Game Audio via OpenAL: Extras

Summary

In the previous Sound tutorial, you learnt the basics of OpenAL, and implemented a useable audio system that extended your basic game engine with an emitter class that could play sound. This time around, you'll be introduced to the concept of streaming sounds, via implementing the ability to play the popular *ogg vorbis* audio format. You'll also see how to extend the sound system to support 'triggered' sounds - handy for GUI output and 'one time only' sounds caused by in-game events.

New Concepts

Triggering sounds, sounds without 3D positioning, streaming sounds, multiple sound buffers, Ogg Vorbis files

Introduction

Our *SoundSystem* is already pretty good, and is capable of playing many different sounds in a reliable way. We can still make it better, though, so to round off this tutorial series, we're going to look at triggering global audio, and streaming audio files too large to fit in memory.

Triggering Audio & Global Audio

As our *SoundSystem* class currently stands, it can play audio samples in 3D space via OpenAL and a special *SoundEmitter* class. That's an incredibly powerful tool that will allow a wide variety of soundscapes to be created, but there's a slight hole in our audio playback capabilities. What about the audio feedback from a graphical user interface? It's common for the buttons in games to make a noise when highlighted or clicked, and some games have video clips with audio as cutscenes between levels. These are examples of *global* audio - sound that we don't want to be processed in 3D space, but to instead just be sent to the speakers as-is. With the way our sound system is currently implemented, we'd need to create a new GameEntity for each of these, and we'd have to keep track of them and delete them as necessary in our gameplay logic. We also have another problem - where do we place them in space? How audio is played back is determined by the sound emitter's position in relation to the listener, but sounds caused by GUI elements we really want to be 'global' sounds, always audible, no matter where the player is in the world.

The ideal solution to how to deal with these triggered types of one-off sound events is to extend the *SoundSystem* to support basic playback of samples without having to send it a new *SoundEmitter*. That might sound slightly backwards given how powerful the ability to control sounds with a *GameEntity* is, but sometimes in our games, we just want to trigger the playback of a sound, and not really care about it afterwards. By adding simple functions to the *SoundSystem* to automate the triggered playback of sounds, it becomes much easier to playback audio, whatever the situation. Having separate functions for one-off sounds in 3D space, and global sounds, allows you to hide the actual implementation of how the sound is played back or made global - differing sound APIs will handle global audio playback in different ways, and relying on external-API specific knowledge throughout your game's code is a **bad** idea - much better to just let the *SoundSystem* deal with it. In the tutorial code, you'll see how to support such triggered audio with the minimum of additional code, allowing the flexibility of 3D *GameEntity*-derived sound, and the simplicity of being able to trigger in-game sounds without having to keep track of lots of temporary entities in your gameplay code.

Compressed Audio

In the previous tutorial, we looked at how audio data is encoded via *Pulse Code Modulation*, and loaded in some uncompressed audio data from a WAV file. Of course, WAV is not the only audio file format out there, there are *compressed* formats, too - such as the ubiquitous **MP3** file format. These compressed sound formats may use the same compression techniques used in file archiving formats like **ZIP** and **RAR** at some point in their compression algorithm, such as run length encoding (shortening repeating bytes to a single byte and an occurrence multiplier) and Huffman coding (building up 'dictionaries' of frequently occurring byte sequences and assigning them short identifier sequences), but will also usually include compression techniques specifically developed to analyse and compress the waveform data of sound. The most common of these is the *discrete Fourier transform* - Fourier transforms turn complex waves into the sum of simpler waves defined by sines and cosines, while discrete transforms do so on discrete 'separate' data such as the digital PCM information. These Fourier transforms can become quite costly, but algorithms known as fast Fourier transforms exist that are much quicker, and it is these that are use when encoding and decoding compressed audio data. The audio hardware in your mobile phones and MP3 players most likely contains a hardware implementation of such a fast Fourier transform, allowing the efficient, real-time decoding of your favourite music tracks.

Ogg Vorbis

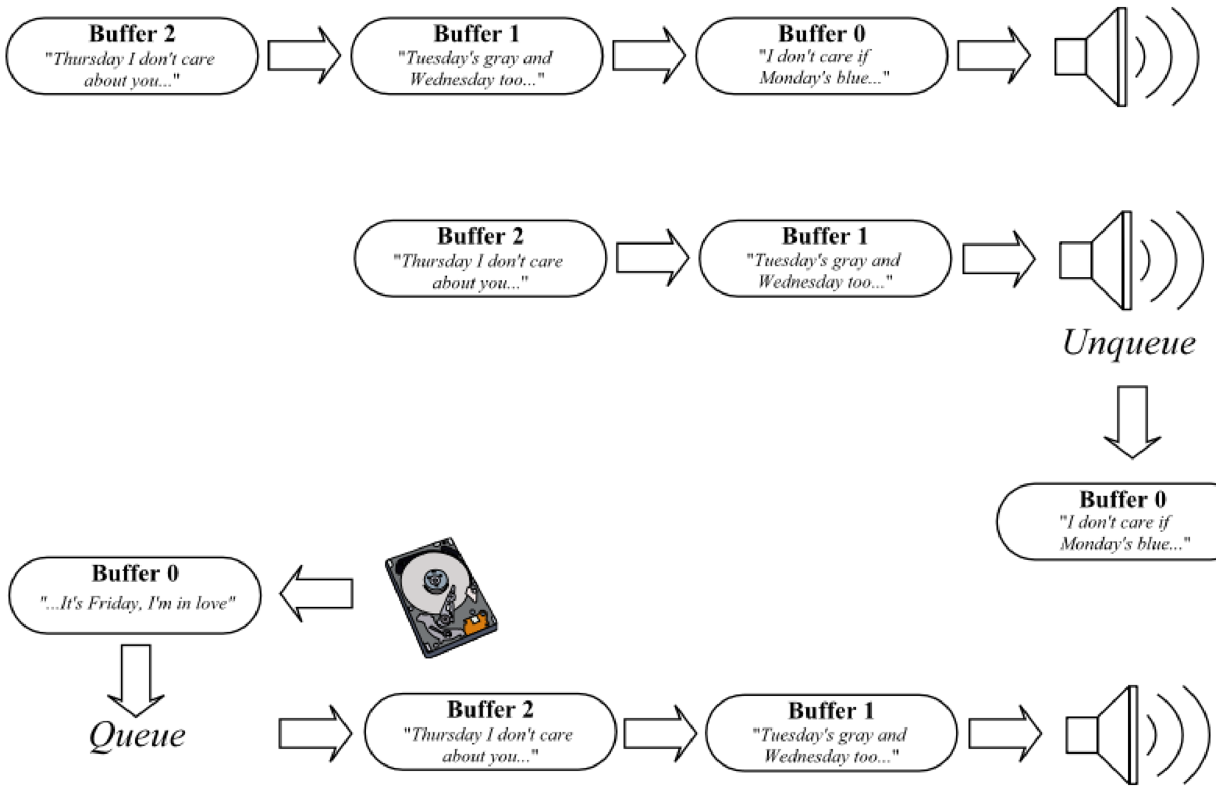
The process of decoding a compressed audio file is generally quite computationally expensive, and writing a loader for such a format, as you did for WAV files, would be too complicated, and likely to be error prone. Instead, external libraries are often employed to handle the actual extraction of PCM data from a compressed file format. The choice of compressed audio to support in an application therefore comes down to the availability of these libraries, and how liberal their license is - some will be free, while some will require some sort of payment, or even royalties on every copy sold.

The MP3 file format falls into this latter category (for commercial software selling over 100,000 units, anyway), so free audio compression libraries are often used instead. One of the most popular free alternatives to MP3 is the *Ogg Vorbis* file format, handled by the Xiph.Org Foundation. Strictly speaking, *Ogg* is only the container format, with *Vorbis* being the name of the specific compression technique used to fill the container, but due to such compressed files having the file extension **.ogg**, it is common for the entire scheme to be simply called *ogg*. Like MP3, ogg supports varying bitrates, as well as a varying number of channels of audio per file. This flexibility, along with the liberal license and efficiency of its decoder, make ogg ideal for compressing music and longer tracks down to a more manageable size.

Streaming Audio

These compressed file formats allow your games to come with 10s of hours worth of sound effects, vocal tracks, and music, all at a reasonably small space requirement. However, remember that at some point, the compressed data must be turned back into raw PCM data, for sending to the sound card. The PCM data for a single music track in CD quality will take up around 80MB - that's quite a lot, especially for consoles or mobile devices! It's therefore not feasible to store an entire track in memory at once; but what we can do, is *stream* the data in as we need it. When playing back audio, we will very nearly always be playing back the PCM data from its beginning to end, so once a particular section of PCM data has been played, it doesn't really need to be in memory at all! In fact, we only need enough data from our PCM data stream to play back a short section at a time - the length of this section is determined by how quickly the next section of data can be loaded from disk and decompressed back into PCM data. If the section loaded up is too short, it may have finished playing before the next section has been loaded and decoded, resulting in playback artefacts (sections of sound repeating, pops and clicks, pauses in playback); conversely, if the section is too long, it will be wasting memory.

The simplest way to handle the streaming of audio is to have a number of short buffers, instead of one big buffer for your streamed audio. These buffers can then be filled up with some PCM audio data, and sent to the sound hardware, one after the other, via a queue. Then, as each buffer is popped off the queue, and it's data 'consumed' by the sound hardware, it can be filled up with new data for the next section of the audio track, and popped back on the queue. Providing that the disk access and audio decoding does not take too long, this will result in seamless, streamed playback of an audio file of any size, without the large memory footprint.


Buffer 2
"Thursday I don't care about you..."
Buffer 1
"Tuesday's gray and Wednesday too..."
Buffer 0
"I don't care if Monday's blue..."
Unqueue
Buffer 0
"...It's Friday, I'm in love"
Queue

An example of how three buffers could be used to seamlessly stream an audio file from a file. Top: Three buffers are filled with PCM data from a song. Middle: As the contents of a buffer is sent to the audio device, it is popped off the queue. Bottom: Popped off buffers can then be filled with more PCM data and pushed back onto the queue.

Example Program

To round off our audio playback system, we're going to add in the ability to playback streaming audio, as well as triggering one-off sounds, such as the global sounds described earlier. To do so, we're going to have to modify our *SoundSystem*, *Sound*, and *SoundEmitter* classes, as well as creating a new class derived from *Sound*, *OggSound* - as its name implies, this will handle the decoding of audio data from an Ogg Vorbis file.

SoundEmitter Class

To enhance our audio playback capability, we're going to add the ability for a *SoundEmitter* to play a 'global' sound, and be able to play back audio from a streaming file type - we're going to add ogg capability later, but the *SoundEmitter* class changes will allow for any streaming file to be supported. Why do we need to change the *SoundEmitter* class to support streaming audio in our audio system? Well, to use streaming audio, we're going to use multiple buffers as described earlier, but we can't just add multiple buffers to the *Sound* class - multiple *SoundEmitter* might be streaming audio from the same file, so they each need their own set of buffers to stream data from the *Sound* in to.

So, we need a few new **public** accessors and a **protected** variable to handle global playback, and some **protected** variables to contain the stream position of a *SoundEmitter*, and the buffers it has streamed data in to. We're using the **define** *NUM_STREAM_BUFFERS* to determine the length of the buffer queue a *SoundEmitter* can support - 3 should be enough for a reasonably fast machine.

Header file

```
#define NUM_STREAM_BUFFERS 3  //Part 2

public:
   bool         GetIsGlobal()               {return isGlobal;}
   void         SetIsGlobal(bool value)     {isGlobal = value;}

protected:
   bool         isGlobal;
   double       streamPos;
   ALuint       streamBuffers[NUM_STREAM_BUFFERS];
```

SoundEmitter.h

Class file

Of course, as with other member variables, we should initialise our new variables to sensible defaults, including setting our *streamBuffers* array to all 0 - which as you'll remember, means 'no value' to OpenAL.

```
void  SoundEmitter::Reset() {
...//We're adding stuff to the existing constructor!!!
   streamPos      = 0;
   isGlobal       = false;
   for(unsigned int i = 0; i < NUM_STREAM_BUFFERS; ++i) {
      streamBuffers[i] = 0;
   }
...
}
```

SoundEmitter.cpp

When setting a new *Sound* on a *SoundEmitter*, we need to add in a little bit of logic to generate the OpenAL buffers to stream the PCM data in to. There's no point in always creating these buffers, as not every *SoundEmitter* will be playing streamed data, so instead we create them when a streaming *Sound* is passed to a *SoundEmitter*, and destroy them when a non-streaming *Sound* is passed.

```
void     SoundEmitter::SetSound(Sound *s) {
   sound = s;
   DetachSource();
   if(sound)   {
      timeLeft = sound->GetLength();
      if(sound->IsStreaming()) { //new bit!
         alGenBuffers(NUM_STREAM_BUFFERS, streamBuffers);
      }
      else{
         alDeleteBuffers(NUM_STREAM_BUFFERS, streamBuffers);
      }
   }
}
```

SoundEmitter.cpp

We must also modify the *AttachSource* function, as we need to queue up a number of buffers-worth of audio data every time we start playing a new streamed *Sound*. On line 37, we loop over the *streamBuffers* array of OpenAL buffers, buffering up streamed audio data from the current *Sound*, using its *StreamData* function. To account for any slight differences in the amount of data we're trying to stream, and the amount actually streamed (due to the end of a *Sound* not filling up a buffer entirely, for example), we keep track of the amount of audio data left to stream in the *streamPos* variable, starting from *timeLeft* (line 35). On line 49, we queue up the newly filled OpenAL buffers on the newly attached OpenAL source. Finally, if the *Sound* isn't streamed, we just seek to the correct place in the PCM data, as in the previous tutorial.

```
23 void      SoundEmitter::AttachSource(OALSource* s) {
24    oalSource = s;
25    if(!oalSource) {
26       return;
27    }
28    oalSource->inUse = true;
29
30    alSourceStop(oalSource->source);
31    alSourcef(oalSource->source, AL_MAX_DISTANCE, radius);
32    alSourcef(oalSource->source, AL_REFERENCE_DISTANCE, radius * 0.2f);
33
34    if(sound->IsStreaming()) {
35       streamPos = timeLeft;   //Seeks to correct position in stream
36       int numBuffered = 0; //How many buffers have we filled?
37       while(numBuffered < NUM_STREAM_BUFFERS) {
38          double streamed = //Stream in some more data
39             sound->StreamData(streamBuffers[numBuffered],streamPos);
40
41          if(streamed) {
42             streamPos -= streamed;
43             ++numBuffered; //Filled another buffer!
44          }
45          else{
46             break;   //Can't fill any more buffers...
47          }
48       }
49       alSourceQueueBuffers(oalSource->source, numBuffered,
50                            &streamBuffers[0]);  //Queue up buffers
51    }
52    else{
53       alSourcei(oalSource->source,AL_BUFFER,sound->GetBuffer());
54       alSourcef(oalSource->source,AL_SEC_OFFSET,
55             (sound->GetLength()/ 1000.0) - (timeLeft / 1000.0));
56    }
57    alSourcePlay(oalSource->source);
58 }
```

SoundEmitter.cpp

When we detach one of OpenAL's sources from a *SoundEmitter*, we need to make sure it's not going to keep playing any audio data - to do so, we unqueue any queued buffers on the source, undoing the effect of attaching a source to a *SoundEmitter*. We can query how many buffers the source has gotten through using the *AL_BUFFERS_PROCESED* named constant, and then using the *alSourceUnqueueBuffers* function to unqueue them, before disabling the sound by setting 0 as the current buffer.

```
59 void     SoundEmitter::DetachSource() {
60 ...//start of function up here
61 alSourceStop(oalSource->source);
62 if(sound && sound->IsStreaming()) { //Part 2
63    int numProcessed = 0;
64    ALuint tempBuffer;
65    alGetSourcei(oalSource->source,AL_BUFFERS_PROCESSED,&numProcessed);
66    while( numProcessed-- ) {
67       alSourceUnqueueBuffers( oalSource->source, 1, &tempBuffer );
68    }
69 }
70 alSourcei(oalSource->source,AL_BUFFER,0);
71 ...//rest of function down here!
72 }
```

SoundEmitter.cpp

Lastly, we must refactor the *UpdateSoundState* function. To handle global sounds with as little re-coding as possible, we're going to simply set the attached OpenAL source's position to be the same as the *SoundSystem's* listener - this will make it play at maximum volume, and also make sure the node always passed the distance check when culling nodes. If it's not playing a global sound, the source's position is updated as normal, to that of the *SoundEmitter*.

```
73 void     SoundEmitter::UpdateSoundState(float msec) {
74 ...//Start of function up here
75    Vector3 pos;
76
77    if(GetIsGlobal()) {
78       pos = SoundSystem::GetSoundSystem()->
79             GetListenerTransform().GetPositionVector();
80    }
81    else{
82       if(target) {
83          pos = target->GetWorldTransform().GetPositionVector();
84       }
85       else {
86          pos = this->position;
87       }
88    }
89    alSourcefv(oalSource->source,AL_POSITION,(float*)&pos);
```

SoundEmitter.cpp

With global playback sorted, we must also add in streaming audio updates. If the current *Sound* is streaming audio, we must see if we need to unqueue or enqueue any more audio buffers. We do so using the *AL_BUFFERS_PROCESED* named constant in conjunction with the *alGetSourcei* function again, which returns how many of its queued buffers have been played. On line 95 onwards, we loop over the number of processed buffers, streaming new data into them, and enqueueing them back to the source. The *alSourceUnqueueBuffers* function places the name of the processed buffer in the *freeBuffer* local variable, which we can then reuse in the *alSourceQueueBuffers* function call. As there's the possibility of reaching the end of a streamed file during this process, on line 103 we check for the streamed data running out of audio, and loop back round to the start of the file if the *SoundEmitter* is expected to loop its audio.

You may notice that in this refactored function, we only set the *AL_LOOPING* parameter to true if the *Sound* is **not** a streaming type - if an OpenAL source with no more buffers left to process has looping set, it will start playing its last buffer again! To prevent this, we don't use OpenAL's looping capability - the audio will still loop due to how we manually loop the *streamPos* and *timeLeft* variables.

```
if(sound->IsStreaming()) {
   int numProcessed;
   alGetSourcei(oalSource->source,AL_BUFFERS_PROCESSED,&numProcessed);
   alSourcei(   oalSource->source,AL_LOOPING ,0);

   while(numProcessed--) {
      ALuint freeBuffer;
      //Pop off the buffer
      alSourceUnqueueBuffers(oalSource->source, 1, &freeBuffer);

      streamPos -= sound->StreamData(freeBuffer,streamPos);
      alSourceQueueBuffers(oalSource->source, 1, &freeBuffer);

      if(streamPos < 0 && isLooping) {
         streamPos += sound->GetLength();
      }
   }
}
else{
   alSourcei(oalSource->source    ,AL_LOOPING ,isLooping ? 1 : 0);
}
...//end of function down here
```

SoundEmitter.cpp

Sound Class

Header file

To support our new sound streaming capability, we need to modify the *Sound* class. The function *IsStreaming*, coupled with a new member variable, *streaming*, determines whether an instance of a *Sound* is being read from memory or disk, while the **virtual** function *StreamData* will perform the data streaming. In this basic implementation, streaming of the basic *Sound* class doesn't do anything, but there's no reason why you couldn't stream data from a large WAV file.

```
public:
   bool            IsStreaming()  {return streaming;}
   virtual double StreamData(ALuint buffer, double timeLeft){
                        return 0.0f;}
protected:
   bool         streaming;
```

Sound.h

Class file

In the **constructor**, we need to initialise the *streaming* variable - most sounds will probably be a small amount of raw PCM data, and so will not be streamed, so **false** should be the default.

```
#include "OggSound.h"    //So that AddSound works!
Sound::Sound() {
   streaming    = false;
   //Rest of constructor goes here!
}
```

Sound.cpp

SoundManager Class

The *AddSound* function in our *SoundManager* needs changing, too. We need to add in support for the ogg file format, so just as we checked for the WAV extension in the previous tutorial, we now check for the OGG extension, to. If the sound to be added is in ogg format, we create a new *OggSound*, and add that to the **static** *sounds* map.

```
void  SoundManager::AddSound(string name) {
   Sound *s = GetSound(name);

   if(!s) {
      string extension = name.substr(name.length()-3,3);

      if(extension == "wav") {
         s = new Sound();
         s->LoadFromWAV(name);
         alGenBuffers(1,&s->buffer);
         alBufferData(s->buffer,s->GetOALFormat(),s->GetData(),
                      s->GetSize(),(ALsizei)s->GetFrequency());
      }
      else if(extension == "ogg") {
         OggSound* ogg = new OggSound();
         ogg->LoadFromOgg(name);

         s = ogg;
      }
      else{
         s = NULL;
         cout << "Incompatible file extension '" << extension
              << "'!" << endl;
      }

      sounds.insert(make_pair(name, s));
   }
}
```

Sound.cpp

OggSound Class

Header file

Now for the new stuff! The *OggSound* class deals with the streaming of data from the *ogg vorbis* file format. It does so via two external libraries - *libogg* to handle the ogg file structure, and *libvorbis* to handle the data within that structure. So, we need to add includes to the *OggSound* header file to enable access to these libraries. We also create a new define, *BUFFERLENGTH*, which will determine the size in bytes of the PCM data each of the OpenAL buffers used to stream the data will contain.

As for the actual class itself, it has two **public** functions: *LoadFromOgg*, which handles the initialising of an ogg file, and is called by the *Sound::AddSound* function, and an **overridden** *StreamData* function, which will fill an OpenAL buffer up with some PCM data decoded from the ogg stream. To keep track of the ogg stream, we need a couple of **protected** member variables, *fileHandle* and *streamHandle*. The *fileHandle* variable contains the information relating to the ogg file itself - the ogg vorbis libraries are written in pure C, and so do not support the more modern C++ filestream data types, so we must keep track of the file using the C equivalent, **FILE**. The *streamHandle* variable keeps track of the actual sound data being pulled out of the ogg file, and is a datatype defined by the ogg vorbis libraries.

```
 1 #pragma once
 2 #include "sound.h"
 3 #include <ogg/ogg.h>
 4 #include <vorbis/codec.h>
 5 #include <vorbis/vorbisenc.h>
 6 #include <vorbis/vorbisfile.h>
 7
 8 #define BUFFERLENGTH 32768
 9
10 class OggSound : public Sound {
11 public:
12    OggSound(void);
13    virtual ~OggSound(void);
14
15    void           LoadFromOgg(string filename);
16    virtual double StreamData(ALuint buffer, double timeLeft);
17
18 protected:
19    FILE*          fileHandle;
20    OggVorbis_File streamHandle;
21 };
```

OggSound.h

Class File

Our **constructor** is simple, it just initialises the *fileHandle* and *streaming* variables. You *could* decode an entire ogg file at once, but as with the *Sound* class, extending this functionality is left up to you!.

```
1 #include "OggSound.h"
2
3 OggSound::OggSound(void)   {
4    streaming   = true;
5    fileHandle  = NULL;
6 }
```

OggSound.cpp

The **destructor** is also pretty simple - it closes the ogg file being streamed from (using the C function *fclose*), and clears up the ogg stream information, using the library function *ov_clear*. All ogg vorbis functions are named using the *og_* prefix, making them quite easy to spot.

```
 7 OggSound::~OggSound(void)  {
 8    if(fileHandle) {
 9       ov_clear(&streamHandle);
10       fclose(fileHandle);
11    }
12 }
```

OggSound.cpp

When we open up an ogg file via the *SoundManager::AddSound* function, the following *LoadFromOgg* function is called. First off, on line 14, the C *FILE* is opened, using the C function *fopen*, which takes a **char pointer** as a filename, and a **char array** tag - we're using **rb**, as we're going to **r**ead **b**inary data. Assuming the file exists, we then fill the *streamHandle* **struct** variable, using the *ov_open* library function, which as you might expect, extracts the stream information from the newly opened *FILE*. From this stream information, we can extract a *vorbis_info* **struct**, which contains information relating to the channel count and frequency rate of the compressed vorbis data, and also the audio length, via the *ov_time_total* function which returns the number of seconds the audio lasts for. Finally, we seek to the start of the audio stream using the *ov_time_seek* function, so we can begin to stream audio out of it.

```
13 void  OggSound::LoadFromOgg(string filename) {
14    fileHandle = fopen(filename.c_str(), "rb");
15 
16    if(!fileHandle) {
17       cout << "Failed to load OGG file '" << filename << "'!" << endl;
18       return;
19    }
20    if(ov_open(fileHandle, &streamHandle, NULL, 0) < 0) {
21       cout << "Failed to get OGG stream handle!" << endl;
22       return;
23    }
24    vorbis_info* vorbisInfo = ov_info(&streamHandle, -1);
25 
26    bitRate     = 16; //For now!
27    channels    = vorbisInfo->channels;
28    freqRate    = (float)vorbisInfo->rate;
29    length      = (float)ov_time_total(&streamHandle,-1) * 1000.0f;
30 
31    ov_time_seek(&streamHandle, 0);
32 }
```

OggSound.cpp

The last function in the *OggSound* class is also the most important - *StreamData*. It's pretty complicated, to, and so will take a little bit of explaining. Whenever *StreamData* is called, it will attempt to stream in some amount of audio data into the OpenAL buffer parameter, starting from the point in the audio stream where there are a number of milliseconds left equal to the second parameter. So, on line 39, we again use the *ov_time_seek* function, to move the stream position to the correct place. If this seek process fails, denoted with a seek parameter value of less than 0, we 'loop around' the audio stream by adding its length, and trying to stream again. This can happen when queuing up multiple buffers near the end of the file - if one buffer reaches the end of the audio data, the next buffer will have a negative *timeLeft* parameter, so this must be accounted for.

```
33 double   OggSound::StreamData(ALuint   buffer, double timeLeft) {
34    char     data[BUFFERLENGTH];
35    int      read         = 0;
36    int      readResult   = 0;
37    int      section;
38 
39    int seek  = ov_time_seek(&streamHandle,(length - timeLeft)/1000.0);
40    if(seek != 0) {
41       return StreamData(buffer,timeLeft + GetLength());
42    }
```

OggSound.cpp

On line 45, we begin the process of actually streaming in some data, which we place in the temporary variable *data*, defined on line 34. While we haven't filled up the data array, we read some data from the vorbis stream, using the *ov_read* function. This function takes a number of parameters - the *streamHandle*, a pointer to the destination of the decoded data, how much data to decode, and some parameters that control the data type of the stream, which we will leave at their default values. The **while** loop will move along the *data* local variable, filling it up with decoded PCM data until it is full, or no more PCM data can be read, before breaking out of the loop.

With the data (hopefully!) read into the *data* variable, it can be buffered into the OpenAL buffer just like we did with the WAV PCM data last tutorial - using the *alBufferData* function. We then **return** the number of milliseconds of PCM data we actually decoded, so we know where to seek to in the stream next time we try to stream some data for a particular *SceneNode*.

```
   while(read < BUFFERLENGTH) {
      readResult = ov_read(&streamHandle, data + read,
                           BUFFERLENGTH - read, 0, 2, 1, & section);

      if(readResult > 0) {
         read += readResult;
      }
      else {
         break;
      }
   }
   if(read > 0) {
      alBufferData(buffer, GetOALFormat(), data,
                   read, (ALsizei)GetFrequency());
   }
   return (float)read/(channels*freqRate*(bitRate / 8.0))*1000.0f;
}
```

OggSound.cpp

SoundSystem Class

Header file

To enable the triggering of sounds arbitrarily in our game code, we need to modify the *SoundSystem* class slightly. We need a couple of new **public** functions - both called *PlaySound*, and enable the triggering the playback of a sound either globally or in 3D space, a **protected** function to update them, and a **protected** vector to store them in.

```
struct SoundPriority;
public:
   void     PlaySound(Sound* s, Vector3 position);
   void     PlaySound(Sound* s, SoundPriority p = SOUNDPRIORTY_LOW);

protected:
   void     UpdateTemporaryEmitters(float msec);
   vector<SoundEmitter*>   temporaryEmitters;
```

SoundSystem.h

Class file

Every time our *SoundSystem* updates, we should also update the temporary 'triggered' emitters under its control, so we need to add a call to the new *UpdateTemporaryEmitters* function in the *Update* function - everything else in the *Update* function can remain as-is.

```
void     SoundSystem::Update(float msec) {
   UpdateTemporaryEmitters(msec); //Add this line!
}
```

SoundSystem.cpp

To perform the actual temporary emitter updating, we need to iterate over the *temporaryEmitters* vector, and either call the *SoundEmitter's Update* function and add it to the *frameEmitters* vector for process by the *CullNodes* function, or **delete** the *SoundEmitter* if it has finished its sound playback.

```cpp
void  SoundSystem::UpdateTemporaryEmitters(float msec) {
   for(vector<SoundEmitter*>::iterator i = temporaryEmitters.begin();
                                      i != temporaryEmitters.end(); ) {
      if((*i)->GetTimeLeft() < 0.0f && !(*i)->GetLooping()) {
         delete (*i);
         i = temporaryEmitters.erase(i);
      }
      else{
         frameEmitters.push_back((*i));
         (*i)->Update(msec);
         ++i;
      }
   }
}
```

SoundSystem.cpp

The last thing we need to do to improve our *SoundSystem* class is to create a couple of *PlaySound* functions - one for a global sound (such as GUI audio feedback), and one for a temporary sound in 3D space (such as an explosion). Both functions create a new *SoundEmitter*, add it to the *temporaryEmitters* vector, and use its accessor functions to set the appropriate state. That's it! Your audio system should now be good to go, and be capable of playing a variety of sounds in different ways.

```cpp
void  SoundSystem::PlaySound(Sound* s, Vector3 position) {
   SoundEmitter* n = new SoundEmitter();
   n->SetLooping(false);
   n->SetTransform(Matrix4::Translation(position));
   n->SetSound(s);
   temporaryEmitters.push_back(n);
}
void  SoundSystem::PlaySound(Sound* s, SoundPriority p) {
   SoundEmitter* n = new SoundEmitter();
   n->SetLooping(false);
   n->SetSound(s);
   n->SetIsGlobal(true);
   n->SetPriority(p);
   temporaryEmitters.push_back(n);
}
```

SoundSystem.cpp

To test the ability to trigger sound effects, we're also going to add a couple of **if** statements to the *UpdateGame* function of the *MyGame* class: one to trigger a global sound, and one to trigger a sound playing at the origin:

```cpp
   if(Window::GetKeyboard()->KeyTriggered(KEYBOARD_1)) {
      SoundSystem::GetSoundSystem()->PlaySound(
         SoundManager::GetSound(soundNames[1]),SOUNDPRIORITY_HIGH);
   }
   if(Window::GetKeyboard()->KeyTriggered(KEYBOARD_2)) {
      SoundSystem::GetSoundSystem()->PlaySound(
         SoundManager::GetSound(soundNames[2]), Vector3());
   }
```

main.cpp

Tutorial Summary

That turned out to be quite a lot of code, didn't it! But, in adding it all to your game engine, you've learnt a little bit about how to efficiently stream in audio data, how to handle external libraries written in C, and made your sound playback more general purpose. That's everything for this short sound workshop series, but what you've learnt is by no means everything there is to know about game sound programming. We've not discussed environmental effects such as *reverb*, how to use *equalisers* to change the treble and bass of played back audio, or the important subject of *audio occlusion* - how the surfaces of a game level block the propagation of sound. The implementation of some of these things will be audio API specific, while others can be made general purpose. So, don't feel like you've been spoiled *too* much by being given a working audio system - there's a whole load of things you can research into to make your game's audio playback system unique!

Further Work

1) Try adding in the ability to load an entire Ogg file into one buffer, and stream an uncompressed WAV. Loading an entire ogg may sound counter-intuitive, but remember it may not just be a music track that is compressed - Ogg Vorbis is just as useful for small sounds like spot effects, too!

2) As well as audio output, OpenAL provides a system for receiving audio input from a connected microphone. Investigate the *alCaptureOpenDevice* and *alcCaptureStart* functions.

3) OpenAL supports the ability to add special effects such as reverb to a source, via the *Effects Extension* system. Read the *Effects Extension Guide* document contained in the OpenAL SDK folder, and try adding a simple reverb to a source.