

Kłaniam się. Cóż to mamy dzisiaj w planie... a tak, wiem. Nasze bryły 3D zapewne już się obracają i przesuwiają w przestrzeni na wszelkie możliwe sposoby :-). Nadszedł więc czas, aby poznać kolejną podstawową "cegielkę", bez której nie możemy się obejść w grafice 3D, czyli rzecz o materiałach i świetle. Umieemy już malować tysiące brył na ekranie, umiemy je bajerancko pokolorować, ale ciągle mamy wrażenie, że czegoś nam brakuje. Wszystkie bryły są jakieś takie... nijakie. Nie wyglądają w każdym razie tak, jakbyśmy sobie tego życzyli. Spróbujcie sobie dla przykładu zrobić sześcian i każdy jego wierzchołek niech będzie w takim samym kolorze. Prawda, że nie jest to nasze wyobrażenie o tym, jak powinien wyglądać np. zielony sześcian? Cóż więc zrobić, żeby wyglądało to prawie jak w rzeczywistości? Ano, w tej lekcji tego właśnie się dowiecie...

```
struct CUSTOMVERTEX
{
    D3DXVECTOR3 position;
    D3DXVECTOR3 normal;
};
```

Zaczynamy, no... jak zwykle... czyli nasze zmienne. Jak widać dzisiaj, pomimo dwóch nowych rzeczy, struktura naszego wierzchołka znacznie się uprościła. Wszystko dzięki zastosowaniu typów z naszej niezastąpionej biblioteki **D3DX**. W poprzednich przykładach każdą współrzędną, czy też inną cechę wierzchołka przedstawialiśmy za pomocą typów podstawowych, takich jak **float** czy **int**. Dzisiaj posłużymy się typami z biblioteki **D3DX**, które są niczym innym, jak strukturami zawierającymi nasze poprzednie dane, zebrane tylko w bardziej elegancki sposób. Typ **D3DXVECTOR3** zawiera, jak można się łatwo domyśleć, współrzędne wektora w przestrzeni ( $x, y, z$ ). My zawsze używaliśmy i zawsze używać będziemy co najmniej tych współrzędnych, żeby cokolwiek narysować w przestrzeni, więc użyjemy sobie w naszym przykładzie właśnie tej eleganckiej strukturki. Ponieważ tym razem będziemy mieli do czynienia z czymś takim jak materiał (o tym mowa później), więc darujemy sobie dzisiaj kolor wierzchołków. Z drugiej strony wprowadzimy do naszego świata następny element, który podwyższy jego realizm a mianowicie światło. Aby jednak móc korzystać ze światła, będziemy potrzebować czegoś takiego jak wektor normalny. W przyrodzie, światło po wyemitowaniu ze źródła zanim trafi do naszego oka, po drodze odbija się od tysięcy jeśli nie milionów obiektów. Za każdym razem, kiedy od czegoś się odbija, część promieni jest pochłaniana przez jakąś powierzchnię, część jest rozpraszana, natomiast reszta jest odbijana w kierunku następnej powierzchni lub naszego oka. I ten proces przebiega do czasu aż światło nie zostanie całkowicie pochłonięte lub nie trafi do naszego oka, abyśmy mogli cokolwiek zobaczyć. Oczywiście obliczanie takich odbić w czasie rzeczywistym za pomocą dzisiejszych komputerów jest po prostu niemożliwe. Wymagałoby to ogromnej ilości obliczeń, nieraz skomplikowanych i analizy zbyt wielkich ilości danych. Dlatego też oświetlanie "komputerowe" polega głównie na przybliżeniu sposobu działania światła w prawdziwym świecie. Przybliżanie to polega na tym, że kolor każdego wierzchołka i piksela na bryle 3D jest obliczany na podstawie bieżącego koloru materiału przypisanego naszemu obiektowi, na podstawie kolorów pikseli w obrazie reprezentującym teksturę nałożoną na nasz obiekt, intensywności światła, które umieszczamy na naszej scenie oraz światła otaczającego. Co to wszystko znaczy powiem już za chwilę, w każdym razie miałem mówić o zmiennych. Aby policzyć efekt oświetlenia, będziemy potrzebować czegoś takiego jak wektor normalny. Każda płaszczyzna (trójkąt) w naszej bryle 3D posiada prostopadły do tej płaszczyzny wektor, który nazywany jest właśnie wektorem normalnym. Kierunek tego wektora jest określony poprzez kolejność, w jakiej zorientowane są kolejne wierzchołki trójkątów (pamiętacie tutorial o niewidocznych powierzchniach?) oraz jakiego typu układu współrzędnych używamy (prawy- czy lewoskrętnego). Wektor normalny skierowany jest w tę samą stronę powierzchni, którą uznamy za widoczną. Na przykład jeśli mamy trójkąt ustawiony równolegle do ekranu i my go widzimy, to wektor normalny będzie skierowany prostopadłe do ekranu w naszą stronę - mam nadzieję, że rozumiecie :-). Aplikacje DirectX3D nie potrzebują metod do liczenia wektorów normalnych do powierzchni. Zostaną one policzone w momencie, w którym będą potrzebne. Są one obecnie używane tylko do liczenia oświetlenia w trybie cieniowania płaskiego, które jednak nie jest zbyt popularne ze względu na niezbyt rewelacyjny efekt, jaki daje w ostateczności. W trybie cieniowania Gourauda, przy liczeniu efektów oświetlenia i teksturowania, DirectX3D używa tzw. normalnych wierzchołków. Cóż to takiego? Jak sama nazwa wskazuje, normalna wierzchołka jest powiązana z konkretnym punktem na naszej bryle 3D, punktem który decyduje o jej końcowym wyglądzie. Najprostszym sposobem na policzenie normalnej wierzchołka jest policzenie normalnych do wszystkich powierzchni, które korzystają z tego wierzchołka (zawierają go jako element składowy trójkąta, z którego są zbudowane) a następnie takie policzenie tego wektora, aby tworzył on jednakowy kąt ze wszystkimi policzonymi normalnymi do powierzchni. Nie jest to jednak metoda najszybsza w większości zastosowań, więc nie raz będziemy musieli wymyśleć inną. W naszym przykładzie jednak ułatwimy sobie nieco zadanie i narysujemy taką figurę, która będzie miała już te normalne ustawione we właściwym kierunku. Przy tym jeszcze jedna ważna sprawa. Przy liczeniu normalnych nie ważne jest, gdzie są one przyłączone. Dla nas ważny będzie tylko i wyłącznie kierunek wektora, który taka normalna wyznacza, ponieważ przy obliczeniach oświetlenia DirectX3D interesuje przede wszystkim kąt, jaki normalna tworzy z promieniami światła. Wszystko na pewno jeszcze wyjdzie w praniu, więc idźmy dalej, aby nie przedłużać.

```
#define D3DFVF_CUSTOMVERTEX ( D3DFVF_XYZ | D3DFVF_NORMAL )
```

No i tutaj, w naszym makrze definiującym format wierzchołków mamy potwierdzenie tego, o czym mówiliśmy wcześniej. Uprościło nam się sporo, użyjemy tylko pozycji wierzchołka i wektora normalnego. Jeśli chodzi o kolor, to wszystko załatwi dla nas materiał i światelka.

```

HRESULT InitD3D( HWND hWnd )
{
    // Create the D3D object.
    if( NULL == ( g_pd3D = Direct3DCreate8( D3D_SDK_VERSION ) ) )
        return E_FAIL;

    // Get the current desktop display mode, so we can set up a back
    // buffer of the same format
    D3DDISPLAYMODE d3ddm;
    if( FAILED( g_pd3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT, &d3ddm ) ) )
        return E_FAIL;

    // Set up the structure used to create the D3DDevice. Since we are now
    // using more complex geometry, we will create a device with a zbuffer.
    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;

    // Create the D3DDevice
    if( FAILED( g_pd3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
        D3DCREATE_SOFTWARE_VERTEXPROCESSING, &d3dpp, &g_pd3dDevice ) ) )
    {
        return E_FAIL;
    }

    // Turn off culling
    g_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_NONE );

    // Turn on the zbuffer
    g_pd3dDevice->SetRenderState( D3DRS_ZENABLE, TRUE );

    return S_OK;
}

```

Nasza funkcja do inicjalizacji trochę się zmienia w stosunku do swoich poprzedniczek. Inicjalizację urządzenia już mamy w małym palcu, tylko dwie małe sprawy:

```

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

..
// Turn on the zbuffer
g_pd3dDevice->SetRenderState( D3DRS_ZENABLE, TRUE );

```

Te trzy linie mają jedną wspólną cechę. Mianowicie włączają nam bufor Z. Co to jest i jak to działa już w zasadzie mówiliśmy, ale tak dla porządku przypomnijmy. Bufor Z służy nam do określania, które piksele na naszej scenie przykrywają inne (znaczy się są narysowane bliżej nas w przestrzeni 3D). Wartość każdego nowo rysowanego piksela na scenie jest za każdym razem sprawdzana z wartością już istniejącego w buforze. Jeśli wartość ta jest mniejsza niż istniejąca (jest bliżej nas), wtedy kolor i współrzędna tego piksela są wpisywane do bufora koloru i bufora Z w miejsce poprzedniego, a jeśli nie, to wtedy ten stary zostaje. Po sprawdzeniu całego bufora scena może zostać wyświetlona poprawnie. Pole **EnableAutoDepthStencil** oznacza dla nas tyle, że nakazujemy Direct3D włączenie bufora Z, abyśmy to my nie musieli się paprać brudną robotą, on jest chętny w każdej chwili liczyć wszystko za nas. Pole to jest ściśle związane z następnym, **AutoDepthStencilFormat**, które będzie zawierać określenie ilu bitowy będzie nasz bufor głębokości. Co oznacza określenie x-bitowy? Ano to samo, co w całym komputerowym świecie. Liczba ta będzie nam determinować, ile możliwych głębokości można zmieścić w takim buforze. W naszym przypadku stała **D3DFMT\_D16** oznacza, że tych głębokości będzie możliwych  $2^{16}$ . Jeśli więc nakreślimy więcej niż  $2^{16}$  trójkątów, z których każdy będzie miał inną głębokość, będzie to oznaczało, że mamy kłopoty. Wtedy będziemy musieli albo pozbyć się w jakiś sposób pewnej ich liczby, albo zwiększyć tę głębokość. Ale nie bądźmy czarnej myśli. Ilość  $2^{16}$  w większości zastosowań nam wystarczy. Co oznacza wywołanie funkcji **SetRenderState()** z podanymi parametrami nie muszą chyba nikomu tłumaczyć.

```

HRESULT InitGeometry()
{

```

```

// Create the vertex buffer
if( FAILED( g_pd3dDevice->CreateVertexBuffer( 50*2*sizeof(CUSTOMVERTEX),
    0, D3DFVF_CUSTOMVERTEX, D3DPOOL_DEFAULT, &g_pVB ) ) )
{
    return E_FAIL;
}

// Fill the vertex buffer. We are algorithmically generating a cylinder
// here, including the normals, which are used for lighting.
CUSTOMVERTEX* pVertices;
if( FAILED( g_pVB->Lock( 0, 0, (BYTE**)&pVertices, 0 ) ) )
    return E_FAIL;

for( DWORD i=0; i<50; i++ )
{
    FLOAT theta = (2*D3DX_PI*i)/(50-1);
    pVertices[2*i+0].position = D3DXVECTOR3( sinf(theta),-1.0f, cosf(theta) );
    pVertices[2*i+0].normal = D3DXVECTOR3( sinf(theta), 0.0f, cosf(theta) );
    pVertices[2*i+1].position = D3DXVECTOR3( sinf(theta), 1.0f, cosf(theta) );
    pVertices[2*i+1].normal = D3DXVECTOR3( sinf(theta), 0.0f, cosf(theta) );
}
g_pVB->Unlock();

return S_OK;
}

```

Za to w metodzie do inicjalizacji (tworzenia) naszej geometrii zmieniło się sporo. Wprawdzie nasz stary znajomy - bufor wierzchołków pozostał, no bo bez niego to ani rusz, ale figura będzie dzisiaj tak dla urozmaicenia bardziej okrągła niż zawsze :-). Sądzę, że samego bufora to już nie muszę omawiać, bo utrwaliło się to Wam w pamięci już wystarczająco, natomiast nasza figura będzie nieco innym, ciekawszym tworem. Dzisiaj po sześcianie i ostrosłupie przyszedł czas na... walec a w zasadzie to rurę, bryłę na której mam nadzieję z powodzeniem uda mi się zaprezentować Wam sposób działania materiału w połączeniu ze światłem. Jak będziemy go rysować? Otóż wykorzystamy naszego dobrego pomocnika, czyli pas trójkątów. Pamiętacie jak go rysowaliśmy? Jeśli nie, zapraszam do poprzednich lekcji, założmy jednak dla uproszczenia, że co nieco jeszcze pamiętacie ;-). Ponieważ mamy mieć tę rurę, więc spróbujmy sobie wyobrazić, jak to powinno wyglądać, żeby dało się narysować... Najprostsza rura to trzy prostokąty złożone krawędziami ze sobą (chyba potraficie sobie to wyobrazić? :-), więc gdybyśmy tak połączyli dużo więcej takich prostokątów, to pewnie uzyskamy dużo lepszą "okrągłość" naszej rury niż z wykorzystaniem tylko 3 ścianek... Ściankę, żeby było najprościej, złożymy sobie z dwóch trójkątów. I w zasadzie wszystko mamy jasne. Jak łatwo będzie zauważyć, wszystko nam doskonale zadziała, jeśli każdy kolejny wierzchołek będzie leżał na przeciwko poprzedniego (chodzi o końce naszej rury) i będziemy je rysować jako kolejne na obwodzie naszej bryły. Czy w skrócie to: punkt na jednym końcu rury, odpowiadający mu punkt na drugim końcu, następnie przesuwamy się trochę dalej na obwodzie i znowu punkt na przeciwnym końcu niż ten ostatni i tak dalej... Nasza prosta pętka zrobi to dla nas w mgnieniu oka:

```

for( DWORD i=0; i<50; i++ )
{
    FLOAT theta = (2*D3DX_PI*i)/(50-1);
    pVertices[2*i+0].position = D3DXVECTOR3( sinf(theta),-1.0f, cosf(theta) );
    pVertices[2*i+0].normal = D3DXVECTOR3( sinf(theta), 0.0f, cosf(theta) );
    pVertices[2*i+1].position = D3DXVECTOR3( sinf(theta), 1.0f, cosf(theta) );
    pVertices[2*i+1].normal = D3DXVECTOR3( sinf(theta), 0.0f, cosf(theta) );
}

```

Jak widać, punkcików takich będziemy mieć pięćdziesiąt, a więc dosyć sporo, jak na taki prosty model. W pętli liczymy najpierw o jaki kąt przesuniemy się w naszej wędrówce po obwodzie rury, następnie wstawiamy do naszego bufora współrzędne nowo powstałego z obliczeń punktu oraz jego normalną. Normalna ma takie same współrzędne jak punkt, co jest dla nas całkowicie zrozumiałe. Ponieważ punkty leżą na obwodzie koła, więc kąt, jaki normalna wierzchołka będzie tworzyć z normalnymi płaszczyzn korzystających z tego wierzchołka, będzie zawsze taki sam - trzeba przyznać, że ułatwiliśmy sobie tutaj bardzo zadanie. I jak powiedziałem wcześniej, nie ważne gdzie leży ta normalna, dla światła ważne jest jaki tworzy ona kąt z jego promieniami. Po zakończeniu wypełniania naszego bufora odblokowujemy go i przystępujemy do ustawienia naszych macierzy świata, widoku i rzutowania. Tutaj też nic dokładnie nie zmienimy, więc może przystąpimy teraz do omówienia kolejnej nowej i ważnej rzeczy, jaką będzie ustawienie obiektowi materiału (ang. *material*) i światła na scenie (ang. *lights*).

```

D3DMATERIAL8 mtrl;
ZeroMemory( &mtrl, sizeof(D3DMATERIAL8) );
mtrl.Diffuse.r = mtrl.Ambient.r = 1.0f;

```

```
mtrl.Diffuse.g = mtrl.Ambient.g = 1.0f;
mtrl.Diffuse.b = mtrl.Ambient.b = 1.0f;
mtrl.Diffuse.a = mtrl.Ambient.a = 1.0f;
g_pd3dDevice->SetMaterial( &mtrl );
```

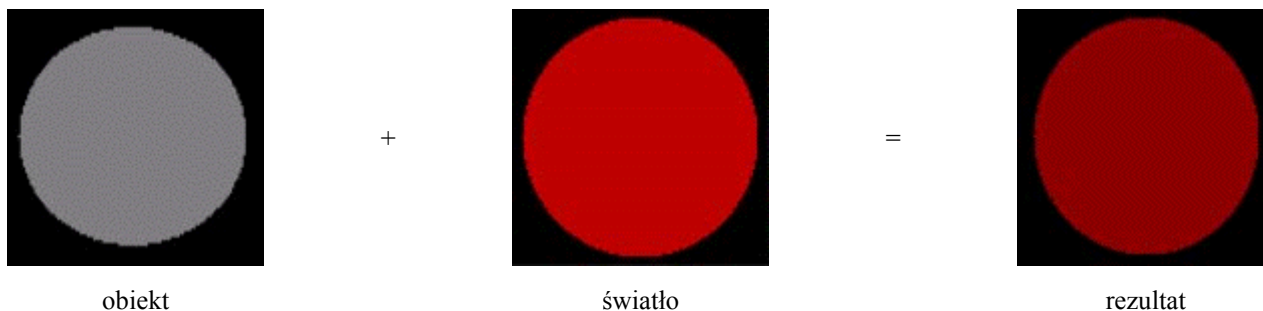
Po pierwsze - materiał. Czym jest materiał? Materiał mówi nam o tym, jak dany obiekt, z którym jest on związany, odbija światło na niego padające. To tak ogólnie. Dokładniej mówiąc i patrząc na to od strony Direct3D, materiał opisuje nam pewne cechy obiektów, które my będziemy renderować, a które będą posiadać dany materiał niejako "na sobie". Direct3D interesuje przede wszystkim:

- jak obiekt odbija światło rozproszone i otaczające (do czego jedno i drugie już za moment),
- jak wyglądają odbłaski światła na obiekcie (czy obiekt lśni, czy jest matowy),
- czy obiekt emituje jakieś własne światło.

Takie cechy materiału opisuje w Direct3D struktura **D3DMATERIAL8**. Zawiera więc ona informacje o tym, ile materiał odbija światła rozproszonego i otaczającego, określa stopień odbłasku materiału oraz ilość emitowanego światła przez sam materiał. Nie muszą chyba dodawać, że właściwości materiału wpływają bezpośrednio na kolor pikseli, które są obliczane i wyświetlane na ekranie monitora w końcowym renderingu. Oprócz właściwości refleksu światła pozostałe 3 są określone przez cztery wartości: składniki koloru (RGB - *Red*, *Green*, *Blue*) oraz stopień przezroczystości, czyli tzw. kanał alfa (poświęćmy temu zagadnieniu oddzielny tutorial). Refleks światła jest określony poprzez inne wartości - swoją siłę oraz kolor tegoż refleksu.

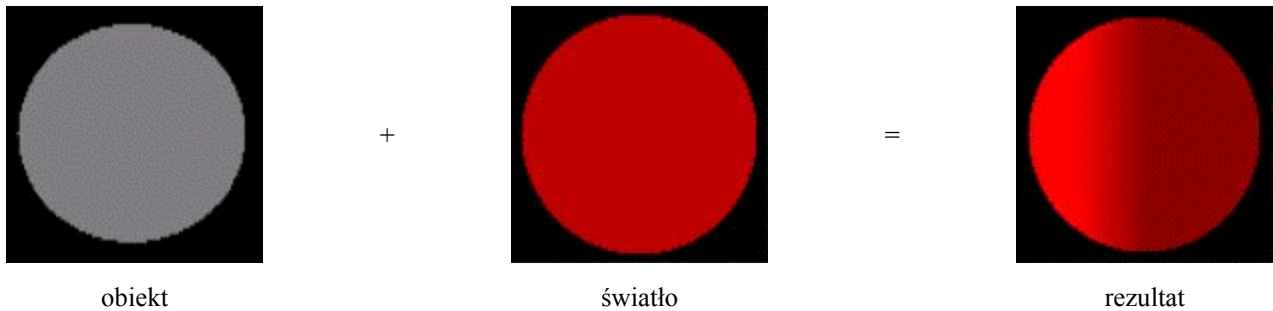
Teraz nadszedł czas, aby sobie powiedzieć trochę o składowych światła. Pierwszym z nich jest tzw. składowa otaczająca (ang. *ambient*). Co ona oznacza? Światło z ustawioną tą właściwością oświetla wszystkie wierzchołki w ten sam sposób. Nie sugeruje się żadnymi normalnymi, odległością od światła, jego kierunkiem, zasięgiem itp. Służy ono po prostu do rozjaśniania (lub ściemniania) oraz ogólnego pokolorowania sceny. Jest to najszybszy sposób oświetlania, co jednak jest okupione najmniejszym stopniem realizmu na naszej oświetlonej w ten sposób scenie. Direct3D zawiera pewną pojedynczą wartość, którą możemy ustawić, bez włączania nowego światła, a która to wartość spowoduje, że scena będzie się zachowywać tak, jakbyśmy włączyli tam światło z ustawioną właściwością ambient. Dzieje się tak dlatego, ponieważ, tak jak napisałem, światło to ma służyć przede wszystkim do korekcji jasności i koloru naszej sceny. Jednak samo zastosowanie światła ze składową ambient naszej scenie realizmu nie doda. Konieczne jest więc zastosowanie tzw. składowej rozproszonej (ang. *diffuse*). Jeśli dla przykładu na naszej scenie narysujemy sobie kulę i oświetlimy ją tylko światłem otaczającym, to kiedy ktoś pierwszy raz popatrzy na naszą scenę, będzie mógł stwierdzić, że ona wcale nie jest 3D. Kula będzie wyglądała po prostu jak zwykłe koło.

Możemy tę sytuację prześledzić na przedstawionych rysunkach:



Zastanówmy się co powoduje, że kula będzie wyglądać jak kula. Najlepiej weźmy do ręki jakąś kuleczkę w jednym kolorze (choćby do zwykłego ping-ponga). I cóż widzimy? Pierwsze, co może rzucić się w oczy, to to, że w różnych miejscach ma ona inne natężenie swojego koloru! Pomimo że cała jest, powiedzmy hmm..., że biała to jednak w jednym miejscu jest nieco ciemniejsza, w innym jaśniejsza. I to jest to! Jeśli spróbowalibyśmy teraz narysować na komputerze zupełnie płaski rysunek kuli, ale zastosujemy tę sztuczkę i znowu pokazemy to tej samej osobie, to ona z całą pewnością stwierdzi, że ta kulka jest już bardziej "3D" niż ta poprzednia, choć tamta jest renderowana przez najnowszy akcelerator za 20 baniek a tą narysowaliśmy w zwykłym Paint-cie. Cóż więc na to poradzić? Otóż tu z ratunkiem pospieszy nam właśnie składowa rozproszona. To właśnie dzięki niej, wykorzystując normalne, odległości, zasięg i inne parametry światła uzyskamy efekt pocienienia naszej kuli, tak aby już po tym fakcie przedstawić naszemu biednemu obiektowi eksperymentów zrenderowaną scenę z pytaniem czy jest już zadowolony. I z całą pewnością będzie, ponieważ nasza kula będzie już prawie jak prawdziwa!

I znów prześledźmy, jak to wygląda na przykładzie:



Dzięki składowej rozproszonej Direct3D policzy najpierw kąt, jaki tworzy normalna z promieniami światła a następnie na jego podstawie powie ile światła jest odbijane do obserwatora i jakim odcieniem koloru ma zostać umieszczony na bryle piksel w danym miejscu. W zależności od rodzaju światła (punktowe, kierunkowe itd.) Direct3D będzie brał różne parametry światła, ale o tym też za chwilę powiemy. Jak łatwo się domyśleć, ten sposób jest o wiele bardziej obliczeniowy, ponieważ trzeba policzyć o wiele, wiele więcej rzeczy niż wykorzystując tylko światło otaczające, ale jeśli nie umieścimy za wiele światła na naszej scenie, to wszystko powinno być w jak najlepszym porządku. Te dwa parametry będą miały dla nas, początkujących decydujące znaczenie. Pozostałych nie będziemy dzisiaj mieszać w naszą scenę. Jeśli nadarzy się okazja, to na pewno jeszcze sobie o nich powiemy.

Ale miałem mówić o materiale, a zapędziłem się i już wiecie prawie wszystko o światłach :-), ale nic to, teraz będzie to jak znalazł. Ale wracając jeszcze do tych materiałów. Co oznaczają te wartości, które podajemy jako składowe struktury **D3DMATERIAL8**? Otóż jest to ilość światła, jaka zostanie odbita przez obiekt wyposażony w nasz materiał, oczywiście robimy to dla każdej składowej oddzielnie. Jeśli więc widzimy np.:

```
mtrl.Diffuse.r = 1.0f;
mtrl.Diffuse.g = 1.0f;
mtrl.Diffuse.b = 1.0f;
mtrl.Diffuse.a = 1.0f;
```

Oznacza to tyle, że nasz materiał odbije wszystko, co przyjdzie do niego ze źródła światła, oczywiście będzie to przeliczone przez inne parametry (kąty, odległość, itp.). Ktoś zada ciekawe pytanie - jaki więc kolor masz nasz obiekt? Ponieważ nie przyczepiliśmy do niego żadnej tekstury, która by to określiła, wycieliśmy kolor wierzchołków więc? Otóż kolor naszego obiektu zależał będzie od dwóch rzeczy: po pierwsze od definicji naszego materiału, ale z drugiej strony od koloru światła jakie będzie na niego padało. Tak samo działa to w rzeczywistym świecie. Ponieważ słońce oświetla wszystko wokół nas światłem białym, które zawiera w swoim widmie wszystkie barwy, więc to materiał decyduje o tym, jaki my widzimy kolor danego obiektu. Ale założmy teraz, że nagle słońce zaczyna świecić światłem, które na przykład zawiera tylko składową niebieską. I co się dzieje? Jeśli obiekty będą posiadać materiał, który nie będzie odbijał światła niebieskiego (składowa **R** będzie miała wartość 0), nic nie będziemy widzieć!!! A teraz w drugą stronę: jeśli nasz obiekt będzie oświetlony światłem białym, a materiał będzie odbijał tylko składową niebieską, to jaki będzie kolor naszego obiektu? Czy już wiecie? :-). Sytuacja ta będzie miała miejsce oczywiście zarówno w przypadku składowej otaczającej (ambient) jak i rozproszonej (diffuse).

Dobrze. Przebrnęliśmy chyba przez najtrudniejsze, więc czas teraz na nieco przyjemniejsze rzeczy, czyli ustawimy sobie na naszej scenie światełko, a może nawet dwa? Światełko w Direct3D jest określane, jak wiele innych obiektów, poprzez strukturę. W tym przypadku będzie to **D3DLIGHT8**. Poszczególne pola będą mówiły o wielu różnych parametrach, takich jak te, o których już mówiliśmy (składowe koloru), a także o zasięgu światła, jego pozycji, kierunku i kilku innych parametrach zależnych od jego typu. Teraz właśnie przyszedł pewnie czas, aby dowiedzieć się czegoś o rodzajach światła. I znowu mamy analogię do świata rzeczywistego. Oczywiście nie da się zamodelować wszystkich rodzajów światła, bo to nie jest możliwe, ale kilka podstawowych zaimplementowano w DirectX z mniejszym lub większym powodzeniem. I tak mamy:

- **Światła punktowe** (ang. *point light*) - charakteryzują się one oczywiście kolorem, jaki posiadają, pozycją w jakiej są umieszczone, ale nie posiadają jednego ściśle określonego kierunku. Światło to świeci we wszystkich kierunkach a analogia jaka mi się tutaj nasuwa to np. żarówka. Ważny będzie też w przypadku takiego światła jego zasięg, który może zostać ograniczony.
- **Światła kierunkowe** (ang. *directional light*) - takie światła posiadają tylko kolor i kierunek w jakim świecą (kierunek definiujemy poprzez wektor). Światła takie emitują promienie równoległe, tzn. że promienie przechodzące przez scenę, będą na niej cały czas w jedną stronę. Światła te nie posiadają zasięgu. Dostateczną analogią z przyrody jest np. światło słoneczne.
- **Reflektor** (ang. *spot light*) - takie światełko z kolei posiada kolor, pozycję oraz kierunek, w którym emituje promienie. Światło padając na powierzchnię tworzy na niej "plamę" (kto wie jak działa latarka, ten wie o czym mówię ;-). Ma ono określony zasięg, można też regulować wygląd "plamy" o której mówiliśmy (tzn. wewnętrzna - jaśniejsza, zewnętrzna ciemniejsza otoczka i ich wzajemne położenie).

Nie będę się wgłębiał w matematykę, bo tą możecie znaleźć w dokumentacji SDK. Nas bardziej będzie interesowało praktyczne wykorzystanie światła i ogólne pojęcie po co nam to wszystko i jak to działa. Jak powiedziałem wcześniej, na naszej scenie będziemy mieli dwa światelka. Dla naszego przykładu użyjemy światła kierunkowych, czyli posiadających kolor i kierunek świecenia. Dla potrzeb naszej sceny zrobimy tak: żeby nie było nudno w tym odcinku, zamiast animować obiekt, będziemy poruszać światelkami. Będą one krążyć wokół naszej rury i będą świecić niejako "przez nią", tzn. będą skierowane przez środek układu współrzędnych. Nasza rura zostanie ustawiona pionowo abyśmy lepiej mogli obserwować wpływ światła na nasz obiekt. Zajmijmy się więc naszym światelkiem:

```
D3DXVECTOR3 vecDir;
D3DLIGHT8 light;
ZeroMemory( &light, sizeof(D3DLIGHT8) );
light.Type = D3DLIGHT_DIRECTIONAL;
light.Diffuse.r = 1.0f;
light.Diffuse.g = 0.0f;
light.Diffuse.b = 0.0f;
vecDir = D3DXVECTOR3(cosf(timeGetTime()/350.0f), 1.0f, sinf(timeGetTime()/350.0f)
);
D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDir );
light.Range = 1000.0f;
g_pd3dDevice->SetLight( 0, &light );
g_pd3dDevice->LightEnable( 0, TRUE );
```

Pierwsze co robimy, to musimy przygotować dla naszej struktury pamięć. Tę metodę znamy już też dobrze, bo stosowaliśmy przecież nie raz. Następnie ustawiamy rodzaj naszego światła. Tak jak powiedziałem, my będziemy mieli na naszej scenie światelko kierunkowe. Określa nam to stała **D3DLIGHT\_DIRECTIONAL**, którą należy przypisać polu **Type** struktury **D3DLIGHT8**. Następna rzecz, którą robimy to ustawienie koloru światła jakim będzie ono świecić. Użyjemy tylko składowej rozproszonej, ponieważ samą scenę rozjaśnimy sobie za pomocą odpowiedniej funkcji (o której pisałem wcześniej), bez używania światła (im mniej światłek na scenie tym lepiej, możecie mi wierzyć). Jak widać, pierwsze z naszych światła będzie miało kolor czerwony, co w połączeniu z białą składową materiału da nam co? Obiekt będzie odbijał wszystko co do niego przyleci, a że od tego światła przybędzie tylko czerwień, więc wróci... no chyba wiadomo co. Następna rzecz to zdefiniowanie w jakim kierunku będzie świecić nasze światelko. Ponieważ ustawienie pozycji światelka w naszym przypadku nic nam nie da, więc musimy posłużyć się polem **Direction**. Aby zaangażować obrót naszego światelka wokół obiektu posłużymy się znowu tą samą metodą obliczenia pozycji co w przypadku tworzenia naszej rury. Zauważmy, że tworzymy wektor, który posiada swój początek w środku układu współrzędnych. Nasze światelko będzie więc świecić od środka układu na zewnątrz rury, poprzez jej ścianę. Aby wszystko było w porządku, powinniśmy jeszcze znormalizować nasz wektor kierunkowy światła. Na czym polega normalizacja? Otóż jest to nadanie jakiemuś obliczonemu wcześniej wektorowi długości równej jeden bez zmiany kierunku. Robi się tak szczególnie w przypadku wektorów używanych potem wewnątrz przez Direct3D do obliczeń, czyli wszelkich normalnych czy właśnie wektorów kierunkowych dla światła. Do tego celu posłuży nam funkcja **D3DXVec3Normalize()**, która jako pierwszy swój parametr pobiera wektor, który będzie zawierał dane znormalizowane z wektora, który podamy jako parametr drugi. Tak więc w tym przypadku upieczemy od razu dwie pieczenie przy jednym ogniu. Znormalizujemy wektor i przypiszemy jego wartości właściwemu polu struktury **D3DLIGHT8**.

```
g_pd3dDevice->SetLight( 0, &light );
g_pd3dDevice->LightEnable( 0, TRUE );
```

No i po wypełnieniu naszej struktury odpowiednimi wartościami pozostaje nam tylko jedna, a właściwie to dwie rzeczy. Pierwsza z nich to powiedzenie naszemu urządzeniu, że te parametry, które ustawiliśmy w naszej strukturze, chcemy przypisać światłu nr 0. Liczba światła, jaką możemy włączyć, jest w zasadzie nieograniczona. Tylko jedna bardzo ważna rzecz. Liczenie oświetlenia przez nasz program ręcznie (software'owo) nie będzie miało na niego pozytywnych wyników. Ilość klatek na sekundę na pewno spadnie drastycznie, żebyśmy posiadali nie wiem jak silny procesor główny. Dlatego też zawsze warto ograniczyć się do liczby światła, jaką może policzyć nam nasza karta. O tej zaś liczbie możemy się dowiedzieć poprzez odczyt pola **MaxActiveLights** struktury **D3DCAPS8**, która będzie zawierała właściwości naszego urządzenia po wywołaniu metody odczytującej te właśnie dane. Ale pamiętajmy od teraz - im mniej światła na scenie - tym lepiej. Ktoś może powiedzieć - zaraz, zaraz, przecież światła nadają naszej scenie realizmu, więc jak to? Owszem to prawda, ale musimy również pamiętać, że światła pożerają ogromne ilości obliczeń, więc ich stosowanie należy ograniczyć do minimum. Resztę efektów trzeba będzie wymyślić inaczej, poprzez zastosowanie map świetlnych i innych podobnych tricków. Drugą rzeczą po powiadomieniu naszego urządzenia, które to ma być światło, jest powiedzenie mu, aby uaktywniło nam to światło. Uaktywnienie go spowoduje, że jego parametry będą uwzględniane przy obliczaniu kolorów pikseli na naszej scenie. Jako parametry tej metody podajemy numer światła, które chcemy uaktywnić (możemy je również deaktywować tą samą metodą, podając jako drugi parametr FALSE). Tak samo postępujemy z naszym drugim światłem. Ono też będzie kierunkowe, jedyną zmianą, jaką poczynimy, będzie kolor, w jakim będzie świecić i jego kierunek. Tym razem będzie ono, powiedzmy, białe i jak będziemy mieli okazję się przekonać, czy faktycznie nasz materiał odbija wszystko, co się do niego dostanie. Nasze drugie światło będzie też świecić w przeciwną stronę niż poprzednie, a jaki to dam efekt końcowy? Zaraz się przekonamy.

```
g_pd3dDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
g_pd3dDevice->SetRenderState( D3DRS_AMBIENT, 0x00202020 );
```

Na koniec jeszcze dwie rzeczy. Pierwsze to uaktywnienie liczenia oświetlenia na naszej scenie. O tej metodzie już wspominałem przy pierwszych lekcjach o wierzchołkach. Właściwie to mogliśmy sobie darować wywołanie tej metody, ponieważ oświetlenie jest domyślnie włączone Direct3D, ale ponieważ jesteśmy porządni, więc sobie tutaj jasno zaznaczymy, że mamy na scenie światelka i ich używamy. Druga metoda, to wyżej wspomniane ustawienie składowej otaczającej (ambient) na scenie. Ponieważ nie używamy oddzielnego światła, to dla naszych potrzeb posłużymy się tą metodą, która ustawi ogólny poziom jasności i koloru na naszej scenie. Wszelkich obliczeń Direct3D dokona sam bez obciążania dodatkowo procesorów jeszcze jednym światłem.

```
VOID Render()
{
    // Clear the backbuffer and the zbuffer
    g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
                      D3DCOLOR_XRGB(0,0,255), 1.0f, 0 );

    // Begin the scene
    g_pd3dDevice->BeginScene();
    // Setup the lights and materials
    SetupLights();

    // Render the vertex buffer contents
    g_pd3dDevice->SetStreamSource( 0, g_pVB, sizeof(CUSTOMVERTEX) );
    g_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
    g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 2*50-2 );

    // End the scene
    g_pd3dDevice->EndScene();

    // Present the backbuffer contents to the display
    g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

No i powoli zbliżamy się do końca naszej wędrówki po już oświetlonym świecie. Nasza funkcja do renderingu. W zasadzie nie mamy tutaj także za wiele nowego. Nawet nam się funkcja uprościła można powiedzieć, bo pomiędzy metodami **BeginScene()** i **EndScene()** mamy wywołanie ledwie 4 funkcji. Ustawiania strumienia danych i rysowania prymitywu nie trzeba chyba wyjaśniać? Wywołanie funkcji SetupLights() jest tutaj trochę mało eleganckie ale cóż, niech już będzie. Za każdym wywołaniem funkcji Render() zostaną obliczone nowe pozycje światel i po policzeniu przez Direct3D oświetlenia i kolorów, otrzymamy na ekranie poniżej załączony obrazek. Po uruchomieniu samego programu efekt będzie na pewno jeszcze ciekawszy :-).

