Witam w dziesiątym już odcinku naszego kursu. Powoli zbliżamy się do końca naszej podróży po podstawach wykorzystania tej biblioteki w praktycznym programowaniu grafiki 3D czasu rzeczywistego. Dziś powiemy sobie o rzeczy, bez której też już nie da się dzisiaj zrobić na nikim wrażenia a mianowicie o mieszaniu (ang. *blending*). Ktoś krzyknie - zaraz, zaraz... miało być o szybach, szklankach i kieliszkach a będzie o mieszaniu? Tak, tak - bo to właśnie dzięki mieszaniu będziemy mogli uzyskać efekt przezroczystości szyby a przyda nam się to w wielu innych sytuacjach również. Ale najpierw opanujmy podstawy, a od najbliższej lekcji zajmiemy się bardziej zaawansowanymi zagadnieniami. Swoją drogą będzie to lekcja trochę z oszukiwaniem, ponieważ trochę sobie w niej uprościmy (czyt. oszukamy), ale mam nadzieję, że DirectX wybaczy nam takie małe lenistwo ;-).

```
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
```

Nasze zmienne. W zasadzie nie zmieniło się nic i doskonale wiemy, po co nam to wszystko. W dzisiejszym odcinku wykorzystamy teksturę do uzyskania efektu przezroczystej ściany. Gdybyśmy wzięli "goły" sześcian, to nasz efekt przezroczystości może i miałby miejsce, ale nie byłby już tak efektowny, jak będzie po zakończeniu tego odcinka. Obiekt urządzenia, obiekt Direct3D i bufor wierzchołków znamy już doskonale i wiemy, po co nam to potrzebne.

```
// A structure for our custom vertex type struct
CUSTOMVERTEX
{
  FLOAT x;
  FLOAT y;
  FLOAT z;
  DWORD color;
  FLOAT tx;
  FLOAT ty;
};

#define D3DFVF_CUSTOMVERTEX ( D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_TEX1 )
```

Nasza struktura dla wierzchołków. W zasadzie nic nowego tu nie wprowadzimy bo i nie ma takiej potrzeby. Będziemy mieć oczywiście pozycje wierzchołków w przestrzeni, ich kolor, który weźmie udział w mieszaniu oraz współrzędne teksturowania, które posłużą do nałożenia tekstury. Na koniec definiujemy nasz własny typ wierzchołków **FVF** (***F**lexible **V**ertex **F**ormat*) wykorzystując te standardowe, znajdujące się w bibliotece.

```
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;
}
```

Następnie dobrze nam znana już funkcja do inicjalizacji urządzeń. Nie zmienia się też nic. Z ważniejszych rzeczy, które powinniśmy sobie utrwalić: wyłączamy ukrywanie ścian, korzystając z metody kolejności trójkątów (skoro obiekty mają być przezroczyste, więc nie ma sensu ukrywać ich odwróconych ścian), wyłączamy światło, żeby za dużo się nie opisać i włączamy bufor Z. Pozostała cześć jest identyczna jak poprzednio. Ogólnie cały przykład jest podobny do poprzedniego, z małymi tylko różnicami w funkcji renderującej.

```
HRESULT InitGeometry()
{
  CUSTOMVERTEX g_Vertices[] =
  {
    // back
    { 1.0f,-1.0f, 1.0f, 0x07FFFFFF, 0.0f, 1.0f,},
    { 1.0f, 1.0f, 1.0f, 0x07FFFFFF, 0.0f, 0.0f,},
    {-1.0f,-1.0f, 1.0f, 0x07FFFFFF, 1.0f, 1.0f,},
     ...
    {-1.0f, 1.0f,-1.0f, 0x7FFFFFFF, 0.0f, 0.0f,},
    { 1.0f,-1.0f,-1.0f, 0x7FFFFFFF, 1.0f, 1.0f,},
    { 1.0f, 1.0f,-1.0f, 0x7FFFFFFF, 1.0f, 0.0f,},
  };

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

  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;
}
```

Nasza figura. Tak jak w poprzednim przykładzie, definiujemy sześcian, który posłuży nam do zaprezentowania efektu półprzezroczystej ściany. Najpierw definiujemy nasze wierzchołki, które posłużą do zbudowania bryły, następnie ładujemy teksturę z pliku znajdującego się na dysku. Następnie już lecimy standardowo - bufor wierzchołków, blokujemy, kopiujemy do niego z pamięci, odblokowujemy i gotowe. Funkcji czyszczącej i ustawiającej macierze już nie muszę chyba omawiać. Są one identyczne jak kilka przykładów wstecz, więc nie będziemy się już nimi zajmować do czasu aż zmienimy naszą politykę dotyczącą wyświetlania wierzchołków - to znaczy kiedy zastosujemy vertex shader, czyli, ogólnie mówiąc, sposób przekształcania wierzchołków.
Jedyną zmianą, jaką da się zauważyć w stosunku do poprzedniego przykładu, jest inna wartość koloru dla poszczególnych wierzchołków. Wielu podejrzliwych zapewne zada znów pytanie jak najbardziej na miejscu - dlaczego? Jak pewnie nietrudno się domyśleć, będzie miało to jakiś, na razie bliżej nieokreślony, związek z naszą dzisiejszą lekcją i nie będą się mylić. Ta wartość posłuży nam właśnie do uzyskania efektu przezroczystości.

Trąbimy o tej przezroczystości już dłuższy czas, ale ciągle nie wiemy skąd ona się właściwie weźmie. Jak napisałem na początku, uzyskamy ją dzięki zmieszaniu kolorów. Ktoś dociekliwy znowu zacznie wnikać - skąd te kolory, jak je będziemy mieszać i po co nam to w ogóle. Już wszystko wyjaśniam. Kiedyś ktoś mądry wymyślił coś takiego, co zwie się **kanałem alfa** (ang. *alpha channel*). Pewnie nie raz zadawaliście sobie pytanie, co to za diabeł i po co on nam robi zamieszanie w naszym jakże prostym i logicznie pojmowanym świecie barw, złożonych z trzech podstawowych kolorów - czerwonego (**R**ed), zielonego (**G**reen) i niebieskiego (**B**lue). Kiedy gramy w stare dobre gierki, nie zastanawiamy się prawie nigdy nad tym jaką one mają grafikę, jak realizowane są poszczególne efekty. Czasem tęsknię za tymi dobrymi czasami, kiedy nie trzeba było się martwić o to, jak zmieszać kolory żeby otrzymać szklankę wyrenderowaną w czasie rzeczywistym :-). Pogoń za doskonałością doprowadziła i w grafice komputerowej do rożnych szalonych wynalazków, jednym z pierwszych był właśnie **kanał alfa**. Cóż to jest takiego? Każdy parający się obróbką grafiki, doskonale wie, co to są składowe koloru, czy to

RGB, czy **CMYK**, czy też jeszcze inne. Każdy kolor, który widzimy na ekranie, jest złożony z trzech barw podstawowych, w przypadku monitorów - czerwonego, zielonego i niebieskiego. Wiele razy na pewno bawiliście programami typu Paint, Photoshop, Corel czy setką innych im podobnych. Tam operowanie składowymi koloru to jedna z podstawowych umiejętności, która gwarantuje już połowiczny sukces naszego przyszłego malarskiego dzieła. Ale ponieważ natura ludzka jest dosyć przewrotna, więc ktoś się kiedyś nudził i wymyślił Internet. Jeśli Internet, to oczywiście natychmiast zaczęły powstawać tysiące stron, mniej lub bardziej wymyślnych. Zaczęto stosować coraz bardziej kosmiczne technologie, aby przyciągnąć potencjalnych "oglądaczy" do swoich stron. Jednym z "bajerów", który miał zachwycić użytkowników stron były obrazki. Na początku strony dosyć monotematyczne i szablonowe, dopiero z czasem zaczęły nabierać... Zaraz, zaraz... miało być o kanale alfa, a ja ściemniam coś o Internecie. Ale już do rzeczy. Ktoś kiedyś wymyślił sobie, że można by zrobić sobie na przykład ładny, kolorowy napis na stronie w postaci obrazka. Miał on jednak wadę w postaci niepożądanego koloru w literach, które zawierały jakieś zamknięte powierzchnie - na przykład *P*, *O* i tym podobne. Wywnioskowano więc, że przydałoby się zrobić te właśnie elementy obrazka przezroczyste. Od tego nie było już daleko do kanału alfa. Jest to nic innego, jak jeszcze jedna składowa kolorów obrazka, która mówi nam w jakim stopniu dany fragment obrazka jest przezroczysty. Dla napisów oczywiście wystarczyłoby zdefiniowanie tylko tego, które obszary są przezroczyste a które nie. Jednak ludziom to oczywiście nie wystarczyło i poszli dalej. Doszli do wniosku, że skoro można potraktować składową kanału alfa jako jeden z elementów składowych obrazka, to czemu nie wykorzystać tego i nie zrobić rożnego stopnia tej przezroczystości. Przecież to wcale nie takie oczywiste, że skoro coś jest przezroczyste, to tego nie widać, prawda? Tak więc, dziś mamy do dyspozycji w grafice kolor "uzbrojony" niejako w jeszcze jedną składową - składową kanału alfa, która mówi jak bardzo obrazek jest przezroczysty. W większości przypadków ta składowa ma taką samą wielkość (jeśli chodzi o rozmiar), jak każda inna znajdująca się w kolorze. Jeśli więc na przykład składowa koloru **RGB** ma po 8 bitów, składowa alfa też ma tyle, co daje 256 rożnych stopni przezroczystości obiektu, który korzysta z dobrodziejstw kanału. Jak na razie nie wymyślono potrzeby stosowania większej ilości bitów na kanał alfa, bo to co jest i tak wystarcza do osiągania przez nas bardzo zadowalających rezultatów. A jak to wszystko ma się do naszego przykładu? Otóż w grafice 3D od początku wiadomo było, że kanał alfa na pewno się przyda. Wprawdzie pierwsze programy nie korzystały z tego, ale w momencie, w którym ktoś stwierdził, że można na scenie nieźle "zamieszać", zaczęły powstawać, jak grzyby po deszczu, programiki wykorzystujące w mniejszym lub większym stopniu właściwości kanału alfa. Dziś mieszanie, bo o tym w zasadzie należy powiedzieć, jako o źródle przezroczystości, stanowi podstawę uzyskiwania efektów, na które nie moglibyśmy sobie pozwolić mając do dyspozycji tylko to, czego nauczyliśmy się do tej pory. Mieszając będziemy robić przezroczyste obiekty, robić wypukłości, korzystać z multiteksturowania (ang. *multitexturing*). Efekty tego typu na pewno jeszcze będziemy mieli okazję zobaczyć, więc Wasza cierpliwość zostanie wystawiona na dużą próbę ;-). Ale powiedzmy sobie w końcu, na czym to całe mieszanie ma polegać. Otóż, żeby zamieszać na scenie, będziemy potrzebować dwóch kolorów. Jeden to wiadomo - będzie nim kolor naszej figury. Drugim będzie kolor umieszczony w buforze, w którym rysujemy. Wykorzystując odpowiednie metody spowodujemy to, że DirectX przy rysowaniu naszej figury, mając do dyspozycji kolor w buforze i naszej figury, odpowiednio zmiesza je i w wyniku otrzymamy to co chcemy - czyli półprzezroczysty sześcian. Wyobraźmy sobie, jak to będzie przebiegać. DirectX wykorzystując kolor naszych wierzchołków, materiału jaki nałożony jest na obiekt, światła umieszczonego na scenie i wreszcie tekstury, którą zaaplikujemy naszemu sześcianowi, będzie starał się narysować coś w buforze. Jednak my zrobimy mu małą niespodziankę i w metodzie renderującej nakażemy przeprowadzenie mieszania. On zaś będzie musiał wziąć pod uwagę wszystkie składniki, o których powiedziałem wyżej i coś z tym zrobić. Pierwsze na co popatrzy to kolor wierzchołków. I co zobaczy?

```
{ 1.0f,-1.0f, 1.0f, 0x07FFFFFF, 0.0f, 1.0f, }
```

Czwarta wartość w tablicy opisującej wierzchołek jest typu **DWORD**, czyli ma 32 bity. Ponieważ kolor w DirectX często właśnie będziemy opisywać za pomocą tego typu, więc wyjaśnijmy sobie co jest czym. Jak napisałem wcześniej, kiedy będziemy używać kanału alfa, będziemy mieli nie 3 a 4 składowe - czerwona, zielona, niebieska i alfę. Aby wystarczyło nam na obsługę grafiki w trybie nawet **True Color**, nasze składowe będą mieć rozmiar po 8 bitów każda. W zasadzie mogłoby się tyczyć to tylko składowych podstawowych (**RGB**), ale żeby nie zaciemniać sobie i nie tworzyć jakiś kosmicznych struktur, wszystko upchniemy w jedną liczbę. Nasza wartość będzie przechowywać, jak wspomniałem, wszystkie cztery składowe, ale w określonej kolejności. I w tym szczególnym przypadku liczba przechowuje w pamięci co następuje:

kanał alfa - czerwony - zielony - niebieski

Tak więc jeśli przyjrzymy się z bliska naszej liczbie, to otrzymamy następujące wartości:

kanał alfa = 7Fh = 127,
czerwony = FFh = 255,
zielony = FFh = 255,
niebieski = FFh = 255

Należy więc zwrócić uwagę, że kanał alfa podajemy jako pierwszy! Dlaczego tak, tego nie wiemy. Może po prostu ktoś miał taki kaprys a może jest to podyktowane jakimiś względami optymalizacji (na przykład szybszy odczyt kolejnych komórek). Ale przecież ważne jest tylko to, żeby zapamiętać w jakiej kolejności będziemy podawać te składowe. Ale wracając do DirectX-a i jego malowania. Pobrał więc wszelkie dane do pamięci, nakazaliśmy urządzeniu przeprowadzić mieszanie z wykorzystaniem kanału alfa, więc ono bierze się ochoczo do roboty. Mając wszelkie dane, oblicza ono kolor piksela, jaki należy umieścić w buforze. To znaczy:

kolor wierzchołka + kolor materiału + kolor światła + kolor tekstury.

i już, już chce umieścić obliczony w ten sposób piksel na ekranie i... no właśnie. I w tym momencie musi przecież zamieszać. Aby to zrobić, musi pobrać z miejsca w buforze, w którym zamierza postawić nowo obliczony piksel, kolor jaki tam się aktualnie znajduje. Grzecznie więc to robi i w tym momencie się zaczyna. Ma dwa kolory - ten obliczony i ten z którym musi zmieszać tego obliczonego. Nic prostszego - wykorzystując pewien wzór robi to bez najmniejszego problemu. A wzór ten wygląda następująco:

kolor docelowy = kolor obliczony* x *wsp1* + *kolor w buforze* x *wsp2

A co oznaczają poszczególne wartości?

kolor docelowy - jest to kolor jaki zostanie zapisany w buforze po przeprowadzeniu operacji mieszania,
kolor obliczony - jest to kolor jaki powstanie po przeprowadzeniu wszelkich obliczeń dotyczących kolorów na bryle,
wsp1 - jest to pewien współczynnik (o którym już za moment bliżej), przez który zostanie pomnożony kolor obliczony,
kolor w buforze - no to chyba wiadomo i nie muszę nikomu tłumaczyć,
wsp2 - taki sam (jeśli chodzi o rodzaj) jak w przypadku współczynnika wsp1.

We wzorze tym widzimy pewne tajemnicze współczynniki. Po co nam one? Rozwińmy może ten wzór bardziej szczegółowo. Dla uproszczenia ;-) wprowadźmy sobie też pewne oznaczenia:

RsC, ***GsC***, ***BsC***, ***AsC*** - składowe koloru źródłowego (obliczonego),
RsF, ***GsF***, ***BsF***, ***AsF*** - składowe współczynnika, przez który pomnożymy kolor,
RdC, ***GdC***, ***BdC***, ***AdC*** - składowe koloru pochodzącego z bufora,
RdF, ***GdF***, ***BdF***, ***AdF*** - składowe współczynnika, przez który pomnożymy kolor z bufora.

No i na czym będzie polegać działanie tych naszych współczynników? Mnożenie koloru obliczonego przez współczynnik będzie wyglądać następująco:

RsC x ***RsF*** - nowa składowa czerwona,
GsC x ***GsF*** - nowa składowa zielona,
BsC x ***BsF*** - nowa składowa niebieska,
AsC x ***AsF*** - nowa składowa niebieska,

Cały bajer tkwi w tym, że ten nasz współczynnik ***sF** jak i ***dF** może przybierać rożne wartości. W DirectX-ie jak i w OpenGL-u współczynniki te są bardzo podobne i w pewien sposób ustandaryzowane. Wartości poszczególnych składowych takiego współczynnika mogą przyjmować wartości ujęte w poniższej tabeli (podam tylko te, które na początek będą bardziej przydatne):

D3DBLEND_ZERO	0	0	0	0
D3DBLEND_ONE	1	1	1	1
D3DBLEND_SRCCOLOR	Rs	Gs	Bs	As
D3DBLEND_INVSRCCOLOR	1-Rs	1-Gs	1-Bs	1-As
D3DBLEND_SRCALPHA	As	As	As	As
D3DBLEND_INVSRCALPHA	1-As	1-As	1-As	1-As
D3DBLEND_DESTALPHA	Ad	Ad	Ad	Ad
D3DBLEND_INVDESTALPHA	1-Ad	1-Ad	1-Ad	1-Ad
D3DBLEND_DESTCOLOR	Rd	Gd	Bd	Ad
D3DBLEND_INVDESTCOLOR	1-Rd	1-Gd	1-Bd	1-Ad

Istnieje jeszcze kilka innych, ale są to dziwadła, którymi na razie i tak nie warto zawracać sobie głowy. Mamy więc nasze współczynniki, więc dowiedzmy się jak one działają. W przypadku **D3DBLEND_ZERO** i **D3DBLEND_ONE** chyba wszystko jest jasne. Składowe koloru zostaną pomnożone przez składowe współczynnika. Jak łatwo zauważyć, jeśli kolor pobrany z bufora pomnożymy przez **D3DBLEND_ONE** (w zasadzie pozostawimy bez zmian) a kolor obliczony przez **D3DBLEND_ZERO** (wyzerujemy), to oczywistym staje się, że nie zobaczymy naszej nowo rysowanej bryły. Jeśli zrobimy odwrotnie, to wiadomo, że kolory nowo rysowanej bryły w pełni (x1) przykryją nam kolory z naszego bufora, bo tych po prostu nie będzie (x0). W przypadku innych współczynników sprawa się oczywiście komplikuje nieco, ale za to efekty zaczynają być bardzo ciekawe ;-). **Rs**, **Gs**, **Bs**, **As** są to poszczególne składowe koloru, które są określone przy tworzeniu wierzchołków. U nas mamy je wszystkie ustawione na FFh oprócz kanału alfa właśnie, który ma wartość 7Fh. Jak widać, oprócz dwóch wyżej wymienionych rodzajów współczynników, wszystkie pozostałe w jakiś tam sposób korzystają właśnie z

kanału alfa przy mnożeniu i potem mieszaniu. Po pomnożeniu przez wartości mniejsze niż FFh, otrzymamy właśnie efekt przezroczystego obiektu, ponieważ nie wszystkie składowe koloru będą miały takie samo natężenie. Dla przykładu, jeśli pomnożymy wartość piksela obliczoną przez współczynnik o "wartości" **D3DBLEND_SRCALPHA**, to wszystkie kolory tego piksela zostaną przemnożone przez wartość kanału alfa. Jeśli będzie on mniejszy od FFh, to wszystkie składowe podstawowe stracą trochę na wartości.
Tak w najmniejszym skrócie wygląda idea całego tego mieszania. Jeśli chodzi o przezroczystość obiektów, to sprawa jest jeszcze w miarę prosta. Gorzej jeśli przyjdzie to wykorzystać przy tworzeniu mapy nierówności, ale może po kolei. Na pocieszenie mogę jeszcze dodać, że jeśli chodzi o samą przezroczystość, to jeszcze nie wszystko. Drugim wielkim problem jest kolejność rysowania przezroczystych obiektów. W zależności od kolejności ich narysowania, efekty mogą być całkowicie rożne! Jeśli mamy zamiar renderować na jednej scenie obiekty przezroczyste i nieprzezroczyste, to na pewno chcielibyśmy, aby nasz bufor *Z* pomógł nam w usuwaniu niewidocznych lub przysłoniętych powierzchni, które znajdują się za naszym przezroczystym obiektem. Jeśli jakieś nieprzezroczyste obiekty na scenie są przed przezroczystymi lub nie, to oczywiście życzymy sobie, aby one też były ukryte przed naszym wzrokiem. Jeśli obiekt przezroczysty jest bliżej, to oczywiście chcemy, aby został on "zmieszany" z tymi nieprzezroczystymi do otrzymania odpowiedniego efektu. Możemy sobie z tym oczywiście poradzić, kombinując z kolejnością rysowanych trójkątów na scenie, ale wszystko będzie pięknie do czasu kiedy albo obiekt, albo kamera nie zaczną się poruszać. Wtedy wszystko zaczyna się komplikować i staje się 10 razy trudniejsze... Rozwiązaniem problemu jest włączenie bufora *Z*, ale tylko przy rysowaniu obiektów nieprzezroczystych. Kiedy mamy zamiar rysować obiekty przezroczyste, należy zablokować zapis do bufora *Z*. Najpierw należy narysować obiekty nieprzezroczyste z włączonym buforem *Z*, następnie zablokować do niego zapis i rysować obiekty przezroczyste. Jeśli obiekty przezroczyste są za tymi nieprzezroczystymi, to nie są one malowane ponieważ ich współrzędne kwalifikują ich do tego, aby ich nie rysować. Jeśli są bliżej, nie przykrywają obiektów nieprzezroczystych ponieważ dane nie mogą być zapisane do bufora głębokości. Zamiast tego ich kolory są mieszane z tymi, które znajdują się w buforze (z bryłami nieprzezroczystymi). Więc na koniec powinienem zadać pytanie - czy wszyscy zrozumieli o co tu chodzi? :-)

```
VOID Render()
{
  g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
D3DCOLOR_XRGB(0,0,0), 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_MINFILTER, D3DTEXF_LINEAR );
  g_pd3dDevice->SetTextureStageState( 0, D3DTSS_MAGFILTER, D3DTEXF_LINEAR );
  g_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
  g_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );
  g_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG2 );
  g_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG2, D3DTA_CURRENT );

  g_pd3dDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );
  g_pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
  g_pd3dDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
  g_pd3dDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );

  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->SetRenderState( D3DRS_ZWRITEENABLE, TRUE );

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

No i nadszedł czas na funkcję renderującą a w niej na nowe, ważne w tej lekcji rzeczy. Ponieważ wiemy mniej więcej na czym ten cały blending, czyli mieszanie polega - przynajmniej tak w skrócie, więc omówmy sobie co jest czym.

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

Ten zestawik to w zasadzie znamy. Ustalamy tutaj sposób obróbki i nakładania naszej tekstury na bryłę. Modulujemy kolor naszej tekstury z kolorami wierzchołków, ustawiamy co będzie źródłem dla naszej operacji - pierwsze źródło to tekstura, drugie to kolor wierzchołków. Na koniec, żeby ładnie to wszystko wyglądało, przefiltrujemy sobie naszą teksturkę, aby sześcianik był piękny i gładki.

```
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG2 );
g_pd3dDevice->;SetTextureStageState( 0, D3DTSS_ALPHAARG2, D3DTA_CURRENT );
```

Metoda ciągle ta sama, tylko parametry się zmieniają, chciałoby się rzec. Cóż to takiego pozwoli nam dziś ustawić nasza metoda do obsługi tekstur? Ponieważ włączamy mieszanie kolorów, więc trzeba umożliwić te operacje dla tekstur. Stała **D3DTSS_ALPHAOP** określa operację, jaka będzie wykonana przy włączonym alpha blendingu, która jest dokładniej określona w trzecim argumencie naszej metody. Argumentem tym jest **D3DTOP_SELECTARG1**, więc wyselekcjonujemy pierwszy argument do operacji blendingu tekstury. W następnym kroku znowu mamy naszą kochaną metodę i jako drugi parametr przyjmuje ona **D3DTSS_ALPHAARG1**. W takim razie jako trzeci parametr musimy podać, co będzie pierwszym argumentem (w zasadzie kolorem) w operacji mieszania. Stała **D3DTA_TEXTURE** jasno mówi, że będzie to nasza tekstura. Następne dwa wywołania robią dokładnie to samo z tym, że będzie to wybranie drugiego argumentu dla naszej operacji. Tym razem stała **D3DTA_CURRENT** jest odpowiednikiem koloru diffuse naszego wierzchołka.

```
g_pd3dDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );
```

Jak pisałem wcześniej, jednym ze sposobów uniknięcia problemów z przezroczystymi obiektami malowanymi na scenie jest blokada zapisu do bufora *Z*, co zostanie dla nas zrobione właśnie dzięki wywołaniu metody **SetRenderState()** z powyższymi argumentami.

```
g_pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
```

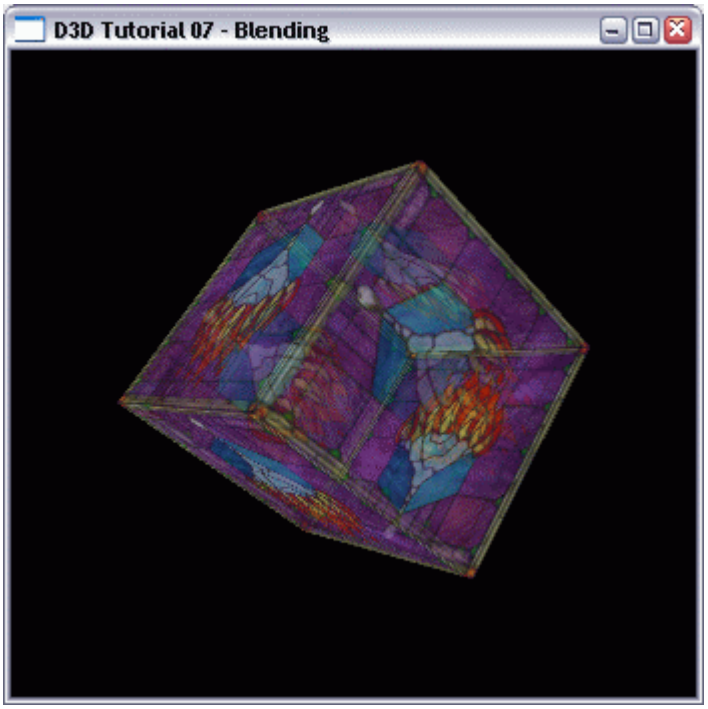
A tutaj włączamy sobie nasz blending. Ustawiając jakiś tam tajemniczy przełącznik w naszym urządzeniu, spowodujemy, że od teraz do czasu jego wyłączenia będzie przeprowadzane mieszanie kolorów na scenie wykorzystując kanał alfa naszych kolorów. I będzie się to właśnie odbywało na podstawie podanego przeze mnie powyżej wzoru.

```
g_pd3dDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
g_pd3dDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
```

Ponieważ będziemy mieli wzór, więc pora ustawić współczynniki, przez jakie będziemy skalować nasze składowe kolorów. Nasza nieoceniona metoda **SetRenderState()** pozwoli nam przypisać współczynniki i dla koloru źródłowego - **D3DRS_SRCBLEND** oraz docelowego - **D3DRS_DESTBLEND**. Tym razem jako drugi parametr podajemy wymieniane już nie raz współczynniki. Wartości, jakie przyjmują, można sobie oczywiście odczytać z tabeli, ale dla naszych skromnych potrzeb możemy sobie przyjąć, że te dwa będą najczęściej ustawiane, aby otrzymać efekt przezroczystego obiektu. Po ustawieniu wszystkiego, oczywiście namalujemy sobie nasz piękny sześcian, który powinien wyglądać mniej więcej jak ten, na obrazku poniżej. Jeśli rysujemy obiekty przezroczyste, należy zawsze pamiętać żelazną zasadę! Rysujemy najpierw te najbardziej oddalone od nas, a na końcu te, które są najbliżej. Więc z każdym narysowaniem musimy posortować ściany ze względu na odległość od obserwatora a szczególnie wtedy, kiedy obiekty lub kamera się poruszają. W naszym przykładzie oczywiście niczego takiego nie robimy, bo w zasadzie malujemy jak leci i skutek może być widoczny, jeśli ustawimy sobie na przykład przezroczystość bardzo bliską nieprzezorczystosci - weźmy dla przykładu wartość koloru FEFFFFFFh. Wyraźnie widać w jakiej kolejności są rysowane ściany, więc pamiętajcie, że zawsze trzeba je posortować przed narysowaniem! W ogóle każdą scenę, w której występuje blending, trzeba rozpatrywać niejako indywidualnie i starać się kombinować, tak aby było najlepiej - nie ma jednej uniwersalnej metody rysowania.

```
g_pd3dDevice->SetRenderState( D3DRS_ZWRITEENABLE, TRUE );
```

Na koniec oczywiście musimy włączyć to, co żeśmy wyłączyli. Gdybyśmy rysowali na scenie więcej obiektów i były one nieprzezroczyste, to mogłoby się nam to przydać. Ale dobrym zwyczajem jest to, żeby zawsze pamiętać, że jeśli coś się włącza i tworzy, to potem należy to wyłączyć i zniszczyć utworzone, zresztą jeszcze nie raz będę Wam o tym truł. Jeśli wszystko się udało skompilować bez błędów i dobrze wszystko pojęliśmy, to w końcu powinniśmy otrzymać na ekranie coś takiego:


D3D Tutorial 07 - Blending