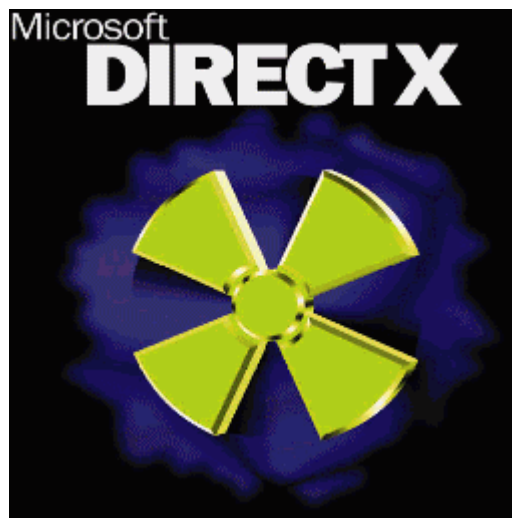


Spielprogrammierung mit DirectX 8

von
Sebastian Pech

www.spech.de
spech@spech.de



Inhaltsverzeichnis

Anhang 1
Anhang 2
Kapitel 1
Kapitel 2

Informationen
History und Wissenswertes
Das erste Dreieck
Texturen

Anhang 1: Informationen

Vorwort

In der Tutorial Sammlung „Spielprogrammierung mit DirectX 8“ werden nach und nach immer mehr Themen hinzugefügt, die bestimmte Bereiche der Spielprogrammierung mit DirectX behandeln. Die einzelnen Kapitel werden nur zum Teil auf einander aufbauen, vielmehr sind sie Problemlösungen für verschiedene Bereiche.

Ich wünsche allen Lesern viel Spaß beim Lesen und ein erfolgreiches Spiel.

Danksagungen

Ich danke ganz besonders Punika, norebo und alle aus dem Zerbie Forum für ihr Hilfe und Unterstützung bei viel zu vielen Fragen ;-)

Vorraussetzungen

Wissen über C++ und Windowsprogrammierung sollte vorhanden sein da ich Grundlegende Dinge nicht mehr erkläre, dazu gibt es genug Tutorials im Internet und Bücher die sich ausgiebig mit diesen Themen befassen.

Links

Folgenden Links bieten weiterführende Informationen und weitere Links:

Deutsch:

<http://www.stefanzerbst.de>

<http://www.games-net.de>

<http://www.untergrund-spiele.de>

Englisch:

<http://www.flipcode.com>

<http://www.gamasutra.com>

Hinweise

Der gesamte Code wird von mir auf gute und einfache Lesbarkeit geschrieben, in richtigen Projekten sollte der Code über mehrere Dateien verteilt werden außerdem sollten Funktionen zu Klassen zusammengefügt werden.

Bei einigen Codes handelt es sich um geringfügig geänderte Dateien aus dem DirectX SDK.

An einigen Stellen werde ich Pfadangaben mit *SDKPfad* beginnen dieser Teil muss durch den entsprechenden Pfad zum DirectX SDK ersetzt werden.

Copyright

Alle Texte und Bilder sind Eigentum des Autors und dürfen nicht ohne seine Genehmigung verändert, veröffentlicht oder kopiert werden.

Quellcode

Der Quellcode zu den einzelnen Dateien gibt es auf www.spech.de -> Tutorials.

Version \ Letzte Änderung

1.1 \ 29.08.02

Anhang 2: History und Wissenswertes

Änderungen

Leider hat sich ein böser Fehler in Version 1.0 eingeschlichen. Die CUSTOMVERTEX Struktur muss bei jeder Reihe ein Semikolon haben. DWORD color; usw.

Ungarische Notation

Oder wie schreibe ich meine Variablen Namen richtig:

g_name	Das g steht für Global
p_name	Das p steht für Pointer (Zeiger)
g_pName	Ein globaler Pointer
hName	Das h steht für Handle
iName	Das i ist ein Integer

Kapitel 1: Das erste Dreieck

Vorbereitungen

Als erstes erstellen wir eine neue, leere Win32Anwendung und fügen einen C++SourceFile hinzu. Danach sollte in den Optionen sichergestellt werden das die Verzeichnisse *SDKPfad\include* und *SDKPfad\lib* verfügbar sind (Tools -> Options -> Directories). Nun müssen wir die Datei *d3d8.lib* linken (Project -> Settings -> Link -> Obejct/Library modules). So genug der Vorbereitung fangen wir mit dem Code an.

Deklarationen und Prototypen

```
#include <d3d8.h>
```

Zu aller erst müssen wir die Header Includen um auf die Funktionen zugreifen zu können.

```
LPDIRECT3D8          g_pD3D          = NULL;
LPDIRECT3DDEVICE8    g_pd3dDevice    = NULL;
LPDIRECT3DVERTEXBUFFER8 g_pVB       = NULL;
```

Nun erstellen wir die Variablen in denen wir unsere Direct3D Objekt speichern um später das Device zu erstellen mit dem wir hauptsächlich unsere Render Operationen durchführen. Als drittes erstellen wir die Struktur für den VertexBuffer die später alle unsere Vertex Daten beinhalten wird.

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z, rhw;
    DWORD color;
};
```

```
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW | D3DFVF_DIFFUSE)
```

Hier definieren wir nun die Struktur die später die Eckpunkte unseres Dreieckes enthalten wird. Der rhw Wert bedeutet das wir vortransformierte Vertices nutzen werden d.h. wir können unsere Koordinaten in Bildschirmpunkten angeben.

```
HRESULT InitD3D( HWND hWnd );
HRESULT InitVB();
VOID Cleanup();
VOID Render();
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam );
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR, INT );
```

Als letzten Punkt müssen wir die Funktionen bekannt machen damit wir uns nicht um eine besondere Reihenfolge der Aufrufe kümmern brauchen, sondern alle Funktionen beliebig platzieren können (natürlich mit Einschränkungen ;-)).

WinMain

Da diese Funktion etwas länger ist werden wir uns diese etwas genauer anschauen.

```
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR, INT )
{
```

Die WinMain Funktion ist der anfangs Punkt einer jeden Win32Anwendung.

```
    WNDCLASSEX wc = { sizeof(WNDCLASSEX), CS_CLASSDC, MsgProc, 0L, 0L,
                     GetModuleHandle(NULL), NULL, NULL, NULL, NULL,
                     "GameWindowsClass", NULL };
    wc.hCursor      = LoadCursor(NULL, IDC_ARROW);
    RegisterClassEx( &wc );
```

```
    HWND hWnd = CreateWindow( "GameWindowsClass", "Spieleprogrammierung mit
    DirectX 8",
                             WS_OVERLAPPEDWINDOW, 100, 100, 640, 480,
```

```
);
GetDesktopWindow(), NULL, wc.hInstance, NULL
```

Hier erstellen wir unsere Fenster Klasse und initialisieren diese sofort, anschließend weisen wir der Klasse noch einen festen Mauscursor zu und registrieren die Klasse dann bei Windows. Im folgenden Schritt erstellen wir dann unser Fenster mit dem Titel „Spieleprogrammierung mit DirectX 8“ und der Größe 640x480 Pixel.

```
if( SUCCEEDED( InitD3D( hWnd ) ) )
{
    if( SUCCEEDED( InitVB() ) )
    {
        ShowWindow( hWnd, SW_SHOWDEFAULT );
        UpdateWindow( hWnd );
```

Nun rufen wir die Funktion zum initialisieren von DirectX auf, wenn dies erfolgreich ist erstellen wir den VertexBuffer, ist auch dies erfolgreich zeigen wir unser Fenster an.

```
MSG msg;
ZeroMemory( &msg, sizeof(msg) );
while( msg.message!=WM_QUIT )
{
    if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    else
        Render();
}
Cleanup();
UnregisterClass( "GameWindowsClass", wc.hInstance );
return 0;
}
```

Dieser Teil ist der MainLoop er läuft solange wie das Programm nicht beendet wird. PeekMessage überprüft ob eine Nachricht vorhanden ist, ist das der Fall wird sie bearbeitet, sonst rufen wir unsere Render Funktion auf. Wenn die Quit Message empfangen wurde rufen wir unsere Cleanup Funktion auf uns melden danach wieder unser Fenster ab.

MsgProc

```
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
    switch( msg )
    {
        case WM_DESTROY:
            PostQuitMessage( 0 );
            return 0;
    }
    return DefWindowProc( hWnd, msg, wParam, lParam );
}
```

Diese kurze Funktion macht nichts anderes als unsere Nachrichten auf unsere Beenden Nachricht zu prüfen und gegebenenfalls das Programm zu beenden.

InitD3D

Die nun folgende Funktion ist eine der wichtigsten in unserem Programm sie kümmert sich um das Initialisieren von DirectX.

```
HRESULT InitD3D( HWND hWnd )
{
    if( NULL == ( g_pD3D = Direct3DCreate8( D3D_SDK_VERSION ) ) )
        return E_FAIL;
```

```

D3DDISPLAYMODE d3ddm;
if( FAILED( g_pd3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT, &d3ddm )
) )
    return E_FAIL;

D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferFormat = d3ddm.Format;

if( FAILED( g_pd3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
hWnd,
                                D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                &d3dpp, &g_pd3DDevice ) ) )
{
    return E_FAIL;
}

return S_OK;
}

```

Zu aller erst erstellen wir unsere D3D Objekt in dem wir einfach die Direct3DCreate8 Funktion aufrufen und ihr den Wert D3D_SDK_VERSION übergeben, sollte dies fehlschlagen werden wir die Funktion sofort mit einem Fehlerwert beenden. Anschließend versuchen wir den aktuellen Bildschirmmodus zu bekommen um unseren BackBuffer mit dem selben Format zu erstellen. Als letztes erstellen wir unsere D3DDevice basierend auf den vorherigen Daten. Sind all diese Schritte erfolgreich ausgeführt werden geben wir S_OK zurück und wenden uns nun dem VertexBuffer zu.

InitVB

Der VertexBuffer übernimmt die Aufgabe all unsere Punkte des Dreiecks zu speichern.

```

HRESULT InitVB()
{
    CUSTOMVERTEX g_Vertices[] =
    {
        { 50.0f, 50.0f, 0.0f, 1.0f, 0xffff0000, },
        { 250.0f, 250.0f, 0.0f, 1.0f, 0xff00ff00, },
        { 50.0f, 250.0f, 0.0f, 1.0f, 0xff00ffff, },
    };

    if( FAILED( g_pd3DDevice->CreateVertexBuffer( 3*sizeof(CUSTOMVERTEX),
                                                0, D3DFVF_CUSTOMVERTEX,
                                                D3DPOOL_DEFAULT, &g_pVB )
    ) )
    {
        return E_FAIL;
    }
}

```

Als ersten Schritt definieren wir unsere drei Eckpunkte des Dreiecks. Die ersten drei Werte sind die X, Y und Z Werte in Bildschirmkoordinaten, danach folgt der RHW Wert dieser sollte immer auf 1 gesetzt werden um Darstellungsfehler zu vermeiden, der letzte Wert ist die Farbe unsere Dreiecks in der Form 0xAlphaRGB. Wenn wir unsere drei Punkte definiert haben erstellen wir unsere VertexBuffer in der Größe Anzahl der Vertices (3) * der Größe unserer Vertex Struktur (sizeof(CUSTOMVERTEX)) und legen fest das DirectX sich um die Festlegung des Speicherortes kümmert.

```

VOID* pVertices;
if( FAILED( g_pVB->Lock( 0, sizeof(g_Vertices), (BYTE*)&pVertices, 0 )
) )
    return E_FAIL;
memcpy( pVertices, g_Vertices, sizeof(g_Vertices) );
g_pVB->Unlock();

```

```
    return S_OK;
}
```

Im zweiten Teil der Funktion müssen wir die Vertex Daten in unseren neuen VertexBuffer schreiben, dazu locken wir ihn und kopieren die Daten in den VB anschließend müssen wir ihn wieder freigeben.

Render

In dieser Funktion kümmern wir uns um das Anzeigen unseres Dreiecks

```
VOID Render()
{
    g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,0),
1.0f, 0 );

    g_pd3dDevice->BeginScene();

    g_pd3dDevice->SetStreamSource( 0, g_pVB, sizeof(CUSTOMVERTEX) );
    g_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
    g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );

    g_pd3dDevice->EndScene();

    g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

Am Anfang unsere Funktion löschen wir alle Daten die im BackBuffer sind indem wir ihn mit schwarzer Farbe übermalen. Danach signalisieren wir das wir anfangen möchten zu zeichnen in dem wir BeginScenen aufrufen. Danach wählen wir als Quelle unserer Geometrie den VertexBuffer und zeigen das ganze mit DrawPrimitive an. DrawPrimitive besteht aus den drei Parametern: Art wie die Punkte verbunden sind, Erstes Vertex, Anzahl der Dreiecke. Zu letzt beenden wir mit EndScene das zeichnen der Szene. Wenn wir unsere Szene beendet haben können wir die Daten auf dem Bildschirm anzeigen.

Cleanup

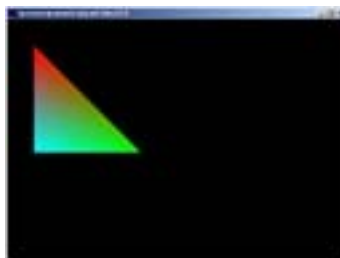
Was erstellt wurde muss auch wieder gelöscht werden das machen wir in dieser Funktion.

```
VOID Cleanup()
{
    if( g_pVB != NULL )
        g_pVB->Release();

    if( g_pd3dDevice != NULL )
        g_pd3dDevice->Release();

    if( g_pD3D != NULL )
        g_pD3D->Release();
}
```

Den Vertex Buffer, das D3DDevice, und das D3DObjekt wird gelöscht falls es erstellt wurde.



Kapitel 2: Texturen

Überlegungen

Texturen sind eigentlich ein recht einfaches Thema. Mit verschiedensten Einstellungen und mehreren Texturen übereinander, lassen sich jedoch eine ganze Menge Effekte erzielen, die den Umfang dieses Tutorials sprengen würden. Wir beschränken uns auf normale Texturen, Transparenz und dem Alphablending.

Dieses Tutorial baut auf dem selben Code auf wie Teil 1 auf. Der aktuelle Code ist in drei Ordner aufgeteilt.

Die erste Textur

Um unsere erste Textur darzustellen ändern wir erst mal unser Dreieck zu einem Viereck. Dazu suchen wir die folgende Funktion:

```
CUSTOMVERTEX g_Vertices[] =
{
    { 50.0f, 50.0f, 0.0f, 1.0f, 0xffffffff, }, // x, y, z, rhw, farbe
    { 250.0f, 250.0f, 0.0f, 1.0f, 0xff00ff00, },
    { 50.0f, 250.0f, 0.0f, 1.0f, 0xff00ffff, },
};
```

und ändern diese zu

```
CUSTOMVERTEX g_Vertices[] =
{
    { 160.0f, 80.0f, 0.0f, 1.0f, 0xffffffff, 0.0f, 0.0f, },
    { 480.0f, 80.0f, 0.0f, 1.0f, 0xffffffff, 1.0f, 0.0f, },
    { 160.0f, 400.0f, 0.0f, 1.0f, 0xffffffff, 0.0f, 1.0f, },
    { 480.0f, 400.0f, 0.0f, 1.0f, 0xffffffff, 1.0f, 1.0f, },
};
```

Nun haben wir ein weiteres Vertex erstellt. Doch warum sind dort auf einmal 2 Parameter dazugekommen? Ganz einfach wir müssen unsere Vertex Struktur noch ändern. Die neue Struktur sieht so aus:

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z, rhw;
    DWORD color;
    FLOAT tu, tv;
};
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW | D3DFVF_DIFFUSE | D3DFVF_TEX1)
```

Hier finden wir unsere beiden neuen Parameter wieder. TU und TV sind nichts anderes als die Koordinaten der Textur auf unserem Viereck.

Unglücklicherweise ist unser Vertex Buffer für ein Viereck nun zu klein. Aus diesem Grund ändern wir die folgenden Funktionen:

```
if( FAILED( g_pd3dDevice->CreateVertexBuffer( 3*sizeof(CUSTOMVERTEX),
0, D3DFVF_CUSTOMVERTEX,
D3DPOOL_DEFAULT, &g_pVB )
) )
```

Wir machen aus der 3 einfach eine 4 und fertig. Zu letzt müssen wir noch einstellen das wir nicht mehr eins sondern 2 Dreiecke rendern möchten. Dazu schreiben wir die DrawPrimitive Funktion so um:

```
g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 2 );
```

Dieser Aufruf bedeutet nun beginne beim ersten Eintrag und zeichne 2 Dreiecke. Wie aber zeichnen wir 2 Dreiecke aus vier Punkten, ganz einfach wir malen einfach eine weitere Linie

an unser erstes Dreieck dran. Nun haben wir unser Viereck und können eine Textur drauf malen.

Nachdem unser Viereck nun die Textur Koordinaten bekommen hat wird es Zeit auch die Textur zu laden. Dazu brauchen wir als erstes eine Variable die unsere Textur aufnimmt.

```
LPDIRECT3DTEXTURE8 g_texLogo = NULL;
```

Das ist die Variable die unsere Textur speichert.

```
if( FAILED( D3DXCreateTextureFromFile(g_pd3dDevice, "test.bmp",  
&g_texLogo)))  
    return E_FAIL;  
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1);  
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
```

Hier wird die Textur „test.bmp“ geladen und in unserer Variablen gespeichert. Die beiden aufrufe darunter sagen dem D3Device das wir eine ganz normale Textur haben. Nun sind wir fast am Ziel.

```
g_pd3dDevice->SetTexture( 0, g_texLogo);
```

Hier setzen wir die Textur die beim Rendern benötigt wird auf unsere eben erstellte Variable.



Transparenz

Auch Transparenz ist keine Zauberei, wir müssen dafür nicht einmal viel Code ändern also los geht es.

```
if( FAILED(D3DXCreateTextureFromFileExA(g_pd3dDevice, "alpha.bmp",  
D3DX_DEFAULT, D3DX_DEFAULT, D3DX_DEFAULT, 0, D3DFMT_UNKNOWN,  
D3DPOOL_MANAGED, D3DX_DEFAULT, D3DX_DEFAULT, D3DCOLOR_XRGB(255,0,255),  
NULL, NULL, &g_texLogo)))  
return E_FAIL;
```

So sieht unsere neue Lade Funktion für die Textur aus, aber keine Angst uns interessieren nur ein paar Parameter. Als erstes übergeben wir wieder unsere D3DDevice, der zweite Parameter ist unsere Datei, der elfte Parameter gibt an das die Farbe 255,0,255 transparent wird und der letzte Parameter ist wieder unsere Textur Variable.

```
pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);  
pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);  
pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
```

Zu guter letzt geben wir noch an das Alphablending aktiviert wird und das die Farbparameter die wir oben angegeben haben nicht angezeigt werden sollen.



Alpha Blending

Mit Hilfe von Alpha Blending können wir einen Körper transparent erscheinen lassen, diese Technik ist in etwa dieselbe wie die der Transparenz, wieder laden wir unsere Textur wie oben beschrieben. Die Transparenz Farbe brauchen wir diesmal nicht.

Um das ganze besser zeigen zu können erstellen wir aber noch ein zweites Viereck für den Hintergrund. Weil wir aber nicht wollen das die beiden Vierecke miteinander verbunden sind und weil wir zwei Texturen nutzen müssen wir zweimal DrawPrimitive und SetTexture aufrufen. Dies sollte aber keine Problem sein.

```
g_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);  
g_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);  
g_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);
```

Ja ganz recht das ist alles was wir brauchen. Wir rufen dies einfach statt der Transparenz auf. Mit den beiden Blend Parametern kann man gut spielen um die verschiedensten Effekte zu erhalten. Die Formel für die Farbe die dabei rauskommt lautet wie folgt:

$FinalColor = SourcePixelColor * SourceBlendFactor + DestPixelColor * DestBlendFactor$

