

Tutorial 1: Create a Window

by Victor Saar, Ingmar Rötzer

Content

This tutorial is the first one in a series about using Direct3D. Although we want to create a fullscreen Direct3D application a window is still necessary. Direct3D relies on this window to communicate with the Windows operating system and therefore we have to start with that. The window is created using the Win32 API, which provides the methods to create and handle it. Short explanations about the used Winapi functions will be provided in this tutorial. For further information visit msdn.microsoft.com or take a look at other resources concerning this topic.

Why do we use object-oriented programming in our projects? Firstly, the source code is arranged more clearly. All the functions and variables that belong together are encapsulated in one class. Creating an application object will automatically initialize the whole application including a window and all Direct3D stuff. Secondly, the development is modular. You can easily add a component, e.g. CSun and that object will provide all methods to handle that part of the application. After reading that first tutorial you might get the impression that OOP is somehow "overpowered" here. But once we have moved on a bit the advantages of this approach will become evident.

In this tutorial we start with the application class but concentrate only on the window for now. After executing the program a window is shown where several settings concerning D3D can be chosen. These settings are saved in a config file. The "Start" button will trigger the D3D initialization (as described in the following tutorial). For now, it just shuts down the app.

Programs are usually split into three parts. These parts are the initialization of all required objects, such as the window and Direct3D, the main part, that is doing nothing so far, and the releasing part, which will release all objects and free the appropriate memory.

The project is divided into the following files shown below. In the following tutorials you will always get a list of the added files. We need two files for the application class, two for the main source and one more for handling the Windows messages. The resource and the manifest file are essential so that the WinXP styles can be used. Do not forget to link to *d3d9.lib* and *comctl32.lib*.

main.h

This is the main header file, which has to be included in every other project file. At the beginning we define a constant *main.h* if this was not done before. This prevents the variables and function prototypes to be declared more than once.

```
#ifndef main_h
#define main_h
```

Then we include the main Windows header, which is needed for the Win32 API. *commctrl.h* is necessary to use the Windows XP styles. *d3d9.h* is the main Direct3D header file. *fstream.h* is for input and output of files. Last but not least *application.h* belongs to our custom application class.

```
//includes
#include<windows.h>
#include<commctrl.h>
#include<d3d9.h>
```

```
#include<fstream.h>
#include"application.h"
```

The following constants define the title of the application and the size of its window. The last two constants identify the buttons.

```
//constants
#define TITLE          "D3D Tut 01: Create Window"
#define WINDOW_X      350
#define WINDOW_Y      320

//Button ID's
#define ID_START      1
#define ID_CANCEL     2
```

The application object is declared as `extern` so we can use it throughout the whole project.

```
//globals
extern CApplication    g_App;
```

This is the function prototype of the window procedure, which handles the Windows messages.

```
//function prototypes
LRESULT CALLBACK      WindowProcedure(HWND, UINT, WPARAM, LPARAM);

#endif
```

main.cpp

This is the main file of all projects containing the main program structure.

At first we include the main project header and create an object of the application class. The standard constructor will be used to create the window and later initialize the Direct3D stuff.

```
#include"main.h"

CApplication      g_App; //application object
```

The main function in Windows applications is called `WinMain`. It is the entry point for a Win32-based application. `hInstance` is a handle to the current instance of the application. The other parameters are not interesting to us. `Message` will contain the windows message to be processed.

```
int WINAPI WinMain(HINSTANCE hInstance,
                  HINSTANCE hPreviousInstance,
                  LPSTR lpcmdline,
                  int nCmdShow)
{
    MSG Message; //contains messages from window
```

The next loop is very important as it is our main program loop. `g_App.GetWindowStatus()` tells us whether the app has stopped. So the loop continues until `g_App.GetWindowStatus()` returns false.

Windows is a message based OS. So every action creates messages sent to the appropriate object. For example, our window receives a message when a button is pressed. `GetMessage()` takes these messages from the window's message queue and stores them in `Message`. Messages that are used for

navigation in dialogs, e.g tabs, are processed separately by `IsDialogMessage()`. All remaining ones are translated and then dispatched to a window procedure.

```
//main program loop
while(g_App.GetWindowStatus())
{
    //get next message from queue
    if(GetMessage(&Message, NULL, 0, 0))
    {
        //process dialog messages, e.g. tabs
        if(!IsDialogMessage(g_App.GetWindowHandle(), &Message))
        {
            TranslateMessage(&Message); //translate messages
            DispatchMessage(&Message); //dispatch message to window procedure
        }
    }
}
```

This is the end of our program.

```
return 0;
} //WinMain
```

winproc.cpp

At first the main header is included. The window procedure receives all messages intended for our window so we can respond to them. Parameters are the handle of the window, the type of the message and two more arguments describing the message.

```
#include "main.h"

LRESULT CALLBACK WindowProcedure(HWND hWnd,
                                  UINT uMessage,
                                  WPARAM wParam,
                                  LPARAM lParam)
{
```

The following `switch` statement checks for various message types contained in `uMessage`. `WM_COMMAND` is sent when a user input occurred in the window. The `wParam` (actually the low order word of it) specifies the type of the command message. Here we respond to our two buttons. The Start button saves the D3D settings and will later initialize Direct3D. For now, it shuts down the app as well as the cancel button does.

```
switch(uMessage)
{
    case WM_COMMAND: //user command on window
    {
        switch(LOWORD(wParam))
        {
            case ID_START: //start button pressed
            {
                g_App.SaveSettings(); //save settings to config
                g_App.SetWindowStatus(false); //quit app, later init D3D here
                break;
            }
            case ID_CANCEL: //cancel button pressed
            {
                DestroyWindow(hWnd); //send WM_DESTROY to window
                break;
            }
        }
    }
}
```

```

}
return 0;
}

```

The next message we respond to is a `WM_KEYDOWN` message. Details about the key pressed can be found in `wparam`. Pressing Escape therefore destroys the window (and the application).

```

case WM_KEYDOWN:
{
switch(wparam)
{
case VK_ESCAPE:
{
DestroyWindow(hWnd); //send WM_DESTROY to window
break;
}
}
}
return 0;
}

```

If the window receives a `WM_DESTROY` message we just set the window status in our application class to be false. Then the main program loop terminates.

```

case WM_DESTROY:
{
g_App.SetWindowStatus(false);
break;
}
}

```

All other messages we do not respond to by ourselves are handled by the `DefWindowProc()` function.

```

//handle remaining messages
return DefWindowProc(hWnd, uMessage, wparam, lparam);
} //WindowProcedure

```

capplication.h

Starting with the constant definition and the inclusion of *main.h*.

```

#ifndef capplication_h
#define capplication_h

#include "main.h"

```

In this file we declare our application class called `CApplication`. The first function is the default constructor which is automatically called when the object is created. Followed by the destructor that is called if the object is destroyed. The third function creates our window. Settings for D3D are loaded and saved by the next two functions. The sixth function will be called in the destructor and helps in destroying the window. Last but not least there are three inline functions that return information about the window handle and state or change the current window state.

```

class CApplication
{
public:
    CApplication(void);
    ~CApplication(void);

```

```

void          InitWindow(void);
void          SaveSettings(void);
void          LoadSettings(void);

void          KillWindow(void);

inline bool   GetWindowStatus(void)
{
    return m_bRunningWindow;
}

inline HWND   GetWindowHandle(void)
{
    return m_hWindow;
}

inline void   SetWindowStatus(bool bRunningWindow)
{
    m_bRunningWindow = bRunningWindow;
}

```

Here are a lot of private member variables now. `m_bRunningWindow` controls the application, it runs as long as the value is true. `m_hWindow` is the handle of the window. All elements on this window, e.g buttons or combo boxes, are also controlled by window handles. The following four variables contain information about the desired screen resolution in 3D-mode, the type of vertex processing and the degree of anisotropy. The last three are specific D3D data types that are explained in one of the following tutorials.

```

private:
    bool          m_bRunningWindow;

    HWND          m_hWindow,
                m_hBtnStart,
                m_hBtnCancel,
                m_hLblResolution,
                m_hCbResolution,
                m_hLblBackBuffer,
                m_hCbBackBuffer,
                m_hLblDepthStencil,
                m_hCbDepthStencil,
                m_hLblVertexProcessing,
                m_hCbVertexProcessing,
                m_hLblMultiSampling,
                m_hCbMultiSampling,
                m_hLblAnisotropy,
                m_hCbAnisotropy;

    DWORD         m_dwWidth,
                m_dwHeight,
                m_dwVertexProcessing,
                m_dwAnisotropy;

    D3DFORMAT     m_ColorFormat,
                m_DepthStencilFormat;

    D3DMULTISAMPLE_TYPE m_MultiSampling;
};

#endif

```

This is our application class so far. It will be extended in the next tutorial where we init Direct3D.

cappplication.cpp

This file covers the implementation of the class. At the beginning we include *main.h*. After that the default constructor initializes most member variables. The screen resolution is set to 800x600 with 16bit color depth using software vertex processing and disabling any antialiasing and anisotropic filtering. These are just failsafe settings. They are normally fetched from the config file. At last the function `InitWindow()` is called that creates our window.

```
#include "main.h"

CApplication::CApplication(void)
{
    m_dwWidth = 800;
    m_dwHeight = 600;
    m_ColorFormat = D3DFMT_R5G6B5;
    m_DepthStencilFormat = D3DFMT_D16;
    m_dwVertexProcessing = D3DCREATE_SOFTWARE_VERTEXPROCESSING;
    m_MultiSampling = D3DMULTISAMPLE_NONE;
    m_dwAnisotropy = 0;

    m_bRunningWindow = true;

    InitWindow();
} //CApplication
```

Here we have the destructor that calls `KillWindow()` to destroy the window.

```
CApplication::~CApplication(void)
{
    KillWindow();
} //~CApplication
```

The next function creates our window. A `WNDCLASSEX` structure describes the window and is filled below. `InitCommonControls()` enables the styles on WindowsXP machines, it has no effect on other windows versions. The last function gets the current desktop size.

```
void CApplication::InitWindow(void)
{
    WNDCLASSEX WindowClass;
    RECT DesktopSize;

    InitCommonControls();

    GetClientRect(GetDesktopWindow(), &DesktopSize);
```

The window class contains information about our window. Do we have a menu, what icons do we use? The most important things are the reference to the window procedure, the application instance returned by `GetModuleHandle(NULL)` and the name of the window class. Apart from that we want a standard icon and cursor and the background should be standard background color. Finally, the class is registered so that it can be used.

```
WindowClass.cbSize = sizeof(WNDCLASSEX); //size of class
WindowClass.style = CS_HREDRAW | CS_VREDRAW; //class style
WindowClass.lpfnWndProc = WindowProcedure; //window procedure
WindowClass.cbClsExtra = 0;
WindowClass.cbWndExtra = 0;
WindowClass.hInstance = GetModuleHandle(NULL); //instance
WindowClass.hIcon = NULL
WindowClass.hCursor = NULL
WindowClass.hbrBackground = GetSysColorBrush(COLOR_BTNFACE);
WindowClass.lpszMenuName = NULL;
WindowClass.lpszClassName = "ClassName"; //name of class
```

```
WindowClass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

RegisterClassEx(&WindowClass);
```

Now `CreateWindowEx()` is used to create the window. The first parameter is the extended window style. Then follows the name of the window class which defines the window. The next thing is the window title that we defined in *main.h*. A combination of flags define the standard window style. The next four parameters are the x and y coordinates of the upper left corner and the width and height of the window. At last there are some uninteresting parameters.

```
m_hWindow = CreateWindowEx(WS_EX_CONTROLPARENT,
    "ClassName",
    TITLE,
    WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU |
    WS_MINIMIZEBOX | WS_VISIBLE,
    (DesktopSize.right - WINDOW_X) / 2,
    (DesktopSize.bottom - WINDOW_Y) / 2,
    WINDOW_X,
    WINDOW_Y,
    NULL,
    NULL,
    GetModuleHandle(NULL),
    NULL);
```

The window elements such as buttons and combo boxes are created using `CreateWindow()` as well. Differences to the main window are the window class names that depend on the type of the element and the various styles, because each element has its own behaviour flags. These elements are treated as child windows since `m_hWindow` is set to be their owner window. The buttons also get their ID's here (9th parameter). To prevent this tutorial from becoming too big only an excerpt of the corresponding code is given. For detailed information refer to the project source.

```
m_hLblResolution = CreateWindow("static", //class name
    "Resolution:", //title
    WS_CHILD | WS_VISIBLE | SS_LEFT,
    20,
    24,
    200,
    18,
    m_hWindow, //parent
    NULL,
    (HINSTANCE)GetWindowLong(m_hWindow,
        GWL_HINSTANCE),
    NULL);
```

All the combo boxes that exist now are still empty. So we fill them with some predefined values. A message is sent to them containing the command to add the following string to the list. After that has been done the settings are loaded from the config file. Again only an excerpt is given.

```
SendMessage(m_hCbResolution, CB_ADDSTRING, 0, (long)"640 x 480");
SendMessage(m_hCbResolution, CB_ADDSTRING, 0, (long)"800 x 600");
...

LoadSettings();
} //InitWindow
```

The following function opens the config file, loads the settings and selects them in the combo boxes of the window. It's a lot of code so please take a look at the source for details. Most of it is self-explanatory.

```
void CApplication::LoadSettings(void)
{
    ...
} //LoadSettings
```

SaveSettings() does basically the same, just the other way round. The current selection is read from the combo box and saved to the config file.

```
void CApplication::SaveSettings(void)
{
    ...
} //SaveSettings
```

At the end the class is unregistered and its memory freed.

```
void CApplication::KillWindow(void)
{
    UnregisterClass("ClassName",GetModuleHandle(NULL));
} //KillWindow
```

This is the whole application class so far. The default constructor creates the window when an object is created and the destructor is called automatically when the object is destroyed. In the following tutorial, the initialization of Direct3D will take place so that the application class is complete.