Witam po raz kolejny. Po przemyśleniu kilku spraw i przejrzeniu kilkudziesięciu przykładowych programów postanowiłem jednak, że nie tak do końca będziemy się zajmować tylko i wyłącznie tutorialami pochodzącymi z DirectX SDK. Postaram się połączyć wszystko co najlepsze, czyli prostotę i atrakcyjność tutoriali z nehe razem z nowościami panów z MS. Będzie więc bardziej po kolei, ale z wykorzystaniem najnowszej technologii czyli DirectX 8. Dzisiaj bierzemy się za macierze, świat i transformacje, czyli "pokręcimy" trochę naszą sceną ;-).
Jak już mamy to w dobrym zwyczaju, zacznijmy od tego, co mamy na początku. W dzisiejszej lekcji zmałpuję trochę z nehe i będzie to podobny przykład do tego, prezentującego obroty w przestrzeni 3D.

```
#include <d3d8.h>

LPDIRECT3D8             g_pD3D       = NULL;
LPDIRECT3DDEVICE8       g_pd3dDevice = NULL;
LPDIRECT3DVERTEXBUFFER8 g_pVB        = NULL;
D3DXMATRIX              matWorldX;
D3DXMATRIX              matWorldY;

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

#define D3DFVF_CUSTOMVERTEX ( D3DFVF_XYZRHW | D3DFVF_DIFFUSE )
```

Sam początek nie wnosi na razie dla nas nic nowego, prócz macierzy (ang. *matrices*), jednak jest to temat dosyć obszerny, więc opisałem go w dalszej części artykułu, która jest dosyć rozległa.

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

Jeśli chodzi o ogólne zmienne, to raczej niewiele się dla nas w tym momencie zmienia. Będziemy więc potrzebować, jak zwykle, globalnego obiektu Direct3D, obiektu urządzenia no i od czasu, kiedy zaczęliśmy malować figury - obiektu bufora wierzchołków.

```
D3DXMATRIX matWorldX;
D3DXMATRIX matWorldY;
```

Zmienne reprezentujące nasze macierze, które posłużą do przekształcania naszego świata zgodnie z naszymi zachciankami. Jeszcze trochę cierpliwości, już za kilka minut wszystko wyjaśni się bliżej.

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

Jeśli chodzi o wierzchołki, to na pierwszy rzut też nie wygląda to jakoś rewolucyjnie, ale jeśli tak dokładniej się przyjrzeć i porównać z poprzednim przykładem to... tak widać, że brakuje nam zmiennej "rhw", która była potrzebna Direct3D przy automatycznym przeliczaniu naszych współrzędnych. Ponieważ w tym przykładzie użyjemy po raz pierwszy nie przetransformowanych wierzchołków (będziemy to musieli w pewnym sensie zrobić sami), więc ta zmienna staje się nam zbędna. A jak wiadomo pamięć jest bardzo cenna, nie mówiąc już o szybkości... wyobraźmy sobie np. milion wierzchołków na scenie, a milion dodatkowych liczb do przeliczenia... no wiadomo...

```
#define D3DFVF_CUSTOMVERTEX ( D3DFVF_XYZ | D3DFVF_DIFFUSE )
```

Tutaj też oczywiście widzimy zmianę. Skoro będziemy uwzględniać tylko współrzędne w przestrzeni i kolor, musimy poinformować Direct3D o tym, co będzie umieszczone w buforze. Jest to chyba oczywiste, że musi się nam zgadzać format wierzchołków z tym, co zadeklarujemy w strukturze je opisującej. W przeciwnym przypadku program przy czytaniu z pamięci sugerując się takim, a nie innym formatem może zacząć nam czytać z pamięci różne śmiecie i w pewnym momencie pójść w jakże przez nas nie lubiane buraki.

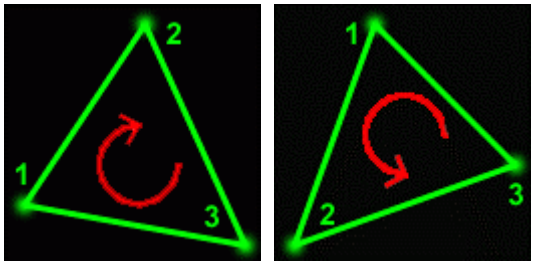
```
HRESULT InitD3D( HWND hWnd )
{
  if( NULL == ( g_pD3D = Direct3DCreate8( D3D_SDK_VERSION ) ) )
      return E_FAIL;
```

```
  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;

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

  return S_OK;
}
```

Funkcja inicjalizacyjna, patrząc globalnie, też się raczej nie zmieniła. Jak widać, przy samym jej końcu mamy jednak dwukrotne wywołanie ważnej metody obiektu urządzenia, która będzie miała w niektórych przypadkach decydujący wpływ na wygląd sceny. Jest to metoda **SetRenderState()**. Służy ona do... właściwie to do robienia mnóstwa rożnych rzeczy. Ogólnie rzecz biorąc ustawia się za jej pomocą określony parametr urządzenia renderującego (jej pierwszy argument typu **D3DRENDERSTATETYPE**) na określoną wartość (typu **DWORD**). Parametry urządzenia mogą być naprawdę przeróżne i jest ich całe mnóstwo, wystarczy zajrzeć do helpa. Z tych ważniejszych i przydatnych, zwłaszcza przy nauce, to między innymi włączanie bufora *Z* (ang. *z-buffer*), zakresów mgły (ang. *fog*), włączanie przezroczystości, zarządzanie buforem szablonu (ang. *stencil buffer*) i światłem (ang. *light*) oraz mnóstwo innych mniej lub bardziej interesujących nas parametrów. My omówimy sobie teraz jednak tylko te, które zastosujemy, a jak przyjdzie czas, to na pewno nie obejdzie się bez następnych.

```
g_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_NONE );
```

Jak nadmieniłem na początku, będziemy na naszej scenie coś obracać. Jeśli pewien przedmiot obraca się wokół jakiejś osi, to wiadomo, że po pewnym czasie powinna pokazać się nam druga strona tego obiektu. Grafika 3D wprowadziła do komputerów trzeci wymiar właśnie po to, aby można było oglądać obiekty ze wszystkich stron. Jak już nam wiadomo, obiekty takie składają się z serii trójkątów, które przylegając do siebie tworzą na ekranie złudzenie obiektu przestrzennego. My widzimy niejako tylko "skorupę" okrywającą cały szkielet, złożony z wierzchołków. Aby jednak móc dojść końca z takimi obiektami, trzeba było wprowadzić pewne zasady, które pomagają jakoś panować nad natłokiem coraz większej liczby trójkątów na scenach całego świata. Powiedzmy sobie więc trochę o niewidocznych krawędziach. Gdybyśmy zawsze rysowali wszystkie możliwe trójkąty na scenie, to nasza scena wyglądałaby jak ich bezładne zbiorowisko bez jakiegokolwiek porządku. Obiekt owszem, może i byłoby widać, ale przy skomplikowanych modelach mogę Was zapewnić, że nie jest to miły widok :-). Wymyślono więc coś takiego jak bufor *Z*. Jak to działa dokładnie, to powiemy sobie w którejś z kolejnych lekcji, tutaj jednak tylko nadmienię, że działanie bufora *Z* opiera się na tym, że współrzędna z, każdego kolejno rysowanego piksela na ekranie, jest badana ze współrzędną piksela, który znajduje się w tym samym miejscu na ekranie i jego współrzędna jest już umieszczona w takim buforze. Jeśli współrzędna z nowego piksela jest mniejsza od poprzedniej, to kolor i ta współrzędna zostają zapisane jako nowe w buforze, przykrywając tym samym poprzednie wartości, jeśli nie - zostaje stary kolor i stara współrzędna. W ten sposób trójkąty, które są "dalej" od nas, są zamalowywane przez te leżące niejako "bliżej". Cała grafika 3D to jednak mnóstwo obliczeń, nierzadko bardzo czasochłonnych, więc czasem bufor *Z* to jednak za mało. Musimy tu zauważyć, że wypełnianie takiego bufora *Z* przy dużych rozdzielczościach i wielu trójkątach musi pochłaniać stanowczo zbyt dużo czasu. Aby ułatwić sobie życie wymyślono więc jeszcze coś takiego jak kolejność rysowania wierzchołków. Jest to technika, która pozwala w dużym stopniu przyśpieszyć wykrywanie niewidocznych krawędzi i w połączeniu z buforem *Z* daje zupełnie zadowalające rezultaty. Na czym to polega? Już tłumaczę: ponieważ każdy trójkąt to 3 wierzchołki, więc wymyślono, że aby zobaczyć taki trójkąt namalowany na ekranie, to trzeba te wierzchołki narysować w odpowiedniej kolejności. Dla przykładu ktoś powiedział: słuchajcie, żeby trójkąty były widoczne, to ich wierzchołki będą musiały być rysowane zgodnie z ruchem wskazówek zegara. Pomysł bardzo prosty a jakże skuteczny. Teraz jeśli namalujemy dla przykładu trójkąt w przeciwną stronę (przeciwnie do ruchu wskazówek zegara), to go po prostu nie zobaczymy!



Dwa powyższe rysunki pokazują zaistniałą sytuację. Jeśli zastosujemy do tych trójkątów naszą konwencję pokazywania widocznych ścian, drugi z tych trójkątów, w którym wierzchołki ułożone są przeciwnie do ruchu wskazówek zegara, nie pokaże się nam na ekranie. Nietrudno zauważyć następnie, że jeżeli obrócimy któryś z naszych trójkątów wokół osi *X* lub *Y*, to kolejność ich wierzchołków zmienia się na przeciwną i tym samym stają się one niewidoczne! Prawda, że proste? W połączeniu z buforem *Z* metoda ta będzie nam bardzo pomocna przy malowaniu bardziej przejrzystych wypełnionych modeli, które już niedługo zaczną się pojawiać na naszej scenie. Jednak rozpisałem się już aż nadto, a miałem omówić metodę **SetRenderState()** z parametrami dotyczącymi definiowania sposobu ukrywania niewidocznych ścian stosując algorytm kolejności wierzchołków. Ponieważ wiemy już, że nasze pakiety, zarówno DirectX jak i OpenGL są w miarę uniwersalne, pozwalają nam więc zdefiniować sposób pracy takiego algorytmu. Możemy zażyczyć sobie uwidocznienie ścian rysowanych tylko w określonym porządku wierzchołków - w prawo lub w lewo a także wyłączyć ukrywanie niewidocznych ścian. I w naszym przykładzie tak właśnie zrobimy. Ponieważ nasze figury będą płaskie i dodatkowo będą się obracać, więc aby nie wyglądało to brzydko, wyłączymy sobie ukrywanie niewidocznych ścian za pomocą metody ustawiającej parametry urządzenia. Stała **D3DRS_CULLMODE** mówi ww. metodzie jakiego parametru urządzenia będzie tyczyła się wartość podana jako drugi jej argument. W naszym przypadku wiadomo już, że będzie się ona tyczyła ukrywania ścian, które uznamy, za niewidoczne. Drugi parametr to wartość mówiąca w jaki sposób urządzenie ma działać dla tego określonego parametru. W naszym przypadku ustawienie stałej **D3DCULL_NONE** spowoduje to, że urządzenie będzie rysować zarówno ściany tylne jak i przednie. Możemy też, w ramach eksperymentów, spróbować ustawić pozostałe dwie wartości, które mamy do dyspozycji czyli **D3DCULL_CW** (ściany będą niewidoczne, jeśli wierzchołki je tworzące będą widoczne na scenie w kolejności zgodnej z ruchem wskazówek zegara) lub **D3DCULL_CCW** jeśli ich kolejność będzie przeciwna. Tak więc zagorzali eksperymentatorzy mogą popróbować swoich sił.

```
g_pd3dDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
```

Drugą sprawą jest światło. Ponieważ używamy w naszej scenie wierzchołków nieoświetlonych, musimy wyłączyć światło, które domyślnie złośliwie jest włączone w Direct3D. Do tego także posłuży nam metoda **SetRenderState()**, której wywołanie widzimy powyżej. Co oznaczają poszczególne parametry nie muszę już chyba szczegółowo wyjaśniać. W ten właśnie sposób pozbywamy się wrednego światła, które spowodowałoby tylko niepotrzebne zamieszanie na naszej scenie.

```
HRESULT InitVB()
{
  CUSTOMVERTEX g_Vertices[] =
  {
    // triangle
    {-1.0f,-1.0f, 0.0f, 0xffff0000, },
    { 1.0f,-1.0f, 0.0f, 0xff0000ff, },
    { 0.0f, 1.0f, 0.0f, 0xff00ff00, },
    // square
    {-1.0f,-1.0f, 0.5f, 0xffff00ff, },
    {-1.0f, 1.0f, 0.5f, 0xffff00ff, },
    { 1.0f,-1.0f, 0.5f, 0xffff00ff, },
    { 1.0f, 1.0f, 0.5f, 0xffff00ff, },
  };

  if( FAILED( g_pd3dDevice->CreateVertexBuffer( 3*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;
}
```

W funkcji inicjującej nasz bufor wierzchołków nie zmienia się nic poza liczbą malowanych na scenie wierzchołków. Narysujemy sobie tutaj dwie figury (trójkąt i kwadrat), więc potrzebne będzie nam 7 wierzchołków. Zauważmy także, że pomimo iż malujemy różne obiekty wszystkie współrzędne ładujemy do jednego bufora.

```
VOID Cleanup()
{
  if( g_pVB != NULL )
      g_pVB->Release();
  if( g_pd3dDevice != NULL )
      g_pd3dDevice->Release();
  if( g_pD3D != NULL )
      g_pD3D->Release();
}
```

Ponieważ już nie wiem co mam wymyślić nowego :-), więc po prostu skopiuję to, co napisałem w poprzedniej lekcji. W następnych sądzę, że damy sobie już z tym spokój, bo nie będziemy w kółko wałkować tego samego... W funkcji czyszczącej (wywoływanej na końcu aplikacji) zwalniamy obiekt bufora wierzchołków. Dobrym zwyczajem, w programowaniu z wykorzystaniem obiektów COM, jest zwalnianie obiektów w kolejności odwrotnej do ich tworzenia, skoro więc utworzyliśmy obiekt bufora jako ostatni, zwolnijmy go jako pierwszy. Najważniejsze jest jednak to, aby obiekt **Direct3D** był zwalniany na końcu, jeszcze raz o tym przypominam, bo to bardzo ważne!!!

```
VOID SetupMatrices()
{
  D3DXMATRIX matView;
  D3DXMATRIX matProj;

  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 );

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

Teraz zajmiemy się kolejną nową rzeczą, bez której ani rusz w grafice 3D, czyli wykorzystaniem rachunku macierzy w naszych przekształceniach. Od tej pory słowo "macierz" będzie się pojawiać w naszych lekcjach nader często, więc postarajcie się zrozumieć najlepiej jak możecie :-). Przyszedł też czas, aby wykorzystać pierwsze metody z pomocniczej biblioteki **D3DX**, o której już była mowa we wstępie do DirectX-a.

```
D3DXMATRIX matView;
D3DXMATRIX matProj;
```

Czym jest macierz? Wielbiciele Matriksa zapewne doskonale wiedzą czym jest to tajemnicze coś, jednak w naszym przypadku będziemy posługiwać się bliższą naszemu sercu definicją matematyczną. Macierz to po prostu tablica liczb. W naszym przypadku jest to tablica dwuwymiarowa, posiadająca kolumny i rzędy (wiersze). Zapomniałem dodać, że aby zrozumieć choć trochę tę lekcję, potrzebny będzie nam podręcznik do matematyki, choćby najprostszy, ale traktujący o rachunku macierzy. Wielu zapyta po co nam taka macierz... Jak już niejednokrotnie wspominałem, aby narysować coś, co daje na ekranie złudzenie trójwymiarowości, trzeba wykonać sporą ilość obliczeń. Są to układy równań (przeważnie trzech). Moglibyśmy, co prawda, robić wszystko ręcznie, wcale nie jest powiedziane, że tak nie można. Ale ci, którzy liznęli choć trochę matematyki, wiedzą, że istnieje coś takiego jak metoda macierzowa rozwiązywania układów równań. Sprowadza się ona do mnożenia odpowiednich elementów macierzy i daje w wyniku pożądane przez nas dane. Ponieważ metoda ta ma szczególne predyspozycje aby zaadoptować ją w zastosowaniach komputerowych, więc sięgnięto po nią i tutaj. Zastosowanie macierzy znakomicie ułatwia, upraszcza a przede wszystkim przyspiesza obliczenia. W naszej praktyce spotkamy macierze, które posiadają 4 rzędy i od 1 do 4 kolumn. Poszczególne elementy w takich macierzach będą reprezentować nam współrzędne w przestrzeni lub wartości, przez które będziemy je przemnażać, aby otrzymać określone efekty.
W Direct3D mamy oczywiście obiekt, który reprezentuje macierz. I tutaj w sukurs po raz pierwszy przyjdzie nam pomocnicza biblioteka, o której już wspominaliśmy czyli **D3DX**. Zawiera ona sporo struktur, funkcji i interfejsów, które okażą się nam niezwykle pomocne. Jedną z takich struktur jest właśnie **D3DXMATRIX**. Wprawdzie w czystym Direct3D także mamy podobną strukturę (**D3DMATRIX**), ale w porównaniu do tej ma ona sporo ograniczeń. Przede wszystkim, jak zapewne wielu studentów wie ;-) - mnożenie macierzy. Jest to zmora początkujących programistów, którzy podczas nauki w szkole są katowani procedurami do operacji na macierzach a ich mnożenie to już szczególnie znienawidzona dziedzina.

Jednak **D3DX** daje nam do ręki strukturę, która posiada wszystkie potrzebne operatory! Będziemy więc mogli dodawać nasze macierze, odejmować, mnożyć a nawet je dzielić! I to wszystko za pomocą jednego operatora. Zapis "matX*matY" na pewno będzie wyglądał lepiej i bardziej przejrzyście niż wywołanie funkcji: MnozMacierz(matX, matY) - prawda?
Odpowiednie wypełnienie pól składowych macierzy, aby osiągnąć określony efekt, będzie się wiązało czasem z niemałym wysiłkiem (będzie kiedyś o cieniach to zobaczycie ;-), ale po cóż mamy naszą kochaną bibliotekę narzędziową. Dostaniemy do ręki funkcje, które same, po podaniu kilku potrzebnych danych, wstawią do takiej macierzy potrzebne wartości a nam pozostanie tylko użycie takiej macierzy w programie. Zapewniam Was, że będzie to naprawdę banalnie proste! Żeby zatem nie przeciągać za bardzo, może ruszmy wreszcie z miejsca.

Gdy patrzymy na scenę z określonego miejsca, to wiadomo - obiekty ułożą się w jakiś tam sposób, jeśli zaczniemy się przesuwać (jako obserwator) obiekty będą zmieniać swoje położenie także. To samo w przypadku zmiany miejsca patrzenia (zmieniamy na przykład obiekt naszego zainteresowania) lub gdy obiekty zaczynają się poruszać (uciekają przed nami? :-)
Jak policzyć takie rzeczy? Otóż będziemy potrzebować do tego celu trzech macierzy:

1. Macierz świata (ang. *world transformation matrix*).
Macierz ta odpowiedzialna jest za umieszczanie obiektów na naszej scenie i wszelkie ich transformacje. Co to znaczy? Jeśli mamy jakiś obiekt na naszej scenie i chcemy go na przykład obrócić, to mamy kilka wyjść. Możemy napisać oczywiście funkcję, która przemnoży nam poszczególne punkty obiektu przez odpowiednie równania i obiekt się obróci, tylko po co to. Przecież my możemy sobie obrócić cały świat - a dokładniej pomnożyć macierz świata przez macierz obrotu i w rezultacie na ekranie wszystko zacznie się obracać. Jak spowodować, aby obracał się tylko określony obiekt, będzie pokazane w dzisiejszej lekcji, wystarczy powiedzieć, że robi się to bardzo prosto. Tak samo ma się sprawa z przesuwaniem i skalowaniem obiektów. Nie mówię już o składaniu takich przekształceń, ale to sprowadza się przecież do banalnego mnożenia macierzy!

2. Macierz widoku (ang. *view transformation matrix*).
Macierz ta odpowiada za ustawienie kamery na scenie. Innymi słowy wystarczy nam skonstruować odpowiedni układ równań (macierz ma się rozumieć), której ustawienie jako macierzy widoku spowoduje takie przeliczenie współrzędnych obiektu, że w końcowym efekcie na ekranie dostaniemy wrażenie bycia w pewnym określonym punkcie i patrzenia w pewnym określonym kierunku.

3. Macierz rzutowania (ang. *projection transformation matrix*).
Wiadomo przecież, że ekran jest płaski a nie żaden tam 3D, więc skąd wrażenie głębi? Po to mamy właśnie macierz rzutowania. To nic innego jak układ równań, z którego, po wstawieniu do niego naszych współrzędnych, wyjdą tylko dwie: x i y, które zostaną namalowane na ekranie w odpowiednim miejscu, dając wrażenie posiadania obiektu trójwymiarowego. Jeśli mamy nasz punkt wejściowy, posiadający współrzędne: x, y, z po prostu przez sam Direct3D jest to on kolejno przekształcany przez 3 układy równań (reprezentowane przez powyższe macierze) aby w końcowej fazie osiąść na scenie we właściwym miejscu. Po kolei więc najpierw jest określane jego miejsce w świecie, następnie współrzędne są przeliczane, aby sprawiać wrażenie patrzenia na niego z określonego miejsca a na końcu współrzędne te są zamieniane na 2D w procesie rzutowania i umieszczane na ekranie - prawda, że proste? :-)

Dawniejszymi czasy, kiedy nie było jeszcze biblioteki **D3DX**, wszystkie te macierze konstruowało się ręcznie. Często wystarczyło się pomylić w jednej literce czy oznaczeniu pola macierzy i godzinami zastanawiało się dlaczego nic nie widać na ekranie, a to po prostu któraś z macierzy była źle policzona. Dziś mamy w ręce narzędzia, które za jednym zamachem tworzą nam odpowiednią macierz bez możliwości jakiejkolwiek pomyłki i znakomicie przyśpieszają tworzenie programów. I teraz właśnie omówimy sobie co to za narzędzia:

```
D3DXMatrixLookAtLH( &matView, &D3DXVECTOR3( 0.0f, 3.0f,-5.0f ),
                              &D3DXVECTOR3( 0.0f, 0.0f, 0.0f ),
                              &D3DXVECTOR3( 0.0f, 1.0f, 0.0f ) );
```

Pierwsza funkcja, jaką wykorzystamy, będzie funkcją służącą do wygenerowania potrzebnej nam macierzy widoku (odpowiedzialnej za kamerę - przypominam). Już sama jej nazwa po części wskazuje (LookAt) do czego służy. Ponieważ na początku umawialiśmy się na lewoskrętny układ współrzędnych (ang. *left-handed coordinate system*), więc tutaj zastosujemy tę wersję funkcji, która da nam macierz właściwą właśnie dla lewoskrętnego układu współrzędnych.
Funkcja pobiera cztery argumenty: pierwszy, to jak nie trudno już zgadnąć, adres zmiennej typu **D3DXMATRIX**, która będzie zawierać naszą macierz widoku. Trzy następne to trochę dziwnie wyglądające stwory, ale w taki właśnie sposób podajemy trzy kolejne wektory, które będą nam potrzebne przy definiowaniu kamery (wywołujemy jawnie konstruktor struktury **D3DXVECTOR**, która zawiera trzy składowe wektora: x, y, z). Pierwszy z nich to punkt, w którym położone będzie nasze oko. Ktoś zawoła, zaraz, zaraz mówisz punkt, a przecież to jest wektor. No tak racja, ale zauważmy, że jeśli połączyć środek układu współrzędnych z jakimś punktem w przestrzeni, to współrzędne (składowe) wektora utworzonego w ten sposób są czym? No właśnie, niczym innym jak tylko współrzędnymi tego punktu. Drugi wektor pokazuje miejsce, na które będziemy patrzeć, oczywiście współrzędne podane mamy w ten sam sposób. Trzeci parametr jest zdefiniowany jako "góra świata" (czyli kierunek pionowy). Jest on potrzebny do wyliczenia macierzy widoku, ponieważ dwa wektory to za mało a on sam po prostu pokazuje w danym momencie kierunek, który uznajemy za górę naszego świata. Przeważnie tę wartość ustawiamy na $x = 0$, $y = 1$ i $z = 0$, co jest chyba zgodne z logiką.

```
g_pd3dDevice->SetTransform( D3DTS_VIEW, &matView );
```

Ponieważ wszystkie nasze operacje na scenie i świecie to tylko zestaw transformacji, więc mając już macierz widoku możemy powiedzieć o tym Direct3D. Metodą **SetTransform()** obiektu urządzenia powodujemy to, że wewnętrzne metody przekształcające naszą scenę jako macierz widoku przyjmą tą, przez nas zdefiniowaną. Metoda ta pobiera dwa argumenty: pierwszy mówi Direct3D jakiego typu macierzą ma być dla niego macierz podana jako drugi argument. Stała **D3DTS_VIEW** oznacza, że (jak już nadmieniliśmy) będzie ona macierzą widoku.

```
D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
```

Następnie zdefiniujemy sobie sposób rzutowania przekształconych przez nas punktów 3D na płaszczyznę ekranu. Będziemy potrzebować do tego macierzy rzutowania. I znowu nie musimy kombinować ze skomplikowanymi obliczeniami, wystarczy nam jedna funkcja **D3DX**, która da nam do ręki odpowiednią macierz. Funkcja **D3DXMatrixPerspectiveFovLH()** (ależ skomplikowana nazwa :-). Pobiera jako argumenty kolejno: kąt widzenia, jaki obejmuje kamera (podany w radianach, w naszym przypadku PI/4), drugi argument to stosunek wysokości do szerokości ekranu (przy nie kwadratowych oknach należy go policzyć!), natomiast dwa ostatnie argumenty oznaczają najbliższą i najdalszą płaszczyznę (ang. *near clipping plane*, *far clipping plane*), poza zakresem których obiekty nie będą po prostu widoczne. Jeśli więc zachce nam się narysować nagle obiekt, który leży o 120 jednostek od nas, to po prostu go nie zobaczymy. Wbrew pozorom jest to bardzo pożyteczne, bo pozwala pozbyć się w prosty sposób ze sceny zbędnych obiektów i tym samym zaoszczędza jakże cenny dla nas czas.

```
g_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
```

I tak samo, jak w przypadku macierzy widoku, mówimy urządzeniu wykorzystując wyżej wspomnianą metodę o tym, że ta macierz będzie dla nas macierzą rzutowania na ekran. Direct3D wykorzystując te dane odpowiednio wypełni macierz a kiedy nadejdzie czas, pomnoży przez nią współrzędne punktów i ukażą się one naszym spragnionym oczom.

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

  D3DXMatrixTranslation( &matWorldX, 1.5f, 0.0f, 0.0f );
  D3DXMatrixRotationY( &matWorldY, timeGetTime()/150.0f );
  g_pd3dDevice->SetTransform( D3DTS_WORLD, &(matWorldY*matWorldX) );
  g_pd3dDevice->SetStreamSource( 0, g_pVB, sizeof(CUSTOMVERTEX) );
  g_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
  g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );

  D3DXMatrixTranslation( &matWorldX,-1.5f, 0.0f, 0.0f );
  D3DXMatrixRotationX( &matWorldY, timeGetTime()/150.0f );
  g_pd3dDevice->SetTransform( D3DTS_WORLD, &(matWorldY*matWorldX) );
  g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 3, 2 );

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

Ponieważ mamy już ustawione dwie z trzech naszych macierzy, czas przystąpić do rzeczy właściwej grafice 3D czyli animacji! Wytłumaczę teraz tylko nowe rzeczy, ponieważ lekcja i tak jest dosyć długa a nie ma powodu, aby ją przeciągać. Po pierwsze - będziemy animować czyli w naszym przypadku obracać, a jak za chwilę się okaże, także przesuwać obiekty. Jak wcześniej napisałem do transformacji obiektów na scenie posłuży nam macierz świata. Napisałem także, że to ona pierwsza znajduje się w łańcuchu, przez który przechodzą kolejno wierzchołki. Dlaczego więc tworzymy nasze macierze dopiero w funkcji rysującej? Pomyślmy przez chwilę... punkty czy obiekty, która zostaną wyświetlone zostaną przeliczone dopiero podczas bezpośredniego ich wyświetlania (metoda **DrawPrimitive()**), więc jest nam w zasadzie to obojętne, kiedy nastąpi ich transformacja w świecie, ważne aby było to przed wywołaniem metody rysującej! Poza tym aby uprościć sobie sprawę, posłużymy się czasem systemowym do określenia kąta obrotu, więc trzeba to obliczać w funkcji, która jest wywoływana co jakiś czas, a Render() nadaje się do tego chyba najlepiej :-).
Po drugie - aby trochę urozmaicić przykład i zmusić Was do większego wysiłku umysłowego, a także aby zaprezentować moc biblioteki **D3DX**, pomnożymy sobie macierze. Mamy dwa obiekty, ale żeby nie robić przykładu prymitywnym, jeden obrócimy sobie wokół osi *X*, a drugi wokół osi *Y*. Ponieważ zdefiniowaliśmy oba na środku układu współrzędnych, aby nie nakładały nam się na siebie, rozsuniemy je trochę w naszym świecie.

```
D3DXMatrixTranslation( &matWorldX, 1.0f, 0.0f, 0.0f );
```

Zmienna "matWorldX" będzie reprezentować naszą macierz przesuwania wzdłuż osi *X*. Funkcja **D3DXMatrixTranslation()** obliczy nam macierz translacji (czyli przesunięcia) o podany jako trzy ostatnie argumenty wektor.

```
D3DXMatrixRotationY( &matWorldY, timeGetTime()/150.0f );
```

Następnie obliczymy sobie macierz obrotu wokół osi. Funkcja **D3DXMatrixRotationY()** da nam do ręki macierz, która, jeśli zostaną przez nią pomnożone punkty naszego obiektu, spowoduje jego obrót w trójwymiarowym świecie. Wykorzystujemy tu funkcję **timeGetTime()**, która zwraca czas jaki upłynął od uruchomienia systemu. Ponieważ funkcja **Render()** jest wywoływana co jakiś czas (bardzo krótki), więc wartości zwracane przez tą funkcję są dla nas rosnącymi wartościami kąta, podanego jako drugi argument. Rozwiązanie co prawda nieeleganckie, ale dla naszego szkoleniowego przykładu zupełnie wystarczające.

```
g_pd3dDevice->SetTransform( D3DTS_WORLD, &(matWorldY*matWorldX) );
```

I tu uwaga! Teraz ustawiamy macierz świata (czyli faktycznie dokonujemy transformacji na naszym obiekcie!). Wiemy już co robi metoda **SetTransform()**, domyślamy się znaczenia stałej **D3DTS_WORLD**, więc co zrobić aby przetransformować nasz obiekt? Jak widać na przykładzie, wystarczy pomnożyć stworzone przez nas macierze, które w rezultacie działania przeciążonego operatora mnożenia * (gwiazdka) dadzą macierz wynikową, która zostanie ustawiona jako macierz świata. Jeśli czytaliście poprzednie lekcje uważnie, to wiecie, że kolejność mnożenia ma tu ogromne znaczenie. Ponieważ chcemy, aby obiekt obracał się nam w miejscu, musimy dokonać pewnej sztuczki. Zamiast kombinować i mnożyć różne równania na obroty wokół jakiegoś punktu, my obrócimy sobie obiekt w miejscu, w którym został utworzony, czyli w środku układu współrzędnych! Bez problemów i niepotrzebnych kombinacji. Następnie nasz obrócony o dany kąt obiekt po prostu przesuniemy w odpowiednie miejsce. Ponieważ nie rysujemy za każdym przekształceniem obiektu a jedynie składamy przekształcenia (obrót + przesuniecie), jako końcowy efekt możemy zauważyć obrót obiektu w miejscu, do którego został on przesunięty! I to wszystko dzieje się dzięki tak wspaniałemu wynalazkowi jak macierz. Mam nadzieję, że teraz wielu przestanie je przeklinać i doceni mękę nad funkcjami do ich mnożenia czy dodawania :-). Na końcu mając ustawione wszystkie już macierze, wywołujemy serię metod urządzenia, które już dobrze znamy czyli ustawienie źródła danych, ustawienie vertex shadera i w końcu narysowanie figury. Metoda **DrawPrimitive()**, mając już odpowiednie macierze, przeliczy nam współrzędne punktów przez odpowiednie równania (przemnoży przez macierze) i otrzymamy to, co na rysunku poniżej. Powyższe operacje wykonujemy oczywiście dla drugiego obiektu, który przesuwamy w drugą stronę i obracamy wokół innej osi. Ponieważ przy każdym wejściu do funkcji **Render()** współrzędne obiektów się nie zmieniają (przecież w buforze są cały czas te same), transformacje wykonywane na tych obiektach przy zmianie np. kąta obrotu wywołują pożądany przez nas skutek.

Mam nadzieję, że choć trochę wyjaśniła Wam się potrzeba zastosowania macierzy w grafice 3D, zrozumieliście na czym polega składanie przekształceń, poznaliście bardzo pożyteczne funkcje z biblioteki **D3DX**, no i święcie wierzę, że teraz kręci się na Waszych monitorach tysiące trójkątów na wszelkie możliwe sposoby :-).

Jak zwykle dołączam gotowy projekt, po skompilowaniu i uruchomieniu którego powinniście otrzymać na ekranie załączony poniżej obrazek.