8

Audio in Games Development

In chapter, the following recipes will be covered:

1. 1 Installing FMOD
2. 2 Adding background music
3. 3 Adding sound effects
4. 4 Creating a sound-effect manager
5. 5 Dealing with multiple sound file names
6. ~~6 Dealing with playing multiple sound effects at the same time~~

# Introduction

One of the most important aspects in games development. However also strangely one of the neglected or one of the most under-rated sections of the game development is audio programming. Just to understand the impact of audio in games, try playing a game like Counter-Strike or Quake with sounds and then try playing the games without sound. It creates a totally huge impact. Audio programming if not done correctly, can lead to games crashing and whole other problems

Hence it is very important to learn the correct way to learn audio programming. Most engines will have an in-built sound component. For others, we need to add the audio component. In this chapter, we will have a look at one of the most popular sound engine. We will also have a look at how to integrate SDL into our C++ code to play audio and sound effects.

# Installing FMOD

1. The first thing to get started is to install FMOD. This is one of the most popular audio engines and used in almost all of the modern game engines. It can also be added to any game engine of your choice.

## Getting ready

To step through this recipe, you will need a machine running Windows.

## How to do it...

In this recipe we will see the different types of source control available to us.

* Go to the following URL <http://www.fmod.org/>
* <http://www.fmod.org/download/>
* There is one authoring tool to edit the audio files.
* However, we should be downloading the FMOD Studio Programmer’s API and Low Level Programmer API.
* It also has plugins for all modern engine like Cocos 2DX, Unreal Engine and Unity3D.

## How it works...

FMOD is a low level API. So it provides call-backs which help us to use the interface of FMOD to play sounds, pause sounds and do a whole lot other things. Because we have the source files, we can build the libraries and also use it in our own engine. FMOD also provides API for Unity3D, which means that the code is also exposed to C# which makes it easier to work with in Unity3D.

# Adding background music

Any game will be incomplete if it did not have any background music. So it becomes very important that we integrate a way to play music in our C++ engine. There are various ways to do that. We are going to use SDL to play music in our game.

## Getting ready

You need a Windows machine and a working copy of Visual Studio. SDL library is also required.

## How to do it...

In this recipe we will find out how easy it is to play a background music.

* Add a source file called Source.cpp

Code Snippet

IAudioDevice\* device = new SDLAudioDevice();

IAudioContext\* context = new SDLAudioContext();

IAudioData\* data = device->CreateAudioFromFile(FILE\_PATH);

class IAudioContext

{

public:

virtual ~IAudioContext() {}

virtual void PlayAudio(AudioObject& ao) = 0;

virtual void PauseAudio(AudioObject& ao) = 0;

virtual void StopAudio(AudioObject& ao) = 0;

private:

};

## How it works...

In this example we are playing a background music for our game. We need to create an interface providing as a wrapper to the existing SDL audio library. Interfaces are also good at providing skeleton architecture which base class can derive from in the future. We require a SDLAudioDevice which is the main handler object to play the music. On top of that we create a pointer to a Audiodata object which creates audio from the file path provided. The device handler object has an inbuilt function called CreateAudioFromFile to help us with the process. Finally, we have an audio context class which has functions to play, pause and stop the audio. Each of the function takes an audio object as a reference which is used to perform the action on our audio file.

# Adding sound effects

Sound effects are a neat way of adding some sense of tension or achievement in the game. Playing, pausing and stopping sound effects work in the same technique that we use for background music which we have seen in the recipe above. However, we can add some variety to the sound files by controlling their position, volume and pitch.

## Getting ready

1. You need to have a working Windows machine.

## How to do it...

* Add a source file called Source.cpp

Code Snippet

struct SampleInfo

{

double volume;

double pitch;

};

SampleInfo info;

info.volume = 1.0;

info.pitch = 0.7298149802137;

AudioObject sound(info, data);

sound.SetPos(0.0);

## How it works...

In this example as we are only looking at the part which involves modifying the pitch, volume and position of the sound file. These 3 mentioned things could be considered as attributes of the sound file. There could be other attributes as well. So the first thing to do would be to create a structure. The structure could be used to store all the attributes of the sound. All we need to do is populate the structure with values as we want them. Finally, we create an audio object and pass in the struct as one of the parameters of the object. The constructor would then initialise the sound to have these properties. Because we attached the properties to the object, it means we can also manipulate them at run-time and lower the volume dynamically if needed. Also the pitch and other properties could also be manipulated in the same way.

# Creating a sound Effect Manager

Not one of the best practices out there, however one of the most common method is to create a manager class for handling the audio. The manager class should ensure that there is only one audio component in the whole game which controls which sound is to be played, paused, looped and so on. Although there are other ways of writing the manager class, this is the most standard practice.

## Getting ready

For this recipe, you will need a Windows machine and Visual Studio

## How to do it...

1. In this recipe we will find out how easy it is to add source control.
2. #pragma once
3. #include <iostream>
4. #include "../lib/SDL2/include/SDL2/SDL.h"
5. #include "iaudiodevice.hpp"
6. #include "iaudiocontext.hpp"
7. #include "audioobject.hpp"
8. #include "sdl/sdlaudiodevice.hpp"
9. #include "sdl/sdlaudiocontext.hpp"
10. #define FILE\_PATH "./res/audio/testClip.wav"
11. class GlobalAudioClass
12. {
13. private:
14. AudioObject\* \_audObj;
15. IAudioDevice\* device = new SDLAudioDevice();
16. IAudioContext\* context = new SDLAudioContext();
17. IAudioData\* data = device->CreateAudioFromFile(FILE\_PATH);
18. SampleInfo info;
19. static GlobalAudioClass \*s\_instance;
20. GlobalAudioClass()
21. {
22. info.volume = 1.0;
23. info.pitch = 0.7298149802137;
24. \_audObj = new AudioObject(info,data);
25. }
26. ~GlobalAudioClass()
27. {
28. //Delete all the pointers here
29. }
30. public:
31. AudioObject\* get\_value()
32. {
33. return \_audObj;
34. }
35. void set\_value(AudioObject\* obj)
36. {
37. \_audObj = obj;
38. }
39. static GlobalAudioClass \*instance()
40. {
41. if (!s\_instance)
42. s\_instance = new GlobalAudioClass();
43. return s\_instance;
44. }
45. };
46. // Allocating and initializing GlobalAudioClass's
47. // static data member. The pointer is being
48. // allocated - not the object inself.
49. GlobalAudioClass \*GlobalAudioClass::s\_instance = 0;

## How it works...

1. In this example, we have written a singleton class to implement the audio manager. The singleton class has all the required sdl headers and other device and context objects required to play the sounds. All those are private, so that it cannot be accessed from other classes. We also make a static pointer to the class and make the constructor private as well. If we need an instance of this audio manager, we have to use the static GlobalAudioClass \*instance() function. The function automatically checks if there is an already create instance, then it returns that instance else it creates a new one. Hence, only one instance of the manager class exist at all times. We can also use the manager to set and get data to the sound file, for example setting the path of the sound file.

# Dealing with multiple sound file names

In games, there will not be one file but multiple sound files that we will have to deal with. Each will have a different name, type and location. So it is not a wise move to define all of them separately. It will work but it will be very messy coding if we have over 20 sound effects in our game. Hence there is a need for a slight improvement to the code.

## Getting ready

1. For this recipe, you will need a Windows machine and an installed version of a SVN client. A versioned project is also necessary.

## How to do it...

In this recipe, you will see how easy it is to deal with multiple sound file names

1. Add a source file called Source.cpp

Code Snippet

1. #pragma once
2. #include <string>
3. using namespace std;
4. class AudioDataHandler
5. {
6. public:
7. AudioDataHandler();
8. ~AudioDataHandler();
9. string GetAudio(int data) // Set one of the enum values here from the driver program
10. {
11. return Files[data];
12. }
13. enum AUDIO
14. {
15. NONE=0,
16. BACKGROUND,
17. BATTLE,
18. UI
19. };
20. private:
21. string Files[] =
22. {
23. "",
24. "Hello.wav",
25. "Battlenn.wav",
26. "Click.wav"
27. }

30. };
31. int main()
32. {
33. AudioDataHandler \_audioData;
34. cout<<\_audioData.GetAudio(AudioDataHandler::BACKGROUND);
35. }

## How it works...

In this example, we have created an audio data handler class. The class has a enum which stores all the logical names of the sounds, for example battle\_music, background\_music and so on. We also have a string array which stores the actual names of the sound files. The order is important and it has to match the order in which we have written the enum. Now that this is created, we can create an object if this class and set and get the audio file name. This enum is stored as integers and starts at 0 by default, the names would serve as index for the string array. So Files[AudioDataHandler::Background] is actually Files[1] which is “Hello.wav” and so the correct file will be downloaded. This is a very neat way of organising the audio data files.