

Witanko. Dzisiaj szybki tutorialik na podstawie poprzedniego. Wiele się już nauczyliśmy, umiemy rysować i obracać światem, ale pewnie wielu kręci nosem na płaskie figury, jakimi do tej pory się posługiwaliśmy. Dzisiaj więc znowu zmałpuję z nehe i będziemy rysować w pełnym 3D. Powstaną dwie figury przestrzenne oparte na poprzednim przykładzie i przy okazji powiemy sobie dokładniej o prymitywach (ang. *primitives*). Ponieważ w przepaściach mojego dysku leży świeżo ściągnięty DirectX 8.1 do przetestowania, więc aby nie przeciągać zaczynamy.

Sądę, że dziś to może sobie odpuścimy omawianie naszego programu od początku, ponieważ musiałbym dokładnie skopiować tekst z poprzedniego przykładu. Zamiast rozwodzić się nad zmiennymi i funkcjami do inicjalizacji, które znamy już na wylot, przystąpimy od razu do analizy kodu, który będzie miał dla nas zasadnicze znaczenie.

```
HRESULT InitGeometry()
{
    CUSTOMVERTEX g_Vertices[] =
    {
        // triangle front
        { -1.0f,-1.0f,-1.0f, 0xff00ff00, },
        { 0.0f, 1.0f, 0.0f, 0xffff0000, },
        { 1.0f,-1.0f,-1.0f, 0xff0000ff, },
        // triangle back
        { 1.0f,-1.0f, 1.0f, 0xff00ff00, },
        { 0.0f, 1.0f, 0.0f, 0xffff0000, },
        { -1.0f,-1.0f, 1.0f, 0xff0000ff, },
        // triangle left
        { -1.0f,-1.0f, 1.0f, 0xff0000ff, },
        { 0.0f, 1.0f, 0.0f, 0xffff0000, },
        { -1.0f,-1.0f,-1.0f, 0xff00ff00, },
        // triangle right
        { 1.0f,-1.0f,-1.0f, 0xff0000ff, },
        { 0.0f, 1.0f, 0.0f, 0xffff0000, },
        { 1.0f,-1.0f, 1.0f, 0xff00ff00, },

        // square front
        { -1.0f,-1.0f,-1.0f, 0xffff00ff, },
        { -1.0f, 1.0f,-1.0f, 0xffff00ff, },
        { 1.0f,-1.0f,-1.0f, 0xffff00ff, },
        { 1.0f, 1.0f,-1.0f, 0xffff00ff, },
        // right
        { 1.0f,-1.0f,-1.0f, 0xffff0000, },
        { 1.0f, 1.0f,-1.0f, 0xffff0000, },
        { 1.0f,-1.0f, 1.0f, 0xffff0000, },
        { 1.0f, 1.0f, 1.0f, 0xffff0000, },
        // back
        { 1.0f,-1.0f, 1.0f, 0xff00ff00, },
        { 1.0f, 1.0f, 1.0f, 0xff00ff00, },
        { -1.0f,-1.0f, 1.0f, 0xff00ff00, },
        { -1.0f, 1.0f, 1.0f, 0xff00ff00, },
        // left
        { -1.0f,-1.0f, 1.0f, 0xff00ffff, },
        { -1.0f, 1.0f, 1.0f, 0xff00ffff, },
        { -1.0f,-1.0f,-1.0f, 0xff00ffff, },
        { -1.0f, 1.0f,-1.0f, 0xff00ffff, },
        // top
        { -1.0f, 1.0f,-1.0f, 0xff0000ff, },
        { -1.0f, 1.0f, 1.0f, 0xff0000ff, },
        { 1.0f, 1.0f,-1.0f, 0xff0000ff, },
        { 1.0f, 1.0f, 1.0f, 0xff0000ff, },
        // bottom
        { -1.0f,-1.0f, 1.0f, 0xffffffff00, },
        { -1.0f,-1.0f,-1.0f, 0xffffffff00, },
        { 1.0f,-1.0f, 1.0f, 0xffffffff00, },
        { 1.0f,-1.0f,-1.0f, 0xffffffff00, },
    };

    if( FAILED( g_pd3dDevice->CreateVertexBuffer( 36*sizeof(CUSTOMVERTEX),
        0, D3DFVF_CUSTOMVERTEX, D3DPOOL_DEFAULT, &g_pVB ) ) )
    {
        return E_FAIL;
    }
}
```

```

}

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;
}

```

Tak jak powiedziałem, dzisiaj będą już figury, a w zasadzie to bryły rysowane w przestrzeni 3D. Może nie będę zbyt odkrywcy jeśli przyznam się od razu, że pomysł na te bryły jest ściągnięty z poprzedniego przykładu. Wprawdzie mam napisane narzędzia, które potrafią z plików generowanych przez 3D Studio Max wczytywać dane obiektów począwszy od współrzędnych, na normalnych i współrzędnych tekstuowania kończąc, ale na nie przyjdzie czas jak już opanujemy podstawy. Do ich nauki wystarczą nam zupełnie banalne bryły, które możemy sobie wygenerować "ręcznie" w naszym programie.

Na naszej scenie będziemy mieć dwie bryły. Pierwsza z nich to ostrosłup o podstawie kwadratu, druga to sześciian. Patrząc na dane podane w tablicy, zauważycie pewnie, że różnią się one dla ostrosłupa i sześcianu. A wszystko dlatego, że każdą z tych brył narysujemy sobie w inny sposób (korzystając z innego prymitywu).

Ostrosłup będziemy składać z pojedynczych trójkątów. Jak zapewne zauważyliście, każda ze ścian tej bryły to oddzielny trójkąt. Ponieważ jestem z natury leniem, więc nie chciało mi się tworzyć podstawy ostrosłupa, ponieważ i tak nie będzie nigdy widoczna (będziemy go obracali tylko wokół osi *Y*), ale jeśli ktoś chce się sprawdzić, może dołożyć jeszcze dwa trójkąty i próbować obracać tę bryłę w inny, bardziej wymyślny sposób.

Natomiast sześciian poskładamy sobie z pasów trójkątów (ang. *triangle strip*). Co to takiego wprawdzie już mówiliśmy, ale nie zaszkodzi o tym napisać jeszcze raz, tym razem ilustrując jednak rysunkami. Ale o tym będzie za moment. Po zadeklarowaniu tablicy i wypełnieniu jej odpowiednimi wartościami, tworzymy bufor wierzchołków i wpisujemy tam wszystkie nasze dane tak, jak zawsze.

```

HRESULT InitD3D( HWND hWnd )
{
    ...
    g_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );
    g_pd3dDevice->SetRenderState( D3DRS_LIGHTING, FALSE );

    return S_OK;
}

```

Skoro mamy mieć pełne bryły 3D, wypada tym razem zatroszczyć się o usuwanie niewidocznych krawędzi. Tutaj bardzo przyda się nam poprzednia lekcja, w której opisałem metodę "uniewidaczniania" powierzchni techniką rysowania trójkątów z wierzchołkami ustawionymi w odpowiedniej kolejności. Ponieważ nasze bryły będą się obracać, więc nie możemy już malować wszystkich trójkątów na scenie tak, jak w poprzednim przykładzie. Gdybyśmy zostawili to tak, jak poprzednio, to otrzymalibyśmy wprawdzie bardzo ciekawe efekty, ale zupełnie przez nas nie pożądane, jak sądzę. Dlatego też pierwsze, co musimy zrobić, to powiedzieć naszemu urządzeniu, że tym razem chcemy, aby rysowało ono nam trójkąty, ale tylko te, które posiadają określoną kolejność wierzchołków. Dla naszego przykładu przyjmijmy sobie, że widoczne trójkąty to te, które posiadają kolejne wierzchołki ułożone zgodnie z ruchem wskazówek zegara (ang. *clockwise* - zgodnie z ruchem wskazówek zegara, *counter clockwise* - przeciwnie do ruchu wskazówek zegara). Skoro tak, to musimy o tym powiadomić nasze urządzenie. Wiemy już, że użyjemy w tym celu metody **SetRenderState()**. Wywołanie jej w postaci:

```
g_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );
```

(a szczególnie jej ostatni parametr - stała **D3DCULL_CCW**) spowoduje, że urządzenie będzie uwidaczniać na ekranie tylko te trójkąty, których wierzchołki będą się układać przy rysowaniu zgodnie z ruchem wskazówek zegara.

Teraz proponuję odwołać się do tablicy ze współrzędnymi naszych brył. Weźmy dla przykładu pierwsze trzy wierzchołki, które w pierwotnym położeniu będą tworzyć część przednią naszego ostrosłupa (płaszczyznę, która jest skierowana w naszą stronę).

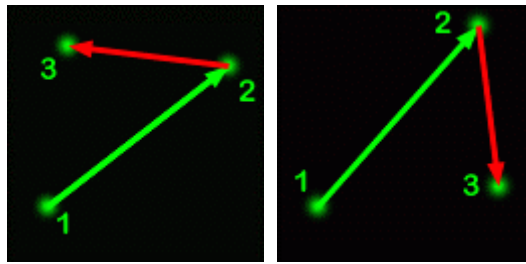
```

// triangle front
{ -1.0f, -1.0f, -1.0f, 0xff00ff00, },
{ 0.0f, 1.0f, 0.0f, 0xffff0000, },
{ 1.0f, -1.0f, -1.0f, 0xff0000ff, },

```

Przeanalizujmy jak będzie wyglądał nasz trójkąt. Pierwszy wierzchołek będzie leżał w lewym dolnym rogu, bliżej nas (jeśli wyobrazimy sobie ostrosłup w jego lokalnym układzie współrzędnych, w położeniu początkowym). Następny wierzchołek, to w zasadzie jest nam obojętne gdzie będzie. Żeby jednak nasza bryła wyglądała jako tako, przyjmijmy ten wierzchołek jako punkt zbiegu ścian ostrosłupa i umieścimy go na "szczytce" naszej bryły. Natomiast położenie ostatniego wierzchołka już nie może być obojętne. Jeśli chcemy, aby nasz trójkąt był widoczny, to nie może on leżeć po lewej stronie odcinka

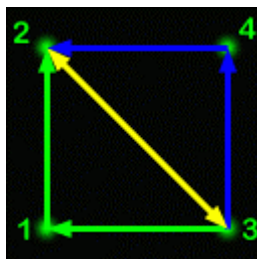
łączącego punkty 1 i 2. Łatwo to będzie sobie wyobrazić, rysując na kartce te wierzchołki w dowolnym położeniu i prowadząc strzałki od jednego do drugiego, wyznaczające kierunek, zgodnie z którym nasze wierzchołki będą ułożone. Jeśli wierzchołek trzeci położymy po lewej stronie odcinka łączącego wierzchołki 1 i 2, to strzałka narysowana od 2 do 3 wyraźnie zaprzeczy kierunkowi zgodnemu z ruchem wskazówek i nasz trójkąt w tym momencie zostanie tylko czarną plamą na czarnym tle... po prostu go nie zobaczymy. W podobny sposób postępujemy ze wszystkimi trójkątami w naszej pierwszej bryle. Poniższe rysunki obrazują te sytuacje:



Co natomiast z tym tajemniczym sześcianiem? Już wyjaśniam, weźmy dla przykładu przednią ścianę naszej drugiej bryły:

```
// square front
{ -1.0f, -1.0f, -1.0f, 0xffff00ff, },
{ -1.0f, 1.0f, -1.0f, 0xffff00ff, },
{ 1.0f, -1.0f, -1.0f, 0xffff00ff, },
{ 1.0f, 1.0f, -1.0f, 0xffff00ff, },
```

I cóż widzimy - mamy cztery wierzchołki. W sumie niby racja, cztery wierzchołki no bo cztery wierzchołki kwadratu tworzącego naszą bryłę. Ale jak już mówiłem nieraz wcześniej - wszystko co namalujemy będzie zbudowane z trójkątów. Więc? Jak to pogodzić? Ano już mówię. Przy rysowaniu sześciannu posłużymy się inną metodą malowania prymitywów. Oczywiście też namalujemy trójkąty, ale tym razem weźmiemy się na sposób i stworzymy dwa trójkąty wykorzystując tylko 4 wierzchołki a nie 6, gdybyśmy rysowali przy zastosowaniu tradycyjnych metod. Na czym będzie polegał nasz trick? Zauważmy, że pierwsze trzy wierzchołki utworzą nam piękny trójkąt (spróbujmy narysować go np. na kartce papieru). W naszym przykładzie posłużymy się prymitywem zwanym pasem trójkątów (ang. *triangle strip*). Pas taki tworzy się wykorzystując narysowane na początku trójkąty, złożone z trzech wierzchołków poprzez dodanie do niego kolejnego wierzchołka oraz wykorzystanie dwóch ostatnich wierzchołków poprzednio narysowanego trójkąta. Tak więc w naszym przykładzie najpierw rysujemy trójkąt, który będzie stanowił połowę naszej kwadratowej ścianki (połowę wyznacza w tym przypadku przekątna tego kwadratu). Łatwo już teraz się domyśleć, że dodając jeden dodatkowy wierzchołek i biorąc dwa ostatnie wierzchołki tego trójkąta, zbudujemy następny trójkąt, który będzie stanowił drugą połowę. Oczywiście musimy wiedzieć, że te dwa wierzchołki z pierwszego trójkąta będą leżały na końcach przekątnej i z trzecim stworzą taką kolejność, że po narysowaniu taki trójkąt będzie widoczny. Jak widać na rysunku i z analizy kodu wynika, że drugi trójkąt jest jednak odwrócony (nie powinien być widoczny... więc?). Jak wynika z dokumentacji SDK, pas trójkątów został właśnie tak pomyślany. Zauważmy, że gdybyśmy chcieli domalować kolejny wierzchołek i tworzyć nowy trójkąt na podstawie dwóch poprzednich, to przy właściwej kolejności drugiego trójkąta (zgodnie z ruchem wskazówek zegara) ten następny (piąty wierzchołek) musiałby leżeć po lewej stronie odcinka łączącego punkty 3 i 4, a co za tym idzie, powstały trójkąt "władowałaby" się nam na istniejący już złożony z wierzchołków 2, 3 i 4. Nie miałoby to większego sensu, więc panowie z MS wymyślili tak, że co drugi trójkąt w pasie będzie odwrócony, a o kolejności wierzchołków, które spowodują widoczność (lub nie) trójkąta będą decydować pierwsze trzy wierzchołki.



W identyczny sposób tworzymy pozostałe ściany naszej bryły, która będzie "szaleć" w przestrzeni. Nietrudno już zauważyć, że za każdym razem dokonujemy tego samego manewru. Teraz proponuję policzyć, ile wierzchołków zaoszczędziliśmy w ten sposób. Gdybyśmy rysowali tradycyjnymi metodami, to tak: 6 ścian, każda po 2 trójkąty, każdy trójkąt po 3 wierzchołki, więc w sumie daje to $6 * 2 * 3 = 36$ wierzchołków do policzenia. Korzystając z naszych pasów trójkątów mamy: 6 ścian po 4 wierzchołki, więc $6 * 4 = 24$. Czyli 12 wierzchołków mniej. Być może wydaje się to niewiele, ale wyobraźmy sobie teraz, że mamy na scenie np. słynne dema z nVidii (Porsche Boxter, straż pożarna czy też skorpion). Ilość wierzchołków w tych demach na modelach sięga niebagatelnej liczby ok. 100 000 (!) Tak, tak - słownie: sto tysięcy wierzchołków! Ilość powalająca na kolana. Oczywiście wszystko to "dzięki temu", że figury malowane są z samych trójkątów. Teraz pomyślmy, ile moglibyśmy zaoszczędzić naszemu biednemu komputerowi wysiłku przy zastosowaniu odpowiedniej kombinacji pasów

trójkątów. Nie wiem na pewno, bo trudno to oceniać, ale sądzę, że liczby zbędnych wierzchołków szły by w tysiące. Oczywiście sposób ten ma pewne wady. Chodzi między innymi o współrzędne mapowania. Ponieważ o teksturach powiemy sobie już następnym razem, więc nadmienię tylko, że problemem w przypadku pasów trójkątów będzie na przykład przypisanie wierzchołków leżącemu na granicy dwóch różnych tekstur różnych współrzędnych mapowania. Oczywiście jest na to sposób i to dosyć prosty (wspomniemy o tym) a tym bardziej docieklwym radzę się przyjrzeć budowie plików pochodzących np. z 3D Studio Max - tam ten problem rozwiązano bardzo prosto i zarazem elegancko. Drugim problemem jest odpowiednie zbudowanie modelu. Ponieważ już nikt w profesjonalnych aplikacjach nie tworzy modeli ręcznie (niech ktoś spróbuje z ręki wpisać 100 000 punktów tak, żeby wyszło Porsche Boxter :-), więc wykorzystuje się do tego celu takie programy jak 3D Studio Max, LightWave czy jeszcze inne. Większość tych programów posługuje się jednak tylko trójkątami, a nie ich pasami, więc na dłuższą metę ta technika na niewiele się nam tu przyda. Ale programy owe mają inną fajną sztuczkę, która pozwala zredukować liczbę wierzchołków do absolutnego minimum. Jest to tzw. indeksowanie. Na czym ono polega? Otóż każdy występujący na scenie wierzchołek jest zapisany w pamięci tylko raz. Ma tam on swoje określone miejsce (podobnie jak w tablicy). Dane dla każdego rysowanego trójkąta (wierzchołki) będą pobierane z właśnie z takiej tablicy. W programie będzie utworzona druga tablica zawierająca indeksy (numery) wierzchołków, z których mają być utworzone kolejne trójkąty. Tak więc program będzie patrzył najpierw na tablicę indeksów i dla każdego następnego trójkąta najpierw będzie odczytywał numery wierzchołków, z jakich będzie zbudowany trójkąt, a potem, wykorzystując te numery, dobierze się do konkretnych współrzędnych w drugiej tablicy i już namaluje piękny trójkąt. W ten sposób nie musimy w danych zawartych w pamięci powtarzać współrzędnych wierzchołków, które leżą w zasadzie w jednym miejscu w przestrzeni. Tak samo przy obliczeniach określonych w transformacji. Zauważmy, że zamiast przeliczać na przykład 10 wierzchołków leżących w tym samym miejscu, my możemy przeliczyć tylko jeden a wyjdzie na to samo, bo wszystkie stykające się akurat w tym miejscu trójkąty będą korzystały z tego samego wierzchołka. Czyż nie pięknie?

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

    D3DXMatrixTranslation( &matWorldXT, 1.5f, 0.0f, 0.0f );
    D3DXMatrixRotationY( &matWorldY, timeGetTime()/150.0f );
    g_pd3dDevice->SetTransform( D3DTS_WORLD, &(matWorldY*matWorldXT) );
    g_pd3dDevice->SetStreamSource( 0, g_pVB, sizeof(CUSTOMVERTEX) );
    g_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
    g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 4 );

    D3DXMatrixTranslation( &matWorldXT,-1.5f, 0.0f, 0.0f );
    D3DXMatrixRotationX( &matWorldX, timeGetTime()/150.0f );
    D3DXMatrixRotationY( &matWorldY, timeGetTime()/150.0f );
    g_pd3dDevice->SetTransform( D3DTS_WORLD, &(matWorldX*matWorldY*matWorldXT) );
    g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 12, 2 ); // front
    g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 16, 2 ); // right
    g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 20, 2 ); // back
    g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 24, 2 ); // left
    g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 28, 2 ); // top
    g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 32, 2 ); // bottom

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

No i w zasadzie to byłoby na tyle. Reszta kodu nie zmienia się znacząco, przez co rozumiem, że nie muszę jej tłumaczyć :-). Widać w funkcji renderującej sporo wywołań metody **DrawPrimitive()**. Ważne jest to, aby jak najwięcej przekształceń sceny zawrzeć w jednej parze metod **BeginScene()** i **EndScene()**. Ich wywołanie sporo kosztuje, więc należy się starać aby było ich jak najmniej za jednym razem, najlepiej, żeby wszystko odbywało się za jednym zamachem. Jak ktoś czegoś nie pamięta, niech zajrzy do lekcji poprzedniej. Jeśli wszystko poszło jak trzeba, otrzymamy oczywiście załączony poniżej obrazek.

