Kłaniam się. Już wiecie o czym będzie dzisiaj i pewnie nie możecie się doczekać, aby ruszyć z czytaniem? Na początek jednak kilka słów - jak zwykle. Ponieważ wiadomo, że apetyt rośnie w miarę jedzenia, więc dzisiaj nie mogło skończyć się inaczej. Umiemy inicjalizować, wiemy sporo o obiektach i metodach, więc przyswajanie nowych przyjdzie nam z łatwością. Dziś nauczymy się wnosić do naszych scen ten koloryt, który zachwyca w większości gier 3D. Jak się przekonamy po lekturze tego artykułu, tekstury są czymś, co nie tylko upiększa naszą scenę i nadaje jej swoistego "życia", ale mogą być też bardzo skutecznym narzędziem w walce o szybkość naszego programu, nieraz pomagając nam uniknąć kosztownych obliczeń i wielu mniej lub bardziej zbędnych przekształceń. Ponieważ tutorialik zapowiada się naprawdę interesująco, zapraszam bez zwłoki do czytania.

```
LPDIRECT3D8             g_pD3D       = NULL;  // Used to create the D3DDevice
LPDIRECT3DDEVICE8       g_pd3dDevice = NULL;  // Our rendering device
LPDIRECT3DVERTEXBUFFER8 g_pVB        = NULL;  // Buffer to hold vertices
LPDIRECT3DTEXTURE8      g_pTexture   = NULL;  // Our texture
```

Zaczynamy jak zwykle od początku. Będziemy mieli kilka nowych rzeczy, o których koniecznie trzeba powiedzieć. Po pierwsze nasze zmienne. Pierwsze trzy z nich już doskonale znamy i wiemy do czego służą. Tak więc mamy po kolei: obiekt Direct3D, nasze nie zastąpione urządzenie oraz bufor wierzchołków, który przechowuje bryły 3D. Nową rzeczą jest obiekt tekstury. Mówiąc najprościej będzie on odpowiedzialny za zarządzanie zasobami tekstur, którymi będziemy się posługiwać w naszym programie. Tak mówimy i mówimy o tych teksturach, ale czym one tak naprawdę są?

(Zaczerpnięte i przetłumaczone z RedBook - OpenGL 1.1)
Wyobraźmy sobie teraz dla przykładu, że pragniemy namalować mur złożony z setek cegieł. Nie mamy czegoś takiego jak tekstura, nie wiemy zupełnie co to jest. Co robimy? Większość z nas zapewne będzie chciała modelować każdą cegłę jako oddzielną bryłę. Można oczywiście sobie to zautomatyzować i napisać jakąś miłą funkcję do generowania muru z takich brył, tylko że...
Po pierwsze taki mur wymaga tysięcy takich brył... tysiące brył to dziesiątki tysięcy wierzchołków, a jakie to ma dla nas znaczenie nie muszę chyba nikomu tłumaczyć. Choćby już z tego względu takie rozwiązanie całkowicie odpada.
Po drugie mur taki wygląda zbyt idealnie. Widzieliście kiedyś mur ceglany, w którym każda cegła byłaby identyczna, miała taki sam kolor i rozmiar?! Owszem, są one podobne, ale bez przesady - nie będzie więc on zbyt realistyczny, a przecież nie o to nam chodzi! Nałożenie tekstury - czyli mapowanie to inaczej naklejanie obrazka, który posiadamy na dysku w określonym formacie (np. *.bmp, *.jpg, *.gif, *.tga) na figurę (wielokąt, bryłę). Co to dla nas oznacza? Wyobraźmy sobie, że mamy zeskanowaną fotografię jakiegoś muru, który chcemy uwidocznić na naszej scenie. Wystarczy stworzyć tylko jeden wielokąt, który będzie odpowiadał kształtem zarysowi oryginalnego muru (w najprostszym przypadku tylko 4 wierzchołki!), "przykleić" do niego nasze zdjęcie i... prawie gotowe. Prawie, ponieważ dzisiaj, przy możliwościach współczesnych akceleratorów grafiki, aby coś wyglądało realistycznie, samo naklejanie nie wystarczy. Gdy dołożymy filtrowanie tekstur (ang. *filtering*), techniki mipmappingu (ang. *mipmapping*), multiteksturowanie (ang. *multitexturing*), mapowanie nierówności (ang. *bump mapping*), odbicia i bufor szablonu (ang. *stencil buffer*), możemy naprawdę osiągnąć efekty, które sprawią, że nasz model nie będzie różnił się prawie niczym od tej rzeczywistości, którą tak pieczołowicie usiłujemy zamodelować. O wszystkich tych rzeczach powiemy sobie, ale na pewno nie dzisiaj, bo to zbyt rozległy temat. Natomiast co do samego teksturowania jeszcze. Mapowanie tekstur zapewnia nam to, że wszystko to, co stanie się z samą bryłą, stanie się też z teksturą - przykład? Jeśli ustawimy sobie mur, tak aby patrzeć na niego z pewnej perspektywy, to wiadomo, że cegły położone dalej od nas będą mniejsze i mniej wyraźnie, niż te leżące bliżej. Jeśli zrobimy to na samych bryłach, wszystko będzie w porządku, tekstury oczywiście zadziałają tak samo. DirectX sam zadba o to, aby odpowiednio przekształcić obrazek, żeby wyglądało to zgodnie z naszymi oczekiwaniami. Czemu jeszcze mogą służyć tekstury:

- w symulatorach lotu odwzorowują roślinność rosnącą na ziemi,
- modelują wzory i obrazy na ścianach,
- sprawiają, że bryły wyglądają jak zbudowane z naturalnych materiałów (marmur, drewno, tkanina, cegła, metal...).

Możliwości i przykładów istnieje w zasadzie nieskończona ilość, każdy może też wymyślać swoje własne. Chociaż najbardziej naturalnym zastosowaniem tekstur wydaje się być ich nakładanie na wypełnione trójkąty, to równie dobrze można robić to z wykorzystaniem punktów czy linii, a kwestia potrzeby takiego działania to już zupełnie inna sprawa. Ponieważ nakładanie tekstur to spory rozdział w grafice 3D (jeśli nie największy) i istnieje wiele rożnych tego aspektów, my skupimy się na absolutnych podstawach jak to robić, żeby od czegoś zacząć a z czasem, gdy przyjdą jakieś bardziej wymagające zastosowania, będziemy omawiali je po kolei. Wszystkiego naprawdę nie da się powiedzieć od razu. Wszystkim tym, którzy chcą wiedzieć wszystko od razu, polecam książkę "Fundamentals of Three-Dimensional Computer Graphics" - Alan Watt (Addison-Wesley Publishing Company, 1990).

Tektura to nic innego, jak prostokątna tablica danych (kolorów, jasności, kanału alfa). Każdy element tablicy, złożony na przykład z koloru, jasności i współczynnika przezroczystości to tak zwany "teksel" (ang. *texel*). Nakładanie tekstur na nasze trójkąty będzie polegało na odpowiednim wybieraniu i nakładaniu we właściwe miejsca trójkąta odpowiednich elementów naszej tekstury (tekseli). Mapowanie tekstur jest takie "magiczne", ponieważ prostokątne obszary są nakładane na nieregularne kształty i jest to robione w bardzo sensowny sposób.

```
D3DXMATRIX matWorldX;     // Matrix for translate via X axis
D3DXMATRIX matWorldY;     // matrix for rotate via X and Y axis
```

Przechodząc dalej w naszym programie mamy macierze, które posłużą do przeprowadzania transformacji. W naszym przypadku nie będziemy przesuwać obiektów, jak to miało miejsce poprzednio, tylko obrócimy sobie sobie nasz sześcian na środku, tam gdzie został stworzony.

```
struct CUSTOMVERTEX
{
  FLOAT x;
  FLOAT y;
  FLOAT z;
  DWORD color;
  FLOAT tx;
  FLOAT ty;
};
```

Nasze wierzchołki. Oczywiście pozostają wymagane do jakichkolwiek transformacji współrzędne w przestrzeni (*x*, *y*, *z*), kolor wierzchołka (ang. *color*), który w zasadzie to moglibyśmy sobie darować, ale... na razie niech zostanie, kto wie, może i do czegoś się nada ;-). Dwie nowe zmienne to **tx** i **ty**. Zapytacie po co to. Jak się zapewne domyślacie, ma to jakiś związek z teksturami i macie całkowitą rację. Są to tak zwane współrzędne mapowania tekstur. Cóż to takiego? Otóż wiadomo, że aby nałożyć teksturę na trójkąt, czy linię czy nawet wierzchołek, musimy wiedzieć przede wszystkim, który punkt tekstury chcemy przypisać właśnie temu konkretnemu wierzchołkowi, prawda? Możemy przecież zażyczyć sobie, aby np. pierwszy wierzchołek naszej figury miał ten teksel tekstury, który znajduje się w lewym dolnym rogu obrazka, który będziemy naklejać, prawym górnym a może zupełnie jeszcze gdzieś indziej. Po to są właśnie współrzędne mapowania. Mówią one Direct3D, który teksel obrazka chcemy nałożyć konkretnemu wierzchołkowi na naszej scenie. Ponieważ obrazek jest przeważnie płaski ;-), więc mamy dwie współrzędne, które zdefiniują nam miejsce z którego będą pobierane teksele i nakładane na naszą bryłę. Sytuacja trochę się komplikuje (ale nie dla nas na szczęście), jeśli posiadamy trójkąt. Wtedy dla każdego wierzchołka podajemy współrzędne tekseli, które chcemy im przypisać. A co z resztą trójkąta? Możemy sobie wyobrazić, jak musi kombinować teraz procedura nakładania tektury, aby wypełnić resztę trójkąta, mając tylko współrzędne na obrazku i "goły" trójkąt. Na całe szczęście nas to w ogóle nie obchodzi, wszystko dzieje się bez naszej wiedzy za sprawą naszej karty grafiki, która ma już zaimplementowane odpowiednie procedury w swoim GPU (***G****raphical* ***P****rocessing* ***U****nit*). Ona poradzi sobie doskonale z takim problemem, odpowiednio wybierze właściwe teksele z obrazka i nałoży we właściwe miejsce na trójkącie, a co najważniejsze zrobi to bardzo, bardzo szybko. Generując naszą bryłę, która ma mieć naklejone ładne obrazki, należy więc już na samym początku postarać się o to, aby posiadała ona odpowiednie dane. Do tego celu znowu mogą nam posłużyć programy, o których wspominałem wcześniej, kombajny typu 3D Studio Max, LightWave, choć są też mniejsze i, co ważniejsze, darmowe narzędzia.

```
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;
  d3dpp.EnableAutoDepthStencil = TRUE;
  d3dpp.AutoDepthStencilFormat = D3DFMT_D16;

  if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
          D3DCREATE_SOFTWARE_VERTEXPROCESSING,
          &d3dpp, &g_pd3dDevice ) ) )
  {
    return E_FAIL;
  }

  g_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_NONE );
  g_pd3dDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
  g_pd3dDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
```

```
  return S_OK;
}
```

Skoro więc wiemy już mniej więcej co to tekstura i po co nam to w ogóle, możemy przystąpić do analizy dalszej części naszego kodu. Tradycyjnie pierwsza funkcja w naszym programie to inicjalizacja urządzenia. W zasadzie posłużymy się tu już naszym standardem, czyli urządzeniem ustawionym do renderingu w oknie i włączonym buforem Z. Wyłączymy sobie też światełko, żeby nie wprowadzało nam za dużo zamieszania w nasz program. Celem jest nauka teksturowania a skomplikowane sceny ze wszystkimi bajerami to mam nadzieję, że pokażą się po lekcjach opisujących podstawy w naszym kursie. Liczę, że nie omieszkacie się pochwalić własnymi osiągnięciami ;-).

Następna rzecz to nasza bryła. Tutaj wracamy do naszego starego dobrego sześcianu. Ma on dwie niewątpliwe zalety. Po pierwsze jest go bardzo łatwo "wpisać" z ręki - czyli krótko mówiąc, zdefiniować jego wierzchołki i inne dane bez pomocy programów do modelowania. Można sobie łatwo wyobrazić jego położenie w przestrzeni i uzmysłowić, jak wyglądać będą jego współrzędne. Po drugie na sześcianie będziemy mogli pokazać dokładnie, o co chodzi z tym całym teksturowaniem. Każdy wierzchołek naszej figury to będą oczywiście jej trzy współrzędne w przestrzeni, kolor każdego wierzchołka oraz rzecz o której mówiliśmy powyżej, czyli współrzędne teksturowania. Wielu z Was zapyta jak działają takie współrzędne. Pisałem już wyżej, ale teraz wytłumaczymy sobie dokładniej. Współrzędna teksturowania wierzchołka nie dotyczy w samej rzeczy jego samego, ale raczej tekstury, która będzie nałożona na ściankę, do której dany wierzchołek należy. Mówi ona, która część tekstury zostanie przypisana danemu wierzchołkowi, aby potem nasze urządzenie mogło na podstawie obrazu tekstury w pamięci, odwzorować ją na naszej bryle dając różnorakie efekty. Jakie wartości powinny przyjmować współrzędne teksturowania? Dobre pytanie. Odpowiedź - zależy co chcemy osiągnąć. Jak już wspomniałem, tekstury to obrazki nakładane głównie na trójkąty. Dawniej dobrym zwyczajem było to, aby obrazy reprezentujące tekstury były zawsze w kształcie kwadratu (miały taką samą liczbę pikseli w poziomie i w pionie). W OpenGL-u dodatkowym wymogiem jest to, aby ilość pikseli w obrazku reprezentującym teksturę była w pionie i poziomie wielokrotnością liczby 2. Dzisiaj, przy możliwościach DirectX-a, takie ograniczenie w zasadzie nie istnieje. Pierwsze akceleratory miały pewne niewygodne ograniczenie, które nie pozwalało na używanie większych tekstur niż 256 pikseli (Voodoo1). Dzisiaj, kiedy karty dysponują nawet 256 MB pamięci (!!!), nie jest to już tak wielki problem. Ale operowanie właśnie teksturami w okolicach 128 i 256 pikseli to już chyba standard, co w połączeniu z obrazkami "zwijalnymi" (takimi, jakie stosuje się np. na tła stron WWW), daje całkiem zadowalające rezultaty. Do prostych zastosowań w zupełności wręcz wystarczają. Ale miałem pisać o DirectX, a znowu odwołałem się do OpenGL-a. DirectX-owi jest w zasadzie obojętne, jakiego rozmiaru będzie tekstura (czy prostokątna, czy też może inna). On, wykorzystując odpowiednie mechanizmy, sam dostosuje nakładany obraz do współrzędnych tekstur i wybierze z niego odpowiednie fragmenty. Wracając jednak do współrzędnych mapowania. Każda z nich odzwierciedla miejsce, z którego mają być pobierane dane z obrazu. Współrzędna (0.0f, 0.0f) mówi nam, że szukany fragment naszego obrazu znajduje się w lewym dolnym rogu obrazu. Analogicznie biorąc (0.0f, 1.0f) to lewy górny róg obrazu, (1.0f, 0.0f) to prawy dolny, no i już wiadomo, że (1.0f, 1.0f) to prawy górny. Jeśli teraz np. wyobrazimy sobie kwadrat na naszej scenie i teksturę np. z obrazem naszego wymarzonego samochodu ;-), to z jej nałożeniem nie powinniśmy mieć już żadnego problemu. Po prostu odpowiednim wierzchołkom naszego kwadratu należy przypisać odpowiadające współrzędne, które spowodują, że samochód pojawi się na naszej figurze. Dobrym przykładem niech będzie fragment kodu z naszego programu:

```
{ 1.0f,-1.0f,-1.0f, 0xFFFF0000, 0.0f, 1.0f, },
{ 1.0f, 1.0f,-1.0f, 0xFFFF0000, 0.0f, 0.0f, },
{ 1.0f,-1.0f, 1.0f, 0xFFFF0000, 1.0f, 1.0f, },

{ 1.0f, 1.0f,-1.0f, 0xFFFF0000, 0.0f, 0.0f, },
{ 1.0f,-1.0f, 1.0f, 0xFFFF0000, 1.0f, 1.0f, },
{ 1.0f, 1.0f, 1.0f, 0xFFFF0000, 1.0f, 0.0f, },
```

Tu pragnę zwrócić uwagę na jeszcze jedną rzecz, o której mgliście wspomniałem na początku. Otóż napisałem, że być może przyda nam się do czegoś kolor naszych wierzchołków. I tak będzie w istocie, co jest szczególnie widoczne właśnie w tym fragmencie kodu. Jak zobaczycie po uruchomieniu programu, kolor wierzchołków, przy zastosowaniu odpowiedniej operacji na teksturze, może mieć duże znaczenie dla jej wyglądu, co widać właśnie na tej naszej, czerwonej ścianie. Więcej szczegółów znajdziecie przy opisie funkcji renderującej.

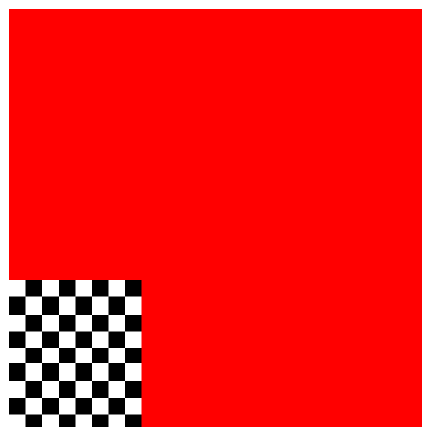
Jakiś maniak motoryzacji może powiedzieć: "Dobra, ja jestem maniakiem i chciałbym, aby mój obrazek pojawił się na sześcianie dziewięć razy." Co zrobić w takim przypadku? Możliwości mamy dwie. Pierwsza to odpowiednio spreparować teksturę - czyli w jakimś programie graficznym z jednej małej teksturki skleić taką, która spełni oczekiwania naszego maniaka i po nałożeniu na obiekt będzie to wyglądać należycie. Drugi sposób to dobrać odpowiednie współrzędne teksturowania na naszej figurze. Napisałem, że prawy górny róg naszej tekstury to współrzędne (1.0f, 1.0f). A co będzie, jeśli ktoś przypisze naszemu wierzchołkowi wartości większe niż te? Najprościej będzie po prostu sprawdzić... czy już wiecie?

```
{ -1.0f,-1.0f,-1.0f, 0xFFFFFFFF, 0.0f, 3.0f, },
{ -1.0f, 1.0f,-1.0f, 0xFFFFFFFF, 0.0f, 0.0f, },
{  1.0f,-1.0f,-1.0f, 0xFFFFFFFF, 3.0f, 3.0f, },
```

```
{ -1.0f, 1.0f,-1.0f, 0xFFFFFFFF, 0.0f, 0.0f, },
{  1.0f,-1.0f,-1.0f, 0xFFFFFFFF, 3.0f, 3.0f, },
{  1.0f, 1.0f,-1.0f, 0xFFFFFFFF, 3.0f, 0.0f, },
```

Tak, tekstura zostanie po prostu powielona tą ilość razy, ile wartość współrzędnej jest większa od 1.0f. Jest to tzw. efekt kafelkowania (ang. *tiling*). Podając np. jako współrzędne teksturowania wartość 3.0f dla wierzchołków, które mają mieć przypisane piksele z prawej strony obrazka, zauważymy, że po nałożeniu tekstury na naszą bryłę, obrazek zostanie niejako "spłaszczony" w poziomie, tak aby w rzeczywistości zmieściły się nam na ścianie trzy takie obrazki. Taki sam efekt uzyskamy oczywiście zmieniając współrzędne w pionie. W tym momencie nasz maniak motoryzacji zapewne stawia nam duże piwo ;-). Ja zaś przy okazji dodam, co się stanie jeśli wartości współrzędnych mapowania będą mniejsze niż jeden. Pewnie nie trudno będzie już sobie to wyobrazić. Po prostu jeśli w naszym przykładzie zamiast współrzędnej 3.0f, oznaczającej prawą stronę naszego obrazka, podamy np. 0.3f, to uzyskamy na naszej bryle tylko 1/3 naszego obrazka. Myślę, że jest to zupełnie zrozumiałe? No dobra, ktoś zapyta, a co będzie jeśli współrzędne teksturowania będą ujemne? Najprościej będzie oczywiście ponownie sprawdzić. Ustawmy dla przykładu jednej z naszych ścian właśnie ujemne współrzędne. I co? I w zasadzie nic, prawda? Tekstury nakładają się dalej, jak się nakładały. Można zauważyć, że będą one po prostu tyle razy powtarzane na ścianie, ile wynosi różnica pomiędzy najniższą a najwyższą wartością we współrzędnych. Jeśli np. weźmiemy sobie jako lewy górny róg współrzędne (-1, -1) a prawy górny (3, 3), to tekstura zostanie powielona na ścianie cztery razy. Jednak dobrym zwyczajem jest zawsze nakładanie tekstur z dodatnimi wartościami, ponieważ wydaje się to bardziej naturalne i logiczne. Tekstura w zasadzie nie posiada jako takich współrzędnych ujemnych, ale jak ktoś lubi sobie utrudniać, to jego sprawa. Trochę powyżej napisałem o tilingu. Jeśli czytacie help dołączony do SDK, to pewnie wiecie, że istnieje coś takiego jak tryby adresowania tekstur. Jest to bardzo fajne coś, co może nam posłużyć do uzyskania "od ręki" kilku ciekawych efektów podczas kafelkowania naszego obrazka po powierzchni naszej bryły. Możemy np. zażyczyć sobie, aby każda kolejna odbitka naszego obrazka na ścianie była po prostu powtarzana bez żadnych zmian. Możemy wymyślić sobie, aby każdy kolejny odbity wiersz i kolumna był odwrócony w stosunku do poprzedniego "lustrzanie". Efekt, o którym mówię, jest dobrze widoczny na obrazku znanym na pewno doskonale z SDK:

Możemy też podać jako współrzędne teksturowania wartości większe niż 1.0f, ale zażądać, aby tylko jeden obrazek został nałożony na naszą ścianę. Wiadomo, że ułoży się on tam gdzie podamy prawidłowe (z zakresu 0.0f - 1.0f) współrzędne. Co zaś z resztą bryły? Otóż w takim przypadku będziemy mieć dwie nowe możliwości. Pierwsza to zastosowanie tak zwanej granicy tekstury. Będzie to po prostu wypełnienie reszty naszej ściany jakimś zadanym kolorem. I znowu dobrze znany przykład:



Tekstura będzie dla nas tą małą szachownicą w lewym dolnym rogu, natomiast resztę naszej figury wypełnimy sobie dla przykładu kolorem czerwonym. Drugi przypadek to nie określanie takich granic. Co więc z naszą ścianą? Otóż reszta ściany zostanie pokolorowana takimi kolorami pikseli, jakie znajdują się w ostatnim prawidłowym rzędzie (wierszu) i kolumnie naszej tekstury. Może brzmi to trochę zawile i niezrozumiale, ale myślę, że poniższy rysunek rozwieje Wasze wątpliwości:

Jak łatwo zauważyć, zmieniłem trochę kolory naszej tekstury, żeby lepiej było widać zamierzony efekt. Po zdefiniowaniu naszej bryły z odpowiednimi współrzędnymi mapowania tekstur i odpowiednim kolorem wierzchołków, który nam się jeszcze dzisiaj przyda, możemy przystąpić do dalszych niezbędnych działań. Pierwszym z nich będzie załadowanie naszej tekstury z dysku:

```
if( FAILED( D3DXCreateTextureFromFile( g_pd3dDevice, "dx8_logo.bmp",
    &g_pTexture ) ) )
{
  return E_FAIL;
}
```

I znowu z pomocą przychodzi nam nieoceniona biblioteka **D3DX**. Dzięki funkcji **D3DXCreateTextureFromFile()** nie będziemy musieli pisać jak dawniej funkcji do wczytywania obrazka z pamięci. Funkcja ta potrafi wczytać nam podany plik o nazwie podanej jako jeden ze swoich argumentów. Funkcja pobiera jako pierwszy argument wskaźnik do naszego urządzenia (oczywiście musi ono być już zainicjalizowane), drugi to wspomniana wyżej nazwa pliku a trzeci to wskaźnik do obiektu tekstury, który zdefiniowaliśmy na początku. Będzie on nam służył do zarządzania naszą teksturą, ale w naszym przykładzie raczej żadnych cudów z nią robić nie będziemy, więc podajemy go tam raczej dla świętego spokoju.

```
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;
```

Następnie widzimy znowu naszych starych znajomych, którzy w przeciwieństwie do nas, nic a nic się nie zmieniają. Czyli nasz bufor wierzchołków, który wypełnimy tym razem współrzędnymi, kolorami i współrzędnymi mapowania. Mając te dane i teksturę zobaczymy, co potrafi nam DirectX namalować tym razem.

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

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

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

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

Funkcji czyszczącej to sądzę, że nawet nie muszę opisywać? Jedyne co dodajemy, to zwalnianie obiektu tekstury, które nie różni się niczym od innych. Utrwalamy natomiast zasadę, że jeśli korzystamy z jakiegoś obiektu, to dobrym zwyczajem będzie potem jego zwalnianie.

```
VOID SetupMatrices()
{
  D3DXMATRIX matView;
  D3DXMatrixLookAtLH( &matView, &D3DXVECTOR3( 0.0f, 0.0f,-6.0f ),
                                &D3DXVECTOR3( 0.0f, 0.0f, 0.0f ),
                                &D3DXVECTOR3( 0.0f, 1.0f, 0.0f ) );
  g_pd3dDevice->SetTransform( D3DTS_VIEW, &matView );

  D3DXMATRIX matProj;
  D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 400.0f/400.0f, 1.0f, 100.0f );
  g_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
}
```

Ta funkcja w zasadzie nie wymaga komentarza. Nie zmienia nam się dokładnie nic i dopóki nie zaczniemy poruszać po naszej scenie kamerą zamiast obiektami, to chyba przestaniemy się nią w ogóle zajmować, ponieważ miejsce na naszym serwerze jest bardzo cenne a jest go bardzo mało...

```
VOID Render()
{
  g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
D3DCOLOR_XRGB(0,0,250), 1.0f, 0 );
  g_pd3dDevice->BeginScene();

  D3DXMatrixRotationX( &matWorldX, timeGetTime()/1500.0f );
  D3DXMatrixRotationY( &matWorldY, timeGetTime()/1500.0f );

  g_pd3dDevice->SetTexture( 0, g_pTexture );
  g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
  g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
  g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
  g_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_DISABLE );

  g_pd3dDevice->SetTextureStageState( 0, D3DTSS_MINFILTER, D3DTEXF_LINEAR );
  g_pd3dDevice->SetTextureStageState( 0, D3DTSS_MAGFILTER, D3DTEXF_LINEAR );

  g_pd3dDevice->SetStreamSource( 0, g_pVB, sizeof(CUSTOMVERTEX) );
  g_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
  g_pd3dDevice->SetTransform( D3DTS_WORLD, &(matWorldX*matWorldY) );
  g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 12 ); // front

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

No i wreszcie nasza funkcja, którą już bardzo zdążyliśmy polubić :-). I jak widzimy, pojawiają się nam znowu jakieś dziwne stwory. Jak nie trudno zaważyć, nadal będą to metody naszego nieocenionego urządzenia renderującego. Tym razem jednak seria dotycząca obsługi tekstur. Żeby więc nie przeciągać, od razu na pierwszy ogień bierzemy pierwszą z nich:

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

Metoda ta przypisuje dany obiekt tekstury (g_pTexture) do zestawu aktualnie używanych na scenie tekstur. Taki zestaw może zawierać na raz do ośmiu takich tekstur. Kolejne tekstury w takim zestawie są numerowane od zera, więc nasza, jako jedyna, będzie miała numer 0 a najwyższa, gdybyśmy taką mieli, miałaby numer 7.

```
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
```

Następnie mamy serię wywołań metody **SetTextureStage()** z dosyć rożnymi parametrami. Sama metoda służy do ustawiania przeróżnych rzeczy dla aktualnie ustawionej tekstury. Pierwszym parametrem jest numer naszej tekstury w zestawie, do której będziemy stosować nasze przekształcenia. Drugi to wartość pochodząca z dosyć kosmicznie (jeśli chodzi o nazwę) wyglądającego typu wyliczeniowego **CONST_D3DTEXTURESTAGESTATETYPE**, który definiuje ni mniej ni więcej, tylko stan aktualnie ustawionej tekstury. Pisząc "stan" mam na myśli sposób jej przekształcania przez DirectX. Typ, który podałem, zawiera sporo wartości, oczywiście nie opiszę tutaj wszystkich - odsyłam jak zwykle do dokumentacji, ale z tych które używamy:

D3DTSS_COLOROP - określa sposób, w jaki kolory tekstury są mieszane z kolorami wierzchołków zdefiniowanymi dla naszej bryły. Stała ta nakazuje urządzeniu takie pokolorowanie naszej bryły, jakie będzie określone przez trzeci argument. W przypadku tej stałej trzecim argumentem będzie wartość z typu wyliczeniowego **CONST_D3DTEXTUREOP**, który definiuje sposoby mieszania kolorów tekstur z kolorami wierzchołków. **D3DTOP_MODULATE** oznacza pomnożenie wartości koloru wierzchołka z wartościami koloru tekstury, która nałożona jest na ten wierzchołek. W ostatecznym rozrachunku nasza bryła przyjmuje kolory tekstury niejako zabarwionej kolorem wierzchołków, które są "pod nią". Oczywiście typów miksowania jest o wiele więcej, niektóre dają naprawdę ciekawe efekty, niektóre posłużą nam do uzyskiwania też efektów specjalnych (np. mapowania nierówności - ang. *bump mapping*), ale o nich będziemy z pewnością jeszcze mówić. Dociekliwych jak zwykle odsyłam do dokumentacji.

```
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
```

Następne wywołanie naszej metody urządzenia korzysta również ze stałej typu **CONST_D3DTEXTURESTAGESTATETYPE**, tylko że tym razem jest to wartość **D3DTSS_COLORARG1**. Wskazuje ona na to, co będzie pierwszym argumentem przy operacji mieszania kolorów z wykorzystaniem operacji **D3DTOP_MODULATE**. Stała podana jako trzeci argument typu **D3DTA_TEXTURE** wskazuje, jak łatwo się domyśleć, że tym argumentem będzie kolor tekstury.

```
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
```

Poprzednio podaliśmy pierwszy argument do naszej operacji mieszania, więc czas na drugi. Tym razem będzie to **D3DTA_DIFFUSE** czyli nic innego, jak ta składowa koloru wierzchołków, która jest określana podczas przeprowadzania cieniowania wierzchołków. Po ustaleniu, jaka będzie nasza operacja i ustaleniu jakie elementy będą brały w niej udział, nasze urządzenie po prostu dokona samo takiej operacji bez ingerencji programowej z naszej strony. Zobaczycie jeszcze w dalszej części kursu, że odpowiednie manipulowanie takimi przekształceniami tekstur i kolorów będzie dawać naprawdę rewelacyjne efekty.

```
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_DISABLE );
```

Ostatnie wywołanie naszej metody do ustalania parametrów teksturowania to ustalenie sposobu ustawiania przezroczystości dla naszego obiektu tekstury. Jak łatwo się domyśleć, trzeci parametr spowoduje wyłączenie takiego obliczania kolorów, które dawałoby efekt przezroczystej ściany. Ale nie martwcie się, o przezroczystych ścianach będzie jak mniemam już w następnej lekcji ;-), więc uzbrójcie się w dużą dawkę cierpliwości.

```
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_MINFILTER, D3DTEXF_LINEAR );
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_MAGFILTER, D3DTEXF_LINEAR );
```

Teraz natomiast powiemy sobie troszeczkę o tzw. filtrowaniu tekstur. Kiedy Direct3D nakłada tekstury na naszą bryłę, każdy piksel tej bryły widoczny na ekranie musi posiadać kolor, który faktycznie jest zawarty na tej nakładanej na niego teksturze. Taki proces nazywa się właśnie filtrowaniem tekstur (ang. *filtering*). Kiedy tekstura jest nakładana na obiekt, to przeprowadzana jest z reguły jedna z dwóch operacji - powiększanie albo pomniejszanie tekstury. Bardzo rzadko się zdarza, aby obraz tekstury pasował rozmiarowo idealnie do naszego obiektu, tak aby po nałożeniu teksele tekstury pokrywały się dokładnie z pikselami, jakie zawiera nasza bryła. Z reguły tekstura jest albo obrazem mniejszym, albo większym niż obiekt, na który jest nakładana i w zależności od tego następuje właśnie jej powiększanie lub pomniejszanie. Powiększanie tekstury następuje wtedy, kiedy usiłujemy jednemu tekselowi tekstury przypisać kilka pikseli z naszej bryły (tekstura jest za mała, aby każdy piksel ściany otrzymał swój własny teksel z tekstury). Kiedy tekstura jest za duża, wtedy nasze urządzenie usiłuje jednemu pikselowi ściany przypisać kilka tekseli z tekstury. Jakie to może przynieść efekty będziemy widzieć bardzo dobrze, jeśli przyjrzymy się np. ścianom w legendarnym Wolfie 3D. Gdy podejdziemy bliżej do ściany, ujrzymy znienawidzoną przez wszystkich zatwardziałych graczy pikselozę, której dziś już nie toleruje się w porządnych grach 3D. Aby uniknąć takich efektów, konieczne jest więc zmieszanie kolorów tekseli nakładanej tekstury, aby otrzymać właściwy kolor, który zostanie nałożony na naszą bryłę. Direct3D daje nam do ręki narzędzia, które ułatwiają w znacznym stopniu korzystanie z takiego dobrodziejstwa jakim jest filtracja tekstur. Oczywiście jak już panowie od wymyślania najbardziej zakręconych algorytmów nas przyzwyczaili, mamy do dyspozycji kilka rodzajów filtrowania. W Direct3D spotkamy się z trzema: filtrowaniem liniowym (ang. *linear filtering*), które zastosujemy u nas, filtrowaniem anizotropowym (ang. *anisotropic filtering*) i filtrowaniem z zastosowaniem tzw. mipmappingu. Możemy też oczywiście nie stosować filtrowania, wtedy na

ekranie otrzymamy, jakże przez nas nie lubianą pikselozę, ale dla celów edukacyjnych można się pokusić o wyłączenie na chwilę filtrowania i popatrzenia sobie jak kiedyś to wyglądało :-). Oczywiście każdy typ filtrowania ma swoje wady i zalety, objawiające się głównie w ilości obliczeń, jakie są potrzebne do jego przeprowadzenia. Najmniej kosztuje oczywiście nie przeprowadzanie zaawansowanego filtrowania, następne w kolejce jest filtrowanie liniowe, potem filtrowanie anizotropowe, na końcu zaś filtrowanie za pomocą mipmap, które pożera najwięcej obliczeń i co gorsza pamięci, ale daje oczywiście najlepsze efekty. Na czym w skrócie polega filtrowanie liniowe, którym my się zajmiemy?

Filtrowanie, którego będziemy używać jest inaczej nazywane bilinearnym (ang. *bilinear*). Najpierw DirectX określa z którego miejsca tekstury będzie pobrany teksel, który zostanie umieszczony w miejscu piksela naszej bryły 3D. Miejsce to na teksturze nazywane jest adresem teksela. Najczęściej zdarza się, że taki adres nie jest liczbą całkowitą (wynika z obliczeń, jakie przeprowadza procesor karty), więc DirectX dopasowuje najbliższy całkowity adres, który znajduje się w pobliżu obliczonej wartości. Gdybyśmy w tym momencie skończyli nasz proces filtrowania, to moglibyśmy nałożyć teksturę na nasz obiekt, ale otrzymalibyśmy właśnie wspomnianą wyżej pikselozę. Żeby więc nie wyglądało to tak strasznie, DirectX bierze dodatkowo z odpowiednimi wagami (inaczej mówiąc stosunkiem do właściwego, najbliższego adresu teksela), leżące w jego najbliższym sąsiedztwie teksele (na górze, dole i po obu jego stronach), następnie miesza je wszystkie razem i nakłada tak obliczony kolor na bryłę, co daje efekt gładkiej, poteksturowanej powierzchni. Takie to właśnie powierzchnie zachwycały oczy szczęśliwych posiadaczy pierwszych akceleratorów z serii VooDoo. Właśnie podczas filtrowania tekstur akceleratory pokazują swoją moc obliczeniową, ponieważ gdyby chcieć robić to za pomocą głównego procesora (CPU - ***C**entral* ***P**rocessing* ***U**nit*), zajęłoby to nam o wiele, wiele więcej czasu...

Reszta naszej funkcji do renderingu nie stanowi już dla nas żadnej tajemnicy. Ustawiamy źródło danych, ustawiamy naszego shadera dla wierzchołków (tak swoją drogą to też będzie niedługo :-), ustawiamy nasze przekształcenia i pozostaje nam już tylko narysowanie naszej figury metodą **DrawPrimitive()** oraz przerzucenie tylnego bufora na ekran. I w zasadzie wszystko jest już jasne. Jeśli dobrze wszystko pojęliście, to po skompilowaniu przykładu, powinniście dostać na ekranie taką piękną bryłę obleczoną w teksturę, jaką widać na załączonym obrazku.