helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

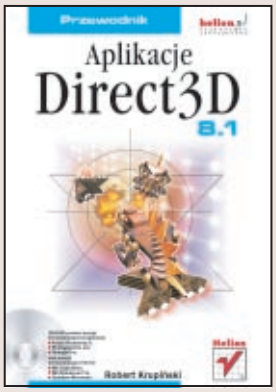
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Aplikacje Direct3D



Autor: Robert Krupiński
ISBN: 83-7197-877-4
Format: B5, stron: 180

Dzięki Direct3D możesz tworzyć nowoczesne gry, symulacje czy programy multimedialne. Jest on podstawowym standardem programowania grafiki trójwymiarowej w systemach operacyjnych zgodnych z Windows. Akceleracja sprzętowa, oferowana przez większość współczesnych kart graficznych oraz bogaty zbiór narzędzi dostępnych w wersji 8.1 umożliwia programowanie zaskakującej i efektownej grafiki 3D. Direct3D dostarcza programiście gotowych interfejsów, uwalniając go jednocześnie od konieczności zaznajamiania się ze wewnętrznymi funkcjami sprzętu.

Blisko 100 przykładowych projektów prezentujących wiele praktycznych zastosowań Direct3D wprowadzi Cię w świat programowania grafiki 3D.

W książce omówiono m.in.:

- Podstawy programowania w Direct3D
- Bufory werteksów
- Zarządzanie złożonymi obiektami (obiekt Mesh)
- Przekształcenia przestrzeni
- Światło i materiał, przeźroczystość
- Operowanie teksturami
- Zarządzanie obiektami leżącymi na jednej płaszczyźnie
- Pisanie kodu niezależnego od rodzaju karty graficznej

Autor zakłada, że Czytelnik potrafi posługiwać się pakietem Visual C++ i posiada umiejętność programowania w tym języku, korzysta z klas MFC, a także jest zaznajomiony z pojęciami dotyczącymi grafiki komputerowej. Jeśli spełniasz te warunki i chcesz kreować własne, trójwymiarowe światy na ekranie komputera, z pewnością pomoże Ci w tym ta książka.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Wprowadzenie 7

Rozdział 1. Aplikacje bazowe 11

Szkielet podstawowy 11
Kreator DirectX AppWizard 14

Rozdział 2. Bufory werteksów 17

Prymitywy 17
FVF 25
Bufory indeksów 30
Dynamiczna zmiana zawartości buforów 32
Łączenie buforów 34

Rozdział 3. Obiekt Mesh 37

Obiekty podstawowe 37
Klasy pomocnicze 40
Teselacja 41
Pliki .x 44

Rozdział 4. Przekształcenia przestrzeni 47

Okno widoku 47
Macierz świata 48
Macierz widoku 50
Macierz rzutowania 53
Kwaterniony 54

Rozdział 5. Światło i materiał 59

Ustawienia kolorów 59
Światło kierunkowe 63
Światło punktowe 64
Światło źródłowe 65
Wiele źródeł światła 66
Materiał 68

Rozdział 6. Blending werteksów 71

Tweening 71
Macierze deformujące 76

Rozdział 7. Alpha Blending 83

Rozdział 8. Bufor szablonu 89

Rozdział 9. Tekst 93

Rozdział 10. Tekstury 97

Tekstury płaskie 97
Tekstury otoczenia 100
Dwa etapy łączenia tekstur 106
Tekstura wichrująca i tekstura obiektu 109
Tekstura wichrująca i tekstura otoczenia 114
Tekstura obiektu, wichrująca i otoczenia 116
Tekstury przestrzenne 117
Dot3 119
Mapy wektorów normalnych i wektorów wichrujących 122

Rozdział 11. Potoki 125

VertexShader 125
PixelShader 136
Program uruchomieniowy 141

Rozdział 12. Z-bufor 143

Rozdział 13. Efekty 145

Rozdział 14. Płaszczyzny tnące 153

Rozdział 15. Mgła 155

Rozdział 16. Sprajty 159

Rozdział 17. Punkty 163

Rozdział 18. Wygaszacz ekranu 167

Rozdział 19. Interakcja 171

Zawartość CD-ROM-u 177

Skorowidz 181

Rozdział 10.
Tekstury

Tekstury zwiększają realizm renderowanych scen, nadają im specyficzny charakter. Jeśli zachowamy ustawienia konfiguracyjne sceny, a zmienimy teksturę, to możemy uzyskać ciekawy efekt, np. sceneria zmieni się z polarnej w pustynną. Dynamiczne zmienianie tekstur, różne sposoby ich łączenia, wyświetlanie kolejnych klatek tekstur na powierzchni obiektów, a nawet zaburzania współrzędnych tekstur — to zabiegi, które pozwalają wprowadzać coraz to nowe rozwiązania, efekty.

W przykładach przedstawimy różne rodzaje tekstur obsługiwanych przez DirectX oraz sposób ich stosowania.

Tekstury płaskie

Definicje struktur FVF dla obiektów zawierających teksturę zostały omówione w rozdziale „Bufory werteksów" na przykładzie projektu z katalogu *\Primitives\FVFPositionTexCoord*. Teraz pokażemy, w jaki sposób przygotować teksturę do wyświetlenia.

W deklaracji klasy `CMyApp` podajemy wskaźnik do interfejsu obsługującego teksturę:

```
private:
    LPDIRECT3DTEXTURE8 m_pTexture;
```

W konstruktorze inicjalizujemy to pole wartością `NULL`. Teksturę można załadować z pliku na kilka sposobów. My przedstawimy dwa z nich:

```
if( FAILED( D3DXCreateTextureFromFile(m_pd3dDevice, "tex.bmp",&m_pTexture) ) )
        return E_FAIL;

if( FAILED( D3DUtil_CreateTexture(m_pd3dDevice, , "tex.bmp",&m_pTexture) ) )
        return E_FAIL;
```

W slocie 0 ustawiamy teksturę jako aktywną, wykorzystywaną do renderingu:

```
m_pd3dDevice->SetTexture( 0,m_pTexture);
```

Obiekty, które posiadają współrzędne *u* i *v*, będą wykorzystywały tę teksturę. Na koniec należy pamiętać o zwolnieniu zasobów tekstury:

```
SAFE_RELEASE(m_pTexture);
```

Kolejnym krokiem będzie dynamiczna zmiana współrzędnych *u* i *v* tekstury. Projekt z katalogu *\Texture\TexRotate* (rysunki 10.1a i 10.1b) implementuje obrót współrzędnych tekstury wokół osi Z:

```
D3DXMATRIX mat;
D3DXMatrixRotationZ(&mat,sinf(m_fTime));
```



Rysunek 10.1a. *Obrót współrzędnych tekstury wokół osi Z*



Rysunek 10.1b. *Obrót współrzędnych tekstury wokół osi Z*

Macierz obrotu wokół osi Z jest umieszczana w zmiennej `mat`.

Macierz obrotów wykorzystujemy do zmiany współrzędnych tekstury. W metodzie `FrameMove` blokujemy bufory werteksów i uaktualniamy pozycje *u* i *v* tekstury:

```
D3DXVECTOR2 v;
DEFVERTEX* pVertices;
```

```
if(SUCCEEDED( m_pVB->Lock( 0, 4*sizeof(DEFVERTEX), (BYTE**)&pVertices, 0 ) ) )
{
    pVertices[0].uv=*D3DXVec2TransformCoord(&v,&m_v0,&mat);

    pVertices[1].uv=*D3DXVec2TransformCoord(&v,&m_v1,&mat);

    pVertices[2].uv=*D3DXVec2TransformCoord(&v,&m_v2,&mat);

    pVertices[3].uv=*D3DXVec2TransformCoord(&v,&m_v3,&mat);
}
m_pVB->Unlock();
```

Funkcja `D3DXVec2TransformCoord` przekształca wektor dwuelementowy, określony w drugim argumencie, przez macierz `mat`.

W przykładzie z katalogu *\Texture\TexTranslate* wykorzystaliśmy inną macierz przekształcenia współrzędnych. Jest to macierz translacji:

```
D3DXMatrixTranslation(&mat,1.3f*sinf(0.78f*m_fTime), 2.3f*cosf(1.3f*m_fTime),0.0f);
```

Rysunki 10.2a i 10.2b przedstawiają teksturę przemieszczaną na powierzchni obiektu.



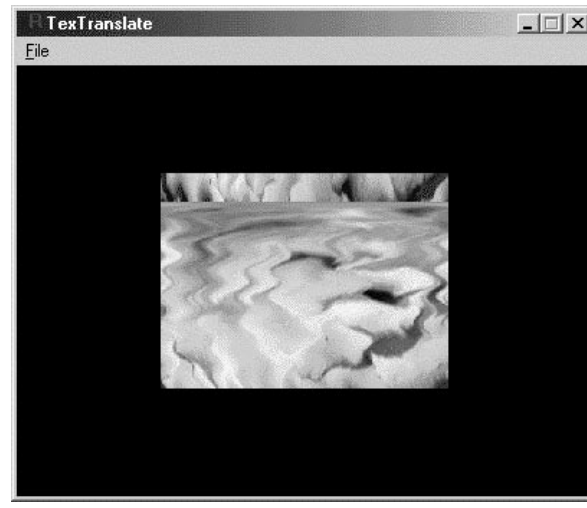
Rysunek 10.2a. *Przemieszczanie tekstury na powierzchni obiektu*

W projekcie *\Texture\TexAnim* wykorzystaliśmy sekwencyjną zmianę tekstury. Kolejna tekstura jest wyświetlana z częstością trzydziestu klatek na sekundę, co sprawia wrażenie animacji tekstury na powierzchni obiektu. Dodatkowo tekstura jest przemieszczana. Definiujemy tablicę tekstur:

```
private:
    LPDIRECT3DTEXTURE8 m_pTextures[NUM_TEXTURES];
```

W tablicy tej będą przechowywane kolejne klatki tekstury. Inicjalizujemy wartością `NULL` wszystkie tekstury:

```
for(int i=0;i<NUM_TEXTURES;i++)
    m_pTextures[i]=NULL;
```



Rysunek 10.2b. *Przemieszczanie tekstury na powierzchni obiektu*

Pamiętamy o zwolnieniu zasobów zajmowanych przez tekstury:

```
for(int i=0;i<NUM_TEXTURES;i++)
    SAFE_RELEASE( m_pTextures[i] );
```

Ładujemy wszystkie tekstury z plików do tablicy:

```
if( FAILED( D3DXCreateTextureFromFile( m_pd3dDevice, "tex0.bmp", &m_pTextures[0] ) ) )
    return E_FAIL;

.............

if( FAILED( D3DXCreateTextureFromFile( m_pd3dDevice, "tex9.bmp", &m_pTextures[9] ) ) )
    return E_FAIL;
```

Zanim rozpoczniemy renderowanie, wybieramy teksturę, która ma być wykorzystana i umieszczamy ją w slocie 0:

```
m_pd3dDevice->SetTexture( 0, m_pTextures[m_nActiveTexture] );
```

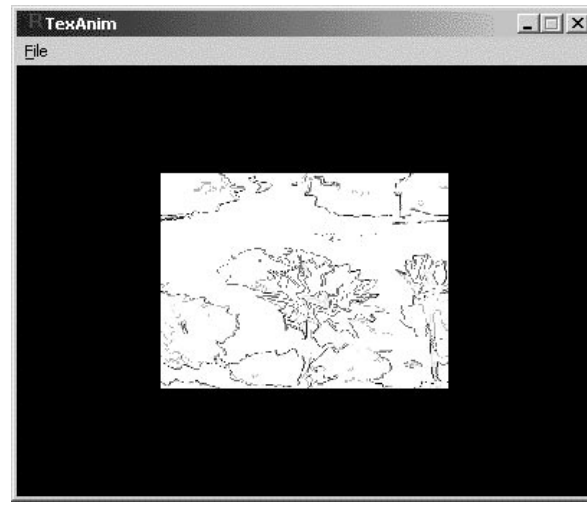
Rysunek 10.3a przedstawia sekwencję animacji, którą uzyskaliśmy. Rysunek 10.3b pokazuje pierwszą bitmapę, a rysunek 10.3c — piątą z sekwencji dziesięciu klatek.

W analizowanych przykładach tekstury zostały umieszczone na czworokątach, lecz nic nie stoi na przeszkodzie, by animować tekstury na powierzchni złożonych obiektów trójwymiarowych.

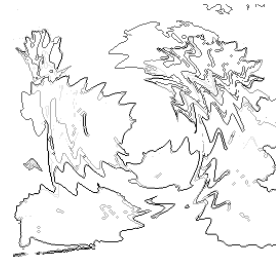
Tekstury otoczenia

Współrzędne tekstury mogą być wygenerowane automatycznie, nie jest konieczne podawanie ich wartości w strukturach werteksów. Powstaje wtedy efekt odbijania się tekstury na obiekcie, nazywany mapowaniem otoczenia (*Environment Mapping*).

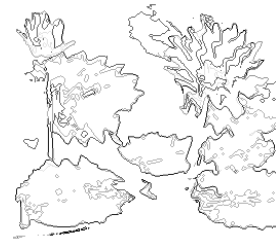
W podanych przykładach pokażemy sposób wykorzystania mapowania.



Rysunek 10.3a. *Wynik wyświetlania sekwencji tekstur na powierzchni obiektu*



Rysunek 10.3b. *Pierwsza klatka animowanej tekstury*



Rysunek 10.3c. *Piąta klatka animowanej tekstury*

Przykład (z katalogu *\Texture\EnvMapping1*) mapowania otoczenia został przedstawiony na rysunku 10.4a. Teksturę otoczenia pokazuje rysunek 10.4b.

Aby otrzymać taki efekt, należy wprowadzić w kodzie:

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT2 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_CAMERASPACENORMAL );
```

Flaga `D3DTSS_TEXCOORDINDEX` informuje urządzenie o sposobie generowania współrzędnych tekstury (wybrane `D3DTSS_TCI_CAMERASPACENORMAL`), zaś flaga `D3DTSS_TEXTURETRANSFORMFLAGS` informuje o liczbie współrzędnych tekstury, przekazywanych do urządzenia renderującego. W tym przypadku są to dwie współrzędne *u* i *v* (`D3DTTFF_COUNT2`). Ustawienia te wykonano dla poziomu 0 (pierwszy argument powyższych funkcji). Dodatkowo ustawiamy macierz przekształcenia współrzędnych tekstury:



Rysunek 10.4a. *Tekstura powstała w wyniku mapowania otoczenia (generowanie współrzędnych tekstury — flaga D3DTSS_TCI_CAMERASPACENORMAL). Do urządzenia przekazywane są dwie współrzędne tekstury*



Rysunek 10.4b. *Tekstura otoczenia*

```
D3DXMATRIX mat;
mat._11 = 0.5f; mat._12 = 0.0f; mat._13 = 0.0f; mat._14 = 0.0f;
mat._21 = 0.0f; mat._22 =-0.5f; mat._23 = 0.0f; mat._24 = 0.0f;
mat._31 = 0.0f; mat._32 = 0.0f; mat._33 = 1.0f; mat._34 = 0.0f;
mat._41 = 0.5f; mat._42 = 0.5f; mat._43 = 0.0f; mat._44 = 1.0f;
m_pd3dDevice->SetTransform( D3DTS_TEXTURE0, &mat );
```

Zmieniając dynamicznie wartości tej macierzy, można także uzyskać efekt przemieszczania się tekstury na powierzchni obiektu (np. odbijanie się otoczenia w szybie poruszającego się samochodu). Flaga `D3DTS_TEXTURE0` oznacza, że ustawiana jest macierz przekształcająca teksturę dla poziomu 0.

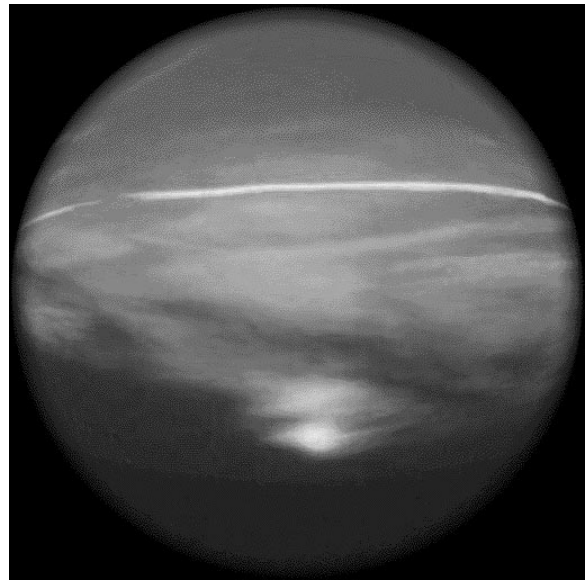
Do urządzenia można przekazać trzy współrzędne, wartość dwóch pierwszych zostanie obliczona przez podzielenie ich przez wartość trzeciej współrzędnej. W efekcie końcowym uzyskamy tylko dwie współrzędne do pobierania koloru z tekstury dwuwymiarowej. W tym przypadku ustawienia wyglądają tak:

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT3
|D3DTTFF_PROJECTED);
```

Jest to jedyna różnica w porównaniu do poprzedniego projektu. Efekt tego typu mapowania przedstawia rysunek 10.5a (projekt z katalogu *\Texture\EnvMapping1p*). Na rysunku 10.5b przedstawiono teksturę otoczenia. Mapowanie, w którym zastosowano flagi `D3DTTFF_PROJECTED`, pomaga wyeliminować błędy, jakie powstają na powierzchni obiektów w kierunku osi *Z* (tekstura nie jest nałożona, lecz rozciągnięta).



Rysunek 10.5*a*. *Mapowanie tekstury otoczenia na obiekt (generowanie współrzędnych tekstury — flaga D3DTSS_TCI_CAMERASPACENORMAL). Do urządzenia przekazywane są trzy współrzędne tekstury*

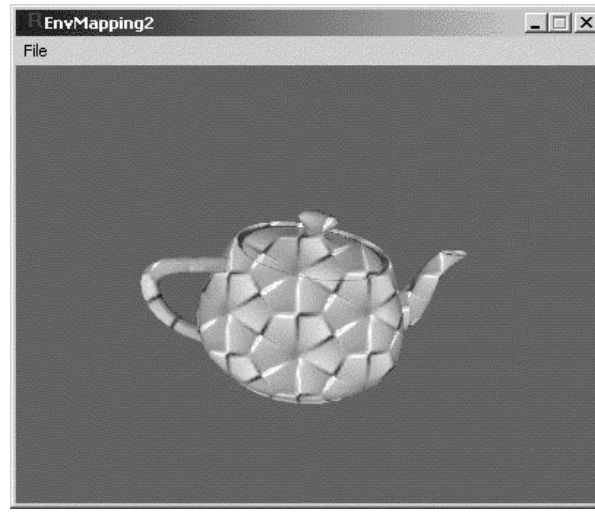


Rysunek 10.5*b*. *Tekstura otoczenia*

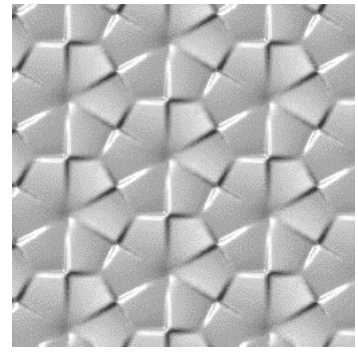
Jeśli zastosujemy flagę D3DTSS_TCI_CAMERASPACEPOSITION, to otrzymamy inny tryb generowania współrzędnych tekstury. Ustawiamy wtedy:

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT2 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_CAMERASPACEPOSITION );
```

W tym przypadku będą również wykorzystane tylko dwie współrzędne tekstury. Wynik mapowania oraz teksturę otoczenia przedstawiają kolejno rysunek 10.6a i b (projekt z katalogu *\Texture\EnvMapping2*).



Rysunek 10.6a. *Mapowanie tekstury otoczenia na obiekt (generowanie współrzędnych tekstury — flaga D3DTSS_TCI_CAMERASPACEPOSITION). Do urządzenia przekazywane są dwie współrzędne tekstury*

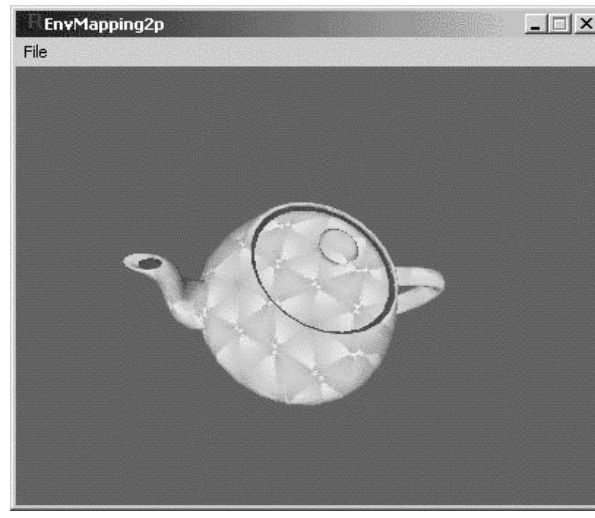


Rysunek 10.6a. *Tekstura otoczenia*

Ten sam zabieg mapowania, ale dla flagi D3DTTFF_PROJECTED przedstawia rysunek 10.7 (projekt z katalogu *\Texture\EnvMapping2p*).

Tryb generowania współrzędnych tekstury D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR stosujemy w przypadku tekstur typu sześciennego. Ustawienia są wtedy następujące:

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT3 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, D3DTSS_TCI_
CAMERASPACEREFLECTIONVECTOR );
```



Rysunek 10.7. *Mapowanie tekstury otoczenia na obiekt (generowanie współrzędnych tekstury — flaga D3DTSS_TCI_CAMERASPACEPOSITION). Do urządzenia przekazywane są trzy współrzędne tekstury*

W tym przypadku generowane są trzy współrzędne, które są przekazywane do urządzenia w celu indeksowania tekstury. Sposób generowania wektora odbicia, który właśnie służy do indeksowanie tekstury, może być modyfikowany za pomocą flagi `D3DRS_LOCALVIEWER`, która przyjmuje wartości typu `BOOL`.

Projekt (*\Texture\EnvMapCube*) przedstawia zastosowanie mapowania sześciennego. Tekstura ma postać sześcianu, na którego każdą ścianę można nałożyć inną płaską teksturę otoczenia.

Definiujemy teksturę sześcienną:

```
private:
    LPDIRECT3DCUBETEXTURE8 m_pTextureCube;
```

Inicjalizujemy ją wartością `NULL` w konstruktorze:

```
m_pTextureCube=NULL;
```

Ładujemy z pliku *.dds* przygotowaną wcześniej teksturę sześcienną:

```
if( FAILED( D3DXCreateCubeTextureFromFile(m_pd3dDevice, "tex.dds", &m_pTextureCube) ) )
    return E_FAIL;
```

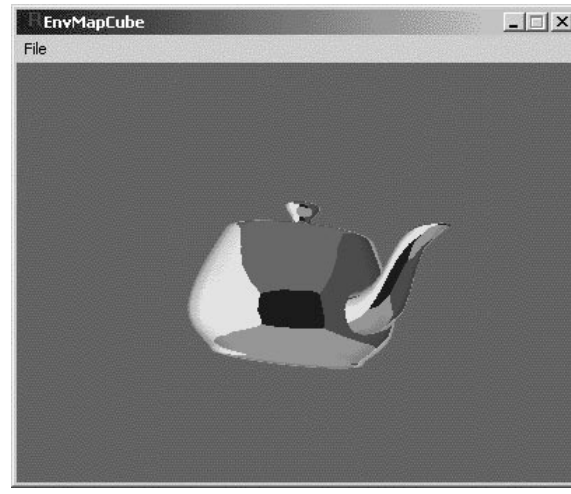
Tekstura została tak przygotowana, aby każda ściana była innego koloru. Efekt nałożenia jej na obiekt jest więc widoczny (rysunek 10.8).

Tekstury *.dds* można wykonać w programie *DxTex.exe* z pakietu SDK 8.1. Ustawiamy tę teksturę jako aktywną w slocie 0:

```
m_pd3dDevice->SetTexture( 0,m_pTextureCube);
```

I dopiero teraz można wykonać rendering obiektu. Na koniec należy zwolnić zasoby tekstury.

```
SAFE_RELEASE(m_pTextureCube);
```



Rysunek 10.8. *Mapowanie tekstury otoczenia na obiekt (generowanie współrzędnych tekstury — flaga D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR). Do urządzenia przekazywane są trzy współrzędne tekstury*

Przed rozpoczęciem działania aplikacja musi się upewnić, czy urządzenie obsługuje ten rodzaj tekstur:

```
if( 0 == ( pCaps->TextureCaps & D3DPTEXTURECAPS_CUBEMAP ) )
    return E_FAIL;
```

W bardziej skomplikowanym przykładzie można na teksturę otoczenia renderować obiekty zdefiniowanej sceny, a następnie wykorzystać tak przygotowaną teksturę do rysowania obiektów zapisywanych w buforze obrazu. Otrzymamy efekt wzajemnego odbijania się obiektów na swoich powierzchniach.

Dwa etapy łączenia tekstur

Jeden werteks może posiadać informacje o współrzędnych dla ośmiu tekstur. W tym podrozdziale przedstawimy sposób łączenia tekstur w dwóch etapach (dla dwóch tekstur). Każdy werteks będzie zawierał współrzędne dla dwóch tekstur `uv` i `uv1`:

```
struct DEFVERTEX
{
    FLOAT x, y, z;
    D3DXVECTOR2 uv;
    D3DXVECTOR2 uv1;
};
```

Definicja FVF dla takiej struktury przybiera postać:

```
#define D3DFVF_DEFVERTEX (D3DFVF_XYZ|D3DFVF_TEX2|D3DFVF_TEXCOORDSIZE2(0)| D3DFVF_
TEXCOORDSIZE2(1))
```

Makro `D3DFVF_TEXCOORDSIZE2(x)` oznacza, że tekstura o indeksie *x* ma dwa parametry *u* i *v*.

Tekstury ładujemy z plików i ustawiamy w odpowiednich slotach 0 i 1:

```
m_pd3dDevice->SetTexture( 0, m_pTextures[m_nActiveTexture] );
m_pd3dDevice->SetTexture( 1, m_pTexture);
```

Tekstura w slocie zerowym będzie się zmieniała, dlatego wybieramy z tablicy jedną z tekstur (będą wyświetlane sekwencyjnie — animacja). Teraz wprowadzamy informacje na temat tego, co będzie się działo z teksturą w danym slocie. Dla etapu 0:

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_SELECTARG1);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP,   D3DTOP_DISABLE);
```

Flaga D3DTSS_TEXCOORDINDEX ustawia indeks informujący, które współrzędne tekstury są przeznaczone dla tej tekstury w strukturze werteksa (indeks 0 to uv). Flaga D3DTSS_COLOROP oznacza rodzaj operacji dla kolorów *rgb*, a flaga D3DTSS_ALPHAOP — operację dla kanału alfa. Flagi D3DTSS_COLORARG1 i D3DTSS_COLORARG2 oznaczają odpowiednio argumenty dla określonej operacji *rgb*. Operacja D3DTOP_SELECTARG1 to wybór argumentu pierwszego D3DTA_TEXTURE, który oznacza teksturę. Argument drugi D3DTA_DIFFUSE, oznaczający kolor, jest ignorowany. Dla kanału alfa operacja jest wyłączona D3DTOP_DISABLE. Wynikowe wartości są przekazywane do kolejnego slotu.

W drugim etapie jako pierwszy argument wybieramy teksturę ze slotu 1 (D3DTA_TEXTURE) oraz jako drugi argument wynik poprzedniej operacji (D3DTA_CURRENT) ze slotu 0:

```
D3DTEXTUREOP m_eTexOp;
m_eTexOp=D3DTOP_MODULATE;

.............

m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 1 );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP,   m_eTexOp);
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_CURRENT );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_ALPHAOP,   D3DTOP_DISABLE);
```

Parametr wyliczeniowy m_eTexOp jest jedną z dostępnych operacji łączenia tekstur:

Operacja	Testowana flaga D3DTEXOPCAPS	Opis
D3DTOP_DISABLE	D3DTEXOPCAPS_DISABLE	wyłączenie
D3DTOP_SELECTARG1	D3DTEXOPCAPS_SELECTARG1	wybór pierwszego argumentu
D3DTOP_SELECTARG2	D3DTEXOPCAPS_SELECTARG2	wybór drugiego argumentu
D3DTOP_MODULATE	D3DTEXOPCAPS_MODULATE	$Arg_1 \cdot Arg_2$
D3DTOP_MODULATE2X	D3DTEXOPCAPS_MODULATE2X	$(Arg_1 \cdot Arg_2) \cdot 2$
D3DTOP_MODULATE4X	D3DTEXOPCAPS_MODULATE4X	$(Arg_1 \cdot Arg_2) \cdot 4$
D3DTOP_ADD	D3DTEXOPCAPS_ADD	$Arg_1 + Arg_2$
D3DTOP_ADDSIGNED	D3DTEXOPCAPS_ADDSIGNED	$Arg_1 + Arg_2 - 0.5$
D3DTOP_ADDSIGNED2X	D3DTEXOPCAPS_ADDSIGNED2X	$(Arg_1 + Arg_2 - 0.5) \cdot 2$
D3DTOP_SUBTRACT	D3DTEXOPCAPS_SUBTRACT	$Arg_1 - Arg_2$
D3DTOP_ADDSMOOTH	D3DTEXOPCAPS_ADDSMOOTH	$Arg_1 + Arg_2 \cdot (1 - Arg_1)$

Typ wyliczeniowy D3DTEXTUREOP obejmuje znacznie większą liczbę operacji. W tabeli zostały podane tylko te, które zostały wykorzystane w opisywanej implementacji.

W trzecim etapie nie będziemy wykonywali żadnych operacji, więc wywołujemy funkcje:

```
m_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLOROP,   D3DTOP_DISABLE );
m_pd3dDevice->SetTextureStageState( 2, D3DTSS_ALPHAOP,   D3DTOP_DISABLE );
```

Sprawdzamy, czy urządzenie wykonuje określoną operację:

```
if(m_d3dCaps.TextureOpCaps&D3DTEXOPCAPS_MODULATE2X)
    pB4->EnableWindow(TRUE);
else
    pB4->EnableWindow(FALSE);
```

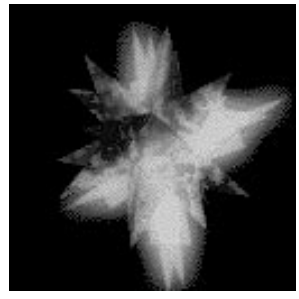
Należy także sprawdzić, czy są dostępne dwa sloty:

```
if(pCaps->MaxTextureBlendStages<2)
    bCapsAcceptable=FALSE;
```

Przykład z katalogu *\Texture\TexOp* to aplikacja MFC, która pozwala przetestować wybrane sposoby łączenia tekstur (rysunek 10.9a). Rysunek 10.9b pokazuje pierwszą teksturę z sekwencji dziesięciu tekstur, a rysunek 10.9c — piątą.

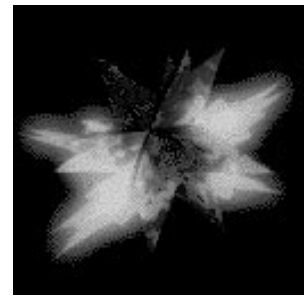


Rysunek 10.9a. *Aplikacja MFC — różne sposoby łączenia tekstur*



Rysunek 10.9b. *Pierwsza klatka animowanej tekstury*

Rysunek 10.9c.
Piąta klatka animowanej tekstury



Różne tekstury można wykorzystać do symulowania oświetlania obiektów światłem monochromatycznym lub wielobarwnym.

Tekstura wichrująca i tekstura obiektu

W tym podrozdziale przedstawimy zastosowanie tekstury wichrującej w slocie 0 oraz tekstury obiektu w slocie 1. Tekstura wichrująca (*Bump Texture*) imituje nierówności powierzchni obiektów. W zależności od formatu tekstura wichrująca może nie tylko modyfikować współrzędne tekstury obiektu, ale i regulować jasność pikseli.

Przykład z katalogu *\Texture\TexBumpTranslate* pokazuje, jak oddziałuje tekstura wichrująca na teksturę obiektu (rysunek 10.10a). Dla porównania tekstura obiektu została przedstawiona na rysunku 10.10b. Na statycznych obrazach widać wyraźnie efekt, kiedy obserwuje się wieżę i dach (na lewo od wieży). Podczas dynamicznego działania programu jest to łatwiejsze do zauważenia.

Rysunek 10.10a.
Tekstura obiektu modyfikowana przez teksturę wichrującą



Tekstura wichrująca jest przemieszczana w poziomie, co daje efekt falowania powierzchni obiektu. Oto operacje, które należy wykonać, jeśli chcemy zastosować teksturę wichrującą.



Rysunek 10.10b. *Tekstura obiektu*

Najpierw sprawdzamy, czy urządzenie obsługuje *BumpMapping* za pomocą flagi `D3DTEXOPCAPS_BUMPENVMAP` oraz czy obsługuje określony format tekstury wichrującej `D3DFMT_V8U8`:

```
if(!( pCaps->TextureOpCaps & D3DTEXOPCAPS_BUMPENVMAP ))
    return E_FAIL;

if(FAILED( m_pD3D->CheckDeviceFormat( pCaps->AdapterOrdinal,
pCaps->DeviceType, Format,0, D3DRTYPE_TEXTURE,D3DFMT_V8U8 ) ) )
        return E_FAIL;
```

Ładujemy teksturę obiektu i umieszczamy ją w slocie 1. Załadowana tekstura wichrująca jest przekształcana, zapisywana w formacie V8U8 i umieszczana w slocie 0:

```
if( FAILED( D3DXCreateTextureFromFile(m_pd3dDevice, "tex.bmp", &m_pTexture)))
    return E_FAIL;

m_pd3dDevice->SetTexture( 1,m_pTexture);

if( FAILED( LoadBumpTextureFromFile("tex1.bmp",&m_pTextureBump)))
    return E_FAIL;

m_pd3dDevice->SetTexture( 0,m_pTextureBump);
```

Tak wygląda kod funkcji, za pomocą której zamienia się format bitmapy z A8R8G8B8 na V8U8:

```
HRESULT CMyApp::LoadBumpTextureFromFile(TCHAR* strTexture,LPDIRECT3DTEXTURE8* ppTexture)
{
    LPDIRECT3DTEXTURE8 pTexture;

    if( FAILED( D3DXCreateTextureFromFile(m_pd3dDevice, strTexture, &pTexture) ) )
    {
        SAFE_RELEASE(pTexture);
            return E_FAIL;
    }

    D3DSURFACE_DESC pDesc;

    if( FAILED(pTexture->GetLevelDesc(0,&pDesc)))
    {
        SAFE_RELEASE(pTexture);
            return E_FAIL;
    }


    if( FAILED( m_pd3dDevice->CreateTexture( pDesc.Width, pDesc.Height, 1, 0,
D3DFMT_V8U8, D3DPOOL_MANAGED, ppTexture) ) )
    {
    SAFE_RELEASE(pTexture);
        return E_FAIL;
    }

    D3DLOCKED_RECT lockrect;

    pTexture->LockRect( 0, &lockrect, 0, 0 );
    DWORD dwSrcPitch = (DWORD)lockrect.Pitch;
    BYTE* pSrcBits = (BYTE*)lockrect.pBits;

    (*ppTexture)->LockRect( 0, &lockrect, 0, 0 );
        DWORD dwDestPitch = (DWORD)lockrect.Pitch;
        BYTE* pDestBits = (BYTE*)lockrect.pBits;

    for( DWORD y=0; y<pDesc.Height; y++ )
        {
        for( DWORD x=0; x<pDesc.Width; x++ )
            {
            *(pDestBits+2*x+1)=*(pSrcBits+4*x+1);//v=g
            *(pDestBits+2*x)=*(pSrcBits+4*x+2);//u=r
        }

        pDestBits+=dwDestPitch;
        pSrcBits+=dwSrcPitch;
    }

    pTexture->UnlockRect(0);
        (*ppTexture)->UnlockRect(0);

    SAFE_RELEASE(pTexture);

    return S_OK;
}
```

Bajty odpowiadające kolorom są umieszczane w pamięci w następującej kolejności: B, G, R, A, B1, G1, R1, A1, B2, G2, R2, A2... Podobnie umieszczane są bajty odpowiadające parametrom wichrującym: *dU*, *dV*, *dU1*, *dV1*, *dU2*, *dV2*...Wartości *dU* i *dV* z przedziału ⟨−128,127⟩ odpowiadają parametrom wichrującym z przedziału od –1 do 1. Parametr *dU* odnosi się do kierunku *X*, zaś parametr *dV* do kierunku *Y*. Funkcja `GetLevelDesc` zwraca w strukturze `D3DSURFACE_DESC` parametry tekstury wejściowej. Wykorzystujemy wysokość i szerokość tekstury załadowanej z pliku do utworzenia tekstury o tych samych wymiarach (`CreateTexture`). Obydwie tekstury blokujemy funkcją `LockRect`, aby można było przepisać dane. Odpowiednio przepisujemy kolor czerwony do *dU* i kolor zielony do *dV*. Po zakończeniu operacji blokady są zwalniane (`UnlockRect`).

Ustawiamy parametry macierzy 2×2 przekształcającej wartości *dU* i *dV*:

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_BUMPENVMAT00, FLOATtoDW(0.01f));
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_BUMPENVMAT01, FLOATtoDW(0.0f));
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_BUMPENVMAT10, FLOATtoDW(0.0f));
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_BUMPENVMAT11, FLOATtoDW(0.01f));
```

Nowe współrzędne tekstury obliczane są według wzorów:

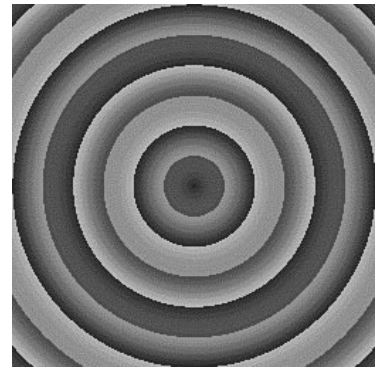
$$dU' = dU \cdot M_{00} + dV \cdot M_{10}$$
$$dV' = dU \cdot M_{01} + dV \cdot M_{11}$$

gdzie M_{00}, M_{01}, M_{10}, M_{11} są parametrami ustawionej poprzednio macierzy 2×2.

Stosujemy operacje tekstury wichrującej dla slotu 0:

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_BUMPENVMAP );
```

Na koniec można wykonać rendering. Rysunek 10.11 przedstawia teksturę wichrującą zastosowaną w przykładzie.



Rysunek 10.11. *Przykładowa tekstura wichrująca (format V8U8)*

Przykład z katalogu *\Texture\TexBumpLTranslate* dotyczy sposobu wykorzystania tekstury wichrującej z luminancją o formacie L6V5U5 (rysunek 10.12).

Sprawdzamy, czy urządzenie obsługuje *BumpMapping* z luminancją (flaga `D3DTOP_BUMPENVMAPLUMINANCE`) oraz czy obsługuje określony format tekstury wichrującej `D3DFMT_L6V5U5`:



Rysunek 10.12. *Zastosowanie tekstury wichrującej z luminancją*

```
if(!( pCaps->TextureOpCaps & D3DTOP_BUMPENVMAPLUMINANCE ))
    return E_FAIL;

if(FAILED( m_pD3D->CheckDeviceFormat( pCaps->AdapterOrdinal,
pCaps->DeviceType, Format,0, D3DRTYPE_TEXTURE,D3DFMT_L6V5U5) ) )
    return E_FAIL;
```

Dane dla tekstury wichrującej ładujemy z pliku za pomocą przygotowanej wcześniej funkcji `LoadBumpTextureFromFile`. W kolorze czerwonym jest przechowywana wartość *dU*, w kolorze zielonym — *dV*, a w niebieskim — wartość luminancji. Dane te są zapisywane za pomocą 16 bitów, tak że 6 bitów jest przeznaczonych dla luminancji, 5 bitów — dla *dU* i 5 bitów — dla *dV*. Należy pamiętać, że w komputerach PC bity są ułożenie według zasady *Little-Endian* (tzn. młodszy bajt znajduje się przed starszym). Część kodu odpowiedzialnego za pakowanie bitów w funkcji `LoadBumpTextureFromFile` wygląda w ten sposób:

```
#define PACKBITS(l,dv,du)((WORD)(((((l)>>2)&0x3f)<<10)|((((dv)>>3)&0x1f)<<5)|
(((du)>>3)&0x1f)))

.............

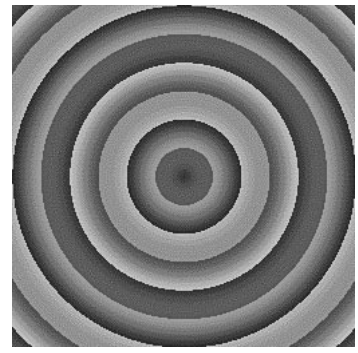
du=*(pSrcBits+4*x+2);//r
dv=*(pSrcBits+4*x+1);//g
lumin=*(pSrcBits+4*x);//b

*(WORD*)(pDestBits+2*x)=PACKBITS(lumin,dv,du);
```

Pozostała część kodu funkcji `LoadBumpTextureFromFile` jest taka sama jak w funkcji omawianej przy okazji projektu *\Texture\TexBumpTranslate*.

Wartości luminancji odpowiadają poziomom 0 – 255. Teksturę wichrującą z luminancją przedstawia rysunek 10.13.

Zanim przeprowadzimy renderowanie obiektu, tekstura wichrująca z luminancją wymaga dwóch dodatkowych parametrów: współczynnika skalowania (flaga `D3DTSS_BUMPENVLSCALE`) i offsetu (flaga `D3DTSS_BUMPENVLOFFSET`) dla luminancji:



Rysunek 10.13. *Przykładowa tekstura wichrująca z luminancją (format L6V5U5)*

```
m_pd3dDevice->SetTextureStageState(0,D3DTSS_BUMPENVLSCALE, FLOATtoDW(2.5f));
m_pd3dDevice->SetTextureStageState(0,D3DTSS_BUMPENVLOFFSET, FLOATtoDW(0.4f));
```

Natężenie piksela w tym przypadku jest obliczane według wzoru:

$$L' = L \cdot S_L + O_L$$

gdzie:

L' — luminancja wyjściowa,

L — luminancja pobrana z tekstury wichrującej,

S_L — współczynnik skalujący,

O_L — offset.

W slocie 0 należy jeszcze zaznaczyć, że wykorzystywana jest tekstura z luminancją:

```
m_pd3dDevice->SetTextureStageState(0,D3DTSS_COLOROP, D3DTOP_BUMPENVMAPLUMINANCE);
```

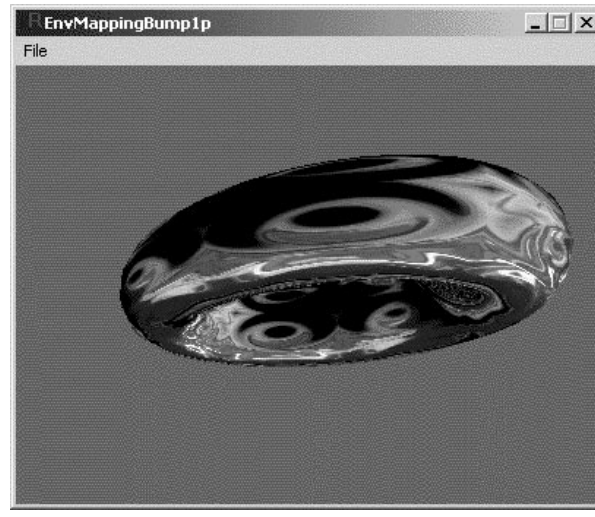
i można wykonać rendering.

Tekstura wichrująca i tekstura otoczenia

Przykłady z poprzedniego podrozdziału zmodyfikujemy w ten sposób, że zamiast tekstury obiektu będzie stosowana tekstura otoczenia. W slocie 0 pozostaje tekstura wichrująca, a w slocie 1 znajdzie się tekstura otoczenia. Należy ustawić urządzenie, aby generowało automatycznie współrzędne dla tekstury o indeksie 1:

```
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT3|
D3DTTFF_PROJECTED);
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, D3DTSS_TCI_
CAMERASPACENORMAL | 1 );
```

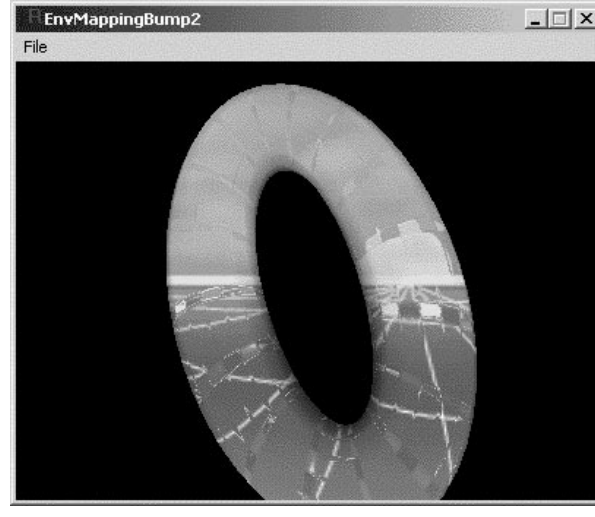
Struktura FVF zawiera teraz tylko jedną parę współrzędnych (dla tekstury wichrującej). Przykład z katalogu *\Texture\EnvMappingBump1p* pokazuje wynik działania tekstury wichrującej i otoczenia (rysunek 10.14).



Rysunek 10.14. *Połączenie tekstury wichrującej i tekstury otoczenia (generowanie współrzędnych tekstury otoczenia — flaga D3DTSS_TCI_CAMERASPACENORMAL). Do urządzenia przekazywane są trzy współrzędne tekstury otoczenia*

Przykład z katalogu *\Texture\EnvMappingBump2* dotyczy następujących parametrów tekstury otoczenia (rysunek 10.15):

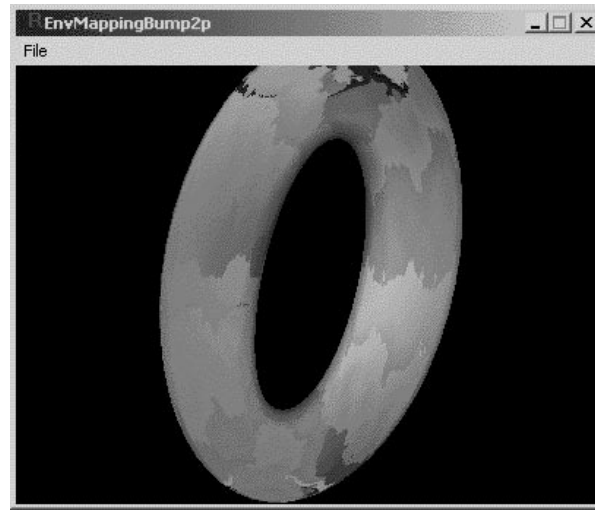
```
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT2 );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_CAMERASPACEPOSITION | 1 );
```



Rysunek 10.15. *Połączenie tekstury wichrującej i tekstury otoczenia (generowanie współrzędnych tekstury otoczenia — flaga D3DTSS_TCI_CAMERASPACEPOSITION). Do urządzenia przekazywane są dwie współrzędne tekstury otoczenia*

zaś przykład z katalogu *\Texture\EnvMappingBump2p* dotyczy parametrów (rysunek 10.16):

```
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT3|
D3DTTFF_PROJECTED);
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, D3DTSS_TCI_
CAMERASPACEPOSITION | 1 );
```



Rysunek 10.16. *Połączenie tekstury wichrującej i tekstury otoczenia (generowanie współrzędnych tekstury otoczenia — flaga D3DTSS_TCI_CAMERASPACEPOSITION). Do urządzenia przekazywane są trzy współrzędne tekstury otoczenia*

Tekstura obiektu, wichrująca i otoczenia

Łączenie trzech tekstur wymaga trzech slotów. Sprawdzamy, czy są one dostępne:

```
if(pCaps->MaxTextureBlendStages<3)
return E_FAIL;
```

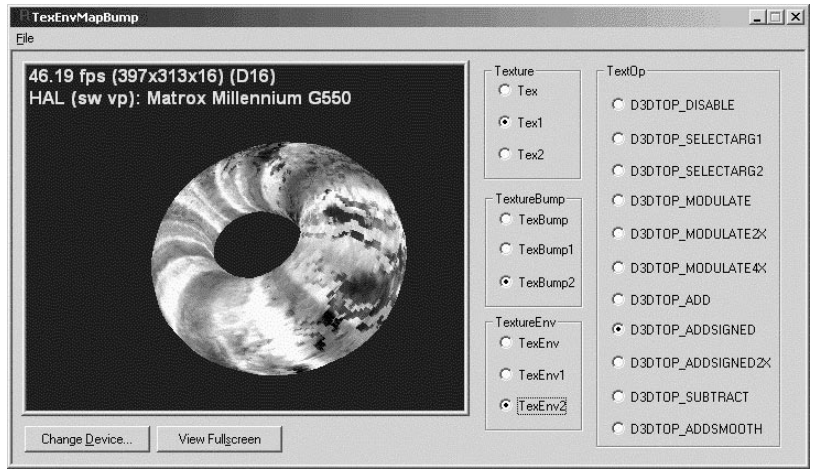
W slocie 0 umieszczamy teksturę obiektu, w slocie 1 — teksturę wichrującą, a w slocie 2 — teksturę otoczenia. Wprowadzamy następujące ustawienia we wszystkich trzech slotach:

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );

m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 1 );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP,   D3DTOP_BUMPENVMAP );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_CURRENT);
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1);
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );

m_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLOROP,   m_eTexOp);
m_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLORARG2, D3DTA_CURRENT);
m_pd3dDevice->SetTextureStageState( 2, D3DTSS_ALPHAOP,   D3DTOP_DISABLE );
m_pd3dDevice->SetTextureStageState( 2, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT3|
D3DTTFF_PROJECTED);
m_pd3dDevice->SetTextureStageState( 2, D3DTSS_TEXCOORDINDEX, D3DTSS_TCI_
CAMERASPACEPOSITION | 2);
```

W slocie 2 wprowadziliśmy zmienną `m_eTexOp`, aby można było zmieniać operację łączenia tekstur. Projekt z katalogu *\Texture\TexEnvMapBump*, przedstawiony na rysunku 10.17, umożliwia podglądanie łączenia trzech tekstur. Użytkownik może wybrać jedną z dziewięciu tekstur (trzy możliwości dla tekstury obiektu, trzy dla tekstury otoczenia i trzy dla tekstury wichrowania). Przełączniki trybu łączenia tekstur zmieniają operacje w slocie 2.



Rysunek 10.17. *Aplikacja MFC pozwala wybrać dowolną teksturę otoczenia, wichrującą, obiektu oraz sposób łączenia tekstur*

Tekstury przestrzenne

Teksturę przestrzenna (*Volume Map*) tworzymy przez dodanie trzeciej współrzędnej do płaskiej tekstury. Otrzymamy wtedy prostopadłościan o określonej szerokości, wysokości i głębokości. Elementem jednostkowym takiej figury jest teksel. Tekstury przestrzenne znajdują zastosowanie w medycynie, są wykorzystywane do przedstawiania kolejnych przekrojów, np. głowy. Taki obraz przestrzenny można oglądać pod dowolnym kątem.

W przykładzie z katalogu *\Texture\TexVolume* bryłę przestrzenną tekstury wypełniamy kolorami i będziemy ją wyświetlali, zmieniając trzecią współrzędną tekstury. Sprawdzamy, czy urządzenie obsługuje tekstury przestrzenne:

```
if( 0 == ( pCaps->TextureCaps & D3DPTEXTURECAPS_VOLUMEMAP ) )
    return E_FAIL;
```

Struktura werteksa z trzema współrzędnymi tekstury `u`, `v` i `w` wygląda w ten sposób:

```
struct DEFVERTEX
{
    FLOAT x, y, z;
    FLOAT u,v,w;
};
```

Deklaracja FVF ma postać:

```
#define D3DFVF_DEFVERTEX (D3DFVF_XYZ|D3DFVF_TEX1|D3DFVF_TEXCOORDSIZE3(0))
```

Deklarujemy wskaźnik do struktury tekstury przestrzennej:

```
private:
    LPDIRECT3DVOLUMETEXTURE8 m_pTexture;
```

i inicjalizujemy w konstruktorze wartością NULL. Na końcu działania aplikacji zwalniamy zasoby interfejsu tekstury:

```
SAFE_RELEASE(m_pTexture);
```

Korzystając z funkcji CreateVolumeTexture, tworzymy teksturę przestrzenną:

```
if( FAILED(m_pd3dDevice->CreateVolumeTexture( 16, 16, 16, 1, 0,
D3DFMT_A8R8G8B8, D3DPOOL_MANAGED, &m_pTexture )))
    return E_FAIL;
```

Funkcja ta utworzy teksturę o wymiarach 16×16×16 i formacie D3DFMT_A8R8G8B8. Wypełniamy teksturę, korzystając z wywołania funkcji:

```
if( FAILED(D3DXFillVolumeTexture(m_pTexture, FillMyTextureVolume,NULL)))
    return E_FAIL;
```

Drugim argumentem jest zdefiniowana wcześniej przez programistę funkcja wypełniająca teksturę. Oto przykład implementacji takiej funkcji:

```
VOID FillMyTextureVolume(D3DXVECTOR4* pOut,D3DXVECTOR3* pTexCoord,D3DXVECTOR3* pTexelSize,
LPVOID pData)
{
    pOut->x=pTexCoord->x;
    pOut->y=pTexCoord->y;
    pOut->z=pTexCoord->z;
}
```

Funkcja ta w pierwszym parametrze zawiera strukturę tekstury *rgba* do wypełnienia, o współrzędnej tekstury określonej w drugim parametrze. Trzeci parametr to wielkość teksela, czwarty to wartość, którą przekazał programista w trzecim parametrze wywołania funkcji D3DXFillVolumeTexture. W tym przypadku będzie to NULL. Przykład przepisuje pozycje w teksturze do wartości kolorów.

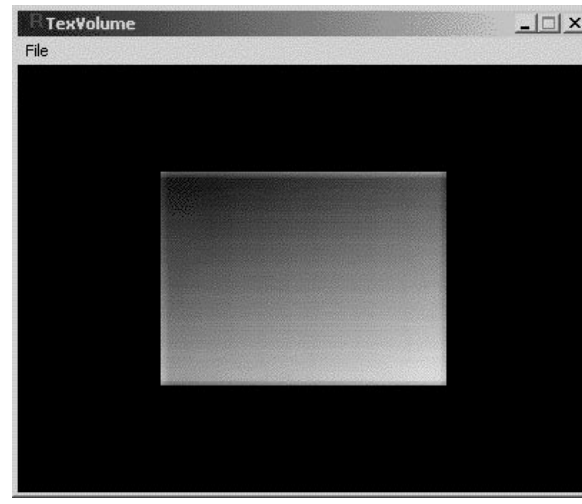
Ustawiamy teksturę w slocie 0. W metodzie FrameMove zmieniamy współrzędną *w*, w efekcie wyświetlane będą kolejne przekroje tekstury na czworokącie:

```
DEFVERTEX* pVertices;
if(SUCCEEDED( m_pVB->Lock( 0, 4*sizeof(DEFVERTEX), (BYTE**)&pVertices, 0 ) ) )
{
    for(int i=0; i<4; i++ )
        pVertices[i].w = m_fTime;

    m_pVB->Unlock();
}
```

Wynik działania programu przedstawia rysunek 10.18.

W projekcie z katalogu *\Texture\TexVolume1* zastosowaliśmy rotację współrzędnych tekstury, tak że na czworokącie wyświetlane są różne przekroje tekstury przestrzennej. Kod pokazuje, jak zmieniają się współrzędne tekstury:



Rysunek 10.18. *Na powierzchni obiektu wyświetlana jest tekstura o wymiarach 16×16×16, ze zmienną współrzędną w*

```
D3DXMATRIX matRot,mat;
D3DXQUATERNION q;
D3DXVECTOR3 vOut;

D3DXMatrixRotationYawPitchRoll( &matRot,m_fTime*1.25f,m_fTime,0.0f);

D3DXMatrixAffineTransformation(&mat,1.0f,&D3DXVECTOR3(0.7071f,0.5f,0.0f),
D3DXQuaternionRotationMatrix(&q,&matRot), &D3DXVECTOR3(-0.20710678f, 0.0f,0.5f));

D3DXVECTOR3 v;
DEFVERTEX* pVertices;
if(SUCCEEDED( m_pVB->Lock( 0, 4*sizeof(DEFVERTEX), (BYTE**)&pVertices, 0 ) ) )
{
    pVertices[0].uvw=*D3DXVec3TransformCoord(&v,&m_v0,&mat);

    pVertices[1].uvw=*D3DXVec3TransformCoord(&v,&m_v1,&mat);
    pVertices[2].uvw=*D3DXVec3TransformCoord(&v,&m_v2,&mat);
    pVertices[3].uvw=*D3DXVec3TransformCoord(&v,&m_v3,&mat);
}
```

Funkcja `D3DXVec3TransformCoord` przekształca wektor trzyelementowy, określony w drugim argumencie, przez uprzednio przygotowaną macierz `mat`.

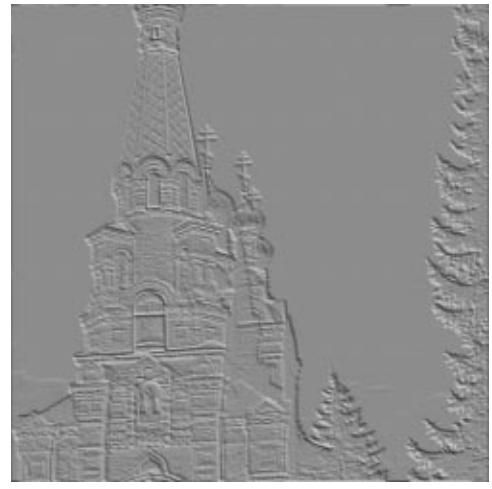
Rysunek 10.19 przedstawia wynik działania programu.

Dot3

Natężenie oświetlenia Lambertowskiego danego punktu obliczamy przez iloczyn skalarny wektora normalnego i wektora w kierunku światła. Jeżeli dana powierzchnia ma teksturę, która zamiast wartości *rgb* ma współrzędne *xyz* wektora normalnego powierzchni i znamy pozycję światła (x, y, z), to możemy obliczyć mapę oświetlenia powierzchni. Przykład *\Texture\TexDot3* pokazuje, jak to zrobić. Ładujemy teksturę zawierającą wektory normalne (rysunek 10.20a) i zwykłą teksturę (rysunek 10.20b), dla której zostały utworzone wektory normalne.



Rysunek 10.19. *Na powierzchni obiektu wyświetlana jest tekstura o wymiarach 16×16×16, ze zmiennymi współrzędnymi* u, v *i* w



Rysunek 10.20*a*. *Tekstura zawierająca wektory normalne*

Sprawdzamy, czy urządzenie obsługuje operacje `D3DTEXOPCAPS_DOTPRODUCT3`:

```
if(!(pCaps->TextureOpCaps & D3DTEXOPCAPS_DOTPRODUCT3))
    return E_FAIL;
```

W slocie 1 umieszczamy teksturę obiektu, która będzie łączona z teksturą wynikową oświetlenia ze slotu 0:

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_DOTPRODUCT3 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_TFACTOR );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP,   D3DTOP_DISABLE   );

m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 1 );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP,   m_eTexOp);
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_CURRENT );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_ALPHAOP,   D3DTOP_DISABLE);
```



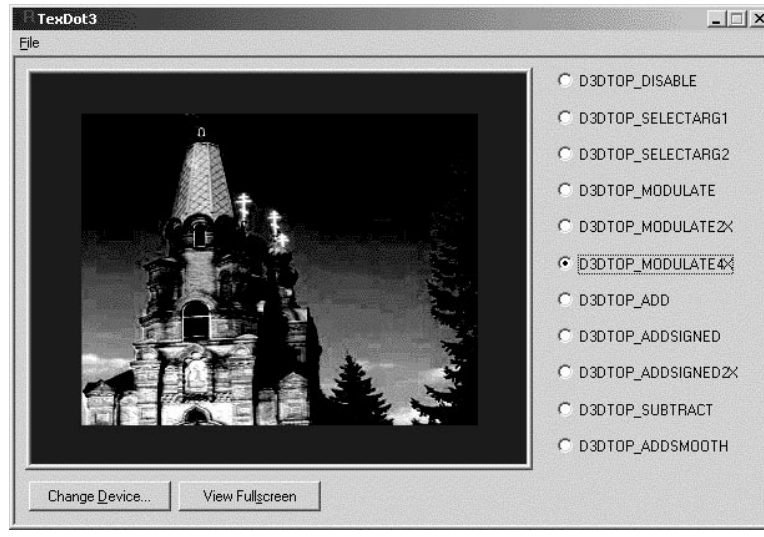
Rysunek 10.20b. *Tekstura źródłowa dla tekstury z wektorami normalnymi*

W slocie 0 została umieszczona tekstura z wektorami normalnymi oraz zastosowana operacja D3DTOP_DOTPRODUCT3. Flaga D3DTA_TFACTOR oznacza, że drugim argumentem jest wektor wskazujący kierunek świecenia światła. Zanim rozpoczniemy renderowanie, ustawiamy jeszcze ten współczynnik (flaga D3DRS_TEXTUREFACTOR):

```
DWORD r = (DWORD)( 127.0f * vLight.x + 128.0f );
DWORD g = (DWORD)( 127.0f * vLight.y + 128.0f );
DWORD b = (DWORD)( 127.0f * vLight.z + 128.0f );

DWORD dwFactor =(r<<16L) + (g<<8L) + b;
m_pd3dDevice->SetRenderState( D3DRS_TEXTUREFACTOR, dwFactor );
```

Współczynnik ten jest zmienny w czasie, co oznacza przemieszczanie się światła. Wynik działania programu przedstawia rysunek 10.21. Użytkownik może dobrać za pomocą jednego z przełączników z okna dialogowego operację łączenia mapy oświetlenia z teksturą obiektu.



Rysunek 10.21. *Wykorzystanie tekstury z wektorami normalnymi*

Mapy wektorów normalnych i wektorów wichrujących

Teraz zajmiemy się sposobem generowania tekstur z wektorami normalnymi (*NormalMap*) oraz tekstur z wektorami wichrującymi (*BumpMap*). Wykorzystanie tekstur z wektorami normalnymi zostało omówione w części dotyczącej map z oświetleniem, zaś tekstur z wektorami wichrującymi — podczas analizy zagadnień związanych z teksturami wichrującymi.

Tekstury z wektorami normalnymi i wektorami wichrującymi będą generowane na podstawie zwykłej tekstury. W pierwszym etapie należy znaleźć gradient w kierunku *X* i w kierunku *Y* w bitmapie. Teoria przetwarzania obrazów oferuje wiele masek do obliczania gradientu (rysunki 10.22a, b i c).

Rysunek 10.22a. *Maski Prewitta*

$$X\begin{bmatrix}-1 & 0 & 1\\ -1 & 0 & 1\\ -1 & 0 & 1\end{bmatrix}\quad Y\begin{bmatrix}-1 & -1 & -1\\ 0 & 0 & 0\\ 1 & 1 & 1\end{bmatrix}$$

Rysunek 10.22b. *Maski Sobela*

$$X\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix}\quad Y\begin{bmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ 1 & 2 & 1\end{bmatrix}$$

Rysunek 10.22c. *Maski gradientu*

$$X\begin{bmatrix}-1 & 1 & 1\\ -1 & -2 & 1\\ -1 & 1 & 1\end{bmatrix}\quad Y\begin{bmatrix}-1 & -1 & -1\\ 1 & -2 & 1\\ 1 & 1 & 1\end{bmatrix}$$

W projekcie *\Texture\GenDotN* wykorzystamy maskę Sobela. Stosując maskę dla kierunku *X* i *Y*, otrzymamy dwa wektory [1,0,*d*X] i [0,1,*d*Y]. Obliczamy ich iloczyn wektorowy:

$$[1,0,dX]\times[0,1,dY]=[-dX,-dY,1]$$

Otrzymany wektor jest gradientem dla danego punktu w obrazie. Jego składowe *x* i *y* wykorzystujemy jako dane dla wektora wichrującego, zaś trzy składowe jako wektor normalny. Dane te należy odpowiednio spakować do składowych *r* i *g* dla tekstury wichrującej (*signed char*) oraz do składowych *r*, *g* i *b* dla tekstury z wektorami normalnymi (*BYTE*). Fragment kodu przedstawia przygotowanie tekstury wichrującej:

```
for( DWORD y=0; y<pDesc.Height; y++ )
{
    for( DWORD x=0; x<pDesc.Width; x++ )
    {
        maskY=(y-1)%pDesc.Height;
        maskX=(x-1)%pDesc.Width;
```

```
            r=(float)(*(pSrcBits+dwSrcPitch*maskY+maskX*4+2));
            g=(float)(*(pSrcBits+dwSrcPitch*maskY+maskX*4+1));
            b=(float)(*(pSrcBits+dwSrcPitch*maskY+maskX*4));

            grey11=0.299f*r +0.587f*g+0.114f*b;

             ..............

            maskY=(y+1)%pDesc.Height;
            maskX=(x+1)%pDesc.Width;

            r=(float)(*(pSrcBits+dwSrcPitch*maskY+maskX*4+2));
            g=(float)(*(pSrcBits+dwSrcPitch*maskY+maskX*4+1));
            b=(float)(*(pSrcBits+dwSrcPitch*maskY+maskX*4));

            grey33=0.299f*r +0.587f*g+0.114f*b;

            dX=grey13+2.0f*grey23+grey33-grey11-2.0f*grey21-grey31;
            dY=grey11+2.0f*grey12+grey13-grey31-2.0f*grey32-grey33;

            D3DXVECTOR3 vec(-dX/255.0f,-dY/255.0f,1.0f);
            D3DXVec3Normalize(&vec,&vec);

            *(pDestBits+dwSrcPitch*y+x*4+2)=char(127.0f*vec.x);//r
            *(pDestBits+dwSrcPitch*y+x*4+1)=char(127.0f*vec.y);//g
            *(pDestBits+dwSrcPitch*y+x*4)=0;//b
            *(pDestBits+dwSrcPitch*y+x*4+3)=0;//a
        }
    }
```

Dla każdego punktu w obrazie, który pokrywa maska, obliczamy luminancję (zmienne `greyXX`). Zastosowano zawijanie tekstury na krawędziach. Składowe wektora są skalowane, wartość skalowania wynosi 127, następnie zapisywane do odpowiednich bajtów tekstury wichrującej.

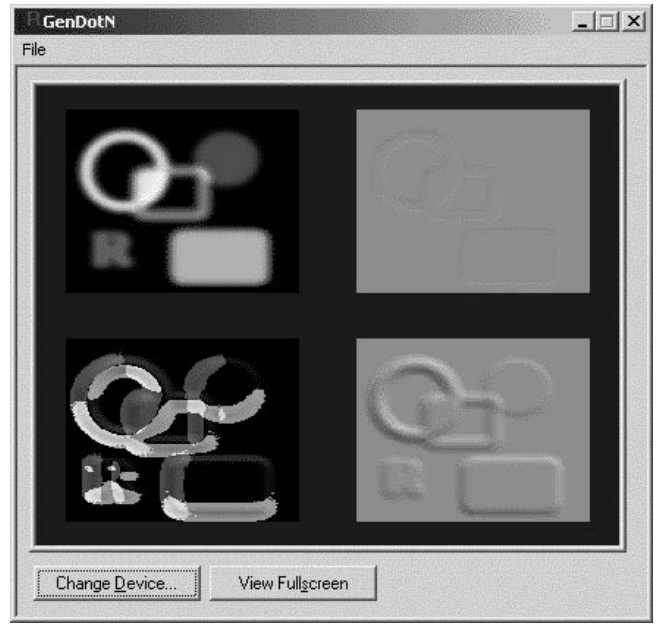
W tym kodzie widać różnicę w porównaniu do kodu wykorzystanego do obliczenia tekstury z wektorami normalnymi. Zastosowana została ta sama maska jak do obliczania gradientu:

```
*(pDestBits+dwSrcPitch*y+x*4+2)=BYTE((vec.x+1.0f) / 2.0f * 255.0f);//r
*(pDestBits+dwSrcPitch*y+x*4+1)=BYTE((vec.y+1.0f) / 2.0f * 255.0f);//g
*(pDestBits+dwSrcPitch*y+x*4)=BYTE((vec.z+1.0f) / 2.0f * 255.0f);//b
*(pDestBits+dwSrcPitch*y+x*4+3)=0;//a
```

Kody te wykorzystaliśmy w projekcie *\Texture\GenDotN*. Użytkownik może załadować tekstury z pliku za pomocą komendy *LoadTexture*. Na podstawie tej tekstury tworzone są dwie tekstury, korzystające z zaprezentowanych wyżej kodów (scena 3. i 4.) (rysunek 10.23).

Scena 2. przedstawia teksturę z wektorami normalnymi, obliczoną za pomocą funkcji Direct3D `D3DXComputeNormalMap`. Wywołanie tej funkcji jest następujące:

```
if(FAILED(D3DXComputeNormalMap(m_pTextureNormal,m_pTexture,NULL,0,
D3DX_CHANNEL_LUMINANCE, 1.0f)))
    return;
```



Rysunek 10.23. *Aplikacja MFC generuje na podstawie tekstury załadowanej z pliku tekstury z wektorami normalnymi oraz teksturę wichrującą*

Wszystkie trzy obliczone tekstury można zachować w plikach, aby wykorzystać je we własnych programach. Wybieramy jedną z komend menu: *SaveDuDvMap* (zapisuje teksturę wichrującą), *SaveNormalMap* (zapisuje teksturę z wektorami normalnymi utworzoną komendą Direct3D `D3DXComputeNormalMap`) lub *SaveNormalMapOwn* (zapisuje teksturę z wektorami normalnymi utworzoną według własnego algorytmu).