

# Tutorial 2: Load and Play Sounds

by Ingmar Rötzer

## Content

So far, we have only initialized DirectX Audio and now the time has come to get your speakers ready. In this tutorial, loading and playing of 2D-sounds will be shown. The new CSound class is introduced that represents a single sound object and contains methods to manipulate its behaviour. By creating several objects of that type it is possible to use different sounds simultaneously.

The csound.\* files are new and contain the CSound class. Many of the other project files have changed. The following parts describe the new sourcecode in detail.

## main.h

Here we just add the sound class header and the extern sound object. The title has to be changed as well.

```
//includes
#include<dmusici.h>
#include<windows.h>

#include"cappplication.h"
#include"csound.h" //sound class header (NEW)
#include"caudio.h"

//constants
#define TITLE "DXAudio Tut 02: Load and Play Sounds"

//globals
extern bool g_bRunning;
extern CApplication g_App;
extern CAudio g_Audio;
extern CSound g_Sound1; //extern sound object (NEW)
```

## main.cpp

The sound object is created here. It is global so that it is accessible from the WindowProcedure() (the reason is explained later on). The name of the sound file is passed to the object.

```
//globals
bool g_bRunning = true;
CApplication g_App;
CAudio g_Audio;
CSound g_Sound1("Trumpet1.wav"); //global sound object (NEW)
```

## caudio.h

A new member function is added that changes the master volume of the application. The two get functions are necessary because the CSound class needs the performance and the loader objects. These functions are declared inline as they are very short and calling them might be faster that way.

```
class CAudio
```

```

{
public:
    CAudio();
    ~CAudio();

    void InitAudio();

    void SetSearchDirectory(const char*);
    void SetMasterVolume(long); //vol. (NEW)

    void KillAudio();

    inline IDirectMusicPerformance8* GetPerformance()
    {
        return m_pPerformance;
    } //get pointer to performance (NEW)

    inline IDirectMusicLoader8* GetLoader()
    {
        return m_pLoader;
    } //get pointer to loader (NEW)

private:
    IDirectMusicPerformance8* m_pPerformance;
    IDirectMusicLoader8* m_pLoader;
};

```

## caudio.cpp

Changing the master volume requires calling `SetGlobalParam()`. Parameters are the type of the object to be changed, the new value and the size of the data type. The volume is an amplification factor applied to the default volume in hundredths of a decibel. The useful range is +10db to -100db.

```

void CAudio::SetMasterVolume(long lVolume)
{
    m_pPerformance->SetGlobalParam(GUID_PerfMasterVolume,
                                   &lVolume,
                                   sizeof(long)); //set master volume
} //SetMasterVolume

```

Before the performance and loader objects can be released by `KillAudio()` all playing music has to be stopped. This is done through the `Stop()` function passing zero everywhere.

```

void CAudio::KillAudio()
{
    m_pPerformance->Stop(NULL,NULL,0,0); //stop all playing music (NEW)

    m_pLoader->Release();

    m_pPerformance->CloseDown();
    m_pPerformance->Release();

    CoUninitialize();
} //KillAudio

```

## csound.h

The sound class holds one sound represented by the `IDirectMusicSegment8` data type. Member functions manage playback and change the behaviour of the soundfile. Pointers to the performance

and loader objects are taken from the `CAudio` class because they are essential for loading and playback of the sound. The standard constructor is responsible for loading and the destructor unloads the sound from memory.

```
class CSound
{
public:
    CSound(char*);
    ~CSound();

    void Play();
    void Stop();

    void SetRepeats(DWORD);
    bool IsPlaying();

private:
    IDirectMusicPerformance8* m_pPerformance;
    IDirectMusicLoader8* m_pLoader;
    IDirectMusicSegment8* m_pSegment;
};
```

## csound.cpp

The standard constructor gets the performance and loader objects from the `CAudio` class. Then the segment is created using `CoCreateInstance()`. The filename has to be converted to wide char. After that the loader object is needed to load a sound from a file. Note that you can only use wav or midi files. Before the sound can be played the band data has to be downloaded to the performance.

```
#include "main.h"

CSound::CSound(char* szFile)
{
    WCHAR wszFile[MAX_PATH];

    m_pSegment = NULL;

    //get performance object from CAudio class
    m_pPerformance = g_Audio.GetPerformance();

    //get loader object from CAudio class
    m_pLoader = g_Audio.GetLoader();

    //create segment
    CoCreateInstance(CLSID_DirectMusicSegment,
        NULL,
        CLSCTX_INPROC,
        IID_IDirectMusicSegment8,
        (void**)&m_pSegment);

    //convert filename to wide char
    MultiByteToWideChar(CP_ACP, 0, szFile, -1, wszFile, MAX_PATH);

    //load sound from file
    m_pLoader->LoadObjectFromFile(CLSID_DirectMusicSegment,
        IID_IDirectMusicSegment8,
        wszFile,
        (void**)&m_pSegment);

    //download band
    m_pSegment->Download(m_pPerformance);
} //CSound
```

The destructor unloads the data from the performance so that the segment can be released from memory.

```
CSound::~CSound()
{
    m_pSegment->Unload(m_pPerformance);
    m_pSegment->Release();
} //~CSound
```

Here the performance is used to play the segment. `DMUS_SEGF_SECONDARY` means that the sound is played as a secondary segment. It is also possible to play a segment as the primary segment but since only one of these can exist at the same time, the former primary segment would stop playing. However, many secondary segments can be used simultaneously. This sound is played on the default audio path that was created by the performance. `PlaySegmentEx()` offers a wide range of options for playback (e.g scheduling sounds or stopping other segments when this one starts) but explaining them all would take too much space. Look up the possible parameters in the DirectX SDK help if you wish.

```
void CSound::Play()
{
    m_pPerformance->PlaySegmentEx(m_pSegment,
                                  0,
                                  NULL,
                                  DMUS_SEGF_SECONDARY,
                                  0,
                                  0,
                                  NULL,
                                  NULL);
} //Play
```

This function simply stops playback of the specified segment. The second parameter takes the time at which to stop the sound. A value of zero means immediately.

```
void CSound::Stop()
{
    m_pPerformance->StopEx(m_pSegment, 0, 0);
} //Stop
```

The number of repetitions of the sound is set with this function. Passing `DMUS_SEG_REPEAT_INFINITE` repeats it forever.

```
void CSound::SetRepeats(DWORD dwRepeats)
{
    m_pSegment->SetRepeats(dwRepeats);
} //SetRepeats
```

This function asks the performance if the sound is currently playing and returns the result.

```
bool CSound::IsPlaying()
{
    return (m_pPerformance->IsPlaying(m_pSegment, NULL) == S_OK);
} //IsPlaying
```

## winproc.cpp

Now it is time for testing. I have included a sample sound in the `/sound` directory but you may use

any other way or midi file. The message loop responds to keyboard input and calls the appropriate function to manipulate the sound.

```

LRESULT CALLBACK WindowProcedure(HWND hWindow,
                                UINT uMessage,
                                WPARAM wParam,
                                LPARAM lParam)
{
    static long lVolume = 500;    //set the amplification factor to 5db

    switch(uMessage)
    {
    case WM_CHAR:
        {
        switch(wParam)
        {
        case 'p':
            {
            g_Sound1.Play();    //Pressing 'P' plays the sound
            break;
            }
        case 's':
            {
            g_Sound1.Stop();    //Pressing 'S' stops it
            break;
            }
        case '+':
            {
            lVolume += 50;
            g_Audio.SetMasterVolume(lVolume); //Pressing '+' increases master vol.
            break;
            }
        case '-':
            {
            lVolume -= 50;
            g_Audio.SetMasterVolume(lVolume); //Pressing '-' decreases master vol.
            break;
            }
        case '1':
            {
            g_Sound1.SetRepeats(1);    //Pressing '1' means no repetitions
            break;
            }
        case '2':
            {
            g_Sound1.SetRepeats(2);    //Pressing '2' repeats the sound
            break;
            }
        case 't':
            {
            if(g_Sound1.IsPlaying())    //Pressing 't' tests playback
                MessageBox(hWindow,
                    "Sound is currently playing!",
                    "Testing",
                    MB_OK | MB_ICONINFORMATION);
            else
                MessageBox(hWindow,
                    "Sound is not playing!",
                    "Testing",
                    MB_OK | MB_ICONINFORMATION);
            break;
            }
        }
        }
    return false;
}

return DefWindowProc(hWindow,uMessage,wParam,lParam);

```

```
}//WindowProcedure
```

Now you are able to use stereo sound in your application. The introduction to 3D-sound will be the topic of the next tutorial.