Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Síntesis de audio 3D con filtros HRTF

3D Sound synthesis with HRTF filters

Alumno: Felipe Andrés Álvarez Avaria

Tutor: José Ignacio Estévez Damas

La Laguna, 12 de marzo de 2021

D. **José Ignacio Estévez Damas**, con N.I.F. 43.786.097-P profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A (N)

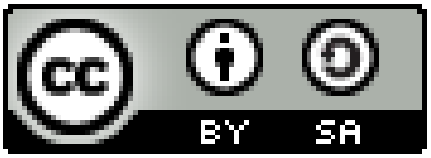
Que la presente memoria titulada:

”*Síntesis de audio 3D con filtros HRTF*”

ha sido realizada bajo su dirección por D. **Felipe Andrés Álvarez Avaria**, con N.I.F. 55.366.894-Y.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 12 de marzo de 2021

Licencia



Resumen

En la industria de los videojuegos el sonido ha quedado relegado a un segundo plano con el masivo desarrollo gráfico producido en las últimas décadas. Esto se debe en parte a las innovaciones tecnológicas que cada vez acercan más el plano virtual a una visión realista (ej. Ray Tracing), y a la disponibilidad de recursos para su investigación. Sin embargo, para que un videojuego sea inmersivo es imprescindible que el sonido acompañe a ese estímulo visual, proporcionando al jugador una experiencia 3D.

El objetivo principal de este trabajo ha sido implementar en C++ un sistema de audio 3D basado en filtros HRTF, que permiten comprender el funcionamiento de la percepción auditiva del ser humano. Esto genera la creación de muestras de audio binaural, que permiten simular la direccionalidad al sonido, aportando naturalidad al sonido en la experiencia de juego.

Palabras clave: HRTF, audio binaural, audio 3D, xAudio2, videojuegos, C++

Abstract

In the video game industry, sound has been left behind with the massive graphic development produced in recent decades. This is due, in part, to technological innovations that increasingly bring the virtual field closer to reality (e.g. Ray Tracing), and to the availability of resources for research. However, for a video game to be immersive it is essential that the sound complement visuals, providing the player with a 3D experience.

The main objective of this work has been to implement in C++ a 3D audio system based on HRTF filters, which allow us to understand the functioning of human auditory perception. This generates the creation of binaural audio samples, which allow the directionality of the sound to be simulated, giving the sound naturalness to in-game experience.

Keywords: *HRTF, binaural audio, audio 3D, xAudio2, video-games, C++*

Índice general

Índice de Figuras

Índice de Tablas

Listings

Capítulo 1

Introducción

El objetivo de este proyecto es desarrollar un sistema de sonido 3D. Estudiaremos las posibilidades tecnológicas de las librerías de audio 3D modernas, y analizaremos las características necesarias para que el jugador pueda disfrutar de una buena simulación de audio, pudiendo esta ser crucial si se juega competitivamente a juegos en primera persona como Counter-Strike o similares.

El Audio 3D no sólo se usa en aplicaciones multimedia como pueden ser el cine o equipos de música home-cinema, sino que también tiene su utilidad para simulaciones en entornos 3D y videojuegos en primera persona. Con un sistema como el que se quiere desarrollar en este proyecto es posible integrarlo en los videojuegos para brindar una nueva dimensión de contenido, la direccionalidad del sonido.

Los videojuegos con las mejoras tecnológicas se han ido cada vez aprovechando más del procesamiento de audio. Empezando por un pequeño conjunto de sonidos simples guardados en las consolas en formato analógico, como alguna versión del Pong, pasando por el audio digital de baja fidelidad y los primeros microprocesadores que manejaban audio (segunda y tercera generación de videoconsolas). Con el tiempo y la mejora de este tipo de microprocesadores, cada vez se obtenía una resolución mayor. Hubo mejoras notables una vez se pasó de los 8bit a los 16bit de audio, vistos por primera vez en la cuarta generación de consolas (PlayStation, XBox), así como cuando por primera vez salieron al mercado sistemas estéreo, los sistemas mono quedaron obsoletos en cuanto a calidad.

Una vez llegamos a calidad CD o 24bits a 44.1Khz de frecuencia de muestreo fue cuando se detuvo ligeramente el interés por la calidad del audio, ya que se escuchaba bastante bien.

Hoy en día no sólo tenemos la posibilidad de audio a mayores calidades, sino que podemos procesar el audio en tiempo real. Este tipo de procesamiento ofrece un mundo nuevo de posibilidades al sonido, transformándolo y adaptándolo a los productos y sistemas que lo integren.

Con un sistema como este es posible ofrecer un sonido mucho más rico en detalles a cualquier experiencia audiovisual, pero este proyecto está enfocado principalmente a los videojuegos, debido a la oportunidad de usar el sonido como otra herramienta más para presentar nuevas ideas y conceptos de jugabilidad en este ámbito.

Capítulo 2

Objetivos

2.1. Objetivo general del Proyecto

Con este proyecto perseguimos la mejora del audio en los videojuegos, mediante el aprovechamiento del concepto de la escucha binaural, crear un sistema de audio que sea capaz de procesar muestras de audio y reproducirlas como muestras de audio estéreo binaural, intentando crear para el oyente una sensación de sonido 3D similar a la de estar físicamente en el lugar donde se producen los sonidos.

Esto se quiere conseguir mediante la implementación de un sistema de audio basado en filtros HRTF, modificando las muestras de audio para su conversión a audio de tipo binaural, aportando la sensación de direccionalidad al sonido, entre otros beneficios.

Además de lo anterior, se pretende estudiar y comprender el tiempo interaural, para mejorar su acotamiento y en un futuro su posible implementación en sistemas gráficos, con la finalidad de conseguir refinar la inmersión 3D del jugador a través de dispositivos como los auriculares.

2.2. Objetivos específicos

2.2.1. Implementación de filtros HRTF

Los filtros HRTF pueden ser una solución a los problemas comunes de fidelidad del audio 3D. Aplicando este tipo de filtro se intenta demostrar que se puede crear un sistema de audio 3D, donde la direccionalidad del sonido puede conseguirse de manera realista, con cualquier tipo de auriculares estéreo. El objetivo de realizar esta implementación es comprobar la dificultad de su uso tanto de manera técnica como computacional. Por otro lado también es preciso intentar entender cómo funciona esta tecnología, para desarrollar las mejoras convenientes.

2.2.2. Sistema de audio 3D Binaural

Con el objetivo de mejorar la calidad de sonido en los videojuegos, se quiere crear una librería o módulo que sea capaz de usar filtros HRTF, entendiendo que usando un sonido y una posición en el espacio tridimensional sea capaz de simular la direccionalidad de los sonidos en un entorno estéreo. Este módulo debería de ser fácilmente integrable en otra aplicación, como por ejemplo, un motor de videojuegos.

2.2.3. Comprender a “bajo nivel” los algoritmos necesarios para lograr la direccionalidad del audio

El enfoque del este proyecto está influenciado por su carácter formativo. En lugar de recurrir a librerías ya existentes que implementan la direccionalidad del audio, se tratará de programar los principales algoritmos sobre los que descansa la obtención de audio 3D. Recurrir a este enfoque no solo es interesante desde el punto de vista formativo, sino que permitirá obtener una librería con menos dependencias y con la que sea más factible la experimentación y ampliación de sus características.

Capítulo 3

Estado del Arte

3.1. Procesamiento de Audio

3.1.1. Transformada de Fourier

La Transformada de Fourier es una transformación matemática que descompone una función perteneciente al dominio del tiempo en otra función que pertenece al dominio de la frecuencia. Esta operación es reversible, haciendo posible transformar de un dominio a otro según necesidad. [2]

Formalmente, la Transformada de Fourier es una aplicación que hace corresponder una función f con otra función g, definida de la siguiente manera:

$$g(\xi) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} f(x) e^{-i\xi x} dx$$

Esta operación requiere que la señal de entrada sea una función continua (real o compleja) y su salida es una función continua de valores complejos, donde el módulo (o magnitud) de estos valores representan la cantidad de esa frecuencia presente en la función original y donde el argumento (o fase) es la fase de la onda sinusoidal base de esa frecuencia.

Las operaciones lineales que se realicen en un dominio tienen su correspondiente operación en el otro dominio, que a veces son más fáciles de ejecutar. La convolución en el dominio del tiempo corresponde a multiplicar en el dominio de la frecuencia, por ejemplo.

DTFT

Por otro lado también disponemos de una versión discreta de esta operación, con la que podemos usar una secuencia de valores como entrada. Se trata de la Transformada de Fourier de Tiempo Discreto (DTFT, Discrete-Time Fourier Transform). Ésta transformada, únicamente evalúa suficientes componentes frecuenciales como para reconstruir el segmento finito que se analiza. Se suele utilizar para analizar muestras de una función continua.

$$X_{2\pi}(\omega) = \sum_{n=-\infty}^{\infty} x[n] e^{-i\omega n}$$

DFT

Además, tenemos la Transformada de Fourier Discreta (DFT, Discrete Fourier Transform), que requiere una secuencia de valores como entrada y devuelve un intervalo muestreado de la DTFT, o lo que es lo mismo, una secuencia discreta de valores complejos.

Formalmente, la DFT transforma una secuencia de N números complejos $\{x_n\} := x_0, x_1, ..., x_{N-1}$ en otra secuencia de números complejos $\{X_k\} := X_0, X_1, ..., X_{N-1}$, definidos de la siguiente manera [3]:

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \\ &= \sum_{n=0}^{N-1} x_n \cdot \left[\cos\left(\tfrac{2\pi}{N}kn\right) - i \cdot \sin\left(\tfrac{2\pi}{N}kn\right)\right] \end{aligned}$$

La función inversa es análoga a la DFT, con distinto signo en el exponente y un factor $\frac{1}{N}$

$$x_k = \frac{1}{N}\sum_{n=0}^{N-1} X_n \cdot e^{\frac{i2\pi}{N}kn}$$

Y según el Teorema de inversión, si aplicamos la inversa de la DFT sobre una DFT obtenemos la función original:

$$x_n = \text{IDFT}\{X_k\} = \frac{1}{N}\left(\text{DFT}\{X_k^*\}\right)^*$$

A diferencia de la DTFT, al usar la DFT se entiende que la secuencia de entrada es un único periodo de una señal periódica que se extiende de forma infinita. En caso de que la entrada no sea de esa naturaleza, tenemos que aplicar una ventana (función) a la secuencia, para evitar discontinuidades en los extremos de la señal analizada.

Ya que vamos a trabajar con muestras de audio discretas, tenemos que utilizar un algoritmo que implemente la DFT.

FFT

La Transformada Rápida de Fourier (FFT, Fast Fourier Transform) es un algoritmo que computa la DFT (y su inversa) de una secuencia. Se trata de un algoritmo eficiente que puede ser ejecutado en un ordenador o en hardware especializado. Normalmente la DFT mediante evaluación directa requiere $O(N^2)$ operaciones aritméticas, mientras que con una FFT se puede obtener el mismo resultado con $O(NlogN)$ operaciones. Debido a esto es el más utilizado para el análisis de señales, ya que permite ejecutarse en tiempo real (sin mucha demora) en un ordenador convencional para aplicar filtros y realizar análisis sobre el sonido o imágenes, entre otros usos.

Vamos a usar el algoritmo de Cooley-Tukey (Ver Apéndice A.1), ya que es una manera intuitiva de calcular FFT sobre una señal.

3.1.2. Overlap-add method

Teniendo en cuenta que la multiplicación de la señal de entrada con el filtro en el dominio de la frecuencia equivale a una convolución circular, tenemos que aplicar una

corrección a la operación, debido a que la convolución circular modificaría el resultado esperado.

El Overlap-add method es un procedimiento para calcular eficientemente la convolución discreta de una señal larga con un filtro FIR, evitando que se produzca el efecto de la convolución circular.

Por definición, sea $x[n]$ la señal de entrada y sea $h[n]$ el filtro de respuesta finita al impulso (FIR), la convolución en el dominio del tiempo es la multiplicación en el dominio de la frecuencia, y podemos tomar de manera discreta esta operación:

$$y[n] = x[n] * h[n] \triangleq \sum_{m=-\infty}^{\infty} h[m] \cdot x[n-m] = \sum_{M}^{m=1} h[m] \cdot x[n-m] \quad \text{(Eq. 1)}$$

Donde los valores de $h[m]$ fuera del rango $[1, M]$ son cero:

$$h[m] = 0 \ \forall \ m \notin [1, M]$$

Esta estrategia se basa en separar la señal de entrada $x[n]$ en segmentos más pequeños y realizar la convolución con el filtro $h[n]$, por cada segmento:

$$x_k[n] \triangleq \begin{cases} x[n+kL] & n = 1, 2, ..., L \\ 0 & \text{en otro caso} \end{cases}$$

Donde L es un tamaño de segmento arbitrario. Entonces tenemos:

$$x[n] = \sum_k x_k[n-kL]$$

Por otro lado, $y[n]$ se puede expresar como una suma de convoluciones más cortas:

$$\begin{aligned} y[n] = (\textstyle\sum_k x_k[n-kL]) * h[n] &= \textstyle\sum_k (x_k[n-kL] * h[n]) \\ &= \textstyle\sum_k y_k[n-kL] \end{aligned}$$

Consiguiendo efectivamente, poder separar la señal en fragmentos superpuestos debido a que rellenamos con ceros y luego sumando las zonas que se superponen.

3.1.3. Filtros HRTF

Los filtros de Función de transferencia relacionada con la cabeza (HRTF, Head-related transfer function) representan la respuesta que tienen los oídos sobre las señales sonoras, según la posición relativa de la fuente del sonido al oyente.

Cuando un sonido se propaga y llega, tanto el tamaño como la forma de la cabeza, orejas, canal auditivo e incluso el tamaño y la forma de las cavidades nasales y orales transforman la percepción el sonido para este sujeto en particular, aumentando algunas frecuencias y disminuyendo otras.

Los humanos reconocemos la localización de un sonido gracias a que cada oído recibe una señal monoaural y luego comparando éstas señales según la diferencia en tiempo de llegada y la intensidad. (ver figura 3.1) Estas señales monoaurales provienen de la interacción de la fuente del

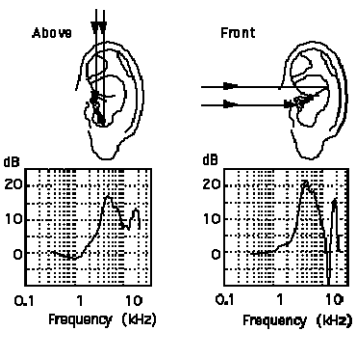


Figura 3.1: Aspecto de filtro HRTF. Autor: Mario Crespo Delgado para hispasonic.com

sonido y la anatomía humana, donde el sonido original se modifica antes de entrar al canal auditivo.

Dado que el sonido percibido varía según nuestra anatomía, un conjunto de filtros HRTF específico no es compatible con todas las personas, ya que puede variar considerablemente de una persona a otra. Sabiendo esto, al diseñar este tipo de filtros se intenta que sean lo más generales posible, consiguiendo un mayor o menor efecto según la persona.

Este tipo de filtros se construyen teniendo en cuenta estos parámetros, normalmente usando un equipo de grabación que consiste en un par de micrófonos localizados donde estarían los tímpanos en un modelo de una cabeza humana hecha de plástico o algún otro material. De esta manera podemos medir la diferencia entre las dos señales monoaurales que llegan a cada micrófono y cómo afectan los elementos indicados anteriormente. Normalmente un equipo de grabación de este tipo se usa en una cámara anecoica o similar, para evitar reflexiones tempranas debido al eco.

Un par de filtros HRTF se pueden usar para sintetizar sonido binaural de tal manera que se escucharía como si la fuente del sonido estuviera en un punto arbitrario del espacio.

3.2. Audio3D aplicado a los videojuegos

El audio 3D en el desarrollo de videojuegos consiste en aplicar distintas técnicas y transformaciones a las muestras de audio para simular el origen de cada sonido en un espacio 3D, además de otros objetivos, como aplicar filtros a las muestras para simular que es escuchada en condiciones del espacio por el que se supone que se transmitiría el sonido (bajo el agua, en una habitación con eco...), además de otros tipos de efectos como la oclusión por obstáculos entre el emisor y el oyente.

3.2.1. Estado actual

Actualmente existen librerías que implementan audio en tres dimensiones, pero con funcionalidad limitada, centrándose en imitar el funcionamiento de una fuente de sonido, para ahorrar en tiempo de procesamiento. Muchas veces se ven sistemas simples adaptados a altavoces estéreo, que permite reproducir una muestra de audio en el canal izquierdo, el derecho o en ambos.

De esta manera con altavoces convencionales estéreo, en un videojuego podemos diferenciar con cierta precisión si un sonido proviene de la izquierda o de la derecha. Por el contrario, si usamos auriculares (in-ear, o diadema), el canal izquierdo es escuchado sólo por el oído izquierdo y el canal derecho sólo por el oído derecho. Al aislar de esta manera el sonido, el usuario podrá identificar la posición de la fuente según si el sonido proviene de un auricular o del otro, pero será una experiencia extraña, ya que este comportamiento no se ve reflejado en la naturaleza.

3.2.2. Algunos ejemplos

Hay proyectos y librerías que implementan mejoras sobre lo comentado, como Razer Surround, que simula un sistema de altavoces 7.1 virtuales para conseguir el efecto de sonido 3D en auriculares estéreo. Este sistema se basa en gestionar un sistema de altavoces 7.1 virtuales, donde hay dos altavoces de frente como en estéreo, pero

añadimos también uno central (para las frecuencias medias-altas, de frente), dos en los lados (izquierda, derecha) y dos atrás, en paralelo a los delanteros. El 'punto 1' de 7.1 indica que también hay otro altavoz extra designado sólo para las frecuencias más bajas (ver figura 3.2)

En el caso de que la aplicación o contenido multimedia pueda trabajar con sonido multicanal 7.1 o 5.1, este sistema funciona relativamente bien, permitiendo contar con una mayor fidelidad del posicionamiento del audio.

En el caso de que la aplicación no soportara más canales de audio que estéreo, no habría una ventaja real, ya que sería similar al sonido estéreo con el que partimos.

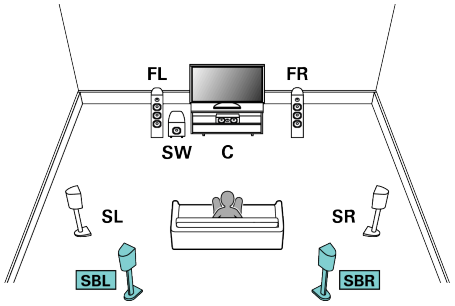


Figura 3.2: Sistema de audio 7.1 Copyright: Maranth (Sound United LLC)

Por otro lado, encontramos que en librerías comerciales existen sistemas de audio en tres dimensiones que son bastante más complejos para simular correctamente las fuentes de audio en un espacio tridimensional, aunque no se utilicen estas funciones en exceso por su alto coste computacional, ya que en muchos casos se trata de una simulación, o aproximaciones de como funciona el sonido en la vida real.

Algunos ejemplos de librerías comerciales son:

- FMOD: Utilizada en gran cantidad de juegos de compañías muy conocidas como Bethesda, RockStar y en juegos populares como Minecraft, Roblox, etc.
- WWISE: Librería que proporciona audio espacial y en donde se considera la propagación del sonido, el tipo de material que refleja el sonido, etc.
- X3DAudio: Es una API que se usa junto a xAudio2, y consigue crear la ilusión de que un sonido proviene de un punto específico en un espacio 3D.

En la misma categoría tenemos sistemas más sofisticados y complejos como Nvidia VRWorks Audio, que proporcionan muchas más características, incluyendo entre ellas el audio 3D binaural y el trazado de rayos para la computación del sonido, con el requisito de que es necesario tener una tarjeta gráfica moderna de la marca Nvidia.

Por otro lado, algunas de las librerías de acceso abierto más populares son:

- Open AL: Popular por su gran tiempo en escena, se ha quedado atrás en cuanto a sus competidores. Aún así, podemos encontrar implementaciones software y open-source de la API como OpenAL-Soft que aplica funciones HRTF (usadas para audio binaural) consiguiendo buenos resultados.
- SoLoud: Mucho más moderna, simple y fácil de utilizar, por ello que no cuente con sistema más avanzado de procesamiento.

3.3. Herramientas y datos experimentales usados en el desarrollo y documentación del proyecto.

En esta sección se enumeran algunas herramientas utilizadas para el desarrollo de este proyecto.

3.3.1. Microsoft Visual Studio

Desarrollado por Microsoft, se trata de un editor de código fuente que se encuentra disponible para Windows y MacOS. Entre sus funciones, incluye control integrado con Git, autocompletado de código, resaltado de sintaxis y refactorización de código. Cuenta además con una amplia gama de herramientas para la depuración.

Para este proyecto usaremos la versión de VS2019, y trabajando sobre Windows 10. Esto nos da la ventaja de que en Windows 10 ya está instalado DirectX, un conjunto de librerías de gráficos desarrollado por Microsoft, donde está integrado xAudio2.

Por último, usaremos el estándar ISO C++14, usando el compilador integrado en el IDE, Visual C++ 14.2.

3.3.2. Audacity

Creado por Dominic Mazzoni y Roger Dannenberg (2000) y publicado como software libre en la plataforma colaborativa “SourceForge”. Es una aplicación que permite la grabación de audio en tiempo real, así como su edición, la observación de las ondas del sonido en parámetros determinados, entre otras muchas funcionalidades.

3.3.3. Matplotlib - python3

Se trata de una de las librerías más populares de python para crear gráficos y tablas. Se usará esta librería para analizar y representar las distintas muestras de audio que procesamos en el programa principal. Además de generar gráficas y diagramas para visualizar la información en la que vamos trabajando.

3.3.4. Filtros HRTF

Los filtros que usaremos para este proyecto han sido obtenidos de la siguiente página web [4]:

`https://sound.media.mit.edu/resources/KEMAR.html`

Se trata de un amplio conjunto de filtros HRTF recogidos de la medición en una cámara anecoica de un sistema de micrófonos de cabeza simulado de la marca KEMAR en 1994, por Bill Gardner y Keith Martin en el MIT Media Lab (ver figura 3.3).

El equipo de grabación usado es un conjunto de micrófonos colocados en los oídos de la cabeza "dummy". Gracias a este sistema, podemos medir cuál es el efecto que tiene un sonido sobre los dos oídos de manera independiente.

Figura 3.3: Bill Gardner y Keith Martin junto al equipo de grabación en una cámara anecoica. Copyright: MIT Media Lab

Los filtros están muestreados a 44.1 KHz, uno diferente para cada oído (izquierdo y derecho) y están disponibles un total de 710 posiciones diferentes, variando en elevación desde los -40º hasta 90º. Todos los filtros se han calculado a una distancia de 1.4m del sistema de grabación. Según la elevación hay un numero variable de mediciones. Se puede comprobar el espacio cubierto por las mediciones en la figura 3.4.

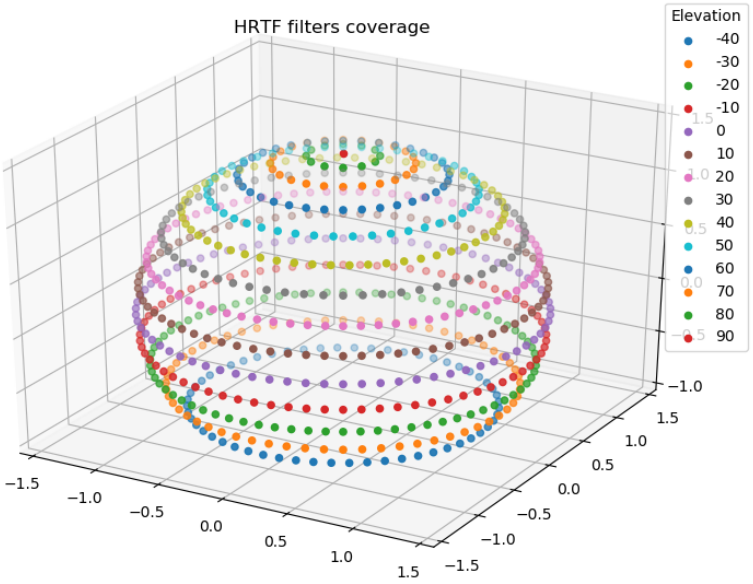


Figura 3.4: Mediciones de los filtros HRTF

Por otro lado también está disponible una versión compacta de estos filtros, ya aplicado el filtro inverso de los altavoces que se usaron para la grabación. Están disponibles en estéreo, por lo que el canal izquierdo corresponde al oído izquierdo y el derecho para el oído derecho.

Además de esto, sólo están disponibles hasta azimut de 180º para todas las elevaciones. Esto no es problema, ya que para simular un sonido proveniente de una fuente con azimut mayor, podemos elegir el filtro opuesto (simétrico respecto al eje x) al azimut que estamos buscando e intercambiando los canales izquierdo y derecho del filtro.

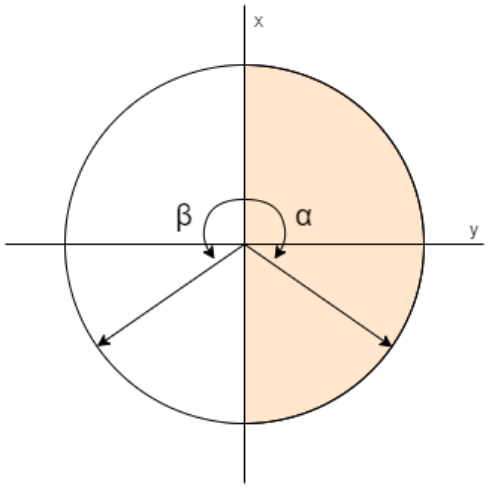


Figura 3.5: Selección de filtro, vista superior

Como vemos en la figura 3.5, si el oyente estuviera en la posición $(0, 0)$ sólo tenemos filtros que estén en el 1º y 4º cuadrante (azimut desde 0º a 180º). Pero como todos los filtros se han hecho a la misma distancia del micrófono, para obtener un filtro con azimut β tal que $\beta = -\alpha$ sólo basta con escoger el filtro correspondiente a un azimut de α e intercambiar el canal izquierdo con el canal derecho.

Este ’truco’ funciona bastante bien, pero tenemos que tener en cuenta que usar el filtro simétrico de esta manera se pierde la información de localización binaural en el plano medio (cerca del eje x).

3.3.5. 3dAudio, Simulación HRTF en python

Se trata de una aplicación de simulación hecha en python [5]. Se crea un escenario en 2D usando el mismo conjunto de filtros HRTF que usaremos aquí. Podemos usar este programa para comparar los resultado obtenidos en nuestro proyecto.

3.3.6. draw.io

Servicio web que ofrece soporte para crear diagramas de todo tipo. Se usará esta herramienta para realizar representaciones visuales de los conceptos tratados en este proyecto.

3.4. Lenguajes de programación

- C++: La librería de audio que vamos utilizar para este proyecto sólo está disponible en este lenguaje, aunque esto nos da la ventaja de procesar el audio en tiempo real al poder trabajar en bajo nivel. Usaremos el estándar ISO C++14.
- Python: Scripts auxiliares para comprobar que el código de las transformaciones es correcto.

3.5. Librería de audio

Se trata de xAudio2, una librería de bajo nivel y baja latencia de audio para Windows y consolas Microsoft. Está orientada a gestionar audio con un gran rendimiento para videojuegos. Soporta audio en 3D básico, múltiples fuentes de audio, múltiples canales, mezcla de audio y capacidad de procesamiento del audio en tiempo real. Además tiene soporte para crear procesamientos de audio personalizados.

3.5.1. Funcionamiento

La librería acepta diferentes formatos PCM como buffer de sonido de entrada:

- PCM de valores enteros 8 bits
- PCM de valores enteros 16 bits (óptimo para xAudio2)
- PCM de valores enteros 20 bits (en contenedor de 24 o 32 bits)
- PCM de valores enteros 24 bits (en contenedor de 24 o 32 bits)
- PCM de valores enteros 32 bits
- PCM de valores punto flotante de 32 bits

Para que esta librería empiece a trabajar, tenemos que inicializar el motor de sonido y darle cierta configuración. xAudio2 utiliza un modelo de 'voices' y grafos de procesamiento. En la figura 3.6 podemos ver el escenario mínimo, donde sólo tenemos una fuente de audio y un sólo dispositivo de salida.

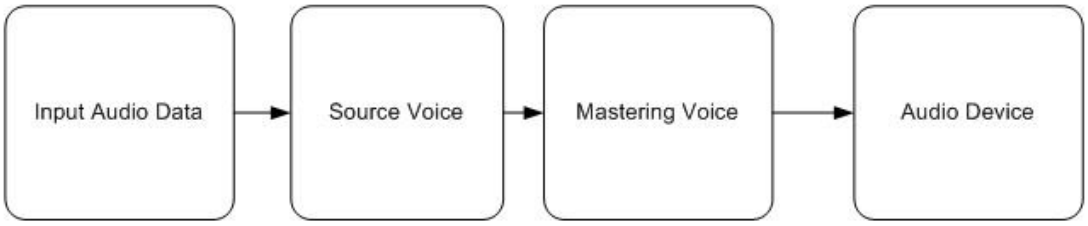


Figura 3.6: Estructura básica xAudio2 Autor: Michael Satran [1]

3.5.2. Modelo de voices

Las 'voices' son los nodos del grafo de procesamiento, donde hay tres tipos diferenciados:

- **Source Voice**: Las 'Source Voices' se encargan de operar sobre los datos de audio que le indiquemos. Normalmente abrimos un fichero de audio, conseguimos el buffer de audio y se lo entregamos a este tipo de voice.
- **Submix Voice**: Las 'Submix Voices' pueden recibir datos de otras 'Source Voices' y de otras 'Submix Voices', mezclando el audio de todas las voices que estén enlazadas y operan sobre estos datos. No se le puede enviar buffer de audio directamente y hay que enlazarla finalmente a una 'Mastering voice' para escuchar el resultado.
- **Mastering Voice**: las 'Mastering Voices' se encargan de escribir los datos de audio de todas las fuentes de las que reciba, en los dispositivos de audio del sistema. No se pueden enviar buffers de audio a una voice de este tipo tampoco. Toda señal que llegue a este punto será escuchada mediante los dispositivos de audio del sistema (altavoces).

3.5.3. Grafos de procesamiento

Para xAudio2 tenemos que definir un grafo de procesamiento, basándose en el uso de las voices, donde el audio va pasando de una voice a otra según su posición en el grafo. Cada nodo se encarga de una función distinta.

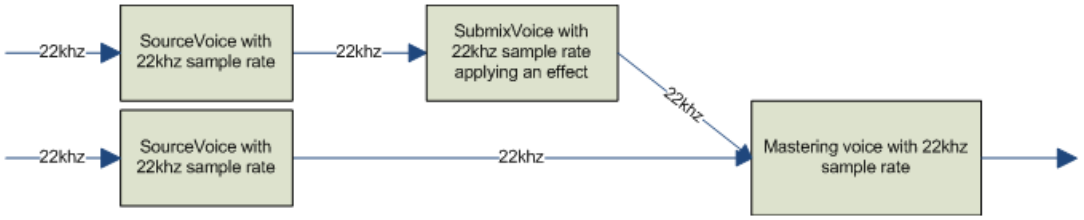


Figura 3.7: Ejemplo de grafo de procesamiento. Autor: Michael Satran [1]

Además de lo indicado, la librería convierte entre diferentes frecuencias de muestreo si es necesario. Por ejemplo si tenemos audio muestreado a 22KHz en una source voice y la enlazamos a una mastering voice que de entrada necesita una señal de 44.1Khz,

automáticamente se produce la conversión de frecuencia de muestreo (SRC, Sample Rate Conversion).

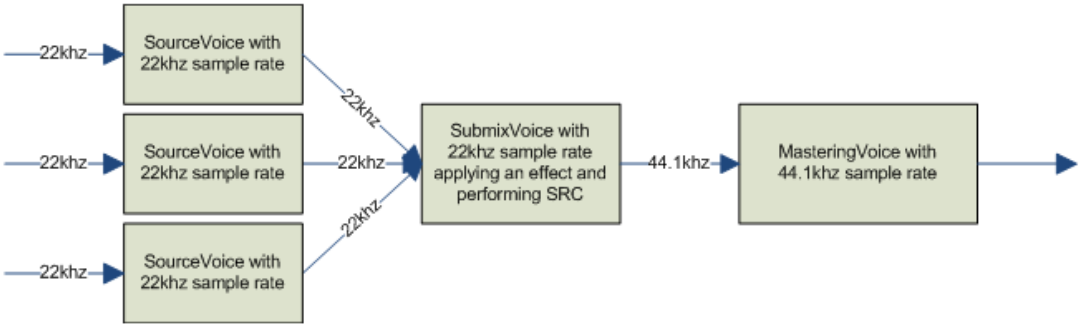


Figura 3.8: Ejemplo de conversión de frecuencia de muestreo. Autor: Michael Satran [1]

Por otro lado, las voices no pueden recibir señales en diferentes frecuencias de muestreo. Todas las señales de entrada a una voice deberán ser de la misma frecuencia de muestreo, como vemos en las figuras 3.8 y 3.9. Tenemos que tener en cuenta que sólo podemos definir la entrada de cada voice una vez la estamos instanciando.

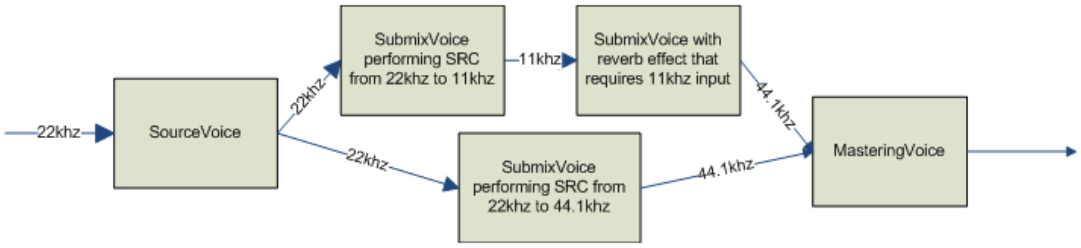


Figura 3.9: Ejemplo de SRC complejo. Autor: Michael Satran [1]

Para enlazar una voice con otra, se realiza en momento de instanciación de la voice.

Primero se crea la instancia de la Mastering Voice, y luego las demás en orden inverso.

En el caso que no indiquemos ninguna voice para enlazar mientras estamos creando la instancia de una Submix Voice o una Source Voice, automáticamente se enlazarán a la última Mastering Voice creada.

3.5.4. API de procesamiento xAPO

La librería permite que cualquier voice del grafo tenga una cadena de efectos asociada. Una vez ya tenemos un grafo de audio, con sus voices correctamente configuradas, podemos añadir efectos preexistentes en la librería como la reverberación o la ganancia, además de la posibilidad de crear efectos personalizados.

Para añadir efectos a una voice, tenemos que instanciar una estructura llamada `XAUDIO2_EFFECT_DESCRIPTOR` asociada a cada efecto que queramos aplicar y otra estructura de `XAUDIO2_EFFECT_CHAIN` indicando cuantos efectos vamos a aplicar y una lista ordenada de estos efectos, para que se vayan aplicando secuencialmente. La entrada del primer efecto es el audio que tenga la voice y cuando acaba de procesar, se lo pasa al segundo efecto, así hasta que ya no queden más efectos en la cadena.

Es posible habilitar y deshabilitar efectos específicos de la cadena, pero tenemos que tener en cuenta una norma: La salida de un efecto y la entrada del siguiente efecto en la cadena resultante deben coincidir. Si no cumplimos esta condición, no podremos apagar o encender los efectos arbitrariamente y simplemente el comando que usemos para esta acción fallará.

Además, un efecto puede cambiar el número de canales de la señal (por ejemplo, de mono a estéreo) pero no puede cambiar la frecuencia de muestreo.

Para empezar a definir nuestro propio efecto, el procedimiento que dicta Microsoft en la documentación de xAudio2 es implementar todos los métodos de la interfaz IXAPO y la interfaz IUnknown. Por otro lado tenemos a nuestra disposición la clase base CXAPOBase que proporciona una implementación básica de los métodos de la interfaces anteriormente nombradas. Podemos extender esta clase e implementar el método IXAPO::Process(). Este método es que el realizará el procesamiento en tiempo real, a discreción de la librería.

3.5.5. Implementación del filtro

Una vez xAudio2 llame al método `IXAPO::Process()` asociado a nuestra instancia de la clase xAPO, entregará por parámetros dos buffers distintos, entre otros parámetros, encapsulados junto a otros datos en la estructura `XAPO_PROCESS_BUFFER_PARAMETERS`.

```
1    STDMETHOD_(void, Process)(
2        UINT32 InputProcessParameterCount,
3        const XAPO_PROCESS_BUFFER_PARAMETERS* pInputProcessParameters,
4        UINT32 OutputProcessParameterCount,
5        XAPO_PROCESS_BUFFER_PARAMETERS* pOutputProcessParameters,
6        BOOL IsEnabled
7    );
```

Listing 3.1: Cabecera del método IXAPO::Process()

Ya que el funcionamiento es en tiempo real, la librería entrega fragmentos del buffer inicial de audio, según el parámetro `pInputProcessParameters[0].ValidFrameCount`, por lo que es conveniente guardar muestras hasta tener el mínimo necesario para empezar a procesar y luego seguir con el método overlap-add utilizando el algoritmo de FFT escogido y el conjunto de filtros para las operaciones de filtrado.

Una vez finalizado, el resultado se lo entregamos a la librería en el campo designado, como vemos en el fragmento de código 3.2.

```
1    void* pvSrc = pInputProcessParameters[0].pBuffer; // input: process this data
2    assert(pvSrc != NULL);
3
4    void* pvDst = pOutputProcessParameters[0].pBuffer; // output: put processed data here
5    assert(pvDst != NULL);
6
7    // Pass-through, copy from input to output
8    memcpy(pvDst, pvSrc, pInputProcessParameters[0].ValidFrameCount * m_uChannels *
    m_uBytesPerSample);
```

Listing 3.2: Entrada y salida dentro del método IXAPO::Process()

Capítulo 4

Desarrollo

4.1. Primera toma de contacto con el procesamiento de audio digital

Para entender mejor los conceptos de procesamiento de audio digital (DSP) y las funciones de transferencia relacionada con la cabeza (HRTF), comenzamos por crear un programa en C++, usando la librería xAudio2, que sea capaz de cargar archivos de sonido, aplicarle un filtro HRTF y reproducir o guardar en fichero el resultado. De esta manera podremos comprobar si podemos hacer uso de un filtro HRTF y luego poder implementarlo en el motor gráfico.

4.1.1. Proyecto de Visual Studio 2019

Antes de nada, necesitamos importar las librerías que vamos a usar. Por defecto en Windows 10, están incluidas las cabeceras xaudio2.h, xapo.h y xapobase.h, por lo que no tenemos que descargarlas de internet.

Primero creamos un proyecto de tipo Windows en Visual Studio 2019, ya sea de consola o con interfaz gráfica. Luego incluimos las librerias modificando la siguiente configuración: En Propiedades del proyecto, Enlazador, Entrada, añadimos 'xaudio2.lib y xapobase.lib' a dependencias adicionales, tal y como vemos en la figura 4.1.

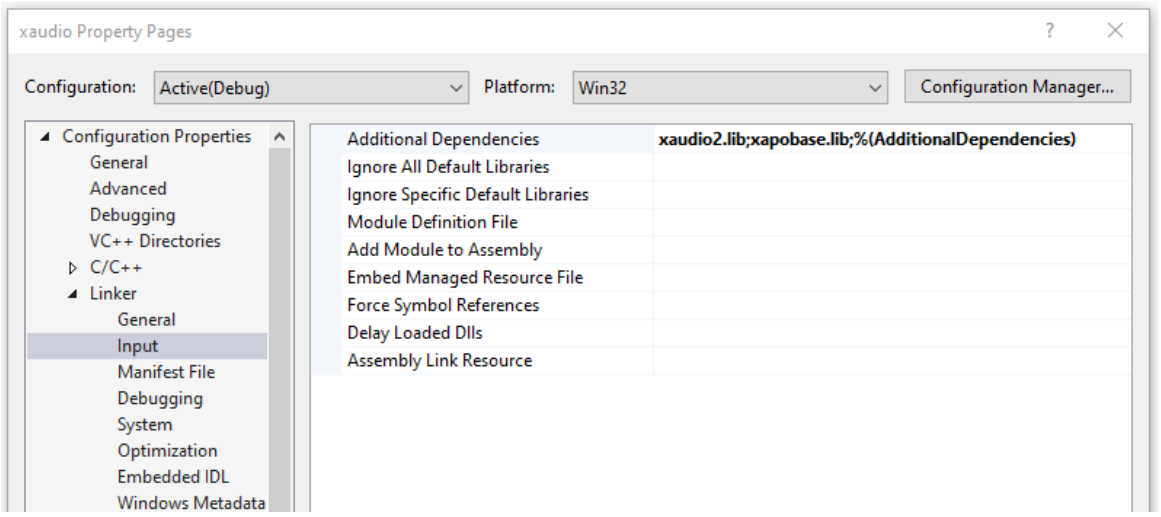


Figura 4.1: Configuración del proyecto VS2019

Con esto, ya tenemos enlazados xAudio2 y la API xAPO al proyecto.

4.1.2. Configuración de xAudio2

A continuación tenemos que inicializar el motor de sonido y le damos una configuración mínima:

1. Inicializamos la librería COM
2. Creamos una instancia de xAudio2
3. Creamos una instancia de Mastering Voice
4. Creamos una instancia de Submix Voice
5. Creamos una instancia de Source Voice
6. Enlazamos la Source Voice con la Submix Voice
7. Cargamos un fichero .wav y lo vinculamos al Source Voice
8. Reproducimos el sonido

Para empezar, muchos métodos y muchas funciones tienen un valor de retorno que se guarda en `HRESULT hr`, se trata de un valor de retorno para comprobación de errores. Podemos usar la función `FAILED()` para averiguar si hubo algún error. Se incluye esta comprobación en el primer fragmento de código, pero se omite en el resto, debido a que se aplica de manera análoga en todos los casos.

Antes de iniciar la librería de xAudio2, tenemos que iniciar la librería COM, que es usada por xAudio2. Para eso, tenemos que usar el método `CoInitializeEx()` y cuando hayamos terminado el trabajo, usamos `CoUninitialize()` para cerrar el acceso a la librería:

```
    HRESULT hr;

    hr = CoInitializeEx(NULL, COINIT_MULTITHREADED); // Required for xAudio2
    if (FAILED(hr)) {
        std::cout << "CoInitializeEx() failed" << std::endl;
        assert(0);
        return false;
    }
    [...]
    CoUninitialize(); // Close COM Library
```

Listing 4.1: Inicialización y Cierre de Librería COM

Para crear una instancia de xAudio2 basta con declarar un ’puntero inteligente’ y pasarlo por parámetros al método que crea la instancia:

```
    // Create a instance of xAudio2 Engine
    Microsoft::WRL::ComPtr<IXAudio2> pXAudio2 = NULL;
    hr = XAudio2Create(&pXAudio2, 0, XAUDIO2_DEFAULT_PROCESSOR);
```

Listing 4.2: Creación de instancia de xAudio2

Creamos la mastering voice y la submix voice. No hace falta indicar ningún enlace, ya que por defecto la librería enlaza la submix voice a la última mastering voice creada.

```
    // Create a mastering voice
    IXAudio2MasteringVoice* pMasterVoice = NULL;
    hr = pXAudio2->CreateMasteringVoice(&pMasterVoice);
    [...]
    // Create a Submix Voice
    IXAudio2SubmixVoice* pSFXSubmixVoice = NULL;
    hr = pXAudio2->CreateSubmixVoice(&pSFXSubmixVoice, 1, 44100, 0, 0);
```

Listing 4.3: Voices de xAudio2

Cargamos un fichero .wav (ver apéndice 2 para la función openWav):

```
    // Start importing .wav
    WAVEFORMATEXTENSIBLE wfx = { 0 };
    XAUDIO2_BUFFER buffer = { 0 };

    const std::string strFileName = "test_audio\\440hz.wav";
    openWav(strFileName.c_str(), wfx, buffer); // Output in wfx and buffer
```

Listing 4.4: Abrir archivo wav en C++ Windows

Ahora podemos crear la Source Voice usando el formato de onda que tenemos en la variable `wfx`, y luego le entregamos el buffer de audio con el método `SubmitSourceBuffer()`:

```
    // Populate structures for xaudio2, so it knows where to send data
    XAUDIO2_SEND_DESCRIPTOR voice;
    voice.Flags = 0;
    voice.pOutputVoice = pSFXSubmixVoice;

    XAUDIO2_VOICE_SENDS sendlist;
    sendlist.SendCount = 1;
    sendlist.pSends = &voice;

    // Create source voice
    IXAudio2SourceVoice* pSourceVoice;
    hr = pXAudio2->CreateSourceVoice(&pSourceVoice, (WAVEFORMATEX*)&wfx, 0, 2.0, NULL, &
    sendlist, NULL);
    [...]
    // Submit buffer to voice
    hr = pSourceVoice->SubmitSourceBuffer(&buffer);
```

Listing 4.5: Codificación del grafo de procesamiento

Y por último, reproducimos el sonido con el método `Start()`:

```
    // Play the sound (when ready)
    hr = pSourceVoice->Start(0);
```

Listing 4.6: Reproducir el buffer de sonido

4.2. Implementación de filtro personalizado

Ya que el objetivo es implementar un sistema de audio binaural que se usará para videojuegos, usaremos en la medida de lo posible PCM en punto flotante de 32 bits, ya que los procesadores modernos tienen buena velocidad de cómputo en punto flotante.

Una vez ya tenemos la estructura básica de xAudio2, podemos desarrollar la clase que se encargará de aplicar el filtro al audio de una voice determinada. Creamos una clase (myXapo), que hará uso de la interfaz xAPO de xAudio2, para aplicar filtros a la

señal. La interfaz nos indica que debemos poner el código de procesamiento en el método `myXapo::Process()`, y que este método se llamará periódicamente con los nuevos trozos de audio.

Podemos copiar el código de ejemplo que se ofrece en la documentación de xAudio2, para obtener una clase xAPO que no tenga efecto, que sólo copie el buffer de entrada en el buffer de salida, para usarlo como base y construir encima, el filtro (ver apéndice A.2):

https://docs.microsoft.com/en-us/windows/win32/xaudio2/how-to–create-an-xapo

Una vez tengamos la clase base, la incluimos en el proyecto. Para poder instanciarla y usarla en xAudio2, tenemos que darle configuración a la estructura `XAPO_REGISTRATION_PROPERTIES`:

```
// We create the necessary structures to load custom XAPOs
XAPO_REGISTRATION_PROPERTIES prop = {};
prop.Flags = XAPOBASE_DEFAULT_FLAG;
prop.MinInputBufferCount = 1;
prop.MaxInputBufferCount = 1;
prop.MaxOutputBufferCount = 1;
prop.MinOutputBufferCount = 1;

// Load custom Xapo instance
IUnknown* pXAPO = new myXapo(&prop);
```

Listing 4.7: Instanciación de la clase myXapo

Para aplicar el efecto a la Submix Voice, tenemos que pasar a esta instancia una estructura de datos `XAUDIO2_EFFECT_CHAIN`, que contendrá un puntero a un array de estructuras `XAUDIO2_EFFECT_DESCRIPTOR`, que a su vez describe cada efecto que vamos a aplicar. Ésta última estructura, contiene un puntero a la instancia de una clase que extienda la interfaz IUnknown (nuestro efecto), además de otros parámetros.

```
// Structure for custom xAPO
XAUDIO2_EFFECT_DESCRIPTOR descriptor;
descriptor.InitialState = true; // Efect applied from start
descriptor.OutputChannels = 1;  // TODO: Change to 2 for HRTF
descriptor.pEffect = pXAPO;     // Custom xAPO to apply

// Structure indicating how many effects we have (1)
XAUDIO2_EFFECT_CHAIN chain;
chain.EffectCount = 1;
chain.pEffectDescriptors = &descriptor;

// Apply effect chain to Submix Voice
hr = pSFXSubmixVoice->SetEffectChain(&chain);
```

Listing 4.8: Crear y aplicar cadena de efectos sobre una voice

Una vez puesta toda esta configuración, el filtro que vamos a desarrollar se aplicará en ejecución sobre el audio que esté administrando la voice.

Por otro lado, tenemos que tener en cuenta que cuando se ejecute el método `myXapo::Process()`, se estará ejecutando en el hilo dedicado al sonido, y es conveniente no realizar ninguna operación que no tengamos certeza de que va a acabar a tiempo, ya que el audio se escucharía con cortes o 'ticks' muy molestos. Por lo tanto no podremos reservar memoria, hacer peticiones web, ni escribir en disco, por ejemplo.

4.2.1. FFT

Vamos a usar el algoritmo de Cooley–Tukey que computa recursivamente la FFT usando el método divide y vencerás, siendo muy intuitiva su implementación:

`https://rosettacode.org/wiki/Fast_Fourier_transform`

En esta implementación se usa `std::complex<double>` para definir los valores individuales y `std::valarray` para definir el conjunto de datos (ver listing 4.9). Por otro lado se usa `std::slice()` para separar los valores en posiciones pares e impares, de tal manera que se ordenan las muestras según el algoritmo de bit-reversal [6].

Además de esto, crearemos dos estructuras para representar los buffers, `data_buffer` y `stereo_data_buffer`.

La estructura `data_buffer` se compone de un puntero a `CArray` y un entero sin signo para designar el espacio utilizado. Con esto se consigue reservar memoria en un inicio y mientras ejecutamos en el hilo de audio, sólo modificamos los valores del CArray y su tamaño útil, sin cambiar el espacio de memoria reservado. Esto significa que podemos trabajar con los CArray en el hilo de sonido, sin bloquearlo.

```
typedef std::complex<double> Complex;
typedef std::valarray<Complex> CArray;

struct data_buffer {
    CArray* pdata;
    unsigned size;
};

struct stereo_data_buffer {
    data_buffer left;
    data_buffer right;
};
```

Listing 4.9: Tipos de datos usados para trabajar los buffers de audio

Ya que esta implementación de FFT requiere que la entrada sea una secuencia de valores complejos, tenemos que transformar la señal de entrada que está representada en valores punto flotante, a las estructuras de datos mencionadas anteriormente. Además rellenamos con 0's hasta un límite, por lo que podemos dejar preparados los arrays para las siguientes operaciones.

```
// Prepare data for FFT, takes data from p and outputs in arr.
// Fills with 0's to match nfft size
inline void transformData(CArray& arr, const float* p, const uint32_t size, const
    uint32_t nfft = 512) {
    if (size > nfft) {
        assert(0);
    }
    arr.resize(nfft);
    // Copy data from pointer
    for (int i = 0; i < size; i += 1) {
        arr[i] = Complex(p[i], 0);
    }
    // Fill with zeros
    for (int i = size; i < nfft; i += 1) {
        arr[i] = Complex(0, 0);
    }
}
```

Listing 4.10: transformData()

En la figura 4.2 podemos ver la onda de sonido que procesa el programa representada en una gráfica tiempo-frecuencia y en la figura 4.3 podemos ver el resultado de aplicar la FFT sobre esta onda. Lo que vemos en esta última figura representado en el eje de abscisas es una frecuencia particular y el eje de ordenadas es la magnitud de cada número complejo, indicando la importancia o 'cuanto' de una frecuencia en particular hay en la señal original.

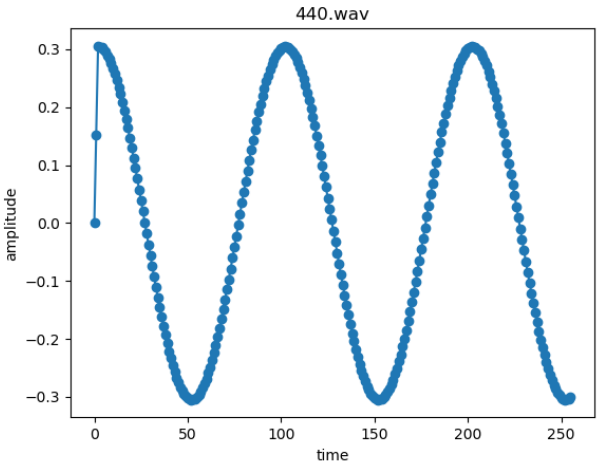


Figura 4.2: Onda sinusoidal de 440Hz.

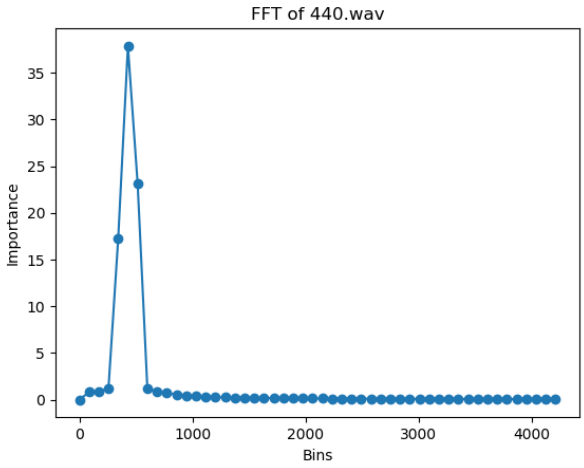


Figura 4.3: FFT sobre la onda de 440Hz.

Para este caso no nos ha hecho falta aplicar una ventana sobre la señal de entrada, ya que este fragmento si que es un periodo de una señal periódica infinita, por lo que el resultado es correcto.

4.2.2. Overlap-add

Antes de empezar es conveniente tener los filtros FIR en el dominio de la frecuencia. Los cargamos de la misma manera que hemos abierto un fichero .wav, separamos los canales, aplicamos la FFT a cada canal y guardamos la información en una estructura `filter_data`. (ver apendices A.1 y A.2)

Para realizar el método Overlap-add empezamos definiendo el tamaño del bloque en el que vamos a trabajar. Necesitamos cumplir la siguiente ecuación:

$$N = L + M - 1$$

Siendo N el tamaño de los datos de entrada para la FFT que vamos a realizar, M es el tamaño del filtro FIR (respuesta al impulso) y L es un valor arbitrario, que representa la longitud de datos de la señal de entrada que seleccionamos para cada bloque. El objetivo es que N sea potencia de 2, para maximizar la eficiencia. [6]

Una manera de aproximar que valores escoger para esta ecuación es usando el tamaño del filtro:

$$N = 8 \cdot 2^{ceiling(log2(M))}$$

De esta manera nos da un valor aproximadamente 8 veces mayor que el tamaño del filtro. Inicialmente se consideran los filtros de $M = 512$ muestras, pero esto daría lugar a computar FFTs de $N = 4096$, por lo que es muy complicado ejecutarse en tiempo real. Por este motivo usaremos la variante compacta de los filtros, que son de $M = 128$, resultando en FFTs de $N = 1024$, siendo mucho más factible de realizar.

Entonces ya sabemos todos los valores de la ecuación:

$$\begin{aligned} N &= L + M - 1 \\ N &= 1024 \\ M &= 128 \\ L &= N - (M - 1) \\ L &= 897 \end{aligned}$$

Para aplicar la ventana usaremos una Hann Window, aplicada sobre toda la señal de entrada. Recordemos que es un paso necesario siempre y cuando la señal que vamos a filtrar no sea un fragmento de una señal periódica.

La librería xAudio2 ofrece un número de muestras determinado en cada llamada de la función IXAPO::Process(). Nuestro trabajo es ir guardando muestras hasta que tengamos al menos L, procesarlas y luego enviarlas. Esto añade un poco de latencia al procesamiento del sonido, pero es la única manera ya que xAudio2 administra el workflow.

Para realizar el overlap-add, vamos a seguir el siguiente diagrama: (ver figura 4.4, Apéndice A.2)

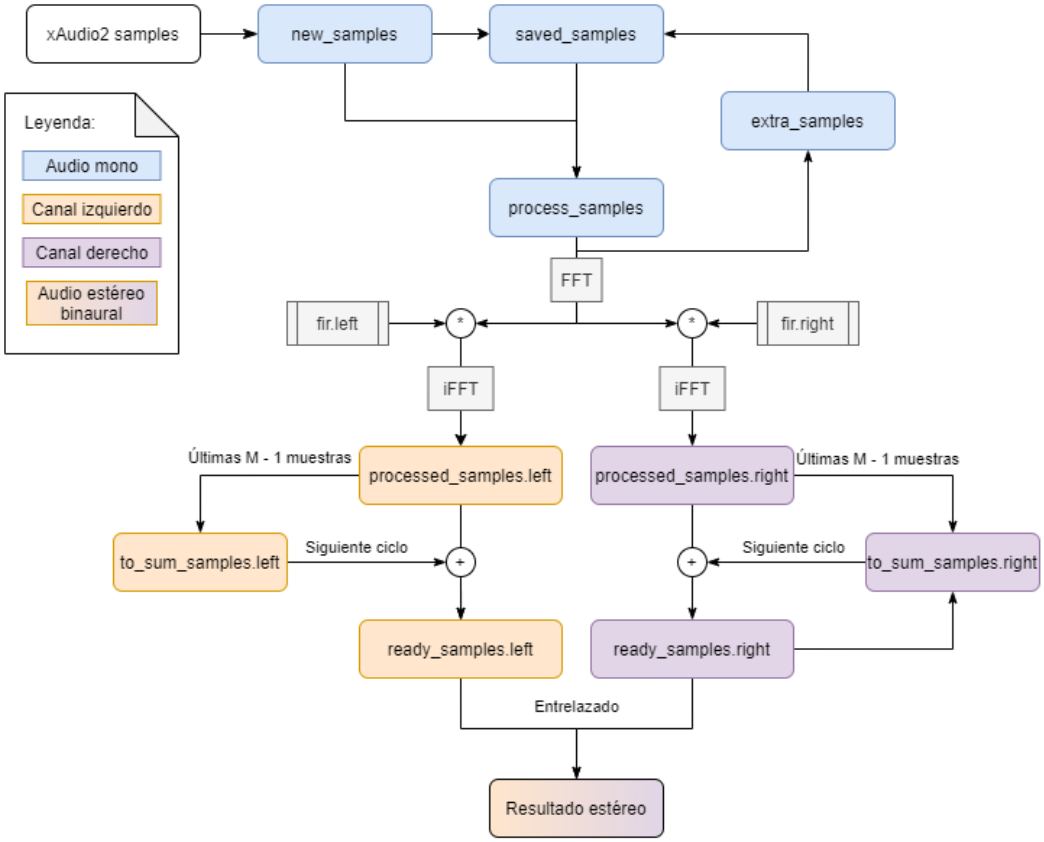


Figura 4.4: Diagrama overlap-add

Siguiendo el diagrama, los nodos azules son estructuras `data_buffer` y los nodos naranjas y violetas son `stereo_data_buffer`. Podemos ver que las muestras que van llegando las dejamos en la variable `new_samples`, para ir traspasándolas a `saved_samples`. Una vez tengamos suficientes muestras guardadas en esta última variable, podemos preparar el procesamiento. Llevamos L muestras a `process_samples` y las sobrantes las devolvemos a `saved_samples`.

Una vez ya tenemos suficientes muestras de la señal original, tenemos que preparar el

bloque de tamaño N que será la entrada a la FFT. Para esto rellenamos con $M-1$ ceros al final de la secuencia, hasta tener $N = L + M - 1$ valores de entrada.

Realizamos la FFT y nos devuelve el dominio de la frecuencia en la variable `process_samples`. En este paso, aplicamos los filtros mediante la multiplicación en el dominio de la frecuencia. Como hemos explicado anteriormente, esta operación da como resultado la convolución en el dominio del tiempo.

A continuación, aplicamos la FFT inversa para devolver al dominio del tiempo y sumamos los últimos $M-1$ valores de la secuencia del primer segmento de datos con los primeros $M-1$ valores del segundo segmento. En el caso de tratarse de la primera vez que procesamos la señal, no tendremos valores guardados para sumar a la secuencia, descartamos los primeros $M-1$ valores de `processed_samples` (de los dos canales).

Finalmente todas las muestras que ya hayan sido procesadas y que no necesitemos para futuras iteraciones, las podemos entregar a xAudio2 para que las envíe a los dispositivos del sistema.

Como ahora se trata de una señal en estéreo (2 canales), tenemos que ponerlos entrelazados, quedando primero la primera muestra del canal izquierdo, seguida de la primera muestra del canal derecho, continuando con la segunda muestra del canal izquierdo... y así hasta terminar la secuencia.

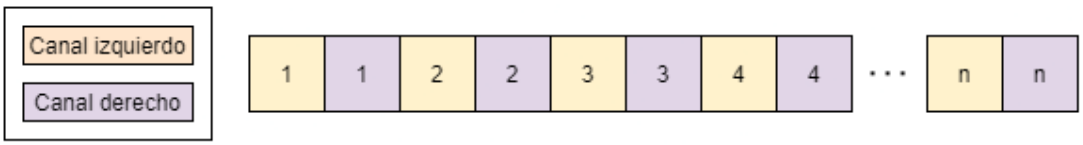


Figura 4.5: Flujo de datos en estéreo

4.3. Funcionamiento correcto de los filtros HRTF

La primera idea es guardar en un fichero el resultado e inspeccionar como cambia la onda, y reproducirlo. Junto al programa en python 3daudio, podemos comparar el resultado y corregir los errores que existan.

Ya que tenemos el método overlap-add implementado, podemos escoger manualmente alguno de los filtros que tenemos disponibles. Lo primero que se va a comprobar es que un sonido pueda identificarse entre izquierda y derecha, para luego comprobar de cuanto grado de precisión dispone el sistema.

Para empezar, escogemos un filtro con azimut de 90º y una elevación de 0º. Con esto esperamos que el sonido se transforme de manera que se escuche como si proviniera de la derecha del oyente. Si guardamos las muestras que tenemos en `ready_samples` podemos ver el resultado gráficamente (ver figura 4.6). Podemos observar que ahora hay dos canales en la onda, y además de eso, el canal derecho está ligeramente desfasado en el tiempo y ligeramente atenuado en comparación con el otro canal. Esto genera el efecto en nuestra percepción comentado anteriormente.

Por otro lado, si escogemos el filtro opuesto, uno con azimut de -90º (o 270º) y misma elevación, podemos ver que el comportamiento es el opuesto, la señal del canal izquierdo está atenuada y desfasada esta vez (ver figura 4.7).

Si intentamos ver la diferencia entre los filtros de elevación 0 y variamos el azimuth entre 0º y 180º (de frente y atrás, respectivamente) no se observa ninguna diferencia entre ellos sobre la onda de 440Hz. Podemos ver en las figuras 4.8 y 4.9 que sólo aparece el

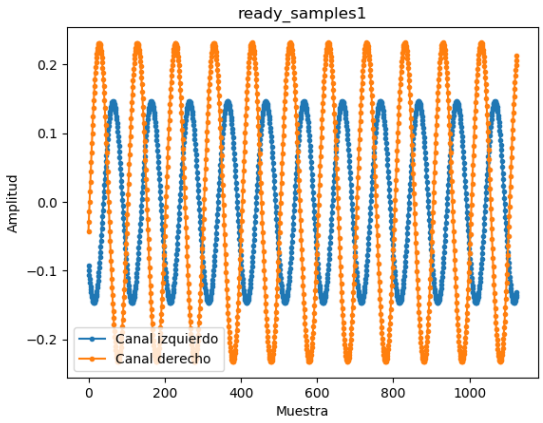


Figura 4.6: Filtro de azimut 90º y elevación 0º sobre la onda de 440Hz.

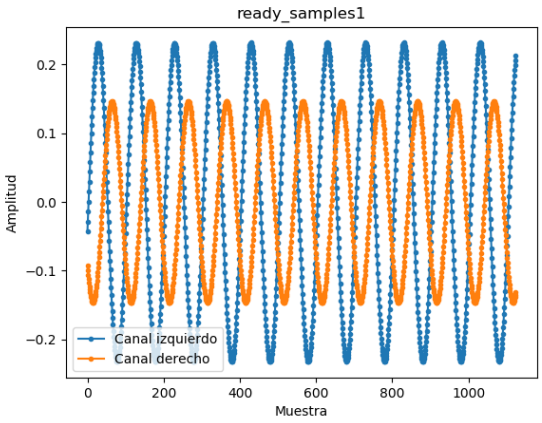


Figura 4.7: Filtro de azimut -90º y elevación 0º sobre la onda de 440Hz.

canal derecho, pero esto se debe a que el canal derecho y el izquierdo son esencialmente idénticos.

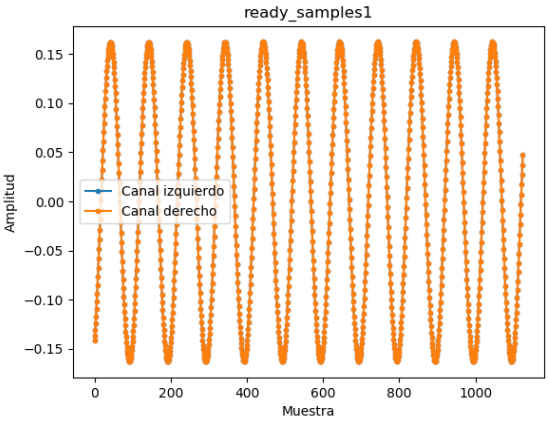


Figura 4.8: Filtro de azimut 0º y elevación 0º sobre la onda de 440Hz.

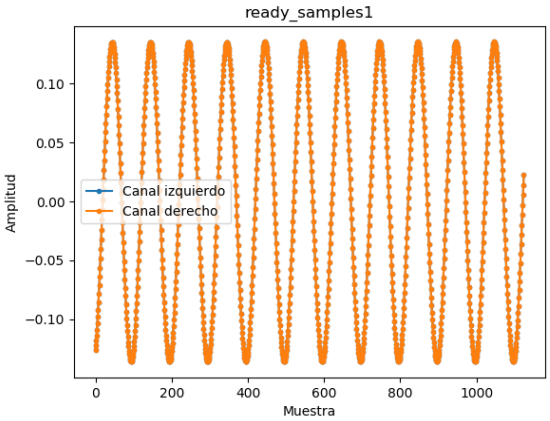


Figura 4.9: Filtro de azimut 180º y elevación 0º sobre la onda de 440Hz.

A continuación, si dejamos fijo el azimut a 90º y variamos la elevación sobre los valores -40º, 0º y 80º, podemos comprobar que entre los valores de elevación -40º y 0º hay diferencias sutiles, como que en -40º los dos canales está ligeramente más atenuados y un poco menos de desfase.

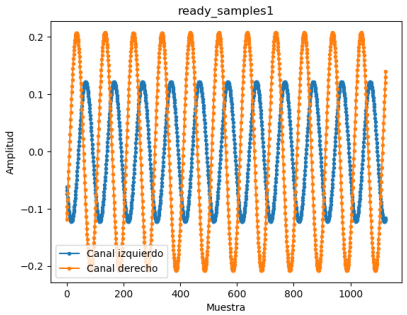


Figura 4.10: Filtro de azimut 90º y elevación -40º sobre la onda de 440Hz.

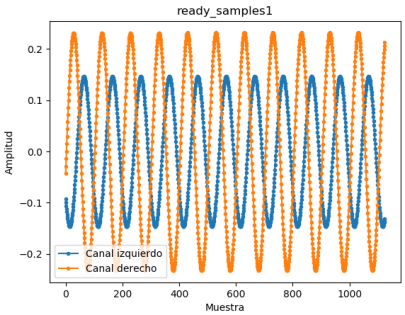


Figura 4.11: Filtro de azimut 90º y elevación 0º sobre la onda de 440Hz.

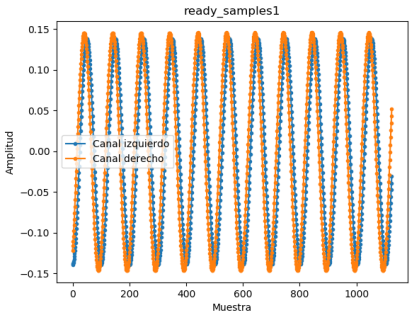
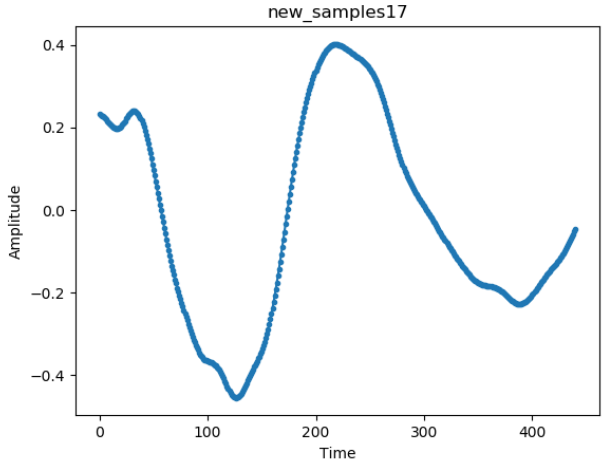


Figura 4.12: Filtro de azimut 90º y elevación 80º sobre la onda de 440Hz.

Por otro lado el filtro con elevación de 80º sólo tiene un pequeño desfase en el canal izquierdo en comparación con la señal original. (ver figuras 4.10, 4.11 y 4.12)

Por último si cambiamos de audio a otro que tenga armónicos, como el sonido de una guitarra, tendría que atenuar más unas frecuencias que otras, además del desfase que ya hemos visto. En la figura 4.13 podemos ver un fragmento de la onda del sonido de la guitarra en mono, antes de pasar por el programa y en la figura 4.14 podemos ver el resultado de la FFT sobre esa porción de la señal.



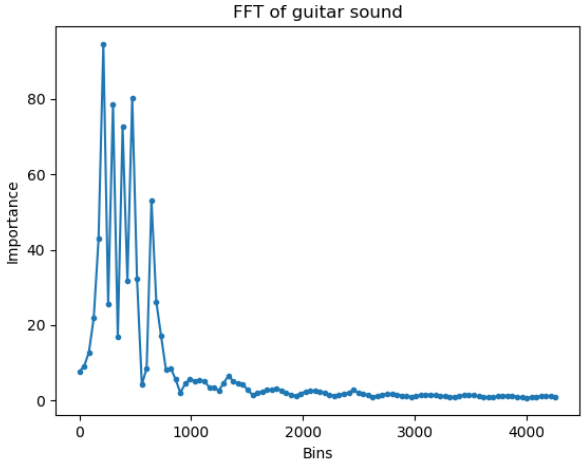
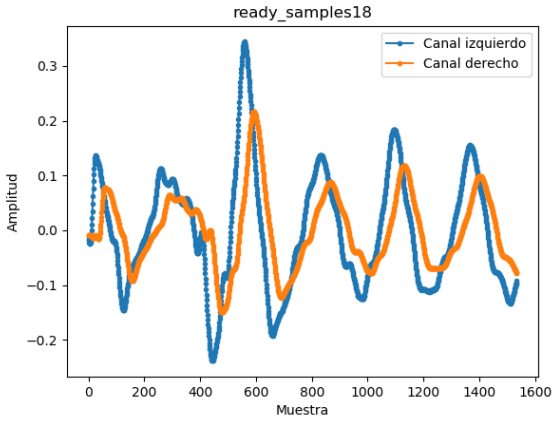


Figura 4.13: Muestras del sonido de una guitarra.

Figura 4.14: FFT del la onda de la guitarra anterior

Luego si le aplicamos los filtros, tenemos el resultado en las figuras 4.15 y 4.16. Se puede apreciar el mismo comportamiento que en la onda sinusoidal de 440Hz, con la diferencia de que parece que algunos picos son menos pronunciados, sin tener en cuenta la atenuación que sufre el canal. Podemos ver como en la región de 250-350 muestras (eje x) los picos en el canal antenuado son más planos que si sólo hubiera atenuación.



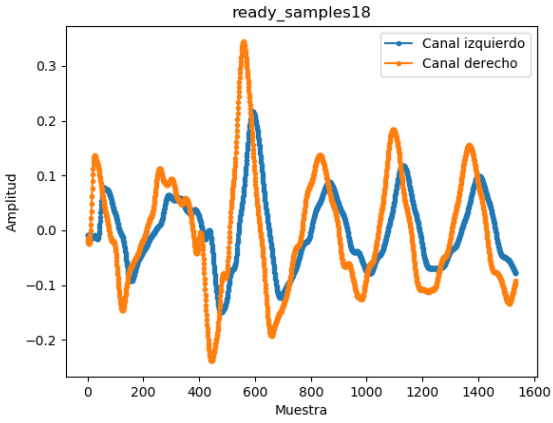


Figura 4.15: Filtro de azimut 90º y elevación 0º sobre la onda de la guitarra.

Figura 4.16: Filtro de azimut -90º y elevación 0º sobre la onda de la guitarra.

Como vemos, el funcionamiento de los filtros es correcto en unas pruebas y ligeramente incorrecto en otras. Por un lado podemos ver como hay diferencias notables entre la izquierda y derecha, pero no entre delante y atrás. Esto se debe a que usamos filtros derivados de un mismo oído.

Teniendo esto en cuenta, escuchando las muestras de audio resultante, es posible distinguir en cierta medida la dirección en la que sería la fuente del sonido.

4.4. Sistema de audio binaural

Con el correcto funcionamiento de la operación de filtrado, ya podemos empezar a desarrollar un sistema que permita cambiar de filtro según la posición de la fuente de audio que queremos simular en el espacio 3D. Ya que tenemos una amplia variedad de respuestas al impulso, podemos ir cambiando la que aplicamos en un momento determinado en momento de ejecución y que modifique el sonido percibido por el usuario y que parezca que el sonido proviene de la fuente de audio, ya sea en movimiento o estática.

Este sistema tiene como entrada un sonido y una posición relativa al oyente. Con estos dos parámetros, podemos simular la direccionalidad de una fuente de sonido. Para poder construir este sistema, tendremos que poder modificar el filtro que estamos aplicando en cada momento, por lo que debemos de poder cambiarlo después de haberla instanciado.

Con el método `IXAPO::SetParameters()` podemos pasarle parámetros adicionales a la xAPO para cambiar su comportamiento en tiempo de ejecución, efectivamente consiguiendo el cambio de filtro.

Para poder usar ese método, tenemos que extender nuestra clase xAPO con otra clase base más, `CXAPOParametersBase`, y además poner en el contructor de myXapo una inicialización al contructor de esta clase con unos parámetros definidos. Después de esto, tenemos que usar el método `CXAPOParametersBase::BeginProcess()` que nos dará acceso a la última configuración que se ha enviado con `IXAPO::SetParameters()`. Por otro lado hay que devolver el control a xAudio2 con `CXAPOParametersBase::EndProcess()`. Esto es necesario para no entrar en condición de carrera con estos datos compartidos entre el hilo del programa y el hilo del sonido.

```
// Nueva cabecera, ahora con BYTE* params y UINT32 params_size, para inicializar el
    constructor
myXapo(XAPO_REGISTRATION_PROPERTIES* pRegistrationProperties, BYTE* params, UINT32
    params_size, filter_data* filters, size_t filters_size, size_t signal_size);

[...]
// al principio de myXAPO::Process()
BYTE* param = BeginProcess();
[...]
// Usamos param para modificar el comportamiento...
[...]
// al final de myXAPO::Process()
EndProcess();
```

Una vez tengamos la clase preparada, podemos desde el programa principal, usar el método `IXAPO::SetParameters()` para pasar información a la clase xAPO.

```
void setPosition(IXAPOParameters* effect, spherical_coordinates position) {
    effect->SetParameters(&position, sizeof(position));
}
```

Listing 4.11: Ejemplo de uso de SetParameters()

Por otro lado también afecta la distancia a la que está la fuente de sonido. Los filtros se grabaron a una distancia específica y se indica que aproximadamente el sonido tardó unas 180 muestras en viajar esa distancia de 1.4m. Por esto podemos deducir que podemos aplicar un desfase a los dos canales de 128.57 muestras por cada metro de distancia de la fuente al oyente.

Además de lo anterior, debemos aplicar un efecto de ganancia si la fuente se empieza

a alejar, para simular la atenuación por distancia. Si se considera que cada vez que duplicamos la distancia la señal se atenúa 6dB, podemos realizar ciertos cálculos para darle más realismo al sistema.

Por otro lado, hay varios factores a tener en cuenta, ya que la distancia no es la única que influye en como se escucha el sonido, sino también el medio en el que se transmite. Consideraremos que el sonido se transmite por el vacío, aunque no sea un concepto real, para simplificar los cálculos.

Ya que tenemos el sistema listo para transformar el audio, sólo nos queda codificar un sistema que convierta coordenadas de un espacio tridimensional en coordenadas esféricas para poder seleccionar los filtros correctamente. Basta con un par de funciones que hagan los cálculos (ver fragmento de código 4.11)

```
struct cartesian_coordinates3d {
  double x;
  double y;
  double z;
};

struct spherical_coordinates {
  double radius;
  double azimuth;
  double elevation;
};

// Gets spherical coords in degrees to cartesian 3D
inline cartesian_coordinates3d cartesian3d_to_spherical(spherical_coordinates input) {
  return cartesian_coordinates3d {
    input.radius * cos(input.azimuth) * cos(input.elevation), // x
    input.radius * sin(input.azimuth) * cos(input.elevation), // y
    input.radius * sin(input.elevation),                      // z
  };
}

// Gets cartesian coords in degrees to spherical in radians
inline spherical_coordinates spherical_to_cartesian(cartesian_coordinates3d input) {
  return spherical_coordinates {
    sqrt(pow(input.x, 2) + pow(input.y, 2) + pow(input.z, 2)),  // radius
    atan2(input.y, input.x),                                    // azimuth
    atan(sqrt(pow(input.x, 2) + pow(input.y, 2)) / input.z),    // elevation
  };
}
```

Listing 4.12: Funciones que convierten de coordenadas cartesianas en 3D a coordenadas esféricas.

Por último, debemos seleccionar dentro de la xAPO el mejor filtro para las coordenadas que tenemos. Para esto, necesitamos las coordenadas esféricas, y buscamos entre los filtros el que más cercano sea a la posición que estamos buscando. Comenzamos por buscar la elevación, ya que según este parámetro habrá más o menos azimut disponibles para escoger. Una vez sabemos la elevación, escogemos el azimut más cercano, e indicamos que filtro es el resultante desde dentro de la clase(ver apéndice A.2).

Capítulo 5

Conclusiones y líneas futuras

El sistema de audio funciona, pero tiene algunos problemas técnicos, relacionados con el procesamiento del audio digital en tiempo real, ya que tiene un grado de dificultad acentuado. Este tipo de procesamiento además conlleva un coste computacional moderado, no siendo adecuado para cualquier dispositivo que use sonido.

Por otro lado, al hacer uso de la librería de audio xAudio2, es posible extrapolar muchos componentes del desarrollo, y poder ejecutar el sistema de sonido en otras plataformas de Microsoft como las consolas Xbox, que si cuentan con hardware dedicado para estas tareas.

Hemos probado los filtros HRTF y son adecuados, producen un cierto efecto de direccionalidad sobre la fuente de sonido. Por desgracia no generan audio binaural de una misma profundidad que el grabado con un equipo especial, pero si que puede ser un buen punto de partida para mejorar este tipo de tecnología. Sin olvidarnos que una de las mejores ventajas es que sólo necesita de un poco de capacidad de procesamiento y unos auriculares estéreo para realizar el efecto.

Para mejorar esta tecnología, se pueden aplicar distintas técnicas sobre el sistema de audio binaural, como por ejemplo interpolar los filtros HRTF disponibles, para obtener más puntos disponibles, y así aumentar ligeramente el nivel de detalle, al poder escoger virtualmente cualquier posición, aunque no haya filtro grabado.

En conclusión, este proyecto indica que para el usuario final, que estará disfrutando del contenido multimedia, sólo notará que con sus dispositivos de sonido actuales podrá hacer uso de esta tecnología del sonido, y no necesitará actualizarlos.

De cara al futuro, un sistema como este podría ser implementado en un motor de videojuegos, otorgando una poderosa herramienta para el desarrollo de nuevas mecánicas haciendo uso de la habilidad de localizar las fuentes de sonido, o simplemente para el aumento del nivel de detalle en los títulos con aspiraciones a ser más fieles a la realidad.

Como opción más novedosa, es recomendable probar un sistema como este con las tecnologías de realidad virtual (VR) y aprovechar que el jugador encarna físicamente a un personaje en primera persona en un mundo tridimensional, para conseguir una experiencia realmente inmersiva.

Es posible usar este sistema para otros tipos de contenido multimedia como el cine y simulaciones realistas en primera persona entre otros. Aunque el mayor beneficio de aplicar una técnica como ésta es que se puede ejecutar en tiempo real, bajo demanda. Para el cine, el sonido se puede computar en la etapa de producción y por lo tanto, consiguiendo mejores resultados, usando métodos de procesamiento más exactos y complejos.

Capítulo 6

Summary and Conclusions

The audio system works, but it has some technical problems, related to the digital audio processing in real time, since it has a steep degree of difficulty. This type of processing also carries a moderate computational cost, not being suitable for every device that use sound.

On the other hand, by making use of the xAudio2 audio library, it is possible to extrapolate many components of the development, and to be able to run the sound system on other Microsoft platforms such as Xbox consoles, which do have dedicated hardware for these tasks.

We have tested the HRTF filters and they are suitable, they produce a certain directionality effect on the sound source. Unfortunately, it does not generate binaural audio of the same depth as recorded with special equipment, but it can be a good starting point to improve this type of technology. Without forgetting that one of the best advantages is that you only need a little processing power and a pair of stereo headphones to perform the effect.

To improve this technology, different techniques can be applied on the binaural audio system, such as interpolating the available HRTF filters, to obtain more available points, and thus slightly increase the level of detail, by being able to choose virtually any position, although no filter is recorded in that position.

In conclusion, this project indicates that for the end user, who will enjoy multimedia content, they will only notice that with their current sound devices they will be able to make use of this sound technology, and will not need to update them.

Looking ahead, a system like this could be implemented in a video game engine, providing a powerful tool for the development of new mechanics, making use of the ability to locate sound sources, or simply to increase the level of detail in titles with aspirations to be more faithful to reality.

As a newer option, it is advisable to test a system like this with virtual reality (VR) technologies and take advantage of the fact that the player physically embodies a first person character in a three-dimensional world, to achieve a truly immersive experience.

It is possible to use this system for other types of multimedia content such as cinema and realistic first person simulations among others. Although the greatest benefit of applying a technique like this is that it can be executed in real time, on demand. For cinema, sound can be computed at the production stage and therefore achieving better results, using more accurate and complex processing methods.

Capítulo 7

Presupuesto

Para este proyecto se ha investigado durante unas 200 horas y se ha desarrollado durante unas 100 horas la librería, siendo el precio de investigación a 10€ por hora y el precio del desarrollo a 20€ por hora.

7.1. Sección Uno

Tipos	**Tiempo**	**Valor**
Investigación desarrollo	150 horas	1500 €
Desarrollo	150 horas	3000 €
Total	300 horas	4500 €

Tabla 7.1: Resumen de tipos

Apéndice A

Código

A.1. Función para cargar los filtros

```
/********************************************************************************
 *
 * Fichero main.cpp
 *
 ********************************************************************************/
 // Function that loads all HRTF filters in the original folder format
int loadFilters(filter_data* filters, size_t* size) {

    const array<int, 14> elevations =
        {-40, -30, -20, -10, 0, 10, 20, 30, 40, 50, 60, 70, 80, 90};
    const array<double, 14> azimuth_inc =
        {6.43, 6.00, 5.00, 5.00, 5.00, 5.00, 5.00, 6.00, 6.43, 8.00, 10.00, 15.00, 30.00,
     91};

    std::string prevPath = "filters";
    std::string name = "elev";
    std::string separator = "\\";
    std::string start = "H";

    unsigned index = 0;

    for (unsigned i = 0; i < elevations.size(); i++) {
        std::string elevation = to_string(elevations[i]);
        std::string folder = name + elevation;

        unsigned j = 0;
        // Load all azimuths for that elevation folder
        while((j + 1) * azimuth_inc[i] <= 180) {

            unsigned azimuth = round(azimuth_inc[i] * j);
            // Build file path
            std::stringstream file;
            file << elevation << "e";
            file << std::setfill('0') << std::setw(3) <<(unsigned) azimuth << "a.wav";

            std::string path =
                prevPath + separator + folder + separator + start + file.str();
            // Start importing .wav
            WAVEFORMATEXTENSIBLE *wfx = new WAVEFORMATEXTENSIBLE({ 0 });
            XAUDIO2_BUFFER *buffer = new XAUDIO2_BUFFER({ 0 });
```

```
            HRESULT hr = openWav(path.c_str(), *wfx, *buffer);
            if (FAILED(hr)) {
                std::cout << "Couldn't open " << path << "." << std::endl;
                return hr;
            }
            // Fill data in the structure for later
            filters[index] = filter_data {
                raw_data {
                    wfx,
                    buffer,
                },
                fir {
                    NULL,       // Later populate with FFT(buffer)
                    NULL,       // Later populate with FFT(buffer)
                },
                elevations[i],  // save elevation
                azimuth,        // save azimuth
            };
            index++;
            j++;

        }

    }
    *size = index - 1;
    return 0;
}
```

A.2. Clase xAPO

```
/*************************************************************************************
 *
 * Fichero myXapo.h
 *
 *************************************************************************************/
class myXapo : public CXAPOBase {

private:
    WORD m_uChannels;
    WORD m_uBytesPerSample;
    UINT32 max_frame_count;

    size_t fft_n;
    size_t step_size;
    size_t fir_size;

    data_buffer new_samples;
    data_buffer saved_samples;
    data_buffer process_samples;

    stereo_data_buffer processed_samples;
    stereo_data_buffer ready_samples;
    stereo_data_buffer to_sum_samples;

    filter_data* hrtf_database;
```

```
    size_t n_filters;

    void changeFilter(spherical_coordinates input);
    unsigned getSampleGap(spherical_coordinates input);
    double getGain(spherical_coordinates input);

public:

    myXapo(XAPO_REGISTRATION_PROPERTIES* pRegistrationProperties, BYTE* params, UINT32
    params_size, filter_data* filters, size_t filters_size, size_t signal_size);
    ~myXapo();

    STDMETHOD(LockForProcess) (UINT32 InputLockedParameterCount,
        const XAPO_LOCKFORPROCESS_BUFFER_PARAMETERS* pInputLockedParameters,
        UINT32 OutputLockedParameterCount,
        const XAPO_LOCKFORPROCESS_BUFFER_PARAMETERS* pOutputLockedParameters)
    {
        //assert(!IsLocked());
        assert(InputLockedParameterCount == 1);
        assert(OutputLockedParameterCount == 1);
        assert(pInputLockedParameters != NULL);
        assert(pOutputLockedParameters != NULL);
        assert(pInputLockedParameters[0].pFormat != NULL);
        assert(pOutputLockedParameters[0].pFormat != NULL);

        m_uChannels = pInputLockedParameters[0].pFormat->nChannels;
        m_uBytesPerSample = (pInputLockedParameters[0].pFormat->wBitsPerSample >> 3);
        max_frame_count = pOutputLockedParameters->MaxFrameCount;

        return CXAPOBase::LockForProcess(
            InputLockedParameterCount,
            pInputLockedParameters,
            OutputLockedParameterCount,
            pOutputLockedParameters);
    }
    STDMETHOD_(void, Process)(UINT32 InputProcessParameterCount,
        const XAPO_PROCESS_BUFFER_PARAMETERS* pInputProcessParameters,
        UINT32 OutputProcessParameterCount,
        XAPO_PROCESS_BUFFER_PARAMETERS* pOutputProcessParameters,
        BOOL IsEnabled)
    {
        BYTE* params = BeginProcess();
        //assert(IsLocked());
        assert(InputProcessParameterCount == 1);
        assert(OutputProcessParameterCount == 1);
        assert(NULL != pInputProcessParameters);
        assert(NULL != pOutputProcessParameters);


        XAPO_BUFFER_FLAGS inFlags = pInputProcessParameters[0].BufferFlags;
        XAPO_BUFFER_FLAGS outFlags = pOutputProcessParameters[0].BufferFlags;

        // assert buffer flags are legitimate
        assert(inFlags == XAPO_BUFFER_VALID || inFlags == XAPO_BUFFER_SILENT);
        assert(outFlags == XAPO_BUFFER_VALID || outFlags == XAPO_BUFFER_SILENT);

        // check input APO_BUFFER_FLAGS
        switch (inFlags)
```

```
        {
        case XAPO_BUFFER_VALID:
        {
            //input: process this data
            void* pvSrc = pInputProcessParameters[0].pBuffer;
            assert(pvSrc != NULL);
            //output: put processed data here
            void* pvDst = pOutputProcessParameters[0].pBuffer;
            assert(pvDst != NULL);

            float* pvSrcf = static_cast<float*>(pvSrc); // input, as float, mono
            float* pvDstf = static_cast<float*>(pvDst); // output, as float, stereo

            UINT32 len = pInputProcessParameters[0].ValidFrameCount * m_uChannels;

            // Prepare array to perform FFT
            transformData(*new_samples.pdata, pvSrcf, len, len);
            new_samples.size = len;

            // Start overlap-add
            size_t old_len = saved_samples.size;
            size_t new_len = old_len + new_samples.size;

            // If we don't have enough samples, save them and don't return anything
            if (new_len < step_size) {
                for (unsigned i = old_len; i < new_len; i++) {
                    (*saved_samples.pdata)[i] = (*new_samples.pdata)[i - old_len];
                }
                saved_samples.size = new_len;
            }
            // If we have enough, process them and put them in a queue to return
            else {
                // Put all needed samples into process_samples
                for (unsigned i = 0; i < old_len; i++) {
                    (*process_samples.pdata)[i] = (*saved_samples.pdata)[i];
                }
                for (unsigned i = old_len; i < step_size; i++) {
                    (*process_samples.pdata)[i] = (*new_samples.pdata)[i - old_len];
                }
                // Pad with 0's to match N
                for (unsigned i = step_size; i < fft_n; i++) {
                    (*process_samples.pdata)[i] = Complex(0, 0);

                }
                process_samples.size = fft_n;

                // Save samples we're not using this cycle
                size_t extra_samples = new_len - step_size;
                size_t index = new_samples.size - extra_samples;

                for (unsigned i = index; i < new_samples.size; i++) {
                    (*saved_samples.pdata)[i - index] = (*new_samples.pdata)[i];
                }
                saved_samples.size = extra_samples;

                //// Start overlap-add ////

                // Apply Hann Window
```

```
141                 applyHannWindow(process_samples);
142
143                 // Convert to freq domain
144                 fft(process_samples);
145
146                 // Set DC bin to 0
147                 (*process_samples.pdata)[0] = Complex(0, 0);
148
149                 spherical_coordinates* coords = (spherical_coordinates*)(params);
150                 changeFilter(*coords);
151
152                 // In case we need to apply a filter in 2nd or 3rd quadrant,
153                 we flip the filters
154                 if (this->flip_filters) {
155                     // Apply convolution, for each channel
156                     for (unsigned i = 0; i < process_samples.size; i++) {
157                         (*processed_samples.left.pdata)[i] =
158                             (*process_samples.pdata)[i] *
159                             (*hrtf_database[this->index_hrtf].fir.right)[i];
160                         (*processed_samples.right.pdata)[i] =
161                             (*process_samples.pdata)[i] *
162                             (*hrtf_database[this->index_hrtf].fir.left)[i];
163                     }
164                 }
165                 else {
166                     // Apply convolution, for each channel
167                     for (unsigned i = 0; i < process_samples.size; i++) {
168                         (*processed_samples.left.pdata)[i] =
169                             (*process_samples.pdata)[i] *
170                             (*hrtf_database[this->index_hrtf].fir.left)[i];
171                         (*processed_samples.right.pdata)[i] =
172                             (*process_samples.pdata)[i] *
173                             (*hrtf_database[this->index_hrtf].fir.right)[i];
174                     }
175                 }
176                 processed_samples.left.size = processed_samples.right.size =
177                 process_samples.size;
178
179                 // Convert back to time domain, each channel
180                 ifft(processed_samples.left);
181                 ifft(processed_samples.right);
182
183                 // Sum last computed frames to first freshly computed ones,
184                 // on each channel
185                 if (to_sum_samples.left.size > 0) {
186                     for (unsigned i = 0; i < to_sum_samples.left.size; i++) {
187                         (*processed_samples.left.pdata)[i] +=
188                             (*to_sum_samples.left.pdata)[i];
189                         (*processed_samples.right.pdata)[i] +=
190                             (*to_sum_samples.right.pdata)[i];
191                     }
192                     to_sum_samples.left.size = to_sum_samples.right.size = 0;
193                 }
194
195                 //processed samples, go to a queue, each channel separately
196                 size_t sizepr = processed_samples.left.size;
197                 size_t overlap = fir_size - 1;
198
```

```
                for (unsigned i = 0; i < sizepr - overlap; i++) {
                    (*ready_samples.left.pdata)[i + ready_samples.left.size] =
                        (*processed_samples.left.pdata)[i];
                    (*ready_samples.right.pdata)[i + ready_samples.right.size] =
                        (*processed_samples.right.pdata)[i];
                }
                ready_samples.left.size = ready_samples.right.size =
                    ready_samples.left.size + sizepr - overlap;

                // save M - 1 to sum, on each channel
                for (unsigned i = sizepr - overlap; i < sizepr; i++) {
                    (*to_sum_samples.left.pdata)[i - (sizepr - overlap)] =
                        (*processed_samples.left.pdata)[i];
                    (*to_sum_samples.right.pdata)[i - (sizepr - overlap)] =
                        (*processed_samples.right.pdata)[i];
                }
                to_sum_samples.left.size = to_sum_samples.right.size =
                    sizepr - (sizepr - overlap);
            }

            // If we have processed samples, return them to xaudio2
            if (ready_samples.left.size > 0) {

                UINT32 limit =
                    min((UINT32)ready_samples.left.size, this->max_frame_count * 2);
                size_t size_obuffer = limit - (limit % 2);

                //Put into output buffer
                for (unsigned i = 0; i < size_obuffer; i+=2) {
                    pvDstf[i]     = (float)(*ready_samples.left.pdata)[i / 2].real();
                    pvDstf[i + 1] = (float)(*ready_samples.right.pdata)[i / 2].real();
                }

                // Each channel outputs half buffer frames
                size_t ch_size = size_obuffer / 2;

                //Rotate and refresh size counter, on each channel
                for (unsigned i = ch_size; i < ready_samples.left.size; i++) {
                    (*ready_samples.left.pdata)[i - ch_size] =
                        (*ready_samples.left.pdata)[i];
                    (*ready_samples.right.pdata)[i - ch_size] =
                        (*ready_samples.right.pdata)[i];
                }
                ready_samples.left.size = ready_samples.left.size - ch_size;
                ready_samples.right.size = ready_samples.right.size - ch_size;


                pOutputProcessParameters[0].ValidFrameCount = size_obuffer;
            }
            // We don't have any processed sample, don't output anything
            else {
                pOutputProcessParameters[0].ValidFrameCount = 0;
            }

            new_samples.size = 0;
            processed_samples.left.size = processed_samples.right.size = 0;
            break;
        }
```

```
257
258         case XAPO_BUFFER_SILENT:
259         {
260             // All that needs to be done for this case is setting the
261             // output buffer flag to XAPO_BUFFER_SILENT which is done below.
262             break;
263         }
264
265         }
266
267         // set buffer flags
268         pOutputProcessParameters[0].BufferFlags = pInputProcessParameters[0].BufferFlags;
             // set destination buffer flags same as source
269         EndProcess();
270     }
271
272 };
```

```
1  /*****************************************************************************
2   *
3   * Fichero myXapo.cpp
4   *
5   *****************************************************************************/
6  #pragma once
7  #include "myXapo.h"
8  #include "filter_data.h"
9
10 myXapo::myXapo(XAPO_REGISTRATION_PROPERTIES* prop, filter_data* filters, size_t
       filters_size) : CXAPOBase(prop) {
11
12     m_uChannels = 1;
13     m_uBytesPerSample = 0;
14
15     index_hrtf = 0;
16     flip_filters = false;
17
18     new_samples.pdata = new CArray(512);
19     new_samples.size = 0;
20
21     fir_size = 128;                             // M
22     fft_n = 8 * pow(2, ceil(log2(fir_size)));  // N
23     step_size = fft_n - (fir_size - 1);        // L
24
25     // Start preparing filters
26     for (unsigned i = 0; i < filters_size; i++) {
27         CArray hrtf_temp = CArray();
28         UINT32 size = filters[i].data.buffer->AudioBytes / 2;
29         // Convert data into CArray
30         transformData(hrtf_temp,
31             (int16_t*)filters[i].data.buffer->pAudioData,
32             size, size);
33         // De-interleave so we have channels separated
34         CArray left  = hrtf_temp[std::slice(0, hrtf_temp.size() / 2, 2)];
35         CArray right = hrtf_temp[std::slice(1, hrtf_temp.size() / 2, 2)];
36         assert(left.size() == right.size());
37         // Copy data to fir structure, so we can have its FFT
38         filters[i].fir.left = new CArray(fft_n);
39         filters[i].fir.right = new CArray(fft_n);
```

```
        for (unsigned j = 0; j < left.size(); j++) {
            (*filters[i].fir.left)[j] = left[j];
            (*filters[i].fir.right)[j] = right[j];
        }
        // Pad with 0's to match N
        for (unsigned j = left.size(); j < fft_n; j++) {
            (*filters[i].fir.left)[j] = Complex(0, 0);
            (*filters[i].fir.right)[j] = Complex(0, 0);
        }
        // Convert to frequency domain
        fft(*filters[i].fir.left);
        fft(*filters[i].fir.right);
    }

    hrtf_database = filters;
    n_filters = filters_size;


    saved_samples.pdata = new CArray(step_size + new_samples.size);
    saved_samples.size = 0;

    process_samples.pdata = new CArray(fft_n);
    process_samples.size = 0;

    processed_samples.left.pdata = new CArray(fft_n);
    processed_samples.left.size = 0;
    processed_samples.right.pdata = new CArray(fft_n);
    processed_samples.right.size = 0;

    to_sum_samples.left.pdata = new CArray(fir_size);
    to_sum_samples.left.size = 0;
    to_sum_samples.right.pdata = new CArray(fir_size);
    to_sum_samples.right.size = 0;

    ready_samples.left.pdata = new CArray(fft_n * 2);
    ready_samples.left.size = 0;
    ready_samples.right.pdata = new CArray(fft_n * 2);
    ready_samples.right.size = 0;

}

myXapo::~myXapo() {
    // Free resources
    delete saved_samples.pdata;
    delete process_samples.pdata;
    delete processed_samples.left.pdata;
    delete processed_samples.right.pdata;
    delete to_sum_samples.left.pdata;
    delete to_sum_samples.right.pdata;
    delete ready_samples.left.pdata;
    delete ready_samples.right.pdata;
}

void myXapo::changeFilter(spherical_coordinates input) {

    unsigned index = -1;

    spherical_coordinates coords = input;
```

```
98
99      this->flip_filters = false;
100     if (coords.azimuth < 0) {
101         flip_filters = true;
102         coords.azimuth = -coords.azimuth;
103     }
104
105     double minDiffE = 500000;
106     int min_Elevation = -1;
107     // Search for good elevation
108     for (int i = 0; i < this->n_filters; i++) {
109         double diffE = abs(this->hrtf_database[i].elevation - input.elevation);
110         if (diffE < minDiffE) {
111             minDiffE = diffE;
112             min_Elevation = this->hrtf_database[i].elevation;
113         }
114     }
115
116     //Search for an azimuth with best elevation found
117     double minDiffA = 500000;
118     int min_Azimuth = -1;
119     for (int i = 0; i < this->n_filters; i++) {
120         if (this->hrtf_database[i].elevation == min_Elevation) {
121
122             double diffA = abs(this->hrtf_database[i].angle - coords.azimuth);
123
124             if (diffA < minDiffA) {
125                 minDiffA = diffA;
126                 min_Azimuth = this->hrtf_database[i].angle;
127             }
128         }
129     }
130     // Get the best filter we found
131     for (int i = 0; i < n_filters; i++) {
132         if (this->hrtf_database[i].angle == min_Azimuth && this->hrtf_database[i].
    elevation == min_Elevation) {
133             index = i;
134             break;
135         }
136     }
137
138     this->index_hrtf = index;
139 }
140
141 unsigned myXapo::getSampleGap(spherical_coordinates input) {
142     const double gapPerMeter = 128.57; // Result of 180 samples per 1.4m
143
144     return (unsigned) round(gapPerMeter * input.radius);
145 }
146
147 double myXapo::getGain(spherical_coordinates input) {
148     // Using 1.4m as reference
149     double distance = 1.4;
150
151     return (input.radius / distance) * 6; // 6dB for every doubling in distance
152 }
```

Bibliografía

[1] M. managed open source, "xaudio2 official documentation." https://docs.microsoft.com/en-us/windows/win32/xaudio2/xaudio2-apis-portal.

[2] J. V. Barchiesi, *Introducción al Procesamiento Digital de Señales*. Valparaíso, Chile: Ediciones Universitarias de la Universidad Católica de Valparaíso, 2008.

[3] J. O. S. III, "Mathematics of the discrete fourier transform (dft) with audio applications." https://www.dsprelated.com/freebooks/mdft/, 2007.

[4] B. Gardner and K. Martin, "Hrtf measurements of a kemar dummy-head microphone." https://sound.media.mit.edu/resources/KEMAR.html, 1994.

[5] F. Pfreundtner and M. Lederle, "Python 3d binaural audio simulation." https://github.com/felixpfreundtner/audio3d, 2016.

[6] S. W. Smith, "The scientist and engineer's guide to digital signal processing." http://www.dspguide.com/, 1997.