UNIVERSITAT POLITÈCNICA DE VALÈNCIA

OpenAL con Python: generación de señales básicas

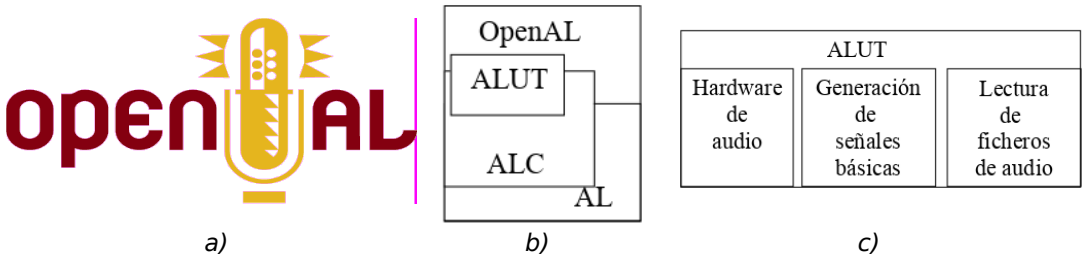
Apellidos, nombre	**Agustí i Melchor, Manuel** (magusti@disca.upv.es)
Departamento	**Departamento de Informática de Sistemas y Computadores (DISCA)**
Centro	Universitat Politècnica de València

1 Resumen de las ideas clave

Es posible desarrollar aplicaciones de audio 3D (audio espacial o envolvente) con un SDK como OpenAL [1], Figura 1a, sobre C/C++ y para diferentes sistemas operativos. El estándar se estructura en dos capas (Figura 1b): ALC y AL que incluyen las operaciones de más bajo nivel que constituyen el motor de audio 3D. De modo opcional al estándar, se ha creado una tercera capa de alto nivel que facilita el desarrollo ofreciendo unas operaciones básicas relacionadas con el audio ya implementadas, es la capa denominada "The OpenAL Utility Toolkit" [2] (abreviado como ALUT[1]) que incluye (Figura 1c) tres bloques:

- Funciones de alto nivel para inicialización y configuración del hardware.
- Generación de señales de sonido básicas, como ondas senoidales, cuadradas, triangulares, dientes de sierra o ruido blanco.
- Y proporcionar un interfaz de acceso a ficheros multiplataforma. Para ello, ALUT implementa un mínimo interfaz de acceso al contenido de ficheros de audio, limitado a la lectura de ficheros WAVE monofónicos o estéreo.

ALUT no es una parte obligatoria del estándar, por lo que se puede encontrar aplicaciones desarrolladas con OpenAL que no hacen uso de ninguna funcionalidad de ALUT.



a) *b)* *c)*

Figura 1: OpenAL: (a) logo, (b) arquitectura interna y (c) servicios de ALUT.

Si se trabaja en Python las opciones que implementan OpenAL no ofrecen la capa de ALUT, así que hay que suplirlas o complementarlas con otras opciones de Python y, en algún caso, habrá que implementarlas. **En este trabajo se aborda** cómo se puede tener un nivel de operatividad comparable con ALUT en lo que respecta a la generación de señales básicas se puede realizar implementando las funciones matemáticas que las definen.

2 Objetivos

Una vez que el lector haya recorrido con detenimiento este documento y explore el código que se adjunta, podrá desarrollar sus propios ejemplos de uso de sonidos simples con OpenAL utilizando Python, en su computador, y con ello:

[1] *The OpenAL Utility Toolkit* (ALUT) <http://distro.ibiblio.org/rootlinux/rootlinux-ports/more/freealut/freealut-1.1.0/doc/alut.html>.

- Explicar qué opciones hay para desarrollar aplicaciones con el motor OpenAL en Python.

- Explorar ejemplos de desarrollo utilizando el lenguaje Python a partir de disponer en memoria de un sonido descomprimido.

- Explorar los principios básicos de síntesis de sonido a través de la generación de formas de ondas básicas.

No es un objetivo de este artículo explicar Python, así que se asume que el lector tiene conocimientos básicos del mismo. Aunque si viene de otros lenguajes como C o Java, encontrará muchas similitudes. Tampoco lo es explicar OpenAL en profundidad, para ello se propone consultar artículos introductorios sobre OpenAL como [7] y [8].

3 Introducción

Para implementar una aplicación que utilice el sonido 3D con OpenAL se hará una breve revisión de la arquitectura que subyace en este estándar. Antes de entrar a ver un ejemplo de desarrollo, se examinará la situación en Python al respecto de la disponibilidad de una implementación de OpenAL y su instalación.

3.1.1 Introducción a OpenAL

OpenAL [1] es un motor de audio 3D, esto es, utiliza audio cargado en memoria para generar sonido espacial (también denominado *surround o audio envolvente)*. Lo que permite *renderizar* el sonido que escucha un oyente inmerso en una escena sonora tridimensional, de modo que, él mismo, se "vea" envuelto entre las fuentes de sonido dispuestas a su alrededor.

Para desarrollar una aplicación multiplataforma que haga uso del sonido 3D se puede implementar accediendo directamente a los servicios que proporciona cada sistema operativo o bien hacer uso de un estándar como OpenAL, véase la Figura 2a. Este nos ofrece un nivel de abstracción sobre el sistema operativo. El esquema básico de trabajo que ofrece OpenAL para "renderizar" el audio, Figura 2b, se basa [3] en dos objetos:

- Los *buffers*, que contienen los diferentes sonidos sin compresión y obtenidos desde ficheros o generados de manera sintética.

- Estos se asignan a objetos de tipo fuente (*source*), que reciben como entrada el sonido y los parámetros que definen su situación, orientación y velocidad, así como también la atenuación, la direccionalidad de la fuente, etc. De esta forma se puede calcular cómo se van a comportar estas fuentes de sonido.

Finalmente, todos los sonidos con *renderizados* en la situación que define la posición del oyente a través del mezclador ("output mix"), para proporcionarle al oyente todos los sonidos parametrizados con los valores globales de la escena como la velocidad de

transmisión del sonido, si se tiene en cuenta el efecto *Doppler*, posición y velocidad del oyente, etc.

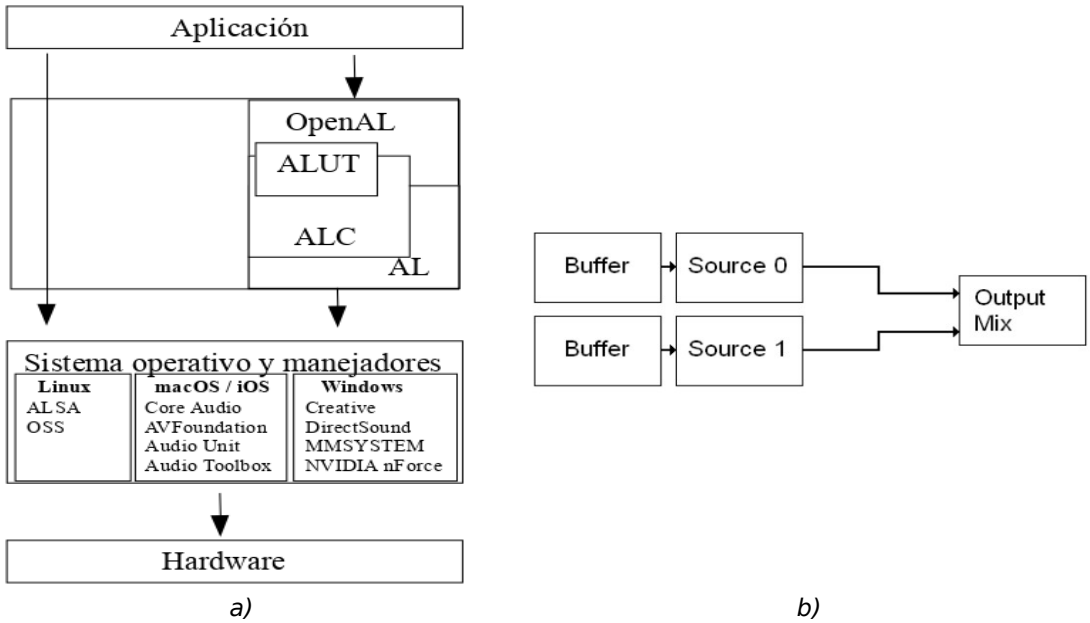


Figura 2: Arquitectura de una aplicación multiplataforma que hace uso del sonido: (a) con los servicios del SO o usando el estándar OpenAL y (b) flujo básico de información en OpenAL (imagen extraída de [3]).

El trabajo relativo a proveer de sonido a los *buffers* se facilita en OpenAL, a través de ALUT, que proporciona operaciones de generación de señales de sonido básicas o de la lectura archivos de audio en formato WAVE. Si el formato no está soportado por ALUT es posible realizar el trabajo manualmente o con una librería específica para ese formato.

En OpenAL los ficheros se leen y se cargan completos en la memoria principal antes de ser reproducidos, es lo que se denomina el modo precarga (*static playback* en la terminología de OpenAL) y también es posible reproducir el audio en continuo (o *streaming*), en este trabajo se asume que se realiza la reproducción en modo precarga.

3.1.2 OpenAL en Python

Si se trabaja en Python, las opciones disponibles para usar sonido envolvente son varias y se puede agrupar en dos grandes vías:

- Desde el apartado de Audio en la página Wiki de Python[2] se habla de *pysonic*, un *wrapper* para FMOD (que ofrece funciones de audio 3D y efectos de sonido). FMOD está disponible para un gran número de plataformas, pero es de código propietario. Además, según el sitio web[3] de descarga de este SDK, la última actualización es de 2013 y, puesto que no es código abierto, depende exclusivamente del fabricante su continuidad.

[2] El Wiki de Python está en <https://wiki.python.org/moin/Audio/>.

[3] Véase en <https://sourceforge.net/projects/pysonic/>.

- En cambio, en el apartado de "PythonGameLibraries" de ese mismo Wiki[4] se habla de *PyOpenAL* [5] como un *binding* de OpenAL para Python. Pero no se ha actualizado desde 2019[5].
  Así que buscando en el repositorio de software para el lenguaje de programación Python, el Índice de paquetes de Python (PyPI), se puede encontrar el proyecto PyAL [5] que proporciona *python-openal 0.4.1* en la última actualización de 2022.

Por lo que el resto del artículo se va a centrar en el uso de PyAL, por lo que sería bueno empezar por instalar PyAL y tener a mano su documentación. Lo primero es sencillo[6], se puede realizar con la orden:

```
$ pip install python-openal
```

La segunda cuestión, la documentación, es un poco más compleja debido a que, como se describe en la documentación de PyAL [6], este consiste en dos módulos y cada uno de ellos tiene un enfoque diferente y está en proceso de completarse:

- `openal`, que es un *wrapper* para la especificación del estándar OpenAL 1.1. Es la parte que la documentación llama "Direct OpenAL interaction" que abarca los niveles AL y ALC y que es en la que se centra este artículo.
  Para poder ver el API (el conjunto de operaciones de OpeAL) que ofrece, hay que acceder al código fuente de los ficheros *al.py* y *alc.py*, *véase [5]*.

- `openal.audio`, que contiene clases de alto nivel y funciones auxiliares que usan OpenAL. Este módulo proporciona una funcionalidad similar a ALUT, pero se aparta de la nomenclatura original de OpenAL, así que lo dejaremos a un lado, por el momento.

4 Desarrollo

El código que se muestra en este apartado se ha creado a partir del ejemplo[7] de la documentación de PyAL [6]. El ejemplo funciona sin errores, pero no genera ningún sonido, así que, en este apartado, se tomará como base ese código, se complementará a partir de lo que se observa en otros ejemplos de PyAL y se ampliará para que haga lo que se espera de una aplicación de sonido: ¡que suene la música! O lo que sea, pero que suene.

Se van a realizar funciones que generen sonidos básicos, viendo de **replicar la funcionalidad de ALUT** para tareas de síntesis de señales básicas (que era uno de tres bloques de operaciones que lo componen, como se veía en la Figura 1c) y que son: generación de ruido blanco y generación de señales básicas (como las que muestran las funciones representadas gráficamente en la Figura 3). Estas señales son enviadas por la

[4] PythonGameLibraries está en la URL <https://wiki.python.org/moin/PythonGameLibraries>.

[5] Información obtenida de <https://pypi.org/project/PyOpenAL/>.

[6] Véase la página del registro de instalación de <https://pypi.org/project/python-openal/>.

[7] Véase en la URL <https://pythonhosted.org/PyAL/openal.html>.

tarjeta de audio a un altavoz, quien las convertirá en sonido audible. La función de ALUT que hace la generación de señales básicas es *alutCreateBufferWaveform* y está encargada de crear un *buffer* y asignarle sonido a partir de la generación del mismo, esto es de forma sintética. Para ello, devuelve una referencia a un *buffer* que contiene un sonido, de la forma de ondas especificada, a la frecuencia indicada (en Hercios), con la fase (en grados entre -180 y +180) y la duración en segundos.

En este apartado se va llevar a cabo una versión similar para el caso de la generación de una señal de sonido basada en ruido blanco y una señal senoidal. Para una introducción a estos términos de puede consultar [9].

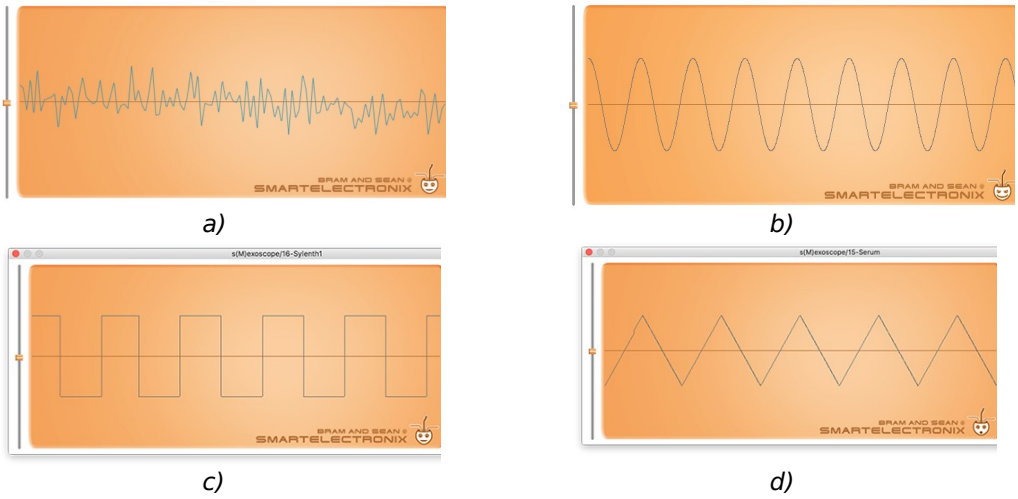


Figura 3: Tipos de estrategias básicas de generación sonidos representadas en forma gráfica: (a) ruido blanco, (b) onda senoidal, (c) onda cuadrada y (d) dientes de sierra. Imágenes de [9].

4.1 Ejemplo sonoro: ruido blanco

El ruido no es, estrictamente hablando puesto que no tiene un patrón de repetición, una forma de onda [9]. sino una opción para la generación de sonido que se utilizan en los instrumentos musicales electrónicos (como los sintetizadores o las cajas de ritmos). Está formado por valores aleatorios y existen diferentes tipos en función de las características de frecuencia que los caracteriza. El que se muestra en este apartado se llama ruido blanco (*whitenoise*[8]) por replicar la señal de ruido que genera ALUT.

Los Listado 1 y Listado 2 muestran el código realizado que implementa el comportamiento de la función de OpenAL *createBufferWaveform* para la generación de una "señal" tipo *Whitenoise*, se pueden encontrar al completo en el github [10] creado al respecto, y que se llama *whitenoise.py*. El Listado 1 es la primera parte de este ejemplo y contiene el código que genera una señal de ruido blanco en un *buffer* de OpenAL. Para ello:

[8] Puede descubrir más cosas al respecto del ruido como fuente de sonido y de los colores del sonido en <https://en.wikipedia.org/wiki/Colors_of_noise>. Y puede experimentar con ellas con el Web Audio API <https://webaudioapi.com/samples/oscillator/>.

- Calcula cuántos valores (*nElements*) son necesarios para generar la señal de las características indicadas como parámetros de la función y rellena un vector con valores aleatorios, líneas 8 a la 10, en el rango de [0 .. 256[(por lo que ocuparán un byte cada valor) y muestra en pantalla la longitud y el contenido del vector, líneas 11 y 12.
- Crea un *buffer*, vacío, guardado su identificador en la variable *bufferID*, líneas 14 y 15.
- Y lo rellena con los datos del vector creado y las propiedades del OpenAL que lo describen, líneas 17 a la 21.
  Para los entendidos en Python, observe que se ha utilizado la función *byte*[9], esto hace que el vector no sea editable. Podríamos haber utilizado *bytearray* si se quisiera poder modificarlo.

```
1.  from openal import al, alc # imports all relevant AL and ALC functions
2.  import time
3.  import random
1.
2.  def createBufferWaveform_whiteNoise( tamanyMostra, freqMostreig,
3.                                       nSegons ):
4.    nCanals = 1 # sempre monofonic per poder espacialitzar
5.    nElements = int(freqMostreig * nCanals ) * nSegons
6.
7.    vector = [None] * nElements
8.    random.seed()
9.    for i in range(0, nElements):
10.       vector[i] =  random.randrange(0, 256, 1)
11.   print( len(vector) )
12.   print( vector )
13.
14.   bufferID = al.ALuint(0)
15.   al.alGenBuffers(1, bufferID)
16.
17.   alformat =  al.AL_FORMAT_MONO8
18.   longitutEnBytes = int(nElements * tamanyMostra/8)
19.
20.   al.alBufferData(bufferID, alformat, bytes(vector),
21.                   longitutEnBytes, freqMostreig)
22.   return bufferID, alformat, longitutEnBytes, freqMostreig
```

Listado 1: Ejemplo de generación de una señal de ruido blanco con PyAL (whitenoise.py).

El Listado 2 es quien dirige la operación, se va a encargar de inicializar OpenAL, entre las líneas 24 a la 41, lo que implicar escoger el dispositivo hardware (en este caso la tarjeta de audio por defecto, aunque no es muy habitual que tengas varias, ¿verdad?) con *alOpenDevice*, (línea 25 a la 29); crea un contexto (*alcCreateContext*, líneas 31 a la 34) y lo establece como por defecto (con *alcMakeContextCurrent*, líneas 36 a 40). Este último paso

[9] Véase más al respecto dela función *byte* en https://docs.python.org/3/library/stdtypes.html#bytes .

es necesario en esta implementación de PyAl, aunque solo se ha creado un contexto y no hay duda de cuál debería escogerse en caso de tener que elegir.

```
    ...
23. def main():
24.     source = al.ALuint()
25.     device = alc.alcOpenDevice(None)      # alutInit
26.     if not device:
27.         error = alc.alcGetError()
28.         # do something with the error, which is a ctypes value
29.         return -1
30.     # Omit error checking
31.     context = alc.alcCreateContext(device, None)
32.     if not context:
33.         error = alc.alcGetError()
34.         return -2
35.
36.     resultat = alc.alcMakeContextCurrent(context)
37.     if not resultat:
38.          alc.alcDestroyContext(context)
39.          alc.alcCloseDevice(device)
40.          return -3
41.     #fi de alutInit
42.
43.     al.alGenSources(1, source)
44.     al.alSourcef(source, al.AL_PITCH, 1)
45.     al.alSourcef(source, al.AL_GAIN, 1)
46.     al.alSource3f(source, al.AL_POSITION, 0, 0, 0)
47.     al.alSource3f(source, al.AL_VELOCITY, 0, 0, 0)
48.     al.alSourcei(source, al.AL_LOOPING, 1)
49.
50.     buf, alformat, longitutEnBytes, freqMostreig =
51.                   createBufferWaveform_whiteNoise( 8, 11025, 10 )
52.     al.alSourceQueueBuffers(source, 1, buf)
53.
54.     al.alSourcePlay( source )    #source.play()
55.     time.sleep( 11 )            #al.alSleep( 2 )
56.
57.     al.alDeleteSources(1, source)
58.     alc.alcDestroyContext(context)       #alutExit
59.     alc.alcCloseDevice(device)           #fi de alutExit
60.     return 0
61.
62. if __name__ == "__main__":
63.     raise SystemExit(main())
```

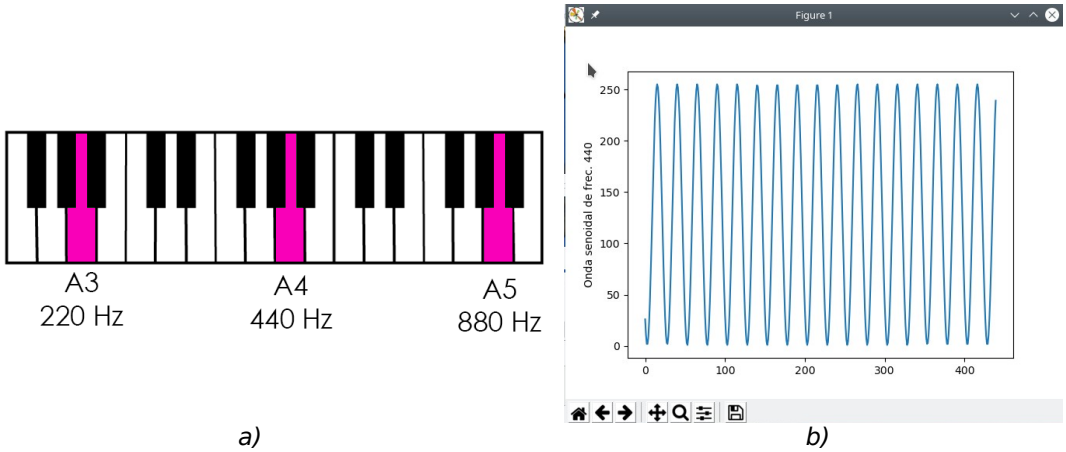
Listado 2: Ejemplo de generación de una señal de ruido blanco con PyAL (whitenoise,py, 2ª parte).

Entre las líneas 43 y la 55 está el bloque de hacer sonar lo que corresponda. Como se puede observar, se crea una fuente y se establecen sus propiedades (líneas 43 a la 48), entre las que destaca la creación de un *buffer* (líneas 50 y 51) con la función del Listado 1 y su asignación (línea 52) con *alSourceQueueBuffer*. Este último paso es necesario siempre en PyAL y previo a *alSourcePlay* (línea 54) que hace, finalmente, que algo suene. La línea 55 hace una espera de 11 segundos, ya que hemos generado una señal de duración 10

segundos, para que de tiempo a escucharla. El bloque final del Listado 2, líneas 57 a la 59, se encargan de liberar los recursos que se han asignado a la aplicación.

4.2 Ejemplo sonoro: tono de 440Hz

Como ejemplo de sonido más similar a una nota de un instrumento (un sintetizador, en la Figura 4a, como podría ser otro instrumento electrónico), en este apartado se generará una señal senoidal, Figura 1b. Es un sonido muy electrónico, puesto que los sonidos reales están compuestos de muchas señales simples: cada instrumento añade sus propias características debido a su geometría y materiales a la interpretación de una misma nota. Aquí se generará la nota LA de la escala central que corresponde, idealmente, con una senoidal de 440 Hz[10].



a) *b)*

Figura 4: Notas musicales y frecuenccia: (a) tres notas musicales (el LA en tres escalas) en un teclado (Imagen de [9]) y (b) la forma de onda que va a reproducir el código de ejemplo.

El Listado 3 muestra parte del ejemplo de código, puesto que es similar al del apartado anterior, así que solo se comentarán las diferencias. En el github [10] está al completo. ¿Qué cambia? Básicamente el código de generación de la señal, el código visualiza la señal (Figura 4b), aunque omitiremos la explicación por brevedad de la exposición

En este caso *createBufferWaveform_sine* se ha construido a modo de réplica de la correspondiente función de ALUT y, a partir de los parámetros que recibe (como la frecuencia, segundos que se indican de duración, etc.) se encarga de calcular el tamaño de un vector que almacena una señal de las características que se le indican.

La diferencia respecto a la generación del ruido blanco del apartado anterior estriba en que se utiliza la ecuación[11] de la línea 19 para generar los valores de los elementos del

[10] Más sobre la relación notas y hercios, en "Frecuencia de ls Notas – Frecuencias usicales (Hz)" <https://parabajoelectrico.com/frecuencia-notas-musicales/#La-frecuencia-del-sonido>.

[11] Adaptada del código fuente de ALUT en el Github de Freealut <https://github.com/vancegroup/freealut> y de python_wave_sample.py <https://gist.github.com/tomazas/156c616d4fe16327c6c7c5cbc27b04d0.

vector. Y, en el programa principal, obviamente, la invocará (líneas 29 y 30) y procederá a reproducirla como en el caso anterior.

```
1.  from openal import al, alc # imports all relevant AL and ALC functions
2.  import time
3.  import math
4.
5.  def createBufferWaveform_sine( frequencia, tamanyMostra, freqMostreig,
    nSegons, fase ):
6.
7.    periode = 1/frequencia
8.    delta = 1.0 / (freqMostreig)
9.
10.   bufferID = al.ALuint(0)
11.   al.alGenBuffers(1, bufferID)
12.
13.   nCanals = 1
14.   nElements = int(freqMostreig * nCanals ) * nSegons
15.   vector = [None] * nElements
16.   angul = 0
17.   t = 0
18.   for i in range(0, nElements):
19.       valor = 127* math.sin( fase + (2 * math.pi * frequencia * t))+128
20.       vector[i] =  int( round(valor) )
21.       t = t + delta
22.
23.   alformat =  al.AL_FORMAT_MONO8
24.   longitutEnBytes = int(nElements * tamanyMostra/8)
25.   al.alBufferData(bufferID, alformat, bytes(vector), longitutEnBytes,
                      freqMostreig)
26.   return bufferID, alformat, longitutEnBytes, freqMostreig
27.
28. def main():
    ...
29.     buf, alformat, longitut, freqMostreig =
                       createBufferWaveform_sine( 440, 8, 11025, 10, 180)
30.     al.alSourceQueueBuffers(source, 1, buf)
31.     al.alSourcePlay(source)
32.     time.sleep( 10 )
    ...
33.     return 0
34. if __name__ == "__main__":
35.     raise SystemExit(main())
```

Listado 3: Código de ejemplo de generación de un tono de 440Hz (ondaSenoidal.py).

5 Conclusión y cierre

Después de haber examinado el contenido de este artículo, el lector habrá podido constatar el uso de OpenAL que se puede hacer actualmente en Python. Habrá visto y, atendiendo a las instrucciones dadas, habrá podido probar cómo suena una aplicación

básica con OpenAL sobre Python que realiza síntesis de sonido a través de la generación de señales básicas.

Puede descargar los ejemplos del repositorio creado a tal efecto en *GitHub* [10] y no cierre este documento sin haberlo comprobado antes. ¡¡ÁNIMO, pruebe a realizar modificaciones en su propia versión y pruebe a implementar otras señales como las ondas cuadradas o triangulares y a ver cómo suenan!!

6 Bibliografía y referencias

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

[2] Panne, S. (2006). "The OpenAL Utility Toolkit (ALUT)". Disponible en <http://distro.ibiblio.org/rootlinux/rootlinux-ports/more/freealut/freealut-1.1.0/doc/alut.html >.

[3] Peacock, D.; Harrison, P.; D'Orta, A.; Carpentier, V. y Cooper, E . (2006).Effects Extension Guide. Disponible en <https://usermanual.wiki/Pdf/Effects20Extension20Guide.90272296/view >.

[4] Página web del proyecto PyOpenAL. Disponible en <https://pypi.org/project/PyOpenAL/>.

[5] Github del proyecto PyAL. Disponible en <https://github.com/JessicaTegner/PyAL>.

[6] Documentación relativa al proyecto PyAL. Disponible en <https://pythonhosted.org/PyAL/>.

[7] Agustí, M. (2011). Introducción al procesado de audio mediante OpenAL. <http://hdl.handle.net/10251/12694 >. Keywords: Procesado de audio, Openal.

[8] Agustí, M.. (2011). Introducción al empleo de técnicas de audio posicional mediante OpenAL. <http://hdl.handle.net/10251/12697 >. Keywords: Audio posicional, Audio envolvente, Audio espacial, Audio 3d, Procesado de audio, Openal.

[9] Stikeys. Comprendiendo las ondas de sonido y las formas de onda. Disponible en <https://stikeys.co/es/tutorials/understanding-sound-waves-and-waveforms/>.

[10] Agustí, M. (2023). OpenAL_examples_PyAL. Github. <https://github.com/magusti/OpenAL_examples/OpenAL_examples_PyAl>.