

Tutorial 3: 3D-Sound

by Ingmar Rötzer

Content

Now the real interesting part begins. This tutorial covers the introduction to 3D-Sound. Getting this to work is far easier than one might expect since we have already done most of the preparation in the preceding two tutorials. A new class called `CSound3D` is added to the project and used instead of the `CSound` class. Much of the code is similar though.

This tutorial is based upon "DirectInput Tutorial 02: Using The Keyboard". There, a pyramid is shown which can be moved in all three dimensions using `DirectInput`. Our sample sound is located just inside this pyramid, hence providing both optical and acoustical feedback. The `CAudio` and `CSound3D` classes are added to this project and I focus only on these. If you have problems with the `Direct3D` and `DirectInput` code see `DirectInput Tutorial 02` and `Direct3D Tutorial 04` for explanation or ignore it since it is not important here.

Small changes to the original `CSound` class are needed to add 3D-Sound support. Most important is the audio path. Formerly, 2D-sounds were played on the default audio path that was created with the performance object. Now we need a 3D capable audio path. Each audio path uses a `DirectSound3DBuffer` to change the properties of the channel (e.g. position). All sounds played on the same audio path have the same properties set via the sound buffer. Therefore, each independent sound needs its own audio path. The constructor of the `CSound3D` class lets the user choose to create a new audio path or use an existing one. The following parts describe the changes in detail.

`csound3d.h`

This class provides the methods and interfaces to manipulate the sound object.

The `SAudioPathRef` structure is used to keep track of the number of sounds that belong to one audio path. A pointer to each audio path and its sound buffer is stored there along with the number of sounds using that audio path. If this reference counter reaches zero, the audio path can be released.

```
#define AUDIO_PATH_LIMIT 64 //limit number of audio paths to 64 (NEW)

struct SAudioPathRef
{
    IDirectMusicAudioPath8* pAudioPath;
    IDirectSound3DBuffer8* p3DBuffer;
    short sRefCount;

    SAudioPathRef()
    {
        pAudioPath = NULL;
        p3DBuffer = NULL;
        sRefCount = 0;
    }
};
```

The class has been extended by some elements. If the constructor receives a pointer to an existing audio path, it is used also by this sound. Otherwise, a new audio path will be created. It is possible to change the volume of this audio path (not the master volume). The static array of `SAudioPathRef` structures stores all newly created audio paths and their usage count.

```

class CSound3D
{
public:
                                CSound3D(char*, IDirectMusicAudioPath8*);
                                ~CSound3D();

    void                        Play();
    void                        Stop();

    void                        ActivateAudioPath(bool); //activate (NEW)
    void                        SetVolume(long, DWORD); //vol. adjust (NEW)
    void                        SetRepeats(DWORD);
    bool                        IsPlaying();

inline IDirectMusicAudioPath8* GetAudioPath()
    {
        return m_pAudioPath;
    } //get pointer to audio path (NEW)

inline IDirectSound3DBuffer8*  Get3DBuffer()
    {
        return m_p3DBuffer;
    } //get pointer to sound buffer (NEW)

private:
    static SAudioPathRef      m_aAudioPathRef[AUDIO_PATH_LIMIT];

    IDirectMusicPerformance8* m_pPerformance;
    IDirectMusicLoader8*      m_pLoader;
    IDirectMusicSegment8*     m_pSegment;
    IDirectMusicAudioPath8*   m_pAudioPath; //ref. to audio path (NEW)
    IDirectSound3DBuffer8*    m_p3DBuffer; //ref. to sound buffer (NEW)
};

```

csound3d.cpp

Let's have a look at the implementation of the class. The following bunch of code is just copied from CSound. It creates a segment and loads the sound from a file.

```

CSound3D::CSound3D(char* szFile, IDirectMusicAudioPath8* pAudioPath)
{
    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 file
    m_pLoader->LoadObjectFromFile(CLSID_DirectMusicSegment,
                                IID_IDirectMusicSegment8,

```



```

m_pSegment->Unload(m_pPerformance);
m_pSegment->Release();

for(unsigned i = 0;i < AUDIO_PATH_LIMIT;++i) //search audio path
{
    if(m_aAudioPathRef[i].pAudioPath == m_pAudioPath)
    {
        m_aAudioPathRef[i].sRefCounter--; //decrement reference counter

        //release audio path when last reference destroyed
        if(m_aAudioPathRef[i].sRefCounter == 0)
        {
            m_p3DBuffer->Release();
            m_pAudioPath->Release();

            m_aAudioPathRef[i].pAudioPath = NULL;
            m_aAudioPathRef[i].p3DBuffer = NULL;
            m_aAudioPathRef[i].sRefCounter = 0;
        }
    }
    break;
}
}
} //~CSound3D

```

The only change here has been made to the last parameter of the `Play` function. Instead of `NULL` (which means default audio path) the custom 3D audio path is supplied on which the sound is then played.

```

void CSound3D::Play()
{
    m_pPerformance->PlaySegmentEx(m_pSegment,
                                  0,
                                  NULL,
                                  DMUS_SEGF_SECONDARY,
                                  0,
                                  0,
                                  NULL,
                                  m_pAudioPath);
} //Play

```

I think this part needs no explanation. If disabled, no sounds can be played on this audio path anymore.

```

void CSound3D::ActivateAudioPath(bool bActivate)
{
    m_pAudioPath->Activate(bActivate);
} //ActivateAudioPath

```

The volume can be changed with this function. Duration is the time period over which the change takes place (e.g. enables fade-out effects).

```

void CSound3D::SetVolume(long lVolume,DWORD dwDuration)
{
    m_pAudioPath->SetVolume(lVolume,dwDuration);
} //SetVolume

```

The functions `Stop()`, `SetRepeats()` and `IsPlaying()` remain unchanged and are not listed again.

main.h

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

```
//includes
#include<windows.h>
#include<commctrl.h>
#include<d3d9.h>
#include<d3dx9.h>
#include<dinput.h>
#include<fstream>
#include<dmusic.h>           //DirectXAudio header

#include"cappplication.h"
#include"cinput.h"
#include"d3ddefs.h"
#include"caudio.h"           //audio class header
#include"csound3d.h"         //3D-sound class header (NEW)

//constants
#define TITLE                 "DXAudio Tut 03: 3D-Sound"

//globals
extern CApplication          g_App;
extern CInput*               g_pInput;
extern CAudio                g_Audio; //extern audio object
```

main.cpp

In this part, the new CSound3D class is used and two sample sounds are played. At first, a global object of the audio class is created.

```
//globals
CApplication    g_App;
CInput*         g_pInput;
CAudio          g_Audio; //global audio object
```

Two objects of CSound3D are created. The second one gets the audio path created by the first object. Therefore, changing the properties of the second sound (actually that of the audio path) changes the first sound as well. `SetPosition()` receives the coordinates of the sound in 3D-space. Here, the coordinates of the pyramid are used. `SetMinDistance()` controls the border where the sound does not get any louder and `SetMaxDistance()` sets the line where the sound diminishes. `DS3D_IMMEDIATE` means that the recalculation of the sound properties is done at once, the alternative method will be used in the fourth tutorial. Note that all these calls use the 3D sound buffer which is received through the Get-function and no such methods are implemented in the class itself. This allows full control over all available settings of the sound buffer.

```
CSound3D Sound1("sounds/fan.wav",NULL),
          Sound2("sounds/wind-thunder.wav",Sound1.GetAudioPath());
float fVelocity = 0.0f;
long lVolume = 0;
bool bSound1Playing = true,bSound2Playing = false;

Sound1.Get3DBuffer()->SetPosition(fX,fY,fZ,DS3D_IMMEDIATE);
Sound1.Get3DBuffer()->SetMinDistance(10.0f,DS3D_IMMEDIATE);
Sound1.Get3DBuffer()->SetMaxDistance(100.0f,DS3D_IMMEDIATE);
Sound1.Play();
```

The following code fragments are inside the game loop and updated every frame. It is possible to change the velocity (used for the Doppler effect) and the volume by pressing certain keys. These

parameters are then updated along with the new position.

```
//increase velocity by pressing '*' on numpad
if(g_pInput->KeyPressed(DIK_MULTIPLY)) fVelocity += 1.0f;

//decrease velocity by pressing '/' on numpad
if(g_pInput->KeyPressed(DIK_DIVIDE)) fVelocity -= 1.0f;

//increase volume by pressing '+' on numpad
if(g_pInput->KeyPressed(DIK_ADD)) lVolume += 10;

//decrease volume by pressing '-' on numpad
if(g_pInput->KeyPressed(DIK_SUBTRACT)) lVolume -= 10;

Sound1.Get3DBuffer()->SetPosition(fX,fY,fZ,DS3D_IMMEDIATE);
Sound1.Get3DBuffer()->SetVelocity(fVelocity,0.0f,0.0f,DS3D_IMMEDIATE);
Sound1.SetVolume(lVolume,0);
```

Why does the following part look so complicated? Well, a small problem arises when we want to use `DirectInput` to play a sound. Since the game loop is executed many times per second on high frame rates, pressing a key would call the `Play()` function several times and the sound is played more than once in a short intervall which sounds terrible. Even the `IsPlaying()` function reacts too slowly to be useful here. So I have introduced a `bool` variable for each sound that is set back when `IsPlaying()` can be used again (after some moments).

```
if(g_pInput->KeyPressed(DIK_1)) //press '1' to play sound A
{
    if(!Sound1.IsPlaying() && !bSound1Playing)
    {
        Sound1.Play();
        bSound1Playing = true;
    }
}
if(g_pInput->KeyPressed(DIK_2)) //press '2' to play sound B
{
    if(!Sound2.IsPlaying() && !bSound2Playing)
    {
        Sound2.Play();
        bSound2Playing = true;
    }
}
if(g_pInput->KeyPressed(DIK_S))
{
    Sound1.Stop();
    Sound2.Stop();
    bSound1Playing = false;
    bSound2Playing = false;
}

if(GetTickCount() % 100 == 0) //reset bool variables after 100 ms
{
    bSound1Playing = false;
    bSound2Playing = false;
}
```

Ok, that is all for now. If you have any questions about the tutorials, just send me an email.