PROJEKTOWANIE GIER 3D

WPROWADZENIE DO TECHNOLOGII DirectX® 11

FRANK D. LUNA

Tytuł oryginału: Introduction to 3D Game Programming with DirectX 11

Tłumaczenie: Krzysztof Wołowski

ISBN: 978-83-246-7474-9

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/wprg3d.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/wprg3d
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

SPIS TREŚCI

CZĘŚĆ II PODSTAWY DIRECT3D

CZĘŚĆ III TEMATY

Rozdział 8

TEKSTURY

Nasze programy są coraz bardziej złożone, jednak szczegółowość prawdziwych obiektów jest na ogół większa niż ta, jaką zapewniają nam kolory na poziomie wierzchołków. **Nakładanie tekstury** (ang. *texture mapping*) to technika pozwalająca odwzorować dane graficzne na trójkąt i zwiększyć dzięki temu wydatnie szczegółowość i realizm sceny. Można na przykład zbudować sześcian i zrobić z niego skrzynkę poprzez nałożenie odpowiedniej tekstury na każdą ścianę (zob. rysunek 8.1).

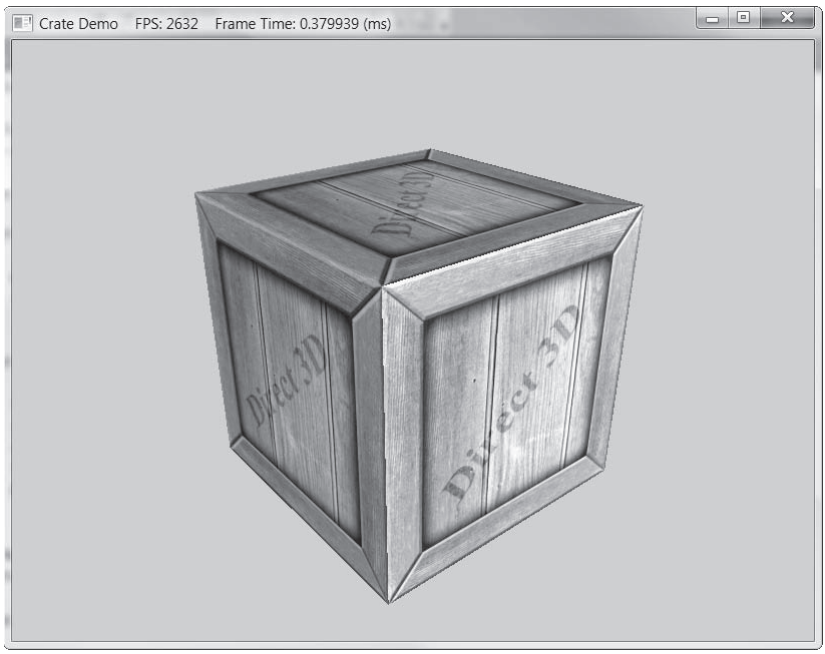
Po przeczytaniu tego rozdziału powinieneś:

1. Wiedzieć, jak określić fragment tekstury do nałożenia na trójkąt.
2. Umieć tworzyć tekstury i włączać ich obsługę.
3. Potrafić filtrować tekstury w celu uzyskania gładszego obrazu.
4. Umieć kafelkować teksturę przy użyciu trybów adresowania.
5. Znać sposoby łączenia tekstur w nowe tekstury, aby uzyskać efekty specjalne.
6. Potrafić tworzyć podstawowe efekty animowanych tekstur.

8.1. PODSUMOWANIE WIADOMOŚCI NA TEMAT TEKSTUR I ZASOBÓW

Być może pamiętasz, że począwszy od rozdziału 4., korzystaliśmy już z tekstur, a w szczególności z bufora głębokości i bufora tylnego — obiektów dwuwymiarowej tekstury reprezentowanych przez interfejs `ID3D11Texture2D`. Dla przypomnienia w tym podrozdziale uporządkujemy zdobytą wcześniej wiedzę o teksturach.

Dwuwymiarowa tekstura jest macierzą danych. Stosuje się ją do przechowywania danych dwuwymiarowego obrazu, przy czym każdy element tekstury zawiera kolor piksela. Nie jest to jednak jej jedyne zastosowanie. Przy zaawansowanej technice zwanej mapowaniem normalnych każdy element tekstury zamiast koloru przechowuje trójwymiarowy wektor. Tak więc, chociaż przyjęło się, że tekstura zawiera dane graficzne, ma ona tak naprawdę dużo szersze



Rysunek 8.1. **Sześcian z teksturą skrzynki w aplikacji „Crate"**

zastosowanie. Teksturę jednowymiarową (`ID3D11Texture1D`) można porównać do jednowymiarowej, a teksturę trójwymiarową (`ID3D11Texture3D`) do trójwymiarowej tablicy elementów. Interfejsy jedno-, dwu- i trójwymiarowych tekstur dziedziczą po `ID3D11Resource`.

W dalszej części rozdziału pokażemy, że tekstury to nie tylko tablice danych. Tekstury mogą również mieć różne poziomy mipmap, a procesor graficzny może na nich wykonywać specjalne operacje, takie jak nakładanie filtrów czy wielokrotne próbkowanie. Ponadto typ przechowywanych w teksturze danych jest ściśle określony; dopuszczalne są tylko specjalne formaty opisywane przez typ wyliczeniowy `DXGI_FORMAT`. Oto niektóre z nich:

1. `DXGI_FORMAT_R32G32B32_FLOAT`: Każdy element ma trzy 32-bitowe zmiennoprzecinkowe składowe.
2. `DXGI_FORMAT_R16G16B16A16_UNORM`: Każdy element ma cztery 16-bitowe składowe odwzorowane na zakres [0, 1].
3. `DXGI_FORMAT_R32G32_UINT`: Każdy element ma dwie 32-bitowe składowe całkowite bez znaku.
4. `DXGI_FORMAT_R16G16B16A16_UNORM`: Każdy element ma cztery 8-bitowe składowe bez znaku odwzorowane na zakres [0, 1].
5. `DXGI_FORMAT_R8G8B8A8_SNORM`: Każdy element ma cztery 8-bitowe składowe ze znakiem odwzorowane na zakres [–1, 1].
6. `DXGI_FORMAT_R8G8B8A8_SINT`: Każdy element ma cztery 8-bitowe składowe całkowite ze znakiem odwzorowane na zakres [–128, 127].
7. `DXGI_FORMAT_R8G8B8A8_UINT`: Każdy element ma cztery 8-bitowe składowe całkowite bez znaku odwzorowane na zakres [0, 255].

Litery R, G, B, A oznaczają odpowiednio czerwony, zielony, niebieski i alfa (od pierwszych liter angielskich słów *red*, *green*, *blue*, *alpha* — *przyp. tłum.*). Jak jednak wspomnieliśmy wcześniej, tekstury nie muszą zawierać informacji o kolorze; na przykład format

```
DXGI_FORMAT_R32G32B32_FLOAT
```

ma trzy składowe zmiennoprzecinkowe, może zatem przechowywać trójwymiarowy wektor o zmiennoprzecinkowych współrzędnych (nie musi to być wektor koloru). Istnieją też formaty **beztypowe**, które rezerwują pamięć, a następnie określają sposób interpretacji danych na dalszym etapie (na podobnej zasadzie działa rzutowanie typów), przy dowiązaniu tekstury do potoku. Na przykład poniższy beztypowy format rezerwuje elementy z czterema 8-bitowymi składowymi, ale nie określa typu danych (jak np. liczba całkowita, liczba zmiennoprzecinkowa czy liczba całkowita bez znaku):

```
DXGI_FORMAT_R8G8B8A8_TYPELESS
```

Teksturę można wiązać z poszczególnymi etapami potoku renderowania; typowym przykładem jest wykorzystanie tekstury jako celu renderowania (Direct3D może zapisywać do tekstury) lub jako zasobu shadera (tekstura jest próbkowana w shaderze). Zasób tekstury stworzony do jednego z tych celów będzie miał specjalne znaczniki miejsca wiązania:

```
D3D11_BIND_RENDER_TARGET | D3D11_BIND_SHADER_RESOURCE
```

wskazujące dwa etapy potoku, z którymi tekstura zostanie powiązana. Właściwie zasoby nie są wiązane bezpośrednio z etapem potoku. Zamiast tego skojarzone z nimi **widoki zasobów** są wiązane z różnymi etapami potoku. W każdym z tych przypadków użyjemy tekstury — Direct3D wymaga utworzenia widoku zasobu tej tekstury podczas uruchamiania programu. Jest to podyktowane głównie wydajnością. W dokumentacji SDK czytamy: „Pozwala to na sprawdzenie poprawności i mapowanie podczas tworzenia widoku, dzięki czemu minimalizujemy sprawdzanie typów podczas wiązania". Tak więc używając tekstury jako celu renderowania i jako shadera, musimy utworzyć dwa widoki: widok celu renderowania (`ID3D11Ren`
`↪derTargetView`) oraz widok zasobu shadera (`ID3D11ShaderResourceView`). Rola widoków zasobów jest dwojaka: informują Direct3D o sposobie wykorzystywania zasobu (z którym etapem potoku zostanie powiązany), a jeśli format zasobu został podczas jego tworzenia określony jako beztypowy, w czasie tworzenia widoku musimy określić typ. Dlatego przy beztypowych formatach elementy tekstury mogą na jednym etapie potoku być liczbami zmiennoprzecinkowymi, a na innym liczbami całkowitymi, co jest równoważne z użyciem operatora `reinterpret_cast`.

Uwaga *W dokumentacji z sierpnia 2013 r. czytamy: „Tworzenie zasobu w całości określonego typu powoduje jego ograniczenie do danego formatu. Pozwala to na optymalizację w trakcie wykonania […]". Dlatego zaleca się tworzenie beztypowych zasobów tylko wtedy, gdy są naprawdę potrzebne. W pozostałych sytuacjach należy określić typ zasobu.*

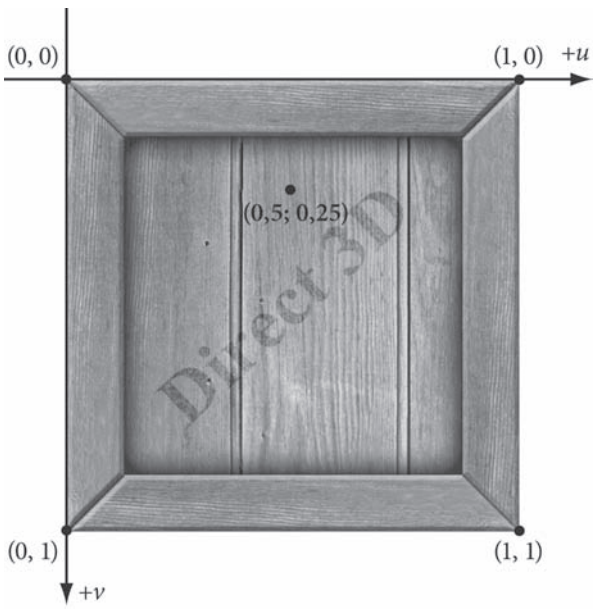
Aby utworzyć widok zasobu, podczas tworzenia zasobu musimy użyć specjalnego znacznika miejsca wiązania. Na przykład jeżeli podczas tworzenia zasobu nie włączono znacznika `D3D11_BIND_SHADER_RESOURCE` (wskazującego, że tekstura powinna zostać powiązana z potokiem jako bufor głębokości/szablonu), utworzenie widoku `ID3D11ShaderResourceView` nie jest dla tego zasobu możliwe. Jeśli spróbujesz to zrobić, otrzymasz następujący komunikat o błędzie:

```
D3D11: ERROR: ID3D11Device::CreateShaderResourceView: A ShaderResourceView cannot be
↪created of a Resource that did not specify the D3D11_BIND_SHADER_RESOURCE BindFlag.
```

W tym rozdziale zajmiemy się tylko wiązaniem tekstur jako zasobów shadera, tak aby nasze shadery pikseli mogły próbkować tekstury i kolorować na tej podstawie piksele.

8.2. WSPÓŁRZĘDNE TEKSTURY

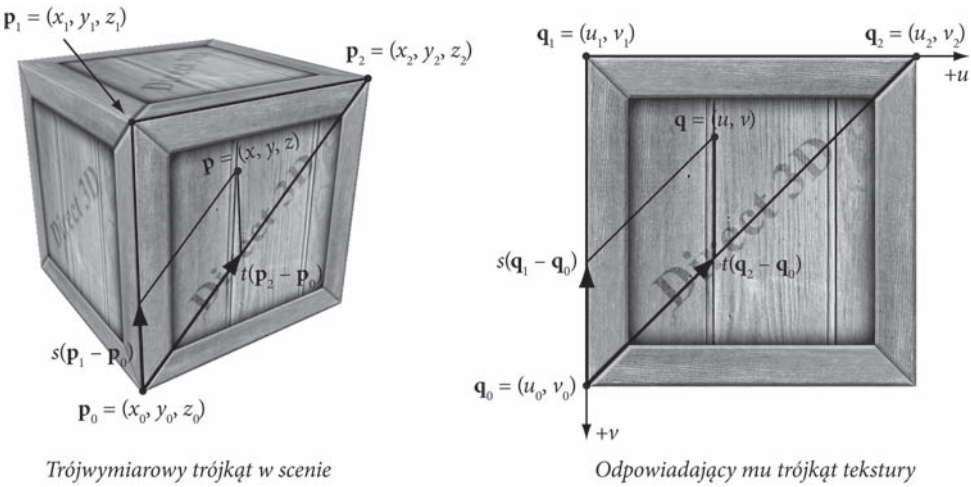
Direct3D korzysta z układu współrzędnych tekstury, w którym *u* jest poziomą, a *v* pionową osią względem obrazu. Współrzędne (*u*, *v*), takie, że $0 \le u, v \le 1$, identyfikują element tekstury zwany **tekselem**. Zwróć uwagę, że na rysunku 8.2 oś *v* jest skierowana w dół. Zauważ też, że Direct3D wykorzystuje znormalizowany przedział [0, 1], dzięki czemu otrzymuje zakres niezależny od wymiarów. Na przykład współrzędne (0,5; 0,5) zawsze reprezentują środkowy teksel bez względu na to, czy rzeczywiste wymiary tekstury to 256×256, 512×512, 1024×1024, czy 2048×2048 pikseli. Podobnie współrzędne (0,25; 0,75) określają teksel znajdujący się na jednej czwartej łącznej szerokości i trzech czwartych łącznej wysokości tekstury. Chwilowo załóżmy, że współrzędne tekstury mieszczą się zawsze w przedziale [0, 1]. Co się dzieje, gdy wyjdą poza ten zakres, wyjaśnimy w dalszej części rozdziału.



Rysunek 8.2. **Układ współrzędnych tekstury nazywany również przestrzenią tekstury**

Każdemu trójkątowi w scenie powinien odpowiadać trójkątny fragment tekstury do nałożenia (zob. rysunek 8.3). Niech $\mathbf{p}_0$, $\mathbf{p}_1$ i $\mathbf{p}_2$ będą wierzchołkami trójkąta w przestrzeni trójwymiarowej o współrzędnych tekstury odpowiednio $\mathbf{q}_0$, $\mathbf{q}_1$ i $\mathbf{q}_2$. Dla dowolnego punktu (*x*, *y*, *z*) trójkąta jego współrzędne tekstury (*u*, *v*) obliczamy, interpolując liniowo współrzędne tekstury wierzchołków na całą powierzchnię trójkąta za pomocą parametrów *s* i *t*, czyli jeśli

$$(x, y, z) = \mathbf{p} = \mathbf{p}_0 + s(\mathbf{p}_1 - \mathbf{p}_0) + t(\mathbf{p}_2 - \mathbf{p}_0)$$



Rysunek 8.3. **Po lewej stronie trójkąt w przestrzeni trójwymiarowej, po prawej wyznaczanie trójkątnego fragmentu dwuwymiarowej tekstury do nałożenia na trójkąt**

przy $s \geq 0$, $t \geq 0$, $s+t \leq 1$, to:

$$(u,v) = \mathbf{q} = \mathbf{q}_0 + s(\mathbf{q}_1 - \mathbf{q}_0) + t(\mathbf{q}_2 - \mathbf{q}_0)$$

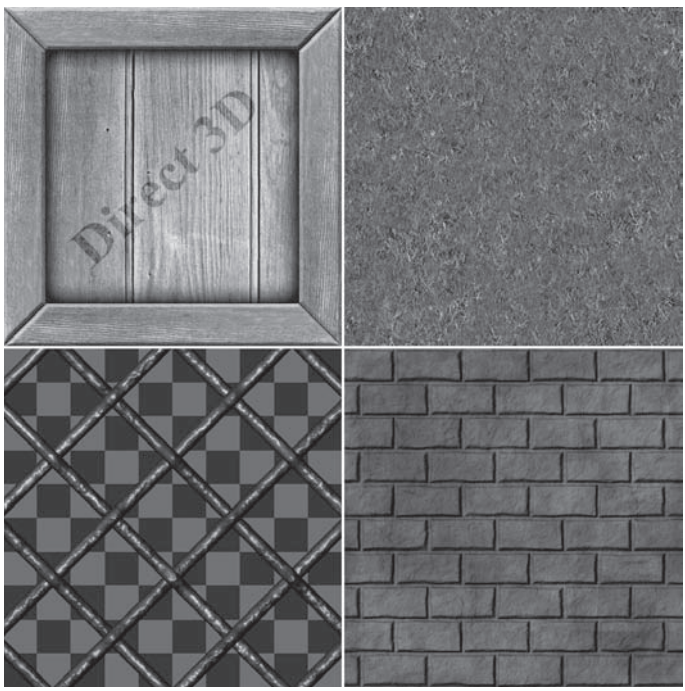
W ten sposób każdemu punktowi trójkąta odpowiada para współrzędnych tekstury.

Aby zaimplementować powyższe rozwiązanie, rozszerzymy naszą strukturę wierzchołka o parę współrzędnych reprezentujących punkt tekstury. Z każdym wierzchołkiem w trójwymiarowej przestrzeni będzie teraz skojarzony wierzchołek płaszczyzny dwuwymiarowej tekstury. Tak więc na każde trzy wierzchołki zdefiniowanego w trójwymiarowej przestrzeni trójkąta będzie przypadać trójkąt zdefiniowany w dwuwymiarowej przestrzeni tekstury.

```
// Prosta 32-bajtowa struktura wierzchołka.
struct Basic32
{
   XMFLOAT3 Pos;
   XMFLOAT3 Normal;
   XMFLOAT2 Tex;
};

const D3D11_INPUT_ELEMENT_DESC InputLayoutDesc::Basic32[3] =
{
   {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0},
   {"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0},
   {"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24, D3D11_INPUT_PER_VERTEX_DATA, 0}
};
```

Uwaga *W wyniku nakładania tekstury możemy uzyskać „niestandardowe" efekty, jeśli trójkąt w teksturze różni się bardzo od trójkąta w scenie. Trójkąt tekstury odwzorowywany na trójkąt w scenie jest wtedy rozciągany i zniekształcany, co rzadko wygląda dobrze. Przykładowo odwzorowanie trójkąta ostrokątnego na trójkąt prostokątny wymaga rozciągania. Generalnie powinniśmy unikać zniekształcania tekstury, chyba że chcemy osiągnąć efekt zniekształcenia.*

Zauważ, że na rysunku 8.3 na każdą ścianę sześcianu nałożyliśmy cały obraz tekstury. Tak jednak nie musi być zawsze. Równie dobrze możemy na geometrię nałożyć jedynie fragment tekstury. Możemy nawet umieścić w jednej teksturze kilka odrębnych obrazów (tworząc tak zwany **atlas tekstur**), a następnie wykorzystać ją przy teksturowaniu kilku obiektów (zob. rysunek 8.4). O tym, który fragment tekstury zostanie nałożony na trójkąt, decydują współrzędne tekstury.



Rysunek 8.4. **Atlas tekstur zawierający cztery fragmenty tekstur. Współrzędne tekstury każdego wierzchołka są ustawione w taki sposób, że poszczególne fragmenty są nakładane na odpowiednie płaszczyzny geometrii**

8.3. TWORZENIE I WŁĄCZANIE OBSŁUGI TEKSTURY

Najczęściej dane tekstury są odczytywane z zapisanego na dysku twardym pliku graficznego, a następnie ładowane do obiektu `ID3D11Texture2D` (w funkcji `D3DX11CreateTextureFromFile`). Zasoby tekstur nie są jednak wiązane bezpośrednio z potokiem renderowania. Zamiast tego tworzy się dla tekstury widok zasobu shadera (`ID3D11ShaderResourceView`) i wiąże widok z potokiem. Całość odbywa się w dwóch krokach:

1. Wywołanie funkcji `D3DX11CreateTextureFromFile`, aby utworzyć obiekt `ID3D11Texture2D` z pliku graficznego na dysku twardym.
2. Wywołanie metody `ID3D11Device::CreateShaderResourceView`, aby utworzyć widok zasobu shadera dla tekstury.

Obydwie te czynności można też wykonać w jednym kroku, za pomocą funkcji D3DX:

```
HRESULT D3DX11CreateShaderResourceViewFromFile(
   ID3D11Device *pDevice,
   LPCTSTR pSrcFile,
   D3DX11_IMAGE_LOAD_INFO *pLoadInfo,
   ID3DX11ThreadPump *pPump,
   ID3D11ShaderResourceView **ppShaderResourceView,
   HRESULT *pHResult
);
```

1. `pDevice`: Wskaźnik do urządzenia D3D, przy użyciu którego tekstura ma zostać utworzona.
2. `pSrcFile`: Nazwa pliku ładowanego obrazu.
3. `pLoadInfo`: Opcjonalne informacje o obrazie. Użyj wartości `null`, aby skorzystać z informacji obrazu źródłowego. Jeżeli podano wartość `null`, wymiary obrazu źródłowego zostaną użyte jako wymiary tekstury, a oprócz tego zostanie wygenerowana pełna sekwencja mipmap (zob. punkt 8.4.2). Ponieważ to nam z reguły wystarcza, wartość `null` stanowi dobry domyślny wybór.
4. `pPump`: Tworzy nowy wątek w celu załadowania zasobu. Aby załadować zasób do wątku głównego, użyj wartości `null`. W tej książce będziemy korzystać wyłącznie z wartości `null`.
5. `ppShaderResourceView`: Zwraca wskaźnik do tworzonego widoku zasobu shadera dla ładowanej z pliku tekstury.
6. `pHResult`: Użyj wartości `null`, jeżeli wartość `pPump` wynosi `null`.

Funkcja pozwala załadować następujące formaty obrazów: BMP, JPG, PNG, DDS, TIFF, GIF i WMP (zob. `D3DX11_IMAGE_FILE_FORMAT`).

Do tekstury i odpowiadającego jej widoku zasobu shadera będziemy się odnosić zamiennie. Możemy na przykład powiedzieć, że wiążemy teksturę z potokiem, podczas gdy tak naprawdę wiążemy jej widok.

W poniższym fragmencie kodu tworzymy teksturę z obrazu o nazwie *WoodCreate01.dds*:

```
ID3D11ShaderResourceView* mDiffuseMapSRV;
HR(D3DX11CreateShaderResourceViewFromFile(md3dDevice,
  L"WoodCrate01.dds", 0, 0, &mDiffuseMapSRV, 0 ));
```

Po załadowaniu tekstury musimy ją jeszcze zapisać w zmiennej efektu, co pozwoli na jej użycie w shaderze pikseli. Obiekt dwuwymiarowej tekstury w pliku *.fx* reprezentuje typ `Texture2D`. Przykładowa deklaracja zmiennej tekstury w pliku efektu mogłaby wyglądać tak:

```
// Danych nienumerycznych nie można zapisywać w obiekcie cbuffer.
Texture2D gDiffuseMap;
```

Zgodnie z komentarzem obiekty tekstur są umieszczane poza buforami stałych. Wskaźnik do zdefiniowanego w efekcie obiektu `Texture2D` (który jest zmienną zasobu shadera) z poziomu aplikacji C++ uzyskamy w następujący sposób:

```
ID3DX11EffectShaderResourceVariable* DiffuseMap;
fxDiffuseMap = mFX->GetVariableByName("gDiffuseMap")->AsShaderResource();
```

Gdy mamy wskaźnik do obiektu `Texture2D` w efekcie, możemy go uaktualnić poprzez interfejs C++:

```
// Ustaw widok zasobu tekstury w zmiennej tekstury w efekcie.
fxDiffuseMap->SetResource(mDiffuseMapSRV);
```

Tak jak w przypadku każdej zmiennej efektu, jeśli chcemy jej nadać inną wartość, musimy wywołać metodę `Apply`:

```
// Ustaw teksturę skrzynki.
fxDiffuseMap->SetResource(mCrateMapSRV);
pass->Apply(0, md3dImmediateContext);
DrawCrate();

// Ustaw teksturę trawy.
fxDiffuseMap->SetResource(mGrassMapSRV);
pass->Apply(0, md3dImmediateContext);
DrawGrass();

// Ustaw teksturę muru.
fxDiffuseMap->SetResource(mBrickMapSRV);
pass->Apply(0, md3dImmediateContext);
DrawBricks();
```

Atlasy tekstur mogą zwiększyć wydajność aplikacji, gdyż dzięki nim aplikacja jest w stanie narysować więcej geometrii w pojedynczym poleceniu rysowania. Załóżmy na przykład, że chcemy użyć atlasu tekstur z rysunku 8.3, zawierającego tekstury skrzynki, trawy i muru. Ustawiając odpowiednie współrzędne tekstury dla poszczególnych obiektów, możemy narysować geometrię w jednym poleceniu rysowania (zakładamy, że nie ma potrzeby zmiany innych parametrów obiektów):

```
// Ustaw atlas tekstur.
fxDiffuseMap->SetResource(mAtlasSRV);
pass->Apply(0, md3dImmediateContext);
DrawCrateGrassAndBricks();
```

Każde kolejne polecenie rysowania powoduje dodatkowe opóźnienie, warto zatem ograniczyć ich liczbę do minimum.

Uwaga *W zasadzie zasób tekstury może zostać wykorzystany w dowolnym shaderze (wierzchołków, geometrii czy pikseli). My będziemy na razie używać tekstur tylko w shaderach pikseli. Wspominaliśmy już wcześniej, że tak naprawdę tekstury to swego rodzaju tablice, ich przydatność w programach cieniujących wierzchołki i geometrię nie powinna zatem dziwić.*

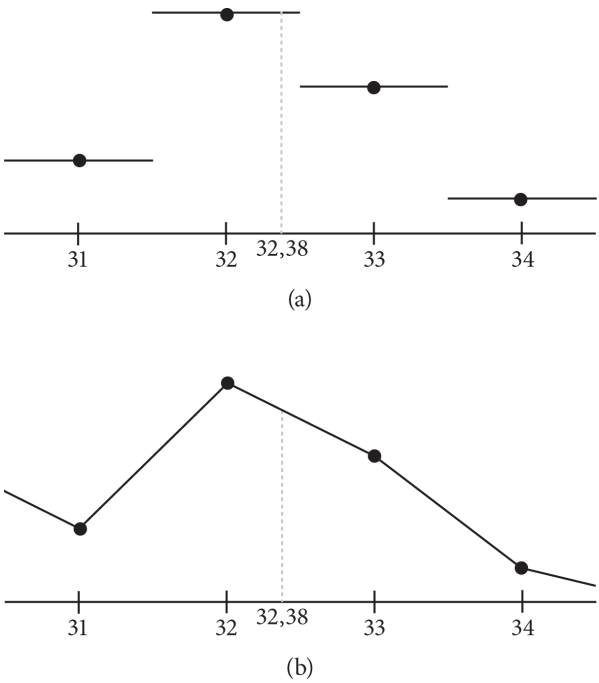
8.4. FILTRY

8.4.1. Powiększanie

Elementy tekstury powinniśmy traktować jak dyskretne próbki koloru będące częścią ciągłego obrazu — nie jak prostokąty, które mają pole. Nasuwa się zatem pytanie: Co, jeśli współrzędne tekstury (u, v) nie pokrywają się z żadnym tekselem? Taka sytuacja jest jak najbardziej możliwa. Przypuśćmy, że gracz zbliża się do muru na tyle, że mur pokrywa cały ekran w scenie. Na potrzeby przykładu załóżmy też, że rozdzielczość monitora to 1024×1024, a rozdzielczość tekstury: 256×256. Tekstura ulega zatem **powiększeniu** (ang. *magnification*), ponieważ jest więcej pikseli do pokrycia niż tekseli. W naszym przykładzie na każdy punkt teksela

przypadają cztery piksele. Dzięki interpolacji współrzędnych tekstury wierzchołków na powierzchni trójkąta każdy piksel otrzymuje parę unikalnych współrzędnych tekstury. Współrzędne tekstury niektórych pikseli nie będą się więc pokrywać z żadnym tekselem. Znając kolory tekseli, możemy przy użyciu interpolacji wyznaczyć kolory pomiędzy nimi. Karty graficzne obsługują dwie metody interpolacji: interpolację stałą i interpolację liniową. W praktyce prawie zawsze korzystamy z tej drugiej.

Rysunek 8.5 ilustruje wymienione metody w przestrzeni jednowymiarowej. Załóżmy, że mamy jednowymiarową teksturę z 256 próbkami i interpolowaną współrzędną tekstury $u = 0{,}126484375$. Ta znormalizowana współrzędna tekstury odnosi się do teksela na pozycji: $0{,}126484375 \cdot 256 = 32{,}38$. Wartość ta leży oczywiście pomiędzy dwoma tekselami, potrzebna jest zatem interpolacja.

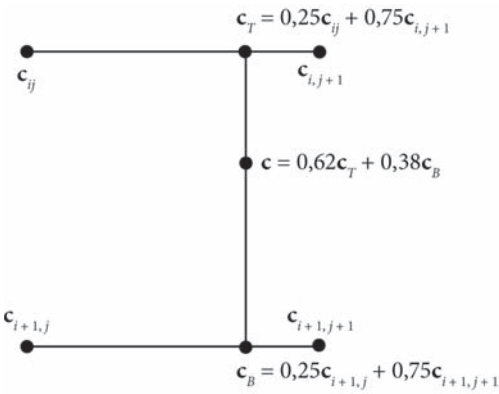


Rysunek 8.5. **(a) Na podstawie punktów tekseli możemy zbudować funkcję schodkową, aby przybliżyć wartości pomiędzy tekselami. Ten sposób nazywamy próbkowaniem metodą najbliższego sąsiada, ponieważ wykorzystuje najbliższy teksel. (b) Na podstawie punktów tekseli możemy zbudować funkcję odcinkowo-liniową, aby przybliżyć wartości pomiędzy tekselami**

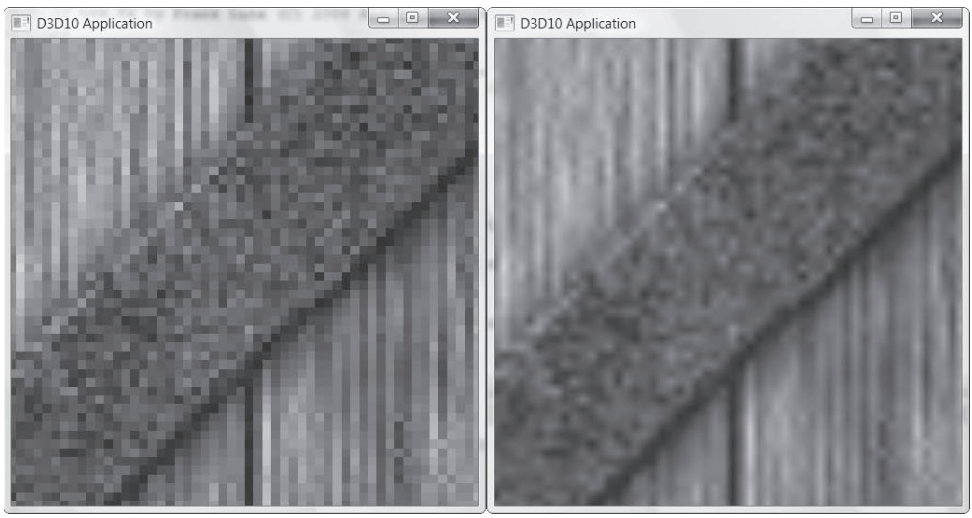
Interpolację na płaszczyźnie dwuwymiarowej nazywamy **interpolacją dwuliniową** (ang. *bilinear*). Pokazano ją na rysunku 8.6. Jeżeli punkt wyznaczany przez współrzędne tekstury znajduje się pomiędzy czterema tekselami, dokonujemy ich liniowej interpolacji w kierunku u, a następnie w kierunku v.

Na rysunku 8.7 pokazano różnicę między interpolacją stałą a interpolacją liniową. Jak widać, ta pierwsza tworzy charakterystyczne bloki. Interpolacja liniowa pozwala uzyskać bardziej wygładzony obraz, ale i tak nie wygląda on tak dobrze jak w przypadku rzeczywistych danych (np. tekstury o wyższej rozdzielczości).

Trzeba podkreślić, że w interaktywnych programach 3D, które dają obserwatorowi pełną swobodę poruszania się, tak naprawdę nie da się rozwiązać do końca problemu powiększania.



Rysunek 8.6. **Mamy tu cztery punkty tekseli: c_{ij}, $c_{i,j+1}$, $c_{i+1,j}$ oraz $c_{i+1,j+1}$. Chcemy przybliżyć kolor w punkcie c pomiędzy czterema punktami tekseli za pomocą interpolacji. W tym przypadku c leży o 0,75 jednostki na prawo od c_{ij} i o 0,38 jednostki poniżej c_{ij}. Najpierw dokonujemy jednowymiarowej interpolacji liniowej między dwoma górnymi kolorami, w wyniku czego otrzymujemy c_T. Podobnie postępujemy w przypadku dwóch dolnych kolorów, otrzymując c_B. Ostateczny wynik, c, uzyskujemy na drodze interpolacji między c_T i c_B**



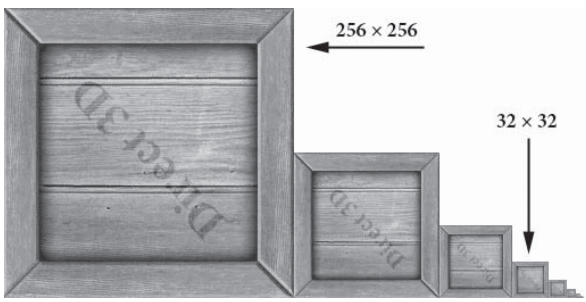
Rysunek 8.7. **Na skutek zbliżenia do sześcianu z teksturą skrzynki następuje powiększenie. Po lewej stronie pokazano wynik interpolacji stałej. Charakterystyczne bloki, z jakich składa się obraz, tłumaczy nieciągłość funkcji (zob. rysunek 8.5a), która powoduje, że zmiany są gwałtowne, a nie łagodne. Po prawej stronie pokazano wynik filtrowania liniowego. Uzyskany w ten sposób obraz jest bardziej wygładzony, co zawdzięczamy ciągłości funkcji interpolującej**

Tekstury widziane z pewnych odległości będą wyglądać wspaniale, ale im bliżej będą oka, tym efekt wizualny może być gorszy. Niektóre gry, aby uniknąć zbyt dużego powiększenia, ograniczają odległość, na jaką wirtualne oko może się zbliżyć do powierzchni. Użycie tekstur o wyższej rozdzielczości może częściowo rozwiązać ten problem.

Uwaga *W kontekście teksturowania stosowanie interpolacji stałej, aby znaleźć wartości tekstury dla współrzędnych tekstury pomiędzy tekselami, nazywamy* ***filtrowaniem punktowym****. Z kolei stosowanie w tym celu interpolacji liniowej nazywamy* ***filtrowaniem liniowym****. Taka terminologia jest stosowana również w Direct3D.*

8.4.2. Pomniejszanie

Pomniejszanie obrazu (ang. *minification*) to działanie odwrotne do powiększania. Podczas pomniejszania liczba odwzorowywanych tekseli jest większa od liczby pikseli. Rozważmy na przykład sytuację, w której mamy mur pokryty teksturą o wymiarach 256×256. Kiedy patrzące na niego oko oddala się, mur staje się coraz mniejszy, do momentu gdy pokrywa tylko obszar o rozmiarze 64×64 pikseli. Obszar 256×256 tekseli musi więc zostać odwzorowany na obszar 64×64 na ekranie. W takiej sytuacji współrzędne tekstury dla pikseli też raczej nie będą się pokrywać z tekselami w teksturze. Interpolacja stała i interpolacja liniowa znajdują więc zastosowanie również w przypadku pomniejszania. Tym razem jednak można zrobić więcej. Wiemy, że częstotliwość próbkowania powinna być zmniejszona z 256×256 tekseli do 64×64 pikseli. Technika mipmappingu oferuje wydajne rozwiązanie tego problemu kosztem trochę większego zużycia pamięci. Podczas inicjalizacji poprzez redukcję próbkowania są tworzone mniejsze wersje tekstury (mipmapy). W ten sposób powstaje sekwencja mipmap (zob. rysunek 8.8). Uśrednianie odbywa się więc z góry, dla poszczególnych rozmiarów mipmap. W czasie działania aplikacji karta graficzna robi dwie rzeczy, w zależności od wybranych przez programistę opcji:



Rysunek 8.8. **Sekwencja mipmap; każda kolejna mipmapa jest dwa razy mniejsza (w obu wymiarach) od poprzedniej, aż do rozmiaru 1×1**

1. Wybiera poziom i używa mipmapy o rozmiarze najbardziej zbliżonym do ekranowej rozdzielczości teksturowanej geometrii, stosując w razie potrzeby interpolację stałą lub interpolację liniową. Sposób ten nazywamy **filtrowaniem punktowym** mipmap, ponieważ przypomina on interpolację stałą — wybieramy po prostu najbliższą mipmapę i wykorzystujemy ją jako teksturę.
2. Wybiera dwie mipmapy o rozmiarze najbardziej zbliżonym do ekranowej rozdzielczości teksturowanej geometrii (jedną większą, a drugą mniejszą niż ekranowa rozdzielczość geometrii). Następnie do tych dwóch poziomów mipmap stosuje filtrowanie stałe lub filtrowanie liniowe, wyznaczając kolor dla każdego z nich. Na koniec dokonuje interpolacji pomiędzy otrzymanymi kolorami. Sposób ten nazywamy **filtrowaniem liniowym** mipmap, ponieważ przypomina on interpolację liniową — dokonujemy interpolacji pomiędzy dwiema najbliższymi mipmapami.

Wybór najbardziej odpowiedniego poziomu szczegółów z sekwencji mipmap pozwala w dużym stopniu ograniczyć obliczenia związane z pomniejszaniem.

8.4.2.1. Tworzenie mipmap

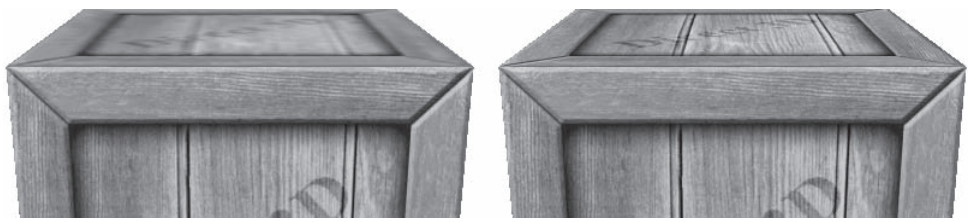
Mipmapy mogą być tworzone bezpośrednio przez grafików lub automatycznie, za pomocą algorytmów filtrujących.

W niektórych formatach plików graficznych, takich jak DDS (ang. *Direct Draw Surface*), poziomy mipmap mogą być przechowywane bezpośrednio w danych pliku. W takim przypadku wystarczy je tylko odczytać — nie ma potrzeby obliczania poziomów mipmap algorytmicznie. Narzędzie DirectX Texture Tool pomaga wygenerować sekwencję mipmap i wyeksportować ją do pliku .DDS. Jeżeli plik graficzny nie zawiera pełnej sekwencji mipmap, zostanie ona stworzona w metodzie `D3DX11CreateShaderResourceViewFromFile` lub `D3DX11 ↪CreateTextureFromFile` przy użyciu określonego algorytmu filtrującego (zapoznaj się ze strukturą `D3DX11_IMAGE_LOAD_INFO` w dokumentacji SDK, a w szczególności z jej składową `MipFilter`). Widzimy zatem, że mipmapping jest w zasadzie automatyczny. Funkcje `D3DX11` wygenerują sekwencję mipmap automatycznie, jeżeli plik źródłowy takiej nie zawiera. I dopóki mipmapping jest włączony, karta graficzna będzie wybierać odpowiednią mipmapę w czasie działania programu.

Uwaga *Zdarza się, że podczas redukcji próbkowania algorytmy filtrujące nie zachowują pewnych istotnych szczegółów. Na przykład na rysunku 8.8 tekst „Direct 3D" na skrzynce znika na niższych poziomach mipmap. Jeżeli jest to niedopuszczalne, twórca może zawsze samodzielnie stworzyć nowe oraz poprawić istniejące poziomy.*

8.4.3. Filtrowanie anizotropowe

Kolejny rodzaj filtrowania to **filtrowanie anizotropowe**. Filtr anizotropowy pomaga niwelować zniekształcenia pojawiające się, kiedy kąt między wektorem normalnym wielokąta a wektorem widzenia kamery jest szeroki (kiedy wielokąt jest ortogonalny do okna widoku). Użycie tego filtra wiąże się z największym kosztem obliczeniowym, ale często warto taki koszt ponieść, żeby usunąć zniekształcenia i artefakty. Rysunek 8.9 przedstawia zrzuty ekranu porównujące filtrowanie anizotropowe z filtrowaniem liniowym.



Rysunek 8.9. **Górna ściana skrzynki jest prawie ortogonalna do okna widoku. Po lewej: W wyniku zastosowania filtrowania liniowego góra skrzynki jest mocno rozmyta. Po prawej: Filtrowanie anizotropowe sprawdza się lepiej przy renderowaniu górnej ściany skrzynki pod tym kątem**

8.5. PRÓBKOWANIE TEKSTUR

Wiemy już, że obiekt `Texture2D` reprezentuje teksturę w pliku efektu. Jest jednak jeszcze jeden obiekt związany z teksturą — obiekt stanu samplera `SamplerState`. W obiekcie stanu samplera definiujemy używane przy teksturze filtry. Oto kilka przykładów:

```
// Użyj filtrowania liniowego przy pomniejszaniu, powiększaniu i mipmappingu.
SamplerState mySampler0
{
   Filter = MIN_MAG_MIP_LINEAR;
};
```

```
// Użyj filtrowania liniowego przy pomniejszaniu, filtrowania punktowego przy powiększaniu i mipmappingu.
SamplerState mySampler1
{
   Filter = MIN_LINEAR_MAG_MIP_POINT;
};

// Użyj filtrowania punktowego przy pomniejszaniu, filtrowania liniowego przy powiększaniu
// i filtrowania punktowego przy mipmappingu.
SamplerState mySampler2
{
   Filter = MIN_POINT_MAG_LINEAR_MIP_POINT;
};

// Użyj interpolacji anizotropowej przy pomniejszaniu, powiększaniu i mipmappingu.
SamplerState mySampler3
{
   Filter = ANISOTROPIC;
   MaxAnisotropy = 4;
};
```

Podczas filtrowania anizotropowego musimy określić wartość parametru z zakresu od 1 do 16. Przy większych wartościach filtrowanie jest bardziej czasochłonne, ale daje lepsze wyniki. Pozostałych permutacji można się domyślić na podstawie tych podanych; wszystkie zostały opisane w dokumentacji SDK w temacie o typie wyliczeniowym `D3D11_FILTER`. W dalszej części rozdziału poznasz pewne inne właściwości samplera, Twój aktualny stan wiedzy o nim wystarczy jednak do zrozumienia pierwszej aplikacji demonstracyjnej.

Znając współrzędne tekstury dla piksela w shaderze pikseli, próbkujemy teksturę w następujący sposób:

```
// Danych nienumerycznych nie można zapisywać w obiekcie cbuffer.
Texture2D gDiffuseMap;

SamplerState samAnisotropic
{
   Filter = ANISOTROPIC;
   MaxAnisotropy = 4;
};

struct VertexOut
{
   float4 PosH : SV_POSITION;
   float3 PosW : POSITION;
   float3 NormalW : NORMAL;
   float2 Tex : TEXCOORD;
};

float4 PS(VertexOut pin, uniform int gLightCount) : SV_Target
{
   float4 texColor = gDiffuseMap.Sample(samAnisotropic, pin.Tex);
   ...
```

Jak widać, do próbkowania tekstury używamy metody `Texture2D::Sample`. W pierwszym argumencie przekazujemy obiekt `SamplerState`, a w drugim współrzędne (u, v) tekstury dla piksela. Metoda zwraca uzyskany drogą interpolacji kolor tekstury w danym punkcie (u, v), korzystając z metod filtrowania określonych w obiekcie `SamplerState`.

Uwaga *Typ HLSL `SamplerState` odzwierciedla interfejs `ID3D11SamplerState`. Stany samplera można też ustawiać na poziomie aplikacji za pomocą metody `ID3DX11Ef`* ↪*`fectSamplerVariable::SetSampler`. Wykorzystuje się przy tym również strukturę `D3D11_SAMPLER_DESC` i metodę `ID3D11Device::CreateSamplerState`. Podobnie jak stany renderowania, stany samplera powinny być tworzone podczas uruchamiania aplikacji.*

8.6. TEKSTURY I MATERIAŁY

Aby zintegrować tekstury z naszym systemem materiałów i oświetlenia, najczęściej mnożymy kolor tekstury przez sumę składników oświetlenia otoczenia i oświetlenia rozproszonego. Składnik światła odbitego jest dodawany na końcu (operację tę nazywamy modulacją z opóźnionym dodaniem).

```
// Modulacja z opóźnionym dodaniem.
litColor = texColor*(ambient + diffuse) + spec;
```

Modyfikacja ta daje nam wartości materiałów otoczenia i rozpraszającego na poziomie piksela, dzięki czemu uzyskujemy wyższą rozdzielczość niż w przypadku materiałów na poziomie obiektu (ponieważ przeważnie trójkąt pokrywa wiele tekseli). Tak więc każdy piksel otrzymuje drogą interpolacji współrzędne tekstury (u, v). Współrzędne te są następnie wykorzystywane do próbkowania tekstury, aby uzyskać kolor, który wpływa na opis materiału dla tego piksela.

8.7. APLIKACJA „CRATE”

Przyjrzymy się teraz najważniejszym etapom nakładania tekstury skrzynki na sześcian (zob. rysunek 8.1).

8.7.1. Określanie współrzędnych tekstury

Metoda `GeometryGenerator::CreateBox` generuje współrzędne tekstury dla prostopadłościanu, tak aby cały obraz tekstury został odwzorowany na każdą jego ścianę. Dla uproszczenia pokazano tylko definicję wierzchołków dla przedniej, tylnej i górnej ściany. Zauważ też, że współrzędne wektorów normalnych i stycznych zostały w konstruktorach `Vertex` pominięte.

```
void GeometryGenerator::CreateBox(float width, float height, float depth,
  MeshData& meshData)
{
   Vertex v[24];

   float w2 = 0.5f*width;
   float h2 = 0.5f*height;
   float d2 = 0.5f*depth;

   // Podaj dane wierzchołków ściany przedniej.
   v[0] = Vertex(-w2, -h2, -d2, ..., 0.0f, 1.0f);
   v[1] = Vertex(-w2, +h2, -d2, ..., 0.0f, 0.0f);
```

```
v[2] = Vertex(+w2, +h2, -d2, ..., 1.0f, 0.0f);
v[3] = Vertex(+w2, -h2, -d2, ..., 1.0f, 1.0f);

// Podaj dane wierzchołków ściany tylnej.
v[4] = Vertex(-w2, -h2, +d2, ..., 1.0f, 1.0f);
v[5] = Vertex(+w2, -h2, +d2, ..., 0.0f, 1.0f);
v[6] = Vertex(+w2, +h2, +d2, ..., 0.0f, 0.0f);
v[7] = Vertex(-w2, +h2, +d2, ..., 1.0f, 0.0f);

// Podaj dane wierzchołków ściany górnej.
v[8] = Vertex(-w2, +h2, -d2, ..., 0.0f, 1.0f);
v[9] = Vertex(-w2, +h2, +d2, ..., 0.0f, 0.0f);
v[10] = Vertex(+w2, +h2, +d2, ..., 1.0f, 0.0f);
v[11] = Vertex(+w2, +h2, -d2, ..., 1.0f, 1.0f);
```

W razie wątpliwości co do sposobu określenia współrzędnych przeanalizuj ponownie rysunek 8.3.

8.7.2. Tworzenie tekstury

Teksturę tworzymy z pliku (a właściwie z widoku zasobu shadera dla tekstury) podczas uruchamiania aplikacji w następujący sposób:

```
// Składowe CrateApp.
ID3D11ShaderResourceView* mDiffuseMapSRV;

bool CrateApp::Init()
{
   if(!D3DApp::Init())
      return false;
   // Należy najpierw zainicjalizować Effects, ponieważ InputLayouts zależy od sygnatur shadera.
   Effects::InitAll(md3dDevice);
   InputLayouts::InitAll(md3dDevice);

   HR(D3DX11CreateShaderResourceViewFromFile(md3dDevice,
     L"Textures/WoodCrate01.dds", 0, 0, &mDiffuseMapSRV, 0));
   BuildGeometryBuffers();

   return true;
}
```

8.7.3. Ustawianie tekstury

Najczęściej dane tekstury są odczytywane w shaderze pikseli. Aby shader pikseli miał do nich dostęp, musimy ustawić widok tekstury (`ID3D11ShaderResourceView`) w obiekcie typu `Texture2D` z pliku .fx. Możemy to zrobić następująco:

```
// Składowa BasicEffect.
ID3DX11EffectShaderResourceVariable* DiffuseMap;

// Pobierz wskaźniki do zmiennych pliku efektu.
DiffuseMap = mFX->GetVariableByName("gDiffuseMap")->AsShaderResource();

void BasicEffect::SetDiffuseMap(ID3D11ShaderResourceView* tex)
{
DiffuseMap->SetResource(tex);
}
```

```
// [kod .FX]
// Zmienna tekstury z pliku efektu.
Texture2D gDiffuseMap;
```

8.7.4. Aktualizacja efektu bazowego

Poniżej pokazano aktualną zawartość pliku *Basic.fx* wzbogaconą o obsługę tekstur:

```
//=============================================================================
// Basic.fx Frank Luna (C) 2011 Wszelkie prawa zastrzeżone.
//
// Efekt bazowy. Aktualnie obsługuje przekształcenia, oświetlenie
// oraz tekstury.
//=============================================================================

#include "LightHelper.fx"

cbuffer cbPerFrame
{
   DirectionalLight gDirLights[3];
   float3 gEyePosW;

   float gFogStart;
   float gFogRange;
   float4 gFogColor;
};

cbuffer cbPerObject
{
   float4x4 gWorld;
   float4x4 gWorldInvTranspose;
   float4x4 gWorldViewProj;
   float4x4 gTexTransform;
   Material gMaterial;
};

// Danych nienumerycznych nie można zapisywać w obiekcie cbuffer.
Texture2D gDiffuseMap;

SamplerState samAnisotropic
{
   Filter = ANISOTROPIC;
   MaxAnisotropy = 4;

   AddressU = WRAP;
   AddressV = WRAP;
};

struct VertexIn
{
   float3 PosL : POSITION;
   float3 NormalL : NORMAL;
   float2 Tex : TEXCOORD;
};

struct VertexOut
{
```

```
    float4 PosH : SV_POSITION;
    float3 PosW : POSITION;
    float3 NormalW : NORMAL;
    float2 Tex : TEXCOORD;
};

VertexOut VS(VertexIn vin)
{
    VertexOut vout;

    // Przekształcenie do przestrzeni świata.
    vout.PosW = mul(float4(vin.PosL, 1.0f), gWorld).xyz;
    vout.NormalW = mul(vin.NormalL, (float3x3)gWorldInvTranspose);

    // Przekształcenie do jednorodnej przestrzeni obcinania.
    vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj);

    // Zwróć atrybuty wierzchołków do interpolacji na powierzchni trójkąta.
    vout.Tex = mul(float4(vin.Tex, 0.0f, 1.0f), gTexTransform).xy;

    return vout;
}

float4 PS(VertexOut pin, uniform int gLightCount, uniform bool gUseTexure) : SV_Target
{
    // Normalizuj ponownie normalną (mogła ulec denormalizacji w procesie interpolacji).
    pin.NormalW = normalize(pin.NormalW);

    // Wektor toEye jest używany przy oświetlaniu.
    float3 toEye = gEyePosW - pin.PosW;

    // Przechowaj odległość tego punktu powierzchni od oka.
    float distToEye = length(toEye);

    // Normalizuj.
    toEye /= distToEye;

    // Wartość domyślna -- element neutralny mnożenia.
    float4 texColor = float4(1, 1, 1, 1);
    if(gUseTexure)
    {
        // Próbkuj teksturę.
        texColor = gDiffuseMap.Sample(samAnisotropic, pin.Tex);
    }

    //
    // Oświetlenie.
    //

    float4 litColor = texColor;
    if(gLightCount > 0)
    {
        // Rozpocznij sumowanie od zera.
        float4 ambient = float4(0.0f, 0.0f, 0.0f, 0.0f);
        float4 diffuse = float4(0.0f, 0.0f, 0.0f, 0.0f);
        float4 spec = float4(0.0f, 0.0f, 0.0f, 0.0f);
```

```
        // Sumuj udział światła z każdego źródła.
         [unroll]
        for(int i = 0; i < gLightCount; ++i)
        {
           float4 A, D, S;
           ComputeDirectionalLight(gMaterial, gDirLights[i],
             pin.NormalW, toEye,
             A, D, S);

        ambient += A;
        diffuse += D;
        spec += S;
        }

        // Modulacja z opóźnionym dodaniem.
        litColor = texColor*(ambient + diffuse) + spec;
    }

    // Pobierz wartość alfa z koloru materiału rozpraszającego i tekstury.
    litColor.a = gMaterial.Diffuse.a * texColor.a;

    return litColor;
}

technique11 Light1
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, PS(1, false)));
    }
}

technique11 Light2
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, PS(2, false)));
    }
}

technique11 Light3
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, PS(3, false)));
    }
}

technique11 Light0Tex
{
    pass P0
    {
```

```
        SetVertexShader(CompileShader(vs_5_0, VS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, PS(0, true)));
    }
}

technique11 Light1Tex
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, PS(1, true)));
    }
}

technique11 Light2Tex
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, PS(2, true)));
    }
}

technique11 Light3Tex
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, PS(3, true)));
    }
}
```

Warto zauważyć, że za pomocą argumentu typu uniform `gUseTexture` możemy decydować, które techniki w pliku *Basic.fx* obejmą teksturowanie, a które nie. Dzięki temu, jeśli potrzebujemy wyrenderować coś, co nie wymaga tekstury, mamy do dyspozycji odpowiednią technikę i nie musimy ponosić kosztów teksturowania. Na tej samej zasadzie wybieramy technikę z liczbą używanych świateł, aby uniknąć niepotrzebnych i czasochłonnych obliczeń oświetlenia.

Dotychczas nie omówiliśmy jeszcze zmiennej bufora stałego `gTexTransform`. Jest ona wykorzystywana w shaderze wierzchołków do przekształcania współrzędnych tekstury:

```
vout.Tex = mul(float4(vin.Tex, 0.0f, 1.0f), gTexTransform).xy;
```

Współrzędne tekstury są dwuwymiarowymi punktami na płaszczyźnie. Możemy je zatem przesuwać, obracać i skalować tak jak wszystkie inne punkty. W tej aplikacji do przekształcenia używamy macierzy jednostkowej, przez co współrzędne tekstury pozostają niezmienione. Jak jednak zobaczymy w podrozdziale 8.9, przekształcenie współrzędnych tekstury pozwala osiągnąć pewne ciekawe efekty. Zwróć uwagę, że aby przekształcić dwuwymiarowe współrzędne tekstury o macierz 4×4, rozszerzamy ją do postaci czterowymiarowego wektora.

```
vin.Tex ---> float4(vin.Tex, 0.0f, 1.0f)
```

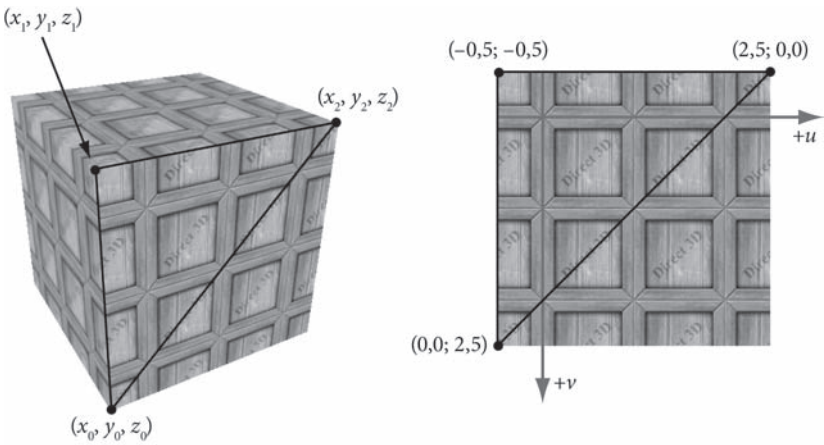
Po wykonaniu mnożenia otrzymany czterowymiarowy wektor jest redukowany do wektora dwuwymiarowego poprzez odrzucenie składowych z i w:

```
vout.Tex = mul(float4(vin.Tex, 0.0f, 1.0f), gTexTransform).xy;
```

8.8. TRYBY ADRESOWANIA

Tekstura wraz z interpolacją stałą i liniową definiują funkcję $T(u, v) = (r, g, b, a)$, której wartości są wektorami. A więc dla współrzędnych tekstury $(u, v) \in [0, 1]^2$ funkcja tekstury T zwraca kolor (r, g, b, a). Direct3D umożliwia nam rozszerzenie dziedziny tej funkcji na cztery różne sposoby (zwane **trybami adresowania**): *wrap*, *border color*, *clamp* oraz *mirror*.

1. *wrap* rozszerza funkcję tekstury, powtarzając obraz w każdym punkcie o współrzędnych całkowitych (zob. rysunek 8.10).

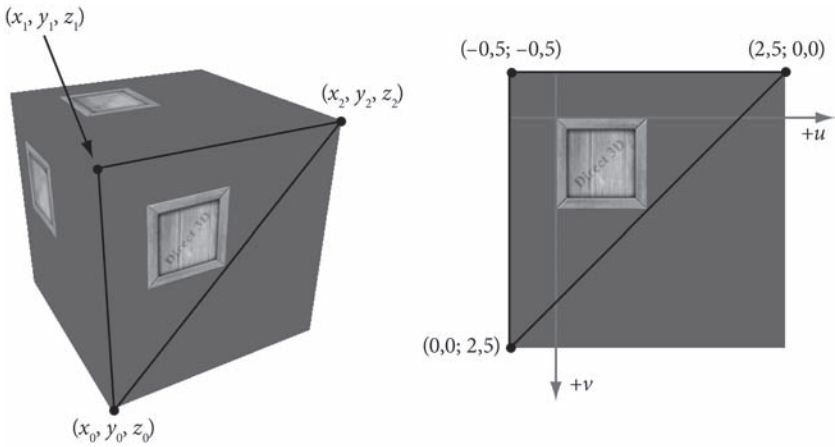


Rysunek 8.10. **Tryb adresowania wrap**

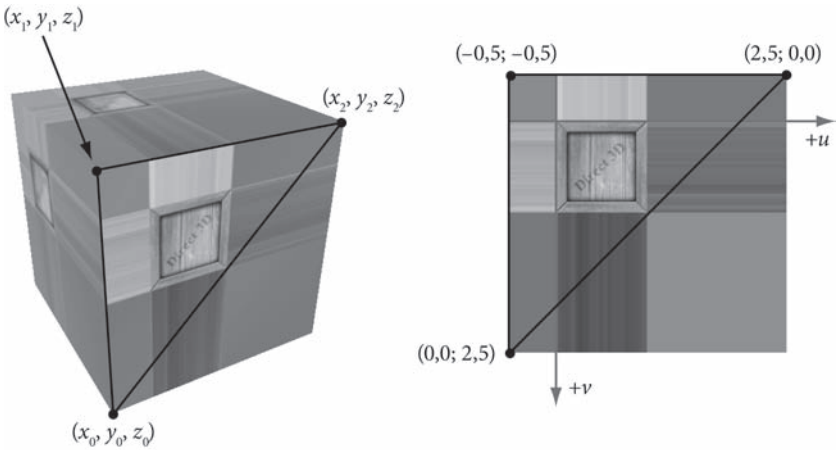
2. *border color* rozszerza funkcję tekstury, odwzorowując współrzędne (u, v) spoza obszaru $[0, 1]^2$ na określony przez programistę kolor (zob. rysunek 8.11).
3. *clamp* rozszerza funkcję tekstury, odwzorowując współrzędne (u, v) spoza obszaru $[0, 1]^2$ na kolor $T(u_0, v_0)$, gdzie (u_0, v_0) to leżący w obszarze $[0, 1]^2$ punkt znajdujący się najbliżej punktu (u, v) (zob. rysunek 8.12).
4. *mirror* rozszerza funkcję tekstury, dodając odbicie lustrzane obrazu w każdym punkcie o współrzędnych całkowitych (zob. rysunek 8.13).

Tryb adresowania jest zawsze określony (domyślna wartość to *wrap*), dzięki czemu dla współrzędnych tekstury spoza przedziału [0, 1] zawsze uzyskamy kolor.

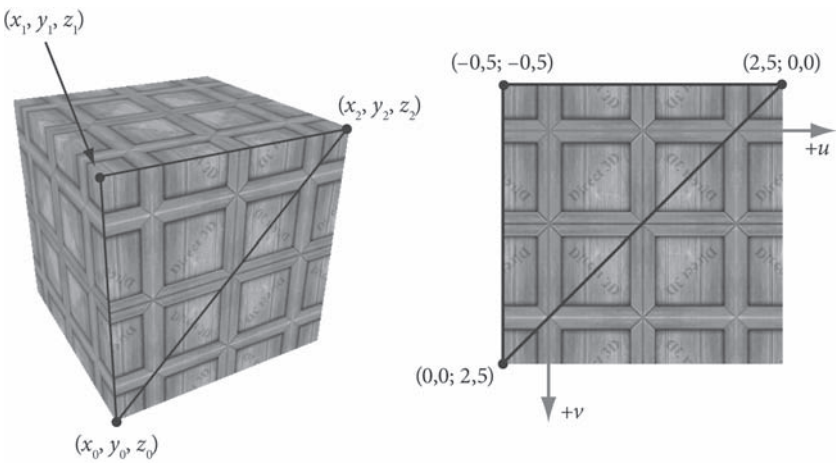
Najczęściej stosowanym trybem adresowania jest prawdopodobnie tryb *wrap*, który pozwala wykorzystać teksturę do pokrycia powierzchni metodą kafelkową. Dzięki temu zyskujemy większą rozdzielczość, bez konieczności dostarczania dodatkowych danych (zawdzięczamy ją powtórzeniom). Przy kafelkowaniu niezwykle istotną kwestią jest widoczność złączeń. Na przykład złączenia kafelków tekstury na skrzynce są widoczne, powtarzający



Rysunek 8.11. **Tryb adresowania border color**

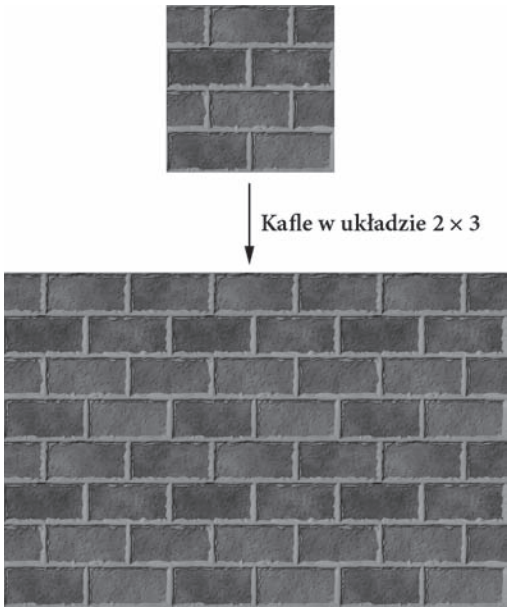


Rysunek 8.12. **Tryb adresowania clamp**



Rysunek 8.13. **Tryb adresowania mirror**

się wzór rzuca się w oczy. Rysunek 8.14 przedstawia natomiast powtórzoną 2·3 razy teksturę muru, w której złączeń nie widać. Taką teksturę nazywamy bezszwową (ang. *seamless*).



Rysunek 8.14. **Ułożona kafelkowo (2×3) tekstura muru. Ponieważ tekstura jest bezszwowa, trudniej dostrzec powtarzający się wzór**

Tryby adresowania określa się w obiektach samplerów. Poniższe przykłady zostały wykorzystane przy tworzeniu rysunków 8.10 – 8.13:

```
SamplerState samTriLinear
{
   Filter = MIN_MAG_MIP_LINEAR;
   AddressU = WRAP;
   AddressV = WRAP;
};

SamplerState samTriLinear
{
   Filter = MIN_MAG_MIP_LINEAR;
   AddressU = BORDER;
   AddressV = BORDER;

   // Niebieski kolor obramowania.
   BorderColor = float4(0.0f, 0.0f, 1.0f, 1.0f);
};

SamplerState samTriLinear
{
   Filter = MIN_MAG_MIP_LINEAR;
   AddressU = CLAMP;
   AddressV = CLAMP;
};

SamplerState samTriLinear
```

```
{
   Filter = MIN_MAG_MIP_LINEAR;
   AddressU = MIRROR;
   AddressV = MIRROR;
};
```

Warto zauważyć, że tryb adresowania można określić osobno dla kierunków u *i* v. *Zachęcamy do eksperymentów z różnymi kombinacjami.*

8.9. PRZEKSZTAŁCANIE TEKSTUR

Współrzędne tekstury reprezentują dwuwymiarowe punkty na płaszczyźnie tekstury. Możemy je zatem przesuwać, obracać i skalować tak jak wszystkie inne punkty. Oto kilka przykładów zastosowania przekształceń tekstury:

1. Rozciągamy teksturę cegieł na całą powierzchnię ściany. Współrzędne tekstury wierzchołków ściany mieszczą się obecnie w przedziale [0, 1]. Skalujemy współrzędne tekstury o 4, aby mieściły się w przedziale [0, 4], w wyniku czego tekstura jest powtarzana cztery razy w poziomie i cztery razy w pionie.
2. Rozciągamy teksturę chmur na tle czystego nieba. Przesunięcie współrzędnych tekstury w funkcji czasu pozwala uzyskać animowane chmury.
3. Obrót tekstury przydaje się przy niektórych efektach cząsteczkowych, można na przykład obracać teksturę kuli ognia w funkcji czasu.

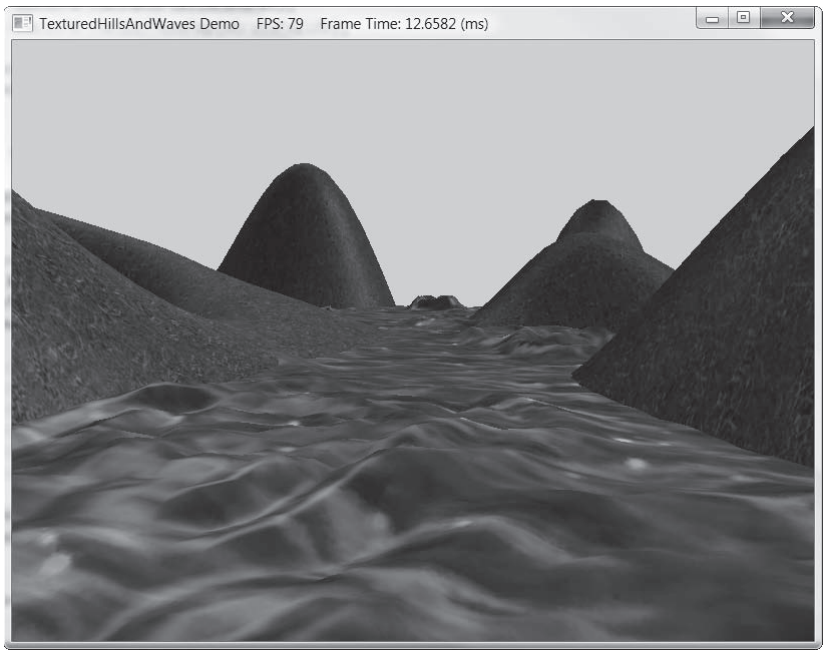
Przekształceń współrzędnych tekstury dokonuje się tak samo jak zwykłych przekształceń. Określamy macierz przekształcenia i mnożymy przez nią wektor współrzędnych tekstury. Na przykład:

```
// Zmienna bufora stałego
float4x4 gTexMtx;

// W programie shadera
vOut.texC = mul(float4(vIn.texC, 0.0f, 1.0f), gTexMtx);
```

Ponieważ pracujemy z dwuwymiarowymi teksturami, interesują nas wyłącznie przekształcenia dwóch pierwszych współrzędnych. Jeśli na przykład macierz tekstury przekształci współrzędną *z*, nie wpłynie to w żaden sposób na współrzędne tekstury.

8.10. APLIKACJA „LAND TEX”

W tej aplikacji scenę zawierającą pagórki i wodę wzbogacimy o tekstury. Pierwszym zadaniem będzie kafelkowanie lądu teksturą trawy. Z racji dużego rozmiaru powierzchni siatki lądu rozciągnięcie tekstury nie byłoby najlepszym rozwiązaniem, jako że zbyt mało tekseli pokrywałoby wtedy jeden trójkąt. Innymi słowy, rozdzielczość tekstury jest zbyt mała w porównaniu do powierzchni, zatem skutkiem rozciągnięcia tekstury byłyby artefakty powiększenia. Dlatego powtarzamy teksturę trawy na powierzchni lądu, aby zwiększyć rozdzielczość. Drugie zadanie będzie polegać na przesuwaniu tekstury wody po geometrii wody jako funkcji czasu. Dzięki ruchowi woda wydaje się bardziej naturalna. Rysunek 8.15 przedstawia zrzut ekranu aplikacji.



Rysunek 8.15. **Zrzut ekranu aplikacji „Land Tex"**

8.10.1. Generowanie współrzędnych tekstury dla siatki

Rysunek 8.16 przedstawia siatkę $m \times n$ na płaszczyźnie xz oraz odpowiadającą jej siatkę znormalizowanej przestrzeni tekstury $[0, 1]^2$. Z rysunku wynika, że współrzędne tekstury wierzchołka ij na płaszczyźnie xz to współrzędne wierzchołka ij siatki w przestrzeni tekstury. Współrzędne wierzchołka ij w przestrzeni tekstury to:

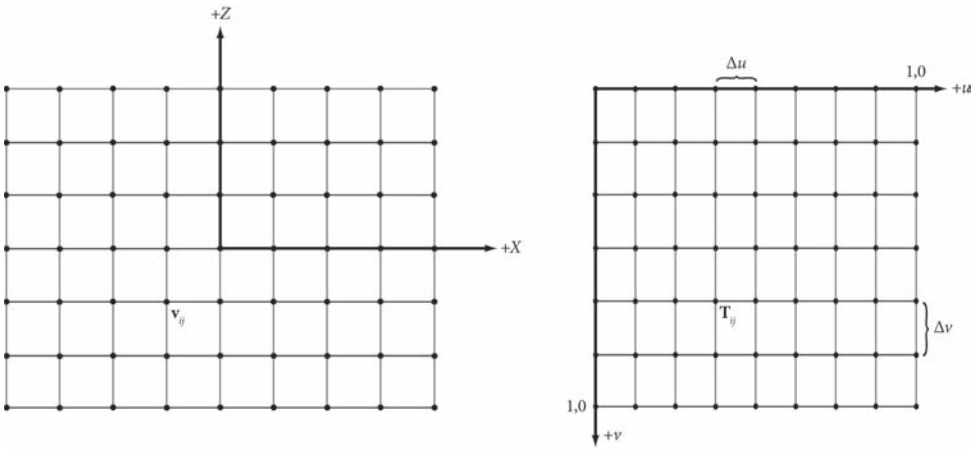
$$u_{ij} = j \cdot \Delta u$$

$$v_{ij} = i \cdot \Delta v$$

gdzie $\Delta u = \frac{1}{n-1}$ oraz $\Delta v = \frac{1}{m-1}$.

A więc aby wygenerować współrzędne tekstury, wykorzystamy w metodzie `GeometryGenerator::CreateGrid` następujący kod:

```
void GeometryGenerator::CreateGrid(float width, float depth, UINT m, UINT n, MeshData&
  meshData)
{
   UINT vertexCount = m*n;
   UINT faceCount = (m-1)*(n-1)*2;

   //
   // Utwórz wierzchołki.
   //

   float halfWidth = 0.5f*width;
   float halfDepth = 0.5f*depth;
```



Rysunek 8.16. **Współrzędne tekstury dla wierzchołka ij siatki w przestrzeni xz odpowiadają wierzchołkowi T_{ij} w przestrzeni uv**

```
    float dx = width / (n-1);
    float dz = depth / (m-1);

    float du = 1.0f / (n-1);
    float dv = 1.0f / (m-1);

    meshData.Vertices.resize(vertexCount);
    for(UINT i = 0; i < m; ++i)
    {
       float z = halfDepth - i*dz;
       for(UINT j = 0; j < n; ++j)
       {
          float x = -halfWidth + j*dx;

          meshData.Vertices[i*n+j].Position = XMFLOAT3(x, 0.0f, z);
          meshData.Vertices[i*n+j].Normal = XMFLOAT3(0.0f, 1.0f, 0.0f);
          meshData.Vertices[i*n+j].TangentU = XMFLOAT3(1.0f, 0.0f, 0.0f);

          // Rozciągnij teksturę na powierzchni siatki.
          meshData.Vertices[i*n+j].TexC.x = j*du;
          meshData.Vertices[i*n+j].TexC.y = i*dv;
       }
    }
}
```

8.10.2. Kafelkowanie

Ustaliliśmy, że siatkę lądu będziemy pokrywać teksturą trawy metodą kafelkową. Jednak wszystkie obliczane dotychczas współrzędne należą do dziedziny $[0, 1]^2$, a więc kafelkowanie nie występuje. Aby ułożyć teksturę kafelkowo, ustawiamy tryb adresowania *wrap* i skalujemy współrzędne tekstury o 5 za pomocą macierzy przekształcenia tekstury. Współrzędne są więc odwzorowywane na dziedzinę $[0, 5]^2$, a powierzchnia siatki lądu jest pokrywana kafelkami w układzie 5×5.

```
XMMATRIX grassTexScale = XMMatrixScaling(5.0f, 5.0f, 0.0f);
XMStoreFloat4x4(&mGrassTexTransform, grassTexScale);
...
```

```
Effects::BasicFX->SetTexTransform(XMLoadFloat4x4(&mGrassTexTransform));
...
activeTech->GetPassByIndex(p)->Apply(0, md3dImmediateContext);
md3dImmediateContext->DrawIndexed(mLandIndexCount, 0, 0);
```

8.10.3. Animacja tekstury

W celu animacji tekstury wody nałożonej na geometrię w metodzie `UpdateScene` korzystamy z funkcji czasu, która przesuwa współrzędne tekstury w płaszczyźnie tekstury. Przy odpowiednio małym przemieszczeniu w następujących po sobie klatkach uzyskamy efekt płynnej animacji. Używamy trybu adresowania *wrap* oraz bezszwowej tekstury, dzięki czemu współrzędne tekstury można dowolnie przesuwać na całej płaszczyźnie przestrzeni tekstury. W zaprezentowanym niżej fragmencie kodu obliczamy wektor przesunięcia dla tekstury wody, a także budujemy i ustawiamy macierz tekstury.

```
// Kafelkuj teksturę wody.
XMMATRIX wavesScale = XMMatrixScaling(5.0f, 5.0f, 0.0f);

// Przesuwaj teksturę w czasie.
mWaterTexOffset.y += 0.05f*dt;
mWaterTexOffset.x += 0.1f*dt;
XMMATRIX wavesOffset = XMMatrixTranslation(mWaterTexOffset.x, mWaterTexOffset.y, 0.0f);

// Połącz skalowanie i przesunięcie.
XMStoreFloat4x4(&mWaterTexTransform, wavesScale*wavesOffset);

...
Effects::BasicFX->SetTexTransform(XMLoadFloat4x4(&mWaterTexTransform));
...
activeTech->GetPassByIndex(p)->Apply(0, md3dImmediateContext);
md3dImmediateContext->DrawIndexed(3*mWaves.TriangleCount(), 0, 0);
```

8.11. FORMATY KOMPRESJI TEKSTUR

Obciążenie procesora graficznego szybko wzrasta wraz z rozszerzającym się spektrum tekstur używanych w naszych scenach (pamiętaj, że tekstury rezydują w pamięci GPU, co ma minimalizować czas dostępu). Aby zmniejszyć to obciążenie, Direct3D obsługuje formaty kompresji tekstur: BC1, BC2, BC3, BC4, BC5, BC6 i BC7:

1. BC1 (`DXGI_FORMAT_BC1_UNORM`): Użyj tego formatu, jeżeli chcesz skompresować format obsługujący trzy kanały kolorów i tylko 1-bitową (włączona lub wyłączona) składową alfa.
2. BC2 (`DXGI_FORMAT_BC2_UNORM`): Użyj tego formatu, jeżeli chcesz skompresować format obsługujący trzy kanały kolorów i tylko 4-bitową składową alfa.
3. BC3 (`DXGI_FORMAT_BC3_UNORM`): Użyj tego formatu, jeżeli chcesz skompresować format obsługujący trzy kanały kolorów i 8-bitową składową alfa.
4. BC4 (`DXGI_FORMAT_BC4_UNORM`): Użyj tego formatu, jeżeli chcesz skompresować format zawierający jeden kanał koloru (np. obraz w skali szarości).
5. BC5 (`DXGI_FORMAT_BC5_UNORM`): Użyj tego formatu, jeżeli chcesz skompresować format obsługujący dwa kanały kolorów.

6. BC6 (`DXGI_FORMAT_BC6_UF16`): Użyj tego formatu, aby skorzystać z kompresji danych HDR (ang. *high dynamic range*).
7. BC7 (`DXGI_FORMAT_BC7_UNORM`): Użyj tego formatu, aby skorzystać z wysokiej jakości kompresji RGBA. Format ten w szczególności zmniejsza liczbę błędów spowodowanych kompresją map normalnych.

Uwaga *Skompresowana tekstura może być użyta tylko jako dane wejściowe etapu cieniowania pikseli potoku renderowania.*

Uwaga *Ponieważ algorytmy kompresji blokowej operują na blokach pikseli o rozmiarze 4×4, wymiary tekstury muszą być wielokrotnością 4.*

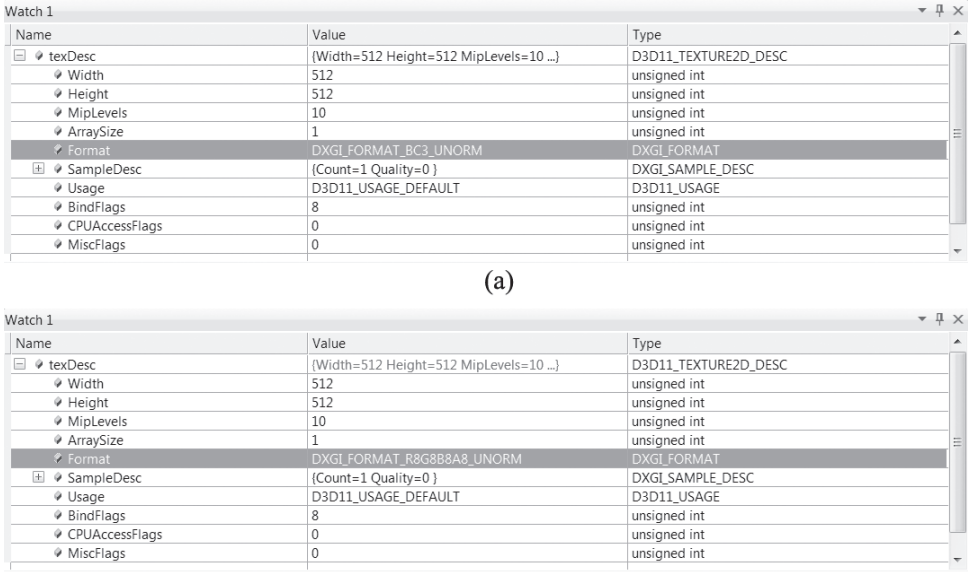
Zaletą danych w tych formatach jest fakt, że mogą być przechowywane w pamięci GPU w skompresowanej postaci, a następnie w momencie użycia dekompresowane na bieżąco przez procesor graficzny.

Direct3D może podczas ładowania dokonać konwersji pliku z nieskompresowanymi danymi graficznymi na format skompresowany, jeśli użyjemy argumentu `pLoadInfo` w funkcji `D3DX11CreateShaderResourceViewFromFile`. Na przykład w poniższym fragmencie kodu ładujemy plik BMP:

```
D3DX11_IMAGE_LOAD_INFO loadInfo;
loadInfo.Format = DXGI_FORMAT_BC3_UNORM;

HR(D3DX11CreateShaderResourceViewFromFile(md3dDevice,
  L"Textures/darkbrick.bmp", &loadInfo, 0, &mDiffuseMapSRV, 0));

// Pobierz teksturę 2D z widoku zasobu.
ID3D11Texture2D* tex;
mDiffuseMapSRV->GetResource((ID3D11Resource**)&tex);

// Pobierz opis tekstury 2D.
D3D11_TEXTURE2D_DESC texDesc;
tex->GetDesc(&texDesc);
```

Na rysunku 8.17a pokazano opis tekstury `texDesc` w debuggerze. Jak widać, zawiera ona żądany format. Gdybyśmy zamiast tego podali w argumencie `pLoadInfo` wartość `null`, użyty zostałby format z obrazu źródłowego (zob. rysunek 8.17b), którym w tym przypadku jest `DXGI_FORMAT_R8G8B8A8_UNORM`.

Alternatywnym formatem jest DDS (ang. *Direct Draw Surface*) przechowujący tekstury bezpośrednio. Aby stworzyć teksturę w tym formacie, należy załadować plik graficzny z poziomu narzędzia DirectX Texture Tool (*DxTex.exe*) znajdującego się w katalogu SDK: *D:\Microsoft DirectX SDK (June 2010)\Utilities\bin\x86*. Następnie wybrać kolejno *Menu/Format/Change Surface Format* oraz format DXT1, DXT2, DXT3, DXT4 lub DXT5 i zapisać plik w formacie DDS. Formaty te to tak naprawdę formaty kompresji tekstur Direct3D 9, ale DXT1 jest taki sam jak BC1, DXT2 i DXT3 takie same jak BC2, a DXT4 i DXT5 takie same jak BC3. Na przykład jeżeli zapiszemy plik jako DXT1, a następnie załadujemy go za pomocą `D3DX11CreateShaderResourceViewFromFile`, tekstura będzie miała format `DXGI_FOR`
`↪MAT_BC1_UNORM`:



Rysunek 8.17. **(a) Tekstura tworzona w skompresowanym formacie DXGI_FORMAT_BC3_UNORM. (b) Tekstura tworzona w nieskompresowanym formacie DXGI_FORMAT_R8G8B8A8_UNORM**

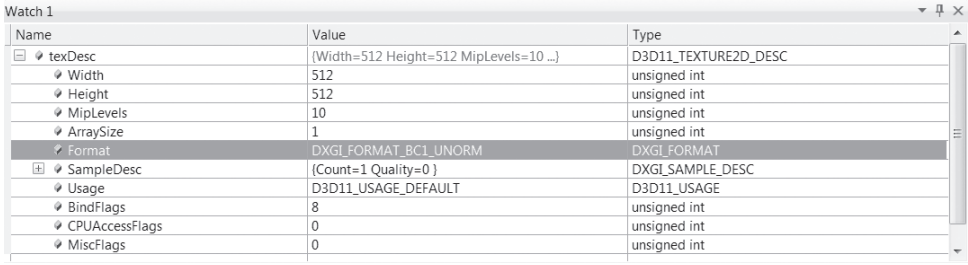
```
HR(D3DX11CreateShaderResourceViewFromFile(md3dDevice,
  L"Textures/darkbrickdxt1.dds", 0, 0, &mDiffuseMapSRV, 0));

// Pobierz teksturę 2D z widoku zasobu.
ID3D11Texture2D* tex;
mDiffuseMapSRV->GetResource((ID3D11Resource**)&tex);

// Pobierz opis tekstury 2D.
D3D11_TEXTURE2D_DESC texDesc;
tex->GetDesc(&texDesc);
```

Zauważ, że jeśli plik DDS używa jednego z formatów kompresji, przekazując wartość `null` w argumencie `pLoadInfo`, wymusimy użycie przez `D3DX11CreateShaderResourceView ↪FromFile` formatu kompresji określonego w pliku.

Do tworzenia tekstur w formatach BC4 i BC5 możesz wykorzystać narzędzie NVIDIA Texture Tools (*http://code.google.com/p/nvidia-texture-tools/*). Jeśli zaś chodzi o BC6 i BC7, DirectX SDK zawiera narzędzie BC6HBC7EncoderDecoder11, które umożliwia konwersję plików tekstur na te formaty. Dostępny jest również pełny kod źródłowy programu, można go zatem zintegrować z własnymi projektami. Co ciekawe, program korzysta przy konwersji z procesora graficznego, o ile karta graficzna obsługuje shadery obliczeniowe. Takie rozwiązanie jest dużo szybsze, niż gdyby to zadanie miał wykonywać procesor główny.

Narzędzie DirectX Texture Tool pozwala też wygenerować sekwencję mipmap (*Menu/Format/Generate Mip Maps*) oraz zapisać je jako plik DDS. Dzięki temu mipmapy są przygotowywane wcześniej i przechowywane razem z plikiem, przy ładowaniu nie są więc potrzebne dodatkowe obliczenia.

Inną zaletą przechowywania tekstur w postaci skompresowanej w plikach DDS jest oszczędność miejsca na dysku twardym.



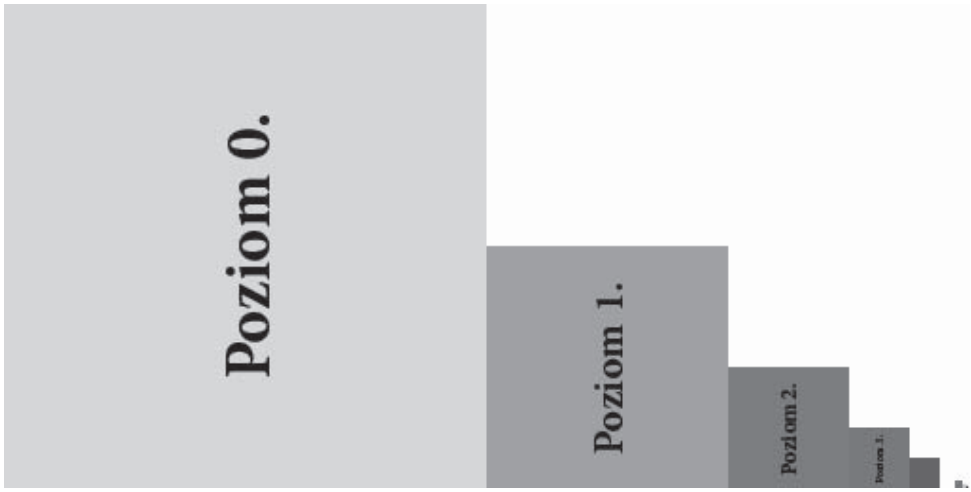
Rysunek 8.18. **Tekstura tworzona w formacie DXGI_FORMAT_BC3_UNORM**

8.12. PODSUMOWANIE

1. Współrzędne tekstury służą do wyznaczania trójkątnego obszaru tekstury, który zostanie nałożony na trójkąt w scenie.
2. Tekstury można tworzyć z przechowywanych na dysku plików graficznych za pomocą funkcji `D3DX11CreateShaderResourceViewFromFile`.
3. Tekstury mogą być filtrowane przy użyciu stanów samplera, w których zdefiniowano odpowiednie filtry pomniejszania, powiększania i mipmappingu.
4. Tryby adresowania decydują o tym, jak współrzędne tekstury spoza przedziału [0, 1] będą obsłużone przez Direct3D. Na przykład czy tekstura powinna być kafelkowana, odbita lustrzanie, a może rozszerzona na podstawie pikseli brzegowych?
5. Współrzędne tekstury podlegają skalowaniu, obrotowi i przesunięciu, tak jak wszystkie punkty. Dokonując małych i stopniowych przekształceń współrzędnych tekstury w każdej klatce, uzyskujemy efekt animowanej tekstury.
6. Kompresja tekstur w formatach Direct3D BC1, BC2, BC3, BC4, BC5, BC6 czy BC7 umożliwia znaczące oszczędności pamięci GPU. Do tworzenia tekstur w formatach BC1, BC2 i BC3 można wykorzystać narzędzie DirectX Texture Tool. Narzędzie NVIDIA Texture Tools (*http://code.google.com/p/nvidia-texture-tools/*) służy natomiast do generowania tekstur w formatach BC4 i BC5, a program SDK BC6HBC7EncoderDecoder11 do generowania tekstur w formatach BC6 i BC7.

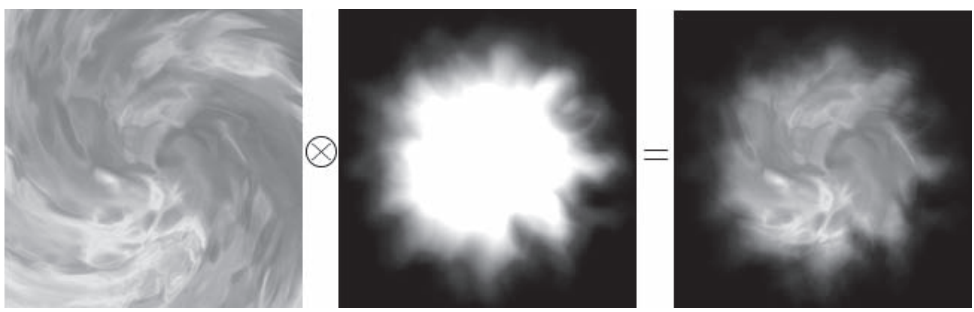
8.13. ĆWICZENIA

1. Poeksperymentuj z aplikacją „Crate”, zmieniając współrzędne tekstury i używając różnych kombinacji trybów adresowania i opcji filtrowania. W szczególności postaraj się uzyskać efekty z obrazków 8.7, 8.9, 8.10, 8.11, 8.12 i 8.13.
2. Za pomocą narzędzia DirectX Texture Tool możemy ręcznie określić poszczególne poziomy mipmap (*File/Open Onto This Surface*). Utwórz plik DDS z sekwencją mipmap, jak na rysunku 8.19. Tekst lub kolor powinny być zróżnicowane, aby łatwiej było rozróżnić poszczególne poziomy. Zmodyfikuj aplikację „Crate”, korzystając z tej tekstury, a następnie przybliż obraz tak, aby było dokładnie widać zmieniające się mipmapy. Wypróbuj zarówno punktowe, jak i liniowe filtrowanie mipmap.



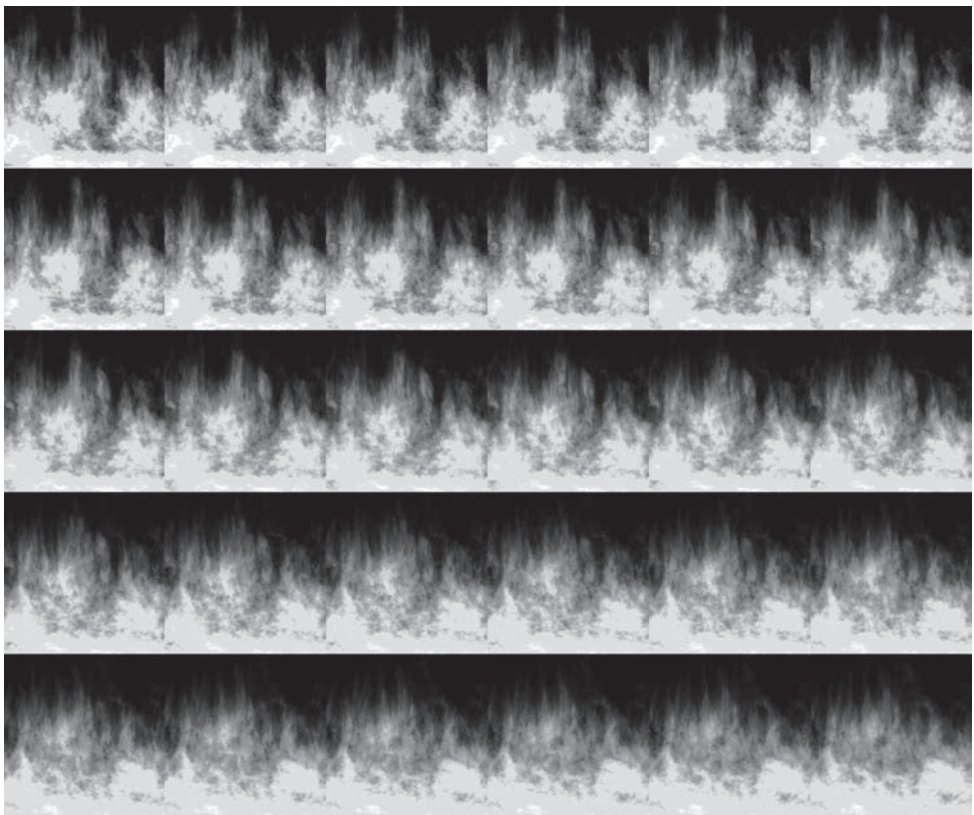
Rysunek 8.19. **Samodzielnie skonstruowana sekwencja mipmap z łatwo rozróżnialnymi poziomami**

3. Dwie tekstury o tych samych rozmiarach można połączyć poprzez różne operacje w celu uzyskania nowego obrazu. Technikę, w której do osiągnięcia pewnego efektu wykorzystujemy więcej niż jedną teksturę, nazywamy **multiteksturowaniem**. Możemy na przykład dodawać, odejmować i mnożyć składowe odpowiadających sobie tekseli dwóch tekstur. Na rysunku 8.20 pokazano wynik mnożenia składowych dwóch tekstur w postaci kuli ognia. W ramach ćwiczenia zmodyfikuj aplikację „Crate", łącząc w shaderze pikseli dwie pierwotne tekstury z rysunku 8.20 tak, aby kula ognia znalazła się na każdej ze ścian sześcianu. Zauważ, że obsługa więcej niż jednej tekstury będzie wymagała modyfikacji pliku *Basic.fx*.



Rysunek 8.20. **Mnożąc składowe odpowiadających sobie tekseli dwóch tekstur, otrzymujemy nową teksturę**

4. Zmodyfikuj rozwiązanie ćwiczenia 3., tak aby tekstura kuli ognia na każdej ścianie kostki obracała się w funkcji czasu.
5. W dołączonych do tego rozdziału plikach znajdziesz katalog zawierający 120 klatek animacji ognia do odtworzenia w czasie 4 sekund (w tempie 30 klatek na sekundę). Na rysunku 8.21 pokazano pierwsze 30 klatek. Zmodyfikuj aplikację „Crate" tak, aby animacja ta była odtwarzana na każdej ze ścian kostki.



Rysunek 8.21. **Klatki wygenerowanej wcześniej animacji ognia**

Załaduj obrazy do tablicy ze 120 obiektami tekstur. Zacznij od tekstury dla pierwszej klatki i co 1/30 sekundy zwiększaj indeks tablicy, przechodząc do kolejnej. Po osiągnięciu 120. tekstury wróć do pierwszej i kontynuuj iterację. Taki proces jest czasami nazywany ***stronicowaniem*** *(ang.* page flipping*), ponieważ przypomina przewracanie kartek książki.*

Przetwarzanie pojedynczych klatek animacji tekstury, jedna po drugiej, jest mało wydajne. Lepiej umieścić je w jednym atlasie tekstur i przesuwać współrzędne tekstury co 1/30 sekundy do następnej klatki animacji. Na potrzeby tego ćwiczenia wystarczy nam jednak ta mało wydajna metoda.

6. Niech $\mathbf{p}_0$, $\mathbf{p}_1$ i $\mathbf{p}_2$ będą wierzchołkami trójkąta w przestrzeni trójwymiarowej o współrzędnych tekstury odpowiednio $\mathbf{q}_0$, $\mathbf{q}_1$ i $\mathbf{q}_2$. Przypomnijmy z podrozdziału 8.2, że dla dowolnego punktu trójwymiarowego trójkąta $\mathbf{p}(s, t) = \mathbf{p}_0+s(\mathbf{p}_1-\mathrm{p}_0)+t(\mathbf{p}_2-\mathrm{p}_0)$, gdzie $s \geq 0$, $t \geq 0$, $s+t \leq 1$, jego współrzędne tekstury (u, v) obliczamy, interpolując liniowo współrzędne tekstury wierzchołków na całą powierzchnię trójkąta za pomocą parametrów s i t:

$$(u,v) = \mathbf{q}_0 + s(\mathbf{q}_1 - \mathbf{q}_0) + t(\mathbf{q}_2 - \mathbf{q}_0)$$

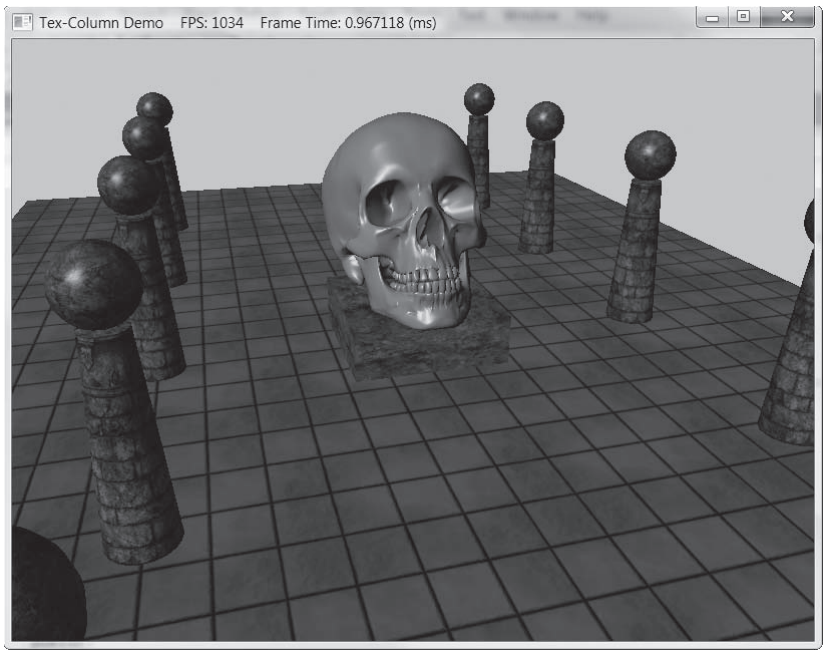
(a) Przy zadanych (u, v) oraz $\mathbf{q}_0$, $\mathbf{q}_1$ i $\mathbf{q}_2$ znajdź wzór na (s, t) wyrażony za pomocą u i v.

Skorzystaj z równania wektorowego $(u, v) - \boldsymbol{q}_0 = s(\boldsymbol{q}_1 - \boldsymbol{q}_0) + t(\boldsymbol{q}_2 - \boldsymbol{q}_0)$.

(b) Wyraź p jako funkcję u i v, czyli znajdź wzór na $\mathbf{p} = \mathbf{p}(u, v)$.

(c) Oblicz wektory $\partial\mathbf{p}/\partial u$ oraz $\partial\mathbf{p}/\partial v$ i podaj ich geometryczną interpretację.

7. Zmodyfikuj aplikację „Lit Skull" z poprzedniego rozdziału, dodając tekstury do podłoża, kolumn i sfer (zob. rysunek 8.22).



Rysunek 8.22. **Scena z kolumnami po nałożeniu tekstur**

SKOROWIDZ

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

Zbuduj wirtualny świat za pomocą DirectX!

DirectX to zestaw funkcji wspierających tworzenie zaawansowanej grafiki w systemie Windows. Historia tego dodatku sięga 1995 roku, kiedy po wprowadzeniu na rynek systemu Windows 95 programiści tworzący gry z dużym oporem rozstawali się z systemem DOS. Od tego czasu DirectX jest nieustannie rozwijany, a gry i trójwymiarowe animacje tworzone z wykorzystaniem najnowszej wersji zapierają dech w piersiach. Jeżeli chcesz poznać możliwości tego dodatku i wykorzystać jego potencjał, to trafiłeś na doskonałą publikację!

Żadna inna nie omawia w tak szczegółowy sposób zagadnień związanych z tworzeniem grafiki i animacji przy użyciu DirectX. Książka została podzielona na trzy części. Pierwsza pozwoli Ci zdobyć podstawową wiedzę na temat fundamentów grafiki trójwymiarowej. Nauczysz się lub przypomnisz sobie, jak prowadzić działania na wektorach, przekształcać te wektory i stosować macierze. W części drugiej szczególny nacisk został położony na poznanie Direct3D. Dowiesz się, jak zainicjalizować system, nanosić tekstury oraz cieniować obiekty. Część trzecia zawiera mnóstwo przydatnych informacji na temat widoku pierwszej osoby, prowadzenia kamery, systemu cząstek, zaawansowanego mapowania oraz animowania. Książka ta jest obowiązkową lekturą każdego programisty zajmującego się grafiką 3D!

Dzięki tej książce:

- poznasz fundamenty grafiki 3D — wektory, macierze i ich przekształcenia
- zaznajomisz się z dostępnymi funkcjami oraz sposobem pracy z Direct3D
- dowiesz się, jak korzystać z tekstur i cieni
- wykorzystasz system cząstek
- poznasz potencjał DirectX

ISBN 978-83-246-7474-9

9 788324 674749

cena: 129,00 zł