UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Reproducción de ficheros Opus con OpenAL: precarga vs "streaming"

Apellidos, nombre	Agustí i Melchor, Manuel (magusti@disca.upv.es)
Departamento	Departamento de Informática de Sistemas y Computadores (DISCA)
Centro	Escola Tècnica Superior d'Enginyeria Informàtica Universitat Politècnica de València

1 Resumen de las ideas clave

OpenAL es [1] un motor de audio 3D capaz de renderizar el sonido que escucha un oyente inmerso en una escena sonora tridimensional, de modo que el usuario se vea envuelto entre las **fuentes de sonido** dispuestas a su alrededor. OpenAL **no se encargará de cargar audio de un fichero** en disco, sino de enlazarlo a una fuente de sonido, cuando ya está cargado en memoria y sin compresión. Esta ha sido una decisión de diseño [2] y [3].

Si se quiere **importar el audio** a partir de un fichero, se ha de recurrir a otras librerías de desarrollo del ámbito multimedia para la lectura y decodificación de formatos de audio. En este artículo, **se verá** el uso del formato de audio OGG, con el esquema de compresión Opus [4] a través de las funciones de la librería *libopus,* para llevar a memoria el audio en un formato que sea compatible con las estructuras de OpenAL. Puesto que el peso de los ficheros de audio puede ser importante, **se explorarán ejemplos de código** que implementan las dos estrategias actuales de reproducción de sonido: precarga y *streaming*.

2 Objetivos

OpenAL almacena el audio en *buffers* de memoria y los asocia a las fuentes (*sources*), que son las encargadas de posicionar el sonido en el espacio y aplicarle efectos. Es labor del desarrollador de una aplicación cargar el sonido a partir de, p. ej., el acceso a ficheros de audio, el uso del micrófono, la síntesis de voz o el uso de un dispositivo MIDI. Cuando la cantidad de datos a cargar es importante, hay que valorar cuándo y cómo se puede hacer. Por ello hay que hablar de las estrategias de reproducción de audio a partir de la precarga ("*static playback*" en la terminología de OpenAL) del audio o de la reproducción en continuo (o en *streaming*).

El presente documento está encaminado a ofrecer una perspectiva inicial de cómo ampliar el conjunto de formatos de ficheros que puede utilizar el motor de audio 3D OpenAL. Como habitualmente el tamaño de los ficheros de audio es grande, es pertinente hablar de cómo abordar su carga en memoria. **No es el objetivo de este documento** dar una solución global para todos los formatos, ni adentrarse en el detalle de cómo es uno por dentro, sino mostrar cómo incorporar uno y que sirva esta discusión para otros casos similares.

A partir del estudio de los ejemplos que se abordan, el lector será capaz de:

- Examinar las estrategias de reproducción de sonido digital y observar las diferencias entre precarga y *streaming*.
- Exponer la estrategia para incorporar el uso del esquema de compresión de audio Opus en OpenAL.
- Instalar y compilar una aplicación que pueda hacer uso del formato Opus sobre OpenAL, con las funciones de la biblioteca *libopus*.

Para plantear la solución, es necesario revisar la relación entre audio digital, su uso en un motor de audio como OpenAL, las posibilidades de uso de un formato de fichero como OGG y la codificación Opus.

Nota: En el texto, cuando se haga referencia a una de las figuras que se incluyen en él, se utilizará la abreviatura "fig." listada en la RAE[1] para este menester.

---

1 Lista de las abreviaturas convencionales más usuales en español. Disponible en <http://www.rae.es/diccionario-panhispanico-de-dudas/apendices/abreviaturas>.

3 Introducción

El sonido se ha venido almacenando en una serie de soportes analógicos como el disco de vinilo o la casete, como muestra la fig. 1. Con la llegada del computador, el sonido se ha de codificar en **"formato digital"** para ser procesado. A partir de esta codificación aparecerán soportes como los discos ópticos y formatos como el CD-Audio y el MP3.

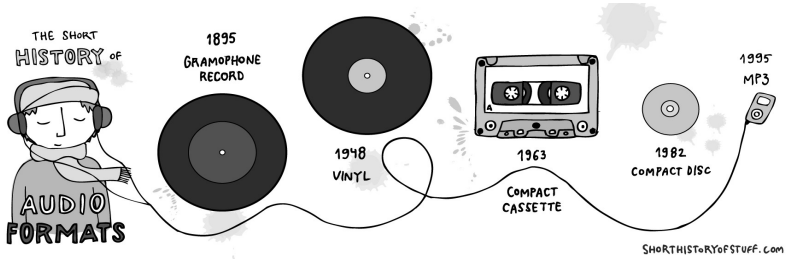

Figura 1: Ejemplos de soportes y formatos de audio. Imagen de [7].

La información de audio ha de ser digitalizada para usarse y almacenarse en un computador. Esto supone la codificación de la señal analógica en digital. El esquema de codificación binario que se utiliza se denomina PCM (de *Pulse Code Modulation*), para referirse a la serie de instantáneas del valor de la señal de sonido que han sido adquiridas con una **frecuencia de muestreo** dada. Estas muestras (*samples*) han sido convertidas en valores numéricos a partir de la elección de un **tamaño de muestra** (o número de bits), que se emplea para indicar su valor en digital dentro del rango dinámico de la señal. Para reproducir el audio, hay que saber exactamente qué valores se han utilizado en la codificación digital para poder interpretar los datos que contiene el fichero.

Además, los ficheros de audio han ido incorporando mecanismos de compresión [5] para reducir la ocupación de los ficheros. Así, se habla de dos tipos de compresión: "sin pérdidas" (*lossless*) y "con pérdidas" (*lossy*). Opus pertenece a este segundo grupo: elimina parte de la información de audio a guardar basándose en características del sistema auditivo humano. Estas técnicas son más complejas que las de sin pérdidas, en tanto en cuanto necesitan tomar más aspectos en consideración para codificar el audio y, por tanto, también para su lectura y decodificación.

Ahora **se plantea utilizar ficheros Opus**. Así que no solo es cuestión de leer el contenido del fichero, sino que además hay que descomprimir la información de audio contenido en él. Por ello va a revisar la operativa de OpenAL y la especificación de Opus para encontrar cómo unir ambos.

3.1 Reproducción y formatos de fichero de audio en OpenAL

OpenAL es capaz de reproducir audio, en PCM, desde memoria. En el caso de que se quiera cargar un fichero (p. ej. en formato WAVE, que es uno de los básicos y más extendidos en el momento de la definición de OpenAL) existen dos aproximaciones: precarga y reproducción en continuo (*streaming*).

La precarga supone llevar a memoria, por completo, el contenido del fichero de audio, para después reproducirlo. La fig. 2 muestra de forma gráfica como la librería *Audio Library Utility Toolkit* (ALUT) [6] complementa a OpenAL, en tanto que es capaz de leer archivos WAVE de disco (con la llamada al sistema *read*) y extraer de ese contenido el audio en PCM para asociarlo (con *alutCreateBufferFromFile*) a un *buffer* de OpenAL.

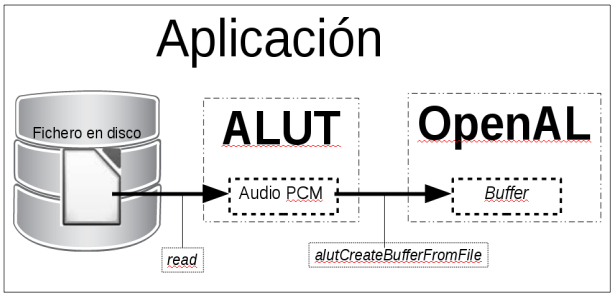

Figura 2: Esquema básico de gestión de formatos de ficheros en OpenAL con ALUT.

Cuando el formato del fichero no es reconocido por ALUT, como p. ej. el caso de Opus, es necesario entender el formato del fichero para ser capaz de extraer el audio, que además puede necesitar ser descomprimido. Para ello es necesario conocer la especificación a fondo del contenedor y el esquema de compresión. La fig. 3 resume, de forma gráfica, los cambios respecto a la fig. 2. En este caso, el uso de la librería *libopusfile* permite extraer el audio descomprimido en PCM para asignárselo a un *buffer* de OpenAL a través de la instrucción *alBufferData.* En cualquier caso la asociación es estática, esto es, se ha cargado totalmente el audio en memoria para asignarlo un *buffer*. Cuando el fichero es grande, esta solución es muy costosa en términos de ocupación de memoria.

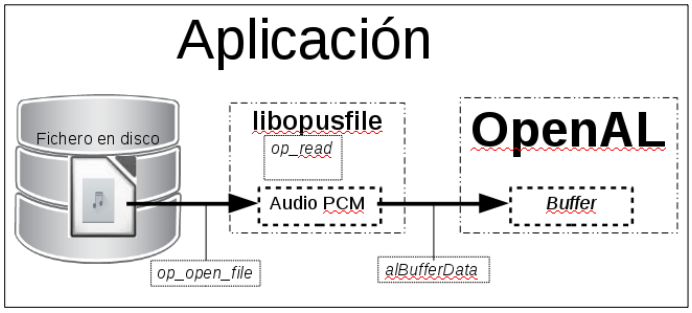

Figura 3: Esquema ampliado de la gestión de formatos de ficheros para OpenAL con librerías externas: en este caso libopus.

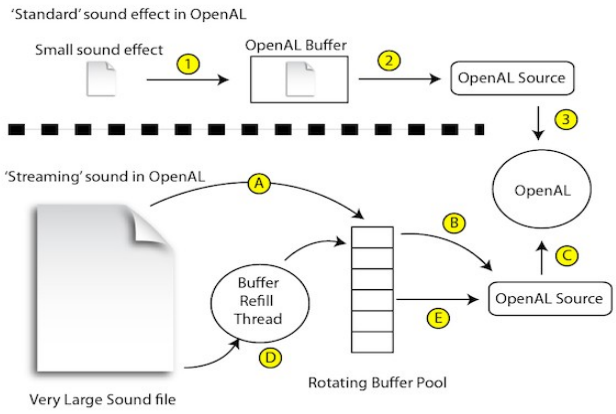

Figura 4: Esquema gráfico de las estrategias de precarga vs streaming en OpenAL. Imagen extraída de [10].

La fig. 4 resume y compara los caminos de la precarga y el streaming, de manera gráfica, estableciendo la similitud y las diferencias entres estas dos

estrategias y marcando con números y letras las diferentes etapas de cada opción.

La aproximación de reproducción en continuo está basada en cargar dinámicamente el audio: esto es, tener en memoria los datos necesarios para tener sonido durante un tiempo, mientras se carga otro "trozo" de audio. Es lo que se conoce como reproducción en continuo o *streaming*. OpenAL no proporciona la funcionalidad de *streaming* sobre un objeto [3], sino que proporciona un mecanismo más general de asociación de una pila de *buffers* a una fuente. Esto permite que la aplicación pueda especificar cuántos, de qué tamaño y si se reutilizan o se descartan tras su uso. Así se consigue maximizar la eficiencia de uso del recurso.

La de *streaming* está dirigida a grandes cantidades de sonido y permite asociar varios trozos de sonido a una fuente, de manera que la reproducción del sonido dependa que exista una pila de *buffers* (paso A de la fig. 4), que son desapilados (B y E) conforme se utilizan (C) y que van siendo rellenados (D), al tiempo que terminan de ser utilizados en la reproducción del audio.

3.2 Formato OGG y compresión Opus

Sobre un fichero en formato OGG, de Xiph.Org [4], se guardan los datos de sonido comprimidos con un esquema de compresión como Vorbis, Opus, Tremor, FLAC, Speex o CELT. Opus es un codec que proporciona buena calidad incluso a valores bajos de ratios de bits[2]. Está libre de patentes[3] y se utiliza desde aplicaciones de transmisión de voz, VoIP, música distribuida por Internet, pasando por el almacenamiento y la difusión en vivo o *streaming*.

¿Cómo se puede utilizar Opus en una aplicación? A través de varias librerías que componen el kit de desarrollo de software (SDK o *Software Development Kit*) de referencia de la especificación del formato Opus: la librería **opusfile**[4], que depende de **libopus** (la referencia de implementación, con operaciones en coma flotante y reales), **libogg** (para definir el acceso al contenedor) y **opusurl** (que implementa el acceso como recursos disponibles bajo descarga.

En este trabajo, se utilizan las funciones de **opusfile**, porque ofrecen la mayor simplicidad en el código al abstraer de los detalles más "íntimos" del formato: los que permiten la identificación, posicionamiento del contenido (*streams*) en el fichero, decodificación y liberación de recursos relativos al contenedor y a la compresión. De esta librería, aquí, utilizaremos el tipo *OggOpusFile* y las funciones *op_open_file,* o*p_channel_count*, *op_pcm_total*, *op_read* y *op_free*.

4 Desarrollo

Para ilustrar con ejemplos prácticos el uso de Opus como extensión a OpenAL, se van a revisar ahora los ejemplos de uso de Opus con OpenAL [9] en modo precarga y *streaming*. Como el código está completo en la referencia indicada, la exposición se centra en las etapas que hace los ejemplos diferentes y se obvian algunos pasos que se dejan fuera de los listados y que se indican con "…".

---

2 Véanse las comparativas en "Codec Landscape. Quality vs Bitrate" <http://opus-codec.org/comparison>.

3 La especificación completa y la implementación de referencia están descritas en el RFC 6716, "Definition of the Opus Audio Codec". <https://tools.ietf.org/html/rfc6716>.

4 El API completo de esta librería se puede consultar en <https://www.opus-codec.org/docs/opusfile_api-0.6/index.html>.

Se necesita instalar los paquetes de desarrollo del SDK de Opus con la orden:

```
$ sudo apt-get install libopusfile-dev libopus-dev
```

Se puede comprobar que están instalados y su versión con las órdenes:

```
$ pkg-config opus --modversion
$ pkg-config opusfile --modversion
```

Para compilar los ejemplos siguientes lo haremos con la ayuda de *pkg-config*, en cuyo caso la compilación del primer ejemplo se puede hacer con:

```
$ gcc opusal.cpp -o opusal `pkg-config openal --cflags --libs` \
   `pkg-config opusfile --cflags –libs`
```

o directamente especificando las librerías, en el segundo ejemplo, con la orden

```
$ gcc opusal_streaming.cpp -o opusal_streaming -I/usr/include/opus
-lopenal -lopusfile
```

4.1 Ejemplo de precarga

El ejemplo de código de *opusal* [9] muestra cómo se puede utilizar el formato Opus en una aplicación propia, mediante el método de precarga para llevar el audio a memoria. Véase el listado 1, que es una versión simplificada de la referencia original, en tanto a que no incluye los pasos de corrección de errores por simplicidad del código mostrado. Se marcan las etapas con un número entre paréntesis al principio de la línea. A diferencia de otros ejemplos de uso de OpenAL, este tiene dos pasos relevantes:

1. El encargado de acceder al fichero OGG y decodificar de él el audio comprimido con Opus. Es el marcado con el número 1.
2. Reproducir el audio. Está descompuesto en dos partes, la propia de reproducir el audio (marcado con **2.1** en el listado 1) y la espera a que termine su reproducción para terminar la aplicación (marcado con **2.2**).

```
int main(int argc, char **argv) {
      ALuint testBuffer,  testSource;
      ...
(1)   load_opus(testBuffer, argv[1]);       // Load the file into the
buffer.
      ...
(2.1) alSourcePlay(testSource);       // Play the source.
      ...
      int sourceState;   // Keep playing until we're finished.
(2.2) do    {
            alGetSourcei(testSource, AL_SOURCE_STATE, &sourceState);
      } while (sourceState != AL_STOPPED);
      ...
      return 0;
}
```

Listado 1: Esquema de programa principal que utiliza el método de precarga.

```
   int load_opus(ALuint buffer, const char *filename) {
      int error = 0;
(1.1) OggOpusFile *file = op_open_file(filename, &error); // Open the file.
      ...
      // Get the number of channels in the current link.
(1.2) int num_channels = op_channel_count(file,-1);
      // Get the number of samples (per channel) in the current link.
(1.3) int pcm_size = op_pcm_total(file,-1);
      ...
      // Opus always uses 16-bit integers, unless the _float are used
      ALenum format;
(1.4) if (num_channels == 1)   {     format = AL_FORMAT_MONO16;     }
      else if (num_channels == 2)    { format = AL_FORMAT_STEREO16;  }
      else {...}
      // Allocate a buffer big enough to store the entire uncompressed file.
(1.5) int16_t *buf = new int16_t[pcm_size*num_channels];
      ...
      // Keep reading samples until we have them all.
(1.6) while (samples_read < pcm_size) {
            // op_read returns number of samples read (per channel), ....
(1.7)       int ns = op_read(file, buf + samples_read*num_channels,
                        pcm_size*num_channels, 0);
            …
            samples_read += ns;
            …
      }
      op_free(file);     // Close the opus file.
      // Send it to OpenAL (which takes bytes).
(1.8) alBufferData(buffer, format, buf, samples_read*num_channels*2, 48000);
      …
   }
```

Listado 2: Secuencia de pasos para la precarga de audio con el SDK de Opus.

El paso 1 asume la complejidad de cargar el fichero Opus. El listado 2 lo muestra con detalle. Está estructurado en dos bloques:

1. Reservar espacio en memoria para todos los datos (1.5), para lo que es necesario leer el formato (1.1) del archivo contenedor (OGG) averiguar las propiedades del mismo que figuran en el archivo: número de canales (1.2) y número de muestras (1.3), para obtener la propiedad equivalente en OpenAL (1.4).
2. Cargar el audio desde fichero, para lo que es necesario ir leyendo trocitos (1.6) y descomprimiéndolos (1.7).

Terminada esta secuencia, los datos ya están en memoria en una variable *buf*, que puede ser asociada (1.8) a una fuente cuyo identificador se guarda en la variable *buffer.*

Hay que señalar un par de comentarios sobre algunos "números mágicos" que aparecen. El primero se refiere a que la función *op_read* siempre devuelve valores de 16 bits, pero hay otras funciones posibles como se puede ver en la documentación de *libopusfile*. El segundo tiene que ver con las operaciones de *libopusfile* que, como dice en la documentación[5], siempre trabajan a 48 kHz. La frecuencia de muestreo original no se preserva en la secuencia comprimida con pérdidas. Está guardada en la cabecera del fichero, así que el resultado de la descompresión puede ser remuestreado, si se desea.

4.2 Ejemplo de reproducción en *streaming*

Con la base del ejemplo de precarga en mente, se puede pasar a explorar cómo emplear la estrategia de reproducción en continuo (*streaming*) que nos permite ir cargando el contenido del fichero (utilizando el SDK de Opus) al tiempo que se va reproduciendo.

```
   int stream_opus(ALuint source, const char *filename) {
      int error = 0;
(1.1) OggOpusFile *file = op_open_file(filename, &error);      // Open
file.
      ...
      // Get the number of channels in the current link.
(1.2) int num_channels = op_channel_count(file,-1);
      ...
      const int num_buffers = 2;      // The number of buffers ...
      ALuint buffers[num_buffers];
      alGenBuffers(num_buffers,buffers);
      for (int cur_buf = 0; cur_buf < num_buffers; ++cur_buf)
            fill_buffer(buffers[cur_buf],file);
      alSourceQueueBuffers(source,num_buffers,buffers);
(2.1) alSourcePlay(source);
      ...
(2.2) while (update_stream(source,file))    {
            // … a _minimum_ time to be kept asleep.
            usleep(1000*1000*960/48000);}
      alSourceUnqueueBuffers(source,num_buffers,buffers);
      ...
      op_free(file);     // Close the opus file.
   }
```

Listado 3: Carga y reproducción de los buffers.

Este ejemplo [9] lo denomina *opusal_streaming*. Se va a ir haciendo mención a los pasos de la estrategia de precarga para comparar. En este caso, la

5 Disponible en <https://www.opus-codec.org/docs/opusfile_api-0.6/index.html>.

secuencia:de pasos se ha redistribuido para dar una solución que permita, como se ha comentado, ir cargando el contenido del fichero a trozos en memoria, que pueden ser reproducidos. Ambas tareas, se simultanean en el flujo de la aplicación. Es por esto que, en este ejemplo, el código que nos interesa está todo encapsulado en la función *stream_opus* (véase listado 3), el resto de líneas se dedican a tareas "administrativas" (como mostrar mensajes de la puesta en marcha o de error, inicializar y liberar recursos asociados a OpenAL) de la aplicación.

La implementación de este ejemplo reduce a dos el número de *buffers,* a utilizar de forma cíclica, para alternar la carga y reproducción de los trozos del fichero de audio, como muestra el listado 3. Es importante hacer notar que no se obtiene el número total de muestras contenidas en el fichero, a diferencia del ejemplo de precarga (véase el identificador 1.3 del listado 2): aquí no se crea un *buffer* de ese tamaño máximo, sino varios de tamaño fijo (*buffer_size*) que son reutilizados. En este código se utilizan operaciones tanto de *libopusfile*.como de OpenAL. Estas últimas se encargan de la reproducción del sonido PCM ya cargado, como en el anterior paso 2.1, pero ahora hay diferencias importantes en la secuencia de instrucciones que espera a que termine la reproducción, puesto que ahora se va haciéndo por trozos.

La función *update_stream*, véase listado 4, muestra el reemplazo del contenido de los buffers y la secuencia de pasos de OpenAL encargada de gestionar una pila de esos *buffers,* para permitir la alternancia cíclica entre los disponibles. Aquí, por tanto, se hace uso de operaciones de OpenAL exclusivamente.

```
   int update_stream(ALuint source, OggOpusFile *file) {
      int num_processed_buffers = 0;  ALuint currentbuffer;

      // How many buffers do we need to fill?
      alGetSourcei(source, AL_BUFFERS_PROCESSED, &num_processed_buffers);

      ALenum source_state;
      alGetSourcei(source, AL_SOURCE_STATE, &source_state);
      if (source_state != AL_PLAYING) { printf("Source not playing!\n");
(2.1)         alSourcePlay(source);}

      // Unqueue a finished buffer, fill it with new data, and re-add it
      while (num_processed_buffers--) {
            alSourceUnqueueBuffers(source, 1 ,&currentbuffer);
            if (fill_buffer(currentbuffer, file) <= 0)   return 0;
            alSourceQueueBuffers(source, 1, &currentbuffer);
      }
      return 1;
   }
```

Listado 4: Secuencia de pasos para el reemplazo de los buffers ya reproducidos.

La carga final del sonido desde fichero se hace en la función *fill_buffer*, listado 5, donde se rellena uno de esos *buffers* de tamaño fijo. Aquí solo se hace uso de

operaciones de *libopusfile*. El tamaño del *buffer* está establecido para 960 muestras, que el autor del código destaca como algo a mejorar en la implementación. Actualmente, *libopusfile* recomienda un espacio mínimo para, al menos, 120 ms de sonido a 48 kHz por canal. Esto supone, aproximadamente unas 5760 muestras por canal.

```
   int fill_buffer(ALuint buffer, OggOpusFile *file) {
      const int buffer_size = 960*2*2; // Let's have a buffer ...
      int16_t buf[buffer_size];        // 960 muestras, 2 canales y 2 bytes

      int samples_read = 0;
(1.2) int num_channels = op_channel_count(file, -1);
      ...
      ALenum format;     // ... the openAL format based on channels
(1.4) if (num_channels == 1)   { format = AL_FORMAT_MONO16; }
      else if (num_channels == 2) { format = AL_FORMAT_STEREO16;     }
      else  { … }
(1.5) while (samples_read < buffer_size) { // keep reading samples.
            // op_read returns number of samples read (per channel),
            // and accepts number of samples which fit in the buffer,
            // not number of bytes.
            int ns = op_read(file, buf + samples_read*num_channels,
                     (buffer_size-samples_read*num_channels), 0);
            …
            if (ns == 0) break;
            samples_read += ns;      }
(1.8) alBufferData(buffer, format, buf, samples_read*num_channels*2, 48000);
      return samples_read;
   }
```

Listado 5: Función encargada de rellenar un buffer.

5 Conclusión

El motor de audio 3D, OpenAL, parte de una especificación reducida para conseguir un impacto mínimo en el consumo de recursos de una aplicación. Sus capacidades de importación de ficheros se pueden ampliar utilizando una librería genérica como ALUT. La aparición de nuevos formatos hace necesario el uso de bibliotecas especializadas en formatos que, cada vez más, incluyen esquemas de compresión más elaborados. En ese sentido, se ha revisado el camino que sigue la información de audio desde fichero hasta los componentes de OpenAL para proponer dos ejemplos de uso del formato de audio comprimido OGG Opus.

Puesto que el peso de los ficheros de audio puede ser importante y para seguir ahondando en estrategias que reduzcan el impacto en el sistema del uso de

audio, estos ejemplos (sin perdida de generalidad) han mostrado cómo implementar las estrategias de reproducción de sonido de precarga y en modo continuo (*streaming*).

El lector puede ahora experimentar con estos ejemplos y decidir, en función del tamaño del sonido a reproducir y el número de veces que se hará, cual es la mejor manera para incorporarlo a su aplicación. Los motores de videojuegos suelen distinguir entre sonidos de efectos (cortos en duración y que se repiten muchas veces) y la música o banda sonora que suele estar presente durante más tiempo. ¿Se te ocurre cómo implementar cada una de estas opciones? ¿Por qué no lo compruebas?

6 Bibliografía

[1] OpenAL. Disponible en <http://www.openal.org>.

[2] Loki Software. (2000). OpenAL 1.0.Specification. Disponible en <https://pdfs.semanticscholar.org/831a/72e74a6f63dafb1ff74dfa5e311f416bc238.pdf>.

[3] OpenAL 1.1 Specification. (2005). Disponible en <http://www.openal.org/documentation/openal-1.1-specification.pdf>.

[4] Xiph.org. Vorbis audio compresión. Disponible en <https://xiph.org/vorbis/>.

[5] RHA. (2014). *An Introduction To High Resolution Audio Formats - Part 1*. Disponible en <https://www.rha-audio.com/ca/blog/92028/an-introduction-to-high-resolution-audio-formats-part-1>.

[6] The OpenAL Utility Toolkit. Disponible en <http://distro.ibiblio.org/rootlinux/rootlinux-ports/more/freealut/freealut-1.1.0/doc/alut.html>.

[7] Lang, Z. (2013). The short history of audio formats. Disponible en <https://shorthistoryofstuff.wordpress.com/2013/06/28/the-short-history-of-audio-formats/ >.

[8] Opus Interactive Audio Codec. Disponible en <http://opus-codec.org/>.

[9] Gow, D.. (2013). Using Opus with OpenAL. Disponible en <https://davidgow.net/hacks/opusal.html >.

[10] Britten, B. (2010). Streaming in OpenAL. Disponible en <http://benbritten.com/2010/05/04/streaming-in-openal/>.