AUDASPACE

Trabajo de Fin de Máster

Master Universitario en Ingeniería Informática

VNiVERSiDAD D SALAMANCA

FEBRERO DE 2016

Autor: Juan Francisco Crespo Galán

Tutor: Francisco José García Peñalvo

D. Francisco José García Peñalvo, profesor titular del Departamento de Informática y Automática de la Universidad de Salamanca

CERTIFICA:

Que el trabajo titulado “Audaspace” ha sido realizado por D. Juan Francisco Crespo Galán, con DNI 71027851X y constituye la memoria del trabajo realizado para la superación de la asignatura Trabajo de Fin de Máster de la Titulación Máster en Ingeniería Informática de esta Universidad.

Y para que así conste a todos los efectos oportunos.

En Salamanca, a 12 de febrero de 2016

D. Francisco José García Peñalvo

Dpto. Informática y Automática

Universidad de Salamanca

RESUMEN

Existe gran cantidad de software que, de una forma u otra, necesita reproducir sonidos. Para paliar esta necesidad, existen múltiples bibliotecas de audio, tanto comerciales como de código abierto, no obstante, hay algunos problemas con las soluciones existentes en la actualidad.

Las bibliotecas de sonido de código abierto que constan, son en su mayoría de bajo nivel y muy específicas, obligando a los programadores a emplear gran cantidad de tiempo aprendiendo los pormenores de la misma y dificultando el proceso de desarrollo. Por otro lado, las que hay de alto nivel suelen ofrecer un conjunto muy limitado de funcionalidades, que en muchos casos puede no ser suficiente.

Existen también bibliotecas comerciales que resuelven estos problemas, pero a menudo son caras, lo que puede ser un gran problema para equipos de desarrolladores que cuenten con presupuesto reducido.

A continuación, se hablará de Audaspace, una biblioteca de audio de código abierto de reciente creación, que intenta eliminar los problemas antes descritos, es decir, es de alto nivel y con un rico conjunto de funcionalidades. Está escrita en C++, pero también ofrece una API en otros lenguajes de programación como C y Python.

En este trabajo se tratarán temas diferentes en relación a Audaspace. Por un lado, se detallará el proceso de desarrollo de diferentes funcionalidades de la biblioteca y algunos conceptos teóricos relacionados con dicho asunto y con el procesado digital de audio. Entre estas funcionalidades podemos destacar la implementación de un sistema eficiente de convolución, sonido biaural y capacidades de control de sonidos de alto nivel.

Dentro de este aspecto técnico, la parte central consiste en el sistema de convolución, que permitirá la aplicación de gran cantidad de filtros de sonido de forma muy rápida y sencilla. Entre estos efectos se encuentra la síntesis de sonido biaural, que permite al oyente situar diferentes sonidos en el espacio mediante auriculares. Esta funcionalidad está presente en Audaspace y ostenta un gran interés hoy en día, gracias al advenimiento de múltiples sistemas de realidad virtual que están volviéndose populares en diferentes campos. En cuanto a las capacidades de control de sonidos de alto nivel, están pensadas para facilitar en gran medida la integración de Audaspace en otros sistemas de software, tales como motores de videojuegos, que necesitan controlar y gestionar gran cantidad de sonidos diferentes.

Por otro lado, también se describirá la experiencia de usar Audaspace como motor de sonido en otro software, detallando los cambios y adaptaciones que fue necesario realizar para su integración.

Por último, se efectuará un análisis de las características de diversas bibliotecas de audio. El objetivo de este estudio es la realización de comparaciones entre Audaspace y otras bibliotecas de su sector, a fin de determinar los pros y los contras de cada una y de demostrar que Audaspace es una opción competitiva, no solo en cuanto a coste, sino también en cuanto a funcionalidades y características.

ABSTRACT

There is a lot of software out there that somehow requires to play sounds. To answer this necessity, developers usually choose one audio library among the multiple ones that are currently available. There are, however, some problems with this kind of libraries at this time.

On the one hand, the open source ones are usually low level, which makes development more complicated and forces developers to spend a lot of time learning them. On the other hand, the few open source, high level ones are often very limited, feature-wise.

There are also commercial audio libraries that are both, high level and feature-rich. These tend, however, to be quite expensive, which is a problem for teams working within a limited budget.

From now on, we'll talk about Audaspace, a recently created open source audio library that tries to eliminate the problems described above, meaning, it's a high level, feature-rich library with no cost. Audaspace is written in C++, but it also offers C and Python APIs.

This work will focus on several aspects related to Audaspace. Firstly, it will describe the development process of several library features, and theoretical concepts related to them, such as an efficient convolution system, binaural sound and high level sound control features.

Within the technical aspect, the convolution system is key, since it makes possible to use a large quantity of filters and effects, including binaural audio, which allows the listener to locate sounds in the space using only a couple of stereo headphones. This feature is present in Audaspace and it has gained special interest lately, thanks to the release of virtual reality hardware that has been becoming popular in several fields. Regarding the high level sound control capacities, their purpose is to ease the integration process in software that needs to manage a large quantity of sounds, such as game engines.

Secondly, the experience of using Audaspace as other software's sound engine will also be described, explaining in detail what changes and adaptations were needed to make the integration possible.

Last but not least, an analysis of several other audio libraries will be carried out. The goal of this is to make comparisons between Audaspace and other similar libraries, showing the benefits and disadvantages of each one and proving that Audaspace is not only competitive price-wise, but also feature-wise.

TABLA DE CONTENIDO

TABLA DE IMÁGENES

TABLA DE FRAGMENTOS DE CÓDIGO

1 INTRODUCCIÓN

Audaspace es una biblioteca de sonido de código abierto. Nació como el motor de audio de la aplicación de modelado 3D *Blender,* y ha sido recientemente lanzada como una biblioteca independiente.

La mayoría de las bibliotecas de audio de código libre suelen tener uno de los siguientes problemas:

- Muchas son de bajo nivel, es decir, son muy versátiles, pero obligan a los usuarios a realizar muchas tareas que en bibliotecas de más alto nivel ya estarían resueltas.
- Si no son de bajo nivel, la mayoría ofrecen un conjunto de funcionalidades reducido, que puede no ser suficiente para muchos casos de uso.

Por su parte, Audaspace permite a sus usuarios acceder de forma muy sencilla a un rico conjunto de funcionalidades a coste cero.

Mi papel en Audaspace no ha consistido en la implementación de toda la biblioteca, sino en la creación de una parte de ella dentro del contexto del proyecto europeo *VALS (Virtual Alliances for Learning Society) Semester of Code* [1] [2] [3] [4].

Este programa trata de promover la relación entre el mundo académico y el empresarial, de forma que se posibilite a los estudiantes la participación virtual en unas prácticas internacionales. El *Semester of Code* consiste en la participación en proyectos de código abierto, donde se abordarán problemas relevantes en el mundo real. Algo muy similar a lo que *Google* lleva tiempo haciendo en su conocido *Summer of Code.*

En concreto, mi trabajo ha consistido en el desarrollo de un conjunto de funcionalidades para Audaspace. A grandes rasgos han sido: Permitir gestionar fácilmente sonidos en entornos interactivos, implementación de un sistema eficiente de convolución de audio, reverberación, sonido biaural y *language bindings*.

La gestión y control de sonidos es una tarea necesaria en muchos campos y, si la biblioteca no ofrece facilidades para lograrlo, se obligará a los usuarios a realizar un esfuerzo mayor que, en algunos casos, podría disuadirles.

Por su parte, el sistema de convolución plantea un desafío por sus altos requisitos de rendimiento, pero permite la creación de una buena cantidad de efectos de sonido, tales como la reverberación y el sonido biaural, que también se han implementado como parte del proyecto y que no están presentes en muchas de las bibliotecas competidoras.

Por último, los *language bindings* permiten llevar nuestra funcionalidad a otros lenguajes de programación con un coste mínimo, lo cual será de gran ayuda para lograr un número mayor de usuarios potenciales.

Otro importante aspecto que se ha trabajado como parte del proyecto es el posicionamiento de Audaspace en su campo. Para esto se han realizado numerosas comparaciones con otras bibliotecas, tanto de código abierto como comerciales; junto con un análisis de la dificultad de integrar Audaspace en otro software que necesita características de sonido.

En este documento se van a desarrollar de forma detallada todos los temas anteriormente introducidos. La estructura es la siguiente:

- En el capítulo dos se detallarán los objetivos del proyecto, tanto de la biblioteca, como de las diferentes pruebas y comparaciones que se han realizado, explicando brevemente cada uno.
- En el tercero se explicarán varios conceptos teóricos relacionados con el mundo del audio. Mencionando problemas frecuentes y posibles soluciones.
- En el capítulo número cuatro, se realizará una descripción breve de las herramientas y bibliotecas utilizadas, nombrando sus principales características y explicando el porqué de la elección.
- En el quinto, se detallarán los aspectos más importantes del desarrollo de cada una de las funcionalidades, así los problemas que surgieron, y el porqué de las decisiones tomadas.
- En el sexto, se explicará cómo se ha utilizado Audaspace para su integración en un software existente.
- En el séptimo se comparará Audaspace con otras bibliotecas similares y se expondrán las ventajas e inconvenientes de ambos.
- Por último, en los capítulos ocho y nueve, se detallarán una serie de conclusiones obtenidas de la realización del proyecto, y un conjunto de campos en los que se podría trabajar en el futuro.

2 OBJETIVOS

Hay que distinguir entre objetivos de las modificaciones realizadas en la librería y objetivos de las pruebas y comparaciones que se han realizado.

2.1 OBJETIVOS DE LA LIBRERÍA

Aquí se presentarán los objetivos, que han de cumplir las partes que han sido añadidas a la biblioteca:

- **Gestión de sonido de alto nivel**: Funciones útiles para el manejo de audio en entornos interactivos:
    - **Gestor de reproducción**: Se debe de permitir manejar todo un conjunto de sonidos que estén reproduciéndose de forma sencilla y a través de un conjunto de categorías definidas por el usuario.
    - **Sonidos mutables**: Se ofrecerán clases que permitan tratar varios sonidos diferentes como uno solo. Esto otorgará gran sencillez al uso de efectos de sonido como pasos, disparos, etc. Que sonarían muy monótonos si se repitiera el mismo sonido de forma continua.
    - **Música dinámica**: Existirá una interfaz que permita al usuario de la biblioteca definir escenas de audio y transiciones entre las mismas; de esta forma será posible cambiar de escena de forma muy sencilla.
- **Sistema de convolución**: La convolución es una operación muy importante a la hora de aplicar filtros de sonido, pero es muy cara computacionalmente. Debido a esto, es necesario crear un sistema que realice esta operación del modo más eficiente posible.
- **Sonido biaural**: Se creará un sistema que permita al oyente identificar la posición de un sonido de un solo canal a través de unos auriculares estéreo convencionales. Para conseguir esto será necesario hacer uso de *Head Related Trensfer Functions* (HRTFs) con las que se realizará un procesado previo del sonido.
- ***Language bindings*:** Se posibilitará el acceso a todas estas funcionalidades a través de los lenguajes C y Python.
- **Multiplataforma*:*** La biblioteca habrá de funcionar a la perfección tanto en Windows como en Linux.

- **API (*Application Programming Interface*)[1] sencilla**: Uno de los objetivos principales es mantener una API sencilla, que permita gran facilidad de uso y aprendizaje por parte de los usuarios de la biblioteca.

2.2 OBJETIVOS DE LAS PRUEBAS Y DEMOSTRACIONES

Se realizarán también diversas demostraciones de uso y comparaciones de la librería con otras similares. Los objetivos que se persiguen son:

- **Demostrar la facilidad de uso:** Se mostrará la sencillez de la API tanto en términos generales como relativos a otras librerías de características similares.
- **Demostrar que puede integrarse en otro producto:** Se realizará la integración de Audaspace en un pequeño videojuego y se explicarán las mejoras que ha supuesto respecto al sistema anterior.
- **Medir en qué características se gana o se pierde:** Las comparaciones han de dictaminar las ventajas e inconvenientes de Audaspace respecto a algunos de sus competidores.

[1] **API:** *Application Programming Interface*. Conjunto de subrutinas, funciones y procedimientos que ofrece una biblioteca como capa de abstracción para ser utilizada por otro *software*.

3 CONCEPTOS TEÓRICOS

3.1 CONVOLUCIÓN

La convolución es una operación matemática, generalmente denotada por '$*$', y que transforma dos funciones *f* y g en una tercera función que en cierto sentido representa la magnitud en la que se superponen *f* y una versión trasladada e invertida de g.

Esta operación tiene diversas aplicaciones en múltiples campos. En esto caso, se hablará de la acústica, donde se puede decir que el eco, es el resultado de la convolución del sonido original, con una función que representa los diferentes objetos que lo reflejan.

En el caso que nos ocupa, no tiene mucho sentido utilizar la definición literal puesto que se utilizarán valores en instantes discretos de tiempo. Esto hace necesaria una aproximación numérica cuya fórmula pude observarse en la Figura 1.

$$y[n] = x[n] * h[n] = \sum_{i=-\infty}^{\infty} x[i]\,.\; h[n-i]$$

Figura 1: Fórmula convolución discreta

Las propiedades de esta operación son:

- **Conmutatividad**: $f * g = g * f$
- **Asociatividad**: $f * (g * h) = (f * g) * h$
- **Distributividad**: $f * (g + h) = (f * g) + (f * h)$
- **Asociatividad con multiplicación escalar**: $a(f * g) = (af) * g = f * (ag)$
- **Teorema de convolución**: En las condiciones adecuadas $\mathcal{F}(f * g) = \big(\mathcal{F}(f)\big) \cdot \big(\mathcal{F}(g)\big)$ Donde $\mathcal{F}$ es la transformada de Fourier[2].

Esto podría ser implementado tal cual, pero la complejidad computacional sería $\mathrm{O}(N^2)$, lo cual lo hace inviable. Para solucionar esto existen algoritmos de convolución rápida. Los más comunes utilizan un algoritmo FFT[3], que permite calcular la transformada discreta de Fourier de forma muy eficiente. De esta forma es posible conseguir una complejidad computacional de $\mathrm{O}(N \log N)$ [5].

[2] **Transformada de Fourier**: Transformación matemática utilizada para transformar señales entre el dominio del tiempo y el dominio de la frecuencia.

[3] ***Fast Fourier Transform*** **(FFT):** Algoritmo que computa la transformada discreta de Fourier de una secuencia (o su inversa) de forma mucho más eficiente. Reduce la complejidad computacional de $\mathrm{O}(N^2)$ a $\mathrm{O}(N \log N)$.

3.1.1 Convolución rápida usando FFT

La forma inmediata de aplicar esto para calcular la convolución de dos secuencias es realizar la FFT de cada una, multiplicar el resultado de ambas punto por punto, y realizar la FFT inversa del resultado de dicha multiplicación. Con esto obtendríamos la convolución de las dos secuencias iniciales.

Para realizar esto, hay que tener en cuenta que el resultado de la convolución tendrá una longitud igual a la suma de las longitudes de los operandos, menos uno ($M + L - 1$). Esto obliga a que el tamaño de la FFT sea, como mínimo, igual a dicha longitud ($N = M + L - 1$); de lo contrario, se producirá solapamiento o *aliasing en* el dominio del tiempo tal como se puede observar en la Figura 2 [6].

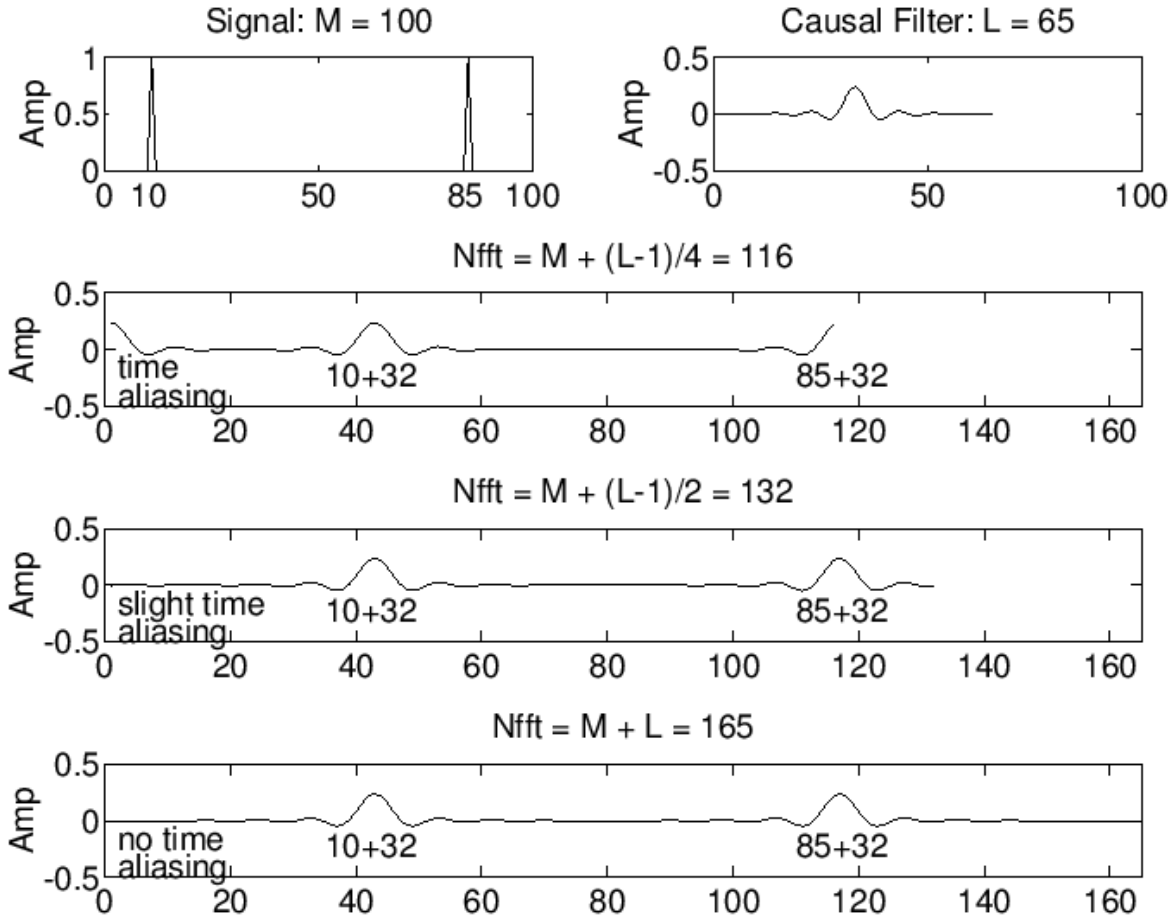

Figura 2: Aliasing en el dominio del tiempo.

Para utilizar una FFT del tamaño correcto, se debe aplicar *zero-padding*[4] a las secuencias de muestras[5] para que su longitud se convierta también en N.

[4] ***Zero-padding*:** Consiste en añadir ceros a una señal en el dominio del tiempo para aumentar su longitud.
[5] **Muestra:** Valor o conjunto de valores en un punto del tiempo y/o espacio.

El *zero-padding* consiste en añadir ceros a la secuencia de muestras para aumentar su longitud. Se puede ver de forma muy clara en la Figura 3 [7].

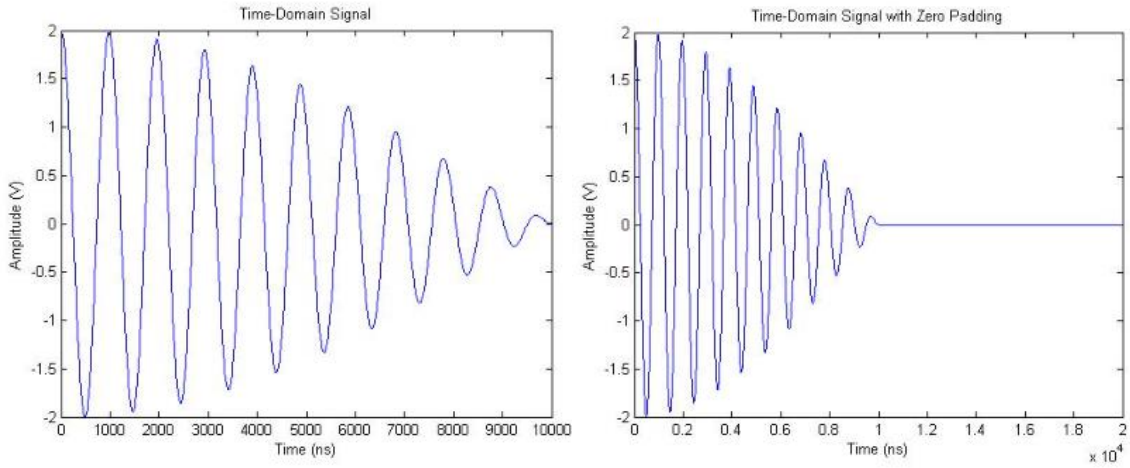

Figura 3: Señal sin zero-padding a la izquierda y con zero-padding a la derecha

Aun con todo esto, dependiendo del caso concreto, puede que usar este método directamente no sea viable. Por ejemplo, es muy frecuente encontrarse con una señal relativamente corta que habrá que convolver con una muy larga. Para estos casos, se suelen utilizar algoritmos que fragmentan la señal larga en partes más manejables, sobre las que se puede operar independientemente. Dos de estos algoritmos son el *overlap-save* y el *overlap-add*, que serán descritos a continuación.

3.1.1.1 Overlap-save

Se denomina *overlap-save* a una forma eficiente de evaluar la convolución discreta entre una señal muy larga y una respuesta al impulso finita (*Finite Impulse Response (FIR)*[6]).

El funcionamiento se basa en dividir la señal larga en varios fragmentos de longitud L y convolverlos independientemente de la forma explicada en el apartado anterior. En este caso no se usará *zero-padding*, por lo que los diferentes resultados parciales presentarán el solapamiento en el dominio del tiempo.

La forma de unir todos estos resultados parciales en un resultado final implica descartar esas partes solapadas y concatenarlos. Para que esto sea posible, es necesario que las diferentes partes de longitud L se superpongan parcialmente (*overlap*) y, de esta forma, se recalculen las partes que se han descartado por solapamiento o *aliasing* [5].

[6] ***Finite Impulse Response*** **(FIR):** Se llama *impulse response* o respuesta al impulso, a la salida que produce un sistema cuando se le presenta una breve señal de entrada, llamada impulso. Si es de duración finita, entonces se puede hablar de *finite impulse response.*

En el Código 1 se puede observar el pseudocódigo detallado de este método, mientras que, en la Figura 4 [8], se presenta un esquema del funcionamiento del mismo.

```
x = señal_larga
h = FIR
M = longitud(h)
overlap = M-1
N = L = 4*overlap  //Puede cambiar, Más rápido con N potencia de 2
step = N-overlap
H = FFT(h, N)
posicion = 0
while(posicion+N <= longitud(x))
      yt = IFFT(FFT(x(1+posicion : N+posicion),N)*H, N)
      y(1+posicion : step+posicion) = yt(M : N)
                              //Se descartan los valores con aliasing
      posicion = posicion+step
end while
```

Código 1: Pseudocódigo overlap-save

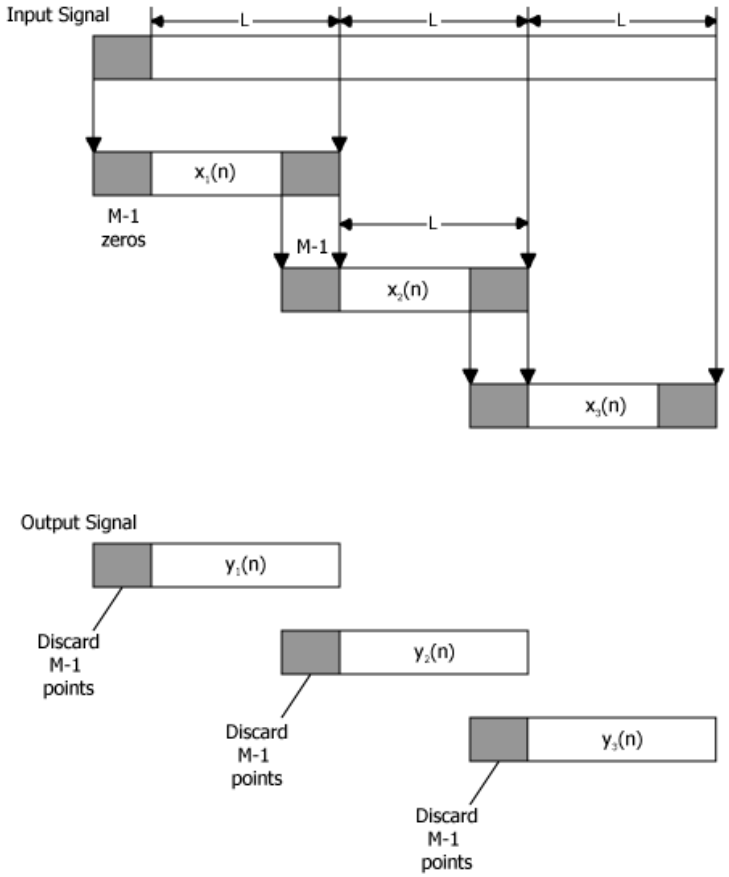

Figura 4: Esquema overlap-save

Para impedir que los primeros elementos del resultado queden corruptos, se pueden añadir $M-1$ ceros al comienzo de la señal, tal como se muestra en la Figura 4.

3.1.1.2 Overlap add

Este método es similar al *overlap-save,* descrito en el apartado anterior, en el sentido de que ambos dividen la señal grande en múltiples fragmentos de longitud L sobre lo que, independientemente, se realiza una convolución para obtener una serie de resultados parciales que luego son unidos en un único resultado final.

No obstante, también presenta varias diferencias. En primer lugar, se hace *zero-padding* para alargar cada fragmento de longitud L a longitud $M + L - 1$. Sobre estos fragmentos de longitud $M + L - 1$ se realizará la convolución con la señal corta. Siendo el resultado $M + L - 1$ elementos sin *aliasing*.

Para unir los diferentes resultados parciales, se han de sumar los últimos $M - 1$ elementos del resultado parcial 1, con los primeros $M - 1$ elementos del resultado parcial 2. Y así sucesivamente [5].

En el Código 2 se puede ver un pseudocódigo más detallado del método *overlap-add* y, en la Figura 5 [8], un esquema de su funcionamiento.

```
x = señal_larga
h = FIR
M = longitud(h)
L = Numero_mayor_que_cero
N = M+L-1      //Puede ser mayor, FFT más rápida con N potencia de 2
Lx = longitud(x)
H = FFT(h, N)
i = 1
y = array con M+Lx-1 ceros
while(i<=Lx)
      yt=IFFT(FFT(x(i : il),N)*H, N)
      k=min(i+N-1, M+Lx-1)
      y(i : k) = y(i : k) + yt(1 : k-i+1)
      i = i+L
end while
```

Código 2: Pseudocódigo overlap-add

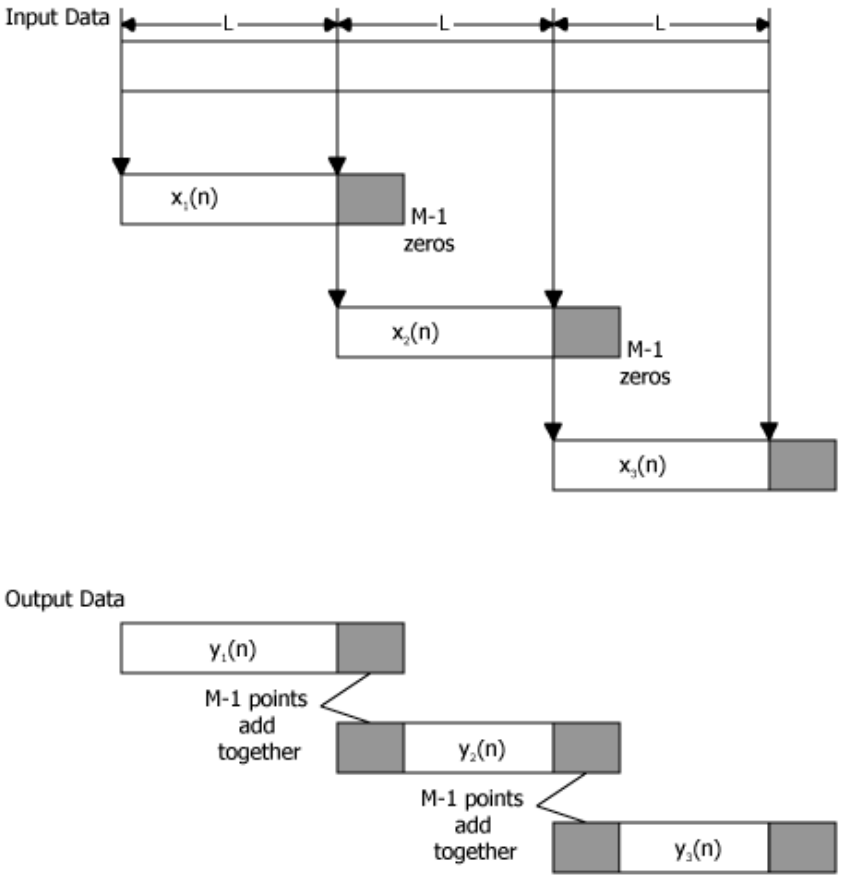

Figura 5: Esquema overlap-add

Overlap-save es algo más rápido que *overlap-add*, puesto que no es necesario hacer sumas. Por esta razón, es el método que finalmente se acabó escogiendo en Audaspace. No obstante, fueron necesarias más optimizaciones para alcanzar el rendimiento deseado. Todo esto será explicado con más detalle en el apartado Aspectos relevantes del desarrollo.

3.2 REVERBERACIÓN

La reverberación es un fenómeno sonoro, que se produce debido a la reflexión del sonido en las diversas superficies del entorno.

Cuando una persona oye un sonido, éste llega a través de dos vías: El sonido directo, y el sonido que se ha reflejado en algún obstáculo. Si el sonido reflejado es inteligible, se denomina eco. Sin embargo, si se percibe como una adición que modifica el sonido original, recibe el nombre de reverberación.

En el campo del procesamiento digital de audio, es posible simular la reverberación mediante la convolución del sonido con una respuesta a impulso.

Esta respuesta a impulso, consiste en un audio pregrabado de las reflexiones, que genera el entorno que se quiere simular, como respuesta ante un sonido de muy corta duración.

3.2.1 Canales de audio y reverberación convolutiva

Un archivo de audio generalmente presenta varios canales que suelen ser reproducidos por diferentes altavoces. Las configuraciones de canales más frecuentes son mono, estéreo, 2.1, 5.1 y 7.1. Siendo el sonido estéreo es especialmente común.

A la hora de realizar la convolución de un sonido, puede que sea necesario hacer una o varias convoluciones por cada canal, dependiendo de la salida que se desea obtener. En la Figura 6 [9] se pueden ver varios ejemplos de esto.

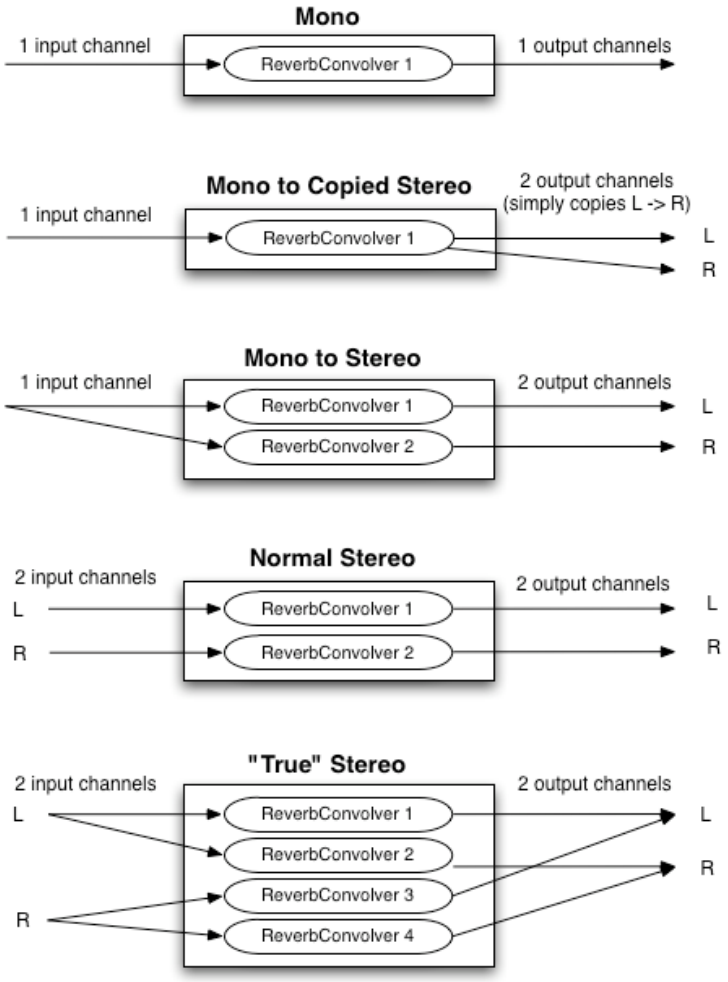

Figura 6: Convolución según canales de audio

Cada *ReverbConvolver* simboliza un sistema que realiza la operación de convolución con una respuesta a impulso diferente. Algo que se observa es que para convolver un sonido de X canales; pueden utilizarse X respuestas a impulso (una para cada canal). Existiendo también la opción de utilizar un solo filtro para todos los canales.

3.3 AUDIO BIAURAL

Cuando se habla de audio biaural, normalmente se hace referencia a reproducir sonidos que un oyente pueda posicionar en el espacio utilizando solamente unos auriculares estéreo.

Normalmente un sonido reproducido por auriculares no puede ser bien posicionado en el espacio por el oyente, pues físicamente se está reproduciendo al lado de la oreja. Esto hace que reproducir un sonido inmersivo, que nos haga sentir dentro de la escena, a través de auriculares, sea un desafío.

Normalmente para conseguir esa sensación se han usado sistemas de sonido 5.1 y 7.1. Es decir, se utilizan cinco o siete fuentes de sonido diferentes para que se pueda distinguir la posición de los sonidos que se emiten. Tal como se muestra en la Figura 7 [10].

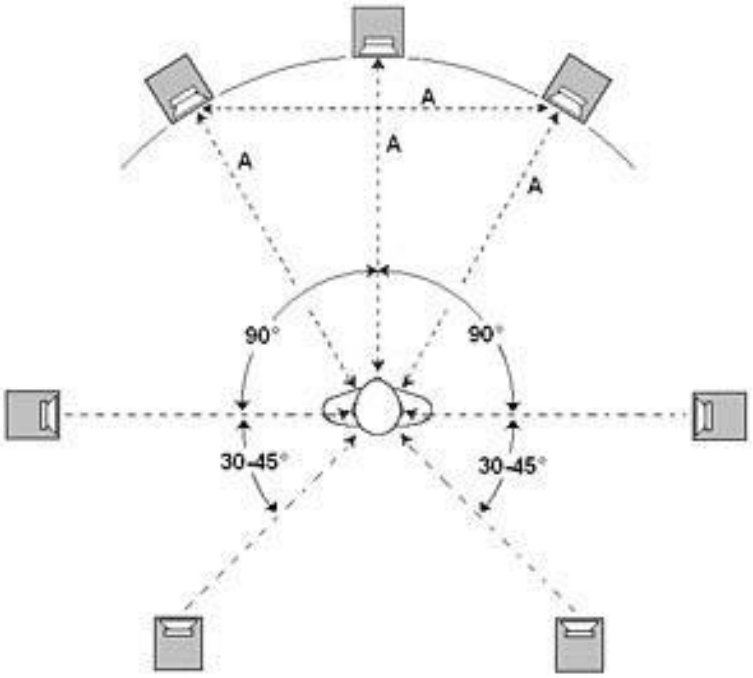

Figura 7: Sistema de sonido 7.1

Este método, obviamente funciona, pues los sonidos se emiten físicamente en la posición aproximada desde la que se emiten en el escenario virtual. No obstante, también presenta varios inconvenientes: Espacio, sonido dependiente de la sala…

Por suerte, existe un método que permite reproducir esto a través de auriculares. Los seres humanos tenemos solamente dos oídos, pero, sin embargo, podemos percibir la posición de los sonidos que nos rodean con mucha precisión.

El hecho de que los oídos estén separados, y eso provoque que haya diferencias en cuándo llega un sonido a cada oído, no es suficiente para explicar cómo posicionamos los sonidos que nos rodean. Los mayores responsables son la forma de nuestras orejas y cabeza, que modifican el sonido dependiendo de la dirección por la que nos llega. Todo esto se puede ver en la Figura 8 [11] y la Figura 9 [12]

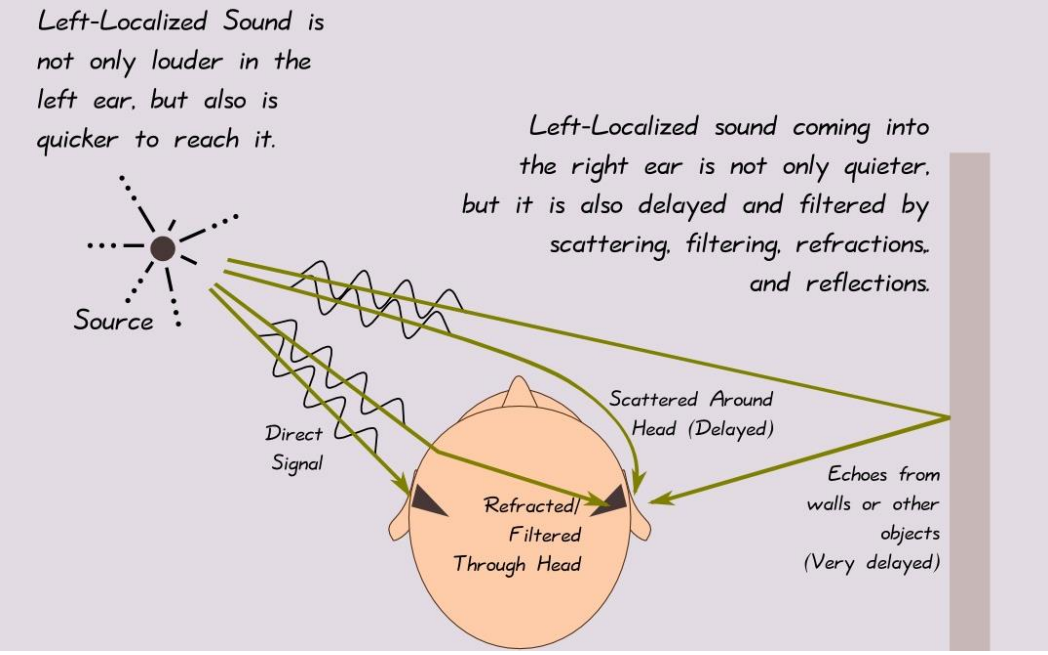

Figura 8: Diferencias en cómo cada oído recibe un sonido

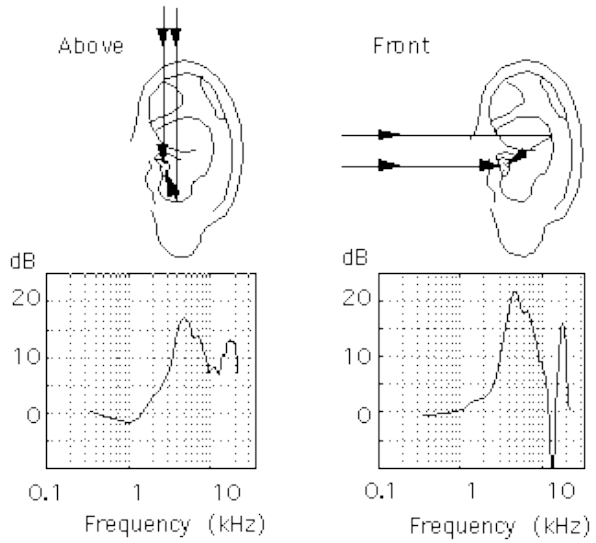

Figura 9: Forma de las orejas modificando un sonido

Es posible simular esos matices en un sonido producido en un entorno virtual gracias al uso de funciones de transferencia relativas a la cabeza, o HRTFs, por sus siglas en inglés (*Head Related Transfer Function*).

3.3.1 HRTFs

Las modificaciones que la forma de nuestra cabeza y orejas producen en un sonido, pueden ser capturadas en una respuesta a impulso.

Para grabarlas, normalmente se utilizan micrófonos en el interior de los oídos de, o bien un humano, o bien una cabeza artificial. Estos micrófonos grabarán respuestas a impulsos emitidos en diferentes posiciones alrededor de la cabeza. Generalmente, se intenta que estas posiciones formen una esfera en torno a la cabeza [13], tal como se observa en la Figura 10 [14].

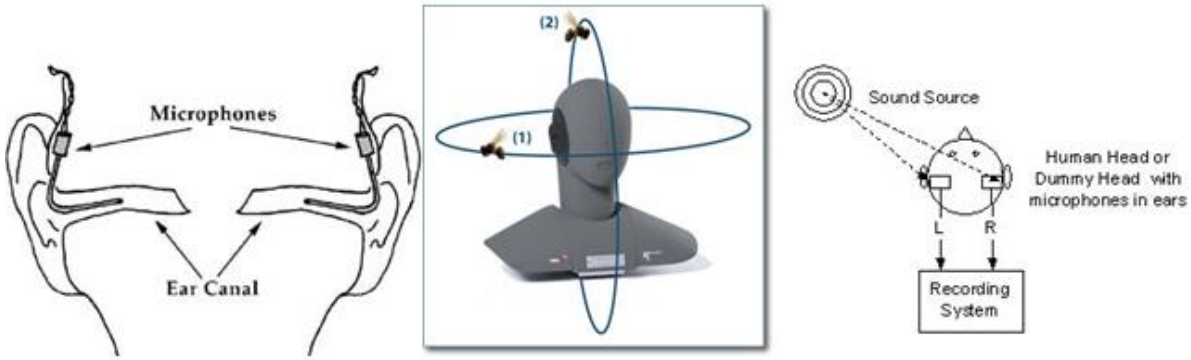

Figura 10: Grabación de HRTFs

Para poder ser usadas en la simulación de sonidos espaciales, debe de haber una distancia reducida entre los puntos desde donde se han grabado las diferentes HRTFs.

3.3.2 Usando la convolución

Para aplicar todo lo anterior es posible utilizar la convolución. Usando un sonido de un único canal como entrada, se aplicarán dos convoluciones, con dos HRTFs diferentes, una para cada oído. La salida será un sonido con dos canales que, si es reproducido mediante auriculares estéreo, producirá la impresión de que suena en una posición determinada del espacio, tal como se observa en la Figura 11 [13].

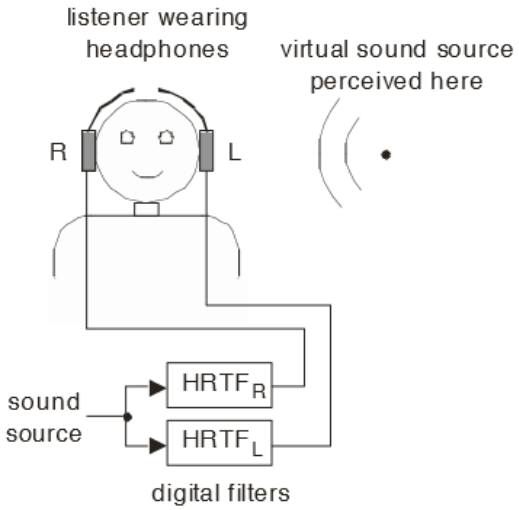

Figura 11: Síntesis de un sonido biaural

3.4 LANGUAGE BINDINGS

Se da el nombre de *language binding* a una especie de puente que permite utilizar una librería escrita en un lenguaje de programación desde otro lenguaje de programación. Por ejemplo, en el caso que nos ocupa, Audaspace está escrita en C++, pero puede ser usada desde C y desde Python gracias a este mecanismo.

Hay diferentes formas de llevar esto a cabo. Una de las más comunes es utilizar C como base común, pues la gran mayoría de los lenguajes de programación o bien están escritos en C (Python, Ruby, Perl…) o bien son compatibles con él (C++, C#...)

Una de las mayores motivaciones para hacer esto es la reutilización de código. En vez de reescribir la misma librería en varios lenguajes, es menos costoso escribir este puente entre lenguajes.

3.5 PATRONES DE DISEÑO

Un patrón de diseño es una solución a un problema de diseño. Ha de ser reutilizable y estar altamente probado.

Durante el desarrollo, se han utilizado varios patrones, aunque hubo un caso en el que se acabó desechando el uso del patrón. A continuación, un resumen de los patrones de diseño usados en este proyecto.

3.5.1 Singleton

Este patrón garantiza que solo se pueda crear una instancia de una clase. Además, proporciona acceso global a dicha instancia.

Figura 12: Singleton

En la Figura 12 se observa el esquema UML del patrón *singleton*, el método subrayado es un método estático.

Se pensó en utilizar este patrón para asegurar que solo existía una instancia de la clase *ThreadPool*, encargada de manejar una reserva de hilos, y a la que se le pueden encargar diversas funciones. Al final esta idea se desechó. Los motivos se explican más claramente en el apartado "Aspectos relevantes del desarrollo", pero tienen que ver con que se detectó que tener varias instancias de la clase, podía tener pequeñas ventajas en casos de uso muy concretos. Además, el uso del patrón tampoco aportaba grandes beneficios que compensaran lo anterior.

3.5.2 Abstract Factory

En programación orientada a objetos, una factoría es un objeto dedicado a crear otros objetos.

En Audaspace se usan factorías para crear dos tipos de objetos, *Devices* y *Readers*. En concreto, se utiliza el patrón a*bstract factory* para la creación de ambos. El diagrama UML de este patrón se puede observar en la Figura 13.

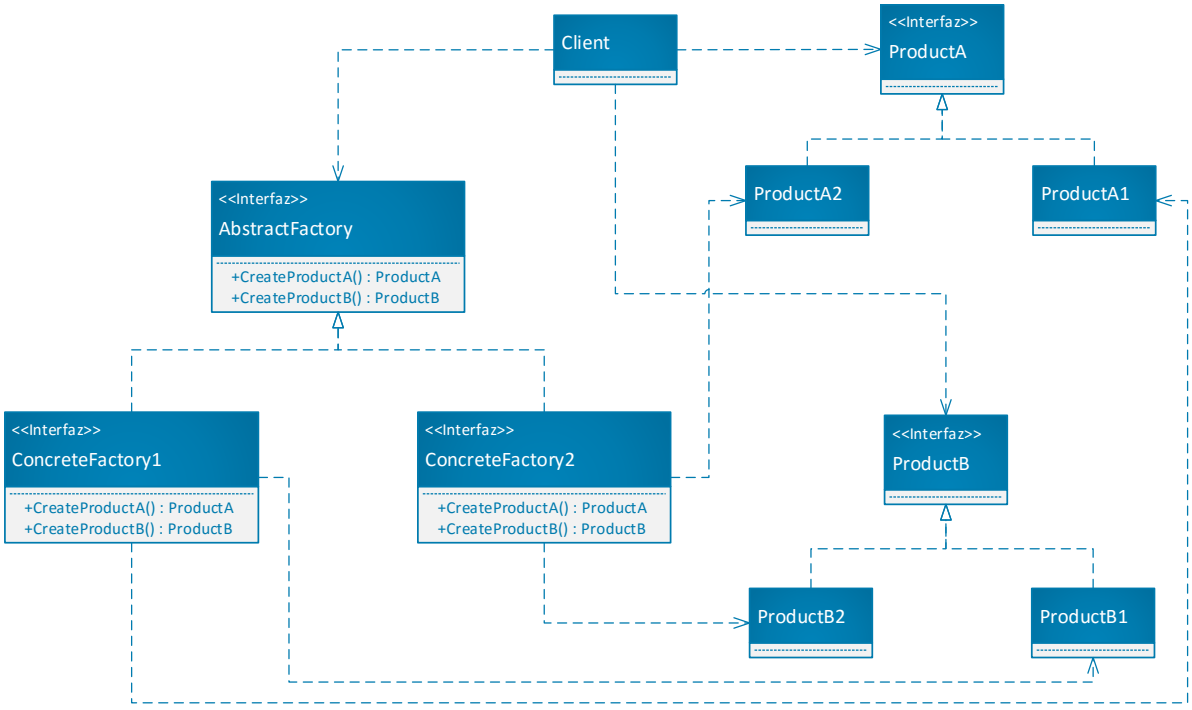

Figura 13: Abstract factory

El uso de este patrón, permite aislar completamente el código del cliente del *Device* concreto que se utilice; siendo éste gestionado por la biblioteca mediante un sistema de prioridades. Además, si se añadieran nuevos tipos de *Device* en futuras versiones, el cliente podría usarlos sin cambiar una sola línea de código.

El otro lugar donde se utiliza este patrón en Audaspace es en la creación de *Readers*. Un *Reader* es un objeto que permite la reproducción de un sonido, y es un método de la interfaz *ISound* el que se encarga de crearlos.

El funcionamiento es el siguiente, en primer lugar, se instancia una implementación de la interfaz *ISound,* que es la factoría. Este objeto representaría un sonido concreto que se puede reproducir cuantas veces se quiera, incluso simultáneamente. Por cada reproducción independiente que se haga, se creará un *Reader* de un tipo concreto que concuerde con el tipo de sonido.

El cliente está completamente aislado del tipo de *Reader* que se utiliza o qué se necesita para su instanciación.

3.5.3 Composite

Este patrón describe como un grupo de objetos puede ser tratado de la misma forma que un solo objeto. En la Figura 14 se observa el diagrama UML de este patón.

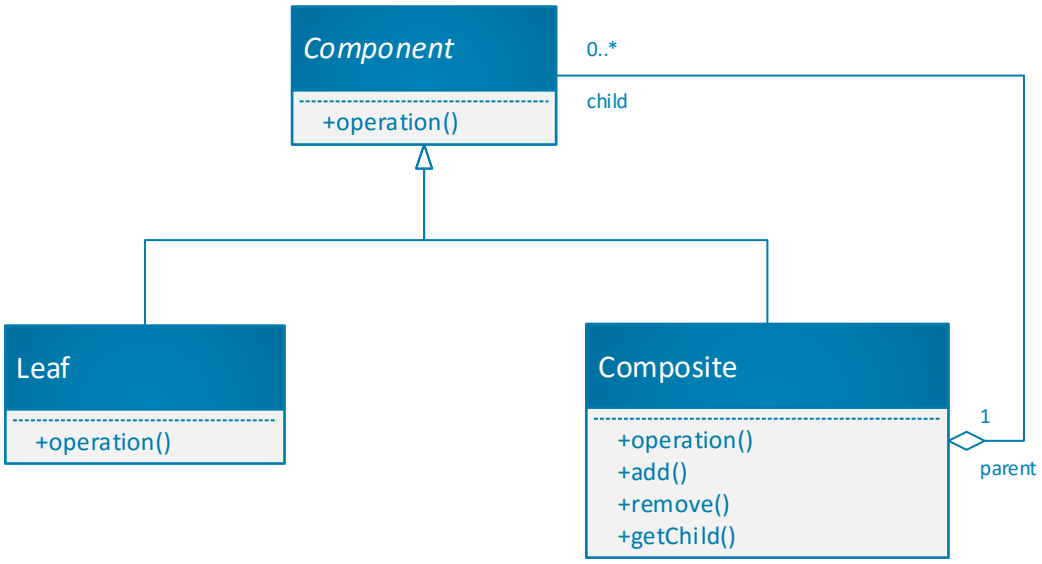

Figura 14: Patrón Composite

La motivación de este patrón, es permitir una forma uniforme de tratar objetos básicos y complejos que pertenecen a la misma jerarquía. En el caso de Audaspace este patrón se utiliza en los *Sounds* y los *Readers*.

Por ejemplo, es muy común que un *Sound* se componga de otros; éste es el caso de los sonidos complejos, que aplican diferentes efectos sobre otros sonidos. Siguiendo el esquema de la Figura 14, es muy sencillo superponer tantos efectos como se quiera a un sonido simple. Siempre manteniendo, además, una forma uniforme de uso, que es definida por la interfaz *ISonud*.

El caso de los *Readers* es equivalente a lo anteriormente descrito. Es de hecho en estos objetos donde de verdad se superponen los efectos, pues los *Sounds* son simples factorías.

4 TÉCNICAS Y HERRAMIENTAS

4.1 VISUAL STUDIO 2015

A día de hoy es la versión más reciente del IDE[7] (*Integrated Development Environment*) creado por Microsoft. Puede ser usado para crear gran variedad de aplicaciones y soporta gran cantidad de lenguajes, como son: *C, C++, C#, Visual Basic, F#, XML, HTML y JavaScript* entre otros.

Visual Studio está disponible para su descarga desde la siguiente página web: https://www.visualstudio.com

Existen múltiples versiones, algunas de pago a diferentes precios según sus características, y una gratuita llamada *Community*, que es la que se ha utilizado en este proyecto.

La licencia de las diferentes versiones de Visual Studio puede descargarse desde el siguiente enlace: https://www.microsoft.com/en-us/download/details.aspx?id=13350

Visual Studio es uno de los IDE más completos a la hora de desarrollar para Windows gracias a un gran elenco de características, entre las que, a mi juicio, destacan:

- **Depurador potente y muy sencillo de utilizar:** Permite una rápida visualización del valor de una variable en cualquier momento, además de medición de tiempos, uso de CPU y uso de memoria. También permite ejecución paso a paso, vuelta atrás, y modificar el código durante la depuración.
- **Soporte para *plugins* y herramientas de terceros:** Esto permite aumentar la funcionalidad de Visual Studio en gran medida. Algunas de ellas, como "*Resharper*" o "*Visual Assist*" están sumamente extendidas entre la comunidad de desarrolladores.
- **Documentación muy completa:** Con ejemplos muy valiosos para aprender y una referencia rápida muy útil cuando se quiere saber qué hace algún método en concreto.

El desarrollo se comenzó utilizando Visual Studio 2013 porque era la versión más reciente en aquel momento. No obstante, a medio desarrollo, se actualizó a la versión 2015 dado que, presentaba notables mejoras en el depurador, y ofrecía un soporte más completo a C++11.

4.2 AUDACITY

Para editar los diferentes efectos sonoros que pueden escucharse en las demostraciones, se ha utilizado este programa.

Audacity es un editor y grabador de sonido multiplataforma y de código abierto. Soporta archivos WAV, AIFF, FLAC, MP2, MP3, Ogg, AC3, ACC y WMA, entre otros.

[7] **IDE:** Entorno integrado de desarrollo. Es una aplicación informática que ofrece a los programadores un conjunto de servicios y herramientas que les facilita el desarrollo de software.

Las funcionalidades de edición que ofrece, demostraron ser más que suficientes para las necesidades de este proyecto. Además, Audacity es gratuito y distribuido bajo licencia *GNU General Public License*.

El programa está disponible para descarga en http://www.audacityteam.org

4.3 GitHub

GitHub es una de las plataformas de desarrollo de software colaborativo más conocidas. En ella, millones de personas alojan sus proyectos utilizando el sistema de control de versiones[8] *Git.*[9]

GitHub es gratuito para proyectos públicos a los que todo el mundo pueda acceder, además, facilita mucho el desarrollo colaborativo, algo muy importante para cualquier proyecto de código abierto. Esto hace a GitHub la plataforma ideal para un proyecto *open source* como es Audaspace. Por si esto fuera poco, GitHub puede ser integrado con la mayoría de IDEs, lo que facilita aún más su uso.

La web oficial de GitHub es https://github.com/

4.4 CMake

CMake es un sistema que controla el proceso de construcción de código (*build*), de forma independiente al sistema operativo y al compilador.

Esta herramienta utiliza un fichero de configuración, llamado CMakeLists.txt, para crear un entorno nativo de construcción que permita la compilación del código fuente. Además, también permite una fácil configuración de las dependencias que se necesiten.

Las ventajas que aporta el uso de CMake vienen sobre todo por el lado de la multiplataforma, pues permite un esquema construcción único para múltiples plataformas.

CMake es gratuito, de código abierto, y se puede descargar desde https://cmake.org/download bajo licencia propia que se encuentra en https://cmake.org/licensing/.

4.5 FFTW

FFTW es una biblioteca escrita en C y que se usa para calcular la transformada discreta de Fourier (DFT) en una o más dimensiones.

[8] **Sistema de control de versiones:** Sistema que permite la gestión de los diversos cambios que se realizan sobre los elementos de algún producto. Debe proporcionar un mecanismo de almacenamiento de los elementos que se controlen, posibilidad de realizar cambios sobre los mismos y un registro histórico de los cambios que haya sufrido cada elemento.

[9] ***Git:*** Es un sistema de control de versiones distribuido. Diseñado por Linus Torvalds.

Es gratuita y se distribuye bajo licencia *GNU General Public License*, pudiéndose descargar en la web http://fftw.org/download.html.

En Audaspace, esta librería es utilizada para calcular la transformada de Fourier cuando se realiza una operación de convolución. De esta forma es posible aplicar el procedimiento descrito en el punto 3.1.1.

Las principales razones que llevaron al uso de esta librería son:

- **Portabilidad:** La librería es portable, pudiéndose ejecutar sin problemas y con buen rendimiento en la gran mayoría de arquitecturas.
- **Velocidad:** Uno los principales puntos fuertes de FFTW es su rapidez. En la página web http://www.fftw.org/benchfft/ se pueden ver diferentes *benchmarks*[10] que comparan su rendimiento con el de otras opciones.

4.6 C++ 11

C++ 11 es el lenguaje de programación en el que Audaspace ha sido escrito. Su elección está motivada por varios aspectos entre los que destacan:

- **Se trata de un lenguaje multiplataforma**: Un aspecto muy importante, pues uno de los objetivos de Audaspace es funcionar en diversas plataformas.
- **Permite utilizar fácilmente bibliotecas escritas en C**: Es fácil utilizar bibliotecas que ofrecen una interfaz en C, como FFTW.
- **Gestión de memoria simplificada**: C++ 11 permite el uso de punteros inteligentes, que permiten, en gran medida, simplificar la gestión de memoria.
- **Soporte para hilos**: C++ 11 permite trabajar con hilos sin necesidad de usar ninguna librería externa.

[10] ***Benchmark:*** Técnica utilizada para medir el rendimiento de un Sistema o componente del mismo.

5 ASPECTOS RELEVANTES DEL DESARROLLO

5.1 NOCIONES BÁSICAS SOBRE AUDASPACE

En primer lugar, es importante explicar algunos conceptos básicos sobre la construcción y el funcionamiento de la versión inicial de Audaspace. Particularmente los conceptos de `Sound`, `Reader`, `Device` y `Handle`, puesto que son muy imprescindibles para entender cómo funcionan las partes desarrolladas.

5.1.1 *Sounds* y *Readers*

En primer lugar, es necesario comenzar por los conceptos de `Sound` y `Reader`, que ya fueron mencionados brevemente en el apartado 3.5.2. En la Figura 15, se puede observar un pequeño diagrama de clases que describe la relación entre `Sounds` y `Readers`.

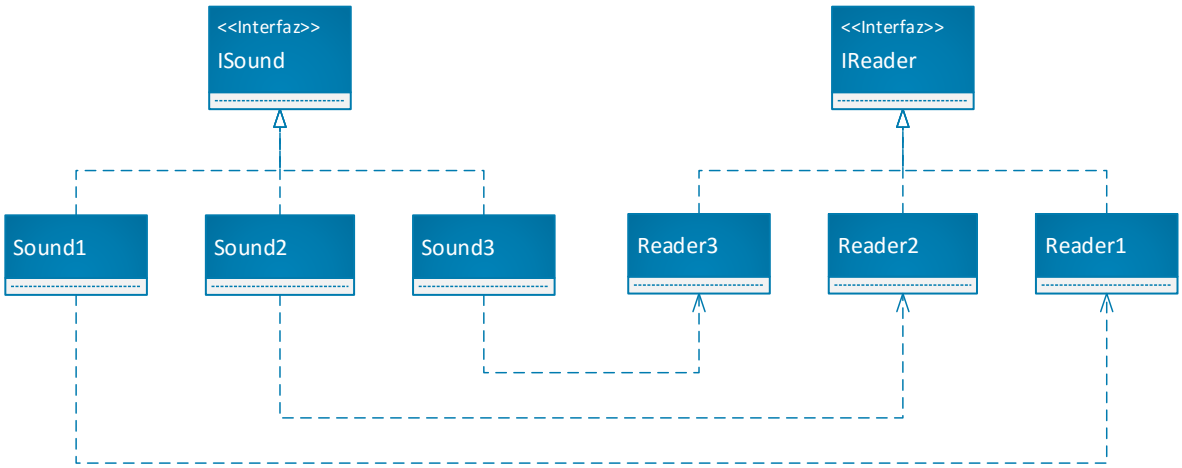

Figura 15: Sounds y Readers

Un `Sound` no es más que una factoría que el cliente instancia y que puede crear `Readers`. Hay muchos `Sounds` diferentes en la biblioteca, y cada uno representa un tipo de sonido con características diferentes (procedencia, efectos, etc.). No obstante, todos implementan la interfaz `ISound` que define la funcionalidad mínima que deben tener.

La interfaz `ISound` obliga a todas las clases que la implementan a poder crear un *Reader* determinado, acorde al tipo de sonido del que se trate. La tarea de ese *Reader* es tanto posibilitar la reproducción del sonido, como el control de dicha reproducción. Al igual que en el caso anterior, todo *Reader* implementa la interfaz `IReader` que define su funcionalidad mínima.

Aunque hacer esto no es algo habitual en Audaspace, pues los usuarios no suelen necesitar manipular `Readers` directamente, en el Código 3, se puede observar un pequeño fragmento de código que instancia un sonido y crea su *Reader*.

```
std::shared_ptr<ISound> sound(std::make_shared<File>("sonido.wav"));
std::shared_ptr<IReader> reader = sound->createReader();
```

Código 3: Instanciación de Sounds y Readers

Un detalle importante es que no hay límite a cuántos `Readers` puede crear un sonido. Esto permite que el mismo sonido pueda reproducirse cuantas veces se desee, incluso de forma simultánea.

Otro aspecto importante, tanto de `Sounds` como de `Readers`, es su uso del patrón *composite* descrito en el apartado 3.5.3. Esto permite superponer fácilmente efectos de sonido definidos en diferentes componentes.

Por ejemplo, en el Código 4 se puede observar cómo se crea un sonido que aplica un efecto de *fade-in*[11] y cambio de tono sobre un sonido básico cargado desde un archivo.

```
std::shared_ptr<ISound> sound(std::make_shared<File>("sonido.wav"));
std::shared_ptr<ISound> pitchSound(std::make_shared<Pitch>(sound, 2.0));
std::shared_ptr<ISound> fadeSound(std::make_shared<Fader>(pitchSound));
```

Código 4: Uso del patrón composite en sonidos

5.1.2 Devices y Handles

Un `Device` representa un objeto que permite la reproducción de `Sounds` a través de sus `Readers`. Para la creación de objetos de este tipo, se utiliza el patrón *abstract factory,* descrito en el punto 3.5.2. En primer lugar, hay que cargar los *plugins* que definen los tipos de `Device` disponibles, acto seguido, se solicita un `DeviceFactory` cuyo tipo concreto dependerá de los *plugins* cargados. Finalmente, utilizando dicha factoría, es posible crear el objeto `Device`, tal como se observa en el Código 5.

Pare reproducir sonidos, un `Device` utiliza un hilo aparte, por lo que es necesario tener cuidado con las condiciones de carrera, aunque la mayoría de acciones son *thread-safe*[12].

```
PluginManager::loadPlugins("");
auto factory = DeviceManager::getDefaultDeviceFactory();
auto device = factory->openDevice();
std::shared_ptr<ISound> sound(std::make_shared<File>("sonido.wav"));
auto handle = device->play(sound);
handle->pause();
```

Código 5: Creación de Device

[11] ***Fade-in, fade-out***: Efecto que implica que un sonido aumente su volumen paulatinamente al empezar su reproducción o lo reduzca al finalizarla respectivamente.

[12] ***Thread-safe:*** Se dice del código que funciona correctamente cuando es ejecutado concurrentemente por varios hilos.

Por su parte, un `Handle`, permite el control de una reproducción en concreto. Acciones como parar, pausar, reanudar, cambiar volumen, reproducir en bucle, etc. Son posibles invocando los métodos de la interfaz `IHandle`. En el Código 5 también se muestra el método de obtención de un `Handle`.

5.2 SONIDOS MUTABLES

Esta funcionalidad trata de implementar un nuevo tipo de sonido que pueda cambiarse a sí mismo, de ahí su nombre. Su utilidad se puede ilustrar con el siguiente ejemplo:

Un personaje de un juego se mueve caminando y emite un sonido con sus pasos. La forma inmediata de hacer eso es reproducir un determinado sonido cada vez que se reproduzca la animación de caminar. No obstante, con este método, los pasos serían muy monótonos y el oyente se acabaría dando cuenta.

Es posible implementar la variación en la aplicación que usa los sonidos, sin embargo, el enfoque en Audaspace es proporcionar esta funcionalidad directamente en la biblioteca, simplificando así su uso en estos casos.

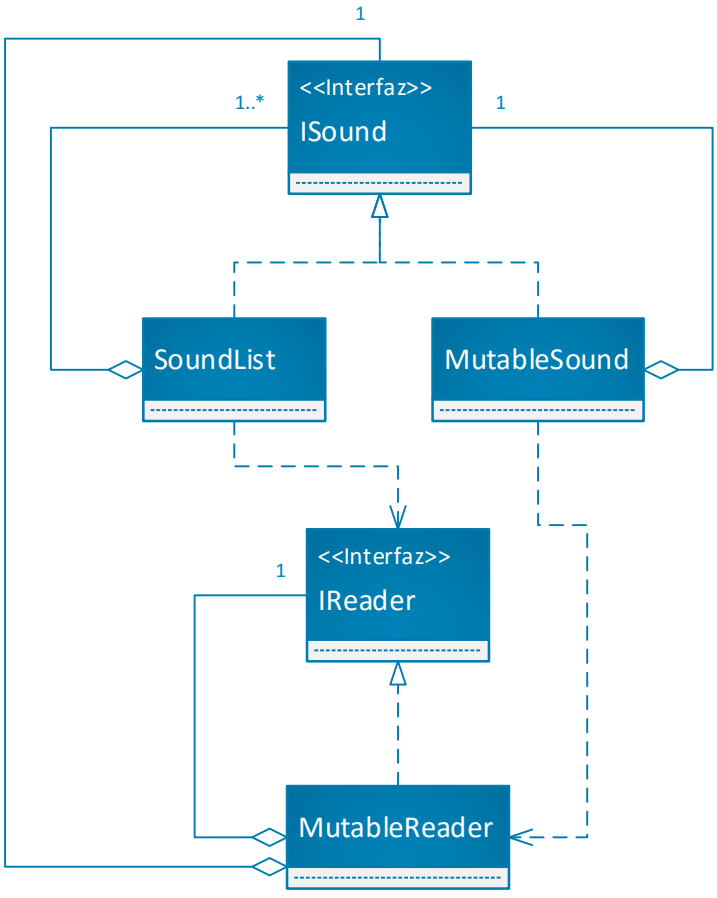

Figura 16: Diagrama de sonidos mutables

En la Figura 16 se muestra el diagrama de clases de esta funcionalidad. Como se puede observar, se está utilizando el patrón *composite* para aplicar los efectos de estas clases a otros sonidos, sin importar sus tipos concretos.

SoundList

La clase `SoundList` define un nuevo tipo de sonido que implementa la interfaz `ISound`. Este nuevo sonido está formado por una agrupación de otros sonidos de diferentes tipos. El trabajo de esta clase es utilizar una lista de sonidos interna para crear un *Reader* diferente en cada reproducción (el tipo de *Reader* que se cree depende del sonido interno que se vaya a reproducir), variando así el verdadero sonido que se reproduce.

No obstante, esto no funcionaría en el caso de una reproducción en bucle, pues el mismo *Reader* sería el que se reproduce una y otra vez. Para solucionar esto se crearon las siguientes clases.

MutableSound

Otro nuevo tipo de sonido, recibe un sonido de cualquier tipo en su constructor y crea con él un *Reader* del tipo `MutableReader`.

MutableReader

Es un *Reader* que recibe un sonido como argumento en su constructor. Su tarea es utilizar ese sonido para crear un nuevo *Reader* interno cada vez que se intente reiniciar la reproducción. Siendo este *Reader* interno el que en realidad ejecuta todas las tareas.

Si el sonido proporcionado es del tipo `SoundList` descrito anteriormente, cada vez que se reinicie la reproducción, y se cree un nuevo *Reader,* éste será diferente.

Sería posible hacer que la clase `SoundList` creara `MutableReaders` directamente, pero en este caso, se perdería la posibilidad de reiniciar la reproducción de los sonidos que componen el objeto `SoundList`, y sería necesario replicar algo de código. Por estas razones, al final se optó por esta solución, que, además, puede ser útil para otros tipos de sonidos.

5.3 GESTOR DE REPRODUCCIÓN

Muchas aplicaciones precisan de un método que permita controlar cantidades elevadas de sonidos. Además, es común que la lista de sonidos utilizados no sea algo estático, más bien lo contrario, suele variar muy a menudo.

Por ejemplo, un videojuego típico. El juego presenta una música de fondo, gran variedad de efectos sonoros producidos por los diferentes elementos del juego, voces… Además, estos sonidos varían a menudo: La música de fondo cambia, los efectos dependen del estado en que se encuentre el juego, etc.

Es muy común también que se permita al jugador silenciar o cambiar el volumen de estos sonidos, generalmente agrupándolos en categorías.

Para lograr todo esto, en Audaspace se ha implementado este módulo cuyo diagrama de clases se puede ver en la Figura 17

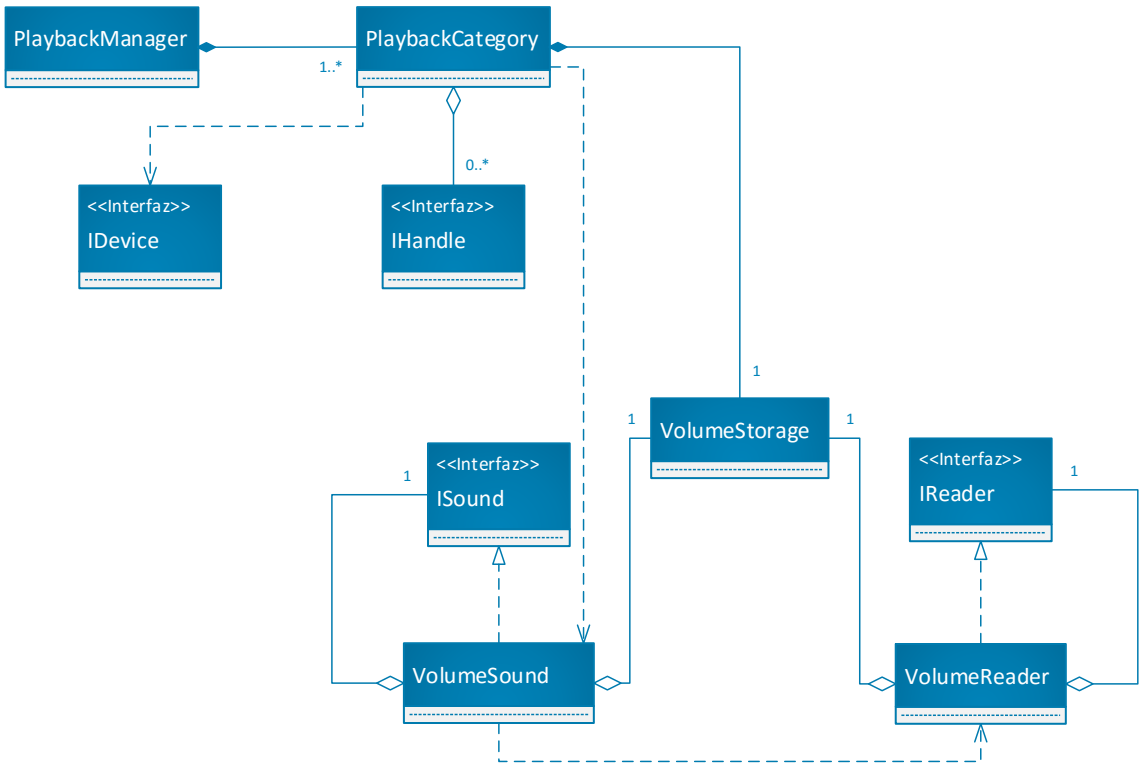

Figura 17: Diagrama de gestor de reproducción

En el diagrama se observa que existe con un `PlaybackManager` que contiene un conjunto de `PlaybackCategories` que, a su vez, contienen un conjunto de `Handles` y usan un tipo especial de sonido, denominado `VolumeSound`.

A continuación, se explicará el porqué de cada una de estas clases.

VolumeStorage

Esta clase es muy sencilla, únicamente contiene un atributo de tipo `float` que representa el volumen de un sonido. Se utiliza en `VolumeSound` y `VolumeReader`, que se describirán a continuación. Su objetivo es servir como un volumen compartido entre varios sonidos.

VolumeSound

Un nuevo tipo de sonido que implementa la interfaz `ISound`. Representa un sonido con volumen compartido. Para conseguir esto, se utiliza el patrón *composite* junto con la clase `VolumeStorage`, que contendrá dicho volumen compartido.

VolumeReader

Es el tipo de *Reader* creado por la clase `VolumeSound`. También utiliza el patrón *composite*, es decir, contiene el *Reader* de otro sonido, cuya salida modifica con el volumen contenido en un objeto del tipo `VolumeStorage`.

PlaybackCategory

Representa una categoría de sonidos. Los sonidos que son asignados a una determinada categoría son convertidos al tipo `VolumeSound`, con un objeto `VolumeStorage` compartido para toda la categoría. Esto es necesario para poder cambiar el volumen de toda la categoría simultáneamente. Además, permite dos grados de control del volumen.

Por ejemplo, dos sonidos de la misma categoría. Cada uno ha de tener un volumen final diferente, pues las fuentes que los emiten están a diferente distancia, pero también es necesario un control de volumen global para la categoría. Con este método, esto sería posible. Se puede variar el volumen de los sonidos de forma independiente con el `Handle` de la reproducción, y a la vez, utilizar el objeto `VolumeStorage` compartido para regular el volumen de la categoría.

Una vez se convierte el sonido al tipo `VolumeSound`, éste será reproducido, almacenando el `Handle` de la reproducción en una lista. Esto plantea el problema de eliminar los `Handles` inválidos, que se van produciendo según las reproducciones que representan finalizan. Para lograr esto se puede asignar a cada `Handle` una clave única, y establecer una función *callback*[13] que sea llamada cuando finalice la reproducción. Esta función *callback* se encargará de limpiar el `Handle` correspondiente.

PlaybackManager

Esta clase se compone de un conjunto de objetos del tipo `PlaybackCategory`. Estas categorías son definidas por el usuario y pueden ser controladas a través de los métodos que proporciona esta clase. Para que una reproducción se pueda manejar a través de esta clase, su sonido debe ser reproducido con el método *play()* de la misma, indicando además un identificado de la categoría a la que ha de pertenecer. Si esa categoría ya existe, el sonido se añadirá, en caso contrario, se creará una categoría nueva. En el Código 6 se puede ver un ejemplo de uso de esta clase.

```
PluginManager::loadPlugins("");
auto factory = DeviceManager::getDefaultDeviceFactory();
auto device = factory->openDevice();
std::shared_ptr<ISound> sound(std::make_shared<File>("sonido.wav"));
PlaybackManager manager(device);
manager.play(sound, 0);
manager.pause(0);
```

[13] **Función *callback***: También llamada función de retrollamada. Consiste en una función que se pasa como argumento a otra función para que sea llamada en algún momento.

Código 6: Uso de PlaybackManager

5.4 MÚSICA DINÁMICA

El objetivo de este módulo es proporcionar una clase que posibilite al usuario la creación de escenas de audio y transiciones entre ellas.

Se puede imaginar de nuevo el ejemplo de un videojuego típico, en el que suena una música de fondo. Cuando el jugador está explorando, la música de fondo será tranquila, sin embargo, si el jugador encuentra enemigos y comienza a luchar, la música ha de cambiar a algo con más tensión. Además, será necesario que la transición se produzca de forma limpia, sin cortes ni saltos extraños.

Para conseguir esto, se ha creado una clase con las relaciones que se muestran en el diagrama de la Figura 18.

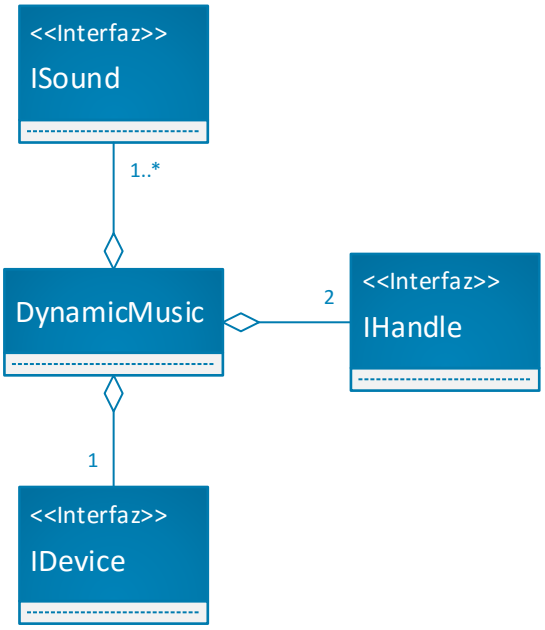

Figura 18: Diagrama DynamicMusic

Se puede ver que la clase `DynamicMusic` agrupa una serie de sonidos que representan las escenas y transiciones, dos `Handles` que permiten controlar lo que se esté reproduciendo en un momento dado (en algunas transiciones se reproducen dos sonidos simultáneamente) y, por último, también se usa un `Device` para la reproducción de las diferentes escenas y transiciones.

La estructura de datos utilizada para los diferentes sonidos es una matriz bidimensional. Esto permite colocar los sonidos de las escenas en la diagonal, mientras que el resto de los elementos son transiciones entre dos escenas determinadas. Tal como se muestra en la Figura 19.

	1	2	3	4
1	Escena 1	-	Transición de ‘1’ a ‘3’	-
2	-	Escena 2	-	-
3	Transición de ‘3’ a ‘1’	-	Escena 3	-
4	-	-	-	Escena 4

Figura 19: Matriz de escenas de sonido

Tras añadir escenas y transiciones con los métodos `addScene()` y `addTransition()`, se podrá cambiar en cualquier momento de una escena a otra con `changeScene()`. Cuando este método se invoca pueden ocurrir dos cosas:

1. **Si se ha definido una transición**: En este caso se espera a que el sonido actual termine de reproducirse, entonces se reproduce el sonido de transición, y, finalmente, la nueva escena. Para lograr esto sin introducir esperas, se hace uso de funciones *callback* que son llamadas cuando finaliza la reproducción en curso.
2. **Si no hay una transición definida**: Si se da este caso, el cambio de escena ocurre sin esperar a que finalice el sonido actual. Para suavizarlo, se introduce un efecto *crossfade*[14] que durará una cantidad configurable de tiempo.

En el Código 7 se puede ver un ejemplo de uso de esta clase. Es importante mencionar que el método `addScene()` devuelve el identificador de la escena que ha de ser usado para acceder a la misma.

```
PluginManager::loadPlugins("");
auto factory = DeviceManager::getDefaultDeviceFactory();
auto device = factory->openDevice();

auto scene1(std::make_shared<File>("scene1.wav"));
auto scene2(std::make_shared<File>("scene2.wav"));
auto transition12(std::make_shared<File>("transition12.wav"));

DynamicMusic manager (device);
int s1 = manager.addScene(scene1);
int s2 = manager.addScene(scene2);
manager.addTransition(s1, s2, transition12);
manager.changeScene(s1);
manager.changeScene(s2);
```

Código 7: Ejemplo de uso de la clase DynamicMusic

[14] ***Crossfade***: Efecto de mezcla de audio que consiste en hacer *fade-out* al sonido que se está reproduciendo a la vez que se hace *fade-in* a un nuevo sonido que empieza a reproducirse.

Una cosa interesante, que es posible con este módulo es utilizar los `MutableSounds` descritos en el apartado 5.2 como escenas. Esto permite que, aunque una escena esté mucho tiempo reproduciéndose, el sonido no resulte monótono para el usuario.

5.5 CLASES DE UTILIDAD

En este módulo se describen dos clases que, aunque fueron creadas para dar soporte a la convolución, son lo suficientemente generales como para usarse para otras cosas en el futuro. En la Figura 20 Se pueden ver estas clases y sus dependencias.

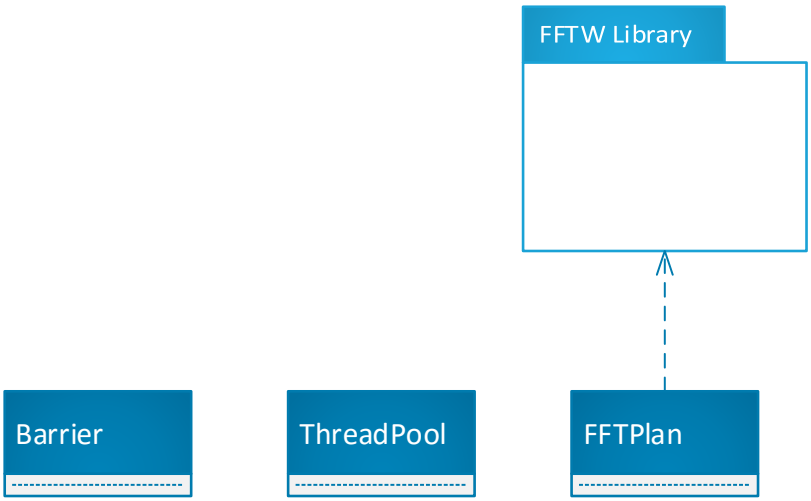

Figura 20: Diagrama clases de utilidad

Barrier

Representa una barrera, es decir, un mecanismo de sincronización de hilos que detiene a todos los que pasen por él hasta que el número de hilos detenidos alcanza un valor configurable.

Esta clase no se utiliza en ninguna parte de la versión final de este proyecto. Fue necesaria para la implementación de uno de los prototipos que se realizaron del módulo de convolución, pero, en posteriores evoluciones del mismo, se utilizó un enfoque más eficiente que hizo que el uso de barreras dejara de ser necesario.

No obstante, la clase se mantiene en la biblioteca, pues se consideró que podría ser útil en el futuro.

ThreadPool

Esta clase implementa una reserva de hilos que puede ser utilizada para paralelizar casi cualquier cosa. En la biblioteca se utiliza para paralelizar la convolución, de forma que aproveche más eficientemente el procesador, pero nada impide a los usuarios asignar otras tareas a esta clase.

Para instanciar esta clase basta con especificar el número de hilos que ha de tener la reserva. En ese momento se crearán los hilos, que quedarán bloqueados hasta que se asignen tareas a realizar.

Para asignar tareas a la reserva de hilos, basta con llamar al método `enqueue()`, proporcionándole como argumentos un puntero a la función que debe ejecutarse y los argumentos que requiera dicha función. Este método es un *template*[15], por lo que es posible asignarle cualquier función, sin importar el tipo de su valor de retorno o argumentos.

Además, el método `enqueue()` devuelve un objeto del tipo `Future`, definido en la biblioteca estándar de C++ 11. Estos objetos pueden usarse tanto para saber si la tarea ha terminado como para recibir su valor de retorno.

En el Código 8 se puede ver un ejemplo sencillo del modo de uso de esta clase.

```
using namespace std;
int task(int a);

auto threadPool(make_shared<ThreadPool>(thread::hardware_concurrency()));
auto future = threadPool->enqueue(task, 2);
future.wait();
return future.get();
```

Código 8: Ejemplo de uso de la clase ThreadPool

Cuando una tarea finaliza, el hilo que la ha ejecutado comienza con una nueva o, si no hay más, se bloquea hasta que se añada otra.

En un principio se planteó crear esta clase siguiendo el patrón *singleton.* No obstante, se identificaron dos inconvenientes. En primer lugar, crear un objeto `ThreadPool` con una cantidad personalizada de hilos, sería menos intuitivo, y, en segundo lugar, usar varias reservas de hilos puede ser útil en el caso de tener varios grupos de tareas que han de ser procesados a la vez, sin tener en cuenta el orden en que han sido añadidos. Esto es debido a que la clase `ThreadPool` utiliza una cola de tareas, por lo que las tareas que se añadan más tarde, también se ejecutarán más tarde.

Finalmente, se desechó la idea del patrón *singleton*, pues las ventajas que ofrecía tampoco eran muy significativas en este caso.

FFTPlan

Esta clase encapsula parte de la funcionalidad de la biblioteca FFTW que se usa para calcular la transformada de Fourier.

[15] ***Templates***: También llamados plantillas. Son una funcionalidad del lenguaje C++. Permiten a una clase operar con tipos genéricos, lo que posibilita que funcione con una gran variedad de tipos sin necesidad de reescribirla.

Permite por tanto el cálculo, tanto de la transformada, como de la transformada inversa de Fourier. A parte del encapsulamiento de una dependencia externa, el otro objetivo que se buscaba con esta clase era una manera de reutilizar los objetos "plan" que ofrece FFTW. Estos objetos pueden crearse rápida o lentamente dependiendo de unos *flags* de configuración, pero cuanto más tarden en instanciarse, más rápidas serán las transformadas de Fourier que los utilicen.

Es por tanto útil crear este objeto con los *flags* de creación lenta durante el periodo de carga, para luego reutilizarlo en gran cantidad de ocasiones.

5.6 CONVOLUCIÓN

El objetivo de este módulo es proporcionar una forma de aplicar convolución a un sonido cualquiera. Además, ésta debe aplicarse según se reproduce el sonido, lo que permitirá aplicar convolución a sonidos extremadamente largos de la misma forma que se aplicaría a otros más cortos.

El esquema de clases que se sigue en este módulo es el que se muestra en la Figura 21.

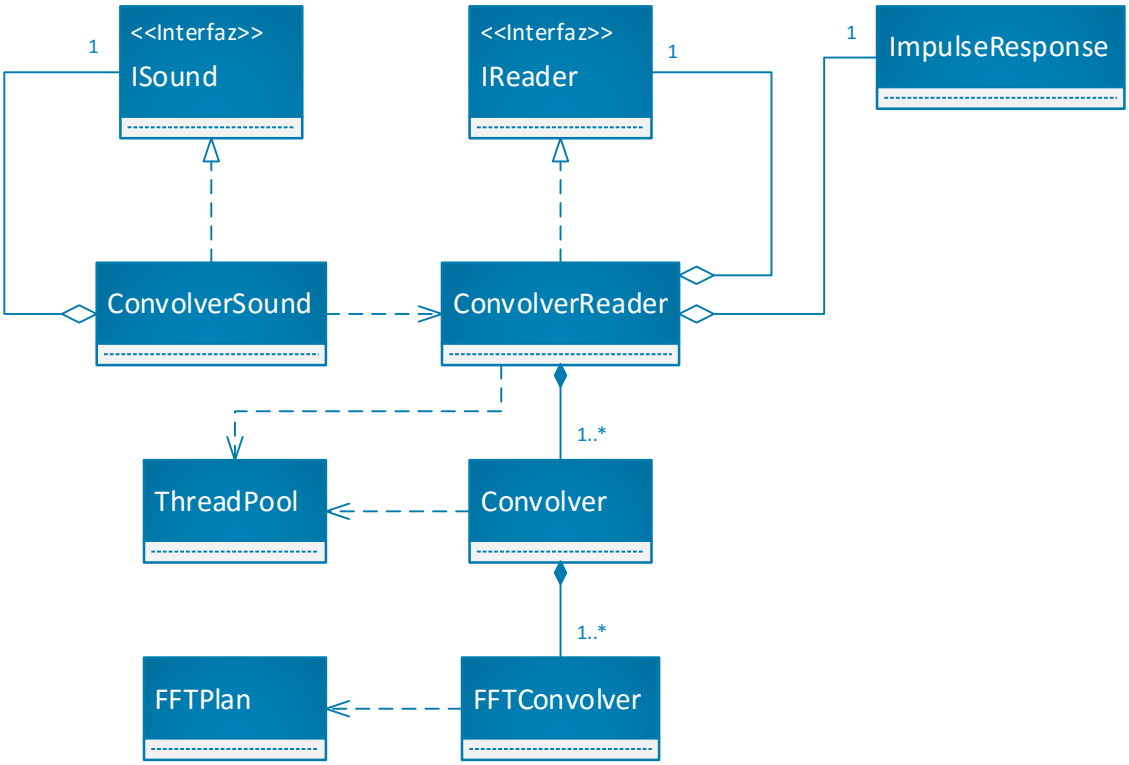

Figura 21: Diagrama del módulo de convolución.

Se puede observar que se vuelve a aplicar el patrón *composite* para crear un nuevo tipo de `Sound` y de `Reader`. Además, se ve que `ConvolverReader` agrega varios objetos de tipo *Convolver* y éstos a su vez agregan varios objetos `FFTConvolver`.

ImpulseResponse

Esta clase representa una respuesta a impulso que es convertida al dominio de la frecuencia y dividida en el momento de la instanciación. Puede utilizarse en cuantas convoluciones como se desee.

ConvolverSound

Nuevo tipo de sonido que una vez más utiliza el patrón *composite*. Su tarea es crear `Readers` que convuelvan un sonido cualquiera con una respuesta a impulso.

ConvolverReader

Reader encargado de aplicar convolución a un sonido. También utiliza el patrón *composite*. Esta clase hace uso de objetos de tipo *Convolver* para realizar dicha operación. Sólo puede ser creado si el sonido al que se ha de aplicar la convolución y la respuesta a impulso tienen el mismo número de canales.

Además, también ha de encargarse tanto de la separación como de la unión de los canales del sonido, ya que se les ha de aplicar la convolución de forma independiente. Un punto importante de esta clase es que paraleliza la convolución de los canales utilizando un objeto `ThreadPool`.

FFTConvolver

Se encarga realizar la convolución utilizando la transformada rápida de Fourier. Para ello, utiliza un objeto `FFTPlan`. Puede utilizarse independientemente para respuestas a impulso muy cortas, pero, normalmente, es usada por objetos de tipo *Convolver*, ya que implementan un modo más eficaz de realizar la convolución.

Convolver

Esta clase se utiliza para realizar la convolución de un sonido con una respuesta a impulso de cualquier longitud. Utiliza objetos `FFTConvolver` para realizar las diferentes convoluciones parciales que requiere el algoritmo que se ha utilizado y que se detallará en el apartado 5.6.1.

También utiliza un objeto `ThreadPool`. Es usado para paralelizar las numerosas convoluciones parciales que se han de realizar, aprovechando así las capacidades de los procesadores modernos que cuentan con varios núcleos.

A continuación, se va a describir con detalle el método de convolución utilizado y cómo se llegó hasta él.

5.6.1 Método de convolución.

Como primer prototipo, se implementó el método *overlap-add* tal cual está descrito en el punto 3.1.1.2, pero su rendimiento con respuestas a impulso de longitud moderada no era para nada bueno, lo que hizo este prototipo inviable.

A continuación, se describirán los diferentes pasos que fueron optimizando el algoritmo hasta su versión final.

5.6.1.1 División en partes de la respuesta a impulso

El siguiente paso implicó dividir la respuesta a impulso utilizada en varias partes, de la misma forma que se divide el sonido en el método *overlap-add*. Esto ayudó de dos maneras:

- En primer lugar, hizo posible reducir el tamaño de las partes del sonido cuando se usaban respuestas a impulso largas, lo que reduce el tiempo de procesamiento necesario para completar la convolución de una parte. Hay que tener en cuenta que deben de tenerse listas un número determinado de muestras cada cierto tiempo. Si el tamaño de parte es muy grande, ese tiempo aumentará y el oyente percibirá discontinuidades en el sonido.
- También permitió distribuir las tareas de convolución de forma más uniforme en el tiempo. En Audaspace, el `Device` encargado de la reproducción de un sonido solicita un número de muestras al Reader cada cierto tiempo. Con el anterior método, todas las operaciones que implicaba la convolución de una parte, se realizaban en el momento de la solicitud. Sin embargo, si se divide también en partes la respuesta a impulso, es posible realizar solo una parte de esas operaciones para responder. El resto serán necesarias para la siguiente solicitud y por tanto podrán ser realizadas en un hilo a parte en el intervalo de tiempo que transcurre entre peticiones.

  En la Figura 22 se puede observar un ejemplo numérico muy sencillo de lo descrito anteriormente.

Sonido	1	2	3	4	5	6	7	8	*	11	12	13	14	15	16	17	18	Filtro
11	34	24																|1|2|*|11|12|
		13	40	28														|1|2|*|13|14|
				15	46	32												|1|2|*|15|16|
						17	52	36										|1|2|*|17|18|
		33	80	48														|3|4|*|11|12|
				39	94	56												|3|4|*|13|14|
						45	108	64										|3|4|*|15|16|
+								51	122	72								|3|4|*|17|18|
				55	126	72												|5|6|*|11|12|
						65	148	84										|5|6|*|13|14|
								75	170	96								|5|6|*|15|16|
										85	192	108						|5|6|*|17|18|
						77	172	96										|7|8|*|11|12|
								91	202	112								|7|8|*|13|14|
										105	232	128						|7|8|*|15|16|
												119	262	144				|7|8|*|17|18|
11	34	70	120	185	266	364	480	497	494	470	424	355	262	144				

Figura 22: Convolución con respuesta a impulso dividida

Se puede observar que, para cada dos números de la salida, solo hace falta la primera convolución parcial de la parte actual y las convoluciones de las partes anteriores.

5.6.1.2 Pre-procesamiento de la respuesta de impulso

Este fue el origen de la clase `ImpulseResponse` que se muestra en el diagrama de la Figura 21.

Consiste en crear un objeto que contenga la respuesta a impulso en el dominio de la frecuencia y dividida en partes. De esta forma, el procesamiento puede realizarse una única vez, y el objeto resultante puede utilizarse para convolver varios sonidos.

5.6.1.3 Paralelización

La siguiente optimización consiste en paralelizar al máximo lo anterior. Es posible hacer esto en dos lugares:

- El primer punto es en los canales del sonido. Es posible paralelizar la convolución de cada canal, puesto que se ha de realizar independientemente. Si hay procesadores suficientes, se puede casi duplicar el rendimiento en un sonido estéreo.
- El otro punto es en las operaciones que se realizan entre las peticiones. Como se puede ver en la Figura 22, todas esas convoluciones parciales son independientes, por lo que pueden ser paralelizadas y su resultado combinado al final de forma adecuada.

Estos cambios obligaron a implementar mecanismos de control de hilos que al final condujeron a la creación de la clase `ThreadPool` descrita en el apartado 5.5.

5.6.1.4 Reducción de FFTs

Hasta ahora el mayor punto de trabajo son las transformadas de Fourier, puesto que se realizan dos por convolución parcial. Una normal, y una inversa. Por suerte es posible reducir esto.

El primer paso es obvio, basta con guardar el resultado de la transformada de Fourier de la parte del sonido que se usa como entrada, en vez de recalcularla para cada convolución parcial.

El segundo paso no es tan trivial, consiste en realizar la acumulación de resultados parciales en el dominio de la frecuencia, para solo tener que hacer una transformación inversa al final. En la Figura 23 se puede ver un esquema de estos cambios [15].

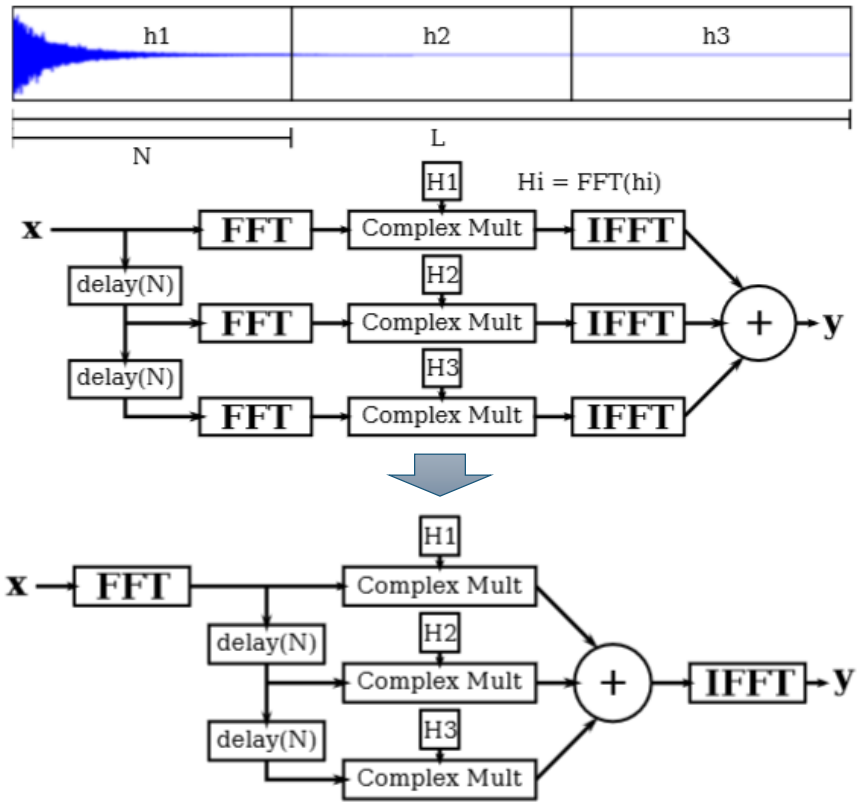

Figura 23: Reducción del número de FFTs usadas para la convolución

Para lograr reducir el número de IFFTs, tal como se muestra en la Figura 23 también fue necesario realizar otros cambios.

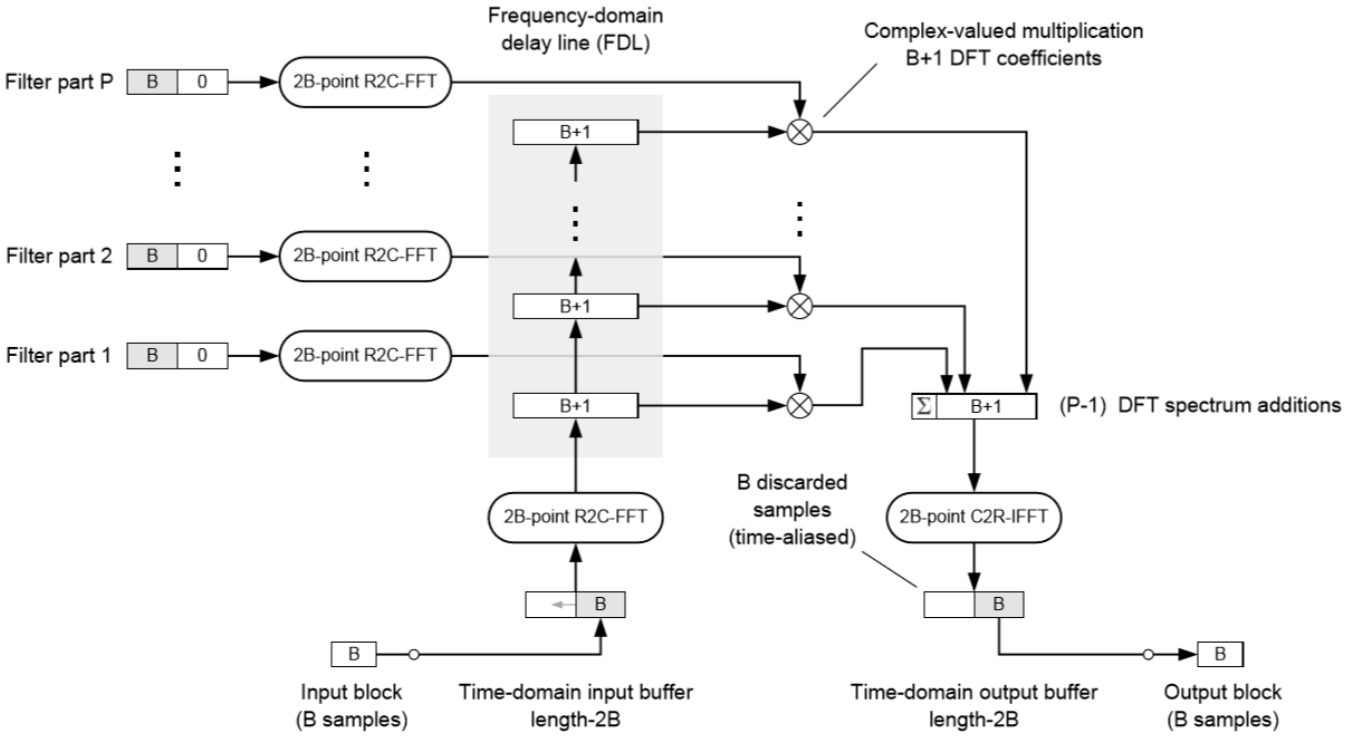

Figura 24: Esquema algoritmo de convolución final

En la Figura 24 se observa un esquema del algoritmo final que se utilizó [16]. El procedimiento es el siguiente:

1. El buffer de entrada actúa como una ventana deslizante de longitud $2B$. Siendo B el tamaño de las partes del sonido. Con cada bloque de entrada, la mitad derecha de buffer será pasada a la izquierda, y la nueva entrada será copiada en la mitad derecha.
2. Se desplaza el *frequency-domain delay line* (FDL) una posición hacia arriba. El FDL no es más que una cola con longitud igual al número de partes de la respuesta a impulso.
3. Se calcula la FFT del buffer de entrada usando como tamaño $2B$ y se introduce el resultado, que tendrá $B + 1$ números complejos, en la primera posición del FDL.
4. Se multiplica punto a punto cada parte del filtro con el contenido de la posición del FDL que les corresponda.
5. Se acumulan los resultados de la multiplicación anterior en un buffer.
6. Se calcula la IFFT del buffer donde se realizó la acumulación. El resultado tendrá longitud $2B$ y se descartará la mitad izquierda del mismo, mientras que la mitad derecha se devuelve como salida.

El resultado es un algoritmo cuyo rendimiento ha mejorado enormemente desde la primera iteración. Aún es posible mejorarlo más, pero se consideró que estos resultados eran suficientemente buenos, por lo que se decidió dejarlo aquí, al menos por el momento.

5.7 AUDIO BIAURAL

Esta funcionalidad depende completamente de la convolución descrita en el apartado anterior. Y su arquitectura se parece mucho también a la del módulo de convolución. Tal como se observa en la Figura 25.

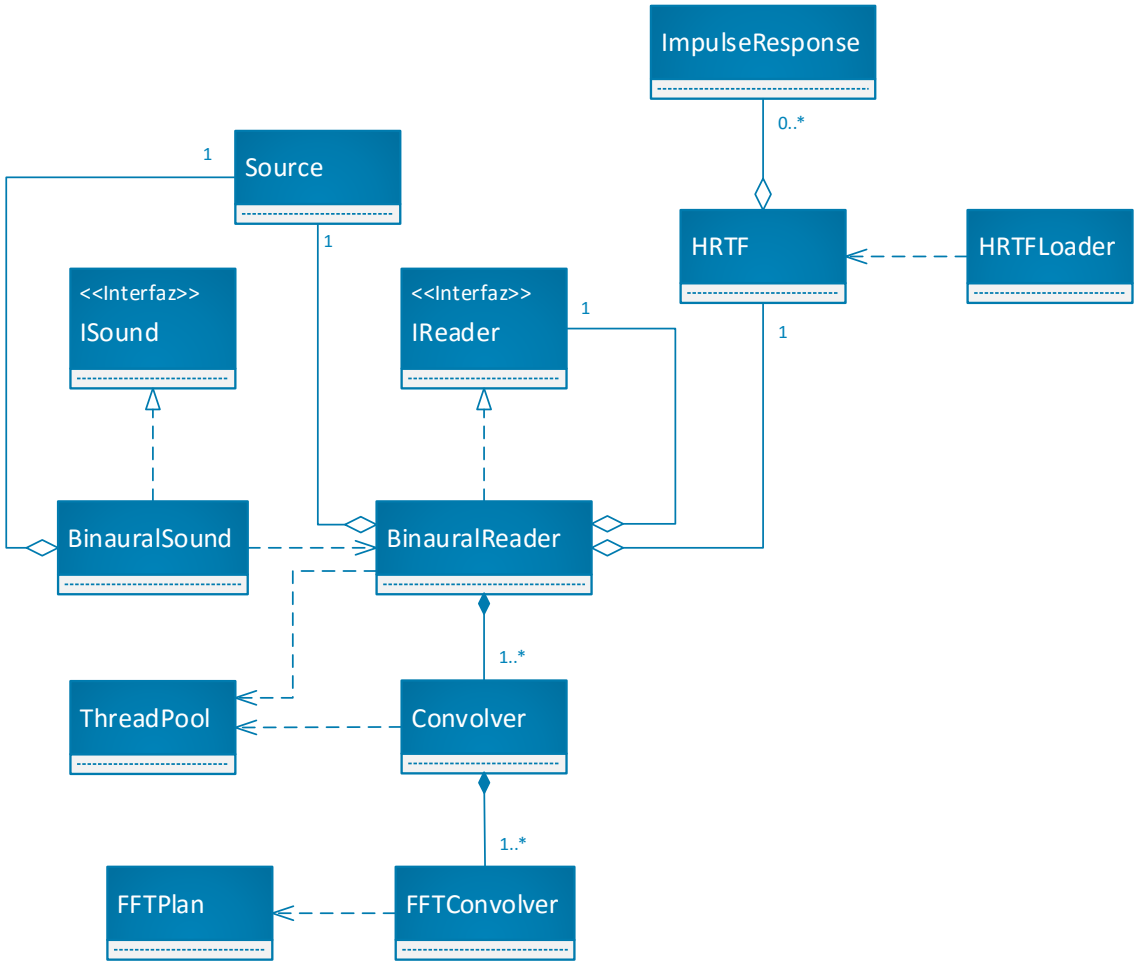

Figura 25: Diagrama del módulo de audio biaural

Como se ve en el diagrama, las clases `Convolver` y `FFTConvolver` donde se implementa el procedimiento de convolución siguen estando presentes. Aunque ahora son usadas por un nuevo `Sound` y `Reader`.

También se pueden ver dos nuevas clases: `Source` y `HRTF`, que son usadas por `BinauralReader`.

Source

Una clase muy sencilla que contiene tres valores: Azimuth[16], elevación y distancia. Los dos primeros son ángulos, mientras que el tercero es un valor en el intervalo $[0, 1]$ que representa la distancia. Siendo 0 el valor más cercano y 1 el más alejado posible.

Representa la dirección desde la que suena un sonido y la distancia a la que lo hace. Al utilizarlo junto a un objeto `BinauralReader`, permite al usuario variar la posición del sonido mientras éste se está reproduciendo.

HRTF

Esta clase representa un conjunto de HRTFs. Para su implementación se utilizan objetos `ImpulseResponse`, conteniendo cada uno una HRTF ya procesada.

Cada objeto `ImpulseResponse` viene identificado por un ángulo de azimuth y un ángulo de elevación, que son definidos cuando se carga. Además, esta clase ofrece un método que devuelve el par de objetos `ImpulseResponse` (uno para cada oreja), que más se acerquen al valor de azimuth y elevación que se solicita.

Por ejemplo, si se solicitan los HRTFs para un `Source` con un azimuth de 20 y una elevación de 10, la clase nos devolverá dos objetos `ImpulseResponse`. Uno marcado con azimuth 20 y elevación 10, para la oreja derecha, y otro marcado con azimuth 340 y elevación 10, para la oreja izquierda. O los más cercanos si esos valores exactos no estuvieran disponibles.

La razón de la diferencia de azimuth entre ambos objetos `ImpulseResponse` es porque nuestras orejas son opuestas la una a la otra.

HRTFLoader

Es una clase auxiliar que permite cargar todos los HRTFs de un directorio, siempre y cuando estos sigan un determinado formato en su nombre, que ha de especificar la oreja para la que han sido grabados y el ángulo de elevación y azimuth que representan. Dado que C++ no ofrece una forma de explorar directorios que sea independiente del sistema operativo, esta clase está implementada de dos formas diferentes. Una para Windows y otra para Linux.

Todos los HRTFs encontrados serán cargados en un nuevo objeto del tipo `HRTF` que será devuelto.

BinauralSound

De nuevo se implementa el patrón *composite* en un sonido. De la misma forma que todas las clases que implementan `ISound` su función es crear `Readers`. En este caso creará un objeto `BinauralReader` que podrá aplicar este efecto a un sonido cualquiera.

[16] **Azimuth:** Dirección horizontal expresada como la distancia angular entre la dirección a un punto fijo y la dirección a el objeto.

BinauralReader

También sigue el patrón *composite*. Funciona de forma muy similar a la clase `ConvolverReader`, pero con algunas diferencias.

- En primer lugar, solo acepta como entrada sonidos con un solo canal que se irán convirtiendo en sonidos con dos canales según son procesados.
- El filtro utilizado para la convolución no es constante, sino que va cambiando según varían los valores contenidos en el objeto `Source`. Para conseguir filtros actualizados, se solicitan al objeto `HRTF` que contiene esta clase.
- Se realizan siempre un mínimo de dos convoluciones simultaneas, una por cada canal de salida. No obstante, en algunas situaciones es necesario realizar cuatro. Esto es debido a que normalmente no se disponen de tantos HRTFs como para que las variaciones entre ellos sean lo suficientemente pequeñas. Lo más normal son variaciones de entre cinco y diez grados entre HRTFs contiguos.

  En esta situación, el usuario oiría pequeños chasquidos según se cambian los HRTFs utilizados al variar la posición del sonido. Para solucionar esto, se realiza una pequeña interpolación entre el sonido que se estaba emitiendo y el que se va a emitir después del cambio de posición. Esto se consigue haciendo cuatro convoluciones simultaneas durante un corto periodo de tiempo. Dos para la posición anterior y dos para la nueva.

A continuación, en el Código 9, se muestra un pequeño ejemplo de cómo reproducir y cambiar la posición de un sonido biaural.

```
using namespace std;
PluginManager::loadPlugins("");
auto factory = DeviceManager::getDefaultDeviceFactory();
auto device = factory->openDevice();
auto plan(make_shared<FFTPlan>(2048, true));
auto tPool(make_shared<ThreadPool>(thread::hardware_concurrency()));

auto hrtfs = HRTFLoader::loadRightHRTFs(fftPlan, ".wav", "hrtfs");
auto source = make_shared<Source>(0, 0, 0);
auto sound = make_shared<File>("sonido.wav");
auto binaural=make_shared<BinauralSound>(sound,hrtfs,source,tPool,plan);

auto handle = device->play(binaural);
source->setAzimuth(5);
source->setElevation(10);
source->setDistance(0.5);
```

Código 9: Ejemplo de uso de sonido biaural:

5.8 *LANGUAGE BINDINGS*

En Audaspace se han implementado *language bindings* a otros dos lenguajes aparte de C++: C y Python.

Tener una API definida en C, ayuda bastante a la hora de crear *language bindings* a otros lenguajes, pues puede actuar de punto de unión con ellos. Además, por suerte, dadas las similitudes de C y C++, ha sido muy sencilla de implementar.

Puesto que C no es un lenguaje orientado a objetos, toda la funcionalidad está disponible a través de funciones donde el primer argumento es siempre el objeto.

En el caso de Python, el puente que hay que realizar es algo más complicado que en el caso anterior, pero tampoco presenta gran dificultad. En este caso sí que se utilizan clases con sus respectivas propiedades y métodos. Para utilizar Audaspace desde Python, basta con utilizar `import aud`.

Ambas API exponen gran parte de la funcionalidad de Audaspace a sus respectivos lenguajes, pero fueron diseñados priorizando la sencillez. Por esta razón, para acceder a toda la versatilidad de la biblioteca, sigue siendo necesario utilizar C++.

6 USANDO AUDASPACE

Para mostrar la facilidad de uso de Audaspace, la biblioteca ha sido usada para proporcionar audio a un videojuego. Dicho videojuego fue creado por mi como demostración de una biblioteca que escribí hace algunos años.

Puesto que su objetivo principal no era el audio, el sonido fue un aspecto que se dejó bastante de lado. A continuación, se describirá el proceso de integración de Audaspace y las mejoras que supuso.

6.1 INTEGRACIÓN EN UN JUEGO

Antes de la integración con Audaspace, el videojuego solamente reproducía sonidos y permitía variar su volumen. Existían dos categorías de sonidos, música y efecto, pero eran gestionadas dentro del código del propio juego.

El primer paso consistió, obviamente en cambiar todos los tipos de datos e inicializaciones de sonidos a la nueva API. Inicialmente se utilizaba un esquema con identificadores de sonidos, de forma que dichos identificadores pudieran usarse para reproducir un sonido. A fin de realizar el mínimo número de cambios, tal comportamiento se emuló mediante un objeto de la clase `unordered_map` de la biblioteca estándar de C++ 11, donde se añadían los diferentes sonidos junto con una clave identificadora.

Otro cambio importante fue el eliminar el código encargado del control de categorías de sonidos para utilizar en su lugar la clase `PlaybackManager` proporcionada por Audaspace y descrita en el apartado 5.3. Este cambio ayudó a eliminar algo de complejidad del código del juego y permitió mantener la funcionalidad.

También se decidió aprovechar la funcionalidad de música dinámica proporcionada por Audaspace (apartado 5.4). Puesto que el videojuego es muy simple, solo se crearon dos escenas de sonido, una para la exploración, y otra para el combate. De esta forma, la música de fondo que suena en cada una de estas situaciones es diferente.

Para implementar esto ha sido necesario añadir un mínimo de código extra al juego pues, anteriormente, este requisito no fue contemplado. Los cambios fueron muy sencillos y consistieron en la creación de un nuevo tipo de mensaje especificando una escena. Es decir, cuando una entidad del juego detecta que ha ocurrido algo, en este caso un combate, notifica al controlador para que cambie la escena de sonido. El cambio de escena se realiza invocando un único método, siendo las transiciones manejadas por Audaspace.

Finalmente, otra nueva funcionalidad que se implementó fue el uso de sonidos mutables. Anteriormente, solo se usaba un sonido para, por ejemplo, los pasos de los personajes. Esto hacía que dichos sonidos se acabaran volviendo repetitivos tras muy poco tiempo.

Para la implementación de esta funcionalidad, no fue necesario cambiar nada del código base del videojuego. Simplemente se alteró la inicialización de los sonidos para que fueran instanciados como objetos de la clase `SoundList`, comentada en el apartado 5.2. Pese a esto, el modo de uso no cambia en absoluto. En Audaspace, gracias a que cualquier tipo de sonido, por especial que sea, implementa la interfaz `ISound`, no hay diferencia alguna entre el modo de reproducción de unos y otros.

Gracias a la sencillez de la API de Audaspace, todos estos cambios llevaron una cantidad mínima de tiempo. Hay que mencionar, que una de las grandes características desarrolladas en este proyecto, el sonido biaural, no ha sido implementada en el videojuego. Esto es debido a que se trata de un videojuego con vista aérea, por lo que la implementación de esta característica no tiene mucho sentido. El sonido biaural está pensado sobre todo para experiencias más inmersivas: Perspectiva de primera persona, realidad virtual, etc. No obstante, para demostrar que es posible hacerlo, se ha creado un nivel especial que muestra en funcionamiento dicha característica.

6.1.1 Nivel especial

Este nivel especial puede ser accedido desde un botón presente en el menú principal (Figura 26).

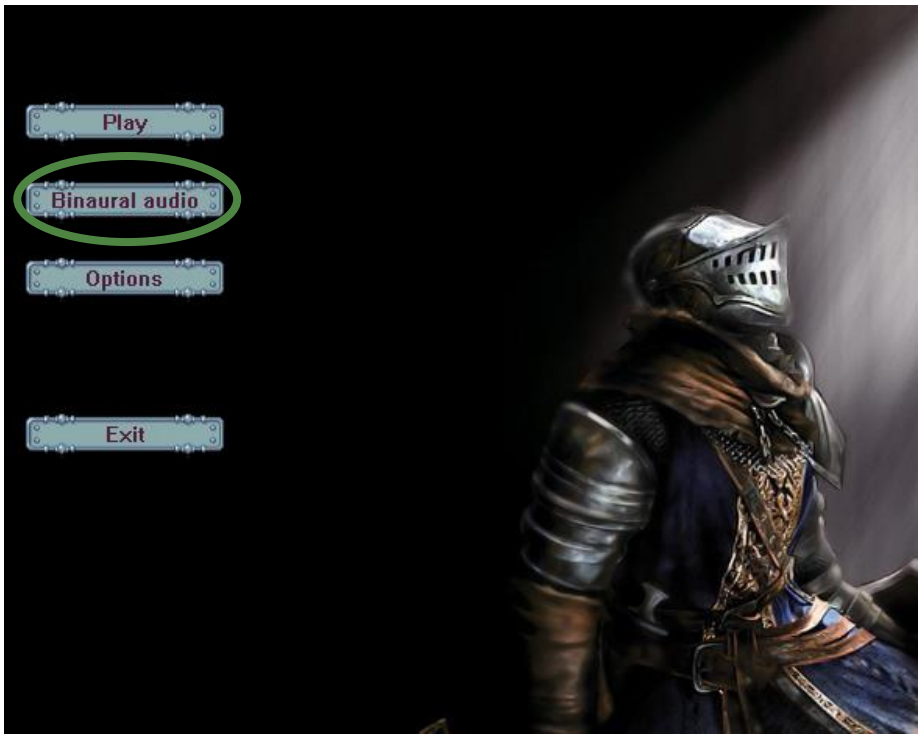

Figura 26: Menú principal del juego de muestra

Este nivel es algo diferente a lo habitual. El sonido será emitido por los movimiento y acciones del jugador, y se mostrará como si lo oyera el personaje situado en medio de la sala, tal como se puede ver en la Figura 27. Además, utilizando la convolución, también se aplicará un efecto de reverberación acorde a la sala.

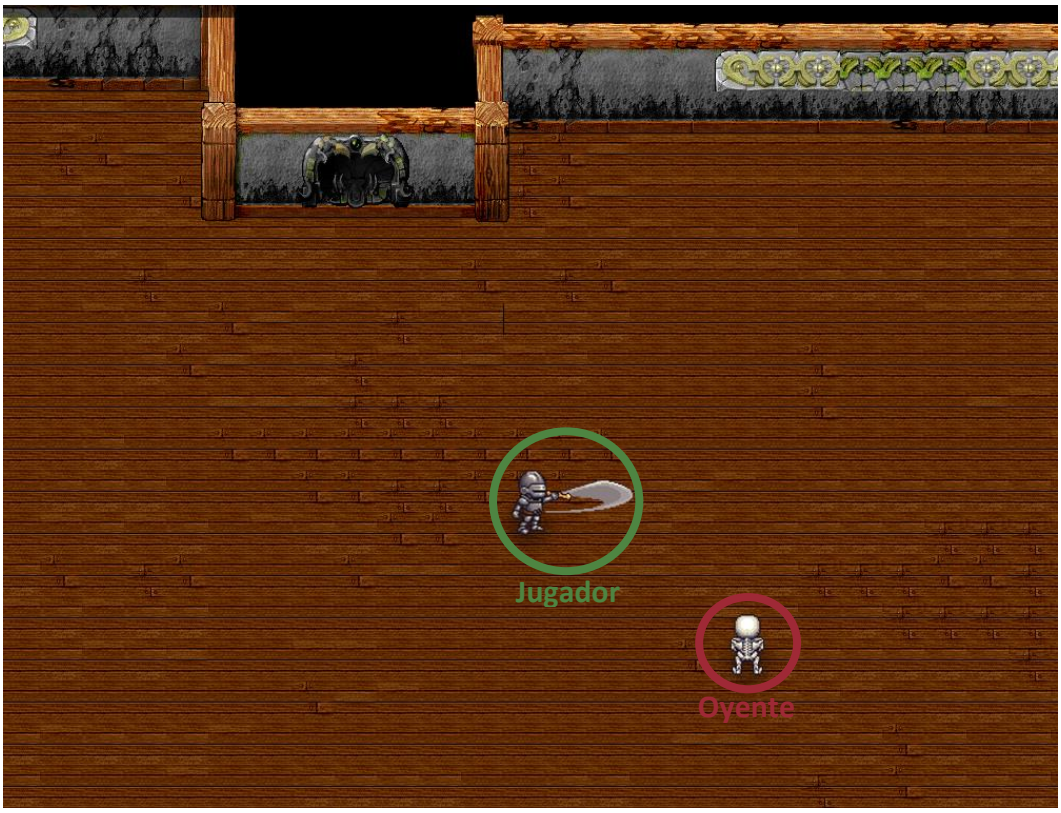

Figura 27: Nivel de demostración de sonido biaural

Para hacer todo esto posible, fue necesario hacer algunos cambios a las entidades del juego. El objetivo es que las entidades que emiten sonidos, puedan variar la posición de origen de los mismos en base al lugar en el que está situado el oyente. Esto se consigue mediante un nuevo mensaje de notificación de posición por parte del oyente, y un procesamiento de dicho mensaje por parte de los receptores.

A parte de lo descrito en el párrafo anterior, solo es necesario cambiar la inicialización de los sonidos, instanciando objetos que apliquen los efectos correspondientes al igual que se hizo en el caso de los sonidos mutables.

6.2 SCRIPT DE SONIDO

A parte de lo anterior, se ha escrito también un script en Python para demostrar parte de la API de Audaspace en este lenguaje. Este script se centrará en mostrar el sonido biaural y la reverberación.

La finalidad del script es múltiple y se puede resumir en los siguientes puntos:

- Mostrar una escena inmersiva en la que el oyente pueda situarse dentro del entorno que escucha. Para ello se usarán unos HRTFs que fueron grabados utilizando una cabeza maniquí KEMAR [17].
- Mostrar el rendimiento del sistema de convolución. En el script, a parte de las tareas de mezcla y reproducción de sonidos, también se realizarán hasta 40 convoluciones simultaneas. Todo esto puede realizarse en tiempo real y sin que se aprecie ninguna ralentización. Incluso en un procesador de dos núcleos a 2GHz solo se observa un pico de un 10% de uso de la CPU.
- Volver a demostrar, esta vez de forma más visible, la simplicidad del API. Como se observa en el script, además de ser bastante corto, la gran mayoría de las líneas de código son usadas para cambiar de posición los diferentes sonidos.
- Servir como demostración de uso de Audaspace en Python.

También se proporciona el código en C++ de un programa que hace exactamente lo mismo que el script anterior para poder comparar ambas API de forma sencilla.

6.3 COROLARIO

Teniendo los dos casos anteriores en mente, se puede asegurar que Audaspace no es para nada difícil de utilizar, y que puede integrarse con mucha facilidad en proyectos ya existentes.

Esta es una gran ventaja, pues es una de las características que más se buscan, junto al coste y a las funcionalidades, a la hora de elegir una biblioteca u otra.

En lo que respecta al coste es muy difícil competir con Audaspace, pues se trata de un proyecto de código abierto, con lo que presenta coste cero y posibilidad de acceder y modificar libremente el código fuente de la biblioteca.

En cuanto a las funcionalidades, en el apartado 7 se realizarán comparaciones entre Audaspace y varias otras bibliotecas de su sector. En estas comparaciones, se mostrarán las diferentes ventajas e inconvenientes que se han derivado del uso de cada una.

7 COMPARACIÓN Y BENCHMARKING

En este capítulo se realizarán diferentes pruebas y comparaciones con otras bibliotecas similares que puedan considerarse competidoras de Audaspace; detallando en que puntos es mejor una u otra y explicando los motivos. Es importante mencionar que, cuando en las tablas se produce un empate, no siempre es un empate exacto. A veces, en una característica, hay ciertas diferencias que se consideran compensadas entre las dos bibliotecas que se comparan.

En cuanto al código de colores:

- **Rojo**: Significa que la característica no está presente o es muy deficiente en comparación a la alternativa.
- **Amarillo**: Significa que la característica está presente, pero le falta algo que la otra opción tiene. Este caso suele ir acompañado de una pequeña descripción.
- **Verde**: Significa que la característica está presente y no hay deficiencias significativas en comparación con la otra biblioteca.

Para la comparación se han elegido diferentes bibliotecas, tanto de código abierto como comerciales. El criterio principal a la hora de escogerlas ha sido la similitud o no similitud con Audaspace en términos de objetivos. Es decir, bibliotecas con las que Audaspace deba competir a la hora de ser elegida por sus potenciales usuarios. En las páginas siguientes se realizarán comparaciones con RtAudio, SndObj, CLAM, Cricket, IrrKlang, BASS y FMOD.

En cuanto a los puntos comparados en cada biblioteca, estos han sido elegidos respondiendo a diferentes características, básicas y avanzadas, que se consideran útiles o incluso necesarias en una biblioteca de audio de alto nivel.

7.1 BIBLIOTECAS DE CÓDIGO ABIERTO

En cuanto a bibliotecas de código abierto, no hay muchas que se puedan considerar competidoras de Audaspace. La mayoría son bibliotecas de bajo nivel que, en varios casos, Audaspace utiliza como dependencias que la ayuden a implementar su funcionalidad. Ejemplos de esto son SDL, Open Al (*software*) y FFMPEG, entre otras.

Por estos motivos, se podría decir que Audaspace está bien posicionada para competir con otras librerías de código abierto de su mismo ámbito. No obstante, a continuación, se presentarán varias comparaciones con otras librerías que, en cierto modo, se asemejan a Audaspace.

7.1.1 RtAudio

RtAudio es una biblioteca de código abierto creada y mantenida por Gary P. Scavone, de la universidad de McGill. Puede ser obtenida en: http://www.music.mcgill.ca/~gary/rtaudio/.

En primer lugar, se han escrito programas de ejemplo con RtAudio para ver su funcionamiento y probar sus características, pero rápidamente fue visible que la funcionalidad de esta librería era muy limitada; solamente proporcionando un modo uniforme de reproducir y grabar audio en varios sistemas operativos.

En la Figura 28 se puede ver una tabla más detalla con la funcionalidad de ambas bibliotecas y los factores en los que una es mejor que la otra.

	Audaspace	RtAudio	Descripción
Dispositivos de entrada	Sí	Sí	
Dispositivos de salida	Sí	Sí	
Audio 3D	Sí	No	
Lectura de ficheros	Sí	No	
Escritura de ficheros	Sí	No	
***Streaming* desde internet**	Sí	No	
Secuenciación de sonidos	Sí	No	
Re-especificación de sonidos	Sí	No	
Generación de ondas	Sí	No	
Filtros y efectos	Sí	No	
Superposición de efectos	Sí	No	
Control de reproducciones avanzado	Sí	No	
Convolución	Sí	No	
Audio biaural	Sí	No	
Sencillez de la API	Muy sencilla	Normal	
Proporciona herramientas	No	No	

	Audaspace	RtAudio	Descripción
Fácilmente extensible	Sí	No	RtAudio no está pensada para ello, pero el hecho de ser de código abierto ayuda un poco.
Documentación	Programas de ejemplo con explicaciones y documentación de clases generada con DoxyGen	Programas de ejemplo con explicaciones y documentación de clases	La documentación de ambas es suficiente, pero no exhaustiva. Es un aspecto que se puede mejorar.
Lenguajes	C++, C y Python	C++	Ambas librerías están implementadas en C++, pero RtAudio no proporciona *language bindings* a otros lenguajes.
Plataformas	Windows, Mac y Linux	Windows, Mac y Linux	
Licencia	Apache License	Licencia propia que permite su uso para cualquier fin.	Ambas licencias permiten el uso para cualquier fin.

Figura 28: Comparación con RtAudio

En cuanto a la sencillez de la API, se puede ver que la ofrecida por RtAudio, es bastante menos cómoda para trabajar. Comparando las funcionalidades que ofrecen tanto Audaspace como RtAudio, se puede observar que la forma de reproducir sonidos es mucho más rápida y sencilla en Audaspace.

Mientras que en Audaspace basta con invocar un método, en RtAudio es necesario definir una función *callback* que escriba a un *stream* los datos del sonido. El método de RtAudio implica, por tanto, bastante más trabajo a realizar por el usuario que el esquema de `Readers` usado por Audaspace.

7.1.2 SndObj

SndObj puede encontrarse en: http://sndobj.sourceforge.net/. Es, al igual que en el caso anterior, una biblioteca de código abierto, y fue creada por Victor Lazzarini.

A fecha de la creación de este documento no ha sido posible obtener la documentación de la biblioteca, pues, aunque se ofrecen enlaces en la web citada más arriba, los ficheros de documentación no están disponibles.

Del mismo modo que en el caso anterior, se ha examinado la biblioteca para ver qué características posee y compararlas con Audaspace. A continuación, en la Figura 29, se presenta una tabla de comparación.

	Audaspace	SndObj	Descripción
Dispositivos de entrada	Sí	Sí	
Dispositivos de salida	Sí	Sí	
Audio 3D	Sí	No	
Lectura de ficheros	Sí	Sí, pero admite muchos menos formatos que Audaspace	SndObj solo admite WAVE, AIFF y MIDI.
Escritura de ficheros	Sí	Sí, pero admite muchos menos formatos que Audaspace	SndObj solo admite WAVE, AIFF y MIDI.
***Streaming* desde internet**	Sí	No	
Secuenciación de sonidos	Sí	No	
Re-especificación de sonidos	Sí	No	
Generación de ondas	Sí	Sí	
Filtros y efectos	Sí	Sí	
Superposición de efectos	Sí	Sí	
Control de reproducciones avanzado	Sí	No	
Convolución	Sí	Sí, pero no divide la respuesta a impulso en partes	El no dividir la respuesta a impulso en partes dificulta la convolución con respuestas a impulso de tamaño medio y grande.
Audio biaural	Sí	No	
Sencillez de la API	Muy sencilla	Sencilla	
Proporciona herramientas	No	No	

	Audaspace	SndObj	Descripción
Fácilmente extensible	Sí	Sí	
Documentación	Programas de ejemplo con explicaciones y documentación de clases generada con DoxyGen	No disponible	La web de SndObj ofrece enlaces a la documentación, pero los ficheros han desaparecido.
Lenguajes	C++, C y Python	C++, Python y Java	Ambas librerías están escritas en C++ y ofrecen una API en Python, pero SndObj proporciona una API en Java y no en C.
Plataformas	Windows, Mac y Linux	Windows, Mac y Linux	
Licencia	Apache License	GNU Public License.	

Figura 29: Comparación con SndObj

El funcionamiento de SndObj se basa en conectar objetos de sonido, concepto similar al que utiliza Audaspace con los *Readers*. No obstante, su API hace más complicado el control de reproducciones de forma independiente a los sonidos.

7.1.3 CLAM

CLAM es una biblioteca de audio de código abierto creada por la universidad Pompeu Fabra de Barcelona. La web del proyecto es: http://clam-project.org/.

Aunque se trata de una librería muy elaborada que, además, ofrece diversas herramientas, lleva sin actualizarse desde hace 5 años y tiene problemas con parte de la documentación, que no está accesible.

Pese a esto, es una opción muy completa que merece la pena explorar. En la Figura 30 se puede ver una comparación con Audaspace.

	Audaspace	CLAM	Descripción
Dispositivos de entrada	Sí	Sí	
Dispositivos de salida	Sí	Sí	
Audio 3D	Sí	Sí	
Lectura de ficheros	Sí	Sí	

	Audaspace	CLAM	Descripción
Escritura de ficheros	Sí	Sí	
***Streaming* desde internet**	Sí	No	
Secuenciación de sonidos	Sí	No	
Re-especificación de sonidos	Sí	No	
Generación de ondas	Sí	Sí	
Filtros y efectos	Sí	Sí	
Superposición de efectos	Sí	Sí	
Control de reproducciones avanzado	Sí	No	
Análisis de audio	No	Sí	
Serialización de objetos	No	Sí	
Convolución	Sí	No	
Audio biaural	Sí	No	
Sencillez de la API	Muy sencilla, pero sin herramientas	Compleja, pero proporciona una herramienta	Clam proporciona una herramienta que permite crear el motor de procesamiento de sonido de forma gráfica.
Proporciona herramientas	No	Sí	Varias herramientas de análisis y generación de audio.
Fácilmente extensible	Sí	Medio	Es extensible por ser de código abierto, pero no documenta ningún modo establecido de extensión.

	Audaspace	CLAM	Descripción
Documentación	Programas de ejemplo con explicaciones y documentación de clases generada con DoxyGen	Programas de ejemplo y documentación de clases	Clam tiene una documentación de clases más detallada que Audaspace, pero los enlaces al resto de explicaciones están rotos.
Lenguajes	C++, C y Python	C++	
Plataformas	Windows, Mac y Linux	Windows, Mac y Linux	
Precio	Apache License	GNU Public License.	

Figura 30: Comparación con Clam

Es necesario destacar que, pese a que la API de CLAM no es muy sencilla, sí que ofrece gran flexibilidad y capacidad de configuración.

Además, uno de los grandes atractivos de CLAM, son sus herramientas. La más interesante, en mi opinión, es *NetworkEditor*, que hace posible utilizar CLAM de forma gráfica. Esto permite, por ejemplo, que diseñadores de audio puedan implementar el sistema que crean conveniente, sin necesitar conocimientos de programación. A parte de esta, también se ofrecen otras herramientas de análisis y generación de audio.

7.2 BIBLIOTECAS GRATUITAS

7.2.1 Cricket

Cricket es una biblioteca de audio multiplataforma de uso gratuito, pero que no es de código abierto. De hecho, la única modalidad de pago que presenta es una licencia que permite acceder al código fuente a cambio de 99$ por desarrollador. Puede localizarse en el siguiente enlace: http://www.crickettechnology.com/

De igual forma que en los casos anteriores, en la Figura 31, se puede ver una tabla comparativa entre Cricket y Audaspace.

	Audaspace	Cricket	Descripción
Dispositivos de entrada	Sí	No	
Dispositivos de salida	Sí	Sí	
Audio 3D	Sí	Sí	
Lectura de ficheros	Sí	Sí	

	Audaspace	Cricket	Descripción
Escritura de ficheros	Sí	No	
***Streaming* desde internet**	Sí	No	
Secuenciación de sonidos	Sí	No	
Re-especificación de sonidos	Sí	No	
Generación de ondas	Sí	No	
Filtros y efectos	Sí	Sí	Aunque implementa menos efectos, En Cricket éstos pueden ser manipulados durante la reproducción. En Audaspace hacer esto es más complicado.
Superposición de efectos	Sí	Sí	
Control de reproducciones avanzado	Sí	No	
Convolución	Sí	No	
Audio biaural	Sí	No	
Sencillez de la API	Muy sencilla	Muy sencilla	
Proporciona herramientas	No	No	
Fácilmente extensible	Sí	Sí	
Documentación	Programas de ejemplo con explicaciones y documentación de clases generada con DoxyGen	Programas de ejemplo con explicaciones y documentación del API	La documentación de Cricket es más completa.
Lenguajes	C++, C y Python	C++, .NET, Java y Objective-C	

	Audaspace	Cricket	Descripción
Plataformas	Windows, Mac y Linux	Windows, Mac, Linux, iOS, Android y Windows Phone 8	
Precio	0€	Entre 0$ y 99$	

Figura 31: Comparación con Cricket

Como se puede ver, Cricket posee un buen abanico de funcionalidades y ofrece una API muy sencilla y fácil de usar. Además, su uso es gratuito, y soporta plataformas móviles, sector al que claramente está enfocada.

7.3 BIBLIOTECAS COMERCIALES

Aquí se comparará Audaspace con algunas bibliotecas comerciales que permiten su uso gratuito para fines no comerciales. Esta es la categoría donde la lucha es más reñida, pero hay que tener en cuenta que algunos de los productos con los que se la compara llevan años en el mercado y ya están muy asentados, mientras que Audaspace comenzó su andadura como biblioteca independiente apenas unos meses antes de la realización de este documento.

Además, muchas de las funcionalidades en las que Audaspace sale perdiendo en estas comparaciones, están planeadas para su implementación en un futuro próximo.

7.3.1 IrrKlang

IrrKlang es una biblioteca de audio de alto nivel creada por Ambiera que puede encontrarse en la siguiente página web: http://www.ambiera.com/irrklang/. Ofrece diferentes precios de licencia, dependiendo del precio que se le quiera poner al producto en el que se va a usar la biblioteca:

- Uso no comercial: **Gratis**
- Precio menor de 20€: **65€**
- Precio menor a 27€: **290€**
- Cualquier precio (incluye soporte prioritario y actualizaciones): **490€**

Además, todas las opciones no gratuitas incluyen la posibilidad de usar IrrKlang como biblioteca estática. Como en casos anteriores, en la Figura 32 se puede observar una tabla comparativa entre Audaspace e IrrKlang.

	Audaspace	IrrKlang	Descripción
Dispositivos de entrada	Sí	Sí	
Dispositivos de salida	Sí	Sí	

	Audaspace	IrrKlang	Descripción
Audio 3D	Sí	Sí	
Lectura de ficheros	Sí	Sí	
Escritura de ficheros	Sí	Sí	
***Streaming* desde internet**	Sí	Sí	
Secuenciación de sonidos	Sí	No	
Re-especificación de sonidos	Sí	No	
Generación de ondas	Sí	Sí	
Filtros y efectos	Sí	Sí	Los efectos de IrrKlang solo están disponibles en Windows, pero pueden ser manipulados durante la reproducción. Algo que Audaspace no puede hacer.
Superposición de efectos	Sí	Sí	
Control de reproducciones avanzado	Sí	No	
Convolución	Sí	No	
Audio biaural	Sí	No	
Sencillez de la API	Muy sencilla	Muy sencilla	
Proporciona herramientas	No	No	
Fácilmente extensible	Sí	Sí	No utiliza *plugins*, pero permite crear nueves efectos y decodificadores

	Audaspace	IrrKlang	Descripción
Documentación	Programas de ejemplo con explicaciones y documentación de clases generada con DoxyGen	Programas de ejemplo con explicaciones y documentación del API	La documentación de IrrKlang es más completa.
Lenguajes	C++, C y Python	C++ y .NET	.NET incluye varios lenguajes (C#, Visual Basic, etc.), por lo que IrrKlang tiene ventaja en este apartado.
Plataformas	Windows, Mac y Linux	Windows, Mac y Linux	
Precio	0€	Entre 0€ y 490€	

Figura 32: Comparación con IrrKlang

IrrKlang es bastante utilizado, gracias tanto a su gran sencillez, como a su bajo precio y variedad de funcionalidades. No obstante, en la tabla superior se puede observar como Audaspace, siendo de código abierto, se sitúa muy cerca, e incluso, tiene algunas funcionalidades adicionales.

7.3.2 BASS

BASS es otra librería comercial muy completa. Ha sido creada por Un4seen Developments y puede localizarse en http://www.un4seen.com/. Las posibilidades de licencia que ofrece son las siguientes:

- Uso no comercial: **Gratis**
- *Shareware*[17] con precio menor a 40€ o no *shareware* con precio inferior a 10€: **125€**
- Un solo producto comercial de cualquier tipo: **950€**
- Un solo producto comercial para iOS o Android: **475€**
- Número ilimitado de productos comerciales: **3.450€**

Todos los precios anteriores son por plataforma, pero las plataformas adicionales tienen un 40% de descuento sobre el precio de la primera. En la tabla de la Figura 33 se puede observar una comparación entre Audaspace y BASS.

[17] ***Shareware:*** Se denomina así al modelo de distribución de *software* en el que el usuario puede probar el producto, por tiempo limitado o con otras restricciones, antes de comprarlo.

	Audaspace	BASS	Descripción
Dispositivos de entrada	Sí	Sí	
Dispositivos de salida	Sí	Sí	
Salidas múltiples simultaneas	No	Sí	
Audio 3D	Sí	Sí	
Lectura de ficheros	Sí	Sí	
Escritura de ficheros	Sí	Sí	
***Streaming* desde internet**	Sí	No	
Secuenciación de sonidos	Sí	No	
Re-especificación de sonidos	Sí	Sí	
Generación de ondas	Sí	No	
Filtros y efectos	Sí	Sí	Los efectos de BASS solo están disponibles en Windows, pero permite manipularlos durante la reproducción.
Superposición de efectos	Sí	Sí	
Control de reproducciones avanzado	Sí	Sí	BASS ofrece funcionalidad más avanzada y permite sincronizar canales entre ellos o con los eventos del software donde se usa
Convolución	Sí	No	
Audio biaural	Sí	No	
Sencillez de la API	Muy sencilla	Normal	
Proporciona herramientas	No	No	

	Audaspace	BASS	Descripción
Fácilmente extensible	Sí	Sí	
Documentación	Programas de ejemplo con explicaciones y documentación de clases generada con DoxyGen	Documentación del API	BASS no ofrece programas de ejemplo, pero la documentación del API es más completa
Lenguajes	C++, C y Python	C++, C, Delphi y Visual Basic. Otros disponibles de forma no oficial	BASS ofrece API en más lenguajes y existen aún más *plugins* no oficiales
Plataformas	Windows, Mac y Linux	Windows, Mac, Linux, Android, Windows Store/Phone, iOS y Linux ARM	
Precio	0€	Entre 0€ y 3.450€	

Figura 33: Comparación con BASS

BASS es una gran biblioteca cargada de funcionalidades muy completas. Su API no es tan sencilla como Audaspace, en parte porque no es orientada a objetos y presenta muchas configuraciones de estructuras. No obstante, esto tampoco la hace extremadamente complicada.

Otro punto muy importante de BASS es su extensibilidad, que ha dado lugar a numerosos *plugins* de terceros. Pese a que estos *plugins* no han sido evaluados en la tabla de la Figura 33, han de ser tenidos en cuenta a la hora de escoger qué biblioteca se quiere utilizar.

7.3.3 FMOD

FMOD es una de las bibliotecas de audio más conocidas y usadas que existen. Se puede decir incluso que se trata del referente del sector. Su página web oficial se encuentra en www.fmod.org.

Las posibilidades de licencia que ofrece son las siguientes, en función del presupuesto con el que se cuente. Además, una licencia solo es válida para un solo producto:

- Uso no comercial: **Gratis**
- Presupuesto inferior a 100.000$: **Gratis con limitación de un producto al año**
- Presupuesto inferior a 100.000$: **500$ primera plataforma y 500$ plataformas adicionales. Sin limitación de productos anuales.**
- Presupuesto inferior a 500.000$: **3.000$ primera plataforma y 1.500$ plataformas adicionales.**

- Presupuesto superior a 500.000$: **15.000$ primera plataforma y 3.000$ plataformas adicionales.**

Además, el uso de los diferentes *plugins* de terceros conlleva licencias adicionales que oscilan entre los 900$ y los 5.000$.

A continuación, en la Figura 34, una tabla comparativa entre FMOD y Audaspace.

	Audaspace	FMOD	Descripción
Dispositivos de entrada	Sí	Sí	
Dispositivos de salida	Sí	Sí	
Múltiples dispositivos simultaneos	No	Sí	
Audio 3D	Sí	Sí	FMOD presenta más posibilidades de configuración para su sonido 3D.
Lectura de ficheros	Sí	Sí	
Escritura de ficheros	Sí	Sí	
***Streaming* desde internet**	Sí	Sí	
Secuenciación de sonidos	Sí	No	
Re-especificación de sonidos	Sí	Sí	
Generación de ondas	Sí	No	
Filtros y efectos	Sí	Sí	FMOD dispone de más efectos y permite manipularlos durante la reproducción.
Superposición de efectos	Sí	Sí	
Control de reproducciones avanzado	Sí	Sí	FMOD ofrece funcionalidad más avanzada en cuanto a eventos y sincronización

	Audaspace	FMOD	Descripción
Convolución	Sí	Sí	
Audio biaural	Sí	Solo usando un *plugin* de terceros	3DCeption de TwoBigEars permite el uso de audio biaural en FMOD
Sencillez de la API	Muy sencilla	Muy sencilla	Ambas API son muy sencillas, pero FMOD proporciona varias herramientas que simplifican aún más el proceso.
Proporciona herramientas	No	Sí	FMOD ofrece una gran cantidad de herramientas de gran calidad
Fácilmente extensible	Sí	Sí	
Integración en otro *software* de desarrollo	No	Unreal Engine 4 y Unity	
Documentación	Programas de ejemplo con explicaciones y documentación de clases generada con DoxyGen	Documentación del API muy completa y varios ejemplos y tutoriales	
Lenguajes	C++, C y Python	C++	
Plataformas	Windows, Mac y Linux	Windows, Windows Store, UWP, Mac, Linux, Android, Windows Phone 8, iOS, PS3, PS4, PSVita, Xbox 360, Xbox One y Wii U.	
Precio	0€	Entre 0$ y 15.000$	

Figura 34: Comparación con FMOD

Como se puede ver, FMOD es la biblioteca más completa que se ha probado y supera a Audaspace en varias categorías. Uno de los mayores puntos fuertes de FMOD son sus diversas herramientas que facilitan mucho la tarea de los diseñadores de audio. En cuanto a la API, FMOD presenta dos API con distinto nivel de abstracción. Una de muy alto nivel y otra de más bajo nivel que permite un mayor control a cambio de un poco más de complejidad.

Además, FMOD también soporta una gran cantidad de plataformas, incluyendo la mayoría de las consolas.

Por otro lado, es la más cara de las librerías analizadas, Y su modelo de licencia por producto y plataforma puede hacerla prohibitiva en algunos casos.

8 CONCLUSIONES

Comencé a colaborar con Audaspace como parte del *VALS Semester of Code* y, en lo que a mí respecta, ha sido una gran experiencia. Me permitió colaborar en un proyecto con un ámbito internacional y, además, trabajar en un campo en el que no tenía experiencia previa, como es el procesamiento de audio.

En cuanto a las funcionalidades desarrolladas por mí, creo que cumplen, e incluso se superan los objetivos establecidos al principio de proyecto.

En primer lugar, como se mostró en el apartado 6, todas las funciones de control de sonidos han resultado muy útiles a la hora de integrar Audaspace en un videojuego, reduciendo la complejidad del mismo, y facilitando la implementación de características más avanzadas que las que existían hasta ese momento en el ejemplo presentado. Todo esto con un tiempo de desarrollo muy reducido.

En segundo lugar, tal como se puede escuchar en las demostraciones, la implementación del sistema eficiente de convolución fue también un éxito, permitiendo realizar un gran número de convoluciones simultaneas en tiempo real y siendo realmente sencillo de utilizar. Además, este sistema es clave en el proyecto, pues posibilita la creación de importantes efectos, como son la reverberación y sonido biaural, los cuales permiten una representación mucho más realista del sonido, sobre todo a través de auriculares.

Otro punto importante son las diferentes comparaciones que se han realizado con otras bibliotecas del mismo ámbito. En ellas se puede ver que Audaspace sale muy bien parada, incluso cuando se la compara con algunas bibliotecas comerciales de gran prestigio y que llevan mucho tiempo en el mercado. Esto, junto con su gran sencillez y su origen *open source*, hacen que sea ideal como alternativa a opciones menos sencillas o mucho más costosas que, actualmente, son utilizadas con asiduidad.

A modo de recuento, algunas de las ventajas más importantes que Audaspace ofrece de cara al usuario son:

- Gran sencillez de uso.
- Gran cantidad de funcionalidades, algunas de ellas muy poco frecuentes en otras bibliotecas.
- Fácilmente expandible.
- Coste cero.
- Código abierto.

9 LÍNEAS DE TRABAJO FUTURO

Audaspace es una librería muy joven que aún puede ser mejorada. Algunos de los aspectos más interesantes en los que se puede trabajar en el futuro son:

- Cambiar a una arquitectura nodal que permita añadir, quitar y configurar efectos a sonidos que ya están reproduciéndose.
- Creación de herramientas que faciliten el uso de Audaspace a diseñadores de audio sin conocimientos de programación.
- Procesamiento de audio en la GPU.
- Reconocimiento y síntesis de voz utilizando bibliotecas como *espeak*.
- Añadir *language bindings* a otros lenguajes como Java, C#, Lua, etc.
- Soporte a plataformas móviles (iOS, Android…)
- Implementar nuevos *plugins* utilizando librerías como *libogg, libtheora, Windows native audio*…

10 AGRADECIMIENTOS

- A Francisco José García Peñalvo – Por realizar todas las funciones correspondientes a la dirección de un Trabajo de Fin de Máster y por darnos a conocer y motivarnos a tomar parte en el *VALS Semester of Code*.
- A Jörg Müller, creador de Audaspace – Por toda la ayuda que me prestó, tanto durante el VALS como después de él.
- A todos los artistas que comparten desinteresadamente recursos de audio con toda la comunidad en internet.

11 GLOSARIO

- **API:** *Application Programming Interface*. Conjunto de subrutinas, funciones y procedimientos que ofrece una biblioteca como capa de abstracción para ser utilizada por otro *software*.
- **Azimuth:** Dirección horizontal expresada como la distancia angular entre la dirección a un punto fijo y la dirección a el objeto.
- ***Benchmark:*** Técnica utilizada para medir el rendimiento de un Sistema o componente del mismo.
- ***Crossfade***: Efecto de mezcla de audio que consiste en hacer *fade-out* al sonido que se está reproduciendo a la vez que se hace *fade-in* a un nuevo sonido que empieza a reproducirse.
- ***Fade-in, fade-out***: Efecto que implica que un sonido aumente su volumen paulatinamente al empezar su reproducción o lo reduzca al finalizarla respectivamente.
- ***Fast Fourier Transform* (FFT):** Algoritmo que computa la transformada discreta de Fourier de una secuencia (o su inversa) de forma mucho más eficiente. Reduce la complejidad computacional de $\mathrm{O}(N^2)$ a $\mathrm{O}(N\log N)$.
- ***Finite Impulse Response* (FIR):** Se llama *impulse response* o respuesta al impulso, a la salida que produce un sistema cuando se le presenta una breve señal de entrada, llamada impulso. Si es de duración finita, entonces se puede hablar de *finite impulse response.*
- **Función *callback***: También llamada función de retrollamada. Consiste en una función que se pasa como argumento a otra función para que sea llamada en algún momento.
- ***Git:*** Es un sistema de control de versiones distribuido. Diseñado por Linus Torvalds.
- **IDE:** Entorno integrado de desarrollo. Es una aplicación informática que ofrece a los programadores un conjunto de servicios y herramientas que les facilita el desarrollo de software.
- **Muestra:** Valor o conjunto de valores en un punto del tiempo y/o espacio.
- ***Shareware:*** Se denomina así al modelo de distribución de *software* en el que el usuario puede probar el producto, por tiempo limitado o con otras restricciones, antes de comprarlo.
- **Sistema de control de versiones:** Sistema que permite la gestión de los diversos cambios que se realizan sobre los elementos de algún producto. Debe proporcionar un mecanismo de almacenamiento de los elementos que se controlen, posibilidad de realizar cambios sobre los mismos y un registro histórico de los cambios que haya sufrido cada elemento.
- ***Templates***: También llamados plantillas. Son una funcionalidad del lenguaje C++. Permiten a una clase operar con tipos genéricos, lo que posibilita que funcione con una gran variedad de tipos sin necesidad de reescribirla.

- ***Thread-safe:*** Se dice del código que funciona correctamente cuando es ejecutado concurrentemente por varios hilos.
- **Transformada de Fourier**: Transformación matemática utilizada para transformar señales entre el dominio del tiempo y el dominio de la frecuencia.
- ***Zero-padding*****:** Consiste en añadir ceros a una señal en el dominio del tiempo para aumentar su longitud.

12 REFERENCIAS

[1] F. J. García Peñalvo, J. Cruz Benito, M. Á. Conde y D. Griffiths, «Semester of Code: Piloting Virtual Placements for Informatics across Europe,» de *Proceedings of Global Engineering Education Conference*, Tallinn, Estonia, 2015.

[2] F. J. García Peñalvo, J. Cruz Benito, M. Á. Conde y D. Griffiths, «Virtual placements for informatics students in open source business across Europe,» de *Frontiers in Education Conference Proceedings*, Madrid, Spain, 2014.

[3] F. J. García Peñalvo, J. Cruz Benito, D. Griffiths, P. Sharples, S. Wilson y M. Johnson, «Developing Win-Win Solutions for Virtual Placements in Informatics: The VALS Case,» de *Proceedings of the Second International Conference on Technological Ecosystems for Enhancing Multiculturality*, New York, USA, 2014.

[4] F. J. García Peñalvo, J. Cruz Benito, D. Griffiths y A. Achilleos, «Virtual placements management process supported by technology: Proposal and firsts results of the Semester of Code,» *Revista Iberoamericana de Tecnologías del Aprendizaje,* 2016.

[5] S. J. Orfanidis, Introduction to Signal Processing, Prentice Hall, 2010.

[6] J. O. Smith, Spectral Audio Signal Processing.

[7] S. Hilbert, «Bitweenie,» 22 4 2013. [En línea]. Available: http://www.bitweenie.com/listings/fft-zero-padding/. [Último acceso: 25 8 2015].

[8] D. L. Jones, «OpenStax CNX,» 21 6 2004. [En línea]. Available: https://cnx.org/contents/sPUi4DGE@5/Fast-Convolution. [Último acceso: 20 8 2015].

[9] «W3C Working Draft Web Audio Api,» [En línea]. Available: https://www.w3.org/TR/webaudio/convolution.html. [Último acceso: 20 8 2015].

[10] «iXBT,» 15 9 2003. [En línea]. Available: http://www.ixbt.com/dvd/ht-room.shtml. [Último acceso: 2015 10 16].

[11] T. Hancock, «Free Software Magazine,» 11 4 2011. [En línea]. Available: http://www.freesoftwaremagazine.com/articles/understanding_surround_and_binaural_sound. [Último acceso: 17 10 2015].

[12] D. o. E. a. C. E. U. o. C. Davis, «The CIPIC Interface Laboratory,» [En línea]. Available: http://interface.cipic.ucdavis.edu/sound/tutorial/psych.html. [Último acceso: 16 10 2015].

[13] J. Bejoy, «Virtual Surround Sound Implementation Using Deccorrelation Filters and HRTF».

[14] B. Hsu, «Bill Hsu.me,» 18 4 2013. [En línea]. Available: http://billhsu.me/?tag=head-tracking. [Último acceso: 20 1 2016].

[15] E. Battenberg y R. Avizienis, «Implementating Real-Time Partitioned Convolution Algorithms on Conventional Operating Systems,» de *Conference on Digital Audio Effects* , Paris, France, 2011.

[16] F. Wefers, Partitioned Convolution Algorithms for Real-Time Auralization, Logos Verlag Berlin, 2014.

[17] B. Gardner y K. Martin, «Music, Mind, and Machine Group,» MIT Media Lab, 27 1 1997. [En línea]. Available: http://sound.media.mit.edu/resources/KEMAR.html. [Último acceso: 3 9 2015].

[18] W. Fohl, J. Reichardt y J. Kuhr, «A System-On-Chip Platform fo rHRTF-Based Realtime Spatial Audio Rendering,» de *The Second International Conference on Creative Content Technologies*, Hamburg, Germany, 2010.