

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

OpenGL. Programowanie gier

Autorzy: Kevin Hawkins, Dave Astle

Tłumaczenie: Jaromir Senczyk

ISBN: 83-7361-035-9

Tytuł oryginału: [OpenGL Game Programming](#)

Format: B5, stron: 630



Coraz szybsze procesory, coraz wydajniejsze karty graficzne – wszystko to powoduje, że programiści gier komputerowych potrafią kreować własne, wirtualne i trójwymiarowe światy, przyciągające gracza bogactwem szczegółów i drobiazgowym odwzorowaniem rzeczywistości (lub wyobraźni twórcy). Tworzenie tak zaawansowanych i skomplikowanych gier nie byłoby możliwe bez bibliotek graficznych, takich jak OpenGL, pozwalających na wyświetlanie trójwymiarowych obiektów przez karty graficzne różnych producentów. OpenGL w ogromnym stopniu przyspiesza pracę programisty.

Książka omawia użycie OpenGL do tworzenia dynamicznych, trójwymiarowych światów gier oraz efektów specjalnych. Przedstawia podstawy tworzenia aplikacji w systemie Windows, teorię grafiki trójwymiarowej, ale główny nacisk kładzie na prezentację funkcjonalności OpenGL. Jednak autor nie poprzestał na opisie. Sama biblioteka nie wystarcza do stworzenia gry. Opisane zostały także niezbędne programistyczne elementy biblioteki DirectX pozwalające na wzbogacenie gry o dźwięk i wygodne sterowanie.

Książka przedstawia:

- Podstawy programowania w Windows, funkcje WGL
- Podstawy teorii grafiki trójwymiarowej
- Maszynę stanów OpenGL i podstawowe elementy grafiki
- Przekształcenia układu współrzędnych i macierze OpenGL
- Kolory i efekty związane z oświetleniem
- Mapy bitowe i tekstury w OpenGL
- Listy wyświetlania i tablice wierzchołków
- Bufory OpenGL
- Krzywe, powierzchnie i powierzchnie drugiego stopnia
- Sposoby tworzenia efektów specjalnych
- Interfejs DirectInput
- Wykorzystanie DirectX Audio
- Trójwymiarowe modele postaci



Spis treści

List od wydawcy serii.....	15
O Autorach.....	17
Przedmowa.....	19
Wprowadzenie	21
Część I Wprowadzenie do OpenGL i DirectX	23
Rozdział 1. OpenGL i DirectX	25
Dlaczego tworzymy gry?	25
Świat gier	26
Elementy gry	26
Narzędzia	28
OpenGL.....	30
Historia OpenGL.....	30
Architektura OpenGL	31
Biblioteka GLU.....	31
Pakiet bibliotek GLUT.....	32
Programy.....	33
DirectX	33
Historia DirectX.....	34
Architektura DirectX	35
DirectX Graphics	36
DirectX Audio	36
DirectInput.....	36
DirectPlay.....	36
DirectShow	36
DirectSetup	37
Porównanie OpenGL i DirectX.....	37
Podsumowanie.....	37
Rozdział 2. Korzystanie z OpenGL w systemie Windows	39
Wprowadzenie do programowania w systemie Windows	39
Podstawowa aplikacja systemu Windows.....	41
Funkcja WinMain()	42
Procedura okienkowa.....	42
Obsługa komunikatów.....	43
Klasy okien.....	44
Określenie atrybutów klasy okna.....	45
Ładowanie ikon i kursora myszy	46

Rejestracja klasy	48
Tworzenia okna	48
Pętla przetwarzania komunikatów	50
Kompletna aplikacja systemu Windows	52
Wprowadzenie do funkcji WGL	55
Kontekst tworzenia grafiki	56
Korzystanie z funkcji WGL	56
Funkcja wglCreateContext()	56
Funkcja wglDeleteContext()	56
Funkcja wglMakeCurrent()	57
Formaty pikseli	58
Pole nSize	58
Pole dwFlags	59
Pole iPixelFormat	59
Pole cColorBits	59
Aplikacja OpenGL w systemie Windows	60
Pełnoekranowe aplikacje OpenGL	67
Podsumowanie	69
Rozdział 3. Przegląd teorii grafiki trójwymiarowej	71
Skalary, punkty i wektory	71
Długość wektora	72
Normalizacja wektora	72
Dodawanie wektorów	73
Mnożenie wektora przez skalar	73
Iloczyn skalarny wektorów	73
Iloczyn wektorowy	74
Macierze	75
Macierz jednostkowa	75
Macierz zerowa	75
Dodawanie i odejmowanie macierzy	76
Mnożenie macierzy	76
Implementacja działań na macierzach	78
Przekształcenia	79
Przesunięcie	80
Obrót	80
Skalowanie	82
Rzutowanie	82
Rzutowanie izometryczne	83
Rzutowanie perspektywiczne	84
Obcinanie	86
Światło	86
Światło otoczenia	87
Światło rozproszone	87
Światło odbijane	88
Odwzorowania tekstur	88
Podsumowanie	89
Część II Korzystanie z OpenGL	91
Rozdział 4. Maszyna stanów OpenGL i podstawowe elementy grafiki	93
Funkcje stanu	93
Podstawowe elementy grafiki	99
Tworzenie punktów w trójwymiarowej przestrzeni	100
Zmiana rozmiaru punktów	101
Antialiasing	101

Odcinki	102
Zmiana szerokości odcinka	103
Antialiasing odcinków	103
Wzór linii odcinka	103
Wielokąty	104
Ukrywanie powierzchni wielokątów	105
Ukrywanie krawędzi wielokątów	106
Antialiasing wielokątów	106
Wzór wypełnienia wielokątów	107
Trójkąty	107
Czworokąty	108
Dowolne wielokąty	109
Przykład wykorzystania podstawowych elementów grafiki	109
Podsumowanie	110
Rozdział 5. Przekształcenia układu współrzędnych i macierze OpenGL	111
Przekształcenia układu współrzędnych	111
Współrzędne kamery i obserwatora	113
Przekształcenia widoku	114
Wykorzystanie funkcji gluLookAt()	114
Wykorzystanie funkcji glRotate*() i glTranslate*()	115
Własne procedury przekształceń widoku	116
Przekształcenia modelowania	117
Rzutowanie	118
Przekształcenie okienkowe	118
OpenGL i macierze	119
Macierz modelowania	119
Przesunięcie	120
Obrót	120
Skalowanie	122
Stos macierzy	123
Model robota	124
Rzutowanie	132
Rzutowanie ortograficzne	133
Rzutowanie perspektywiczne	134
Okno widoku	135
Przykład rzutowania	136
Wykorzystanie własnych macierzy przekształceń	138
Ładowanie elementów macierzy	138
Mnożenie macierzy	139
Przykład zastosowania macierzy definiowanych	139
Podsumowanie	140
Rozdział 6. Kolory, łączenie kolorów i oświetlenie	141
Czym są kolory?	141
Kolory w OpenGL	142
Głębia koloru	143
Sześćcian kolorów	143
Model RGBA w OpenGL	143
Model indeksowany w OpenGL	144
Cieniowanie	145
Oświetlenie	147
Oświetlenie w OpenGL i w realnym świecie	147
Materiały	148

Normalne.....	148
Obliczanie normalnych.....	149
Użycie normalnych.....	151
Normalne jednostkowe.....	152
Korzystanie z oświetlenia w OpenGL.....	153
Tworzenie źródeł światła.....	158
Położenie źródła światła.....	159
Thumienie.....	160
Strumień światła.....	160
Definiowanie materiałów.....	163
Modele oświetlenia.....	164
Efekty odbicia.....	166
Ruchome źródła światła.....	167
Łączenie kolorów.....	173
Przezroczystość.....	174
Podsumowanie.....	179
Rozdział 7. Mapy bitowe i obrazy w OpenGL.....	181
Mapy bitowe w OpenGL.....	181
Umieszczanie map bitowych.....	182
Rysowanie mapy bitowej.....	183
Przykład zastosowania mapy bitowej.....	183
Wykorzystanie obrazów graficznych.....	185
Rysowanie obrazów graficznych.....	185
Odczytywanie obrazu z ekranu.....	187
Kopiowanie danych ekranu.....	187
Powiększanie, pomniejszanie i tworzenie odbić.....	188
Upakowanie danych mapy pikseli.....	188
Mapy bitowe systemu Windows.....	188
Format plików BMP.....	189
Ładowanie plików BMP.....	190
Zapis obrazu w pliku BMP.....	191
Pliki graficzne Targa.....	193
Format plików Targa.....	193
Ładowanie zawartości pliku Targa.....	194
Zapis obrazów w plikach Targa.....	196
Podsumowanie.....	197
Rozdział 8. Odwzorowania tekstur.....	199
Odwzorowania tekstur.....	199
Przykład zastosowania tekstury.....	200
Mapy tekstur.....	205
Tekstury dwuwymiarowe.....	205
Tekstury jednowymiarowe.....	206
Tekstury trójwymiarowe.....	206
Obiekty tekstur.....	207
Tworzenie nazwy tekstury.....	207
Tworzenie i stosowanie obiektów tekstur.....	207
Filtrowanie tekstur.....	208
Funkcje tekstur.....	209
Współrzędne tekstury.....	209
Powtarzanie i rozciąganie tekstur.....	210
Mipmapy i poziomy szczegółowości.....	212
Automatyczne tworzenie mipmap.....	213

Animacja powiewającej flagi.....	213
Objaśnienia.....	214
Implementacja.....	214
Ukształtowanie terenu.....	223
Objaśnienia.....	223
Implementacja.....	227
Podsumowanie.....	235
Rozdział 9. Zaawansowane odwzorowania tekstur.....	237
Tekstury wielokrotne.....	237
Sprawdzanie dostępności tekstur wielokrotnych.....	238
Dostęp do funkcji rozszerzeń.....	239
Tworzenie jednostek tekstury.....	240
Określanie współrzędnych tekstury.....	241
Przykład zastosowania tekstur wielokrotnych.....	242
Odwzorowanie otoczenia.....	250
Torus na niebie.....	250
Macierze tekstur.....	253
Mapy oświetlenia.....	255
Stosowanie map oświetlenia.....	255
Wieloprzebiegowe tekstury wielokrotne.....	261
Podsumowanie.....	265
Rozdział 10. Listy wyświetlania i tablice wierzchołków.....	267
Listy wyświetlania.....	267
Tworzenie listy wyświetlania.....	268
Umieszczanie poleceń na liście wyświetlania.....	268
Wykonywanie list wyświetlania.....	269
Uwagi dotyczące list wyświetlania.....	271
Usuwanie list wyświetlania.....	271
Listy wyświetlania i tekstury.....	272
Przykład: animacja robota z użyciem list wyświetlania.....	273
Tablice wierzchołków.....	274
Obsługa tablic wierzchołków w OpenGL.....	275
Stosowanie tablic wierzchołków.....	276
glDrawArrays().....	278
glDrawElements().....	278
glDrawRangeElements().....	279
glArrayElement().....	279
Tablice wierzchołków i tekstury wielokrotne.....	280
Blokowanie tablic wierzchołków.....	280
Przykład: ukształtowanie terenu po raz drugi.....	281
Podsumowanie.....	284
Rozdział 11. Wyświetlanie tekstów.....	285
Czcionki rastrowe.....	285
Czcionki konturowe.....	289
Czcionki pokryte teksturą.....	292
Podsumowanie.....	299
Rozdział 12. Bufory OpenGL.....	301
Bufory OpenGL — wprowadzenie.....	301
Konfiguracja formatu pikseli.....	301
Opróżnianie buforów.....	304
Bufor koloru.....	305
Podwójne buforowanie.....	305
Buforowanie stereoskopowe.....	306

Bufor głębi	306
Funkcje porównania głębi	307
Zastosowania bufora głębi	307
Bufor powielania	316
Przykład zastosowania bufora powielania	319
Bufor akumulacji	324
Podsumowanie	326
Rozdział 13. Powierzchnie drugiego stopnia	327
Powierzchnie drugiego stopnia w OpenGL	327
Styl powierzchni	328
Wektory normalne	328
Orientacja	329
Współrzędne tekstury	329
Usuwanie obiektów powierzchni	329
Dyski	329
Walce	331
Kule	332
Przykład użycia powierzchni drugiego stopnia	333
Podsumowanie	336
Rozdział 14. Krzywe i powierzchnie	337
Reprezentacja krzywych i powierzchni	337
Równania parametryczne	338
Punkty kontrolne i pojęcie ciągłości	338
Ewaluatory	339
Siatka równoodległa	342
Powierzchnie	343
Pokrywanie powierzchni teksturami	346
Powierzchnie B-sklejane	350
Podsumowanie	354
Rozdział 15. Efekty specjalne	355
Plakatowanie	355
Przykład: kaktusy na pustyni	357
Zastosowania systemów cząstek	359
Cząstki	359
Położenie	360
Prędkość	360
Czas życia	360
Rozmiar	361
Masa	361
Reprezentacja	361
Kolor	361
Przynależność	362
Metody	362
Systemy cząstek	362
Lista cząstek	362
Położenie	362
Częstość emisji	363
Oddziaływania	363
Atrybuty cząstek, zakresy ich wartości i wartości domyślne	363
Stan bieżący	364
Łączenie kolorów	364
Reprezentacja	364
Metody	364

Menedżer systemów cząstek	365
Implementacja.....	365
Tworzenie efektów za pomocą systemów cząstek.....	368
Przykład: śnieżyca.....	369
Mgła.....	373
Mgła w OpenGL.....	374
Mgła objętościowa.....	375
Odbicia.....	375
Odbicia światła.....	376
Obsługa bufora głębi.....	376
Obsługa skończonych płaszczyzn za pomocą bufora powielania.....	377
Tworzenie nieregularnych odbić.....	378
Odbicia na dowolnie zorientowanych płaszczyznach.....	378
Cienie.....	379
Cienie statyczne.....	379
Rzutowanie cieni.....	380
Macierz rzutowania cieni.....	380
Problemy z buforem głębi.....	381
Ograniczanie obszaru cieni za pomocą bufora powielania.....	381
Obsługa wielu źródeł światła i wielu zacienianych powierzchni.....	382
Problemy związane z rzutowaniem cieni.....	382
Bryły cieni w buforze powielania.....	383
Inne metody.....	384
Przykład: odbicia i cienie.....	384
Podsumowanie.....	387

Część III Tworzymy grę.....389

Rozdział 16. DirectX: DirectInput.....391

Dlaczego DirectInput?	391
Komunikaty systemu Windows.....	391
Interfejs programowy Win32.....	394
Win32 i obsługa klawiatury.....	394
Win32 i obsługa manipulatorów.....	396
DirectInput.....	397
Inicjacja interfejsu DirectInput.....	397
Wartości zwracane przez funkcje DirectInput.....	398
Korzystanie z DirectInput.....	399
Dodawanie urządzeń.....	399
Tworzenie urządzeń.....	400
Tworzenie wyliczenia urządzeń.....	400
Sprawdzanie możliwości urządzenia.....	404
Wyliczenia obiektów.....	405
Określanie formatu danych urządzenia.....	405
Określanie poziomu współpracy.....	407
Modyfikacja właściwości urządzenia.....	407
Zajmowanie urządzenia.....	408
Pobieranie danych wejściowych.....	408
Bezpośredni dostęp do danych.....	408
Buforowanie danych.....	409
„Odpytywanie” urządzeń.....	409
Kończenie pracy z urządzeniem.....	409
Odwzorowania akcji.....	410
Tworzenie podsystemu wejścia.....	410
Przykład zastosowania systemu wejścia.....	418
Podsumowanie.....	420

Rozdział 17. Zastosowania DirectX Audio	421
Dźwięk	421
Dźwięk i komputery	423
Cyfrowy zapis dźwięku	423
Synteza dźwięku	424
Czym jest DirectX Audio?	425
Charakterystyka DirectX Audio	426
Obiekty ładujące	426
Segmenty i stany segmentów	426
Wykonanie	427
Komunikaty	427
Kanały wykonania	427
Syntezator DSL	427
Instrumenty i ładowanie dźwięków	427
Ścieżki dźwięku i bufor	428
Przepływ danych dźwięku	428
Ładowanie i odtwarzanie dźwięku w DirectMusic	429
Inicjacja COM	430
Tworzenie i inicjacja obiektu wykonania	430
Tworzenie obiektu ładującego	431
Załadowanie segmentu	431
Ładowanie instrumentów	432
Odtwarzanie segmentu	432
Zatrzymanie odtwarzania segmentu	433
Kontrola odtwarzania segmentu	434
Określanie liczby odtworzeń segmentu	434
Kończenie odtwarzania dźwięku	435
Prosty przykład	435
Zastosowania ścieżek dźwięku	445
Domyślna ścieżka dźwięku	446
Standardowe ścieżki dźwięku	446
Odtwarzanie za pomocą ścieżki dźwięku	447
Pobieranie obiektów należących do ścieżek dźwięku	449
Dźwięk przestrzenny	450
Współrzędne przestrzeni dźwięku	450
Percepcja położenia źródła dźwięku	450
Bufor efektu przestrzennego w DirectSound	451
Parametry przestrzennego źródła dźwięku	451
Odległość minimalna i maksymalna	453
Tryb działania	453
Położenie i prędkość	454
Stożki dźwięku	454
Odbiorca efektu dźwięku przestrzennego	455
Przykład zastosowania efektu przestrzennego	456
Podsumowanie	468
Rozdział 18. Modele trójwymiarowe	469
Formaty plików modeli trójwymiarowych	469
Format MD2	470
Implementacja formatu MD2	472
Ładowanie modelu MD2	476
Wyświetlanie modelu MD2	480
Pokrywanie modelu teksturą	482
Animacja modelu	483

Klasa CMD2Model.....	488
Sterowanie animacją modelu	498
Ładowanie plików PCX.....	501
Podsumowanie.....	503
Rozdział 19. Modelowanie fizycznych właściwości świata.....	505
Powtórka z fizyki.....	505
Czas.....	505
Odległość, przemieszczenie i położenie.....	506
Prędkość	508
Przyspieszenie.....	509
Siła	510
Pierwsza zasada dynamiki	511
Druga zasada dynamiki	511
Trzecia zasada dynamiki.....	511
Pęd.....	511
Zasada zachowania pędu	512
Tarcie.....	513
Tarcie na płaszczyźnie.....	513
Tarcie na równi pochyłej	514
Modelowanie świata rzeczywistego	516
Rozkład problemu	516
Czas.....	517
Wektor.....	521
Płaszczyzna	526
Obiekt	529
Zderzenia obiektów	531
Sfery ograniczeń.....	532
Prostopadłościanny ograniczeń	533
Zderzenia płaszczyzn	535
Obsługa zderzenia.....	538
Przykład: hokej	538
Świat hokeja.....	539
Lodowisko	539
Krażek i zderzenia	544
Gracz.....	550
Kompletowanie programu.....	553
Podsumowanie.....	559
Rozdział 20. Tworzenie szkieletu gry.....	561
Architektura szkieletu SimpEngine	561
Zarządzanie danymi za pomocą obiektów klasy CNode	562
Zarządzanie obiektami: klasa CObject.....	566
Trzon szkieletu SimpEngine.....	570
System wejścia.....	572
Klasa CEngine	573
Cykl gry	574
Obsługa wejścia	575
Klasa CSimpEngine.....	576
Kamera	577
Świat gry.....	580
Obsługa modeli.....	580
System dźwięku.....	581
System cząstek	583
Podsumowanie.....	583

Rozdział 21. Piszemy grę: „Czas zabijania”	585
Wstępny projekt	585
Świat gry	586
Przeciwnicy	588
Sztuczna inteligencja przeciwnika	589
Ogro	590
Sod	592
Rakiety i eksplozje	592
Interfejs użytkownika gry	594
Korzystanie z gry	594
Kompilacja gry	595
Podsumowanie	596
Dodatki	597
Dodatek A Zasoby sieci Internet	599
Programowanie gier	599
GameDev.net	599
Wyszukiwarka informacji związanych z programowaniem gier	600
flipCode	600
Gamasutra	600
OpenGL	600
NeHe Productions	600
OpenGL.org	601
Inne strony poświęcone OpenGL	601
DirectX	601
DirectX Developer Center	601
Lista mailingowa DirectX	601
Inne zasoby	602
ParticleSystems.com	602
Real-Time Rendering	602
Informacja techniczna dla programistów	602
Artykuły dotyczące efektu mgły	602
Dodatek B Dysk CD	603
Struktura plików na dysku CD	603
Wymagania sprzętowe	603
Instalacja	604
Typowe problemy i ich rozwiązywanie	604
Skorowidz	605

Rozdział 9.

Zaawansowane odwzorowania tekstur

W rozdziale tym przedstawione zostaną zaawansowane techniki tworzenia tekstur, które pozwolą kolejny raz na zwiększenie realizmu tworzonej grafiki. Omówione zostaną następujące zagadnienia:

- ♦ tekstury wielokrotne;
- ♦ mapy otoczenia;
- ♦ macierz tekstur;
- ♦ mapy oświetlenia;
- ♦ wieloprzebiegowe tworzenie tekstur wielokrotnych.

Zastosowanie każdego z wymienionych efektów powoduje, że grafika trójwymiarowa nabiera jeszcze doskonalszego wyglądu.

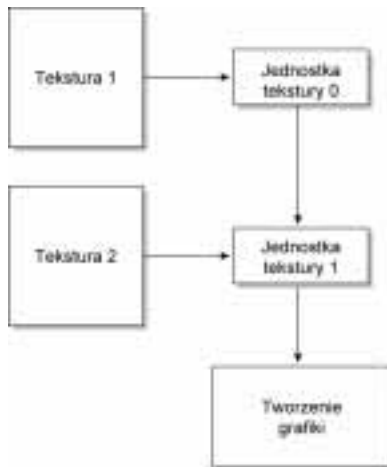
Tekstury wielokrotne

Dotychczas wykorzystywana była pojedyncza tekstura dla każdego wielokąta. OpenGL umożliwia jednak pokrycie wielokąta sekwencją tekstur, czyli tworzenie *tekstur wielokrotnych*. Możliwość ta stanowi obecnie jedynie *rozszerzenie* specyfikacji OpenGL. Rozszerzenia interfejsu programowego OpenGL zatwierdzone są przez radę ARB (*Architectural Review Board*) nadzorującą rozwój specyfikacji OpenGL. Tekstury wielokrotne stanowią opcjonalne rozszerzenie OpenGL i nie są dostępne w każdej implementacji.

Tekstury wielokrotne tworzone są za pomocą sekwencji *jednostek tekstury*. Jednostki tekstury omówione zostaną szczegółowo w dalszej części rozdziału. Obecnie wystarczy jedynie informacja dotycząca tego, że każda jednostka tekstury reprezentuje pojedynczą teksturę, a także to, że podczas tworzenia tekstury wielokrotnej każda jednostka tekstury przekazuje wynik jej zastosowania kolejnej jednostce tekstury aż do utworzenia końcowej tekstury wielokrotnej. Rysunek 9.1 ilustruje proces tworzenia tekstur wielokrotnych.

Rysunek 9.1.

Tekstura wielokrotna powstaje na skutek nałożenia na wielokąt więcej niż jednej tekstury za pomocą jednostek tekstur



W procesie tworzenia tekstury wielokrotnej można wyróżnić cztery główne etapy.

1. Sprawdzenie, czy implementacja OpenGL udostępnia możliwość tworzenia tekstur wielokrotnych.
2. Uzyskanie wskaźnika funkcji rozszerzenia.
3. Tworzenie jednostki tekstury.
4. Określenie współrzędnych tekstury.

Teraz należy przyjrzeć się bliżej każdemu z wymienionych etapów.

Sprawdzanie dostępności tekstur wielokrotnych

Przed rozpoczęciem stosowania tekstur wielokrotnych trzeba najpierw sprawdzić, czy udostępnia je używana implementacja OpenGL. Tekstury wielokrotne są rozszerzeniem specyfikacji OpenGL zaaprobowanym przez radę ARB. Listę takich rozszerzeń udostępnianych przez konkretną implementację można uzyskać wywołując funkcję `glGetString()` z parametrem `GL_EXTENSIONS`. Lista ta posiada postać łańcucha znaków, w którym nazwy kolejnych rozszerzeń rozdzielone są znakami spacji. Jeśli w łańcuchu tym znaleziona zostanie nazwa `"GL_ARB_multitexture"`, to oznacza to dostępność tekstur wielokrotnych.

W celu sprawdzenia dostępności konkretnego rozszerzenia OpenGL można skorzystać z dostępnej w bibliotece GLU funkcji `gluCheckExtension()` o następującym prototypie:

```
GLboolean gluCheckExtension(char *extName, const GLubyte *extString);
```

Funkcja ta zwraca wartość `true`, jeśli nazwa rozszerzenia `extName` znajduje się w łańcuchu `extString` lub wartość `false` w przeciwnym razie.

Łańcuch zwrócony przez funkcję `glGetString()` można także przeszukać indywidualnie. Oto przykład implementacji odpowiedniej funkcji:

```

bool InStr(char *searchStr, char *str)
{
    char *endOfStr;           // wskaźnik ostatniego znaku łańcucha
    int idx = 0;

    endOfStr = str + strlen(str); // ostatni znak łańcucha

    // pętla wykonywana aż do osiągnięcia końca łańcucha
    while (str < endOfStr)
    {
        // odnajduje położenie kolejnego znaku spacji
        idx = strcspn(str, " ");

        // sprawdza str i wskazuje poszukiwaną nazwę
        if ( (strlen(searchStr) == idx) && (strncmp(searchStr, str, idx) == 0))
        {
            return true;
        }

        // nie jest to poszukiwana nazwa, przesuwamy wskaźnik do kolejnej nazwy
        str += (idx + 1);
    }
    return false;
}

```

Funkcja ta zwraca wartość `true`, jeśli łańcuch `str` zawiera łańcuch `searchStr` przy założeniu, że `str` składa się z podłańcuchów oddzielonych znakami spacji. Aby sprawdzić, czy dana implementacja umożliwia stosowanie tekstur wielokrotnych, można skorzystać ostatecznie z poniższego fragmentu kodu:

```

char *extensionStr;           // lista dostępnych rozszerzeń

// pobiera listę dostępnych rozszerzeń
extensionStr = (char*)glGetString(GL_EXTENSIONS);

if (InStr("GL_ARB_multitexture", extensionStr))
{
    // tekstury wielokrotne są dostępne
}

```

Po sprawdzeniu dostępności tekstur wielokrotnych kolejnym krokiem będzie uzyskanie wskaźników funkcji rozszerzeń.

Dostęp do funkcji rozszerzeń

Aby korzystać z rozszerzeń OpenGL na platformie Microsoft Windows musimy uzyskać dostęp do funkcji rozszerzeń. Funkcja `wglGetProcAddress()` zwraca wskaźnik żądanej funkcji rozszerzenia. Dla tekstur wielokrotnych dostępnych jest sześć funkcji rozszerzeń:

- ♦ `glMultiTexCoordiARB` (gdzie $i=1..4$) — funkcje te określają współrzędne tekstur wielokrotnych;
- ♦ `glActiveTextureARB` — wybiera bieżącą jednostkę tekstury;
- ♦ `glClientActiveTextureARB` — wybiera jednostkę tekstury, na której wykonywane będą operacje.

Poniższy fragment kodu służy do uzyskania dostępu do wymienionych funkcji rozszerzeń:

```
PFNGLMULTITEXCOORD2FARBPROC glMultiTexCoord2fARB = NULL;
PFNGLACTIVETEXTUREARBPROC glActiveTextureARB = NULL;
PFNGLCLIENTACTIVETEXTUREARBPROC glClientActiveTextureARB = NULL;
glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC)
    wglGetProcAddress("glMultiTexCoord2fARB");
glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC)
    wglGetProcAddress("glActiveTextureARB");
glClientActiveTextureARB = (PFNGLCLIENTACTIVETEXTUREARBPROC)
    wglGetProcAddress("glClientActiveTextureARB");
```

Po wykonaniu tego kodu można tworzyć tekstury wielokrotne za pomocą funkcji `glMultiTexCoord2fARB()` i `glActiveTextureARB()`.

Tworzenie jednostek tekstury

Tworzenie tekstur wielokrotnych odbywa się poprzez nakładanie wielu jednostek tekstury. Jednostka tekstury składa się z obrazu tekstury, parametrów filtrowania, stosu macierzy tekstury, a ponadto posiada zdolność automatycznego tworzenia współrzędnych tekstury.

Aby skonfigurować parametry jednostki tekstury, trzeba najpierw wybrać bieżącą jednostkę tekstury za pomocą funkcji `glActiveTextureARB()` zdefiniowanej w następujący sposób:

```
void glActiveTextureARB(GLenum texUnit);
```

Po wykonaniu tej funkcji wszystkie wywołania funkcji `glTexImage*()`, `glTexParameter*()`, `glTexEnv*()`, `glTexGen*()` i `glBindTexture()` dotyczyć będą jednostki tekstury określonej przez parametr *texUnit*. Parametr *texUnit* może posiadać wartość `GL_TEXTURE0_ARB`, gdzie *i* jest liczbą całkowitą z przedziału od 0 do wartości o jeden mniejszej od dopuszczalnej liczby jednostek tekstur. Na przykład `GL_TEXTURE0_ARB` oznacza pierwszą z dostępnych jednostek tekstur. Liczbę jednostek tekstur dopuszczalną dla danej implementacji OpenGL uzyskać można wykonując poniższy fragment kodu:

```
int maxTextUnits; // maksymalna liczba jednostek tekstur
glGetIntegerv(GL_MAX_TEXTURE_UNITS_ARB, &maxTextUnits);
```

Jeśli w wyniku wywołania funkcji `glGetIntegerv()` uzyskana zostanie wartość 1, oznacza to będzie, że dana implementacja nie umożliwia tworzenia tekstur wielokrotnych.

Jednostki tekstur można konfigurować za pośrednictwem obiektów tekstur. Informacja, którą zawiera obiekt tekstury, jest automatycznie przechowywana z jednostką tekstury. Aby korzystać z jednostek tekstur, wystarczy więc — podobnie jak w przypadku zwykłych tekstur — utworzyć obiekt tekstury, a następnie aktywować jednostkę tekstury za pomocą funkcji `glActiveTextureARB()`. Następnie należy z wiązać obiekty tekstur z odpowiadającymi im jednostkami tekstur. Ilustruje to następujący fragment kodu:

```
// obiekty tekstur
int texObjects[2];
...
glGenTextures(2, texObjects); // tworzy dwa obiekty tekstur
```

```
// tworzy pierwszą teksturę
glBindTexture(GL_TEXTURE_2D, texObjects[0]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, 64, 64, GL_RGB, GL_UNSIGNED_BYTE, texOne);
...
// tworzy drugą teksturę
glBindTexture(GL_TEXTURE_2D, texObjects[1]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, 64, 64, GL_RGB, GL_UNSIGNED_BYTE, texTwo);
...
// tworzy jednostki tekstur za pomocą obiektów tekstur
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, smileTex->texID);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, checkerTex->texID);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
...
```

Powyższy fragment kodu tworzy najpierw dwa obiekty tekstur, a następnie kreuje za ich pomocą dwie jednostki tekstur używane do tworzenia tekstur wielokrotnych.

Określanie współrzędnych tekstury

Teraz, gdy wiadomo już, w jaki sposób tworzyć jednostki tekstur, należy zapoznać się ze sposobem ich użycia. Jako że pojedynczy wielokąt będzie więcej niż jedną teksturą, trzeba zdefiniować dla niego także więcej niż jeden zestaw współrzędnych tekstury. Dla każdej jednostki tekstury należy więc dysponować osobnym zbiorem współrzędnych tekstury i zastosować je dla każdego wierzchołka z osobna. Aby określić współrzędne tekstury dwuwymiarowej można zastosować funkcję `glMultiTexCoord2fARB()` zdefiniowaną w następujący sposób:

```
void glMultiTexCoord2fARB(GLenum texUnit, float coords);
```

Inne wersje tej funkcji umożliwiają określenie współrzędnych tekstu jedno-, trój- oraz czterowymiarowych także za pomocą wartości innych typów niż `float`. Na przykład funkcja `glMultiTexCoord3dARB()` pozwala określić współrzędne tekstury trójwymiarowej za pomocą wartości typu `double`. Korzystając z tych funkcji należy podać najpierw jednostkę tekstury, a następnie jej współrzędne. Oto przykład:

```
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 0.0f);
```

Powyższe wywołanie funkcji nadaje bieżącemu wierzchołkowi współrzędne (0.0, 0.0) pierwszej jednostki tekstur. Podobnie jak w przypadku funkcji `glTexCoord2f()` trzeba

określić najpierw współrzędne wszystkich używanych jednostek tekstur, a dopiero potem zdefiniować wierzchołek. Ilustruje to poniższy fragment kodu:

```
glBegin(GL_QUADS);
// lewy, dolny wierzchołek
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 0.0f);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 0.0f);
glVertex3f(-5.0f, -5.0f, -5.0f);

// prawy, dolny wierzchołek
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 0.0f);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 0.0f);
glVertex3f(5.0f, -5.0f, -5.0f);

// prawy, górny wierzchołek
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 1.0f);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 1.0f);
glVertex3f(5.0f, 5.0f, -5.0f);

// lewy, górny wierzchołek
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 1.0f);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 1.0f);
glVertex3f(-5.0f, 5.0f, -5.0f);
glEnd();
```

Fragment ten rysuje kwadrat pokryty dwiema teksturami. Dla każdego wierzchołka definiuje współrzędne obu tekstur.

Należy pamiętać, że przy stosowaniu tekstur wielokrotnych funkcja `glTexCoord2f()` pozwala na zdefiniowanie współrzędnych tekstury tylko dla pierwszej jednostki tekstury. W takim przypadku wywołanie funkcji `glTexCoord2f()` jest więc równoważne wywołaniu `glMultiTexCoord2f(GL_TEXTURE0_ARB, ...)`.

Przykład zastosowania tekstur wielokrotnych

Teraz analizie zostanie poddany kompletny przykład programu, który pokrywa sześcian dwiema teksturami pokazanymi na rysunku 9.2.

Rysunek 9.2.

*Dwie tekstury,
którymi pokryty
zostanie sześcian*



A oto pierwszy fragment kodu programu:

```
//////// Definicje
#define BITMAP_ID 0x4D42           // identyfikator formatu BMP
#define PI 3.14195

//////// Includes
#include <windows.h>               // standardowy plik nagłówkowy Windows
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```

#include <gl/gl.h>                // standardowy plik nagłówkowy OpenGL
#include <gl/glu.h>                // plik nagłówkowy biblioteki GLU
#include "glex.h"                  // plik nagłówkowy rozszerzeń OpenGL

///// Typy
typedef struct
{
    int width;                    // szerokość tekstury
    int height;                   // wysokość tekstury
    unsigned int texID;           // obiekt tekstury
    unsigned char *data;          // dane tekstury
} texture_t;

///// Zmienne globalne
HDC g_HDC;                       // globalny kontekst urządzenia
bool fullScreen = false;         // true = tryb pełnoekranowy
                                   // false = tryb okienkowy
bool keyPressed[256];            // tablica przyciśnięć klawiszy
float angle = 0.0f;              // kąt obrotu
float radians = 0.0f;            // kąt obrotu kamery w radianach

///// Zmienne myszy i kamery
int mouseX, mouseY;              // współrzędne myszy
float cameraX, cameraY, cameraZ; // współrzędne kamery
float lookX, lookY, lookZ;       // punkt wycelowania kamery

///// Zmienne tekstur
texture_t *smileTex;
texture_t *checkerTex;

///// Zmienne związane z teksturami wielokrotnymi
PFNGLMULTITEXCOORD2FARBPROC glMultiTexCoord2fARB = NULL;
PFNGLACTIVETEXTUREARBPROC glActiveTextureARB = NULL;
PFNGLCLIENTACTIVETEXTUREARBPROC glClientActiveTextureARB = NULL;
int maxTextureUnits = 0;         // dopuszczalna liczba jednostek tekstur

```

Plik nagłówkowy *glex.h* należy dołączać wtedy, gdy korzysta się z jakichkolwiek rozszerzeń OpenGL zatwierdzonych przez ARB. Plik ten zawiera definicje wszystkich rozszerzeń. Jego kopia umieszczona została także na dysku CD.

Powyższy fragment kodu prezentuje także sposób wykorzystania struktur do przechowywania informacji o teksturze i jej danych. Typ `texture_t` umożliwia zapamiętanie obiektu tekstury, jej szerokości i wysokości oraz danych. Upraszcza to proces ładowania i stosowania tekstur w OpenGL.

Omawiany program będzie umożliwiać sterowanie kamerą OpenGL w podobny sposób do programu prezentującego rzeźbę terenu, który przedstawiony został w rozdziale 8. Gdy mysz będzie poruszać się poziomo, sześcian będzie obracany wokół osi y. Natomiast przy wykonywaniu pionowych ruchów myszą, można spowodować oddalanie się bądź przybliżanie kamery do sześcianu.

Pierwszą z funkcji, którą definiuje program, jest omówiona już wcześniej funkcja `InStr()`. Zwraca ona wartość `true`, jeśli łańcuch `searchStr` zostanie odnaleziony wewnątrz łańcucha `str`, który zawiera wiele nazw oddzielonych znakami spacji.

```

bool InStr(char *searchStr, char *str)
{
    char *endOfStr;// wskaźnik ostatniego znaku łańcucha
    int idx = 0;

    endOfStr = str + strlen(str);// ostatni znak łańcucha

    // pętla wykonywana aż do osiągnięcia końca łańcucha
    while (str < endOfStr)
    {
        // odnajduje położenie kolejnego znaku spacji
        idx = strchr(str, " ");

        // sprawdza, czy str wskazuje poszukiwaną nazwę
        if ( (strlen(searchStr) == idx) && (strcmp(searchStr, str, idx) == 0))
        {
            return true;
        }

        // nie jest to poszukiwana nazwa, przesuwamy wskaźnik do kolejnej nazwy
        str += (idx + 1);
    }
    return false;
}

```

W celu umożliwienia korzystania z tekstur wielokrotnych program definiuje funkcję `InitMultiTex()`, która sprawdza, czy dostępne jest rozszerzenie o nazwie `"GL_ARB_multitexture"`, a ponadto pozyskuje wskaźniki funkcji rozszerzeń. Funkcja `InitMultiTex()` zwraca wartość `false`, jeśli tworzenie tekstur wielokrotnych nie jest dostępne.

```

bool InitMultiTex()
{
    char *extensionStr;// lista dostępnych rozszerzeń

    extensionStr = (char*)glGetString(GL_EXTENSIONS);

    if (extensionStr == NULL)
        return false;

    if (InStr("GL_ARB_multitexture", extensionStr))
    {
        // zwraca dopuszczalną liczbę jednostek tekstur
        glGetIntegerv(GL_MAX_TEXTURE_UNITS_ARB, &maxTextureUnits);

        // pobiera adresy funkcji rozszerzeń
        glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC)
            wglGetProcAddress("glMultiTexCoord2fARB");
        glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC)
            wglGetProcAddress("glActiveTextureARB");
        glClientActiveTextureARB = (PFNGLCLIENTACTIVETEXTUREARBPROC)
            wglGetProcAddress("glClientActiveTextureARB");

        return true;
    }
    else
        return false;
}

```

Kolejna funkcja, `LoadTextureFile()`, ładuje pojedynczą teksturę i umieszcza jej opis w strukturze typu `texture_t` i zwraca wskaźnik tej struktury. Parametrem funkcji `LoadTextureFile()` jest nazwa pliku. W celu załadowania jego zawartości wywołuje ona funkcję `LoadBitmapFile()` i umieszcza informacje o załadowanym obrazie w strukturze typu `texture_t`. Funkcja `LoadTextureFile()` tworzy także obiekt tekstury.

```
texture_t *LoadTextureFile(char *filename)
{
    BITMAPINFOHEADER texInfo;
    texture_t *thisTexture;

    // przydziela pamięć strukturze typu texture_t
    thisTexture = (texture_t*)malloc(sizeof(texture_t));
    if (thisTexture == NULL)
        return NULL;

    // ładuje obraz tekstury i sprawdza poprawne wykonanie tej operacji
    thisTexture->data = LoadBitmapFile(filename, &texInfo);
    if (thisTexture->data == NULL)
    {
        free(thisTexture);
        return NULL;
    }

    // umieszcza informacje o szerokości i wysokości tekstury
    thisTexture->width = texInfo.biWidth;
    thisTexture->height = texInfo.biHeight;

    // tworzy obiekt tekstury
    glGenTextures(1, &thisTexture->texID);

    return thisTexture;
}
```

Jako że opanowana już została umiejętność ładowania tekstury, można zapoznać się z kolejną funkcją, która może załadować wszystkie tekstury używane w programie i skonfigurować je tak, by możliwe było ich wykorzystanie do tworzenia tekstur wielokrotnych. Funkcja `LoadAllTextures()` ładuje wszystkie tekstury, określa ich parametry, tworzy mipmapy i wiąże obiekty tekstur z jednostkami tekstur.

```
bool LoadAllTextures()
{
    // ładuje pierwszą teksturę
    smileTex = LoadTextureFile("smile.bmp");
    if (smileTex == NULL)
        return false;

    // ładuje drugą teksturę
    checkerTex = LoadTextureFile("chess.bmp");
    if (checkerTex == NULL)
        return false;

    // tworzy mipmapy z filtrowaniem liniowym dla pierwszej tekstury
    glBindTexture(GL_TEXTURE_2D, smileTex->texID);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
}
```

```

glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, smileTex->width, smileTex->height,
    GL_RGB, GL_UNSIGNED_BYTE, smileTex->data);

// tworzy mipmapy z filtrowaniem liniowym dla drugiej tekstury
glBindTexture(GL_TEXTURE_2D, checkerTex->texID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, checkerTex->width, checkerTex->height,
    GL_RGB, GL_UNSIGNED_BYTE, checkerTex->data);

// wybiera pierwszą jednostkę tekstury
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, smileTex->texID); // wiąże jednostkę z pierwszą teksturą

// wybiera drugą jednostkę tekstury
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, checkerTex->texID); // wiąże jednostkę z drugą teksturą

return true;
}

```

Funkcja Initialize() inicjuje dane programu:

```

void Initialize()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // tło w czarnym kolorze

    glShadeModel(GL_SMOOTH);                // cieniowanie gładkie
    glEnable(GL_DEPTH_TEST);                 // usuwanie ukrytych powierzchni
    glEnable(GL_CULL_FACE);                  // brak obliczeń dla niewidocznych stron wielokątów
    glFrontFace(GL_CCW);                     // niewidoczne strony posiadają porządek wierzchołków
                                           // przeciwny do kierunku ruchu wskazówek zegara

    glEnable(GL_TEXTURE_2D); // włącza tekstury dwuwymiarowe

    InitMultiTex();

    LoadAllTextures();
}

```

Sześcian narysować można za pomocą rozbudowanej wersji funkcji DrawCube() znanej z poprzednich przykładów. Będzie ona korzystać z funkcji glMultiTexCoord2f() w miejsce funkcji glTexCoord2f() w celu zdefiniowania współrzędnych jednostek tekstur dla każdego wierzchołka sześcianu:

```

void DrawCube(float xPos, float yPos, float zPos)
{
    glPushMatrix();
    glTranslatef(xPos, yPos, zPos);
    glBegin(GL_QUADS);
        glNormal3f(0.0f, 1.0f, 0.0f);        // górna ściana

```

```

glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 0.0f);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 0.0f);
glVertex3f(0.5f, 0.5f, 0.5f);

glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 1.0f);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 1.0f);
glVertex3f(0.5f, 0.5f, -0.5f);

glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 1.0f);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 1.0f);
glVertex3f(-0.5f, 0.5f, -0.5f);

glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 0.0f);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 0.0f);
glVertex3f(-0.5f, 0.5f, 0.5f);
glEnd();
glBegin(GL_QUADS);
    glNormal3f(0.0f, 0.0f, 1.0f);           // przednia ściana

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 1.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 1.0f);
    glVertex3f(0.5f, 0.5f, 0.5f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 1.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 1.0f);
    glVertex3f(-0.5f, 0.5f, 0.5f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 0.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 0.0f);
    glVertex3f(-0.5f, -0.5f, 0.5f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 0.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 0.0f);
    glVertex3f(0.5f, -0.5f, 0.5f);
glEnd();
glBegin(GL_QUADS);
    glNormal3f(1.0f, 0.0f, 0.0f);           // prawa ściana

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 1.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 1.0f);
    glVertex3f(0.5f, 0.5f, 0.5f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 0.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 0.0f);
    glVertex3f(0.5f, -0.5f, 0.5f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 0.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 0.0f);
    glVertex3f(0.5f, -0.5f, -0.5f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 1.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 1.0f);
    glVertex3f(0.5f, 0.5f, -0.5f);
glEnd();
glBegin(GL_QUADS);
    glNormal3f(-1.0f, 0.0f, 0.0f);          // lewa ściana

```

```

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 1.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 1.0f);
    glVertex3f(-0.5f, 0.5f, 0.5f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 1.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 1.0f);
    glVertex3f(-0.5f, 0.5f, -0.5f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 0.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 0.0f);
    glVertex3f(-0.5f, -0.5f, -0.5f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 0.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 0.0f);
    glVertex3f(-0.5f, -0.5f, 0.5f);
glEnd();
glBegin(GL_QUADS);
    glNormal3f(0.0f, -1.0f, 0.0f);           // dolna ściana

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 0.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 0.0f);
    glVertex3f(-0.5f, -0.5f, 0.5f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 1.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 1.0f);
    glVertex3f(-0.5f, -0.5f, -0.5f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 1.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 1.0f);
    glVertex3f(0.5f, -0.5f, -0.5f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 0.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 0.0f);
    glVertex3f(0.5f, -0.5f, 0.5f);
glEnd();
glBegin(GL_QUADS);
    glNormal3f(0.0f, 0.0f, -1.0f);          // tylna ściana

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 0.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 0.0f);
    glVertex3f(0.5f, -0.5f, -0.5f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 0.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 0.0f);
    glVertex3f(-0.5f, -0.5f, -0.5f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 1.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 1.0f);
    glVertex3f(-0.5f, 0.5f, -0.5f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 1.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 1.0f);
    glVertex3f(0.5f, 0.5f, -0.5f);
glEnd();
glPopMatrix();
}

```

Większość kodu funkcji `Render()` znana jest z poprzednich przykładów. Określa ona położenie kamery i rysuje sześcian.

```
void Render()
{
    radians = float(PI*(angle-90.0f)/180.0f);

    // wyznacza położenie kamery
    cameraX = lookX + sin(radians)*mouseY;
    cameraZ = lookZ + cos(radians)*mouseY;
    cameraY = lookY + mouseY / 2.0f;

    // wycelowuje kamerę w punkt (0,0,0)
    lookX = 0.0f;
    lookY = 0.0f;
    lookZ = 0.0f;

    // opróżnia bufor ekranu i głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    // umieszcza kamerę
    gluLookAt(cameraX, cameraY, cameraZ, lookX, lookY, lookZ, 0.0, 1.0, 0.0);

    // skaluje sześcian do rozmiarów 15x15x15
    glScalef(15.0f, 15.0f, 15.0f);

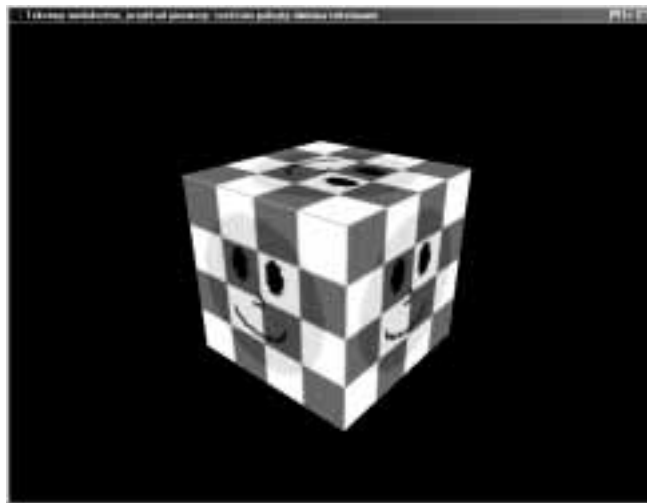
    // rysuje sześcian o środku w punkcie (0,0,0)
    DrawCube(0.0f, 0.0f, 0.0f);

    SwapBuffers(g_HDC);          // przełącza bufor
}
```

Po umieszczeniu omówionych funkcji w szkieletcie standardowej aplikacji systemu Windows uzyskany zostanie program, którego działanie ilustruje rysunek 9.3.

Rysunek 9.3.

*Przykład
zastosowania
tekstury wielokrotnej*



Tekstury wielokrotne mogą być źródłem doskonałych efektów i dlatego z pewnością warto z nimi poeksperymentować trochę więcej. Pora jednak na omówienie kolejnego doskonałego narzędzia tworzenia zaawansowanych efektów graficznych: odwzorowania otoczenia.

Odwzorowanie otoczenia

Odwzorowanie otoczenia polega na stworzeniu obiektu, którego powierzchnia wydaje się odbiciem jego otoczenia. Na przykład doskonale wypolerowana srebrna kula odbijać będzie na swojej powierzchni elementy otaczającego ją świata. Właśnie taki efekt można uzyskać stosując odwzorowanie otoczenia. W tym celu nie trzeba jednak wcale tworzyć obrazu odbicia otoczenia na powierzchni efektu. W OpenGL wystarczy jedynie utworzyć teksturę reprezentującą otoczenie obiektu, a OpenGL automatycznie wygeneruje odpowiednie współrzędne tekstury. Zmiana położenia obiektu wymaga wyznaczenia nowych współrzędnych tekstury, by można było uzyskać efekt odbicia zmieniającego się razem z poruszającym się obiektem.

Najlepszy efekt zastosowania odwzorowania otoczenia uzyskać można używając tekstur utworzonych za pomocą „rybiego oka”. Zastosowanie takich tekstur nie jest absolutnie konieczne, ale znacznie zwiększa realizm odwzorowania otoczenia. Współrzędne tekstury *można* też obliczać indywidualnie, jednak zmniejsza to zwykle efektywność tworzenia grafiki.

Aby skorzystać w OpenGL z możliwości odwzorowania otoczenia, niezbędny jest poniższy fragment kodu:

```
glTexGenf(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGenf(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
// następnie wybiera teksturę otoczenia i rysuje obiekt
```

I to wszystko! Teraz należy przeanalizować przykład zastosowania możliwości odwzorowania otoczenia.

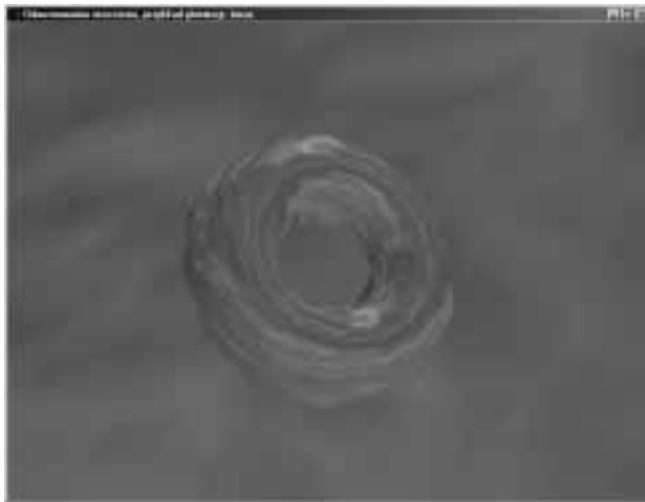
Torus na niebie

Omówiony teraz zostanie przykład zastosowania możliwości odwzorowania otoczenia w programie, który rysować będzie torus pokryty teksturą otoczenia. Rysunek 9.4 przedstawia końcowy efekt działania programu.

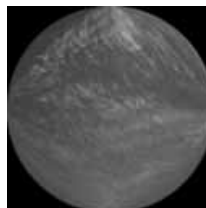
Teksturę otoczenia utworzyć można na podstawie tekstury nieba, do której zastosowany zostanie efekt rybiego oka za pomocą dowolnego edytora obrazów dysponującego taką możliwością. Rysunek 9.5 przedstawia otrzymaną teksturę otoczenia.

Rysunek 9.4.

*Torus pokryty
teksturą otoczenia*

**Rysunek 9.5.**

*Tekstura otoczenia
uzyskana
na podstawie
tekstury nieba*



Teraz należy przyjrzeć się kodowi programu. Poniżej przedstawione zostały jego fragmenty związane z tworzeniem tekstur i rysowaniem grafiki:

```
typedef struct
{
    int width;                // szerokość tekstury
    int height;               // wysokość tekstury
    unsigned int texID;       // obiekt tekstury
    unsigned char *data;      // dane tekstury
} texture_t;

...
float angle = 0.0f;          // kąt obrotu torusa
texture_t *envTex;           // tekstura otoczenia
texture_t *skyTex;           // tekstura nieba
...

bool LoadAllTextures()
{
    // ładuje obraz tekstury otoczenia
    envTex = LoadTextureFile("sky-sphere.bmp");
    if (envTex == NULL)
        return false;

    skyTex = LoadTextureFile("sky.bmp");
    if (skyTex == NULL)
        return false;
}
```

```

// tworzy mipmapy z filtrowaniem liniowym dla tekstury otoczenia
glBindTexture(GL_TEXTURE_2D, envTex->texID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, envTex->width, envTex->height,
    GL_RGB, GL_UNSIGNED_BYTE, envTex->data);

// tworzy mipmapy z filtrowaniem liniowym dla tekstury nieba
glBindTexture(GL_TEXTURE_2D, skyTex->texID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, skyTex->width, skyTex->height,
    GL_RGB, GL_UNSIGNED_BYTE, skyTex->data);

return true;
}
...
void Render()
{
    // zwiększa kąt obrotu
    if (angle > 360.0f)
        angle = 0.0f;

    angle = angle + 0.2f;

    // opróżnia bufory ekranu i głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBindTexture(GL_TEXTURE_2D, skyTex->texID);
    glBegin(GL_QUADS);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3f(-200.0f, -200.0f, -120.0f);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3f(200.0f, -200.0f, -120.0f);
        glTexCoord2f(1.0f, 1.0f);
        glVertex3f(200.0f, 200.0f, -120.0f);
        glTexCoord2f(0.0f, 1.0f);
        glVertex3f(-200.0f, 200.0f, -120.0f);
    glEnd();

    // odsuwa obiekt i obraca go względem wszystkich osi układu współrzędnych
    glTranslatef(0.0f, 0.0f, -100.0f);
    glRotatef(angle, 1.0f, 0.0f, 0.0f);
    glRotatef(angle, 0.0f, 1.0f, 0.0f);
    glRotatef(angle, 0.0f, 0.0f, 1.0f);

    // konfiguruje odwzorowanie otoczenia
    glTexGenf(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
    glTexGenf(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);

```

```
// wybiera teksturę otoczenia
glBindTexture(GL_TEXTURE_2D, envTex->texID);

// rysuje torus o promieniu wewnętrznym równym 10 jednostek
// i promieniu zewnętrznym 20 jednostek
auxSolidTorus(10.0f, 20.0f);

glFlush();
SwapBuffers(g_HDC); // przełącza bufor
}
```

Jak łatwo zauważyć, stosowanie odwzorowania otoczenia w OpenGL jest niezwykle proste, o ile pozwoli się maszynie OpenGL automatycznie wyznaczać współrzędne tekstury. Współrzędne te można obliczać indywidualnie, aby uzyskać inne efekty specjalne, ale zagadnienie to nie będzie tutaj omawiane.

Macierze tekstur

W rozdziale 5. omówione zostały sposoby wykonywania przekształceń wierzchołków, takich jak przesunięcie, obrót i skalowanie za pomocą macierzy modelowania. Przedstawiona została także koncepcja stosu macierzy umożliwiająca hierarchiczne tworzenie grafiki.

Takie same możliwości w przypadku tekstur OpenGL udostępnia za pomocą macierzy tekstur i *stosu macierzy tekstur*. Na przykład funkcję `glTranslatef()` można zastosować do przesunięcia tekstury na odpowiednią powierzchnię. Podobnie funkcję `glRotatef()` można wykorzystać do obrotu układu współrzędnych tekstury i uzyskać w efekcie obrót tekstury. Gra „American McGee’s Alice” firmy Electronic Arts and Rogue Entertainment jest najlepszym przykładem niezwykle efektownych efektów, jakie można osiągnąć poprzez manipulację macierzą tekstur.

Wykonywanie operacji na macierzach tekstur jest bardzo łatwe. Stosuje się w tym celu udostępniane przez OpenGL funkcje `glMultMatrix()`, `glPushMatrix()`, `glPopMatrix()` oraz funkcje przekształceń. Najpierw jednak trzeba poinformować maszynę OpenGL, że wykonywane będą operacje na macierzy tekstur:

```
glMatrixMode(GL_TEXTURE);
```

Od tego momentu na macierzy tekstur i stosie macierzy tekstur można wykonywać dowolne operacje. Po ich zakończeniu należy polecić maszynie OpenGL powrót do macierzy modelowania, aby zamiast tekstur przekształcać obiekty.

Poniżej zaprezentowany został fragment kodu ilustrujący sposób wykonywania obrotu tekstury:

```
// opróżnia bufor ekranu i głębi
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
// wybiera tryb operacji na macierzy tekstur
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
```

```

glRotatef(angle, 0.0f, 0.0f, 1.0f); // obraca teksturę
// przywraca tryb operacji na macierzy modelowania
glMatrixMode(GL_MODELVIEW);

glBindTexture(GL_TEXTURE_2D, texID); // wybiera teksturę

// rysuje czworokąt pokryty teksturą
glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f);
    glVertex3f(-20.0f, -20.0f, -40.0f);
    glTexCoord2f(1.0f, 0.0f);
    glVertex3f(20.0f, -20.0f, -40.0f);
    glTexCoord2f(1.0f, 1.0f);
    glVertex3f(20.0f, 20.0f, -40.0f);
    glTexCoord2f(0.0f, 1.0f);
    glVertex3f(-20.0f, 20.0f, -40.0f);
glEnd();

```

Kod ten wybiera najpierw macierz tekstury jako bieżącą macierz. Następnie ładuje do niej macierz jednostkową, zanim zastosuje do bieżącej macierzy funkcję obrotu `glRotatef()`. Na skutek jej użycia tekstura zostaje obrócona o pewien kąt względem osi z. Po wykonaniu obrotu wybiera macierz modelowania jako bieżącą macierz i rysuje czworokąt pokryty obróconą teksturą. Proste, prawda?

Gdyby po czworokącie zostały narysowane kolejne obiekty, to także zostałyby one pokryte obróconą teksturą. Rozwiązanie tego problemu zaprezentowane zostało poniżej:

```

// opróżnia buforę ekranu i głębi
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
// wybiera tryb operacji na macierzy tekstur
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glRotatef(angle, 0.0f, 0.0f, 1.0f); // obraca teksturę
// przywraca tryb operacji na macierzy modelowania
glMatrixMode(GL_MODELVIEW);

glBindTexture(GL_TEXTURE_2D, texID); // wybiera teksturę

// rysuje czworokąt pokryty teksturą
glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f);
    glVertex3f(-20.0f, -20.0f, -40.0f);
    glTexCoord2f(1.0f, 0.0f);
    glVertex3f(20.0f, -20.0f, -40.0f);
    glTexCoord2f(1.0f, 1.0f);
    glVertex3f(20.0f, 20.0f, -40.0f);
    glTexCoord2f(0.0f, 1.0f);
    glVertex3f(-20.0f, 20.0f, -40.0f);
glEnd();

// resetuje macierz tekstur dla następnych obiektów
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glMatrixMode(GL_MODELVIEW);

// rysuje następne obiekty
...

```

Wystarczy jedynie załadować macierz jednostkową do macierzy tekstur, aby wykonane przekształcenia tekstury nie miały wpływu na sposób rysowania tekstur na kolejnych obiektach. Warto wypróbować różne przekształcenia tekstur i uzyskać nowe, ciekawe efekty, które można będzie zastosować w grach.

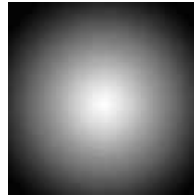
Mapy oświetlenia

Zadaniem *map oświetlenia* jest symulacja statycznego oświetlenia powierzchni. Mapa oświetlenia jest teksturą, która opisuje sposób oświetlenia powierzchni. Mapy oświetlenia znajdują coraz powszechniejsze zastosowanie dzięki wprowadzeniu sprzętowej obsługi tekstur wielokrotnych. Choć mapy oświetlenia stosowane są głównie w celu uzyskania efektu statycznego oświetlenia, można je wykorzystać także do tworzenia cieni.

Rysunek 9.6 przedstawia mapę oświetlenia symulującą efekt strumienia światła padającego na powierzchnię pod kątem prostym. Jeśli pokryty nią zostanie wielokąt, to uzyskany zostanie efekt oświetlenia strumieniem światła prostopadłym do jego powierzchni.

Rysunek 9.6.

Przykład mapy oświetlenia



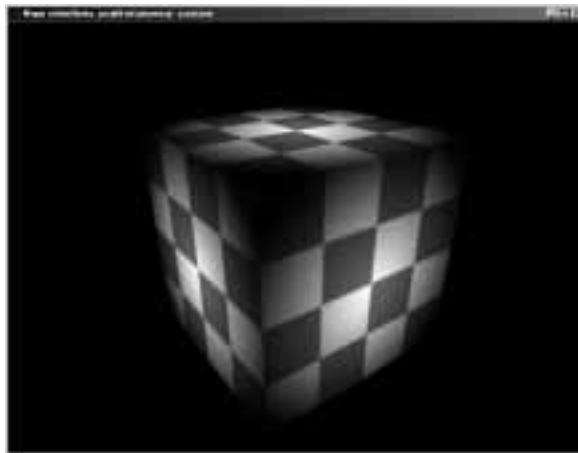
Mapy oświetlenia są teksturami wykorzystującymi jedynie odcienie szarości. Aby uzyskać efekt oświetlenia, mapę taką należy połączyć z teksturą pokrywającą wielokąt.

Stosowanie map oświetlenia

Mapę oświetlenia stosuje się na przykład do uzyskania efektu oświetlenia sześcianu pokrytego teksturą szachownicy. Trzeba nałożyć ją na każdą ze ścian sześcianu, by uzyskać efekt pokazany na rysunku 9.7.

Rysunek 9.7.

Zastosowanie mapy oświetlenia



Ładując mapę oświetlenia z pliku należy zadbać o to, by jej piksele opisane były za pomocą odcieni szarości. Poniżej zamieszczona została zmodyfikowana wersja funkcji `LoadBitmapFile()` ładującej mapę oświetlenia z 24-bitowego pliku *BMP* jako obraz w odcieniach szarości:

```
unsigned char *LoadGrayBitmap(char *filename, BITMAPINFOHEADER *bitmapInfoHeader)
{
    FILE *filePtr;                // wskaźnik pozycji pliku
    BITMAPFILEHEADER bitmapFileHeader; // nagłówek pliku
    unsigned char *bitmapImage;    // dane obrazu
    int imageIdx = 0;              // licznik pikseli
    unsigned char tempRGB;         // zmienna zamiany składowych

    unsigned char *grayImage;     // obraz w odcieniach szarości
    int grayIdx;                  // licznik pikseli w odcieniach szarości

    // otwiera plik w trybie "read binary"
    filePtr = fopen(filename, "rb");
    if (filePtr == NULL)
        return NULL;

    // wczytuje nagłówek pliku
    fread(&bitmapFileHeader, sizeof(BITMAPFILEHEADER), 1, filePtr);

    // sprawdza, czy jest to plik formatu BMP
    if (bitmapFileHeader.bfType != BITMAP_ID)
    {
        fclose(filePtr);
        return NULL;
    }

    // wczytuje nagłówek obrazu
    fread(bitmapInfoHeader, sizeof(BITMAPINFOHEADER), 1, filePtr);

    // ustawia wskaźnik pozycji pliku na początku danych obrazu
    fseek(filePtr, bitmapFileHeader.bfOffBits, SEEK_SET);

    // przydziela pamięć buforowi obrazu
    bitmapImage = (unsigned char*)malloc(bitmapInfoHeader->biSizeImage);

    // przydziela pamięć potrzebną do przechowania obrazu
    // w odcieniach szarości,
    // trzy razy mniejszą niż rozmiar obrazu RGB

    grayImage = (unsigned char*)malloc(bitmapInfoHeader->biSizeImage / 3);

    // sprawdza, czy udało się przydzielić pamięć
    if (!bitmapImage)
    {
        free(bitmapImage);
        fclose(filePtr);
        return NULL;
    }

    // wczytuje dane obrazu
    fread(bitmapImage, 1, bitmapInfoHeader->biSizeImage, filePtr);
}
```

```

// sprawdza, czy dane zostały wczytane
if (bitmapImage == NULL)
{
    fclose(filePtr);
    return NULL;
}

grayIdx = 0;
for (imageIdx = 0; imageIdx < bitmapInfoHeader->biSizeImage; imageIdx+=3)
{
    grayImage[grayIdx] = bitmapImage[imageIdx];
    grayIdx++;
}

free(bitmapImage);
// zamyka plik i zwraca wskaźnik bufora zawierającego wczytany obraz
fclose(filePtr);
return grayImage;
}

```

Po załadowaniu mapy oświetlenia można posługiwać się nią jak zwykłą teksturą z jedną różnicą: format jej trzeba zawsze definiować jako `GL_LUMINANCE`, a nie jako `GL_RGB`. A oto przykład:

```

// mapa oświetlenia jako mipmapa
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_LUMINANCE, width, height, GL_LUMINANCE,
    GL_UNSIGNED_BYTE, lightmapData);
// lub pojedyncza tekstura
glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, width, height, 0, GL_LUMINANCE,
    GL_UNSIGNED_BYTE, lightmapData);

```

Aby zastosować mapę oświetlenia, należy posłużyć się mechanizmem tekstur wielokrotnych i włączyć tryb nakładania tekstury `GL_MODULATE`. Trzeba zatem przyjrzeć się najważniejszym fragmentom kodu tego programu:

```

typedef struct
{
    int width; // szerokość tekstury
    int height; // wysokość tekstury
    unsigned int texID; // obiekt tekstury
    unsigned char *data; // dane tekstury
} texture_t;

// tekstury
texture_t *checkerTex; // mapa oświetlenia
texture_t *lightmapTex; // tekstura szachownicy

// funkcje tworzenia tekstur wielokrotnych
PFNGLMULTITEXCOORD2FARBPROC glMultiTexCoord2fARB = NULL;
PFNGLACTIVETEXTUREARBPROC glActiveTextureARB = NULL;
PFNGLCLIENTACTIVETEXTUREARBPROC glClientActiveTextureARB = NULL;
int maxTextureUnits = 0;

bool InitMultiTex()
{
    char *extensionStr; // lista dostępnych rozszerzeń

```



```

extensionStr = (char*)glGetString(GL_EXTENSIONS);

if (extensionStr == NULL)
    return false;

if (strstr(extensionStr, "GL_ARB_multitexture"))
{
    // pobiera dopuszczalną liczbę jednostek tekstur
    glGetIntegerv(GL_MAX_TEXTURE_UNITS_ARB, &maxTextureUnits);

    // pobiera adresy funkcji tworzenia tekstur wielokrotnych
    glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC)
        wglGetProcAddress("glMultiTexCoord2fARB");
    glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC)
        wglGetProcAddress("glActiveTextureARB");
    glClientActiveTextureARB = (PFNGLCLIENTACTIVETEXTUREARBPROC)
        wglGetProcAddress("glClientActiveTextureARB");

    return true;
}
else
    return false;
}

```

Funkcja InitMultiTex() sprawdza dostępność mechanizmu tworzenia tekstur wielokrotnych dla danej implementacji OpenGL. Tym razem wykorzystuje ona funkcję `strstr()`, która zwraca wartość `NULL`, jeśli drugi z jej parametrów nie jest podłańcuchem pierwszego.

Kolejną z funkcji jest omówiona wcześniej funkcja ładowania mapy oświetlenia z pliku:

```

unsigned char *LoadGrayBitmap(char *filename, BITMAPINFOHEADER *bitmapInfoHeader)
{
    FILE *filePtr;                // wskaźnik pozycji pliku
    BITMAPFILEHEADER bitmapFileHeader; // nagłówek pliku
    unsigned char *bitmapImage;    // dane obrazu
    int imageIdx = 0;              // licznik pikseli
    unsigned char tempRGB;         // zmienna zamiany składowych

    unsigned char *grayImage;     // obraz w odcieniach szarości
    int grayIdx;                  // licznik pikseli w odcieniach szarości

    // otwiera plik w trybie "read binary"
    filePtr = fopen(filename, "rb");
    if (filePtr == NULL)
        return NULL;

    // wczytuje nagłówek pliku
    fread(&bitmapFileHeader, sizeof(BITMAPFILEHEADER), 1, filePtr);

    // sprawdza, czy jest to plik formatu BMP
    if (bitmapFileHeader.bfType != BITMAP_ID)
    {
        fclose(filePtr);
        return NULL;
    }
}

```

```
// wczytuje nagłówek obrazu
fread(bitmapInfoHeader, sizeof(BITMAPINFOHEADER), 1, filePtr);

// ustawia wskaźnik pozycji pliku na początku danych obrazu
fseek(filePtr, bitmapFileHeader.bfOffBits, SEEK_SET);

// przydziela pamięć buforowi obrazu
bitmapImage = (unsigned char*)malloc(bitmapInfoHeader->biSizeImage);

// przydziela pamięć potrzebną do przechowania obrazu
// w odcieniach szarości,
// trzy razy mniejszą niż rozmiar obrazu RGB

grayImage = (unsigned char*)malloc(bitmapInfoHeader->biSizeImage / 3);

// sprawdza, czy udało się przydzielić pamięć
if (!bitmapImage)
{
    free(bitmapImage);
    fclose(filePtr);
    return NULL;
}

// wczytuje dane obrazu
fread(bitmapImage, 1, bitmapInfoHeader->biSizeImage, filePtr);

// sprawdza, czy dane zostały wczytane
if (bitmapImage == NULL)
{
    fclose(filePtr);
    return NULL;
}

grayIdx = 0;
for (imageIdx = 0; imageIdx < bitmapInfoHeader->biSizeImage; imageIdx+=3)
{
    grayImage[grayIdx] = bitmapImage[imageIdx];
    grayIdx++;
}

free(bitmapImage);
// zamyka plik i zwraca wskaźnik bufora zawierającego wczytany obraz
fclose(filePtr);
return grayImage;
}
```

Następna funkcja, `LoadLightmap()`, przypomina funkcję `LoadTextureFile()` z poprzednich przykładów, ale korzysta z usług funkcji ładowania mapy oświetlenia `LoadGrayBitmap()`:

```
texture_t *LoadTextureFile(char *filename)
{
    BITMAPINFOHEADER texInfo;
    texture_t *thisTexture;
```

```

// przydziela pamięć strukturze typu texture_t
thisTexture = (texture_t*)malloc(sizeof(texture_t));
if (thisTexture == NULL)
    return NULL;

// ładuje obraz tekstury i sprawdza poprawne wykonanie tej operacji
thisTexture->data = LoadGrayBitmap(filename, &texInfo);
if (thisTexture->data == NULL)
{
    free(thisTexture);
    return NULL;
}

// umieszcza informacje o szerokości i wysokości tekstury
thisTexture->width = texInfo.biWidth;
thisTexture->height = texInfo.biHeight;

// tworzy obiekt tekstury
glGenTextures(1, &thisTexture->texID);

return thisTexture;
}

```

Również funkcja `LoadAllTextures()`, która prezentowana jest poniżej, została zmodyfikowana pod kątem użycia mapy oświetlenia, dla której — w odróżnieniu od zwykłych tekstur — należy zastosować wartości `GL_LUMINANCE` i `GL_MODULATE`:

```

bool LoadAllTextures()
{
    // ładuje teksturę szachownicy
    checkerTex = LoadTextureFile("chess.bmp");
    if (checkerTex == NULL)
        return false;

    // ładuje mapę oświetlenia
    lightmapTex = LoadLightmap("lmap.bmp");
    if (lightmapTex == NULL)
        return false;

    // tworzy teksturę szachownicy
    glBindTexture(GL_TEXTURE_2D, checkerTex->texID);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
    glubuild2Dmipmaps(GL_TEXTURE_2D, GL_RGB, checkerTex->width, checkerTex->height,
        GL_RGB, GL_UNSIGNED_BYTE, checkerTex->data);

    // tworzy mapę oświetlenia
    glBindTexture(GL_TEXTURE_2D, lightmapTex->texID);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    glubuild2Dmipmaps(GL_TEXTURE_2D, GL_LUMINANCE, lightmapTex->width,
        lightmapTex->height, GL_LUMINANCE, GL_UNSIGNED_BYTE, lightmapTex->data);
}

```

```
// wybiera pierwszą jednostkę tekstury
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, checkerTex->texID); // wiąże teksturę szachownicy

// wybiera drugą jednostkę tekstury
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, lightmapTex->texID); // wiąże mapę oświetlenia

return true;
}
```

Pozostała część kodu jest identyczna z pierwszym przykładem programu tworzenia tekstur wielokrotnych.

Wieloprzebiegowe tekstury wielokrotne

Brak obsługi mechanizmu tekstur wielokrotnych przez konkretną implementację OpenGL nie oznacza, że nie można uzyskać takich efektów innym sposobem. Rozwiązanie polega na symulacji tekstur wielokrotnych za pomocą doboru odpowiednich funkcji łączenia i tworzeniu końcowego efektu w wielu przebiegach rysowania grafiki. Metoda ta jest zwykle wolniejsza od tekstur wielokrotnych, które coraz częściej obsługiwane są sprzętowo, ale umożliwia uzyskanie podobnych efektów.

Wieloprzebiegowe tworzenie sceny polega na odpowiedniej zmianie zawartości bufora głębi i trybów łączenia kolorów w kolejnych przebiegach. Aby uzyskać taki efekt jak w przypadku tekstur wielokrotnych, należy wykonać poniższy fragment kodu:

```
// pierwszy przebieg rysowania
glBindTexture(GL_TEXTURE_2D, tex1);
DrawTexturedCube(0.0f, 0.0f, 0.0f);

// drugi przebieg rysowania
glEnable(GL_BLEND); // włącza łączenie kolorów
glDepthMask(GL_FALSE); // wyłącza zapis do bufora głębi
glDepthFunc(GL_EQUAL);

glBlendFunc(GL_ZERO, GL_SRC_COLOR);

glBindTexture(GL_TEXTURE_2D, checkerTex->texID);
DrawTexturedCube(0.0f, 0.0f, 0.0f);

// przywraca poprzedni stan maszyny OpenGL
glDepthMask(GL_TRUE);
glDepthFunc(GL_LESS);
glDisable(GL_BLEND);
```

Poprzez zmianę sposobu łączenia kolorów uzyskać można efekt tekstury wielokrotnej. Poniższe fragmenty kodu pokazują, w jaki sposób można uzyskać w wielu przebiegach taki sam efekt jak w przypadku pierwszego przykładu tekstur wielokrotnych:

```

bool LoadAllTextures()
{
    // ładuje obraz pierwszej tekstury
    smileTex = LoadTextureFile("smile.bmp");
    if (smileTex == NULL)
        return false;

    // ładuje obraz drugiej tekstury
    checkerTex = LoadTextureFile("chess.bmp");
    if (checkerTex == NULL)
        return false;

    // tworzy pierwszą teksturę
    glBindTexture(GL_TEXTURE_2D, smileTex->texID);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
    glBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, smileTex->width, smileTex->height,
        GL_RGB, GL_UNSIGNED_BYTE, smileTex->data);

    // tworzy drugą teksturę
    glBindTexture(GL_TEXTURE_2D, checkerTex->texID);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    glBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, checkerTex->width, checkerTex->height,
        GL_RGB, GL_UNSIGNED_BYTE, checkerTex->data);

    return true;
}

void DrawTexturedCube(float xPos, float yPos, float zPos)
{
    glPushMatrix();
    glTranslatef(xPos, yPos, zPos);
    glBegin(GL_QUADS);
        glNormal3f(0.0f, 1.0f, 0.0f);           // górna ściana

        glTexCoord2f(1.0f, 0.0f);
        glVertex3f(0.5f, 0.5f, 0.5f);

        glTexCoord2f(1.0f, 1.0f);
        glVertex3f(0.5f, 0.5f, -0.5f);

        glTexCoord2f(0.0f, 1.0f);
        glVertex3f(-0.5f, 0.5f, -0.5f);

        glTexCoord2f(0.0f, 0.0f);
        glVertex3f(-0.5f, 0.5f, 0.5f);
    glEnd();
    glBegin(GL_QUADS);
        glNormal3f(0.0f, 0.0f, 1.0f);           // przednia ściana

        glTexCoord2f(1.0f, 1.0f);
        glVertex3f(0.5f, 0.5f, 0.5f);

```

```
glTexCoord2f(0.0f, 1.0f);
glVertex3f(-0.5f, 0.5f, 0.5f);

glTexCoord2f(0.0f, 0.0f);
glVertex3f(-0.5f, -0.5f, 0.5f);

glTexCoord2f(1.0f, 0.0f);
glVertex3f(0.5f, -0.5f, 0.5f);
glEnd();
glBegin(GL_QUADS);
glNormal3f(1.0f, 0.0f, 0.0f);           // prawa ściana

glTexCoord2f(0.0f, 1.0f);
glVertex3f(0.5f, 0.5f, 0.5f);

glTexCoord2f(0.0f, 0.0f);
glVertex3f(0.5f, -0.5f, 0.5f);

glTexCoord2f(1.0f, 0.0f);
glVertex3f(0.5f, -0.5f, -0.5f);

glTexCoord2f(1.0f, 1.0f);
glVertex3f(0.5f, 0.5f, -0.5f);
glEnd();
glBegin(GL_QUADS);
glNormal3f(-1.0f, 0.0f, 0.0f);         // lewa ściana

glTexCoord2f(1.0f, 1.0f);
glVertex3f(-0.5f, 0.5f, 0.5f);

glTexCoord2f(0.0f, 1.0f);
glVertex3f(-0.5f, 0.5f, -0.5f);

glTexCoord2f(0.0f, 0.0f);
glVertex3f(-0.5f, -0.5f, -0.5f);

glTexCoord2f(1.0f, 0.0f);
glVertex3f(-0.5f, -0.5f, 0.5f);
glEnd();
glBegin(GL_QUADS);
glNormal3f(0.0f, -1.0f, 0.0f);         // dolna ściana

glTexCoord2f(1.0f, 0.0f);
glVertex3f(-0.5f, -0.5f, 0.5f);

glTexCoord2f(1.0f, 1.0f);
glVertex3f(-0.5f, -0.5f, -0.5f);

glTexCoord2f(0.0f, 1.0f);
glVertex3f(0.5f, -0.5f, -0.5f);

glTexCoord2f(0.0f, 0.0f);
glVertex3f(0.5f, -0.5f, 0.5f);
glEnd();
glBegin(GL_QUADS);
glNormal3f(0.0f, 0.0f, -1.0f);         // tylna ściana
```

```

        glTexCoord2f(0.0f, 0.0f);
        glVertex3f(0.5f, -0.5f, -0.5f);

        glTexCoord2f(1.0f, 0.0f);
        glVertex3f(-0.5f, -0.5f, -0.5f);

        glTexCoord2f(1.0f, 1.0f);
        glVertex3f(-0.5f, 0.5f, -0.5f);

        glTexCoord2f(0.0f, 1.0f);
        glVertex3f(0.5f, 0.5f, -0.5f);
    glEnd();
    glPopMatrix();
}

void Render()
{
    radians = float(PI*(angle-90.0f)/180.0f);

    // wyznacza położenie kamery
    cameraX = lookX + sin(radians)*mouseY;
    cameraZ = lookZ + cos(radians)*mouseY;
    cameraY = lookY + mouseY / 2.0f;

    // wycelowuje kamerę w punkt (0,0,0)
    lookX = 0.0f;
    lookY = 0.0f;
    lookZ = 0.0f;

    // opróżnia bufor ekranu i głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    // umieszcza kamerę
    gluLookAt(cameraX, cameraY, cameraZ, lookX, lookY, lookZ, 0.0, 1.0, 0.0);

    // skaluje sześcian do rozmiarów 15x15x15
    glScalef(15.0f, 15.0f, 15.0f);

    // pierwszy przebieg rysowania
    glBindTexture(GL_TEXTURE_2D, smileTex->texID);
    DrawTexturedCube(0.0f, 0.0f, 0.0f);
    // drugi przebieg rysowania
    glEnable(GL_BLEND); // włącza łączenie kolorów
    glDepthMask(GL_FALSE); // wyłącza zapis do bufora głębi
    glDepthFunc(GL_EQUAL);

    glBlendFunc(GL_ZERO, GL_SRC_COLOR);

    glBindTexture(GL_TEXTURE_2D, checkerTex->texID);
    DrawTexturedCube(0.0f, 0.0f, 0.0f);

    // przywraca poprzedni stan maszyny OpenGL
    glDepthMask(GL_TRUE);
    glDepthFunc(GL_LESS);
    glDisable(GL_BLEND);
    glFlush();
    SwapBuffers(g_HDC); // przełącza bufor
}

```

Efekt uzyskany w wyniku wieloprzebiegowego tworzenia tekstur nie będzie się różnić od efektów uzyskanych za pomocą tekstur wielokrotnych. Zwykle jednak tworzony będzie wolniej, ponieważ mechanizm tekstur wielokrotnych korzysta z możliwości ich sprzętowej obsługi. Jednak tworzenie tekstur wielokrotnych w wielu przebiegach pozwala na dodatkowe eksperymentowanie z różnymi efektami uzyskiwanymi przez zmianę funkcji łączenia kolorów.

Podsumowanie

OpenGL umożliwia pokrycie powierzchni wielokątą sekwencją tekstur tworzącą *teksturę wielokrotną*.

W procesie tworzenia tekstur wielokrotnych wyróżnić można cztery etapy: sprawdzenie dostępności tekstur wielokrotnych, uzyskanie wskaźników funkcji rozszerzenia, stworzenie jednostki tekstury i określenie współrzędnych tekstur.

Odwzorowanie otoczenia pozwala rysować obiekty odbijające otaczający je świat na podobieństwo wypolerowanej srebrnej kuli.

Stos macierzy tekstur umożliwia wykonywanie przesunięć, obrotów i skalowań tekstur. Przekształcenia tekstur umożliwiają uzyskanie ciekawych efektów.

Mapy oświetlenia służą do symulacji statycznego oświetlenia obiektu za pomocą odpowiedniej tekstury reprezentującej światło padające na daną powierzchnię.