Universidad Zaragoza
1542

Trabajo Fin de Grado

Auralización de espacios acústicos mediante trazado de rayos

Autor

Julia López Cambra

Director

José Ramón Beltrán Blázquez

Departamento de Ingeniería Electrónica y Comunicaciones

Escuela de Ingeniería y Arquitectura

2016

AURALIZACIÓN DE ESPACIOS ACÚSTICOS MEDIANTE TRAZADO DE RAYOS

RESUMEN

El objetivo de este proyecto es crear una librería de código C++ que calcule las propiedades acústicas de una escena mediante el método de trazado de rayos. Para ello habrá que definir el modelo 3D de los escenarios, que servirá de entrada la librería, el trazado de caminos acústicos a través del mismo, el cálculo de una respuesta impulsional, y por último el filtrado de la señal de sonido a través de dicha respuesta al impulso.

La parte correspondiente al trazado de rayos se basa en un proyecto preexistente, llamado GSound, creado en la Universidad de Carolina del Norte, EEUU. El tratamiento de señal se crea basado en los algoritmos y técnicas que se emplean habitualmente para este propósito: reverberadores de convolución y reverberadores artificiales mediante filtros IIR.

AURALIZATION OF ACOUSTIC SCENES USING RAY TRACING

ABSTRACT

In this project, a C++ code library that calculates a given scene's acoustic properties using a ray tracing technique is created. First, the description of 3D scenes within the code is formalized, then tracing of acoustic paths is addressed, an impulse response extracted from these paths, and lastly audio filtering through the impulse response implemented.

The ray tracing part of the project is based on a previous project, called GSound, developed in the University of North Carolina, USA. The signal processing part is designed following the algorithms and techniques commonly used for this purpose: convolutional reverberation and IIR algorithms.

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

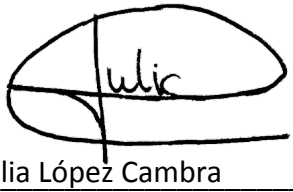
D./Dª. Julia López Cambra,

con nº de DNI 73019028P en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Auralización de espacios acústicos mediante trazado de rayos

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 20 de noviembre de 2016



Fdo: Julia López Cambra

Índice

1. Introducción

El objetivo de este trabajo es elaborar un código capaz de recrear de forma realista la propagación del sonido desde una fuente hasta un oyente a través de un escenario predefinido, tal que proporcione audio binaural en la posición del oyente. El sistema debería funcionar en tiempo real, y adaptarse a distintos escenarios.

Se pretende presentar este código en forma de librería para poder ser importado y utilizado por software externo (motores de videojuegos, por ejemplo).

Para la propagación se parte de un proyecto ya existente, GSound [1], que implementa en lenguaje C++ un algoritmo de trazado de rayos, e incluye además un código básico para el manejo de los escenarios. GSound está construido como un programa completo y autosuficiente: dentro del código se definen los datos de entrada (audio y escenarios), que son fijos, y el manejo de periféricos para llevar los datos de salida a los altavoces.

La mayor parte del trabajo se desarrollará utilizando Visual Studio, donde se realizarán varias modificaciones sobre GSound: por una parte, será necesario mejorar el soporte que se le da a los escenarios, de modo que sea compatible con al menos un formato estándar. Por otra, habrá que cambiar completamente la estructura de entrada y salida de datos, eliminando la parte del manejo de archivos de sonido, de forma que podamos introducir un escenario y obtener la respuesta impulsional que lo caracteriza.

Lo anterior se incluirá dentro de una librería dedicada a recrear una cadena de procesado de audio, que deberá ser capaz de leer de un archivo de audio, filtrar utilizando la respuesta al impulso anterior, y proporcionar datos de salida al ritmo al que se necesiten.

Además de compilar la librería, hará falta crear un entorno en el que mostrar cómo utilizarla. Para esto se utilizará OpenFrameworks sobre el mismo Visual Studio, una herramienta pensada para manipular contenido audiovisual y programar interacciones con facilidad. En él se crearán algunos escenarios, a los que se añadirá el procesado de audio.

2. Principios teóricos

En el ámbito de los videojuegos y los simuladores se hacen grandes avances en cuanto al realismo gráfico y en la IA, mientras que en general se dedica mucho menos esfuerzo a la mejora de la calidad del audio de los mismos.

Los principios teóricos que se introducen a continuación son implementados en GSound y en este proyecto de modo que obtenemos un motor de simulación de audio capaz de, en tiempo real, renderizar el sonido emitido en las fuentes tal y como se escucharía en el punto en el que se encuentra el receptor, teniendo en cuenta el escenario actual.

2.1. Auralización

El término auralización fue introducido por M. Kleiner en 1990 [2], quien lo define como: *proceso de presentación audible, por modelización física o matemática del campo sonoro de una fuente en un espacio, de tal forma que simulan la experiencia sonora en una determinada posición en el espacio modelado.*

Originalmente, se diseñaron métodos de auralización para analizar la calidad acústica de recintos en fase de diseño. Sin embargo, se ha extendido su aplicación a sistemas de realidad virtual y videojuegos, permitiendo aumentar la sensación de inmersión [3]. En estos casos, la exactitud del cálculo no es tan importante como la rapidez, optando normalmente por utilizar aproximaciones que permitan simular en tiempo real.

El audio producido podrá ser monoaural (un canal de salida) o binaural (dos canales de salida). La característica distintiva de una representación binaural respecto de otras es que ésta tiene en cuenta que el oyente no es un único punto en el espacio, sino que representa la posición de los dos oídos, con lo que se consigue mayor realismo y sensación de audio tridimensional.

El cálculo de audio binaural requiere hallar dos respuestas impulsionales, una para cada componente del receptor [4]:

- $h_R(t) \rightarrow$ respuesta al impulso en el oído derecho del receptor
- $h_L(t) \rightarrow$ respuesta al impulso en el oído izquierdo del receptor

2.2. Propagación de sonido

La propagación de audio se suele modelar como una combinación de varios fenómenos. Cualquier sonido que llega a un receptor se puede dividir en tres componentes [1]:

- Sonido directo: se transmite directamente de la fuente al receptor.

- Reflexiones tempranas: abarcan los primeros ecos de un sonido, que llegan al oyente justo después del sonido directo. Estas contribuciones pueden venir de reflexiones especulares en superficies cercanas, reflexiones difusas o difracción por los bordes de los objetos contenidos en la escena. Las reflexiones tempranas dan al oyente una idea del tamaño del entorno y de las características más destacadas.
- Reverberación tardía: es el último componente que escucha el receptor, en el que se acumulan muchas reflexiones de mayor orden cuya amplitud va decayendo.

2.3. Trazado de rayos

El trazado de rayos [5] es un algoritmo empleado en cualquier ámbito en el que se quiera simular los efectos de la propagación de onda (además de en audio, en imagen, principalmente). Consiste en trazar rayos partiendo del punto donde se encuentra el emisor; y seguir su trayectoria a través del escenario. Los rayos, según el camino que hayan seguido, representarán caminos directos, o los efectos de la reflexión y refracción. Estos resultados se utilizarán para calcular las respuestas impulsionales izquierda y derecha en cada instante de la simulación. En el Anexo 1 se explica el funcionamiento de esta técnica, así como los detalles de la implementación de GSound.

2.4. Reverberación

La reverberación tardía se caracteriza por una alta densidad de ecos, causados por un gran número de reflexiones de alto orden, que llegan de todas direcciones [6].

Al contrario que las reflexiones tempranas, en las que calcularemos cada eco por separado (mediante trazado de rayos), para la reverberación se utiliza una aproximación perceptual, es decir, se intentan reproducir las principales características perceptuales de la reverberación tardía. Los algoritmos que resultan de esto se basan en filtros de tipo IIR (respuesta impulsional infinita).

3. Procesado de la señal

En este apartado se describe, de forma general, la estructura del proyecto. Se presentan todas las etapas del tratamiento de la señal de audio, desde el archivo de entrada hasta llegar a los periféricos de salida. Los detalles de cada etapa y de su implementación en C++ se verán en los siguientes apartados.

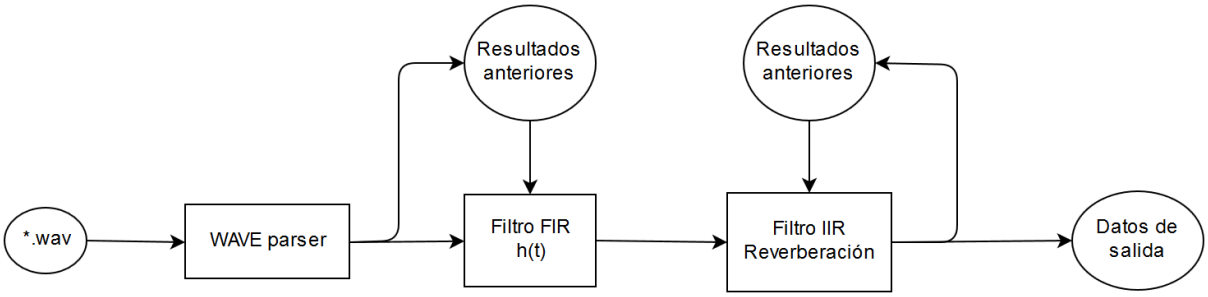


Figura 1: Diagrama de flujo que describe las etapas de procesado de la señal.

3.1. Entrada de datos

Por simplicidad, admitimos solamente un formato de entrada de audio, el *WAVE* (extensión *.wav). Para poder operar con éste, lo primero será disponer de una herramienta capaz de leer la información que contiene.

Un archivo *WAV* [7] comienza con una cabecera que contiene parámetros como el número de canales, tasa de muestreo, tamaño de la sección de datos, bits por muestra, etc. A continuación se encuentran los datos, intercalando las muestras de cada canal de una en una; en el caso del estéreo: L R L R ...

Así pues, la clase que, dentro del código, se encargue de la entrada consistirá básicamente en un *parser*: deberá leer todos los datos de la cabecera, pasar al resto del programa los parámetros que se necesiten, y finalmente leer las muestras de audio según se vayan solicitando.

3.2. Trazado de rayos

La primera modificación de la señal consistirá en filtrar por una respuesta impulsional finita (FIR), calculada mediante trazado de rayos con GSound.

Para el cálculo del filtrado disponemos de dos opciones: convolución de la señal de entrada con la $h(t)$ o pasar al dominio frecuencial mediante FFT. El uso de una u otra

dependerá de las características del filtro (principalmente del número de muestras que tenga).

Por otra parte, ya que el procesado se realiza en tiempo real, se calculará el resultado de N en N muestras. En consecuencia, como indica la figura 1 tendremos que mantener un búfer con muestras de entrada pasadas para convolucionar con los retardos de $h(t)$.

3.3. Reverberación

El último paso consiste en añadir un efecto de reverberación tardía artificial, ya que el trazado de rayos sólo simula las reflexiones tempranas. La literatura proporciona varias posibles implementaciones, creadas con filtros de tipo *comb* y *allpass*, ambos con funciones de transferencia de tipo IIR. Se propone utilizar la arquitectura propuesta por Schroeder [8], de la cual derivan las demás, por ser más sencilla (una implementación compleja podría ser problemática para la condición de tiempo real).

4. Respuesta al impulso y filtrado

4.1. Cálculo de la respuesta impulsional del escenario

Podemos caracterizar acústicamente cualquier escenario mediante su respuesta al impulso, fijadas una posición de emisor y receptor concretas. En el Anexo 1 se explica cómo GSound simula la propagación de audio en una escena. Aquí tratamos de convertir ese trazado de rayos en una función $h(t)$ que represente las propiedades acústicas del entorno.

Cada camino de propagación trazado representa una distancia recorrida por el sonido y un valor de atenuación (debido a esa distancia, o a la absorción de los sólidos del entorno si se dan reflexiones). Así, podemos convertir cada uno en un impulso que añadiremos a la respuesta del filtro.

Un impulso se caracteriza por:

- Un valor de retardo: calculado a partir de la distancia total recorrida en el camino de propagación, y la velocidad del sonido.
- Una amplitud: calculada a partir de la intensidad de la fuente (parámetro que elegimos al inicializarla), la atenuación introducida por el aire, y la atenuación introducida por las propiedades de los materiales.

4.2. Filtrado de la señal de entrada

Una vez obtenida la respuesta al impulso, el siguiente paso será convolucionar con la señal de entrada. Las muestras de audio llegarán al filtro a través de un búfer de longitud fija, por lo que la convolución se calcula para N muestras, donde N es la longitud de dicho búfer.

$$y(n) = x(n) * h(t) \qquad n = 0..N \tag{4.1}$$

Cada vez que el programa solicite audio para la salida, se realizará este cálculo, con nuevos datos de entrada y tal vez con una nueva $h(t)$, recalculada para tener en cuenta los posibles cambios en la posición de los integrantes de la escena.

En la ecuación 4.1 aparecen $x(n)$ e $y(n)$ como dependientes de n, mientras que $h(t)$ está en función de t. Esto se debe a que, debido a la manera de generar la respuesta impulsional, el eje temporal de ésta no está restringido a los instantes de muestreo por los que se rigen las señales de entrada y salida.

A la hora de convolucionar, nos encontramos con que el instante en el que ocurre un impulso de $h(t)$ no coincide con el instante al que corresponde ninguna muestra de $x(t)$, sino que se encuentra entre dos muestras. Esto se describe gráficamente en la figura 2.

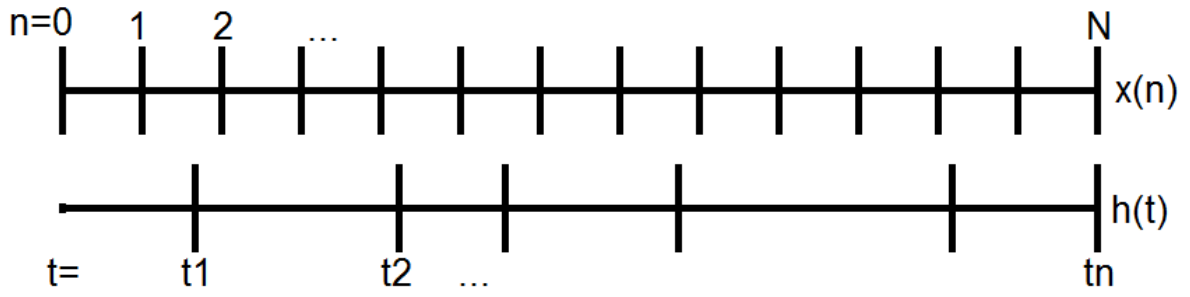


Figura 2: Ilustración de una posible superposición de los ejes temporales de la señal de entrada y la respuesta al impulso del filtro.

La solución a este problema consiste en, para cada elemento de $h(t)$, interpolar las dos muestras de $x(t)$ que se encuentren más cercanas dentro del eje temporal. Utilizando como ejemplo la figura 2, a la muestra $h(t_1)$ le correspondería una $x(t_1)$, que sería el resultado de interpolar $x(n = 1)$ con $x(n = 2)$. Como la separación temporal entre muestras es pequeña, una interpolación lineal será suficiente.

A partir de todo lo anterior podemos expresar el cálculo de una muestra de salida en función de la señal de entrada y la función del filtro de la siguiente manera:

$$y(n) = x(n - t_1)h(t_1) + x(n - t_2)h(t_2) + ... + x(n - t_N)h(t_N) \tag{4.2}$$

Donde cada $x(n - t_i)$ se calcula por interpolación lineal a partir de las muestras de $x(n)$ más cercanas mediante la expresión:

$$\begin{aligned} x(n - t_i) &= x(k + 1)d + x(k)(1 - d) \\ d &= \frac{t_i - kt_s}{t_s} \end{aligned} \tag{4.3}$$

El parámetro t_s es el tiempo por muestra, es decir, la inversa de la frecuencia de muestreo. El parámetro d se utiliza para ponderar la media entre los dos valores de $x(n)$, como muestra la figura 3.

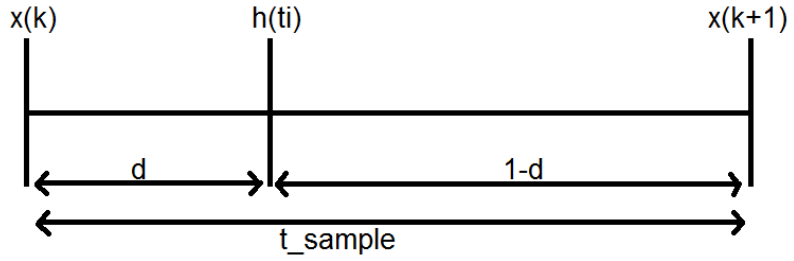


Figura 3: Explicación gráfica de las variables de la ecuación 4.3.

5. Escenarios

El programa necesita conocer la geometría de la escena donde se realiza la propagación de sonido. Para describirla, habitualmente se descompone en una malla formada por polígonos.

5.1. Formato nativo (*.sm*)

La implementación de GSound utiliza un formato propio para la representación de los escenarios 3D. Existe una clase específica para almacenar la información del mallado, llamada `SoundMesh`. De esta forma, por cada escena que queramos incluir, simplemente hay que crear una nueva instancia de esta clase.

Dentro de `SoundMesh`, el escenario se describe mediante tres listas:

- Vértices: cada uno consiste simplemente en tres números reales, representando las tres coordenadas cartesianas. Los vértices se identifican mediante su posición en la lista.
- Materiales: hace falta una manera de especificar las propiedades acústicas de cada pieza del mallado. Existe una clase `SoundMaterial` preparada para guardar todos los datos necesarios sobre un material concreto (se detalla más adelante).
- Triángulos: el mallado de la geometría está restringido a triángulos. Cada uno de ellos queda descrito mediante tres índices, que apuntan a tres vértices de la lista anterior, y un cuarto índice, que indica cual de los materiales anteriores corresponde a esta pieza del escenario.

Además, contamos con la funcionalidad añadida de poder guardar una instancia de `SoundMesh` en un archivo externo de tipo binario, con la extensión **.sm*. Cargar un escenario de esta forma es mucho más rápido que si tuviéramos que especificar uno a uno cada vértice, material y triángulo; por ejemplo leyéndolos de otro fichero externo. Esto es útil si una escena aparece más de una vez, se podría guardar y así sólo se procesa la primera vez que aparece.

5.2. Formato Wavefront OBJ

Aunque dispongamos del formato **.sm*, todavía hace falta una manera de especificarle una geometría a una instancia vacía de la clase `SoundMesh`, sin partir de otra `SoundMesh` anterior. Para conseguir esto debemos implementar una función que lea algún archivo de mallado estándar y vaya construyendo un escenario. Para ello, elegimos el formato Wavefront OBJ.

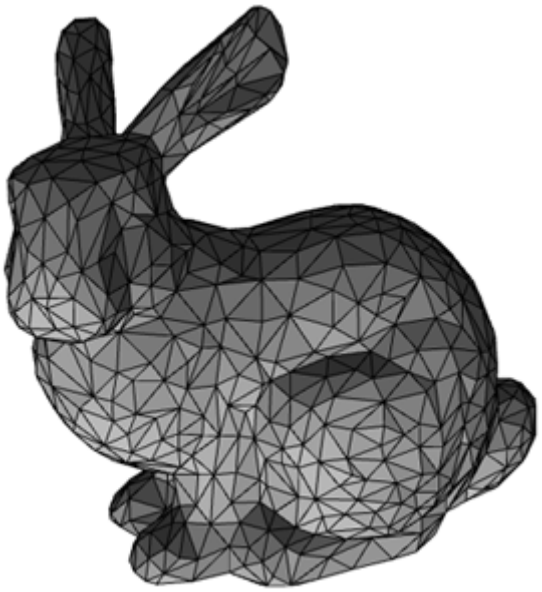


Figura 4: Ejemplo de mallado 3D formado por triángulos.

5.2.1. Elección de OBJ

Existen muchos formatos para describir una geometría 3D [9], algunos de ellos muy extendidos; sin embargo nos basta con implementar un *parser* para leer un único formato. Elegimos Wavefront OBJ por varios motivos:

- Por un lado, es uno de los formatos más utilizados. Todos los programas que se utilizan habitualmente para la edición de escenas 3D (Blender, Unity, ...) permiten trabajar con él.
- Por otro lado, es un fichero ASCII, lo cual nos permite manipularlo con facilidad.
- Es un estándar abierto. Esto es imprescindible, necesitamos conocer la forma exacta en que se escribe la información.
- Otra ventaja es que es un formato muy sencillo. El `SoundMesh` proporciona una descripción muy básica de la geometría (sólo vértices y triángulos). Mientras que otros estándares incluyen muchas opciones (como por ejemplo añadir física, animaciones...), OBJ incluye poco más que una enumeración de vértices y polígonos.

5.2.2. Especificación del formato

A continuación se muestra un ejemplo de los elementos más comunes que contiene un OBJ [10].

```
# Materiales:
mtllib [external .mtl file name]
usemtl [material name]
# Vértices:
v 0.123 0.234 0.345 1.0
```

```
v ...
# Coordenadas de textura:
vt 0.500 1 [0]
vt ...
# Normales:
vn 0.707 0.000 0.707
vn ...
# Geometría curva (free form geometry)
vp 0.310000 3.210000 2.100000
vp ...
# Caras poligonales:
f 1 2 3
f 3/1 4/2 5/3
f 6/4/1 3/5/3 7/6/5
f ...
```

Un fichero de este tipo contiene un dato en cada línea. El tipo de dato se identifica por el carácter o grupo de caracteres hasta el primer espacio, especificando después su valor.

Para los elementos de tipo vértice (todos los que empiezan por `v*`) se especifican tres números reales, correspondientes a las tres coordenadas cartesianas. Las caras se definen mediante una lista de números naturales, tres como mínimo, que corresponden con los índices de los vértices que forman el polígono descrito. Cada índice puede llevar otros asociados correspondientes a texturas y normales. Por esto, los vértices siempre se definen primero y las caras al final, como se ve en la figura **??**.

5.2.3. Lectura de OBJ

Como se ha visto antes, `SoundMesh` hace una representación muy sencilla del escenario, utilizando sólo los elementos imprescindibles. Además, los formatos estándar como OBJ están pensados para usarlos en entornos gráficos, por lo que tienen información sobre texturas (imágenes que se usan para colorear los polígonos) que no nos es de utilidad.

Para, partiendo de cualquier fichero **.obj*, conseguir pasar toda la información relevante a un `SoundMesh`, se crea un *parser*.

A continuación se detallan los datos que extraemos:

1. Líneas que comienzan con `v`: cada una de estas se convierte directamente en un vértice de nuestro mallado. Sólo leemos las líneas de tipo `v`, mientras que las `vt`, `vp`, `vn` se ignoran.

2. Líneas que comienzan con `f`: en este caso debemos solucionar la discrepancia entre el formato OBJ, que permite describir polígonos con cualquier número de lados, y SoundMesh, que se restringe a triángulos. El uso de polígonos de más de tres lados es bastante común, y no sería razonable esperar que el usuario siempre proporcione mallados triangulares.

   La solución consiste en, cada vez que se lee un polígono, descomponerlo usando una librería de triangulación.

3. Líneas que comienzan con `usemtl`: este comando hace referencia a los materiales utilizados. Estas líneas aparecen entremezcladas con las de tipo `f`: cada vez que aparece `usemtl`, se especifica un material que se asignará a todas las caras poligonales que se describen a continuación, hasta que aparezca otro `usemtl`.

   La descripción de los materiales en GSound contiene solamente parámetros que describen la interacción de éstos con las ondas acústicas, mientras que en OBJ se centra en su aspecto visual. Es decir, OBJ no proporciona ninguna información sobre éstos que nos sea de utilidad.

   Lo único que podemos extraer para nuestro escenario, es una lista de los materiales utilizados, con el nombre que se les da, y en qué triángulos se utiliza cada uno. Sin embargo, tendríamos que obtener los parámetros que describen sus propiedades acústicas por otros medios.

5.3. Triangulación

Como se ha mencionado antes, se necesitará un método para triangular los polígonos que puedan aparecer en la lectura de un escenario. En este caso, hemos utilizado una API escrita en C++ [11], para poder incluirla fácilmente en el proyecto.

5.3.1. *Ear clipping*

Hay varias maneras de implementar una triangulación, y no todas tienen las mismas capacidades. El código elegido sólo puede procesar polígonos planos y sin agujeros, tanto cóncavos como convexos, utilizando un método llamado *ear clipping* [12]. Consideramos que esto será suficiente, ya que no es habitual encontrar polígonos complejos en modelos 3D (se desaconsejan a los diseñadores).

Este método se basa en el hecho de que, dado cualquier polígono simple, sin agujeros, con al menos cuatro vértices, éste tiene al menos dos 'orejas', es decir, triángulos con dos lados correspondientes a los bordes del polígono y el tercer lado en el interior. El algoritmo encuentra una de estas orejas, la recorta del polígono, y repite hasta que sólo quedan tres vértices (como se muestra en el ejemplo de la figura 5).

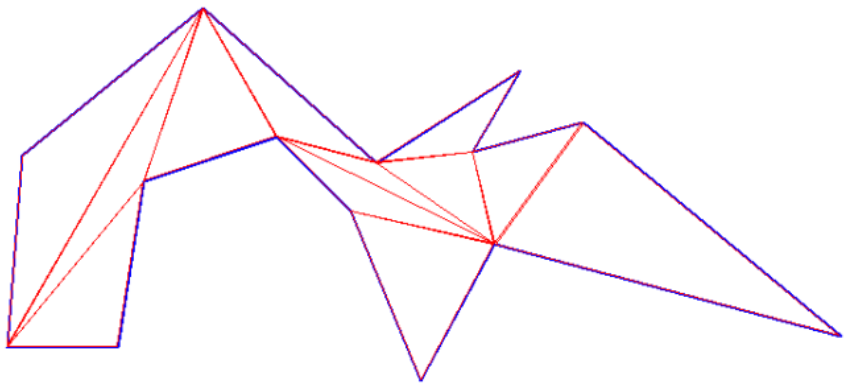


Figura 5: Ejemplo de descomposición de un polígono cóncavo en triángulos mediante *ear clipping*.

5.4. Materiales

5.4.1. Propiedades acústicas de los materiales

Especificaremos el comportamiento de un material mediante tres conjuntos de coeficientes de atenuación:

- Coeficientes de atenuación para el sonido que es reflejado en una superficie del material en cuestión: cuando una onda sonora rebota en una superficie, el sonido resultante estará atenuado según esta cantidad.
- Coeficientes de transmisión para el sonido que pasa a través de una barrera del material. Cuando una onda sonora choca con una superficie, la parte de éste que no es reflejada es o bien absorbida (se disipa como calor) o se transmite al otro lado de la barrera. Multiplicar por estos coeficientes determina el sonido que atraviesa el material.
- Coeficientes de atenuación para el sonido que ha atravesado un material. Estos valores están especificados en unidades de atenuación por unidad de longitud. Es decir, si un rayo atraviesa D unidades de este material, el sonido final habrá sido atenuado a la potencia D.

Cada clase de coeficiente se da como un conjunto de valores y no como uno sólo para poder incluir su comportamiento frente a distintas frecuencias. Cada grupo de coeficientes representa una respuesta frecuencial, muestreada para ocho puntos diferentes: 67 Hz, 125 Hz, 250 Hz, 500 Hz, 1 kHz, 2 kHz, 4 kHz y 8 kHz.

Muchas fuentes nos describen las propiedades de los materiales de esta manera, en función de la frecuencia. En todo caso, si sólo se conociera un valor medio, podemos simplemente aproximar por una función constante. En la figura 6 se incluyen las propiedades de absorción de algunos materiales comunes [13]. Materiales como el ladrillo, el mármol y el yeso tienen una absorción muy baja, lo que significa que reflejarán más. Otros como el cemento y, más aún, los textiles, tienen una absorción muy alta, lo que significa que buena parte de la onda incidente no se reflejará.

5.4.2. Paso de parámetros al escenario

Necesitaremos también una vía por la cual dar valores concretos a estos coeficientes en cada material empleado en el mallado. Para ello nos basamos en las líneas del OBJ de tipo `usemtl`, mencionadas en la página 17.

Crearemos un fichero de texto (*.txt) en el que, con un formato muy sencillo, se especificarán todos los datos necesarios para representar cada material. Este archivo complementará al OBJ a la hora de introducir nuevos escenarios. Aun así, para cubrir todos los posibles casos, se programará el *parser* de modo que sea posible no proporcionar este fichero, o que no aparezcan en él todos los materiales, y en estos casos el programa utilizará un material por defecto predefinido.

Este archivo consistirá en grupos de cuatro líneas, donde la primera de ellas contiene el nombre del material en cuestión, y las otras tres los parámetros. En la primera línea establecemos `mat` como palabra clave para reconocer el inicio de un nuevo material,

seguido del nombre del mismo. Para relacionarlo con el OBJ, se deben utilizar los mismos nombres aquí y en las líneas tipo `usemtl`. Las demás líneas contienen cada una una respuesta frecuencial, con los ocho valores mencionados antes. A continuación se muestra un fichero de ejemplo con dos materiales:

```
mat material1
0.9 0.9 0.9 0.9 0.9 0.4 0.1 0 //coef. transmisión
0 0.45 0.9 0.9 0.9 0.6 0.3 0 //coef. aten. reflexión
0.5 0.43 0.36 0.29 0.21 0.14 0.07 0 //coef. aten. transmisión

mat material2
0.6 0.6 0.6 0.6 0.3 0.1 0 0 //coef. transmisión
0 0.4 0.8 0.8 0.8 0.3 0.2 0 //coef. aten. reflexión
0.7 0.56 0.45 0.34 0.3 0.15 0 0 //coef. aten. transmisión
```

Table 5-1 Absorption Coefficients of General Building Materials and Furnishings

Materials	Absorption Coefficients (Hz) 125	250	500	1000	2000	4000
Brick, unglazed	0.03	0.03	0.03	0.04	0.05	0.07
Brick, unglazed, painted	0.01	0.01	0.02	0.02	0.02	0.03
Carpet, heavy, on concrete	0.02	0.06	0.14	0.37	0.60	0.65
Same, on 40-oz hairfelt or foam rubber	0.08	0.24	0.57	0.69	0.71	0.73
Same, with impermeable latex backing on 40-oz hairfelt or foam rubber	0.08	0.27	0.39	0.34	0.48	0.63
Concrete block, coarse	0.36	0.44	0.31	0.29	0.39	0.25
Concrete block, painted	0.10	0.05	0.06	0.07	0.09	0.08
Fabrics:						
Light velour, 10 oz/yd^2, hung straight, in contact with wall	0.03	0.04	0.11	0.17	0.24	0.35
Medium velour, 14 oz/yd^2, draped to half area	0.07	0.31	0.49	0.75	0.70	0.60
Heavy velour, 18 oz/yd^2, draped to half area	0.14	0.35	0.55	0.72	0.70	0.65
Floors: Concrete or terrazzo	0.01	0.01	0.015	0.02	0.02	0.02
Linoleum, asphalt, rubber or cork tile on concrete	0.02	0.03	0.03	0.03	0.03	0.02
Wood	0.15	0.11	0.10	0.07	0.06	0.07
Wood parquet in asphalt on concrete	0.04	0.04	0.07	0.06	0.06	0.07
Glass:						
Large panes of heavy plate glass	0.18	0.06	0.04	0.03	0.02	0.02
Ordinary window glass	0.35	0.25	0.18	0.12	0.07	0.04
Gypsum Board, $\frac{1}{2}$" nailed to 2 × 4s, 16" oc	0.29	0.10	0.05	0.04	0.07	0.09
Marble or glazed tile	0.01	0.01	0.01	0.01	0.02	0.02
Openings:						
Stage, depending upon furnishings			0.25–0.75			
Deep balcony, upholstered seats			0.50–1.00			
Grills, ventilating			0.15–0.50			
Plaster, gypsum or lime, smooth finish on tile or brick	0.013	0.015	0.02	0.03	0.04	0.05
Plaster, gypsum or lime, rough finish on lath	0.14	0.10	0.06	0.05	0.04	0.03
Same, with smooth finish	0.14	0.10	0.06	0.04	0.04	0.03
Plywood paneling, $\frac{3}{8}$" thick	0.28	0.22	0.17	0.09	0.10	0.11
Water surface, as in a swimming pool	0.008	0.008	0.013	0.015	0.020	0.025
Air, sabins/100 ft^3 @ 50% RH				0.9	2.3	7.2
Absorption of Seats and Audience, sabins per square foot of seating area						
Audience, seated in upholstered seats, per square foot of floor area	0.60	0.74	0.88	0.96	0.93	0.85
Unoccupied cloth-covered upholstered seats, per square foot of floor area	0.49	0.66	0.80	0.88	0.82	0.70
Unoccupied leather-covered upholstered seats, per square foot of floor area	0.44	0.54	0.60	0.62	0.58	0.50
Wooden pews, occupied, per square foot of floor area	0.57	0.61	0.75	0.86	0.91	0.86
Chairs, metal or wood seats, each, unoccupied	0.15	0.19	0.22	0.39	0.38	0.30

Figura 6: Absorción en función de la frecuencia de algunos materiales comunes [13].

6. Reverberación

En este apartado se presenta el algoritmo utilizado para simular el efecto de reverberación tardía. Como se ha mencionado anteriormente, hemos dividido la simulación en dos grandes bloques: propagación temprana y tardía, ya que poseen diferentes características físicas y perceptuales [14].

6.1. Reverberador de Schroeder

Los primeros reverberadores artificiales basados en el procesado de señales discretas fueron construidos por Schroeder [8], y la mayor parte de los diseños que le siguieron consistieron en mejoras y modificaciones de aquéllos [14].

La estructura del filtro es sencilla, tal como se describe en la figura 7.

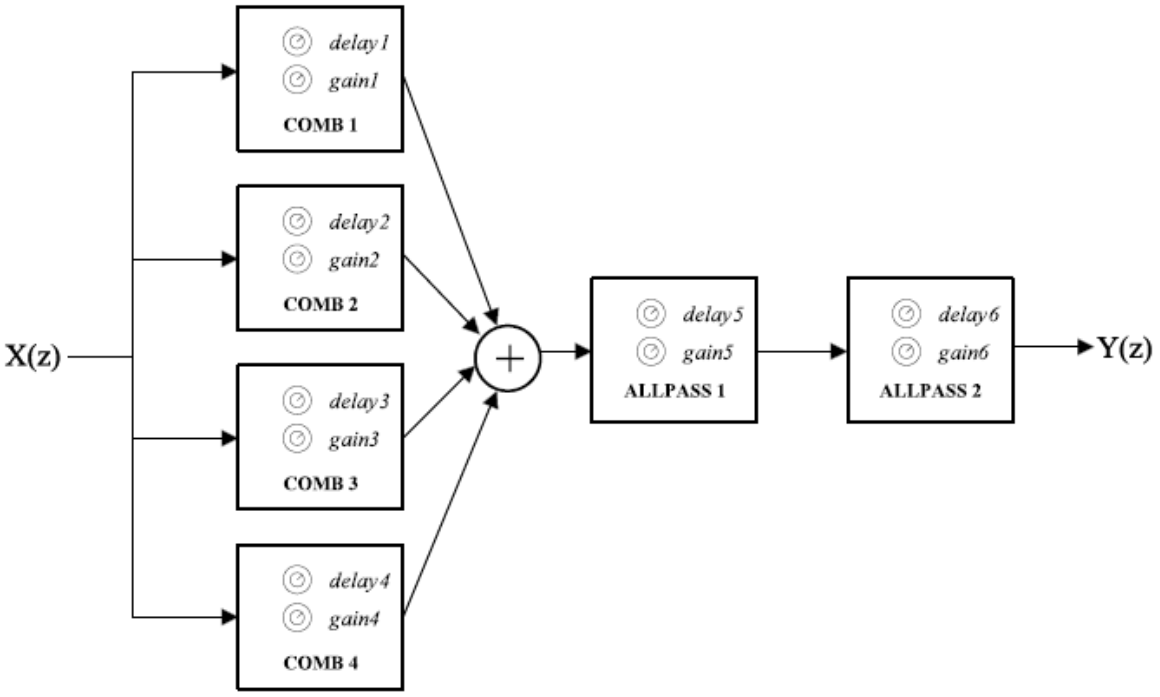


Figura 7: Reverberador de Schroeder con cuatro filtros *comb* en paralelo y dos filtros *allpass* en serie.

6.1.1. Filtros *comb*

El filtro *comb*, que aparece en la figura 8, consiste en un retardo cuya salida es realimentada a la entrada. Su función de transferencia tiene la expresión [6]:

$$H(z) = \frac{z^{-k}}{1 - gz^{-k}} \tag{6.1}$$

Es decir, queda caracterizado estableciendo un valor de ganancia g y un retardo k (expresado en número de muestras). A partir del diagrama de bloques (figura 8), podemos expresar la salida del filtro en el dominio del tiempo en la forma:

$$y(n) = x(n - k) + gy(n - k) \tag{6.2}$$

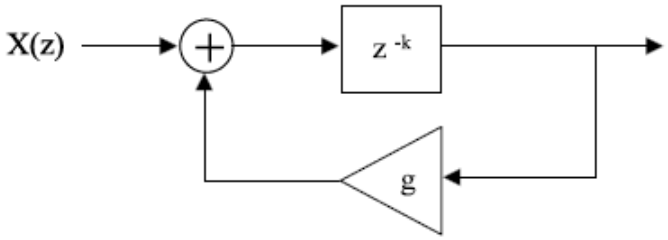


Figura 8: Filtro *comb*.

La respuesta en frecuencia de este filtro contiene k picos periódicos, correspondientes a la frecuencia del polo. Estos picos se corresponden con las frecuencias para las que se producirá reverberación, mientras que las frecuencias en mínimos del filtro decaerán rápidamente [14]. Es por esto que se utilizan varios *comb* en paralelo; con la intención de que conjuntamente cubran todo el espectro.

6.1.2. Filtros *allpass*

El filtro *allpass*, que aparece en la figura 9, se forma mediante una modificación del *comb*. Su función de transferencia es de la forma [6]:

$$H(z) = \frac{-g + z^{-k}}{1 - gz^{-k}} \tag{6.3}$$

Como vemos, este filtro también queda caracterizado estableciendo un valor de ganancia g y un retardo k. A partir del diagrama de bloques (figura 9), podemos expresar la salida del filtro en el dominio del tiempo en la forma:

$$y(n) = -gx(n) + x(n - k) + gy(n - k) \tag{6.4}$$

Los polos del filtro *allpass* son los mismos que los del filtro *comb* anterior, pero ahora aparecen también ceros conjugados; lo que resulta en una respuesta en frecuencia plana (de magnitud uno para todas las frecuencias). La función que cumplen en el algoritmo es incrementar la densidad de ecos para todas las frecuencias [14].

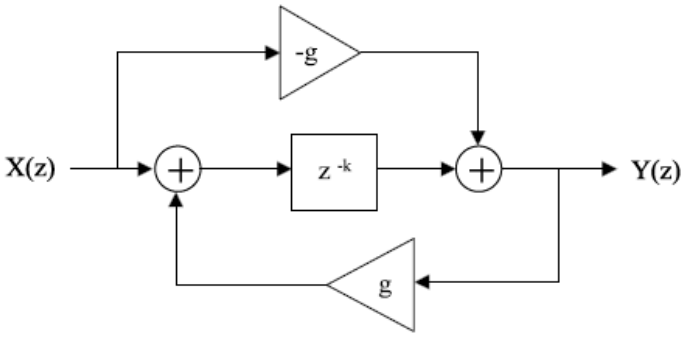


Figura 9: Filtro *allpass*.

6.1.3. Parámetros

Para poder utilizar el sistema que acabamos de presentar, es necesario establecer el valor de retardo y ganancia para los seis filtros.

Para los filtros *comb*, Schroeder estableció que los retardos se deberían elegir tales que la ratio del mayor al menor no sea mayor de 1,5; y más concretamente, que éstos se encuentren en el rango entre 30 y 45 ms. La ganancia de estos filtros se rige por la expresión:

$$g = 10^{-\frac{3k}{F_s T_r}} \tag{6.5}$$

Donde F_s es la frecuencia de muestreo, y T_r es el tiempo de reverberación en segundos. La frecuencia de muestreo vendrá dada por el archivo de entrada de audio, pero el tiempo de reverberación es un parámetro que será necesario ajustar correctamente.

Para los filtros *allpass*, Schroeder propuso retardos de 5 ms y 1,7 ms, y ambas ganancias estarían ajustadas a 0,7. Estos valores pueden modificarse, en particular cuando se pasa como entrada al algoritmo un conjunto de reflexiones tempranas.

6.2. Integración de la reverberación en la respuesta del sistema

La reverberación es solamente una parte de la respuesta de un escenario dado. Como representa la figura 10, primero se reciben el sonido directo y las reflexiones tempranas, ambos tratados en el apartado 4. A esto le sigue la reverberación calculada con los filtros de Schroeder. A diferencia de las implementaciones habituales, y como puede verse en la figura 10, la salida de la respuesta al impulso de las reflexiones iniciales se lleva al reverberador artificial haciendo, de este modo, que el sonido esté correlado y sea más realista.

Sin embargo, esta cola de impulsos, tal como se obtiene de la salida del reverberador, no está colocada justo después de las primeras reflexiones, como debería; ni tampoco presenta la amplitud adecuada para que el decaimiento sea continuo. En su lugar, está superpuesta al resto de la respuesta al impulso. La solución consiste en un sistema como el que muestra la figura 11, con un retardo y una ganancia adecuadas.

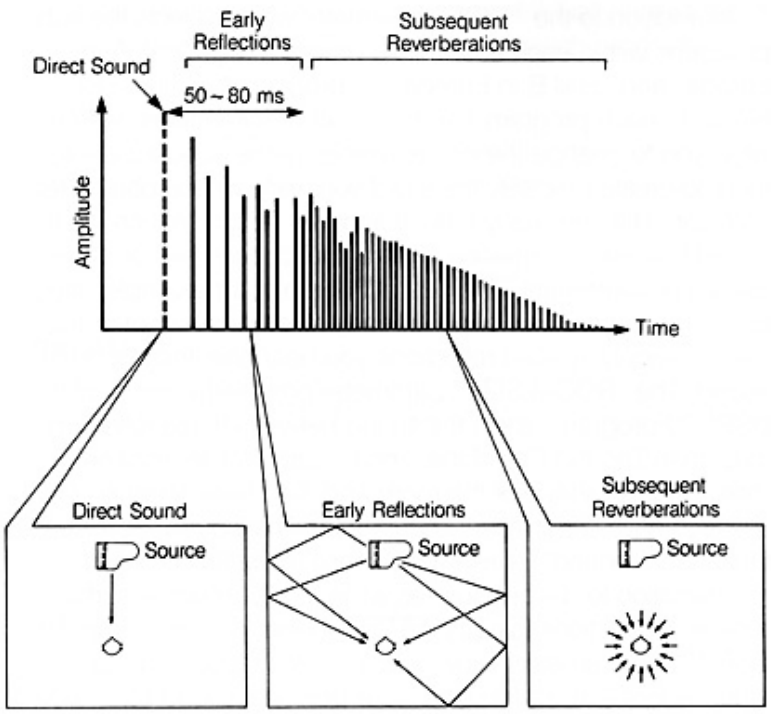


Figura 10: Respuesta al impulso correspondiente a una habitación, donde se distinguen las distintas fases de la propagación.

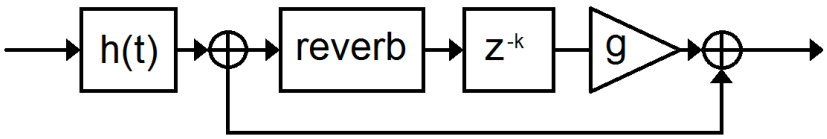


Figura 11: Diagrama de procesado de la señal, en el que se muestra cómo integrar la reverberación calculada con el resto de la respuesta.

6.2.1. Ajuste del retardo

El objetivo es encontrar la muestra de $h(t)$ tras la cual todos los impulsos son tan pequeños y están tan separados en el tiempo que ya no los consideramos relevantes. Pegaremos la cola de reverberación inmediatamente después de esta muestra.

No existe un método definido para encontrar este punto de la respuesta impulsional, por lo que tenemos que crear uno propio. Para ello, observamos la distribución de las amplitudes y la separación temporal de los impulsos en algunas $h(t)$ de ejemplo que hemos generado usando un escenario sencillo (en la figura 15 se encuentra la respuesta impulsional utilizada para este ejemplo).

Vemos que cerca del final de la $h(t)$ las muestras se dispersan y se encuentran más separadas, lo que será el principal indicador a la hora de encontrar el valor de retardo que buscamos.

Definimos la siguiente función en Matlab, aplicada al vector de amplitudes (h) y al vector de tiempos (t) de la respuesta al impulso:

$$R = \frac{diff\,(diff\,(cumsum(t)))}{diff\,(h)} \tag{6.6}$$

Donde:

$$cumsum(x) = [x_1; x_2 + x_1; x_3 + x_2 + x_1; \dots] \tag{6.7}$$

$$diff\,(x) = [x_2 - x_1; x_3 - x_2; x_4 - x_3; \dots] \tag{6.8}$$

$$diff\,(diff\,(x)) = [x_3 - 2x_2 + x_1; x_4 - 2x_3 + x_2; \dots] \tag{6.9}$$

En las imágenes 13 y 14 se representa el proceso que se sigue al aplicar la ecuación 6.6. Concretamente, la figura 13 muestra los resultados parciales del numerador y denominador de la ecuación, y la figura 14 muestra el resultado total.

La ecuación funciona de la siguiente manera:

1. La derivada de la suma acumulada del vector de tiempos será siempre creciente, pero crecerá más deprisa donde las muestras de $h(t)$ estén muy separadas, ya que de un elemento al siguiente el valor temporal subirá mucho; mientras que para muestras muy juntas apenas crecerán ni el retardo ni la derivada.

2. La segunda derivada mostrará picos en aquellos puntos donde la primera crece más rápidamente. Precisamente uno de esos picos será el punto que buscamos.

3. Por otra parte, la derivada del vector de amplitudes empieza en un valor negativo grande, y tiende a cero (siempre desde el lado negativo). Al dividir lo anterior por este vector, las muestras del resultado que corresponden a valores de amplitud pequeños quedan divididas por un valor cercano a cero (resultando valores más altos), mientras que aquellas correspondientes a valores de amplitud grandes se dividen por un valor más alto (quedan atenuadas en el resultado).

Finalmente, se busca el mínimo valor dentro de R, aunque descartando el último valor del vector, que corresponde a la última muestra de la $h(t)$ (hacemos la suposición de que el punto de la respuesta impulsional que buscamos no va a ser el último, basada en los casos estudiados). En la figura 15 se muestra la $h(t)$ que se ha usado de ejemplo, con el punto que se ha calculado para colocar la reverberación.

6.2.2. Ajuste de la ganancia

Tras colocar la reverberación en su sitio dentro del eje temporal, tenemos que ajustarla en amplitud. Para ello, supongamos que g_1 es la amplitud que tendría una muestra de $h(t)$ en el instante en el que colocamos la primera muestra de reverberación. Esta será la amplitud que deba tener finalmente la primera muestra de la respuesta al impulso del sistema de Schroeder.

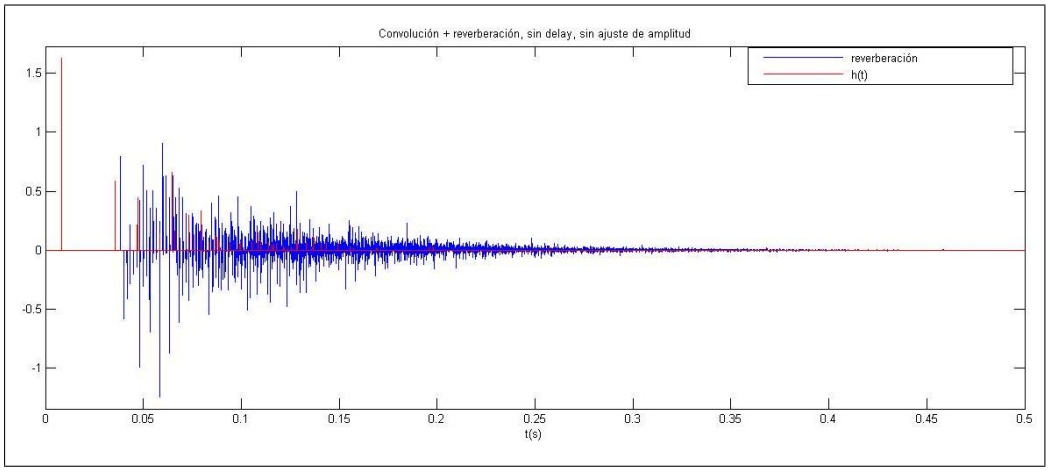
Idealmente, calcularíamos g_1 ajustando la pendiente de $h(t)$ a una exponencial decreciente, para luego interpolar en base a ésta. Sin embargo, podemos suponer que, en el punto en el que hacemos el ajuste, la amplitud ha descendido mucho y la pendiente es prácticamente horizontal. Esto nos lleva a aproximar g_1 por el valor de la muestra anterior de la respuesta impulsional.

Por otra parte, la amplitud que realmente tiene esta primera muestra es conocida, ya que depende sólo de las ganancias de los filtros *allpass*. Concretamente, cada filtro aporta un factor igual a 0,7, según los valores que hemos establecido en el apartado anterior. Así, la ganancia que presenta el reverberador es $g_2 = 0{,}49$.

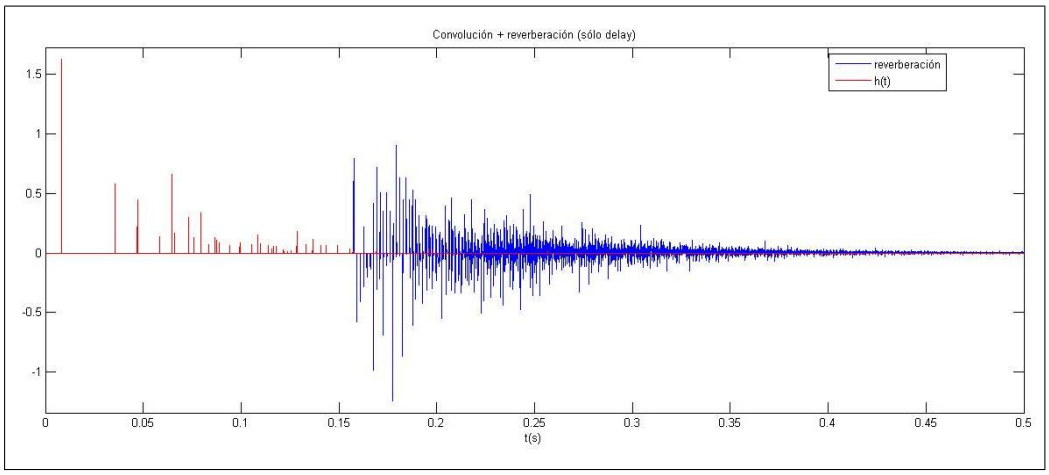
Finalmente, el ajuste en amplitud queda como:

$$g = \frac{g_1}{g_2} = \frac{h(t_1)}{0,49} \tag{6.10}$$

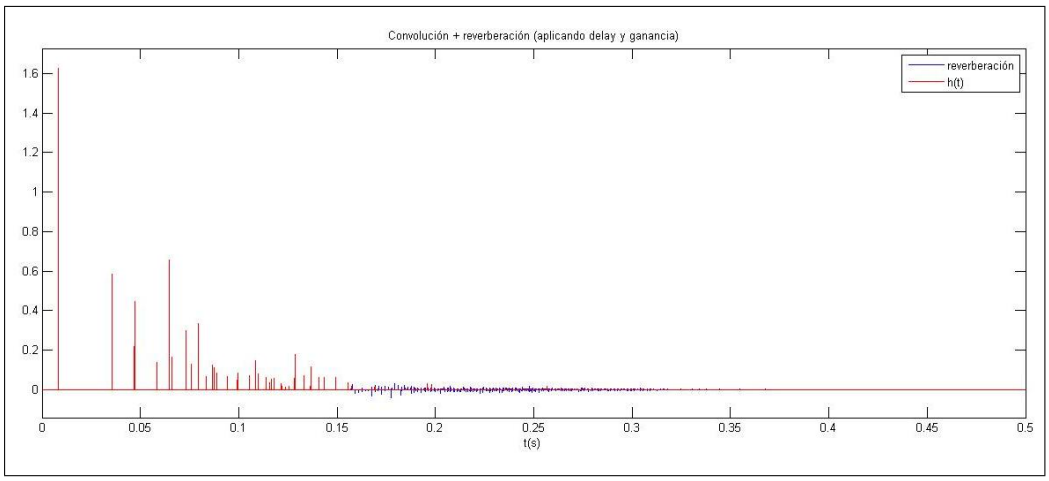
Donde t_1 corresponde con el retardo calculado en el apartado anterior.



(a) Respuesta al impulso (sonido directo y reflexiones tempranas, en rojo), en comparación con la salida del reverberador (en azul), sin ajustar retardo ni ganancia.



(b) Respuesta al impulso (en rojo), y salida del reverberador (en azul) con el retardo ajustado, pero no la ganancia.



(c) Respuesta al impulso (en rojo), y salida del reverberador (en azul) con el retardo y la ganancia ajustados.

Figura 12: Proceso de ajuste de la respuesta del reverberador, en el eje temporal y el de amplitud, dentro de la respuesta global.

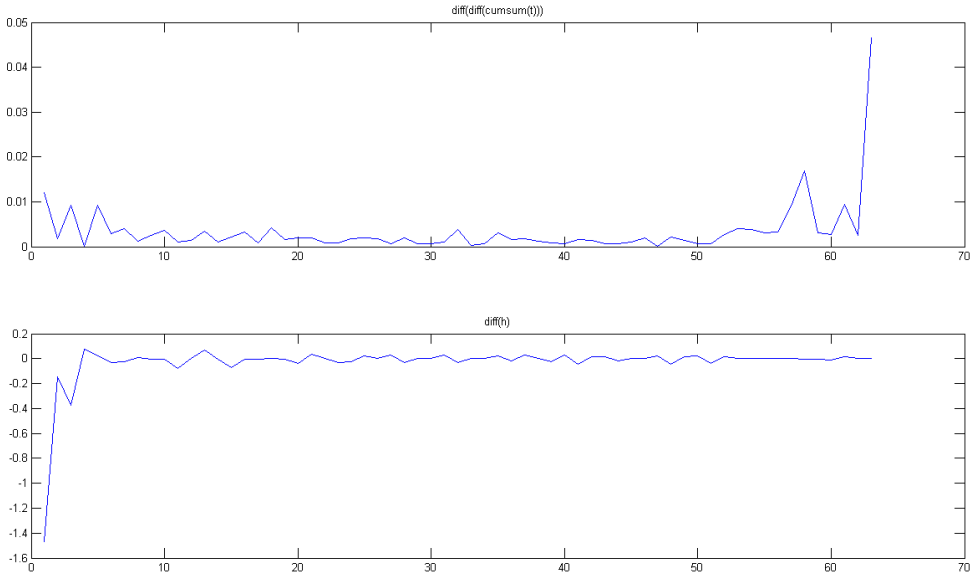


Figura 13: Resultado del numerador y denominador de la ecuación 6.6, para la $h(t)$ de ejemplo.

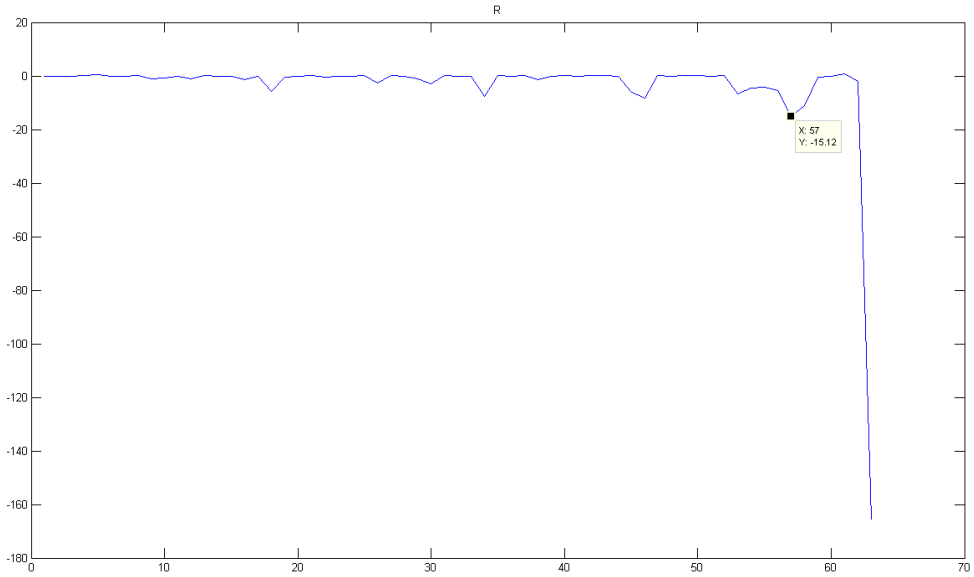


Figura 14: Resultado de la ecuación 6.6, para la $h(t)$ de ejemplo.

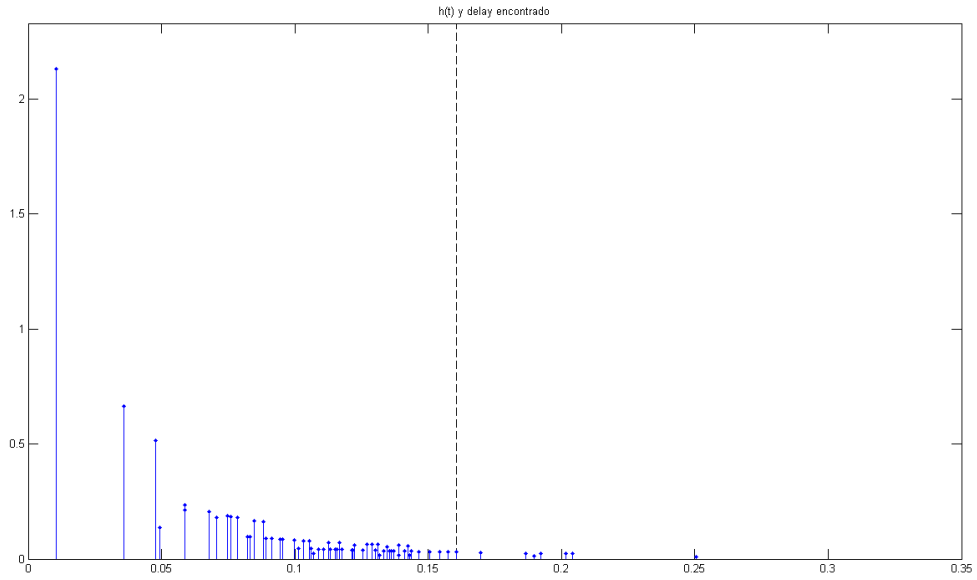


Figura 15: Respuesta impulsional de ejemplo, con el retardo donde se va a colocar la cola de reverberación, calculado por la ecuación 6.6.

7. Otros aspectos

En este apartado se exponen todos los detalles de la implementación que es necesario ajustar y que no se han mencionado anteriormente.

7.1. Parámetros del mallado

Una de las primeras cosas que el programa deberá hacer al iniciar una ejecución es cargar un escenario para el GSound. En el punto 5 se ha explicado cómo extraer la estructura triangulada, pero debemos especificar también los siguientes datos para que ésta quede completamente definida dentro del entorno:

- Posición, dentro de un sistema de coordenadas global, donde fijamos el escenario.
- Escala: podemos asignar un factor de escala al mismo escenario. Este factor determina también la unidad de longitud interna de GSound de modo que equivalga a un metro en unidades físicas reales.

7.2. Parámetros del emisor y el receptor

Para poder calcular una respuesta impulsional necesitamos conocer, además del escenario, la posición de la(s) fuente(s) y del receptor. Pasaremos estos parámetros a GSound cada vez que se modifiquen, o periódicamente, y calcularemos la nueva $h(t)$ cada vez que sean actualizados.

El oyente también se caracteriza por su orientación en el espacio. En GSound se expresa la direccionalidad de éste mediante una matriz de rotación 3x3. Una matriz de rotación es aquella que, si la multiplicamos con un vector de coordenadas (x, y, z), da como resultado el vector rotado.

Para calcular dicha matriz, partiremos de las coordenadas que en aeronáutica se conocen como *yaw*, *pitch*, *roll*. Este sistema expresa la orientación de un objeto mediante tres coordenadas (x, y, z) expresadas en grados, que representan la rotación en cada eje tal como aparece en la figura 16.

La matriz de rotación se puede obtener a partir de estos parámetros mediante las siguientes expresiones [15]:

$$R(\alpha, \beta, \gamma) = R_z(\alpha)R_y(\beta)R_x(\gamma) \tag{7.1}$$

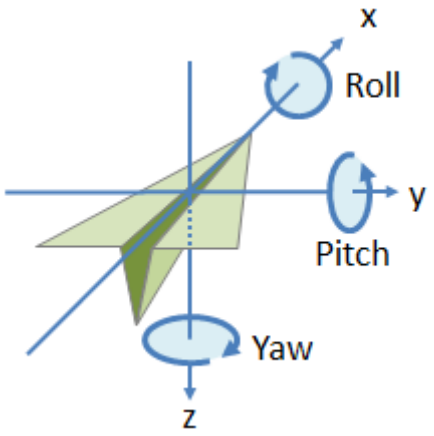


Figura 16: Coordenadas *yaw*, *pitch*, *roll*.

$$R_z(\alpha) = \begin{pmatrix} cos(\alpha) & -sen(\alpha) & 0 \\ sen(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{7.2}$$

$$R_y(\beta) = \begin{pmatrix} cos(\beta) & 0 & sen(\beta) \\ 0 & 1 & 0 \\ -sen(\beta) & 0 & cos(\beta) \end{pmatrix} \tag{7.3}$$

$$R_x(\gamma) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\gamma) & -sen(\gamma) \\ 0 & sen(\gamma) & cos(\gamma) \end{pmatrix} \tag{7.4}$$

La característica de orientación se emplea en GSound para calcular una ganancia diferente para el canal izquierdo y el derecho, de modo que el oyente perciba la dirección de origen del sonido.

Por último, para caracterizar una fuente de audio especificamos su intensidad sonora, y su atenuación lineal y cuadrática, que se utilizarán para calcular la amplitud y el decaimiento de la señal, en función de la distancia recorrida.

7.3. Parámetros del trazado de rayos

Al motor de trazado de rayos se le especifica el número de caminos de propagación que debe lanzar inicialmente (hay que recordar que no todos llegarán a ser útiles). Cuanto mayor establezcamos este parámetro más completa será la simulación, y tendremos mayor densidad de impulsos en el filtro $h(t)$, pero más tiempo llevará también el cálculo del mismo. Los valores manejados habitualmente son entre 100 y 1000.

En principio, si no establecemos ninguna restricción, un rayo podría estar circulando por tiempo ilimitado por un escenario: se propagaría, y chocaría con los sólidos de la escena, perdiendo amplitud, pero, si no establecemos un mecanismo para eliminarlo, seguirá su camino indefinidamente.

Para solucionarlo, definimos dos parámetros limitantes:

- Número máximo de interacciones: un camino de propagación se da por terminado cuando ha tenido más de N interacciones (ya sean una reflexión, difracción o difusión) con el escenario, porque suponemos que en tal punto ya acarrea una atenuación muy alta. Se suele dar valor 4 a este parámetro.
- Tiempo máximo de propagación: si se da el caso de que un rayo encuentra menos obstáculos de los establecidos en el punto anterior durante un tiempo dado, también lo finalizamos. Elegimos un valor lo bastante alto como para que la atenuación introducida por la distancia sea considerable (un valor habitual es 0,5 s).

7.4. Parámetros del reverberador

Para calcular las ganancias de los filtros del reverberador, es necesario establecer primero el parámetro tiempo de reverberación (T_r). En GSound se ha implementado un método que estima el área de las superficies del escenario, y a partir de esto el T_r. Para la estimación del área se utiliza una propagación de rayos desde la fuente. Ya que sólo buscamos un dato aproximado, esta propagación se hará con un número de caminos mucho menor que la realizada desde el receptor. En los escenarios en los que se realizan pruebas, encontramos que este parámetro oscila entre 0,25 y 0,5 s, aproximadamente.

8. Creación de una librería

Todos los procedimientos descritos hasta aquí los empaquetamos dentro de una librería, de forma que podamos acceder a todas las funcionalidades de forma sencilla. La vamos a organizar en dos partes: una se encargará de gestionar el escenario y el cálculo de la respuesta impulsional (lo relativo a GSound), y la otra del procesado de la señal, así como la lectura de archivos de entrada.

8.1. GSound

En el proyecto en Visual Studio incluimos una carpeta de archivos fuente basados en GSound, que implementan todo lo que necesitamos de éste (el motor de trazado, la descripción interna de los escenarios, ...). En todo este código se han realizado las modificaciones necesarias, y se han eliminado algunas clases para dejar en funcionamiento sólamente la parte que nos es útil.

Para poder manejarlo y presentarlo de forma sencilla creamos una clase, llamada `GSoundHandler`, que agrupará las llamadas a funciones de GSound, y almacenará los parámetros necesarios, de forma más conveniente.

`GSoundHandler` incluye funciones para:

1. Inicializar las variables que necesita el motor de renderizado (escenario, fuentes, receptor, ...).

   - `void initializeRender(float maxDelayTime)`
     Inicializa las clases que calculan el trazado de rayos y la respuesta impulsional. Se le especifica el tiempo máximo que permitimos la propagación de cada rayo.
   - `void initializeDemo(std::string meshFile, std::string format, float scale, ofVec3f position)`
     Inicializa un escenario, descrito en el archivo especificado e indicando también el formato (OBJ o **.sm*), con el receptor en la posición $(0, 0, 0)$ y sin ninguna fuente. Además podemos proporcionar un *offset* a la posición del escenario y escalarlo.
   - `void addSource(ofVec3f sourcePosition, float intensity, float linearAttenuation, float quadraticAttenuation)`
     Añade una fuente a la escena, especificando su posición, intensidad y atenuación (lineal y cuadrática).

2. Cambiar la posición y orientación del receptor, y la posición del emisor:

- `void moveListener(ofVec3f newListenerPosition)`
  Modifica la posición del receptor.
- `void rotateListener(ofVec3f newListenerRotation)`
  Cambia la orientación del receptor. Este vector representa las coordenadas *pitch*, *yaw* y *roll* (como x, y, z respectivamente).
- `void moveSource(ofVec3f newSourcePosition)`
  Establece la posición de la fuente.

3. Recalcular la respuesta impulsional:

   - ```
     gsound::ImpulseResponse doSoundPropagation(
                         int maxListPathDepth, int maxListNumRays,
                           int maxSrcPathDepth, int maxSrcNumRays)
     ```
     Aquí concretamos el número de caminos de propagación, de receptor y de fuente (que se usa para estimar el tiempo de reverberación); y el máximo número de reflexiones u otras interacciones con el escenario que puede tener cada rayo.

4. Crear a partir de un fichero OBJ uno equivalente con el formato nativo de GSound **.sm*.

   - ```
     bool saveMesh(std::string OBJFile, std::string fileName,
                                                    bool overwrite)
     ```
     Esta función permite crear archivos en el formato **.sm*. Usando estos, la función `initializeDemo` es más rápida ya que no tiene que ejecutar el *parser* que procesa los OBJ, con lo que reducimos el tiempo de inicialización. Por lo tanto, su uso es opcional.

8.2. Procesado de señal

De forma similar a como gestionamos el trazado de rayos, creamos una clase `InputHandler`, que contiene código para recoger muestras de audio de un fichero, y realizar los cálculos del procesado de señal.

En primer lugar, para manejar ficheros **.wav*, empleamos una clase `Sample` diseñada precisamente para éste propósito. Esta clase tiene una función que abre el archivo y lee la cabecera, de la que recogemos como parámetros el número de canales de la pista, y la velocidad de muestreo. Dispone de otra función que lee una muestra de audio y la devuelve. Con esto queda solucionada la entrada de datos.

La otra tarea que implementa `InputHandler` es el procesado: utilizando lo anterior, recogerá muestras suficientes para llenar un búfer. A continuación, conociendo la respuesta al impulso calculada por `GSoundHandler`, calcula la convolución con el método explicado en el apartado 4. Por último, pasa el resultado de dicha convolución por el filtro de reverberación y devuelve un `float*` del mismo tamaño que la entrada, con el audio final. El tamaño de búfer que se lee en cada ejecución de la función se establece en el programa principal (se explicará con más detalle en 9).

Finalmente, en esta clase creamos los siguientes métodos:

- `void setup(string file, bool isLooping, int outputChannels, float maxDelayTime)`
  Aquí se inicializa la clase `Sample` con el archivo indicado. El *flag* `isLooping` establece si el archivo se reproduce en bucle o una sola vez. Además, especificamos el número de canales de salida que espera el resto del programa (dos por defecto). Por último, `maxDelayTime` es el mismo tiempo que se utiliza en la función anterior para calcular la respuesta impulsional; aquí se necesita ya que hará falta almacenar la entrada y salida anterior en un búfer, y este parámetro determina el tamaño del mismo.

- `float * getAudio()`
  Esta función utiliza la clase `Sample` para leer N (según el tamaño del búfer empleado) muestras y devolverlas en un *array*.

- `float * convolution(float * input, gsound::ImpulseResponse response)`
  Esta función realiza la convolución, además de calcular la cola de reverberación y proporcionar la señal de salida completamente procesada.

- `int getSampleRate()`
  `int getChannels()`
  Son métodos que devuelven la tasa de muestreo y el número de canales del archivo leído.

9. OpenFrameworks

OpenFrameworks es un conjunto de herramientas de código abierto en C++, diseñado para trabajar de forma sencilla e intuitiva con gráficos, audio, vídeo...

Hemos elegido esta herramienta como base para crear una aplicación en la que probar la librería de trazado de rayos. De hecho, también se ha adaptado la clase `GSoundHandler` para que utilice tipos de datos propios de OpenFrameworks, concretamente el tipo `ofVec3f`.

La aplicación consistirá en un entorno gráfico sencillo en el que podamos ver el mismo escenario que se carga en GSound, y movernos por él mediante entradas de teclado. También deberá gestionar la salida de audio a los periféricos.

En los siguientes apartados se presentará la estructura básica de una aplicación en este entorno [16], y la integración en ella de la librería que hemos creado.

9.1. Función `setup`

La función `setup` solamente se ejecuta una vez al comenzar la aplicación, y se utiliza para inicializar los objetos o variables que se vayan a utilizar más adelante.

Las clases descritas en el apartado 8 se inicializan tal como se describe a continuación. Previamente, declaramos cada una de la siguiente manera, junto con una variable para la respuesta impulsional:

```
// Declaraciones:
GSoundHandler * gsoundHandler;
InputHandler inHandler;
gsound::ImpulseResponse response;
```

```
// Inicializar gsoundHandler con el constructor:
gsoundHandler = new GSoundHandler();

// Inicializar las clases encargadas del trazado de rayos:
gsoundHandler->initializeRender(maxDelayTime);

// Cargar el escenario:
gsoundHandler->initializeDemo("data/mesh/cubo", "obj",
                                        meshScale, meshPosition);

// Colocar el listener en el escenario:
gsoundHandler->moveListener(listenerPosition);
gsoundHandler->rotateListener(camera.getOrientationEuler());
```

```
// Añadir una fuente al escenario:
gsoundHandler->addSource(sourcePosition, 6, 1, 0);

// Un InputHandler acompaña a la fuente, manejando su archivo:
inHandler.setup("sounds/acoustics.wav", true, outputChannels,
                                                    maxDelayTime);
```

9.2. Funciones `update` y `draw`

Estas funciones son llamadas en un bucle infinito, primero `update` y después `draw`, hasta finalizar la aplicación. La primera, `update`, se utiliza para actualizar el estado de la aplicación, y realizar los cálculos que se requieran. Después, `draw` se encarga de dibujar en la pantalla. Es posible ajustar la frecuencia a la que se ejecutan, modificando el *framerate*.

En la función `update` tenemos que recoger la posición y la orientación del receptor, y calcular el trazado de rayos de nuevo. El código correspondiente es el siguiente:

```
// Actualiza la posición del receptor:
gsoundHandler->moveListener(listenerPosition);

// Actualiza la orientación del receptor:
gsoundHandler->rotateListener(listenerRotation);

// Calcula una nueva respuesta impulsional:
response = gsoundHandler->doSoundPropagation(4, 1000, 4, 10);
```

9.3. Función `audioOut`

Esta función solicita N muestras de audio cada vez que necesita más para volcarlas a la salida. Para N fijamos un número concreto (normalmente 256), que será el tamaño de búfer con el que trabajar.

Además este método se ejecuta en un *thread* diferente del principal, lo cual nos resulta muy conveniente, ya que es aquí donde se realizará todo el cálculo necesario para filtrar la señal de entrada.

Aquí, la aplicación deberá tomar N muestras de audio y procesarlas con la última respuesta impulsional calculada. El código correspondiente a esto sería como sigue:

```
// Recoger audio del archivo:
input = inHandler.getAudio();

// Filtrar el audio:
output = inHandler.convolution(input, response);
```

9.4. Funciones para gestionar interrupciones

Además de las anteriores, normalmente se necesitan algunas funciones para gestionar interacciones de entrada/salida básicas. En concreto, se dispone de métodos para capturar la entrada de teclado y ratón, de los que empleamos el `keyPressed`. Con este es suficiente para crear un entorno en el que el usuario pueda moverse por el escenario en todas las direcciones (utilizando las cuatro flechas, y las teclas 'WASD' para el movimiento y la rotación del oyente).

10. Conclusiones

Al finalizar el proyecto hemos logrado extraer de GSound un código que puede ser exportado, por una parte. También hemos conseguido escribir una serie de métodos, contenidos en varias clases de C++, que funcionan como una librería; por lo que se han alcanzado los objetivos principales planteados al inicio del mismo.

A lo largo del desarrollo del trabajo, hemos tenido que hacer ajustes y modificar algunos aspectos de la idea original. En particular, cambiamos la plataforma sobre la que implementamos la librería de trazado de rayos: inicialmente se pensó en utilizar Unity, que se utiliza para crear videojuegos. Sin embargo, decidimos utilizar OpenFrameworks ya que éste se utiliza directamente sobre Visual Studio, lo que haría más sencilla la integración de la librería. Además, OpenFrameworks proporciona una forma muy cómoda de gestionar la salida de audio, en la que sólo tenemos que llenar de datos un búfer mientras la herramienta se ocupa de las interrupciones y del *thread* correspondiente.

Por otro lado, aunque al principio pensamos en la posibilidad de compilar la librería, por ejemplo creando un archivo **.dll*, finalmente nos decidimos en contra de esta idea. Durante el desarrollo del trabajo, hemos usado varias librerías y clases en C++ creadas por otras personas. Encontrar estas clases en forma de código fuente nos ha sido muy útil para poder ver cualquier detalle de su funcionamiento, o incluso modificarlas para servir mejor a nuestros propósitos. Así pues, decidimos dejar el código abierto a cualquier mejora o modificación futura.

La parte del proyecto que ha presentado más dificultades ha sido trabajar con GSound, ya que es un código bastante cerrado, con una arquitectura compleja y muy entralazada. Resultó complicado partirlo en distintos módulos, para aprovechar sólo parte de su funcionalidad, y utilizarlo en otro entorno.

Finalmente, algunas ideas que no se han podido llevar a cabo dentro del proyecto: por una parte, desplazar el procesado relativo al trazado de rayos a un *thread* dedicado exclusivamente a esta tarea. Esto podría formar parte de la librería, o hacerlo como parte del programa de demostración en OpenFrameworks.

Opcionalmente, se podría añadir la compatibilidad con otros formatos, tanto de mallado 3D como de audio.

Por último, se podría considerar combinar este sistema con uno que realice la simulación acústica utilizando un método de elementos finitos: se separaría la señal de entrada en frecuencia, y se procesaría con trazado de rayos las altas frecuencias, y con elementos finitos las bajas, para luego juntar todo a la salida.

11. Bibliografía

[1] Schissler, C. and Manocha, D., 2011, February. Gsound: Interactive sound propagation for games. In Audio Engineering Society Conference: 41st International Conference: Audio for Games. Audio Engineering Society.

[2] Kleiner, M., Dalenbäck, B.I. and Svensson, P., 1993. Auralization-an overview. Journal of the Audio Engineering Society, 41(11), pp.861-875.

[3] Montell Serrano, R.E., 2011. Sistemas de realidad virtual para el estudio del campo acústico de edificios del patrimonio artístico-cultural.

[4] Isbert, A.C., 1998. Diseño acústico de espacios arquitectónicos (Vol. 4). Univ. Politèc. de Catalunya. P. 371.

[5] Wikipedia. Ray Tracing (physics).
[https://en.wikipedia.org/wiki/Ray_tracing_(physics)]

[6] Beltrán, J.R. and Beltrán, F.A., 2002. Matlab implementation of reverberation algorithms. Journal of New Music Research, 31(2), pp.153-161.

[7] Digital Audio. Creating a WAV (RIFF) file.
[http://www.topherlee.com/software/pcm-tut-wavformat.html]

[8] Schroeder, M. R. 1962. Natural Sounding Artificial Reverberation J. Audio Engineering Society. Vol. 10, N0. 3.

[9] Wikipedia. Polygon Mesh: File Formats.
[https://en.wikipedia.org/wiki/Polygon_mesh#File_formats]

[10] Wikipedia. Wavefront .obj file.
[https://en.wikipedia.org/wiki/Wavefront_.obj_file]

[11] Ratcliff, J.W., 2006. Efficient Polygon Triangulation.

[12] Eberly, D., 1998. Triangulation by ear clipping. Geometric Tools, LLC.

[13] Handbook for sound engineers. The new audio cyclopedia 2nd ed-G Ballou (Ed)

[14] Gardner, W.G. 1998. Chapter 3. Reverberation Algorithms, in Kahrs, M. and Brandenburg, K. Editors. Applications of Digital Signal Processing to Audio and Acoustics. Kluwer Academic Publishers.

[15] Steven M. LaValle, 2006. Yaw, pitch, and roll rotations.
[http://msl.cs.uiuc.edu/planning/node102.html]

[16] Arturo Castro. How openFrameworks works: setup, update, draw [http://openframeworks.cc/ofBook/chapters/how_of_works.html]

[17] Botteldooren, D., 1995. Finite-difference time-domain simulation of low-frequency room acoustic problems. The Journal of the Acoustical Society of America, 98(6), pp.3302-3308.

[18] Kay, T.L. and Kajiya, J.T., 1986, August. Ray tracing complex scenes. In ACM SIGGRAPH computer graphics (Vol. 20, No. 4, pp. 269-278). ACM.

[19] Wikipedia. Bounding Volume Hierarchy. [https://en.wikipedia.org/wiki/Bounding_volume_hierarchy]

[20] Taylor, M., Chandak, A., Mo, Q., Lauterbach, C., Schissler, C. and Manocha, D., 2010. i-sound: Interactive gpu-based sound auralization in dynamic scenes. Tech. Rep. TR10-006.

ANEXOS

Anexo 1: Trazado de rayos

Principio teórico

En física, el trazado de rayos es un método empleado para calcular los caminos de ondas o partículas a través de un entorno con regiones que tienen distintas características de propagación, absorción y reflexión. En estas circunstancias, los frentes de onda pueden curvarse, cambiar de dirección o reflejarse en superficies, complicando el análisis. El trazado de rayos resuelve este problema lanzando cierta cantidad de haces idealmente estrechos (rayos) y haciéndolos avanzar a través del medio en pasos discretos. Propagando un gran número de rayos se puede obtener un análisis detallado de casos complejos.

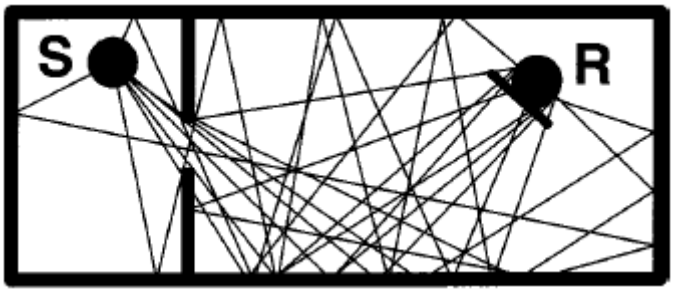


Figura 17: Ejemplo de trazado de rayos en un entorno sencillo en dos dimensiones. La trayectoria de éstos, desde el emisor (S) hasta el receptor (R), representa las reflexiones causadas por las paredes de la escena.

El método asume que el frente de onda puede modelarse con una gran cantidad de rayos muy estrechos, y que cada rayo es recto al menos localmente (es decir, en una pequeña distancia). En cada paso, el motor de trazado hará avanzar los rayos dicha distancia, y calculará la nueva dirección de cada uno teniendo en cuenta la geometría y las propiedades materiales del entorno. Con la nueva dirección se repite el proceso hasta calcular caminos completos. Si la simulación incluye objetos sólidos (normalmente siempre será así) se comprobará si el rayo intersecta con alguno en cada paso, ajustando su trayectoria donde se encuentre una colisión (dando lugar a reflexiones). Otros parámetros de la onda, como su intensidad, también se pueden actualizar a cada paso.

Algoritmo para aplicaciones en tiempo real

El objetivo de este sistema de propagación de audio es poder simular la interacción del sonido con el escenario en entornos dinámicos y en tiempo real. Por ello, la elección del algoritmo utilizado es crucial.

Existen otras soluciones, numéricas y geométricas, para calcular la propagación de audio. Los métodos numéricos consisten en resolver la ecuación de ondas mediante aproximaciones de diferencias finitas en el dominio del tiempo (*FDTD* [17]). Son precisos y producen los resultados más fieles a la realidad, pero requieren mucho tiempo y memoria, por lo que no pueden usarse en ningún sistema en tiempo real [1].

Los métodos geométricos son, en general, más manejables ya que, al contrario que los anteriores, suelen permitir movimiento en la escena. El más preciso es el de las fuentes imagen, que también está basado en rayos. Las fuentes son reflejadas recursivamente sobre cada superficie plana en la escena para formar fuentes imagen, que representan los caminos de reflexión correspondientes (ver figura 18). Este método es muy lento, ya que el tiempo de ejecución crece exponencialmente con la profundidad de reflexión.

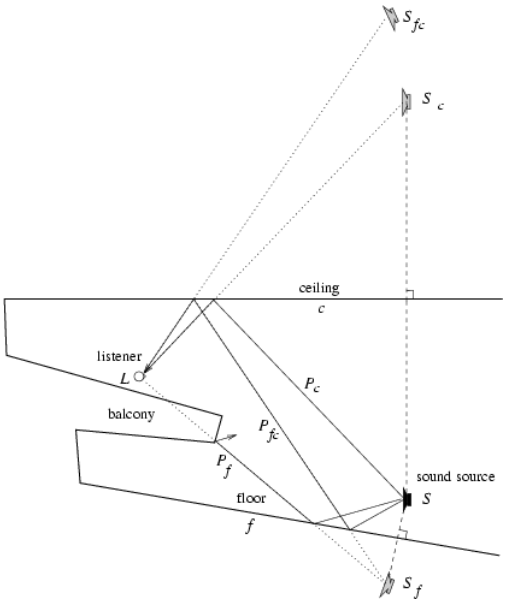


Figura 18: Ejemplo de método de fuentes imagen. Las fuentes imagen S_c y S_{fc} representan reflexiones de primer y segundo orden, y son visibles para el oyente; mientras que la reflexión del suelo S_f queda escondida por el saliente (*balcony*).

Otras soluciones geométricas conocidas son similares al trazado de rayos, utilizando en su lugar haces bi o tridimensionales (trapecios o pirámides truncadas). Debido a la complejidad que introduce trazar volúmenes a través del medio, tampoco alcanzan tiempo real.

El trazado de rayos para propagación de audio tiene varias ventajas sobre las técnicas anteriores: es más sencillo manejar escenas dinámicas, requiere menos procesamiento previo, y por su simplicidad es más adecuado para simulación en tiempo real.

Bounding Volume Hierarchy

Para implementar un algoritmo de este tipo en un escenario de, por ejemplo, un videojuego o un simulador, lo primero será contar con una representación válida del entorno. Ésta consistirá en algún tipo de descripción geométrica que podrá ser más o menos detallada. Debido a la cantidad de información en escenas complejas, el proceso de búsqueda de colisiones de un rayo con un sólido puede ser muy costoso.

Para reducir la carga computacional y así acelerar el cálculo se suele recurrir a una aproximación de los objetos de la escena por volúmenes de geometrías sencillas que los envuelven. Éstos a su vez son jerarquizados en una estructura de árbol. Esta técnica se conoce como *Bounding Volume Hierarchy* (*BVH*) [18]. En la figura 19 se representa una posible agrupación de varios objetos relativamente complejos utilizando rectángulos como figuras envolventes.

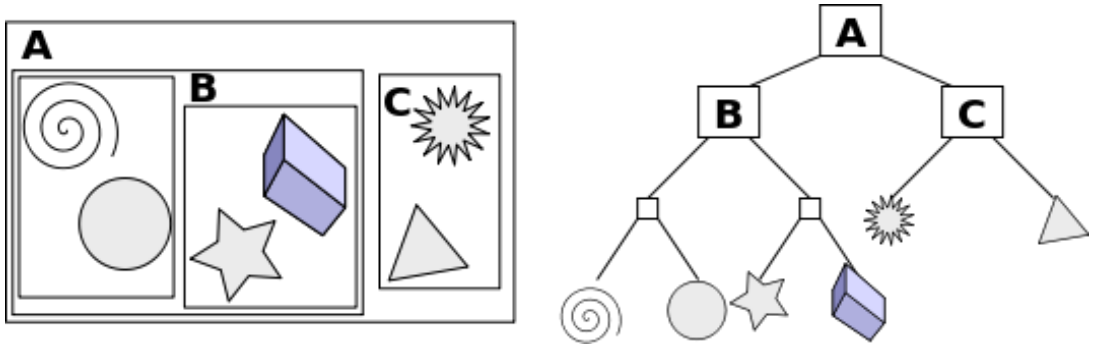


Figura 19: Ejemplo de *BVH* para unas pocas figuras, usando rectángulos como volúmenes simplificados.

La simplificación radica en que, cada vez que se traza un rayo, se comprueba si hay colisión sólo con los nodos 'padre' de la jerarquía, y sólo en el caso de que así sea se desciende por el árbol y se hace el cálculo teniendo en cuenta la geometría real en lugar de la simplificada [19]. Por ejemplo, en la figura 19 se buscaría para los rayos trazados una colisión con A, y sólo si se encuentra se repetiría el cálculo para B o C y posteriormente para los objetos complejos que corresponda.

Implementación en GSound

Trabajo previo

GSound está construido sobre un sistema previo, iSound [20], que también implementa el trazado de rayos.

iSound utiliza un algoritmo de trazado hacia delante, lo cual quiere decir que los rayos parten de cada fuente de audio del escenario. Se lanzan uniformemente distribuidos en todas las direcciones, y son propagados hasta alcanzar un límite de recursión definido por el usuario. La propagación se realiza via reflexión especular o difracción.

Para encontrar los puntos en los que hay difracción, se hace un procesado previo de la escena en el que se identifica qué triángulos (teniendo en cuenta que el modelado 3D del escenario se compone de triángulos) tienen un borde difractante. Al lanzar los rayos, se comprueba cuáles están muy cerca de dichos bordes (se considerará cerca si se encuentran a una distancia menor que un umbral dado). Posteriormente, se crean rayos secundarios desde la región encerrada por el triángulo adyacente.

Por otro lado, se define una esfera, de radio dado por el usuario, alrededor del oyente. Los rayos anteriores que colisionen con esta esfera se guardan en una lista de posibles caminos de propagación. La información que se guarda de cada rayo es el conjunto de triángulos con los que ha interaccionado.

Por último, se comprueban todos los posibles caminos de propagación encontrados haciendo el recorrido de vuelta desde el oyente, para descartar aquellos que presenten oclusión.

Tanto para los rayos trazados desde las fuentes como desde el receptor, se utiliza el método de fuente imagen, replicando el punto de origen en cada reflexión o refracción.

Ahora que se tiene una lista de caminos válidos, para cada uno se calculan algunos datos: la distancia total de la fuente al oyente, la dirección del oyente a la primera fuente imagen, y un valor de atenuación dependiente de la frecuencia causado por la interacción con los materiales.

Trazado de rayos desde el receptor

Un cambio importante de GSound es que lanza rayos desde el receptor (representa la propagación hacia atrás) [1]. Se observó que las reflexiones y difracciones tempranas que se perciben con más intensidad tienden a concentrarse en el área alrededor del oyente. Al emitir rayos desde cada fuente, sólo unos pocos llegan al receptor, y éstos pueden no ser los más relevantes perceptualmente, necesitando en ese caso aún más rayos para encontrar todos los caminos de propagación necesarios. El método propuesto trata de encontrar el mayor número posible de caminos válidos y al mismo tiempo tener que descartar menos. Tiene la ventaja añadida de que el volumen de cálculo no crece con el número de fuentes (puede haber más de una fuente de audio pero sólo un receptor).

El algoritmo empieza lanzando una distribución uniforme de rayos desde la posición del oyente. Éstos son propagados de la misma manera que en iSound. Al intersectar con un triángulo de la escena, se produce una reflexión especular.

Del mismo modo, los caminos encontrados son validados después partiendo de las fuentes, comprobando si existe un camino de reflexión válido, usando un método de fuentes imagen similar al anterior.

Mientras se crean los caminos de propagación desde el oyente, éstos se almacenan en una serie de tablas: todos los rayos con una reflexión en una, con dos en otra tabla, y así hasta la máxima profundidad de reflexión que se haya establecido. Estas listas se comprueban para cada rayo, de modo que nunca se producen duplicados; y se actualizan cada vez que se encuentra una colisión.

Finalmente, para cada camino de propagación válido, se calculan los mismos parámetros que para iSound: distancia total, dirección, y atenuación dependiente de la frecuencia.

Conservación de los caminos de propagación

Dada la naturaleza aleatoria de los rayos usados para determinar la visibilidad, las secuencias válidas obtenidas de un *frame* a otro son a menudo inconsistentes. Esto resulta en caminos de propagación que aparecen y desaparecen aun cuando tanto la fuente como el receptor están quietos.

GSound resuelve este problema conservando las listas de caminos visibles mencionadas en el apartado anterior. Al principio de cada *frame* todas las secuencias de triángulos se comprueban para ver si los rayos son todavía válidos. Se borran todos aquellos que no lo sean, y los demás se conservan. Después, se hace todo el procedimiento de trazado de rayos y se actualizan las listas.

De esta forma, la cantidad de secuencias se mantiene más estable, obteniendo mejor coherencia entre *frames*, lo que es un problema común en otras implementaciones. Pero la mayor ventaja es que hace falta lanzar muchos menos rayos en cada iteración. La colección de caminos de propagación se refina y actualiza a cada paso, en lugar de crearla desde cero.

Implementación paso a paso

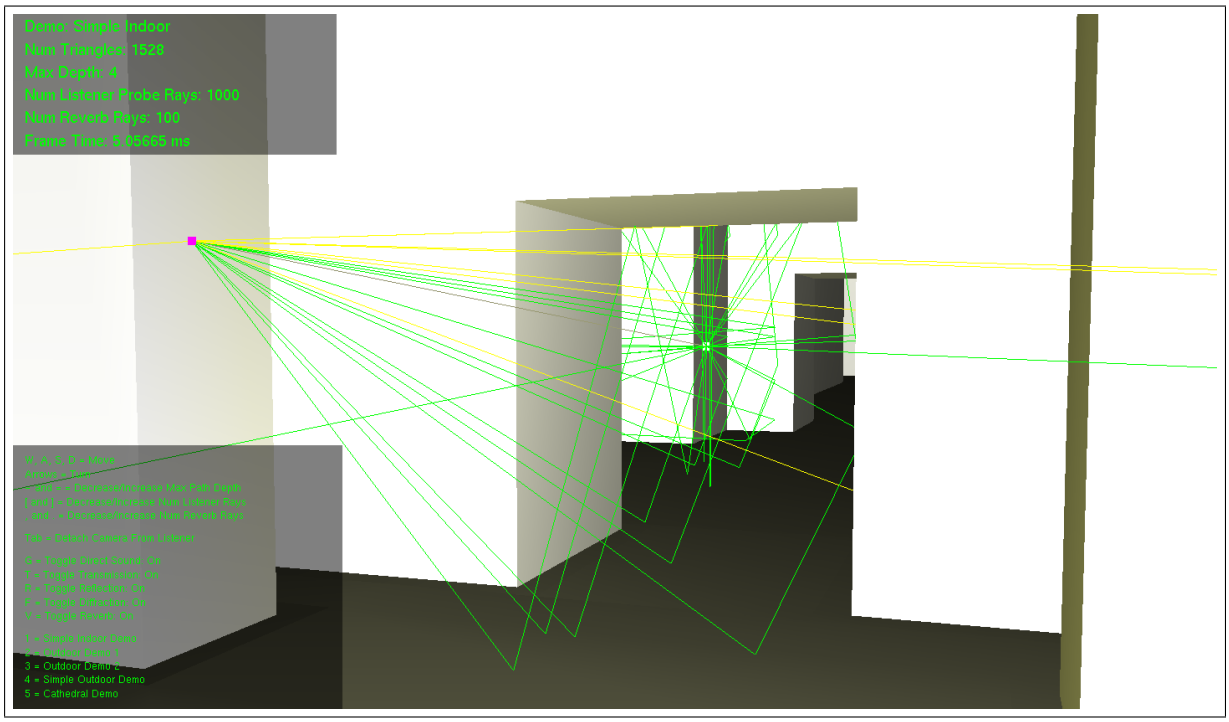
En la figura 21 se presenta un diagrama con el paso a paso de la implementación en GSound de trazado de rayos. Los primeros pasos corresponden a la carga e inicialización del escenario y los caminos de propagación.

A continuación, se lleva a cabo un proceso iterativo para cada uno en el que se hace avanzar una pequeña distancia, se comprueba si ha colisionado con algún objeto del escenario, y se actualiza la dirección de propagación en consecuencia. Se repite el procedimiento hasta que el rayo alcanza un máximo establecido de iteraciones o de reflexiones (lo que ocurra antes).

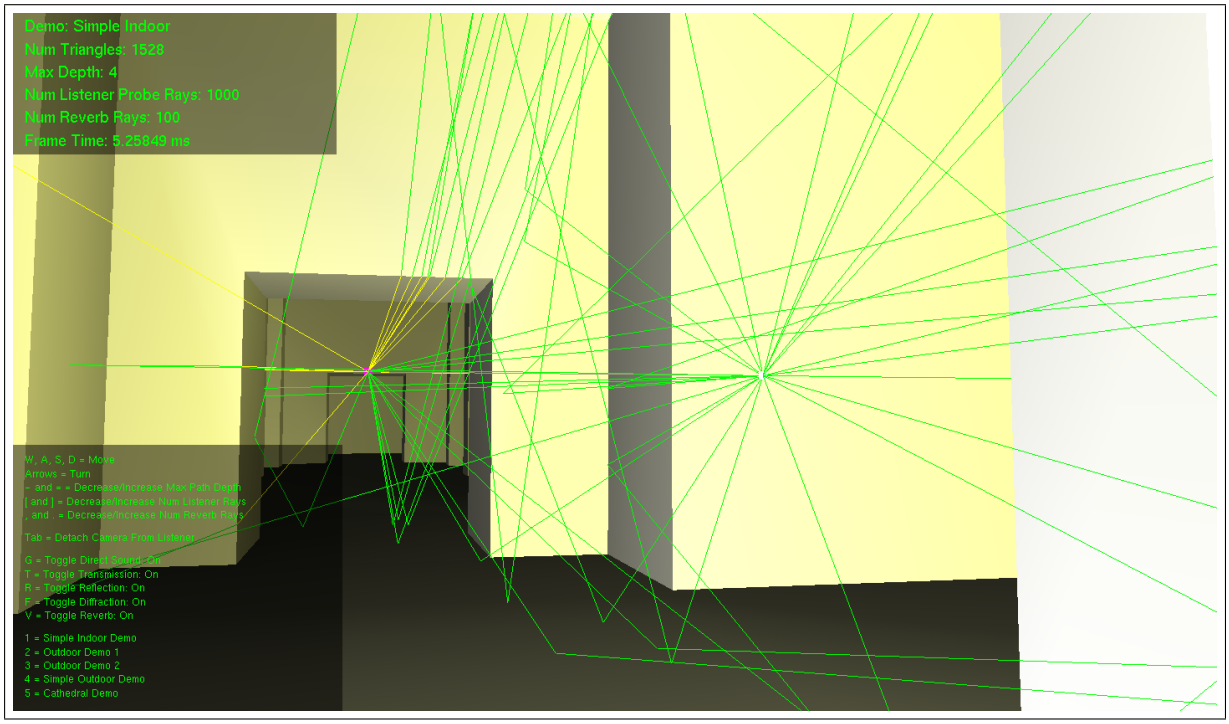
Una vez que todos los rayos han alcanzado el final de su camino, se comprueba cuáles han llegado hasta las fuentes de sonido. Para ello, se crea una esfera alrededor de cada una, y se buscan las colisiones. Todos los rayos que no alcancen estas esferas se eliminan.

Para los caminos que conservamos, se realiza la propagación en el otro sentido, del mismo modo que antes, y se descartan los que presenten oclusión.

En este punto, se tiene una lista definitiva de caminos de propagación, todos ellos relevantes. Sólo queda calcular parámetros, para cada uno, de distancia recorrida, dirección con la que parte de la fuente, y atenuación total.



(a) Desde el punto de vista del emisor.



(b) Desde el punto de vista del oyente.

Figura 20: Visualización del trazado de rayos de GSound. El punto rosa representa al emisor, y el punto blanco al receptor. Se puede ver cómo del oyente parten rayos uniformemente repartidos en todas las direcciones, y cómo del emisor sólo parten aquellos caminos relevantes para el cálculo. En verde se representa la propagación con reflexiones, en amarillo la refracción, y en gris el camino directo.

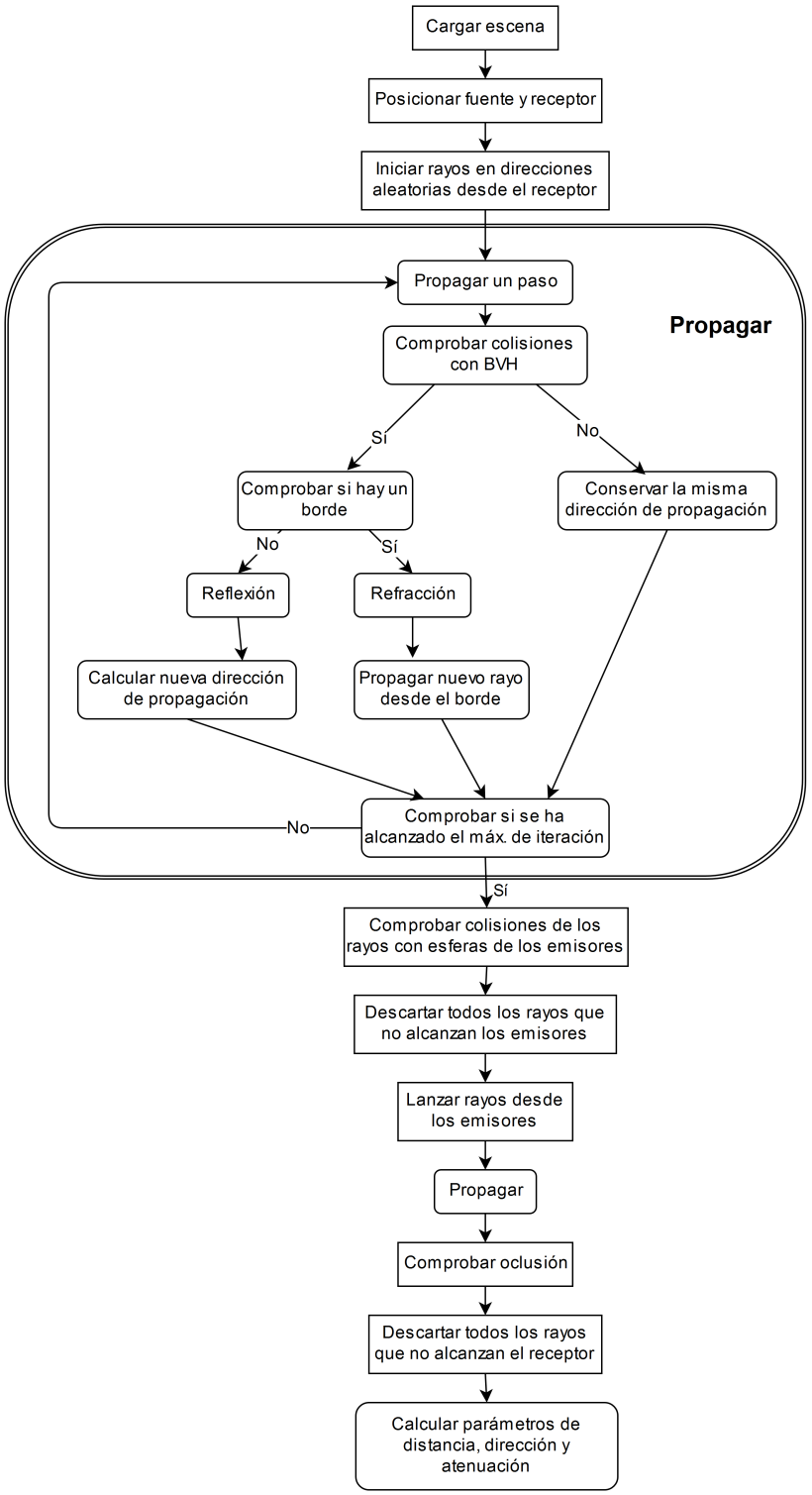


Figura 21: Diagrama de flujo de la implementación de trazado de rayos. Los cuadros con esquinas redondas corresponden a acciones que se ejecutan una vez para cada rayo, y los demás se realizan una vez para todo el programa.