

# Tutorial 2: Init Direct3D

by Victor Saar, Ingmar Rötzer

## Content

Till now we created a window, which handles the messages from Windows and started programming the application class. In this tutorial we finish the application class. We will init Direct3D with the appropriate parameters. There are two ways of doing this. The easy and the hard way. In this tutorial we are concentrating on the easy one. Usually we have to enumerate all adapters, devices and modes, which are available on the system, but here we start from knowing the capabilities of the system. We use the default adapter, so when you have two or more graphic cards in your system the standard will be used. We will set the screen resolution, the color bits, the depth buffer bits and the stencil buffer bits.

After that we have to initialise the scene. That means we must create and set a projection matrix, which ports the 3D coordinates of the vertices to 2D coordinates of the screen. This is also a very important thing. We will do this using the Direct3DX utility library.

We don't have to add more files, because everything we do will be added to the existing files from the first tutorial. Only the *application.h* and *application.cpp* will be changed and two include files are added to *main.h*.

Now some things to the theory. To use Direct3D in our applications we must first create the main object. It is used to first enumerate all needed information about your system and later to create the device. The device is used to render the scene. We send all information to the device and it will render it with all transformations and lighting to a render surface. You usually need two different surfaces to work with Direct3D. The first is the front buffer, which holds the current screen data. The second is the back buffer, on which the device renders our scene. When you want to see something on the screen you have to copy the back buffer. This means that you switch the pointers of the two surfaces. The result is that you see the rendered scene on the screen. So you can render a new frame to the surface and the graphic adapter can display the old frame simultaneously. This is called page flipping.

There is one more problem we have to solve. The problem of lost devices. When somebody hits *ALT+TAB* the device gets lost. That means all information you sent to the device gets lost, such as the projection matrix or the render states.

Additionally to the library files, which were added in the last tutorial we have to add *d3dx9.lib* to the project. For more information take a look in the first tutorial.

## main.h

To use Direct3D and Direct3DX we must include the appropriate header files.

```
#include<d3d9.h> //Direct3D header
#include<d3dx9.h> //Direct3DX header (NEW)
```

## application.h

We need several new functions to handle Direct3D. What they do will be explained when we come to the source code, but some hints are given in the comments. The `inline` functions at the end are interface functions, that are used to get or change the application status.

```

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

    void InitWindow(void);
    void InitD3D(void);           //init Direct3D (NEW)
    void InitScene(void);        //init scene (NEW)
    void CheckDeviceCaps(void);  //check caps (NEW)
    void SaveSettings(void);
    void LoadSettings(void);

    bool CheckDevice(void);      //device lost? (NEW)

    void KillWindow(void);
    void KillD3D(void);          //free memory (NEW)

    inline bool GetWindowStatus(void)
    {
        return m_bRunningWindow;
    }

    inline bool GetD3DStatus(void)
    {
        return m_bRunningD3D;
    }

    inline LPDIRECT3DDEVICE9 GetDevice(void)
    {
        return m_pDirect3DDevice;
    }

    inline HWND GetWindowHandle(void)
    {
        return m_hWindow;
    }

    inline DWORD GetWidth(void)
    {
        return m_dwWidth;
    }

    inline DWORD GetHeight(void)
    {
        return m_dwHeight;
    }

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

    inline void SetD3DStatus(bool bRunningD3D)
    {
        m_bRunningD3D = bRunningD3D;
    }
}

```

We also need some new private objects and variables. At the beginning we add a `bool` for the main Direct3D loop, but later more to that. The main one is the Direct3D object. It is needed to create and handle the rest of the Direct3D stuff. The second is the device, which represents the graphic card. Then we have the `m_PresentParameters`, which hold the information for the device and the device capabilities structure, which is used to check for device functionality. The next variable is the projection matrix, which ports the 3D coordinates to 2D coordinates. This matrix will be made with help of the next four variables. The aspect ratio of the viewport, the field of view and the near and far clipping plane. The rest of the variables is already known from the last tutorial and wont be explained again.

```

private:
    bool m_bRunningWindow,

```

```

        m_bRunningD3D;           //Direct3D loop (NEW)

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;

LPDIRECT3D9    m_pDirect3DObject;   //Direct3D object (NEW)
LPDIRECT3DDEVICE9 m_pDirect3DDevice;
D3DPRESENT_PARAMETERS m_PresentParameters; //device parameters (NEW)
D3DCAPS9      m_DeviceCaps;        //device capabilities (NEW)

DWORD
    m_dwWidth,
    m_dwHeight,
    m_dwVertexProcessing,
    m_dwAnisotropy;

D3DFORMAT
    m_ColorFormat,
    m_DepthStencilFormat;

D3DMULTISAMPLE_TYPE m_MultiSampling;

D3DXMATRIX
float
    m_matProjection;   //projection matrix (NEW)
    m_fAspectRatio,   //viewport ratio (NEW)
    m_fFieldOfView,   //view angle (NEW)
    m_fNearPlane,     //near clipping plane (NEW)
    m_fFarPlane;      //far clipping plane (NEW)
};

```

## capplication.cpp

In the constructor we first set the pointers to `NULL` and define the variables, that are used to create the projection matrix. The field of view has to be in radians. Direct3DX offers some macros to convert angles between degrees and radians. You may wonder why we don't call `InitDirect3D()` here. We cannot do this, because here we don't know the chosen settings. We will initialize Direct3D between the loop of the settings dialog and the main Direct3D loop.

```

CApplication::CApplication(void)
{
    //define member variables
    m_pDirect3DObject = NULL;
    m_pDirect3DDevice = NULL;

    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_fFieldOfView = D3DX_PI / 4.0f; //view angle (NEW)
    m_fNearPlane = 1.0f;             //near clipping plane (NEW)
    m_fFarPlane = 1000.0f;          //far clipping plane (NEW)
    //aspect ratio of viewport (NEW)
}

```

```

m_fAspectRatio = (float)m_dwWidth / (float)m_dwHeight;

m_bRunningWindow = true;
m_bRunningD3D = false;           //Direct3D loop (NEW)

//create window
InitWindow();
} //CApplication

```

To the destructor we only add the clean-up function for Direct3D.

```

CApplication::~CApplication(void)
{
KillDirect3D(); //release Direct3D objects (NEW)
KillWindow(); //destroy window
} //~CApplication

```

Now comes the initialisation function. At first we create the Direct3D object. The parameter is only an information for Direct3D. When it fails we get a message and the program quits.

```

void CApplication::InitD3D(void)
{
//create Direct3D object
if((m_pDirect3DObject = Direct3DCreate9(D3D_SDK_VERSION)) == NULL)
{
MessageBox(m_hWindow,"Direct3DCreate9() failed!","InitD3D()",MB_OK);
m_bRunningD3D = false;
}
}

```

The next thing we have to do is to define what kind of device we want. The structure `m_PresentParameters` holds the information, which we need to create the device with the appropriate mode. The first parameter is clear. We want a fullscreen application. Then we set the swap effect to `D3DSWAPEFFECT_DISCARD`, because we don't need the displayed frame anymore. This is also important when we want to use anti-aliasing, because it is one of the swap effects that supports multisampling.

```

//size
ZeroMemory(&m_PresentParameters,sizeof(m_PresentParameters));
//fullscreen
m_PresentParameters.Windowed = false;
//throw away last frame
m_PresentParameters.SwapEffect = D3DSWAPEFFECT_DISCARD;

```

Now we come to the important information for the device. First we declare that we want our own depth and stencil buffer format. The following parameters are chosen from the settings dialog on startup. The possible values for the depth stencil buffer are `D3DFMT_D16`, `D3DFMT_D15S1`, `D3DFMT_D24X8`, `D3DFMT_D24X4S4` and `D3DFMT_D24S8`. As you can see there are 16-bit and 32-bit formats. The support of these formats depends on your hardware. The number after the `D` is the number of depth buffer bits and after the `s` of the stencil buffer bits. `x` bits are unused. The next is the handle to the window, that creates the device.

```

//own depth/stencil format
m_PresentParameters.EnableAutoDepthStencil = true;
//depth stencil format
m_PresentParameters.AutoDepthStencilFormat = m_DepthStencilFormat;
//window handle
m_PresentParameters.hDeviceWindow = m_hWindow;

```

Here we have the width, height and color format of the application. This is again taken from the settings dialog. Again we have 16-bit and 32-bit formats. Possible values are `D3DFMT_R5G6B5`, `D3DFMT_X1R5G5B5`, `D3DFMT_A8R8G8B8` and `D3DFMT_X8R8G8B8`. The numbers behind R, G, B and A define the bits for the red, green, blue and alpha channel of the back buffer. X is again for unused bits. The last parameter is the multisampling type. Possible values are either `D3DMULTISAMPLE_NONE` or `D3DMULTISAMPLE_n_SAMPLES`, where *n* is the number of samples (1-16).

```
m_PresentParameters.BackBufferWidth = m_dwWidth;           //screen width
m_PresentParameters.BackBufferHeight = m_dwHeight;         //screen height
m_PresentParameters.BackBufferFormat = m_ColorFormat;      //color depth
m_PresentParameters.MultiSampleType = m_MultiSampling;     //anti-aliasing
```

Now we have all information we need to create the device. At first we want the default adapter, which is usually the first graphic card installed on the system. Then we choose the HAL device type (`D3DDEVTYPE_HAL`), because we want hardware acceleration for the application. If your graphics card doesn't support hardware acceleration you can choose one of the software devices (`D3DDEVTYPE_REF`, `D3DDEVTYPE_SW`). After that we have the window handle. The next parameter is the vertex processing. This could either be software (`D3DCREATE_SOFTWARE_VERTEXPROCESSING`) or hardware (`D3DCREATE_HARDWARE_VERTEXPROCESSING`). The second is only possible if your hardware consists of a hardware TnL unit. Otherwise you must use software vertex processing. The next parameter is the address of the structure `m_PresentParameters`, which holds the device information. Last, but not least, the device itself.

```
if (FAILED(m_pDirect3DObject->CreateDevice(D3DADAPTER_DEFAULT,
                                           D3DDEVTYPE_HAL,
                                           m_hWindow,
                                           m_dwVertexProcessing,
                                           &m_PresentParameters,
                                           &m_pDirect3DDevice)))
{
    MessageBox(m_hWindow, "CreateDevice() failed!", "InitD3D()", MB_OK);
    m_bRunningD3D = false;
}
```

At the end we deactivate the cursor, because it isn't needed in our Direct3D applications. Now were Direct3D is initialized we have to call `InitScene()` and `CheckDeviceCaps()`. The second wont be explained, because it has no functionality, yet. It'll later be used to check necessary device capabilities.

```
ShowCursor(false);

InitScene();
CheckDeviceCaps();
} //InitD3D
```

Now where we have created the device we initialize the scene. First we create the projection matrix with help of the Direct3DX function `D3DXMatrixPerspectiveFovLH()`. Here we use the variables defined in the constructor. The function creates with the aspect ratio, the field of view and the near and far clipping plane a projection matrix for a left-handed coordinate system. The left-handed coordinate system is the common one for Direct3D. At the end we send the matrix to the device. The next thing is to set the filter for all texture stages. The minification and magnification filters are set to bi-linear filtering (`D3DTEXF_LINEAR`). The mip filter is set to `D3DTEXF_ANISOTROPIC`. The level of anisotropy can be chosen in the settings dialog and is set here as the maximum.

```
void CApplication::InitScene(void)
{
```

```

D3DXMatrixPerspectiveFovLH(&m_matProjection,
                           m_fFieldOfView,
                           m_fAspectRatio,
                           m_fNearPlane,
                           m_fFarPlane);
m_pDirect3DDevice->SetTransform(D3DTS_PROJECTION,&m_matProjection);

for(unsigned i = 0;i < 8;++i)
{
    m_pDirect3DDevice->SetSamplerState(i,
                                       D3DSAMP_MINFILTER,
                                       D3DTEXF_LINEAR);
    m_pDirect3DDevice->SetSamplerState(i,
                                       D3DSAMP_MAGFILTER,
                                       D3DTEXF_LINEAR);
    m_pDirect3DDevice->SetSamplerState(i,
                                       D3DSAMP_MIPFILTER,
                                       D3DTEXF_ANISOTROPIC);
    m_pDirect3DDevice->SetSamplerState(i,
                                       D3DSAMP_MAXANISOTROPY,
                                       m_dwAnisotropy);
}
} //InitScene

```

The next function checks if the device got lost. We get the current cooperative level and check if it is lost, if we can restore it or if everything is okay. The device will get lost when you press ALT+TAB. Then we cannot restore it and return `false`, so that the render part in `main.cpp` will not processed. When we return to the program the return value is `D3DERR_DEVICENOTRESET` and we can reset the device. We again do this with the function `Reset()` and must then init the scene again, because every information got lost. In this case we return `true`, so that we can go on with rendering. If you reset the device you also have to recreate all buffers created with `D3DPOOL_DEFAULT`.

```

bool CApplication::CheckDevice(void)
{
    //check device
    switch(m_pDirect3DDevice->TestCooperativeLevel())
    {
        case D3DERR_DEVICELOST: return false;
        case D3DERR_DEVICENOTRESET:
        {
            //reset device
            if(FAILED(m_pDirect3DDevice->Reset(&m_PresentParameters)))
            {
                MessageBox(m_hWindow,"Reset() failed!","CheckDevice()",MB_OK);
                return false;
            }

            //init scene again
            InitScene();

            return true;
        }
        default: return true;
    }
} //CheckDevice

```

At the end we have to clean-up, of course. With this function we release all objects, that were used for Direct3D. This function is used in the destructor.

```

void CApplication::KillDirect3D(void)
{
    if(m_pDirect3DDevice != NULL)
    {

```

```

m_pDirect3DDevice->Release();
m_pDirect3DDevice = NULL;
}

if(m_pDirect3DObject != NULL)
{
m_pDirect3DObject->Release();
m_pDirect3DObject = NULL;
}
} //KillDirect3D

```

## winproc.cpp

In this file we just have to update some of the messages. First we add everything to the start button, that is needed to initialize Direct3D. We activate the Direct3D main loop and call `InitDirect3D()`. One thing we also have to do is to set the focus on our window, because it loses the focus when we quit the message loop of the settings dialog.

```

case ID_START:
{
g_App.SaveSettings();
g_App.SetWindowStatus(false);
g_App.SetD3DStatus(true); //activate D3D loop
SetFocus(hWindow); //set window focus
g_App.InitD3D(); //init Direct3D
break;
}

```

When we get a `WM_DESTROY` message, the Direct3D loop has to be finished, too.

```

case WM_DESTROY:
{
g_App.SetWindowStatus(false);
g_App.SetD3DStatus(false); //stop Direct3D loop
break;
}

```

## main.cpp

The following part is necessary to quit the program when the initialization of Direct3D failed, because between two loops we usually create all further objects like `DirectInput`, etc. We cannot do this, because we have no valid Direct3D device, therefore we have to avoid this program part.

```

if(!g_App.GetD3DStatus())
return Message.wParam;

```

The main render part will only be processed if the device is not lost. In the `if` clause we first clear the screen by calling `Clear()`. The first two parameters define rectangles, which can be used to clear only parts of the buffers. In our case we want to clear the whole buffers, therefore we set these parameters to `0` and `NULL`. The next flag defines, which buffers should be cleared. Possible values are `D3DCLEAR_TARGET` for the current render target, `D3DCLEAR_ZBUFFER` for the depth buffer and `D3DCLEAR_STENCIL` for the stencil buffer. The next three parameters define, which values are used to clear the buffers. In this case we fill the back buffer with black, the depth buffer with `1.0f` and the stencil buffer with `0`. At the end we call `Present()`, which presents the contents of the back buffer on the screen. Between these two functions everything in the next tutorials will be drawn.

```
while(g_App.GetD3DStatus())
{
    if(PeekMessage(&Message, NULL, 0, 0, PM_REMOVE))
    {
        TranslateMessage(&Message);
        DispatchMessage(&Message);
    }

    if(g_App.CheckDevice()) //check device (NEW)
    {
        g_App.GetDevice()->Clear(0,
                                NULL,
                                D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                                D3DCOLOR_XRGB(0,0,0),
                                1.0f,
                                0);

        //copy entire screen (NEW)
        g_App.GetDevice()->Present(NULL, NULL, NULL, NULL);
    }
}
```

The application class is finished now. In the next tutorial we start drawing some polygons.