Game Audio via OpenAL

Summary

In the *Graphics For Games* module, you learnt how to use OpenGL to create complex graphical scenes, improving your programming skills along the way, and learning about data structures such as scene graphs. In this workshop, you'll see how to add sounds to your game worlds using the OpenAL sound library.

New Concepts

Sound in games, OpenAL, PCM audio, binary file formats, FourCC codes, WAV files, limited resource management

Introduction

Audio has played an important part in gaming almost as long as there have been games to play - even *Pong* back in 1972 had simple sound effects. We've moved a long way from then; the 80s brought dedicated audio hardware such as the Commodore 64's SID chip that could play 3 simultaneous synthesised sounds, and later the Amiga brought the ability to play audio samples to the home gaming market. In modern gaming hardware, we can expect to hear many simultaneous sounds and music tracks, often in surround sound. Game developers now employ dedicated sound engineers that will carefully adjust the sounds in each game release to create an immersive aural experience - making sure that each individual sound is uniquely identifiable and correctly equalised, and that every music track suits the situation they will be played in.

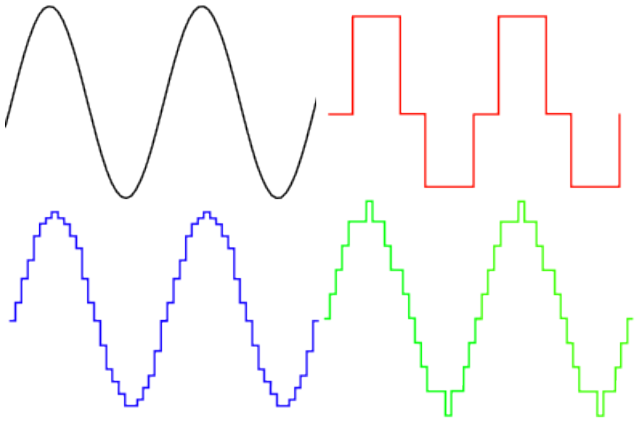
At the heart of a game's audio experience is the code that plays back the game sounds, and calculates which speakers they should use - the *sound system*. Although we can't hope to compete with the complex sound systems of AAA games, we should still be able to make a robust, simple system for the addition of sound in our 3D games, and that's what this workshop will assist you in creating.

Audio Data

Before we even begin to explore how to add in-game audio to our games, it's worth while having a brief overview of what sound actually *is*. Sound is simply the propagation of pressure waves through a medium, such as water or air. If we were to take a look at a sound wave using an oscilloscope, it'd look something like this:

A single continuous wave, with an *amplitude* (how large the peaks and troughs of the wave are - the larger the amplitude, the louder the sound), and a *frequency* (how often these peaks and troughs occur - the higher the frequency, the higher pitched the sound is).

This poses a problem for computers. Sound waves are continuous, yet computers store discrete, digital values - so how do we store our game audio? The answer is *Pulse-Code Modulation*, or **PCM** for short. To digitally represent sound, the sound is regularly sampled, and its amplitude stored as a value. PCM data has two important components - *sample rate* (measured in *hertz*, meaning cycles per second, just like a computer's CPU) which determines how often a sample of the sound is taken, and *bit depth*, which determines the range of amplitude values that can be stored. As there is limited bit depth, and so only a finite number of values that can be stored, the sampled values of a sound wave are *quantised* - rounded to the nearest value that the bit depth can hold. Here's an example of our sound wave being transformed into PCM data:



Clockwise from top left: A section of the original analogue wave, low bit depth and frequency, medium bit depth and frequency, hight bit depth and frequency

You should be able to see how both bit depth and sample rate determine the final quality of the digitised sound. To give you an idea of what the commonly used values for bit depth and sample rate are, audio CDs encode their data at a sample rate of 44,056hz (over 44k samples a second!), and a bit rate of 16 (65,536 unique values). The audio hardware in your computer can natively handle PCM data, and can turn it back into analog data, for outputting via speakers or headphones.

WAV Files

One of the most popular file formats to store PCM data is IBM / Microsoft's WAV file. Unlike the text files you'll have been reading and writing to so far, WAV is a 'binary' file format, meaning it contains no human-readable data. Well, *almost*. WAV files use something known as *FourCCs*, short for Four-Character Code, to identify the start of data structures within the file, commonly known as 'chunks'. As the name suggests, a FourCC is a simple 4-byte identifier, with the byte values in the range that represents readable characters. If you were to open a WAV file in a text editor such as *Notepad++*, you'd see the phrase **RIFF** at the start of the file, and then **WAVE** a few characters later - FourCCs that identify the file is of type RIFF (a container file format, meaning it could contain any sort of binary data), with WAVE data in it.

Here's a more detailed example of a WAV file's contents:

Size	Description	Value
4	FourCC	"RIFF"
4	Chunk Size	8
4	RIFF type	"WAVE"
	n additional chunks of type:	
4	FourCC	*Variable*
4	Chunk Size	32 bits
	Chunk Data	
	EOF	

After the initial file header containing RIFF and WAVE, we have a number of chunks, each of which has a FourCC, a size value, and a number of bytes matching that size value - this structure allows the file to be read in correctly, and searched through if necessary. WAV files can have many types of chunk, containing information on MIDI playback, que indicators within the file data, and even space for text notes. The most important two chunks, and the only two we'll be processing in the code later on, are the *format* chunk, and the *data* chunk. Here's what the format chunk looks like:

Size	Description	Value
4	FourCC	"fmt "
4	Chunk Size	16+extra
2	Compression code	1-65535
2	No. of channels	1-65535
4	Sample rate	32 bits
4	Avg. bytes per second	32 bits
2	Block align	1-65535
2	Sig. bits per sample	1-65535
2	Extra bytes	0-65535
	0-65535 Extra Bytes	

Like other WAV chunks, it begins with a FourCC and a size, before defining a number of important characteristics on the PCM data contained within the WAV. We can determine the number of channels the PCM data has (i.e. whether it is mono, stereo, 5.1, and so on), the sample rate, and bit rate. WAV files can contain either uncompressed or compressed PCM data (although uncompressed is far more common), so a compression code is also defined. Format chunks sometimes also specify additional data to support the compression type, so another size value is defined, determining the number of this extra data.

Finally, this is what the data chunk contains:

Size	Description	Value
4	FourCC	"data"
4	Chunk Size	32 bits
	Sample Data	

Not much to it! It's basically just a big lump of binary data, containing either compressed or uncompressed PCM data. All that's really worth pointing out is how the number of channels is handled by the PCM sample data: for each given 'time slice' of data (remember the 44,056 samples per second for a CD?), the PCM data for each channel is stored, one after the other, then the next samples, and so on. This interleaving of sample data per channel allows multichannel sound to be easily 'streamed' from the data source, without having to read data from different parts of the file.

Audio Hardware

Much as with graphical output, sound is usually handled by dedicated hardware, which may have a wide variety of capabilities. Inexpensive 'on board' sound chips might be little more than a digital to analogue signal converter connected to some audio jacks, while more expensive hardware may have support for natively decoding compressed audio, and contain sophisticated signal processors to add effects such as reverb. Higher end sound hardware will have a number of discrete channels, or *voices* that it can support (that is, the number of concurrent PCM sounds it can process into analogue data and output at one time, a value that is generally independent from whether the PCM data is mono, stereo etc.), while lower-end hardware might perform all PCM data mixing in software, outputting to a single combined channel. Depending on the exact audio features of the hardware, there might even be some amount of dedicated RAM, to buffer sound samples for higher performance.

OpenAL

As with graphics cards, the majority of the varying hardware features of sound devices are hidden behind an API, which will in turn target the vendor-provided drivers of the hardware. There's a few of these APIs around: *DirectSound* for Windows, *XAudio2* for Windows and Xbox 360, and *ALSA* for Linux.

One of the most pervasive audio APIs is *OpenAL*, currently hosted and maintained by Creative Labs. As its name suggets, OpenAL aims to be for audio what OpenGL is for graphics - a cross-platform, easy to use API, that supports both hardware acceleration, and software fallbacks where necessary. Unlike OpenGL, which has an Architecture Review Board to discuss features and improvements to the language (currently handled by the Khronos Group), no such collaborative entity currently exists for OpenAL, with all recent development being undertaken by Creative Labs, developers of the SoundBlaster series of sound cards. Like OpenGL, OpenAL is a giant 'state machine', with its API functions adding commands to an internal stack, which are popped off one after the other as they are processed.

Implementations of OpenAL exist for Windows, Apple Mac and iOS devices, Android, and the Xbox 360. Sony's Playstation 3 has an OpenAL-like sound API called x, making OpenAL a popular choice for cross platform programs, and a firm understanding of how to use it an advantage for a budding game developer.

OpenAL allows a number of sound sources to be played in 3D space, with all of the processing required to determine the volume of sound required in each speaker to generate the spatial effect handled automatically. It supports a number of parameters to determine how samples are played back, and how distance from a sound source effects sound playback volume. OpenAL defines a number of constructs to control the eventual sound output - the Device, the Context, the Listener, Sources, and Buffers.

The Device

As its name implies, the *device* represents your audio hardware. You don't need to worry much about the device other than selecting one to use when initialising your sound system - you may have multiple sound devices in your PC, and some hardware displays itself as multiple devices for hardware and software PCM playback, so a device must be chosen - OpenAL can be instructed to select the 'best' device it finds, though.

The Context

Similar to OpenGL, OpenAL uses a *context* to maintain the relationship between the data structures used to calculate the final outputted audio. Generally you'll only have one context, which will be created when you initialise your application, and destroyed on its exit.

The Listener

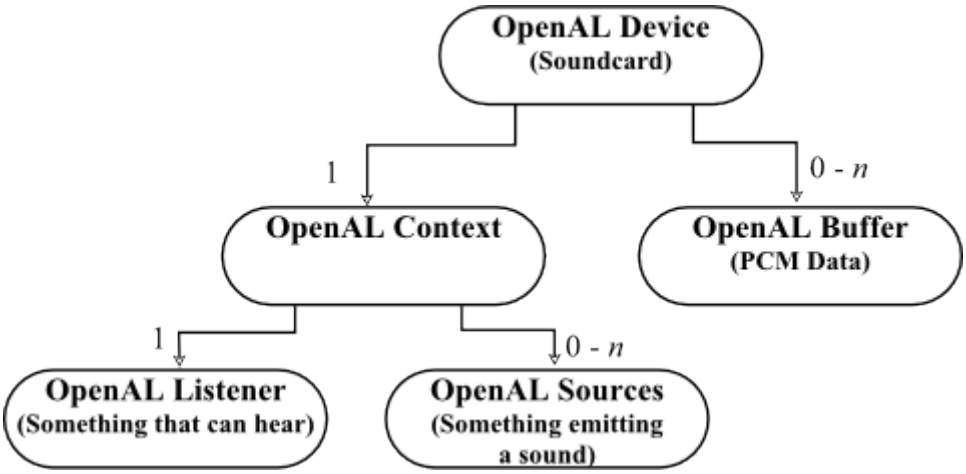
In order to calculate the volumes and directions of the sounds your application plays in 3D space, OpenAL needs a frame of reference - the *listener*. Think of this as being where the microphone is positioned - you'll *probably* have the listener set to the same position as the camera, but you don't necessarily *have* to. In OpenAL the listener object keeps track of things like position, orientation, and master volume. OpenAL only has a single listener, so you don't even have to create or delete it - OpenAL just has a set of API functions to directly handle listener properties. Rather than the matrix representation we use in OpenGL to keep track of the camera, in OpenAL the listener is defined by three vectors - a position, a 'forward' direction, and an 'up' direction. We need an 'up' component to a listeners component as a single direction vector doesn't provide any information as to the rotation *around* that vector - imagine that for some reason you're standing on your head, the sounds you hear to either side will be inverse to if you were being sensible and standing the right way up!

Sources

Next, we have *sources*. Source objects define where a sound is playing in 3D space, keeping track of position, direction, and information like whether the sound is looping indefinately, its pitch, and its volume. We only need a direction vector for a sound source, as an emitted sound has no concept of an 'up' vector - if you turn a speaker upside down, it'll still sound the same. Sources also keep track of which sound sample they are playing, and and which position in playback they are at.

Buffers

Buffers are simply the OpenAL structure for PCM data - they keep track of bit rate, sample frequency, as well as holding the actual PCM data. Unlike listeners and sources, buffers aren't a property of a single context, but defined at the device level - all of your OpenAL contexts share the same PCM data.



An overview of OpenAL and its data structures

Positional Audio

A good 3D game requires audio that is accurately placed in 3D space - it helps immersion, and helps the player locate the sources of sound. To create the effect of a sound being in 3D space, its volume is changed as the player moves away or towards its source, and is panned between the left speaker / headphone to the right in accordance to the relative position and orientation between the player and the sound source. So a sound that is directly to the left of the player is played almost entirely out of the left speaker, while a sound in front would be heard in the left and right speaker equally (or perhaps out of the center speaker in a 5.1 audio setup), and if the player spins on the spot, the sound pans across the speakers to give the illusion of the sound moving in space. For this reason, OpenAL can only play mono sounds (those that have a single channel)in 3D space - there's no way to accurately mix multiple channels in 3D space. A sound that has multiple channels will always be played without any 3D mixing in OpenAL, so if you add a sound to your game that doesn't seem to sound right, check how many channels it has!

Creating an Audio System

Like OpenGL, OpenAL is just an API - if you want to actually use it in your games, you're going to have to do a bit of coding, and create some classes that actually target the API. How to intuitively add audio capability to your games is also something that takes consideration - the audio system should be able to 'look after itself', and not require lots of rewriting code.

Consider how sound is actually used in the games - what *types* of sound are there? There's music, which probably loops around, or changes track; there's 'one off' sounds triggered by in-game menus (which are generally 'global' in that they aren't calculated in 3D space, just sent straight to the speakers); and then there's the actual 'in game' audio in 3D space - it too might be 'one off' (the

sound of the player's gun firing), attached to an in-game object (a character speaking while walking down a corridor) or looping ambience (a dripping tap, the sound of rain hitting a window etc). An audio system should be able to handle all of these cases with ease.

Performance

Finally, there's also the issue of performance to consider. Earlier it was mentioned that some sound hardware can handle multiple, independent PCM sound decodings, allowing several *voices* to be mixed in hardware at once. Well, what if there isn't any hardware capability? In the drivers, all of the sounds being played will be mixed together in software before being sent to the sound card. In either case, there's a limit to the amount of sounds we can process at once - either the limit of the sound hardware's voices, or how many sounds can be mixed before performance starts to drop. In OpenAL, this software mixing is limited to a maximum of 128 sources (and may be less when dealing with hardware-only mixing), so we can't just throw sounds at the audio system and expect everything to play as anticipated - we must treat OpenAL sources as a limited resource, and only give them to 'important' sounds.

So, what counts as an *important* sound? First off, we want to cull any sounds that are too far away - a mouse sneezing 100 miles away probably doesn't warrant taking up one of our precious sound sources! OpenAL defines each active sound source as having a radius, used to determine how loud a sound should be - outside of this radius, the sound is considered to be silent, while the sound gets louder as the listener gets closer to the center of the radius.

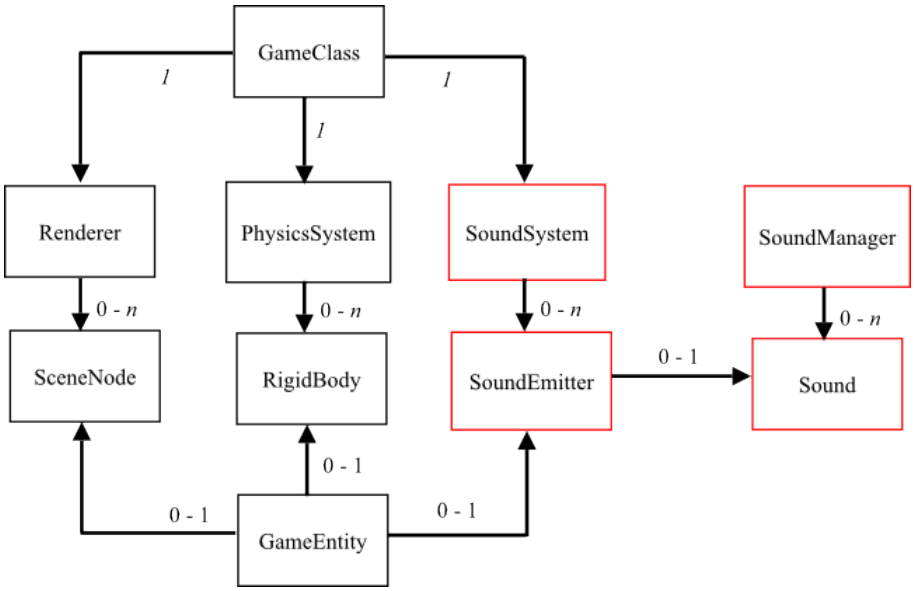
That's the first thing we'll get rid of, then - sounds that the listener is too far away from to hear. That'll probably drastically reduce the number of sources we need! But we *might* still run out of free sources - perhaps the player is hammering a button that makes a beeping noise, or the game is running on a mobile device with a limited number of channels. In such cases, we still need to be able to sort sounds. Imagine our audio system only have a single channel left, which sound should it play - a sneezing mouse right next to you, or the sound of the enemy firing his gun at you from behind cover up ahead? If we only sorted by radius, perhaps the mouse with the snuffly nose would be heard, and that annoying camper gets to headshot you in silence. As well as radius, it's good if sounds have a *priority* as well - a simple integer value will do, with higher values being assigned to more important noises. With both radius and priority taken into account, it's far more likely that the important sounds will receive a source.

It's worth noting that the process of assigning one of the limited audio sources available is a constant process - as sounds complete, sources should be returned to the resource pool to use elsewhere, and as the player and game objects move about, sounds may move beyond their hearing radius. So, just as the *Renderer* class must process the scene hierarchy every frame in order to determine what you should see, a well-behaving sound system will process it's entities every frame to determine what you should *hear*.

Integration

The previous sections have provided an overview of the considerations to make when designing and building sound playback capability, but what about the practicalities of implementing it in a game engine? In the previous few modules, we have been building up our own simple game engine, with support for graphics rendering and rigid body physics, so lets try and implement a sound system into that!

You should now be pretty familiar with the idea of 'subsystem' classes being connected to our game by some sort of controlling class - our *Renderer* class wrapped up graphics rendering and was controlled by *SceneNodes*, and our *PhysicsSystem* operated on *RigidBodies*. So it makes sense to follow this design pattern again - we're going to have a *SoundSystem* class that encapsulates an audio API (in this case OpenAL), and which operates on a number of *SoundEmitter* classes, which are in turn attached and detached using *GameEntities*. Our new class layout is going to look like this:



An overview of the new systems being introduced - new classes are in red

Once we've implemented these, we can add new *GameEntities* to the game world that can play sound - a character speaking to the player might have a *SoundEmitter* to play back their vocals, while another *GameEntity* might only have a *SoundEmitter* and not a *SceneNode* or a *RigidBody* - such a class could be used to playback 'ambient' sound around the player. As well as the *SoundEmitters* created and attached via *GameEntities*, the *SoundSystem* is also going to be able to spawn temporary *SoundEmitters* - useful for 'one off' sounds such as aural feedback from your in-game GUI or gunfire from the player's weapon.

To play back sound data, we need some sort of class to load and process this data from files - the *Sound* class. Finally, we are going to have a class to keep track of these loaded in *Sounds* - a *SoundManager*. This class will have a collection of *Sounds*, and functions to load and retrieve them - by encapsulating all of this functionality we can better keep track of data loaded into memory, and prevent accidentally loading in a file multiple times.

Example program

To show off our new sound playback capability, we're going to take another look at the code from the Game Technologies module. In it, we started with a simple scene of a cube robot standing on the ground. That's not very exciting, so we're going to make things a bit more interesting by spawning some more *GameEntities* randomly throughout the world, that have *SoundEmitters* attached to them, each of which plays a random sound. OK that's still not a very interesting game, but it will demonstrate how to get sound into your own (hopefully more interesting!) game projects.

The *Sound* class

We begin our example program by taking a look at how to implement WAV file loading, via the *Sound* class.

Header file

We start off with the header file, as usual. We're dealing with files and filename strings, so we have a few STL **includes**. We also need to include the OpenAL API header file, as we'll be using OpenAL buffer constsructs. We're also going to define a **struct**, *FMTCHUNK*, which defines the WAVE file format chunk, described earlier - you'll see how it's used shortly.

```
#pragma once

#include <string>
#include <iostream>
#include <fstream>

#include "../OpenAL 1.1 SDK/include/al.h"

using namespace std;

struct FMTCHUNK {
   short           format;
   short           channels;
   unsigned long   srate;
   unsigned long   bps;
   short           balign;
   short           samp;
};
```

Sound.h

Next, we have the actual *Sound* class itself, starting with some accessors for the data we're going to be pulling out of our WAVE files. Note how in this particular class, our **constructor** and **destructor** are **protected**! We only want new *Sounds* to be created via the SoundManager class, which is going to make sure we don't accidentally load in duplicate WAVE files.

```
class Sound {
 friend class SoundManager;
public:
   char*           GetData()       {return data;}
   int             GetBitRate()    {return bitRate;}
   float           GetFrequency()  {return freqRate;}
   int             GetChannels()   {return channels;}
   int             GetSize()       {return size;}
   ALuint          GetBuffer()     {return buffer;}

   ALenum          GetOALFormat();
   float           GetLength();

protected:
   Sound();
   ~Sound(void);

   void            LoadFromWAV(string filename);
   void            LoadWAVChunkInfo(ifstream &file, string &name,
                                    unsigned int &size);

   char*    data;

   float    length;
   int      bitRate;
   float    freqRate;
   int      size;
   int      channels;

   ALuint   buffer;
};
```

Sound.h

Class file

The **constructor** and **destructor** for our *Sound* class are pretty simple - the only heap memory we'll be allocating is the data variable, which of course must be **deleted** in the **destructor**.

```
1  #include "Sound.h"
2
3  map<string, Sound*> Sound::sounds;
4
5  Sound::Sound() {
6     buffer      = 0;
7     bitRate     = 0;
8     freqRate    = 0;
9     length      = 0;
10    data        = NULL;
11 }
12
13 Sound::~Sound(void)  {
14    delete data;
15 }
```

Sound.cpp

Internally, OpenAL defines the contents of PCM data via a *format* enumerator, and it is useful to be able to determine this format for a *Sound* instance. *GetOALFormat* will do just that - OpenAL PCM formats define the bit rate, and channel count, so a couple of **if** statements (and **ternary** statements!) will determine the correct OpenAL enumerator for a given sound. If for some reason the loaded PCM data is of an unusual type (very high bitrate, or 5.1 channels, perhaps), it will return a 'safe' default value to attempt to play the sound with.

```
16 ALenum Sound::GetOALFormat()  {
17    if(GetBitRate() == 16)  {
18       return GetChannels() == 2 ? AL_FORMAT_STEREO16:AL_FORMAT_MONO16;
19    }
20    else if(GetBitRate() == 8) {
21       return GetChannels() == 2 ? AL_FORMAT_STEREO8:AL_FORMAT_MONO8;
22    }
23    return AL_FORMAT_MONO8;
24 }
```

Sound.cpp

Next, we're going to define is *GetLength* - this simply returns the number of seconds of audio the PCM data encodes. The equation isn't hard to figure out - for a given amount of data, it's length will be that size divided up by the channels, frequency, and bitrate - and we'll even keep that as a variable so we only have to calculate it once!

```
25 float Sound::GetLength() {
26    return length;
27 }
```

Sound.cpp

That's all the basics of the *Sound* class out of the way, now for the tricky bit - loading in a WAVE file. As described earlier, WAVE files are made up of *chunks* - so we need to create a loop that will iterate over and process the file's chunks. First though, we have to actually open the file, using an *ifstream* - with an extra tag, *ios::binary* used to denote that we're loading in binary data. If we can't load the file we instantly **return**, otherwise, starting on line 35, we start the process of loading in the WAVE chunks. To grab the chunk info we're going to use a little helper function, *LoadWAVChunkInfo*, which will read the chunk's FourCC and size into the local variables defined on lines 35 and 36.

With the chunk information loaded, we can check the FourCC - we're only concerned about the format and data chunks, but we also need to handle the WAVE header, which includes the FourCC "WAVE" - we need to skip those four bytes, which we do using the *ifstream* function *seekg* (so called because it *seeks* through the file to the desired position for the stream's internal *get* pointer). Really, we should check the validity of the file by reading these bytes and checking they equal "WAVE", but for now we'll assume that it's a correct file structure.

If the chunk is a format chunk, we read in some data into a temporary *FMTCHUNK* **struct**, and extract the bit rate, frequency rate,and number of channels from it. If it's data, we just initialise some memory, and read the file into it. Some WAVE files may be of a compressed format, in which case we'd have to do some extra processing on the resulting data to turn it into OpenAL compatible PCM data, but we're going to assume it's uncompressed for now - remember, we can check whether it's compressed data or not by checking the appropriate field in the *FMTCHUNK* **struct**.

If the chunk is of a type we aren't interetested in (MIDI data, for instance) we just skip over it via the *seekg* function (line 59). Once that's all done, we close the file, as there's nothing else we need from it, so all we have left to do is work out the *Sound's* length.

```
28 void  Sound::LoadFromWAV(string filename) {
29    ifstream file(filename.c_str(),ios::in|ios::binary);
30    if(!file) {
31       cout << "Failed to load WAV file '" << filename << "'!" << endl;
32       return;
33    }
34
35    string       chunkName;
36    unsigned int chunkSize;
37
38    while(!file.eof()) {
39       LoadWAVChunkInfo(file,chunkName, chunkSize);
40
41       if(chunkName == "RIFF") {
42          file.seekg(4, ios_base::cur);
43       }
44       else if(chunkName == "fmt ") {
45          FMTCHUNK fmt;
46
47          file.read((char*)&fmt,sizeof(FMTCHUNK));
48
49          bitRate  = fmt.samp;
50          freqRate = (float)fmt.srate;
51          channels = fmt.channels;
52       }
53       else if(chunkName == "data") {
54          size     = chunkSize;
55          data     = new char[size];
56          file.read((char*)data,chunkSize);
57       }
58       else{
59          file.seekg(chunkSize, ios_base::cur);
60       }
61    }
62    file.close();
63    length = (float)size / (channels*freqRate*(bitRate/8.0f))*1000.0f;
64 }
```

Sound.cpp

The final function we need to define is *LoadWAVChunkInfo*. It takes in a reference to an *ifstream*, *string*, and *unsigned int*, and reads 8 bytes from the *ifstream*, filling in the *name* and *size* references.

```
65 void  Sound::LoadWAVChunkInfo(ifstream &file, string &name,
66                                unsigned int &size) {
67    char chunk[4];
68    file.read((char*)&chunk,4);
69    file.read((char*)&size,4);
70
71    name = string(chunk,4);
72 }
```

Sound.cpp

The *SoundManager* class

The *Sound* class is pretty simple, and is controlled by another very simple class - the *SoundManager*. All it has is 3 **static** functions, and a **static** *map* - even its *constructor* and *destructor* are empty! This is in fact an entirely static class - it has no member instance data or functions, and so there's no need to ever instantiate one, thus its *constructor* and *destructor* are also made **protected**, just like the *Sound* class. It's three static functions are pretty self explanatory - they *Add* a sound to the **static** map, *retrieve* a sound, and **delete** all of the sounds in the map. As you should have noticed, the *SoundManager* class has been made a friend of the *Sound* class, which enables it to call its **protected** functions - in this case we're using this ability to allow the *SoundManager* to call the *Sound* **constructor**, as we only want *Sounds* to ever be made by the manager, for correct 'book keeping'.

Header file

```
1  #pragma once
2
3  #include <map>
4  #include "../nclgl/Sound.h"
5
6  using std::map;
7
8  class SoundManager {
9  public:
10    static void    AddSound(string n);
11    static Sound*  GetSound(string name);
12
13    static void    DeleteSounds();
14 protected:
15    SoundManager(void);
16    ~SoundManager(void);
17
18    static map<string, Sound*> sounds;
19 };
```

SoundManager.h

Class file

The first function we'll look at is the *AddSound* function - we need to ensure duplicated sounds don't end up in the **static** map, and we also need to make sure the data we're loading in is valid. Checking for duplicates is easy - our *GetSound* function is going to do that for us, so we can check the return value of that and only create a new *Sound* if it is NULL. If it is NULL, we're going to make a new

Sound, and check what file format the filename is. A fully fledged sound system might support many different file formats, so our *AddSound* function needs to be able to handle them all, and call the appropriate functions. For now, we're going to add in WAVE file support, by using the **substr** function on line 5 to extract the final three characters of the input file name - we're going to assume if the file extension is wav, then it is a WAVE file. If it is, we'll call our *LoadFromWAV* function to extract the PCM data from the file, and then call a couple of OpenAL functions.

As with OpenGL, OpenAL uses numerical 'names' to identify its data structures, so, on line 10, we generate a new buffer name using the function *alGenBuffers*. WAVE files don't need much processing beyond getting the PCM data and ancillary information such as bit rate and frequency out of them, so we can buffer the data straight into OpenAL using the function *alBufferData*. Once the PCM data is in OpenAL, we could **delete** the data of the *Sound* if we like, just remember to NULL the pointer afterwards, so the **destructor** doesn't break!

```
1  void  SoundManager::AddSound(string name) {
2     if(!GetSound(name)) {
3        Sound *s = new Sound();
4
5        string extension = name.substr(name.length()-3,3);
6
7        if(extension == "wav") {
8           s->LoadFromWAV(name);
9
10          alGenBuffers(1,&s->buffer);
11
12          alBufferData(s->buffer,s->GetOALFormat(),s->GetData(),
13                       s->GetSize(),(ALsizei)s->GetFrequency());
14       }
15       else{
16          cout << "Invalid extension '" << extension << "'!" << endl;
17       }
18       sounds.insert(make_pair(name, s));
19    }
20 }
```

SoundManager.cpp

Our final two **static** functions are *GetSound* and *DeleteSounds*. As their names suggest, *GetSound* will get a *Sound* instance from the static map, or return NULL if no *Sound* paired with the given name exists, while *DeleteSounds* will empty and **delete** the contents of the map. Why do we need to do this? As we're not deleting them anywhere else, our *Sound* instances will otherwise persist, and won't be deleted. Even if the static map itself is deconstructed when it falls out of scope on program exit, maps don't explicitly delete their contained objects, so it must be done manually.

```
21 Sound*      SoundManager::GetSound(string name) {
22    map<string, Sound*>::iterator s = sounds.find(name);
23    return (s != sounds.end() ? s->second : NULL);
24 }
```

SoundManager.cpp

```
25 void  SoundManager::DeleteSounds() {
26    for(map<string, Sound*>::iterator i = sounds.begin();
27        i != sounds.end(); ++i) {
28       delete i->second;
29    }
30 }
```

SoundManager.cpp

The *SoundEmitter* class

Now we have the code to load in sounds, we need to be able to play them! To do this, we're going to make a *SoundEmitter* class, which will act as the connector between our in-game entities, and the *SoundSystem* class we will be looking at shortly.

Header file

First off, some **includes** - we need to know about *SceneNodes*, *Sounds*, and the *SoundSystem* class we're going to create later. Then, we define an **enum** called *SoundPriority* - we'll use this to assign an easy to remember value to represent how important a sound is. Using named constants removes the ambiguity around using a **integer** for priority - is 0 a low or a high priority? It's much easier to work this out if it equates to the **enum** *SOUNDPRIORTY_LOW*!

```
1  #pragma once
2  #include "scenenode.h"
3  #include "Sound.h"
4  #include "SoundSystem.h"
5
6  enum SoundPriority {
7     SOUNDPRIORTY_LOW,
8     SOUNDPRIORITY_MEDIUM,
9     SOUNDPRIORITY_HIGH,
10    SOUNDPRIORITY_ALWAYS
11 };
12
13 struct OALSource;
```

SoundEmitter.h

Now on to the *SoundEmitter* class itself. For convenience, we're going to have a couple of **constructors** - a **default** one, and one which takes a *Sound* to start playing immediately. This gives us a bit of choice in how we use our *SoundEmitters* - things like ambient sound emitters in the levels of your game might have a *Sound* attached to them straight away, while maybe an NPC will have a *SoundEmitter* for when he or she speaks one of many different phrases. As we have multiple **constructors**, we're going to keep the initialisation common to both of them in another function - *Reset*. We also have a **destructor**, as usual.

```
14 class SoundEmitter   {
15 public:
16    SoundEmitter(void);
17    SoundEmitter(Sound* s);
18    ~SoundEmitter(void);
19
20    void                 Reset();
```

SoundEmitter.h

Next, some **public** accessors. Our *SoundEmitter* needs to be able to get and set a *Sound* to play, and a priority, at the least, but we're going to add a few extra features in too - our *SoundEmitter* is going to be able to have a volume set, be able to loop if we want, and have a maximum radius in which it can be heard. As with our *RigidBody* class, our *SoundEmitter* class will also have a *target* - but this time, instead of using this variable to **set** the position of a *SceneNode*, we're going to use it to **get** the position, so we know where in our game world the sound is playing from. As all of these will be trivial in terms of their compiled code, we're going to make all of these accessor functions *inline*, too.

```
21    void                SetSound(Sound *s);
22    inline Sound*       GetSound()                  {return sound;}
23
24    inline void         SetPriority(SoundPriority p){priority = p;}
25    inline SoundPriority GetPriority()              {return priority;}
26
27    inline void         SetVolume(float volume)     {
28                           volume = min(1.0f, max(0.0f, volume));
29                        }
30    inline float        GetVolume()                 {return volume;}
31
32    inline void         SetLooping(bool state)  {isLooping = state;}
33    inline bool         GetLooping()            {return isLooping;}
34
35    inline void         SetRadius(float value)      {
36                           radius = max(0.0f, value);
37                        }
38    inline float        GetRadius()           {return radius;}
39
40    inline float        GetTimeLeft()         {return timeLeft;}
41
42    inline OALSource*   GetSource()           {return currentSource;}
43
44    void                SetTarget(SceneNode *s)    { target = s;}
```

SoundEmitter.h

As we're dealing with a limited number of OpenAL sources, we need functions to attach and detach them from a *SoundEmitter*, and how to compare the priority of two *SoundEmitter*, so that we can sort them. We also need a function to update the internal state of the *SoundEmitter* - how long of its current sound is left to play, the current position and volume of the OpenAL source attached to it, and so on. We also have a number of member variables, including a pointer to an *OALSource* **struct** - we'll define that when we define the *SoundSystem* later, but as you can probably guess, it encapsulates an OpenAL source.

```
45    void                AttachSource(OALSource* s);
46    void                DetachSource();
47
48    static bool CompareNodesByPriority(SoundEmitter *a,SoundEmitter* b);
49
50    virtual void   Update(float msec);
51
52 protected:
53    Sound*         sound;
54    OALSource*     currentSource;
55    SoundPriority  priority;
56    Vector3        position;
57    float          volume;
58    float          radius;
59    bool           isLooping;
60    float          timeLeft;
61 };
```

SoundEmitter.h

Class file

Our class definition begins with the easy bits - **constructors** and **destructors**. Both **constructors** call our *Reset* function, with the overridden **constructor** also immediately calling *SetSound*. Our **destructor** is pretty simple - all it has to do is make sure its OpenAL source is detached, as otherwise we might end up permanently losing our limited number of sound sources if they get attached to a temporary *SoundEmitter* such as a gunshot. The *Reset* function sets our *SoundEmitter* member variables to some sensible defaults.

```
1  #include "SoundEmitter.h"
2
3  SoundEmitter::SoundEmitter(void) {
4     Reset();
5  }
6  SoundEmitter::SoundEmitter(Sound* s) {
7     Reset();
8     SetSound(s);
9  }
10
11 void  SoundEmitter::Reset() {
12    priority       = SOUNDPRIORTY_LOW;
13    volume         = 1.0f;
14    radius         = 500.0f;
15    timeLeft       = 0.0f;
16    isLooping      = true;
17    currentSource  = NULL;
18    sound          = NULL;
19 }
20
21 SoundEmitter::~SoundEmitter(void)   {
22    DetachSource();
23 }
```

SoundEmitter.cpp

Our sound system is going to order *SoundEmitters* by their priority, using the **STL sort** function. To do so, we need a function that takes in two *SoundEmitters* and returns a **bool** - you may remember we did this to sort *SceneNodes* by their distance from the camera in an earlier tutorial. Although we're using an **enum** to store our priority, remember an **enum** value is just an **unsigned integer**, and so can easily be compared against.

```
24 bool  SoundEmitter::CompareNodesByPriority(SoundEmitter *a,
25                                            SoundEmitter* b) {
26    return (a->priority > b->priority) ? true : false;
27 }
```

SoundEmitter.cpp

In order to play a new sound, we need to 'reset' how long is left of the sound to play to the length of the new sound. We're also going to detach the current source, if any, to make sure the old sound doesn't continue playing.

```
28 void     SoundEmitter::SetSound(Sound *s) {
29    sound = s;
30    DetachSource();
31    if(s) {
32       timeLeft = s->GetLength();
33    }
34 }
```

SoundEmitter.cpp

Now for the important bit - attaching and detaching OpenAL sound sources to our *SoundEmitters*. OpenAL sources are represented by the *OALSource* **struct**, which has two values, a **bool** to mark whether it's in use or not, and an OpenAL 'name' for the source. To attach a *SoundEmitter* to the source, we need to first mark the *OALSource* as in use, and then set two important values - the radius of the *SoundEmitter*, and the OpenAL reference distance - this determines how quickly the sound coming from the source fades away with distance, and marks the point at which the volume has faded by 50%. In both cases, we use the *alSourcef* function, which behaves in a similar way to many OpenGL functions - the function name determines what the function operates on (in this case a source), the *f* determines it is **float** data we are using, the first parameter is an OpenAL name, then an API defined constant (in this case *AL_MAX_DISTANCE* and *AL_REFERENCE_DISTANCE*), and finally the value to send.

Then, using the *alSourcei* function, we attach an OpenAL *buffer* to the source - the buffer of the current *Sound*. We can set the current playing position for the current sound using the *AL_SEC_OFFSET* constant, which should be set to the length of the *Sound*, minus how much of the *Sound* there is left to play - remember, when a source is attached, the source doesn't know how much of a sample is 'supposed' to have been played. For example, imagine a radio playing a song in a game; we might move out of its radius and then back towards it, so we need to make sure it is always playing its song from the correct place. lastly, we instruct the OpenAL sound source to start playing, appropriately enough with the *alSourcePlay* function.

```
35 void     SoundEmitter::AttachSource(OALSource* s) {
36    currentSource = s;
37
38    if(!currentSource) {
39       return;
40    }
41    currentSource->inUse = true;
42
43    alSourcef(currentSource->source, AL_MAX_DISTANCE, radius);
44    alSourcef(currentSource->source, AL_REFERENCE_DISTANCE,
45              radius * 0.2f);
46    alSourcei(currentSource->source,AL_BUFFER,sound->GetBuffer());
47    alSourcef(currentSource->source,AL_SEC_OFFSET,
48             (sound->GetLength()/ 1000.0) - (timeLeft / 1000.0));
49    alSourcePlay(currentSource->source);
50 }
```

SoundEmitter.cpp

Detaching a sound source is easier - we mark it as not in use, and silence it by setting its volume to 0, and using the *alSourceStop* API function to stop it from continuing playback. Just to make sure the 'old' sound doesn't continue to play, we also use the *alSourcei* function in conjunction with the *AL_BUFFER* constant again, sending a value of 0 - just like in OpenGL, a value of 0 'turns off' a feature.

```
51 void     SoundEmitter::DetachSource() {
52    if(!oalSource) {
53       return;
54    }
55    alSourcef(oalSource->source,AL_GAIN,0.0f);
56    alSourceStop(oalSource->source);
57    alSourcei(oalSource->source,AL_BUFFER,0);
58
59    oalSource->inUse = false;
60    oalSource = NULL;
61 }
```

SoundEmitter.cpp

Next, we need to define the *Update* function, which updates the currently attached OpenAL source (if one exists), as well as calculating how long of the current sound is left (line 63). On line 65, we increment how long of the sample is left if the sound is supposed to loop - play its sound sample over and over again. Beginning on line 68, we begin the process of updating the current OpenAL source. We must tell it the *SoundEmitters* current position, whether it is looping or not, its volume, and its radius. Why do we have to set these values every frame? First off, the *SoundEmitter* might have a different source attached than the previous frame, and accessor functions like *SetVolume* might also have been called. It's easier to just always set these values than it is to add in logic for detecting when an attached source has changed, and whether a value has changed. On line 72 set the OpenAL source's position to that of our *SoundEmitter* - if it has a current target *SceneNode*, we get its position, otherwise we'll just use the value of the member variable position.

```
62 void     SoundEmitter::Update(float msec) {
63    timeLeft -= msec;
64
65    while(isLooping && timeLeft < 0.0f) {
66       timeLeft += sound->GetLength();
67    }
68
69    if(oalSource) {
70       Vector3 pos;
71
72       if(target) {
73          pos = target->GetWorldTransform().GetPositionVector();
74       }
75       else {
76          pos = this->position;
77       }
78
79       alSourcefv(oalSource->source,AL_POSITION,(float*)&pos);
80
81       alSourcef(oalSource->source, AL_GAIN       ,volume);
82       alSourcei(oalSource->source, AL_LOOPING    ,isLooping ? 1 : 0);
83       alSourcef(oalSource->source, AL_MAX_DISTANCE, radius);
84       alSourcef(oalSource->source, AL_REFERENCE_DISTANCE,radius*0.2f);
85    }
86 }
```

SoundEmitter.cpp

That's everything for the *SoundEmitter* class. We could abstract out the OpenAL specific parts of our *SoundEmitter* class, via an *OpenALSoundEmitter* subclass, if we wanted to have a choice of audio libraries. But for the purposes of this tutorial, the current design is fine.

The SoundSystem class

Time for the final, but most important class - the sound system itself! We're going to make a *SoundSystem* class to encapsulate all of the workings of our sound engine, just like *OGLRenderer* and *Renderer* encapsulate all of the tricky parts of graphics rendering. Like the *PhysicsSystem*, it is a **Singleton** - a way of making a single, globally accessible class. Not everyone is a fan of the **Singleton** 'design pattern' - some even call it an 'anti-pattern', but for our needs, it's ideal. A **Singleton** is a type of class where only one instance is ever created, and is globally accessible via a **static** accessor function. It will let us access the sound system from anywhere in our game's code, allowing sounds to be played from anywhere without having to pass a **pointer** around. As we only need one sound system, it makes sense to limit the class to a single instantiation.

Header file

Before we create the *SoundSystem* class, though, we need to include OpenAL, and define a **struct** - the *OALSource* **struct** introduced earlier. In it we only have two values, a **bool** to determine whether the source is in use or not, and an OpenAL 'name' for the source.

```
1  #pragma once
2
3  #include <vector>
4  #include <algorithm>
5
6  #include "Sound.h"
7  #include "SoundEmitter.h"
8  #include "../OpenAL 1.1 SDK/include/al.h"
9  #include "../OpenAL 1.1 SDK/include/alc.h"
10
11 using std::vector;
12
13 class SoundEmitter;
14
15 struct OALSource {
16    ALuint   source;
17    bool     inUse;
18
19    OALSource(ALuint src) {
20       source   = src;
21       inUse    = false;
22    }
23 };
```

SoundSystem.h

Now for the *SoundSystem* class itself. We're going to have the **constructor** and **destructor protected**, just like with the *Sound* class, and instead handle the lifetime of our *SoundSystem* using two **static** functions *Initialise* and *Destroy*, as well as the *GetSoundSystem* accessor function mentioned earlier.
Our *Initialise* function takes an optional **unsigned integer**, *channels*. This can be used to limit the number of individual sources OpenAL will try to generate - handy for limiting the performance hit of our new *SoundSystem* on resource limited devices!

```
24 class SoundSystem {
25 public:
26    static void Initialise(unsigned int channels = 32) {
27       if(!instance) { instance = new SoundSystem(channels);}
28    }
29
30    static void Destroy() {delete instance;}
31
32    inline static SoundSystem* GetSoundSystem() {return instance;}
```

SoundSystem.h

In our *SoundSystem*, we need to have a listener object of some sort, to determine the spatialisation and volume of the sounds we want played. We could use a *Camera* for this purpose, or maybe a *GameEntity*; instead, we'll use a simple Matrix4 containing the orientation and position of whatever we want to be our sound 'origin', and update it every frame inside the *UpdateGame* function of the *GameClass*. This way, it doesn't matter what class type we take our positional information from, as long as we can turn it into a matrix. Our only other public functions are an *Update* function, which takes a millisecond **float** value, and a *SetMasterVolume* function.

We also need some way of adding and removing *SoundEmitters* from the control of our *SoundSystem*, in the same way as we do for *SceneNode's* in the *Renderer*, and *RigidBodies* in the *PhysicsSystem*.

```
33    void  SetListenerTransform(const Matrix4&transform) {
34       listenerTransform = transform;
35    }
36    Matrix4 GetListenerTransform() {
37       return listenerTransform;
38    }
39    void     AddSoundEmitter(SoundEmitter* s) {emitters.push_back(s);}
40    void     RemoveSoundEmitter(SoundEmitter*s);
41
42    void     Update(float msec);
43
44    void     SetMasterVolume(float value);
```

SoundSystem.h

In our **protected** section, we have our **constructor** and **destructor**, and some functions that will update the OpenAL listener, obtain, attach and detach sources, and culling nodes from processing if they are too far away. For our member variables, we have a vector to store our OpenAL sources, a **vector** to store the *SoundEmitters* the *SoundSystem* will be processing in the current update, a **pointer** to a listener *SceneNode*, a master volume, and a couple of OpenAL specifics - one for an OpenAL context, and one for a device, both of which were covered earlier.

```
45 protected:
46    SoundSystem(unsigned int channels = 32);
47    ~SoundSystem(void);
48
49    void        UpdateListener();
50    void        UpdateTemporaryEmitters(float msec);
51
52    void        DetachSources(vector<SoundEmitter*>::iterator from,
53                   vector<SoundEmitter*>::iterator to);
54    void        AttachSources(vector<SoundEmitter*>::iterator from,
55                   vector<SoundEmitter*>::iterator to);
56
57    void        CullNodes();
58    OALSource*  GetSource();
```

SoundSystem.h

Lastly, we have a **static pointer** to an instance of a *SoundSystem* - this is what makes this class a '**Singleton**' - it has one actual instance which will always be returned and operated on.

```
59    Matrix4              listenerTransform;
60    float                masterVolume;
61    ALCcontext*          context;
62    ALCdevice*           device;
63    SceneNode*           listener;
64
65    vector<OALSource*>      sources;
66    vector<SoundEmitter*>   emitters;
67    vector<SoundEmitter*>   frameEmitters;
68
69    static SoundSystem*  instance;
70 };
```

SoundSystem.h

Class file

As the **Singleton** pattern used for our *SoundSystem* means we have a **static** member variable, we must define it, so the first thing we do after including the header file, on line 3, is to define it as defaulting to NULL - the *Initialise* static function checks for NULL to ensure only one *SoundSystem* is created.

In our **constructor**, we start by outputting a list of OpenAL compatible devices (line 12). It is possible to select a specific device from this list if we want, but for the purposes of this tutorial we're just going to let OpenAL decide what the best device for our OpenAL context is - passing a NULL value to the OpenAL function *alcOpenDevice* function will make OpenAL choose for us. On line 21 we output the name of the device OpenAL chose, and then, on line 23, we initialise a new OpenAL context on the device, and then make it the currently active context (we could have multiple contexts if we liked, but only one will be processed at a time).

On line 26 we tell OpenAL which *distance model* to use for its volume calculations - we can have sounds that get exponentially quieter as the listener moves away from them, or, as we're going to use, linearly quieter with distance. We're also going to clamp the maximum volume of each sound source, so we don't get too large a disparity in volume as we move towards a source.

```
1  #include "SoundSystem.h"
2
3  SoundSystem* SoundSystem::instance = NULL;
4
5  SoundSystem::SoundSystem(unsigned int channels) {
6     listener       = NULL;
7     masterVolume   = 1.0f;
8
9     cout << "Creating SoundSystem!" << endl;
10
11    cout << "Found the following devices: "
12         << alcGetString(NULL,ALC_DEVICE_SPECIFIER) << endl;
13
14    device = alcOpenDevice(NULL); //Open the 'best' device
15
16    if(!device) {
17       cout << "SoundSystem creation failed! No valid device!" << endl;
18       return;
19    }
```

SoundSystem.cpp

```
20    cout << "SoundSystem created with device: "
21         << alcGetString(device,ALC_DEVICE_SPECIFIER) << endl;
22
23    context = alcCreateContext(device,NULL);
24    alcMakeContextCurrent(context);
25
26    alDistanceModel(AL_LINEAR_DISTANCE_CLAMPED);
```

SoundSystem.cpp

That's the basics of OpenAL all set up - not much to it! We must do one last thing though, and generate an appropriate number of OpenAL sources to attach to the *SoundEmitters* in our game. OpenAL doesn't currently have any easy method of determining how many sources it has available, so all we can do is keep asking it to give us new sources until we either a) run out of available sources, or b) hit our maximum number of channels. To generate a new OpenAL source, we simply use the *alGenSources* function, passing it a **reference** to an **unsigned int** - just like with buffers, and just like with OpenGL. If the operation completed successfully, we can add the name to a new *OALSource*,

and put it in our sources vector. If it failed, determined by the return value of the debug function *alGetError*, we break out of the generation loop.

```
27    for(unsigned int i = 0; i < channels; ++i) {
28       ALuint source;
29
30       alGenSources(1,&source);
31       ALenum error = alGetError();
32
33       if(error == AL_NO_ERROR)   {
34          sources.push_back(new OALSource(source));
35       }
36       else{
37          break;
38       }
39    }
40    cout << "SoundSystem has " << sources.size()
41         << " channels available!"  << endl;
42 }
```

SoundSystem.cpp

To destroy our *SoundSystem*, we tell OpenAL not to use its context any more, then delete all of our *OALSources* and the sources they represent, before finally destroying our context and closing the OpenAL device.

```
43 SoundSystem::~SoundSystem(void)  {
44    alcMakeContextCurrent(NULL);
45    for(vector<OALSource*>::iterator i = sources.begin();
46        i != sources.end(); ++i) {
47       alDeleteSources(1, &(*i)->source);
48       delete (*i);
49    }
50    alcDestroyContext(context);
51    alcCloseDevice(device);
52 }
```

SoundSystem.cpp

OpenAL has its own internal master volume, which we can set with a positive **float**. We're going to sanity check our value to clamp it between 0.0 and 1.0 for safety, using a combination of the *max* and *min* functions. Once the volume has been clamped, it can be sent to OpenAL using the OpenAL function *alListenerf*, which as you might expect, is used to control OpenAL state in relation to the listener object.

```
53 void     SoundSystem::SetMasterVolume(float value) {
54    value = max(0.0f, value);
55    value = min(1.0f, value);
56    masterVolume = value;
57    alListenerf(AL_GAIN,masterVolume);
58 }
```

SoundSystem.cpp

Every time we update the *SoundSystem*, we need to update the listener position and orientation, taken from the world-space transformation matrix of the *SceneNode* that has been set as the listener object. Both the position and orientation are set using the *alListenerfv* function, which takes a **pointer** to some **float** data - a *Vector3* in the case of position, and two *Vector3s* for the orientation. Orientation is determined via an up direction and a forward direction, both of which we can get

from the the transformation matrix directly - the y ('up') and z ('forward') axis rotations, respectively.

```
59 void  SoundSystem::UpdateListener() {
60    if(listener)   {
61       Vector3 worldPos = listenerTransform.GetPositionVector();
62
63       Vector3 dirup[2];
64       //forward
65       dirup[0].x = -listenerTransform.values[2];
66       dirup[0].y = -listenerTransform.values[6];
67       dirup[0].z = -listenerTransform.values[10];
68       //Up
69       dirup[1].x = listenerTransform.values[1];
70       dirup[1].y = listenerTransform.values[5];
71       dirup[1].z = listenerTransform.values[9];
72
73       alListenerfv(AL_POSITION,(float*)&worldPos);
74       alListenerfv(AL_ORIENTATION,(float*)&dirup);
75    }
76 }
```

SoundSystem.cpp

Now for the *Update* function, which should be called by an application's main loop, passing the number of milliseconds that have passed. Every update, our *SoundSystem* must do the following:

1) Update the listener object
2) Update all sound emitters sound states
3) Cull nodes that won't be heard, or who don't have sounds etc
4) Sort the remaining nodes by their priority
5) Detach sources from nodes that are too low priority when there's not enough sources to go around
6) Attach sound sources to high priority nodes that don't have sources
7) Clear the *frameEmitters* vector, ready for the next frame.

This process will ensure that the most important sounds in your game world are played, at the expense of temporarily losing low priority sounds if there are not enough channels to go around. Even if a *SoundEmitter* is to be 'culled' this frame (due to being too far away, or whatever other reason needed), it should still be updated via its *Update* function - it may get a sound source next frame, and so it must always accurately reflect its sound playback state. Emitters that *aren't* culled are instead placed in a *frameEmitters* vector, which is then sorted if necessary, and OpenAL sources attached to them.

```
77 void     SoundSystem::Update(float msec) {
78    UpdateListener();
79
80    for(vector<SoundEmitter*>::iterator i = emitters.begin();
81        i != emitters.end(); ++i) {
82       frameEmitters.push_back((*i));
83       (*i)->Update(msec);
84    }
85
86    CullNodes();
```

SoundSystem.cpp

```
87      if(frameEmitters.size() > sources.size()) {
88          std::sort(frameEmitters.begin(), frameEmitters.end(),
89                    SoundEmitter::CompareNodesByPriority);
90
91          DetachSources(frameEmitters.begin()+(sources.size()+1),
92                        frameEmitters.end());
93          AttachSources(frameEmitters.begin(),
94                        frameEmitters.begin() + sources.size());
95      }
96      else{
97          AttachSources(frameEmitters.begin(), frameEmitters.end());
98      }
99
100     frameEmitters.clear();
101 }
```

SoundSystem.cpp

To correctly allocate our limited OpenAL sources, we must cull *SoundEmitters* that definitely won't need one this update - those that are too far away, aren't currently playing a sound, or who have completed playing their sound. We can easily work out if a node is too far away by calculating its distance from the listener by taking the *Length* of the direction vector between them - if it is greater than the *SoundEmitter's* radius, it's too far away and should be culled. As we iterate through the list, nodes that are culled are removed from the current frame's vector of nodes, and we make sure it doesn't have a source, by calling *DetachSource*.

```
102 void  SoundSystem::CullNodes() {
103     for(vector<SoundEmitter*>::iterator i = frameEmitters.begin();
104         i != frameEmitters.end(); ) {
105         SoundEmitter*e = (*i);
106
107         float length;
108
109         if(e->target) {
110             length = (listenerTransform.GetPositionVector() -
111             e->target->GetWorldTransform().GetPositionVector()).Length();
112         }
113         else{
114             length = (listenerTransform.GetPositionVector() -
115             e->position).Length();
116         }
117
118         if(length > e->GetRadius() || !e->GetSound() ||
119                     e->GetTimeLeft() < 0) {
120             e->DetachSource();
121             i = frameEmitters.erase(i);
122         }
123         else{
124             ++i;
125         }
126     }
127 }
```

SoundSystem.cpp

To easily attach and detach sound sources from the nodes in our vector, we have two helper functions, that both take **iterators**. Using these, we can call a function on only a subset of the vector - in this case, we want to sort the vector by its priority, and detach sources from *SoundEmitters* at the end of the vector, and attach them to those at the front. For example, if we have 8 sound

sources, and 10 *SoundEmitters* ordered from highest to lowest priority wanting sources, we make sure that *SoundEmitters* 9 and 10 have any source they are using detached from them, to guarantee that *SoundEmitters* 1 to 8 get sources. When attaching sources to nodes, there's no point attaching a new source if a *SoundEmitter* already has one attached (from a previous frame, for example), so on line 131, we check for that before giving a *SoundEmitter* a new source, obtained via another helper function, *GetSource*.

```
128 void  SoundSystem::DetachSources(vector<SoundEmitter*>::iterator from,
129                                   vector<SoundEmitter*>::iterator to) {
130    for(vector<SoundEmitter*>::iterator i = from; i != to; ++i) {
131       (*i)->DetachSource();
132    }
133 }
134
135 void  SoundSystem::AttachSources(vector<SoundEmitter*>::iterator from,
136                                   vector<SoundEmitter*>::iterator to) {
137    for(vector<SoundEmitter*>::iterator i = from; i != to; ++i) {
138       if(!(*i)->GetSource()) {
139          (*i)->AttachSource(GetSource());
140       }
141    }
142 }
```

SoundSystem.cpp

The last function we need is used to obtain a source - we simply iterate over the vector of *OALSources* we filled up in the **constructor**, and find the first that is not marked as in use, mark it so, and return it.

```
143 OALSource*  SoundSystem::GetSource() {
144    for(vector<OALSource*>::iterator i = sources.begin();
145        i != sources.end(); ++i) {
146       OALSource*s = *i;
147       if(!s->inUse) {
148          return s;
149       }
150    }
151    return NULL;
152 }
```

SoundSystem.cpp

SoundSystem Initialisation

To use our new sound playback capability, we simply have to call its *Initialise* function somewhere in the main function:

```
1 SoundSystem::Initialise();
```

main.cpp

When we're done with it, we should call *Destroy* - but don't forget to also call *DeleteSounds* on the *SoundManager* class we defined earlier!

```
2 SoundManager::DeleteSounds();
3 SoundSystem::Destroy();
```

main.cpp

In order to keep our *SoundSystem* updated, we need to call its *Update* function somewhere - the *UpdateCore* function of the *GameClass* seems a good place!

```
4 SoundSystem::GetSoundSystem()->Update( msec );
5 //SoundSystem::GetSoundSystem()->Update((1000.0f / (float)RENDER_HZ));
```

main.cpp

Finally, to test your new sound system, you might want to make a function inside the *MyGame* class that will place a *SoundEnity* (a *GameEntity* with a *SoundEmitter* member variable) somewhere in your gameworld, maybe with a sphere mesh so you can see where it's been placed (handy for debugging!):

```
1  GameEntity* MyGame::BuildSoundEntity() {
2     float size = 300 + (rand()%300);
3
4     SceneNode* s = new SceneNode(sphere);
5
6     s->SetModelScale(Vector3(size,size,size));
7     s->SetBoundingRadius(size);
8     s->SetColour(Vector4(1,1,1,0.6)); //Make node transparent, too
9  //Pick a sound from a list of filenames
10    Sound*snd = SoundManager::GetSound(soundNames[rand()%NUM_SOUNDS]);
11 //and set it on a new SoundEntity
12    SoundEntity*g = new SoundEntity(snd, s, NULL);
13    //Randomly place it in the world somewhere
14    Vector3 randpos = Vector3((rand()%10)*256, 0.0f,(rand()%10)*256);
15    randpos -= Vector3((rand()%10)*256, 0.0f,(rand()%10)*256);
16
17    s->SetTransform(Matrix4::Translation(randpos));
18    //Connect it to all of our core systems
19    g->ConnectToSystems();
20
21    return g;
22 }
```

MyGame.cpp

Tutorial Summary

In this workshop tutorial, you've learned how sound is encoded as PCM data, and how to load this data from a WAV file. You've learned a bit about how the OpenAL API works, and what goes into making a working sound system, including how to deal with a limited number of sound output resources. From this basic sound system, you should be able to create aurally rich environments for your games, and enhance their immersion. To round off this audio workshop, next time we'll be looking at a few extra features to enhance your sound system - triggering the playing of sounds both in 3D space and straight to the audio hardware, and streaming audio data from a file, which will allow effective playing of long samples such as music.

Further Work

1) OpenAL allows a source to have a pitch set on it, by using *alSourcef* and a parameter type of **AL_PITCH**. Try giving each SoundEmitter a varying pitch to see how the pitch effects a sound. If a pitch of 0.5 is used, how much longer will a sound take to play? What effect will this have on the *timeLeft* variable?

2) So far, our *SoundEmitters* have just had a *radius*, meaning sound is emitted equally in all directions. However, OpenAL also allows a cone of sound, with differing volumes inside and outside of the cone. Investigate the **AL_CONE_OUTER_ANGLE** parameter. How will you determine the direction a *SoundEmitter* is facing in, to set the direction of the cone?

3) As well as pitch, OpenAL supports the calculation of the *doppler effect* - how the pitch of a sound changes in relation to the relative velocities of the listener and sound source. Think about the siren on a police car or ambulance, it's pitch appears to change as the vehicle drives past. Investigate the *velocity* component of a source and the listener, and the usage of the functions *alDopplerFactor* and *alSpeedOfSound*. If doppler effects pitch, will it effect playback rate?

4) Try setting the optional parameter of the *Sound::Initialise* function in the main file to a low value like 4 - you should hear that some sounds get disabled, as there aren't enough channels to play them. Try changing the **for** loop to add in lots of new *SoundEmitters*! Once you've got your head around the sound system and how it works, try combining it with your coursework projects to make the in game nodes have sounds attached to them.

Credits

All of the sounds provided in this workshop are courtesy of the contributors to **Freesound.org** - a collaborative database of Creative Commons Licensed sounds. The original authors of the sounds are as follows:

S: arpegthing.wav by Votives
http://www.freesound.org/people/Votives/sounds/139320/

S: The Road - A Melody From The Past.mp3 by Vosvoy
http://www.freesound.org/people/Vosvoy/sounds/142010/

S: Sheeheep.wav by HerbertBoland
http://www.freesound.org/people/HerbertBoland/sounds/75190/

S: war.wav by Syna-Max
http://www.freesound.org/people/Syna-Max/sounds/56900/

S: steps-on-stone01.ogg by Erdie
http://www.freesound.org/people/Erdie/sounds/41579/

S: LaserRocket2.wav by EcoDTR
http://www.freesound.org/people/EcoDTR/sounds/36847/

S: canon.aif by man
http://www.freesound.org/people/man/sounds/14615/