# FMOD



## FMOD: APIs + Editor de eventos

https://www.fmod.com/

Descargar (previo registro) FMOD Studio Suite: conjunto completo de herramientas para la creación e integración de sonido en videojuegos.

- FMOD Engine: dos APIs en C/C++ (también javascript y C# en Unity):
  - FMOD Core Programmer's API ⇝ posicionamiento 3D, mezcla, reverberación, streaming... (similar a OpenAL, pero más completa).
  - FMOD Studio API: carga y reproducción de bancos de sonido generados con FMOD Studio.
- FMOD Studio: herramienta *Middleware* para el diseño de sonido interactivo ⇝ DAW orientado a videojuegos (desde 2010 como FMOD Ex/Designer)
  - Editor de audio no lineal. Formato propio de bancos de sonido: colección estructurada de sonidos + parámetros de reproducción (eventos).

  Integración con Unity, Unreal, CryEngine, etc.

✓ En constante actualización, muy eficiente.
✓ Buena documentación (navegable, intuitiva). Muchos vídeos (anticuados pero útilesz)
✓ Uso libre en proyectos no comerciales (no es código abierto)





FMOD Core API



## FMOD Core API

- Independiente de FMOD Studio.
- Librería de renderizado de audio 3D: posicionamiento 3D, mezcla en tiempo real, etc. Similar a OpenAL, pero mucho más completa.
- Multiplataforma: Windows, Linux, Mac... Android, iOS, Game Cube, PS2, XBox, Nintendo...
- Soporta multitud de formatos de sonido (wav, ogg, mp3, midi, módulostracker,...)
- Efectos en tiempo real, extesionalidad (plugins),...
- ...funciones de análisis de espectro, drivers ASIO (Steinberg, baja latencia), ...

✓ Utilizado en más de 2000 juegos!! (https://www.fmod.com/games).
  - BioShock, Crysis, Diablo 3, Guitar Hero, Start Craft II, World of Warcraft.
  - Integrada en varios motores de videojuegos.

## Filosofía de FMOD

- FMOD ≈ mesa de mezclas con distintos canales.

  channel en FMOD ≈ fuente de sonora (*source* en OpenAL)
- En cada canal se puede cargar un sonido que se puede gestionar independientemente: reproducir, parar, pausar...
- Cada canal puede situarse en una posición en el espacio y se le pueden asignar efectos.
- Varios canales pueden agruparse en otro canal virtual para manipularlos conjuntamente.
- Repertorio muy completo de funciones para controlar los canales.
- Permite asignar prioridades a los canales, virtualizar, ...



## Preparando el proyecto (Windows)

Librería estática Core API:

- Enlazar la librería

  `api/core/lib/fmod[L][64]_vc.lib`
  - `L` es para habilitar los logs para depuración
  - `64` indica arquitectura de 64 bits (si no se pone nada es 32).

Librería dinámica (dll) asociada (se necesita accesible en tiempo de ejecución)

`fmod[L][64].dll`



## Cabeceras

Accesibles en los directorios:

- `/api/core/inc` para el Core API
- `/api/studio/inc` para el API FMOD Studio (lo usaremos más adelante)

```
#include "fmod.hpp"       // para utilizar el wrapper C++
#include "fmod_errors.h"  // para manejo de errores
...
```



## Estructura del Core API

Clases básicas:

- System:
  - inicialización de la librería
  - selección de hardware y drivers
  - carga y reproducción de sonidos
  - control del listener
  - consumo de CPU, ...
- Sound (unifica el tratamiento los distintos formatos de sonido):
  - control de buffers de sonido (se utiliza explícitamente para tareas específicas, menos frecuentes).
  - Puntos de sincronización de pistas (manejo de loops en tiempo real)
- Channel (hereda de ChannelControl). Control de parámetros de los canales:
  - volumen, panorama, etc
  - atributos para sonido 2D-3D, doppler, pitch
  - control de reproducción (play, pause, stop)

Más clases:

- ChannelGroup (hererda de ChannelControl): permite agrupar canales y obtener/modificar parámetros para todos ellos simultáneamente
- DSP: diseño y control de filtros, mezclas, reverberaciones... Posibilidades de expansión para el usuario.
- Geometry: simulación de recintos acústicos con obstrucción, oclusión, ...
- Reverb3D: manejo de reverbs posicionales



# Inicialización

Para facilitar la gestión de errores:

```
using namespace FMOD;

// para salidas de error
void ERRCHECK(FMOD_RESULT result){
  if (result != FMOD_OK){
    std::cout << FMOD_ErrorString(result) << std::endl;
    // printf("FMOD error %d - %s", result, FMOD_ErrorString(result));
    exit(-1); }
}
```

Inicialización básica:

```
int main() {
  System *system;
  FMOD_RESULT result;
  result = System_Create(&system);      // Creamos el objeto system
  ERRCHECK(result);

  // 128 canales (numero maximo que podremos utilizar simultaneamente)
  result = system->init(128, FMOD_INIT_NORMAL, 0);  // Inicializacion de FMOD
  ERRCHECK(result);
```



# Dispositivos y cierre del sistema

- Se puede seleccionar el dispositivo/driver de salida (`System::getNumDrivers, System::getDriver, System::setDriver`)
- puede además detectarlos automáticamente y cambiarlos dinámicamente (por ejemplo, si se conectan una tarjeta USB durante la ejecución).

Para cerrar el sistema:

```
result=system->release();
ERRCHECK(result);
```



# Carga de un sonido

```
Sound *sound1;
result = system->createSound(
    "Battle.wav",     // path al archivo de sonido
    FMOD_DEFAULT,     // valores (por defecto en este caso: sin loop, 2D)
    0,                // informacion adicional (nada en este caso)
    &sound1);         // handle al buffer de sonido
```

Asignación a canal y reproducción del sonido:

```
Channel *channel;
result = system->playSound(
    sound,    // buffer que se "engancha" a ese canal
    0         // grupo de canales, 0 sin agrupar (agrupado en el master)
    false,    // arranca sin "pause" (se reproduce directamente)
    &channel); // devuelve el canal que asigna
// el sonido ya se esta reproduciendo
```

La variable `channel` permite modificar los parámetros de ese canal:

```
result = channel->setVolume(0.7f);
```

Para hacerlo efectivo,EN CADA VUELTA DEL BUCLE PPAL.:

```
result = system->update();
```

(simplePlayer,battle)

# Carga de sonidos

Hemos cargado el sonido con:

```
system->createSound(..., FMOD_DEFAULT,...)
```

Esto significa que lo cargamos con la primera de las siguientes alternativas:

- `FMOD_LOOP_OFF`, `FMOD_LOOP_NORMAL`, `FMOD_LOOP_BIDI`: sin o con loop.
- `FMOD_2D` y `FMOD_3D`: posicionamiento 2D o 3D
- `FMOD_3D_INVERSEROLLOFF`, `FMOD_3D_LINEARROLLOFF`, `FMOD_3D_CUSTOMROLLOF`, etc
- etc

`FMOD_DEFAULT`: sonido 2D, sin loop
Pueden combinarse distintos modos (del tipo enumerado correspondiente) con el operador "`|`"

- FMOD soporta directamente 20 formatos de sonido (wav, aiff, flac, midi, mod, mpe, ogg, xm, it, etc). Y además, formato FSB, formato propio de FMOD.
- Con independencia del formato, los sonidos se cargan con `createSound` (FMOD se encarga de distinguirlos y cargarlos apropiadamente).



# Reproducción de sonidos

Para reproducir hemos hecho:

```
Channel *channel;
result = system->playSound(
  sound,   // buffer que se "engancha" a ese canal
  0        // grupo de canales, 0 sin agrupar (agrupado en el master)
  false,   // arranca sin "pause" (se reproduce directamente)
  &channel); // devuelve el canal que asigna
```

- "0" indica el grupo de canales (en este caso sin agrupar) FMOD permite el agrupamiento de canales (`ChannelGroup`) en un canal virtual: permite modificar parámetros de todo el grupo de manera conjunta.
- Arrancar el sonido con *pause* lo deja disponible en memoria para:
  - modificar parámetros antes de la reproducción.
  - evitar latencias (por ejemplo, para sincronizar pistas).
- FMOD obtiene un canal libre (se podría indicar un canal concreto, pero no es habitual): devuelve un nuevo canal cada vez que se carga un sonido.





- La reproducción termina:
  - con `channel->Stop()`, o
  - cuando eln sonido termina de reproducirse (y no está en loop)

  En ambos casos: se libera el canal asociado.
- Puede pausarse la reproducción (sin liberar el canal) con `channel->setPaused(true)` y reiniciarse con `channel->setPaused(false)`
- También hay un método `channel->getPaused(&paused)`
  - Podemos hacer un método

```
void TogglePaused(FMOD::Channel* channel) {
  bool paused;
  channel->getPaused(&paused);
  channel->setPaused(!paused);
}
```

- Hay que liberar el buffer de sonido después de utilizarlo:

```
result = sound->release(); ERRCHECK(result);
```



# Afinando la reproducción

Es posible comenzar la reproducción de un canal en un instante de tiempo (o sample) dado:

```
channel->setPosition(500, FMOD_TIMEUNIT_MS);
```

Comienza medio segundo después del instante inicial (también pueden utilzarse samples PCM, bytes, etc).

También podemos definir de antemano el número de repeticiones de un sonido en loop:

```
// se repite indefinidamente
channel->setLoopLoopCount(-1)

// Se repite una sola vez
channel->setLoopCount(0);

// Tres veces
channel->setLoopCount(2);
```

# Formatos en los buffers

Los sonidos pueden cargarse de tres modos en los buffers:

- Samples: se cargan las muestras (PCM) en memoria tal cual. Útil para los efectos de sonido que utilizan poca memoria.
- Streams: muestras grandes que ocuparían mucha memoria. Se cargan un buffer circular (como hacíamos en OpenAL). Los streams se reproducen de manera secuencial: no se puede hacer reproducción múltiple (simultáneo) del sonido de un stream. Útiles para música, pistas de voz, sonidos de ambiente, etc.
- Samples comprimidos (mp3, vorbis, etc): se dejan comprimidos en memoria. No tienen las limitaciones de los streams, ocupan menos en memoria, pero hay que descomprimir en tiempo real!

Por defecto, `System::createSound` los carga como samples (sin compresión).

- Pueden cargarse como streams con `System::createStream` (o con `System::createSound` y el valor `FMOD_CREATESTREAM`). Se puede hacer streaming incluso desde sonidos de internet.
- Pueden cargarse como samples comprimidos con `System::createSound` y el valor `FMOD_CREATECOMPRESSEDSAMPLE`



# Modificando parámetros en tiempo real (renderizado)

- Volumen: `channel->setVolume(val);`
  `val` (float) en el intervalo [0,1] (silencio, volumen normal).
- Silencio: `channel->setMute(true);` y `channel->setMute(false);`
  Silencia el canal. Cuando se reactiva conserva el volumen que tuviese previamente.
- Modificación del volumen en sonidos multicanal. Por ejemplo, en para un sonido estéreo:

```
channel->setMixLevelsOutput(frontleft,frontright,center,...,backleft,backright);
```

- Modificación del pitch: `channel->setPicth(2.0f);`
  También existe `channel->setFrequency(rateHz);` (frecuencia de reproducción en Hz.)



# Un pequeño reproductor

```
printf("[P] Pausar/Despausar\n[V/v] Subir/bajar volumen\n[Q] Salir\n");

bool paused;   float volume=1.0;
while (true) {
  if (kbhit()) {
    int key = getch();
    if ((key == 'P') || (key=='p')) {
      result=channel->getPaused(&paused);  ERRCHECK(result);
      result=channel->setPaused(!paused);  ERRCHECK(result);}
    else if (key=='V') {
      if (volume<1.0) {
        volume=volume+0.1;
        result = channel->setVolume(volume);  ERRCHECK(result);
        printf("Volume: %f\n",volume); }
    else if (key=='v'){
      if (volume>0) {
        volume=volume-0.1;
        result = channel->setVolume(volume);  ERRCHECK(result);
        printf("Volume: %f\n",volume); } }
    else if ((key=='Q') || (key=='q')) break;
  }
  result = system->update();
}
```

(simplePlayer2)



# Panorama en estéreo y en mono

(Para sonidos 3D se utiliza posicionamiento 3D, como veremos)

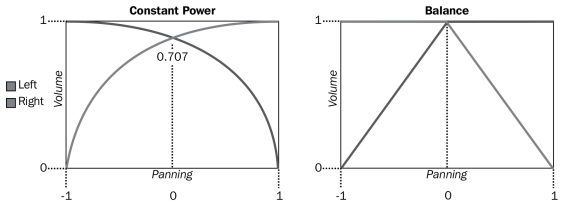
Para sonidos en 2D, tanto en mono como en estéreo manejo de panorama con:

```
channel->setPan(-1.0f);
```

-1.0 a la izquierda; 0 centrado (por defecto); 1 a la derecha

Pero no se comporta igual en mono y estéreo:

- En mono no se hace lineal, sino según un modelo constant power:
  pan = 0.0 significa 71 % left; 71 % right.
  ⇝ suavizar la transición de uno a otro canal según la intensidad *percibida*
- En estéreo pan = 0.0: 100 % left; 100 % right (simplePlayer3)




Getting Started with C++ Audio Programming for Game Development, D. Gouveia

## Agrupando canales

FMOD permite crear grupos de canales, añadir canales a los mismos y crear jerarquías:

```
// creamos un grupo de canales ``channelGroup''
FMOD::ChannelGroup* channelGroup;
system->createChannelGroup("grupo1", &channelGroup);

// aniadimos un canal existente, channel, al grupo
channel->setChannelGroup(channelGroup);
// se puede anidir un canal a un grupo directamente con
// playSound(...,group,...,...)   mas eficiente!


// aniadimos este grupo como hijo de otro grupo ``anotherGroup''
channelGroup->addGroup(anotherGroup);

// hay un ``master'' (raiz del arbol de grupos) que se puede acceder asi:
ChannelGroup* masterGroup;
system->getMasterChannelGroup(&masterGroup);
```

Los grupos, en un videojuego permiten agrupar los canales para cada categoría de sonidos: música, diálos, efectos sonoros, etc.
(ejemplos core channelGroup)



## Manejando grupos de canales

En general, un grupo, admite operaciones similares a las de un canal aislado

```
// Parar todos los canales del grupo
channelGroup->stop();

// Silenciar, pausar
channelGroup->setMute(true);
channelGroup->setPaused(true);

// ajustar volumen
channelGroup->setVolume(0.5f);

// duplicar pitch
channelGroup->setPitch(2.0f);
```

Todos estos cambios se propagan en la jerarquía hacia abajo (en el árbol de grupos y canales), sin sobreescribir los valores de cada canal individual.
La aplicación de parámetros sigue distintas políticas:

- Mute, pause: el valor (negativo) del grupo prevalece sobre el de los hijos. Si el padre está en pausa, los hijos también; pero si el canal no está pausado, se considera el valor de los hijos.
- Volumen, pitch: los valores del grupo se multiplican por los de los hijos. Por ejemplo, un canal con volumen 0.8 en un grupo con volumen 0.5 sonará a volumen 0.4.





# FMOD: sonido 3D



## Sonido 3D. Parámetros globales

La percepción del sonido en 3D depende de múltiples factores: posición, velocidad, direccionalidad, reverberación, obstáculos, etc.
En FMOD el sonido 3D se maneja esencialmente a través de los canales (manipulando parámetros sobre los canales). Pero hay algunos parámetros del sistema (no del canal). En particular:

- rolloff: atenuación con la distancia
- doppler: variación de frecuencia por la velocidad de aproximación entre listener y source

Ambos toman valores en el intervalo [0,1]:

- Por defecto 1.0; 0.0 anula el efecto; 1.0 es el valor “natural” físicamente
- Valores por encima de 1 son posibles: exageración del fenómeno físico (hiper-realidad)

```
float doppler = 1.0f, rolloff = 1.0f;
system->set3DSettings(doppler, 1.0, rolloff);
```

- El parámetro central 1.0 es un factor de escalado de la distancia: útil para modificar globalmente (y en proporción) las dimensiones del escenario de cara al motor de sonido.
- FMOD trabaja con unidades del S.I.: metros, segundos, etc.

## Colocando el listener

FMOD utiliza el sistema de la mano izquierda (modificable):

+X derecha  +Y arriba  +Z adelante

`FMOD_VECTOR` es un struct predefinido con los campos X, Y, Z

```
FMOD_VECTOR
  listenerPos = {0,0,0}, // posicion del listener
  listenerVel = {0,0,0}, // velocidad del listener
  up = {0,1,0}, // vector up: hacia la ``coronilla''
  at = {1,0,0}; // vector at: hacia donde mira

// colocamos listener
system->set3DListenerAttributes(0, &listenerPos, &listenerVel, &up, &at);
```

El primer parámetro indica el listener (FMOD puede trabajar con varios p.e. en juegos con pantalla partida). Se pueden crear con `set3DNumListeners`.

Por qué pasan los vectores por referencia? ⇝ pasan por referencia constante por eficiencia.



## Sonido 3D. Carga de sonidos

Para que FMOD pueda operar en 3D, la carga de sonido ya debe indicarlo:

```
Sound *sound;
ERRCHECK(
  system-createSound(
     "siren08.wav",              // arhivo de sonido
     FMOD_3D | FMOD_LOOP_NORMAL, // para REPRODUCIR en 3D y en loop
     0,
     &sound1));
```

~~IMPORTANTE: para posicionar sonidos, estos deben ser monofónicos~~ (no puede posicionar sonidos esté

**Los sonidos estéreo (y multicanal) pueden posicionarse en 3D!**
El sonido estéreo se separa en 2 voces mono, que se posicionan separadamente en 3D (en general, con sonidos multicanal se hace lo mismo)
Con `Channel::set3DSpread(grados)` puede controlarse la *amplitud del estéreo*:

- 0º: suena en mono, situado en 3D
- 90º: canal izquierdo 45º a la izquierda, canal derecho 45º a la derecha
- 360º: sonido mono situado en la posición opuesta.



## Creación del canal

Igual que antes, creamos sonido:

```
Channel *canalSirena;
ERRCHECK(
  system->playSound(
    sound,          // sonido
    0               // grupo (master)
    true,           // arrancamos en pause
    &canalSirena)); // handle al canal
```

Creamos canal y arrancamos en pause:

```
// situamos canal en 3D
FMOD_VECTOR
  pos={19,0,0}, // posicion
  vel={0,0,0};  // velocidad

canalSirena->set3DAttributes(&pos, &vel);
// por que pasan por referencia? ...
```

Lo habitual en un videojuego es que estos parámetros se ajusten en cada frame.
Ajustamos volumen del canal (igual que antes):

```
float volumenSirena=0.8f;
canalSirena->setVolume(volumenSirena);
```



## Integración del sonido en el juego

En general,

- Se asocia el listener con la cámara del motor gráfico,
- Se asocia un canal (o grupo de canales) a cada entidad del juego,
- Todos los cambios de de la cámara o las entidades
  - posición, velocidad, orientación

  deben realizarse también en los objetos sonoros

El loop principal del juego tiene que incluir la llamada

system->update()

para que los cambios sonoros surtan efecto

(ejemplos/mezclaPosicionamiento/pasos.py  openAL/sirena)

# Velocidad de las entidades (canales)

La velocidad de una entidad puede venir dada por el *estado* del juego, o bien, puede tener qu calcularse como diferencia entre la posición actual y la que tenía en frame anterior

. . . pero con un ajuste temporal: dividir entre el tiempo (*elapsed*) transcurrido en segundos (todas las unidades están en SI):

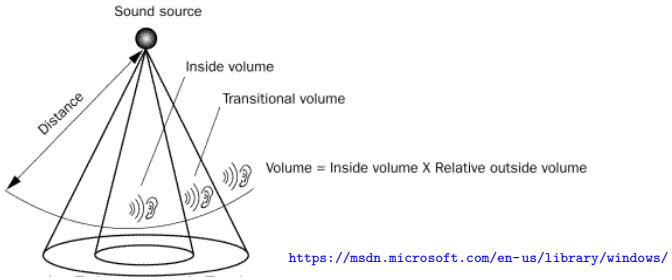
```
FMOD_VECTOR vel;

vel.x = (pos.x - lastPos.x) / elapsed;
vel.y = (pos.y - lastPos.y) / elapsed;
vel.z = (pos.z - lastPos.z) / elapsed;
```

Nota: en el libro de David Gouveia en vez de dividir, multiplica por `elapsed` (?)



# Direccionalidad: conos

Recordemos que la sireccionalidad (y orientación) de las entidades, se simula mediante conos:




2 conos concéntricos con eje en la dirección del sonido:

- Interior del cono interior: el sonido no sufre atenuación.
- Exterior del cono exterior: el sonido no es audible
- Zona intermedia entre ambos conos: la atenuación calculada por interpolación.



# Direccionalidad de los sonidos

- Por defecto todos los sonidos son omnidireccionales (se emite por igual en todas las direcciones del espacio).
- Pero podemos dar dirección a un sonido a través de conos de proyección (asociados al canal).

```
FMOD_VECTOR dir = { 1.0f, 2.0f, 3.0f }; // vector de direccion de los conos

channel->set3DConeOrientation(&dir);
channel->set3DConeSettings(30.0f, 60.0f, 0.5f); // angulos en grados
```

Los ángulos corresponden a:

- `insideconeangle`: ángulo del cono interior, donde no hay atenuación por la dirección (por defecto 360º)
- `outsideconeangle`: ángulo del cono exterior, donde el sonido se atenúa. La atenuación se calcula por interpolación (por defecto 360º)
- `outsidevolume`: volumen fuera del cono exterior (por defecto 1.0)



# Rangos de audición del canal

Podemos controlar el rango donde actúa la atenuación (rolloff). Como veíamos es una forma de "desnormalizar" el volumen de los sonidos (ejemplo reactor y mosquito).

```
float minDistance = 1.0f, maxDistance = 10000.0f;
channel->set3DMinMaxDistance(minDistance, maxDistance);
```

- `minDistance`: distancia a partir de la cual el sonido comienza a atenuarse
- `maxDistance`: distancia a partir de la cual el sonido no se atenúa más (el volumen no es necesariamente 0.0)

Entre esas dos distancias el volumen se atenúa según un modelo de atenuación. Por defecto `FMOD_3D_INVERSEROLLOFF`: poco coste computacional. El modelo logarítmico es más realista pero más costoso.

Por ejemplo, para `minDistance` podemos dar el valor 0.1 para un mosquito y 50 para el sonido de una explosión. En general, debemos dar valores suficientemente grandes para `maxDistance` para que el modelo de atenuación tenga *margen de operación*.

(`fmod/misEjemplos/planoSirena`)

# Simulación de recintos acústicos



# Recinto acústico: reverb (posicional)

*La reverberación es un fenómeno acústico que permite que el sonido persista en el tiempo cuando la fuente sonora ha dejado de producirlo.*

- De manera simplificada, puede pensarse como una sucesión de ecos con muy poco retardo entre ellos, producidos por el rebote en las distintas superficies del recinto.
- Depende fundamentalmente del recinto donde se produce y escucha el sonido.

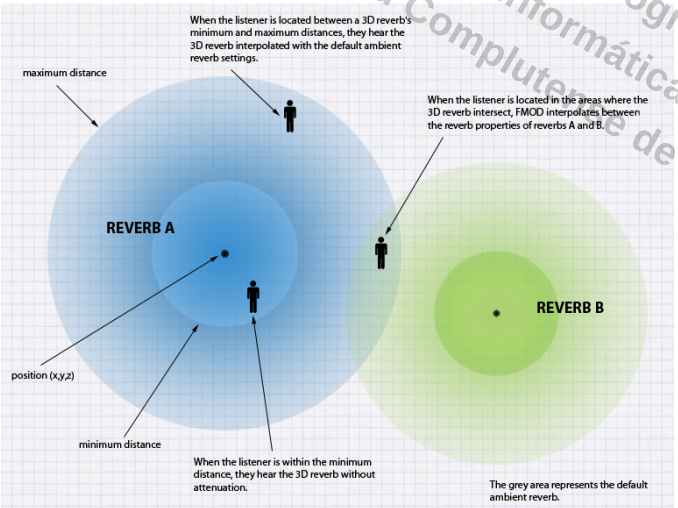
La simulación de la reverberación en un juego puede mejorar notablemente el realismo en las escenas, proporcionando mucha información sobre los recintos (tamaño, forma, materiales, etc).



# Reverb en FMOD

FMOD permite definir múltiples reverbs (virtuales), cada una con:

- Posición en 3D
- Zonas de *influencia*: 2 esferas concéntricas de distancias mínima y máxima.
- Propiedades de la reverb






El listener escuchará una mezcla ponderada de las reverb que actúan sobre la posición donde está.

- Es posible simular nuevas reverb mezclando presets (con las mismas esferas de acción)
- Los sonidos 2D comparten la misma instancia de reverb.

Creando una reverb y situándola en 3D:

```
Reverb *reverb;
result = system->createReverb3D(&reverb);
FMOD_REVERB_PROPERTIES prop2 = FMOD_PRESET_CONCERTHALL;
reverb->setProperties(&prop2);

FMOD_VECTOR pos = { -10.0f, 0.0f, 0.0f };
float mindist = 10.0f, maxdist = 20.0f;
reverb->set3DAttributes(&pos, mindist, maxdist);
```

- Múltiples presets:
  `FMOD_PRESET_ROOM` `FMOD_PRESET_BATHROOM` `FMOD_PRESET_CONCERTHALL`
  `FMOD_PRESET_CAVE` `FMOD_PRESET_UNDERWATER` ...
- Las reverb pueden activarse o desactivarse con `reverb->setActive(false)`
- La mezcla *dry/wet* se hace por canal con `Channel::setReverbProperties`
- Para liberar el objeto `reverb->release()`

## Recinto acústico: obstrucción y oclusión

Recordemos:

- Obstrucción: emisor y oyente en el mismo recinto, con un obstáculo entre ellos. Las ondas sonoras que pasan a través del obstáculo están atenuadas y filtradas; las reflejadas no están afectadas por el obstáculo.
- Oclusión: emisor y oyente en distinto recinto. Todas las ondas sonoras tienen que atravesar un obstáculo (y están atenuadas y filtradas).

En FMOD, clase `Geometry`: permite crear una malla de polígonos (ej. triángulos) para representar obstáculos.

```
FMOD::Geometry* geometry;
system->createGeometry(maxPoligons, maxVertices, &geometry);
```

⇝ crea una una estructura optimizada para gestionar los polígonos: minimizar solapamiento!



## Creación de polígonos y ubicación del objeto

```
int polygonIndex; // Indice al poligono para referenciarlo despues

geometry->addPolygon(
  float directocclusion,        // 0.0 no atenua, 1.0 atenua totalmente
  float reverbOcclusion,        // atenuacion de la reverberacion
  bool doubleSided,             // atenua por ambos lados o no
  int numVertices,              // numero de vertices (>=3)
  const FMOD_VECTOR *vertices,  // vector de numVertices vertices
  int *polygonindex);           // indice del poligono generado
```

- Todos los vértices deben estar en el mismo plano.
- Tienen que ir ordenados en sentido antihorario
- Los polígonos deben ser convexos y con área positiva.

Posición de los vértices relativa a la posición (o centro) del objeto, utilizada en `setPosition`:

- `setPosition(FMOD_VECTOR pos)`: sitúa el objeto geometry en `pos`, relativa al espacio del listener y los canales.
- `setRotation(FMOD_VECTOR *forward, FMOD_VECTOR *up)`: orientación del objeto
- `setScale(FMOD_VECTOR scale)`: escala relativa de `geometry` en las 3 dimensiones por separado

(`fmod/python/geometry/geoRev.py`)



## Efectos

FMOD incorpora una amplia paleta de efectos (DSP):

- Normalización, compresión, distorsión, filtros, equalizadores, chorus, echo, tremolo, delay, etc.

Creación de un DSP de FMOD nativo:

```
FMOD::DSP* dsp;
system->createDSPByType(FMOD_DSP_TYPE_ECHO, &dsp);

// apliacion a un canal (puede aplicarse a un grupo o al sistema)
channel->addDSP(dsp, 0);

// parametros del efecto
dsp->setParameter(FMOD_DSP_ECHO_DECAYRATIO, 0.75f);
```

(core effects)

(ver tutorial FMOD *DSP Architecture and Usage*)



## Reverb por convolución!!

Se utiliza un archivo wav con el patrón de la reverb (IR, Impulse response), para modelar las reflexiones del sonido hacia el listener

```
// creamos unidad DSP para la reverb por convolucion
FMOD::DSP* reverbUnit;
result = system->createDSPByType(FMOD_DSP_TYPE_CONVOLUTIONREVERB, &reverbUnit);
result = reverbGroup->addDSP(FMOD_CHANNELCONTROL_DSP_TAIL, reverbUnit);

// cargamos archivo wav con el modelo de ``impulse response''
FMOD::Sound* irSound;
result = system->createSound(Common_MediaPath("standrews.wav"),
                             FMOD_DEFAULT | FMOD_OPENONLY, NULL, &irSound);
```

... aplicamos. Ver detalles en ejemplo (convolution_reverb)

Modelo IR basado en “St Andrew’s Church”, Audiolab, University of York

http://www.openairlib.net/auralizationdb/content/st-andrews-church

## Combinando DSPs

Sonido de radio: distorsión + filtro paso alto

```
FMOD::DSP* distortion;
system->createDSPByType(FMOD_DSP_TYPE_DISTORTION, &distortion);
distortion->setParameter(FMOD_DSP_DISTORTION_LEVEL, 0.85f);

FMOD::DSP* highpass;
system->createDSPByType(FMOD_DSP_TYPE_HIGHPASS, &highpass);
highpass->setParameter(FMOD_DSP_HIGHPASS_CUTOFF, 2000.0f);

channel->addDSP(distortion, 0);
channel->addDSP(highpass, 0);
```



## Captura de sonido

FMOD también permite capturar sonido (de micrófono)
Ejemplos (core record, record_enumeration)
Otro ejemplo, con combinación de DSPs walkie