UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

A UNITY BASED VALIDATION MODEL FOR RENDERING ALGORITHMS OF MOTION-TRACKED BINAURAL RECORDINGS

A Degree Thesis

Submitted to the Faculty of the

Escola Tècnica d' Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Clara Rendé Subias

In partial fulfilment

of the requirements for the degree in

Science and Telecommunication Technologies
ENGINEERING

Advisor: Markus Hädrich

Berlin, January 2017

Abstract

In 2004, motion-tracked binaural (MTB) sound was introduced as a new method for capturing, recording and reproducing spatial sound. In November 2010, the Technische Universität Berlin (TUB) started to evaluate MTB recordings and developed a C rendering application. One of the most direct applications of such technology is to create computer games. For this purpose, the goal of this project is to use Unity, which is one of the most known game development framework, to apply this new concepts in a practical situation. The project is based on the synchronization between a 2D game and a real-time rendered sound using several of the MTB algorithms. The main challenges were to develop a C++ based dll that implements the MTB logic, to create a Unity executable game and to integrate both technologies to produce the three-dimensional sound effect. The C code takes into account several input music mono channels and a rotation angle controlled by the game itself. As a final result, a complete Windows application was developed.

Resum

El 2004, el so “Motion-Tracked Binaural” (MTB) es va introduir com un nou mètode per capturar, enregistrar i reproduir el so espacial. El novembre de 2010 la Technische Universität Berlin (TUB) va començar a avaluar enregistraments de MTB i va desenvolupar una aplicació de renderització en C. Una de les aplicacions més directes d'aquest tipus de tecnologia és la creació de jocs d'ordinador. Per aquest motiu, l'objectiu d'aquest projecte és utilitzar el Unity, un dels programes de desenvolupament de jocs més coneguts, per aplicar aquests nous conceptes en una situació pràctica. El projecte es basa en la sincronització d’un joc en 2D i un so renderitzat a temps real utilitzant diversos dels algoritmes de MTB. Els principals reptes del projecte van ser desenvolupar una dll en C++ que implementa la lògica de MTB, crear el joc executable de Unity i integrar ambdues tecnologies per produir l'efecte de so tridimensional. El codi implementat en C té en compte diversos canals d’entrada mono de música i un angle de rotació controlat pel joc en si. Com a resultat final s’ha desenvolupat una aplicació completa de Windows.

Resumen

En 2004, el sonido “Motion-Tracked Binaural” (MTB) fue introducido como un nuevo método para capturar, grabar y reproducir sonido espacial. En noviembre de 2010, la Technische Universität Berlin (TUB) comenzó a evaluar las grabaciones MTB y desarrolló una aplicación de renderización en C. Una de las aplicaciones más directas de tal tecnología es crear juegos de computadora. Para ello, el objetivo de este proyecto es utilizar Unity, que es uno de los programas más conocidos para el desarrollo de juegos, para aplicar estos nuevos conceptos en una situación práctica. El proyecto se basa en la sincronización entre un juego 2D y un sonido renderizado a tiempo real utilizando varios algoritmos de MTB. Los principales retos fueron desarrollar una dll basada en C++ que implemente la lógica MTB, crear el juego ejecutable de Unity e integrar ambas tecnologías para producir el efecto sonoro tridimensional. El código C tiene en cuenta varios canales de entrada mono de música y un ángulo de rotación controlado por el juego en sí. Como resultado final, se ha desarrollado una aplicación completa de Windows.

To my parents Toni and Pia

and my sister Pia.

Acknowledgements

I would like to express my gratitude and gratefulness to Dr. Stefan Weinzierl, head of TU-Berlin Audiokommunikation Department, for giving me the opportunity of carrying out my thesis.

Secondly, to Markus Hädrich, supervisor of the project, and to David Ackermann, both members of the Audiokommunikation Department, for providing me with the appropriate help.

Moreover, my thankfulness to Professor Climent Nadeu Camprubí for his support as co-tutor of the project at home university, Telecom Barcelona-UPC.

Furthermore, to my father Toni Rendé, who is also a telecommunications engineer, for providing me with the appropriate programming background along with wise advices and all his support.

Finally, to the Technische Universität of Berlin for hosting me during this period and for providing me with the necessary tools and infrastructures in order to develop the project.

Revision history and approval record

Revision	Date	Purpose
0	01/12/2016	Document creation
1	10/01/2017	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Clara Rendé Subias	Clara.rende@gmail.com
Markus Hädrich	Markus.haedrich@tu-berlin.de
David Ackermann	David.ackermann@tu-berlin.de

Written by:		Reviewed and approved by:	
Date	01/12/2016	Date	10/01/2017
Name	Clara Rendé Subias	Name	Markus Hädrich
Position	Project Author	Position	Project Supervisor

Table of contents

List of Figures

List of Tables and equations

1. Introduction

Motion Tracked Binaural (MTB) sound is a head-related multi-channel microphone recording technology proposed by Professor V. Ralph Algazi, an Audio Engineering Society (AES) member, in the CIPIC Interface Laboratory in 2004. It was introduced as a new method for capturing, recording, and reproducing spatial sound. MTB uses a circular array of microphones which are mounted on the horizontal circumference of a (mostly) spherical rigid shell with the diameter of an average head. Taking into account the results of Prof. Algazi published on AES Journal ([19]), two professors from the Audio Communication department of the Technische Universität Berlin (TUB), Mr. Alexander Lindau and Mr. Sebastian Roos, started a project in November 2010 to evaluate MTB recordings. Their research was explained in the VDT International Convention ([1]). From then, the Audio Communication department began to implement a MTB rendering application. Its purpose was to assess MTB reproduction quality in an integrative manner and as a function of the number of microphones (8, 16, 24, and 32), the type of interpolation algorithm (5 methods) and the audio content (noise, music, speech).

This project has been developed to implement a complete rendering application as a Unity plugin. Unity is a cross-platform game engine used to develop video games for PC, consoles, mobile devices and websites. The way Unity handles plugins is through the use of dynamic-link libraries (.dll) and C# scripts. Scripts allow library function calls according to events and suitable attribute passing from Unity to the C code. The final goal of the project is to apply the MTB sound concept to a Windows based standalone executable Unity game. Due to the fact that this project includes several different technologies, environments and resources to be related in a short period of time, a feasible approach was to identify one main goal together with some optional additional improvements. Yet, a previous work to understand the whole rendering application, review the C++ and C# programming languages and study in-depth the Unity engine was carried out in order to satisfy such purposes. Other important previous issues were to focus on concepts like implementing circular buffers, applying digital filters, knowing the Microsoft multimedia library and also understanding the wav file format, among others.

The developed Unity game consists on a human body in front of a concert hall. This figure can be rotated using the keyboard arrow keys. In a real-time basis, the rotation angle serves to render the stereo sound according to multi-channel pre-recorded signals and specific rendering algorithms. Among the five MTB proposed algorithms, four out of five were implemented and tested along the process. Some new algorithms (modifications or improvements from the previous ones) were also developed. Once accomplished this goal, further optional improvements initially identified were the following ones. On the one hand, to develop and deploy a standalone app for Android or iPhone based on the above concept. On the other hand, to further develop the smartphone app including the head-mounted display and headphones.

This document goes through a theoretical part to introduce the MTB concept and a practical part with all the necessary code implemented to create the plugin. Finally, some graphical results are shown and analysed on different charts.

1.1. Requirements and specifications

In regards with the project requirements the following has to be considered:

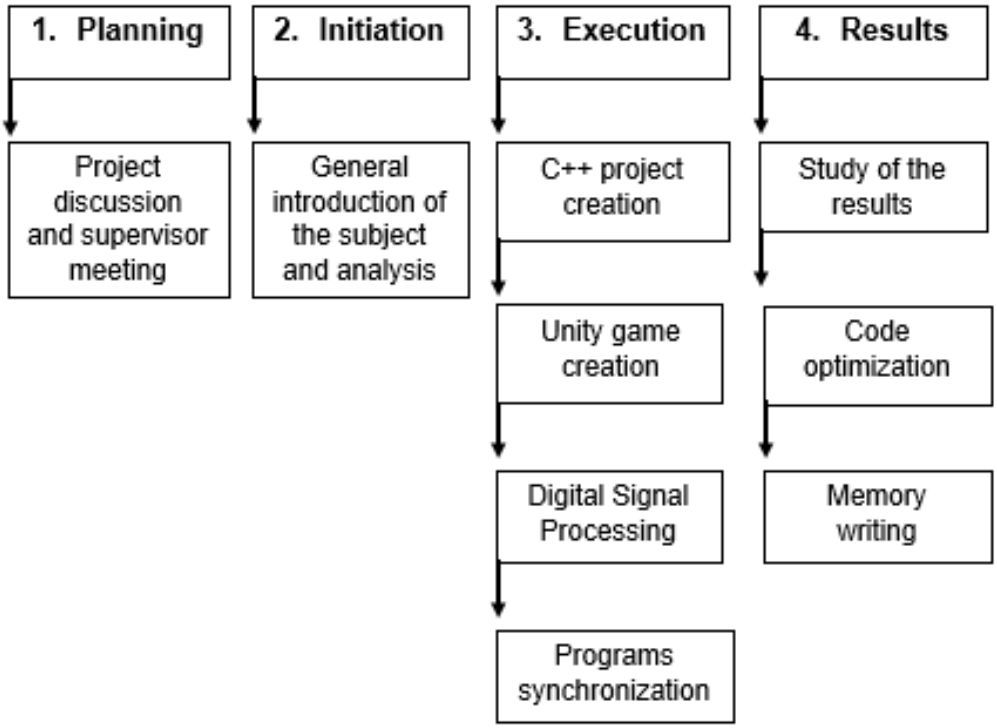
- Development of a Unity game for Windows platform that can serve to validate the sound rendering algorithms provided by the MTB technology.
- Development of needed classes and methods to get music samples from wav files.
- Implementation of C++ algorithms to combine the different recorded audio samples to generate the stereo output taking into account the receiver rotation angle.
- Testing and validation of the MTB concepts.
- Code debugging and testing.
- Project description and results reporting.

As for the project specifications:

- The pre-recorded file format is wav.
- The wav files consist of16 monophonic files with samples of 16 bits at a frequency of 44.1 kHz. Each file was obtained through one of the 16 microphones which are mounted on the horizontal circumference of the spherical rigid shell.
- The input channels are mono and can have a variable length.
- The output of the system is stereo.
- The Microsoft multimedia library is specific to the Windows platform.
- The music has to be reproduced without clicks or discontinuities.
- The program can be executed one-shot or in a looping mode.
- The user can choose the rendering algorithm from the game interface.

1.2. Work plan

1.2.1. Work breakdown structure



1.2.2. Work tasks & milestones

Task: General introduction of the subject and analysis	WP ref: WP1 (sheet 1 of 5)
Short description: To get familiar with the subject.	Planned start date: 20/09/2016 Planned end date: 20/10/2016
Internal task T1: To read the project documentation (Algazi's pdfs and the rendering application) **Internal task T2**: To study the required programming tools (Visual Studio IDE, Unity) **Internal task T3**: To review the C++ and C# programming languages **Internal task T4**: To study the WAV file format **Internal task T5**: To study the Windows Multimedia library. Testing its methods with simple wav files	Start event: 20/09/2016 End event: 01/11/2016

Task: C++ project creation	WP ref: WP2 (sheet 2 of 5)
Short description: To create the audio C++ project in visual studio in order to export it then as a dynamic-link library (.dll) and synchronize it with the Unity game.	Planned start date: 20/10/2016 Planned end date: 10/11/2016
Internal task T6: To configure the output audio device **Internal task T7**: To create the input buffers: use of dynamic memory allocation and memory request to the heap **Internal task T8**: To test the playback of simple wav files. Increase of the number of channels, from 1 to 16 **Internal task T9**: To implement the first interpolation algorithms, without a dynamic rendering (not taking into account the head rotation). Addition of variations of those algorithms **Internal task T10**: To export the project as a dll	Start event: 01/11/2016 End event: 20/11/2016

Task: Unity game creation	WP ref: WP3 (sheet 3 of 5)
Short description: To create the game in Unity in order to simulate the human head rotation.	Planned start date: 10/11/2016 Planned end date: 20/11/2016
Internal task T11: To create the scene and the C# script **Internal task T12**: To synchronize both projects (C# and C++ scripts **Internal task T13**: To debug both projects. Creation of two versions of the C++ project: an executable to trace errors and the dll	Start event:20/11/2016 End event: 10/12/2016

Task: Digital Signal Processing	WP ref: WP4 (sheet 4 of 5)
Short description: To study the different sample processing techniques in order to implement the most complex interpolation algorithms of the rendering application.	Planned start date: 20/11/2016 Planned end date: 10/12/2016
Internal task T14: To study the filters: high pass filter, low pass filter **Internal task T15**: To search and study an audio library in order to implement FFT	Start event: 10/12/2016

Internal task T16: To implement the other interpolation algorithms using dynamic rendering, taking into account the head rotation.	End event: 20/12/2016

Task: Results	WP ref: WP5 (sheet 5 of 5)
Short description: To study the results and to report them	Planned start date: 01/12/2016 Planned end date: 10/01/2017
Internal task T17: To create charts in order to represent the wav files samples once processed **Internal task T18**: To optimize the code in order to achieve a faster execution **Internal task T19**: To test the application with different pre-recorded wav files. To study, download and use different audio tools in order to modify the wav samples: Audacity and Multi Tracker tool. **Internal task T20**: To write the memory	Start event: 01/12/2016 End event: 10/01/2017

1.2.3. Incidences & work plan modifications

After one month trying to get familiar with Unity by my own, which is not an easy task due to its complexity, wide functionality available and so many technologies involved (C++, C#, plugins, scripting), we realized the final goal was not feasible for a bachelor thesis due to the available time frame. It would have been difficult to get a working VR standalone app in four months.

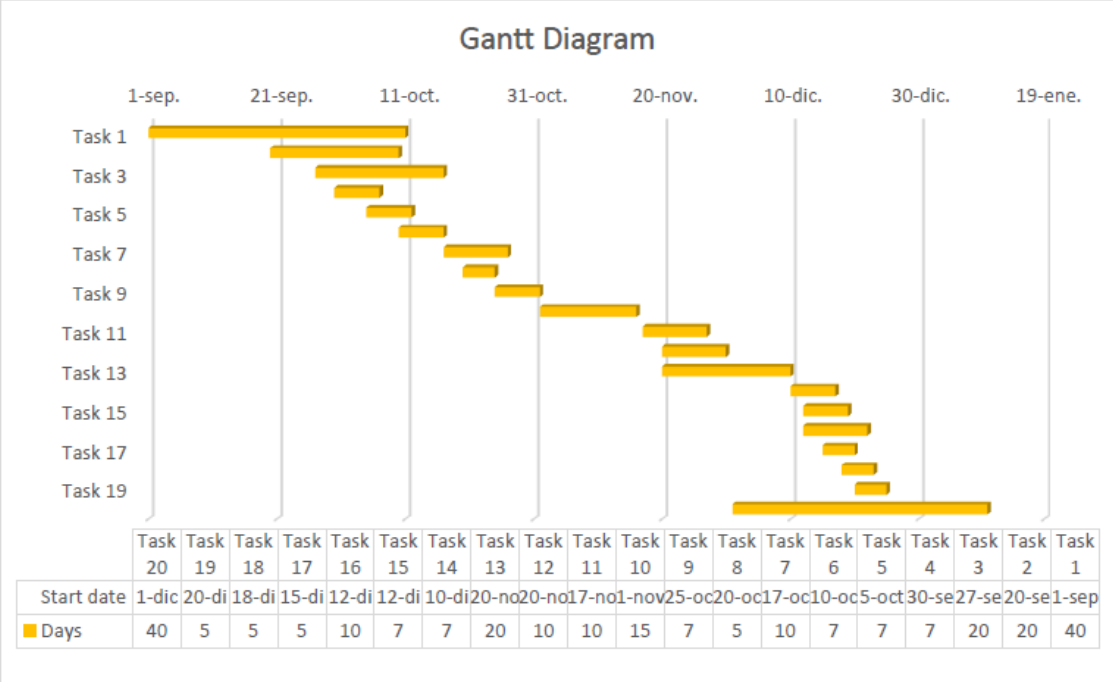
In order to define a feasible project requirement, we modified the main goal of the project and added some additional and optional further targets to assure an auto-content result, as explained in the introduction. Therefore, the final main goal is to create a Unity game for a Windows platform. The game consists on a human body in front of a concert hall. This figure can be rotated using the keyboard arrow keys. In a real-time basis, the rotation angle serves to render the stereo sound according to multi-channel pre-recorded signals and specific rendering algorithms.

Furthermore, once the first goal is accomplished, future improvements can be done. On the one hand, the development and deployment of a standalone app for Android or iPhone based on the above concept. On the other hand, the improvement of the smartphone app including the head-mounted display and headphones.

Finally, another issue I had during the development of this project was the lack of computer RAM. The project was developed using first four input channels. That is, four wav files. Once the program was working properly, the number of channels was increased up to 16. By increasing this value, the computer needed more RAM and, when

running the program, crashed. In order to solve this problem I had to extend the RAM memory from initially 4GB to 8GB. After the change, the program worked correctly.

1.2.4. Gantt diagram



	Task 20	Task 19	Task 18	Task 17	Task 16	Task 15	Task 14	Task 13	Task 12	Task 11	Task 10	Task 9	Task 8	Task 7	Task 6	Task 5	Task 4	Task 3	Task 2	Task 1
Start date	1-dic	20-di	18-di	15-di	12-di	12-di	10-di	20-no	20-no	17-no	1-nov	25-oc	20-oc	17-oc	10-oc	5-oct	30-se	27-se	20-se	1-sep
Days	40	5	5	5	10	7	7	20	10	10	15	7	5	10	7	7	7	20	20	40

#	Task	Start date	Days
Task 1	Project documentation	1-sep.	40
Task 2	Programming tools (Visual Studio and Unity)	20-sep.	20
Task 3	C++ and C# review	27-sep.	20
Task 4	WAV file format	30-sep.	7
Task 5	Windows Multimedia Library	5-oct.	7
Task 6	Output audio device	10-oct.	7
Task 7	Input buffers creation	17-oct.	10
Task 8	Testing playback wav files	20-oct.	5
Task 9	First interpolation algorithms (no dynamic rendering)	25-oct.	7
Task 10	Project exportation as a dll	1-nov.	15
Task 11	Unity scene creation and C# script	17-nov.	10
Task 12	Synchronization of both projects	20-nov.	10
Task 13	Debugging both projects	20-nov.	20
Task 14	Study of the filters	10-dic.	7
Task 15	Study audio libraries to implement FFT	12-dic.	7
Task 16	Other interpolation algorithms (dynamic rendering)	12-dic.	10
Task 17	Creation of charts and interpretation of results	15-dic.	5
Task 18	Optimization of the code	18-dic.	5
Task 19	Testing application	20-dic.	5
Task 20	Reporting results	1-dic.	40

2. State of the art

Virtual reality (VR) technologies are evolving fast and becoming more important in the day to day of people. Their main goal is to create a feeling of “presence”, making the user feel lost in the scene, forgetting reality and being absorbed and immersed in the virtual. Both three-dimensional video and audio effects are really important to generate realistic sensations that replicate a real environment.

On the one hand, 3D videos are already much achieved to such an extent that today some companies have started working on the streaming of VR videos. This is the case of GoPro, whose platform has bet for it and have recently introduced Omni, its VR camera to make videos in 360º and GoPro VR, its VR application. Their goal is to be able to broadcast live. Moreover, other services like YouTube are also working on this new technology. One year ago the video platform of Google started to admit VR videos and now it goes a step further. It will allow to relay live the VR videos. In addition, YouTube has held meetings with camera manufacturers and 360º glasses so that both devices are compatible with the video platform.

On the other hand, audio can still be exploited a lot to achieve complete immersion. On the current VR videos the audio is usually flat (2D). Furthermore, most Virtual Acoustic Environments (VAEs) suffers from a lack of naturalism. For VR to be truly immersive, it needs convincing sound to match. Some companies have already begun to develop software to improve this aspect and recreate not only the video in 3D, but also the audio. For instance, the Two Big Ears Company, which was founded in 2013, design immersive and interactive audio applications and tools with a focus on mobile and emerging technologies. Their mission is to make VR audio succeed across all devices and platforms. They use real-time rendering of the objects with head and positional tracking. The methodology looks quite similar to the one the MTB principle is using (explained in the next point of this document). However, there is not much public information about the algorithms they are using. Therefore, we cannot make a good comparison with the MTB sound principle.

3. Methodology

3.1. The MTB sound principle

The MTB reproduction principle works as the next scheme shows.

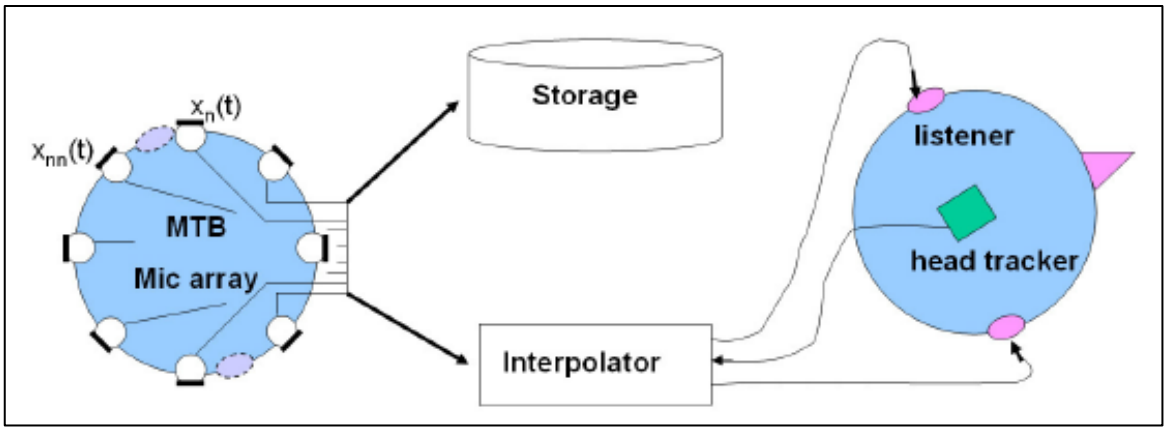


Figure 1. Sketch of the MTB reproduction principle

Firstly, on the left side there is the spherical rigid shell with the set of microphones placed at the horizontal circumference of the sphere. Secondly, on the right side there is the listener's head wearing headphones and a head tracker. The sphere is connected to a storage where all the pre-recorded audio files are saved. In addition, the sphere is also connected to an interpolator, which will apply the required interpolation algorithm to the wav files samples taking into account the human head rotation. The head tracker will determinate the position of the listener's ears and the two microphone signals $x_n(t)$ and $x_{nn}(t)$ that are closest to that position. Finally, the ear signals are constructed by interpolating between these microphones while following head movements. Ear signals can be reconstructed instantaneously or off-line using stored MTB recordings.

MTB discretization and interpolation

There are some obvious major system parameters that can be chosen in order to obtain faster or more accurate results. Those parameters are:

1. The number of microphones
2. The interpolation algorithm
3. The array geometry
4. The angular position of the interpolated microphone signal.

In this project both the number of microphones and the type of interpolation algorithm were varied in order to study their effects in the audio plugin application. The number of microphones were varied from 4 to 16. As to the type of interpolation, different algorithms were implemented. Those are described later.

Binaural recording is a method of recording sound that uses two microphones with the intent to create a 3D stereo sound sensation for the listener of actually being in the room

with the performers or instruments. In 2004, Algazi proposed this new binaural recording technique, the MTB, which promises an efficient solution for the recording and rendering of dynamic binaural ambient scenes. By means of interpolation algorithms, the multichannel signal recorded with the MTB array can be used to reconstruct two audio signals in a way as if they were recorded approximately at a listener's ear position. The rigid sphere of the MTB array acts as an obstacle for sound propagation introducing frequency dependent interaural level (ILD) and time (ITD) differences which are similar to those occurring in binaural hearing. The MTB reproduction can graphically be thought of as listening through headphones to the microphone being rotated according to one's current head orientation. Depending on the complexity of the interpolation algorithm used, the MTB signal can be played back in real-time adapting to head orientation.

3.2. The WAV format

The input of the audio plugin application implemented in this project consists of sixteen mono wav files. This is the reason why an in-depth study of this format was done in order to understand it and to be able to implement then the rendering algorithms

The most common wav audio format is uncompressed audio in the linear pulse code modulation (LPCM) format. LPCM is also the standard audio coding format for audio CDs, which store two-channel LPCM audio sampled 44,100 times per second with 16 bits per sample.

The wav file format is broken up into RIFF chunks. It includes the specific data structure as shown in the following figure.

File offset (bytes)	Field name	Field Size (bytes)
0	ChunkID	4
4	ChunkSize	4
8	Format	4
12	SubChunk1ID	4
16	Subchunk1Size	4
20	AudioFormat	2
22	NumChannels	2
24	SampleRate	4
28	ByteRate	4
32	BlockAlign	2
34	BitsPerSample	2
36	Subchunk2ID	4
40	SubChunk2Size	4
44	Data	Subchunk2Size

Figure 2. Structure of the wav file

The blue zone refers to the RIFF chunk descriptor (generic file container format for storing data in tagged chunks), whose format here is the WAVE, which requires two sub-chunks: "fmt" (green zone) and "data" (red zone).

1. **ChunkID** contains the letters “RIFF” in ASCII form.
2. **ChunkSize** is the size of the rest of the chunk following this number. That is, the entire file inbytes minus 8 bytes for the two previous fields not included in this count: ChunkID and ChunkSize
3. **Format** contains the letters “WAVE”.

The green zone refers to the “fmt” sub-chunk, which describes the format of the sound information in the data sub-chunk.

4. **SubChunk1ID** contains the letters “fmt”
5. **Subchunk1Size** is 16 for PCM (pulse-code modulation, to digitally represent sampled analog signals). It is the size of the rest of the Subchunk which follows this number.
6. **AudioFormat**: PCM = 1 for linear quantization. Other numbers for different forms of compression.
7. **NumChannels**: =1 if the file is mono, =2 if the file is stereo.
8. **SampleRate**: samples per second. Is the sampling frequency in Hz (8000, 44100, etc).
9. **ByteRate**: bytes per second. To obtain its value we calculate: SampleRate*NumChannels*BitsPerSample/8.
10. **BlockAlign** is the number of bytes for one sample including all channels. To obtain its value we calculate: NumChannels*BitsPerSample/8 (2=16bit if mono, 4=16bit if stereo).
11. **BitsPerSample** is the number of bits per sample.

The red zone refers to the “data” sub-chunk, which indicates the size of the sound information and contains the raw sound data.

12. **Subchunk2ID** contains the letters “data”.
13. **SubChunk2Size** is the number of bytes in the data. To obtain its value we calculate: NumSamples*NumChannels*BitsPerSample/8.
14. **Data** is the actual sound data.

In the C code of the application, the wav’s header was defined as a typedef struct, shown as follows.

```
typedef struct  WAV_HEADER              // Header structure of the WAV file
{
    /* RIFF Chunk Descriptor */
    uint8_t         RIFF[4];            // RIFF Header Magic header
    uint32_t        ChunkSize;          // RIFF Chunk Size
    uint8_t         WAVE[4];            // WAVE Header
    uint8_t         fmt[4];             // FMT header
    uint32_t        Subchunk1Size;      // Size of the fmt chunk
    uint16_t        audioFormat;        // Audio format 1=PCM,6=mulaw,7=alaw,
                                        // 257=IBM Mu-Law, 258=IBM A-Law, 259=ADPCM
    uint16_t        numOfChan;          // Number of channels 1=Mono 2=Stereo
    uint32_t        samplesPerSec;      // Sampling Frequency in Hz
    uint32_t        bytesPerSec;        // bytes per second
    uint16_t        blockAlign;         // 2=16-bit mono, 4=16-bit stereo
    uint16_t        bitsPerSample;      // Number of bits per sample
    uint8_t         Subchunk2ID[4];     // "data"  string
    uint32_t        sampledDataLen;     // Sampled data length(bytes)
} wavHdr, *pwavHdr;
```

Figure 3. Structure of the wav file header

What differentiates a mono file from a stereo one is as follows. In monaural sound one single channel is used. It can be reproduced through several speakers, but all speakers are still reproducing the same copy of the signal. In stereophonic sound more channels are used (typically two). Two different channels can be used and make one feed one speaker and the second channel feed a second speaker to create directionality, perspective, space. And this is what was tried to achieve with the different rendering algorithms, to alter the samples of the different input channels to create a three-dimensional sound.

As mentioned, the input channels are mono. The mono samples contain 2 bytes (16 bits) separated into two blocks, the least significant byte (LSB) and the most significant byte (MSB). The output of the application is stereo. The samples are sent to the output audio device following the next order: left, right, left, right and so on, in order to reproduce the sound.

3.3. The Microsoft multimedia library

The audio plugin application uses the Microsoft multimedia library, a windows-specific library which enables applications to use sound and video. In order to import and use the library in the project, it was required to add the MMSystem.h header file in the source file, which declares a set of procedures and functions related to the specific multimedia hardware control. The corresponding dynamic-link library winmm.dll, which provides access to the original windows multimedia audio API (WinMM), was imported as well.

The methods explained below were used to open the audio device, play the wav files sound and close the audio device.

On the one hand, the waveOutOpen() function was used to open the given waveform-audio output device for playback. In order to play the sound the audio playback should be continuous, so the audio driver needs to use input data buffers all the time to avoid silences or skips. Therefore, the samples data stream were initially organized in several chained buffers of fixed lengths. In order to play back the samples, the thewaveOutPrepareHeader() function was called first to prepare the waveform-audio data block for playback. Then waveOutWrite() was called to play the data. This function sends the fixed data block to the given waveform-audio output device. Finally, it was necessary to wait the method callback and control the buffer number that had just finished playing, fill it with new sample data and call waveOutWrite() in a loop. Each call of function waveOutWrite() is creating a new thread that reads the data buffer in a synchronized mode with our data filling method.

The waveOutUnprepareHeader() function was called after the device drive had finished with a data block and before freeing the data buffer to clean up and complement the preparation performed by the waveOutPrepareHeader() function.

On the other hand, the waveOutReset() function was used to finally close the audio device handle and stop feeding buffers to the sound driver. This function stops playback on the given waveform-audio output device and resets the current buffer position to zero. All pending playback buffers are marked as done and returned to the application.

The playback process was executed by a single thread so that the program could execute other processes at the same time. In order to manage the access to the sample buffers, a further synchronization mechanism was implemented to avoid data inconsistences. To control this critical area different functions were used, such as EnterCriticalSection(), LeaveCriticalSection() and deleteCriticalSection().

Finally, some other functions from the multimedia library related to the heap were also used. The heap is the portion of memory where dynamically allocated memory resides. Memory allocated from the heap will remain allocated until one of the following occurs: the memory is freed or the program terminates. In the program, there were used the GetProcessHeap() function, which gets the memory set aside for dynamic allocation, the HeapAlloc() function, which allocates a block of memory from a heap, and the HeapFree() function, which frees a memory block allocated from a heap.
For further references there is the official Microsoft website explaining all its methods ([21]).

3.4. The audio plugin application

The audio plugin application works as follows. Sixteen mono wav files are initially taken into account. Each one was previously recorded with a particular microphone belonging to the 16 microphone array contained on the spherical rigid shell.

The process makes an initial loop in order to open each wav file in binary reading mode. For each file, it is initially obtained its header, which contains the most significate information like total data size (in bytes), total number of samples and so on. Depending on these particular values, a specific amount of memory is demanded to the heap to allocate all the samples in the different input buffers. Then, each file is iterated to fill the buffers with the samples and finally we close all the file pointers. At this point there are 16 input buffers containing just the data samples of each wav file. Actually, the maximum amount of memory needed is limited by fixing a maximum input buffer size. If any of the wav files reaches this maximum size, the data is truncated and the maximum input buffer size is set. To compensate this size limitation, there is the possibility to run the music in looping mode. These input buffers are FIFO buffers and all of them contain the same amount of samples.

After reading all the input mono channels, the output audio device is initialized with the appropriate parameters (sample rate, sample size, number of channels...). As mentioned, there are 16 bits per sample, a sampling frequency of 44,1kHz and two output channels, the left and the right (stereo). The Microsoft multimedia library methods are used. So all the specific hardware control functions are based on these methods.

At running time, the content of the input buffers is transferred into intermediate blocks of 1024 bytes each one. All these blocks are filling the final output buffer which is formed by a configurable amount of fixed size data segments linked together. This activity is synchronized with the driving of the output device handled by the Microsoft multimedia methods. Depending on the user's head rotation, the final rendered stereo image is gotten. To generate the final stereo rendering image, several different algorithms were implemented, from a very simple one to a more sophisticated alternatives. With this intention, the application goes from 16 input buffers to two output channels, which are sent to the output audio device, one to the left output channel and the other one to the right, following this order, getting the music to be reproduced and creating the three-dimensional sound effect. Finally, once the whole music is reproduced, the audio device is closed.

When implementing the rendering algorithms, which take into account the head rotation, dynamic rendering is used. In other words, every 5,8ms a block of 1024 bytes is processed considering the same head rotation. After 5,8ms the head rotation is read again. So the rotation is the same for every block of 1024 bytes.

(1) $$\frac{1024\ \text{bytes}}{\text{block}} * \frac{1\ \text{sample}}{4\ \text{bytes}} = \frac{256\ \text{samples}}{\text{block}}$$

(2) $$\frac{256\ \text{samples}}{\text{block}} * \frac{1\ \text{sec}}{44100\ \text{samples}} = \frac{5{,}8\ \text{ms}}{\text{block}}$$

1 mono sample = 2 bytes, distributed as follows: | High (1byte) | Low (1byte) |

1 stereo sample = 2 mono samples = 4 bytes: | LHB LLB RHB RLB |

where: LHB: Left High Byte

LLB: Left Low Byte

RHB: Right High Byte

RLB: Right Low Byte

The next figure summarizes the process explained above.

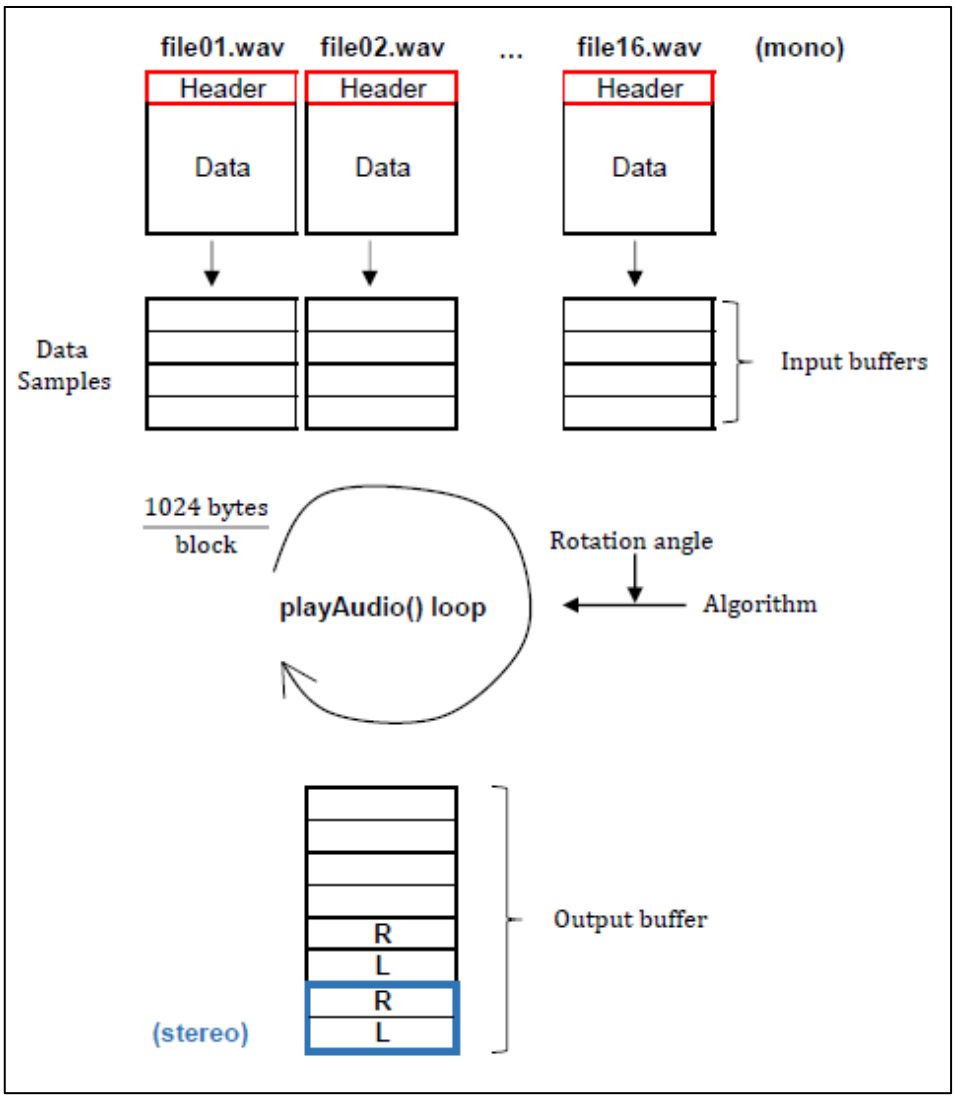


Figure 4. Buffering scheme of the audio plugin application

3.5. The rendering algorithms

In order to reconstruct the 3D audio image, several algorithms were implemented in the MTB rendering application. In this project, some of those rendering algorithms initially defined by Mr. Roos and Mr. Lindau, were implemented. Additionally, some extra methods were added with specific modifications respect the originals. Main differences between algorithms are the number of input channels to be taken into account, the consideration or not of the head rotation angle (dynamic rendering) and the digital processing algorithm of the samples. Those different implementations are explained below following an order of increasing complexity.

In order to explain every implementation, the following figure needs to be taken into account. It consists of a human head, seen from a top view, with its nose, ears and eyes. The rotation angle is thus indicated by the nose. The chR and chL variables refer to each single output channel, the right and the left respectively.

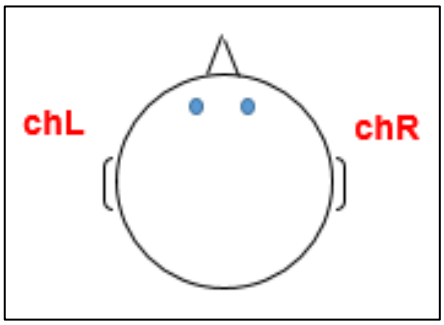


Figure 5. Listening perspective. Human head representation

Depending on the number of input channels the head can be divided into more or less imaginary sections that will set the threshold boundaries for selecting the closest microphone to each listener's ear.

For instance, using 4 or 8 channels with 0 rotation angle would be represented as:

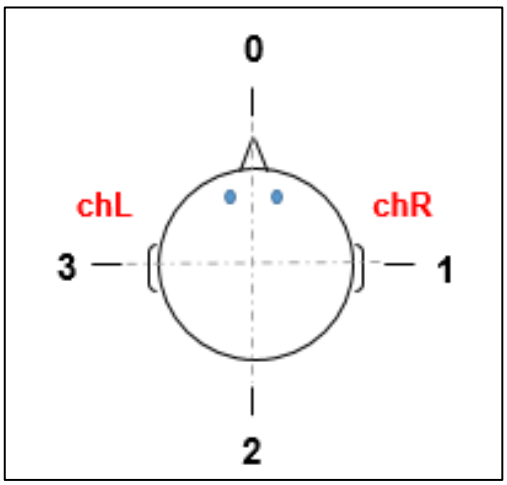


Figure 6. Human head with 4 channels

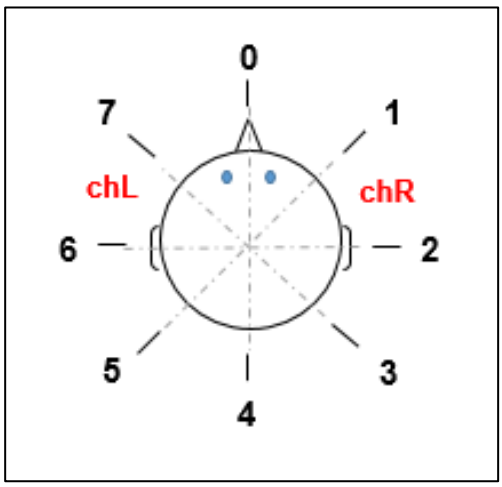


Figure 7. Human head with 8 channels

Numbers in bold are referred to each microphone and the discontinuous lines are the boundaries of each section.

All algorithms were implemented taking into account the rotation angle of the right channel. Once the rotation angle from Unity was obtained, 90 degrees were added to it to correctly locate the right ear. Once obtained it, getting the angle of the left ear was quite easy (just taking the mirror image of the above, which is always 180 degrees).

1. Mono channel

The first implementation is the most simple, the mono channel one. It was a first approach to understand how the whole process of reading the input buffer blocks works. It also creates a reference output that can be compared to other algorithms. As the name indicates, it consists of sending just one file to both outputs (left and right). It is used just one channel, a file recorded with one microphone. As a result we obtain a flat sound, the same info for both ears.

2. Stereo channel

The second implementation is stereo, the right and left channels no longer receive the same samples. This is the traditional rendering. In this case, the sum of the samples of half of the microphones, the ones located closer to the right ear, are sent to the right output channel. In order to not exceed the number of bits of each sample (maximum 16 bits, as previously mentioned), each sample is divided by half of the total number of channels. That is, if there are 16 input channels, every sample of the 8 microphones closest to the right ear is divided by 8. The result of this weighted sum is sent to the output right channel. The same procedure is followed for the left channel with the other half of microphones. The rotation angle is not yet taken into account.

As an example, if there are 8 channels, the output channels receive the addition of the weighted samples of the following microphones:

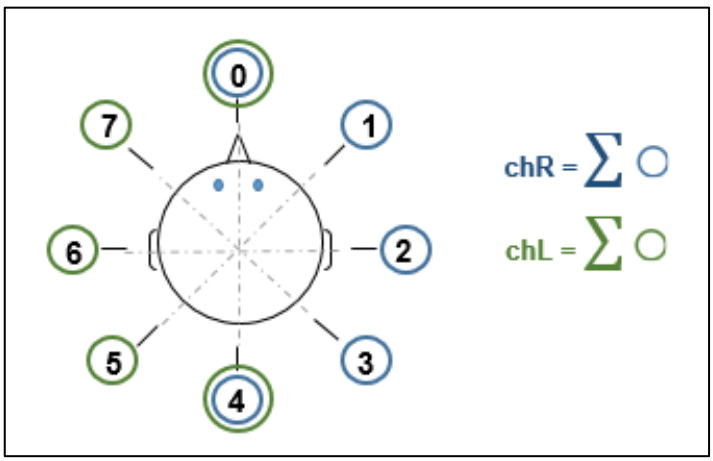


Figure 8. Stereo channel

3. Addition

This algorithm sets the addition of all the samples of each microphone to both left and right output channels. In this occasion, each sample is first divided by the total number of microphones to avoid exceeding the 16 bits when doing the addition. Then, all the weighted samples are summed and the result is sent to both output channels. This is not rendering a stereo image.

To realize what would happen if the samples were not weighted, in the document Results section there have been represented two specific charts. One of the charts was obtained dividing the samples (weighted addition) and the other one without performing the weighting (addition without weighting). As seen in the graph a signal clipping was produced as a consequence of channel saturation.

4. Full Range Nearest Microphone Selection (FR-NM)

This is the first algorithm implemented using the documentation of the MTB rendering application. It is the more direct alternative but not very sophisticated. In this approach the mic located next to each of the listener's ear positions is taken. The acoustic diagram is fragmented into N cake sections (N being the number of available microphones) each sizing ΔΦ=360º/N. Within a section, it is always taken the same mic. There is no accurate mic weighting.

This solution presents a discontinuous behavior. Each time when crossing the different boundaries that define the acoustic space, switching clicks and strange sonic artifacts can be expected to become audible.

5. Full Range Linear Interpolation (FR-LI)

The second solution of the MTB rendering application consists on linearly cross fading between two adjacent microphone's signals. In other words, the signal x(t) at the ear's position can be interpolated from the output of the nearest microphone ($x_n(t)$) and the output of the following one ($x_{nn}(t)$) according to the equation:

(4) $$x(t) = (1 - w) * Xn(t) + w * Xnn(t)$$

where the weight w is determined as the ratio of the angle between the ear and the currently nearest microphone β and the average angular microphone distance ΔΦ (w = β/ΔΦ).

V. R. Algazi also determined the minimum number of microphones needed to keep MTB's magnitude response within ±3dB deviation below a certain frequency f_{up}. But with that equation, when setting f_{up} to 20 kHz a large number of microphones would be needed, which is not efficient.

In addition, in order to eliminate comb filter clicks and sonic artifacts as resulting from the interpolated MTB signal, a low pass (anti-aliasing) filter could be used. But the high frequencies are important too so improved and more efficient algorithms are required.

In the program, to determine and implement the equation (1), there were used two new variables: phi1 and phi2 in order to weigh the microphones. Therefore, the equation (1) can also be written like:

(5) $x(t) = \frac{Xnn(t)*phi1 + Xn(t)*phi2}{phi1+phi2}$, being phi1 the smallest region and phi2 the largest (phi1+phi2 equals the section). Therefore, it is given less weight (phi1) to the next nearest microphone and more weight (phi2) to the nearest microphone.

So, in order to set the right channel, the section of the right ear is divided into two sub-sections, phiR1 and phiR2, and then the channels are weighted for those values as explained above.

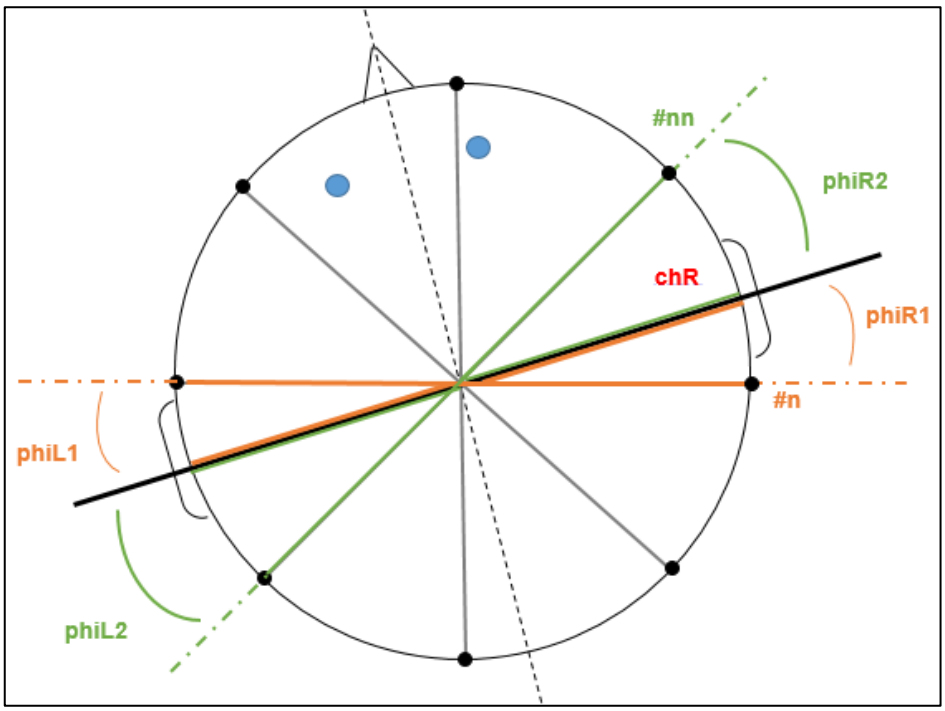


Figure 9. Sub-sections of each ear

It has been proven that, despite the fact that the following implementations had to be better than this one, the best results (based on empirical listening tests) were achieved with the 5th algorithm as defined on MTB rendering application (the TB-SI, not yet explained) and this algorithm, the FR-LI.

6. Low Pass Filter

A simple LPF algorithm was implemented in order to study how audio samples were altered.

The equation used is:

(6) y(n) = y(n-1) – (BETA*[y(n-1)-x(n)])

where BETA = 0.025. The effect of this filter is to smooth the signal reducing the amount of high frequency components of the signal and allowing the low frequency ones. The BETA variable determines the filter smoothing effect.

This implementation can be further improved using the filters used in the MTB rendering application.

7. High Pass Filter

This resource was added to increase accuracy by enhancing the nearest microphone high frequency directivity effect.

The following graph shows how listening angles are affected by frequency. A low frequency is not affected by the listening rotation angle.

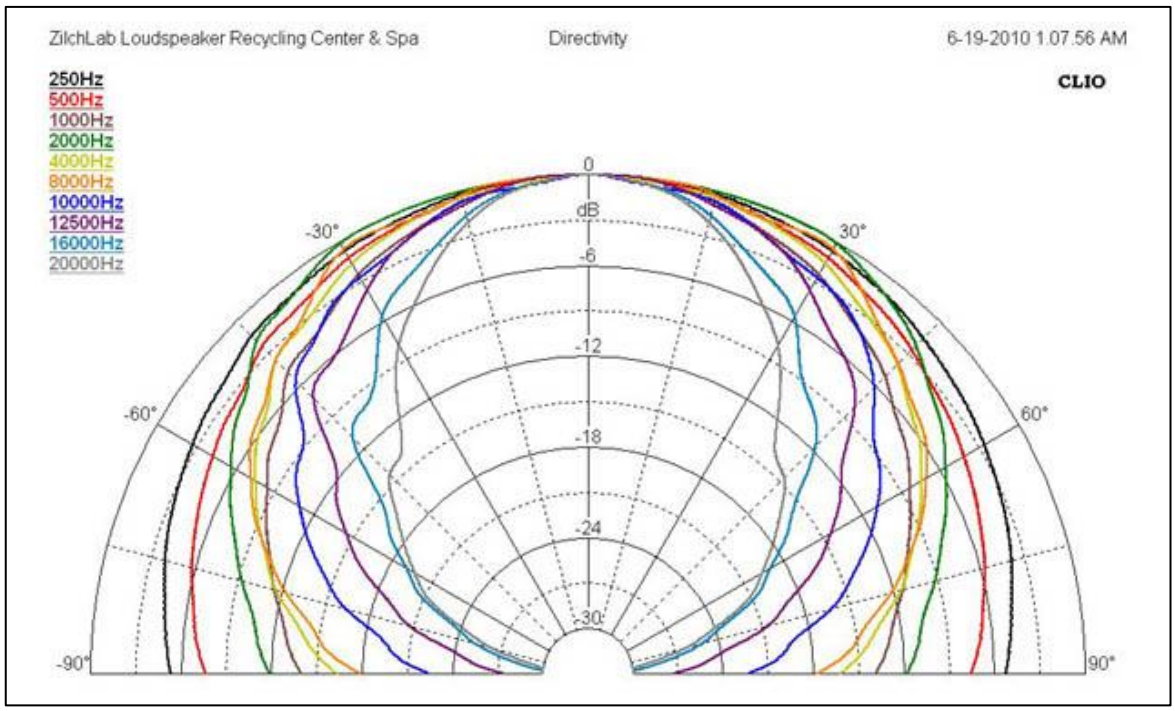


Figure 10. Sub-sections of each ear ([20])

The equation used is:

(7) $y(n) = ALPHA * (y(n-1) + x(n) – x(n-1))$

where ALPHA = 0.5. The effect of this filter is to preserve signal ripple by reducing the amount of low frequency components. The ALPHA variable determines the filter effect.

This implementation can also be further improved using the filters used in the MTB rendering application.

8. Two Band Fixed Microphone Interpolation (TB-FM)

This algorithm reproduces the higher frequencies from a high pass filtered fixed omnidirectional microphone, a complementary microphone. Thus, high frequency spectral energy is restored, but spatial information will be lost. With this solution, not only would the disturbance of the spatial auditory perception be a problem, but also the decision of where the complementary microphone should be located to achieve the best performance.

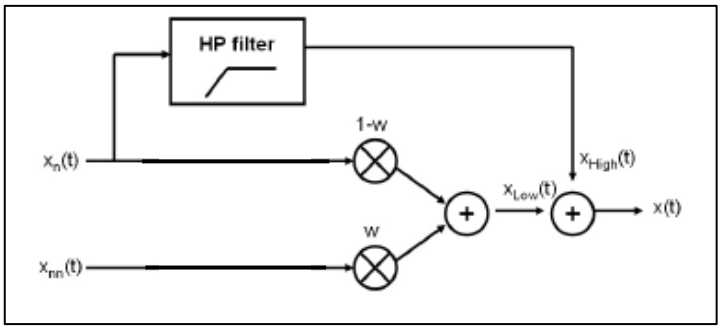


Figure 11. Scheme of the TB-FM algorithm ([1])

9. Two Band Nearest Microphone Selection (TB-NM)

An improvement to the previous algorithm would be applying the FR-NM algorithm to the high frequency audio range in order to improve stability of source localization. Low frequency content is derived from continuously interpolating between nearest and next nearest microphone signals, $x_n(t)$ and $x_{nn}(t)$. High frequency content is derived through switching to nearest microphone signal $x_n(t)$.Thus, the algorithm would work as follows:

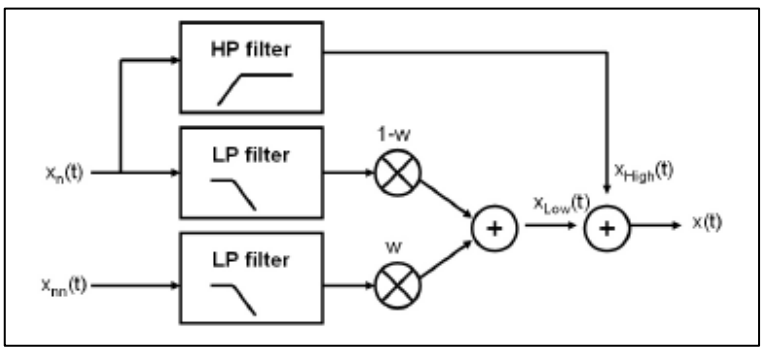


Figure 12. Scheme of the TB-NM algorithm ([1])

10. Two Band Spectral-Interpolation Restoration (TB-SI)

Finally, this algorithm offers the best performance but it is also the most complex. It is the last algorithm presented in the MTB rendering application. Using fast Fourier transform, linear interpolation can also be conducted in real-time in the spectral domain. So that the initial equation would become:

(8) $M_c(\omega)= (1-w)* M_n(\omega) + w* M_{nn}(\omega)$

And the block diagram:

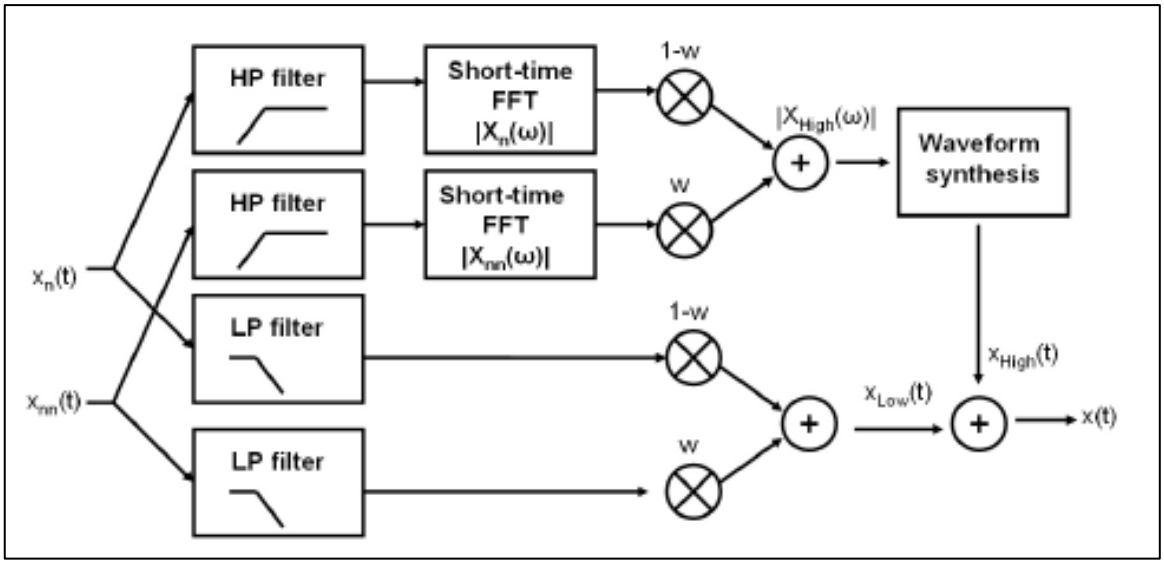


Figure 13. Scheme of the TB-SI algorithm ([1])

Low frequency content is derived from continuously interpolating between nearest and next nearest microphone signals, $x_n(t)$ and $x_{nn}(t)$. High frequency content is derived through interpolation of short-time magnitude spectra and phase reconstruction during waveform synthesis.

3.6. The Unity game

The game user interface created in Unity consists of a human body figure sitting in a concert hall. Figure's head can be rotated in order to simulate the listening rotation angle.

Figure 14. Game user interface

At the beginning the user has to select the rendering algorithm that wants to try.

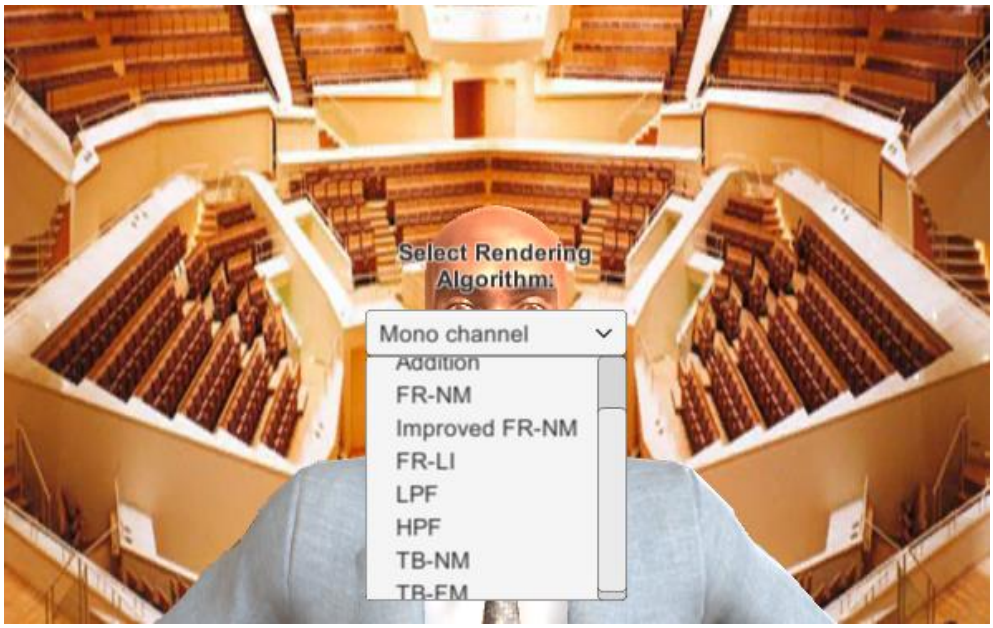


Figure 15. Selection menu

Apart from designing the game scene, a C# script was implemented in order to synchronize the C++ code embedded in the dll with the Unity game. The C# script has different options that can be configured before executing the application.

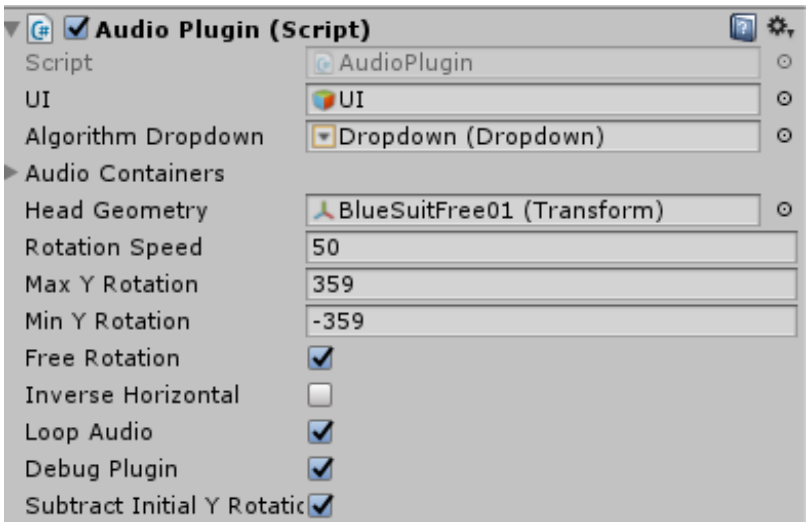


Figure 16. Configuration panel of the audio plugin script

The *Audio Containers* contain all the wav files, 16 in our case.

The *Free Rotation* option allows the user to rotate the head with no degree limitations. In other words, if it is selected the human can be rotated 360 degrees. If it is not selected, the user can choose the value of the boundaries (max Y rotation and min Y rotation variables) for the rotation angle.

The *Inverse Horizontal* option allows the user to swap the right and left arrow keys from the keyboard.

The *Loop Audio* option allows to execute the program in a looping mode. Thus, the music never stop playing.

The *Debug Plugin* option allows to execute the program in the debug mode. That is, an output text file is created when running the game as a way to debug the program to find and correct errors.

The *Subtract Initial Y Rot* option can be used in order to adapt the degree values depending on the initial position of the human body (looking at the front or at the back). If it is selected, it subtracts the initial position to the current Y rotation value.

Once the C program was synchronized with the Unity game, it was not easy to debug the C++ code to track errors. To solve this problem, a standalone executable version in C++ was also developed. That is, the same audio plugin program was copied into a new executable project originating an .exe file instead of a .dll. This executable simplified the code debugging and avoided building Unity game every time new modifications were done.

Finally, to run Unity game, user only needs to run the executable file created by Unity and arrange the folder containing all the audio files.

4. Results

In order to understand how different algorithms work and the way they combine input samples to drive the output audio device, several graphs have been created. These graphs contain the samples of both channels right and left once processed.

In the following figures 16 input channels (wav files) and also fixed rotation angles (0 or 180 degrees) were used in order to check the differences between algorithms. There were used always the same amount of samples, 200 samples (from sample 100 to 300) belonging to the beginning of the files. The charts are in the time domain.

- Mono Channel

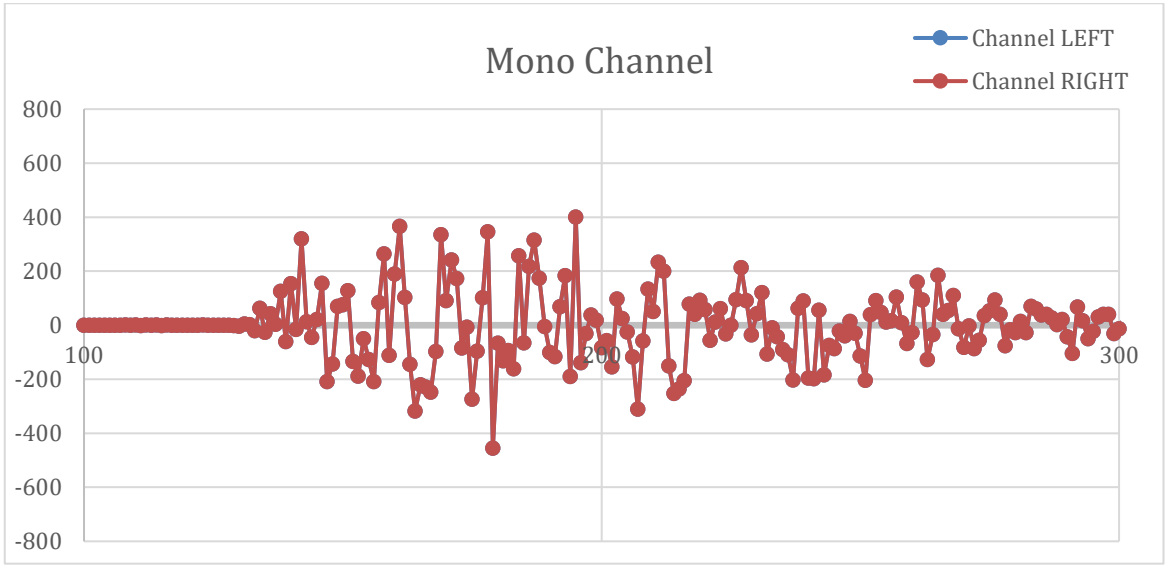


Figure 17. Mono Channel chart

Only one color can be identified because the two output channels values are the same and are therefore superimposed.

- Stereo Channel

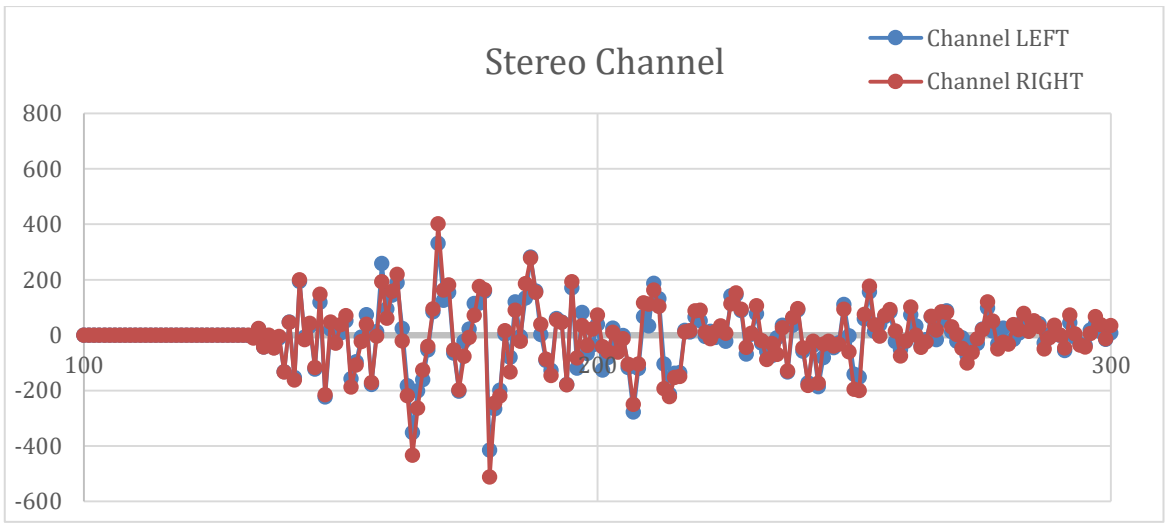


Figure 18. Stereo Channel chart

Both channels can be distinguished. The samples shown per each channel are the result of the addition of half of the total number of microphones.

- Weighted addition

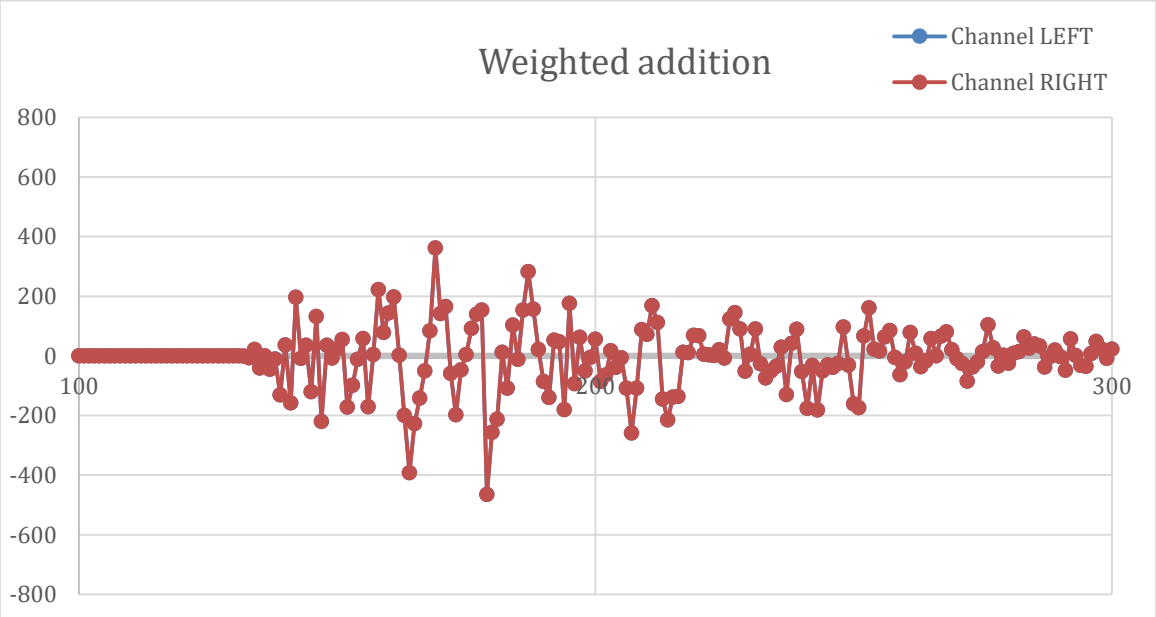


Figure 19. Weighted addition chart

The same information is sent to both channels again. This is, the addition of the samples of all the microphones, first weighted.

In order to see what would happen if each sample was not weighted before the addition, the following chart was represented.

- Addition without weighting

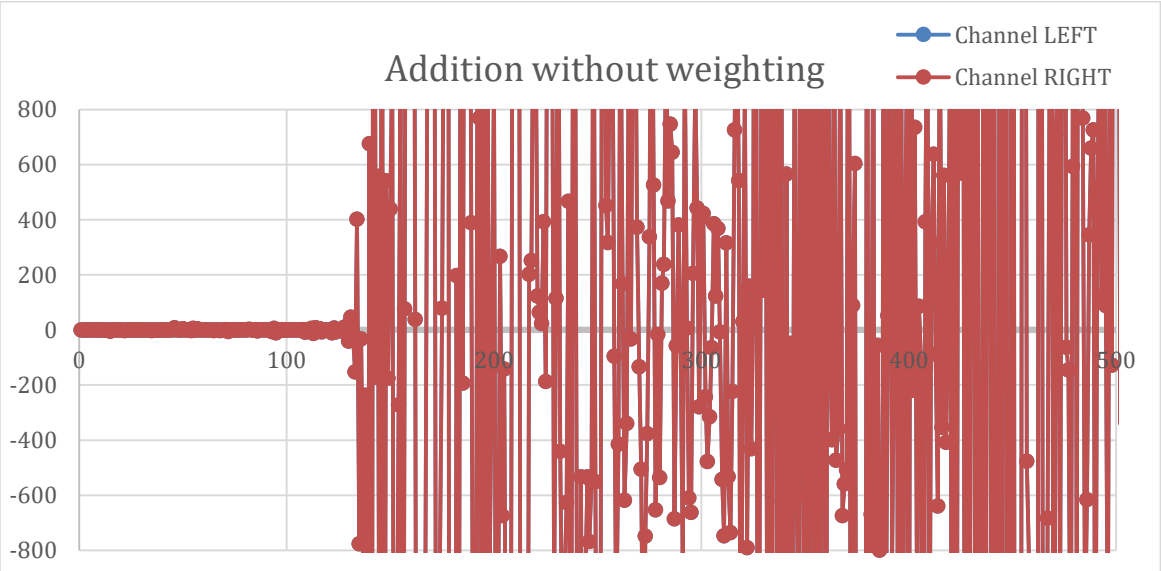


Figure 20. Addition without weigthing chart

The samples exceed the limits of the chart.

- Full Range Nearest Microphone Selection (FR-NM)

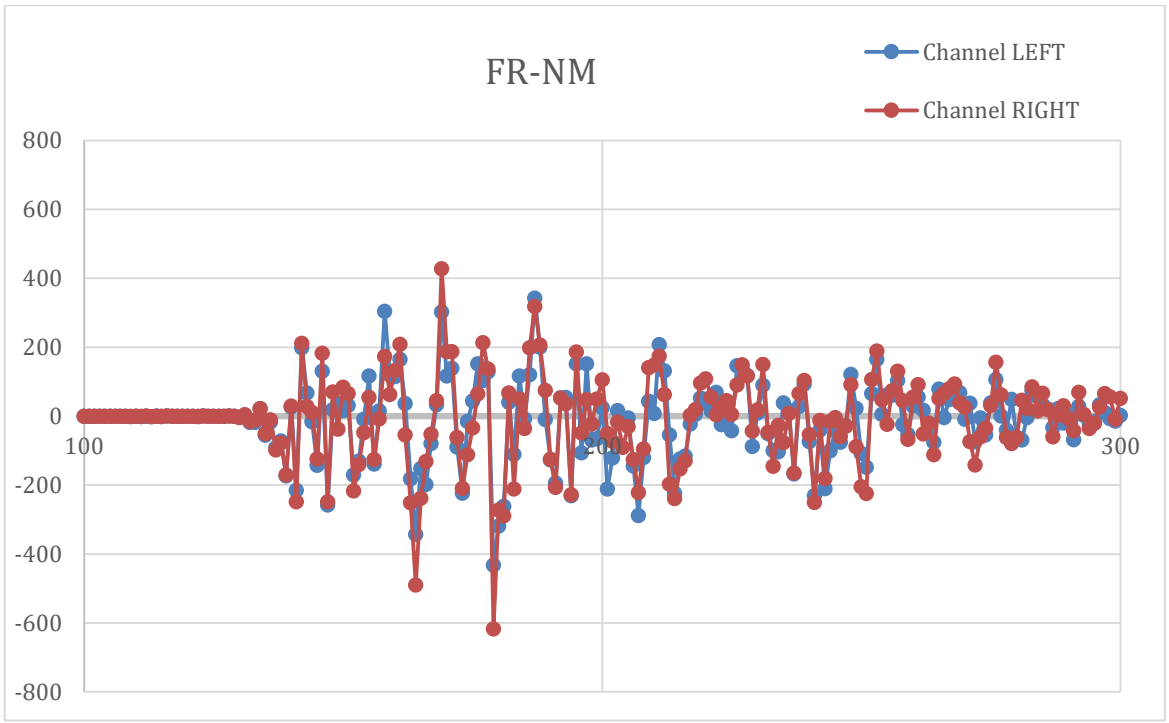


Figure 21. FR-NM chart

This case is quite similar to the above stereo situation. However, instead of picking up half of the closest microphones to the right ear, only the nearest one is taken. When tacking into account more signals, results tend to average.

- Full Range Linear Interpolation (FR-LI)

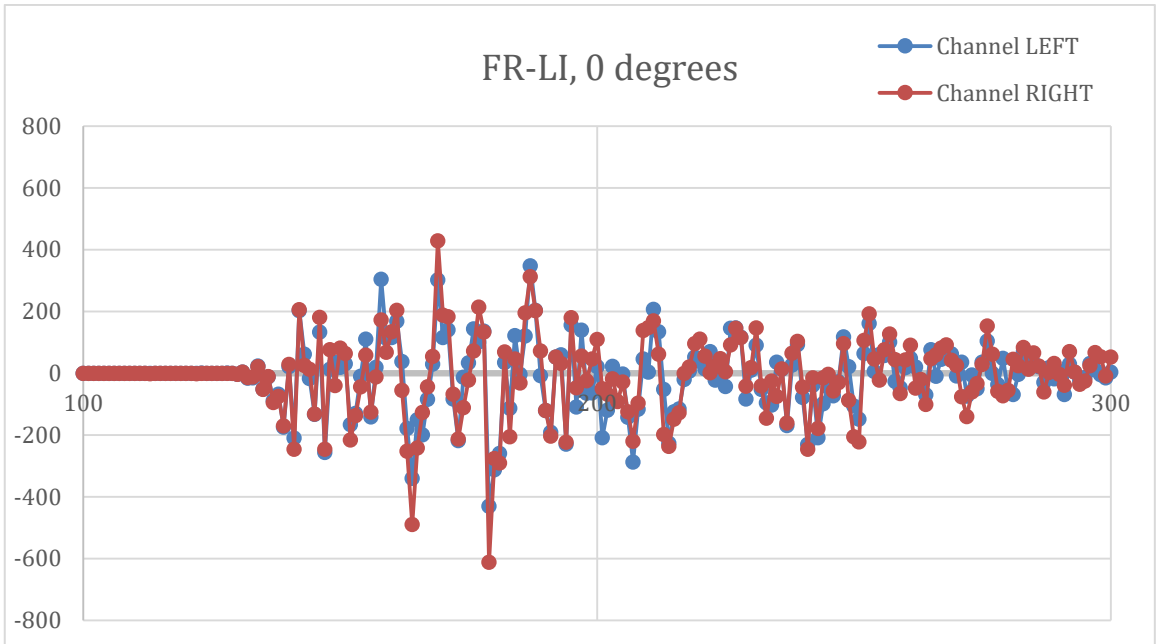


Figure 22. FR-LI, 0 degrees chart

This result is quite similar to the FR-NM one. Two adjacent microphone's signals are linearly cross faded interpolating the signal from the nearest microphone and the next nearest microphone.

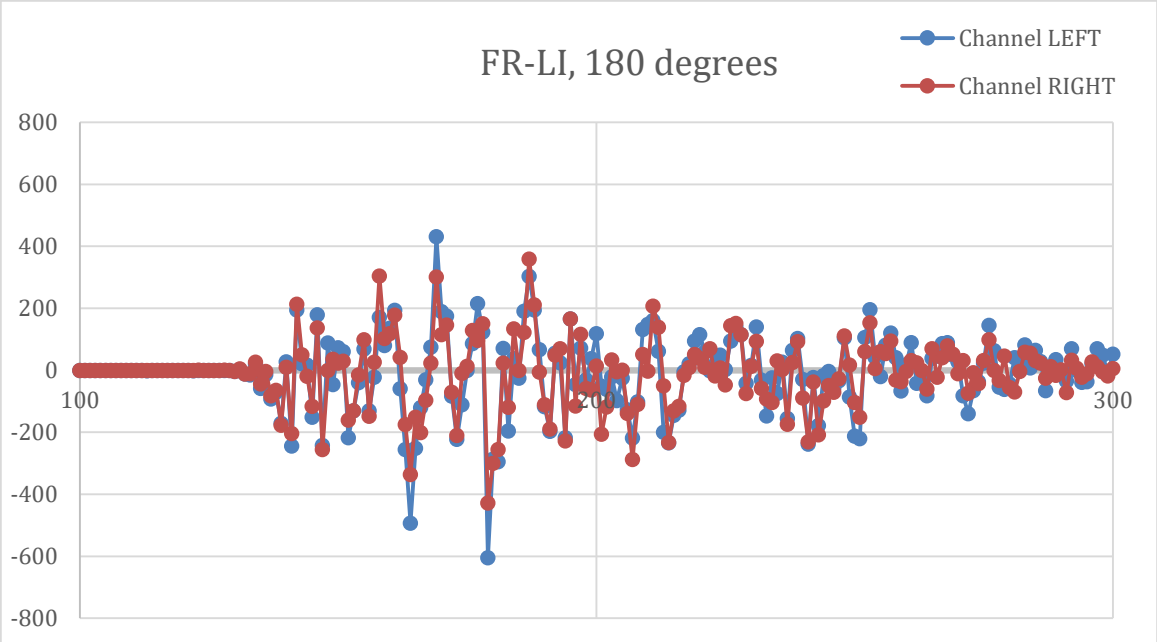


Figure 23. FR-LI, 180 degrees chart

In this case, it has only been changed the rotation angle from 0 degrees to 180 degrees in order to check the results when taking different number of microphones.

- Low Pass Filter (LPF)

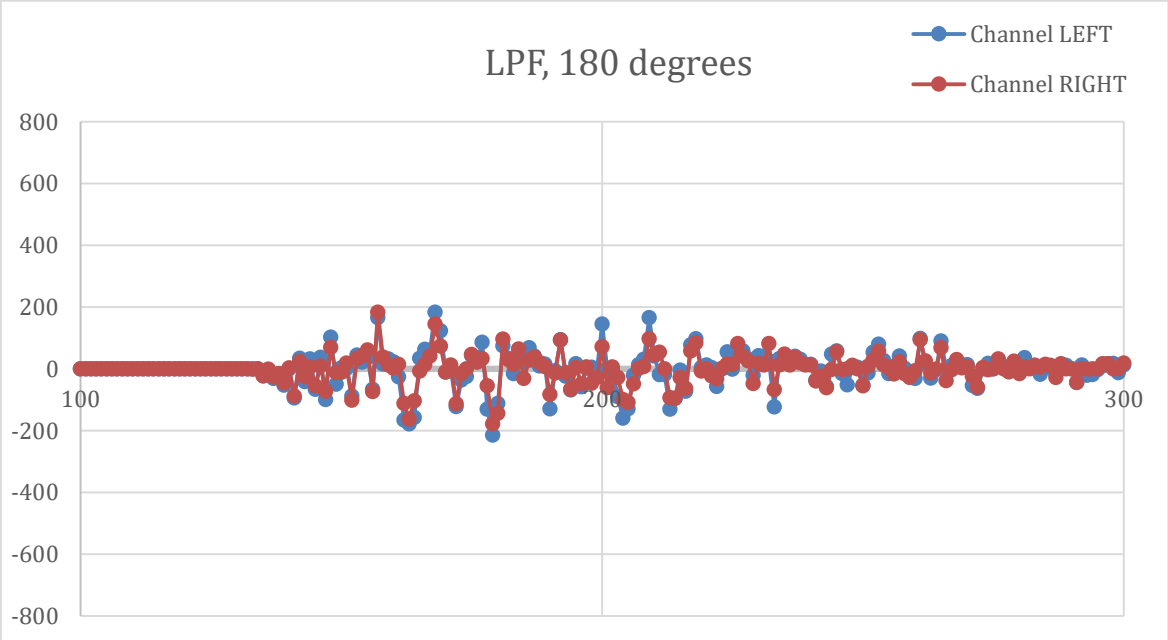


Figure 24. LPF chart

In this case, it can be noticed how the low pass filter allows only the soft part of the signal.The highest peak has a value of 183 instead of 431 like in the previous cases.

- High Pass Filter (HPF)

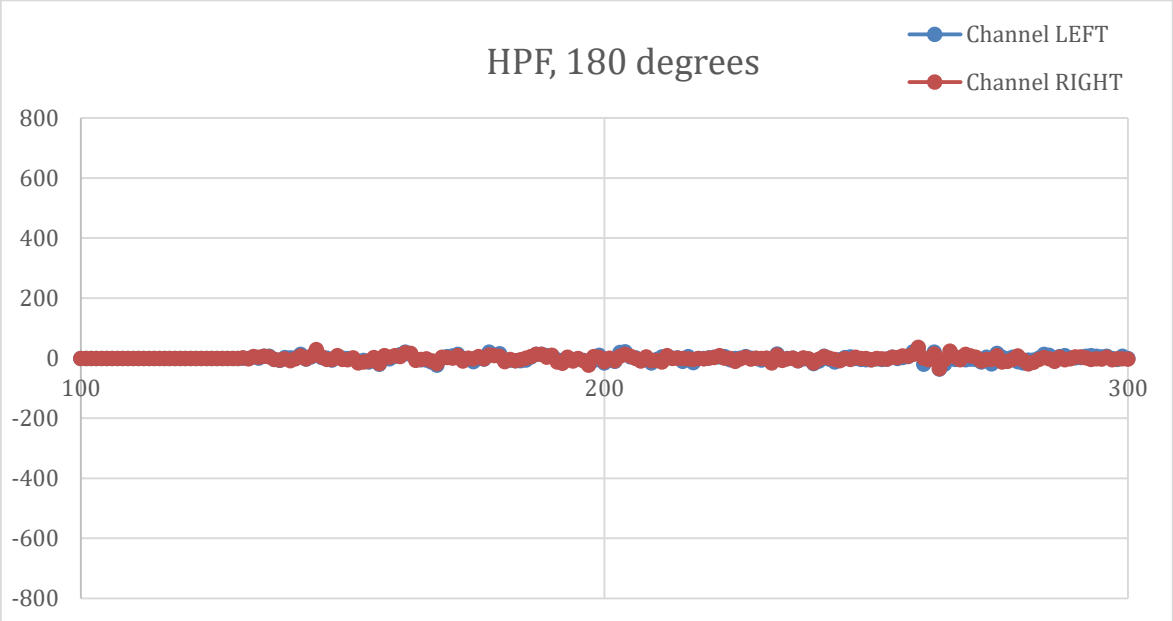


Figure 25. HPF chart

In this case, it can be noticed how the high pass filter preserves only the ripple of the signal.

- Two Band Fixed Microphone Interpolation (TB-FM)

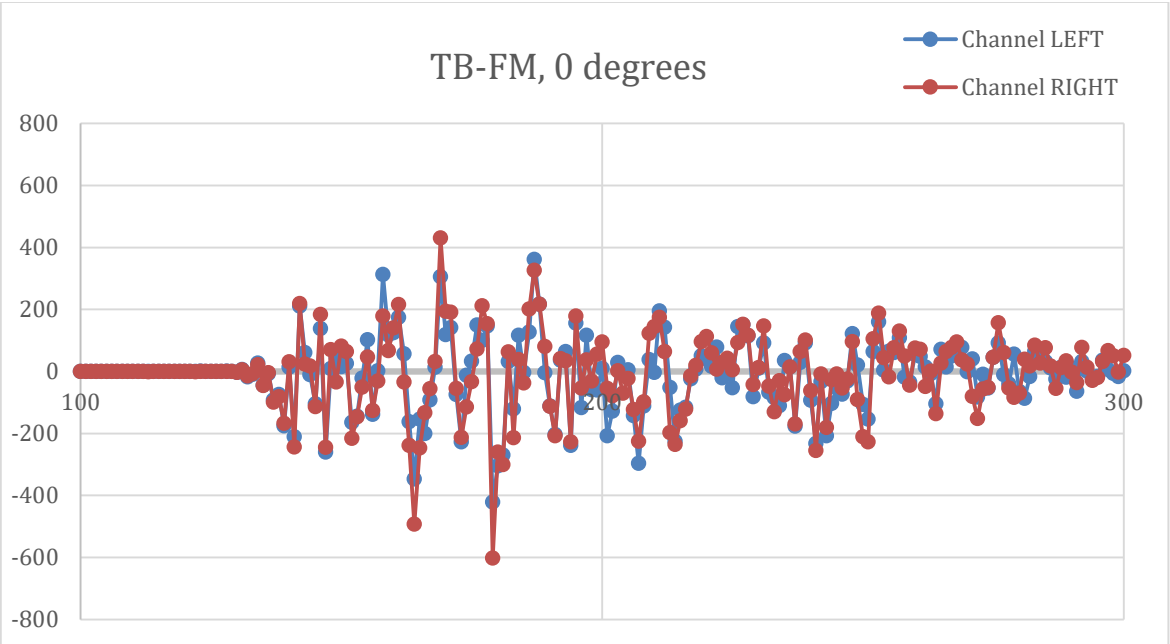


Figure 26. TB-FM chart

This result is quite similar to the FR-LI one. The only system difference comparing it to the FR-LI algorithm is the addition of a HPF. Other music files may provide more different results but, in this project, wav files belong to the recording of a person's speech. In general, voice frequency band ranges from approximately 300 Hz to 3400 Hz, which are

not high values. That is the reason why there cannot be seen many differences between these charts.

- Two Band Nearest Microphone Selection (TB-NM)

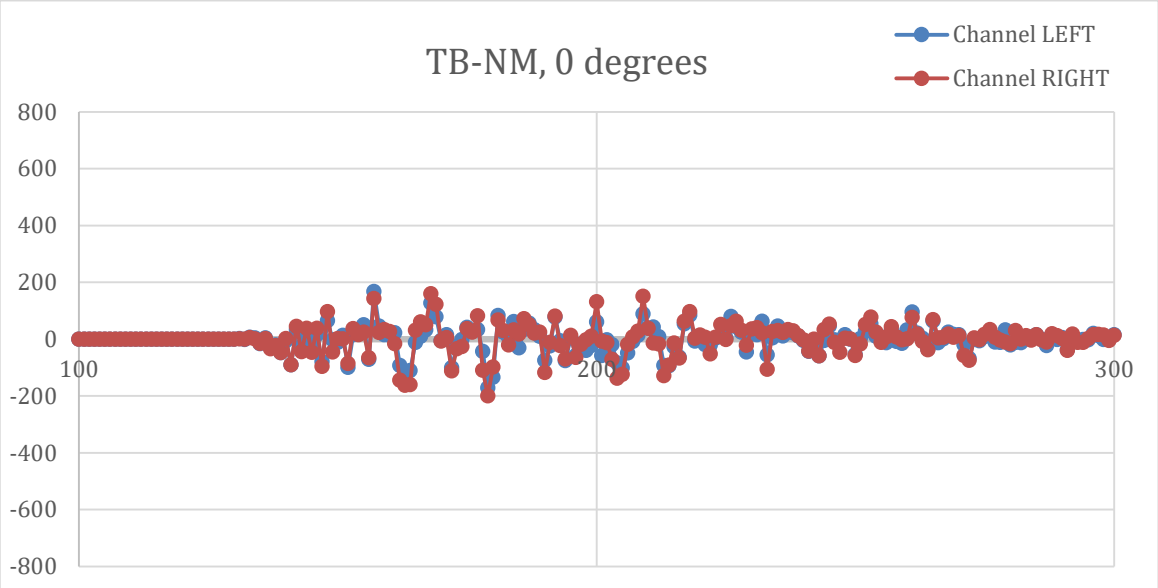


Figure 27. TB-NM chart

By adding the low pass filters the results change greatly. The sample values are quite attenuated comparing them to the original ones.

5. Budget

This section is to provide a general figure to summarize total project costs. Just to get a raw value, we've considered that three engineers have worked in the project providing costly efforts.

Project Staff

Clara Rendé

- Position: field junior engineer
- Euros/hour: 10 € / hour

Markus Hädrich

- Position: Project supervisor
- Euros/hour: 60 € / hour

David Ackermann

- Position: Project contributor
- Euros/hour: 50 € / hour

Project Timing

Start date: 20/09/2016

End date: 10/01/2017

Staff	Working days	Total hours	Total Salary
Engineer 1	75	600	6.000 €
Engineer 2	75	10	600 €
Engineer 3	75	10	500 €

Table 1. Staff project costs

Software

Unity software license: Free (Unity personal), 35€ (Unity plus), 125€ (Unity pro)

Total Cost (approximate): 7.100 € + 0€ = 7.100 €

6. Conclusions and future development:

The main goal of the project has been successfully achieved: the implementation of a 3D sound rendering application as a Unity plugin.

Referring to the future development, there are multiples possibilities to be done.

First, the fifth interpolation algorithm from the MTB rendering application has not been yet implemented due to the fact that it requires a powerful machine in running time. It is the Two Band Spectral-Interpolation Restoration (TB-SI), which uses the short-time Fourier transform. Therefore, the first step would be to implement it.

Moreover, in order to implement the 5th algorithm and also to improve the ones that use the DSP filters, the libraries used in the rendering application can be used. For the moment, basic equations for the high and low pass filters were used because it was not an added value issue for the project to fully develop a filter.

Furthermore, the two optional additional goals mentioned in the introduction can be also implemented. The development and deployment of a standalone app for Android or iPhone and also the improvement of this smartphone app using a head-mounted display and 3D sound over headphones. To do so, the Unity audio SDK has to be used instead of the Microsoft multimedia library just to be able to deploy the game into several playback devices and not only the Windows platform.

Finally, the application latency is a bit appreciable. The sound does change after releasing the arrow keys for a short time. This could be due fact that the input key control process done by the operating system is a synchronous process. This is, all the pressed keys are stored in an intermediate buffer that is read in a timed based frequency. If the keys are pressed continuously, the intermediate buffer is filled and a second process synchronised with it has also to read this critical buffer. That process is controlled by the operating system itself. So, it is not possible to improve this little delay.

Moreover, the whole process is interrupted every 5,8ms per block, which is not relevant. If the process was interrupted every 10ms nothing would change so we have a margin of error. The efficiency also depends on the computer. A good computer will let the program run faster. A minimum analysis of resources impact (memory and CPU usage) was done and the results are the following ones:

- Memory usage: 32,7 MB (out of 8GB)
- CPU (% of all processors): first 25ms: 25%, then: maximum 5%

These results are practically negligible. However, what this does not mean is that there can be no more relevant impacts on slightly more complex applications. Therefore, another future step to be done would be an accurate analysis of the impact of latency and performance. The Diagnostic tools window in Visual Studio could be used.

Bibliography:

[1] Lindau, Alexander; Roos, Sebastian. “Perceptual evaluation of discretization and interpolation for motion-tracked binaural (MTB) recordings”. TU-Berlin, November 2010.

[2] Algazi, V. Ralph; Avendano, C.; Duda, Richard O. (2001): "Estimation of a Spherical- Head Model from Anthropometry." In: *J. Audio Eng. Soc*., Vol. 49, No. 6, pp. 472-479.

[3] Algazi, V. Ralph; Duda, Richard O.; Thompson, Dennis M. (2004): "Motion-Tracked Binaural Sound." In: *J. Audio Eng. Soc*., Vol. 52, No. 11, pp. 1142-1156.

[4] Algazi, V. Ralph, “Dynamic binaural sound capture and reproduction”, U.S. Patent 7 333 622, Feb 19, 2008

[5] Algazi, V. Ralph; Duda, Richard O.; Thompson, Dennis M.; Avendano, C.: "The cipic HRTF database." In: *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*., New York, October 2001.

[6] Algazi, V. R., Avendano, C., and Duda, R. O. (2001a). ‘‘Elevation localization and head-related transfer function analysis at low frequencies,’’ *J. Acoust. Soc. Am*. 109, 1110–1122.

[7] Algazi, V. R., and Duda, R. O. (2002). ‘‘Approximating the head-related transfer function using simple geometric models of the head and torso,’’ *J. Acoust. Soc. Am*. 112, 2053–2064.

[8] Hans-Petter Halvorsen: “Introduction to Visual Studio and C#”. University College of Southeast Norway, 2016.

[9] Acodemy: “Learn C++ in a day”. Acodemy, 2015.

[10] Ray Yao: “C++ in 8 hours”. Ray Yao, 2015.

[11] Jonathan Linowes: “Unity Virtual Reality Projects”. Packt Publishing Ltd, 2015.

[12] John P.Doran: “Unity Game Development Blueprints”. Packt Publishing Ltd, 2014.

[13] Simon Jackson: “Unity 3D UI Essentials”. Packt Publishing Ltd, 2015.

[14] Claudio Scolastici: “Unity 2D Game Development Cookbook”. Packt Publishing Ltd, 2014.

[15] Alan Thorn: “Mastering Unity Scripting”. Packt Publishing Ltd, 2015.

[16] Janine Suvak: “Learn Unity3D Programming with Unity Script”. Heinz Weinheimer, 2014.

[17] Alex Okita: “Learning C# Programming with Unity 3D”. Taylor & Francis Group, 2015.

[18] Lee ZhiEng: “Building a Game with Unity and Blender”. Packt Publishing Ltd, 2015.

[19] V.R. Algazi, R.O. Duda and D.M. Thompson, “Motion-Tracked Binaural Sound,” J. Aud. Eng. Soc., Vol. 52, No. 11, pp. 1142-1156, November 2004.

[20] Dr. Earl Geddes, “Directivity in Loudspeaker Systems”, GedLeeLCC

[21] Microsoft: “Wafeform Audio Reference”, (https://msdn.microsoft.com/es-es/library/windows/desktop/dd743833(v=vs.85).aspx)

Glossary

MTB: Motion-Tracked Binaural

FR-NM: Full Range Nearest Microphone Selection

FR-LI: Full Range Linear Interpolation

TB-FM: Two Band Fixed Microphone Interpolation

TB-NM: Two Band Nearest Microphone Selection

TB-SI: Two Band Spectral-Interpolation Restoration

HPF: High Pass Filter

LPF: Low Pass Filter

HMD: Head-Mounted Display

VR: Virtual Reality

AES: Audio Engineering Society

TUB: Technische Universität Berlin

VDT: Verband Deutscher Tonmeister

VAEs: Virtual Acoustic Environments

ITD: Interaural Level Differences

ITD: Interaural Time Differences

LPCM: Linear Pulse Code Modulation

PCM: Pulse Code Modulation

ASCII: American Standard Code for Information Interchange

RIFF: Resource Interchange File Format

WAVE: Waveform Audio File Format

LSB: Least Significant Byte

MSB: Most Significant Byte

IDE: Integrated Development Environment

DSP: Digital Signal Processing

API: Application Programming Interface

FFT: Fast Fourier Transform

DLL: Dynamic-Link Library

CIPIC: Common Programming Interface for Communications

FIFO: First In First Out

SDK: Software Development Kit