

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

Translation Copyright © 2014 HELION SA

Original Copyright © 2012 by Mercury Learning and Information LLC. All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

SPIS TREŚCI

Podziękowania	19
Wstęp	21
Docelowy odbiorca	22
Wymagana wiedza	22
Potrzebne narzędzia programistyczne i sprzęt	23
Biblioteka D3DX	23
Dokumentacja SDK i przykładowe aplikacje	23
Czytelność	25
Przykładowe programy i dodatkowe źródła online	26
Przygotowanie projektu aplikacji w Visual Studio 2010	26
Tworzenie projektu Win32	26
Dołączanie bibliotek DirectX	27
Ustawianie ścieżek wyszukiwania	28
Wprowadzanie kodu źródłowego i budowa projektu	30

CZĘŚĆ I PODSTAWY MATEMATYCZNE

Rozdział 1. Algebra wektorów	35
1.1. Wektory	35
1.1.1. Wektory i układy współrzędnych	36
1.1.2. Lewo- i praworęczne układy współrzędnych	38
1.1.3. Podstawowe działania na wektorach	38
1.2. Długość i wektory jednostkowe	40
1.3. Iloczyn skalarny	42
1.3.1. Ortogonalizacja	44
1.4. Iloczyn wektorowy	46
1.4.1. Pseudoiloczyn wektorowy dla przestrzeni dwuwymiarowych	47
1.4.2. Ortogonalizacja z iloczynem wektorowym	47
1.5. Punkty	48
1.6. Wektory XNA Math	49
1.6.1. Typy wektorowe	50
1.6.2. Metody ładujące i metody zapisujące	51
1.6.3. Przekazywanie argumentów	52
1.6.4. Stałe wektorowe	53
1.6.5. Przeciążone operatory	54
1.6.6. Przydatne funkcje	54
1.6.7. Funkcje ustawiające	55

1.6.8.	Funkcje wektorowe	56
1.6.9.	Błąd obliczeń zmiennoprzecinkowych	59
1.7.	Podsumowanie	60
1.8.	Ćwiczenia	61
Rozdział 2.	Algebra macierzy	65
2.1.	Definicja	65
2.2.	Mnożenie macierzy	67
2.2.1.	Definicja	67
2.2.2.	Mnożenie macierzy przez wektor	68
2.2.3.	Łączność	69
2.3.	Macierz transponowana	69
2.4.	Macierz jednostkowa	70
2.5.	Wyznacznik macierzy	71
2.5.1.	Macierze zredukowane	72
2.5.2.	Definicja	72
2.6.	Macierz dołączona	74
2.7.	Macierz odwrotna	74
2.8.	Macierze XNA	76
2.8.1.	Typy macierzowe	76
2.8.2.	Funkcje macierzowe	78
2.8.3.	Przykładowy program z macierzami XNA	78
2.9.	Podsumowanie	80
2.10.	Ćwiczenia	81
Rozdział 3.	Przekształcenia	85
3.1.	Przekształcenia liniowe	85
3.1.1.	Definicja	85
3.1.2.	Reprezentacja macierzowa	86
3.1.3.	Skalowanie	87
3.1.4.	Obrót	89
3.2.	Przekształcenia afiniczne	91
3.2.1.	Współrzędne jednorodne	91
3.2.2.	Definicja i reprezentacja macierzowa	92
3.2.3.	Przesunięcie	93
3.2.4.	Macierze afiniczne dla skalowania i obrotu	95
3.2.5.	Interpretacja geometryczna macierzy przekształcenia afinicznego	95
3.3.	Składanie przekształceń	97
3.4.	Przekształcenia zamiany współrzędnych	98
3.4.1.	Wektory	99
3.4.2.	Punkty	100
3.4.3.	Reprezentacja macierzowa	101
3.4.4.	Łączność i macierze zamiany współrzędnych	102
3.4.5.	Macierze odwrotne i macierze zamiany współrzędnych	102
3.5.	Macierz przekształcenia kontra macierz zamiany współrzędnych	103
3.6.	Funkcje przekształcające XNA Math	105
3.7.	Podsumowanie	106
3.8.	Ćwiczenia	107

CZĘŚĆ II PODSTAWY DIRECT3D

Rozdział 4. Inicjalizacja Direct3D	115
4.1. Uwagi wstępne	115
4.1.1. Przegląd Direct3D	115
4.1.2. Technologia COM	116
4.1.3. Tekstury i formaty zasobów	116
4.1.4. Łańcuch wymiany i stronicowanie	117
4.1.5. Buforowanie głębokości	118
4.1.6. Widoki zasobów tekstury	121
4.1.7. Teoria wielokrotnego próbkowania	122
4.1.8. Wielokrotne próbkowanie w Direct3D	124
4.1.9. Możliwości sprzętu	124
4.2. Inicjalizacja Direct3D	125
4.2.1. Tworzenie urządzenia i kontekstu	126
4.2.2. Sprawdzenie obsługiwanego poziomu jakości antialiasingu z 4-krotnym próbkowaniem	128
4.2.3. Opis łańcucha wymiany	129
4.2.4. Tworzenie łańcucha wymiany	130
4.2.5. Tworzenie widoku celu renderowania	131
4.2.6. Tworzenie bufora i widoku głębokości/szablonu	132
4.2.7. Powiązanie widoków z etapem łączenia wyników	135
4.2.8. Ustawienie okna widoku	135
4.3. Czas i animacja	137
4.3.1. Licznik wydajności	137
4.3.2. Klasa GameTimer	138
4.3.3. Okresy między klatkami	139
4.3.4. Czas całkowity	140
4.4. Szkielet przykładowych aplikacji	143
4.4.1. D3DApp	144
4.4.2. Metody niezwiązane ze szkieletem	145
4.4.3. Metody szkieletu	146
4.4.4. Statystyki klatek	147
4.4.5. Obsługa komunikatów	148
4.4.6. Tryb pełnoekranowy	150
4.4.7. Aplikacja „Init Direct3D”	151
4.5. Wykrywanie błędów w aplikacjach Direct3D	152
4.6. Podsumowanie	154
4.7. Ćwiczenia	155
Rozdział 5. Potok renderowania	159
5.1. Złudzenie trójwymiarowości	159
5.2. Reprezentacja modelu	162
5.3. Kolory w grafice komputerowej	163
5.3.1. Działania na kolorach	165
5.3.2. Kolor 128-bitowy	165
5.3.3. Kolor 32-bitowy	166
5.4. Etapy potoku renderowania	168
5.5. Etap zbierania danych	168

5.5.1.	Wierzchołki	168
5.5.2.	Topologia prymitywów	168
5.5.3.	Indeksy	173
5.6.	Etap cieniowania wierzchołków	175
5.6.1.	Przestrzeń lokalna i przestrzeń świata	175
5.6.2.	Przestrzeń widoku	179
5.6.3.	Rzutowanie i jednorodna przestrzeń obcinania	182
5.7.	Etapy teselacji	190
5.8.	Etap cieniowania geometrii	191
5.9.	Obcinanie	191
5.10.	Etap rasteryzacji	193
5.10.1.	Przekształcenie okna widoku	193
5.10.2.	Usuwanie niewidocznych powierzchni	194
5.10.3.	Interpolacja atrybutów wierzchołków	195
5.11.	Etap cieniowania pikseli	197
5.12.	Etap łączenia wyników	197
5.13.	Podsumowanie	197
5.14.	Ćwiczenia	198

Rozdział 6. Rysowanie w Direct3D 203

6.1.	Wierzchołki i formaty wejścia	203
6.2.	Bufory wierzchołków	208
6.3.	Indeksy i bufory indeksów	212
6.4.	Przykładowy shader wierzchołków	215
6.5.	Bufor stały	218
6.6.	Przykładowy shader pikseli	219
6.7.	Stany renderowania	221
6.8.	Efekty	224
6.8.1.	Pliki efektów	224
6.8.2.	Kompilacja shaderów	226
6.8.3.	Łączenie efektów z aplikacją C++	229
6.8.4.	Rysowanie za pomocą efektów	230
6.8.5.	Wczesna kompilacja efektu	231
6.8.6.	Framework efektów jako „generator shaderów”	233
6.8.7.	Kod asemblera	237
6.9.	Aplikacja „Box”	242
6.10.	Aplikacja „Hills”	248
6.10.1.	Generowanie wierzchołków siatki	248
6.10.2.	Generowanie indeksów siatki	251
6.10.3.	Funkcja wyznaczająca wysokość	252
6.11.	Aplikacja „Shapes”	254
6.11.1.	Tworzenie siatki walca	255
6.11.2.	Tworzenie siatki sfery	259
6.11.3.	Tworzenie siatki sfery geodezyjnej	259
6.11.4.	Kod aplikacji	262
6.12.	Ładowanie geometrii z pliku	266
6.13.	Dynamiczne bufory wierzchołków	266
6.14.	Podsumowanie	269
6.15.	Ćwiczenia	271

Rozdział 7. Oświetlenie	275
7.1. Interakcja między światłem i materiałem	276
7.2. Wektory normalne	277
7.2.1. Obliczanie wektorów normalnych	279
7.2.2. Przekształcenia wektorów normalnych	280
7.3. Prawo Lamberta	283
7.4. Światło rozproszone	283
7.5. Światło otoczenia	285
7.6. Światło odbite	285
7.7. Podsumowanie modelu	288
7.8. Określanie kolorów materiałów	289
7.9. Światła równoległe	291
7.10. Światła punktowe	291
7.10.1. Wygaszanie	292
7.10.2. Zasięg	293
7.11. Światła reflektorowe	293
7.12. Implementacja	294
7.12.1. Struktury oświetlenia	294
7.12.2. Upakowanie struktur	296
7.12.3. Implementacja świateł kierunkowych	298
7.12.4. Implementacja świateł punktowych	299
7.12.5. Implementacja świateł reflektorowych	300
7.13. Aplikacja „Lighting”	301
7.13.1. Plik efektu	301
7.13.2. Kod aplikacji C++	304
7.13.3. Obliczanie normalnych	306
7.14. Aplikacja „Lit Skull”	307
7.15. Podsumowanie	309
7.16. Ćwiczenia	310
Rozdział 8. Tekstury	313
8.1. Podsumowanie wiadomości na temat tekstur i zasobów	313
8.2. Współrzędne tekstury	316
8.3. Tworzenie i włączanie obsługi tekstury	318
8.4. Filtry	320
8.4.1. Powiększanie	320
8.4.2. Pomniejszanie	323
8.4.3. Filtrowanie anizotropowe	324
8.5. Próbkowanie tekstur	324
8.6. Tekstury i materiały	326
8.7. Aplikacja „Crate”	326
8.7.1. Określanie współrzędnych tekstury	326
8.7.2. Tworzenie tekstury	327
8.7.3. Ustawianie tekstury	327
8.7.4. Aktualizacja efektu bazowego	328
8.8. Tryby adresowania	332
8.9. Przekształcanie tekstur	335
8.10. Aplikacja „Land Tex”	335
8.10.1. Generowanie współrzędnych tekstury dla siatki	336

8.10.2. Kafelkowanie	337
8.10.3. Animacja tekstury	338
8.11. Formaty kompresji tekstur	338
8.12. Podsumowanie	341
8.13. Ćwiczenia	341
Rozdział 9. Mieszanie kolorów	345
9.1. Równanie mieszania	346
9.2. Operacje mieszania	347
9.3. Współczynniki mieszania	347
9.4. Stan mieszania	348
9.5. Przykłady	351
9.5.1. Mieszanie bez zapisu koloru	351
9.5.2. Dodawanie i odejmowanie kolorów	352
9.5.3. Mnożenie kolorów	352
9.5.4. Przezroczystość	353
9.5.5. Mieszanie kolorów a bufor głębokości	354
9.6. Kanały alfa	355
9.7. Obcinanie pikseli	356
9.8. Mgła	360
9.9. Podsumowanie	365
9.10. Ćwiczenia	366
Rozdział 10. Szablonowanie	369
10.1. Formaty buforów głębokości/szablonu i ich czyszczenie	370
10.2. Test szablonu	371
10.3. Blok stanu głębokości/szablonu	372
10.3.1. Ustawienia głębi	372
10.3.2. Ustawienia szablonu	373
10.3.3. Tworzenie i wiązanie stanu głębokości/szablonu	374
10.3.4. Stany głębokości/szablonu w plikach efektów	375
10.4. Implementacja luster płaskich	376
10.4.1. Symulowanie lustra	376
10.4.2. Definiowanie stanów głębokości/szablonu lustra	378
10.4.3. Rysowanie sceny	379
10.4.4. Kolejność nawijania i odbicia	381
10.5. Implementacja cieni płaskich	382
10.5.1. Cienie światła równoległych	382
10.5.2. Cienie światła punktowych	385
10.5.3. Uogólniona macierz cienia	386
10.5.4. Zapobieganie podwójnemu mieszanu przy użyciu bufora szablonowego	386
10.5.5. Kod cienia	387
10.6. Podsumowanie	388
10.7. Ćwiczenia	389
Rozdział 11. Cieniowanie geometrii	395
11.1. Programowanie shaderów geometrii	396
11.2. Aplikacja „Tree Billboards”	400
11.2.1. Założenia	400

11.2.2.	Struktura wierzchołka	403
11.2.3.	Plik efektu	403
11.2.4.	Obiekt SV_PrimitiveID	408
11.3.	Tablice tekstur	409
11.3.1.	Założenia	409
11.3.2.	Próbkowanie tablicy tekstur	410
11.3.3.	Ładowanie tablic tekstur	410
11.3.4.	Podzасыby tekstury	414
11.4.	Technika alpha-to-coverage	415
11.5.	Podsumowanie	416
11.6.	Ćwiczenia	418

Rozdział 12. Shader obliczeniowy 421

12.1.	Wątki i grupy wątków	422
12.2.	Prosty shader obliczeniowy	425
12.3.	Zasoby danych wejściowych i wyjściowych	426
12.3.1.	Wejściowe tekstury	426
12.3.2.	Tekstury wyjściowe i widoki nieuporządkowanego dostępu (UAV)	426
12.3.3.	Indeksowanie i próbkowanie tekstur	428
12.3.4.	Zasoby ustrukturyzowanych buforów	430
12.3.5.	Kopiowanie wyników shadera obliczeniowego do pamięci systemowej	433
12.4.	Wartości systemowe identyfikujące wątki	436
12.5.	Bufor dopisywania i bufor konsumowany	437
12.6.	Pamięć współdzielona i synchronizacja	439
12.7.	Aplikacja „Blur”	440
12.7.1.	Algorytm rozmywania	440
12.7.2.	Renderowanie do tekstury	443
12.7.3.	Implementacja rozmycia	445
12.7.4.	Program shadera obliczeniowego	449
12.8.	Inne źródła	454
12.9.	Podsumowanie	455
12.10.	Ćwiczenia	457

Rozdział 13. Etapy teselacji 459

13.1.	Typy prymitywów teselacji	460
13.1.1.	Teselacja i shader wierzchołków	461
13.2.	Shader powłoki	461
13.2.1.	Stały shader powłoki	461
13.2.2.	Shader powłoki punktów kontrolnych	463
13.3.	Etap teselatora	465
13.3.1.	Przykłady teselacji czworokątnej łąty	465
13.3.2.	Przykłady teselacji trójkątnej łąty	466
13.4.	Shader dziedziny	466
13.5.	Teselacja czworokąta	468
13.6.	Czworokątne łąty Béziera trzeciego stopnia	471
13.6.1.	Krzywe Béziera	471
13.6.2.	Powierzchnie Béziera trzeciego stopnia	474

13.6.3. Kod do obliczeń powierzchni Béziera trzeciego stopnia	475
13.6.4. Określanie geometrii łąt	476
13.7. Podsumowanie	478
13.8. Ćwiczenia	479

CZĘŚĆ III TEMATY

Rozdział 14. Kamera pierwszej osoby	483
14.1. Przekształcenie widoku	483
14.2. Klasa kamery	485
14.3. Implementacje wybranych metod	486
14.3.1. Metody zwracające wartość XMVECTOR	486
14.3.2. Metoda SetLens	487
14.3.3. Dodatkowe informacje o ostrośłupie widzenia	487
14.3.4. Przekształcanie kamery	488
14.3.5. Budowa macierzy widoku	489
14.4. Użycie klasy kamery	490
14.5. Podsumowanie	491
14.6. Ćwiczenia	492
Rozdział 15. Instancjonowanie i usuwanie powierzchni poza ostrośłupem widzenia	493
15.1. Instancjonowanie sprzętowe	493
15.1.1. Shader wierzchołków	494
15.1.2. Przesyłanie strumieniowe danych instancji	495
15.1.3. Rysowanie danych instancji	496
15.1.4. Tworzenie bufora instancji	497
15.2. Bryły brzegowe i ostrośłupy widzenia	498
15.2.1. Biblioteka XNA Collision	498
15.2.2. Prostopadłościany	498
15.2.3. Sfery	503
15.2.4. Ostrośłupy widzenia	503
15.3. Usuwanie powierzchni poza ostrośłupem widzenia	508
15.4. Podsumowanie	511
15.5. Ćwiczenia	512
Rozdział 16. Wskazywanie obiektów	515
16.1. Przekształcenie ekranu do okna rzutu	517
16.2. Promień wskazujący w przestrzeni świata/lokalnej	520
16.3. Przecięcie promienia i siatki obiektu	521
16.3.1. Przecięcie promienia i prostopadłościanu AABB	522
16.3.2. Przecięcie promienia i sfery	523
16.3.3. Przecięcie promienia i trójkąta	524
16.4. Aplikacja demonstracyjna	526
16.5. Podsumowanie	527
16.6. Ćwiczenia	527

Rozdział 17. Mapowanie sześciennie	529
17.1. Mapowanie sześciennie	529
17.2. Tekstury otoczenia	531
17.2.1. Ładowanie i używanie tekstur sześciennych w Direct3D	534
17.3. Teksturowanie nieba	535
17.4. Modelowanie odbić	537
17.5. Dynamiczne tekstury sześciennie	539
17.5.1. Budowa tekstury sześcienniej i widoków celu renderowania	539
17.5.2. Budowa bufora głębokości i okna widoku	542
17.5.3. Ustawianie kamery	543
17.5.4. Rysowanie do tekstury sześcienniej	544
17.6. Dynamiczne tekstury sześciennie w shaderze geometrii	545
17.7. Podsumowanie	547
17.8. Ćwiczenia	548
Rozdział 18. Mapowanie normalnych i mapowanie przemieszczeń	551
18.1. Do czego nam to potrzebne?	551
18.2. Mapy normalnych	552
18.3. Przestrzeń tekstury (przestrzeń styczna)	555
18.4. Przestrzeń styczna do wierzchołka	557
18.5. Przekształcanie z przestrzeni stycznej do przestrzeni obiektu	558
18.6. Mapowanie normalnych w kodzie shadera	559
18.7. Mapowanie przemieszczeń	563
18.8. Mapowanie przemieszczeń w kodzie shadera	565
18.8.1. Typ prymitywu	565
18.8.2. Shader wierzchołków	566
18.8.3. Shader powłoki	567
18.8.4. Shader dziedziny	569
18.9. Podsumowanie	570
18.10. Ćwiczenia	571
Rozdział 19. Renderowanie terenu	575
19.1. Mapy wysokości	575
19.1.1. Tworzenie mapy wysokości	577
19.1.2. Ładowanie pliku RAW	578
19.1.3. Wygładzanie	579
19.1.4. Widok zasobu shadera mapy wysokości	582
19.2. Teselacja terenu	583
19.2.1. Budowa siatki	583
19.2.2. Shader wierzchołków terenu	586
19.2.3. Współczynniki teselacji	587
19.2.4. Mapowanie przemieszczeń	589
19.2.5. Szacowanie wektora stycznego i normalnego	590
19.3. Usuwanie łąt poza ostrosłupem widzenia	591
19.4. Teksturowanie	596
19.5. Wysokość terenu	598
19.6. Podsumowanie	601
19.7. Ćwiczenia	602

Rozdział 20. Systemy cząsteczek i wyjście strumieniowe	605
20.1. Reprezentacja cząsteczek	605
20.2. Ruch cząsteczek	607
20.3. Losowość	609
20.4. Mieszanie kolorów i systemy cząsteczek	611
20.5. Wyjście strumieniowe	613
20.5.1. Tworzenie shadera geometrii dla wyjścia strumieniowego	613
20.5.2. Wyjście strumieniowe bez renderowania	614
20.5.3. Tworzenie bufora wierzchołków dla wyjścia strumieniowego	616
20.5.4. Wiązanie z etapem wyjścia strumieniowego	616
20.5.5. Odwiązywanie od etapu wyjścia strumieniowego	616
20.5.6. Automatyczne rysowanie	617
20.5.7. Przełączanie buforów wierzchołków	617
20.6. System cząsteczek bazujący na procesorze graficznym	618
20.6.1. Efekty cząsteczkowe	618
20.6.2. Klasa ParticleSystem	618
20.6.3. Emitery	620
20.6.4. Inicjalizacja bufora wierzchołków	620
20.6.5. Metoda Update/Draw	621
20.7. Ogień	623
20.8. Deszcz	627
20.9. Podsumowanie	632
20.10. Ćwiczenia	633
Rozdział 21. Mapowanie cieni	635
21.1. Renderowanie głębokości sceny	635
21.2. Rzutowanie ortograficzne	638
21.3. Współrzędne rzutowe tekstury	640
21.3.1. Implementacja	642
21.3.2. Punkty poza ostrosłupem widzenia	643
21.3.3. Rzutowanie ortograficzne	643
21.4. Mapowanie cieni	644
21.4.1. Opis algorytmu	644
21.4.2. Obciążanie głębokości i aliasing	645
21.4.3. Filtrowanie PCF	648
21.4.4. Budowa mapy cieni	652
21.4.5. Współczynnik cienia	656
21.4.6. Test zacielenia	657
21.4.7. Renderowanie mapy cieni	661
21.5. Duże jądra PCF	661
21.5.1. Funkcje DDX i DDY	662
21.5.2. Rozwiązanie problemu dużego jądra PCF	663
21.5.3. Alternatywne rozwiązanie problemu dużego jądra PCF	665
21.6. Podsumowanie	666
21.7. Ćwiczenia	667

Rozdział 22. Okluzja otoczenia	669
22.1. Okluzja otoczenia na podstawie śledzenia promieni	669
22.2. Okluzja otoczenia przestrzeni ekranu	674
22.2.1. Przebieg renderowania normalnych i głębokości	674
22.2.2. Przebieg okluzji otoczenia	675
22.2.3. Przebieg rozmycia	683
22.2.4. Wykorzystanie mapy okluzji otoczenia	686
22.3. Podsumowanie	689
22.4. Ćwiczenia	689
Rozdział 23. Siatki	693
23.1. Format .m3d	694
23.1.1. Nagłówek	694
23.1.2. Materiały	695
23.1.3. Podzbiory	696
23.1.4. Wierzchołki i indeksy trójkątów	697
23.2. Geometria siatki	699
23.3. Klasa BasicModel	700
23.4. Aplikacja „Mesh Viewer”	701
23.5. Podsumowanie	701
23.6. Ćwiczenia	702
Rozdział 24. Kwaterniony	703
24.1. Liczby zespolone	704
24.1.1. Definicje	704
24.1.2. Interpretacja geometryczna	705
24.1.3. Postać biegunowa i obroty	705
24.2. Algebra kwaternionów	707
24.2.1. Definicja i podstawowe działania	707
24.2.2. Wzory skróconego mnożenia	708
24.2.3. Własności	708
24.2.4. Konwersje	709
24.2.5. Sprzężenie i norma	709
24.2.6. Kwaterniony odwrotne	710
24.2.7. Postać biegunowa	711
24.3. Kwaterniony jednostkowe i obroty	712
24.3.1. Operator obrotu	712
24.3.2. Kwaternionowy operator obrotu w postaci macierzy	714
24.3.3. Zapis macierzy w postaci kwaternionowego operatora obrotu	716
24.3.4. Złożenie	717
24.4. Interpolacja kwaternionów	718
24.5. Funkcje kwaternionowe XNA Math	723
24.6. Aplikacja „Quaternions”	723
24.7. Podsumowanie	728
24.8. Ćwiczenia	729

Rozdział 25. Animacja postaci	731
25.1. Hierarchie układów odniesienia	731
25.1.1. Ujęcie matematyczne	733
25.2. Siatki-skóry	735
25.2.1. Definicje	735
25.2.2. Przekształcenie kość – szkielet	736
25.2.3. Przekształcenie siatka – kość	736
25.2.4. Animacja szkieletu	737
25.2.5. Obliczanie przekształcenia finalnego	738
25.3. Uśrednianie wierzchołków	740
25.4. Ładowanie danych animacji w formacie .m3d	743
25.4.1. Dane wierzchołka skóry	743
25.4.2. Przekształcenia siatka – kość	744
25.4.3. Hierarchia	745
25.4.4. Dane animacji	746
25.4.5. Klasa M3DLoader	748
25.5. Aplikacja „Skinned Mesh”	749
25.6. Podsumowanie	750
25.7. Ćwiczenia	751
Dodatek A Wprowadzenie do programowania Windows	753
A.1. Zagadnienia podstawowe	754
A.1.1. Zasoby	754
A.1.2. Zdarzenia, kolejka komunikatów, komunikaty i pętla komunikatów	754
A.1.3. Interfejs graficzny	755
A.1.4. Unicode	755
A.2. Prosta aplikacja Windows	755
A.3. Analiza prostej aplikacji Windows	760
A.3.1. Dołączanie plików, zmienne globalne i prototypy	761
A.3.2. Funkcja WinMain	761
A.3.3. Struktura WNDCLASS i jej rejestracja	762
A.3.4. Tworzenie i wyświetlanie okna	763
A.3.5. Pętla komunikatów	765
A.3.6. Procedura okna	766
A.3.7. Funkcja MessageBox	768
A.4. Ulepszona pętla komunikatów	768
A.5. Podsumowanie	769
A.6. Ćwiczenia	770
Dodatek B Język HLSL	771
Typy zmiennych	771
Typy skalarne	771
Typy wektorowe	771
Typy macierzowe	772
Tablice	773
Struktury	774
Słowo kluczowe typedef	774

Modyfikatory zmiennych	774
Rzutowanie typów	775
Słowa zastrzeżone i operatory	775
Słowa kluczowe	775
Operatory	775
Przeływ programu	777
Funkcje	778
Funkcje użytkownika	778
Funkcje wbudowane	779

Dodatek C Wybrane zagadnienia geometrii analitycznej 783

C.1. Promienie, proste i odcinki	783
C.2. Równoległoboki	784
C.3. Trójkąty	785
C.4. Płaszczyzny	786
C.4.1. Płaszczyzny XNA Math	787
C.4.2. Wzajemne położenie punktu i płaszczyzny	787
C.4.3. Obliczanie płaszczyzny	788
C.4.4. Normalizacja płaszczyzny	789
C.4.5. Przekształcanie płaszczyzny	789
C.4.6. Najbliższy punkt na płaszczyźnie względem danego punktu	789
C.4.7. Przecięcie promienia i płaszczyzny	790
C.4.8. Odbijanie wektorów	791
C.4.9. Odbijanie punktów	791
C.4.10. Macierz odbicia	792
C.5. Ćwiczenia	793

Dodatek D Rozwiązania wybranych ćwiczeń 795

Rozdział 1.	795
Rozdział 2.	798
Rozdział 3.	802
Rozdział 5.	810
Rozdział 8.	816
Rozdział 10.	818
Rozdział 11.	819
Rozdział 12.	819
Rozdział 14.	819
Rozdział 15.	820
Rozdział 18.	823
Rozdział 21.	823
Rozdział 24.	825
Dodatek C	832

Bibliografia 837

Skorowidz 841

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).

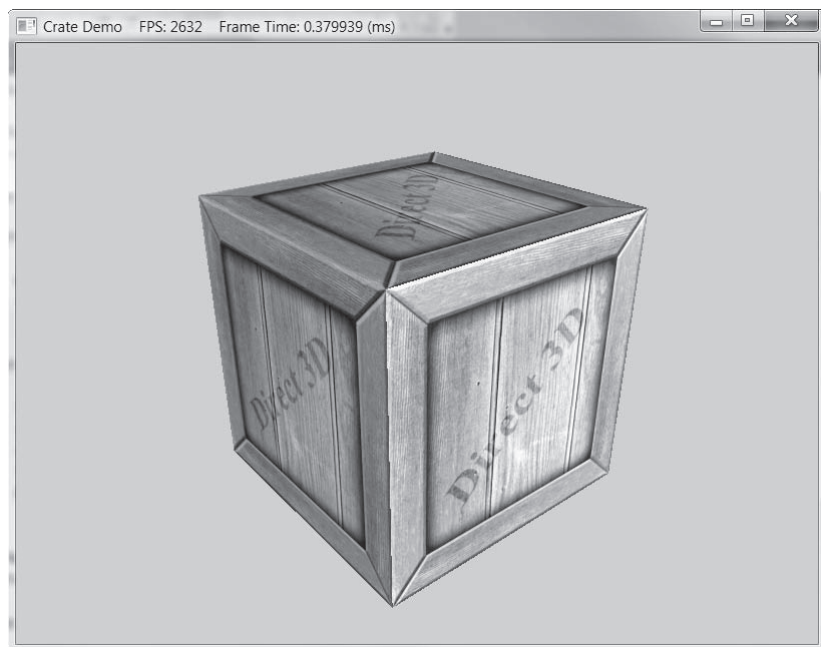
Po przeczytaniu tego rozdziału powinieneś:

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. Sześcián 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* — *przyt. 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 (ID3D11RenderTargetView) oraz widok zasobu shadera (ID3D11ShaderResourceView). Rola widoków zasobów jest dwójaka: 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.

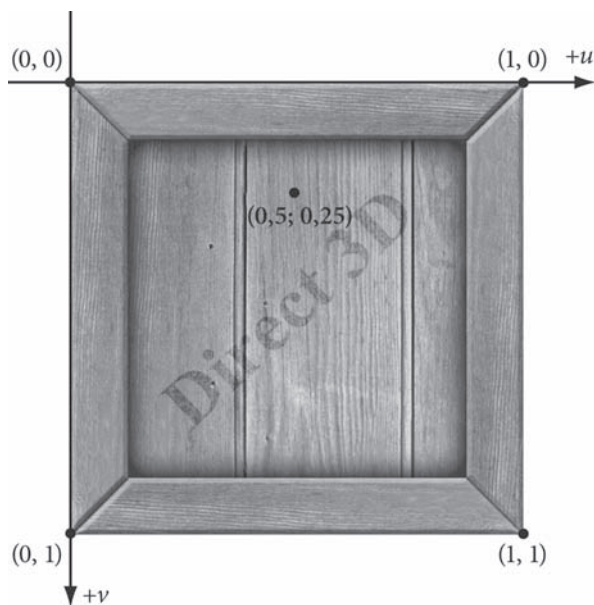
Aby utworzyć widok zasobu, podczas tworzenia zasobu musimy użyć specjalnego znacznika miejsca wiązania. Na przykład jeżeli podczas tworzenia zasobu nie włączono znacznika `D3D11_BIND_SHADER_RESOURCE` (wskazującego, że tekstura powinna zostać powiązana z potokiem jako bufor głębokości/szablonu), utworzenie 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 próbować tekstury i kolorować na tej podstawie piksele.

8.2. WSPÓŁRZĘDNE TEKSTURY

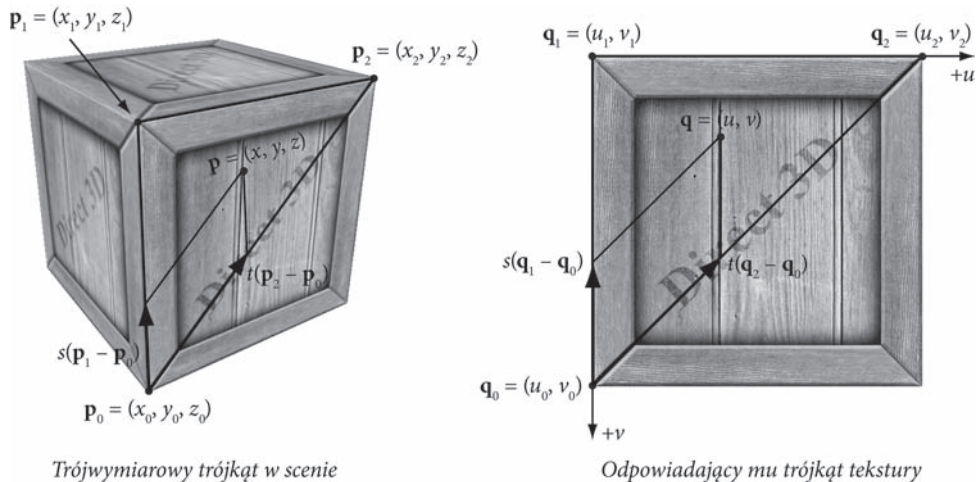
Direct3D korzysta z układu współrzędnych tekstury, w którym u jest poziomą, a v pionową osią względem obrazu. Współrzędne (u, v) , takie, że $0 \leq u, v \leq 1$, identyfikują element tekstury zwany **tekselem**. Zwróć uwagę, że na rysunku 8.2 oś v jest skierowana w dół. Zauważ też, że Direct3D wykorzystuje znormalizowany przedział $[0, 1]$, dzięki czemu otrzymuje zakres niezależny od wymiarów. Na przykład współrzędne $(0,5; 0,5)$ zawsze reprezentują środkowy teksele 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 teksele znajdujący się na jednej czwartej łącznej szerokości i trzech czwartych łącznej wysokości tekstury. Chwilowo założmy, 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 wyznaczenie 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.

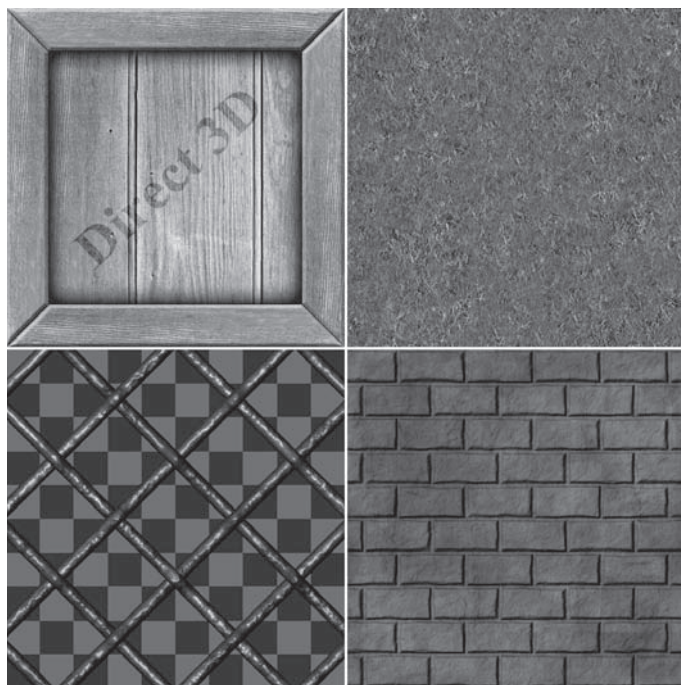
```
struct Basic32
{
    XMFLOAT3 Pos;
    XMFLOAT3 Normal;
    XMFLOAT2 Tex;
};

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 teksturuwaniu 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,
    LPCWSTR 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 wystarczy, 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`).

Uwaga  *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.

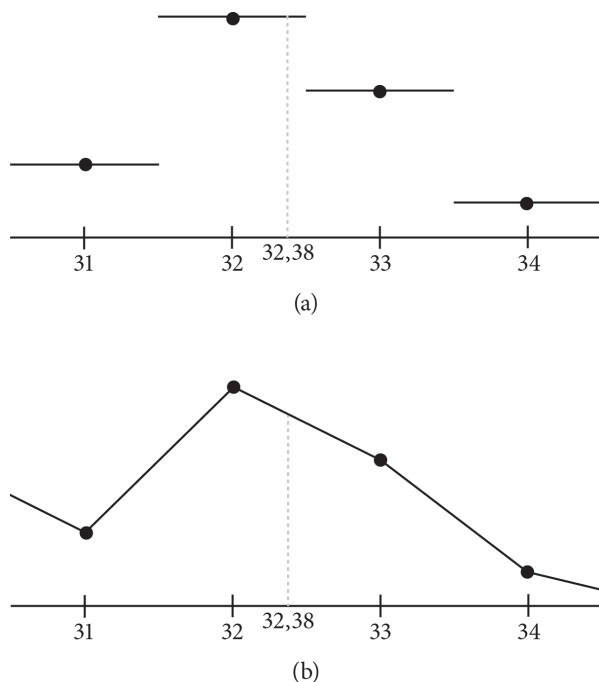
8.4. FILTRY

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 założmy 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 tekselei, 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 tekselela na pozycji: $0,126484375 \cdot 256 = 32,38$. Wartość ta leży oczywiście pomiędzy dwoma tekselemi, potrzebna jest zatem interpolacja.

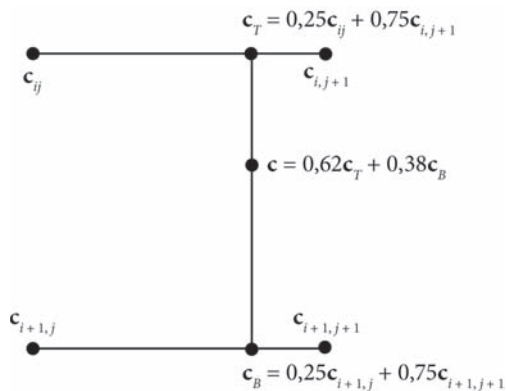


Rysunek 8.5. (a) Na podstawie punktów tekselei możemy zbudować funkcję schodkową, aby przybliżyć wartości pomiędzy tekselemi. Ten sposób nazywamy próbkowaniem metodą najbliższego sąsiada, ponieważ wykorzystuje najbliższy tekselel. (b) Na podstawie punktów tekselei możemy zbudować funkcję odcinkowo-liniową,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

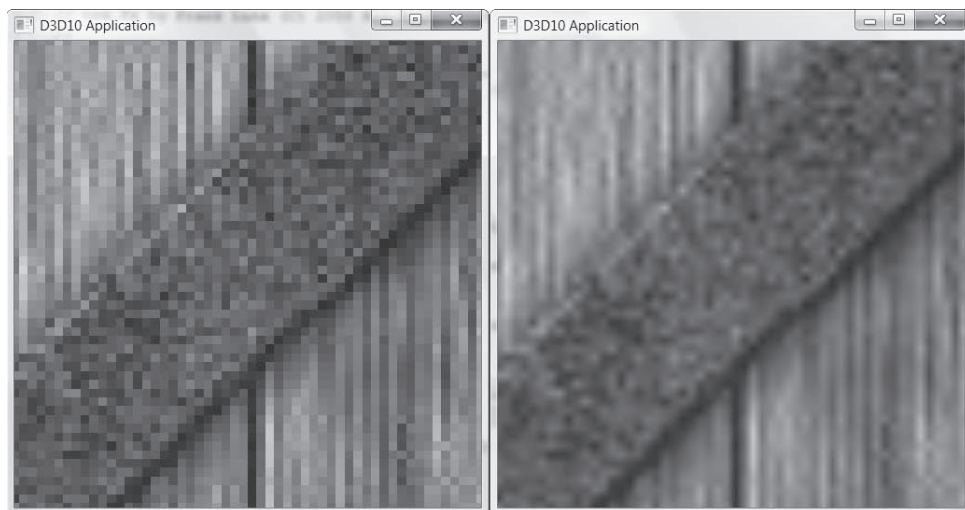
Interpolację na płaszczyźnie dwuwymiarowej nazywamy **interpolacją dwuliniową** (ang. *bilinear*). Pokazano ją na rysunku 8.6. Jeżeli punkt wyznaczony przez współrzędne tekstury znajduje się pomiędzy czterema tekselemi, dokonujemy ich liniowej interpolacji w kierunku u , a następnie w kierunku v .

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 teksele: 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 teksele 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

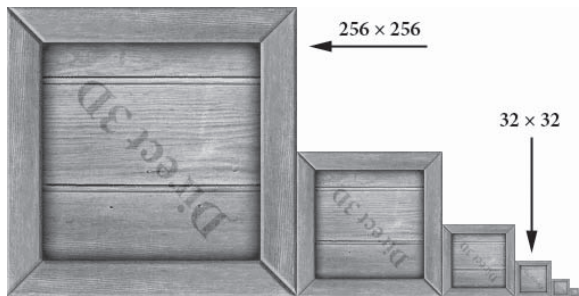
Tekstury widziane z pewnych odległości będą wyglądać wspaniale, ale im bliżej będą oka, tym efekt wizualny może być gorszy. Niektóre gry, aby uniknąć zbyt dużego powiększenia, ograniczają odległość, na jaką wirtualne oko może się zbliżyć do powierzchni. Użycie tekstur o wyższej rozdzielczości może częściowo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

Uwaga

W kontekście teksturuowania stosowanie interpolacji stałej, aby znaleźć wartości tekstury dla współrzędnych tekstury pomiędzy tekselemi, 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 teksele jest większa od liczby pikseli. Rozważmy na przykład sytuację, w której mamy mur pokryty teksturą o wymiarach 256×256 . Kiedy patrząc na niego oko oddala się, mur staje się coraz mniejszy, do momentu gdy pokrywa tylko obszar o rozmiarze 64×64 pikseli. Obszar 256×256 teksele musi więc zostać odwzorowany na obszar 64×64 na ekranie. W takiej sytuacji współrzędne tekstury dla pikseli też raczej nie będą się pokrywać z tekselemi 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 teksele do 64×64 pikseli. Technika mipmapingu 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

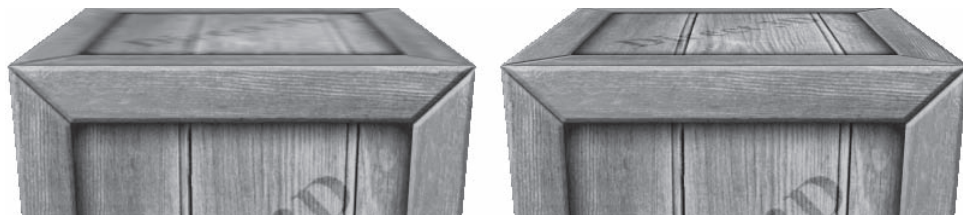
Mipmapy mogą być tworzone bezpośrednio przez grafików lub automatycznie, za pomocą algorytmów filtrujących.

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 `D3DX11CreateTextureFromFile` 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, próbujemy 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`.



Typ HLSL `SamplerState` odzwierciedla interfejs `ID3D11SamplerState`. Stany samplera można też ustawiać na poziomie aplikacji za pomocą metody `ID3DX11EffectSamplerVariable::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 teksteli). 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 prostopadłości, 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(!ID3DApp::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 gUseTexture) : 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(gUseTexture)
    {
        // Próbuj 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 `glUseTexture` 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 niezmiennione. 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)
```

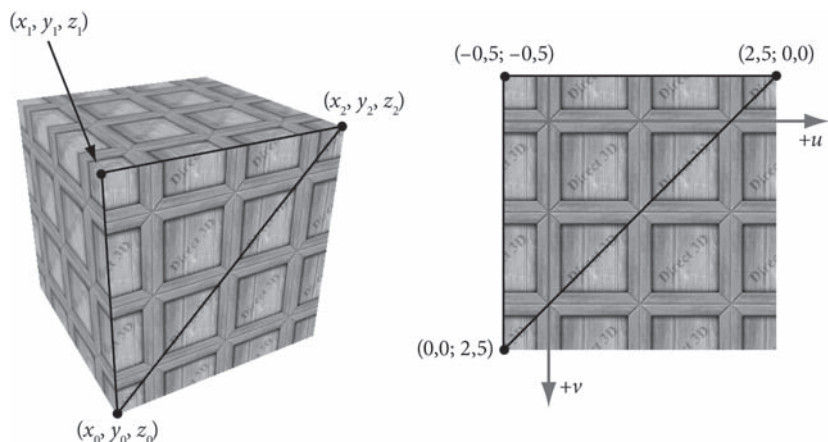
Po wykonaniu mnożenia otrzymany czterowymiarowy wektor jest redukowany do wektora dwuwymiarowego poprzez odrzucenie składowych z i w :

```
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).

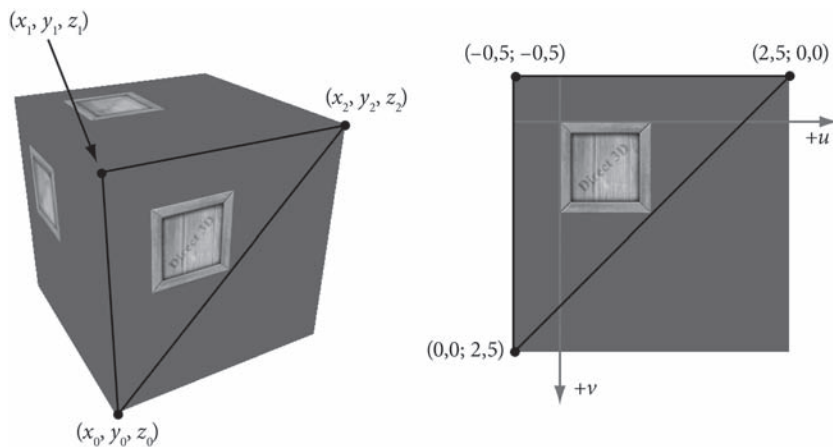


Rysunek 8.10. Tryb adresowania wrap

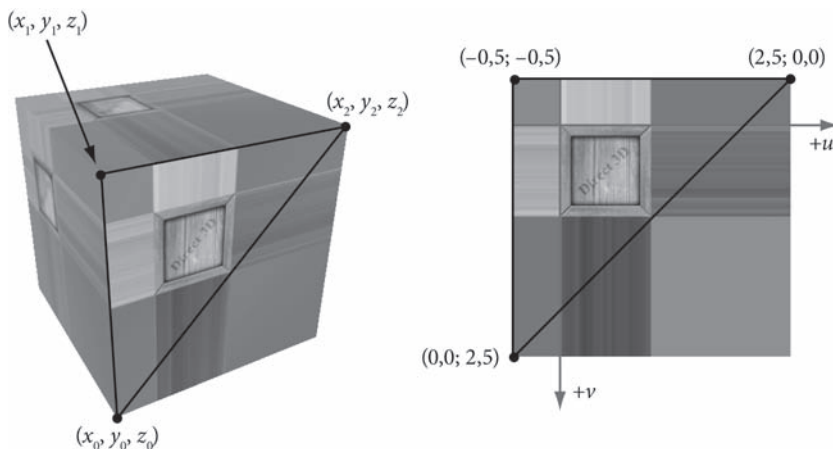
2. *border color* rozszerza funkcję tekstury, odwzorowując współrzędne (u, v) spoza obszaru $[0, 1]^2$ na określony przez programistę kolor (zob. rysunek 8.11).
3. *clamp* rozszerza funkcję tekstury, odwzorowując współrzędne (u, v) spoza obszaru $[0, 1]^2$ na kolor $T(u_0, v_0)$, gdzie (u_0, v_0) to 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.

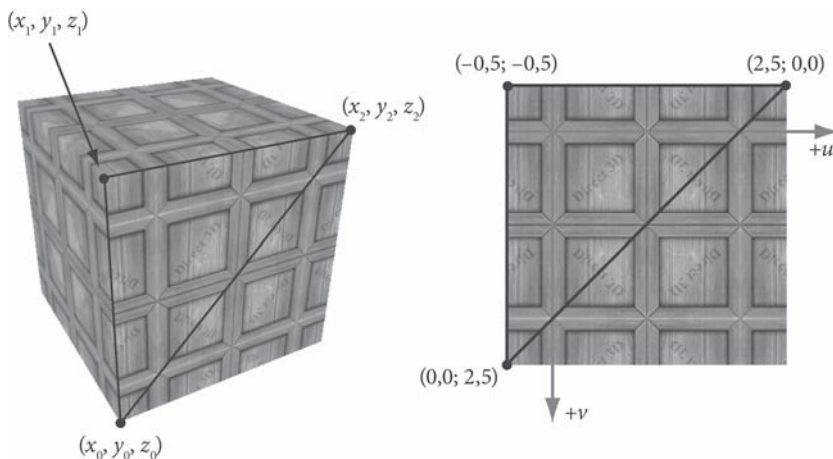
Najczęściej stosowanym trybem adresowania jest prawdopodobnie tryb *wrap*, który pozwala wykorzystać teksturę do pokrycia powierzchni metodą kafelkową. Dzięki temu zyskujemy większą rozdzielczość, bez konieczności dostarczania dodatkowych danych (zawdzięczamy ją powtórzeniom). Przy kafelkowaniu niezwykle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



Rysunek 8.11. Tryb adresowania border color

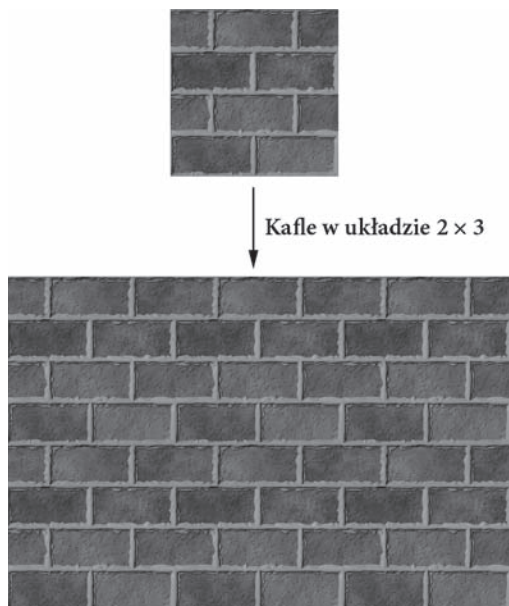


Rysunek 8.12. Tryb adresowania clamp



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;
};
```

Uwaga

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:

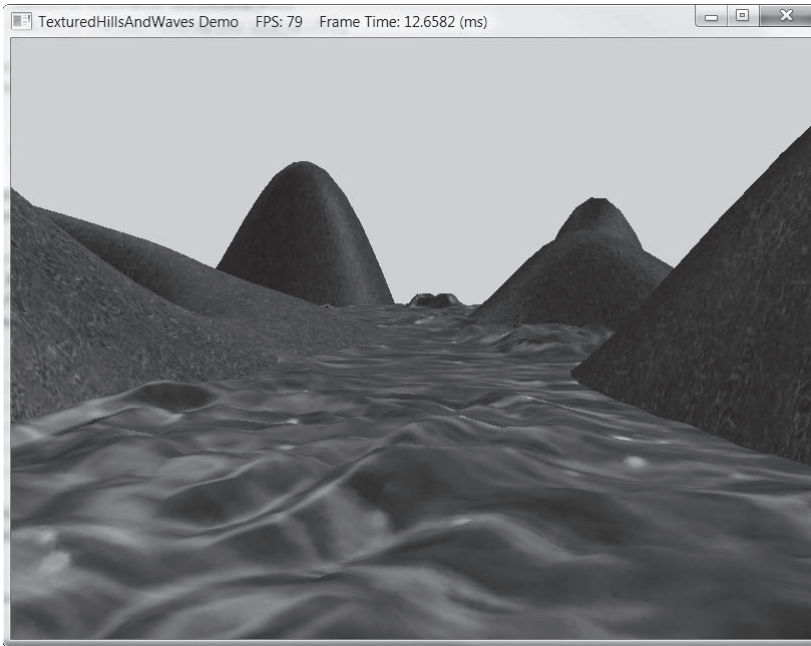
```
// Zmienna bufora stałego
float4x4 gTexMtx;

// 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 współrzedną 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 łądu teksturą trawy. Z racji dużego rozmiaru powierzchni siatki łądu rozciągnięcie tekstury nie byłoby najlepszym rozwiązaniem, jako że zbyt mało teksele pokrywałyby 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 łądu, 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:

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

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

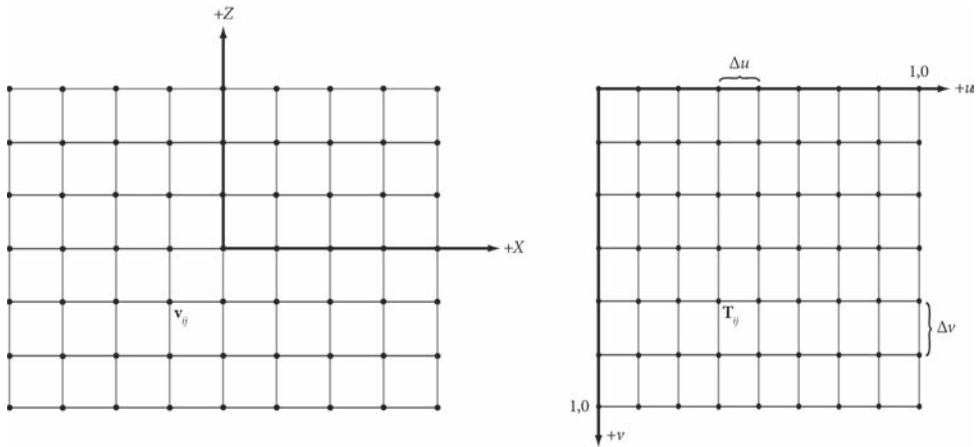
gdzie $\Delta u = \frac{1}{n-1}$ oraz $\Delta v = \frac{1}{m-1}$.

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 łądu 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 łądu 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 4x4, 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_FORMAT_BC1_UNORM`:

Name	Value	Type
[-] texDesc	(Width=512 Height=512 MipLevels=10 ...)	D3D11_TEXTURE2D_DESC
Width	512	unsigned int
Height	512	unsigned int
MipLevels	10	unsigned int
ArraySize	1	unsigned int
Format	DXGI_FORMAT_BC3_UNORM	DXGI_FORMAT
[+] SampleDesc	{Count=1 Quality=0 }	DXGI_SAMPLE_DESC
Usage	D3D11_USAGE_DEFAULT	D3D11_USAGE
BindFlags	8	unsigned int
CPUAccessFlags	0	unsigned int
MiscFlags	0	unsigned int

(a)

Name	Value	Type
[-] texDesc	(Width=512 Height=512 MipLevels=10 ...)	D3D11_TEXTURE2D_DESC
Width	512	unsigned int
Height	512	unsigned int
MipLevels	10	unsigned int
ArraySize	1	unsigned int
Format	DXGI_FORMAT_R8G8B8A8_UNORM	DXGI_FORMAT
[+] SampleDesc	{Count=1 Quality=0 }	DXGI_SAMPLE_DESC
Usage	D3D11_USAGE_DEFAULT	D3D11_USAGE
BindFlags	8	unsigned int
CPUAccessFlags	0	unsigned int
MiscFlags	0	unsigned int

(b)

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 `D3DX11CreateShaderResourceViewFromFile` 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 shaderów obliczeniowych. 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.

Name	Value	Type
[-] texDesc	(Width=512 Height=512 MipLevels=10 ...)	D3D11_TEXTURE2D_DESC
Width	512	unsigned int
Height	512	unsigned int
MipLevels	10	unsigned int
ArraySize	1	unsigned int
Format	DXGI_FORMAT_BC1_UNORM	DXGI_FORMAT
[+] SampleDesc	(Count=1 Quality=0)	DXGI_SAMPLE_DESC
Usage	D3D11_USAGE_DEFAULT	D3D11_USAGE
BindFlags	8	unsigned int
CPUAccessFlags	0	unsigned int
MiscFlags	0	unsigned int

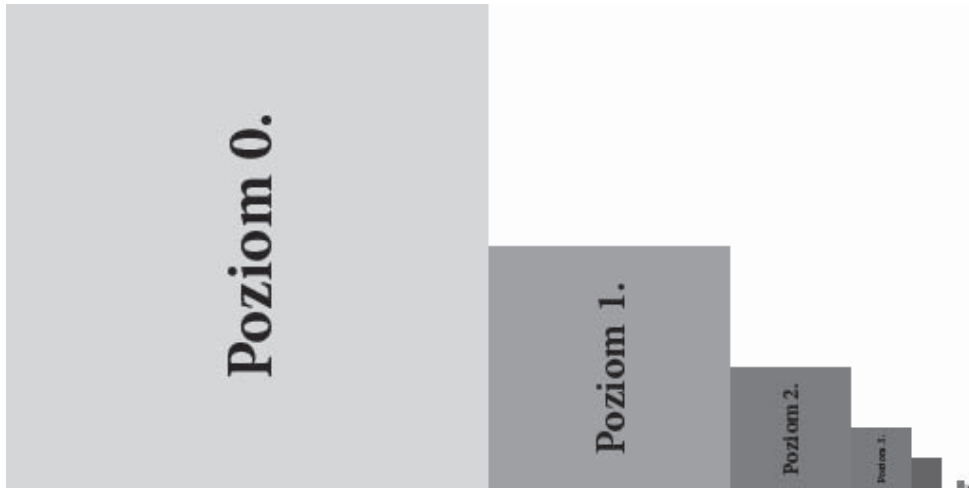
Rysunek 8.18. Tekstura tworzona w formacie DXGI_FORMAT_BC3_UNORM

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 obsługiwane 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.



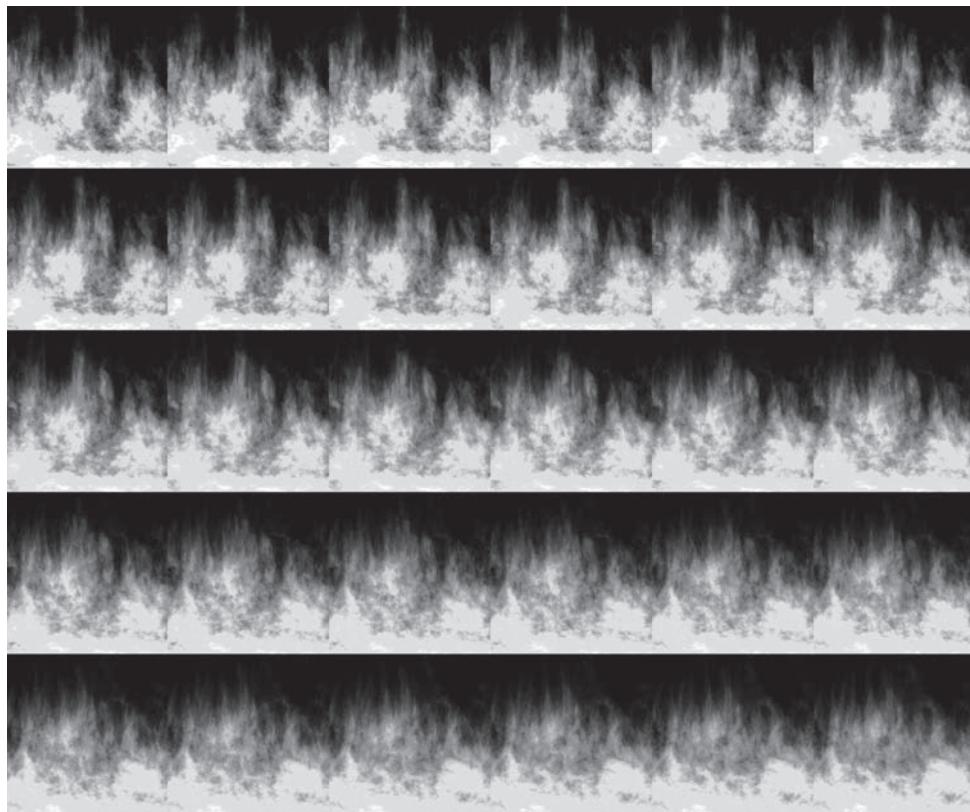
Rysunek 8.19. Samodzielnie skonstruowana sekwencja mipmap z łatwo rozróżnialnymi poziomami

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 tekstele 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 tekstele 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. Zaczynij 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 $\mathbf{p}(s, t) = \mathbf{p}_0 + s(\mathbf{p}_1 - \mathbf{p}_0) + t(\mathbf{p}_2 - \mathbf{p}_0)$, 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 .

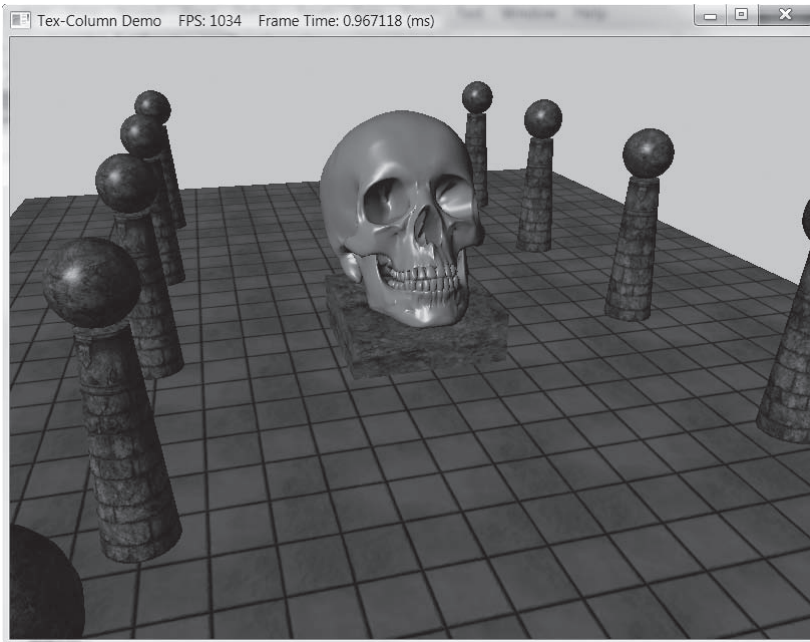


Skorzystaj z równania wektorowego $(u, v) - \mathbf{q}_0 = s(\mathbf{q}_1 - \mathbf{q}_0) + t(\mathbf{q}_2 - \mathbf{q}_0)$.

(b) Wyraż \mathbf{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

ABGR, 273
aktualizacja
 systemu, 615, 633
 tekstury, 666
algebra
 macierzy, 65
 wektorów, 35
algorytm
 obcinania, 192
 rozmywania, 440
algorytmy rzutowania tekstur, 200
aliasing, 122, 645, 650, 667
animacja, 137, 724
 ognia, 343
 postaci, 731
 szkieletu, 737
 tekstury, 338
antialiasing, 122, 128
antialiasing MSAA, 415
API, application programming interface, 21,
 115, 753
aplikacja
 Ambient Occlusion, 671
 Blend, 359
 Blur, 440, 450
 Box, 242
 Crate, 326
 Cube Demo, 549
 Cube Map, 532
 Hills, 248

 Init Direct3D, 151
 Instancing and Culling, 510
 Land Tex, 336
 Lighting, 301
 Lit Skull, 307
 Mesh Viewer, 701
 Mirror, 389
 Picking, 527
 Quaternions, 723
 Shapes, 254
 Skinned Mesh, 749
 Skull, 267
 Tree Billboard, 403
 Waves, 269
 Windows, 755, 760, 769
ARGB, 167
artefakty
 cieni, 645
 renderowania, 360
atlas tekstur, 318, 320

B

baza TBN, 558
bazowa pozycja wierzchołka, 215
biblioteka
 D3DX, 23, 49, 224
 Direct3D 11, 21
 MSDN, 753
 Open Asset Import Library, 702
 XNA Collision, 498, 512, 673
 XNA Math, 33, 50, 60

biblioteki DirectX, 27
 billboard, 606
 billboard, 400, 416
 błąd, 152, 232
 filtra PCF, 662
 obliczeń zmiennoprzecinkowych, 59
 bryła widzenia, 638
 budowa
 bufora głębokości, 542
 macierzy widoku, 489
 mapy cieni, 652
 tekstury sześcienniej, 539
 bufor
 bezpośredni, 433
 dopisywania, 437, 456
 głębokości, 120, 354, 370, 542
 głębokości/szablonu, 637
 indeksów, 212, 270
 instancji, 497
 konsumowany, 437, 456
 przedni, 117
 stały, 218
 szablonowy, 369, 377, 388
 tylny, 117, 377
 ustrukturyzowany, 430, 456
 wierzchołków, 168, 208, 212, 270, 616, 620
 z typem, 432
 buforowanie potrójne, 118
 bufory
 dynamiczne, 269, 271
 stałe, 270

C

cień, 162, 382, 635
 oderwany, 647
 światła punktowych, 385
 światła równoległych, 382
 z przezroczystością, 386
 cieniowanie
 geometrii, 191, 395, 416
 kreskówkowe, 311
 wierzchołków, 175
 COM, Component Object Model, 116, 154
 CPS, counts per second, 137
 czas całkowity, 140
 cząsteczka, 605
 deszczu, 627
 ognia, 623

część wspólna
 ostrosłupa widzenia i prostopadłościanu, 507
 ostrosłupa widzenia i sfery, 506
 czworokątne łaty Béziera, 471

D

dane animacji, 746
 DDS, Direct Draw Surface, 324
 deszcz, 627
 Direct3D, 115
 Direct3D 11, 21
 długość
 jednostkowa, 41
 odcinka, 40
 dodawanie
 kolorów, 352
 wektorów, 50
 dokumentacja
 DirectX, 24
 MSDN, 138
 SDK, 23
 dołączanie
 bibliotek, 28
 plików, 761
 dopełnienie algebraiczne elementu, 74
 dostęp do
 shadera wierzchołków, 207
 tekstury, 456
 wierzchołków, 270
 dostępność, accessibility, 671
 duże jądra PCF, 661, 663, 665
 DXGI, 131
 dynamiczne
 bufory wierzchołków, 266
 tekstury sześcienniej, 539, 545
 dyrektywa #include, 150
 działania na
 kolorach, 165
 wektorach, 38, 60
 dzielenie perspektywiczne, 188, 384

E

efekt
 bazowy, 328
 odbłyску, 289
 schodów, 122

efekty
 cząsteczkowe, 618
 niestandardowe, 317
 renderowania, 224

emitery, 620

etap
 cieniowania geometrii, 191, 198
 cieniowania pikseli, 197
 cieniowania wierzchołków, 175, 198
 łączenia wyników, 197
 obcinania, 198
 rasteryzacji, 193, 198
 teselatora, 465
 zbierania danych, 198

etapy
 potoku renderowania, 169
 teselacji, 190, 198, 459

F

filtrowanie
 anizotropowe, 324
 liniowe, 322, 323
 PCF, 648, 661, 664
 punktowe, 322, 323
 wygładzające, 579

filtry, 320

filtry porównujące, 651

format
 DDS, 324
 DXT5, 356
 indeksów, 213
 M3D
 dane animacji, 746
 indeksy trójkątów, 697
 materiały, 695
 nagłówek, 694
 podzbiory, 696
 siatka – kość, 744
 tablica hierarchii, 745
 wierzchołek skóry, 743
 wierzchołki, 697
 RAW, 578
 UNORM, 654
 wejścia, 204

formaty
 beztypowe, 117
 kompresji tekstur, 338
 typów wyliczeniowych, 116
 wierzchołków, 203

FPS, frames per second, 147, 155

framework efektów, 224, 233

funkcja
 CreateDXGIFactory, 130
 CreateWindow, 753, 763
 D3DX11CreateEffectFromMemory, 233
 D3DX11CreateShaderResourceViewFromFile,
 534
 D3DX11CreateTextureFromFile, 318
 frac, 650
 Gaussa, 442
 MessageBox, 768
 położenia, 608
 porównująca, 651
 prędkości, 607
 saturate, 361
 ściśle rosnąca, 188
 Tick, 139
 WinMain, 761
 XMMatrixPerspectiveFovLH, 189
 XMPlaneFromPointNormal, 788

funkcje
 kwaternionowe, 723
 ładujące, loading functions, 51
 macierzowe, 78
 pochodne, 662
 przekształcające, 105
 ustawiające, 55
 użytkownika, 778
 wbudowane, 779
 wektorowe, 56
 zapisujące, storage functions, 51

G

generowanie
 indeksów siatki, 251
 koloru, 252
 krajobrazu, 578
 losowej tekstury, 610
 losowych próbek, 676
 mapy SSAO, 678
 mapy wysokości, 565
 punktów zasłaniających, 678
 shaderów, 233, 235
 siatki, 306
 wierzchołków siatki, 248, 249
 współrzędnych tekstury, 336

geometria analityczna, 783

geometria
 łąt, 476
 siatki, 699
 głębokość
 piksela, 645
 sceny, 635
 GPU, graphics processing unit, 21
 grupa
 efektów, 225
 wątków, 422, 455
 GUI, Graphical User Interface, 755

H

hierarchia siatki, 752
 hierarchie układów odniesienia, 731
 HLSL, high level shading language, 215

I

identyfikator
 grupy, 436
 indeksu wątku, 437
 wątku w grupie, 436
 wierzchołka, 417
 wykonania wątku, 436
 identyfikatory prymitywów, 420
 iloczyn
 skalarny, 40, 42, 60, 281
 wektorowy, 46, 60, 90
 implementacja
 billboardingu, 402
 cieni płaskich, 382
 czasu całkowitego, 141
 rozmycia, 445
 światel kierunkowych, 298
 światel punktowych, 299
 światel reflektorowych, 300
 indeksowanie tekstur, 428
 indeksy, 173, 212, 214
 siatki, 251
 tablicy tekstur, 414
 trójkątów, 697
 inicjalizacja
 bufora wierzchołków, 620
 Direct3D, 125
 instancjonowanie, 177, 511
 instancjonowanie sprzętowe, 493
 instrukcje SSE2, 50

intensywność koloru, 355
 interfejs
 graficzny użytkownika, GUI, 755
 ID3D11BlendState, 221
 ID3D11DepthStencilState, 221, 369, 389
 ID3D11Device, 126, 154
 ID3D11DeviceContext, 126, 154
 ID3D11RasterizerState, 221
 ID3D11Texture2D, 313
 IDXGI11Effect, 229
 IDXGISwapChain, 117
 programowania aplikacji, API, 21, 115, 753
 interpolacja
 atrybutów wierzchołków, 195
 dwuliniowa, 321
 klatek kluczowych, 726
 kwaternionów, 718
 liniowa, 322
 normalnych, 279
 z korekcją perspektywiczną, 195

J

jądra PCF, 661
 jądro rozmycia, 441
 jednorodna przestrzeń obcinająca, 189
 jednostka NDC, 186
 jednostka urojona, 704
 język HLSL, 215, 771–781

K

kafelkowanie, 337
 kamera, 484, 543
 kamera pierwszej osoby, 483
 kanał alfa, 117, 353, 358, 359
 katalog Common, 28, 309
 katalog projektu, 31
 kąt
 między kwaternionami, 718
 między wektorami, 43, 810
 pola widzenia, 183, 184
 kierunek kamery, 180
 klasa
 BasicModel, 700, 702
 Camera, 485
 D3DApp, 144
 Effects, 309
 GameTimer, 138, 142

- GeometryGenerator, 254
 - M3DLoader, 748
 - ParticleSystem, 618
 - SkinnedModel, 749
 - XMMATRIX, 77, 78
 - XMVECTOR, 61
 - klatka kluczowa, key frame, 724
 - kod
 - asemblera, 232, 237
 - błędu, 153
 - cienia, 387
 - cieniujący geometrię, 416
 - do obliczeń powierzchni Béziera, 475
 - shadera, 217
 - shadera obliczeniowego, 425
 - wirtualnego klawisza, 767
 - kolejka komunikatów, 754, 769
 - kolejność
 - nawijania, 381
 - odbicia, 381
 - składowych, 772
 - kolor, 163, 197
 - 128-bitowy, 165
 - 32-bitowy, 166
 - materiału odbijającego, 286, 288
 - materiału otoczenia, 288
 - materiału rozpraszającego, 288
 - mgły, 361, 366
 - kombinacja liniowa, 69, 86
 - kompilacja
 - efektu, 231
 - shaderów, 226, 228
 - kompresja tekstur, 338, 341
 - komunikaty, 754
 - błędu, 127, 130, 315
 - WM_ACTIVATE, 149
 - WM_CLOSE, 770
 - WM_EXITSIZEMOVE, 149
 - WM_SIZE, 149
 - kontekst, 126
 - natychmiastowy, 128
 - odroczone, 128
 - konwersja współrzędnych wierzchołków, 180
 - kopiowanie wyników shadera, 433
 - koszt kopiowania, 435
 - krzywe Béziera, 472, 479
 - kwaternion, 707, 728
 - czysty, 709, 728
 - jednostkowy, 710, 728
 - norma, 709
 - normalizacja, 721
 - odwrotny, 710
 - postać biegunowa, 711
 - sprężenie, 709
 - własności, 708
 - złożenie, 717
 - kwaternionowy operator obrotu, 714, 716, 728
- ## L
- liczba zespolona
 - część rzeczywista, 704
 - część urojona, 704
 - interpretacja geometryczna, 705
 - postać biegunowa, 705
 - licznik wydajności, 137, 138, 155
 - linia rzutowana, 196
 - liniowa interpolacja, 196
 - lista
 - linii, 170
 - łat, 173
 - punktów, 170
 - trójkątów, 172
 - lokalna oś z kamery, 181
 - lokalny układ współrzędnych, 733, 751
 - losowe wektory, 677
 - losowość systemów cząsteczek, 609
 - lustro płaskie, 376
- ## Ł
- ładowanie
 - danych animacji, 743
 - danych skóry, 748
 - geometrii, 266
 - pliku RAW, 578
 - tablic tekstur, 410
 - tekstur sześciennych, 534
 - łańcuch wymiany, 117, 129, 130
 - łata, 591
 - łaty Béziera, 471
 - łączenie
 - buforów wierzchołków, 214
 - efektów, 229
 - wyników, 135
 - łączność
 - dodawania, 67
 - mnożenia macierzy, 106

M

- macierz, 65
 - cienia kierunkowego, 384
 - cienia uogólniona, 386
 - dołączona, 74, 81
 - dopełnień algebraicznych, 800
 - dziecko – rodzic, 734
 - gWorldViewProj, 218
 - jednostkowa, 70, 80, 281
 - obrotu, 91, 95, 106, 716, 826
 - odbicia, 792
 - odwrotna, 74, 81, 102
 - ortogonalna, 90, 106
 - osobliwa, 74
 - przejścia, 102
 - przekształcenia afinicznego, 95, 813
 - przekształcenia liniowego, 87
 - przesunięcia, 94, 106, 201
 - rzutowania, 187
 - rzutowania ortograficznego, 640
 - rzutowania perspektywicznego, 189, 199
 - skalowania, 88, 95, 106, 804
 - świata, 179, 262
 - świata billboardu, 606
 - transponowana, 69, 80, 90, 281
 - widoku, 483, 489, 492, 811
 - zamiany współrzędnych, 102, 103
 - zredukowana, 72
- macierze XNA, 76
- makro ZeroMemory, 223
- mapa
 - cieni, 635, 644, 646
 - kaskadowa cieni, 667
 - mieszania, 596
 - normalnych, 552, 565
 - okluzji, 686
 - okluzji otoczenia, 673, 686
 - przemieszczeń, 564
 - SSAO, 675, 678
 - wysokości, 563, 575, 583, 601
- mapowanie
 - cieni, 667
 - normalnych, 116, 551, 559
 - otoczenia, 531
 - przemieszczeń, 457, 551, 563, 571, 589
 - sześcienne, 529
- materiał, 276, 326
- metoda
 - Append, 397
 - Apply, 230
 - CreateBox, 326
 - CreateDepthStencilView, 135
 - CreateRenderTargetView, 132
 - CreateInputLayout, 206
 - CreateShaderResourceView, 318
 - CreateTexture2D, 135
 - Dispatch, 455
 - Draw, 263
 - DrawScene, 147
 - GetPassByIndex, 230
 - GetVariableByName, 229
 - IASetVertexBuffers, 211
 - Init, 146
 - MsgProc, 147
 - OnResize, 146
 - ReleaseCOM, 132
 - RSSetViewports, 136
 - SampleCmpLevelZero, 651
 - SetLens, 487
 - Update/Draw, 621
 - UpdateScene, 147
- metody
 - ładujące, 51
 - szacunkowe, 58
 - zapisujące, 51
- mgła, 360, 362, 366
- mierzenie czasu, 143
- mieszanie
 - addytywne, 354, 612, 633
 - bez zapisu koloru, 351
 - kolorów, blending, 345, 611
 - multiplikatywne, 352, 354
 - składowych, 366
 - subtraktywne, 352, 354
- migotanie obrazu, 117, 154
- mipmapa, 323
- mnożenie
 - kolorów, 352
 - kwaternionów, 708, 712, 728
 - macierzy, 67, 80, 97, 102, 281
 - macierzy przez wektor, 68
 - składowych, 165
- model oświetlenia
 - globalny, 277
 - lokalny, 277
- model RGB, 164

modelowanie
 fal morskich, 572
 odbić, 537
 światła, 640
 modulacja, 165
 moduł
 kwaternionu, 728
 liczby zespolonej, 705
 modyfikator inout, 397
 modyfikatory zmiennych, 774
 możliwości sprzętu, feature levels, 124
 MSA, 128
 multiteksturowanie, 342

N

nadpróbkowanie, 122, 123
 nakładanie tekstury, texture mapping, 313
 narzędzia programistyczne, 23
 narzędzie
 DirectX Texture Tool, 324, 340, 533
 DXTex, 356
 natężenie światła, 292, 293
 NDC, normalized device coordinates, 185
 norma kwaternionu, 709
 normalizacja
 płaszczyzny, 789
 wektora, 41, 60
 wektora normalnego, 789
 normalna, 277
 normalne wierzchołków, 278, 309

O

obciążanie głębokości, 645, 648
 obcinanie, 191
 obcinanie pikseli, 356
 obiekt SV_PrimitiveID, 408
 obiekty
 stanu mieszania, 351
 typu billboard, 401
 obliczanie
 indeksu podzasołu, 415
 mgły, 364
 normalnych, 306
 okluzji, 673
 okluzji otoczenia, 689
 oświetlenia, 279
 płaszczyzny, 788
 powierzchni Béziera, 475

przekształcenia finalnego, 738
 wektorów normalnych, 279
 współczynników teselacji, 568
 obliczenia zmiennoprzecinkowe, 59
 obrót, 89
 kwaternionu, 714
 liczby zespolonej, 706
 obsługa
 danych siatek-skór, 743
 komunikatów, 148
 PCF, 651
 tablic, 298
 tekstury, 318
 teselacji, 582
 odbicie
 punktu, 791
 wektora, 791, 793
 zwierciadlane, 285
 odbłysek, 552, 553
 odcinek, 784
 odejmowanie
 kolorów, 352
 punktów, 92
 oderwanie cienia, 646
 odległość punktu od płaszczyzny, 787, 789
 odstępy czasowe, 139
 odwrotność
 kwaternionu, 728
 macierzy, 75
 macierzy rzutowania, 200
 ogień, 623
 ograniczenia ostrosłupa widzenia, 192
 okluzja otoczenia, ambient occlusion, 669, 689
 okluzja otoczenia przestrzeni ekranu, 674, 689
 okno
 ChooseColor, 164
 rzutu, 183
 widoku, viewport, 135, 542
 określanie geometrii łąt, 476
 operacje
 mieszania, 347
 SIMD, 455
 operator, 775
 binarny, 347
 obrotu, 712
 operatory wektorowe, 54
 opis
 formatu wejścia, 204
 łańcucha wymiany, 129

ortogonalizacja
 2D, 44
 3D, 45
 Grama-Schmidta, 45
 z iloczynem wektorowym, 47
 osnowy, warps, 423
 ostrosłup, 811
 ostrosłup widzenia, 182, 191, 200, 487, 506
 oś świata, 182
 oświetlenie, 275

P

pakiet SDK, 23
 paleta macierzy, 751
 paleta macierzy kości, 740
 pamięć współdzielona, 439
 parametr
 połyskliwości, 286, 288
 s, 361, 366
 zasięgu, 293
 parametry wygaszania, 292, 293
 PCF, percentage closer filtering, 649, 661, 664
 perpendicular, prostopadły, 43
 perspektywa
 atmosferyczna, 360
 liniowa, 160
 peter-panning, 646
 pętla komunikatów, 754, 765, 768
 piksel, 220
 piksele
 docelowe, 345
 źródłowe, 345, 366
 plik
 Basic.fx, 328
 d3dApp.cpp, 143
 d3dUtil.h, 143, 152
 LightHelper.fx, 295, 301
 LightHelper.h, 294
 TreeSprite.fx, 403
 WinNT.h, 755
 xnacollision.h, 513
 pliki
 .cpp, 31
 .fbx, 693
 .fx, 215
 .fxo, 232
 .h, 31
 .m3d, 694

.md3, 693
 .sln, 26
 .vcxproj, 26
 .x, 693
 DDS, 340
 efektów, 215, 224
 RAW, 578
 płaszczyzna, 786
 płaszczyzny XNA Math, 787
 pochodne cząstkowe, 663, 817
 podpróbkowanie, 122
 podstawa walca, 258
 podwójne mieszanie, 386
 podzasyby tekstury, 414
 podział trójkąta, 398
 połączenie
 linii, 170
 trójkątów, 171
 połyskliwość, 286
 pomiar czasu, 137
 pomniejszanie, 323
 poruszanie kamerą, 492
 potok, 154
 potok renderowania, 159, 168, 198
 powiązanie widoków, 135
 powierzchnia boczna walca, 256
 powierzchnie Béziera, 474, 479
 powiększanie, 320, 322
 poziom
 mip, mip slice, 414
 mipmap, 414
 tablicy, array slice, 414
 półprzestrzeń, 786
 prawo
 Lamberta, 283
 Snelliusa, 548
 prędkość chwilowa cząsteczki, 607
 procedura okna, 754, 766, 769
 procesory
 główne, CPU, 21, 421
 graficzne, GPU, 421, 435
 program
 Bryce, 577
 Dark Tree, 577
 shadera obliczeniowego, 449
 Terragen, 531, 577, 578
 programowanie
 GPGPU, 421, 435
 shaderów geometrii, 396

- shaderów obliczeniowych, 454
- sterowane zdarzeniami, 756
- projekt
 - DirectX, 27
 - Win32, 27
- promień, 784
 - rozmycia, 441, 450
 - wskazujący, 515, 520, 527
- prosta, 783
- prostopadłościan, 498–501, 513
 - otaczający, 501
- prototypy, 761
- próbkowanie
 - 4-krotne, 128
 - głębokości, 646
 - mapy cieni, 649
 - tablicy tekstur, 410
 - tekstur, 324, 428
 - wielokrotne, 122, 130
- prymitywy, 395
- prymitywy z sąsiedztwem, 172
- przebieg, pass, 224, 271
 - okluzji otoczenia, 675
 - rozmycia, 683
- przeciążanie operatorów, 54, 57, 61
- przecięcie
 - promienia i płaszczyzny, 790
 - promienia i trójkąta, 672
 - prostopadłościanu, 522
 - prostopadłościanu i płaszczyzny, 595
 - sfery, 523
 - siatki, 521
 - trójkąta, 524
- przecinanie, 527
- przekazywanie argumentów, 52
- przekształcanie
 - kamery, 488
 - płaszczyzn, 789
 - tekstur, 335
- przekształcenia
 - afiniczne, 91, 92
 - aktywne, 105
 - liniowe, 85, 802
 - ortograficzne, 639
 - tożsamościowe, 93
- przekształcenie
 - bryły sztywnej, 95
 - ekranu, 517
 - finalne, 736, 738
 - kość – szkielet, 736, 751
 - okna widoku, 193
 - położenia wierzchołka, 216
 - przesunięcia, 805
 - punktu, 94–97
 - siatki – kość, 736, 751
 - skalowania, 87, 804
 - sześcianu, 104
 - ścianania, 201
 - wektorów normalnych, 280
 - widoku, 483
 - zamiany współrzędnych, 98, 103
- przełączenie, presenting, 117
- przełączanie buforów wierzchołków, 617
- przemienność dodawania, 67
- przepływ
 - programu, 777
 - prymitywów, 614
- przepustowość, 422
- przesłanie obiektów, 160
- przestrzeń
 - jednorodna, 182, 226
 - lokalna, 175, 226
 - łączenia, bind space, 735
 - NDC, 641
 - styczna, 555, 558
 - świata, 175, 226
 - tekstury, 555
 - widoku, 179, 226
- przesunięcie, 93, 95, 805
- przesyłanie strumieniowe, 495
- przetwarzanie równoległe, 455
- przezroczystość, 353
- przyspieszenie chwilowe cząsteczek, 607
- pseudoiloczyn wektorowy, 47
- punkt, 48
- punkt ekranu, 518, 824
- punkty tekseli, 321, 322

R

- rasteryzator, 647
- reguła kciuka
 - lewej ręki, 90
 - prawy ręki, 46
- rejstry SIMD, 50
- rekonstrukcja położenia, 675
- relacja rodzic – dziecko, 750

renderowanie, 121, 221
 do tekstury, 443, 666
 głębokości, 635, 674
 kolorowego prostopadłościanu, 242
 lustra, 378
 mapy cieni, 661
 normalnych, 674
 odroczone, 308
 sceny do mapy cieni, 648
 terenu, 575
 reprezentacja przekształceń, 106
 RGB, 164, 276
 rodzaje
 światła, 288
 źródeł światła, 290
 rozdzielność mnożenia, 67
 rozmycie, 440, 445, 683, 686
 rozmycie gaussowskie, 441, 443
 równanie
 mieszania, 346, 352, 365, 612
 parametryczne trójkąta, 785
 płaszczyzny, 786
 promienia, 790
 równoległoboku, 784
 wektorowe prostej, 783
 wektorowe trójkąta, 785
 równoległobok, 784
 różnica
 macierzy, 67
 między punktami, 49
 wektorów, 40
 różnice centralne, 590
 ruch cząsteczek, 607
 rysowanie, 203
 automatyczne, 617
 danych instancji, 496
 do tekstury sześciennnej, 544
 obiektów, 263
 sceny, 379
 systemu, 615, 633
 za pomocą efektów, 230
 rzut, 183
 ortogonalny, 43
 ortograficzny, 638, 643, 666, 824
 rzutowanie, 182
 tekstur, 640, 666
 typów, 775
 wierzchołków, 185

S

sampler porównania, 651
 scena, 548
 sekwencja ruchu, animation clip, 737
 sfera, 506
 geodezyjna, 259, 260
 otaczająca, 503
 shader
 dziedziny, 466, 479, 569
 geometrii, 191, 217, 269, 395, 416
 geometrii dla wyjścia strumieniowego, 613
 obliczeniowy, 421, 422, 457
 pikseli, 219, 270, 346
 powłoki, 461, 464, 478, 567
 powłoki punktów kontrolnych, 463
 wierzchołków, 175, 215, 269, 461, 494, 566
 wierzchołków terenu, 586
 siatka, 248–251, 337, 693–702
 sfery, 259
 trójkątów, 162, 197
 walca, 255
 terenu, 602
 siatki-skóry, 735
 SIMD, Single Instruction, Multiple Data, 50, 61
 skalowanie, 87, 89, 804
 skalowanie okluzji, 679
 składanie przekształceń, 97
 składowa alfa, 197
 skóra, 735
 słowa kluczowe w HLSL, 775
 słowo kluczowe typedef, 774
 sprzężenie
 kwaternionu, 709
 zespolone, 704
 SSAO, 675, 687, 689
 SSE2, Streaming SIMD Extensions 2, 50, 79
 stałe wektorowe, 53
 stały shader powłoki, 461
 stan
 głębokości/szablonu, 372, 375, 388
 głębokości/szablonu lustra, 378
 mieszania, 348, 350
 rasteryzatora, 647
 stany renderowania, 221, 270
 statystyki klatek, 147
 stopień
 swobody, 785
 teselacji, 463

stożek odbicia, 285
 strategia N lat, 464
 stronicowanie, page flipping, 117, 343
 struktura, 774
 D3D11_BLEND_DESC, 348
 D3D11_BUFFER_DESC, 208
 D3D11_INPUT_ELEMENT_DESC, 204, 495
 D3D11_MAPPED_SUBRESOURCE, 268
 D3D11_RASTERIZER_DESC, 222
 D3D11_RENDER_TARGET_BLEND_DESC,
 349
 D3D11_SUBRESOURCE_DATA, 209
 D3D11_TEXTURE2D_DESC, 132
 D3D11_VIEWPORT, 517
 Vertex, 211
 wierzchołka, 403
 WNDCLASS, 762
 struktury
 C++, 297
 HLSL, 296
 oświetlenia, 294
 strumień wyjściowy, 633
 suma
 macierzy, 67
 punktu i wektora, 49
 wektorów, 40
 sygnatura wejściowa, 206
 sygnatury shadera geometrii, 397
 symulowanie lustra, 376
 synchronizacja, 440
 system cząsteczek, particle system, 605, 632
 szablonowanie, 369, 370
 szacowanie
 okluzji, 670
 wektora stycznego, 590
 szkielet, 735
 szum, 683

Ś

ścieżka do DirectX SDK, 28
 ścieżki wyszukiwania, 30
 ślad macierzy, 716
 śledzenie promieni, ray casting, 670
 światło, 276
 kluczowe, 308
 odbite, 277, 285, 287
 otoczenia, 285, 288, 310
 punktowe, 291, 305

reflektorowe, 293
 rozproszone, 283, 288
 równoległe, 291, 309
 słoneczne, 305
 tylne, 308
 wypełniające, 308

T

tablica tekstur, 414
 tablice, 773
 tablice tekstur, 409, 417
 technika, technique, 224
 alpha-to-coverage, 416, 417
 PCF, 651
 renderowania, 271
 SSAO, 675, 687
 technologia COM, 116
 teksel, 316
 tekstura, 116, 154, 313, 326, 341, 597
 clamp, 451
 dwuwymiarowa, 313
 jednowymiarowa, 314
 mapy cieni, 657
 otoczenia, 531, 532
 sześcienna, 534, 539, 547
 trójwymiarowa, 314
 wyjściowa, 426
 teksturowanie, 596
 teksturowanie nieba, 535
 teselacja, 190, 459, 478, 587
 czworokąta, 468
 czworokątnej łaty, 465
 sprzętowa, 478
 terenu, 583
 trójkątnej łaty, 462
 test
 głębokości, 392
 nożyc, 274
 porównania mapy cieni, 651
 przecięcia siatki, 521
 szablonu, 371, 388
 zacienienia, 657, 662
 zasłaniania, 678
 topologia prymitywów, 170
 trajektoria
 cząsteczki, 633
 lotu kuli, 608
 transformacja, *Patrz* przekształcenie

transpozycja macierzy, 69, 81
 trójkąt, 785
 trójkąty sąsiednie, 172
 trójwymiarowość, 160
 tryb

- adresowania, 332
- border color, 333
- clamp, 333
- mirror, 333
- wrap, 332

 chodzenia, 601
 fractional_even, 603
 pełnoekranowy, 150
 tworzenie

- bufora, 132
- indeksów, 212
- instancji, 497
- wierzchołków, 211, 616

 łańcucha wymiany, 130
 macierzy, 773
 mapy wysokości, 577
 mipmap, 323
 obiektu stanu mieszania, 416
 okna, 763
 projektu Win32, 26
 siatki, 255, 259
 tekstur, 577
 tekstury, 327
 tekstury sześcienniej, 533
 urządzenia, 126
 widoku celu renderowania, 131
 typ

- D3D_FEATURE_LEVEL, 124
- prymitywu, 565
- Texture2DArray, 409
- XMVECTOR, 54
- XMVECTORF32, 53

 typy

- macierzowe, 76, 772
- prymitywów teselacji, 460
- skalarne, 771
- strumieniowe, 397
- wektorowe, 50, 771

U

UAV, unordered access view, 426
 uchwyty okna, 761

układ współrzędnych, 36

- billboardu, 401, 606
- kamery, 181, 484, 491
- leworęczny, 38
- praworęczny, 38
- tekstury, 316

 Unicode, 755
 urządzenie odniesienia, reference device, 126
 ustawianie

- kamery, 543
- tekstury, 327

 ustawienia

- głębi, 372
- okna widoku, 135
- szablonu, 373

 usuwanie lat, 591

- niewidocznych powierzchni, 194
- powierzchni, 508

 uśrednianie

- normalnych wierzchołków, 279
- pikseli, 581
- wierzchołków, 740, 742, 751

 użycie klasy kamery, 490

V

Visual Studio 2010, 26

W

wagi rozmycia gaussowskiego, 443
 walec, 256
 wariancja, 667
 wartość

- maski, 371
- odniesienia szablonu, 371
- wektora, 41
- wektora XMVECTOR, 55, 486

 wątek, 422
 wątek renderowania, 128
 wczesna kompilacja efektu, 231
 wejściowe tekstury, 426
 wektor, 35, 60

- jednostkowy ortogonalny, 277
- normalny, 277, 280
- odbicia, 288
- ortogonalny, 42, 44, 47
- ortonormalny, 90
- położenia, 48

- styczny, 280, 590
- światła, 283, 291, 293
- wyszukiwania, 530, 547
- zerowy, 39
- wektory
 - bazy standardowej, 86
 - kolumnowe, 66
 - mapy normalnych, 553
 - wierszowe, 66
 - XNA Math, 49
- wiązanie stanu głębokości/szablonu, 374
- widok
 - celu renderowania, 132
 - nieuporządkowanego dostępu, UAV, 426
 - zasobu shadera, 582
- widoki zasobów tekstury, 121
- wielkość wektorowa, 35
- wielokrotne próbkowanie, 122–124
- wielomian bazowy Bernsteina, 473
- wierzchołek, 176, 203, 269, 403
- wierzchołki
 - siatki, 248
 - trójkąta, 168
- wizualizacja
 - normalnych, 419
 - złożoności głębi, 393
- właściwości siatki, 580
- wskazywanie obiektów, 515, 527
- wskaznik do
 - bloku pamięci, 267
 - bufora, 118, 131
 - IDXGIFactory, 130
 - interfejsu, 230
 - kontekstu urządzenia, 128
 - obiektów technik, 230
 - obiektu, 319
 - tablicy, 210
 - urządzenia, 319
 - zmiennej, 229
- współczynnik
 - cienia, 656
 - mieszania, 347
 - mieszania celu, 346
 - mieszania źródła, 346
 - okluzji, 671
 - proporcji obrazu, 145, 183, 184
 - skalowania, 648
 - teselacji, 461, 587
- współrzędne
 - barycentryczne, 786, 808
 - jednorodne, 91, 107
 - NDC, 187
 - przestrzeni świata, 819
 - przestrzeni widoku, 813, 819, 825
 - rzutowe tekstury, 640, 642
 - tekstury, 316, 326, 331, 341
 - wektora, 37
- wtyczka do Adobe Photoshop, 555
- wygładzanie, 292
- wyglądanie, 579
- wyjście strumieniowe, 613, 616
- wyjście strumieniowe bez renderowania, 614
- wykorzystanie mapy okluzji, 686
- wykrywanie błędów, 152, 155
- wymiary tekstur, 411
- wysokość terenu, 598, 600
- wyświetlanie okna, 763
- wyznacznik macierzy, 71, 80
- wzory
 - Cramera, 71, 525
 - skróconego mnożenia, 708
- wzór na
 - interpolację sferyczną, 720
 - normalną, 279
 - obrót, 90
 - odwrotność macierzy, 75
 - oświetlenie reflektorowe, 294
 - pole równoległoboku, 809
 - projekcję cienia, 382, 385
 - promień wskazujący, 520, 521
 - wartość wektora, 40

Z

- zamiana współrzędnych układu, 808
- zapis równania rzutu, 187
- zasięg, 293
- zasoby, 754
- zbieranie danych, 168
- zbiór prymitywów, 410
- zdarzenia, 754
- zdarzenia myszy, 150
- zliczenia, 155
- złożoność głębi, 390, 391

zmiana

- formatu tekstury, 357

- kolejności składowych, 772

- rozmiaru buforów, 118

- układu współrzędnych, 98, 100

- punkt, 100

- reprezentacja macierzowa, 101

- wektor, 99

- zmiennie globalne, 761

- znormalizowana wartość głębokości, 188

- znormalizowane współrzędne urządzenia, 185, 193

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄŻKI**
- 3. ZBIERAJ PROWIZJĘ**

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

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 DirectX3D. 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 DirectX3D
- dowiesz się, jak korzystać z tekstur i cieni
- wykorzystasz system cząstek
- poznasz potencjał DirectX

helion.pl
księgarnia
internetowa

nr katalogowy 19774



Księgarnia internetowa:
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900



0 601 339900

MERCURY
LEARNING AND INFORMATION LLC



Helion

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/nowosci>

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

siegnij po WIĘCEJ



KOD KORZYSCI

ISBN 978-83-246-7474-9



9 788324 674749

cena: 129,00 zł

Informatyka w najlepszym wydaniu