TECHNISCHE UNIVERSITÄT ILMENAU
Institut für Praktische Informatik und Medieninformatik
Fachgebiet Elektronische Medientechnik

Media Project

Realization of a mobile virtual loudspeaker setup for the playback of multi-channel audio content (“Loudspeakers to Go”)

presented by

Pranav Sharma, Syed Muhammad Ahmed

Supervisor:

Dr.-Ing. Stephan Werner
Dr.-Ing. Florian Klein

Ilmenau, the June 19, 2023

Contents

Abstract

In this media project, we address the limitations of traditional headphone listening experiences by developing a dynamic binaural synthesis system that virtually reproduces a 5-channel surround speaker setup over headphones. While loudspeakers offer an immersive audio experience, headphones are typically constrained by in-head localization. Our system aims to overcome this drawback by utilizing binaural playback and allowing users to experience virtual surround sound while on the move.

We have designed and implemented a system that accounts for both horizontal and vertical head movements, ensuring the virtual loudspeaker positions remain stationary relative to the user's perspective. Incorporating body and head sensors on a BACKPACK1 and headphones respectively, our system allows for translational movements, with the virtual setup adapting to the listener's position. Furthermore, the system integrates virtual room acoustics, simulating various environments in which the virtual speaker setup is located.

To enhance user interaction, a customizable user interface is provided, enabling users to change the virtual room environment, pause, play, or change audio, calibrate head, and body pose, and toggle head and body tracking. Additionally, the user interface displays real-time head movements in the azimuthal plane, allowing users to visualize their orientation within the virtual space. The interface also offers proper feedback, ensuring users have a clear understanding of their interactions and system performance.

Our pose dynamic binaural synthesis system provides a novel, immersive listening experience that meets the requirements of a modern, mobile audience by bridging the gap between headphones and loudspeakers. The user interface's incorporation of real-time head movement tracking and feedback further enhances user engagement and overall experience.

2 Motivation and Introduction

2.1 Motivation

The rapidly advancing realm of audio technology persistently expands the limits of immersive experiences, captivating listeners and enhancing their lives through the profound impact of sound.The motivation behind our project is to contribute to this growing field by developing a real-time dynamic binaural synthesis system that adapts to the listener's orientation, providing a seamless and captivating listening experience. We hope to inspire further research and innovation in this domain by investigating the intersection of spatial audio, virtual environments, and human perception. Our commitment to advancing binaural synthesis and providing users with an unprecedented sense of presence and immersion drives us to create a system that not only meets current expectations but also lays the groundwork for audio technology's future.With this project, we aim to open doors to new possibilities and redefine the very essence of auditory experiences.

2.2 Introduction

In recent years, there has been a growing interest in the development of immersive and dynamic listening experiences, especially in the field of binaural synthesis[1]. Dynamic binaural synthesis helps in the creation of a realistic and engaging acoustic environment that adapts to the listener's movement and position.

This media project focuses on creating a real-time dynamic binaural synthesis system for use with headphones, with an emphasis on utilizing head and body poses to create an innovative solution for listeners on the go, where they can feel like the sound system is moving with them. This tool can help in further development and application in various fields related to audio technology and immersive sound experiences.

Our research is divided into five chapters, each of which will help us in a better understanding of the project. Chapter 3, "Fundamentals of Augmented Auditory Reality", discusses about the the fundamentals and techniques of spatial audio that help in creating an immersive auditory experience,

Chapter 4 focuses on "Dynamic Binaural Synthesis", explaining some basics of Binaural rendering along with the methodology and implementation of our proposed system that accounts for listener and speaker orientations.

In Chapter 5, we explore "pyBinSim: A Python Tool for Auralization of Audio", discussing the use of this tool in our project to facilitate the auralization process.

In Chapter 6, we explain the "System Architecture and Design", for the Loudspeakers to go

Project.

Chapter 7 details the procedure for the experimental validation of the System to go application.

Chapter 8 is dedicated to “Documentation”, providing a comprehensive guide on how to utilize the developed tool and system, ensuring that users can effectively interact with and benefit from our research.

Chapter 9 presents “Discussion and Outlook”, where we evaluate the project’s results, identify potential improvements, and explore the future applications and developments that could emerge from our research. Through this media project, we aim to advance the field of binaural synthesis and contribute to the ongoing pursuit of creating engaging and immersive audio experiences for users on the move.

Chapter 10, titled ”Summary”, provides a brief recap of our entire project. This includes our main aims, how we approached them, what we found out, and what we think could be improved. We also touch on how we might improve it in the future and where this project could help us in research area.

3 Fundamentals of Augmented Auditory Reality

This chapter focuses on the basic concepts of spatial audio and the binaural synthesis technique which helps in creating augmented auditory reality using headphones to enhances the listener experience.

3.1 Spatial Audio

Headphones are a popular way for people to listen to audio. However, they come with certain limitations that can make the sound feel like it's coming from inside the listener's head instead of from the outside world. This can lead to an unnatural listening experience and listener fatigue. Traditional headphone listening may not be the best choice for situations where the listener wants to feel like they are in the middle of the action, such as when playing games or using virtual reality. With the use of advanced digital signal processing algorithms, audio can be synthesized to create a perception of sound coming from different locations in a virtual acoustic environment. This considers the room's acoustic properties, such as reverberations, early reflections, late reflections, direct sound, source position, and the orientation of both the listener and the source. To achieve a spatial audio system, various techniques like binaural rendering, ambisonics, and object-based audio are used. As stated by one research paper [2],

> "Being able to plausibly or authentically reproduce acoustic reality is one of the most important drivers of audio-technical development, for example for room simulations, for music or speech reproduction".

Then the aim of such system becomes to seamlessly combine real and virtual acoustics in a plausible manner. To accomplish this, real room measurements are taken with relevant cues and essential information to maintain plausibility.

3.2 Head Related Transfer Functions (HRTFs)

Head-Related Transfer Functions are mathematical representations that characterize how an individual's unique head, torso, and outer ear (pinna) shape influence the sound reaching each ear[3]. HRTFs capture the spatial cues, such as interaural time differences and interaural level differences, which our brain uses to localize sound sources in three-dimensional space. By applying HRTFs to audio signals, binaural audio can be synthesized, providing listeners with a realistic spatial listening experience over headphones. Since HRTFs are highly individual, personalized HRTFs can improve localization accuracy and spatial perception compared to generic

HRTFs.In certain conditions HRTFs are Linear Time Invarient (LTI) systems [4] so they can be represented as

$$y(t) = h(t) \circledast x(t) = \int_{-\infty}^{\infty} h(\tau)x(t-\tau)d\tau \tag{3.1}$$

where $x(t)$ is the input signal to the system, $y(t)$ is the output signal, $h(t)$ is the transfer function of the system and $\circledast$ denotes the convolution. To represent it in frequency domain we can apply Fourier Transform to $x(t)$, $y(t)$ and $h(t)$, it can be then represented as

$$Y(f) = H(f) \, . \, X(f) \tag{3.2}$$

which can be written as

$$H(f) = \frac{Y(f)}{X(f)} \tag{3.3}$$

where $Y(f)$, $H(f)$ and $X(f)$ are the output, transfer function and input of the system respectively in frequency domain. The inverse Fourier transform of equation 3.3 for the $H(f)$ will result in the transfer function of the system $h(t)$ in time domain[4] as shown below in 3.1. So we can input a known signal like sine sweep, measure the output signal from the in ear microphones and then we can determine the HRTFs.

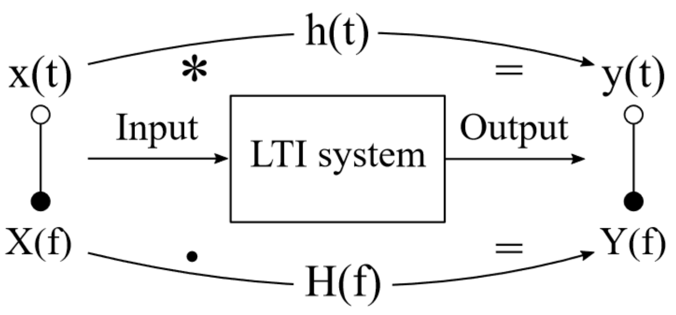


Figure 3.1: LTI system in time and frequency domain[5]

In this case, HRTF filter acts like an LTI system that modifies the sound waves based on the location and direction of the sound source. This is why it's possible to create virtual sound sources that appear to be coming from different directions around us by applying different HRTF filters to the same sound wave.

3.3 Binaural Room Impulse Responses (BRIRs)

Binaural Room Impulse Responses are impulse responses that capture the acoustic properties of a specific environment, including its reverberation characteristics, as well as the spatial cues necessary for binaural hearing. BRIRs are typically recorded or synthesized by combining a room's impulse response with HRTFs. When convolving an audio signal with BRIRs, the resulting sound simulates the listener's experience of the sound source in the given environment with spatial properties, such as direction, distance, and elevation, when played back over headphones.A simple method to get BRIRs from RIRs and HRTFs which are individualized to the same head and torso as the used HRTFs is given by Menzer et al., 2011, p. 404 in a journal article [6]. The SDM sound field parametrization method can be used to reconstruct BRIRs for arbitrary head orientations. This method involves using a [1xN] vector containing the pressure RIRs and a [3xN] matrix indicating the DOA for each sample, along with a HRIRs dataset, to create a set of BRIRs. The BRIRs are constructed by finding the nearest HRIRs to each sound event in each head orientation using rotation matrices, and then calculating a weighted sum of delayed HRIRs multiplied by instantaneous pressure values. The final formula for the BRIRs at a given head orientation is given by equation 3.4, which involves summing over all positions in the room of the delayed and scaled HRIRs as described by Amengual Garí et al., 2019, p. 162 [7].

$$BRIR^{u}(t) = \sum_{n=1}^{N} p_n HRIR_{\hat{k}_u^n} \circledast \delta(t-n) \tag{3.4}$$

p_n represents the instantaneous pressure at the nth position in the room, $HRIR_{\hat{k}_u^n}$ refers to the HRIR for the given source/receiver orientation at the nth position in the room and for the appropriate ear, $\circledast$ indicates convolution in time domain, $\delta(t-n)$ is a delta function that is shifted by n samples in time, and u is an index representing the ear for which the BRIRs is being calculated and t is the time index for the BRIRs.

3.4 Sound Localization and Perception

Sound localization and perception refer to the human auditory system's ability to identify the location, distance, and elevation of sound sources in the environment. Our brain processes various auditory cues, including ITD, ILD, and spectral cues caused by the filtering properties of our ears, to determine the position of sound sources as shown in 3.2. Accurate sound localization is essential for navigating our environment, understanding complex auditory scenes, and creating a sense of immersion in applications like virtual and augmented reality. The spherical coordinate system is often used to describe sound source positions in spatial audio, consisting of radius (r), azimuth (θ), and elevation (ϕ). Azimuth represents the horizontal angle in the medial plane, elevation refers to the vertical angle in the sagittal plane, and radius denotes the distance from the listener to the sound source.

3.5 Tait-Bryan angles

Tait-Bryan angles are widely used in spatial audio applications, as they provide a convenient way to describe the orientation of a listener's head or the direction of a sound source in three-dimensional space as shown in Figure 3.3. By combining Tait-Bryan angles with other spatial audio techniques, such as binaural audio and room impulse response modeling, it is possible to create highly realistic and immersive auditory experiences. For example, Tait-Bryan angles can be used to accurately model the effects of head movements on the perceived direction and distance of sound sources, which is essential for creating realistic virtual and augmented reality environments. However, it is important to be aware of the limitations and potential sources of

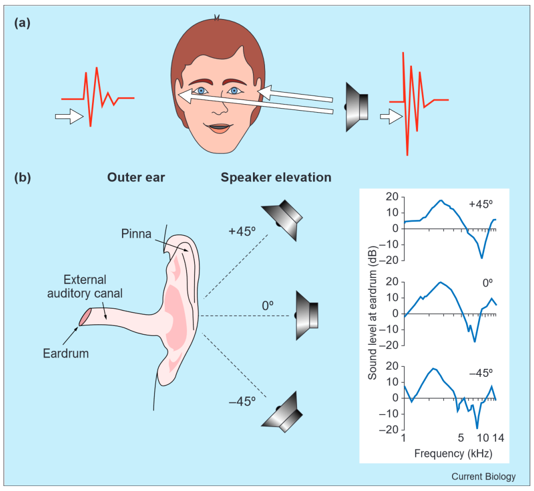


Figure 3.2: Figure 1. Sound localization cues. (a) Binaural cues of interaural time and level difference are produced by the extra time taken for sound to reach the far ear over the near ear (interaural time delay), and the sound shadow of the far ear due to the presence of the head (interaural level difference). (b) Convolutions of the outer ear impose direction-dependent shaping, measured at the eardrum, of the spectrum of a sound. These filtering effects will also contribute to the interaural level difference [8].

error associated with Tait-Bryan angles. For instance, different conventions for defining the axes of rotation can lead to differences in the interpretation of the angles, and the use of Euler angle sequences can result in unwanted singularities or ”gimbal lock” when certain orientations are reached. Therefore, it is important to carefully consider the choice of coordinate system and rotation order when working with Tait-Bryan angles in spatial audio applications.

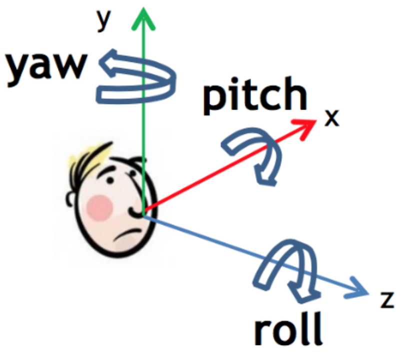


Figure 3.3: Euler angles defined in the head coordinate system [9]

3.6 Quaternions

Quaternions were introduced to generalize complex numbers. They have many applications in mathematics [10], computer graphics [11], and physics (aerospace, robotics, etc.) [12]. A quaternion q is defined as $q = a + bi + cj + dk$, where a, b, c, and d are real and $i^2 = j^2 = k^2 = ijk = -1$. Quaternions are often used as a parametrization of 3D rotations, especially for rotation interpolation. We recall that the set of 3D rotations can be mapped to the unit-norm quaternions under a one-to-two mapping, i.e., each 3D rotation matrix maps to two antipodal unit-norm quaternions: q and $-q$.

Spherical linear interpolation (slerp) consists of the following principle [12]:

$$\text{slerp}(q_1, q_2, \gamma) = q_1(q_1^{-1}q_2)^{\gamma}, \tag{3.5}$$

where q_1 and q_2 are respectively the starting and ending quaternions, and $0 \leq \gamma \leq 1$ is the interpolation factor. This is equivalent to

$$\text{slerp}(q_1, q_2, \gamma) = \frac{\sin((1-\gamma)\Omega)}{\sin(\Omega)}q_1 + \frac{\sin(\gamma\Omega)}{\sin(\Omega)}q_2, \tag{3.6}$$

where $\Omega = \cos^{-1}(q_1 \cdot q_2)$ is the angle between q_1 and q_2, and $q_1 \cdot q_2$ is the dot product of q_1 and q_2. This boils down to interpolating along the grand circle (or geodesics) on a unit sphere in 4D with a constant angular speed as a function of γ. To ensure that the quaternion trajectory follows the shortest path on the sphere [11], the relative angle between successive unit-norm quaternions needs to be checked to choose between $\pm q_2$.

4 Dynamic Binaural Synthesis

In this chapter we will use the previously gained knowledge, to dive deeper into the basics Pose Dynamic Binaural Synthesis for three degrees of freedom of head movements.

4.1 Binaural Synthesis

The term Binaural refers to the signals received by our left and right ears drums.Major cues that are responsible to give sense of direction and distance are interaural time differences, interaural, phase interaural level differences.Low frequencies are determined in the horizontal plane by interaural phase differences, medium frequencies by interaural time differences, and high frequencies by interaural level differences.The ability of human hearing to determine direction and distance to single sound sources is required for the creation of a 'auditive image' of the orchestra and the room [13]. In binaural synthesis, the desired audio signals are convolved with binaural room impulse responses to produce spatial audio[14]. Binaural synthesis exploits these principles by reconstructing the pressure signals at a listener's ears, based on the measurement or the simulation of binaural impulse responses[13].

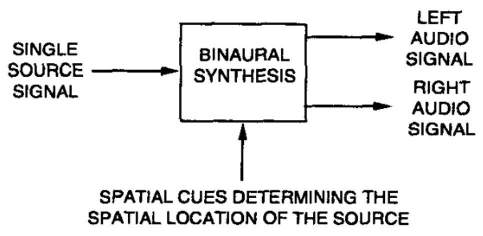


Figure 4.1: Synthesis of spatial placement of one source [13]

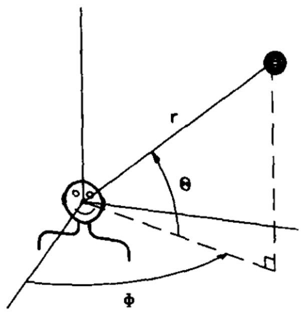


Figure 4.2: Sound source and listener in a free field [13]

In figure 4.2, the distance between the listener and the source is denoted by r, and the angle of incidence is defined by the azimuth ϕ and the elevation by θ.

4.2 Spatial Decomposition Method (SDM)

The method is based on the assumption that the measured sound field can be represented as a series of discrete acoustic events, each of which is represented by a sample in the RIRs with a corresponding direction-of-arrival (DOA) information[7]. The resulting image sources can be applied to multichannel convolution reverberation engines and room acoustics analysis. It estimates the DOA with the open microphone array in a three-dimensional arrangement with an omnidirectional microphone at the geometrical center. Time Difference of Arrival (TDOA) is used to estimate the difference in arrival times of a sound signal at different microphones with respect to geometric center of the array. It can be estimated with the help of Cross-correlation and direct weighting of the windowed RIRs between array elements. Due to increasing echo density, previous assumption is violated when multiple acoustic events arrive at the receiver simultaneously [7].Figure 4.3 shows how spatial sound can be extracted from the spatial room impulse response.

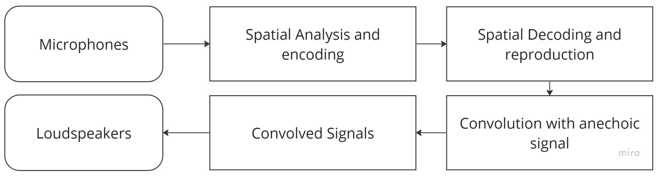

Microphones
Spatial Analysis and encoding
Spatial Decoding and reproduction
Convolution with anechoic signal
Convolved Signals
Loudspeakers
miro

Figure 4.3: The general processing applied in auralization using room impulse response measurements or simulations [15]

4.3 Pose based Dynamic Binaural Rendering

Dynamic binaural synthesis based on measured BRIRs can provide a very high degree of realism [16]. In this project Neumann KU 100 dummy head [17]is used as HRTF datasets. RIRs are taken for four rooms along with DOA, which is calculated separately using open microphone array and then quantized to 50 directions using a Lebedev grid[18]. After determining the DOA, it becomes possible to assign a direction and a head-related transfer function to each reflection. Using this information, it is then possible to calculate a binaural room impulse response that accurately represents the sound field with the help of SDM[15] , which also includes DOA Postprocessing (smoothing and quantization) and reverberation correction process RT-Mod+AP equalization steps as described in Amengual et al. 2020[18] . Responses for the head orientations are defined by arbitrarily rotating the DOA vectors. These Binaural room filters which contains information related to orientation and position for the three rotational degrees of freedom of head movements of listener and source are then used for real-time pose based dynamic binaural synthesis by convolving them with desired audio. As said in one research by Papadakis et al., 2011, p. 2 [19]

> "System latency (time delay) and its visible consequences are fundamental Virtual Environment (VE) deficiencies that can hamper spatial awareness and memory."

In the case of multi-channel audio, long filters need to be convolved in real time hence increasing the system latency. As stated by Werner et al., 2021, p. 10 [14]

> "In case of BRIRs rendering, long filters have to be convolved with the source signal in real-time for several sound sources at the same time. The state-of-the-art solution for this use case is a blocked convolution (overlap-add or overlap-save) with uniformly partitioned filters. This solution is significantly faster than using non-partitioned filters, but the computational complexity increases linearly with increased filter size and decreased block size. In case of limited computational power, a compromise between filter length and block size (which directly relates to the delay induced by the convolution) has to be found. To overcome this limitation, filters can be partitioned nonuniformly (e.g., short segments for the direct sound and early reflections and long segments for the late reverb)."

This has been applied in the project by changing DS filters (direct sound filters), ER filters (early reflection filters) and LR filters (late reverberation filters) based on the head and body orientation of virtual source and listener. This process has been shown in the figure 4.4.

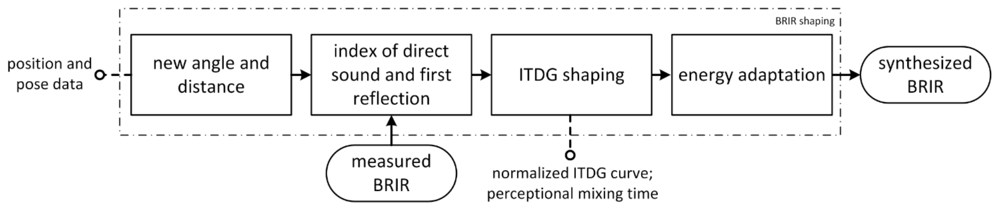


Figure 4.4: Block diagram of the synthesis approach adapting the initial time delay gap (ITDG) and using BRIRs of one measured position [14]

4.4 Differential Tracking

In this project, the system moves with the listener virtually. The position of the listener with respect to the 5 channels remains fixed, and vice versa. One sensor is placed on the headphones, while the other is placed on the body. Both sensors are connected to the local machine via Universal Serial Bus (USB) and continuously transmit real-time data to the local port, which is further translated by Open Sound Control (OSC) messages. The only things that change are head and body orientations (yaw and pitch). The tool takes the values from the head sensor and subtracts them from the corresponding values from the body sensor values. This allows the system to account for any offset between the listener's head and body orientation, as shown in figure 4.5. So, if there is no offset and the listener turns the body, the sound image in their head remains unchanged. And if the body orientation is fixed, the head translations shift the desired audio image accordingly.

By doing so, the virtual speakers appear to move with the listener, regardless of their body position. The positions of the virtual speakers and the listener remain static, while only the orientation of the listener's head and body changes. This task of binauralizing the desired audio is done with the help of a Python tool called pyBinSim [20], which is an open-source tool for real-time dynamic binaural synthesis implemented in Python on top of PyAudio. PyAudio is a cross-platform I/O library that has been used for the playback of the audio [21]. The next chapter provides more detailed insights about pyBinSim.

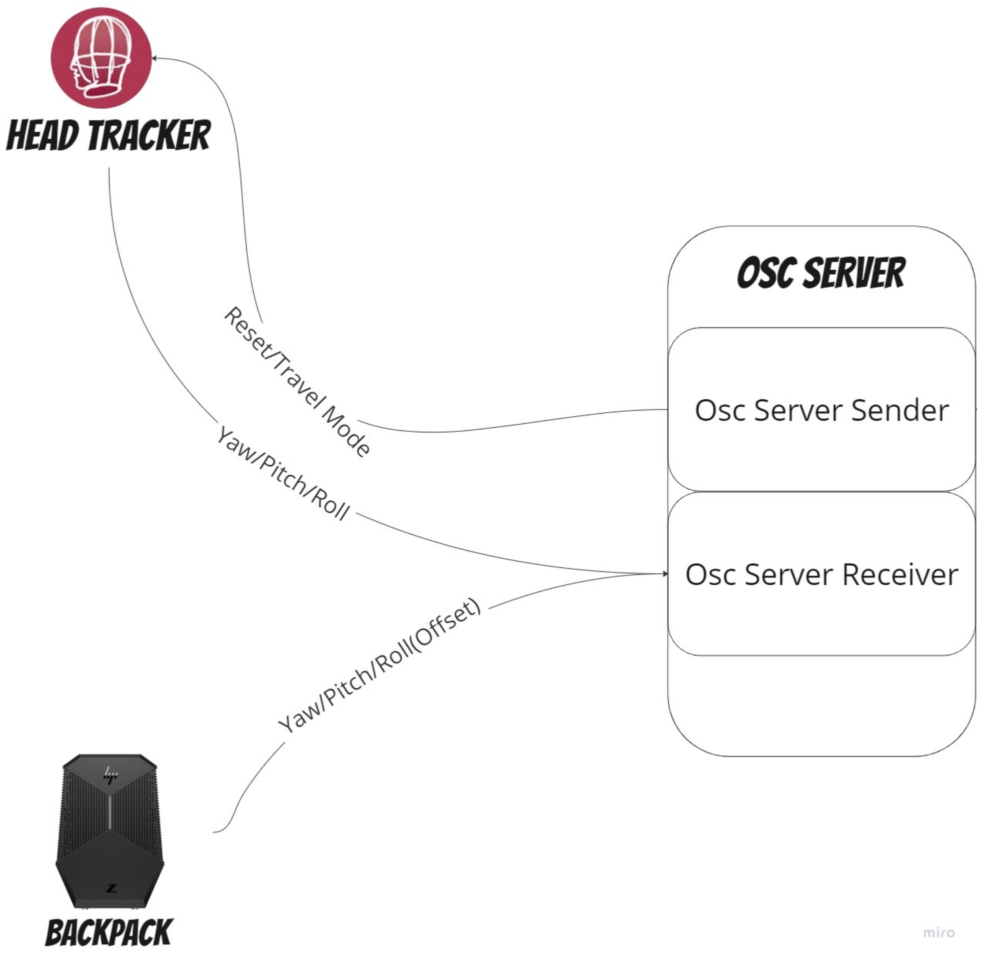


Figure 4.5: Using head and body sensor data to update pybinsim

5 pyBinSim: A Python Tool for Auralization of Audio

pyBinSim is an open source tool made in python language, it implements the uniformly partitioned convolution with the overlap-save approach. The basic principle of the pyBinSim is explained by Neidhardt et al., 2017, p. 2 [20],

> "Depending on the number of input channels (wave-file input or live audio input) the corresponding number of virtual sound sources is created. The filter for each sound source can be selected and activitated via OSC messages. The messages basically contain the number index of the source for which the filter should be switched and an identifier string to address the correct filter. The correspondence between parameter value and filter is determined by a filter list which can be adjusted individually for the specific use case. The simple filter assignment provides high flexibility when using this tool for your project. If only horizontal head rotation should be taken into account, it is advisable to use the azimuth angle as a parameter. If it is the goal, to switch virtual rooms, one parameter could be used so switch between different filter sets. For applications considering position-tracking and head-tracking, the six parameters could be used to address the 6DOF (six degrees of freedom): x, y, z, yaw, pitch and roll."

In our project, we are only focusing on the horizontal and vertical movements. The position of the listener and sources are fixed in the virtual world.

5.1 Configuration Parameters

There are multiple parameters that can be configured at startup or during runtime[21] which are shown in Figure 5.1.These parameters control various aspects of the pyBinSim controller, such as audio file input, processing settings, and output characteristics.

5.2 Flow of pyBinSim

Figure 5.1 explains how pyBinSim responds to the OSC requests it receives from the client.

pyBinSim_Processor() runs pyBinSim in the background to binauralize the desired audio based on the requests received by the OSC Receivers. We have used the multiprocessing library,

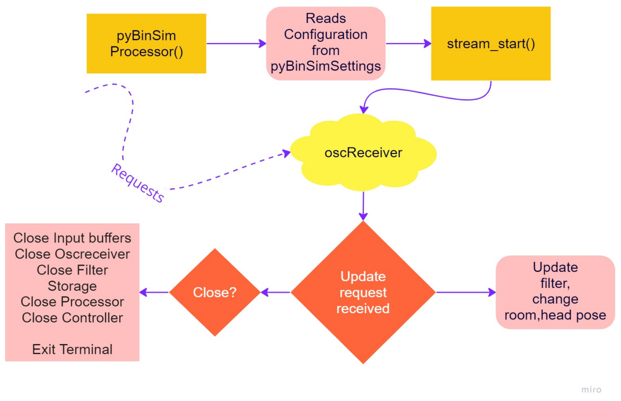


Figure 5.1: Flow of startup and request handling by pyBinSim

Parameter	**Description**
soundfile	Defines the *.wav file(s) played at startup (up to maxChannels channels). Accepts multiple files separated by #.
blockSize	Number of samples processed per block. Low values reduce delay but increase CPU load.
filterSize	Filter size for filters in the list. Must be a multiple of blockSize.
maxChannels	Maximum number of sound sources/audio channels. Must match or exceed the number of channels in soundFile(s).
samplingRate	Sample rate for filters and soundfiles. No automatic sample rate conversion.
enableCrossfading	Enables crossfade between audio blocks. Set ’False’ or ’True’.
useHeadphoneFilter	Enables headphone equalization. Requires filter with the identifier HPFILTER. Set ’False’ or ’True’.
loudness Factor	Factor for output loudness. Clipping may occur.
loopSound	Enables looping of sound file(s). Set ’False’ or ’True’.
newroom	Selects new room environment from filter database. Four available rooms.
close	Command to close components and exit the windows terminal.

Table 5.1: Configuration parameters and their descriptions

which uses subprocesses instead of threading, to create separate functions for the controller and processor as well as to run both processes simultaneously, utilizing the multiple processors of the CPU [22]. All the real-time binaural rendering of the desired multichannel audio is done in this block. It is responsible for updating filters on new room requests or based on the current head and body orientation. If the user closes the program, it closes all the input buffers, clears up the cache, and stops all the ongoing running processes before exiting the terminal, as shown in Figure 5.1.

6 System Architecture and Design

In this chapter we will discuss about the design, architecture, data management, scalability of our tool and details of the room database used in this project .

6.1 Overview

The motivation behind this project is to create a system that allows listeners to experience multi-channel audio as if they were sitting in the same room as the speakers. Portability is also a key aspect of this project, enabling the listener to take the system with them on the go. In this setup, the position of the listener remains static in the virtual world with coordinates (x,y,z) - in our case, the origin (0,0,0). The speakers have fixed coordinates in the virtual world with respect to each other and the listener, and the distance from the listener to each speaker remains fixed.

If the head orientation is fixed, the setup moves along with the listener based on the body orientation (yaw and pitch), as shown in Figure 6.2. If the body orientation is fixed, then real-time head movements are translated by the pyBinSim software, as described in Figure 6.1. This means that the sound appears to come from the same virtual position, even as the user's head moves. A simple 5-channel setup consists of speakers at the center (C), front right (FR), front left (FL), surround right (SR), and surround left (SL).

To account for body movements, this system uses a body sensor to detect the user's body orientation, and subtracts this offset from the head tracking data. By doing so, the system can maintain a consistent virtual position of the speakers, even as the user moves their entire body and head. For portability, we have used the HP VR Backpack PC, which has a powerful processor and can be easily carried as a backpack even on the go. For sensors, we used the Bridgehead Head Tracker 1 by Superware [23] and the Intel RealSense T265 sensor [24].

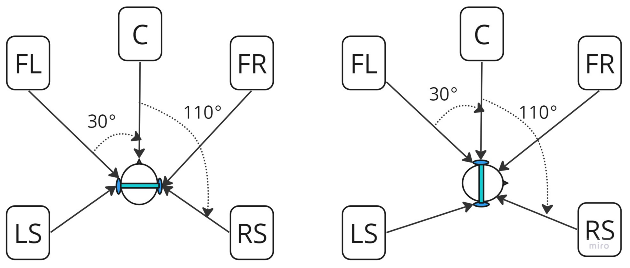


Figure 6.1: Translation of head movements by the system in virtual world setup if body orientation is fixed.

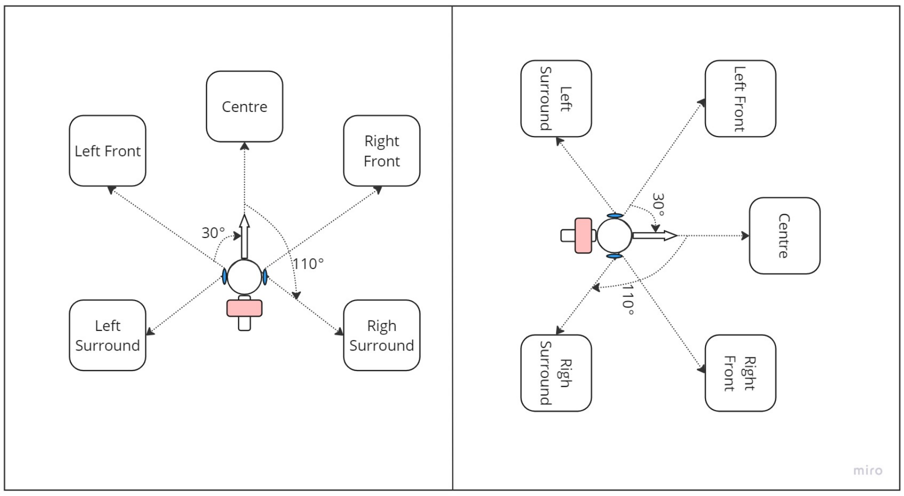


Figure 6.2: Translation of body movements by the system in virtual world setup if head orientation is fixed

6.2 System Architecture

The system architecture broadly consists of pyBinSim processor, pyBinSim controller and the User Interface part.

6.2.1 PyBinSim Processor

The PyBinSim processor is the central processing block of this tool, responsible for the binaural rendering of the desired audio. It receives its initial parameter settings, as described in 5.1, from the PyBinSim setting file generated by the MATLAB script. Using these values, it initializes the audio blocks and filter sizes. Since the MATLAB script generates the SDM-rendered BRIRs in .mat format, PyBinSim also parses and loads the .mat file into the PyBinSim variables. The OSC receiver is initialized with the start of PyBinSim and constantly checks for update requests regarding room changes or filter updates received from the PyBinSim controller.

OSC Request Handling

Open Sound Control (OSC) is a communication protocol used for transmitting control data between different devices and applications over a network. In this implementation, the OscReceiver class sets up four OSC servers on different ports to handle various types of incoming messages, such as DS filter updates, ER filter updates, LR filter updates, and miscellaneous controls like changing rooms, pausing audio playback, or adjusting loudness. It does so by mapping OSC message addresses to specific handler functions in the dispatchers. It continuously listens for incoming OSC messages and calls the appropriate handler function upon receiving a request. By efficiently handling these OSC messages, the OscReceiver class allows for seamless, real-time control of the PyBinSim system. If it receives a stop command, it stops the OSC servers and ends the process.

Pose Handling

The pose data received by PyBinSim includes listener and source orientations, positions, and custom values. The create_key() method concatenates their respective values and returns a comma-separated string. The static methods from _filterValueList() take a filter_value_list as input and create corresponding Pose or SourcePose objects based on the length of the input list. In this way, pose data is handled in the PyBinSim project, allowing for efficient parsing and representation of listener and source positions and orientations in a 3D space.

Filter Request Handling

In PyBinSim, we have discrete filters like DS, ER, and LR, so we have different block sizes for each filter, which are multiples of sound block sizes. It also parses and loads the .mat file that contains synthesized BRIRs into the PyBinSim variables. It then further parses the filter values and creates a dictionary from them. It searches in the dictionary for the available key and returns the corresponding filter. When no filter is found, the defaultFilter is returned, resulting in silence. Additionally, we have added a request_new_room_file() function, which retrieves the filter database file name, loads and parses the values in the PyBinSim variables, and applies the changes to the corresponding filter values.

Real-time Convolution

PyBinSim initializes variables and tensors needed for audio processing and performs the convolution on input audio buffers. It takes care of filling the frequency-domain delay lines (FDLs), performing complex multiplications and accumulations, and converting the result back to the time domain using inverse fast Fourier transform (IFFT). If interpolation is enabled, the method crossfades between the previous and current filters to provide smooth transitions and updates the impulse responses.

Sound and Buffer Handling

PyBinSim is capable of managing audio file reading and buffering for the PyBinSim system. It handles buffer operations, looping, and transitions between sound files. It is initialized with a block size, number of channels, sampling rate, and loop flag. It includes methods for adding sound or silence, reading from the buffer, loading sound files, and requesting new or next sound files. Processing and buffering audio input blocks are done separately in PyBinSim. It has methods to fill the buffer, process the input blocks, and get the processing counter.

6.2.2 pyBinSim controller

This process acts like a remote control to the pyBinSim. It provides the actual user interface to the listener and calls the functions which send requests via OSC client to the pyBinSim. It also handles the data from the body sensor and head sensor and uses differential tracking to give the input values to pyBinSim with the help of multithreading.

Configuration and Initializing

After running the Matlab script for BRIRs, two text files named pyBinSimSetting_SourcesListenerDefs and pyBinSimSettings_isoperare are created. In the former, all the details regarding source position for all channels, listener position, and range of values for yaw and pitch along with their resolution are stored, as shown in Figure 6.3. The latter one contains all the information regarding the paths and configuration parameters (refer to 6.3) required to initialize the pyBinSim processor, as shown in 6.2.1.

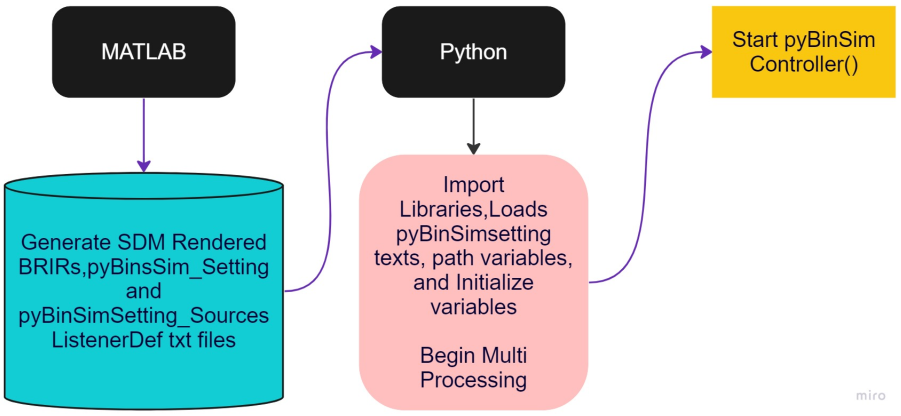

MATLAB
Generate SDM Rendered BRIRs,pyBinsSim_Setting and pyBinSimSetting_Sources ListenerDef txt files
Python
Import Libraries,Loads pyBinSimsetting texts, path variables, and Initialize variables
Begin Multi Processing
Start pyBinSim Controller()
miro

Figure 6.3: Data initialization for pyBinSim controller

The pyBinSim controller makes a list of variables for position and orientation of listeners as well as the range and resolution of yaw and pitch values from the information in pyBinSimSetting_SourcesListenerDefs.txt. It then initializes the OSC client with the help of the SimpleUDPClient to send messages to the OSC server[25]. It also initializes the OSC server to receive requests from the head sensor and takes the data from the body server in a different thread using the RealSense library. After receiving the data from both head and body sensors, it processes this data and sends the update requests to the pyBinSim processor, as shown in Figure 6.4.

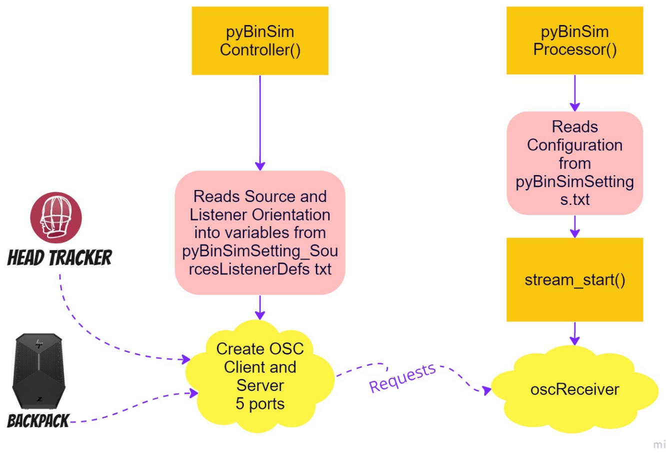

pyBinSim Controller()
Reads Source and Listener Orientation into variables from pyBinSimSetting_SourcesListenerDefs txt
Create OSC Client and Server 5 ports
HEAD TRACKER
BACKPACK
Requests
pyBinSim Processor()
Reads Configuration from pyBinSimSettings.txt
stream_start()
oscReceiver
miro

Figure 6.4: Collection of data from sensors and transmitting as OSC messages

User Interface

A User Interface (UI) can greatly improve the usability and user experience for individuals with varying levels of computer literacy [26]. Since it is not always possible to use both hands while on the go, a simple UI with buttons and good heuristics can solve this problem. We have used Tkinter for our project, which is a standard Python library used for creating graphical user interfaces. It provides a set of tools and widgets that can be used to create windows, buttons, text boxes, menus, and other interactive elements of a GUI [27]. Since we have created a UI with buttons and sliders, this can help the listener easily manipulate audio, change rooms, adjust loudness, visualize head movements in the azimuthal plane, reset body and head orientations, start and stop head tracking, and switch between different travel modes. We have also taken into consideration proper feedback in the form of text fields and changing button text and colors, as shown in Figure 6.5.

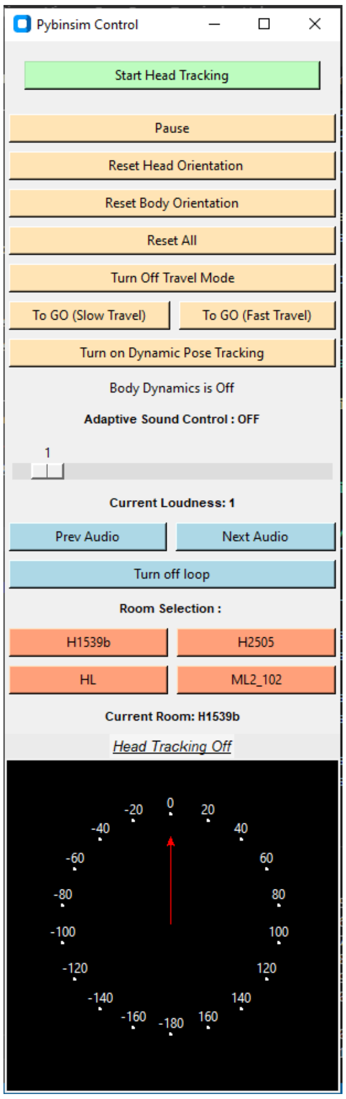


Figure 6.5: User Interface for the pyBinSim Controller

This user interface is initialized after the pyBinSim controller creates the OSC server and client. Each button has a specific function in the background, which is called every time the listener

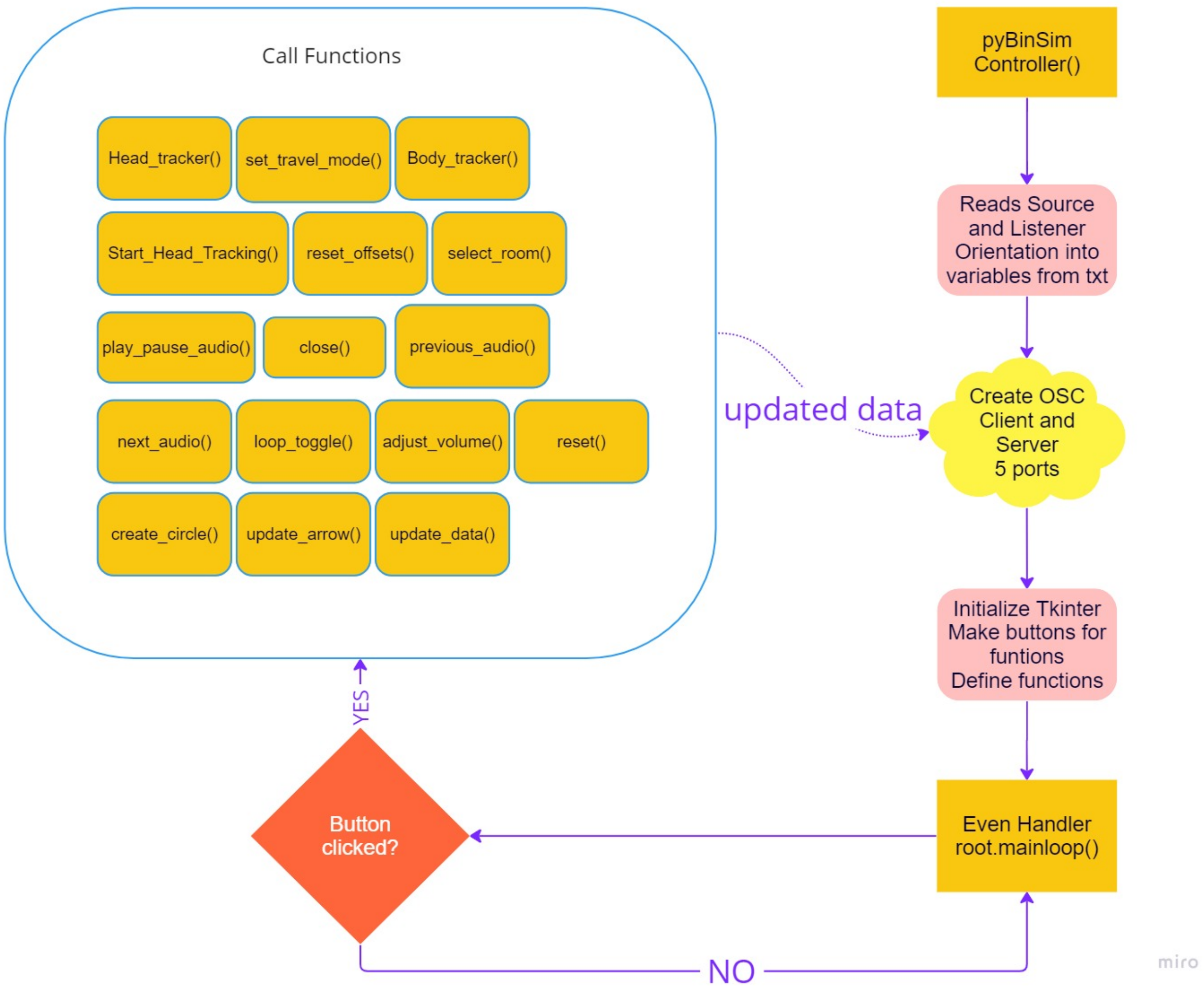
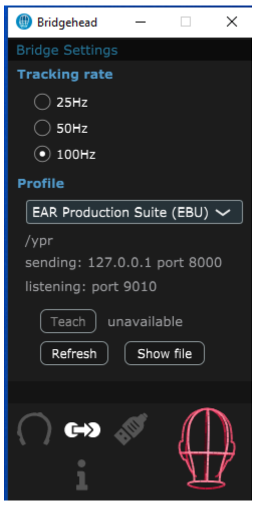


Figure 6.6: Event handling with UI

clicks the button. Tkinter uses an event-driven handler for this purpose. It has a main loop that keeps checking for button presses and selects the specific function behind it. Depending on the function called, the OSC message requests are sent to the pyBinSim processor via the OSC client. The figure 6.6 depicts this process.

6.3 System Design

To facilitate seamless interaction with the pyBinSim, we have developed a dedicated pyBinSim controller. This controller is designed to execute specific functions in response to user input, triggered by button clicks from the listener.



Bridge Settings

Tracking rate

25Hz
50Hz
[✓ 100Hz]

Profile

EAR Production Suite (EBU)

/ypr
sending: 127.0.0.1 port 8000
listening: port 9010

unavailable

Figure 6.7: Bridgehead Head Tracker

Head tracker

The data coming from the head sensor is received via an OSC server listening to the port 8000 and assigned it to global variables. We have used the head tracker made by Supperware[24], which has an interface as shown in Figure 6.7. This sensor sends the data with OSC protocol. We have used the inbuilt profile named Ear Production Suite (EPS) and data is received at 100Hz. This sensor also gives the flexibility to reset the head orientation since it has its own OSC server which listens at port 9010. The OSC server runs in a separate thread to prevent it from blocking the main thread. Once started, the OSC server continuously listens for incoming messages, and the function handles the received data. The listening port can change if a different profile is selected, so it is important to use the EPS profile in our case.

Body tracker

The RealSense pipeline gets the pose data from a RealSense device, which is placed on the backpack system. A stop event is used to terminate the function and stop the pipeline. The received data is processed to calculate the yaw, pitch, and roll values in degrees. The body tracker function runs continuously until the stop event is set. A separate thread is created to run the body tracker function so that it can run in the background without blocking the main thread. The start() method is used to start the pipeline to stream the pose data, and the stop() method is called to stop the pipeline when the function is terminated using the stop event.

Adjust volume

Based on the values of the slider between 1 and 15 from the UI, it changes the volume level of audio playback.

Body tracker toggle

This enables or disables the dynamic body tracking of the listener's movements.

Start Head Tracking

Activates head tracking to start dynamic binaural audio rendering based on the listener's head orientation and updates the data and visualization of the head orientation in the UI.

Set travel mode

On the move, the head tracker can be re-centered automatically when the listener changes head direction. The travel modes are designed to keep the listener's head automatically aligned if they are in a moving vehicle, so their reference frame changes as it moves about. Both travel modes will wait until the listener is a certain angle off-axis for a certain period of time (so glances around will not cause it to take effect) and then start pulling the yaw angle back towards zero degrees. The difference is that the fast mode is tuned to take effect more readily and corrects by about 10 degrees per second, while the slow mode corrects at a rate of about 4 degrees per second.

Reset

It sends an OSC message to the head sensor to re-center itself and update the data in pyBinSim accordingly.

Reset offsets

It resets and re-centers the body position of the listener.

Pause play audio

Pauses or resumes audio playback.

Loop toggle

Enables or disables looping of the current audio track or playlist.

Next audio

Skips to the next audio track in the current playlist or sequence.

Prev audio

Returns to the previous audio track in the current playlist or sequence.

Select room

As of now, we have H1539, H2505, HL, and ML2_102 rooms. Based on the synthesized BRIRs database, room files are indexed, and separate buttons are made in the UI to call for different room environments. If a room is selected, the appropriate filter database file path is fed to the pyBinSim processor. The processor takes the file path, parses, and loads the variables again based on the new filter value list, and at last, updates the required filters. Since we have also updated pyBinSim to receive room change requests based on the path of the room filter database, it becomes easy to scale it to many rooms without having to load all the database in the variables and use them.

Update data

This function calculates the current yaw, pitch, and roll values, normalizing them to a specific range. By selecting the nearest available data from a given set of angles, the function feeds the data to the processor for the binaural rendering. For each audio channel, this function prepares and sends the appropriate parameters to update the direct sound, early reflection, and late reflection filters accordingly. Since the data from the head and body sensor is received at a high rate, it is not possible to update pyBinSim this frequently. We selected a time delay of 0.25 seconds so that there are no audible artifacts while rendering in real time.

Visualization of head tracking

The UI features an interactive circular display representing the head's orientation in the yaw axis, complete with angle markers and a dynamic arrow that responds to changes in the listener's position. As the listener moves their head, the arrow updates its position accordingly, providing a clear and intuitive visualization of the head tracking process. Additionally, a label near the center of the circle indicates the current yaw angle, enhancing the user's understanding of their head orientation. This visual real-time feedback enables users to monitor their real-time head movements in the virtual world, as shown in Figure 6.5.

6.4 Room Database

We have used the SDM rendered BRIRs for four rooms as described below:

Listening Lab (HL)

An 82 m^2 room with a RT of 0.255 seconds. It meets the acoustic requirements of ITU-R BS.1116-3 standard [28] and is designed for optimal sound reproduction as shown in 6.8.

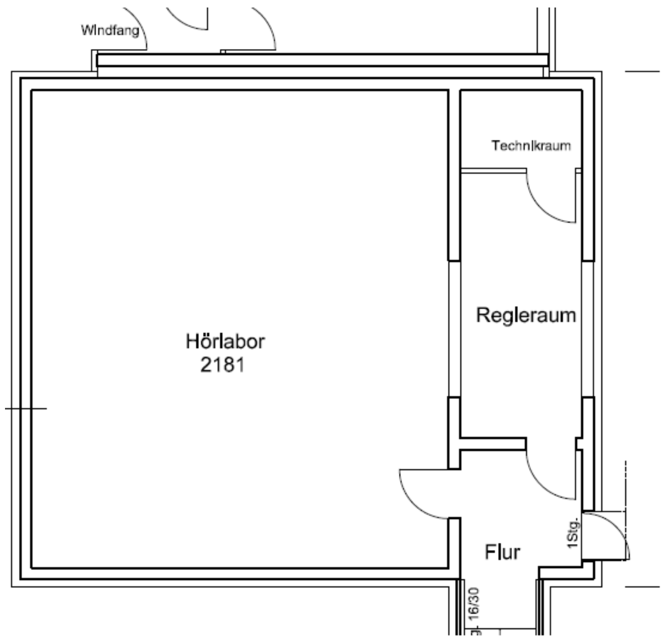


Figure 6.8: Listening Lab (HL)

Editorial room Media Lab 2 (ML2-102)

This 32 m^2 room has a RT of 0.52 seconds. It features a quadratic shape, an optimized ceiling, and two window fronts, providing an ideal environment for editing and media-related tasks as shown in 6.9.

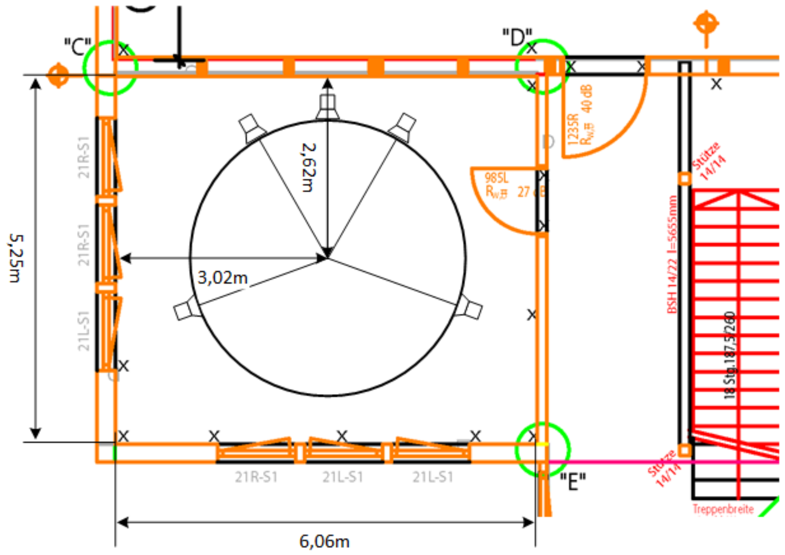


Figure 6.9: Editorial room Media Lab 2 (ML2-102)

Seminar room (H2505)

With a size of 18 m^2 and a RI of 1.2 seconds, this compact room is used for seminars and presentations. It offers balanced acoustics for effective communication as shown in 6.10.

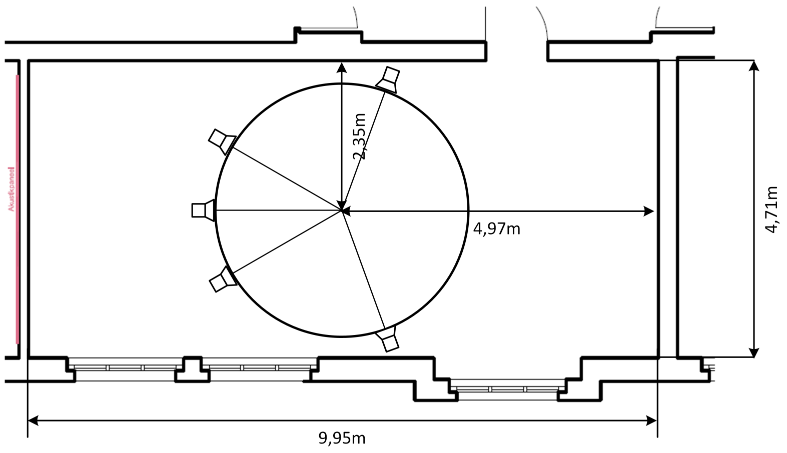


Figure 6.10: Seminar room (H2505)

Audio/Video Lab (H1593b)

Spanning 27 m^2 with a RT of 0.28 seconds, this slightly asymmetric lab is optimized for audio and video production. It ensures low reverberation and favorable acoustic properties for accurate recordings as shown in 6.11.

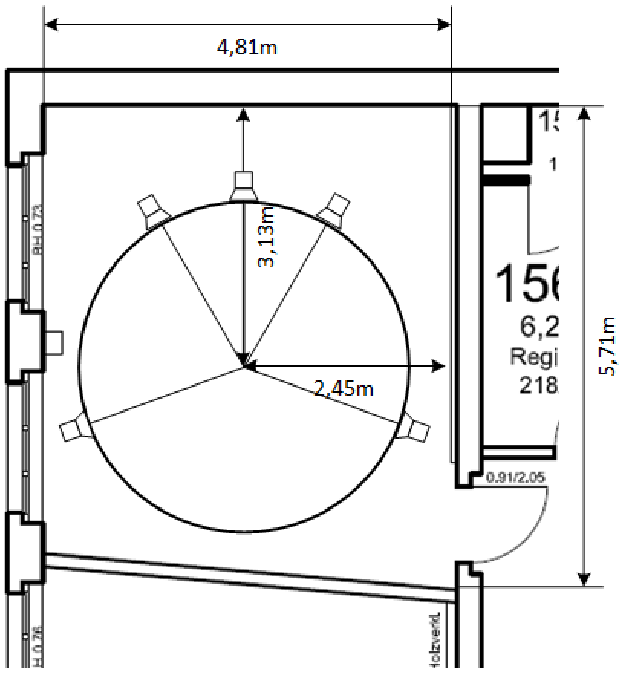


Figure 6.11: Audio/Video Lab (H1593b)

7 Experimental Setup and Validation

The experiment was structured to test the listener's perception of sound and spatial attributes in different conditions. The experiment was broken into three stages, with the listener's head and body movements gradually incorporated into the assessment.

This validation test was conducted in Listening lab of TU Ilmenau. To ensure stability during movement and prevent any offset in body movements, the RealSense camera was securely attached to the top of the Backpack PC. The Backpack PC itself was powered by external batteries, providing the necessary mobility for the experiment. The head sensor was connected to the Backpack PC, enabling the measurement and tracking of head movements.

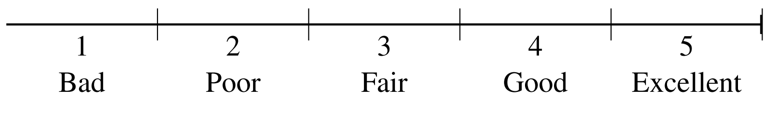
7.1 Listening Test Procedure

Prior to commencing the main stages of the test, an initial assessment was conducted to ensure the listeners were comfortable with the sound loudness. A square area was marked in the room within which listeners could move and interact with the system during the subsequent stages.

Four different sound files were used for the test:

- `C_5ch-48k-16b--ID-male` (5 channel, centre speaker only)
- `FL_FR_5ch-48k-16b--ID-male` (5 channel, Front left and front right speaker)
- `BL_BR_5ch-48k-16b--ID-male` (5 channel, Back left and Back right speaker)
- `orig_mechanism` (5 channel immersive audio)

Listeners were asked to rate the perceived quality of sound using the following scale:



Stage 1: Room Change Detection and Loudness Check

In the first stage, the listeners were asked the following questions while standing at a position with no head tracking and no body tracking on:

- Can you perceive a change when I modify the virtual room? (Yes/No)

- How is the audio source perceived in the room? (Quality) (1-5)
- How well can you separate individual sound sources? (Localization) (1-5)

Stage 2: Test with head tracking on

For the second stage of the experiment, the head tracking feature was activated. Listeners were then asked to freely move their heads and walk within the marked square, responding to the following questions:

- How would you rate the stability of sound sources as you move your head? (Stability) (1-5)
- Did you notice any delay between your head movement and the change in sound? (Latency) (1-5)
- How accurately could you localize the sound source while moving your head? (Localization) (1-5)
- How would you rate the externalization of sound with head movement? (Envelopment) ((a) in-head (b) very close to the head or eyes (c) in the room)

Stage 3: Test with both head and body tracking

The final stage of the experiment incorporated both the head tracking and body tracking. The listeners were asked to move freely, both their heads and bodies, within the marked square. They were asked the following questions:

- How would you rate the stability of sound sources as you move your body? (Stability) (1-5)
- Did you notice any delay between your body movement and the change in sound? (latency) (1-5)
- As you move your body, do the sound sources seem to move with you in the azimuthal plane? (System move with you) (1-5)
- How accurately could you localize the sound source while moving your body? (Envelopment) (1-5)

7.2 Results

A total of 5 subjects participated in the validation test. The loudness level was set to 10 for all subjects in all the stages.

7.2.1 Stage 1: Room Change Detection and Loudness Check

In the first stage, all subjects were able to perceive when the virtual room was changed to which everyone agreed. The violin plot for the quality and localization, when head and body trackers are not turned on, are shown in Figure 7.1.

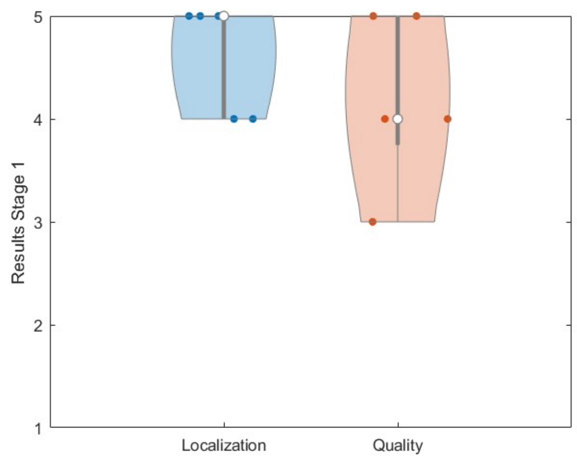


Figure 7.1: violin plot - Stage 1

7.2.2 Stage 2: Test with Head Tracking On

During the second stage, all subjects reported feeling as if they were sitting inside a room. The results for the stability of sound sources, latency between head movement and sound change, and the accuracy of sound source localization are presented in the figure 7.2.

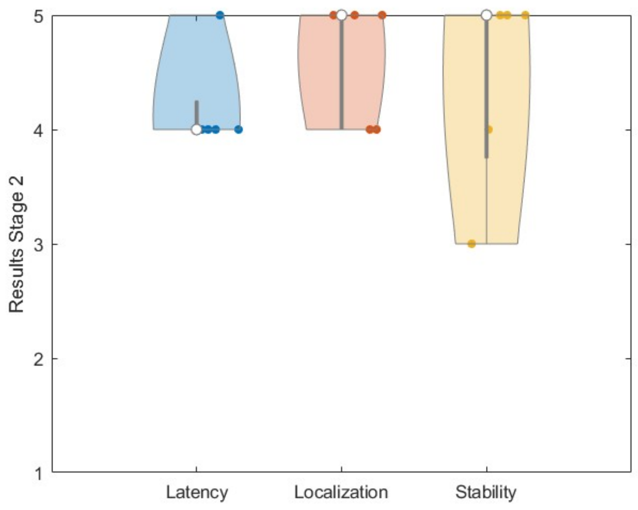


Figure 7.2: violin plot - Stage 2

7.2.3 Stage 3: Test with Head and Body Tracking

In the third stage, all subjects reported feeling as if they were sitting inside the room, with the sound sources moving accordingly. The results for the stability of sound sources, latency between body movement and sound change, the perception of sound source movement with the body, and the accuracy of sound source localization are shown in the figure 7.3.

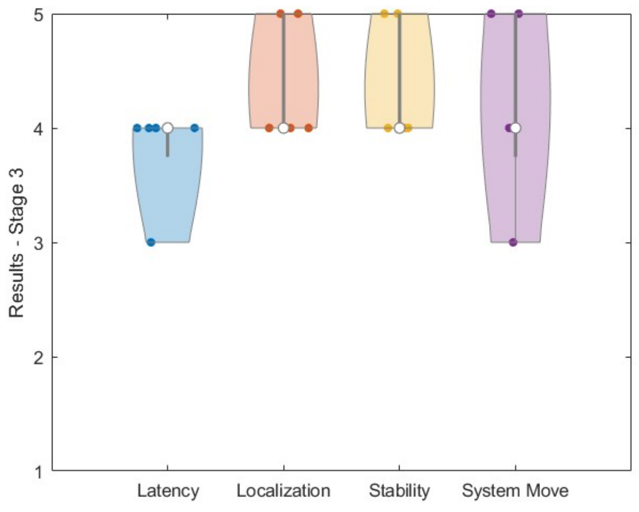


Figure 7.3: violin plot - Stage 3

At last subjects were asked to rate the over all experience using the system. The result is shown in 7.4

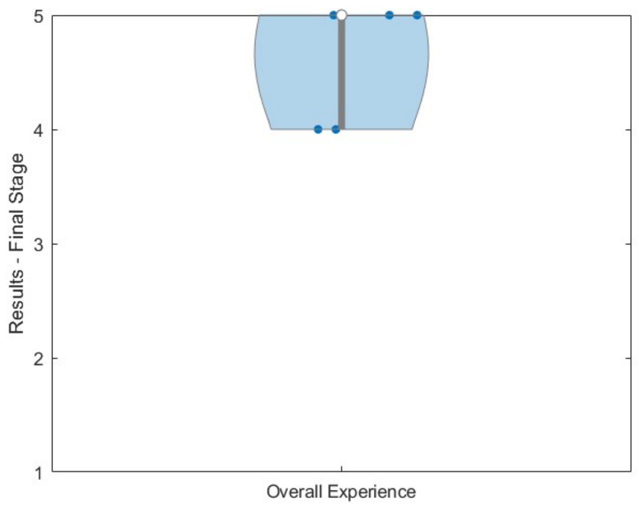


Figure 7.4: violin plot - Final Stage

8 Documentation

In this chapter, we will discuss the experimental setup and design decisions, project structure, event handling, scalability, portability, OSC communications, hardware, and software requirements in more detail.

8.1 Hardware and Software description

8.1.1 Hardware Specifications

The system specifications used in the making of this project are as follows:

System Information

Time of this report:	4/25/2023, 12:41:22
Machine name:	BACKPACK1
Operating System:	Windows 10 Enterprise 64-bit (10.0, Build 19045)
System Manufacturer:	HP
System Model:	HP Z VR Backpack G1 Workstation
BIOS:	F.41 (type: UEFI)
Processor:	Intel(R) Core(TM) i7-7820HQ CPU @ 2.90GHz (8 CPUs), ˜2.9GHz
Memory:	32768MB RAM
Available OS Memory:	32692MB RAM
DirectX Version:	DirectX 12

Display Devices

Card name:	NVIDIA Quadro P5200
Manufacturer:	NVIDIA
Chip type:	Quadro P5200
DAC type:	Integrated RAMDAC
Device Type:	Full Device (POST)
Display Memory:	32593 MB
Dedicated Memory:	16248 MB
Shared Memory:	16345 MB

8.1.2 Software Specifications

Below is the list of software along with the version number used in this project:

Software Used	Versions
Visual Studio Code	1.77.3
Python	3.9
Mathworks Matlab	R2022b
Bridgehead	1.19
Visual Studio Code	1.77.3
Anaconda	22.9.0

8.2 Experimental setup

In our experiment, we set up our system in the listening lab, where we developed and implemented it. One key aspect of our project is addressing mobility and portability concerns. Since the system can be carried in a backpack, we integrated the Chrome Remote Desktop plug-in for the Google Chrome web browser into our setup. This allowed us not only to connect to a monitor screen using an HDMI cable but also to access the system without a monitor, providing more flexibility and convenience in various situations. Furthermore, Chrome Remote Desktop can also be used from a smartphone, tablet, or laptop, on any operating system where Google Chrome can be installed. Another approach could be to incorporate a wearable screen for accessing the system; however, this would be a more expensive option.

8.3 Folder structure and Matlab Script

In this project, we are using the following folder structure:

Folder Structure

- **project_root**
  - 01_SDM_IR_RecTool
  - 02_SDM_BRIR_QuantizedDOA
    - App_BinauralSDM_QuantizedDOA_fixPos_BinSim.m
    - functions
  - 03_BinSim_fixPos
    - UI_for_pyBinSim.py
    - pyBinSimSetting_SourcesListenerDefs.txt
    - pyBinSimSettings_isoperare.txt
  - data

    * HPIRs
    * HRIRs
    * RIRs
    * SDMRenderedBRIRs
    * Signals
  - SDMtools

Generate SDM Rendered BRIRs using Matlab Script

SDM rendered BRIRs are generated using App_BinauralSDM_QuantizedDOA_fixPos_BinSim.m. To run this script, several Matlab packages must be installed.

SDM Toolbox : The SDM Toolbox is a collection of Matlab functions and scripts for spatial room impulse response analysis and synthesis using the Spatial Decomposition Method [15]. It contains several standard default loudspeaker setups, such as 5.1, which can be used directly.

Signal Processing Toolbox : The Signal Processing Toolbox^{TM}offers functions and apps for managing, analyzing, pre-processing, and extracting evenly and unevenly sampled signals. The toolbox contains tools for filter design and analysis, resampling, smoothing, trend elimination, and power spectrum estimation [29].

Statistics and Machine Learning Toolbox : This toolbox includes functions and apps for describing, analyzing, and modeling data. Descriptive statistics, visualizations, and clustering are available for exploratory data analysis [30].

After installing the required packages, the script can be run. It prompts the user to select an input file from the set of RIRs in the folder structure. Once the room is selected, the script outputs the rendered BRIRs into the SDMRenderedBRIRs folder inside the data folder. Additionally, it generates two text files named pyBinSimSetting_SourcesListenerDefs.txt and pyBinSimSettings_isoperare.txt in the 03_BinSim_fixPos folder 8.3.

8.4 Running pyBinSim

In order to run the PyBinSim processor 6.2.1, we need to install the PybinSim library. We installed it using the pybinsim-ConvolverMultisourceMat folder, but it can also be installed in an Anaconda environment.

Installing Pybinsim $ conda create –name binsim python=3.9 numpy
$ conda activate binsim

$ pip install pybinsim

After installing pyBinSim, it can be initialized using the ”pyBinSimSettings.txt” file, which is generated by the Matlab script in the 03_BinSim_fixPos folder, as shown below:

Algorithm 1 Python code to start pyBinSim

```
1: Import pybinsim
2: Import logging
3: pybinsim.logger.setLevel(logging.DEBUG)        ▷ defaults to INFO
4: SetLogLevel WARNING                            ▷ for printing warnings only
5: procedure BINSIM(pyBinSimSettings.txt)
6:     binsim.stream_start()
7: end procedure
```

This initializes the pyBinSim processor, which yields output in the terminal as shown in 8.1.

```
- __init__.py:24 - INFO - Starting pybinsim v1.2.4
- application.py:132 - INFO - BinSim: init
- application.py:196 - INFO - Block size smaller than direct
   ↪ sound filter size: Zero Padding DS filter
- filterstorage.py:127 - INFO - FilterStorage: init
- filterstorage.py:189 - INFO - Loading mat format filters
- filterstorage.py:199 - INFO - Loading mat variable : binsim_1
- filterstorage.py:199 - INFO - Loading mat variable : binsim_2
- filterstorage.py:199 - INFO - Loading mat variable : binsim_3
- filterstorage.py:199 - INFO - Loading mat variable : binsim_4
- filterstorage.py:199 - INFO - Loading mat variable : binsim_5
- osc_receiver.py:39 - INFO - oscReceiver: init
- osc_receiver.py:346 - INFO - Serving on ('127.0.0.1', 10000)
- osc_receiver.py:352 - INFO - Serving on ('127.0.0.1', 10001)
- osc_receiver.py:358 - INFO - Serving on ('127.0.0.1', 10002)
- osc_receiver.py:364 - INFO - Serving on ('127.0.0.1', 10003)
```

Listing 8.1: pyBinSim output in Terminal

This runs in an infinite loop in the terminal. The pyBinSim controller can control it using OSC messages. A separate script can also be made to run on a different terminal to control pyBinSim via OSC messages 6.3. A simple explanation for this can be the use of keybindings, e.g., if the listener presses the space key, it toggles the audio playback. However, it is not possible to move around from one place to another with the keyboard, mouse, and even monitor screen in hand. An on-screen keyboard or wireless mini keyboard can be an alternative solution for this problem using a portable screen attached to the system, but it is difficult to control pyBinSim in two terminals alongside using virtual or real keyboards. We have solved this problem in two parts. Rather than using a keyboard and screen, we are using Chrome Remote Desktop and a User Interface. With this, the listener can securely access their system on the go, using a phone, tablet, or another computer. To solve the problem of two terminals, a better solution can be to use the multiprocessing capabilities of the PC and run in the same script, as shown below with the help of the multiprocessing Python library [22] as depicted in Algorithm 2.

Algorithm 2 Using multiprocessing with pyBinSim

```
if __name__ == '__main__' then
    Initialize process1 ← Process(target=pyBinSim_Controller)
    Initialize process2 ← Process(target=pyBinSim_Processor)
    process1.start()
    process2.start()
    process1.join()
    process2.join()
end if
```

8.5 OSC Messages for pyBinSim

After pyBinSim initializes the OSC servers, which is being done here by osc_receiver.py, pyBinSim processor can be controlled using particular sets of OSC messages as defined below. Each of these string commands is mapped to a corresponding handler function using the Python OSC dispatcher. After slicing the filter value list, the data handlers are mapped to these string commands for different direct sound, early reflection, and late reflection filters. These functions process the tracking data and apply the required filters using dictionary key-value pair values in the PyBinSim processor.

Early reflection filters:

- /pyBinSim_early_Filter
- /pyBinSim_early_Filter_Short
- /pyBinSim_early_Filter_Orientation
- /pyBinSim_early_Filter_Position
- /pyBinSim_early_Filter_Custom
- /pyBinSim_early_Filter_sourceOrientation
- /pyBinSim_early_Filter_sourcePosition

Late reverberation filters:

- /pyBinSim_late_Filter
- /pyBinSim_late_Filter_Short
- /pyBinSim_late_Filter_Orientation
- /pyBinSim_late_Filter_Position
- /pyBinSim_late_Filter_Custom
- /pyBinSim_late_Filter_sourceOrientation
- /pyBinSim_late_Filter_sourcePosition

Miscellaneous commands:

- /newroom
- /pyBinSimFile

- /pyBinSimPauseAudioPlayback
- /pyBinSimloopAudio
- /pyBinSimPauseConvolution
- /pyBinSim_sd_Filter
- /pyBinSimLoudness
- /HeadphoneFilter
- /close

We have added the /newroom string command to change the acoustic environment by selecting different rooms from the new filter database path received by it and the /close command to close the pyBinSim by clearing all the buffers, closing filters, and exiting the terminal. Below are the descriptions for the handler functions used by them.

Algorithm 3 Handling new room file

```
Require: $identifier$ (string), $roompath$ (string)
Ensure: Sets the room path in the current configuration and updates the room file list
1: function HANDLE_ROOM_FILE_INPUT($self, identifier, roompath$)
2:     assert $identifier \equiv$ /newroom
3:     self.currentConfig.set('filterDatabase', roompath)
4:     self.log.info("RoomPath:  ".format(roompath))
5:     self.roomFileList $\leftarrow roompath$
6: end function
```

After getting the update request for the new room file, it then, in the main application.py of the pyBinSim library, iterates through all the channels from 0 to nChannels - 1. For each channel, it retrieves the current filter values for DS, ER, and LR. Then, it obtains the corresponding filters for each filter value list using the filterStorage object. In this way, filters are applied to the corresponding convolvers for each channel as shown in 8.2.

```
current_roomfile_list = binsim.oscReceiver.get_room_file_list()
if current_roomfile_list:
    binsim.filterStorage.request_new_room_file(
       ↪ current_roomfile_list)
    # Update Filters and run each convolver with the current
       ↪ block
    for n in range(binsim.nChannels):

        # Get new Filter
        ds_filterValueList = binsim.oscReceiver.
           ↪ get_current_ds_filter_values(n)
        ds_filter = binsim.filterStorage.get_ds_filter(Pose.
           ↪ from_filterValueList(ds_filterValueList))
        binsim.ds_convolver.setIR(n, ds_filter)

        # Get new early reverb Filter
        early_filterValueList = binsim.oscReceiver.
           ↪ get_current_early_filter_values(n)
        early_filter = binsim.filterStorage.get_early_filter(
           ↪ Pose.from_filterValueList(early_filterValueList))
        binsim.early_convolver.setIR(n, early_filter)

        # Get new late reverb Filter
        late_filterValueList = binsim.oscReceiver.
           ↪ get_current_late_filter_values(n)
        late_filter = binsim.filterStorage.get_late_filter(Pose
           ↪ .from_filterValueList(late_filterValueList))
        binsim.late_convolver.setIR(n, late_filter)
```

Listing 8.2: Updating filters in application.py

Since we have to close pyBinSim processor and receiver together using the UI, we made a close handler function 8.3 inside the pyBinSim processor as well. Hence before closing the UI, an OSC message is sent to the pyBinSim processor to terminate its run.

```
    def handle_shutdown(self, identifier, value):

        assert identifier == "/close"
        if value == 'True' :
            self.close_var = True
            print("Closing....")
```

Listing 8.3: Handling close command

After this, in application.py of the pyBinSim, all the functions that are currently running are closed as depicted by algorithm 4.

Algorithm 4 Closing pyBinSim processor

```
Require: $close\_var$ (boolean)
Ensure: Terminates the pyBinSim application by closing all the buffers, filters, and convolvers
 1: function CLOSE_PYBINSIM_PROCESSOR($self, close\_var$)
 2:     if $close\_var$ then
 3:         self.oscReceiver.terminate()
 4:         self.stream.stop_stream()
 5:         self.stream.close()
 6:         self.p.terminate()
 7:         self.filterStorage.close()
 8:         self.ds_convolver.close()
 9:         self.early_convolver.close()
10:         self.late_convolver.close()
11:     end if
12: end function
```

When the user sends the close command, the close handler function in the pyBinSim OSC receiver sets the close_var to True. In the main application.py, the close_pybinsim_processor function checks if the close_var is True. If it is, the function terminates the pyBinSim application by closing all the buffers, filters, and convolvers.

In conclusion, we have now implemented the necessary handler functions to update the room file, apply the filters for direct sound, early reflections, and late reverberations in the pyBinSim processor, and close the application. By integrating these handler functions with the UI, we can now efficiently control the pyBinSim processor through the UI, allowing for a more seamless and user-friendly experience.

8.6 Data Handling in pyBinSim processor

We are getting two senor Inputs from the body and head continuously at fast rates. With head sensor sending data at 100 Hz and body sensor sending data at 200 Hz, it becomes crucial to handle both the data, apply calculation, and input it to pyBinsim processor.

For the head sensor the data is coming via OSC protocol. The bridgehead sends the data to the OSC server of pyBinSim controller.After receiving the data from sensor , it is assigned to the global variables respectively as shown in 8.4 to be used for further calculation.

```
def Head_tracker(address, *args):
    global yaw_sensor, pitch_sensor, roll_sensor
    yaw_sensor = args[0]
    pitch_sensor = args[1]
    roll_sensor = args[2]

dispatcher = Dispatcher()
dispatcher.map("/ypr", Head_tracker)
def start_osc_server(stop_event):
    server =  osc_server.ThreadingOSCUDPServer(("
       ↪ localhost", 8000), dispatcher)
    while not stop_event.is_set():
        server.handle_request()
    server.server_close()

# Create a stop event to signal the OSC server thread
   ↪ to stop running
stop_event = threading.Event()

# Start_Head_Tracking the OSC server thread
osc_thread = threading.Thread(target=start_osc_server,
   ↪ args=(stop_event,))
osc_thread.start()
```

Listing 8.4: Collecting bridgehead sensor data in pyBinSim controller

Body pose data is collected from the real sense tracker t265. It has its own library named pyrealsense2 which we imported as alias rs. Values from the sensor acts as an offset to the values for head tracking data, as we want the system to move with the subject 6.2.

```
        def Body_tracker(stop_event):
            # Declare RealSense pipeline, encapsulating the
               ↪ actual device and sensors
            global yaw_offset_rs,pitch_offset_rs,yaw_out_rs,
               ↪ pitch_out_rs,yaw_rs,pitch_rs
            pipe = rs.pipeline()
            # # Build config object and request pose data_rs
            cfg = rs.config()
            cfg.enable_stream(rs.stream.pose)
            # # Start streaming with requested config
            pipe.start(cfg)
            # Offsets for OSC values
            pitch_rs = 0
            yaw_rs = 0
            try:
                while not stop_event.is_set():
                    time.sleep(0.01)
                    frames = pipe.wait_for_frames()
                    pose = frames.get_pose_frame()
                    data_rs = pose.get_pose_data()
                    w = data_rs.rotation.w
                    x = -data_rs.rotation.z
                    y = data_rs.rotation.x
                    z = -data_rs.rotation.y
                    pitch_rs = -m.asin(2.0 * (x * z - w * y)) *
                       ↪  180.0 / m.pi
                    yaw_rs = m.atan2(2.0 * (w * z + x * y), w *
                       ↪  w + x * x - y * y - z * z) * 180.0 /
                       ↪  m.pi
                    pitch_current_rs = (pitch_rs -
                       ↪ pitch_offset_rs)
                    yaw_current_rs   = (yaw_rs   -
                       ↪ yaw_offset_rs)
                    yaw_out_rs = yaw_current_rs
                    pitch_out_rs = pitch_current_rs
            except KeyboardInterrupt:
                pipe.stop()
```

Listing 8.5: Collecting realsense sensor data in pyBinSim controller

In this code 8.5, we have used yaw_out_rs and pitch_out_rs as the variables that will yield the value of current body position output. Unlike head sensor , we do not have a reset in body sensor, so we have to do reset manually using the code 8.6. We subtract the offset values from the current values to yield the output of the body pose.

```
            def reset_offsets():
            global yaw_offset_rs, pitch_offset_rs
            yaw_offset_rs = yaw_rs
            pitch_offset_rs = pitch_rs
```

Listing 8.6: Resetting the audio by storing the current values to the offsets

To proceed, it is necessary to update the data from both sensors and transmit the OSC message to the pyBinSim processor. To achieve this, we have calculated the difference between the values obtained from the head and body sensors. Furthermore, we have ensured that the values range between -180 to 180. We have also noted that the yaw values of the head sensor are positive when rotated counterclockwise, whereas for the body sensor, it is the opposite. Therefore, we must consider the respective signs to maintain the source position in the azimuthal plane constant. The code for updating the data is provided in 8.7.

```
        def update_data():
            global body_orientation
            if body_orientation:
                yaw_current   = -(yaw_sensor + yaw_out_rs )
                pitch_current = (pitch_sensor - pitch_out_rs)
            else:
                yaw_current   = -(yaw_sensor )
                pitch_current = (pitch_sensor)
            #Making range from -180 to 180
            if yaw_current >= 180:
                        yaw_current = yaw_current-360
            if pitch_current >= 180:
                        pitch_current = pitch_current-360

            if yaw_current <= -180:
                         yaw_current = yaw_current+360
            if pitch_current <= -180:
                        pitch_current = pitch_current+360
            # Choose nearest available data
            yaw_out  = min(availableAngles_yaw, key=lambda x:
               ↪ abs(x - yaw_current))
            pitch_out  = min(availableAngles_pitch,default=0,
               ↪ key=lambda x: abs(x - pitch_current))

            for iChan in range(0, numChan):
                # send data to switch ds and early filter

                binSimParameters_ds = [iChan, yaw_out,
                   ↪ pitch_out, 0, listenerPosition[0],
                   ↪ listenerPosition[1], listenerPosition[2],
                   ↪  sourceOrientation[iChan][0],
                   ↪ sourceOrientation[iChan][1],
                   ↪ sourceOrientation[iChan][2],
                   ↪ sourcePosition[iChan][0], sourcePosition[
                   ↪ iChan][1] ,sourcePosition[iChan
                   ↪ ][2],0,0,0]
                binSimParameters_early = [iChan, yaw_out,
                   ↪ pitch_out, 0, listenerPosition[0],
                   ↪ listenerPosition[1], listenerPosition[2],
                   ↪  sourceOrientation[iChan][0],
                   ↪ sourceOrientation[iChan][1],
                   ↪ sourceOrientation[iChan][2],
                   ↪ sourcePosition[iChan][0], sourcePosition[
                   ↪ iChan][1] ,sourcePosition[iChan
                   ↪ ][2],0,0,0]
                oscClient_ds.send_message("/pyBinSim_ds_Filter"
                   ↪ , binSimParameters_ds)
                oscClient_early.send_message("/
                   ↪ pyBinSim_early_Filter",
                   ↪ binSimParameters_early)
```

Listing 8.7: Updating data to send to pyinSim processor

The offsets are calculated and transmitted to the OSC server of the pyBinSim processor via OSC string commands. These commands call the necessary handler functions, which update the audio callback.

8.7 Designing User Interface using Tkinter

At the beginning of the project, we identified all the essential functionalities required for the project. This helped us stay on track and create UI functions accordingly.

We decided to use Tkinter, which is based on the Tool Command Language programming language. It is a standard library with a user-friendly interface [27]. It is event-driven and executes the corresponding function when a button is pressed. In addition, it has custom variants such as Custom Tkinter, which we utilized in our project to create the UI window. An example of creating a simple head tracking button is shown in 8.8.

```
import customtkinter as ctk
import tkinter as tk
root = ctk.CTk()
root.configure(bg='#F5F5F5')
root.title("Pybinsim␣Control")
root.protocol("WM_DELETE_WINDOW", Close)

# Defining a common frame

frame = tk.Frame(root)
frame.pack(fill="both", expand=True)


# Start_Head_Tracking

sensor_frame = tk.Frame(frame)
sensor_frame.pack(fill="both")
sensor_frame = tk.Button(sensor_frame, text="Start␣Head
   ↪ ␣Tracking", command=Start_Head_Tracking)
sensor_frame['background'] = '#BDFCBF'
sensor_frame.pack(side="right", fill="both", expand=
   ↪ True,padx=18, pady=18)
```

Listing 8.8: Making a start head tracking button in Tkinter,float,floatplacement=H

The above-mentioned button has the text ”Start Head Tracking”. When clicked, it calls the function ”Start_Head_Tracking()”, as described in 8.9. We have ensured proper feedback, so the user knows if they have clicked the button and what the current status of the button is. Additionally, feedback in the form of a text field is available to describe the current status of the pyBinSim processor.

```
def Start_Head_Tracking():
    global stop_loop
    if sensor_frame["text"] == "Start␣Head␣Tracking":
        # Start the animation
        for i in range(3):
            sensor_frame.config(background="#FF0000")
            root.after(10, sensor_frame.config, {"
               ↪ background": "#BDFCBF"})
            root.after(10, sensor_frame.config, {"
               ↪ background": "#FF5733"})
        sensor_frame.config(text="Stop␣Head␣Tracking")
        label.config(text="Head␣Rotation␣Angle")
        stop_loop = False
        while sensor_frame and sensor_frame["text"] ==
           ↪ "Stop␣Head␣Tracking" and not stop_loop:
            update_data()
            update_arrow(yaw_sensor)
            root.update()
            time.sleep(0.25)
    else:
        sensor_frame.config(text="Start␣Head␣Tracking")
        root.after(0, sensor_frame.config, {"background
           ↪ ": "#BDFCBF"})
        label.config(text="Head␣Tracking␣Off")
```

Listing 8.9: Start head tracking function

Similarly, we have functions for all the buttons in the UI 6.5 that run in the background and update the pyBinSim processor.

To close the pyBinSim processor and controller together, one can simply close the UI. The function mentioned in 8.10 handles the rest automatically.

```
def Close():
    global stop_loop
    stop_loop = True
    oscClient_misc.send_message("/close", str("True"))
    stop_event.set()  # Set the stop event to signal
       ↪ the OSC server thread to stop running
    stop_event_rs.set()
    rs_thread.join()
    osc_thread.join()  # Wait for the OSC server thread
       ↪ to stop running
    root.quit()  # Quit the main window
```

Listing 8.10: Function for closing both pyBinSim processor and controller with UI

8.7.1 Visualizing the Head Movements in the Azimuthal Plane

The Bridgehead software has a UI that displays the current pose of the user's head. We were inspired by this and came up with the idea of creating a head movement visualizer in the azimuthal plane. We defined a circle with values ranging from -180 to 180, with a resolution of 20°. An arrow represents the current position of the head in the azimuthal plane, with the current angle value at the center. This visualization allows the user to easily assess their orientation and adjust it if necessary. If the initial orientation is incorrect, resetting the orientation will cause the arrow to point at the origin again. This allows the listener to not only judge the angle from the audio playback but also see their current head pose in the virtual world with virtual speakers, as shown in figure 8.1.

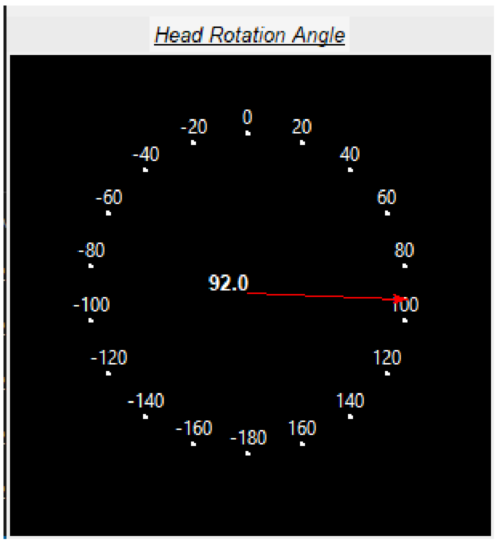


Figure 8.1: Dynamic yaw value indicator in UI

8.8 Scalability and Portability

As of now, we have only used four rooms, but the database can be expanded as much as desired. Since we have indexed the SDM rendered BRIRs files, we can easily create and send file names with string commands to the pyBinSim processor using the buttons, as shown in 8.11.

The drawback of this method is the slight delay caused by the time taken to load and update the filters. This delay may result in some artifacts in the audio playback while changing the filters, but these can be eliminated by adding a time delay. The advantage of sending the filenames is memory management. If the database of rooms is large, loading all the data into Python variables before processing would take up a significant amount of memory. Although updating them using custom values is possible (the last three digits in pyBinSim-Setting_SourcesListenerDefs.txt), our method is better since it is more scalable for large room databases.

We are using a backpack system that can be carried on the go by the listener. We tried using Chrome Remote Desktop to operate the system from a different laptop, as well as from handheld devices. The UI can be easily operated with touch gestures on handheld devices, making it more convenient while traveling. The Bridgehead sensor is integrated into the Beyerdynamic headphones that we used in the listening lab, and the body sensor is placed on top of the backpack. As the backpack remains stationary with respect to the body position while traveling, both sensors continuously send data to the backpack system, which is powered by batteries. It was also observed that low battery power can affect the rendering in pyBinSim, as it uses multiprocessing and consumes a lot of power.

```
            def select_room(selected_room_file):
                oscClient_misc.send_message("/newroom", str(
                   ↪ filter_database_path+room_files[
                   ↪ selected_room_file]))
                current_room_label.config(text="Current␣Room:␣"
                   ↪  + room_files[selected_room_file].replace
                   ↪ ("_5LS_M_binsim_struct.mat", "").replace(
                   ↪ "SDM_", ""), font=("Arial", 8,"bold"),fg=
                   ↪ "black")

        def H1539():
                select_room(0)
        def H2505():
                select_room(1)
        def HL():
                select_room(2)
        def ML2_102():
                select_room(3)
```

Listing 8.11: Change rooms with help of buttons in UI

9 Discussion and Outlook

9.1 Scope of the Project

In this media project, we have successfully developed and evaluated a dynamic binaural synthesis system that virtually reproduces a 5-channel surround speaker setup over headphones. The system overcomes the limitations of traditional headphone listening experiences by accounting for both horizontal and vertical head movements, as well as incorporating body sensors to maintain a consistent virtual position of the speakers. This system is portable and adaptable to users' head and body movements in real-time. Our user interface enhances user engagement by allowing users to control the pyBinSim audio playback, change the room environment, and visualize their orientation within the virtual space. The incorporation of proper feedback ensures that users have a clear understanding of their interactions and system performance. The validation tests also shows good results for localization of individual source and envelopment of surround sound audio for all the stages. Latency got a rating of 3.8, as subjects could feel there was some delay when on the move. This can be easily improved by increasing the blocksize from 1024 to 512 with better optimized system. Subject gave an average score of 4.2 on a 5 scale when asked if they feel system is moving along with them. And overall system had the rating of 4.6 on a 5 scale. Hence, the scope of our project has been successfully accomplished.

9.2 Future Improvements

We have also added the room change feature in pyBinSim, which allows the listener to change the room acoustics as desired. Furthermore, since we have indexed the SDM rendered files and changing the filters using them in real-time, some artifacts may be heard if time delay is not applied. A better optimization of this tool can overcome this issue. Since we are using different sensors with varying sensitivity, slight biases may occur when we give input to the pyBinSim processor after applying differential tracking. This can be improved by using better sensors with low latency. It is also important to keep the body sensor stable while moving so that it remains static in reference to the body movements; otherwise, there can be slight offsets. To solve this, ubiquitous sensors that are small in size and can be easily attached to the listener's body can be used. This system also has the potential to go wireless, so that wires do not hinder the listener's movement. For this, wireless low latency trackers that send data via Bluetooth, cloud-based technology (e.g., Amazon Web Services), or using sockets with local connections from which we can load the data into the pyBinSim can be used. Additionally, sometimes there can be confusion with front-back localization even with head tracking alone. Visual clues can help to solve this problem. This can be done with the help of a visualizer in the UI, which also describes the position of the object, or by using virtual reality in combination with it.

9.3 Future Scope

The dynamic binaural tool presented in this study has potential for further research and development in various areas. By expanding on the initial findings, we can explore research directions to improve and refine the capabilities of the system and our understanding of auditory perception.

9.3.1 Perception of Virtual Environments

An important area of research is understanding how various virtual room environments and their acoustic properties affect users' perception of sound. By conducting experiments to test factors such as room sizes, shapes, materials, and reverberation times, we can gain more insights into creating immersive and realistic spatial audio. This information can be utilized to optimize the system for specific applications, such as virtual reality, gaming, or teleconferencing.

9.3.2 Spatial Cognition and Localization

Studying the relationship between head and body movements, virtual speaker positions, and the listener's spatial cognition can enhance the perception of localization in the virtual surround sound system. By investigating how users perceive the location of sound sources in the virtual environment, researchers can develop strategies to improve localization accuracy and create more convincing auditory illusions.

9.3.3 Personalized HRTFs

Integrating individualized Head-Related Transfer Functions (HRTFs) into the system can significantly improve the accuracy and realism of the binaural synthesis, resulting in a more personalized and immersive audio experience for users. Research can focus on developing methods for acquiring personalized HRTFs, either through direct measurements or algorithmic estimation based on anthropometric data. Additionally, studies can investigate the perceptual benefits of personalized HRTFs compared to generic ones.

9.3.4 Auditory Adaptation and Fatigue

Long-term use of the dynamic binaural synthesis system could potentially affect auditory adaptation and fatigue. Research can be conducted to determine if users experience any negative effects, such as increased listening effort, discomfort, or changes in their ability to localize sounds accurately. If such effects are identified, possible mitigations can be explored, including adaptive rendering algorithms or incorporating breaks and rest periods during prolonged listening sessions.

9.3.5 Auditory Scene Analysis

The project can be expanded to study how listeners perceive and segregate different sound sources within the virtual environment. This could involve developing complex auditory scenes with multiple overlapping sound sources and evaluating the system's ability to render them in a way that allows listeners to accurately identify and separate the individual components. Research in this area can lead to improvements in the system's capacity to create more realistic and engaging auditory experiences.

9.3.6 Sound Quality and Subjective Preferences

Investigating the influence of various virtual room parameters and system settings on subjective sound quality can help optimize the system for better user satisfaction. This might involve conducting psychoacoustic experiments and surveys to determine user preferences and perceived sound quality across different conditions, such as varying reverberation levels, speaker configurations, or background noise levels. The findings from such studies can be used to develop guidelines for system designers to create tailored audio experiences that meet user expectations.

9.3.7 Integration with Visual Stimuli

Combining dynamic binaural synthesis with visual stimuli in virtual or augmented reality applications could be investigated to better understand the cross-modal effects on spatial perception and user experience. Research in this area could look into how matching or mismatching audio and visual cues affect the overall feeling of presence, immersion, and understanding of space. Studies might evaluate the possible advantages of combining the dynamic binaural system with other senses, like touch feedback, to create more engaging and immersive multisensory experiences.

Summary

The media project has successfully designed and tested a dynamic binaural synthesis system, which enables users to experience a 5-channel surround speaker setup via headphones. This system overcomes traditional headphone listening limitations, offering an immersive audio experience similar to loudspeakers, by allowing for both horizontal and vertical head movements. Utilizing body and head sensors on a backpack and headphones respectively, the system remains responsive to the user's movements. The user interface offers functionalities such as control over the pyBinSim audio playback, change in the room environment, and visualization of their orientation within the virtual space.

In the validation test, the system received positive feedback, especially for localization, stability, envelopment of surround sound audio in all the stages. The overall user experience got an average score of 4.6 on a 5-point scale. While subjects did report some latency when moving, this can be improved with system optimization and smaller blocksize of 512.

For future enhancements, the project has considered optimizing the backpack system further so that battery is not drained quickly, using better sensors with low latency, ensuring stability of the body sensor, creating a wireless system, and providing more visual clues to solve front-back localization confusion.

The system's potential future scope includes further research in areas such as understanding user perception of various virtual room environments and their acoustic properties, improving localization accuracy, integrating personalized Head-Related Transfer Functions (HRTFs), studying long-term effects like auditory adaptation and fatigue, and conducting research on auditory scene analysis. Future research can also look into combining this dynamic binaural synthesis system with visual stimuli to understand the effects on spatial perception and user experience.

Bibliography

[1] D. Yao, H. Xu, J. Li, R. Xia, and Y. Yan, "Binaural rendering technology over loudspeakers and headphones," *Acoustical Science and Technology*, vol. 41, pp. 134–141, Jan. 2020.

[2] C. Kuhn-Rahloff, *a Contribution to the Understanding of the Inner Reference in Perceptual MeasurementsProzesse der Plausibilitätsbeurteilung am Beispiel ausgewählter elektroakustischer Wiedergabesituationen: Ein Beitrag zum Verständnis der "inneren Referenz" perzeptiver Messungen*. PhD thesis, Technische Universität Berlin, Apr. 2011. Publisher: Technische Universität Berlin.

[3] H. Gamper, "Head-related transfer function interpolation in azimuth, elevation, and distance," *The Journal of the Acoustical Society of America*, vol. 134, p. EL547, Dec. 2013.

[4] S. Li and J. Peissig, "Measurement of Head-Related Transfer Functions: A Review," *Applied Sciences*, vol. 10, p. 5014, Jan. 2020. Number: 14 Publisher: Multidisciplinary Digital Publishing Institute.

[5] M. Vorländer, *Auralization: fundamentals of acoustics, modelling, simulation, algorithms and acoustic virtual reality*. Berlin: Springer, 1st ed ed., 2008. OCLC: ocn171111969.

[6] F. Menzer, C. Faller, and H. Lissek, "Obtaining Binaural Room Impulse Responses From B-Format Impulse Responses Using Frequency-Dependent Coherence Matching," *IEEE Transactions on Audio, Speech & Language Processing*, vol. 19, pp. 396–405, Feb. 2011.

[7] S. Amengual Garí, W. Brimijoin, H. Hassager, and P. Robinson, "Flexible binaural resynthesis of room impulse responses for augmented reality research," in *Proceedings of the 23rd International Congress on Acoustics*, Sept. 2019.

[8] D. R. Moore and A. J. King, "Auditory perception: The near and far of sound localization," *Current Biology*, vol. 9, pp. R361–R363, May 1999.

[9] Y. Chen, Y. Yu, and J.-M. Odobez, "Head Nod Detection from a Full 3D Model," in *2015 IEEE International Conference on Computer Vision Workshop (ICCVW)*, (Santiago, Chile), pp. 528–536, IEEE, Dec. 2015.

[10] P. Mahé, S. Ragot, and S. Marchand, "First-order ambisonic coding with quaternion-based interpolation of pca rotation matrices," in *EAA Spatial Audio Signal Processing Symposium*, pp. 7–12, 2019.

[11] K. Shoemake, "Animating rotation with quaternion curves," in *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pp. 245–254, 1985.

[12] A. J. Hanson, *Visualizing Quaternions*. Elsevier, 2006.

[13] H. Møller, "Fundamentals of binaural technology," *Applied Acoustics*, vol. 36, pp. 171–218, Dec. 1992.

[14] S. Werner, F. Klein, A. Neidhardt, U. Sloma, C. Schneiderwind, and K. Brandenburg,

"Creation of Auditory Augmented Reality Using a Position-Dynamic Binaural Synthesis System—Technical Components, Psychoacoustic Needs, and Perceptual Evaluation," *Applied Sciences*, vol. 11, p. 1150, Jan. 2021. Number: 3 Publisher: Multidisciplinary Digital Publishing Institute.

[15] S. Tervo, J. Pätynen, A. Kuusinen, and T. Lokki, "Spatial Decomposition Method for Room Impulse Responses," *Journal of the Audio Engineering Society*, vol. 61, pp. 16–27, Jan. 2013.

[16] A. Lindau, T. Hohn, and S. Weinzierl, "Binaural Resynthesis for Comparative Studies of Acoustical Environments," in *Proceedings of the Audio Engineering Society 123rd Convention*, Audio Engineering Society, May 2007.

[17] B. Bernschütz, *A Spherical Far Field HRIR/HRTF Compilation of the Neumann KU 100*. German Acoustical Society (DEGA), Mar. 2013.

[18] S. Amengual Garí, J. Arend, P. Calamia, and P. Robinson, "Optimizations of the Spatial Decomposition Method for Binaural Reproduction," *Journal of the Audio Engineering Society*, vol. 68, pp. 959–976, Dec. 2020.

[19] G. Papadakis, K. Mania, M. Coxon, and E. Koutroulis, "The effect of tracking delay on awareness states in immersive virtual environments: an initial exploration," in *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*, VRCAI '11, (New York, NY, USA), pp. 475–482, Association for Computing Machinery, Dec. 2011.

[20] A. Neidhardt, F. Klein, N. Knoop, and T. Köllmer, "Flexible Python Tool for Dynamic Binaural Synthesis Applications," in *Proceedings of the Audio Engineering Society 142nd Convention*, Audio Engineering Society, May 2017.

[21] pyBinSim, "pyBinSim/pyBinSim," Apr. 2023. original-date: 2017-03-29T19:24:48Z.

[22] "multiprocessing — Process-based parallelism." [Online]. Available: https://docs.python.org/3/library/multiprocessing.html. Accessed: Apr. 24, 2023.

[23] "Head Tracker 1." [Online]. Available: https://supperware.co.uk/headtracker-overview. Accessed: Apr. 24, 2023.

[24] "Tracking camera T265." [Online]. Available: https://www.intelrealsense.com/tracking-camera-t265/. Accessed: Apr. 24, 2023.

[25] "Client — python-osc 1.7.1 documentation." [Online]. Available: https://python-osc.readthedocs.io/en/latest/client.html#simpleudpclient-class. Accessed: Apr. 24, 2023.

[26] Darejeh, "A REVIEW ON USER INTERFACE DESIGN PRINCIPLES TO INCREASE SOFTWARE USABILITY FOR USERS WITH LESS COMPUTER LITERACY," *Journal of Computer Science*, vol. 9, pp. 1443–1450, Nov. 2013.

[27] A. D. Moore, *Python GUI Programming with Tkinter: Develop responsive and powerful GUI applications with Tkinter.* Packt Publishing Ltd, 2018.

[28] International Telecommunication Union, "Itu-r recommendation bs.1116-3: Methods for the subjective assessment of small impairments in audio systems including multichannel sound systems," ITU-R Recommendation BS.1116-3, ITU, 2015.

[29] "Signal Processing Toolbox." [Online]. Available:https://de.mathworks.com/products/signal.html.

Accessed: Apr. 25, 2023.

[30] “Statistics and Machine Learning Toolbox.” [Online]. Available: https://de.mathworks.com/products/statistics.html. Accessed: Apr. 25, 2023.

List of Abbreviations

HRTFs Head Related Transfer Functions

LTI Linear Time Invariant

RIRs Room Impulse Responses

BRIRs Binaural Room Impulse Responses

SDM Spatial Decomposition Method

DOA Direction Of Arrival

HRIRs Head Related Impulse Responses

ITD Interaural Time Difference

ILD Interaural Level Difference

TDOA Time Difference Of Arrival

RT-Mod+AP Reverberation Time-Modified Acoustic Power

VE Virtual Environment

ITDG Initial Time Delay Gap

USB Universal Serial Bus

OSC Open Sound Control

DOF Degrees Of Freedom

C Center

FR Front Right

FL Front Left

SR Surround Right

SL Surround Left

DS Direct Sound

ER Early Reflections

LR Late Reverberation

FDLs Fractional Delay Lines

IFFT Inverse Fast Fourier Transform

UI User Interface

GUI Graphical User Interface

EPS	Ear Production Suite
HDMI	High-Definition Multimedia Interface
I/O	Input/Output

List of Symbols

- $y(t)$: Output signal of the system in the time domain.
- $x(t)$: Input signal of the system in the time domain.
- $h(t)$: Transfer function of the system in the time domain.
- τ: Time-shift variable used in the convolution integral.
- $Y(f)$: Output signal of the system in the frequency domain.
- $X(f)$: Input signal of the system in the frequency domain.
- $H(f)$: Transfer function of the system in the frequency domain.
- f: Frequency variable.
- r: Radius or distance from the listener to the sound source in the spherical coordinate system.
- θ: Azimuth angle in the spherical coordinate system, representing the horizontal angle in the medial plane.
- ϕ: Elevation angle in the spherical coordinate system, representing the vertical angle in the sagittal plane.
- $BRIR^u(t)$: Binaural Room Impulse Response for the u-th ear at time t.
- $^\circ$: Degree is used to represent angles.
- t: Time variable.
- N: Total number of discrete time samples.
- p_n: The amplitude scaling factor for the n-th sample.
- $HRIR_{\hat{k}_u^n}$: Head Related Impulse Response for the u-th ear corresponding to the direction $\hat{k}_u^n$.
- $\circledast$: Convolution operator.
- $\delta(t-n)$: Dirac delta function, representing a unit impulse at time n.
- n: Discrete time index.
- $\hat{k}_u^n$: Direction vector for the uth ear corresponding to the n-th sample.
- q: Quaternion
- a: Real component of quaternion q
- b: Coefficient of i in quaternion q
- c: Coefficient of j in quaternion q

- d: Coefficient of k in quaternion q
- i: Imaginary unit satisfying $i^2 = -1$
- j: Imaginary unit satisfying $j^2 = -1$
- k: Imaginary unit satisfying $k^2 = -1$
- q_1: Starting quaternion for interpolation
- q_2: Ending quaternion for interpolation γ: Interpolation factor ($0 \leq \gamma \leq 1$)
- Ω: Angle between q_1 and q_2
- slerp$(\cdot)$: Spherical linear interpolation function

List of Figures

List of Algorithms

Listings

Declaration of Independence

We Declare that we have carried out and written this work independently. Sources, literature and tools used by us are marked as such.

Signatures

Pranav Sharma

[illegible]

Pranav Sharma

Syed Muhammad Ahmed

Representation of work

The present work is a collective work. Work is distributed in an agile manner.

Project Development

Task	**Team Member**
Planning	Pranav Sharma
Code Development	Pranav Sharma
Usability Testing	Pranav Sharma
Experimental Setup	Syed Muhammad Ahmed
Integration Testing	Syed Muhammad Ahmed
System Testing	Syed Muhammad Ahmed

Table 10.1: Work Distribution in the Project

Report Making

The table below shows the editing of the sections done by Pranav Sharma and Syed Muhammad Ahmed

Section Name	Editor
Abstract	Pranav Sharma
1 Motivation and Introduction	Pranav Sharma
1.1 Motivation	Pranav Sharma
1.2 Introduction	Pranav Sharma
2 Fundamentals of Augmented Auditory Reality	Syed Muhammad Ahmed
2.1 Spatial Audio	Syed Muhammad Ahmed
2.2 Head Related Transfer Functions (HRTFs)	Pranav Sharma
2.3 Binaural Room Impulse Responses (BRIRs)	Pranav Sharma
2.4 Sound Localization and Perception	Syed Muhammad Ahmed
2.5 Tait-Bryan angles	Syed Muhammad Ahmed
2.6 Quaternions	Syed Muhammad Ahmed
3 Dynamic Binaural Synthesis	Pranav Sharma
3.1 Binaural Synthesis	Pranav Sharma
3.2 Spatial Decomposition Method (SDM)	Pranav Sharma
3.3 Pose based Dynamic Binaural Rendering	Pranav Sharma
3.4 Differential Tracking	Pranav Sharma
4 pyBinSim: A Python Tool for Auralization of Audio	Syed Muhammad Ahmed
4.1 Configuration Parameters	Syed Muhammad Ahmed
4.2 Flow of pyBinSim	Pranav Sharma

Table 10.2: Section distribution for Pranav Sharma and Syed Muhammad Ahmed (Part 1)

Section Name	Editor
5 System Architecture and Design	Pranav Sharma
5.1 Overview	Pranav Sharma
5.2 System Architecture	Pranav Sharma
5.2.1 PyBinSim Processor	Pranav Sharma
5.2.2 pyBinSim controller	Syed Muhammad Ahmed
5.3 System Design	Pranav Sharma
5.4 Room Database	Pranav Sharma
6 Experimental Setup and Validation	Pranav Sharma
6.1 Listening Test Procedure	Pranav Sharma
6.2 Results	Pranav Sharma
6.2.1 Stage 1: Room Change Detection and Loudness Check	Pranav Sharma
6.2.2 Stage 2: Test with Head Tracking On	Pranav Sharma
6.2.3 Stage 3: Test with Head and Body Tracking	Pranav Sharma
7 Documentation	Syed Muhammad Ahmed
7.1 Hardware and Software description	Pranav Sharma
7.1.1 Hardware Specifications	Syed Muhammad Ahmed
7.1.2 Software Specifications	Pranav Sharma
7.2 Experimental setup	Syed Muhammad Ahmed
7.3 Folder structure and Matlab Script	Syed Muhammad Ahmed
7.4 Running pyBinSim	Syed Muhammad Ahmed
7.5 OSC Messages for pyBinSim	Pranav Sharma
7.6 Data Handling in pyBinSim processor	Pranav Sharma
7.7 Designing User Interface using Tkinter	Pranav Sharma
7.7.1 Visualizing the Head Movements in the Azimuthal Plane	Pranav Sharma
7.8 Scalability and Portability	Pranav Sharma
8 Discussion and Outlook	Syed Muhammad Ahmed
8.1 Scope of the Project	Pranav Sharma
8.2 Future Improvements	Pranav Sharma
8.3 Future Scope	Syed Muhammad Ahmed
8.3.1 Perception of Virtual Environments	Syed Muhammad Ahmed

Table 10.3: Section distribution for Pranav Sharma and Syed Muhammad Ahmed (Part 2)