

Tutorial 3: Draw Polygons

by Victor Saar

Content

In this tutorial we will draw the first polygons. There will be two objects, a triangle with three different and a quad with one color. Before we can do this we set the needed render states and create some more requirements for drawing. We must create our own specific vertex format and structure with the needed variables included. Then we will draw the polygons with the help of vertex buffers.

We add one file to the project. It is needed for the specific vertex format and is called *d3ddefs.h*. The most will be done in the *main.cpp*. There are only some changes in *application.cpp*.

application.cpp

Before we create our own vertex structure we set the needed render states for the device. They describe how the scene will be rendered. We set the ambient light to maximum and disable lighting at first, because we don't have any lights, but we want to see something. We deactivate the culling of the back side of the polygons, by setting the cull mode to `D3DCULL_NONE`. At last we enable depth buffering so that hidden polygons aren't drawn. These lines are added to the function `InitScene()`.

```
//maximum ambient light (NEW)
m_pDirect3DDevice->SetRenderState(D3DRS_AMBIENT,RGB(255,255,255));
//lighting disabled (NEW)
m_pDirect3DDevice->SetRenderState(D3DRS_LIGHTING,false);
//don't cull backside (NEW)
m_pDirect3DDevice->SetRenderState(D3DRS_CULLMODE,D3DCULL_NONE);
//enable depth buffering (NEW)
m_pDirect3DDevice->SetRenderState(D3DRS_ZENABLE,D3DZB_TRUE);
```

Now we add at the end of `InitScene()` the following lines. They make Direct3D know how the specific vertex structure is constructed. We call the vertex format `D3DFVF_CUSTOMVERTEX`. It describes the vertex structure.

```
m_pDirect3DDevice->SetFVF(D3DFVF_CUSTOMVERTEX); //set vertex format (NEW)
```

d3ddefs.h

Here we must define the vertex format. We define that we have a vertex with a x, y and z (`D3DFVF_XYZ`) coordinate and a diffuse color value (`D3DFVF_DIFFUSE`).

```
//constants
//x, y, z and color
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_DIFFUSE)
```

And at the end of the file we define the vertex structure. It has three `float` values for the coordinates and a `DWORD` value for the color and is called `D3DVERTEX`.

```
//structures
struct D3DVERTEX
{
float fx,
```

```

    fY,
    fZ;
    DWORD dwColor;
};

```

main.cpp

In this file we modify the `winMain()`. At the beginning we have to add some variables. The first two are the needed vertex buffers. One for the triangle and one for the quad. Then we have a pointer, which we need to copy the data of the polygons into the vertex buffers.

```

int WINAPI WinMain(HINSTANCE hInstance,
                  HINSTANCE hPreviousInstance,
                  LPSTR lpcmdline,
                  int nCmdShow)
{
    MSG Message;
    //vertex buffer for triangle (NEW)
    LPDIRECT3DVERTEXBUFFER9 pTriangleVB = NULL;
    //vertex buffer for quad (NEW)
    LPDIRECT3DVERTEXBUFFER9 pQuadVB = NULL;
    //pointer to vertex data (NEW)
    VOID* pData;

```

After that we have two arrays of the data type `D3DVERTEX`. They include the x, y and z coordinate and the diffuse color value as we defined in the vertex format. We set the polygons 10 units into the screen by setting the z value to 10, because we want to see them. Though Direct3D is not able to draw quads to the screen, we only need four vertices, because we use triangle strips, which means the last two points of the first triangle are the two first of the second. The triangle will have three different colors. One for each vertex. Direct3D will interpolate the colors between them.

```

//vertex data for triangle (NEW)
D3DVERTEX aTriangle[] = {{-2.0f, 1.0f, 10.0f, 0xffff0000},
                        {-3.0f, -1.0f, 10.0f, 0xff00ff00},
                        {-1.0f, -1.0f, 10.0f, 0xff0000ff}};
//vertex data for quad (NEW)
D3DVERTEX aQuad[] = {{1.0f, -1.0f, 10.0f, 0xffffffff00},
                    {3.0f, -1.0f, 10.0f, 0xffffffff00},
                    {1.0f, 1.0f, 10.0f, 0xffffffff00},
                    {3.0f, 1.0f, 10.0f, 0xffffffff00}};

```

Before we can use the vertex buffers we must create them with the help of the device. The first parameter is the size of the array, which holds the data. Then we say that we only want to write, because it is faster (`D3DUSAGE_WRITEONLY`). The next is the vertex format and how the memory is managed. We set this to managed (`D3DPOOL_MANAGED`). The last is the pointer to the vertex buffers. For more information please take a look in the SDK documentation.

```

g_App.GetDevice()->CreateVertexBuffer(sizeof(aTriangle),
                                     D3DUSAGE_WRITEONLY,
                                     D3DFVF_CUSTOMVERTEX,
                                     D3DPOOL_MANAGED,
                                     &pTriangleVB,
                                     NULL);
g_App.GetDevice()->CreateVertexBuffer(sizeof(aQuad),
                                     D3DUSAGE_WRITEONLY,
                                     D3DFVF_CUSTOMVERTEX,
                                     D3DPOOL_MANAGED,
                                     &pQuadVB,
                                     NULL);

```

Now we fill the vertex buffers with data. First we lock them and give over the pointer, which points to the begin of the data area in the vertex buffers. Then we copy the data from the arrays and unlock both then.

IMPORTANT NOTE: One of the Wine developers informed us about a bug in the below code snippet. The value of the second parameter (sizeToLock) of both `*->Lock()` methods is wrong. It should be the size of the data, or zero to lock the entire buffer. Please consider this when using the code because I don't have the time to correct all the tutorials.

```
//lock buffer (NEW)
pTriangleVB->Lock(0, sizeof(pData), (void**)&pData, 0);
//copy data to buffer (NEW)
memcpy(pData, aTriangle, sizeof(aTriangle));
//unlock buffer (NEW)
pTriangleVB->Unlock();

//lock buffer (NEW)
pQuadVB->Lock(0, sizeof(pData), (void**)&pData, 0);
//copy data to buffer (NEW)
memcpy(pData, aQuad, sizeof(aQuad));
//unlock buffer (NEW)
pQuadVB->Unlock();
```

At the end we must draw the things to the screen. This has to be done between the clearing of the back buffer and the `Present()` call. First we tell Direct3D that we start drawing now using `BeginScene()`. Then we set with `SetStreamSource()` the vertex buffers as source for the drawing. After that we call the draw functions. For the triangle we use a triangle list (`D3DPT_TRIANGLELIST`). The next two parameters are the first triangle to draw and how many we want to draw. The same with the quad, but we use triangle strips (`D3DPT_TRIANGLESTRIP`). At the end we tell Direct3D that we stop drawing now.

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

    //start drawing (NEW)
    g_App.GetDevice()->BeginScene();

    //set source (NEW)
    g_App.GetDevice()->SetStreamSource(0, pTriangleVB, 0, sizeof(D3DVERTEX));
    //draw triangle (NEW)
    g_App.GetDevice()->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);

    //set next source (NEW)
    g_App.GetDevice()->SetStreamSource(0, pQuadVB, 0, sizeof(D3DVERTEX));
    //draw quad (NEW)
    g_App.GetDevice()->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);

    //stop drawing (NEW)
    g_App.GetDevice()->EndScene();
    g_App.GetDevice()->Present(NULL, NULL, NULL, NULL);
}
```