Universidad Zaragoza
1542

Trabajo Fin de Grado

Implementación de sonido binaural para aplicaciones de realidad aumentada

Autor

Ignacio Ayora Morante

Director

José Ramón Beltrán Blázquez
Departamento de Ingeniería Electrónica y Comunicaciones

Escuela de Ingeniería y Arquitectura
2013/2014

Implementación de sonido binaural para aplicaciones de realidad aumentada

RESUMEN

El objetivo principal de este trabajo es la realización de un plugin que permita procesar sonido binaural (audio en 3D) dentro del programa Unity, destinado al diseño y desarrollo de videojuegos en 3D.

Actualmente Unity cuenta con herramientas simples de generación de sonido espacial. Estas herramientas permiten atenuar el sonido en función de la distancia y generar diferentes espacios de reverberación para simular distintos ambientes sonoros.

Sin embargo, no dispone de ninguna herramienta de posicionamiento realista de la fuente de sonido. Este es el objetivo principal del proyecto. Dotar a Unity de la capacidad de posicionamiento y posterior reproducción de una fuente de sonido en el espacio.

El desarrollo del trabajo se ha llevado a cabo primero en el entorno Matlab, ya que facilita la depuración y corrección de errores además de ofrecer la posibilidad de representar las distintas variables, para después traspasarlo al lenguaje C#, con el que se compila el plugin final.

Contenido

1. INTRODUCCIÓN

El objetivo principal de este trabajo es la realización de un plugin que permita procesar sonido binaural (audio en 3D) dentro del programa Unity, destinado al diseño de videojuegos 3D.

Actualmente Unity cuenta con herramientas simples de generación de sonido espacial. Estas herramientas permiten atenuar el sonido en función de la distancia y generar diferentes espacios de reverberación para simular distintos ambientes sonoros.

Sin embargo, no dispone de ninguna herramienta de posicionamiento realista de la fuente de sonido. Este es el objetivo principal del proyecto. Dotar a Unity de la capacidad de posicionamiento y posterior reproducción de una fuente de sonido en el espacio.

Este informe se ha estructurado en dos partes muy diferenciadas. La primera de ellas se basa en explicar teórica y detalladamente las ideas más importantes del trabajo como pueden ser los principios del audio en 3D o el método de interpolación utilizado.

Tras esto, se comentará en orden cronológico, las 3 fases seguidas para el desarrollo del trabajo. La primera de ellas, se basa en el entorno Matlab, herramienta muy útil para realizar una primera aproximación al problema, aprendiendo de manera más visual, gracias a sus gráficas, cómo funcionan los filtros HRTF. La segunda fase se desarrolla dentro del entorno de desarrollo Visual Studio, con el que programaremos el plugin final basado en el lenguaje de programación C#. Finalmente, la tercera fase se centra en Unity, con la elaboración de una escena en la que se pueda testar y demostrar el correcto funcionamiento del plugin.

2. PRINCIPIOS TEÓRICOS

2.1 AUDIO 3D. AUDIO BINAURAL

Para realizar un plugin que implemente sonido binaural dentro del entorno de desarrollo de videojuegos Unity, lo primero de todo es saber que el sonido binaural hace referencia al sonido en 3D elaborado mediante los filtros HRTF. Dichos filtros se explican detalladamente en el apartado 2.2.

Ahora bien, el sonido 3D no es más que una modificación llevaba a cabo sobre el sonido estéreo, otorgando al receptor la posibilidad de distinguir distintas posiciones de la fuente sonora dentro del espacio tridimensional. Para ello, hay que saber que en el sistema auditivo, la sensación tridimensional está relacionada con la diferencia de amplitud y tiempo con la que recibe cada oído la señal sonora. Es decir, la localización de los sonidos en el espacio se consigue con el procesamiento por separado de la información de cada oreja y con la posterior comparación de fase y nivel entre ambas señales (Wikipedia. Escucha Binaural s.f.).

En definitiva, para conseguir un sonido en 3D, tenemos que 'engañar' al cerebro humano, modificando de alguna forma las propiedades de una determinada señal de audio. En el caso del sonido binaural, los filtros HRTF son los encargados de modificar la señal.

El espacio tridimensional, cuando estamos hablando de audio 3D, y más concretamente del uso de los filtros HRTF, se hace uso de las coordenadas esféricas, mediante las cuales podemos situar cada punto del espacio mediante los ángulos 'phi' (Ψ), 'theta' (ϕ) y el radio 'r'. Ver Figura 2.1.

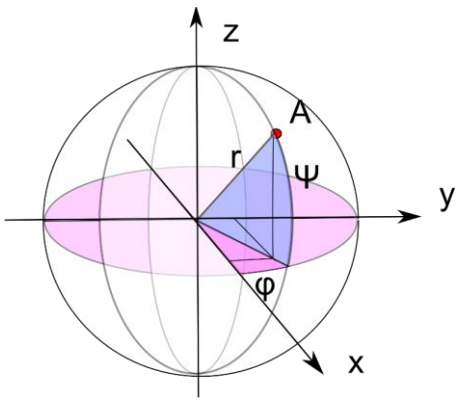


Figura 2.1.- Representación de las coordenadas esféricas.

2.2 HRTF

Los filtros HRTF, "Head-Related Transfer Function", que en español se podrían traducir como funciones de transferencia relacionadas con la cabeza, son filtros que caracterizan la respuesta en frecuencia del sistema hombros-cabeza-oído frente a una fuente de audio. (Mengqiu Zhang 23–25 November 2009)

Midiéndose en campo lejano, las HRTF son funciones del valor de acimut (Ver Figura 2.2) y la elevación de la fuente (Ver Figura 2.3), y esto permite al oyente determinar la dirección del emisor. (Beltrán Blázquez)

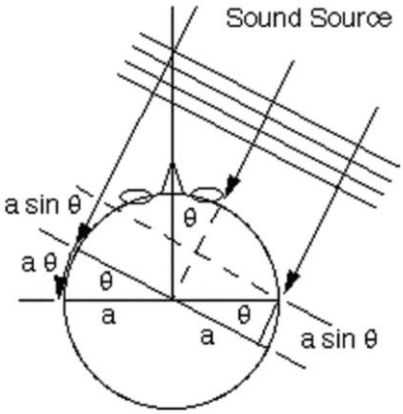


Figura 2.2.- Representación Acimut

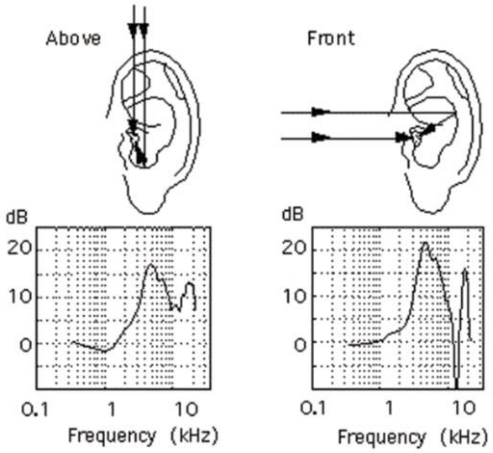


Figura 2.3.- Representación Elevación

Ambas variables, acimut y elevación, se corresponden respectivamente a las variables theta (ϕ) y phi (Ψ), que junto al radio r, representan el espacio tridimensional en coordenadas esféricas tal y como se mostró en la Figura 2.1. El valor del radio está normalizado para las distintas HRTF. En caso de querer modificar la distancia, habría que atenuar o amplificar convenientemente la señal.

Es importante destacar que las HRTF se obtienen realizando medidas del sistema hombros-cabeza-oído sobre distintos maniquís dentro de una cámara anecoica. Estas medidas varían bastante según las características de cada sujeto, por lo que hay que tener en cuenta que las bases de datos o bien, ofrecen varios filtros, uno por cada sujeto o maniquí, o bien proporcionan un único conjunto de filtros HRTF, los cuales son el resultado de promediar todos los demás.

En las bases de datos que se proporcionan varios sujetos distintos, es importante elegir aquel que mejor se adapte a nuestras características físicas, ya que de lo contrario, no percibiremos la sensación tridimensional de forma óptima. Siguiendo esta idea, hay que tener en cuenta que debido a la asimetría del cuerpo humano, también se distingue entre la oreja derecha e izquierda, y se proporcionarán filtros distintos para cada una de ellas.

El espacio tridimensional está constituido por infinitos puntos, pero obviamente, las bases de datos HRTF no pueden contener un filtro por cada uno de ellos. Únicamente se proporcionan los filtros asociados a un número finito de puntos del espacio, los cuales varían según el creador de la base de datos HRTF. Si se desea obtener el filtro de un punto del espacio que no está disponible en la base de datos HRTF, será necesario elegir aquel punto más cercano que sí este determinado por un filtro HRTF de la base de datos, o bien, realizar un método de interpolación. En el caso de este trabajo, se va a realizar un método de interpolación para obtener los filtros HRTF de aquellos puntos no disponibles en la base de datos. El método utilizado se explica en el apartado 2.4.

Una vez elegido un punto del espacio y obtenido su filtro HRTF mediante los valores de acimut y elevación, el siguiente paso para la obtención del audio binaural consiste en filtrar la señal de audio con el filtro HRTF obtenido. De esta forma, el sonido generado simulará provenir del lugar elegido.

Actualmente, dependiendo de quien haya sido el creador, existen varias bases de datos HRTF. A continuación se van a explicar la CIPIC y la MIT ya que fueron las dos principales candidatas para ser usadas en este trabajo, aunque finalmente se decidió usar la CIPIC.

- CIPIC Database (CIPIC- Center for Image Processing and Integrated Computing University of California 1 Shields Avenue Davis)

  La base de datos CIPIC HRTF es una base de datos de dominio público, con mediciones HRTF de alta resolución espacial de 45 sujetos diferentes, incluyendo el maniquí KEMAR con orejas tanto grandes como pequeñas.

  Esta base de datos clasifica los distintos filtros según la persona o sujeto y distingue entre 50 valores distintos de elevación y de 25 en acimut, abarcando desde los -45 a los 275 grados, y de los -80 a los 80 grados respectivamente. Además, incluye las medidas antropométricas de los distintos sujetos para su uso en estudios técnicos.

  Una de las ventajas de esta base de datos es que se encuentra disponible en lenguaje Matlab, proporcionando además un script de apoyo que muestra su comportamiento tanto en tiempo como en frecuencia ayudando considerablemente a la comprensión de las HRTF.

  Su desventaja es que se encuentra únicamente escrito en Matlab presentando un gran problema a la hora usar dicha base de datos dentro de un entorno de desarrollo con otro lenguaje de programación distinto, como puede ser C#. Esto ha obligado a que a la hora de crear el plugin se haya traducido únicamente los filtros relacionados a un sujeto, concretamente el 18, ya que hacerlo con toda la base de datos entera (45 sujetos) se presentaba inviable en tiempo.

- **MIT Database** (Gardner y Martin)

Al igual que CIPIC, la base de datos MIT es de dominio público con mediciones HRTF de alta resolución. Abarca hasta 710 posiciones diferentes del espacio recorriendo elevaciones desde los -40 hasta los 90 grados. Es importante destacar, que para cada valor de elevación, hay un rango distinto de valores de acimut. Esto hace que la distribución en el espacio no sea regular, y se convierte en una desventaja ya que complica la realización de la triangulación de Delaunay, necesaria dentro del método de interpolación.

Como ventaja se podría destacar que esta base de datos está implementada en el lenguaje de programación C++, por lo que el desarrollo del plugin se podría haber realizado directamente en dicho lenguaje, evitando traducciones de lenguajes. Además, C++ es bastante más universal y popular que C# dentro de la comunidad de programadores, por lo que la búsqueda de algunas funciones como la FFT o la triangulación de Delaunay en versión free para su uso dentro del plugin hubiera sido más fácil. Por otro lado, hay que decir que en Unity, la importación de plugins basados en C#, se puede hacer tanto con la versión Indie como con la Pro, mientras que los basados en C++ únicamente se pueden importar usando la versión Pro de Unity.

Una vez vistas las ventajas y desventajas de cada una, la elección de la base de datos CIPIC está motivada principalmente por el hecho de que la distribución en el espacio de los puntos con un filtro HRTF asociado es regular. Eso hace que más tarde el método de interpolación sea mucho más sencillo y rápido.

2.3 DISCONTINUIDADES

Tal y como se explica en el Apartado 2.1, el sonido binaural se obtiene aplicando los filtros HRTF sobre una señal de audio. Con ellos modificamos la señal, simulando la posición de la fuente sonora en un punto concreto del espacio tridimensional.

Ahora bien, si queremos simular que la fuente sonora se desplaza de posición, deberemos trocear o dividir la señal del audio en varios fragmentos, y aplicar a cada uno de ellos un filtro que simule una posición diferente. A cada uno de los fragmentos de la señal los denominaremos ‘frames’. En el caso de Unity, cada frame consta de 1024 muestras, siendo la frecuencia de muestreo de 44100Hz.

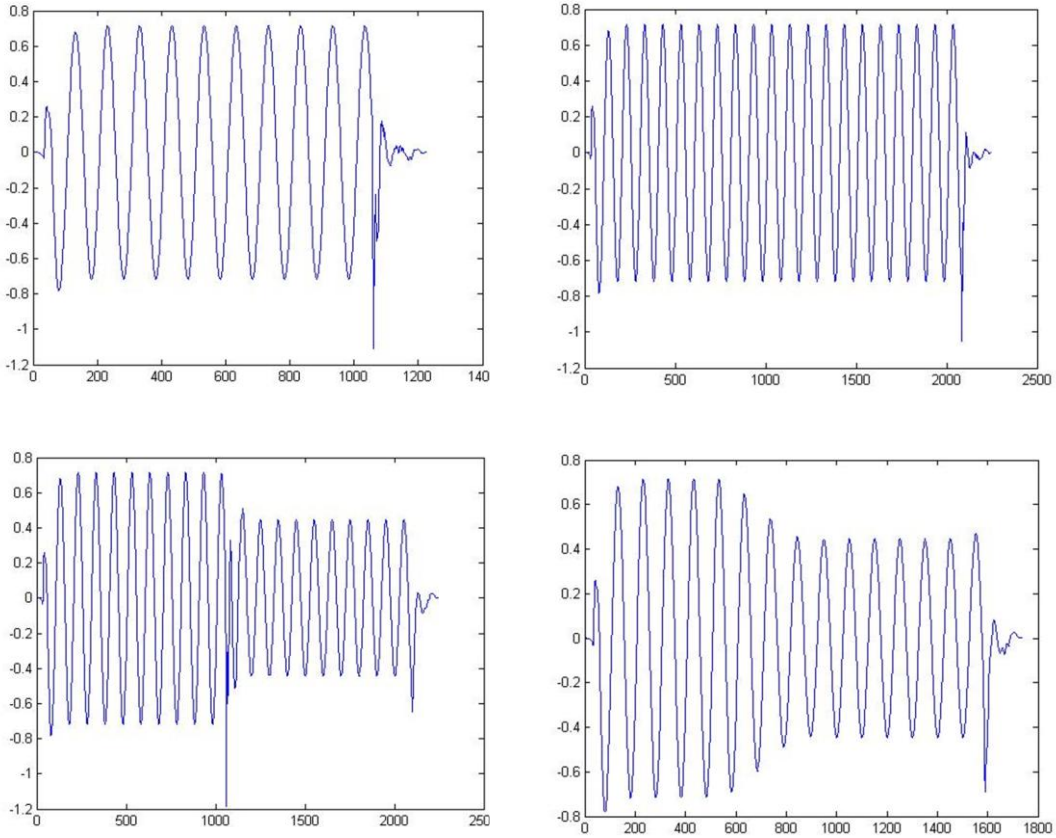


Figura 2.4- Representación HRTF

a) Filtro HRTF aplicado sobre un frame.
b) Overlap-Add aplicado sobre dos frames filtrados con el mismo filtro.
c) Overlap-Add aplicado sobre dos frames filtrados con filtros distintos.
d) Weighted Overlap-Add, aplicado sobre dos frames filtrados con filtros distintos.

El problema de aplicar varios filtros distintos sobre una señal de audio, es la aparición de discontinuidades producidas por el cambio de filtro a lo largo del tiempo. Para solucionarlo se ha optado por utilizar el método "Weighted Overlap-add".

Tal como se puede apreciar en la Figura 2.4.a, al aplicar un filtro HRTF determinado sobre un frame de audio, en este caso un seno puro, aparece al final del mismo un trozo de tamaño igual al tamaño del filtro menos uno, correspondiente al transitorio. Si a dos frames consecutivos se le aplica el mismo filtro a cada uno de ellos por separado, y se solapan con un número de muestras igual al tamaño del filtro menos uno, obtenemos la señal b) de la Figura 2.4, en la que podemos observar que no hay ningún tipo de problema en la unión de frames.
En cambio, si a dos frames consecutivos, se le aplica a cada uno de ellos un filtro distinto y se solapan con el número de muestras indicado anteriormente, obtenemos la gráfica c) de la Figura 2.4, en la que podemos ver como aparece una discontinuidad entre frames. Esto se traduce en un ruido bastante audible de la señal, en forma de golpes o 'clicks' sonoros.
La última gráfica de la Figura 2.4, es decir, la d), muestra como la introducción del método "Weighted Overlap-add", reduce en gran medida la discontinuidad existente en la anterior gráfica.

2.3.1 OVERLAP-ADD

El método "Overlap-add" (Wikipedia. Overlap-Add) es una forma de realizar el filtrado sobre una señal digital de manera mucho más eficiente. En vez de realizar la convolución del filtro con toda la señal de audio, éste método divide la señal en varios trozos, llamados frames, y realiza una convolución por separado con cada uno de ellos. De esta forma, el cálculo es mucho más eficiente ya que pasamos de realizar una única convolución sobre una señal de gran tamaño, a realizar varias convoluciones sobre señales mucho más pequeñas. Finalmente, se unen los frames filtrados, realizando un pequeño solapamiento de las muestras finales, tal y como se puede ver en la Figura 2.5.

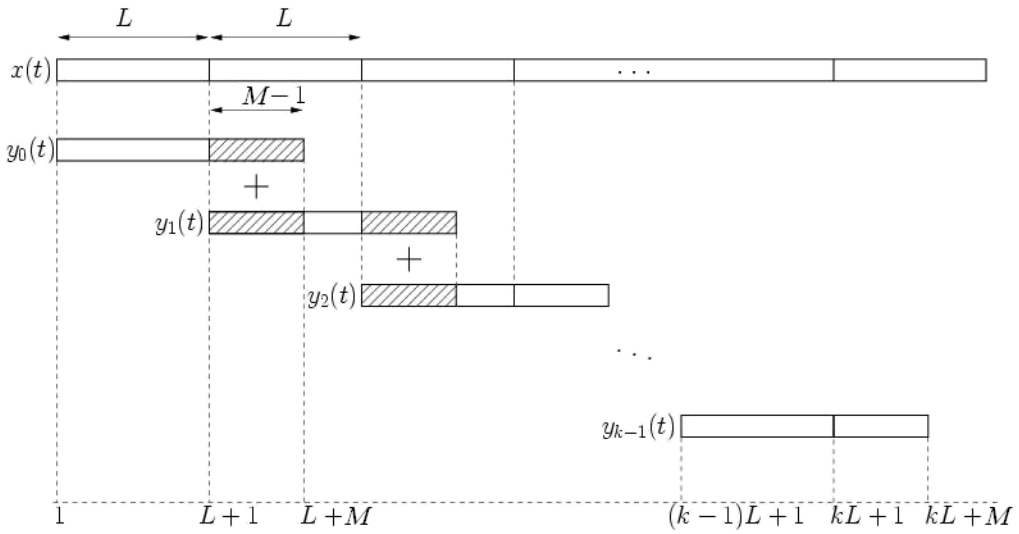


Figura 2.5.- Representación gráfica del método de Overlap-Add

Para entender perfectamente el funcionamiento de este método hay que recordar, que al realizar la convolución de una señal de tamaño L, con un filtro de tamaño M, el resultado será un vector de tamaño igual a L+M-1. Este exceso de tamaño respecto al vector inicial, es el que está asociado al transitorio, y por tanto será el trozo de señal que se solapará al siguiente frame.

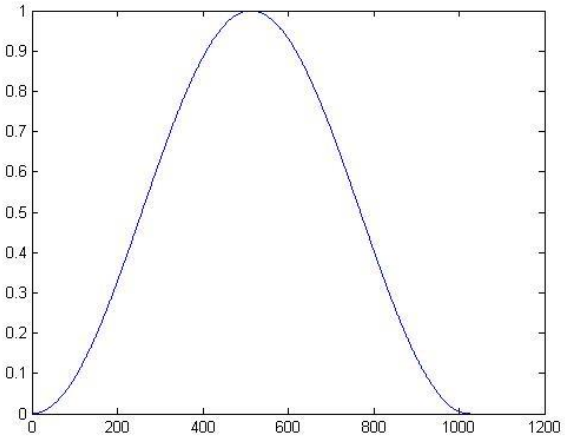
Para que el método "Overlap-Add" funcione correctamente, y el resultado final sea el mismo que en el caso de haber aplicado el filtrado sobre la señal completa, el tamaño del vector y_k, al que llamaremos N, debe cumplir lo siguiente. $N \geq L+M-1$.

Ahora bien, esto será así siempre y cuando se aplique el mismo filtro a todos los frames. En caso de querer aplicar un filtro distinto a cada uno de los frames, el método "Overlap-add", no arreglará por sí solo el problema de las discontinuidades y sus respectivos 'clicks' sonoros. Esto se aprecia perfectamente en la Figura 2.4, concretamente en las gráficas b) y c), ya que ambas muestran la aplicación del método "Overlap-Add" sobre dos frames consecutivos, con la diferencia de que en la primera gráfica el filtro aplicado es el mismo para los dos frames, mientras que en la segunda se han aplicado dos filtros distintos sobre los dos frames de audio.

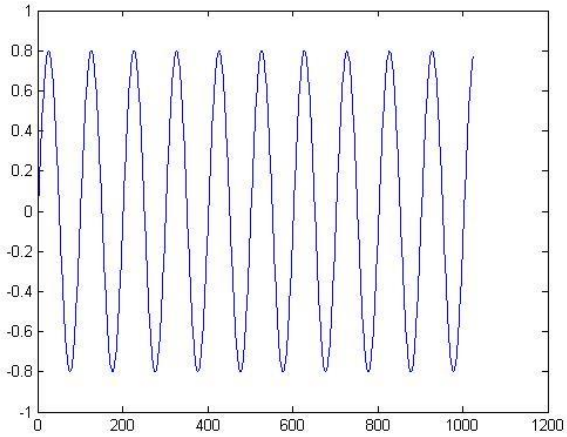
2.3.2 WEIGHTED OVERLAP-ADD

Para minimizar las discontinuidades aparecidas, debemos aplicar el método "Weighted Overlap-add" (Smith), cuyo resultado se puede ver en la gráfica d) de la Figura 2.4.

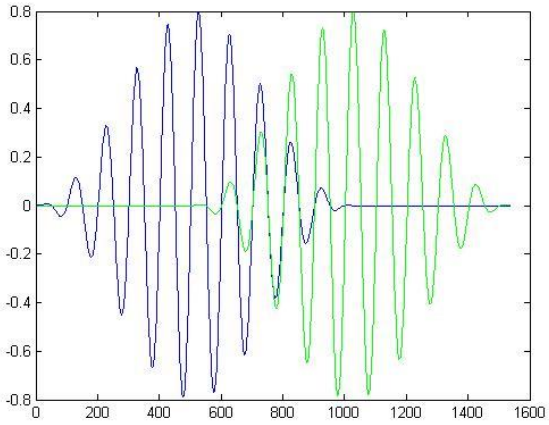
Dicho método se basa en el "Overlap-Add" y la única diferencia entre ellos es que ahora se realiza un enventanado después del filtrado y antes del solapamiento con el objetivo de suavizar la discontinuidad entre un frame y otro. Por lo tanto, el procedimiento a seguir se basa en trocear la señal en varios frames, y aplicarle a cada uno de ellos un filtro HRTF. El vector resultante del filtrado lo multiplicamos por una ventana de Hanning como la mostrada en la gráfica a) de la Figura 2.8, para después realizar el solapamiento.



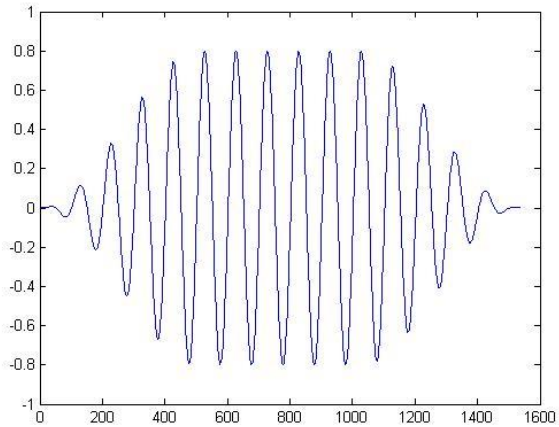
a). Ventana de Hanning.



b). Frame de audio a enventanar.



c). Demostración solapamiento con enventanado.



d). Resultado final tras la suma.

Figura 2.8.- Método "Weighted Overlap-Add" aplicado sobre dos frames de audio.

En la Figura 2.8 podemos ver todos los pasos seguidos para realizar correctamente el "Weighted Overlap-Add". Partimos de una señal de audio, en este caso un seno como el de la gráfica b), el cual filtramos con un filtro HRTF. Después se multiplica con la ventana de Hanning de la gráfica a) y obtenemos los frames suavizados de la gráfica c).

Si sumamos las muestras de ambos frames, obtenemos la señal de la gráfica d) ya que la ventana que hemos usado, la de Hanning, está diseñada de tal forma que la suma de sus colas dan como resultado 1, evitando alterar la escala de amplitud de la señal.

2.4 INTERPOLACIÓN

El objetivo final de este trabajo es poder realizar una interpolación de los filtros HRTF, de tal forma que podamos obtener cualquier punto que queramos del espacio tridimensional. Tras una pequeña fase de investigación y búsqueda de información, se pueden destacar 4 métodos distintos (Gamper):

a) "The normalised VBAP weights".
b) "Inverse distance weighting".
c) "Bilinear interpolation of 3 measurement points".
d) "Bilinear interpolation of 4 measurement points".

De entre todos ellos, el elegido para este trabajo ha sido el tercero, "Bilinear Interpolation of 3 measurements points" debido a su efectividad y facilidad de adaptación tanto al entorno Matlab como el de Unity. El esquema a seguir para llevar a cabo este método, se puede ver en la Figura 2.9.

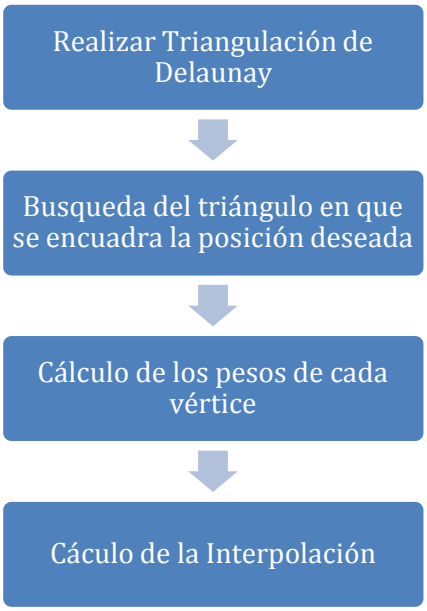


Figura 2.9.- Pasos a seguir en la interpolación.

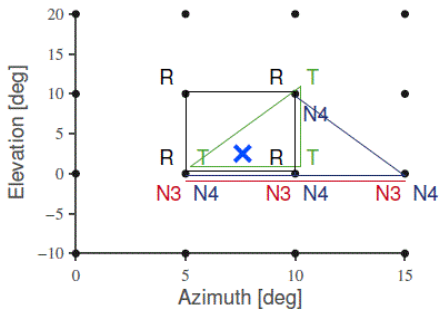


Figura 2.10.- Interpolación de 3 y 4 puntos.

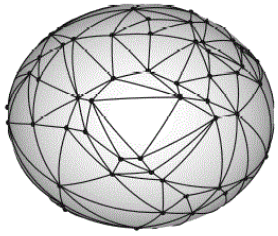


Figura 2.11.- Triangulación sobre una esfera.

Como su propio nombre indica, el método “Bilinear Interpolation of 3 measurements points” realiza la interpolación basándose en tres puntos distintos de medida y calculando sus pesos. Para escoger los tres puntos de medida, lo primero que tenemos que hacer es realizar la triangulación de Delaunay partiendo de los puntos del espacio en los que existe un filtro HRTF disponible en la base de datos. Una vez realizada la triangulación, los tres puntos de medida se corresponderán con los tres vértices del triángulo en que se encuentra inscrito el punto que queremos interpolar. La Figura 2.10 muestra una ‘X’ que representa el punto en el que queremos obtener el filtro interpolado, inscrita dentro de un cuadrado negro (“Bilinear Interpolation of 4 measurements points”), y de un triángulo verde (“Bilinear Interpolation of 3 measurements points”). Sabiendo que el triángulo que lo encuadra es el de color verde, deberemos obtener los filtros HRTF que se encuentren en los puntos marcados por sus tres vértices, es decir los puntos (5,0), (10,0) y (10,10) y calcular el filtro interpolado con la suma ponderada de los tres. El peso que se aplica a cada filtro está determinado por la distancia euclidea existente entre el punto ‘X’ y el vértice correspondiente.

2.4.1 TRIANGULACIÓN DE DELAUNAY

La triangulación de Delaunay (Wikipedia. Triangulación de Delaunay), es una red de triángulos que cumple la condición de Delaunay. Esta condición dice que la circunferencia circunscrita de cada triángulo de la red no debe contener ningún vértice de otro triangulo. Un ejemplo de triangulación realizada sobre una esfera, se puede ver en la Figura 2.11.

Como ya se ha comentado anteriormente, la base de datos CIPIC proporciona sus filtros HRTF distribuidos de una forma regular a lo largo del espacio tridimensional. Esto quiere decir, que para cada valor de elevación, tenemos siempre el mismo número de valores de acimut, lo que hace que se obtenga una triangulación de Delaunay como la mostrada en la Figura 2.12.

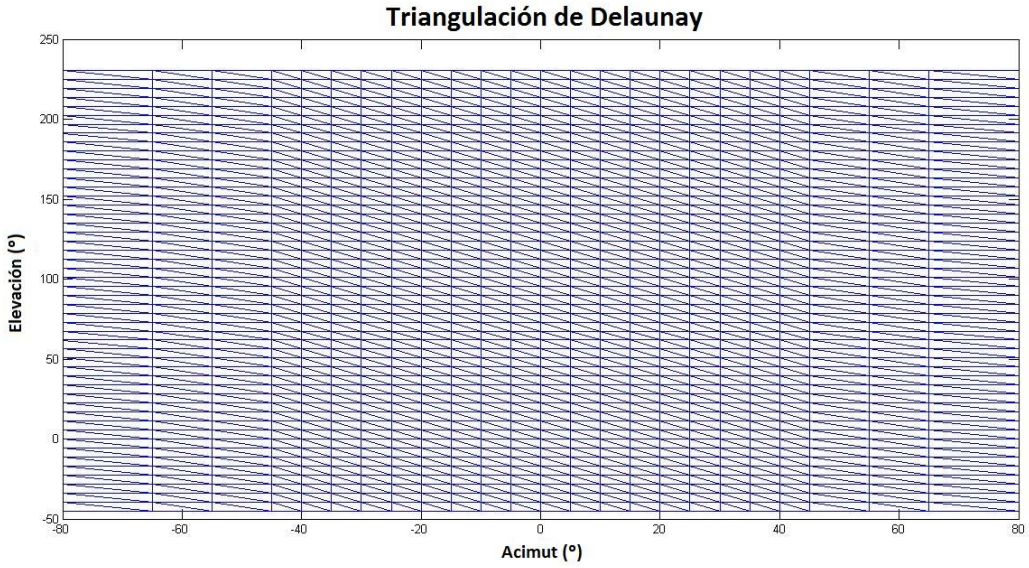


Figura 2.12.- Triangulación de Delaunay realizada sobre la base de datos CIPIC.

2.4.2 BÚSQUEDA DEL TRIÁNGULO

Una vez obtenida la triangulación de Delaunay, ya tenemos el espacio perfectamente delimitado en varios triángulos cuyos vértices se corresponden con aquellos puntos del espacio que contienen un filtro HRTF.

Por lo tanto, cualquier dupla de acimut y elevación, tendrá siempre un triángulo que la encuadre, siendo sus vértices los tres puntos de medida necesarios para realizar el método de interpolación elegido, el "Bilinear Interpolation of 3 measurements points".

Sabiendo esto, el siguiente paso consiste en conocer o calcular dentro de qué triángulo se encuentra el punto que queremos interpolar.

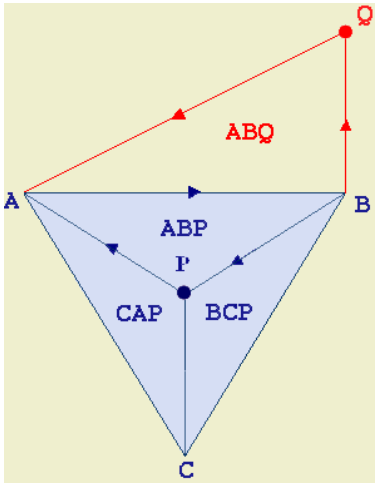


Figura 2.13. Representación de dos triángulos y un punto P inscrito dentro de uno de ellos.

El algoritmo seguido en este trabajo se basa en el concepto de orientación (Departamento de Matemática Aplicada. Universidad Politécnica de Madrid). La orientación de cada triángulo se determina de acuerdo a la dirección del movimiento cuando se visitan los vértices en el orden especificado.

Fijándonos en la Figura 2.13, el punto P se encuentra dentro del triángulo formado por los vértices ABC, pero no está dentro del triángulo ABQ. Visualmente es fácil, pero a la hora de calcularlo con ordenador ya no es tan intuitivo.

El algoritmo que se va a seguir consiste en comprobar si la orientación de los triángulos formados por dos vértices más el punto P es la misma que la del triángulo principal. En caso de no coincidir todas las orientaciones, se deduce que el punto no está inscrito en ese triángulo.

Es decir, para el caso del triángulo ABC de la Figura 2.13, la orientación del mismo ha de coincidir con la orientación del triángulo ABP, el BCP, y el CAP. Mientras que para el caso del triángulo ABQ, la orientación debe coincidir con la de los triángulos ABP, BQP y QAP.

Ambos triángulos principales ABQ y ABC comparten el subtriángulo ABP, por lo tanto sabiendo la orientación de cada uno de ellos, y viendo cual coincide, sabremos a qué triangulo pertenece el punto P. Fijándonos de nuevo en la Figura 2.13, se puede ver que el triángulo ABC tiene su orientación en sentido horario, mientras que el triángulo ABQ tiene sentido anti horario. Sabiendo que el subtriángulo ABP tiene sentido horario, podemos descartar que el punto P se encuentre dentro del triángulo ABQ, ya que sus orientaciones no coinciden.

Ahora bien, que ABP y ABC coincidan en su orientación no asegura que P pertenezca a ese triángulo, ya que aún faltan por comprobar los otros dos subtriángulos CAP y BCP. En este caso, ambos subtriángulos tienen orientación en sentido horario por lo que ahora sí, al coincidir las tres orientaciones, podemos asegurar que el punto P pertenece al triángulo ABC.

Una vez comprendido visualmente el método, pasamos a explicar el algoritmo importante, el que usaremos después en nuestro programa para conocer las orientaciones de cada triángulo y así deducir en qué triangulo se encuentra el punto P. La fórmula es la siguiente:

$$(A1.x - A3.x)*(A2.y - A3.y) - (A1.y - A3.y)*(A2.x - A3.x)$$

Si el resultado de la ecuación es mayor o igual que cero, la orientación es positiva. En cambio, si el resultado es menor que cero, quiere decir que la orientación del triángulo es negativa.

Hay que tener en cuenta que 'A1', 'A2' y 'A3' representan los tres vértices del triángulo que estemos calculando en ese momento, y que las indicaciones '.x' e '.y' indican las coordenadas en abscisas o en ordenadas del vértice correspondiente.

Por ejemplo, si el triángulo ABC, se encuentra en las siguientes coordenadas cartesianas. A = (0,10), B= (0,0) y C = (10,0), la orientación obtenida sería positiva, y se habría calculado de la siguiente forma:

$$(0 - 10)*(0 - 0) - (10 - 0)*(0 - 10) = 100 \geq 0$$

Éste cálculo se realizaría con todos los triángulos, y de esa forma, comparando todas las orientaciones obtenidas, se deduciría cuál es la ubicación del punto P.

Una vez conocido el triángulo en cuestión, ya sabemos cuáles son los tres puntos de medida que necesitábamos para realizar la interpolación, ya que éstos se corresponden con los filtros determinados por los valores de acimut y elevación de los tres vértices.

2.4.3 CÁLCULO DE LOS PESOS Y DE LA INTERPOLACIÓN

Tras realizar la triangulación de Delaunay, y haber encontrado el triángulo que contiene el punto objetivo, hay que calcular los pesos de cada uno de sus vértices. En este caso, el cálculo de los pesos (Gamper) se basa en medir la distancia Euclidiana existente entre el punto 'P' y los tres vértices del triángulo 'ABC'. Después se normalizan los valores obtenidos. Las fórmulas seguidas son las siguientes:

$$d(A,P) = \sqrt{(P.x - A.x)^2 + (P.y - A.y)^2} \qquad Peso(A) = \frac{d(A,P)}{d(A,P) + d(B,P) + d(C,P)}$$

Con los pesos ya calculados, únicamente falta realizar la interpolación. Esto es tan sencillo como obtener los filtros HRTF de cada uno de los vértices del triángulo, y realizar una suma ponderada aplicando los pesos calculados. Recordar que se tiene que hacer por separado el canal izquierdo y derecho, ya que los filtros no son los mismos.

2.5 AUDIO EN UNITY

El audio en Unity está compuesto por 3 grupos principales. Los "AudioSource" que son los objetos emisores, los "AudioListener" que son los receptores u oyentes, y los "AudioClip" que son los que contienen los archivos de audio. (Beltrán Blázquez)(Unity)

2.5.1 AudioSource

"AudioSource" es la clase encargada de reproducir un "AudioClip". En la Figura 2.15 podemos apreciar en su parte superior, como el "AudioClip" que tiene asociado en este caso el "AudioSource" es "helicopteroLa", el cual es un sonido 2D. La reproducción de sonido 2D o 3D se elige dentro del "AudioClip".

El "AudioSource" tiene varias propiedades destinadas a configurar la emisión del sonido de tal forma que podemos aplicar filtros, seleccionar la reproducción en bucle o asignar un volumen adecuado al sonido. Además, según sea un sonido 2D o 3D, se podrán seleccionar unos ajustes u otros.

Dentro de una escena puede haber múltiples "AudioSources".

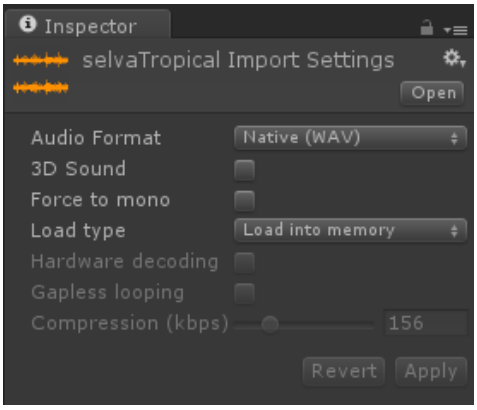


Figura 2.14.- Propiedades de un "AudioClip".

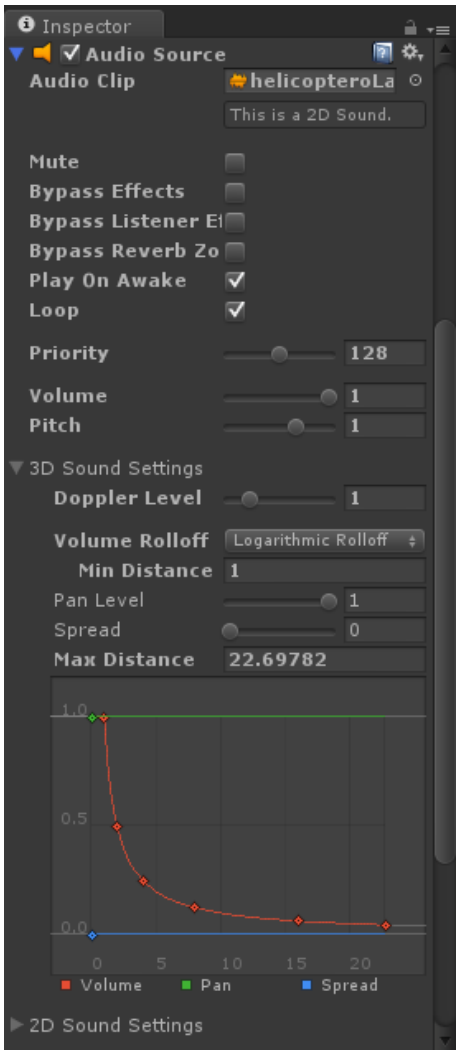


Figura 2.15.- Propiedades de un "AudioSource".

2.5.2 AudioClip

Clase que contiene el archivo de audio que se va a reproducir en el “AudioSource”. Unity es capaz de importar los siguientes formatos: .aif, .wav, .mp3 y .ogg.
Fijándonos en la Figura 2.14, podemos ver como marcando la casilla correspondiente, indicamos a Unity que queremos que reproduzca el archivo de audio en 3D.
En cambio, si generamos el audio 3D mediante el plugin creado en este trabajo, es necesario indicarle a Unity que el archivo es 2D.

2.5.3 AudioListener

Es la clase destinada a recibir el sonido generado por los “AudioSources” actuando como un micrófono. Solo puede haber un “AudioListener” por escena, y éste siempre estará por defecto asociado a la cámara.

2.6 FUNCIÓN OnAudioFilterRead (UNITY)

El objetivo que se busca con el desarrollo del plugin, es el procesado de audio 3D en tiempo real. El propio Unity, si queremos disponer de las muestras de audio en tiempo real, nos ofrece la posibilidad de obtenerlas gracias a la función OnAudioFilterRead, aunque ésta no existió hasta la versión 3 de Unity. En antiguas versiones era necesario hacer uso de las funciones getData() y setData() para poder tener acceso a las muestras de audio, y procesarlas según convenga. Con la primera de ellas se obtiene un array que contiene la señal de audio de un clip de sonido concreto, mientras que con la segunda función podemos establecer o asignar nuestro propio array de sonido dentro del clip de sonido elegido. Con estas funciones, estamos cambiando el propio clip de audio, por lo que en caso de configurar una reproducción en bucle, y procesar audio en tiempo real, una vez que lleguemos al segundo ciclo, las muestras que obtendremos serán las modificadas en el primer bucle, y no las muestras originales del clip de sonido.

A partir de la versión 3 de Unity, se introdujo la función ‘OnAudioFilterRead’ con la que podemos obtener y modificar el array de audio proveniente del AudioListener, insertando nuestro propio filtro dentro de la cadena de audio del DSP. En este caso estamos modificando las muestras del buffer de salida de la cadena de audio, por lo que no afectará al propio clip de sonido, y evitamos problemas en las reproducciones en bucle.

Fijándonos en la página de referencia de Unity (Unity), podemos ver que esta función se hereda de la clase MonoBehaviour, y que tiene dos parámetros.

MonoBehaviour.OnAudioFilterRead(float[], int)

El primero de ellos, es un *array* de audio pasado por referencia, y que contiene valores comprendidos entre -1.0f y 1.0f. Al estar pasado por referencia, las modificaciones que hagamos sobre esta variable, serán las que se reproduzcan después en la cadena de audio. Para evitar problemas, lo que hacemos dentro del script de Unity, es copiar los valores de dicha variable sobre una auxiliar, la cual procesamos con el plugin, para finalmente copiar el array procesado en el array inicial proporcionado por la propia función.

El segundo parámetro es un número de tipo entero que indica el número de canales que se está reproduciendo. En nuestro proyecto va a ser siempre estéreo con dos canales. Para gestionar estos dos canales, Unity entrelaza las muestras de cada canal dentro de un único array, siendo las muestras pares las correspondientes al canal izquierdo, y las muestras impares las correspondientes al canal derecho.

La función OnAudioFilterRead se activa cada vez que el buffer de audio está completo, y teniendo en cuenta que la frecuencia de muestreo es de 44100Hz, y el tamaño del buffer de cada canal es de 1024 muestras, la función se activará aproximadamente cada 23ms. Hay que recordar, que al ser estéreo se realiza un entrelazado de los dos canales, por lo que el tamaño final del array que proporciona la función OnAudioFilterRead es de 2048 muestras.

3. DESARROLLO DEL TRABAJO

Una vez explicados las principales ideas que se aplican en este trabajo, pasamos a explicar el desarrollo que se ha seguido durante su elaboración.

Como al principio los filtros HRTF y el audio en 3D era un tema desconocido, se prefirió dividir el trabajo en 2 fases, de tal forma que al principio se invirtiera tiempo en familiarizarse con la base de datos HRTF y realizar distintas pruebas, para después, programar el plugin final.

Siguiendo esta idea, empezaremos con el programa Matlab, con el que se ha realizado una primera pequeña aproximación al objetivo final, y seguiremos con la programación del plugin en el lenguaje de programación C#, mediante el entorno de desarrollo de Visual Studio. Éste último plugin será el que importemos finalmente en una escena creada dentro de Unity.

3.1 MATLAB

Matlab es un lenguaje de alto nivel y un entorno interactivo para el cálculo numérico, la visualización y la programación. La elección de este programa está motivada por su gran sencillez de uso, la posibilidad de representar gráficamente las variables, y el hecho de tener muchas funciones importantes ya implementadas de forma predeterminada como son la transformada de Fourier directa e inversa, o la triangulación de Delaunay.

El desarrollo en Matlab se ha llevado a cabo realizando pequeños hitos, para poco a poco llegar al objetivo final.

3.1.1 PRIMERA FASE

La primera fase consiste en introducirnos en el mundo de las HRTF, aprender qué son, cómo se utilizan y para qué sirven. Como ya se ha explicado dentro del Apartado 2.1, existen varias bases de datos HRTF distintas, y en este trabajo se ha optado por la CIPIC.

La ventaja principal de la CIPIC HRTF Database es que está implementada en lenguaje Matlab, y por lo tanto podemos hacer uso de cualquiera de los filtros que hay disponibles para cada sujeto. (En el desarrollo del plugin únicamente se ha traducido la base de datos del sujeto número 18, ya que hacerlo con todos era inviable). La diferencia entre un sujeto y otro está en que cada persona se sentirá más o menos identificada con cada uno de ellos en función de sus características físicas y en consonancia, apreciará mejor o peor el sonido binaural. La importación dentro de Matlab se realiza con la siguiente sentencia:

```
load 'subject_018\hrir_final.mat';
```

Una vez cargada la base de datos, solo falta obtener los filtros que buscamos en función del acimut y de la elevación. Recordar que se deben obtener por separado los filtros correspondientes a cada uno de los canales, puesto que las orejas de las personas no son perfectamente simétricas y hay ligeras diferencias entre los filtros:

```
h_l = squeeze( hrir_l( nAcimut, nElevacion, : ) );
```

En las sentencias anteriores hay que tener en cuenta que los valores de acimut y elevación que introduzcamos deben corresponderse con alguno de los que ofrece la base de datos, ya que si no saltará un error. También hay que tener en cuenta que no hay que introducir el valor de acimut y elevación en grados, sino que hay que indicar la posición que ocupa la elevación y el acimut deseados dentro del vector de acimuts y elevaciones. Dicho vector consta de 25 valores comprendidos entre -80° y 80° en acimut, y de 50 valores comprendidos entre -45° y 230’625° en elevación.

Una vez obtenidos ambos filtros, filtramos la señal de audio completa para simular que una fuente de audio se encuentra en una posición concreta y estática del espacio. Para filtrar la señal se ha tenido en cuenta dos posibles métodos. El primero de ellos se basa en la convolución de la señal de audio con el filtro HRTF, mientras que el segundo método consiste en hacer uso de la transformada Fourier y multiplicar ambas señales en el dominio transformado. Es decir:

```
y_l = conv (audio, h_l);                    y_l = ifft(fft(audio)*fft(h_l));
```

Finalmente, cuando ya se han filtrado ambos canales, hay que unirlos dentro de una única variable para poder reproducirlo en formato estéreo.

3.1.2 SEGUNDA FASE

En la primera fase, se aplicó un único filtro a toda la señal de audio por lo que el resultado era la simulación de una fuente estática, sin movimiento. En la segunda fase pasamos a aplicar varios filtros a lo largo de la señal con el objetivo de obtener un efecto de movimiento de la fuente sonora. Para conseguirlo, debemos trocear la señal en pequeños trozos, llamados *frames*, y aplicarle a cada uno de ellos un filtro distinto.

Las primeras pruebas se basaron en movimientos circulares, tanto horizontales como verticales alrededor del emisor. Al reproducir estos movimientos, se podía apreciar perfectamente el desplazamiento de la fuente de sonido, pero aparecía un pequeño ruido en forma de 'clicks' o golpes cada vez que había un cambio de filtro. Este problema debido a las discontinuidades ha sido bastante importante a lo largo del trabajo ya que costó bastante dar con la solución de forma efectiva, y por eso se explica detalladamente en el Apartado 2.3. En dicho apartado también se explica el método seguido para corregir el problema, y que finalmente consistió en aplicar un "Weighted Overlap-Add" usando la ventana de Hanning.

3.1.3 TERCERA FASE

Una vez conseguido filtrar una señal de audio con varios filtros distintos proporcionando algo de movimiento a la fuente, es el momento de realizar el objetivo final del trabajo, la interpolación de los filtros. El método seguido se explica en el Apartado 2.4.

Para realizar esta tarea, se ha creado dentro de Matlab una función llamada "interpolador". Dicha función es ejecutada por otra función a la que se ha llamado "Audio3D" y que es la encargada de realizar el Overlap-Add y el enventanado. Ambos archivos se pueden encontrar en los ficheros adjuntos de este trabajo junto a un script con el nombre "main".

Los resultados obtenidos con este método de interpolación junto al Overlap-Add y el enventanado con la ventana de Hanning, son bastante satisfactorios, dando como resultado un sonido binaural muy convincente y libre de ruidos. En la programación en C# se adoptará el modelo de programación seguido en estos ficheros de Matlab.

3.2 VISUAL STUDIO

Microsoft Visual Studio es un entorno de desarrollo integrado (IDE) para sistemas operativos Windows. Soporta múltiples lenguajes de programación tales como C++, C#, Visual Basic .NET, F#, Java, Python, Ruqby, PHP.

Visual Studio permite a los desarrolladores crear aplicaciones, sitios y aplicaciones web, así como servicios web en cualquier entorno que soporte la plataforma .NET. Así se pueden crear aplicaciones que se comuniquen entre estaciones de trabajo, páginas web, dispositivos móviles, dispositivos embebidos, consolas, etc. (Wikipedia. Microsoft Visual Studio)

3.2.1 ESTRUCTURA DEL PLUGIN

El objetivo principal en Visual Studio, es la de desarrollar un plugin que podamos importar en Unity3D y que procese audio binaural en tiempo real. Para ello, vamos a seguir la estructura utilizada anteriormente en Matlab, con la diferencia de que ahora no disponemos de funciones como la convolución o la transformada de Fourier. Para poder realizar dichos cálculos, vamos a hacer uso de la librería Alglib (ALGLIB® - numerical analysis library, 1999-2014.), la cual ofrece una enorme cantidad de funciones para el cálculo matemático de todo tipo. Esta librería tiene licencia gratuita y se encuentra disponible en Internet.

El leguaje elegido para la programación del plugin dentro de Visual Studio es C#. El motivo de la elección de este lenguaje frente a C++, se basa principalmente en el hecho de que es necesario disponer de la versión Unity PRO para poder importar un plugin que esté basado en C++. En cambio un plugin que esté programado en C# se podrá importar tanto en la versión Indie como en la PRO.

En la Figura 3.1, se muestra la estructura que se ha seguido para elaborar el plugin. En dicha figura, se han representado tanto las clases como los métodos, diferenciándose en que las primeras están coloreadas en naranja claro, mientras que las segundas tienen un color azulado. Las dos clases llamadas “DatabaseIzq” y “DatabasDer”, están destinadas a almacenar los filtros HRTF de la base de datos CIPIC correspondientes al sujeto 18. Recordar que la base de datos estaba disponible en lenguaje Matlab, y se ha tenido que traducir al lenguaje C#, siendo éste el motivo de que sólo se encuentren los filtros correspondientes a un sujeto.

Para poder acceder a los filtros almacenados dentro de “DatabaseIzq” y de “DatabaseDer” desde otras clases, se ha creado el método público “getFiltro”.

La tercera clase en cuestión, llamada “Audio3D” es la clase principal, y contiene el método “aplicarAudio3D()” el cuál se encarga internamente de invocar al resto de métodos necesarios para llevar a cabo el procesado del audio binaural.

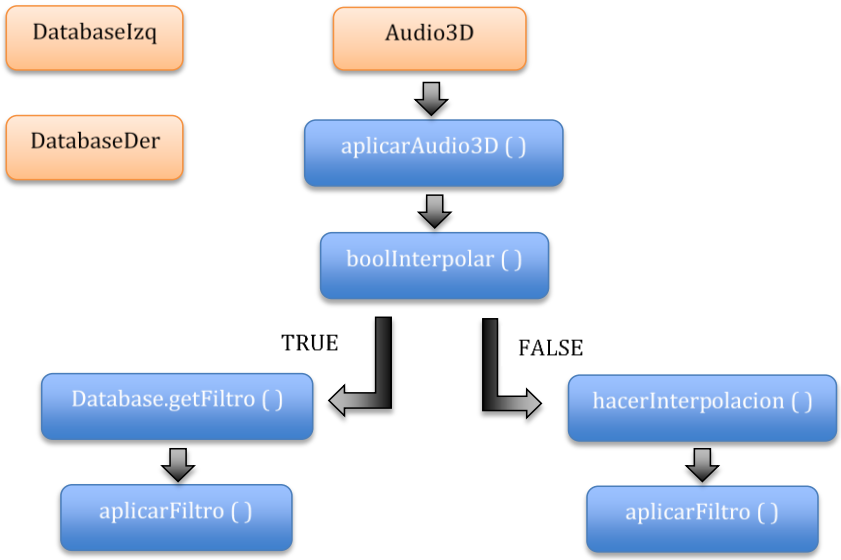


Figura 3.1.- Estructura seguida en la programación del plugin. El color naranja representa las Clases, mientras que el azul representa métodos de la Clase.

A continuación se explica brevemente la función que desempeña cada uno de los métodos que se han creado dentro del plugin:

- **aplicarAudio3D (ref audio, acimut, elevación, audioNext, out cola):** método público encargado de llevar a cabo el procesado del sonido binaural.
  Parámetros:
  I. **ref audio:** Vector de floats pasado por referencia que contiene el audio a procesar. Devuelve el sonido ya filtrado.
  II. **acimut:** Entero que contiene el valor del acimut en grados.
  III. **elevacion:** Entero que contiene el valor de la elevación en grados.
  IV. **audioNext:** Vector de floats que contiene el frame de audio posterior.
  V. **out cola:** Devuelve el trozo de señal que habrá que sumar al siguiente frame para realizar el "Overlap-Add".

  Éste método, tal como se ve en el esquema anterior, llama a la función "boolInterpolar" y según su resultado, realiza unas acciones u otras, hasta obtener el filtro adecuado, y devolver el audio debidamente filtrado.

- **interpolar = boolInterpolar (acimut, elevacion, out nAcimut, out nElevacion):** método privado, encargado de comprobar si es necesario realizar la interpolación.
  Parámetros:
  I. **interpolar:** Booleano que vale 'true' en caso de ser necesaria la interpolación. De no ser necesario, valdría 'false'.
  II. **acimut:** Entero que contiene el valor del acimut en grados.
  III. **elevacion:** Entero que contiene el valor de la elevación en grados.
  IV. **nAcimut:** Entero que contiene el índice dentro del vector de acimuts disponibles dentro de las HRTF.

V. **nElevacion:** Entero que contiene el índice dentro del vector de acimuts disponibles dentro de las HRTF.

- **hacerInterpolacion (acimut, elevacion, out filtroIzq, out filtroDer):** método de carácter privado, utilizado por el método "aplicarAudio3D" para realizar el algoritmo de interpolación y obtener el filtro adecuado.
  Parámetros:
  I. **acimut:** Entero que contiene el valor del acimut en grados.
  II. **elevacion:** Entero que contiene el valor de la elevación en grados.
  III. **filtroIzq:** Vector que contiene el filtro interpolado del canal izquierdo.
  IV. **filtroDer:** Vector que contiene el filtro interpolado del canal derecho.

  Internamente realiza los algoritmos descritos en el Apartado 2.4, correspondientes a la triangulación de Delaunay, búsqueda del triángulo que encuadra al punto objetivo, cálculo de los pesos, y obtención del filtro interpolado.

- **aplicarFiltro (ref audio, filtroIzq, filtroDer):** método de carácter privado, utilizado en última instancia para aplicar el filtro adecuado a la señal de audio correspondiente.
  Parámetros:
  I. **audio:** Vector que contiene el audio a procesar.
  II. **filtroIzq:** Vector que contiene el filtro a aplicar en el canal izquierdo.
  III. **filtroDer:** Vector que contiene el filtro a aplicar en el canal derecho.

  El filtrado se puede realizar tanto con una convolución como con una multiplicación en el dominio transformado de Fourier. Para poder realizar dichas operaciones, se ha contado con la plataforma de análisis numérico y biblioteca de extracción de datos *Alglib* (ALGLIB® - numerical analysis library, 1999-2014.), la cual se encuentra disponible en varios lenguajes de programación (C++, C#, Pascal, VBA) y cuenta con licencia gratuita.

Una vez que se ha programado todo lo descrito anteriormente, llega el momento de compilar el código y generar el plugin que usaremos dentro de Unity.

3.3 UNITY

Unity es un ecosistema de desarrollo de juegos: un poderoso motor de renderizado totalmente integrado con un conjunto completo de herramientas intuitivas y flujos de trabajo rápido para crear contenido 3D interactivo; publicación multiplataforma sencilla; miles de activos de calidad, listos para usar en la Tienda de Activos y una Comunidad donde se intercambian conocimientos (Unity).

3.3.1 PRIMEROS PASOS

Una vez creado el plugin destinado a procesar el audio binaural, es el momento de crear un proyecto en Unity con el que poder testear y probar su correcto funcionamiento. Es importante destacar que Unity es una plataforma que cuenta con una comunidad de gente muy activa a la hora de compartir, y ayudarse mutuamente en la elaboración y programación dentro de los proyectos. Esto hace que haya un montón de videos y de foros con tutoriales y referencias de código que ayudan de gran manera a iniciarnos en este mundo.

Con todas estas ayudas, y después de algún proyecto de prueba, se abordó la creación del proyecto final con un modelo de carácter sencillo, y usando siempre formas predefinidas o de forma gratuita dentro de la UnityStore.

3.3.2 INTEGRACIÓN DE LA DLL

Una vez creada la escena, pasamos a integrar nuestro plugin dentro del proyecto. Como ya se ha comentado anteriormente, el propio Unity ofrece una gran cantidad de información y tutoriales acerca de su plataforma, además de disponer de un foro oficial en el que personas de todo el mundo participan activamente resolviendo las dudas de lo demás por muy simples que sean. Gracias a todo esto, y al propio canal de Unity en Youtube (www.youtube.com), en el que se cuelgan una gran variedad de video tutoriales, la integración del plugin en el proyecto ha sido bastante sencilla.

La fórmula consiste en crear una carpeta llamada ‘plugins’ dentro del directorio ‘Assets’, e incluir el plugin en su interior. Después se crea un script en lenguaje C#, y siguiendo las indicaciones de la página de referencia de Unity, se importa la librería con el siguiente comando:

```
Proyecto.Audio3D miAudio = new Proyecto.Audio3D();
```

Con esta sentencia lo que estamos haciendo es declarar una variable de la clase "Audio3D" con la que podremos invocar al método "aplicarAudio3D"

```
miAudio.aplicarAudio3D( ref audioActual, acimut, elevación, audioNext, out cola );
```

El motivo de incluir como parámetros tanto el "audioActual" como el "audioNext", junto a una variable "cola", se debe al método "Weighted Overlap-Add" explicado en el Apartado 2.3.2. Dichas variables, "audioActual" y "audioNext" hacen referencia tal y como indican sus nombres, a los vectores que contienden el frame de audio actual, y el inmediatamente posterior. Como en Unity nos encontramos con audio en tiempo real, no es posible obtener el frame futuro de forma directa, sino que tenemos que aplicar un delay de un frame de audio dentro de nuestro script.

La variable "cola" es un parámetro de salida del método "aplicarAudio3D" que contiene las muestras sobrantes de realizar la convolución del frame actual con el filtro HRTF interpolado, ya que hay que recordar que el tamaño del vector de salida en una convolución es igual a la suma de los tamaños de las dos variables de entrada, menos uno. Es decir, si tenemos conv(A,B) = C, el tamaño de C, será igual a tam(A)+tam(B)-1. Como el tamaño del frame de audio es de 1024 muestras, y el vector de audio de salida será mayor que esas 1024 muestras, las sobrantes se guardan en la variable "cola". Dicha variable ya se encuentra apropiadamente enventanada, y habrá que sumarla dentro del propio script de Unity, a las muestras iniciales del próximo frame, para así realizar el "Weighted Overlap-Add".

3.3.3 CÁLCULO DEL ACIMUT Y DE LA ELEVACIÓN

Al principio, los valores de acimut y elevación que introducíamos en la llamada al método "aplicarAudio3D", se escogían de tal forma que se realizara un barrido circular alrededor del oyente manteniendo la elevación constante y variando el acimut. Después se repetía el procedimiento variando los valores de elevación pero manteniendo los valores de acimut. Esto se hacía estableciendo los valores dentro del propio script antes de ejecutarlo.

Tras esas primeras pruebas, se pasó a obtener en tiempo real los valores de acimut y elevación de un objeto respecto a la "MainCamera" que es donde se encuentra el "AudioListener".

Debido a que tiene que ser en tiempo real, vamos a calcular continuamente los valores de acimut y elevación dentro de la función Update(). Esta función se ejecuta en cada frame de imagen, por lo que siempre tendremos los valores actualizados para su uso dentro de la función "OnAudioFilterRead" la cual se activa cada 20 milisegundos.

El proyecto creado en Unity, se ha diseñado con un "first person controller", lo que quiere decir que el usuario maneja un personaje en primera persona, teniendo la capacidad de mover la cámara a su antojo. Esto repercute en que el cálculo del acimut y de la elevación no hay que hacerlo respecto al propio muñeco, sino que también hay que tener en cuenta el movimiento de la cámara.

Sabiendo esto, el cálculo del acimut y la elevación se basa en obtener el vector que apuntadesde nuestra “MainCamera” hasta el objeto que contiene el “AudioSource”. Una vez obtenido dicho vector, usando relaciones trigonométricas podemos calcular tanto θ como ϕ, que se corresponden respectivamente a la elevación y el acimut.

Para conocer el vector que apunta desde nuestra cámara hasta el objeto en cuestión, Unity dispone de una función llamada “InverseTransformPoint” dentro de la clase “Transform”.

Como parámetro hay que introducir el “GameObject” al que está asociado el “AudioSource”. El resultado es una variable de tipo “Vector3”.

```
VectorRelativo = Camera.main.transform.InverseTransformPoint(transform.position);
```

3.3.4 PROYECTO FINAL EN UNITY

El proyecto final se ha dividido en 2 escenas, sin contar el menú interactivo encargado de seleccionar una u otra. Ambas escenas cuentan con un único “AudioSource” y un único “AudioListener”. La diferencia radica en que en la primera escena, el “AudioSource” se encuentra fijo y sin ningún tipo de movimiento, y el objetivo es encontrarlo guiándote por el audio que él mismo genera. La segunda escena consta de un “AudioSource” asociado a un helicóptero que se desplaza de forma continua y predeterminada a lo largo del mapa.

En ambas escenas el “AudioListener” está asociado al usuario, concretamente a la “MainCamera”.

3.4 CONCLUSIÓN FINAL DEL TRABAJO

Como ya se comentó en el apartado de Matlab, al usar varios filtros distintos sobre una señal dividida en trozos, aparece el problema de las discontinuidades en cada cambio de filtro HRTF. Estas discontinuidades se traducen en ‘clicks’ o golpes que resultan muy molestos auditivamente.

La solución aportada fue el “Weighted Overlap-add”, con la dificultad de que en Unity, estamos procesando audio en tiempo real con la función “OnAudioFilterRead”, por lo que únicamente disponemos de las muestras correspondientes al frame de audio actual. La solución

adoptada es retrasar un frame completo el procesado del audio, de tal forma que el frame de audio que tratamos y devolvemos a la cadena de audio, no es el que nos da en ese instante la función “OnAudioFilterRead”, sino que es la que nos dio en el anterior ciclo, y que hemos guardado en una variable auxiliar.

Hay que destacar también que al usar la función “OnAudioFilterRead” estamos cogiendo las muestras del buffer de salida de audio, por lo que en caso de haber más de un “AudioSource” en la escena, el buffer tendría los sonidos de todos los “AudioSources” ya mezclados. Esto es una limitación, ya que el plugin creado procesa el audio binaural respecto a una dupla de acimut y elevación concreta determinada por un único “GameObject”.

Si quisiéramos tratar audio binaural sobre varios objetos distintos, deberíamos aplicar nuestro plugin sobre cada uno de ellos por separado, y siempre antes de que hayan sido mezclados. Esto se podría intentar realizar con las funciones “getData()” y “setData()” explicadas en la parte de Anexos, pero su implementación se deja para posteriores proyectos.

Bibliografía

ALGLIB® - numerical analysis library, 1999-2014. s.f. http://www.alglib.net/.

Beltrán Blázquez, JoséRamón. «Audio Inmersivo para entornos de realidad virtual.» *Oculus day*.

CIPIC- Center for Image Processing and Integrated Computing University of California 1 Shields Avenue Davis, CA 95616-8553. *The CIPIC HRTF Database*. http://interface.cipic.ucdavis.edu/sound/hrtf.html.

Departamento de Matemática Aplicada. Universidad Politécnica de Madrid. *Punto interior de un triángulo*. http://www.dma.fi.upm.es/mabellanas/tfcs/kirkpatrick/Aplicacion/algoritmos.htm#puntoInterior.

Gamper, Hannes. «Selection and Interpolation of head-realted transfer functions for rendering moving virtual sound sources.» *Proc. of the 16th Int. Conference on Digital Audio Effects (DAFx-13), Maynooth, Ireland, September 2-5, 2013*.

Gardner, Bill, y Keith Martin. *HRTF Measurements of a KEMAR Dummy-Head Microphone.* s.f. http://sound.media.mit.edu/resources/KEMAR.html.

Mengqiu Zhang, Wen Zhang, Rodney A. Kennedy, and Thushara D. Abhayapala. «HRTF measurement on KEMAR manikin.» 23–25 November 2009.

Smith, Julius Orion. *Weighted Overlap Add*. http://www.dsprelated.com/dspbooks/sasp/Weighted_Overlap_Add.html.

Unity. s.f. www.unity3d.com.

—. *Unity. OnAudioFilterRead*. https://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.OnAudioFilterRead.html.

Wikipedia. Escucha Binaural. http://es.wikipedia.org/wiki/Escucha_binaural.

Wikipedia. Microsoft Visual Studio. http://es.wikipedia.org/wiki/Microsoft_Visual_Studio.

Wikipedia. Overlap-Add. s.f. http://en.wikipedia.org/wiki/Overlap_add.

Wikipedia. Triangulación de Delaunay. http://es.wikipedia.org/wiki/Triangulaci%C3%B3n_de_Delaunay.

ANEXOS

I. ARCHIVOS MATLAB

En el CD adjunto se puede encontrar el path de Matlab con los scripts finales, además de una versión intermedia que se ha querido incluir para demostrar los problemas que surgieron a mitad de trabajo y la forma de resolverlos. Todos los scripts creados están debidamente comentados para facilitar su entendimiento.

El path se divide en tres carpetas distintas. La primera de ellas, con el nombre "CIPIC_hrtf_database" contiene la base de datos CIPIC con los filtros HRTF, además de algún script en el que se muestran de forma gráfica el funcionamiento de estos filtros. Esta carpeta no ha sido modificada, y se puede descargar desde internet de forma gratuita a través del siguiente enlace:

http://interface.cipic.ucdavis.edu/sound/hrtf.html

I.I VERSIÓN INTERMEDIA

La segunda carpeta "Versión intermedia" contiene un script en el que aún no se ha incluido la interpolación de filtros, únicamente realiza el filtrado de una señal de audio con varios filtros distintos. La razón por la que se ha incluido es que en ella aparece el problema de los 'clicks' o golpes ruidosos debidos a las distintas discontinuidades. La solución a este problema es como ya se ha comentado, el método "Weighted Overlap-add".

Dentro de la carpeta hay un script y una función. Ejecutando el primero, estamos llamando internamente al segundo. Nótese que el archivo de audio por defecto es SilbidoOeste, incluido en el path, pero se puede poner otro distinto siempre y cuando sea un archivo .wav.

Los parámetros de entrada de la función son cuatro, de los que hay que destacar el segundo mediante el cual se puede elegir si queremos que la función filtre la señal de audio entera con los mismos valores de acimut y elevación, o si queremos que simule el movimiento de la fuente sonora alrededor del oyente, en cuyo caso podrá ser un barrido horizontal o un barrido vertical empezando en la parte delantera y finalizando en la parte trasera.

A continuación se incluyen gráficas tanto de la señal original como de las filtradas con el fin de visualizar los cambios generados con los filtros HRTF.

SEÑAL --> Seno.wav
Toda la señal con el mismo filtro. Acimut = 0°, Elevación = 0°.

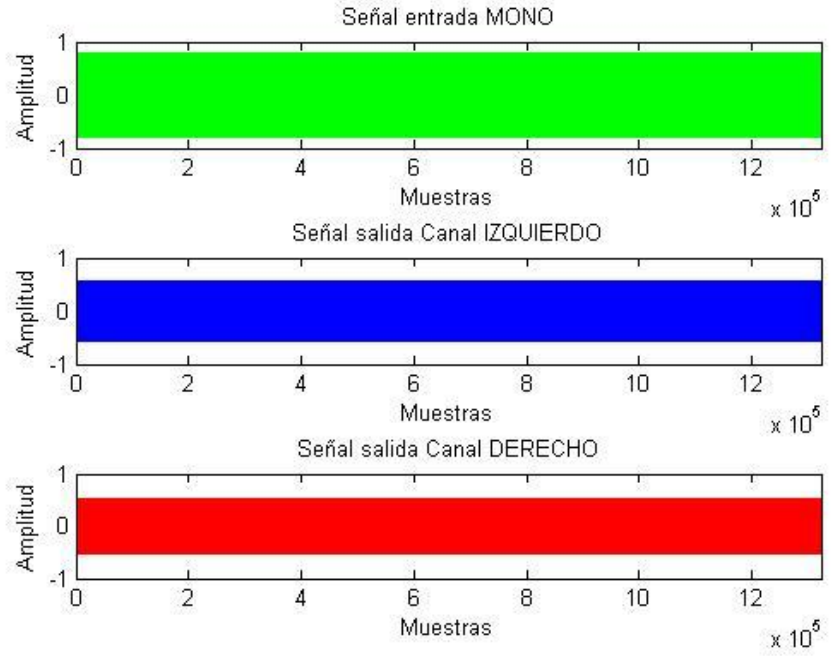


Figura I.I.1. Representación temporal.

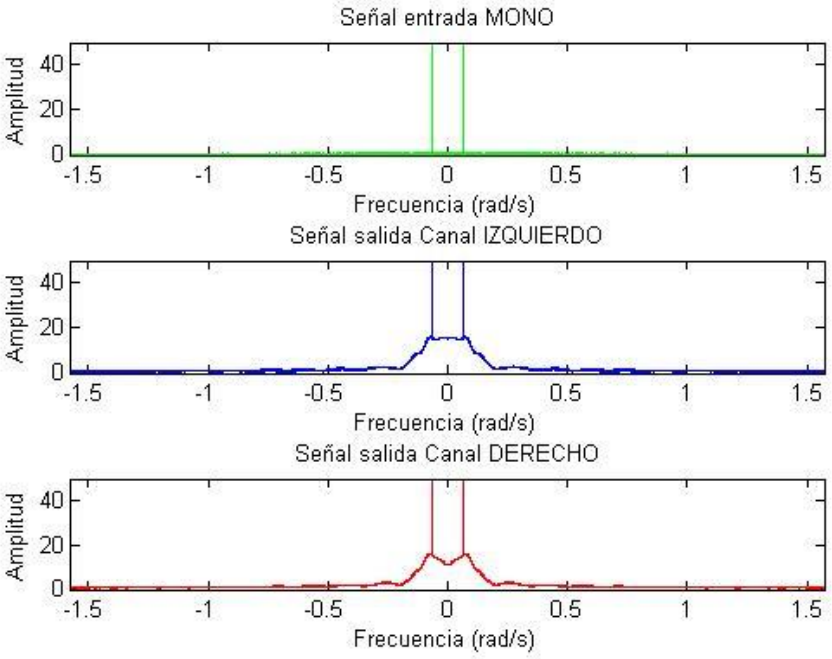


Figura I.I.2. Representación frecuencial.

SEÑAL --> Seno.wav
Toda la señal con el mismo filtro. Acimut = -65°, Elevación = 0°.

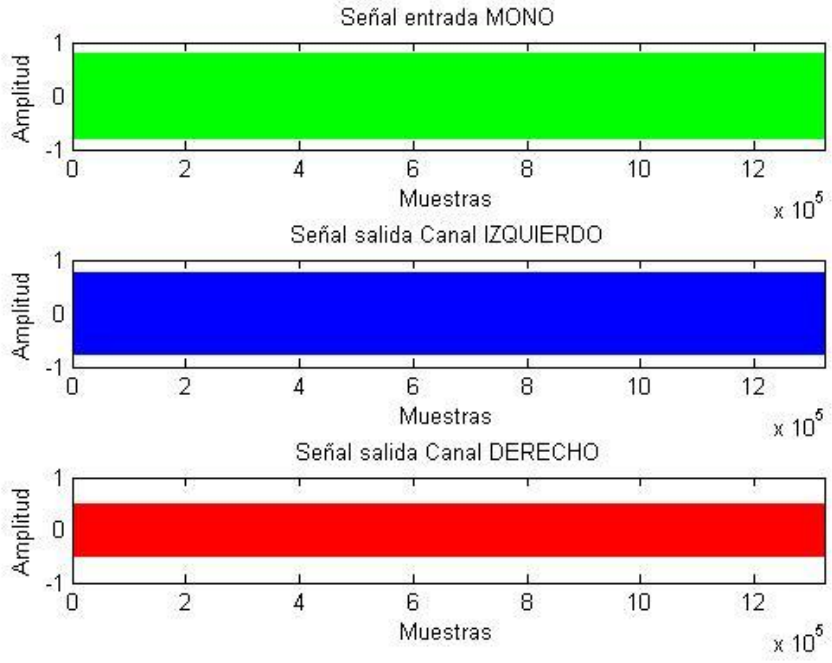


Figura I.I.3. Representación temporal.

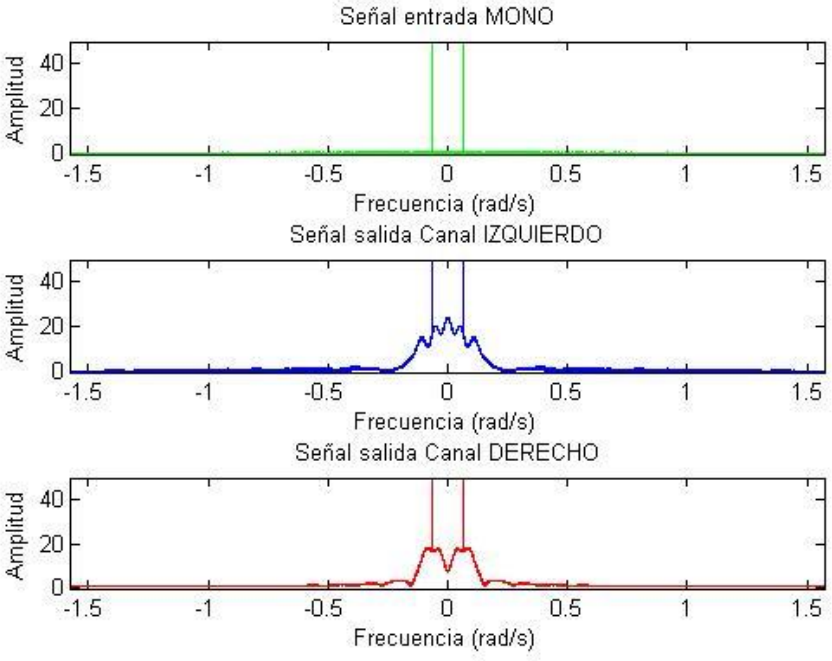


Figura I.I.4. Representación frecuencial.

SEÑAL --> Seno.wav
Toda la señal con el mismo filtro. Acimut = 0°, Elevación = 180°.

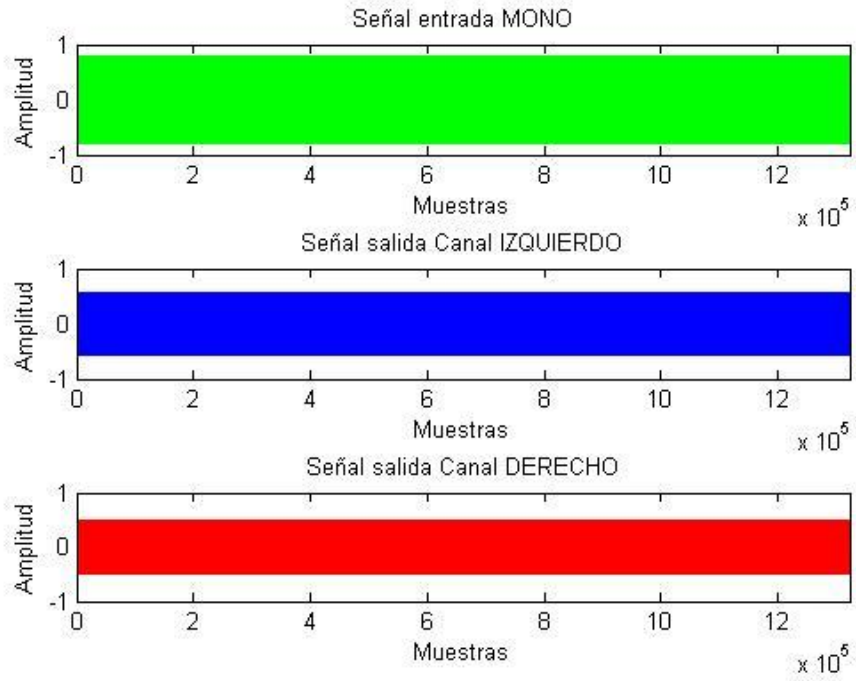


Figura I.I.5. Representación temporal.

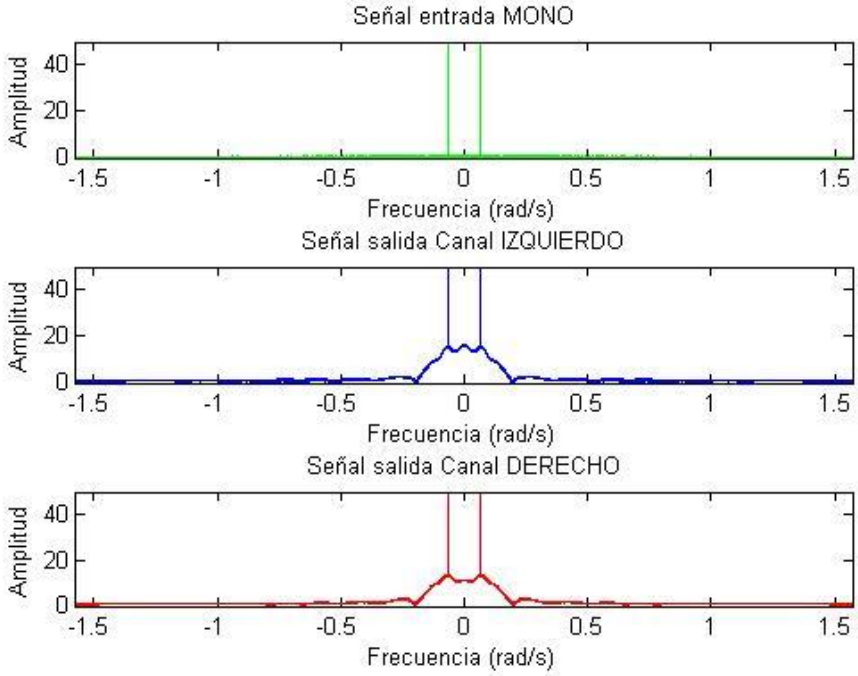


Figura I.I.6. Representación frecuencial.

SEÑAL --> SilbidoOeste.wav
Toda la señal con el mismo filtro. Acimut = 0°, Elevación = 0°.

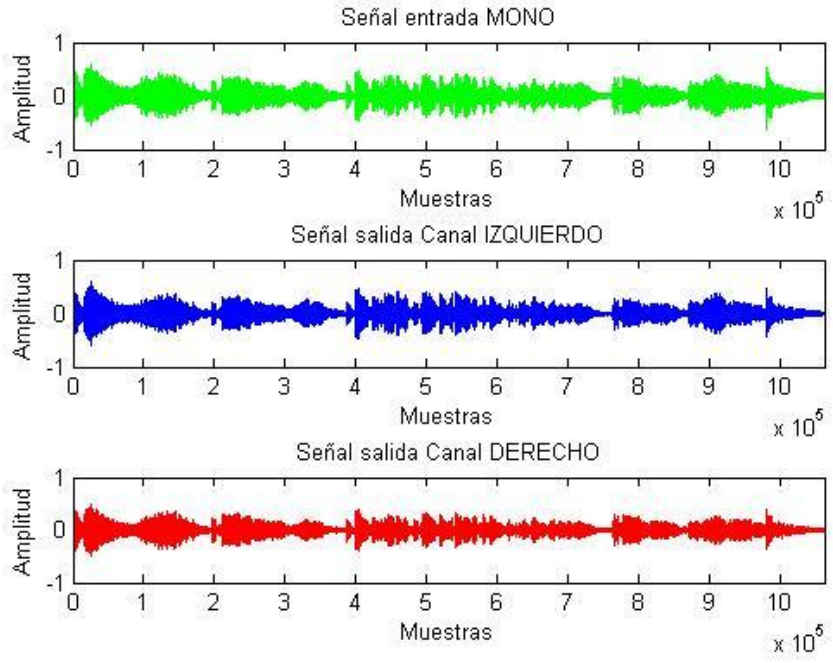


Figura I.I.7. Representación temporal.

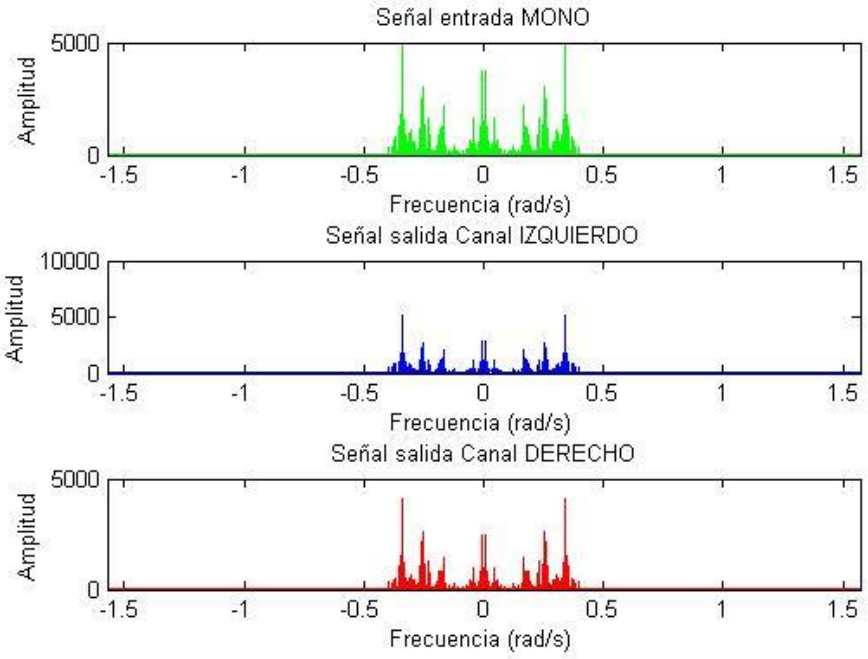


Figura I.I.8. Representación frecuencial.

SEÑAL --> SilbidoOeste.wav
Toda la señal con el mismo filtro. Acimut = 35°, Elevación = 0°.

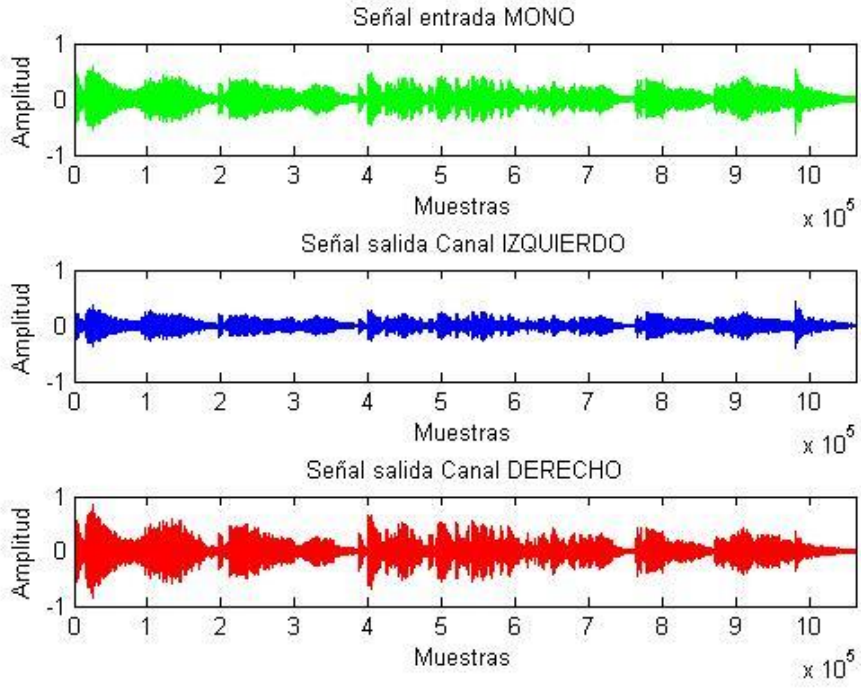


Figura I.I.9. Representación temporal.

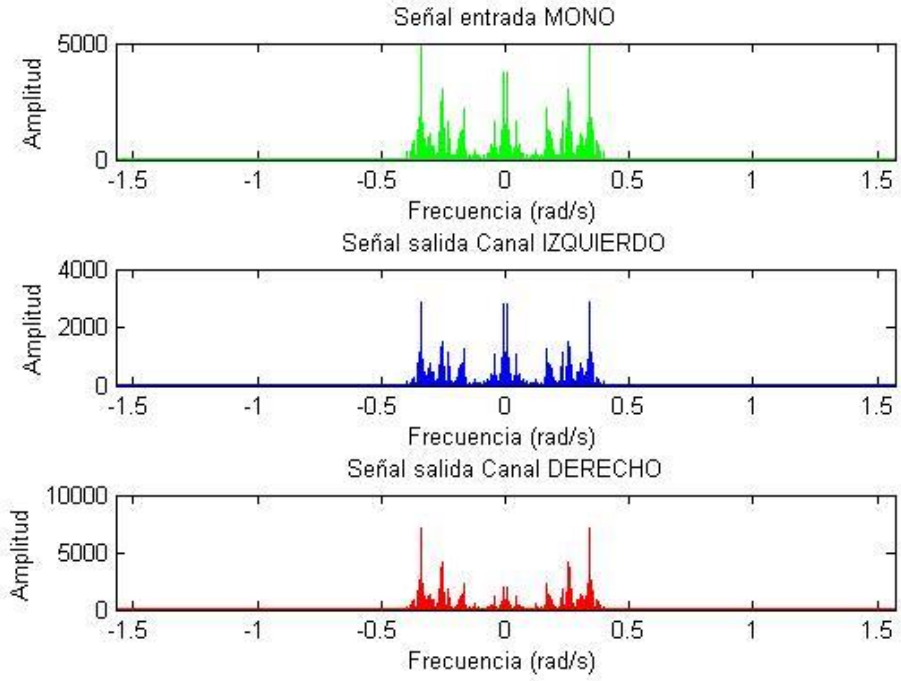


Figura I.I.10. Representación frecuencial.

SEÑAL --> SilbidoOeste.wav
Toda la señal con el mismo filtro. Acimut = 0°, Elevación = -45°.

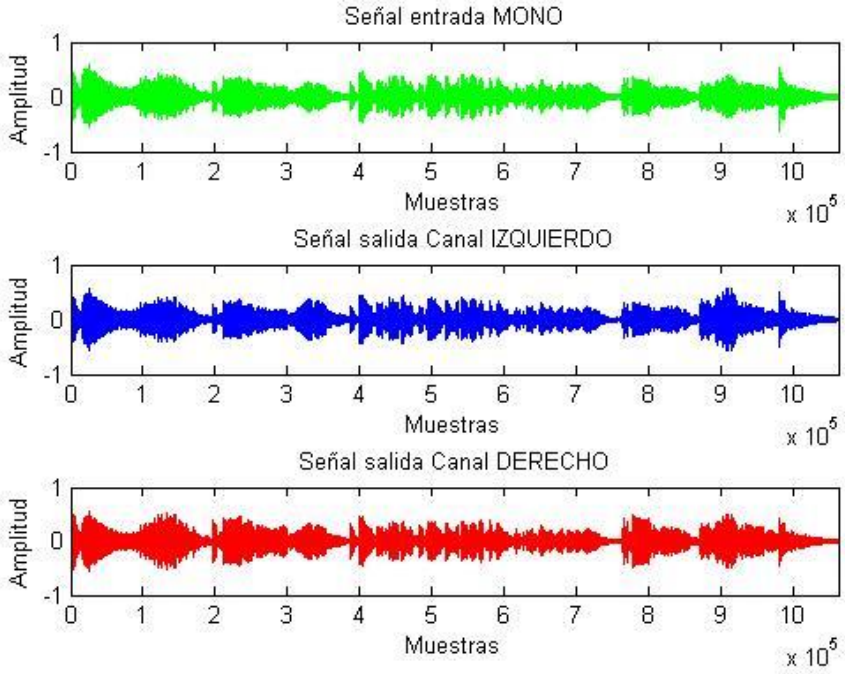


Figura I.I.11. Representación temporal.

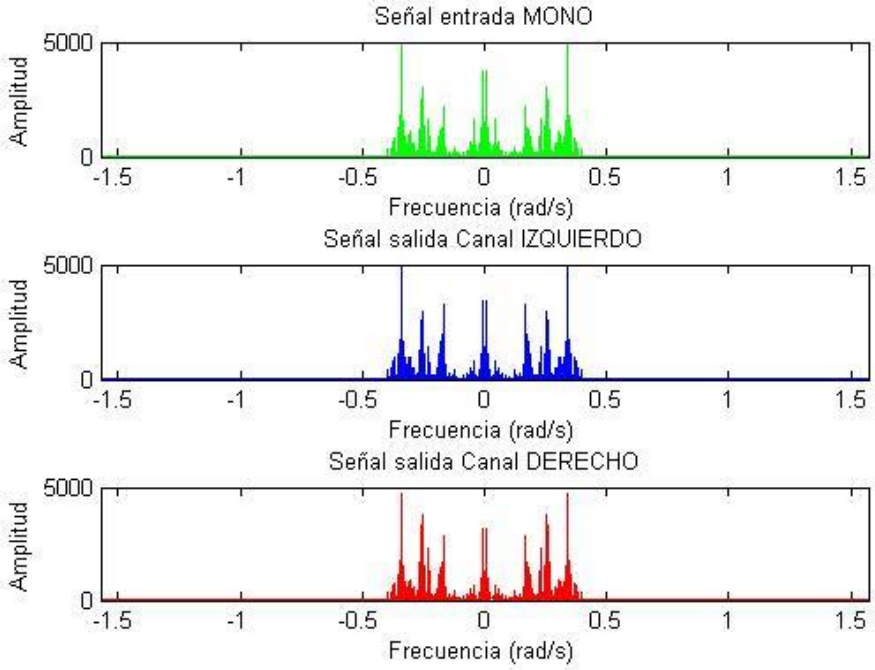


Figura I.I.12. Representación frecuencial.

SEÑAL --> Seno.wav
Recorrido vertical alrededor del oyente, empezando en -45°, y terminando en 230° pasando por 0°, 90° y 180° en elevación.

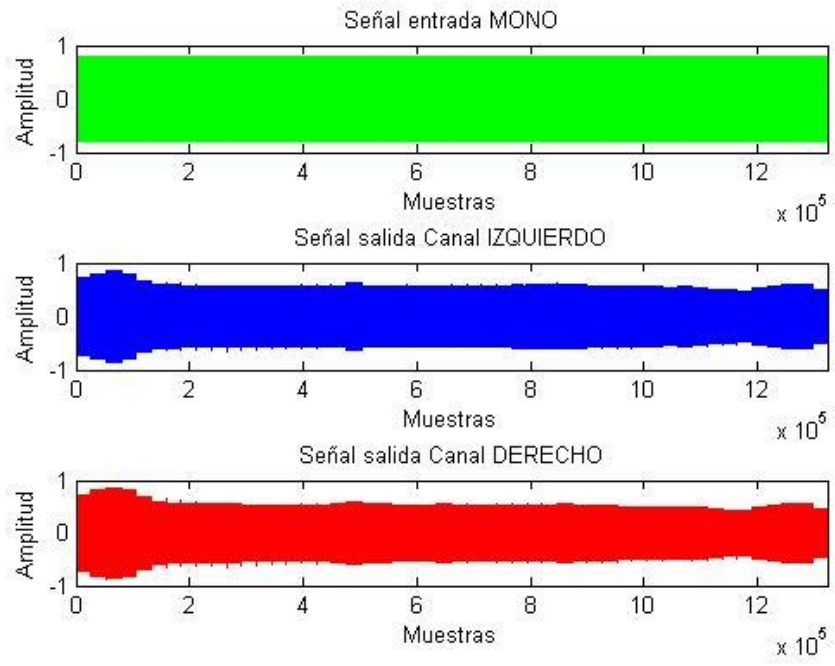


Figura I.I.13. Representación temporal.

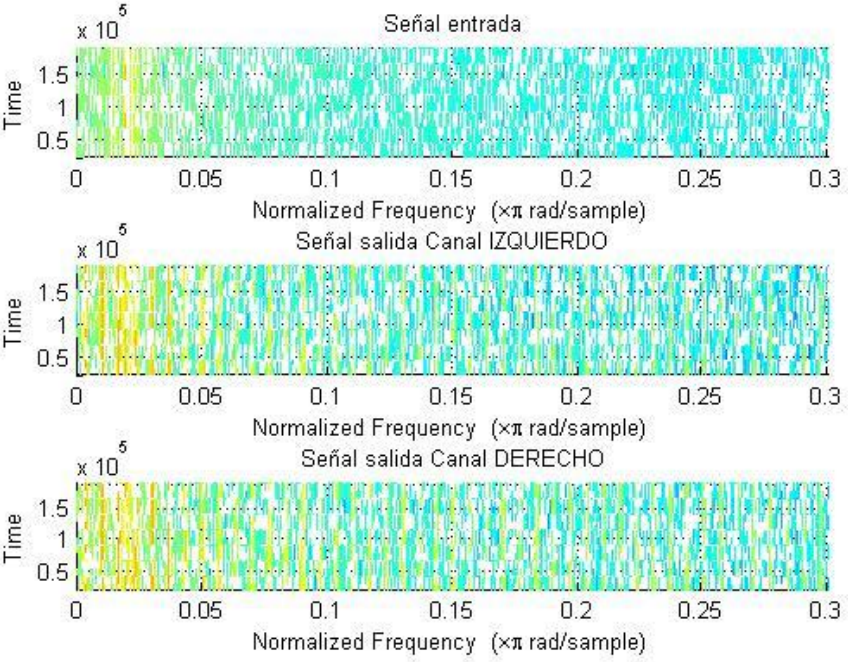


Figura I.I.14. Espectrograma.

SEÑAL --> SilbidoOeste.wav
Recorrido vertical alrededor del oyente, empezando en -45°, y terminando en 230° pasando por 0°, 90° y 180° en elevación.

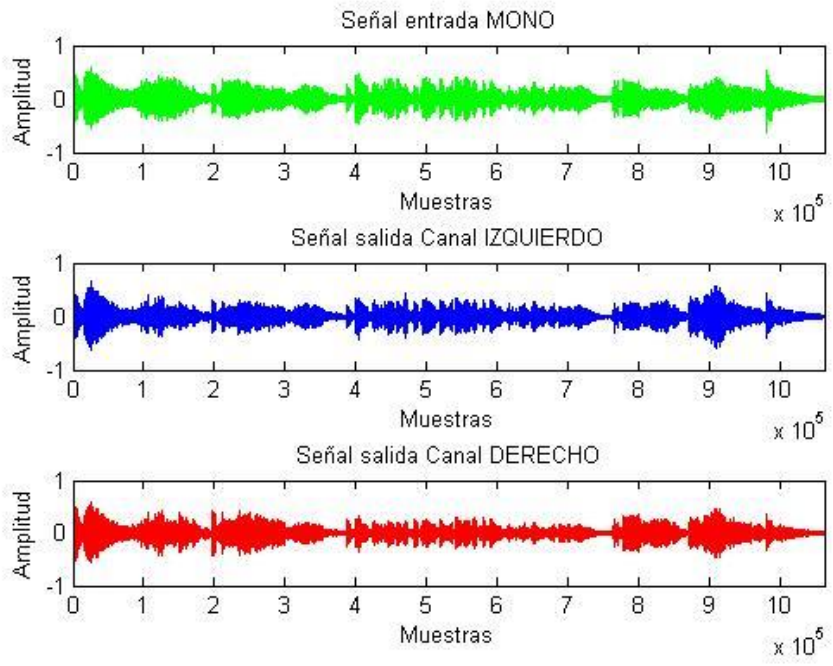


Figura I.I.15. Representación temporal.

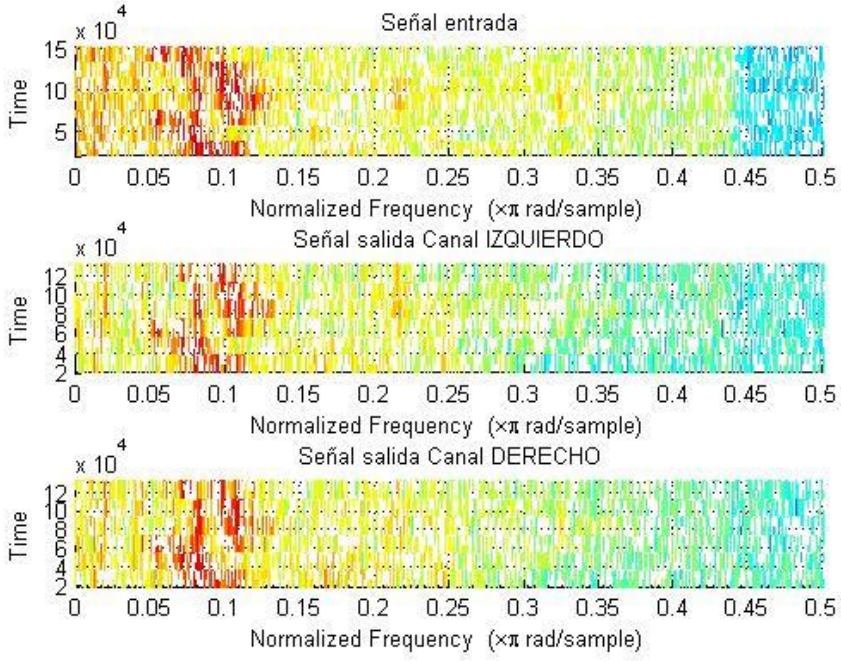


Figura I.I.16. Espectrograma.

SEÑAL --> Seno.wav
Recorrido horizontal alrededor del oyente, empezando en -80°, y terminando en -80° pasando por 0°, 80° y 180° en acimut.

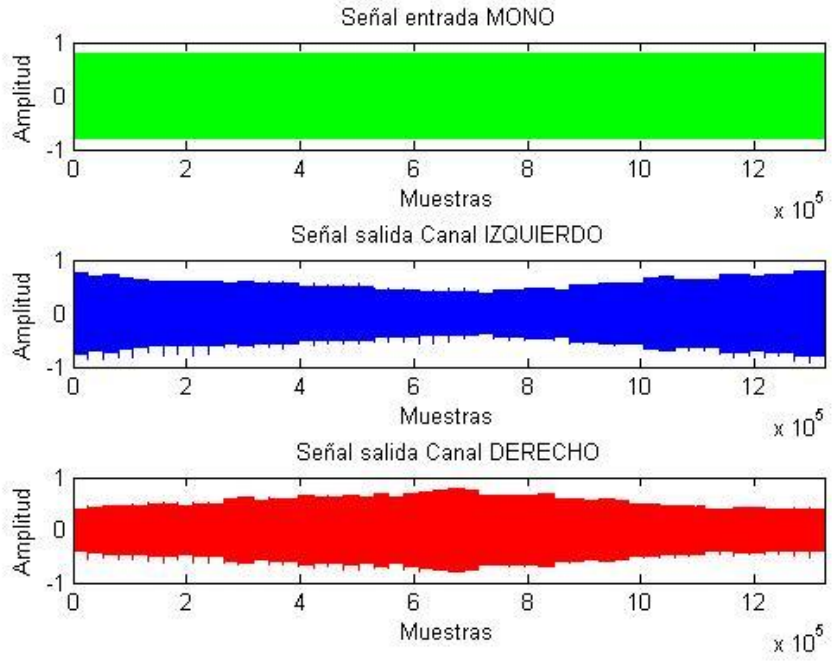


Figura I.I.17. Representación temporal.

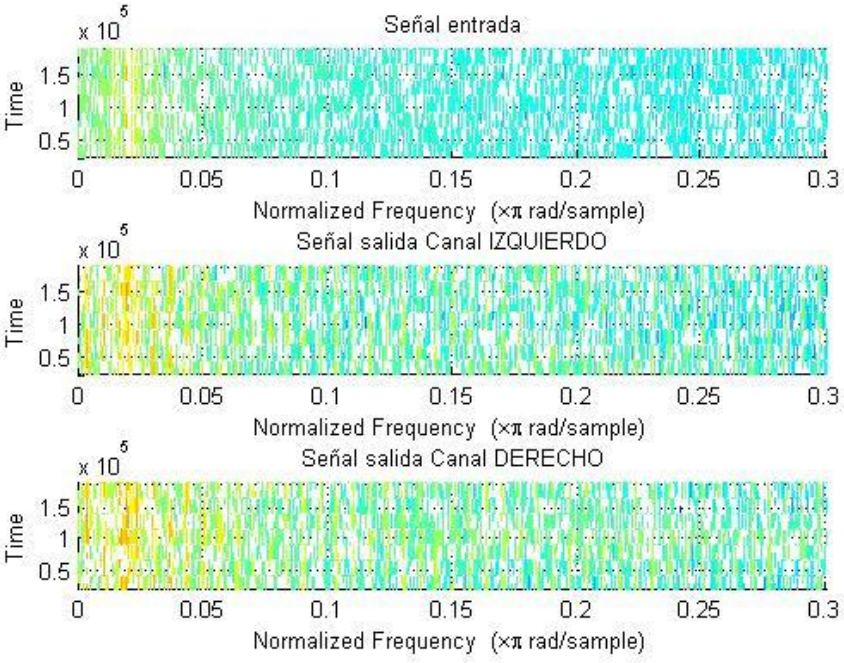


Figura I.I.18. Espectrograma.

SEÑAL --> SilbidoOeste.wav
Recorrido horizontal alrededor del oyente, empezando en -80°, y terminando en -80° pasando por 0°, 80° y 180° en acimut.

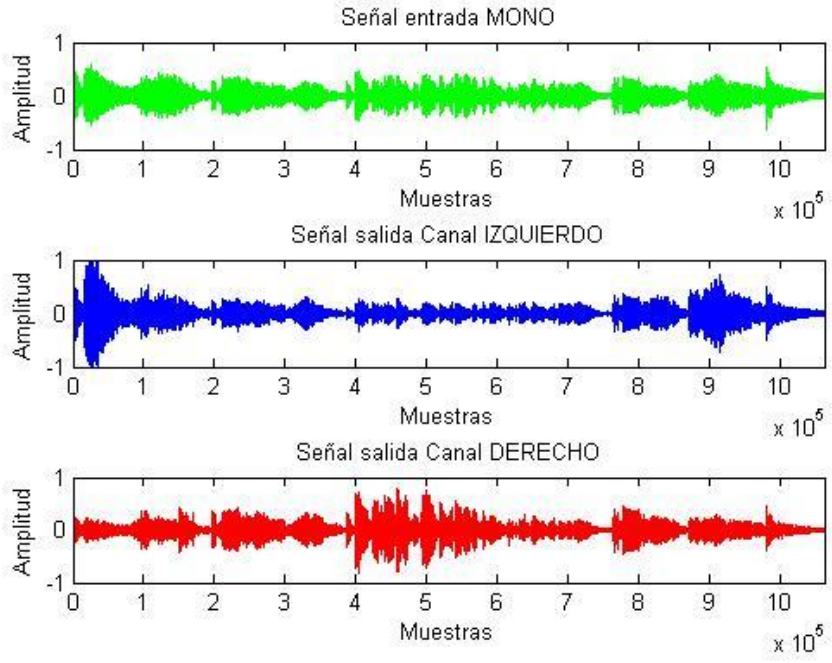


Figura I.I.19. Representación temporal.

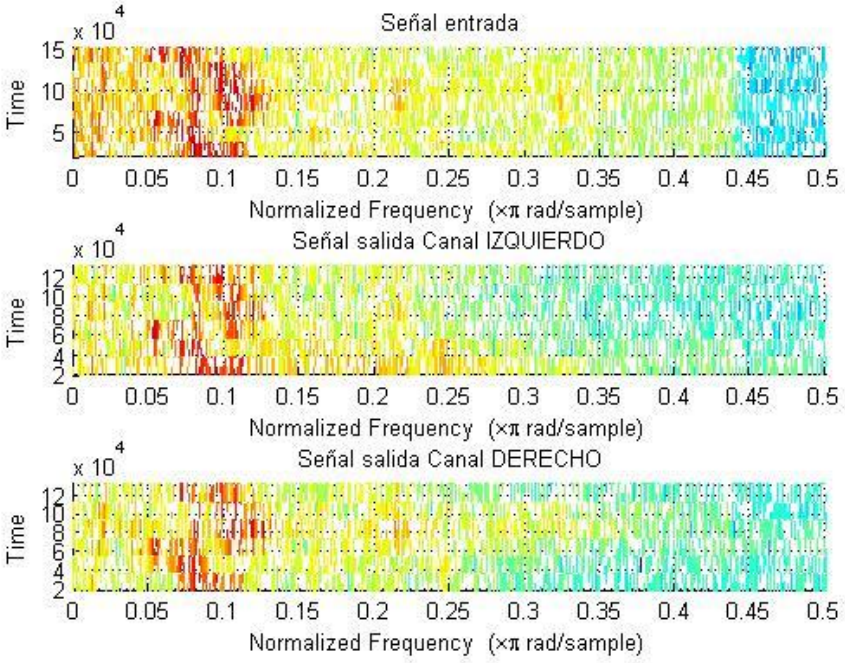


Figura I.I.20. Espectrograma.

I.II VERSIÓN FINAL

La tercera carpeta en cuestión corresponde a la versión final en la que ya se realiza la interpolación de los filtros HRTF. Dentro de la carpeta se encuentran un script llamado “main” y una función llamada “Audio3D”.

La función “Audio3D” es la encargada de realizar todo el proceso de interpolación y de filtrado de la señal. Así mismo, en su interior también se realiza la triangulación de Delaunay, aunque en el plugin basado en C# omitimos este paso, ya que el resultado de la triangulación va a ser siempre el mismo, y por lo tanto, no es necesario realizarlo continuamente. Basta con cargar el resultado obtenido previamente cuando sea necesario.

El script “main” se encarga de llamar a la función “Audio3D”, y de representar al final los resultados obtenidos en gráficas adecuadas para la tarea.

Nótese que el archivo de audio por defecto es SilbidoOeste, incluido dentro del path adjunto, pero se puede cualquier otro siempre y cuando sea un archivo .wav.

A continuación se incluyen gráficas tanto de la señal original como de las filtradas con el fin de visualizar los cambios generados con los filtros HRTF.

SEÑAL --> SilbidoOeste.wav
Recorrido horizontal alrededor del oyente, empezando en -80°, y terminando en -80° pasando por 0°, 80° y 180° en acimut.

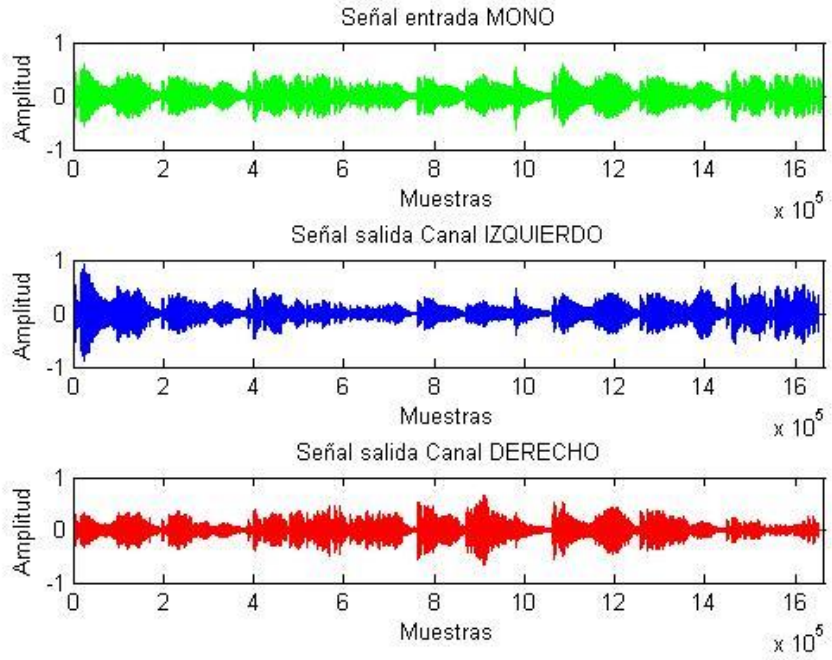


Figura I.II.1. Representación temporal.

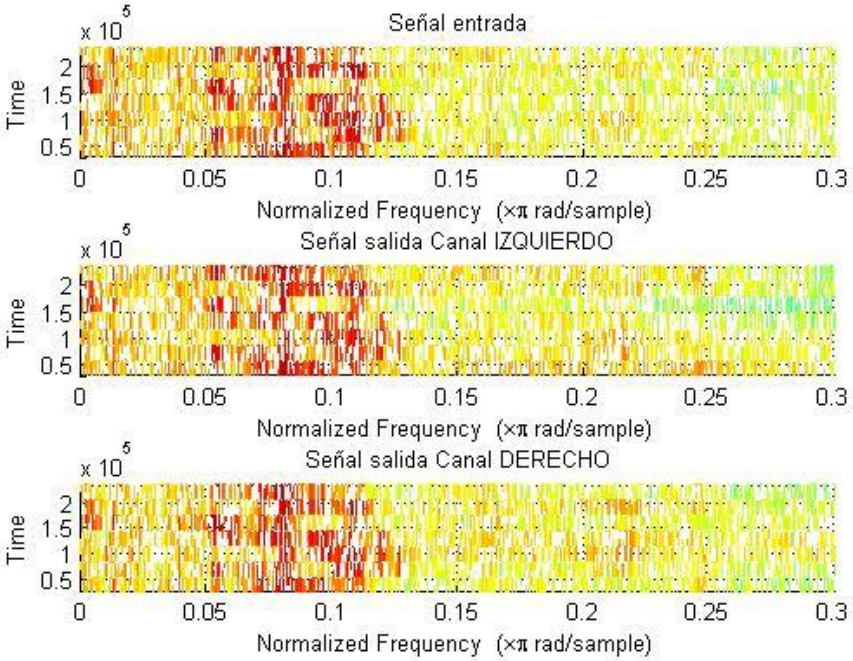


Figura I.II.2. Espectrograma.

SEÑAL --> SilbidoOeste.wav
Recorrido vertical alrededor del oyente, empezando en -45°, y terminando en 230° pasando por 0°, 90° y 180° en elevación.

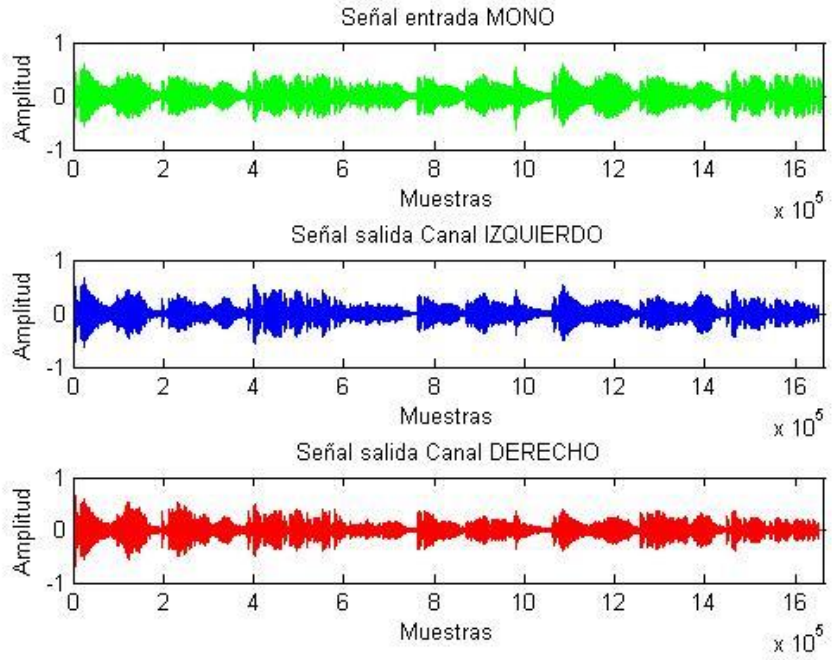


Figura I.II.3. Representación temporal.

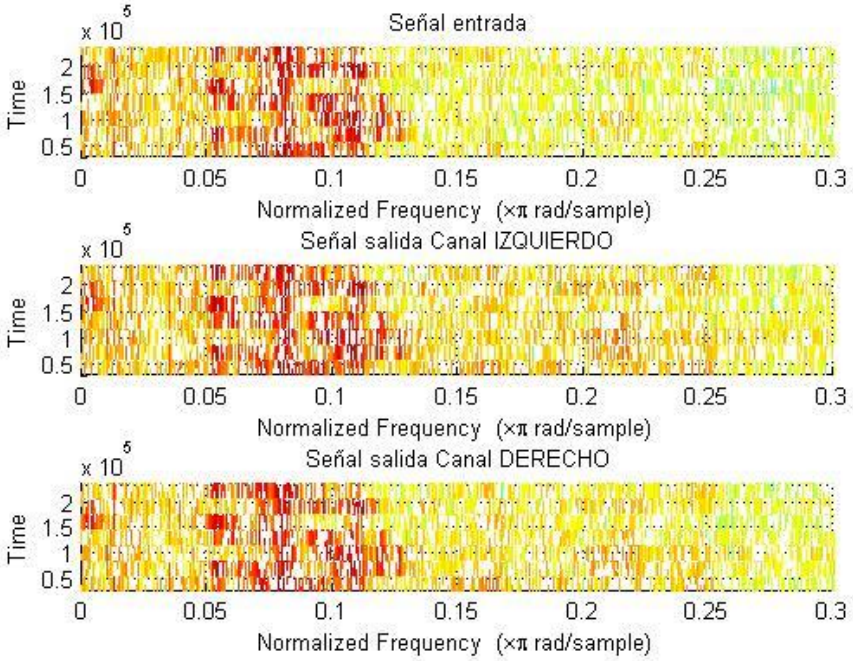


Figura I.II.4. Espectrograma.

II. FUNCIONES GetData() y SetData().UNITY

Aunque finalmente estas funciones no se han usado en el proyecto final, sí que han sido de gran relevancia en el proceso de desarrollo del plugin, y por lo tanto se merecen un pequeño inciso.

Ambas funciones han sido importantes a la hora de buscar una solución al problema de la disponibilidad de muestras futuras, lo cual es un requisito imprescindible para aplicar el método "Weighted Overlap-Add" y así evitar el ruido producido por las discontinuidades de los filtros HRTF.

Las funciones "getData()" y "setData()", a diferencia de la función "OnAudioFilterRead", no están relacionadas con el buffer de audio de salida, sino que están relacionadas con el propio clip de audio que tiene asociado un "GameObject" concreto. Por lo tanto, la principal diferencia con respecto a la función "OnAudioFilterRead", es que ahora sí que podríamos procesar cada "AudioSource" por separado ya que no estamos obteniendo el buffer de salida de audio donde ya están todas las "AudioSource" mezcladas, si no que estamos modificando directamente las muestras del clip de audio asociado al "GameObject" del "AudioSource".

Con la función GetData(float[], int) obtenemos las muestras del clip de audio, mientras que con la función SetData(float[], int) hacemos lo contrario, establecemos las muestras que queramos dentro del clip de audio, modificándolo.

El vector en el que queremos guardar o con el que queremos modificar el audio, se introduce como primer parámetro, y de su tamaño dependerá el número de muestras que se tomarán o modificarán en el clip de sonido. El segundo parámetro está destinado a indicar el offset, es decir, el número de la primera muestra a partir de la cual tomaremos las restantes del clip de sonido.

Por ejemplo, si nuestro vector tiene un tamaño de 1024, y establecemos la variable offset en 5, con la función "GetData()", estaremos guardando en nuestro vector de tamaño 1024, aquellas muestras del clip de sonido comprendidas entre la muestra 5 y la 1029. Con la función "SetData()" modificamos las muestras del clip de audio comprendidas entre la muestra 5 y la 1029 con los valores del vector que le introduzcamos como parámetro.

De esta forma, haciendo uso de la variable de offset, podemos acceder a las muestras futuras, procesarlas, y devolvérselas al clip de audio.

Las dificultades que presentan estas funciones, y que han motivado que se descarten su uso son por un lado la reproducción en bucle, ya que el hecho de estar modificando el propio clip de audio, hace que en la segunda reproducción del archivo, las muestras ya no se correspondan con el clip original, y sí a las modificadas en el primera reproducción, por lo que estaríamos filtrando una señal ya filtrada anteriormente con otros valores de acimut y elevación.

Otro problema que acarrean estas funciones es que éstas no cuentan con una interrupción como la de "OnAudioFilterRead" que hace que se active cada vez que el buffer está completo (1024 muestras), por lo que si queremos que sea un procesado de audio en tiempo real, debemos marcarnos nosotros nuestro propio 'timing' ya sea creando nuestras propias interrupciones, o

usando las funciones de delay de Unity de tal forma que se activen aproximadamente cada 20 milisegundos y así podamos realizar el procesado del audio cada 1024 muestras.

También hay que tener en cuenta que “OnAudioFilterRead” se ejecuta en un hilo distinto al principal, por lo que la carga de procesado que tenga el plugin el cual invocamos dentro de dicha función, no afectará en principio al hilo principal. Al usar las funciones “getData()” y “setData()” tendríamos que tener en cuenta esto, y crear los hilos nosotros mismos con el fin de que el plugin creado no afecte en exceso al transcurso del programa principal.

III. CÓMO USAR EL PLUGIN EN UNITY

Lo primero que tenemos que hacer, si queremos usar el plugin de procesado de audio en 3D dentro de Unity, es crear una carpeta llamada 'Plugins' dentro de la carpeta principal 'Assets', e incluir en su interior el plugin tal y como se puede ver en la Figura III.1. Se recomienda usar el mismo nombre 'Proyecto' para el plugin, ya que las siguientes directrices se explican siguiendo esta nomenclatura. En caso de cambiarlo, habrá que tener en cuenta que será necesario cambiar otros campos para que funcione correctamente.

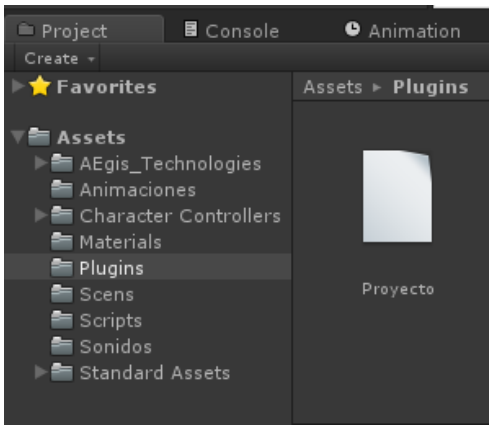


Figura III.1.- Incluir el plugin del proyecto dentro de la carpeta "Assets/Plugins".

Una vez añadido el plugin al proyecto, debemos crear un script basado en C# que se encargará de importar el plugin y de realizar las operaciones oportunas para ponerlo en marcha.

Éste script debe adjuntarse a aquel "GameObject" que contenga el "AudioListerner", acordándose siempre de que no puede haber más de un "AudioListener" en la escena si queremos evitar problemas. Normalmente el "AudioListener" está asociado a la "MainCamera" tal como se puede ver en la Figura III.2.

En dicha figura también podemos apreciar que el nombre del script en este caso es "AudioFilter2" aunque puede tener cualquier nombre, y que el script tiene dos parámetros públicos llamados "Mi Audio Source" y "Distancia Max". En el primero de ellos hay que introducir el "GameObject" que tenga asociado el "AudioSource", en este caso un helicóptero, mientras que el segundo parámetro se usa para determinar la distancia máxima a partir de la cual ya no se oirá el sonido del objeto.

Figura III.2.- Script que importa el plugin adjunto al "GameObject" que contiene el "AudioListener".

Añadiendo dichos archivos y siguiendo las pautas indicadas, el plugin creado se podrá usar en cualquier proyecto de Unity, siempre y cuando haya un único "AudiosSource" en la escena.

IV. CONTENIDO DEL SCRIPT “AudioFilter2”. UNITY

El script “Audio Filter 2” ha sido creado para poder llevar a cabo la importación del plugin en cualquier escena, encargándose de que todo se ejecute correctamente. Como se ha explicado en el anexo anterior y se puede apreciar en su Figura III.4, este script debe estar asociado a la cámara, para que así el procesado de audios se haga adecuadamente.

El contenido de dicho script se puede ver dividido en tres figuras distintas, concretamente en la Figura IV.1, IV.2 y IV.3.

La primera de ellas muestra la clase principal con todas las variables declaradas entre las que destacan dos, ambas de carácter público. Una de ellas es de tipo “MiObjeto”, y que hemos creado nosotros mismos para poder pasar como variable el objeto respecto al que estamos procesando el audio. Es decir, ésta variable será con la que obtenemos los valores de acimut y elevación del objeto que contiene el “AudioSource” de la escena.

La otra variable importante del Script “Audio Filter 2”, es la declarada en la línea 9 de la Figura IV.1. En ella estamos creando una variable de nombre “miDll” de la clase “Audio3D”. Esta clase es la clase principal creada dentro del plugin, por lo que declarando esta variable es como si estuviéramos importando el plugin.

El resto de variables creadas están destinadas a almacenar las distintas muestras de audio, con el fin de realizar el delay necesario para ejecutar correctamente el “Weighted Overlap-Add”.

```
1  using UnityEngine;
2  using System.Collections;
3  using System.Runtime.InteropServices;
4
5
6  public class AudioFilter2 : MonoBehaviour {
7
8      public MiObjeto miAudioSource;
9      public Proyecto.Audio3D miDll = new Proyecto.Audio3D();
10
11     double acimut = 0;
12     double elevacion = 0;
13
14     float[] miAudio = new float[2048];
15     float[] audioOld = new float[2048];
16     float[] audioNext = new float[1024];
17
18     float[] cola = new float[1024];
19     float[] auxCola = new float[1024];
20
```

Figura IV.1.- Captura del Script ‘Audio Filter 2’.

Una vez creada la variable “miDll”, ya podremos invocar su método interno “aplicarAudio3D()” dentro de la función “OnAudioFilterRead”, tal y como se puede ver en la Figura IV.2. Los dos bucles ‘for’ sirven para copiar los distintos vectores según convenga y realizar con ello el delay de un frame de audio.

Por otro lado, en la línea 48 de la Figura IV.2, se puede apreciar el cálculo realizado para la atenuación del audio en función de la distancia máxima, la cual es un parámetro público que se puede cambiar desde la propia escena de Unity. La atenuación sigue una distribución lineal.

```
34
35      void OnAudioFilterRead(float[] data, int channels) {
36      float _atenuacion = 0;
37
38          for (int i = 0; i < 2048; i++)
39          {
40              miAudio[i] = audioOld[i];
41              audioOld[i] = data[i];
42              if(i < 1024) {
43                  audioNext[i] = data[i];
44              }
45          }
46
47          if (distancia > 5) {
48               _atenuacion = (float)(1 - (distancia - 5)/distanciaMax);
49          } else {
50              _atenuacion = 1f;
51          }
52          if (_atenuacion < 0) {
53              _atenuacion = 0f;
54          }
55
56          //Debug.Log("Acimut " + acimut + " Elevacion " + elevacion);
57          //Debug.Log(_atenuacion +" " +  distancia );
58
59          miDll.aplicarAudio3D (ref miAudio, acimut, elevacion, audioNext, out cola);
60
61
62          for (int i = 0; i < 1024; i++)
63          {
64              data[i] = _atenuacion * (miAudio[i] + auxCola[i]);
65              auxCola[i] = cola[i];
66
67              data[1024 + i] = _atenuacion * miAudio[1024 + i];
68          }
69
70      } /// Fin OnAudioFilterRead
```

Figura IV.2.- Captura del Script ‘Audio Filter 2’.

Finalmente en la Figura IV.3, se puede ver la función “coordenadas()” que ha sido creada para realizar el cálculo de las variables de acimut y elevación del objeto que contiene el “AudioSource” respecto a la cámara, además de calcular su distancia. La función “coordenadas()” se invoca continuamente desde la función “Update()”, por lo que sus valores van a estar siempre actualizados.

En la línea 81 de la Figura IV.3 se calcula el vector que apunta desde la “MainCamera” hasta el “GameObject” que contiene el “AudioSource” de la escena. Con ese vector, ya podemos calcular en las líneas 82 y 83, los valores de phi y theta mediante cálculos trigonométricos, así como el valor de la distancia, que no es más que la magnitud del propio vector.

De la línea 87 a la 98 de la Figura IV.3, hay varios ‘if´s’ destinados a realizar los ajustes oportunos para determinar los valores de acimut y elevación. Estos reajustes se deben por ejemplo a que el valor de phi abarca un rango entre-180º y 180º, mientras que el acimut que el plugin maneja va desde-90º hasta 90º; ya que los valores mayores de 90º y menores de-90º, que serían los que se encuentran a la espalda del oyente, se procesan asignando una elevación mayor a los 180º.

```
73
74      void coordenadas ( out double _acimut, out double _elevacion){
75          Vector3 cameraRelative;
76          float phi;
77          float theta;
78
79
80
81          cameraRelative = Camera.main.transform.InverseTransformPoint(miAudioSource.transform.position);
82          phi = Mathf.Atan2(cameraRelative.x, cameraRelative.z) * Mathf.Rad2Deg;
83          theta = 90- Mathf.Acos(cameraRelative.y / cameraRelative.magnitude) * Mathf.Rad2Deg;
84
85          distancia = (double)cameraRelative.magnitude;
86
87          _acimut = phi;
88          if (phi < -90) {
89              _acimut = -180 - phi;
90          }
91          if (phi > 90) {
92              _acimut = 180 - phi;
93          }
94          if (phi > 90 || phi < -90) {
95              _elevacion = 180 - theta;
96          } else {
97              _elevacion = theta;
98          }
99
100     } //Fin coordenadas()
```

Figura IV.3.- Captura del Script ‘Audio Filter 2’.