

Tutorial 1: Initialization

by Ingmar Rötzer

Content

In the following tutorials I am going to introduce you to DirectXAudio. It starts with the initialization of DirectXAudio and later tutorials will cover loading and playing of 2D-sounds up to managing several 3D-sound sources.

In previous DirectX versions DirectSound and DirectMusic were two discrete components. Since DirectX 8, DirectMusic interfaces present all necessary components for loading and playing all sounds. DirectSound is still there to manipulate sound buffers on the low-level but that is not done here.

This tutorial is based on DirectX Graphics Tutorial 01: Create a Window so that the window created there is used to demonstrate certain audio features. In this tutorial the window remains empty. A new CAudio class is created to initialize and manage the DirectX audio objects. Creating an object of this class will automatically initialize all audio components. Later, other member functions for manipulating audio features will be added. Destroying the object closes down the audio interfaces and unloads them from memory.

The project consists of the files listed below. The caudio.* files are new and contain the CAudio class, the other files are taken from the first D3D tutorial and I have only listed the changes made to them. Do not forget to link *dxguid.lib* to your project.

main.h

The DirectX and audio class headers are added. The title has to be changed. At last the main audio object is declared extern for use in all source code files.

```
//includes
#include<dmusic.h>      //DirectXAudio header (NEW)
#include<windows.h>

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

//constants
#define TITLE          "DXAudio Tut 01: Initialization"

//globals
extern bool            g_bRunning;
extern CApplication   g_App;
extern CAudio          g_Audio; //extern audio object (NEW)
```

main.cpp

Only the audio object has to be added here.

```
//globals
bool            g_bRunning = true;
CApplication   g_App;
CAudio          g_Audio; //global audio object (NEW)
```

I have replaced PeekMessage() by GetMessage() as it halts the application if there are no

messages in the queue and does not block other programs. However, in a game `PeekMessage()` is the right choice so that the application is running continuously and can update the game even if there are no window messages.

```
while(g_bRunning)
{
    if(GetMessage(&Message, NULL, 0, 0)) //process window messages (NEW)
    {
        TranslateMessage(&Message);
        DispatchMessage(&Message);
    }
}
```

winproc.cpp

I have made some small changes to this file but none that change functionality.

```
#include "main.h"

LRESULT CALLBACK WindowProcedure(HWND hWnd,
                                  UINT uMessage,
                                  WPARAM wParam,
                                  LPARAM lParam)
{
    switch(uMessage)
    {
        case WM_KEYDOWN:
        {
            switch(wParam)
            {
                case VK_ESCAPE:
                {
                    DestroyWindow(hWnd);
                    break;
                }
            }
            return false;
        }
        case WM_DESTROY:
        {
            g_bRunning = false;
            PostQuitMessage(0);
            return false;
        }
    }

    return DefWindowProc(hWnd, uMessage, wParam, lParam);
} //WindowProcedure
```

capplication.cpp

Only `InitWindow()` has changed in this class. The background color is set to the standard windows color.

```
bool CApplication::InitWindow(void)
{
    WNDCLASSEX WindowClass;

    WindowClass.cbSize = sizeof(WNDCLASSEX);
    WindowClass.style = CS_HREDRAW | CS_VREDRAW;
    WindowClass.lpfnWndProc = WindowProcedure;
    WindowClass.cbClsExtra = 0;
```

```

WindowClass.cbWndExtra = 0;
WindowClass.hInstance = GetModuleHandle(NULL);
WindowClass.hIcon = NULL;
WindowClass.hCursor = NULL;
WindowClass.hbrBackground = GetSysColorBrush(COLOR_BTNFACE); //color (NEW)
WindowClass.lpszMenuName = NULL;
WindowClass.lpszClassName = "ClassName";
WindowClass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

if (!RegisterClassEx(&WindowClass))
{
    MessageBox(m_hWindow,
        "RegisterClassEx() failed!",
        "CreateApplicationWindow()",
        MB_OK);
    return false;
}

```

A different window behaviour and style (`WS_OVERLAPPEDWINDOW`) is passed to `CreateWindowEx()`. Furthermore, the size of the window has changed. `ShowCursor(false)` has been removed because the application is not in fullscreen mode and the windows cursor might be useful;-)

```

if (!(m_hWindow = CreateWindowEx(0,
                                "ClassName",
                                TITLE,
                                WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                                0, 0, 320, 240,
                                NULL, NULL,
                                GetModuleHandle(NULL), NULL)))
{
    MessageBox(m_hWindow,
        "CreateWindowEx() failed!",
        "CreateApplicationWindow()",
        MB_OK);
    return false;
}

return true;
} //CreateApplicationWindow

```

audio.h

Now the preparation is finished and the real work starts. At first, the header constant is defined to prevent the header from being included more than once.

```

#ifndef audio_h
#define audio_h

```

Here the class `CAudio` is defined. It has a constructor which calls `InitAudio()` to initialize `DirectXAudio`. Then a default search directory can be set. this is needed to load sound files. The destructor calls `KillAudio()` which releases all interfaces.

```

class CAudio
{
public:
    CAudio(); //standard constructor
    ~CAudio(); //destructor

    void    InitAudio(); //initializes DXAudio

    void    SetSearchDirectory(const char*); //search directory

```

```
void KillAudio(); //release DXAudio interfaces
```

Then two private objects are declared. The performance object is the most important one when using DirectXAudio. It handles anything related with playback. The loader object is responsible for loading sound files.

```
private:
    IDirectMusicPerformance8* m_pPerformance; //performance object
    IDirectMusicLoader8*      m_pLoader; //loader object
};

#endif
```

caudio.cpp

At the beginning main.h is included. The constructor sets the performance and the loader object to NULL and then calls `InitAudio()`.

```
CAudio::CAudio()
{
    m_pPerformance = NULL;
    m_pLoader = NULL;

    InitAudio();
} //CAudio
```

The destructor just calls `KillAudio()`.

```
CAudio::~CAudio()
{
    KillAudio();
} //~CAudio
```

Since DirectXAudio is all COM (Component Object Model) it must be initialized first otherwise any other function will fail. To get the performance and loader objects `CoCreateInstance()` is used. The first parameter is the class identifier (CLSID) of the object to be retrieved. The fourth parameter contains a reference to the identifier and the last one is the address of the variable that receives the object.

Then the performance object is used to initialize the performance by calling `InitAudio()`. In the first two parameters it is possible to specify own `IDirectMusic` and `IDirectSound` objects. Passing NULL in both means that they are created and used internally by the performance. The next parameter is our window handle. A default audiopath is created with the following argument and 64 channels are allocated to it. `DMUS_AUDIOF_ALL` enables all audio features and the last parameter means that default values are used for the synthesizer.

```
void CAudio::InitAudio()
{
    CoInitialize(NULL); //initialize COM

    CoCreateInstance(CLSID_DirectMusicLoader,
                    NULL,
                    CLSCTX_INPROC,
                    IID_IDirectMusicLoader8,
                    (void*)&m_pLoader); //create loader object
```

```

CoCreateInstance(CLSID_DirectMusicPerformance,
                NULL,
                CLSCTX_INPROC,
                IID_IDirectMusicPerformance8,
                (void**)&m_pPerformance); //create performance object

m_pPerformance->InitAudio(NULL,
                          NULL,
                          NULL,
                          DMUS_ATH_PATH_DYNAMIC_STEREO,
                          64,
                          DMUS_AUDIOF_ALL,
                          NULL); //initialize DirectMusic & DirectSound

```

Here the current directory is retrieved, the subdirectory *sounds* is appended and then `SetSearchDirectory()` is called.

```

char szSearchPath[MAX_PATH];

GetCurrentDirectory(MAX_PATH, szSearchPath);
strcat(szSearchPath, "\\sounds");
SetSearchDirectory(szSearchPath);
} //InitAudio

```

The default search directory string has to be a wide char so `MultiByteToWideChar()` converts our standard string to wide char. Then this directory is set for all music types through the loader object.

```

void CAudio::SetSearchDirectory(const char* szPath)
{
    WCHAR wszSearchPath[MAX_PATH];

    MultiByteToWideChar(CP_ACP,
                        0,
                        szPath,
                        -1,
                        wszSearchPath, MAX_PATH); //convert to wide char

    m_pLoader->SetSearchDirectory(GUID_DirectMusicAllTypes,
                                wszSearchPath,
                                false); //set as default search path
} //SetSearchDirectory

```

When the audio object is destroyed `KillAudio()` is called. At first the loader object is released from memory. The performance object has to be closed down before it can be released. At last COM is uninitialized.

```

void CAudio::KillAudio()
{
    m_pLoader->Release(); //release loader object

    m_pPerformance->CloseDown(); //close down performance object
    m_pPerformance->Release(); //release performance object

    CoUninitialize(); //uninitialize COM
} //KillAudio

```

That is all for now. `DXAudio` has been initialized and is prepared to play sounds. I have left out any error handling code in this project to keep it simple. The next tutorial covers loading and playing of 2D-sounds.