

Witam. Dziś wreszcie się zacznie. Wiemy już coś niecoś o pakiecie DirectX, wiemy trochę o DirectX Graphics i znamy podstawowe pojęcia w grafice 3D. Tak więc nadszedł czas, aby wreszcie zabrać się za właściwą rzecz, czyli za analizę przykładowych kodów. Na pierwszy ogień idzie pierwszy przykład z DirectX SDK, dotyczący tworzenia podstawowych urządzeń oraz przygotowania okna do renderingu. Od tej pory zmienimy nieco sposób prowadzenia lekcji. Będziemy omawiać poszczególne fragmenty kodu, co pomoże nam lepiej zrozumieć po co to wszystko. Więc zaczynamy!

Jak napisałem we wcześniejszych lekcjach, cały pakiet DirectX jest oparty na technologii COM. My, programiści dostajemy do ręki zestaw obiektów i udostępnianych przez nie interfejsów (zbiorów metod), dzięki którym będziemy mogli korzystać z dobrodziejstw współczesnej elektroniki, jakimi są niewątpliwie nasze najnowsze akceleratory grafiki. Nas będą interesować oczywiście obiekty z modułu DirectX Graphics. Na początek będziemy potrzebować dwóch podstawowych, bez których na pewno nie da się pracować. W tym miejscu od razu na początku zaznaczmy sobie, że typy zmiennych, jakie będziemy spotykać w naszych programach, będą czasem znacznie odbiegać od tych, dotychczas dobrze znanych.

```
#include <d3d8.h>
```

Na samym początku dołączamy oczywiście wszelkie niezbędne pliki nagłówkowe. Do utworzenia pierwszej poprawnie pracującej aplikacji korzystającej z DirectX Graphics będzie nam potrzebny tylko ten, wyżej wymieniony. Ktoś zapyta: dobrze, ale przecież mamy aplikację Windows, więc na pewno trzeba jeszcze dołączyć **windows.h**. I oczywiście będzie miał rację, ale... jeśli komuś się chciało przeglądać plik **d3d8.h**, to napewno zauważył, że został tam już dołączony ten jakże pożyteczny plik. Mamy to więc niejako z głowy. W sumie jest to oczywiste, ponieważ sam DirectX korzysta ze zmiennych i funkcji Windows, to musi o nich wiedzieć. Mamy więc potrzebne nagłówki - co dalej?

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

Aby móc korzystać z dobrodziejstw naszego akceleratora grafiki, musimy na samym początku stworzyć dwa obiekty. Pierwszy z nich jest to obiekt Direct3D. O COM-ie wiemy już prawie wszystko ;-), więc domyślamy się, że obiekt ten posłuży nam między innymi do uzyskania dostępu do innych składników pakietu. Jest to pierwszy obiekt, jaki musimy stworzyć, aby dobrać się do pozostałych i ostatni, który zwalniamy. Jeśli czytaliście uważnie lekcję o COM-ie, to wiecie, że obiekty te charakteryzują się czasem życia, na który możemy bezpośrednio wpływać. Ale jeśli chcemy mieć dostęp do różnych obiektów z modułu DirectX Graphics, to obiekt Direct3D powinien zawsze być obecny. Jeśli kończymy nasz program, "życie" może też zakończyć nasz główny obiekt Direct3D. Co daje nam ten obiekt poza tym, że będziemy za jego pomocą mogli się dobrać do innych? Posiada kilka pożytecznych metod, które pozwolą nam na pobieranie ilości urządzeń 3D w naszym systemie, określenie ich możliwości i ogólnych właściwości. Umożliwia to aplikacjom wybranie urządzenia, z którego będą korzystać, bez bezpośredniego tworzenia obiektu takiego urządzenia (co to jest obiekt urządzenia już tłumaczę). Drugi obiekt, obiekt urządzenia reprezentuje moduł, który będzie wykonywał wszystkie polecenia, dające w ostatecznym rozrachunku obraz na monitorze. Tak więc za jego pomocą będziemy dokonywać transformacji obiektów, ich oświetlenia i rzutowania na płaszczyzny, które zostaną potem wyświetlone na monitorze, oczywiście wykorzystując odpowiednie jego metody. Jak zobaczymy na przykładzie, umiejętne ich wykorzystanie pozwoli na osiągnięcie oszałamiających rezultatów :-), no ale po kolei. Jak widać w przykładowym kodzie, tworzymy dwie zmienne, które w zasadzie są wskaźnikami (zaznajomieni z notacją węgierską chyba się domyślają) na obiekty typu **Direct3D** i **Direct3DDevice**. Jak przekonamy się już niedługo, w większości przypadków będziemy się posługiwać właśnie wskaźnikami.

```
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;

    return S_OK;
}
```

Co oznaczają poszczególne linie? Wytłumaczmy sobie może po kolei:

```
if( NULL == ( g_pD3D = Direct3DCreate8( D3D_SDK_VERSION ) ) )
    return E_FAIL;
```

Pamiętacie z lekcji o COM-ie jak zastanawialiśmy się skąd wziąć wskaźnik do pierwszego obiektu? Wtedy napisałem, że my użyjemy funkcji, która da nam do ręki taki wskaźnik. No i jak widać na przykładzie, nie jest to takie straszne, jak mogłoby się wydawać, prawda? Funkcja (tu mała uwaga, w tym momencie jest to właśnie funkcja, ponieważ nie jest ona składnikiem żadnego obiektu!) zwraca nam wskaźnik do obiektu Direct3D. Jako parametr funkcji należy podać stałą

**D3D\_SDK\_VERSION**. Zapewnia ona zgodność naszej aplikacji z dołączonymi nagłówkami. Będzie ona inkrementowana (zwiększana) za każdym razem, gdy nastąpią jakieś zmiany (np. w nagłówkach), które będą wymagały przebudowania aplikacji. My jednak przyjmijmy do wiadomości po prostu, że jako parametr tej funkcji podajemy taką, a nie inną stałą. W przeciwnym przypadku wywołanie funkcji nie powiedzie się i wskaźnik na obiekt Direct3D przyjmie wartość NULL, która spowoduje, że nasza aplikacja nie uruchomi się.

```
D3DDISPLAYMODE d3ddm;
if( FAILED( g_pD3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT, &d3ddm ) ) )
    return E_FAIL;
```

Mając już dostęp do pierwszego obiektu (mamy do niego wskaźnik), możemy się pokusić o wykorzystanie którejs z jego metod. Ponieważ nasza pierwsza aplikacja będzie pracowała w trybie okienkowym, konieczne będzie pobranie właściwości aktualnie ustawionego trybu graficznego, ponieważ będzie to nam potem potrzebne przy tworzeniu obiektu urządzenia. Jak widać na przykładzie, wywołanie metod obiektów COM nie różni się niczym szczególnym od wywołania metod klas w języku C++, co powinno nas tylko cieszyć. Dokładny opis metod, struktur i funkcji znajdziecie oczywiście w dokumentacji SDK, no ale jeśli sobie zażyczyacie, to będziemy je opisywać na stronie, będzie może jaśniej i zrozumialej. Metoda **GetAdapterDisplayMode()** pobiera, jak widać, dwa argumenty. Pierwszy z nich to urządzenie, dla którego chcemy uzyskać ustawiony na nim tryb graficzny. Dzisiaj trudno już spotkać w komputerze dwa urządzenia wyświetlające, ponieważ współczesne karty są zintegrowane z akceleratorami w jedno, ale w zamierzonych czasach VooDoo 1 taka sytuacja była normą. Oczywiście obiekt Direct3D posiada, jak powiedziałem, metody do wyszukiwania wszystkich urządzeń w systemie, no ale przeważnie będzie znajdował tylko jedno. Nawet jeśli mamy jeszcze kilka urządzeń, zawsze któreś z nich wyświetla obraz, prawda? Jest ono wtedy nazywane urządzeniem domyślnym i dla niego Direct3D ma specjalny numer, który określa stałą **D3DADAPTER\_DEFAULT**. Oczywiście przy zmianie urządzenia wyświetlającego ten numer automatycznie zostaje mu przypisany z racji pełnionego urzędu, nie muszą chyba dodawać, że taka sama sytuacja ma miejsce, gdy mamy jedno urządzenie. Założmy jednak dla uproszczenia, że mamy jedną kartę, która aktualnie pracuje jako ta, która "daje" nam obraz na monitor. Pragniemy dla niej uzyskać ustawiony aktualnie tryb graficzny. Jako drugi parametr podajemy adres struktury, która będzie zawierała dane o naszym trybie graficznym. Struktura ta (**D3DDISPLAYMODE**) zawiera informacje o aktualnej rozdzielczości, ilości bajtów na piksel (formacie) i częstotliwości odświeżania. Po wywołaniu tej metody, w zmiennej "d3ddm" będziemy mieli wszystkie interesujące nas informacje.

```
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferFormat = d3ddm.Format;
```

Następnie przygotowujemy się do uzyskania dostępu do naszego urządzenia 3D. Zanim to jednak zrobimy, musimy ustalić kilka rzeczy. Przede wszystkim w jakim trybie będziemy rysować, pełnoekranowym czy w okienku. Jest to bardzo ważne dla sterownika, ponieważ przy aplikacji pełnoekranowej wykorzystywane są pełne możliwości akceleratora grafiki, natomiast w trybie okienkowym niektóre funkcje działają wolniej. Dokonujemy tego wypełniając strukturę

**D3DPRESENT\_PARAMETERS**, która zawiera wszystkie potrzebne dane przy wywołaniu funkcji tworzącej obiekt urządzenia. Najpierw zamyślimy wszystkie pola tej struktury (funkcja **ZeroMemory()**). Jak wspomnieliśmy wcześniej, będziemy uczyć się pisać w DirectX w okienku, więc pole o nazwie **Windowed** ustawiamy na **TRUE**. Drugie pole, **SwapEffect** ma związek z łańcuchami przełączania i ustawienie go na taką, a nie inną wartość, pozwala sterownikowi wybrać najbardziej efektywną metodę przełączania buforów pamięci (jeśli chodzi o szybkość). Jeśli czytaliście uważnie lekcje poświęcone animacji, wiecie, że będziemy korzystać z buforów. Będziemy rysować w tzw. buforze tylnym (ang. *back buffer*), niewidocznym dla nas, a następnie kopiować jego zawartość na powierzchnię przednią (ang. *front buffer*). Aby wszystko przebiegało bez niespodzianek, musimy jednak pamiętać o bardzo ważnej rzeczy. Aby nic się nam nie pokręciło, należy się upewnić, że tylny bufor będzie posiadał taki sam format (liczba bajtów na piksel) jak bufor przedni, zwłaszcza, że aplikacja pracuje w trybie okienkowym. Nie zachowanie tej zasady spowoduje, że próba utworzenia obiektu urządzenia nie powiedzie się i zakończymy szybko naszą karierę jako twórcy nowego Quake'a :-). Po to właśnie pobieraliśmy właściwości aktualnie ustawionego trybu graficznego, aby teraz z tego skorzystać. Polu **BackBufferFormat** przypisujemy zawartość pola **Format** struktury **D3DDISPLAYMODE**, więc wymuszamy to, że tylny bufor będzie miał taki sam format pikseli jak aktualny bufor przedni, co pozwoli na bezproblemowe przetrzymywanie danych z bufora tylnego do przedniego. Jest to oczywiście format aktualnie ustawionego trybu graficznego.

```
if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
D3DCREATE_SOFTWARE_VERTEXPROCESSING,
```

```
&d3dpp, &g_pd3dDevice ) ) )
```

```
return E_FAIL;
```

Mając potrzebne dane w strukturze **D3DPRESENT\_PARAMETERS**, wywołujemy następną metodę obiektu Direct3D - **CreateDevice()**. Metoda ta tworzy nam, po podaniu kilku parametrów, obiekt naszego urządzenia, którego wykorzystanie pozwoli z kolei na wyświetlanie sceny i manipulowanie nią w sposób, w jaki sobie tego zażyczymy. Pierwszy parametr **D3DADAPTER\_DEFAULT** mówi nam, podobnie jak w przypadku pobierania aktualnego trybu graficznego, z którego sterownika będziemy korzystać przy tworzeniu i wyświetlaniu naszej sceny. Drugi parametr **D3DDEVTYPE\_HAL** określa, jakiego typu urządzenia renderującego będziemy używać. Większość posiadaczy słabszego sprzętu wie o tym, że można uruchomić większość gier na tzw. emulacji software'owej, jeśli nie posiada się odpowiedniego akceleratora. Jednak szybkość działania takiej emulacji pozostawia bardzo wiele do życzenia, a czasem w ogóle nie ma sensu. W określonych przypadkach możliwe jest wykorzystanie specjalnych rozkazów procesorów głównych (MMX, 3D-Now!), które w jakiś tam niewielki sposób przyspieszają wykonywanie poszczególnych instrukcji, ale przy wymaganiach dzisiejszych aplikacji jest to zdecydowanie i tak za mało. Ponieważ dzisiaj większość komputerów wyposażona jest w akceleratory grafiki i każdy graficzny procesor coś tam potrafi, więc korzystanie z emulacji programowej odłożmy na razie na półkę, choć niewykluczone, że kiedyś do tego powrócimy. Trzeci parametr to po prostu uchwyt okna, w którym będzie wyświetlany nasz obraz (główne okno naszej aplikacji). Jasne więc staje się tutaj to, że obiekt urządzenia tworzymy po zarejestrowaniu i utworzeniu naszego okna! Jeśli jeszcze ktoś nie wie jak tworzyć okna i co to znaczy rejestrować klasę okna w systemie, zapraszam na strony z kursem API :-). Czwarty parametr definiuje ogólne zachowanie się urządzenia renderującego, tzn. mniej więcej sposób w jaki będzie przetwarzać dane, które my będziemy mu przekazywać. Ustawienie tego parametru na wartość **D3DCREATE\_SOFTWARE\_VERTEXPROCESSING**, daje możliwość pracy naszej aplikacji na kartach bez sprzętowego przetwarzania wierzchołków. Tą właściwość posiada, jak na razie, niewiele akceleratorów, ale sytuacja już zmienia się na lepsze ;-). Piąty parametr to struktura, którą wypełnialiśmy powyżej, mówiąca, między innymi, w jakim trybie uruchamiamy naszą aplikację, jak będą przełączane bufor obrazu, oraz jaki format posiada tylny bufor. Ostatni, szósty parametr to adres wskaźnika do poszukiwanego przez nas od dawna obiektu urządzenia. Jeśli wszystko odbyło się prawidłowo i otrzymaliśmy nasz wskaźnik, to ta metoda zwróci nam wartość **D3D\_OK**. Warto zaznajomić się trochę z zestawem błędów DirectX, ponieważ pomoże to w trudnej sztuce, jaką jest wyszukiwanie błędów w aplikacjach wykorzystujących DirectX. Dobrze jest też badać jakie błędy zwracają wszelkie metody obiektów, co znakomicie ułatwia zlokalizowanie potencjalnych niezgodności.

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

Następna funkcja, jaką omówimy, będzie to funkcja zwalniana obiekty przy zakończeniu programu. Ponieważ korzystamy z COM-ów więc, jak wspominałem w lekcji ich dotyczącej, skorzystamy teraz jawnie z metody, którą dziedziczy każdy obiekt COM wraz z interfejsem **IUnknown**, służącej do zakończenia "życia" obiektu i zwolnienia zajmowanej przez niego pamięci. Ponieważ stworzyliśmy w naszej aplikacji tylko dwa obiekty, jeden typu **DIRECT3D8** i drugi **DIRECT3DDEVICES8**, więc nie będzie tu wielkiej filozofii, przypominam tylko, że obiekt Direct3D kasujemy jako ostatni (ponieważ on daje dostęp do pozostałych obiektów Direct3D). Jasne jest więc, że jeśli zniszczymy najpierw jego, to stracimy możliwość dostania się do pozostałych, zaśmiecających jakże cenną pamięć.

```
VOID Render()
{
    if( NULL == g_pd3dDevice )
        return;

    g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255), 1.0f, 0
);
    g_pd3dDevice->BeginScene();
    g_pd3dDevice->EndScene();

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

Podobnie jak poprzednio, przeanalizujemy tę funkcję linia po linii:

```
if( NULL == g_pd3dDevice )
    return;
```

Mając już okno, mając obiekty, możemy przystąpić do jakże długo wyczekiwanego momentu jakim jest rysowanie. W tym momencie to może słowo nieco na wyrost, ponieważ nie narysujemy jeszcze nic, ale będziemy mieli podstawę do tego, aby w następnej lekcji zacząć już zabawę z wierzchołkami, które zaczną się pojawiać na naszej scenie 3D. Na początku funkcji renderującej upewniamy się, czy nasze urządzenie, za pomocą którego chcemy rysować, na pewno istnieje. Tego fragmentu nie trzeba chyba nikomu wyjaśniać.

```
g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB( 0, 0, 255 ), 1.0f, 0 );
```

Jeśli już upewnimy się, że wszystko jest w porządku, możemy wywołać pierwszą metodę obiektu urządzenia. Ponieważ każda kolejna faza ruchu musi być namalowana na "czystej kartce", więc przed narysowaniem czegokolwiek musimy sobie taką kartkę wyczyścić. Urządzenie 3D oferuje nam taką metodę. Metoda **Clear()** powoduje wyczyszczenie tylnego bufora, który jest pierwotnym celem renderingu naszej sceny. Posiada ona możliwość czyszczenia kilku prostokątów zawierających się w tylnym buforze, których to tablica jest określona w drugim parametrze. Ponieważ drugi parametr ustawiamy na **NULL** (czyścimy całą powierzchnię bufora), to ten pierwszy musi być ustawiony na 0. Drugi parametr zawiera wskaźnik do tablicy przechowującej struktury typu **D3DRECT**, które określają prostokąty zdefiniowane w celu ich wyczyszczenia w tylnym buforze. Jeśli ustawiamy ten parametr na **NULL**, to (tak jak pisałem przy poprzednim parametrze), wyczyszczona będzie cała powierzchnia. Trzeci parametr to typ powierzchni, która ma zostać wyczyszczona. Możemy tworzyć powierzchnię nie tylko z przeznaczeniem do renderingu, ponieważ jeśli chcemy wykorzystywać na przykład bufor Z, czy też bufor szablonu (ang. *stencil buffer*), to trzeba będzie tworzyć dla nich specjalne powierzchnie. W naszym przykładzie mamy tylko bufor służący do renderingu, więc tylko on zostaje wyczyszczony. Czwarty parametr to kolor, jakim zostanie wyczyszczona powierzchnia. **D3DCOLOR\_XRGB()** to makro, które przekształci podane wartości na strukturę **D3DCOLOR**, która powinna zostać podana tutaj jako parametr. Następny parametr to nowa wartość dla bufora Z, w naszym przykładzie na razie jest bez znaczenia, więc ustawiamy ją na wartość 1.0f. Ostatni parametr to wartość potrzebna do działania bufora szablonu, więc w tym momencie też dla nas bez znaczenia, najlepiej ustawić ją na 0).

```
g_pd3dDevice->BeginScene();
g_pd3dDevice->EndScene();
```

Wszystkie obiekty 3D, które będą wyświetlane na ekranie, będą znajdować się na tzw. scenie. Nie mam tu na myśli tej, znanej z teatru. Będzie to swojego rodzaju scena wirtualna, fragment przestrzeni 3D, w której umieścimy nasze obiekty. Wśród ludzi zajmujących się grafiką 3D określenie "scena" przyjęło się już na stałe i stanowi dla nich rzecz tak naturalną, jak wierzchołek i tekstura. Twórcy DirectX Graphics poszli dalej i nawet metody urządzenia mają ją w swojej nazwie. Gdy chcemy wyprowadzić na ekran naszą scenę 3D, musimy o tym powiedzieć naszemu urządzeniu rysującemu. Do tego celu służy para metod, która podobnie jak ma to miejsce w OpenGL-u, powoduje to, że wszystko, co umieścimy pomiędzy nimi, będzie miało bezpośredni wpływ na to, co otrzymamy na ekranie. Są to metody przedstawione powyżej. Każda rozpoczęta scena musi być oczywiście zakończona. Wskazane jest także, aby jak największa ilość instrukcji rysujących była zawarta pomiędzy wywołaniami tych funkcji, a najlepiej wszystkie. Wielokrotne wywoływanie tej pary funkcji w jednej rysowanej klatce może nam doskonale spowolnić rendering. Tak więc pamiętajmy o tym!

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

Ta metoda urządzenia powoduje przerzucenie zawartości tylnego bufora (lub następnego w kolejce, jeśli jest ich więcej niż jeden) na przednią powierzchnię (ang. *front buffer*), czyli, innymi słowy, wyrysowanie naszej sceny na ekranie. Pierwszy parametr określa prostokąt na tylnym, buforze z jakiego kopiowane są dane, jeśli ma wartość **NULL**, to kopiowany jest cały bufor. Ważna rzecz, o jakiej należy wspomnieć to to, że jeśli prostokąt wykracza poza ramy powierzchni bufora, to jest on obcinany do jej granic. Drugi parametr to prostokąt docelowy na powierzchni przedniego bufora i tak samo jak w sytuacji tylnego bufora, jeśli przekroczone są granice powierzchni, to prostokąt jest obcinany. Trzeci parametr określa okno (uchwyt), którego obszar klienta jest traktowany jako cel rysowania sceny. Jeśli podamy **NULL**, to aktualnym uchwytem jest zawartość pola **hWndDeviceWindow** struktury **D3DPRESENT\_PARAMETERS**. W naszym przypadku jest to okno, które stworzyliśmy specjalnie dla celów aplikacji.

Jak widać funkcja renderująca jest bardzo prosta, ale może ona przybrać pokaźne rozmiary, jeśli będziemy renderować sporą liczbę obiektów i zmieniać im często właściwości takie jak teksturowanie, przezroczystość itp. Wszystko to jednak powinniśmy zrobić pomiędzy parą metod obiektu urządzenia renderującego **BeginScene()** i **EndScene()**. Tworzenie podstawy programu, rejestracja okna, tworzenie pętli komunikatów i wywołanie poszczególnych funkcji opisałem w tutorialu przedstawiającym podstawy programowania w Windows i nie ma potrzeby powtarzać tego drugi raz. Po skompilowaniu projektu i uruchomieniu programu powinniśmy otrzymać na ekranie załączony poniżej obrazek.

