components**: Useful for removing the direct sound path, which could be simulated separately using a Listener Model, or eliminating early reflections, which might be simulated using the Image Source Method in a hybrid approach.
- **Truncating Late Impulse Components**: Reduces computational cost when working with long BRIRs and allows combining the BRIR convolution with simpler models for late reverberation tails, such as Feedback Delay Networks (FDN).

This flexibility provides greater control over computational efficiency and ensures compatibility with hybrid modeling approaches.

Windowing a Room Impulse Response

Windowing a Room Impulse Response

Windowing a Room Impulse Response

Listener & Environment Model based on BRIR direct convolution

The **Listener Direct BRIR Convolution Model** module enables spatial audio rendering from multiple sound sources. It uses direct convolution with the binaural room impulse responses (BRIR[1]) to simulate both direct sound and reverberation, providing a complete representation of the acoustic interaction between the source and the listener. If the impulse responses do not include information about the direct path, the simulation is limited to the reverberation of the environment[2].

Each sound source is processed independently, considering its relative position with respect to the listener. In the final step, the model combines all channels, left and right separately, to produce a single output per ear.

Convolutions are performed in the frequency domain using the uniform partitioned convolution algorithm. These impulse responses are provided by the **BRIR service module** for each audio frame. To ensure proper functioning, the service module must be configured beforehand. More information can be found in the service section.

Distance Attenuation

While not entirely accurate, this model also includes a distance simulation, which is disabled by default. This is a simulation of the distance attenuation due to propagation in space. It is based on the **inverse square law**, which states that the intensity of sound decreases proportionally to the square of the distance to the source. This phenomenon, termed d**distance-based attenuation**, is a critical factor in our perception of sound intensity.

The input signal is **attenuated based on the distance** between the source and the listener, replicating the natural reduction in sound intensity over distance. This means that a fixed amount of attenuation is applied to the signal each time the distance is doubled from a reference distance. By default, this attenuation has a value of -3 dB and the reference distance is 1 metre. Thus, the applied attenuation is calculated using the following expression:

\(attenuation = 10^{(Distance Attenuation Factor/ -3 dB) * log_{10}(ReferenceDistance / distance)}\)

ARCHITECTURE

The internal block diagram of this class is as follows:

Listener Direct BRIR Convolution Model Internal diagram

Listener Direct BRIR Convolution Model Internal diagram.

CONFIGURATION OPTIONS

This model allows configuration by calling its methods or by BRT internal commands:

- **Model (on/off)**: Silent when off.
- **Spatialization (on/off)**: Transparent when off.
- **Interpolation (on/off)**: When switched on, BRIRs are calculated at the exact position (relative source-listener position). For this purpose, barycentric interpolation is performed, starting from the three closest points. When it is switched off, the HRIR with the closest position are chosen.
- **BRIR to be used**: The BRIR service module to be used for rendering. The system supports dynamic, hot-swapping of the service module being used.
- **Distance Attenuation (on/off)**: No attenuation is applied when disabled.
- **Distance Attenuation Factor**: Sets a new distance attenuation factor, default is -3dB.

CONNECTIONS

Modules to which it supports connections:

```
- Source models
```

Modules to which it connects:

```
- Listener
- Bilateral Filter
```

r C++ developer

- **File**: /include/ListenerModels/ListenerAmbisonicVirtualLoudspeakersModel.hpp
- **Class name**: CListenerAmbisonicVirtualLoudspeakersModel
- **Inheritance**: CListenerModelBase
- **Namespace**: BRTListenerModel
- **Classes that instance**:
- BRTProcessing::CHRTFConvolverProcessor

CLASS INHERITANCE DIAGRAM

Listener Direct BRIR Convolution Model Internal diagram

Listener Direct BRIR Convolution Model Internal diagram.

HOW TO INSTANTIATE

```
// Assuming that the ID of this listener model is contained in
_listenerModelID.
brtManager.BeginSetup();
std::shared_ptr<BRTListenerModel::CListenerAmbisonicVirtualLoudspeakersMo
=
brtManager.CreateListenerModel<BRTListenerModel::CListenerAmbisonicVirtua
brtManager.EndSetup();
if (listenerModel == nullptr) {
    // ERROR
}
```

HOW TO CONNECT

Connect it to a listener.

```
// Assuming that the ID of this listener is contained in _listenerID
and
// that the ID of this listener model is contained in
_listenerModelID.
std::shared_ptr<BRTBase::CListener> listener =
brtManager.GetListener(_listenerID);
if (listener != nullptr) {
    brtManager.BeginSetup();
    bool control = listener->ConnectListenerModel(_listenerModelID);
    brtManager.EndSetup();
}
```

Connect a source model to it.

```
// Assuming that the soundSource could be a ID(string) or a
std::shared_ptr<BRTSourceModel::CSourceModelBase>;
std::shared_ptr<BRTListenerModel::CListenerModelBase> listenerModel =
brtManager-
>GetListenerModel<BRTListenerModel::CListenerModelBase>(_listenerModelID)
if (listenerModel != nullptr) {
    bool control = listenerModel->ConnectSoundSource(soundSource);
}
```

PUBLIC METHODS

```
void EnableModel() override
void DisableModel() override

void EnableSpatialization() override
void DisableSpatialization() override
bool IsSpatializationEnabled() override

void EnableInterpolation() override
void DisableInterpolation() override
bool IsInterpolationEnabled() override

void EnableDistanceAttenuation() override
void DisableDistanceAttenuation() override
bool IsDistanceAttenuationEnabled() override

bool SetDistanceAttenuationFactor(float _distanceAttenuationFactorDB)
override
float GetDistanceAttenuationFactor() override

void ResetProcessorBuffers()

bool SetHRBRIR(std::shared_ptr<BRTServices::CHRBRIR> _listenerBRIR)
override
std::shared_ptr<BRTServices::CHRBRIR> GetHRBRIR() const override
void RemoveHRBRIR() override

bool ConnectSoundSource(const std::string & _sourceID) override
bool
ConnectSoundSource(std::shared_ptr<BRTSourceModel::CSourceModelBase>
_source) override
bool DisconnectSoundSource(const std::string & _sourceID) override
bool
DisconnectSoundSource(std::shared_ptr<BRTSourceModel::CSourceModelBase>
_source) override

void UpdateCommand() override
```

1. A BRIR captures the acoustic characteristics of a room from the perspective of a specific listener, as it is recorded using microphones placed in the listener's ears. This is why we refer to this module as both a listener and environment model. ↩
2. This is the typical way we use the library. ↩

Listener Model based on BRIR convolution in the Ambisonics domain

The **Listener Ambisonic Reverberant Virtual Loudspeakers (RVL) Model** enables spatial audio rendering from multiple sound sources. It performs convolution in the ambisonic domain using binaural room impulse responses (BRIR[1]) and the selected ambisonic order (currently up to third order). This process simulates both direct sound and reverberation, providing a complete representation of the acoustic interaction between the source and the listener. If the impulse responses do not include information about the direct path, the simulation is limited to the reverberation of the environment[2]. The process consists of two main stages: ambisonic encoding and ambisonic convolution/decoding.

In the first stage, a ambisonic encoding is performed for each input sound source. This encoding generates (N) ambisonic channels per ear, where (N) depends on the selected ambisonic order, and per sound source.

In the second stage, two tasks are performed simultaneously: convolution with impulse responses and ambisonic decoding (for more details, see the AmbisonicBIR section). There are two convolution/decoding blocks, one for each ear, allowing independent processing for the left and right channels. Each block begins by separately mixing the ambisonic channels generated during the encoding stage. It then performs convolution in the frequency domain for each channel using precomputed impulse responses stored in the AmbisonicBIR service module. These impulse responses represent the ambisonic mixture of the virtual loudspeaker responses, enabling both convolution and ambisonic decoding to be executed simultaneously. Finally, the output is mixed and transformed back into the time domain, resulting in the final signal for the corresponding ear.

This modular approach ensures efficient and precise spatial audio rendering. The linearity of the operations is leveraged to reduce computational overhead by minimizing the number of convolutions required. The result is a highly realistic simulation of the direct sound path in the ambisonic domain, offering support for scalable ambisonic orders up to the third order.

For further details on the functionality of the Bilateral Ambisonic Encoder and the Ambisonic Domain Convolver, refer to their respective sections in the documentation.

Distance Attenuation

While not entirely accurate, this model also includes a distance simulation, which is disabled by default. This is a simulation of the distance attenuation due to propagation in space. It is based on the **inverse square law**, which states that the intensity of sound decreases proportionally to the square of the distance to the source. This phenomenon, termed d**distance-based attenuation**, is a critical factor in our perception of sound intensity.

The input signal is **attenuated based on the distance** between the source and the listener, replicating the natural reduction in sound intensity over distance. This means that a fixed amount of attenuation is applied to the signal each time the distance is doubled from a reference distance. By default, this attenuation has a value of -3 dB and the reference distance is 1 metre. Thus, the applied attenuation is calculated using the following expression:

\(attenuation = 10^{(Distance Attenuation Factor/ -3 dB) * log_{10}(ReferenceDistance / distance)}\)

ARCHITECTURE

The internal block diagram of this class is as follows:

Listener Ambisonic Reverberant Virtual Loudspeakers (RVL) Model - Internal diagram

Listener Ambisonic Reverberant Virtual Loudspeakers (RVL) Model - Internal diagram.

CONFIGURATION OPTIONS

This model allows configuration by calling its methods or by BRT internal commands:

- **Model (on/off)**: Silent when off.
- **BRIR to be used**: The BRIR service module to be used for rendering. The system supports dynamic, hot-swapping of the service module being used.
- **Ambisonic Order**: The order of the ambisonic coding to be used. Currently only orders between 1 (default) and 3 are valid.
- **Ambisonic Normalization**: The ambisonic normalization to be used. The available options are: N3D (default), SN3D, maxN
- **Distance Attenuation (on/off)**: No attenuation is applied when disabled.
- **Distance Attenuation Factor**: Sets a new distance attenuation factor, default is -3dB.

CONNECTIONS

Modules to which it supports connections:

```
- Source models
```

Modules to which it connects:

```
- Listener
- Bilateral Filter
```

r C++ developer

- **File**: /include/ListenerModels/ListenerAmbisonicReverberantVirtualLoudspeakersModel.hpp
- **Class name**: CListenerAmbisonicReverberantVirtualLoudspeakersModel
- **Inheritance**: CListenerModelBase
- **Namespace**: BRTListenerModel
- **Classes that instance**:
- BRTProcessing::CAmbisonicDomainConvolverProcessor
- BRTProcessing::CBilateralAmbisonicEncoderProcessor

CLASS INHERITANCE DIAGRAM

Listener Ambisonic Reverberant Virtual Loudspeakers (RVL) Model class diagram

Listener Ambisonic Reverberant Virtual Loudspeakers (RVL) Model - class diagram.

HOW TO INSTANTIATE

```
// Assuming that the ID of this listener model is contained in
_listenerModelID.
brtManager.BeginSetup();
std::shared_ptr<BRTListenerModel::CListenerAmbisonicReverberantVirtualLou
=
brtManager.CreateListenerModel<BRTListenerModel::CListenerAmbisonicReverb
brtManager.EndSetup();
if (listenerModel == nullptr) {
    // ERROR
}
```

HOW TO CONNECT

Connect it to a listener.

```
// Assuming that the ID of this listener is contained in _listenerID
and
// that the ID of this listener model is contained in
_listenerModelID.
std::shared_ptr<BRTBase::CListener> listener =
brtManager.GetListener(_listenerID);
if (listener != nullptr) {
    brtManager.BeginSetup();
    bool control = listener->ConnectListenerModel(_listenerModelID);
    brtManager.EndSetup();
}
```

Connect a source model to it.

```
// Assuming that the soundSource could be a ID(string) or a
std::shared_ptr<BRTSourceModel::CSourceModelBase>;
std::shared_ptr<BRTListenerModel::CListenerModelBase> listenerModel =
brtManager-
>GetListenerModel<BRTListenerModel::CListenerModelBase>(_listenerModelID)
if (listenerModel != nullptr) {
    bool control = listenerModel->ConnectSoundSource(soundSource);
}
```

PUBLIC METHODS

```
void EnableModel() override
void DisableModel() override

bool SetAmbisonicOrder(int _ambisonicOrder) override
int GetAmbisonicOrder() override

bool SetAmbisonicNormalization(Common::TAmbisonicNormalization
_ambisonicNormalization) override
bool SetAmbisonicNormalization(std::string _ambisonicNormalization)
override
Common::TAmbisonicNormalization GetAmbisonicNormalization() override

void EnableDistanceAttenuation() override
void DisableDistanceAttenuation() override
bool IsDistanceAttenuationEnabled() override

bool SetDistanceAttenuationFactor(float _distanceAttenuationFactorDB)
override
float GetDistanceAttenuationFactor() override

bool SetHRBRIR(std::shared_ptr<BRTServices::CHRBRIR> _listenerBRIR)
override
std::shared_ptr<BRTServices::CHRBRIR> GetHRBRIR() const override
void RemoveHRBRIR() override

bool
ConnectSoundSource(std::shared_ptr<BRTSourceModel::CSourceModelBase>
_source) override
bool ConnectSoundSource(const std::string & _sourceID) override
bool
DisconnectSoundSource(std::shared_ptr<BRTSourceModel::CSourceModelBase>
_source) override
bool DisconnectSoundSource(const std::string & _sourceID) override

void ResetProcessorBuffers()
void UpdateCommand() override
```

1. A BRIR captures the acoustic characteristics of a room from the perspective of a specific listener, as it is recorded using microphones placed in the listener's ears. This is why we refer to this model as both a listener and environment model. ↩
2. This is the standard way we use the library. ↩

2.4 Environment Models

2.4.1 Environment Models

Environmental simulations play a crucial role in achieving realistic binaural audio rendering by replicating how sound interacts with its surroundings. These simulations model the acoustic behavior of spaces, including reflections, reverberation, and the effects of air propagation, to recreate an immersive auditory experience that closely resembles real-world scenarios. By combining environmental models with listener and source models, the BRT Library ensures accurate spatial audio reproduction.

What we call environment models in the BRT Library are models that simulate the propagation of sound in a given acoustic environment using geometric algorithms based on virtual sources. Currently, the BRT Library provides two environmental models:

- Free Field Environment Model: Simulates direct sound propagation in open spaces, accounting for propagation delay, distance-based attenuation, and filtering effects caused by the medium.
- SDN Environment Model: Implements room acoustics simulation using Scattering Delay Networks[1], a computationally efficient algorithm for modeling reflections and reverberation in enclosed spaces.

And we are working to incorporate the following:

- *ISM*: Simulates room early reflections using the Image Source Method in complex room shapes *(Under development)*.
- *Hybrid: ISM + Convolution*: Simulates room reverberation where early reflections are modeled using the Image Source Method, and the reverberant tail is simulated through convolution with a BRIR *(Under development)*.

1. de Sena, E., Hacihabiboglu, H., & Cvetkovic, Z. (2011, February 2). Scattering Delay Network: An Interactive Reverberator for Computer Games. 41st International Conference: Audio for Games. ↩

2.4.2 Free Field Environment Model

The **Free Field Environment Model** simulates sound propagation in a free-field environment, focusing on the direct sound path. Specifically, it models the propagation delay, distance-based attenuation, and filtering effects caused by the medium.

In a free-field environment, sound propagates without any interaction with obstacles or boundaries, such as walls or objects, allowing it to travel unimpeded from the source to the listener. This type of propagation is characterized by the **inverse-square law**, which states that the sound intensity decreases proportionally to the square of the distance from the source. This phenomenon, known as **distance-based attenuation**, plays a crucial role in how we perceive the loudness of sounds in open spaces.

Additionally, sound waves traveling through a medium, such as air, are subject to **frequency-dependent filtering**. Higher frequencies are more susceptible to attenuation due to energy absorption by the medium, which results in a natural filtering effect as the distance increases. This filtering can significantly impact the timbre of sounds heard at greater distances.

The model also accounts for **propagation delay**, which represents the time it takes for sound to travel from the source to the listener. This delay is determined by the speed of sound in the medium, which is approximately 343 m/s in air under standard atmospheric conditions. By accurately simulating these factors, the Free Field Environment Model provides a realistic representation of sound propagation in open environments.

Functional overview

The **Free Field Environment Model** simulates sound propagation in open spaces. It processes monaural input signals connected from source models and generates virtual monaural output sources that are connected to listener models. For each source connected to the model, a corresponding virtual output source is created to represent the processed signal. The user only needs to specify the listener model to which this environment model is connected, after which sources can be freely added or removed as needed.

SIGNAL PROCESSING CHAIN

The model’s signal processing involves three main stages. First, the input signal is **attenuated based on the distance** between the source and the listener, replicating the natural reduction in sound intensity over distance. This means that a fixed amount of attenuation is applied to the signal each time the distance is doubled from a reference distance. By default, this attenuation has a value of -6.0206 dB and the reference distance is 1 metre. Thus, the applied attenuation is calculated using the following expression:

\(attenuation = 10^{(Distance Attenuation Factor/ -6.0206 dB) * log_{10}(ReferenceDistance / distance)}\)

Although *not yet implemented*, a **filtering** stage will eventually simulate the frequency-dependent attenuation effects caused by propagation through air

Finally, the signal passes through a **delay line** that accurately models the propagation delay corresponding to the distance between the source and listener. This sequence ensures that the output signals reflect realistic free-field sound propagation.

DYNAMIC ADAPTATION

The delay line is dynamically updated to account for changes in the relative positions of the source and the listener, maintaining accurate modeling of propagation effects in real-time. This dynamic behavior makes the Free Field Environment Model a vital component for simulating direct sound paths in spatial audio systems.

Configuration Options

This model allows configuration by calling its methods or by BRT internal commands:

- **Model (on/off)**: Silent when off.
- **Gain (float)**: Extra gain to be applied to the model output.
- **Distance Attenuation (on/off)**: No attenuation is applied when disabled.
- **Propagation Delay (on/off)**: No delay is applied when disabled.
- **Distance Attenuation Factor**: Sets a new distance attenuation factor, default is -6.0206dB.

Connections

Modules to which it supports connections:

```
- Source models
```

Modules to which it connects:

```
- Listener models
```

r C++ developer

- **File**: /include/EnvironmentModels/FreeFieldEnvironmentModel.hpp
- **Class name**: CFreeFieldEnvironmentModel
- **Inheritance**: CEnviromentModelBase
- **Namespace**: BRTEnvironmentModel
- **Classes that instance**:
- BRTEnvironmentModel::CFreeFieldEnvironmentProcessor

Class inheritance diagram

Free field Model Internal diagram

Free field Model Internal diagram.

How to instantiate

```
// Assuming that the ID of this environment model is contained in
_environmentID.
brtManager.BeginSetup();
std::shared_ptr<BRTEnvironmentModel::CFreeFieldEnvironmentModel>
environmentModel =
brtManager.CreateEnvironment<BRTEnvironmentModel::CFreeFieldEnvironmentMo
brtManager.EndSetup();
if (environmentModel == nullptr) {
    // error
}
```

How to connect

Connect it to a listener model.

```
// Assuming that the ID of this environment model is contained in
_environmentModelID and
// that the ID of this listener model is contained in
_listenerModelID.
std::shared_ptr<BRTListenerModel::CListenerModelBase> listenerModel =
brtManager.GetListenerModel<BRTListenerModel::CListenerModelBase>(_listen
if (listenerModel != nullptr) {
    brtManager.BeginSetup();
    bool control = listenerModel-
>ConnectEnvironmentModel(_environmentModelID);
    brtManager.EndSetup();
}
```

Connect a source model to it.

```
// Assuming that the ID of this source model is contained in _sourceID
and
// that the ID of this environment is contained in
_environmentModelID.
std::shared_ptr<BRTEnvironmentModel::CEnviromentModelBase>
environmentModel = brtManager-
>GetEnvironmentModel<BRTEnvironmentModel::CEnviromentModelBase>(_environm
if (environmentModel != nullptr) {
    bool control = environmentModel->ConnectSoundSource(_sourceID);
}
```

Public methods

```
void EnableModel()
void DisableModel()

void SetGain(float _gain)
float GetGain()

void EnableDistanceAttenuation()
void DisableDistanceAttenuation()
bool IsDistanceAttenuationEnabled()
```

```
void EnablePropagationDelay()
void DisablePropagationDelay()
bool IsPropagationDelayEnabled()

bool SetDistanceAttenuationFactor(float _distanceAttenuationFactorDB)
float GetDistanceAttenuationFactor()

bool
ConnectSoundSource(std::shared_ptr<BRTSourceModel::CSourceModelBase>
_source)
bool ConnectSoundSource(const std::string & _sourceID)

bool
DisconnectSoundSource(std::shared_ptr<BRTSourceModel::CSourceModelBase>
_source)
bool DisconnectSoundSource(const std::string & _sourceID)

void ResetProcessorBuffers()

void UpdateCommand()
```

2.4.3 SDN Environment Model

The **SDN Environment Model** implements an acoustic room simulation using **Scattering Delay Networks (SDNs)**. This approach models the acoustics of a room by employing acoustic reverberators, as described by Enso De Sena et al. in the paper Efficient Synthesis of Room Acoustics via Scattering Delay Networks.

Scattering Delay Networks simulate the acoustics of an enclosure using a network of delay lines connected by scattering junctions. The parameters of the model are derived from the physical properties of the simulated room, enabling a realistic approximation of the room's acoustic behavior. SDNs accurately model first-order reflections and make progressively coarser approximations for higher-order reflections. The algorithm supports unequal and frequency-dependent wall absorption, directional sound sources, and microphones, making it versatile for various room configurations.

Functional overview

The **SDN Environment Model** simulates the acoustics of a shoebox-shaped room by processing input signals from connected source models and generating virtual output sources linked to the corresponding listener models. For each connected source, the model creates seven virtual output sources: six representing the reflections from the walls, ceiling, and floor, and one representing the direct sound path. The user only needs to specify the listener model to which the SDN Environment Model is connected, allowing sources to be freely added or removed as required.

For each source, the model constructs scattering nodes and waveguides corresponding to the six reflective surfaces of the room. These virtual sources are positioned based on the geometry of the room and the relative positions of the source and the listener, with the delay and attenuation for each source calculated accordingly. The additional virtual source representing the direct sound path ensures the simulation includes both the reflections and the direct propagation.

To configure the simulation accurately, users must define the room’s dimensions and the absorption coefficients for each wall. The model assumes the room’s center is located at the origin of the coordinate system (0, 0, 0), ensuring a consistent spatial framework for the simulation. The shoebox-shaped room is described by three parameters:

- **Length**: The dimension of the room along the X-axis.
- **Width**: The dimension of the room along the Y-axis.
- **Height**: The dimension of the room along the Z-axis.

Furthermore, the definition of absorption coefficients is required for each wall, which can be defined as either frequency-independent or frequency-dependent (9 bands).

Architecture

The internal block diagram of this class is as follows:

SDN environment Model Internal diagram

SDN environment Model Internal diagram.

Configuration Options

This model allows configuration by calling its methods or by BRT internal commands:

- **Model (on/off)**: Silent when off.
- **Gain (float)**: Extra gain to be applied to the model output.
- **Direct path (on/off)**: Direct path silent when off.
- **Reverb path (on/off)**: Reverb path silent when off.
- **Setup ShoeBox Room (dimensions)**: Set the room geometry. Only shoebox rooms (six walls) are allowed for the SDN model
- **Setup Room Wall Absorption (absorption)**: Set walls absortion coefficients. Absorptions are set for nine octave bands: 62.5Hz, 125Hz, 250Hz, 500Hz, 1KHz, 2KHz, 4KHz, 8KHz and 16KHZ

Connections

Modules to which it supports connections:

```
- Source models
```

Modules to which it connects:

```
- Listener models
```

r C++ developer

- **File**: /include/EnvironmentModels/SDNEnvironmentModel.hpp
- **Class name**: CSDNEnvironmentModel
- **Inheritance**: CEnviromentModelBase
- **Namespace**: BRTEnvironmentModel
- **Classes that instance**:
- BRTEnvironmentModel::CSDNEnvironmentProcessor
- Common::CRoom

Class inheritance diagram

Free field Model Internal diagram

Free field Model Internal diagram.

How to instantiate

```
// Assuming that the ID of this environment model is contained in
_environmentID.
brtManager.BeginSetup();
std::shared_ptr<BRTEnvironmentModel::CSDNEnvironmentModel>
environmentModel =
brtManager.CreateEnvironment<BRTEnvironmentModel::CSDNEnvironmentModel>(_
brtManager.EndSetup();
if (environmentModel == nullptr) {
    // error
}
```

How to connect

Connect it to a listener model.

```
// Assuming that the ID of this environment model is contained in
_environmentModelID and
// that the ID of this listener model is contained in
_listenerModelID.
std::shared_ptr<BRTListenerModel::CListenerModelBase> listenerModel =
brtManager.GetListenerModel<BRTListenerModel::CListenerModelBase>(_listen
if (listenerModel != nullptr) {
    brtManager.BeginSetup();
    bool control = listenerModel-
>ConnectEnvironmentModel(_environmentModelID);
    brtManager.EndSetup();
}
```

Connect a source model to it.

```
// Assuming that the ID of this source model is contained in _sourceID
and
// that the ID of this environment is contained in
_environmentModelID.
std::shared_ptr<BRTEnvironmentModel::CEnviromentModelBase>
environmentModel = brtManager-
>GetEnvironmentModel<BRTEnvironmentModel::CEnviromentModelBase>(_environm
if (environmentModel != nullptr) {
    bool control = environmentModel->ConnectSoundSource(_sourceID);
}
```

Public methods

```
void EnableModel() override
void DisableModel() override

void SetGain(float _gain) override
float GetGain()

bool SetupShoeBoxRoom(float length, float width, float height)
Common::CRoom GetRoom()

bool SetRoomWallAbsortion(int wallIndex, float absortion)
bool SetRoomAllWallsAbsortion(float _absortion)
bool SetRoomWallAbsortion(int wallIndex, std::vector<float>
absortionPerBand)
bool SetRoomAllWallsAbsortion(std::vector<float> absortionPerBand)

void EnableDirectPath() override
void DisableDirectPath() override
bool IsDirectPathEnabled() override

void EnableReverbPath() override
void DisableReverbPath() override
bool IsReverbPathEnabled() override

bool
ConnectSoundSource(std::shared_ptr<BRTSourceModel::CSourceModelBase>
_source) override
bool ConnectSoundSource(const std::string & _sourceID) override

bool
DisconnectSoundSource(std::shared_ptr<BRTSourceModel::CSourceModelBase>
_source) override
bool DisconnectSoundSource(const std::string & _sourceID) override

void ResetProcessorBuffers()

void UpdateCommand() override
```

2.5 Bilateral Filters

2.5.1 Bilateral Filters

The CBilateralFilter class provides a mechanism for filtering binaural audio signals, with support for configurable filter stages. It can process signals considering both static and dynamic listener and source positions. The class allows for flexibility in enabling or disabling the processor, adjusting filter coefficients, and resetting buffers for new processing cycles.

Functional Overview

This model implements binaural filtering through second-order sections (SOS). It applies filtering based on the source and listener positions, allowing for spatially accurate sound filtering in 3D environments. The filter uses a chain of second-order filters for both the left and right ears and applies various methods to configure, enable, disable, and process the audio signals.

KEY METHODS:

- **Setup** : Initializes the binaural filter by setting up the number of filter stages for both left and right ears.
- **Set Coefficients** : Stores the filter coefficients for the left and right ears.
- **Enable Processor** : Enables the filter processor, allowing it to process signals.
- **Disable Processor** : Disables the filter processor, preventing it from processing audio signals.
- **Is Processor Enabled** : Returns a boolean indicating whether the processor is enabled.
- **Process** : Filters the input audio signal using the binaural filter, considering the positions and orientations of the source and listener.
- **Reset Process Buffers** : Resets the buffers used in the processing chain for both left and right ears, clearing any previous signal data.

2.6 Services Modules

2.6.1 Service Modules

Service modules are auxiliary components designed to manage and supply the essential resources required for process models to perform binaural audio synthesis algorithms effectively. These resources include various key elements, such as the Head-Related Transfer Function (HRTF), which is critical for simulating the direct sound path perceived by the listener model; the Binaural Room Impulse Response (BRIR), which enables accurate simulation of room reverberation effects; and directivity data, which is vital for capturing the directional characteristics of sound sources. Additionally, second-order filters are employed to accurately model near-field effects, ensuring precise spatial audio reproduction even in close-proximity scenarios. Together, these resources form the foundation for creating immersive and realistic binaural audio experiences.

Currently, five service modules are implemented:

- HRTF: Stores head-related impulse responses indexed by azimuth and elevation.
- BRIR: Stores room-related impulse responses indexed by azimuth and elevation.
- Directivity TF: Stores transfer functions of a sound source based on the position of the listener and the sources.
- SOS Filters: Stores coefficients for second-order sections of a filter, which can be fixed or vary based on distance, azimuth, and/or elevation.
- Ambisonincs BIR: Stores the impulse responses of the virtual loudspeakers in the ambisonic domains, in order to achieve a process with simultaneous impulse responses convolution and ambisonic decoding.

2.6.2 HRTF Service Module

The HRTF service module is responsible for handling the reading, storage, and processing of HRTFs. These functions are essential for enabling the listener model to render spatial audio (read more). This module optimizes the accessibility and organization of the HRTF data, ensuring efficient processing and integration within the overall rendering pipeline.

An HRTF consists of a collection of HRIRs, each associated with specific measurement directions, including azimuth (left-right) and elevation (up-down) and distance. The BRT is designed to handle arbitrary distributions of HRIR measurement directions, meaning it does not rely on a regular or complete directional grid and imposes no minimum density requirements. The HRTF Service Module is in charge of estimating HRIRs for the exact direction and distance of the source if these are not explicitly included in the HRTF data. For this estimation, the algorithm selects the three nearest points at which the HRTF was measured, and performs a barycentric interpolation among the HRIRs corresponding to these three locations. In this way, this module stores the HRTF in the form of a regular grid, enabling more efficient use of resources during binaural rendering. For more details about this grid, click here.

Interpolating between HRIRs with varying ITDs can lead to issues, such as audible artifacts and reduced rendering quality. To address this, the HRTF Service Module handles ITDs independently from the interpolation and convolution processes. To do so, user-imported HRIRs should be provided with ITDs stored separately. The ITDs to be added after interpolation can be either estimated by interpolating among those corresponding to the three closest HRIRs, or synthesised using data about the location of the sound source (specifically the interaural azimuth) and the head circumference of the listener.

Each HRIR is partitioned in chunks to match the input buffer length. This is done in order to use the UPOLS convolution. In addition, an FFT is applied to each of the HRIR partitions and stored in memory by the Service Module, since the convolution is done in the frequency domain.

The following diagram illustrates the processing performed on the HRTF table before it is stored. This processing is performed offline, ensuring that, in real-time, we have a regular table that is faster to access.

HRTF offline process

HRTF offline process.

Functional Overview

The methods in the **HRTF class** are designed to facilitate the loading and management of HRTFs within the renderer. This service allows loading multiple HRTFs as needed, enabling flexible spatial audio configurations. To load an HRTF file, a separate Reader class is required to parse the file and add HRIRs to the HRTF class one by one. The process begins by calling the **Begin Setup** method to initialize the configuration. HRIRs are then added incrementally using the **Add HRIR** method. Once all the data has been loaded, the **End Setup** method finalizes the setup, generating a complete HRTF table for use. Additionally, the class provides various configuration options, such as setting the sampling rate, adjusting the angular grid resolution, enabling the Woodworth ITD model, and defining head and ear properties, as detailed in the methods below. These features ensure precise and efficient integration of HRTFs into the rendering pipeline.

Configuration Options

The methods provided by this service are as follows.

- **Begin Setup**: Prepares the HRTF object for initializing and starts the configuration.
- **End Setup**: Finalizes the setup process and store in the configured HRTF data.
- **Set Sampling Rate**: Specifies the sampling rate for the HRTF data
- **Add HRIR**: Adds a Head-Related Impulse Response (HRIR) to the HRTF dataset for a specific position.
- **Set/Get file name**: Assigns or retrieves the file name of the HRTF data to be used.
- **Set/Get sampling step for the grid**: Defines or retrieves the angular step size used to sample the HRTF grid.
- **Enable/Disable Woodworth ITD**: Toggles the application of the Woodworth ITD formula for HRTF processing.
- **Get HRIR Partitioned**: Retrieves the interpolated and partitioned stored HRTF for an specific direction.
- **Get HRIR Delay**: Calculates and returns the delay associated with the HRIR for a specific position.
- **Set/Get Head Radius**: Configures or retrieves the radius of the listener's head model.
- **Set ear position**: Sets the position of the ears relative to the head model.

r C++ developer

- **File**: /include/LServiceModules/HRTF.hpp
- **Class name**: CHRTF
- **Inheritance**: CServicesBase
- **Namespace**: BRTServices

Class inheritance diagram

HRTF Class diagram

HRTF Class diagram.

How to instantiate and load

```
// Assuming SOFA_FILEPATH contains the SOFA filename including the
path
std::shared_ptr<BRTServices::CHRTF> hrtf =
std::make_shared<BRTServices::CHRTF>();
bool hrtfSofaLoaded = AppUtils::LoadSofaFile(SOFA_FILEPATH,
hrtf);
    if (!hrtfSofaLoaded) {
        // ERROR
    }
```

How to connect it to a listener

```
// Assuming that the ID of this listener is contained in _listenerID
and
// that the HRTF is already lsuccessfuly loaded into hrtf.
std::shared_ptr<BRTBase::CListener> listener = brtManager-
>GetListener(listenerID);
listener->SetHRTF(hrtf);
```

Public methods

```
int32_t GetHRIRLength() const
void SetGridSamplingStep(int _samplingStep)
int GetGridSamplingStep()

bool BeginSetup(int32_t _HRIRLength,
BRTServices::TEXTRAPOLATION_METHOD _extrapolationMethod) override
void AddHRTFTable(T_HRTFTable&& newTable)
void AddHRIR(double _azimuth, double _elevation, double _distance,
Common::CVector3 listenerPosition, THRIRStruct&& newHRIR) override
bool EndSetup() override

void EnableWoodworthITD()
void DisableWoodworthITD()
bool IsWoodworthITDEnabled()

const std::vector<CMonoBuffer<float>>
GetHRIRPartitioned(Common::T_ear ear, float _azimuth, float
_elevation, bool runTimeInterpolation, const Common::CTransform& /*
_listenerLocation*/ ) const override
THRIRPartitionedStruct GetHRIRDelay(Common::T_ear ear, float
_azimuthCenter, float _elevationCenter, bool runTimeInterpolation,
Common::CTransform& _listenerLocation)
const int32_t GetHRIRNumberOfSubfilters() const
const int32_t GetHRIRSubfilterLength() const override
bool IsHRTFLoaded()
float GetHRTFDistanceOfMeasurement() override

void SetTitle(std::string _title) override
void SetDatabaseName(std::string _databaseName) override
void SetListenerShortName(std::string _listenerShortName) override

void SetFilename(std::string _fileName) override
std::string GetFilename() override

void SetHeadRadius(float _headRadius) override
float GetHeadRadius() override

void SetEarPosition(Common::T_ear _ear, Common::CVector3
_earPosition) override
void SetCranialGeometryAsDefault() override
```

```
Common::CVector3 GetEarLocalPosition(Common::T_ear _ear)

void SetSamplingRate(int _samplingRate)
```

2.6.3 BRIR Service Module

The BRIR service module is responsible for managing the reading, storage, and processing of BRIRs. These functions are crucial for enabling the system to simulate room reverberation and provide accurate spatial audio rendering (read more). This module ensures that the BRIR data is well-organized and easily accessible, facilitating efficient processing and seamless integration within the overall audio rendering pipeline.

An BRIR consists of a collection of BRIRs, each associated with specific measurement directions, including azimuth (left-right) and elevation (up-down) and distance. The BRT is designed to handle arbitrary distributions of BRIR measurement directions, meaning it does not rely on a regular or complete directional grid and imposes no minimum density requirements. The HR-BRIR Service Module is in charge of estimating BRIRs for the exact direction and distance of the source if these are not explicitly included in the HRTF data. For this estimation, the algorithm selects the three nearest points at which the BRIR was measured, and performs a barycentric interpolation among the BRIRs corresponding to these three locations. In this way, this module stores the BRIR in the form of a regular grid, enabling more efficient use of resources during binaural rendering. For more details about this grid, click here.

Each BRIR is partitioned in chunks to match the input buffer length. This is done in order to use the UPOLS convolution. In addition, an FFT is applied to each of the HRIR partitions and stored in memory by the Service Module, since the convolution is done in the frequency domain.

The following diagram illustrates the processing performed on the HRTF table before it is stored. This processing is performed offline, ensuring that, in real-time, we have a regular table that is faster to access.

HRTF offline process

BRIR offline process.

Functional Overview

The methods in the HRBRIR class provide the tools necessary for configuring and managing Head-Related BRIR data within the renderer. This service supports loading multiple BRIRs, offering flexibility in spatial audio setups. To load a BRIR file, a dedicated Reader class is necessary to parse the file and add the BRIRs to the HRBRIR class step by step. The process begins with initializing the configuration using **Begin Setup** and concludes with **End Setup**, which locks in the loaded data. Additional methods enable setting sampling rates, managing file names, defining sampling steps for the BRIR grid, and retrieving partitioned BRIRs for specific directions. This class also includes the SetWindowingParameters method, which sets the parameters for the windowing process to optimize the impulse response.

Configuration Options

The methods provided by this service are as follows.

- **Begin Setup**: Prepares the HRBRIR object for initializing and starts the configuration.
- **End Setup**: Finalizes the setup process and store in the configured HRBRIR data.
- **Set Sampling Rate**: Specifies the sampling rate for the HRBRIR data
- **Add BRIR**: Adds a Head-Related Impulse Response (HRBRIR) to the HRBRIR dataset for a specific position.
- **Set/Get file name**: Assigns or retrieves the file name of the HRBRIR data to be used.
- **Set/Get sampling step for the grid**: Defines or retrieves the angular step size used to sample the HRBRIR grid.
- **Get BRIR Partitioned**: Retrieves the interpolated and partitioned stored HRBRIR for an specific direction.
- **SetWindowingParameters**: Sets the parameters for the windowing process, defining the midpoint and rise time for both fade-in and fade-out windows in the impulse response.

r C++ developer

Section under construction

2.6.4 Directivity Service Module

The Directivity TF service module is responsible for handling the reading, storage, and processing of the transfer functions of the directivity of a sound source. These functions are essential for enabling the source model to render the directionality of a sound source (read more). This module optimizes the accessibility and organization of the Directivity TF data, ensuring efficient processing and integration within the overall rendering pipeline.

A Directivity TF table consists of a collection of Directivity Transfer Functions, each associated with specific measurement directions, including azimuth (left-right) and elevation (up-down) and distance. The BRT is designed to handle arbitrary distributions of mesurement, meaning it does not rely on a regular or complete directional grid and imposes no minimum density requirements. The Directivity Service Module is in charge of estimating the Directivity transfer function for the exact direction and distance of the listener if these are not explicitly included in the loaded table. For this estimation, the algorithm selects the three nearest points at which the Directivity was measured, and performs a barycentric interpolation among the transfer functions corresponding to these three locations. In this way, this module stores the Directivity TF in the form of a regular grid, enabling more efficient use of resources during binaural rendering. For more details about this grid, click here.

Functional Overview

Directivity class methods manage the setup, configuration, and retrieval of directivity transfer functions (TFs). They allow initialization with specific parameters, adding TFs for given directions, retrieving existing TFs with optional runtime interpolation, and setting or querying the resampling step. The process of loading a new directivity table begins by calling **BeginSetup**, followed by adding transfer functions one by one using **AddDirectivityTF**, and finalizing the configuration with **EndSetup**. To load the directivity data, a Reader class is used to read the data files and load them into the class's directivity table. These features support efficient handling and processing of directional audio data.

Configuration Options

The methods provided by this service are as follows.

- **Begin Setup**: Initializes the configuration with parameters for transfer function length and extrapolation method.
- **End Setup**: Finalizes the setup process and validates the configuration.
- **Set/Get ResamplingStep**: Defines ans retrieves the resampling step size for directivity data grid.
- **Get Directivity TF Length**: Returns the length of the directivity transfer functions.
- **AddDirectivityTF**: Adds a directivity transfer function (TF) for a specific azimuth and elevation.
- **GetDirectivityTF**: Retrieves a directivity transfer function for a given azimuth and elevation, with optional runtime interpolation.

Directivity TF format

A set of requirements must be met to the directionality data to work properly in the BRT:

- **Frequency distribution**: the frequency distribution must be linear from 0 to the Nyquist frequency minus one. The number of samples must be the same as configured in the BRT. By default, BRT has a number of samples of 256, but it is configurable using the BeRTA application.
- **Casual filters**: the filters defining directivity must be causal. Therefore, the transfer functions must have a phase that corresponds to a causal filter.

r C++ developer

Section under construction

2.6.5 SOS Filter

The BRT simulates sources in the near-field implementing a compensation for the HRTF conventional processing, where the main algorithm is the convolution with an HRTF measured at a fixed distance. The renderer considers sources in the near fields when they are located at distances lower than 2 meters to the listener’s head. The implemented approach uses the model presented by Romblom & Cook (2008).

These difference filters are implemented as IIR filters adjusted to match the described transfer function. The BRT includes two biquad filters for each ear, where the coefficients for these filters depend on both the distance of the sound source and its interaural azimuth. These filters are pre-calculated and stored in a file as a look- up table. This process can be considered as an HRIR correction since it is applied in series with the HRIR selected and interpolated in the previous stages of the pipeline.

When the binaural spatialisation is performed a problem arises when the source or the listener are moving, since some audible artefacts can appear in the signal. In this particular case, those artefacts can be caused as the near-field correction filters have to change from frame to frame.

In order to minimise this problem, at every frame each biquad filter is applied using both the previous and the new coefficients, and a linear cross-fading is performed to produce the output.

Functional Overview

This class handles the configuration of second-order section (SOS) filter coefficients. It allows for the addition of SOS filter coefficients based on specific azimuths and distances. The class also provides functionality to retrieve the SOS filter coefficients based on the ear's position, distance, and azimuth.

Configuration Options

The methods provided by this service are as follows.

- **Add Coefficients**: Adds new second-order section (SOS) filter coefficients for a given interaural azimuth and distance.
- **Set Ear Position**: Sets the position of the left or right ear within the 3D space.
- **Get SOS Filter Coefficients**: Retrieves the second-order section (SOS) filter coefficients for a specific ear, distance, and interaural azimuth.

2.6.6 Ambisonic BRIR

The Ambisonic BRIR Convolution Module of the Binaural Rendering Toolbox enables the spatial audio rendering of multiple sound sources in an ambisonic environment. By leveraging the convolution with Binaural Room Impulse Responses (BRIR) in the ambisonic domain, this module processes the input sources considering their positions and the listener's position and orientation. It supports ambisonic encoding of orders 1, 2, or 3, and uses to decode virtual loudspeakers located at the vertices of regular polyhedra (octahedron for first-order, icosahedron for second-order, and dodecahedron for third-order).

In order to optimize the process, the module convolves the ambisonic signal directly with the ambisonic mix of the virtual loudspeakers' impulse responses, subsequently mixing all right and left channels separately, as it's shown in the diagram shown below. Uniformly partitioned convolution is used to compute the convolution in the frequency domain.

The diagram below illustrates the key components of the system and how they work together. From handling input parameters to generating and processing ambisonic signals, the system employs ambisonics and convolution techniques to accurately simulate 3D audio environments. The components and their respective functionalities are detailed below.

- **Input Handling**: The module accepts a set of audio sources along with their positions in space, as well as the position and orientation of the listener.
- **Ambisonic Encoding**: The relative positions of each source with respect to the listener are used to encode the sources into an ambisonic format of the specified order (1st, 2nd, or 3rd).
- **Virtual Loudspeaker**: Impulse responses are got from the BRIR file for those positions of teh virtual loudspeakers, arranged at the vertices of a regular polyhedron corresponding to the ambisonic order.
- **Convolution**: Rather than convolving each virtual loudspeaker signal with the HRIR individually, the module convolves the ambisonic signal directly with the ambisonic mix of the BRIRs for the virtual loudspeakers.
- **Channel Mixing**: The convoluted signals are mixed into separate left and right channels, resulting in the final binaural audio output.

HRTF offline process

Ambisonic BRIR model diagram.

Functional Overview

To properly load and configure a file using this class, the process must begin with the **Begin Setup** method, which initializes the system with a specified ambisonic order and normalization scheme. This step sets up the necessary parameters for subsequent operations. Once all required configurations and impulse responses have been added, the setup process is finalized using the **End Setup** method. This ensures that the class is ready for use, as indicated by the IsReady method, which verifies if the Ambisonic BIR is loaded and operational. During the setup, additional methods can be used to enhance the configuration. For instance, **Add Impulse Response** allows adding a new impulse response associated with a specific channel and listener position. Finally, the system provides methods for retrieving configuration details, such as **Get Channel Partitioned IR**, which fetches the partitioned impulse response for a given channel and ear, taking into account the listener's current position and orientation.

Configuration Options

The methods provided by this service are as follows.

- **Begin Setup**: Initializes the setup process with the specified parameters.
- **End Setup**: Completes the setup process and confirms the system is ready.
- **Reset**: Resets the system to its initial state.
- **Is Ready**: Checks if the system is ready and the BRIR is loaded.
- **Add Impulse Response**: Adds a new impulse response associated with a specific channel and listener position.
- **Get Channel Partitioned IR**: Retrieves the partitioned impulse response for a channel and a specific ear, considering the listener's position and orientation.
- **Add Impulse Responses From HRIR**: Adds multiple impulse responses from a shared HRIR resource.

BRIR Format

A BRIR (Binaural Room Impulse Response) must be loaded. Readers can be used to load BRIR SOFA files. Depending on the Ambisonic order, the direction of the virtual loudspeakers are searched in the BRIR file. If some of them is not found, an interpolation among the three closest directions is performed. If different listener positions are provided, a different set of virtual loudspeakers will be created for every listener position. When rendering, the one with the closest position to the listener is selected to convolve. To determine if two positions are the same, a resolution of 1 meter is used.

Depending on how the BRIR is captured, several cases may be considered

- One fixed listener with several source positions around it
- One listener position with different orientations and one fixed source
- Different listener positions with sources around

r C++ developer

Section under construction

2.7 SOFA Readers

2.7.1 Readers

The SOFA Reader class is designed to handle the reading and parsing of audio resource data from files in the SOFA (Spatially Oriented Format for Acoustics) format. SOFA is an open standard used for storing spatial acoustic data, such as Head-Related Transfer Functions (HRTFs), Binaural Room Impulse Responses (BRIRs), and other directional audio characteristics. This format is widely adopted in spatial audio applications due to its flexibility and compatibility with 3D audio systems.

This reader is provided to simplify the resource loading process for the BRT. The renderer includes a set of service models that offer interfaces for integrating spatial audio data into the library. The SOFA Reader bridges the gap between the SOFA format and the BRT's service models, ensuring efficient data loading and compatibility.

Data type and conventions supported by this reader are:

- **HRTF and BRIR reader** supports any convention with data types *FIR* or *FIR-E*, such as SimpleFreeFieldHRIR.
- **Directivity TF reader** supports any convention with data types *TF*, such as FreeFieldDirectivityTF.
- **SOS filter** reader supports any convention with data types *SOS*, such as SimpleFreeFieldHRSOS.

Configuration Options

The methods provided by this class are as follows.

- **Get Sample Rate From Sofa**: Extracts the sample rate specified in a given SOFA file.
- **Read HRTF From Sofa**: Reads the HRTF data from a SOFA file and loads into a service class, supporting specific spatial resolution and extrapolation methods.
- **Read SOS Filters From Sofa**: Reads the near-field compensation SOS filters from a SOFA file and loads into the corresponding service class.
- **Read DirectivityTF From Sofa**: Loads source directivity transfer functions from a SOFA file, configuring the spatial resolution and extrapolation method.
- **Read BRIR From Sofa**: Extracts BRIR data from a SOFA file, loading into a service class while applying windowing parameters for smooth transitions.

r C++ developer

Section under construction

2.8 Infrastructure Entities

2.8.1 Infrastructure Entities

The **Infrastructure Entities** encompass the medium- and low-level modules of the BRT Library. These modules form the foundational elements upon which the high-level models are built and operate. While they are essential to the library's functionality and architecture, understanding them is not required unless you plan to extend the library or modify its architecture.

MIDDLE LAYER

The **Middle Layer** includes a collection of signal processing and service modules that provide the necessary building blocks for high-level models. Signal processing modules handle tasks such as convolution and filtering, while service modules manage critical data like impulse responses. Developers looking to contribute new algorithms or models to the library will need to understand this layer, as it serves as the foundation for extending the library's functionality.

BOTTOM LAYER

The **Bottom Layer** represents the core of the library. It consists of fundamental classes and templates that define the modular architecture and implement mechanisms for interconnecting modules. This layer ensures the flexibility and scalability of the library, making it possible to adapt its architecture to various needs. Developers interested in modifying or extending the library's architecture will primarily work with this layer.

2.8.2 Processing Modules

Processing Modules

The **Processing Modules** encompass a variety of signal processing components used by high-level models to perform simulations and audio rendering tasks. These modules handle essential operations such as convolution, filtering, and encoding, acting as the core signal processing units for the BRT Library's functionality.

Modular Design

To promote flexibility and ease of development, the implementation of these modules typically separates signal processing from the connectivity required within the library. This design choice facilitates both the development of new modules and their reuse in other contexts. Each processing module is generally implemented with two distinct classes or files [1]:

- A **signal processing class**, which is independent of the BRT Library’s architecture but may rely on common components such as type definitions and vector classes. E.g. *MySignalProcessing.hpp*
- A **processor class**, which extends the signal processing class by integrating connectivity features required for interaction with other library modules. E.g. *MySignalProcessingProcessor.hpp*

For instance, in the case of the **HRTF Convolver**, the class `HRTFConvolver.hpp` implements the convolution logic without depending on connections to any other BRT module. On the other hand, the class `HRTFConvolverProcessor.hpp` integrates this functionality within the library while adding the necessary mechanisms for connecting to other modules.

Reusability and Integration

This modular design not only streamlines development within the BRT Library but also enhances the reusability of signal processing components. By decoupling core signal processing functions from library-specific connectivity, these modules can be easily integrated into external systems or reused in other audio processing contexts, making them highly versatile and portable.

LIST OF SIGNAL PROCESSOR MODULES

- **Ambisonic Domain Convolver & Processor**: The module performs ambisonic convolution for the N channels of each of the M different input sources, resulting in a monaural signal.
- **Ambisonic Encoder**: Encodes sound sources into an ambisonic format for spatial audio processing and rendering.
- **Bilateral Ambisonic Encode & Processor**: Process an audio source by applying delay and spatial expansion, simulating near-field effects through binaural filtering, and finally encoding the signals into left and right Ambisonic channels for spatial audio rendering.
- **Binaural Filter**: Implement binaural filtering from second-order stages.
- **Directivity TF Convolver**: Process audio signals by applying a source's directional transfer function (Directivity TF) to simulate realistic sound propagation.
- **Distance Attenuator**: Implements distance-based attenuation to simulate the natural decrease in sound intensity over distance.
- **Uniform Partitioned Convolution**: Implements efficient frequency-domain convolution using uniformly partitioned overlap-save methods, optimized for long impulse responses.
- **HRTF Convolver & Processor**: Manage the convolution of the input signals, using the Uniform Partitioned Convolution.
- **Near Field Effect Processor**: Integrate connectivity features to manage binaural filter processing

1. There are processing modules that are used internally by other modules or signal processors. In this type of module, the class that adds connectivity functionality has not been implemented. ↩

Ambisonic Domain Convolver Processor

The **Ambisonic Domain Convolver Processor** is a system designed for processing Ambisonic audio channels, performing frequency-domain transformations, convolutions, and mixing to produce spatially accurate sound for binaural listening. The processor handles multiple input channels and applies several processing stages, including FFT, convolution with Ambisonic BIR, and final mixing to create the output for the listener's ears.

ARCHITECTURE

The internal block diagram of this class is as follows:

BAmbisonic Domain Convolver Processor - Internal diagram

Ambisonic Domain Convolver Processor - Internal diagram.

As it is shown in the diagram, the **Ambisonic Domain Convolver Processor** takes multiple Ambisonic channels, applies FFT processing, convolves with Ambisonic BIR to create spatial effects, mixes the processed channels, and delivers the final output as binaural ear samples, simulating an immersive listening experience for the listener.

Key Components and Flow:

1. **Channel Mixer**: Each mixer processes the respective channels (`channel 1` through `channel N`) from each sound source, for further frequency-domain transformation.
2. **FFT Processor**: The *FFT Processor* blocks apply a Fast Fourier Transform. Each processed channel goes to Uniform Partitioned Convolution.
3. **Uniform Partitioned Convolution**: The frequency-domain channels are passed into the *Uniform Partitioned Convolution* blocks. These blocks convolve the signals with the *AmbisonicBIR*.
4. **All Channels - Mixer**: The convolved outputs from all channels (`Channel 1` through `Channel N`) are combined in the *All Channels - Mixer*. This step ensures the final output integrates contributions from all Ambisonic channels.
5. **Final Convolution and Output**: The mixed signal (`Channel mixed`) undergoes a final convolution step in the *FFT : UniformPartitionedConvolution* block. The result is output as *Ear samples*, simulating spatial audio perception at the listener's ears.

r **C++ developer**

(In progress)

Bilateral Ambisonic Encoder Processor

The **Bilateral Ambisonic Encoder Processor** processes an audio source by applying delay and spatial expansion, simulating near-field effects through binaural filtering, and finally encoding the signals into left and right Ambisonic channels for spatial audio rendering.

ARCHITECTURE

The internal block diagram of this class is as follows:

Bilateral Ambisonic Encoder Processor - Internal diagram

Bilateral Ambisonic Encoder Processor - Internal diagram.

Key Components and Flow

The **Bilateral Ambisonic Encoder Processor diagram** describes the internal workflow for encoding a source signal into Ambisonic channels for both left and right audio streams.

1. **AddDelayExpansionMethod**: The source signal is first passed through the `left` and `right` *AddDelayExpansionMethod* blocks. These methods apply a delay and spatial expansion to the signal to account for spatial positioning and propagation effects.
2. **Near Field Effect - Binaural Filter**: The processed `left` and `right` signals are then passed through a *Binaural Filter*. This stage simulates near-field audio effects implementing binaural filtering from second-order stages.
3. **AmbisonicEncoder**: The filtered `left` and `right` signals are fed into their respective *AmbisonicEncoder* blocks. These encoders process the samples and output *N Ambisonic Channels* for both left and right streams.
4. **Output**: The final output consists of *Left N Ambisonics Channels* and *Right N Ambisonics Channels*, which represent spatially encoded versions of the source signal.

r **C++ developer**

(In progress)

Uniformly Partitioned Convolution

For both anechoic and reverberation paths, the BRT utilizes the Uniformly Partition Overlap-Save (UPOLS) convolution method in the frequency domain, as described by Wefers (2015). This FFT-based approach divides the impulse response (IR) into multiple blocks, each matching the frame size (N). These blocks are processed individually as separate IRs using a standard overlap-save technique. This method enables efficient convolution with long IRs by breaking them into shorter, more manageable segments. It is particularly advantageous for long IRs, such as BRIRs.

The figure shows the whole process of the UPOLS in the case of the HRIR, but it is the same for the BRIR. The partitions and FFTs of the HRTF are computed offline, where the whole HRTF is partitioned in blocks of length N (same length as the frame size) and stored. At run-time, for every audio frame, a new input buffer arrives, the content of the input block buffer is shifted N samples to the left and the new input of N samples is then placed on the right (see bottom-left part of the diagram). Then, the whole input buffer is transformed into the frequency domain using a 2N-point real-to-complex FFT and stored in a delay-line, following an overlap-save scheme. In this way, in each audio frame, the input signal stored in the delay line is shifted up by one frame slot. During the audio frame, each delayed input buffer is convolved with each HRIR segment, by a multiplication in the complex domain. Finally, all the multiplications results are mixed and transformed back into the time-domain. To do so, the BRT implement a 2N-point complex-to-real IFFT, where the first N points are discarded, as we are using an overlap-save scheme. These sample removal and the size used for the FFT and IFFT of 2N-point are implemented to avoid aliasing in the time domain. In addition, zero-padding is used to complete the signal buffer in case it is needed.

Additionally, UPOLS assumes a stable impulse response, which is suitable for static sources and listeners. To accommodate the changes in the HRIR caused by the movement of sources or the listener, the BRT introduces a new delay line for the partitioned impulse response (top-left grey box). For each audio frame, both delay lines (HRIR and input buffer signal) are shifted by one frame slot. This allows a new HRIR, corresponding to the source's position, to be inserted into the delay line at each frame, with its first segment multiplied by the current audio frame. In subsequent frames, the HRIR remains in the delay line, convolving its remaining segments with the incoming audio frames. This method significantly reduces the number and intensity of artifacts caused by the movement of sources.

HRTF offline process

Uniformly Partition Overlap-Save (UPOLS).

FUNCTIONAL OVERVIEW

The methods of the UPOLS convolution class are designed to efficiently manage and execute frequency-domain convolution for both anechoic and reverberation paths. The process begins with the **Setup method**, where the input size, block size and number of blocks are configured to prepare the system for operation. Convolution can then be performed using **ProcessUPConvolution method**, which processes the input signal with the given impulse responses (IRs) and outputs the result. For scenarios with moving source and listener, **Process UPConvolution With Memory** extends this functionality. These methods collectively enable efficient convolution with long IRs by leveraging the Uniformly Partitioned Overlap-Save (UPOLS) algorithm.

CONFIGURATION OPTIONS

The methods provided by this class are as follows.

- **Setup**: Configures the system for UPOLS convolution with specified input and IR settings.
- **Process UPConvolution**: Executes convolution of the input signal with the provided impulse responses, generating the output.
- **Process UPConvolutionWithMemory**: Performs convolution while retaining intermediate states, for scenarios with moving sources and listener.
- **Calculate IFFT**: Computes the inverse FFT of a buffer for frequency-to-time domain transformation.
- **Reset**: Resets the system, clearing all internal states and buffers for a fresh start.

r **C++ developer**

Section under construction

2.8.3 Common

Common

- **Biquad filter**: Provides tools for applying second-order IIR filtering to audio signals. It supports filter setup using specified coefficients.
- **Buffer**: Manages temporary storage of audio data during processing, ensuring efficient data flow through the renderer.
- **Conventions**: Defines and enforces data format standards and spatial audio conventions, such as coordinate systems.
- **Cranial Geometry**: Models the geometry of the human head to enhance spatial audio accuracy by accounting for head-related effects.
- **Envelope Detector**: Extracts the amplitude envelope of an audio signal.
- **Error Handler**: Manages error detection and reporting during audio processing to ensure robustness and debugging support.
- **FFSG**: Implement the fast Fourier transforms by Takuya OOURA Copyright(C) 1996-2001.
- **Filters Chain**: A modular pipeline that allows chaining multiple audio filters for sequential processing of sound signals.
- **FFT calculator**: Audio signal processor that calculates FFTs and IFFTs.
- **IR Windowing**: Applies a window function to impulse responses (IRs).
- **Profiler**: Monitors and measures the performance of various components within the renderer, tracking the CPU usage of the rendering process.
- **Quaternion**: Represents rotational transformations in 3D space, crucial for accurately positioning sound sources and listeners.
- **Room**: Enable the creation and management of 3D room geometries and acoustic properties, supporting predefined shapes and wall configurations.
- **Transform**: Handles spatial transformations like translation, rotation, and scaling for positioning sources or listeners.
- **Vector3**: Represents 3D vectors, often used for positions, directions, or velocity in spatial audio calculations.
- **Wall**: Models a physical wall to simulate reflections and obstructions in room acoustics.
- **Wave Guide**: Simulates wave propagation in 3D space, often used for modeling sound in complex environments.

3. Applications

3.1 BRT Applications: General Overview

The **Binaural Rendering Toolbox (BRT)** includes a collection of applications designed to provide researchers and developers with ready-to-use tools for creating, testing, and deploying immersive audio experiences. These applications leverage the core functionalities of the BRT Library and extend them into practical, real-time tools for diverse use cases. This section introduces the main applications, their features, and intended use cases.

3.1.1 Introduction

The BRT Applications are standalone software tools built around the BRT Library, enabling seamless integration of binaural rendering into various workflows. These applications are tailored for both experimental research and real-world implementation, offering high configurability and interoperability with existing systems. From immersive auditory experiments to wearable solutions, BRT Applications are designed to meet the demands of modern psychoacoustics and virtual acoustics research.

3.1.2 Key Features

1. **Interoperability**:

- All BRT Applications support the **Open Sound Control (OSC)** protocol for external control, allowing integration with other tools and frameworks such as MATLAB, PureData, and Unity 3D.
- Compatible with standard file formats like **SOFA** and **WAV**, ensuring smooth data exchange and reproducibility.

2. **Reproducibility**:

- Applications record all relevant parameters of binaural simulations, facilitating accurate reproduction and analysis of experimental results.
- Support for a new SOFA-based convention for annotated audio, capturing dynamic spatial configurations alongside binaural audio.

3.1.3 Applications

BeRTA Renderer (BRT Audio Renderer)

BeRTA Renderer is the flagship application of the BRT Toolbox, providing a high degree of configurability for real-time spatial audio rendering. Key functionalities include:

- **Real-time Rendering**:
- Dynamic control of sound source locations, listener positions, and other spatial parameters via OSC commands.
- Support for multiple sound sources and HRTF swapping in real time.
- **Configurable Virtual Scenarios**:
- User-defined configurations for different buffer sizes, sample rates, and rendering parameters.
- GUI-based feedback for monitoring application status, including rendering states and OSC commands.
- **Data Saving and Export**:
- Saves all parameters and audio in the annotated audio format for reproducibility.
- Exports simulation data for further analysis or use in auditory modeling.

BeRTA GUI (BRT Audio Renderer GUI)

BeRTA GUI is a graphical interface (Windows x64 and MacOS) to control BeRTA Renderer via OSC.

Wearable Solutions

BRT Toolbox also includes applications designed for portable, low-latency rendering on wearable devices:

- **BeRTA-mini**:
- A lightweight version of BeRTA tailored for wearable platforms, combining a BELA audio board, IMU sensors (gyroscope, magnetometer, accelerometer), and headphones.
- Designed for on-the-go binaural rendering with high responsiveness and accuracy.
- **Tracker Interoperability**:
- Integrates with MBIENTLAB’s MetaMotionRL and other motion tracking devices.
- Converts tracking data into OSC commands for seamless interaction with other BRT applications.

Future Applications

Several additional tools are under development or planned for future releases, including: - Multi-listener rendering applications. - Tools for simulating advanced reverberation models (e.g., convolution-based and hybrid approaches). - Hearing loss and hearing aid simulations.

3.1.4 Usage

BRT Applications are ideal for: - Psychoacoustic experiments requiring dynamic spatial configurations. - Integration into existing experimental setups with OSC-controlled workflows. - Real-time virtual acoustics rendering for VR/AR applications and wearable devices.

These applications are designed to minimize the coding effort required to deploy sophisticated audio rendering capabilities, making them accessible to a wide range of users.

For more information about the BRT Applications and their functionalities, visit the BRT GitHub Repository.

3.2 BeRTA Rendeder

3.2.1 BeRTA Renderer

BeRTA Renderer is the flagship application of the BRT Toolbox, providing a high degree of configurability for real-time spatial audio rendering. The application integrates the BRT library and works for Windows x64 (Windows 10 and 11) and MacOS Universal.

Key functionalities include:

- **Real-time Rendering**:
- Dynamic control of sound source locations, listener positions, and other spatial parameters via OSC commands.
- Support for multiple sound sources and HRTF swapping in real time.
- **Configurable Virtual Scenarios**:
- User-defined configurations for different buffer sizes, sample rates, and rendering parameters.
- GUI-based feedback for monitoring application status, including rendering states and OSC commands.
- **Data Saving and Export**:
- Saves all parameters and audio using the proposed annotated audio SOFA convention format for reproducibility.
- Exports simulation data for further analysis or use in auditory modeling.

The application is fully controlled through OSC (Open Sound Control), resulting in a graphical interface focused primarily on reporting rendering status. The GUI consists of terminal-like sections that display critical information, such as rendering parameters (e.g., enabled or disabled features) and the settings of audio sources and the listener. Additional terminals provide detailed logs of OSC commands received and sent, actions executed by BeRTA, and error messages detected by BeRTA's built-in diagnostic system.

HRTF offline process

BeRTA Renderer Interface

3.3 BeRTA GUI

BeRTA GUI is a cross-platform interface (Windows x64 and macOS) for controlling the BeRTA Renderer via OSC. It is a demonstration tool intended to facilitate the exploration of the main BRT features, allows to test (or show) custom audio scenes using a specific configuration of BeRTA Renderer.

Specifically, **BeRTA GUI** enables seamless setup and management of the renderer, including launching, relaunching with custom settings, and loading essential resources like HRTFs and BRIRs. The GUI provides real-time control over sound sources, listener movement, and rendering features while visually displaying the scene's layout.

3.3.1 The key functionalities include:

- **Setting BeRTA Renderer**:
- Launching BeRTA GUI Will launch BeRTA Renderer and connect both.
- BeRTA GUI can relaunch BeRTA Renderer with different custom setting files.
- Sending OSC commands to load all type of resources (HRTFs, BRIRs, Directivities, Near-field compensations).
- **Real Time Control**:
- Sending commands to play, pause, stop or mute sound sources.
- Sending commands to set line-in sources.
- Sending commands to move listener and source (6DoF)
- Sending commands to switch on the fly rendering characteristics in BeRTA Render (as HRTF of near-field compensation)
- **Scene display**:
- Showing graphically the position and orientation of sources with respect to the listener.
- Receiving and showing updates on the scene sent to BeRTA Renderer by a third application (e.g. an app running in Unity which updates the Listener position)

HRTF offline process

BeRTA GUI Interface

3.4 OSC Commands

3.4.1 OSC commands defined in BRT

Part of the BRT components is the definition of a set of OSC commands which the BRT Renderer application understands to dynamically configure the virtual scene. This is a complete list of all commands:

OSC CONTROL COMMANDS

- `/control/actionResult` : Retrieve the result of the action performed in another application after receiving an OSC command.
- `/control/bufferFrames` : Query the number of buffered audio frames.
- `/control/connect` : Establish a connection with an IP and port.
- `/control/disconnect` : Terminate the connection and unsubscribe from updates.
- `/control/frameSize` : Request the audio frame size per channel.
- `/control/ping` : Check if BeRTA is listening by sending an echo.
- `/control/sampleRate` : Request the current audio sample rate.
- `/control/version` : Retrieve the current BeRTA version.
- `/control/playCalibration` : Start the calibration process.
- `/control/setCalibration` : Set the calibration values.
- `/control/playCalibrationTest` : Play the calibration test sound at a specified volume.
- `/control/stopCalibrationTest` : Stop the calibration test sound playback.
- `/control/getSoundLevel` : Get the current sound level.
- `/control/setSoundLevelLimit` : Set the sound level limit to a specified volume.
- `/control/soundLevelAlert` : Alert sent when the output sound level exceeds the threshold set in the limiter.

OSC OVERALL COMMANDS

- `/play` : Starts playback of file sources and streaming from input channels.
- `/pause` : Pauses file sources and stops streaming from input channels.
- `/stop` : Stops playback of file sources and streaming from input channels.
- `/removeAllSources` : Removes all sound sources.
- `/playAndRecord` : Records spatialized sound and data during playback.
- `/record` : Records spatialised sound and data without real-time playback.
- `/enableModel` : Enables or disables a model.
- `/modelGain` : Sets the output gain of the model in dB.

RESOURCES COMMANDS

- `/resources/enableWoodworthITD` : Enables or disables the calculation of the ITD using the Woodworth formula.
- `/resources/getBRIRInfo` : Gets information about one of the loaded Binaural Room Impulse Responses (BRIR).
- `/resources/getDirectivityTFInfo` : Gets information about one of the loaded Directivity Transfer Functions.
- `/resources/getHRTFHeadRadius` : Retrieves the head radius of one of the loaded HRTFs.
- `/resources/getHRTFInfo` : Gets information about one of the loaded HRTFs.
- `/resources/getSOSFiltersInfo` : Gets information about one of the loaded sets of Near Field Compensation filters.
- `/resources/loadBRIR` : Loads a new Binaural Room Impulse Response from a SOFA file.
- `/resources/loadDirectivityTF` : Loads a Directivity Transfer Function from a SOFA file.
- `/resources/loadHRTF` : Loads a new HRTF from a SOFA file and assigns an identifier.
- `/resources/loadSOSFilters` : Loads a set of Second Order Section filters from a SOFA file.
- `/resources/removeBRIR` : Removes a Binaural Room Impulse Response from the loaded resources.
- `/resources/removeDirectivityTF` : Removes a Directivity Transfer Function from the loaded resources.
- `/resources/removeHRTF` : Removes an HRTF from the loaded resources.
- `/resources/removeSOSFilters` : Removes a set of Near Field Compensation filters from the loaded resources.
- `/resources/restoreHRTFHeadRadius` : Sets the HRTF head radius to the value stored in the SOFA file.
- `/resources/setHRTFHeadRadius` : Sets the head radius to be stored in the HRTF.

SOURCE MODELS COMMANDS

- `/source/addLineIn` : Adds a new source from an audio input channel.
- `/source/enableDirectivity` : Enable or disable the directivity of a sound source.
- `/source/gain` : Set the gain of a sound source in dB.

- `/source/location` : Set the global location of a sound source.
- `/source/loadSource` : Loads a new sound file and assigns an identifier.
- `/source/loop` : Set loop mode for a sound source.
- `/source/mute` : Mutes a specific sound source.
- `/source/orientation` : Set the orientation (yaw, pitch, roll) of a sound source.
- `/source/pause` : Pauses a specific sound source.
- `/source/play` : Plays a specific sound source.
- `/source/playAndRecord` : Record a file of specified duration with the source's audio and spatial information.
- `/source/record` : Record a file of specified duration with the source's audio and spatial information without real-time playback.
- `/source/removeSource` : Removes a sound source from the system.
- `/source/setDirectivity` : Assign a directivity to a sound source.
- `/source/solo` : Mutes all sources except the specified one.
- `/source/stop` : Stops a specific sound source.
- `/source/unmute` : Unmutes a previously muted sound source.
- `/source/unsolo` : Unmute all sources except the specified one.

LISTENER MODELS COMMANDS

- `/listener/enableInterpolation` : Enable or disable interpolation among HRIRs.
- `/listener/enableITD` : Enable or disable the simulation of Interaural Time Difference (ITD).
- `/listener/enableModel` : Enable or disable a listener model. *(Deprecated, use '/enableModel' instead.)*
- `/listener/enableNearFieldEffect` : Enable or disable Near Field Compensation (NFC) with HRTF.
- `/listener/enableParallaxCorrection` : Enable or disable parallax correction for direction of arrival.
- `/listener/location` : Set the global location of the listener in x, y, z coordinates.
- `/listener/orientation` : Set the orientation of the listener in yaw, pitch, roll.
- `/listener/setAmbisonicsNormalization` : Set the Ambisonics normalization type.
- `/listener/setAmbisonicsOrder` : Set the Ambisonic encoding order.
- `/listener/setBRIR` : Set a Binaural Room Impulse Response (BRIR) for the listener.
- `/listener/setHRTF` : Set a Head-Related Transfer Function (HRTF) for the listener.
- `/listener/setSOSFilters` : Set filters for Near Field Compensation (NFC) after HRTF convolution.

ENVIRONMENT MODELS COMMANDS

- `/environment/enableDirectPath` : Toggle the direct path source in the environment model.
- `/environment/enableModel` : Enable or disable a specified environment model. *(Deprecated, use '/enableModel' instead.)*
- `/environment/enableReverbPath` : Toggle the reverb path sources in the environment model.
- `/environment/setShoeBoxRoom` : Set up a shoebox-shaped room with specified dimensions.
- `/environment/setWallAbsorption` : Set absorption coefficients for a wall.

BINAURAL FILTERS COMMANDS

- `/binauralFilter/setSOSFilter` : Set up sos filter for an specific listener.
- `/binauralFilter/enableModel` : Enable or disable a specified binaural filter. *(Deprecated, use '/enableModel' instead.)*

3.4.2 Control commands

The OSC control commands allow to establish and manage connections with BeRTA over the network or locally. Below are detailed the basic commands for initiating and closing connections, as well as obtaining system and audio configuration information. Each command is designed to provide immediate feedback.

/CONTROL/CONNECT

Open an OSC connection to the indicated IP and port to send the information back. After opening the communicating the following response is sent back: same command with the IP and listening port of BeRTA. Once the connection is stablished, the sender is considered as a suscriber to all the updates which berta sends back.

Syntax

`/control/connect <string ip> <int port>`

`ip` : IP address of the sender which BeRTA will use to send back messages. Use `localhost` if it is running in the same machine

`port` : OSC port in which the sender islistening for reply and update messages. BeRTA will send all replies and update messages to this port.

Return

A message is sent back to the sender indicating IP and listening port of BeRTA: `/control/connect <string ip> <int port>` . Although this information is already known by the sender, this is away of acknowdeging that the connection is stablished and the sender is subscribed to updates.

Example

BeRTA receives: `/control/connect localhost 10011` .

BeRTA sends back to the sender: `/control/connect localhost 10017` .

/CONTROL/DISCONNECT

Stops the communication through which BeRTA sent information back. This is equivalent to unsibscribe the sender from updates sent back by BrRTA. When this message is received the same message is sent back.

Syntax

`/control/disconnect`

Return

An echo is sent back to the sender to confirm the reception of the command: `/control/disconnect` .

Example

BeRTA receives: `/control/disconnect` .

BeRTA sends back to the sender: `/control/disconnect` .

/CONTROL/PING

Responds with the same message to indicate that BeRTA is listening to the external app.

Syntax

`/control/ping`

Return

An echo is sent back to the sender to confirm the reception of the command: `/control/ping` .

Example

BeRTA receives: `/control/ping` .

BeRTA sends back to the sender: `/control/ping` .

/CONTROL/VERSION

Responds with a message to indicate the version of BeRTA.

Syntax

`/control/version`

Return

An echo is sent back to the sender including the version of BeRTA: `/control/version <string version>` .

Example

BeRTA receives: `/control/version` .

BeRTA sends back to the sender: `/control/version BeRTA-Renderer v3.0.0` .

/CONTROL/SAMPLERATE

Asks for the sample rate which BeRTA is using. This parameter is defined in the used settings file.

Syntax

`/control/sampleRate`

Return

An echo is sent back to the sender including the sample rate: `/control/sampleRate <int sample_rate>` .

`sample_rate` : Sample rate used by BeRTA in samples per second.

Example

BeRTA receives: `/control/sampleRate`

BeRTA sends back to the sender: `/control/sampleRate 48000`.

/CONTROL/FRAMESIZE

Asks for the size of the audio frame which BeRTA is using. This parameter is defined in the used settings file.

Syntax

`/control/frameSize`

Return

An echo is sent back to the sender including the frame size: `/control/frameSize <int frame_size>`.

`frame_size` : size of the audio frame in bytes per channel (BeRTA produces a stereo output for each listener. This parameter indicated the size of the frame of only one channel).

Example

BeRTA receives: `/control/frameSize`.

BeRTA sends back to the sender: `/control/frameSize 512`.

/CONTROL/BUFFERFRAMES

Asks for the number of audio frames that are stored in the buffer to prevent audio dropout or system failure due to frame underrun. Since the operating system is not real-time, there can be instances where it prioritizes other tasks, potentially delaying the provision of a new audio frame. By setting an adequate buffer size, the system ensures continuous audio playback, even if the operating system momentarily allocates resources to higher-priority processes. This buffer acts as a safeguard against timing inconsistencies, allowing the audio processing thread to retrieve preloaded frames, maintaining uninterrupted audio output. However, as a trade-off, increasing the number of buffered frames results in higher system latency. To minimize latency, this parameter should be set to 0, which eliminates buffering but requires the system to deliver new audio frames in real time without delay. This parameter is defined in the used settings file.

Syntax

`/control/bufferFrames`

Return

An echo is sent back to the sender including the number of frames buffered: `/control/bufferFrames <int buffer_frames>`.

`buffer_frames` : number of buffered frames.

Example

BeRTA receives: `/control/bufferFrames`.

BeRTA sends back to the sender: `/control/bufferFrames 2`.

/CONTROL/ACTIONRESULT

Command received when an action has been carried out from the application (triggered by a previously received OSC command). It informs about the outcome of the executed action.

Syntax

`/control/actionResult <string actionCommand> <string id> <bool success> <string description>`

Example

BeRTA receives: `/control/actionResult /resources/removeHRTF HRTF1 true HRTF HRTF1 removed`

Calibration

/CONTROL/PLAYCALIBRATION

Available from BeRTA v3.4.0

Start the calibration process by playing an audio file at the dBFS volume specified in the parameter. For more details, refer to the calibration section.

Syntax

`/control/playCalibration <float leveldBFS> <int channelNumber>`

`leveldBFS` : The signal level at which the calibration audio was played, measured in dBFS.

`channelNumber` : *Available from BeRTA v3.7.0* This indicates the output channels of the audio interface to which the command applies. This value must always be an even number. For example, if channel 0 is selected, channels 0–1 are affected; if 2 is selected, channels 2–3 are affected. If no value is specified, the output channels of the first configured listener are used instead.

Return

`/control/actionResult /control/playCalibration <string actionName> <boolean success> <string description>`.

The return confirmation refers to the action `calibration`, indicating `success=true` if the calibration has been successfully performed and `success=false` if not. In both cases a `description` is added to give more details.

Example

BeRTA receives: `/control/playCalibration -40 0`

BeRTA sends back to the sender: `/control/actionResult /control/playCalibration calibration true "Calibration sound played"`.

/CONTROL/SETCALIBRATION

Available from BeRTA v3.4.0

Set the calibration values using the dBFS playback volume and the dBSPL output level measured in the headphones, both specified in the parameters. For more details, refer to the calibration section.

Syntax

`/control/setCalibration <float leveldBFS> <float leveldBSPL> <int channelNumber>`

`leveldBFS`: The signal level at which the calibration audio was played, measured in dBFS.

`leveldBSPL`: The output sound level measured at the headphones (expressed in dBSPL) when the system is calibrated and delivering the dBFS specified in the leveldBFS parameter.

`channelNumber`: *Available from BeRTA v3.7.0* This indicates the output channels of the audio interface to which the command applies. This value must always be an even number. For example, if channel 0 is selected, channels 0–1 are affected; if 2 is selected, channels 2–3 are affected. If no value is specified, the output channels of the first configured listener are used instead.

Return

`/control/actionResult /control/setCalibration <string actionName> <boolean success> <string description>`.

The return confirmation refers to the action `calibrationDone`, indicating `success=true` if the calibration has been successfully performed and `success=false` if not. In both cases a `description` is added to give more details.

Example

BeRTA receives: `/control/setCalibration -40 60 0`

BeRTA sends back to the sender: `/control/actionResult /control/setCalibration calibrationDone true "Calibration done with -40 dB FS and 60 dB SPL."`.

/CONTROL/PLAYCALIBRATIONTEST

Available from BeRTA v3.4.0

Play the calibration test sound at a volume adjusted to match the dBSPL specified in the parameter, allowing verification with the sound level meter. See the calibration section for more details.

Syntax

`/control/playCalibrationTest  <float leveldBSPL> <int channelNumber>`

`leveldBSPL`: The signal level (in dBSPL) measured at the headphones when reproducing the calibration audio.

`channelNumber`: *Available from BeRTA v3.7.0* This indicates the output channels of the audio interface to which the command applies. This value must always be an even number. For example, if channel 0 is selected, channels 0–1 are affected; if 2 is selected, channels 2–3 are affected. If no value is specified, the output channels of the first configured listener are used instead.

Return

`/control/actionResult /control/playCalibrationTest <string actionName> <boolean success> <string description>`.

The return confirmation refers to the action `calibrationTest`, indicating `success=true` if the calibration has been successfully performed and `success=false` if not. In both cases a `description` is added to give more details.

Example

BeRTA receives: `/control/playCalibrationTest 80 2`

BeRTA sends back to the sender: `/control/actionResult /control/playCalibrationTest calibrationTest false "ERROR BeRTA has not been calibrated yet"`.

/CONTROL/STOPCALIBRATIONTEST

Available from BeRTA v3.4.0

Stop the calibration test sound playback.

Syntax

`/control/stopCalibrationTest <int channelNumber>`

`channelNumber`: *Available from BeRTA v3.7.0* This indicates the output channels of the audio interface to which the command applies. This value must always be an even number. For example, if channel 0 is selected, channels 0–1 are affected; if 2 is selected, channels 2–3 are affected. If no value is specified, the output channels of the first configured listener are used instead.

Return

`/control/actionResult /control/stopCalibrationTest <string actionName> <boolean success> <string description>`.

The return confirmation refers to the action `calibrationTest`, indicating `success=true` if the stop of the calibration has been successfully performed and `success=false` if not. In both cases a `description` is added to give more details.

Example

BeRTA receives: `/control/stopCalibrationTest 0`

BeRTA sends back to the sender: `/control/actionResult /control/stopCalibrationTest calibrationTest true "Calibration test stopped."`.

/CONTROL/GETSOUNDLEVEL

Available from BeRTA v3.4.0

Get the current sound level in dBSPL for left and right channel. For more details, refer to the calibration section.

Syntax

`/control/getSoundLevel <string listener_id>`

`listener_id`: identifier assigned to the listener.

Return

`/resources/getSoundLevel <string listener_id> <float leftChannelSoundLevel> <float rightChannelSoundLevel>`

The return confirmation refers to the `listener_id`, indicating the sound level for the left (`leftChannelSoundLevel`) and the right channel (`rightChannelSoundLevel`).

Example

BeRTA receives: `/control/getSoundLevel`

BeRTA sends: `/control/getSoundLevel 60 62`

Safety Limiter

/CONTROL/SETSOUNDLEVELLIMIT

Available from BeRTA v3.4.0

Set the sound level limit to the dBSPL value specified in the parameter. See the safety limitter for more details.

Syntax

`/control/setSoundLevelLimit <float levelLimitdBSPL> <int channelNumber>`

`levelLimitdBSPL`: The maximum allowable sound level, in dBSPL, used by the limiter to ensure safe listening levels for the user.

`channelNumber`: *Available from BeRTA v3.7.0* This indicates the output channels of the audio interface to which the command applies. This value must always be an even number. For example, if channel 0 is selected, channels 0–1 are affected; if 2 is selected, channels 2–3 are affected. If no value is specified, the output channels of the first configured listener are used instead.

Return

`/control/actionResult /control/setSoundLevelLimit <string actionName> <boolean success> <string description>`.

The return confirmation refers to the action `calibrationTest`, indicating `success=true` if the set has been successfully performed and `success=false` if not. In both cases a `description` is added to give more details.

Example

BeRTA receives: `/control/setSoundLevelLimit 100 0`

BeRTA sends back to the sender: `/control/actionResult /control/setSoundLevelLimit calibrationTest false "ERROR Sound level limit has not been set becasuse BeRTA has not been calibrated."`.

/CONTROL/SOUNDLEVELALERT

Available from BeRTA v3.4.0

Sent when the output sound level exceeds the threshold set in the sound level limiter. This is a send-only command from BeRTA. It does not receive any parameters but triggers an alert when the limit is surpassed. See the safety limitter for more details.

Syntax

`/control/soundLevelAlert`

Example

BeRTA sends: `/control/soundLevelAlert 60 62`

3.4.3 Overall commands

These commands affects all sound sources and therefore, they don't need the `/source` prefix.

/PLAY

Starts playing back all sources comming from audio files and starts streaming all sources comming form input channels.

Syntax

`/play`

Return

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/play`.

/STOP

Stops playing back all sources comming from audio files and also stops streaming all sources comming form input channels.

Syntax

`/stop`

Return

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/stop`.

/PAUSE

Pauses all sources comming from audio files and stops streaming all sources comming form input channels. While file sources will resume with a later play at the same point they were paused, for sources connected to an input channel the effect of `/pause` is identical to the effect of `/stop`.

Syntax

`/pause`

Return

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/pause`.

/REMOVEALLSOURCES

Removes all sources.

Syntax

`/removeAllSources`

Return

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/removeAllSources`.

/PLAYANDRECORD

Records a file of the specified duration with spatialised sound (wav) and other data (mat) corresponding to the playback of all sources from the beginning. If the type is mat, the produced file will follow the structure of the AnnotatedReceiverAudio SOFA convention. Before starting the recording, all sources are stopped and played back since the begining.

Syntax

`/playAndRecord <String filename> <String type> <float time>`

`filename`: indicates the name of the file and must include the path, either relative or absolute. If a relative path is used it will be calculated from the data folder that can be found in the same folder as the BeRTA executable. The extension will be added by the application if necessary. If there is a file with the same name, it’ll not be overwritten, an ordinal number will be added to the end of the new file name.

`type`: indicates the extension of the file: wav (not implemented yet) or a mat (matlab binary data container).

`time`: indicates the duration of the recording in seconds. If time is -1 the recording will finish automatically when the source stops.

Return

`/control/actionResult /playAndRecord <string filename> <boolean success> <string description>`.

The return confirmation refers to the `filename`, indicating `success=true` if the action has been successfully performed and `set=false` if not. In both cases a `description` is added to give more details.

It sends an echo to all subscribers excepting the sender: `/playAndRecord <String filename> <String type> <float time>`

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/playAndRecord c:/tmp/file.mat mat 10`

When the recording is finished (in this example after 10 seconds), BeRTA sends back to all subscribers: `/control/actionResult /playAndRecord c:/tmp/file.mat true Recording completed. File saved : c:/tmp/file.mat` or `/control/actionResult  /playAndRecord c:/tmp/file.mat false ERROR:Recording failed. File could not be created.`

/RECORD

Available from BeRTA v3.7.0

Generates a file of the specified duration with spatialised sound (wav) and other data (mat) by processing all sources from the beginning *without real-time playback*. This allows the operation to be executed significantly faster. If the type is mat, the produced file will follow the structure of the AnnotatedReceiverAudio SOFA convention. All sources are stopped and processed from the beginning before the recording starts.

Syntax

`/record <String filename> <String type> <float time>`

`filename`: indicates the name of the file and must include the path, either relative or absolute. If a relative path is used it will be calculated from the data folder that can be found in the same folder as the BeRTA executable. The extension will be added by the application if necessary. If there is a file with the same name, it'll not be overwritten—an ordinal number will be added to the end of the new file name.

`type`: indicates the extension of the file: wav (not implemented yet) or mat (MATLAB binary data container).

`time`: indicates the duration of the recording in seconds. **This value must be greater than 0**. Unlike `/playAndRecord`, the special value `-1` is *not* allowed.

Return

`/control/actionResult /record <string filename> <boolean success> <string description>`.

The return confirmation refers to the `filename`, indicating `success=true` if the action has been successfully performed and `success=false` if not. In both cases a `description` is added to give more details.

It sends an echo to all subscribers except the sender: `/record <String filename> <String type> <float time>`

Example

BeRTA receives and echoes back to all subscribers except the sender: `/record c:/tmp/file.mat mat 10`

When the processing is finished (in this example after 10 seconds), BeRTA sends back to all subscribers:
`/control/actionResult /record c:/tmp/file.mat true Recording completed. File saved : c:/tmp/file.mat`
or
`/control/actionResult /record c:/tmp/file.mat false ERROR:Recording failed. File could not be created.`

/ENABLEMODEL

Available from BeRTA v3.6.0

This command enables or disables a model. This function is implemented in all listener, environment models and in all binaural filters. The model to be enabled or disabled is identified by an identifier defined in the configuration file used. When a model is deactivated it does not process the input signal and provides:

- *Silence* on its output in the case of listener models.
- Let the signal *pass through unaltered* in the case of environment models and binaural filters.

Syntax

`/enableModel <string model_id> <boolean enable>`

`model_id`: identifier assigned to the model.

`enable`: If true (1), enables the model. If false (0), the model is disabled and its output will be silent.

Return

`/control/actionResult /enableModel <string model_id> <bool enable> <string description>`

The return confirmation refers to the `model_id`, indicating `enable=true` if the model has been enabled and `enable=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/enableModel DirectPath true`

BeRTA sends back to the sender: `/control/actionResult /enableModel DirectPath true "Listener model DirectPath enabled."`

/MODELGAIN

Available from BeRTA v3.6.0

This command sets the output gain of the model in dB. This is a gain that is applied to the model output after processing. A gain of 0 dB, default value, indicates that no gain is applied to the output samples after processing. The model to be enabled or disabled is identified by an identifier defined in the used settings file.

Syntax

`/modelGain <string model_id> <float gain>`

`model_id`: identifier assigned to the model.

`gain`: A floating value, expressed in dB, representing the gain. The model output is multiplied by \(10^{gain / 20}\).

Return

`/control/actionResult /modelGain <string model_id> <bool setGain> <string description>`

The return confirmation refers to the `model_id`, indicating `setGain=true` if the action has been successfully performed and setGain=false if not. In both cases a description is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/modelGain DirectPath -3`

BeRTA sends back to the sender: `/control/actionResult /modelGain DirectPath true "Listener model DirectPath gain updated to -3dB."`

3.4.4 Resources commands

These commands are responsible for managing the resources used in the virtual audio scene.

/RESOURCES/LOADHRTF

Loads a new HRTF in a SOFA file from the specified path and assign an identifier to it. If there is already another HRTF with the same identifier, it is substituted by the new one. The SOFA file must use FIR or FIR-E as data type but can follow any convention using these data types. When loading, the HRTF is spatially resampled to fit the HRTF grids. This way the real time processor can get fast enough the three HRIRs to be interpolated and convolved.

Syntax

`/resources/loadHRTF <string HRTF_id> <string HRTF_SOFAFile_path> <float spatialResolution>`

`HRTF_id` : Identifier to be assigned to the HRTF for later references to it. If there is already another HRTF with the same identifier, it is substituted by the new one. Otherwise, a new HRTF is created.

`HRTF_SOFAFile_path` : Specifies a SOFA file containing an HRTF. it can be either relative or absolute. If a relative path is used it will be calculated from the data folder that can be found in the same folder as the BeRTA executable.

`spatialResolution` : This parameter indicates the interpolation step in the horizontal plane of zero elevation. To learn more about this grid and the interpolation mechanism, please refer to the HRTF grids.

Return

`/control/actionResult /resources/loadHRTF <string HRTF_id> <boolean loaded> <string description>`

The return confirmation refers to the `HRTF_id` ,indicating `loaded=true` if the HTYF is successfuly loaded and `loaded=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/resources/loadHRTF HRTF1 c:/tmp/hrtf.sofa 5`

BeRTA sends back to the sender: `/control/actionResult /resources/loadHRTF HRTF1 true HRTF HRTF1 loaded successfully, with a spatial resolution of 5, from the file: c:/tmp/hrtf.sofa` or `/control/actionResult /resources/loadHRTF HRTF1 false HRTF sofa file couldn't be loaded from HRTF1;` .

/RESOURCES/REMOVEHRTF

Removes an HRTF from the loaded resources.

Syntax

`/resources/removeHRTF <string HRTF_id>`

`HRTF_id` : Identifier of the HRTF to be removed.

Return

`/control/actionResult /resources/loadHRTF <string HRTF_id> <bool removed> <string description>`

The return confirmation refers to the `HRTF_id` ,indicating `removed=true` if the HRTF is successfuly removed and `removed=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/resources/removeHRTF HRTF1`

BeRTA sends back to all subscribers: `/control/actionResult /resources/removeHRTF HRTF1 true HRTF HRTF1 removed` or `/control/actionResult /resources/removeHRTF HRTF1 false error: HRTF not found in the list`

/RESOURCES/GETHRTFINFO

Gets some information about one of the loaded HRTFs. this information consists of the filename and the spatial resolution used to resample it in the HRTF grid.

Syntax

`/resources/getHRTFInfo <string HRTF_id>`

`HRTF_id` : Identifier of the HRTF of which we are requesting information.

Return

`/resources/getHRTFInfo <string HRTF_id> <string HRTF_SOFAFile> <float spatialResolution>`

The return message is sent back to the sender and refers to the `HRTF_id`, indicating the name of the SOFA file (`HRTF_SOFAFile`) from which the HRTF was loaded, as well as the `spatialResolution` used to resample it when loaded.

Example

BeRTA receives: `/resources/getHRTFInfo HRTF1`

BeRTA sends back: `/resources/getHRTFInfo HRTF1 hrtf.sofa 5`

/RESOURCES/SETHRTFHEADRADIUS

Sets the head radius to be stored into the HRTF. This value affects: (1) the position of the ears used in the parallax correction when getting HRIRs; and (2) the calculation of the ITD when using the simulation based on the Woodworth formula whcih is activated or deactivated by the `/resources/enableWoodworthITD` command. If this command is not received, the position of receivers in the HRTF SOFA file is used as the default value. See `/resources/restoreHRTFHeadRadius` as a related command.

Syntax

`/resources/setHRTFHeadRadius <string HRTF_id> <float headRadius>`

`HRTF_id` : Identifier of the HRTF of which we are setting the new head radius.

`headRadius` : New head radius in meters. Only positives values are accepted.

Return

`/control/actionResult /resources/setHRTFHeadRadius <string HRTF_id> <bool set> <string description>`

The return confirmation refers to the `HRTF_id`, indicating `set=true` if the HRTF is successfuly set and `set=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/resources/setHRTFHeadRadius HRTF1 0.09`

BeRTA sends back to the sender: `/control/actionResult /resources/setHRTFHeadRadius HRTF1 true success`

/RESOURCES/GETHRTFHEADRADIUS

Gets the head radius of one of the HRTFs loaded as a resource.

Syntax

`/resources/getHRTFHeadRadius <string HRTF_id>`

`HRTF_id` : Identifier of the HRTF of which we are requesting the head radius.

Return

A message is sent back to the sender `/resources/getHRTFHeadRadius <string HRTF_id> <float headRadius>`, indicating the `headRadius` of the HRTF `HRTF_id`.

Example

BeRTA receives: `/resources/getHRTFHeadRadius HRTF1`

BeRTA sends back to the sender: `/resources/getHRTFHeadRadius HRTF1 0.09`

/RESOURCES/RESTOREHRTFHEADRADIUS

Sets the head radius of the HRTF to the one stored in the SOFA file as the position of receivers. This value affects: (1) the position of the ears used in the parallax correction when getting HRIRs; and (2) the calculation of the ITD when using the simulation based on the Woodworth formula whcih is activated or deactivated by the `/resources/enableWoodworthITD` command.

Syntax

`/resources/restoreHRTFHeadRadius <string HRTF_id>`

`HRTF_id` : Identifier of the HRTF for which we are setting the head radius.

Return

`/control/actionResult /resources/restoreHRTFHeadRadius <string HRTF_id> <bool restored> <string description>`

The return confirmation refers to the `HRTF_id`, indicating `restored=true` if the HRTF is successfuly set and `restored=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/resources/restoreHRTFHeadRadius HRTF1`

BeRTA sends back to the sender: `/control/actionResult /resources/restoreHRTFHeadRadius HRTF1 true success`

/RESOURCES/ENABLEWOODWORTHITD

Enables or disables the calculation of the ITD based on the Woodworth formula, considering head radius, or it is extracted from the delays in the HRTF SOFA file. The Woodworth formula needs the head radius set by commands `/resources/setHRTFHeadRadius` and `/resources/restoreHRTFHeadRadius`. By default, the use of the Woodworth formula is disabled.

Syntax

`/resources/enableWoodworthITD <string HRTF_id> <boolean enable>`

`HRTF_id` : Identifier of the HRTF for which we are setting the head radius.

`enable` : If enable is true (1), the ITD calculation is performed using the Woodworth formula. If enable is false (0), the ITD is simulated from the delays read from the SOFA file.

Return

`/control/actionResult /resources/enableWoodworthITD <string HRTF_id> <bool enabled> <string description>`

The return confirmation refers to the `HRTF_id`, indicating `enabled=true` if the HRTF is successfuly set and `enabled=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/resources/enableWoodworthITD HRTF1 true`

BeRTA sends back to the sender: `/control/actionResult /resources/enableWoodworthITD HRTF1 true success`

/RESOURCES/LOADDIRECTIVITYTF

Loads a new Directivity Transfer Function in a SOFA file from the specified path and assign an identifier to it. If there is already another Directivity with the same identifier, it is substituted by the new one. The SOFA file must use TF as data type with the same number of frequency bins as the frame size configured in BeRTA. When loading, the DirectivityTF is spatially resampled to fit the BRT grids. This way the real time processor can get fast enough the three Directivities to be interpolated and convolved.

Syntax

`/resources/loadDirectivityTF <string directivityTF_id> <string directivityTF_SOFAFile_path> <float spatialResolution>`

`directivityTF_id` : Identifier to be assigned to the directivityTF for later references to it. If there is already another directivityTF with the same identifier, it is substituted by the new one. Otherwise, a new directivityTF is created.

`directivityTF_SOFAFile_path` : Specifies a SOFA file containing a directivityTF. it can be either relative or absolute. If a relative path is used it will be calculated from the data folder that can be found in the same folder as the BeRTA executable.

`spatialResolution` : This parameter indicates the interpolation step in the horizontal plane of zero elevation. To learn more about this grid and the interpolation mechanism, please refer to the BRT grids.

Return

`/control/actionResult /resources/loadDirectivityTF <string directivityTF_id> <boolean loaded> <string description>`

The return confirmation refers to the `directivityTF_id`, indicating `loaded=true` if the directivityTF is successfuly loaded and `loaded=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/resources/loadDirectivityTF DirectivityTF1 c:/tmp/directivityTF.sofa 15`

BeRTA sends back to the sender: `/control/actionResult /resources/loadDirectivityTF DirectivityTF1 true Directivity directivityTF1 loaded` or `/control/actionResult /resources/loadDirectivityTF DirectivityTF1 false Directivity sofa file couldn't be loaded from c:/tmp/directivityTF.sofa`.

/RESOURCES/REMOVEDIRECTIVITYTF

Removes a Directivity Transfer Function from the loaded resources.

Syntax

`/resources/removeDirectivityTF <string directivityTF_id>`

`directivityTF_id` : Identifier of the directivityTF to be removed.

Return

`/control/actionResult /resources/removeDirectivityTF <string directivityTF_id> <bool removed>`

The return confirmation refers to the `directivityTF_id`, indicating `removed=true` if the directivityTF is successfuly removed and `removed=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/resources/removeDirectivityTF directivityTF1`

BeRTA sends back to the sender: `/control/actionResult /resources/removeDirectivityTF directivityTF1 true`

`DirectivityTF directivityTF1 removed` or `/control/actionResult /resources/removeDirectivityTF directivityTF1 false ERROR deleting DirectivityTF.  directivityTF1 not found in the list`

/RESOURCES/GETDIRECTIVITYTFINFO

Gets some information about one of the loaded Directivity Transfer Functions. This information consists of the filename and the spatial resolution used to resample it in the BRT grid.

Syntax

```
/resources/getDirectivityTFInfo <string directivityTF_id>
```

`directivityTF_id` : Identifier of the DirectivityTF of which we are requesting information.

Return

```
/resources/getDirectivityTFInfo <string directivityTF_id> <string directivityTF_SOFAFile> <float spatialResolution>
```

The return message is sent back to the sender and refers to the `directivityTF_id`, indicating the name of the SOFA file (`directivityTF_SOFAFile`) from which the directivity was loaded, as well as the `spatialResolution` used to resample it when loaded.

Example

BeRTA receives: `/resources/getDirectivityTFInfo directivityTF1`

BeRTA sends back: `/resources/getDirectivityTFInfo directivityTF1 directivityTF.sofa 15`

/RESOURCES/LOADSOSFILTERS

Loads a new set of Second Order Section (SOS) filters in a SOFA file from the specified path and assign an identifier to it. If there is already another set of SOS filters with the same identifier, it is substituted by the new one. The SOFA file must use SOS as data type.

Syntax

```
/resources/loadSOSFilters <string SOSFilters_id> <string SOSFilters_SOFAFile_path>
```

`SOSFilters_id` : Identifier to be assigned to the NFC Filters for later references to it. If there is already another set of NFC Filters with the same identifier, it is substituted by the new one. Otherwise, a new set of NFC Filters is created.

`SOSFilters_SOFAFile_path` : Specifies a SOFA file containing a set of NFC Filters. It can be either relative or absolute. If a relative path is used it will be calculated from the data folder that can be found in the same folder as the BeRTA executable.

Return

```
/control/actionResult /resources/loadSOSFilters <string SOSFilters_id> <boolean loaded>
```

The return confirmation refers to the `SOSFilters_id`, indicating `loaded=true` if SOS NFC Filters are successfuly loaded and `loaded=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/resources/loadSOSFilters SOSF1 c:/tmp/SOSilters.sofa`

BeRTA sends back to the sender: `/control/actionResult /resources/loadSOSFilters NFCF1 true SOS filter SOSF1 loaded successfully, from the file: c:/tmp/SOSilters.sofa` or `/resources/loadSOSFilters SOSF1 false ERROR deleting SOS filter.  SOSF1 not found in the list`.

/RESOURCES/REMOVESOSFILTERS

Removes a set of Near Field Compensation (NFC) filters from the loaded resources.

Syntax

```
/resources/removeSOSFilters <string SOSFilters_id>
```

`SOSFilters_id` : Identifier of the NFC Filters to be removed.

Return

```
/control/actionResult /resources/removeSOSFilters <string SOSFilters_id> <bool removed> <string description>
```

The return confirmation refers to the `SOSFilters_id`, indicating `removed=true` if the NFC Filters are successfuly removed and `removed=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/resources/removeSOSFilters SOSF1`

BeRTA sends back to the sender: `/control/actionResult /resources/removeSOSFilters SOSF1 true SOSF1 removed from the list` or `/control/actionResult /resources/removeSOSFilters SOSF1 false ERROR deleting SOS Filters.  SOSF1 not found in the SOS Filters list`

/RESOURCES/GETSOSFILTERSINFO

Gets some information about one of the loaded set of Near Field Compensation (NFC) filters. This information consists of the name of the file from which it was loaded.

Syntax

`/resources/getSOSFiltersInfo <string SOSFilters_id>`

`SOSFilters_id` : Identifier of the NFC Filters of which we are requesting information.

Return

`/resources/getSOSFiltersInfo <string SOSFilters_id> <string SOSFilters_SOFAFile>`

The return message is sent back to the sender and refers to the `SOSFilters_id` , indicating the name of the SOFA file (`SOSFilters_SOFAFile`) from which the filters were loaded.

Example

BeRTA receives: `/resources/getSOSFiltersInfo NFCF1`

BeRTA sends back: `/resources/getSOSFiltersInfo NFCF1 NFCFilters.sofa`

/RESOURCES/LOADBRIR

Parameters renamed from BeRTA v3.6.0

Loads a new Binaural Room Impulse Response (BRIR) from a SOFA file at the specified path and assign an identifier to it. If there is already another BRIR with the same identifier, it is substituted by the new one. The SOFA file must use FIR or FIR-E as data type but can follow any convention using these data types. When loading, the HRTF is spatially resampled to fit the BRT grids.

Syntax

`/resources/loadBRIR <string BRIR_id> <string BRIR_SOFAFile_path> <float spatialResolution> <float fadeInBegin> <float riseTime> <float fadeOutCutoff> <float fallTime>`

`BRIR_id` : Identifier to be assigned to the BRIR for later references to it. If there is already another BRIR with the same identifier, it is substituted by the new one. Otherwise, a new BRIR is created.

`BRIR_SOFAFile_path` : Specifies a SOFA file containing a BRIR. It can be either relative or absolute. If a relative path is used it will be calculated from the data folder that can be found in the same folder as the BeRTA executable.

`spatialResolution` : This parameter indicates the interpolation step in the horizontal plane of zero elevation. To learn more about the interpolation mechanism, please refer to BRT grids.

The loaded BRIR can be windowed. For this purpose, a fade-in and fade-out mechanism has been enabled using raised-cosines. The parameter `fadeInBegin` indicates the midpoint (50%) of the rising slope of the window, in seconds, the `risetime` indicates the rising time, also in seconds. Analogously, the parameter `fadeOutCutoff` indicates the midpoint (50%) of the falling slope of the window, and `fallTime` indicates the falling time, all in seconds.

Return

`/control/actionResult /resources/loadBRIR <string BRIR_id> <boolean loaded>`

The return confirmation refers to the `BRIR_id` , indicating `loaded=true` if the BRIR is successfuly loaded and `loaded=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/resources/loadBRIR BRIR1 c:/tmp/BRIR.sofa 15 0.005 0.001 0.5 0.01`

BeRTA sends back to the sender: `/control/actionResult /resources/loadBRIR BRIR1 true BRIR BRIR1 loaded successfully, with a spatial resolution of 15, from the file: c:/tmp/BRIR.sofa` or `/control/actionResult /resources/loadBRIR BRIR1 false BRIR sofa file couldn't be loaded from c:/tmp/BRIR.sofa` .

/RESOURCES/REMOVEBRIR

Removes a Binaural Room Impulse Response (BRIR) from the loaded resources.

Syntax

`/resources/removeBRIR <string BRIR_id>`

`BRIR_id` : Identifier of the BRIR to be removed.

Return

`/control/actionResult /resources/removeBRIR <string BRIR_id> <bool removed> <string description>`

The return confirmation refers to the `BRIR_id` , indicating `removed=true` if the BRIR successfuly removed and `removed=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/resources/removeBRIR BRIR1`

BeRTA sends back to all subscribers: `/control/actionResult /resources/removeBRIR BRIR1 true BRIR BRIR1 removed` or just to the sender: `/control/actionResult /resources/removeBRIR BRIR1 false ERROR deleting BRIR.  BRIR1 not found in the list`

/RESOURCES/GETBRIRINFO

Gets some information about one of the loaded Binaural Room Impulse Response (BRIR). This information consists of the name of the file from which it was loaded, the spatial resolution and all the parameters of the window with which the BRIR is windowed.

Syntax

```
/resources/getBRIRInfo <string BRIR_id>
```

`BRIR_id` : Identifier of the BRIR of which we are requesting information.

Return

```
/resources/getBRIRInfo <string BRIR_id> <string BRIR_SOFAFile> <float spatialResolution> <float fadeInWindowThreshold> <float fadeInWindowRiseTime> <float fadeOutWindowThreshold> <float fadeOutWindowFallTime>
```

The return message is sent back to the sender and refers to the `BRIR_id`, indicating the name of the SOFA file (`BRIR_SOFAFile`) from which the filters were loaded, the spatial resolution at which it is resampled and all the parameters defining the windowing: `fadeInWindowThreshold` and `fadeInWindowRiseTime` for the fade-in rising slope; and `fadeOutWindowThreshold` and `fadeOutWindowFallTime>` for the fade-out falling slope.

Example

BeRTA receives: `/resources/getBRIRInfo BRIR1`

BeRTA sends back: `/resources/getBRIRInfo BRIR1 BRIR.sofa 15 0.005 0.001 0.5 0.01`

3.4.5 Source commands

These commands provide a comprehensive set of tools for managing and controlling audio sources within the virtual scene. These commands allow for the addition, manipulation, and control of sound sources, including their playback, position, orientation, and various audio properties.

/SOURCE/LOADSOURCE

Adds a new source loading a sound file (wav, mp3 and aif formats supported). Adding a new source does not imply playing it back. When adding the source an ID string is associated. That ID can be used later to control the source using other OSC commands.

Syntax

`/source/loadSource <string source_id> <string source_path> <string source_model> [<string modelToConnectTo>]`

Specifying a sound file with `source_path`, a new sound file is loaded from the specified path and an identifier `source_id` is assigned to it. If there is already another source with the same identifier, it is substituted by the new one. Otherwise, a new source is created.

`source_model` indicates the source model to be used, possible values are: `OmnidirectionalModel`, `DirectivityModel`. See Source Models for more details.

`modelToConnectTo` is optional and indicates to which listener model this source will be connected. If left blank, the source will be connected to all listener models that have been indicated in the application configuration.

Return

`/control/actionResult /source/loadSource <string source_id> <bool loaded> <string description>` .

The return confirmation refers to the `source_id`, indicating `loaded=true` if the source is successfuly loaded and `loaded=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/loadSource source1 speech.wav`

BeRTA sends back to the sender: `/control/actionResult /source/loadSource source1 true Audio source source1 loop enabled` or `/source/loadSource source1 ERROR: The audio source source1 doesn't exist.`

/SOURCE/ADDLINEIN

Adds a new source getting it from an audio input channel. Adding a new source does not imply playing it back. When adding the source an ID string is associated. That ID can be used later to control the source using other OSC commands.

Syntax

`/source/addLineIn <string source_id> <int linein_channel> <string source_model> [<string modelToConnectTo>]`

`source_id`: identifier assigned to the sound source for further references to it

`linein_channel`: indicates the input channel of the selected audio device to be linked to this sound source.

`source_model` indicates the source model to be used, possible values are: `SimpleModel`, `DirectivityModel`. See Source Models for more details.

`modelToConnectTo` is optional and indicates to which listener model this source will be connected. If left blank, the source will be connected to all listener models that have been indicated in the application configuration.

Return

`/control/actionResult /source/addLineIn <string source_id> <bool loaded> <string description>`

The return confirmation refers to the `source_id`, indicating `loaded=true` if the source is successfuly loaded and `loaded=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/addLineIn source1 1 DirectivityModel`

BeRTA sends back to the sender: `/control/actionResult /source/addLineIn source1 true AUDIO LINE-IN source1 configured succesfully` or `/source/addLineIn source1 false error: not a valid channel`

/SOURCE/REMOVESOURCE

Removes a sound source from the system.

Syntax

`/source/removeSource <string source_id>`

`source_id`: identifier assigned to the sound source for further references to it

Return

`/control/actionResult /source/removeSource <string source_id> <bool removed> <string description>`

The return confirmation refers to the `source_id`, indicating `removed=true` if the source is successfuly removed and `removed=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/remove source1`

BeRTA sends back to the sender : `/control/actionResult /source/remove source1 true Audio source removed: source1` or `/control/actionResult /source/remove source1 false ERROR: The audio source source1 doesn't exist.`

/SOURCE/PLAY

Plays a specific source, starting to send frames to the library from either the audio file or the input channel. In the first case, it starts from the beginning of the file unless the source was previously started and a `/source/pause` command was sent before. In such case it resumes at the point it was paused.

Syntax

`/source/play <string source_id>`

`source_id` : identifier assigned to the sound source for further references to it

Return

`/control/actionResult /source/play <string source_id> <bool played> <string description>`

The return confirmation refers to the `source_id`, indicating `played=true` if the source is successfuly played and `played=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/play source1`

BeRTA sends back to the sender: `/control/actionResult /source/play source1 true Audio source source1 played back`

/SOURCE/STOP

Stops a specific source. In the case of a source comming from a file, this resets the source so that the next `/source/play` command will start at the beginning of the file.

Syntax

`/source/stop <string source_id>`

`source_id` : identifier assigned to the sound source for further references to it

Return

`/control/actionResult /source/stop <string source_id> <bool stopped> <string description>`

The return confirmation refers to the `source_id`, indicating `stopped=true` if the action has been successfully performed and `stopped=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/stop source1`

BeRTA sends back to the sender: `/control/actionResult /source/stop source1 true Audio source source1 removed.`

/SOURCE/PAUSE

Pauses a specific source. In the case of a source comming from a file, the next `/source/play` command will resume, sarting at the same point where it was paused. If the `source_id` does not exist, the command is ignored.

Syntax

`/source/pause <string source_id>`

`sourc e_id` : identifier assigned to the sound source for further references to it

Return

`/control/actionResult /source/pause <string source_id> <bool paused> <string description>`

The return confirmation refers to the `source_id`, indicating `paused=true` if the action has been successfully performed and `paused=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/pause source1`

BeRTA sends back to the sender: `/control/actionResult /source/pause source1 true Audio source source1 paused.`

/SOURCE/MUTE

Mutes a specific source. In the case of a source comming from a file, the application keeps getting the sound stream from the file at the same rate, but silently. The `/source/unmute` command will make the source sound again. If the `source_id` does not exist, the command is ignored.

Syntax

```
/source/mute <string source_id>
```

`source_id` : identifier assigned to the sound source for further references to it

Return

```
/control/actionResult /source/mute <string source_id> <bool muted> <string description>
```

The return confirmation refers to the `source_id`, indicating `muted=true` if the action has been successfully performed and `muted=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/mute source1`

BeRTA sends back to the sender: `/control/actionResult /source/mute source1 true Audio source source1 muted.`

/SOURCE/UNMUTE

Unmutes a specific source. In the case of a source comming from a file previously muted, the application has kept getting the sound stream from the file at the same rate, but silently. The `/source/unmute` command will make the source sound again. If the `source_id` does not exist, the command is ignored.

Syntax

```
/source/unmute <string source_id>
```

`source_id` : identifier assigned to the sound source for further references to it

Return

```
/control/actionResult /source/unmute <string source_id> <bool unmuted> <string description>
```

The return confirmation refers to the `source_id`, indicating `unmuted=true` if the action has been successfully performed and `unmuted=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/unmute source1`

BeRTA sends back to the sender: `/control/actionResult /source/unmute source1 true Audio source source1 unmuted.`

/SOURCE/SOLO

Mutes all sources but the the one indicated by this command. In the case of sources comming from files, the application keeps getting the sound stream from them at the same rate, but silently. The `/source/unsolo` command will make all the sources sound again. Individual commands `/source/unmute` addressed to those sources muted as a consecuence of an `/source/solo` command will have the same effect making them sound again. If the `source_id` does not exist, the command is ignored.

Syntax

```
/source/solo <string source_id>
```

`source_id` : identifier assigned to the sound source for further references to it

Return

```
/control/actionResult /source/solo <string source_id> <bool solo> <string description>
```

The return confirmation refers to the `source_id`, indicating `solo=true` if the action has been successfully performed and `solo=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/solo source1`

BeRTA sends back to the sender: `/control/actionResult /source/solo source1 true Audio source source1 playing solo.`

/SOURCE/UNSOLO

Unmutes all sources but the the one indicated by this command. If the `source_id` does not exist, the command is ignored.

Syntax

`/source/unsolo <string source_id>`

`source_id` : identifier assigned to the sound source for further references to it

Return

`/control/actionResult /source/unsolo <string source_id> <bool unsolo> <string description>`

The return confirmation refers to the `source_id`, indicating `unsolo=true` if the action has been successfully performed and `unsolo=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/unsolo source1`

BeRTA sends back to the sender: `/control/actionResult /source/unsolo source1 true All audio sources unmuted.`

/SOURCE/LOOP

Sets loop mode for the identified source. If loop mode is enabled, when reaching the end, the source will start again. If loop mode is disabled, when reaching the end, the source will stop.

Syntax

`/source/loop <string source_id> <boolean enable>`

`source_id` : identifier assigned to the sound source for further references to it

`enable` : if true (1) loop mode is enabled. If false (0), loop mode is disabled

Return

`/control/actionResult /source/loop <string source_id> <bool loop> <string description>`

The return confirmation refers to the `source_id`, indicating `loop=true` if the action has been successfully performed and `loop=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/loop source1`

BeRTA sends back to the sender: `/control/actionResult /source/loop source1 true Audio source source1 enabled.`

/SOURCE/LOCATION

Set location of source. Position is set in global x,y,z coordinates, expressed in meters. The new location is sent to all remotes except the one has sent the message.

Syntax

`/source/location <string source_id> <float x> <float y> <float z>`

`source_id` : identifier assigned to the sound source for further references to it

`x` : X global coordinate, expressed in metres. As a reference, X axis is positive to the front.

`y` : Y global coordinate, expressed in metres. As a reference, Y axis is positive to the left.

`z` : Z global coordinate, expressed in metres. As a reference, Z axis is positive to up.

Return

Any confirmation message is sent back to the sender.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/location source1 2 0 0`

/SOURCE/ORIENTATION

Set orientation of the source. Orientation is set in egocentric coordinates yaw, pitch and roll, expressed in radians and applied in that order (Yaw, Pitch and Roll). An orinetation of (0, 0, 0) corresponds to be oriented towards positive X. The new orientation is sent to all remotes except the one has sent the message.

Syntax

`/source/orientation <string source_id> <float yaw> <float pitch> <float roll>`

`source_id` : identifier assigned to the sound source for further references to it

`yaw` : Yaw, expressed in radians. Positive to the right.

`pitch` : Pitch, expressed in radians. Positive upwards.

`roll` : Roll, expressed in radians. Positive to the right.

Return

Any confirmation message is sent back to the sender.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/orientation source1 0 0 0`

/SOURCE/GAIN

Sets source gain in dB. 0 dB indicates that the source is kept as it is read from the file or the input channel.

Syntax

`/source/gain <string source_id> <float gain>`

`source_id` : identifier assigned to the sound source for further references to it. If the source doesn't exist, the command is ignored.

`gain` : Gain, expressed in dB. the source, as red from the file or the input channel is multiplied by 10^(gain/20)

Return

`/control/actionResult /source/gain <string source_id> <bool setGain> <string description>`

The return confirmation refers to the `source_id`, indicating `setGain=true` if the action has been successfully performed and `setGain=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/gain source1 -3`

BeRTA sends back to the sender: `/control/actionResult /source/gain source1 true Audio source source1 gain updated to -3 dB`

/SOURCE/ENABLEDIRECTIVITY

Switch on or off the directivity of the source. For directivity to work, a DirectivityTF must first be loaded from a SOFA file (`/resources/loadDirectivityTF`) and then assigned to the source (`/source/setDirectivity`).

Syntax

`/source/enableDirectivity <string source_id> <boolean enable>`

`source_id` : identifier assigned to the sound source. if this the source doesn't exist, the command is ignored

`enable` : if true (1) directivity is enabled. If false (0), directivity is disabled

Return

`/control/actionResult /source/enableDirectivity <string source_id> <bool enableDirectivity> <string description>`

The return confirmation refers to the `source_id`, indicating `enableDirectivity=true` if the action has been successfully performed and `enableDirectivity=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/enableDirectivity source1 true`

BeRTA sends back to the sender: `/control/actionResult /source/enableDirectivity source1 true Audio source source1 directivity enabled`

/SOURCE/SETDIRECTIVITY

Assigns a directivity which has been previously loaded using the command `/resources/loadDirectivityTF`.

Syntax

`/source/setDirectivity <string source_id> <string DirectivityTF_id>`

`source_id` : identifier assigned to the sound source for further references to it

`DirectivityTF_id` : identifier of the directivity which was assigned when using the command `/resources/loadDirectivityTF`

Return

`/control/actionResult /source/setDirectivity <string DirectivityTF_id> <bool setDirectivity> <string description>`

The return confirmation refers to the `source_id`, indicating `setDirectivity=true` if the action has been successfully performed and `setDirectivity=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/setDirectivityTF source1 DirectivityTF1`

BeRTA sends back to the sender: `/control/actionResult /source/setDirectivityTF DirectivityTF1 true DirectivityTF: DirectivityTF1 has been set to SoundSource: source1`

/SOURCE/PLAYANDRECORD

Records a file of the specified duration with the delivered binaural sound and all the spatial information of one source and listener (location and orientation). The sound corresponds to the playback of this source from the beginning. The file includes the structure needed to create a SOFA file with the new AnnotatedReceiverAudio convention

Syntax

`/source/playAndRecord <string source_id> <string filename> <string type> <float seconds>`

`source_id` : identifier of the source to be played back and recorded.

`string filename` : indicates the name of the file and must include the path, either relative or absolute. If a relative path is used it will be calculated from the data folder that can be found in the same folder as the BeRTA executable. The extension will be added by the application if necessary. If there is a file with the same name, it'll not be overwritten, an ordinal number will be added to the end of the new file name.

`type` : indicates the extension of the file: wav (not implemented yet) or a mat (matlab binary data container).

`seconds` : indicates the duration of the recording. If seconds is -1 the recording will finish automatically when the source stops.

Return

`/control/actionResult /source/playAndRecord <string source_id> <bool recorded> <string description>`

The return confirmation refers to the `source_id`, indicating `recorded=true` if the action has been successfully performed and `recorded=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/source/playAndRecord SoundSource1 C:/tmp/testingRecord mat 3`

BeRTA sends back to the sender: `/control/actionResult /source/playAndRecord source1 true "Recording completed. File saved: C:/tmp/testingRecord`

/SOURCE/RECORD

Available from BeRTA v3.7.0

Generates a file of the specified duration with the delivered binaural sound and all the spatial information of one source and listener (location and orientation), by processing the source from the beginning *without real-time playback*. This allows the operation to be executed significantly faster. The file includes the structure needed to create a SOFA file with the new AnnotatedReceiverAudio convention.

Syntax

`/source/record <string source_id> <string filename> <string type> <float seconds>`

`source_id` : identifier of the source to be processed and recorded.

`filename` : indicates the name of the file and must include the path, either relative or absolute. If a relative path is used it will be calculated from the data folder that can be found in the same folder as the BeRTA executable. The extension will be added by the application if necessary. If there is a file with the same name, it'll not be overwritten—an ordinal number will be added to the end of the new file name.

`type` : indicates the extension of the file: wav (not implemented yet) or mat (MATLAB binary data container).

`seconds` : indicates the duration of the recording. **This value must be greater than 0.** Unlike `/source/playAndRecord`, the special value `-1` is *not* allowed.

Return

`/control/actionResult /source/record <string source_id> <bool recorded> <string description>`

The return confirmation refers to the `source_id`, indicating `recorded=true` if the action has been successfully performed and `recorded=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribers except the sender: `/source/record SoundSource1 C:/tmp/testingRecord mat 3`

BeRTA sends back to the sender: `/control/actionResult /source/record SoundSource1 true`

`"Recording completed. File saved: C:/tmp/testingRecord"` or	`/control/actionResult /source/record SoundSource1 false` `"ERROR: Recording failed. File could not be created."`

3.4.6 Listener commands

In this section, we describe the various OSC commands used to control the listener. These commands not only manage basic parameters like position and orientation but also define the models applied for simulation. These models come with a range of controllable parameters and can simulate either just the listener's head through a Head-Related Transfer Function (HRTF), or include the surrounding environment using a Binaural Room Impulse Response (BRIR), as the BRIR models the combined impulse response of a head situated within a room. See listener models for more information.

/LISTENER/LOCATION

Set location of listener. Position is set in global x,y,z coordinates, expressed in meters. The new location is sent to all remotes except the one has sent the message.

Syntax

`/listener/location <string listener_id> <float x> <float y> <float z>`

`listener_id` : identifier assigned to the listener.

`x` : X global coordinate, expressed in metres. As a reference, X axis is positive to the front.

`y` : Y global coordinate, expressed in metres. As a reference, Y axis is positive to the left.

`z` : Z global coordinate, expressed in metres. As a reference, Z axis is positive to up.

Return

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/listener/location defaultListener 0 0 0`

/LISTENER/ORIENTATION

Sets orientation of the listener. Orientation is set in egocentric coordinates yaw, pitch and roll, expressed in radians and applied in that order (Yaw, Pitch and Roll). An orinetation of (0, 0, 0) corresponds to be oriented towards positive X. The new orientation is sent to all remotes except the one has sent the message.

Syntax

`/listener/orientation <string listener_id> <float yaw> <float pitch> <float roll>`

`listener_id` : identifier assigned to the listener.

`yaw` : Yaw, expressed in radians. Positive to the right.

`pitch` : Pitch, expressed in radians. Positive upwards.

`roll` : Roll, expressed in radians. Positive to the right.

Return

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/listener/orientation defaultListener 0.1 0 0`

/LISTENER/SETHRTF

Sets an HRTF using the HRTF id. The HRTF should be load previously using the command `/resources/loadHRTF` . After set the HRTF the command `listener/enableSpatialization` is called automatically. The HRTF is assigned to all the listener models which may accept it. See listener models for more information. Currently, these are the Listener models accepting it:

- Listener Model based on Direct Convolution with HRTF.
- Listener Model based on convolution with HRTF in the Ambisonic domain

Syntax

`/listener/setHRTF <string listener_id> <string HRTF_id>`

`listener_id` : identifier assigned to the listener.

`HRTF_id` : identifier assigned to the HRTF.

Return

`/control/actionResult /listener/setHRTF <string listener_id> <bool set> <string description>`

The return confirmation refers to the `listener_id` , indicating `set=true` if the action has been successfully performed and `set=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/listener/setHRTF defaultListener HRTF1`

BeRTA sends back to the sender: `/control/actionResult /listener/setHRTF defaultListener true "HRTF HRTF1 has been set to listener defaultListener"`

/LISTENER/SETBRIR

Sets a BRIR using the BRIR id. The BRIR should be loaded previously using the command `/resources/loadBRIR`. After receiving this command `listener/enableSpatialization` is called automatically. The BRIR is assigned to all the models linked to the listener which may accept it. See the Listener+Environment models based on Room Impulse Responses for more infortmation. These models currently are:

- Listener + Environment Model based on Direct Convolution with BRIR.
- Listener + Environment Model based on convolution with BRIR in the Ambisonic domain

Syntax

`/listener/setBRIR <string listener_id> <string BRIR_id>`

`listener_id` : identifier assigned to the listener.

`BRIR_id` : identifier assigned to the BRIR.

Return

`/control/actionResult /listener/setBRIR <string listener_id> <bool set> <string description>`

The return confirmation refers to the `listener_id`, indicating `set=true` if the action has been successfully performed and `set=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/listener/setBRIR defaultListener BRIR1`

BeRTA sends back to the sender: `/control/actionResult / listener/BRIR1 defaultListener true "BRIR BRIR1 has been set to listener defaultListener"`

/LISTENER/SETSOSFILTERS

Sets a set of filters for the Near Field Compensation to be applied afgter the convolution with the HRTF. The filters are identified by an id, which was defined when they were previously loaded using the command `/resources/ loadSOSFilters`. The NFC filters are assigned to all the models linked to the listener which may accept it. These currently are:

- Listener Model based on Direct Convolution with HRTF.
- Listener Model based on convolution with HRTF in the Ambisonic domain

Syntax

`/listener/setSOSFilters <string listener_id> <string NFCFilters_id>`

`listener_id` : identifier assigned to the listener.

`NFCFilters_id` : identifier assigned to the Near Field Compensation filters.

Return

`/control/actionResult /listener/setSOSFilters <string listener_id> <bool set> <string description>`

The return confirmation refers to the `listener_id`, indicating `set=true` if the action has been successfully performed and `set=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/listener/setSOSFilters DefaultListener NFCFilters1`

BeRTA sends back to the sender: `/control/actionResult / listener/setSOSFilters defaultListener true "SOSFilters NFCFilters1 has been set to listener defaultListener"`

/LISTENER/ENABLESPATIALIZATION

In the case of a Listener Model based on Direct Convolution with HRTF and Listener Model based on BRIR, this command switch on or off the simulation of the spatial audio. In case it is off, the sound is played without spatialization.

Syntax

`/listener/enableSpatialization <string listener_id> <boolean enable>`

`listener_id` : identifier assigned to the listener.

`enable` : If true (1), enables spatilization for the listener. If false (0), spatilization is disabled.

Return

`/control/actionResult /listener/enableSpatialization <string listener_id> <bool enable> <string description>`

The return confirmation refers to the `listener_id`, indicating `enable=true` if the spatialization has been enabled and `enable=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/listener/enableSpatialization DefaultListener true`

BeRTA sends back to the sender: `/control/actionResult /listener/enableSpatialization defaultListener true "Spatialization enabled"`

/LISTENER/ENABLEINTERPOLATION

IIn the case of a Listener Model based on Direct Convolution with HRTF and Listener Model based on BRIR, this command switch on or off the interpolation among HRIRs. In case it is on, a barycentric interpolation is applied among the three nearest HRIRs surrounding the source direction of arrival. See HRTF grids for more information. If there are several models based on direct convolution with HRTF linked to the listener, this command will affect all of them.

Syntax

`/listener/enableInterpolation <string listener_id> <boolean enable>`

`listener_id` : identifier assigned to the listener.

`enable` : If true (1), enables interpolation of the HRTF of the listener. If false (0), interpolation is disabled.

Return

`/control/actionResult /listener/enableInterpolation <string listener_id> <bool enable> <string description>`

The return confirmation refers to the `listener_id`, indicating `enable=true` if the model has been enabled and `enable=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/listener/enableInterpolation DefaultListener true`

BeRTA sends back to the sender: `/control/actionResult /listener/enableInterpolation defaultListener true "Interpolation enabled to defaultListener"`

/LISTENER/ENABLENEARFIELDEFFECT

This command switches on or off the Near Field Compensation (NFC) applied together with the convolution with the HRTF. For the NFC to work, a set of NFC filters must first be loaded from a SOFA file (`/resources/loadNFCFilters`) and then assigned to the listener model (`/listener/setNFCFilters`). If there are several models with the NFC feature linked to the listener, this command will affect all of them. The models with this feature currently are:

- Listener Model based on Direct Convolution with HRTF.
- Listener Model based on convolution with HRTF in the Ambisonic domain

Syntax

`/listener/enableNearFieldEffect <string listener_id> <boolean enable>`

`listener_id` : identifier assigned to the listener.

`enable` : If true (1), enables NFC. If false (0), NFC is disabled.

Return

`/control/actionResult /listener/enableNearFieldEffect <string listener_id> <bool enable> <string description>`

The return confirmation refers to the `listener_id`, indicating `enable=true` if the model has been enabled and `enable=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `` `/listener/enableNearFieldEffect DefaultListener true ``

BeRTA sends back to the sender: `/control/actionResult /listener/enableNearFieldEffect DefaultListener true "NearField Effect enabled to DefaultListener"`

/LISTENER/ENABLEITD

This command switches on or off the simulation of ITD separate from the intrinsic ITD which could be included as initial delays in HRIRs. For the ITD simnulation to make sense, HRIRs stored in the HRTF SOFA file should be aligned. Then the ITD can be extracted from the initial delays stored in the Data.Delay field of the SOFA structure. However, the ITD simulation can also use a spherical head modelto sintesize ITD. See also the commands `/resources/enableWoodworthITD` and `/resources/setHRTFHeadRadius` for more information. If there are several models with the ITD simulation feature, this command will affect all of them. The models with this feature currently are:

- Listener Model based on Direct Convolution with HRTF.
- Listener Model based on convolution with HRTF in the Ambisonic domain

Syntax

`/listener/enableITD <string listener_id> <boolean enable>`

`listener_id` : identifier assigned to the listener.

`enable` : If true (1), enables ITD. If false (0), ITD is disabled.

Return

`/control/actionResult /listener/enableITD <string listener_id> <bool enable> <string description>`

The return confirmation refers to the `listener_id`, indicating `enable=true` if the model has been enabled and `enable=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/listener/enableITD DefaultListener true`

BeRTA sends back to the sender: `/control/actionResult /listener/enableITD DefaultListener true "ITD enabled to DefaultListener"`

/LISTENER/ENABLEPARALLAXCORRECTION

This command switches on or off the parallax correction used to calculate the direction of arrival to each of the two ears. To do so, the head radius is needed and it is taken from the HRTF SOFA file or the radius provided by the `/resources/setHRTFHeadRadius` command. If there are several models with this parallax correction feature, this command will affect all of them. The models with this feature currently are:

- Listener Model based on Direct Convolution with HRTF.
- Listener Model based on convolution with HRTF in the Ambisonic domain

Syntax

`/listener/enableParallaxCorrection <string listener_id> <boolean enable>`

`listener_id` : identifier assigned to the listener.

`enable` : If true (1), enables parallax correction. If false (0), parallax correction is disabled.

Return

`/control/actionResult /listener/enableITD <string listener_id> <bool enable> <string description>`

The return confirmation refers to the `listener_id`, indicating `enable=true` if the model has been enabled and `enable=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/listener/enableParallaxCorrection DefaultListener true`

BeRTA sends back to the sender: `/control/actionResult /listener/enableParallaxCorrection DefaultListener true "Parallax Correction enabled to DefaultListener"`

/LISTENER/ENABLEMODEL

Deprecated since BeRTA v3.6.0, use '/enableModel' instead.

This command switches on or off a listener model. When a listener model is disabled it does not process the input signal and provides silence at its output. This feature must be implemented in all listener models. The listener model to be enabled or disabled is didentified by an identifier defined in the used settings file.

Syntax

`/listener/enableModel <string listenerModel_id> <boolean enable>`

`listenerModel_id` : identifier assigned to the model.

`enable` : If true (1), enables the model. If false (0), the model is disabled and its output will be silent.

Return

`/control/actionResult /listener/enableModel <string listenerModel_id> <bool enable> <string description>`

The return confirmation refers to the `listenerModel_id`, indicating `enable=true` if the model has been enabled and `enable=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/listener/enableModel DirectPath true`

BeRTA sends back to the sender: `/control/actionResult / listener/enableModel DirectPath true "Listener model DirectPath enabled."`

/LISTENER/SETAMBISONICSORDER

Sets the order of the Ambisonic encoding used by the listener model. The listener model is didentified by an identifier defined in the used settings file. This command can be only used with models based on Ambisonics. If it is referred to a different model it will be ignored. The models with this feature currently are:

- Listener Model based on convolution with HRTF in the Ambisonic domain
- Listener + Environment Model based on convolution with BRIR in the Ambisonic domain

Syntax

`/listener/setAmbisonicsOrder <string listenerModel_id> <int ambisonicsOrder>`

`listenerModel_id` : identifier assigned to the model.

`ambisonicsOrder` : Ambisonic order. Possible values are 1, 2 and 3.

Return

`/control/actionResult /listener/setAmbisonicsOrder <string listenerModel_id> <bool set> <string description>`

The return confirmation refers to the `listener_id`, indicating `set=true` if the action has been successfully performed and `set=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/listener/setAmbisonicsOrder DirectPath 3`

BeRTA sends back to the sender: `/control/actionResult / listener/setAmbisonicsOrder DirectPath true "Ambisonics order set to 3"`

/LISTENER/SETAMBISONICSNORMALIZATION

Sets the normalization type used in the Ambisonic encoding used by the listener model. The listener model is didentified by an identifier defined in the used settings file. This command can be only used with models based on Ambisonics. If it is referred to a different model it will be ignored. The models based on Ambisonics currently are:

- Listener Model based on convolution with HRTF in the Ambisonic domain
- Listener + Environment Model based on convolution with BRIR in the Ambisonic domain

Syntax

`/listener/setAmbisonicsNormalization <string listenerModel_id> <string ambisonicsNormalization>`

`listenerModel_id` : identifier assigned to the model.

`ambisonicsNormalization` : Ambisonic normalization type. Possible values are: `N3D`, `SN3D`, `maxN`.

Return

`/control/actionResult /listener/setAmbisonicsNormalization <string listenerModel_id> <bool set> <string description>`

The return confirmation refers to the `listener_id`, indicating `set=true` if the action has been successfully performed and `set=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/listener/setAmbisonicsNormalization ReverbPath N3D`

BeRTA sends back to the sender: `/control/actionResult / listener/setAmbisonicsNormalization DirectPath true "Ambisonics normalization set to N3D"`.

/LISTENER/ENABLEDISTANCEATTENUATION

Available from BeRTA v3.6.0

This command enables or disables the distance simulation in the listener model, *only available in the combined listener/environment models*. This simulation consists in applying a global attenuation, where doubling the distance between the source and the listener reduces the sound level by a predefined attenuation value.

Syntax

`/listener/enableDistanceAttenuation <string listenerModel_id> <boolean enable>`

`listenerModel_id` : identifier assigned to the model.

`enable` : If true (1), enables the simulation of the distance attenuation. If false (0), disable the simulation of the distance attenuation.

Return

`/control/actionResult /listener/enableDistanceAttenuation <string listenerModel_id> <bool enabled> <string description>`

The return confirmation refers to the `listenerModel_id`, indicating `enabled=true` if the distance attenuation has been enabled and `enabled=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/listener/enableDistanceAttenuation ReverbPath true`

BeRTA sends back to the sender: `/control/actionResult /listener/enableDistanceAttenuation ReverbPath true "Distance attenuation enabled in the model (ReverbPath)."`.

/LISTENER/SETDISTANCEATTENUATIONFACTOR

Available from BeRTA v3.6.0

This command allows you to change the attenuation factor per distance, in decibels. This is the amount of attenuation applied to the signal each time the distance is doubled from the reference distance. By default, this attenuation has a value of -3 dB and the default reference distance is 1 meter. The value to set must always be negative. For more information, see the section on the listener and environment combined models.

Syntax

`/listener/setDistanceAttenuationFactor <string listenerModel_id> <float distanceAttenuationFactor>`

`listenerModel_id`: identifier assigned to the model.

`distanceAttenuationFactor`: a float value, expressed in dB, representing the distance attenuation factor. This value must be negative.

Return

`/control/actionResult /listener/setDistanceAttenuationFactor <string listenerModel_id> <float distanceAttenuationFactor> <string description>`

The return confirmation refers to the `listenerModel_id`, indicating if the attenuation factor `distanceAttenuationFactor` has been set or not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/listener/setDistanceAttenuationFactor ReverbPath -3`

BeRTA sends back to the sender: `/control/actionResult /listener/setDistanceAttenuationFactor ReverbPath true "Distance Attenuation Factor updated in the model (ReverbPath) to -3".`

3.4.7 Environment commands

This section covers the OSC commands responsible for / controlling the environment. These commands handle parameters related to the characteristics of the space, such as room geometry and the absorption profiles of the walls, particularly in models based on geometrical acoustics.

/ENVIRONMENT/ENABLEMODEL

Deprecated since BeRTA v3.6.0, use '/enableModel' instead.

This command switches on or off an environment model. When an environment model is disabled it does not process the input signal and provides silence at its output. this feature must be implemented in all environment models. The environment model to be enabled or disabled is didentified by an identifier defined in the settings file.

Syntax

`/environment/enableModel <string environmentModel_id> <boolean enable>`

`environmentModel_id` : identifier assigned to the model.

`enable` : If true (1), enables the model. If false (0), the model is disabled and its output will be silent.

Return

`/control/actionResult /environment/enableModel <string environmentModel_id> <bool enable> <string description>`

The return confirmation refers to the `environmentModel_id`, indicating `enable=true` if the model has been enabled and `enable=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/environment/enableModel SDN true`

BeRTA sends back to the sender: `/control/actionResult /environment/enableModel SDN true "Environment model SDN enabled".`

/ENVIRONMENT/SETSHOEBOXROOM

Sets up a shoebox room with six walls (four walls, plus floor and ceiling). The six walls are automatically created with the centre of the room at 0, 0, 0. The following ID are assigned for each wall:

- `0` front wall (positive x)
- `1` left wall (positive y)
- `2` right wall (negative y)
- `3` back wall (negative x)
- `4` floor (negative z)
- `5` ceiling (positive z)

Syntax

`/environment/setShoeBoxRoom <string environmentModel_id> <float length> <float width> <float height>`

`environmentModel_id` : identifier assigned to the model.

`length` : extension of the room in the X axis, expressed in metres.

`width` : extension of the room in the Y axis, expressed in metres.

`heigth` : extension of the room in the Z axis, expressed in metres.

Return

`/control/actionResult /environment/setShoeBoxRoom <string environmentModel_id> <bool set> <string description>`

The return confirmation refers to the `environmentModel_id`, indicating `set=true` if the action has been successfully performed and `set=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/environment/setShoeBoxRoom SDN 6 8 3`

BeRTA sends back to the sender: `/control/actionResult /environment/setShoeBoxRoom SDN true "ShoeBox Room set to SDN (Length, Width, Height): [6, 8, 3]"`

/ENVIRONMENT/SETWALLABSORPTION

Sets the absorption coeficients of a wall. These can be frequency independent, expressed with just one number between 0 and 1, or frequency dependent, expressed in thac case with nine numbers with absorptions corresponding to 62.5Hz, 125Hz, 250Hz, 500Hz, 1KHz, 2KHz, 4KHz, 8KHz and 16KHz.

Syntax

`/environment/setWallAbsorption <string environmentModel_id> <int wall_id> <float absorption_fullband>`

`/environment/setWallAbsorption <string environmentModel_id> <int wall_id> <float abs62> <float abs125> <float abs250> <float abs500> <float abs1K> <float abs2K> <float abs4K> <float abs8K> <float abs16K>`

`environmentModel_id` : identifier assigned to the model.

`wall_id` : index of the wall to which the absorption apllies.

`absoption_fulband` : absorption coefficient which is frequency independent.

`abs62` : absorption coefficient of the band centered at 62.5Hz.

`abs125` : absorption coefficient of the band centered at 125Hz.

`abs250` : absorption coefficient of the band centered at 250Hz.

`abs500` : absorption coefficient of the band centered at 500Hz.

`abs1K` : absorption coefficient of the band centered at 1KHz.

`abs2K` : absorption coefficient of the band centered at 2KHz.

`abs4K` : absorption coefficient of the band centered at 4KHz.

`abs8K` : absorption coefficient of the band centered at 8KHz.

`abs16K` : absorption coefficient of the band centered at 16KHz.

Return

`/control/actionResult /environment/setWallAbsorption <string environmentModel_id> <bool set> <string description>`

The return confirmation refers to the `environmentModel_id`, indicating `set=true` if the action has been successfully performed and `set=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Examples

BeRTA receives and echoes back to all subscribiers but the sender: `environment/setWallAbsorption SDN 0 0.1 0.2 0.3 0.4 0.5 0.4 0.3 0.2 0.1`

BeRTA sends back to the sender: `/control/actionResult /environment/setWallAbsorption SDN true "Wall absorption set to SDN, wall 0 : [0.1, ...]"`

/ENVIRONMENT/ENABLEDIRECTPATH

This command switches on or off the virtual source coresponding to the direct path, which actually is not virtual, but the real one. Disabling it will produce silence at its output. This feature can be implemented in all environment models.

Syntax

`/environment/enableDirectPath <string environmentModel_id> <boolean enable>`

`environmentModel_id` : identifier assigned to the model.

`enable` : If true (1), enables the direct path source. If false (0), mutes the direct path source.

Return

`/control/actionResult /environment/enableDirectPath <string environmentModel_id> <bool enabled> <string description>`

The return confirmation refers to the `environmentModel_id`, indicating `enabled=true` if the direct path has been enabled and `enabled=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/environment/enableDirectPath SDN false`

BeRTA sends back to the sender: `/control/actionResult /environment/enableDirectPath SDN true "Environment model SDN Direct path enabled"`

/ENVIRONMENT/ENABLEREVERBPATH

This command switches on or off all the virtual sources but the one of the direct path, which actually is not virtual. Disabling them will produce silence at all their outputs. This feature can be implemented in all environment models.

Syntax

`/environment/enableReverbPath <string environmentModel_id> <boolean enable>`

`environmentModel_id` : identifier assigned to the model.

`enable` : If true (1), enables the reverb path sources. If false (0), mutes all reverb path sources.

Return

`/control/actionResult /environment/enableReverbPath <string environmentModel_id> <bool enabled> <string description>`

The return confirmation refers to the `environmentModel_id`, indicating `enabled=true` if the model direct path has been enabled and `enabled=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/environment/enableReverbPath SDN false`

BeRTA sends back to the sender: `/control/actionResult /environment/enableReverbPath SDN true "Environment model SDN Reverb path enabled"`.

/ENVIRONMENT/ENABLEPROPAGATIONDELAY

This command enables or disables the simulation of propagation delay throughout the entire environment, allowing users to control whether or not the system accounts for delays in signal propagation during audio rendering.

Syntax

`/environment/enablePropagationDelay <string environmentModel_id> <boolean enable>`

`environmentModel_id`: identifier assigned to the model.

`enable`: If true (1), enables the simulation of the propagation delay. If false (0), disable the simulation of the propagation delay.

Return

`/control/actionResult /environment/enablePropagationDelay <string environmentModel_id> <bool enabled> <string description>`

The return confirmation refers to the `environmentModel_id`, indicating `enabled=true` if the propagation delay has been enabled and `enabled=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/environment/enablePropagationDelay FreeFieldEnvironmentModel true`

BeRTA sends back to the sender: `/control/actionResult /environment/enablePropagationDelay FreeFieldEnvironmentModel true "Propagation delay in the FreeFieldEnvironmentModel model enabled."`.

/ENVIRONMENT/ENABLEDISTANCEATTENUATION

Available from BeRTA v3.6.0

This command enables or disables the distance simulation in the environment model, if available. This simulation consists in applying a global attenuation, where doubling the distance between the source and the listener reduces the sound level by a predefined attenuation value.

Syntax

`/environment/enableDistanceAttenuation <string environmentModel_id> <boolean enable>`

`environmentModel_id`: identifier assigned to the model.

`enable`: If true (1), enables the simulation of the distance attenuation. If false (0), disable the simulation of the distance attenuation.

Return

`/control/actionResult /environment/enableDistanceAttenuation <string environmentModel_id> <bool enabled> <string description>`

The return confirmation refers to the `environmentModel_id`, indicating `enabled=true` if the distance attenuation has been enabled and `enabled=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/environment/enableDistanceAttenuation FreeFieldEnvironmentModel true`

BeRTA sends back to the sender: `/control/actionResult /environment/enableDistanceAttenuation FreeFieldEnvironmentModel true "Distance attenuation enabled in the model (FreeFieldEnvironmentModel)."`.

/ENVIRONMENT/SETDISTANCEATTENUATIONFACTOR

Available from BeRTA v3.6.0

This command allows you to change the attenuation factor per distance, in decibels. This is the amount of attenuation applied to the signal each time the distance is doubled from the reference distance. By default, this attenuation has a value of -6.0206 dB and the default reference distance is 1 meter. The value to set must always be negative. For more information, see the section on the environment model.

Syntax

`/environment/setDistanceAttenuationFactor <string environmentModel_id> <float distanceAttenuationFactor>`

`environmentModel_id`: identifier assigned to the model.

`distanceAttenuationFactor`: a float value, expressed in dB, representing the distance attenuation factor. This value must be negative.

Return

`/control/actionResult /environment/setDistanceAttenuationFactor <string environmentModel_id> <bool setAttenuation> <string description>`

The return confirmation refers to the `environmentModel_id`, indicating if the attenuation factor has been set or not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/environment/setDistanceAttenuationFactor FreeField -3`

BeRTA sends back to the sender: `/control/actionResult /environment/setDistanceAttenuationFactor FreeField true "Distance Attenuation Factor updated in the model (FreeField) to -3".`

3.4.8 Binaural Filter commands

This section covers the OSC commands responsible for controlling the binaural filters using second order section filters.

/BINAURALFILTER/SETSOSFILTER

This command set the resource (the Second Order Section filters) to the simulation of the binaural filter model.

Syntax

`/binauralFilter/setSOSFilter <string model_id> <string sosfilter_id>`

`model_id` : identifier assigned to the model.

`sosfilter_id` : identifier assigned to the SOS filter resource.

Return

`/control/actionResult /binauralFilter/setSOSFilter <string model_id> <bool success> <string description>`

The return confirmation refers to the `model_id`, indicating `success=true` if the set has been done successfully and `enabled=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/binauralFilter/setSOSFilter DefaultListener Earmuffs`

BeRTA sends back to the sender: `/control/actionResult /binauralFilter/setSOSFilter DefaultListener true "SOS Filters (Earmuffs) has been set to listener (DefaultListener)"`

/BINAURALFILTER/ENABLEMODEL

Deprecated since BeRTA v3.6.0, use '/enableModel' instead.

This command enables or disables the simulation of binaural filter.

Syntax

`/binauralFilter/enableModel <string sosfilter_id> <boolean enable>`

`sosfilter_id` : identifier assigned to the model.

`enable` : If true (1), enables the simulation of binaural filter. If false (0), disable the simulation of the binaural filter.

Return

`/control/actionResult /binauralFilter/enableModel <string sosfilter_id> <bool enabled> <string description>`

The return confirmation refers to the `sosfilter_id`, indicating `enabled=true` if the binaural filter has been enabled and `enabled=false` if not. In both cases a `description` is added to give more details.

In case of success, an echo is sent to all subscribers except the sender, using the same syntax as the received message.

Example

BeRTA receives and echoes back to all subscribiers but the sender: `/binauralFilter/enableModel Earmuffs true`

BeRTA sends back to the sender: `/control/actionResult /binauralFilter/enableModel Earmuffs true "Binaural filter (Earmuffs) enabled"`.

3.5 BRT OSC API for Unity - Integration Package

This package enables the integration of a Unity scene with applications from the BRT library (BRT Application) using OSC commands. It can be download from our repository.

OSC communication

OSC communication between Unity and BRT Application

3.5.1 Package Contents

`BRTManager.cs`

A dual-purpose class designed to:

- Serve as an example of how to use the package.
- It can be used and extended by anyone, as it implements some important OSC commands.

`BRTCommands.cs`

This class handles the implementation, sending, and receiving of OSC commands.

Note: This class is still under development. Bugs may be present, and not all commands from the BRT interface are currently implemented. For more details about the available OSC commands, refer to the BRT OSC Documentation.

`OSC.cs`

A third-party library that implements the OSC protocol. Configure the IP and port for the remote application and the listening port for the Unity app in this script.

For the scripts described above to work, they must be added to the same object, as illustrated in the following image.

Scripts impoted in Unity.

Scripts impoted in Unity.

`BRTAudioListener.cs`

This script sends the listener's position and orientation to the BRT renderer via OSC commands.

To use this:

- Attach the script to your scene's camera object, or Instantiate our prefab (`CenterEarAnchor.prefab`) in the scene, as a child of a camera object.
- Enter the listener ID as defined in the BeRTA Renderer configuration file, see section.

`CenterEarAnchor.prefab`

A prefab designed to be placed under the "center eye" object in a virtual reality scene. Its purpose is to apply an offset, positioning it virtually at the center of the listener's head. This prefab also includes the `BRTAudioListener.cs` script.

CenterEarAnchor prefab in the Unity inspector

CenterEarAnchor prefab in the Unity inspector.

`BRTAudioSource.cs`

This script is designed to communicate the position and orientation of a virtual sound source. It must be installed on objects representing sound sources in the Unity virtual world. These values are sent every time they change. It will also load the source in BeRTA Renderer at startup.

To use this:

- Attach the script to your scene's objects
- Assign a unique ID
- Enter the full name of the audio file that the BeRTA Renderer should load.

BRTAudioSource.cs

BRTAudioSource.cs

3.5.2 Getting Started: Basic Unity Setup

This guide explains how to set up a basic Unity scene that communicates with **BeRTA** using our Unity integration package. The setup includes **one static audio source** and **one listener**.

Step-by-Step Guide

1. IMPORT THE UNITY PACKAGE

Import the Unity package provided for BeRTA integration into your Unity project.

2. CREATE THE CORE BRT GAMEOBJECT

- In your Unity scene, create an **empty GameObject**.
- Add the following scripts as components:
- `BRTManager.cs`
- `BRTCommands.cs`
- `OSC.cs`
- Open the `OSC.cs` component and configure:
- The **remote IP and port** to match the BeRTA application.
- The **local listening port** for incoming OSC messages.

3. CREATE AN AUDIO SOURCE AND ASSIGN BRTCOMMANDS

- Create a new **GameObject** to act as your **audio source**.
- In the script attached to this source (e.g., `BRTAudioSource.cs`), **locate the `BRTCommands` field and drag the GameObject containing `BRTCommands.cs` into it**.
- In the audio file field, enter the absolute path to the .wav file you want BeRTA to play.

4. CONFIGURE THE LISTENER AND ASSIGN BRTCOMMANDS

- Drag the `CenterEarAnchor.prefab` into your scene as a child of the **Main Camera**.
- This prefab includes the `BRTAudioListener.cs` script and positions the listener at the center of the user's head.
- In the `BRTAudioListener.cs` script, **locate the `BRTCommands` field and drag the GameObject containing `BRTCommands.cs` into it**.

5. LAUNCH BERTA

- Open and run the **BeRTA application**.
- With Unity and BeRTA running, the **OSC connection will be established**, and audio will be rendered in real time.

Notes

- `BRTManager.cs`, `BRTCommands.cs`, and `OSC.cs` must be attached to the **same GameObject**.
- Both the **audio source** and the **listener prefab** must reference this GameObject through their `BRTCommands` field.
- Ensure your network settings and firewall allow OSC communication.

3.5.3 License

Except for the OSC.cs class, which was not developed by us, all other scripts in this package are distributed under the MIT license.

3.6 BRT OSC API for Matlab

Section under construction

3.7 Configuration File Setup

This section explains how to define a configuration file in JSON format that is read when the application starts. If the program is launched without arguments, it will look for a file called `settings.json` in the same folder. Alternatively, a custom configuration file can be provided as an argument, for example, `mysettings.json`. This file defines the audio interface settings, the model architecture, and, optionally, a set of resources to be re-loaded at the begining and a complete scenario with sources and listeners.

3.7.1 General Structure

The JSON configuration file is organized into several key sections that define settings such as general parameters, model architecture, resources (HRTFs, BRIRs, etc.), and scene configuration. Below is a guide on how to construct each section.

1. General Settings

These settings control general application parameters.

- **SampleRate**: The sample rate in Hz. Example: `48000`.
- **BufferSize**: The size of the buffer in samples. Example: `512`.
- **BufferFrames**: Controls stream latency. Larger values may provide better performance but increase latency.
- **OSCListenPort**: The port on which the OSC server listens for messages.
- **ExtrapolationMethod**: The method of extrapolation. Available options: `ZeroInsertion`, `NearestPoint`.
- **LogFile** : Selects whether to save a log file or not. `true`, `false`.
- **LogFilePath** : Folder where the log file is stored. By default the application saves the log files in the subfolder ‘data’ inside the installation folder. On Windows, if the installation folder is ‘Program Files’, you will need to run the program with administrator permissions.

Example:

```
"GeneralSettings": {
  "SampleRate": 48000,
  "BufferSize": 512,
  "BufferFrames": 4,
  "OSCListenPort": 10017,
  "ExtrapolationMethod": "NearestPoint",
  "LogFile": true,
  "LogFilePath" : "",
},
```

2. Models Architecture

This section defines the models used for sound rendering, including listeners and environment models.

- **Listeners**: Defines the physical listener(s) by ID. Example: `["DefaultListener"]`.
- **ListenerModels**: Specifies the listener models, identified by an ID and model type. The available types include `ListenerDirectHRTFConvolution`, `ListenerDirectBRIRConvolutionModel`, `ListenerAmbisonicVirtualLoudspeakersModel`, and `ListenerAmbisonicReverberantVirtualLoudspeakersModel`.
- **EnvironmentModels**: Specifies the environment models, such as the `SDNEnvironmentModel` and the `FreeFieldEnvironmentModel`.
- **Binaural Filters**: Specifies the binaural filter used. The available binaural filter model is the `SOSBinauralFilter`.
- **ConnectSourcesTo**: Defines which models the audio sources connect to when loaded. Sources can be connected to either listener models or environment models.
- **Model2ModelConnections**: Defines how are the interconnections between the different models (environment, listening and binaural filters). The outputs of the `OriginID` models will be connected to the inputs of the `DestinationID` models.
- **ConnectToListener**: Specifies how the listener models are connected to the listener. The output of the listener models is connected to the listener for mixing.

Example:

```
    "ModelsArchitecture": {
        "Listeners": ["DefaultListener"],
        "ListenerModels": [
            {"ID": "DirectPath", "Model":
"ListenerDirectHRTFConvolution"},
            {"ID": "ReverbPath", "Model":
"ListenerAmbisonicReverberantVirtualLoudspeakersModel"}
        ],
        "EnvironmentModels": [{"ID": "FreeField", "Model":
"FreeFieldEnvironmentModel"}],
        "BinauralFilters":[],
        "ConnectSourcesTo":["FreeField", "ReverbPath"],
        "Model2ModelConnections": [{"OriginID":"FreeField",
"DestinationID":"DirectPath"}],
        "ConnectToListener":[
      {"ModelID":"DirectPath", "ListenerID":"DefaultListener"},
            {"ModelID":"ReverbPath", "ListenerID":"DefaultListener"}
        ],
   },
```

3. Resources

This section lists the resources such as HRTFs, BRIRs, NearField Filters, and DirectivityTFs that are loaded when the application starts.

- **HRTFs**: A list of HRTFs (Head-Related Transfer Functions) to load. If no HRTFs should be loaded, leave the list empty (`[]`).
- **BRIRs**: A list of BRIRs (Binaural Room Impulse Responses) to load. BRIRs can provide spatial audio information specific to a particular room. If none should be loaded, leave this list empty (`[]`).
- **SOSFilters**: Defines the SOS filters to load. If no SOS Filter should be loaded, leave the list empty (`[]`)
- **DirectivityTFs**: A list of Directivity Transfer Functions (TFs) to apply to sound sources. These describe how sound is emitted in different directions from a sound source.

Example:

```
"Resources" : {
        "HRTFs": [{"ID": "HRTF1", "fileName": "resources//HRTF//
3DTI_HRTF_SADIE_II_D2_256s_48000Hz_resampled5.sofa",
"spatialResolution": 5}],
        "BRIRs": [{"ID": "BRIR1", "fileName": "resources//BRIR//
3DTI_BRIR_Trapezoid_48000Hz_3D.sofa", "spatialResolution": 15,
"fadeInWindowThreshold" : 0, "fadeInWindowRiseTime": 0,
"fadeOutWindowThreshold" : 0, "fadeOutWindowRiseTime": 0}],
        "SOSFilters": [
            {"ID": "DefaultNFFilters", "fileName": "resources//
SOSFilters//NearFieldCompensation_ILD_1.2m_48Khz.sofa"},
            ],
        "DirectivityTFs": [],
    },
```

4. Sound Sources

This section defines the sound sources to be loaded at startup. Each source must specify the full filename and the model used for rendering. The available source model options are `OmnidirectionalModel` or `DirectivityModel`.

- **ID**: A unique identifier for each sound source.
- **fileName**: The full path to the sound file to be used.
- **sourceModel**: Specifies the rendering model for the sound source. The two available models are `OmnidirectionalModel` and `DirectivityModel`.

Example:

```
    "SoundSources": [
        {"ID": "SoundSource1", "fileName": "resources//
MusArch_Sample_48kHz_Anechoic_FemaleSpeech.wav",
"sourceModel":"OmnidirectionalModel"},
        {"ID": "SoundSource2", "fileName": "resources//
MusArch_Sample_48kHz_Anechoic_MaleSpeech.wav",
"sourceModel":"OmnidirectionalModel"}
    ],
```

5. Scene Configuration

This section allows you to define the scene by specifying commands in OSC (Open Sound Control) format. These commands configure various parameters like looping sound sources, setting HRTFs, and configuring environmental properties such as room dimensions and wall absorption coefficients.

- **command**: The OSC command to execute.
- **parameters**: The parameters that accompany each OSC command.

Example:

```
    "SceneConfiguration": [
        {"command": "/source/loop", "parameters": ["SoundSource1",
"true"]},
        {"command": "/listener/setHRTF", "parameters":
["DefaultListener", "HRTF1"]},
        {"command": "/listener/setSOSFilters", "parameters":
["DefaultListener", "DefaultNFFilters"]},
        {"command": "/listener/setBRIR", "parameters":
["DefaultListener", "BRIR1"]}
    ],
```

3.8 Annotated Receiver Audio SOFA Convention

The AnnotatedReceiverAudio is a proposed SOFA (Spatially Oriented Format for Acoustics) convention to record and store (binaural) audio data at the receivers, annotated with geometric information. It is being currently discussed and considered as work in progress for the SOFA standard.

SOFA is an open standard developed to store, exchange, and manage spatial acoustic data efficiently. It is widely used in the fields of 3D audio, virtual reality, and acoustics research to handle data such as Head-Related Transfer Functions (HRTFs), Binaural Room Impulse Responses (BRIRs), and other spatially oriented measurements.

The AnnotatedReceiverAudio convention aims at storing all the raw data during an experiment with participants to be reused for validation of new auditory models, or to reproduce the experiment. All the details can be seen here in the SOFA web site.

3.8.1 Key Fields

- **Data.Receiver**: stores the audio samples received by the ears (receivers).
- **M**: timestamp of measurements, it indicates the instant of time at which each of the captures is made.
- **ReceiverPosition**: specifies the 3D spatial positions of the ears (receivers) relative to the listener position.
- **ListenerPosition**: listener head positions.
- **ListenerUp**: listener orientations, up vector
- **ListenerView**: listener orientations, view vector.
- **EmitterPosition**: defines the 3D positions of the sound sources (emitters) relative to the source positions.
- **SourcePosition**: positions of the virtual set.
- **Response**: the subject's responses. If an experiment is being performed, it allows storing the subject's response under these conditions.

BeRTA Renderer is able to record files using this AnnotatedReceiverAudio convention including the delivered binaural audio together with all the above information siynchronized with the audio. See `/playAndRecord` and `/source/playAndRecord` for more information about the OSC commands to use this feature. these commands can also be sent by BeRTA GUI.

An example of AnnotatedReceiverAudio sofa file can be download from our repository.

3.9 Calibration

BeRTA provides a calibration option, which allows establishing the relationship between the system's dBFS (Decibels Full Scale) and the SPL (Sound Pressure Level) in dB of the output signal.

Calibration is essential in 3D audio systems to ensure that the perceived loudness of sounds matches real-world expectations. In digital audio, dBFS represents the signal level relative to the maximum possible amplitude, where 0 dBFS corresponds to the system's peak. However, real-world loudness is measured in dB SPL, which quantifies sound pressure in the air.

By calibrating BeRTA, users can map the digital signal levels to actual sound pressure levels, ensuring consistency across different playback environments. This is particularly useful for applications requiring precise loudness control.

3.9.1 Calibration Process in BeRTA

The calibration process ensures that the system's digital audio levels (dBFS) correspond accurately to real-world sound pressure levels (dB SPL). It consists of **four steps**, two performed via software and two requiring physical measurements. In summary, the process involves generating a calibration signal within BeRTA, measuring its output with a sound level meter, applying the calibration settings in the system, and verifying that the calibration is accurate.

Step-by-Step Calibration

1. **Generating the Calibration Signal**: The process begins by sending the **Start Calibration**[1] OSC command. This command triggers BeRTA to generate a pink noise signal at a specified dBFS level, which is provided as a parameter. *(Software step)*
2. **Measure the output**: The user must measure the output signal from one of the headphone channels using a sound level meter. The measured value is the sound pressure level (dB SPL) of the output signal. We recommend setting the sound level meter to slow mode and used the C-weighting curve. *(Physical step)*.
3. **Applying Calibration**: Once the measurement is taken, the user sets the calibration using the **SetCalibration**[1] command in BeRTA. This step establishes the relationship between the system's digital audio levels (dBFS) and the actual sound pressure level (dB SPL). *(Software step)*
4. **Verifying Calibration** *(optional)*: To ensure that the calibration has been correctly applied, the **PlayCalibrationTest**[1] OSC command can be used. This command specifies the desired dB SPL level, and BeRTA emits a pink noise signal at that volume. The user can then verify the output level using a sound level meter. *(Software and Physical step)*

After completing these steps, the system is calibrated, ensuring that the audio output maintains a consistent and accurate loudness reference for immersive and professional applications.

3.9.2 Monitoring Sound Levels

Once calibrated, BeRTA will display the real-time signal level[2] in dB SPL in the bottom-right section of the interface. Additionally, users can query the current signal level programmatically using the **GetSoundLevel**[1] OSC command.

To estimate the signal level, BeRTA calculates the RMS of the final digital signal and converts it to dBFS. Since the system has been calibrated using a reference SPL measurement provided by the user, BeRTA can apply the calibration factor to approximate the actual dB SPL level experienced by the listener.

Although this estimation is generally accurate, it is important to note that it remains an approximation. Several factors can introduce discrepancies, including:

- **Hardware limitations**: The analog components of the system, such as the amplifier and headphones, may not be capable of producing the expected sound pressure level, leading to saturation or distortion.
- **Frequency response variations**: The system may not accurately reproduce certain frequency components due to the frequency response limitations of the playback devices, such as headphones or speakers.
- **Non-linear distortions**: The listener’s headphones may introduce non-linear distortions that alter the actual perceived sound pressure level.

Despite these potential variations, the real-time SPL estimation provides a useful reference for maintaining consistent loudness levels across different playback environments.

1. For more details, see OSC Commands. ↩↩↩↩
2. The displayed SPL values are estimations based on the calibration factor and may not account for all variations introduced by hardware limitations and environmental factors. ↩

3.10 Safety Limiter

3.10.1 Importance of Setting a Maximum Signal Level

In spatial audio applications, ensuring safe listening levels is crucial to preventing hearing damage and avoiding unintended excessive sound levels. Prolonged exposure to high sound pressure levels (SPL) can cause hearing fatigue or even permanent hearing loss. Additionally, sudden loud sounds can be uncomfortable or startling for users, potentially compromising the immersive experience. To mitigate these risks, BeRTA includes a **Safety Limiter** that allows users to define a maximum SPL threshold, preventing output levels from exceeding safe limits.

3.10.2 Functionality

Once the calibration process has been completed, users can configure a **safety control** in BeRTA. This control continuously monitors the signal level in real time. If the signal exceeds the predefined safety threshold, BeRTA automatically limits it to the maximum allowed level, displays a warning in the graphical interface, and sends an alert command to notify connected clients.

The safety threshold is set using the **setSoundLevelLimit** OSC command, where the user specifies the maximum allowed SPL level. For further details, refer to the OSC Commands section.

When the signal surpasses the limit, BeRTA sends an alert message via the **soundLevelAlert** OSC command to all clients that have established an OSC connection with the BeRTA Renderer. This ensures that external applications are immediately informed of potential safety violations.

By implementing the Safety Limiter, BeRTA helps maintain a controlled and safe listening environment, protecting users while ensuring consistent playback levels across different use cases.