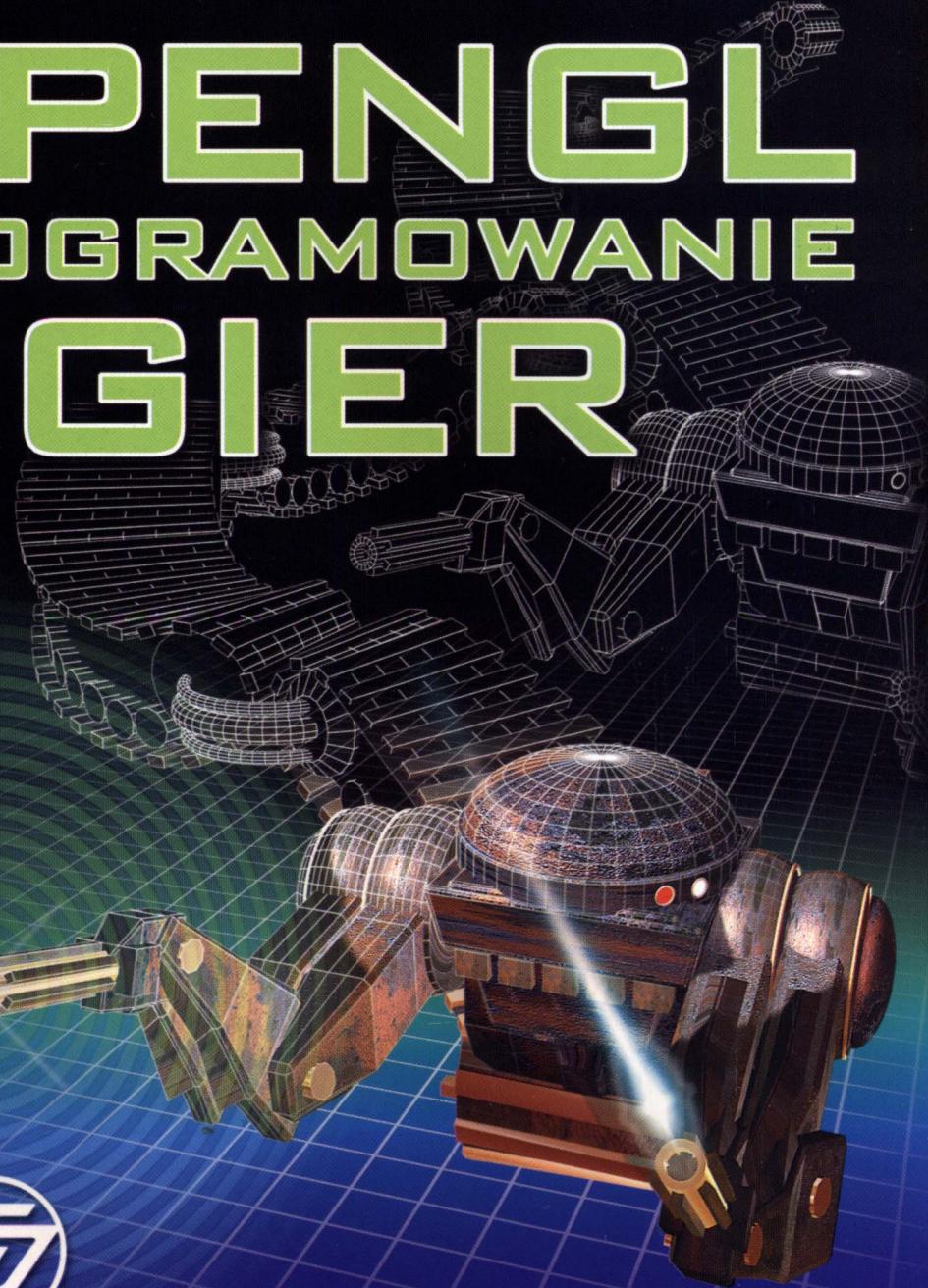


PROGRAMOWANIE GIER

ZAWIERA CD



# OPENGL PROGRAMOWANIE GIER



KEVIN HAWKINS  
DAVE ASTLE

[helion.pl](http://helion.pl)

Helion



# Kompletny przewodnik po programowaniu trójwymiarowych gier

- Dzięki OpenGL nauczysz się tworzyć własne gry 3D
- Poznasz tajemnice takich przebojów jak Quake czy Doom III
- Używając DirectX Audio wzbogacisz gry o efekty dźwiękowe
- Zdobędziesz podstawową wiedzę na temat programowania w Windows
- Nauczysz się tworzyć i wykorzystywać modele postaci

*Osobiście pojawienie się tej książki przyprawia mnie o dreszcz emocji. Po raz pierwszy programiści otrzymują książkę poświęconą programowaniu gier za pomocą OpenGL. Jeśli nawet nie viązasz swojej kariery zawodowej z tworzeniem gier komputerowych, to lektura tej książki pomoże Ci lepiej zrozumieć technologię, która wykorzystywana jest do tworzenia współczesnych gier, a także zdobyć umiejętności, które mogą być przydatne w innych dziedzinach programowania.”*

Mark J. Kilgard, NVIDIA Corporation  
Autor „OpenGL Programming for the X Window System” oraz pakietu OpenGL Utility Toolkit (GLUT)

**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 wyobrażni 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 programiście 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 DirectX
- Wykorzystanie DirectX Audio
- Trójwymiarowe modele postaci

Szukasz dobrej książki? Zamów najnowszy katalog: <http://helion.pl/katalog>  
Zamów informacje o nowościach: <http://helion.pl/novosci>  
Zamów cennik: <http://helion.pl/cennik>

ISBN 83-7361-035-9

## Wydawnictwo Helion

ul. Chopina 6, 44-100 Gliwice  
 skr. poczt. 462, 44-100 Gliwice  
 (32) 230-98-63, (32) 231-22-19  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
<http://helion.pl>



9 788373 610354

**helion.pl**  
księgarnia  
internetowa

Cena 72,00 zł

Informatyka w najlepszym wydaniu

PROGRAMOWANIE GIER

# OPENGL PROGRAMOWANIE GIER

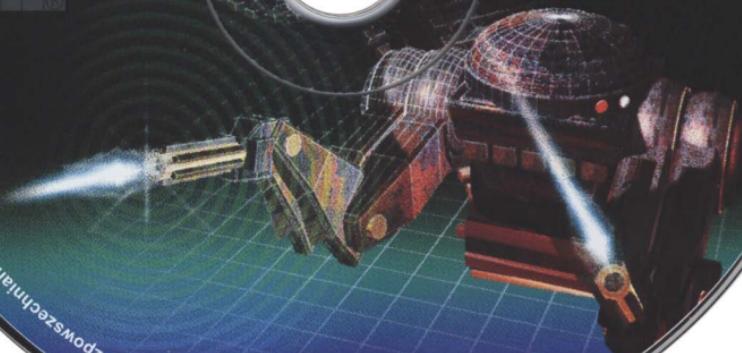
helion.pl

Helion



ISBN 83-7361-035-9  
OpenGL. Programowanie gier  
Wydawnictwo Helion  
www.helion.pl

Księgarnia internetowa <http://helion.pl>



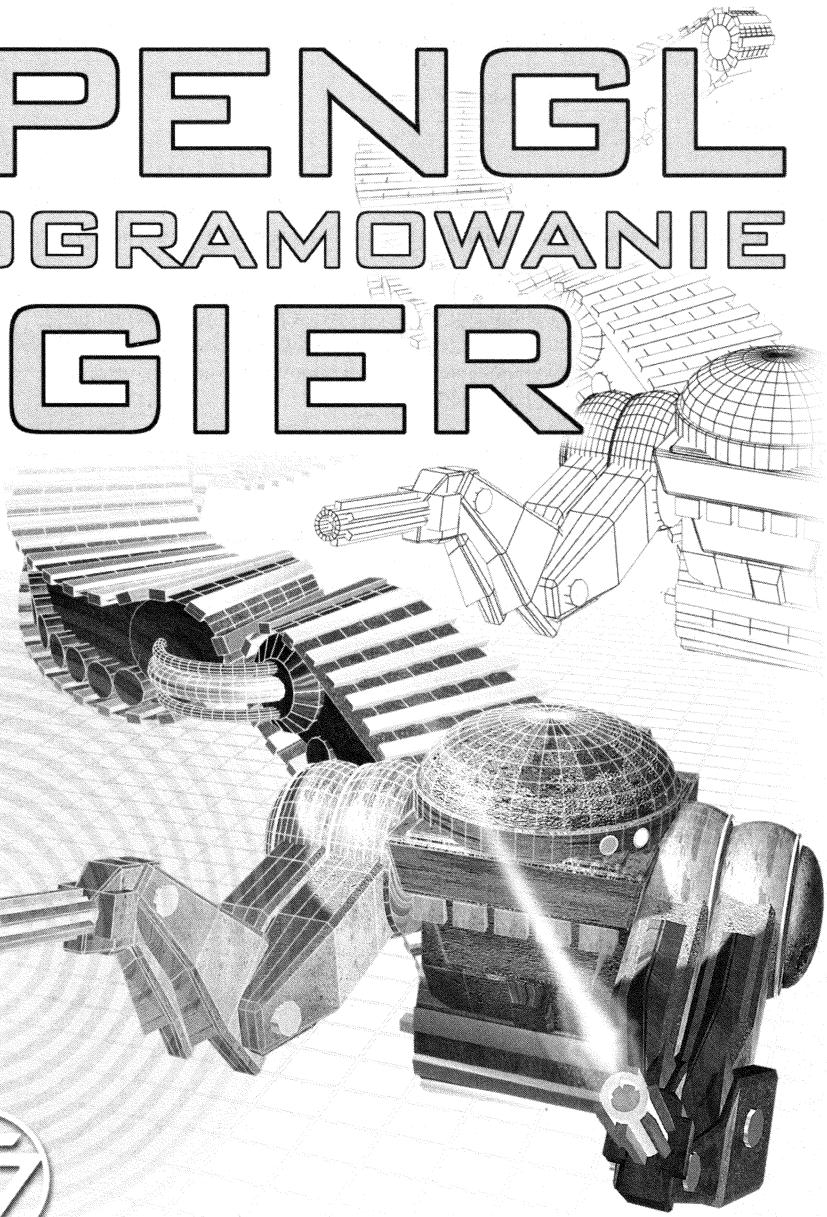
CD-ROM reklamowy przeznaczony do rozpowszechniania wylatania z klaszki

**PROGRAMOWANIE GIER**

ZAWIERA CD



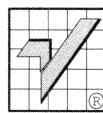
# **OPENGL PROGRAMOWANIE GIER**



**KEVIN HAWKINS  
DAVE ASTLE**

**helion.pl**

**Helion**



Tytuł oryginału: OpenGL Game Programming

Tłumaczenie: Jaromir Senczyk

ISBN: 83-7361-035-9

Copyright © 2002 Course Technology

Translation copyright © 2003 by Wydawnictwo Helion.

Polish language edition published by Wydawnictwo Helion.

Copyright © 2003

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 jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopowanie 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 właścicielami.

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

Wydawnictwo HELION  
ul. Chopina 6, 44-100 GLIWICE  
tel. (32) 231-22-19, (32) 230-98-63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

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

Printed in Poland.

Druk: Zakład Poligraficzny W. Wiliński, Gliwice, ul. Chopina 6, tel./fax 231-32-16

*Mojej rodzinie i przyjaciołom*  
— Kevin Hawkins

*Lissie*  
— Dave Astle

# Podziękowania

Wiele osób pomogło mi napisać tę książkę. Za okazane mi wsparcie chciałbym podziękować całej mojej rodzinie i przyjaciółom. Dziękuję Ci, Karo, za to, że wytrzymałaś ze mną przez te wszystkie miesiące, zwłaszcza wtedy, kiedy było mi szczególnie ciężko. Dziękuję też za wsparcie Toddowi i Tuckerowi. A także kolegom z zespołu i trenerom za to, że pytając mnie codziennie o postęp w pracy nad książką dodatkowo motywowali mnie do wysiłku. Szczególne podziękowania należą się Dave'owi, który mogąc poświecić mniej czasu naszemu projektowi zdecydował się jednak pomóc mi napisać tę książkę.

Dziękuję wszystkim pracownikom wydawnictwa Prima, a zwłaszcza Jody Kennen i Kelly Talbot, które umożliwiły mi napisanie tej książki i wspierały podczas jej tworzenia. Dziękuję André LaMothe'owi, który udzielał nam rad dotyczących kształtu naszej pracy. A także Ernestowi „Tanstaaflowi” Pazerze, który uświadomił mi, że szansę zrobienia niektórych rzeczy otrzymujemy tylko raz.

Na końcu pragnę podziękować wszystkim, którzy przyczynili się do poszerzenia mojej wiedzy podczas pisania tej książki. Na szczególne podziękowania zasłużyli: Bas Kuennen, Mark Kilgard, Mark DeLoura, André LaMothe, Richard „Keebler” Benson, Jordan „jdm” Maynard, Nate Miller, Nate Robins i Jeff „NeHe” Molofee.

— Kevin Hawkins

Powstanie tej książki możliwe było dzięki pracy wielu osób. Największe podziękowania należą się oczywiście Kevinowi, który poświęcił jej więcej czasu i wysiłku ode mnie. Większość materiału zawartego w tej książce jest owocem jego pracy. Chciałbym także podziękować mojej żonie za poświęcenie, które umożliwiło mi pracę nad tą книгą, mimo że w tym czasie była w stanie błogosławionym. A także czwórkę moich dzieci: Rebecę, Evanowi, Elisie i Tylerowi za to, że pozwalały mi pracować, a także za to, że wciągały mnie do zabawy. Dziękuję także Nathanowi, który znajdował się wtedy w brzuszu mamy. Szczególne podziękowania należą się Madonne za pomoc okazaną mi przez te wszystkie lata.

Pozostanę także dłużnikiem Ernesta „Tanstaafla” Pazerzy, który namówił mnie do napisania tej książki. Mimo związanego z tym stresu, było to jednak niezwykłe doświadczenie. Chciałbym także podziękować wszystkim pracownikom wydawnictwa Prima, a zwłaszcza Jody Kennen i Kelly Talbot za ich stałe wsparcie oraz André LaMothe'owi za konsultacje dotyczące kształtu naszego przedsięwzięcia.

Na końcu chcę podziękować wszystkim, którym zawdzięczam wiedzę, umożliwiającą mi napisanie tej książki: Chuckowi Hansenowi (mojemu wykładowcy grafiki na Uniwersytecie Utah), Jeffowi „NeHe” Molofee (który stworzył dysk CD towarzyszący tej książce) oraz Markowi Kilgardowi, Nate Robins, Nate Miller, Scottowi „Druidowi” Franke i Richowi „Keeblerowi” Bensonowi.

— Dave Astle

# Spis treści

<b>List od wydawcy serii.....</b>	<b>15</b>
<b>O Autorach.....</b>	<b>17</b>
<b>Przedmowa.....</b>	<b>19</b>
<b>Wprowadzenie .....</b>	<b>21</b>
<b>Część I Wprowadzenie do OpenGL i DirectX .....</b>	<b>23</b>
<b>Rozdział 1. OpenGL i DirectX .....</b>	<b>25</b>
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
<b>Rozdział 2. Korzystanie z OpenGL w systemie Windows .....</b>	<b>39</b>
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
Pieniężne aplikacje OpenGL .....	67
Podsumowanie .....	69
<b>Rozdział 3. Przegląd teorii grafiki trójwymiarowej .....</b>	<b>71</b>
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
<b>Część II Korzystanie z OpenGL.....</b>	<b>91</b>
<b>Rozdział 4. Maszyna stanów OpenGL i podstawowe elementy grafiki .....</b>	<b>93</b>
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
<b>Rozdział 5. Przekształcenia układu współrzędnych i macierze OpenGL.....</b>	<b>111</b>
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
<b>Rozdział 6. Kolory, łączenie kolorów i oświetlenie .....</b>	<b>141</b>
Czym są kolory?.....	141
Kolory w OpenGL.....	142
Głębia koloru .....	143
Sześciyan 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
Tłumienie .....	160
Strumień światła.....	160
Definiowanie materiałów .....	163
Modele oświetlenia .....	164
Efekty odbicia .....	166
Ruchome źródła światła.....	167
Łączanie kolorów .....	173
Przezroczystość.....	174
Podsumowanie .....	179
<b>Rozdział 7. Mapy bitowe i obrazy w OpenGL.....</b>	<b>181</b>
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
<b>Rozdział 8. Odwzorowania tekstur .....</b>	<b>199</b>
Odwzorowania tekstur.....	199
Przykład zastosowania tekstury.....	200
Mapy tekstu.....	205
Tekstury dwuwymiarowe .....	205
Tekstury jednowymiarowe .....	206
Tekstury trójwymiarowe .....	206
Obiekty tekstu .....	207
Tworzenie nazwy tekstury .....	207
Tworzenie i stosowanie obiektów tekstu .....	207
Filtrowanie tekstu.....	208
Funkcje tekstu .....	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
Uksztaltowanie terenu .....	223
Objaśnienia .....	223
Implementacja.....	227
Podsumowanie .....	235
<b>Rozdział 9. Zaawansowane odwzorowania tekstur.....</b>	<b>237</b>
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
<b>Rozdział 10. Listy wyświetlania i tablice wierzchołków .....</b>	<b>267</b>
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: uksztaltowanie terenu po raz drugi.....	281
Podsumowanie .....	284
<b>Rozdział 11. Wyświetlanie tekstów.....</b>	<b>285</b>
Czcionki rastrowe.....	285
Czcionki konturowe .....	289
Czcionki pokryte teksturową.....	292
Podsumowanie .....	299
<b>Rozdział 12. Bufory OpenGL.....</b>	<b>301</b>
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
<b>Rozdział 13. Powierzchnie drugiego stopnia .....</b>	<b>327</b>
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
<b>Rozdział 14. Krzywe i powierzchnie .....</b>	<b>337</b>
Reprezentacja krzywych i powierzchni .....	337
Równania parametryczne .....	338
Punkty kontrolne i pojęcie ciągłości.....	338
Ewaluator .....	339
Siatka równoodległa.....	342
Powierzchnie .....	343
Pokrywanie powierzchni teksturami .....	346
Powierzchnie B-sklejane .....	350
Podsumowanie .....	354
<b>Rozdział 15. Efekty specjalne .....</b>	<b>355</b>
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
Oddziaływanie .....	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: DirectX.....	<b>391</b>
Dlaczego DirectX?.....	391
Komunikaty systemu Windows.....	391
Interfejs programowy Win32 .....	394
Win32 i obsługa klawiatury .....	394
Win32 i obsługa manipulatorów .....	396
DirectInput .....	397
Inicjacja interfejsu DirectX.....	397
Wartości zwracane przez funkcje DirectX.....	398
Korzystanie z DirectX.....	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
Odzworzowania akcji.....	410
Tworzenie podsystemu wejścia .....	410
Przykład zastosowania systemu wejścia.....	418
Podsumowanie .....	420

<b>Rozdział 17. Zastosowania DirectX Audio .....</b>	<b>421</b>
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 bufory .....	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
<b>Rozdział 18. Modele trójwymiarowe .....</b>	<b>469</b>
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
<b>Rozdział 19. Modelowanie fizycznych właściwości świata.....</b>	<b>505</b>
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 równej pochyłej.....	514
Modelowanie świata rzeczywistego .....	516
Rozkład problemu .....	516
Czas.....	517
Wektor .....	521
Płaszczyzna .....	526
Obiekt.....	529
Zderzenia obiektów .....	531
Sfery ograniczeń .....	532
Prostopadłościany ograniczeń .....	533
Zderzenia płaszczyzn .....	535
Obsługa zderzenia.....	538
Przykład: hokej .....	538
Świat hokeja.....	539
Lodowisko .....	539
Krążek i zderzenia.....	544
Gracz.....	550
Kompletowanie programu.....	553
Podsumowanie .....	559
<b>Rozdział 20. Tworzenie szkieletu gry.....</b>	<b>561</b>
Architektura szkieletu SimpEngine .....	561
Zarządzanie danymi za pomocą obiektów klasy CNode .....	562
Zarządzanie obiektyami: 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

<b>Rozdział 21. Piszemy grę: „Czas zabijania” .....</b>	<b>585</b>
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
<b>Dodatki .....</b>	<b>597</b>
<b>Dodatek A Zasoby sieci Internet.....</b>	<b>599</b>
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
<b>Dodatek B Dysk CD.....</b>	<b>603</b>
Struktura plików na dysku CD .....	603
Wymagania sprzętowe.....	603
Instalacja .....	604
Typowe problemy i ich rozwiązywanie.....	604
<b>Skorowidz.....</b>	<b>605</b>

# List od wydawcy serii

„OpenGL. Programowanie gier” jest jedną z pierwszych książek serii Prima Game Development i prawdopodobnie jedną z bardziej ambitnych. Napisano już wiele książek poświęconych programowaniu OpenGL i programowaniu grafiki trójwymiarowej, ale żadna z nich nie była poświęcona wykorzystaniu możliwości OpenGL do programowania gier. „OpenGL. Programowanie gier” wypełnia tę lukę, a nawet stawia sobie bardziej ambitne cele. Programowanie w OpenGL pozostaje bowiem w dość luźnym związku z tworzeniem aplikacji na platformie Windows. I właśnie ta książka integruje aspekty programowania aplikacji Windows z programowaniem OpenGL w kontekście tworzenia gier. Za to należą się jej twórcom, Kevinowi Hawkinsowi i Dave’owi Astle’owi (współzałożycielom GamDev.net), duże brawa.

Półki mojej biblioteki zapełnia wiele książek poświęconych OpenGL. Jednak z punktu widzenia programisty gier żadna z nich nie odpowiadała w pełni moim potrzebom. Właśnie na taką książkę jak niniejsza pozycja oczekiwali programiści gier. „OpenGL. Programowanie gier” pozwala rozpocząć przygodę z programowaniem gier nawet nowicjuszom. Na wstępie dokonany zostaje krótki przegląd problematyki gier, a za nim pojawia się porównanie specyfikacji OpenGL i DirectX. Następnie przedstawione zostają zasady tworzenia grafiki trójwymiarowej w systemie Windows z wykorzystaniem OpenGL zamiast Direct3D. Po zaprezentowaniu podstawowej wiedzy na temat tworzenia aplikacji OpenGL na platformie Windows nadchodzi właściwy czas na omówienie teorii grafiki trójwymiarowej i wyjaśnienie poprzez odpowiednie ilustracje takich zagadnień jak przekształcenia, oświetlenie, obcinanie, rzutowanie i odwzorowywanie tekstur. Książka ta byłaby warta uwagi nawet wtedy, gdy kończyłaby na tym omówienie grafiki trójwymiarowej. Ma jednak do zaoferowania czytelnikowi jeszcze rozdziały poświęcone bardziej zaawansowanym zagadnieniom — listom wyświetlania, powierzchniom trójwymiarowym i efektom specjalnym. Zwłaszcza przedstawienie problematyki powierzchni trójwymiarowych jest szczególnie warte uwagi.

Po omówieniu zagadnień grafiki trójwymiarowej następuje powrót do problematyki programowania gier i przejście do omówienia zastosowań specyfikacji DirectX, którego nie można znaleźć w żadnej książce poświęconej OpenGL! Przedstawione zostają komponenty DirectX i DirectSound — niezbędne elementy każdej gry na platformie Windows.

Dostarczając w ten sposób wiedzy na temat wszystkich technologii potrzebnych do stworzenia gry, „OpenGL. Programowanie gier” poświęca końcowe rozdziały przedstawieniu modelowania praw fizyki, projektowaniu kompletnego szkieletu gry i stworzeniu przykładowej gry ilustrującej problematykę omówioną w książce.

Książka ta znajdowała się w przygotowaniu od dłuższego już czasu, jednak jestem przekonany, że czytelnicy docenią dodatkowy włożony w nią wysiłek, który pozwolił uczynić ją kwintesencją wiedzy dotyczącej programowania gier i OpenGL.

Z poważaniem

*André LaMothe*  
Maj 2001

# O Autorach

**Kevin Hawkins** jest absolwentem informatyki Uniwersytetu Embry-Riddle w Daytona Beach na Florydzie. Tam też zdobywa obecnie tytuł magisterski w dziedzinie inżynierii oprogramowania. Jego doświadczenie programisty obejmuje blisko 7 lat programowania w C i C++, OpenGL i DirectX oraz w wielu innych językach i interfejsach programowych. Kevin jest także współzałożycielem i prezesem GamDev.net ([www.gamedev.net](http://www.gamedev.net)), najbardziej wszechstronnej witryny w Internecie, która poświęcona jest programowaniu gier. Jest też zawodnikiem uniwersyteckiej drużyny bejsbolowej. Wolny czas poświęca także lekturze, grom komputerowym, grze na gitarze i swojej dziewczynie, Karze.

**Dave Astle** ukończył informatykę na Uniwersytecie Utah, gdzie specjalizował się w grafice komputerowej, w badaniach nad sztuczną inteligencją, programowaniu sieci komputerowych oraz opracowywaniu teorii i projektowania kompilatorów. Programowaniem gier zajął się prawie 20 lat temu, a obecnie pracuje jako programista w firmie WaveLink. Jest też współzałożycielem i dyrektorem GameDev.net, wiodącej witryny społeczności programistów gier. Pracuje też jako główny programista we własnej firmie Myopic Rhino Games ([www.myopicrhino.com](http://www.myopicrhino.com)), która jest niezależnym producentem gier przeznaczonych na rynek masowy. Gdy nie pochłania promieniowania emitowanego przez monitor, słucha muzyki, czyta książki, jeździ na rolkach, zbiera figurki nosorożców, ćwiczy próbując osiągnąć proporcje Jasona Halla i poświęca czas dzieciom.



# Przedmowa

Gry komputerowe stanowią doskonałą rozrywkę. Tworzą niezwykłe światy będące wyzwaniem dla zręczności i inteligencji. Takim samym wyzwaniem jest także tworzenie gier. Muszę przyznać, że nie istniał chyba nigdy dotąd bardziej odpowiedni moment do rozpoczęcia nauki programowania gier. Jeśli ktoś planuje zająć się tą niełatwą sztuką, to otrzymuje właśnie odpowiedni do tego podręcznik.

Moim zdaniem najbardziej niezwykłym aspektem tworzenia gier jest dzisiaj możliwość łatwego wykorzystania ogromnego potencjału tkwiącego w najnowszych kartach graficznych dostępnych dla popularnego sprzętu klasy PC. W niedalekiej przeszłości sytuacja wyglądała zupełnie inaczej. Odpowiedni sprzęt graficzny dostępny był jedynie na dwóch odległych biegunach systemów komputerowych. Z jednej strony odpowiednią wydajność posiadały stacje graficzne pracujące pod kontrolą systemu Unix wytwarzane przez mojego poprzedniego pracodawcę — firmę Silicon Graphics. Ze względu na koszt tego sprzętu był on dostępny jedynie dla naukowców, inżynierów i artystów pracujących w dużych firmach. Chociaż dla stacji tych napisano kilka ciekawych gier, nikt nie kupował ich z tego właśnie powodu. Natomiast na przeciwnym biegunie istniały konsole gier dostępne dla szerokiego ogółu konsumentów. Jednak ich programowanie było zarezerwowane tylko dla wąskiej grupy profesjonalistów, którzy posiadali dostęp do wyspecjalizowanych środowisk tworzenia gier.

Wysoki koszt zaawansowanych stacji graficznych wymusił powstanie efektywnego i łatwego interfejsu programowego, który pozwoliłby w pełni wykorzystywać tkwiący w nich potencjał możliwości tworzenia grafiki trójwymiarowej. Firma Silicon Graphics stworzyła taki interfejs nazwany IRIS GL, gdzie GL oznaczało bibliotekę graficzną (*graphics library*). Interfejs ten stał się podstawą do opracowania przez wszystkich producentów stacji roboczych wspólnego interfejsu OpenGL. Obecnie jest on wykorzystywany przez tysiące aplikacji w nauce, przemyśle, medycynie i animacji komputerowej. Także firma Microsoft zaimplementowała bibliotekę OpenGL — najpierw dla systemu operacyjnego Windows NT, a następnie systemu Windows 95 i ich kolejnych wersji.

OpenGL zaprojektowano przede wszystkim z myślą o tworzeniu trójwymiarowej grafiki wspomaganej sprzętowo. Każde polecenie OpenGL wykonywane przez stację graficzną firmy Silicon Graphics obsługiwane jest sprzętowo. Pierwsze implementacje OpenGL dla komputerów klasy PC wykonywały polecenia OpenGL programowo i dlatego nie były demonem prędkości. Na szczęście opracowanie odpowiedniego sprzętu graficznego dla tej klasy systemów komputerowych okazało się tylko kwestią czasu. Obecnie na przykład rodzina procesorów graficznych GeForce opracowana przez firmę NVIDIA wykonuje

sprzętowo polecenia OpenGL. Coraz doskonalsze układy półprzewodnikowe i innowacyjne architektury procesorów graficznych doprowadziły do sytuacji, w której układy graficzne współczesnych komputerów PC są kilkanaście razy szybsze niż odpowiadające im układy stacji graficznych Silicon Graphics sprzed zaledwie kilku lat (a także wielokrotnie tańsze!). Obecna ewolucja specyfikacji OpenGL podyktowana jest w dużej mierze rozwojem sprzętu graficznego klasy PC i jego zastosowaniami w grach.

Powstała w ten sposób dość niezwykła sytuacja, w której najdoskonalszy interfejs do tworzenia profesjonalnej grafiki trójwymiarowej jest równocześnie najlepszym interfejsem do tworzenia grafiki gier. OpenGL jest obecnie standardem tworzenia trójwymiarowej grafiki nie tylko na platformie Windows PC, ale także dla komputerów firmy Apple pracujących pod kontrolą najnowszego systemu OS X. Oczywiście OpenGL jest także dostępny dla stacji roboczych firm Sun, HP, IBM i Silicon Graphics. OpenGL stał się także istotnym elementem oprogramowania o ogólnie dostępnym kodzie źródłowym. Takie implementacje OpenGL dostępne są na przykład dla systemu operacyjnego Linux. Nie zdziwiłbym się także, gdyby wkrótce standard OpenGL został przyjęty także przez producentów konsoli gier.

Jeśli ktoś korzysta z najnowszych gier dostępnych dla sprzętu PC, to może słyszał już o grach wykorzystujących specyfikację OpenGL i na pewno grał w niektóre z nich. Do najbardziej znanych tytułów należą: seria kolejnych wersji gry *Quake* oraz oczekiwana gra *Doom 3* — obydwie firmy id Software, *Half Life* firmy Valve; opracowane przez Raven Software *Soldier of Fortune* i *Star Trek: Voyager — Elite Force*; *Tribes 2* firmy Dynamic i *Serious Sam* wyprodukowany przez Croteam. Twórcy tych uznanych tytułów byli kiedyś nowicjuszami, którymi powodowała pasja tworzenia trójwymiarowych światów gier. Każdy ma szansę na to, by dołączyć do nich!

Książka ta omawia proces tworzenia kompletnej gry. Oczywiście zasadniczym jej elementem jest trójwymiarowa grafika, dlatego też na pierwszy plan wysuwa się przedstawienie specyfikacji OpenGL. Książka omawia także inne kluczowe aspekty programowania gry: obsługę wejścia za pomocą DirectX Input, dźwięku i muzyki za pomocą DirectX Audio; ładowanie plików graficznych zawierających obrazy tekstur oraz ładowanie modeli obiektów; wykorzystanie praw fizyki do obsługi zderzeń obiektów; projektowanie architektury gry. Jednak książka ta stanowi dopiero początek przygody z programowaniem gier. Kontynuację stanowić mogą przykłady umieszczone na dysku CD oraz inne źródła wymienione w dodatkach.

Pojawienie się tej książki przyprawia mnie o dreszcz emociji — po raz pierwszy programiści otrzymują książkę poświęconą programowaniu gier za pomocą OpenGL!

Jeśli nawet czytelnik tej książki nie wiąże swojej kariery zawodowej z tworzeniem gier komputerowych, to jej lektura pomoże lepiej zrozumieć technologię, która wykorzystywana jest do tworzenia współczesnych gier, a także pozwoli zdobyć umiejętności, które mogą być przydatne także w innych dziedzinach programowania. Zastosowania OpenGL nie ograniczają się bowiem tylko do gier. Ale w tą grę zagrajmy!

Mark J. Kilgard

Inżynier oprogramowania graficznego, NVIDIA Corporation

Autor *OpenGL Programming for the X Window System*

oraz pakietu OpenGL Utility Toolkit (znanego także pod nazwą GLUT)

# Wprowadzenie

Zapraszamy do lektury „OpenGL. Programowanie gier”! Książka ta pokazuje, w jaki sposób można korzystać z zaawansowanych bibliotek graficznych i innych podczas programowania gier. Stanowi efekt pracy autorów trwającej ponad rok. Mamy nadzieję, że stanie się doskonałym podręcznikiem tworzenia gier.

Jak łatwo jest wywnioskować na podstawie tytułu, grafikę gier będziemy tworzyć korzystając z biblioteki OpenGL. Ponieważ OpenGL jest wyłącznie biblioteką graficzną, a gry składają się także z wielu innych elementów, takich jak efekty dźwiękowe, muzyka, obsługa wejścia, to uzupełnimy ją o bibliotekę DirectX.

Zadaniem tej książki jest wypełnienie luki w literaturze przeznaczonej dla programistów gier. Dostępnych jest wiele książek poświęconych grafice OpenGL i równie wiele dotyczących programowania DirectX. Jak dotąd jednak żadna z nich nie próbowała połączyć zastosowania obu specyfikacji.

Mimo że przedstawiamy w naszej książce specyfikacje OpenGL i DirectX, to nie było naszym zamysłem stworzenie kompletnego ich omówienia. Omawiamy jedynie te komponenty, które przydatne są podczas programowania gier. W końcowej części książki podajemy informacje o innych źródłach zawierających więcej informacji na temat OpenGL i DirectX.

Książka nasza podzielona jest na trzy części.

Zadaniem części I jest dostarczenie wiadomości, które stworzą podstawy do omówienia bardziej zaawansowanych zagadnień pojawiających się w następnych częściach. Przedstawiamy więc historię specyfikacji OpenGL i DirectX omawiając jednocześnie ich ogólny sposób działania. Ponieważ docelową platformą naszych gier będzie system Windows, omawiamy także specyfikę tworzenia aplikacji dla tego systemu. Pierwszą część książki kończy omówienie teoretycznych podstawa grafiki trójwymiarowej.

Część II koncentruje się na omówieniu możliwości biblioteki OpenGL, które są przydatne przy projektowaniu gier. Wszystkie rozdziały tej części książki zawierają przykłady programów ilustrujących przedstawiane możliwości OpenGL. Kompletnie, działające programy stanowią także punkt wyjściowy do eksperymentów z OpenGL przeprowadzanych na własną rękę.

Zadaniem części III jest doprowadzenie do stworzenia kompletnego szkieletu programowego gier wykorzystującego OpenGL do tworzenia grafiki oraz DirectSound i DirectInput do obsługi dźwięku i wejścia gry. Kulminacyjnym punktem tej części książki jest stworzenie kompletnej gry wykorzystującej opracowany szkielet i wiele technik omówionych we wcześniejszych rozdziałach.

Do książki dołączyliśmy dysk CD zawierający wiele programów demonstracyjnych, gier i bibliotek stanowiących doskonałe uzupełnienie książki.

Proponujemy zatem przejść do lektury zasadniczej części książki.

# **Część I**

# **Wprowadzenie**

# **do OpenGL i DirectX**



## Rozdział 1.

# OpenGL i DirectX

Przed przystąpieniem do programowaniem gier warto zapoznać się z narzędziami, które będą wykorzystywane. Będą nimi dwa interfejsy programowe: OpenGL i DirectX. Pierwszy z nich wykorzystywany będzie do tworzenia grafiki trójwymiarowej i innych elementów wizualnych, natomiast drugi do interakcji z graczem, tworzenia dźwięku oraz gier, w których może uczestniczyć wielu graczy. W tym rozdziale omówiona zostanie historia powstania obu interfejsów, ich podstawowe możliwości, a ponadto czytelnik zostanie wprowadzony w tematykę gier.

W bieżącym rozdziale przedstawione zostaną takie zagadnienia jak:

- ◆ specyfika gier;
- ◆ podstawy OpenGL;
- ◆ składniki interfejsu DirectX;
- ◆ porównanie interfejsów OpenGL i DirectX.

## Dlaczego tworzymy gry?

Gry komputerowe, które dotychczas zwykło się uważa za rozrywkę dla dzieci, stały się rynkiem wartym kilka miliardów dolarów. Ostatnie lata określiły w tej mierze niezwykle dynamiczny trend, którego granice trudno jest przewidzieć. Eksplozja rynku interaktywnej rozrywki sprawiła, że dzisiaj właśnie ten sektor napędza rozwój technologiczny komputerów PC oraz pomaga prowadzić nowe badania w dziedzinach związanych z tworzeniem grafiki i sztuczną inteligencją. Ten niezwykły rozwój z pewnością przyciąga wielu ludzi do przemysłu gier, ale czy jest on głównym powodem, dla którego tworzymy gry?

Z rozmów z wieloma osobami zajmującymi się tworzeniem gier wyłania się jeden zasadniczy powód, dla którego zajęli się tą dziedziną i odnieśli sukces: *zabawa*. Tworzenie gier słusznie uważa się za jedno z bardziej kreatywnych zajęć związanych z programowaniem — dowodzą tego niezwykłe gry, które powstały w ostatnich latach. Na przykład „Half-Life” firmy Valve Software odmieniła zupełnie dotychczasowe pojęcie gry komputerowej. Programiści i projektanci przyciągani są do przemysłu tworzenia gier dzięki perspektywie kreowania nowych światów, których doświadczać będą wkrótce tysiące, jeśli nie miliony innych ludzi. Tworzenie gier polega na odkrywaniu nowych światów i nowych technologii. Jak ujął to Michael Sikora, niezależny twórca gier, jest ono jak „niekończąca się podróż”. Właśnie dlatego tworzymy gry.

## Świat gier

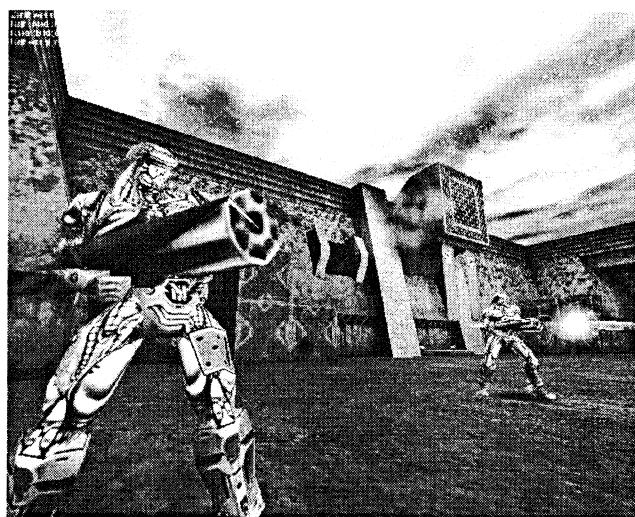
Okolo dziesięciu lat temu firma id Software wypuściła na rynek grę „Wolfenstein 3D”. Rzuciła ona na kolana znawców i miłośników gier dzięki wykorzystaniu trójwymiarowej grafiki i stworzeniu zadziwiających światów, w których gracze spędzali długie godziny. Gra ta zapoczątkowała wyścig twórców gier na niespotykaną dotąd skalę. W 1993 roku pojawiła się kolejna gra firmy id Software — „Doom”, która wykorzystywała udoskonalony moduł tworzenia grafiki trójwymiarowej gry „Wolfenstein”. Po raz kolejny firma id Software wyznaczyła w ten sposób nowy standard gier wykorzystujących grafikę trójwymiarową. Po kilku latach światem gier wstrząsnął „Quake”. Po raz pierwszy gra ta stworzyła w pełni trójwymiarowy świat, w którym postacie przestały tylko udawać trójwymiarowość i zyskały możliwość poruszania się o 6 stopniach swobody. Jedynym ograniczeniem w przypadku tworzenia nowych światów stała się teraz skończona możliwość przetwarzania wielokątów przez procesor maszyny i ich wyświetlania przez kartę graficzną. „Quake” wprowadził także możliwość uczestniczenia w grze dużej liczby graczy obecnych w sieci lokalnej lub sieci Internet.

Od czasu wprowadzenia gry „Quake” nowe osiągnięcia prezentowane są co kilka miesięcy. Wprowadzono sprzętowe przetwarzanie grafiki 3D, które podwaja swoją efektywność średnio co pół roku. Wszystko to sprawia, że obecnie tworzenie gier jest eksplutujące jak nigdy dotąd. Rysunek 1.1 pokazuje ujęcie jednego z ostatnich osiągnięć w tej dziedzinie — gry „Unreal Tournament” firmy Epic.

Rysunek 1.1.

Ujęcie z gry

*Unreal Tournament*



## Elementy gry

Można by zapytać teraz, w jaki sposób powstaje gra. Aby odpowiedzieć na tak postawione pytanie, trzeba uświadomić sobie, że gra na najniższym poziomie abstrakcji jest niczym innym jak programem komputerowym. Współcześnie zdecydowana większość

programów jest efektem pracy zespołów programistów, z których każdy pracuje nad pewnym fragmentem projektu. Fragmenty te łączone są na etapie finalnym w spójną całość. Nie inaczej wygląda tworzenie gier (z tą jednak różnicą, że powstanie gry nie ogranicza się wyłącznie do pracy programistów). Artyści-graficy tworzą obrazy i scenariusz gry. Efekty ich pracy wykorzystywane są przez projektantów, którzy opracowują mapy kolejnych poziomów gry powołując w ten sposób do życia nowy, wirtualny świat. Technicy dźwięku i muzycy przygotowują efekty dźwiękowe, które podnoszą realizm gry, a ponadto muzykę, która tworzy niepowtarzalną atmosferę. Wszystkie te komponenty łączone są potem przez programistów w jedną, działającą całość.

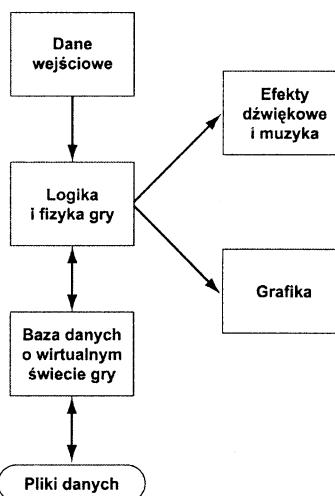
Aby możliwe było właściwe wykorzystanie wiedzy i umiejętności specjalistów z tak różnych dziedzin, projekt gry musi zostać podzielony na szereg elementów, takich jak:

- ◆ grafika;
- ◆ informacje wejściowe;
- ◆ dźwięki i muzyka;
- ◆ logika gry i sztuczna inteligencja;
- ◆ działanie gry w sieci;
- ◆ interfejs użytkownika i system menu.

Każda z tych dziedzin może zostać następnie podzielona na szereg jeszcze bardziej wyspecjalizowanych obszarów. Na przykład logika gry wykorzystywać może mechanikę obiektów oraz system cząstek, a grafika może być tworzona tak w dwu, jak i w trzech wymiarach. Rysunek 1.2 prezentuje uproszczoną architekturę gry.

**Rysunek 1.2.**

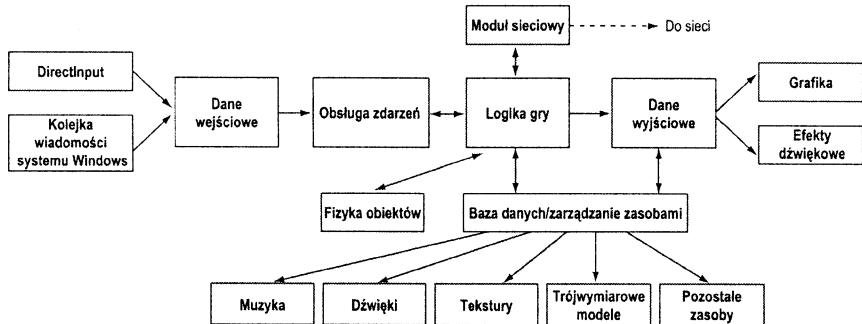
Gra składa się z różnych podsystemów



Każdy z wyróżnionych elementów komunikuje się z innymi elementami. Logika gry stanowi element centralny, w obrębie którego podejmowane są decyzje o przetwarzaniu informacji wejściowej i tworzeniu informacji wyjściowej. Architektura pokazana na rysunku 1.2 jest wyjątkowo uproszczona. Rysunek 1.3 pokazuje, jak mogłaby wyglądać jej bardziej rozbudowana wersja.

**Rysunek 1.3.**

Bardziej zaawansowana architektura gry



Rysunek 1.3 pokazuje, że bardziej zaawansowana gra wymaga bardziej złożonej architektury. Wymaganych jest więcej szczegółowych komponentów implementujących specyficzną funkcjonalność wymaganą do właściwego działania gry. Warto w tym miejscu uświadomić sobie, że gry stanowią szczególnie złożoną kompozycję najróżniejszych technologii i dlatego ich projektowanie powinno odbywać się na jeszcze wyższych poziomach abstrakcji, niż ma to miejsce w przypadku typowego oprogramowania. Projektując grę tworzy się dzieło sztuki i w taki sposób należy traktować cały proces twórczy. Dlatego też nie trzeba obawiać się tworzenia nowatorskich projektów gier i wykorzystania dostępnych technologii w niestandardowy sposób. Nie istnieje jeden ustalony sposób tworzenia gier, podobnie jak nie istnieje na przykład jeden sposób malowania obrazów. Należy starać się o wprowadzenie innowacji i tworzyć nowe standardy!

## Narzędzia

Aby w pełni skorzystać z materiału zawartego w tej książce, należy zaopatrzyć się w dodatkowe oprogramowanie. Oczywiście potrzebny będzie kompilator języka C++. Jako że pisząc tę książkę założyliśmy, że potencjalny czytelnik potrafi programować w języku C++, możemy także przyjąć, że posiada on także odpowiedni kompilator. Wszystkie przykłady programów zamieszczone w tej książce powstały przy użyciu kompilatora Microsoft Visual C++. Nie powinny jednak pojawiać się żadne problemy związane z ich uruchomieniem przy użyciu innego kompilatora języka C++ dla systemu Windows.

Oprócz kompilatora niezbędne będą także biblioteki i pliki nagłówkowe interfejsów OpenGL oraz DirectX. Muszą one zostać zainstalowane tak, by mógł odnaleźć je kompilator języka C++. Jeśli czytelnik jeszcze ich nie zainstalował lub nie jest pewien, czy zrobił to dobrze, to powinien skorzystać z poniższych wskazówek.

Ponieważ specyfikacja interfejsu OpenGL nie jest aktualizowana zbyt często, to prawdopodobnie kompilator używany przez czytelnika zawiera już jej najnowszą wersję. Jeśli kompilatorem tym jest Visual C++, to można mieć pewność, że tak właśnie jest. W każdym innym przypadku trzeba sprawdzić, czy odpowiednie pliki znajdują się w podkatalogach katalogu, w którym zainstalowany został kompilator. W tym celu należy odnaleźć katalog plików nagłówkowych (nazwany najczęściej *include*) oraz katalog bibliotek (najczęściej *lib*). Katalog plików nagłówkowych powinien zawierać podkatalog o nazwie *gl*, w którym powinny znajdować się pliki *gl.h*, *glu.h* i *glaux.h*. Natomiast w katalogu bibliotek powinny być umieszczone pliki *opengl32.lib*, *glu32.lib* i *glaux.lib*. Jeśli, co mało

prawdopodobne, brakować będzie jednego lub więcej z wymienionych plików, to należy uzupełnić go korzystając z plików znajdujących się na dysku CD. Należy odnaleźć i ściągnąć plik o nazwie *opengl95.exe*, rozpakować go i skopiować pliki bibliotek do katalogu bibliotek kompilatora. Później trzeba w katalogu plików nagłówkowych utworzyć podkatalog o nazwie *gl* i umieścić w nim pliki nagłówkowe.



Jako że OpenGL stanowi przykład architektury otwartej, w sieci Internet można napotkać wiele wersji bibliotek i plików nagłówkowych. Dwie najważniejsze implementacje OpenGL pochodzą z firm Silicon Graphics i Microsoft. Ponieważ firma Silicon Graphics nie kontynuuje już prac nad implementacją OpenGL na platformę Windows, można korzystać jedynie z implementacji firmy Microsoft.

Po skompletowaniu plików bibliotek i nagłówków trzeba sprawdzić jeszcze, czy dostępna jest najnowsza wersja sterownika OpenGL dla wykorzystywanej karty graficznej. Można sprawdzić to na stronach internetowych producenta karty lub skorzystać z dostępnego bezpłatnie programu GLSetup (<http://www.glssetup.com>).

Na dysku CD umieszczony także został pakiet DirectX 8.0 SDK. Jeśli czytelnik nie posiada zainstalowanej tej lub nowszej wersji pakietu, to powinien zainstalować go, aby skorzystać z przykładów zamieszczonych w dalszej części książki. Jeśli czytelnik posiada starszą wersję pakietu, to najpierw powinien odinstalować ją korzystając z Panelu sterowania, a dopiero potem zainstalować pakiet DirectX 8.0 SDK.

Po zainstalowaniu obu pakietów SDK należy sprawdzić, czy kompilator został odpowiednio skonfigurowany i potrafi odnaleźć biblioteki i pliki nagłówkowe. Jeśli czytelnik umieścił biblioteki i pliki nagłówkowe OpenGL zgodnie z powyższymi zaleceniami i pozwolił programowi instalacyjnemu pakietu DirectX zaktualizować instalację kompilatora, to krok ten nie jest konieczny. W przeciwnym razie należy skonfigurować kompilator Visual C++ w przedstawiony niżej sposób.

1. Należy rozwinąć menu *Tools* i wybrać polecenie *Options*.
2. Należy wybrać zakładkę *Directories*.
3. Do listy katalogów należy dodać katalog, w którym umieszczone zostały pliki nagłówkowe, a następnie przesunąć go na szczyt listy tak, by nowe wersje plików nagłówkowych były znajdywane przed ich starszymi wersjami dostarczonymi z kompilatorem Visual C++.
4. Należy wybrać zakładkę *Libraries*.
5. Należy dodać do listy katalog, w którym umieszczone zostały pliki bibliotek w identyczny sposób jak w przypadku plików nagłówkowych.

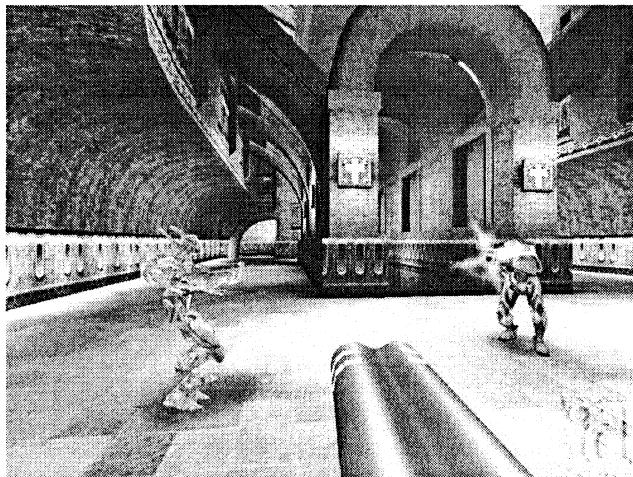
Za każdym razem, gdy będzie tworzony nowy projekt, należy umieścić w nim odpowiednie biblioteki. Istnieje kilka sposobów na określenie wykorzystywanych plików bibliotek. Jednym z lepszych jest wybranie komendy *Settings* z menu *Project*, wybranie zakładki *Link* i dodanie nazwy biblioteki w linii *Object/library modules*. Na przykład dla wszystkich programów korzystających z OpenGL można umieścić w tej linii nazwy plików *opengl32.lib* i *glu32.lib*.

# OpenGL

Specyfikacja OpenGL określa interfejs programowy udostępniający programiście możliwości graficzne platformy, na której działać będzie tworzona aplikacja. OpenGL jest biblioteką niskiego poziomu, która umożliwia generowanie zaawansowanej grafiki, dostępną dla wszystkich najważniejszych platform w różnych konfiguracjach sprzętowych. Zaprojektowano ją z myślą o różnego rodzaju aplikacjach, takich jak na przykład komputerowe wspomaganie projektowania czy gry. Wiele gier dostępnych na rynku wykorzystuje OpenGL. Przykładem może być pokazana na rysunku 1.4 gra *Quake 3* firmy id Software.

Rysunek 1.4.

Gra *Quake 3*



Specyfikacja OpenGL zawiera jedynie operacje graficzne niskiego poziomu, aby umożliwić programiście pełną kontrolę nad tworzoną grafiką i zwiększyć uniwersalność biblioteki. Operacje te mogą zostać wykorzystane do stworzenia własnej biblioteki graficznej wyższego poziomu. Przykładem takiego działania może być biblioteka GLU (*OpenGL Utility Library*) dostarczana razem z większością dystrybucji biblioteki OpenGL. Należy zwrócić przy tym uwagę na to, że OpenGL jest wyłącznie biblioteką operacji graficznych i — w przeciwieństwie do DirectX — nie umożliwia operacji związanych z tworzeniem dźwięku, interakcją z użytkownikiem, tworzeniem aplikacji sieciowych ani żadnych innych operacji niezwiązanych z tworzeniem grafiki.

## Historia OpenGL

Specyfikacja OpenGL została opracowana przez firmę Silicon Graphics, Inc. (SGI) jako interfejs służący do tworzenia grafiki, który jest niezależny od platformy. Od roku 1992 rozwój tej specyfikacji prowadzony jest przez OpenGL Architecture Review Board (ARB), w skład której wchodzą przedstawiciele wiodących producentów sprzętu i oprogramowania — firm ATI, Compaq, Evans & Sutherland, Hewlett-Packard, IBM, Intel, Intergraph, nVidia, Microsoft oraz Silicon Graphics. Zadaniem ARB jest przygotowywanie specyfikacji OpenGL i tym samym dyktowanie funkcjonalności kolejnych dystrybucji interfejsu OpenGL tworzonych przez producentów.

Gdy pisaliśmy tę książkę, najnowszą wersją specyfikacji OpenGL była wersja 1.2. Jako że specyfikacja OpenGL istnieje już dość długo, fakt, że dostępna jest dopiero wersja 1.2, sugeruje, że specyfikacja ta nie jest aktualizowana zbyt często.

Ponieważ specyfikację OpenGL tworzono początkowo z myślą o zastosowaniach związanych jedynie z profesjonalnymi stacjami graficznymi, to tym bardziej była ona wyściągająca dla zwykłych komputerów osobistych. Jednak gwałtowny rozwój możliwości kart graficznych komputerów osobistych w ostatnich latach sprawił, że coraz więcej z nich nie było wykorzystywanych przez specyfikację OpenGL. Dlatego też producenci kart graficznych zaczęli tworzyć własne *rozszerzenia* specyfikacji OpenGL, które z czasem mogły stać się częścią oficjalnego standardu. Specyfikacja OpenGL 1.2 była na przykład pierwszą wersją, która udostępniała wsparcie możliwości wymaganych przez twórców gier (na przykład odwzorowanie wielu tekstur jednocześnie). Jest więc bardzo prawdopodobne, że gry będą miały także znaczący wpływ na postać kolejnych wersji specyfikacji.

## Architektura OpenGL

Specyfikacja OpenGL stanowi zbiór kilkuset funkcji udostępniających możliwości sprzętu graficznego. Wykorzystuje ona maszynę stanów, której stany opisują sposób wykonywania operacji graficznych. Korzystając z interfejsu programowego OpenGL można określić wiele aspektów maszyny stanów, takich jak na przykład bieżący kolor, oświetlenie, sposób łączenia kolorów i tak dalej. Sposób tworzenia grafiki jest więc określony przez bieżącą konfigurację maszyny stanów. Dlatego też ważne jest, by właściwie rozumieć znaczenie poszczególnych stanów maszyny dla sposobu jej działania. Często popełnianym błędem podczas korzystania z biblioteki OpenGL jest niewłaściwy wybór stanu, co skutkuje nieprzewidzianymi efektami operacji graficznych. W książce nie zostaną omówione w całości maszyny stanów OpenGL, ale przedstawione będą te wycinki, które związane będą z wykorzystywanymi operacjami.

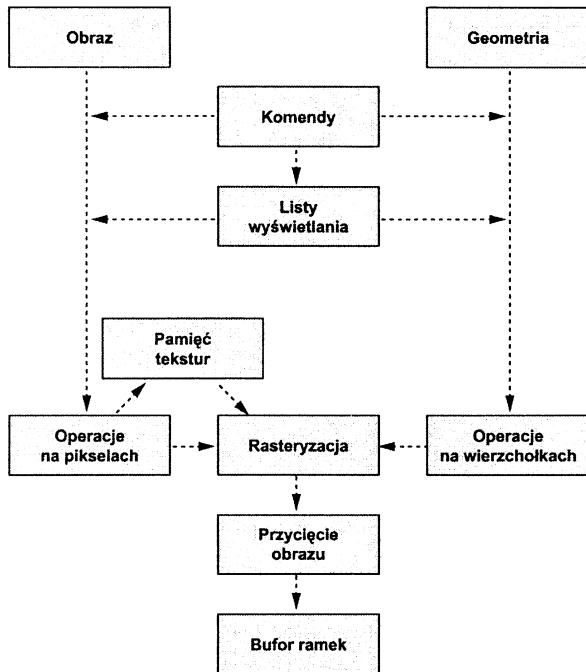
Jądro biblioteki OpenGL stanowi potok tworzenia grafiki przedstawiony na rysunku 1.5. Na tym etapie nie trzeba dokładnie rozumieć znaczenia wszystkich wykonywanych operacji. Najważniejsze jest to, by uszwiadomić sobie, że wszystko, co jest widoczne na ekranie, stanowi rezultat wykonania szeregu operacji w potoku. Większość z tych operacji wykonywana jest na szczęście automatycznie przez bibliotekę OpenGL.

Interfejs programowy OpenGL stanowi w systemie Windows alternatywę dla interfejsu GDI (*Graphics Device Interface*). Zadaniem interfejsu GDI jest ukrycie specyfiki różnych kart graficznych przed programistą tworzącym aplikacje systemu Windows. Jako że interfejs ten zaprojektowano z myślą o typowych aplikacjach, jego zasadniczą wadą z punktu widzenia programisty tworzącego gry jest niska efektywność. Interfejs OpenGL omija interfejs GDI i operuje bezpośrednio na sprzęcie. Rysunek 1.6 prezentuje hierarchię programowych interfejsów graficznych w systemie Windows.

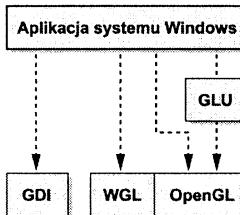
## Biblioteka GLU

Biblioteka GLU (*OpenGL Utility Library*) uzupełnia bibliotekę OpenGL o funkcje graficzne wyższego poziomu. Oferuje ona proste funkcje obudowujące wywołania funkcji OpenGL oraz skomplikowane komponenty umożliwiające wykorzystanie zaawansowanych technik tworzenia grafiki. Należą do nich:

**Rysunek 1.5.**  
Potok tworzenia  
grafiki OpenGL



**Rysunek 1.6.**  
Interfejs programowy  
OpenGL w systemie  
Windows



- ◆ skalowanie obrazów dwuwymiarowych;
- ◆ tworzenie trójwymiarowych brył, takich jak kule czy walce;
- ◆ automatyczne tworzenie mipmap na podstawie pojedynczego obrazu;
- ◆ wsparcie tworzenia powierzchni NURBS;
- ◆ wsparcie operacji kafelkowania wielokątów wklęsłych;
- ◆ specjalizowane przekształcenia i ich macierze.

Jeśli czytelnik nie zna części powyższych terminów, to nie ma powodu do niepokoju. Wyjaśnione zostaną stopniowo w dalszej części książki przy korzystaniu z poszczególnych możliwości biblioteki GLUT.

## Pakiet bibliotek GLUT

Pakiet bibliotek GLUT (*OpenGL Utility Toolkit*) zawiera zestaw pomocniczych bibliotek i jest dostępny dla najważniejszych platform. Biblioteki te uzupełniają OpenGL o możliwość

tworzenia menu, okien czy interakcji z użytkownikiem. Są też niezależne od platformy, co umożliwia łatwe przenoszenie wykorzystujących je aplikacji (na przykład z systemu Unix do Windows).

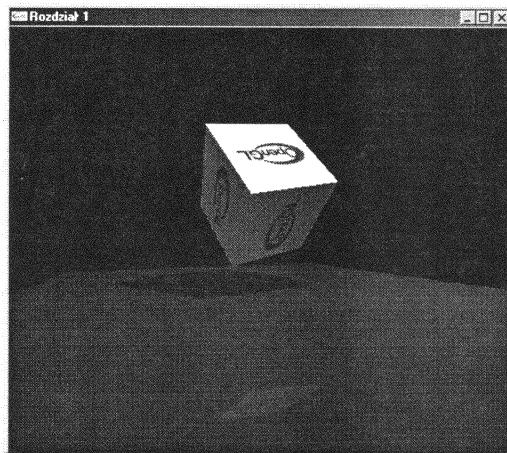
Biblioteki GLUT są łatwe w użyciu i choć nie dostarczają możliwości porównywalnych z oferowanymi przez system operacyjny, są zupełnie wystarczające w przypadku tworzenia prostych aplikacji i programów demonstracyjnych.

W przypadku dość złożonych gier możliwości bibliotek GLUT są zbyt ograniczone i dla tego nie będą wykorzystywane w przykładach zawartych w tej książce. Więcej informacji na temat bibliotek GLUT czytelnik znajdzie na stronie <http://reality.sgi.com/mjk/glut3/>.

## Programy

Należy teraz wyprzedzić nieco bieg książki i przyjrzeć się programom, które będą tworzone. Rysunek 1.7 prezentuje animację oświetlonego i pokrytego teksturą sześcianu. Kod źródłowy tego programu znajduje się na dysku CD. Chociaż znaczna jego część może jeszcze nie być zrozumiałą dla czytelnika, to jednak przykład ten daje ogólny pogląd na to, co będzie tematem kolejnych rozdziałów. Przykład ten — jako jedyny w tej książce — wykorzystuje też biblioteki GLUT. To wyjątkowe rozwiązanie, które pozwala jednak uproszczyć kod źródłowy programu przed omówieniem tworzenia aplikacji systemu Windows. Jeśli czytelnik zdecyduje się na skompilowanie tego programu, to powinien najpierw skopiować z dysku CD pliki bibliotek i nagłówków GLUT w podobny sposób jak w przypadku plików biblioteki OpenGL.

**Rysunek 1.7.**  
*Prosty program  
korzystający  
z biblioteki OpenGL*



## DirectX

DirectX jest zbiorem interfejsów programowych opracowanych przez firmę Microsoft w celu zapewnienia bezpośredniego dostępu do sprzętu w systemie Windows. Każdy z tych interfejsów udostępnia funkcje dostępu do urządzeń lub emuluje urządzenie, jeśli nie występuje ono w danej konfiguracji. Funkcje te umożliwiają tworzenie grafiki dwu-

i trójwymiarowej, wykorzystanie ogromnej liczby typów urządzeń wejściowych, mikrowanie dźwięku, tworzenie gier działających w sieci oraz wykorzystanie wielu formata plików multimedialnych. Do interfejsów programowych DirectX należą:

- ◆ DirectDraw;
- ◆ Direct3D;
- ◆ DirectInput;
- ◆ DirectSound;
- ◆ DirectMusic;
- ◆ DirectPlay;
- ◆ DirectShow.

W zamierzeniu firmy Microsoft DirectX miał umożliwić tworzenie na platformie Windows aplikacji multimedialnych działających efektywnie, a ponadto niezależnych od konkretnej konfiguracji sprzętowej. Dopiero jednak w wersji 7 poradzono sobie z istotnymi ograniczeniami poprzednich wersji, a producenci sprzętu zaczęli rozwiązywać problemy konfliktów sprzętowych, które były zmorą nękającą użytkowników interfejsów programowych DirectX. Dodatkowo wersja ta wprowadziła także kilka bibliotek pomocniczych, takich jak na przykład D3DX. Wprowadzenie tej biblioteki pozwoliło uniknąć wielokrotnego wywoływania funkcji inicjalizacji, które utrudniało wcześniej korzystanie z DirectX. Wersja 8 obok wprowadzenia wielu nowych możliwości przyniosła także zupełnie nową, przeprojektowaną architekturę DirectX. Trzeba zatem przywrócić się bliżej historii DirectX, aby zobaczyć, w jakim kierunku zmierza rozwój tego rozwiązania.

## **Historia DirectX**

W czasach panowania systemu operacyjnego DOS twórcy gier korzystali z bezpośredniego dostępu do sprzętu. Dostęp do przerwań, kart dźwiękowych, portów urządzeń wejściowych, karty grafiki VGA umożliwiał im wykorzystanie możliwości sprzętowych dokładnie w taki sposób, jak to zaprojektowali. Gdy na rynku pojawił się system Windows 3.1, nie wzbudził on większego zainteresowania twórców gier, głównie ze względu na ograniczenia efektywnościowe, jakie na nich nakładał.

Jednak tworzenie gier dla systemu DOS także nie było wolne od poważnych problemów. Na rynku pojawiało się coraz więcej różnych urządzeń i tym samym jeszcze szybciej wzrosła liczba możliwych konfiguracji sprzętu klasy PC. Doszło nawet do tego, że tworzenie sterowników różnych urządzeń pochłaniało więcej nakładów niż programowanie samej gry!

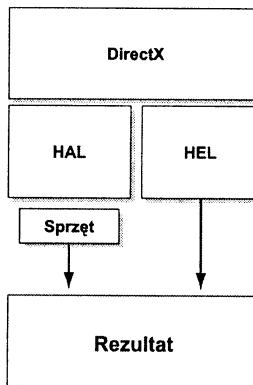
Wprowadzenie przez firmę Microsoft systemu Windows 95 stworzyło nadzieję na zmianę tej sytuacji. Technologia Plug and Play uprościła instalację nowych urządzeń w systemie — w tym kart graficznych, dźwiękowych i urządzeń wejścia. System Windows 95 wprowadził nowy sposób zarządzania zasobami, który ułatwiał zarządzanie urządzeniami i czynił niezależność aplikacji od urządzeń bardziej realną. System nadal jednak nie zawierał żadnych rozszerzeń, które umożliwiłyby efektywne tworzenie gier.

DirectX stworzono więc, aby uczynić z systemu Windows platformę interesującą także dla twórców gier. Projektanci DirectX zrozumieli, że w tym celu należy udostępnić szereg efektywnych bibliotek niskiego poziomu, które umożliwiają programistom gier pełną kontrolę nad tworzonym dziełem. Kolejnym celem DirectX było także uwołnienie twórców gier od tworzenia sterowników urządzeń, za które odtąd odpowiedzialni stali się ich producenci. Inną istotną możliwością DirectX było umożliwienie równoczesnego wykonywania zwykłych aplikacji i gier. Technologia DirectX — dysponując takimi możliwościami — miała też zapewniać efektywność działania gier zbliżoną do osiąganej dotychczas w systemie DOS. Rozwój technologii DirectX na przestrzeni kilku ostatnich lat doprowadził do stworzenia jednego z lepszych interfejsów programowych dostępnych dla twórców gier i aplikacji multimedialnych. Przyjrzyjmy się więc bliżej architekturze DirectX.

## Architektura DirectX

W przekazywaniu żądań DirectX do urządzeń pośredniczą dwie warstwy: warstwa abstrakcji sprzętu HAL (*Hardware Abstraction Layer*) oraz warstwa emulacji sprzętu HEL (*Hardware Emulation Layer*). Na etapie inicjalizacji DirectX sprawdza, czy dana konfiguracja umożliwia sprzętową realizację różnych operacji. Jeśli dana operacja posiada możliwość sprzętowej realizacji, to do jej wykonania wykorzystywana jest warstwa HAL. W przeciwnym razie wykonanie operacji jest emulowane programowo przez warstwę HEL. Ilustruje to rysunek 1.8.

Rysunek 1.8.  
Architektura DirectX



Jak łatwo zauważyć, warstwa HAL korzysta z usług sprzętu, a warstwa HEL sama implementuje pożądaną funkcjonalność. Taka architektura umożliwia stopniowe wykorzystywanie pojawiających się możliwości sprzętowych. Można na przykład założyć, że odpowiednia konfiguracja zawiera kartę grafiki, która umożliwia sprzętową realizację odwzorowań zakłócających dla grafiki trójwymiarowej, ale nie wspiera sprzętowo odwzorowań otoczenia. Jeśli ktoś kupi najnowszą grę, która wykorzystuje oba rodzaje odwzorowań, to warstwa HAL będzie realizować odwzorowania zakłócające przy użyciu możliwości sprzętowych, a warstwa HEL dostarczy programowej implementacji odwzorowań otoczenia. Po pewnym czasie można będzie wymienić kartę grafiki na nową, która będzie realizować sprzętowo oba rodzaje odwzorowań. Odwzorowania otoczenia będą w takim wypadku realizowane z lepszą efektywnością, ponieważ ich realizację przejmie warstwa HAL i będą one wykonywane sprzętowo.

Innym aspektem architektury DirectX jest zestaw tworzących ją interfejsów programowych. Wymienione zostały one już wcześniej, więc teraz zostaną przedstawione bliżej ze wskazaniem na te, które będą wykorzystywane w przykładach zawartych w książce.

## **DirectX Graphics**

Interfejs DirectX Graphics stanowi połączenie interfejsów DirectDraw i Direct3D dostępnych w poprzednich wersjach DirectX. Jako że do tworzenia grafiki wykorzystywane będą możliwości oferowane przez interfejs programowy OpenGL, możliwości DirectX Graphics nie będą dla odbiorcy interesujące.

## **DirectX Audio**

DirectX Audio stanowi połączenie interfejsów DirectSound dostępnych w poprzednich wersjach DirectX. Direct Audio umożliwia realizację kompletnej ścieżki dźwiękowej z wykorzystaniem możliwości sprzętowych kart dźwiękowych, ładowania dźwięków DLS, obiektów DMO (*DirectX Media Objects*) oraz efektów pozycjonowania źródeł dźwięku w przestrzeni. Komponent DirectX Audio stanowi mikser dźwięków o ogromnych możliwościach tworzenia różnego rodzaju efektów wykorzystywanych w grach i aplikacjach multimedialnych. W książce tej przedstawione zostanie wykorzystanie DirectX Audio do implementacji efektów dźwiękowych w programach demonstracyjnych i grach.

## **DirectInput**

Interfejs DirectInput umożliwia programiście dostęp do ogromnej liczby najróżniejszych urządzeń wejściowych oraz zawiera wsparcie siłowego sprzężenia zwrotnego dla specjalizowanych manipulatorów. Komponent DirectInput omija kolejkę komunikatów wykorzystywaną przez system Windows i działa bezpośrednio ze sterownikami urządzeń w celu zapewnienia najkrótszego czasu reakcji tak pożądanej w grach. Komponent DirectInput będzie wykorzystywany we wszystkich przykładach, które wymagać będą interakcji z użytkownikiem.

## **DirectPlay**

DirectPlay stanowi zestaw narzędzi upraszczających komunikację w lokalnych sieciach komputerowych, sieci Internet i za pomocą modemów. Narzędzia te pozwalają łatwo odnaleźć sesje gry działające na innych maszynach i zapewniają komunikację pomiędzy nimi. Przykłady przedstawione w książce nie będą wymagały korzystania z komponentu DirectPlay.

## **DirectShow**

DirectShow umożliwia odtwarzanie plików multimedialnych w formatach *AVI* i *MPG*. Komponent został dołączony do DirectX dopiero w wersji 8, a wcześniej dostępny był oddzielnie. Nie będzie omawiany w książce.

## DirectSetup

DirectSetup umożliwia instalację DirectX przez aplikację. Jako że DirectX jest dość skomplikowanym produktem, interfejs DirectSetup znaczco upraszcza proces jego instalacji. W tej książce nie zostanie jednak omówione jego wykorzystanie.

# Porównanie OpenGL i DirectX

Temat ten jest przedmiotem wielu dyskusji i sporów. Pozostanie nim jeszcze przez długi czas. Tutaj zamierzamy jedynie ograniczyć się do zestawienia możliwości oferowanych przez oba interfejsy.

Pierwszą istotną różnicą jest to, że interfejs DirectX nie służy jedynie do tworzenia grafiki, ale posiada także komponenty umożliwiające tworzenie efektów dźwiękowych, interakcji z użytkownikiem, aplikacji sieciowych i multimedialnych. Natomiast OpenGL jest interfejsem przeznaczonym wyłącznie do tworzenia grafiki. Należy porównać zatem możliwości komponentu graficznego interfejsu DirectX z możliwościami interfejsu OpenGL.

Obydwa interfejsy wykorzystują tradycyjne potokowe tworzenie grafiki. Taki sposób tworzenia grafiki stosowany jest od zarania grafiki komputerowej. Pewne modyfikacje wprowadziło w nim wykorzystanie nowych rozwiązań sprzętowych, ale zasadnicza idea pozostała taka sama.

Obydwa interfejsy opisują punkty wierzchołkowe za pomocą zestawu danych zawierających współrzędne określające ich położenie w przestrzeni oraz inne informacje o punktach. Podstawowe elementy grafiki (punkty, linie, trójkąty) definiowane są za pomocą uporządkowanych zbiorów wierzchołków. Jednak sposób, w jaki informacja o wierzchołkach wpływa na tworzenie tych elementów, jest różny dla obu interfejsów.

Istnieje także wiele innych różnic. Ilustruje je tabela 1.1, w której zestawione zostały możliwości oferowane przez oba interfejsy.

## Podsumowanie

W rozdziale tym przedstawione zostały interfejsy OpenGL i DirectX. Interfejs OpenGL będzie wykorzystywany w dalszej części książki do tworzenia grafiki trójwymiarowej, a interfejs DirectX do tworzenia efektów dźwiękowych i interakcji z użytkownikiem.

Interfejs programowy DirectX stworzono, aby umożliwić aplikacjom systemu Windows bardziej bezpośredni i efektywny dostęp do możliwości sprzętu. Interfejs DirectX korzysta z warstw HAL i HEL, które umożliwiają realizację operacji DirectX niezależnie od tego, czy dana konfiguracja dysponuje odpowiednim sprzętem. W książce tej wykorzystywane będą komponenty DirectX Audio oraz DirectInput.

Skoro mamy już ogólny pogląd na możliwości interfejsów OpenGL i DirectX, to pora zabrać się do programowania!

**Tabela 1.1.** Porównanie interfejsów OpenGL i DirectX

Cecha	OpenGL	DirectX 8
Łączenie wierzchołków w jeden	Nie	Tak
Dostępność dla wielu systemów operacyjnych	Tak	Nie
Mechanizm rozszerzeń	Tak	Tak
Prowadzenie prac nad rozwojem specyfikacji	Gremium złożone z przedstawicieli wielu firm	firma Microsoft
Dostępność pełnej specyfikacji	Tak	Nie
Oświetlenia dwustronne	Tak	Nie
Tekstury przestrzenne	Tak	Nie
Bufory Z niezależne od sprzętu	Tak	Nie
Bufory akumulacyjne	Tak	Nie
Pełnoekranowy antialiasing	Tak	Tak
Rozmycie na skutek ruchu	Tak	Tak
Głębokość pola	Tak	Tak
Tworzenie obrazów stereoskopowych	Tak	Nie
Atrybuty określające rozmiary punktów i szerokość linii	Tak	Nie
Picking	Tak	Nie
Krzywe i powierzchnie parametryczne	Tak	Nie
Bufory geometrii	Listy wyświetlania	Bufory wierzchołków
Emulacja programowa	Nie jest realizowany sprzętowo	Pozwala aplikacji określić sposób realizacji
Interfejs	Wywołania procedur	COM
Aktualizacje	Wydawane co roku przez ARB lub za pomocą rozszerzeń	Co roku
Dostępność kodu źródłowego	Przykładowa implementacja	Microsoft DDK

## Rozdział 2.

# Korzystanie z OpenGL w systemie Windows

Programiści, którzy nie mieli jeszcze okazji do tego, by zająć się tworzeniem programów dla systemu Windows, często uważają, że jest to niezwykle zagmatwane. Jeśli jednak właściwie zrozumie się sposób działania tego systemu i rolę poszczególnych funkcji, to okaże się, że tworzenie programów w systemie Windows jest także niezwykle łatwe. W rozdziale tym przedstawiony zostanie krok po kroku sposób tworzenia aplikacji w systemie Windows. Jeśli czytelnik nie przepadał dotąd za systemem Windows, to może polubi go po przeczytaniu tego rozdziału. Przedstawione w nim zostaną następujące zagadnienia:

- ◆ architektura systemu Windows;
- ◆ procedura okienkowa;
- ◆ klasa okien;
- ◆ zasoby;
- ◆ pętla przetwarzania komunikatów systemu Windows;
- ◆ funkcje WGL;
- ◆ formaty pikseli;
- ◆ korzystanie z OpenGL w systemie Windows;
- ◆ pełnoekranowe aplikacje OpenGL.

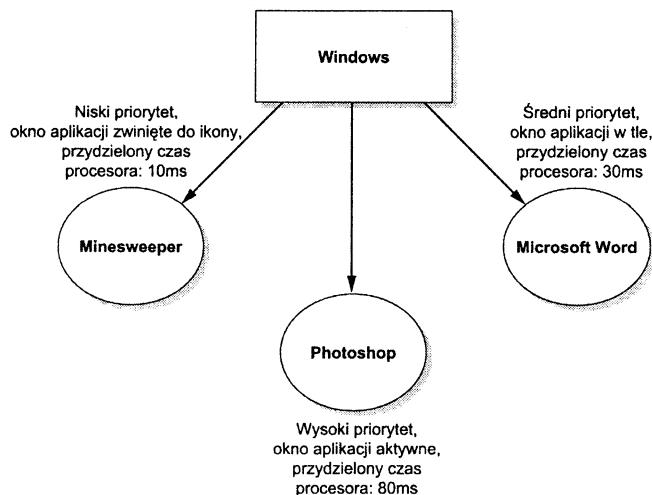
## Wprowadzenie do programowania w systemie Windows

Jako że wszystkie tworzone w tej książce programy będą działać na platformie Windows, trzeba najpierw zapoznać się ze specyfiką tworzenia programów działających w tym systemie. W rozdziale tym przedstawiony zostanie odpowiedni zasób informacji, który wystarczał będzie do tworzenia prostych aplikacji obudowujących OpenGL lub innych prostych aplikacji, takich jak na przykład edytory zasobów gry.

System Windows opracowany przez firmę Microsoft umożliwia jednocześnie działanie wielu aplikacji lub *procesów*. Każdy proces uzyskuje pewien *przedział czasu*, w którym jego wykonanie nie zostanie przerwane przez inny proces. Wielkość tego przedziału czasu ustalana jest przez *zarządcę procesów*, który stanowi część systemu odpowiedzialną za szeregowanie procesów. Zarządcy procesów zapewnia, że każdy proces uzyskuje odpowiednią ilość czasu w zależności od priorytetu procesu oraz bieżącego stanu systemu. Rysunek 2.1 prezentuje sposób, w jaki system Windows zarządza wykonaniem wielu procesów.

**Rysunek 2.1.**

Wykonanie  
wielu procesów  
w systemie Windows

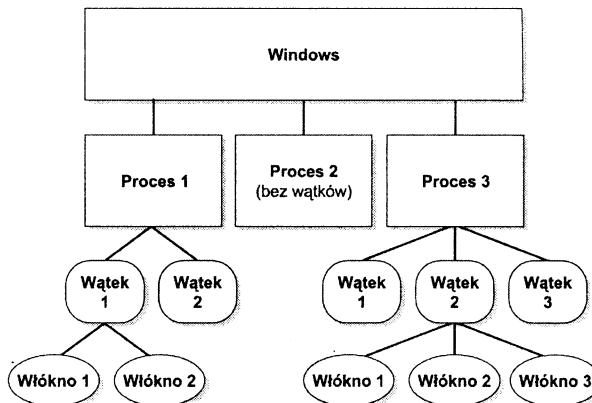


Dla twórców gier interesującą cechą systemu Windows jest jego *wielowątkowość*. Oznacza ona możliwość rozbiecia każdego procesu na wątki, z których każdy wykonuje własny kod. Rozwiązanie takie umożliwia wielozadaniowość w obrębie pojedynczej aplikacji. Podobnie jak procesy wątki są szeregowane na podstawie ich priorytetów i przyznawany jest im odpowiedni czas wykonania. Dzięki wykorzystaniu wątków gra może więc wykonywać równocześnie wiele różnych działań. Okazuje się jednak, że to jeszcze nie koniec możliwości związanych z wielozadaniowością.

Najnowsze wersje systemu Windows (98, ME) wprowadzają kolejną możliwość podziału wykonania procesu w postaci *włókna*. Każdy wątek może składać się z wielu włókien wykonujących niezależne działania. Rysunek 2.2 przedstawia hierarchię procesów, wątków i włókien wykonywanych w wielozadaniowym systemie Windows.

Kolejną istotną cechą sposobu wykonywania aplikacji systemu Windows jest *sterowanie za pomocą zdarzeń*. Oznacza ono, że działanie aplikacji odbywa się na podstawie zdarzeń, które otrzymują one od systemu operacyjnego. Aplikacja może na przykład oczekiwać na naciśnięcie przez użytkownika klawisza. Po naciśnięciu klawisza system operacyjny rejestruje to zdarzenie i przesyła je do aplikacji. Na podstawie otrzymanego zdarzenia aplikacja podejmuje decyzję o sposobie dalszego działania. Obsługa zdarzeń wykonywana jest przez tak zwaną *procedurę okienową*. Aplikacja sprawdza w pętli własną *kolejkę zdarzeń*. Jeśli oczekują w niej zdarzenia, to podejmowana jest ich obsługa.

**Rysunek 2.2.**  
Hierarchia procesów,  
wątków i włókien  
w systemie Windows



Na tym polega w skrócie działanie aplikacji w systemie Windows. Od programów tworzonych dla innych platform odróżnia je przede wszystkim to, że aplikacje Windows muszą obsługiwać zdarzenia generowane przez system. Po przyswojeniu podstawowej wiedzy można przyjrzeć się działaniu programu w systemie Windows.

## Podstawowa aplikacja systemu Windows

Programowanie w systemie Windows nie jest wcale trudne. Pierwszą aplikacją będzie — jak zwykle w takich przypadkach — prosty program, który wyświetli na ekranie komunikat powitania. Przeglądając książki poświęcone podstawom programowania w języku C lub C++ łatwo można natrafić na podobny kod źródłowy:

```

// Program wyświetlający komunikat powitania
#include <stdio.h>

void main()
{
    printf("Witamy!\n");
}
  
```

Program ten będzie działać poprawnie na przykład w systemie DOS, ale dla systemu Windows trzeba stworzyć jego nową wersję.

```

#define WIN32_LEAN_AND_MEAN // "odchudza" aplikację Windows

#include <windows.h>           // główny plik nagłówkowy systemu Windows

// punkt, w którym rozpoczyna się wykonywanie aplikacji
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
    // wyświetla okno dialogowe zawierające tekst powitania
    MessageBox(NULL, "\tWitamy!", "Moja pierwsza aplikacja Windows", 0);
    return 0;
}
  
```

I to wszystko. Jest to prawdopodobnie najprostsza z możliwych aplikacji systemu Windows. Wyświetla ona okno dialogowe zawierające tekst powitania.

Teraz należy przyjrzeć się bliżej sposobowi tworzenia bardziej zaawansowanych aplikacji systemu Windows.

## Funkcja WinMain()

Wykonanie programu w systemie DOS zaczyna się od wywołania funkcji `main()`. Projektanci systemu Windows postanowili być nieco bardziej kreatywni i główną funkcję aplikacji systemu Windows nazwali `WinMain()`. Jej prototyp przedstawia się następująco:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,  
int nShowCmd)
```

Funkcja ta zwraca wartość typu `int`, a `WINAPI` określa konwencję wywołań stosowaną przez wszystkie 32-bitowe wersje systemu Windows (różną od zwykłej konwencji wywołań funkcji języka C).

Funkcji `WinMain()` system przekazuje podczas uruchomienia aplikacji przedstawione niżej parametry.

- ◆ Parametr `hInstance` zawierający uchwyt instancji aplikacji, przez którą należy rozumieć jej pojedynczą działającą kopię. Uchwyt ten wykorzystywany jest przez system do odwoływania się do instancji aplikacji przy obsłudze zdarzeń, przetwarzaniu komunikatów i innych operacjach.
- ◆ Parametr `hPrevInstance` posiadający zawsze wartość `NULL`. Parametr ten wywodzi się z systemu Windows 3.1, w którym posiadał wartość różną od `NULL`, gdy w tle wykonywane były już inne instancje aplikacji.
- ◆ Parametr `lpCmdLine` przekazujący wskaźnik do łańcucha znaków, który zawiera parametry uruchomienia aplikacji podane w linii poleceń. Jeśli na przykład użytkownik uruchomi aplikację korzystając z okna dialogowego *Uruchom*, w którym wpisał `aplikacja.exe parametr 1`, to łańcuch wskazywany przez `lpCmdLine` będzie miał postać parametr `1`.
- ◆ Parametr `nShowCmd` określa sposób prezentacji aplikacji na ekranie po jej uruchomieniu.

## Procedura okienkowa

Aplikacja Windows musi obsłużyć komunikaty wysypane do niej przez system Windows. `WM_CREATE`, `WM_SIZE` czy `WM_MOVE` stanowią jedynie kilka elementów ogromnego zbioru komunikatów, które może otrzymywać aplikacja. Aby system Windows mógł komunikować się z aplikacją, niezbędna jest funkcja zwana *procedurą okienkową* (zwyczajnie o nazwie `WndProc`), która określać będzie sposób obsługi poszczególnych komunikatów o zdarzeniach. Procedura ta nazywana jest często także *procedurą obsługi zdarzeń*. Funkcja `WndProc` otrzymuje komunikat od systemu Windows, przetwarza go i podejmuje odpowiednie działania.

A oto prototyp procedury okienkowej:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
```

Typem zwracanym przez procedurę okienkową jest `LRESULT`, który w systemie Windows zdefiniowany jest jako `long`. `CALLBACK` oznacza konwencję wywołań stosowaną w przypadku

funkcji wywoływanych przez system Windows. Nazwa procedury okienkowej reprezentuje adres funkcji i dzięki temu może być ona wywoływana przez system, ponieważ przekażemy jej adres systemowi tworząc klasę okna.

Pierwszym parametrem procedury okienkowej jest uchwyt okna *hwnd*. Parametr ten ma znaczenie jedynie wtedy, gdy jednocześnie otwartych jest wiele okien tej samej klasy. Umożliwia on wtedy sprawdzenie, którego okna dotyczy komunikat, zanim podjęta zostanie odpowiednia akcja.

Kolejny z parametrów — *message* — reprezentuje identyfikator komunikatu, który należy obsłużyć w procedurze okienkowej. Identyfikatory komunikatów rozpoczynają się od przedrostka *WM\_*.

Trzeci i czwarty parametr — *wParam* i *lParam* — stanowią rozszerzenie parametru *message*. Zawierają one dodatkową informację, która nie mogła być przekazana za pośrednictwem identyfikatora komunikatu.

## Obsługa komunikatów

W celu obsługi komunikatów procedura okienkowa wykorzystuje zwykle warunkową instrukcję wyboru *switch*. Chociaż system Windows może wysyłać do aplikacji kilkaset różnych komunikatów, to jednak procedura okienkowa powinna zapewnić jedynie obsługę tych, które są istotne dla danej aplikacji. W pozostałych przypadkach z reguły system wykonuje domyślną obsługę komunikatów. Poniżej przedstawiamy przykładowy kod procedury okienkowej wykorzystującej instrukcję *switch*.

```
HRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT paintStruct;
    switch(message)
    {
        case WM_CREATE:           // w odpowiedzi na ten komunikat tworzone jest okno
        {
            return 0;
        }

        case WM_CLOSE:           // komunikat o zamknięciu aplikacji
        {
            PostQuitMessage(0);   // wysyła komunikat o zakończeniu aplikacji
            return 0;
        }

        case WM_PAINT:            // zawartość okna powinna zostać odrysowana
        {
            BeginPaint(hwnd, &ps); // odrysowuje okno
            EndPaint(hwnd, &ps);
        }

        default:
            break;
    }
    // przekazuje nieobsłużone komunikaty systemowi
    return (DefWindowProc(hwnd, message, wParam, lParam));
}
```

W obecnej postaci procedura okienkowa prawie nie zawiera kodu obsługi zdarzeń — potrafi jedynie wysłać komunikat o zakończeniu pracy aplikacji. Stanowi jednak dobrą ilustrację ogólnego sposobu obsługi zdarzeń za pomocą instrukcji switch. Pierwszym z obsługiwanych komunikatów jest WM\_CREATE, który wysyłany jest do aplikacji podczas tworzenia jej okna. Obsługując ten komunikat można dokonać inicjalizacji aplikacji bądź, jak w tym przypadku, nie wykonywać żadnych działań i natychmiast zakończyć wykonywanie procedury okienkowej za pomocą polecenia return.

Kolejnym z obsługiwanych komunikatów jest WM\_CLOSE, który jest wysyłany do aplikacji, gdy okno określone przez parametr hwnd jest zamykane. Obsługa tego komunikatu polega typowo na umieszczeniu w kolejce komunikatu WM\_QUIT sygnalizującego zakończenie pracy aplikacji. W tym celu wywołujemy funkcję PostQuitMessage().

Ostatni z zaprezentowanych komunikatów — WM\_PAINT — wysyłany jest, gdy konieczne jest odrysowanie zawartości okna. Obsługując go należy odrysować zawartość okna. W tym przypadku polega ono na wywołaniu funkcji BeginPaint i EndPaint, które powodują walidację obszaru okna i odrysowanie jego tła za pomocą pędzla określonego przez klasę okna.

Jeśli komunikat, który otrzymała aplikacja, nie jest jednym z wymienionych wyżej, to jego obsługę należy przekazać funkcji domyślnej obsługi komunikatów DefWindowProc(). W ten sposób można zyskać pewność, że jeśli komunikat nie zostanie obsłużony przez procedurę okienkową aplikacji, to obsłuży go system.

## Klasy okien

Przez *klasę okna* należy rozumieć zbiór atrybutów, które są wykorzystywane przez system podczas tworzenia okna. Każde okno musi należeć do pewnej klasy. Klasa ta posiada własną procedurę okienkową, która jest wspólna dla wszystkich okien danej klasy. W ten sposób każde okno może przetwarzać komunikaty systemu Windows i obsługiwać je za pomocą procedury okienkowej klasy, do której należy. Klasy okienkowe zdefiniowane są dla wszystkich elementów interfejsu użytkownika systemu Windows: przycisków, okien, list, pól edycji i innych. Wartości atrybutów poszczególnych klas są różne dla różnych elementów interfejsu użytkownika i decydują o ich postaci i zachowaniu.

Aplikacja musi najpierw zdefiniować i zarejestrować klasę okien, zanim utworzy okno danej klasy. Definiując klasę okna posługujemy się strukturą WNDCLASSEX. Stanowi ona rozszerzenie stosowanej dawniej struktury WNDCLASS o dwa pola. Pierwsze z nich określa rozmiar struktury, a drugie uchwyty małej ikony okna. Struktura WNDCLASSEX zdefiniowana jest następująco:

```
typedef struct _WNDCLASSEX {
    UINT        cbSize;           // rozmiar struktury WNDCLASSEX
    UINT        style;            // styl okna
    WNDPROC    lpfnWndProc;       // adres procedury okienkowej
    int         cbClsExtra;        // dodatkowe informacje o klasie
    int         cbWndExtra;        // dodatkowe informacje o oknie
    HINSTANCE   hInstance;        // uchwyty instancji aplikacji
    HICON       hIcon;             // uchwyty dużej ikony okna
    HCURSOR    hCursor;           // uchwyty kurSORA myszy
```

```

HBRUSH    hbrBackground;      // kolor tła okna
LPCWSTR   lpszMenuName;     // nazwa głównego menu
LPCWSTR   lpszClassName;    // nazwa klasy okna
HICON     hIconSm;          // uchwyt małej ikony okna
} WNDCLASSEX;

```

Aby stworzyć klasę okna, trzeba więc zdefiniować poniższą zmienną:

```
WNDCLASSEX windowClass;           // klasa okna aplikacji
```

## Określenie atrybutów klasy okna

Wypełnienie struktury opisującej klasę okna nie przedstawia większej trudności. Pole cbSize określa rozmiar struktury WNDCLASSEX i wobec tego inicjujemy je w poniższy sposób:

```
windowClass.cbSize = sizeof(WNDCLASSEX);
```

Pole hInstance musi zawierać uchwyt bieżącej instancji aplikacji, który otrzymany został jako parametr funkcji WinMain(). Natomiast w polu lpfnWndProc należy umieścić adres procedury okienkowej. Jeśli zdefiniowana zostanie w pokazany wcześniej sposób, to adres ten należy umieścić w polu lpfnWndProc jak poniżej:

```
windowClass.lpfnWndProc = WndProc;
```

Pole stylu okna opisuje ogólne właściwości okna i należy wypełnić je za pomocą predefiniowanych znaczników. Korzystając z operatora sumy bitowej można utworzyć ich dowolną kombinację. Tabela 2.1 prezentuje dostępne znaczniki.

**Tabela 2.1.** Znaczniki stylu okna

Znacznik	Znaczenie
CS_DBLCLKS	System będzie wysyłał do procedury okienkowej komunikaty o dwukrotnym kliknięciu myszą, gdy nastąpi ono w obszarze okna danej klasy
CS_CLASSDC	Tworzy jeden, wspólny kontekst urządzenia dla wszystkich okien klasy
CS_GLOBALCLASS	Pozwala aplikacji utworzyć okno danej klasy niezależnie od wartości parametru hInstance przekazanego funkcji CreateWindowEx
CS_HREDRAW	Powoduje odrysowanie okna jeśli jego szerokość uległa zmianie
CS_NOCLOSE	Blokuje możliwość zamknięcia okna za pomocą systemowego menu okna
CS_OWNDC	Tworzy osobny kontekst urządzenia dla każdego z okien klasy
CS_PARENTDC	Udostępnia region przycięcia okna nadrzędnego oknu podziemnemu, by mogło ono rysować w obszarze okna nadrzędnego
CS_SAVEBITS	Zapisuje w postaci mapy bitowej obszar ekranu przesłonięty przez okno; styl ten przeznaczony jest dla małych okien (na przykład dialogowych lub menu), które pojawiają się na krótko i usuwane są z ekranu, zanim wykonane zostaną inne akcje związane z wyglądem ekranu
CS_VREDRAW	Powoduje odrysowanie okna, jeśli jego wysokość uległa zmianie

Wszystkie one dostępne są dla programisty definiującego klasę okna, ale na razie wystarczy wykorzystać tylko dwa:

```
windowClass.style = CS_HREDRAW | CS_VREDRAW;
```

Pola cbClsExtra i cbWndExtra otrzymają wartość 0. Pole cbClsExtra określa liczbę dodatkowych bajtów, które powinny zostać przydzielone do struktury klasy okna, a pole cbWndExtra liczbę dodatkowych bajtów przydzielanych do każdej instancji okna. W naszym przypadku nie będziemy z nich korzystać.

## Ładowanie ikon i kurSORA myszy

Klasa okien określa także postać kurSORA myszy i ikon okien. Zasoby te można zdefiniować indywidualnie lub skorzystać z gotowych, dostarczanych przez system. Przyjrzymy się zatem temu zagadnieniu bliżej.

Klasa okien określa dwa rodzaje ikon okien: małą i dużą. Mała ikona posiada zazwyczaj wymiary 16×16 pikseli i wyświetlana jest przez systemowe menu okna bądź na pasku zadań, gdy aplikacja jest do niego zwinięta. Duże ikony wykorzystywane są przez system do prezentacji plików, katalogów i pulpitów. Funkcja LoadIcon() umożliwia załadowanie obu rodzajów ikon i posiada następujący prototyp:

```
HICON LoadIcon(HINSTANCE hInst, LPCSTR lpszName);
```

Ładuje ona ikonę określoną za pomocą łańcucha *lpszName* i zwraca uchwyt ikony. Parametr *hInst* określa uchwyt modułu zawierającego ikonę. Aby skorzystać z ikon dostarczanych przez system Windows należy określić jego wartość jako NULL, a jako drugi parametr jedną z makrodefinicji dostępnych dla zasobów systemu Windows. W tym przypadku będzie to makrodefinicja IDI\_APPLICATION, która określa domyślną ikonę aplikacji. Tabela 2.2 prezentuje makrodefinicje dla niektórych ikon oferowanych przez system.

**Tabela 2.2.** Typy ikon

Wartość	Opis
IDI_APPLICATION	Domyślna ikona aplikacji
IDI_ASTERISK	Ikona w kształcie gwiazdki
IDI_ERROR	Ikona w kształcie dłoni
IDI_EXCLAMATION	Wykrzyknik
IDI_HAND	Ikona w kształcie dłoni
IDI_INFORMATION	Ikona w kształcie gwiazdki
IDI_QUESTION	Znak zapytania
IDI_WARNING	Wykrzyknik
IDI_WINLOGO	Logo systemu Windows

Aby związać ikonę z definiowaną klasą okna, trzeba wykonać poniższą operację:

```
windowClass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

W ten sposób określone zostało to, że główną ikoną klasy okien będzie domyślna ikona aplikacji. Małą ikonę okien określa się w ten sam sposób:

```
windowClass.hIconSm = LoadIcon(NULL, IDI_WINLOGO);
```

Tym razem będzie nią logo systemu Windows.

Załadowanie kursora myszy odbywa się w podobny sposób do załadowania ikon. Wykorzystuje się w tym celu funkcję LoadCursor() zdefiniowaną następująco:

```
HCURSOR LoadCursor(HINSTANCE hInst, LPCSTR lpszName);
```

Ładuje ona kursor określony przez łańcuch *lpszName* i zwraca uchwyt kursora. Podobnie jak w przypadku funkcji LoadIcon() pierwszy parametr funkcji LoadCursor() określa uchwyt modułu, który zawiera mapę kursora. Jeśli parametrowi *hInst* nadana zostanie wartość NULL, to można będzie skorzystać z jednego z kursorów zdefiniowanych w systemie Windows. Aby kursor miał standardową postać strzałki, należy skorzystać z makrodefinicji IDC\_ARROW. Inne typy dostępnych kursorów prezentuje tabela 2.3.

**Tabela 2.3.** Typy kursorów

Wartość	Opis
IDC_APPSTARTING	Standardowa strzałka z małą klepsydrą
IDC_ARROW	Standardowa strzałka
IDC_CROSS	Celownik
IDC_HELP	Strzałka ze znakiem zapytania
IDC_IBEAM	Kursor tekstowy
IDC_NO	Przekreślony okrąg
IDC_SIZEALL	Poczwórna strzałka
IDC_SIZENESW	Podwójna strzałka wskazująca umowne kierunki: północno-wschodni i południowo-zachodni
IDC_SIZENS	Podwójna strzałka wskazująca umowne kierunki: północny i południowy
IDC_SIZENWSE	Podwójna strzałka wskazująca umowne kierunki: północno-zachodni i południowo-wschodni
IDC_SIZEWE	Podwójna strzałka wskazująca umowne kierunki: zachodni i wschodni
IDC_UPARROW	Pionowa strzałka
IDC_WAIT	Klepsydra

Aby związać z daną klasą okien określony kursor, należy więc wykonać poniższą operację:

```
windowClass.hCursor = LoadCursor(NULL, IDC_ARROW);
```

W ten sposób okna tej klasy korzystać będą ze standardowego kursora w postaci strzałki. Jako że omówione w ten sposób zostały najważniejsze atrybuty definicji klasy okien, można przyjrzeć się teraz sposobowi, w jaki tworzy się i rejestruje definicję klasy:

```
WNDCLASSEX windowClass; // klasa okna
windowClass.cbSize = sizeof(WNDCLASSEX); // rozmiar opisującej ją struktury
windowClass.style = CS_HREDRAW | CS_VREDRAW; // odrysowanie na skutek
                                                // zmiany rozmiarów
windowClass.lpfnWndProc = WndProc;           // adres procedury okienkowej
windowClass.cbClsExtra = 0;
windowClass.cbWndExtra = 0;
windowClass.hInstance = hInstance;             // instancja aplikacji
windowClass.hIcon = NULL;                     // bez ikony
windowClass.hCursor = LoadCursor(NULL, IDC_ARROW); // standardowa strzałka
windowClass.hbrBackground = NULL;              // bez pędzla tła
windowClass.lpszMenuName = NULL;               // bez menu
```

```
windowClass.lpszClassName = "MojaKlasa";           // nazwa klasy okien
windowClass.hIconSm = NULL;                         // bez małych ikon

RegisterClassEx(&windowClass)                      // rejestruje klasę
```

Całkiem proste, prawda?

## Rejestracja klasy

Po zdefiniowaniu klasy okien należy ją zarejestrować korzystając z funkcji RegisterClassEx() o następującym prototype:

```
ATOM RegisterClassEx(CONST WNDCLASSEX *lpWClass);
```

Wywołać ją można w poniższy sposób:

```
RegisterClassEx(&windowClass)
```

Należy zwrócić uwagę na to, że funkcja ta umożliwia zarejestrowanie klasy okna tylko na podstawie definicji za pomocą struktury WNDCLASSEX, a nie wykorzystywanej wcześniej struktury WNDCLASS. Po zarejestrowaniu klasy okien można stworzyć już pierwsze okno.

## Tworzenia okna

Okna systemu Windows tworzy się za pomocą funkcji CreateWindow() lub CreateWindowEx(). W programach zaprezentowanych w tej książce wykorzystywana będzie funkcja CreateWindowEx(), ponieważ stanowi ona rozszerzoną wersję funkcji tworzenia okien i korzysta w pełni z klas okien. Trzeba więc przyznać się prototypowi funkcji CreateWindowEx():

```
HWND CreateWindowEx(
    DWORD dwExStyle,           // rozszerzony styl okna
    _LPCTSTR lpClassName,      // wskaźnik do nazwy zarejestrowanej klasy
    _LPCTSTR lpWindowName,     // wskaźnik do nazwy okna
    DWORD dwStyle,             // styl okna
    int x,                     // pozycja okna w poziomie
    int y,                     // pozycja okna w pionie
    int nWidth,                // szerokość okna
    int nHeight,               // wysokość okna
    HWND hWndParent,           // uchwyty okna nadzawanego
    HMENU hMenu,                // uchwyty menu lub identyfikator okna podzewanego
    HINSTANCE hInstance,        // uchwyty instancji aplikacji
    LPVOID lpParam);           // wskaźnik do danych związanych z tworzeniem okna
```

Komentarze po prawej stronie parametrów wyjaśniają ich znaczenie. Szerszego omówienia wymagają następujące parametry:

- ◆ *lpClassName* — parametr ten określa nazwę klasy okna, które ma zostać utworzone (na przykład aby utworzyć okno klasy zarejestrowanej w poprzednim przykładzie, będzie musiał mieć wartość "MojaKlasa");
- ◆ *lpWindowName* — określa tekst, który pojawi się na belce okna aplikacji (na przykład "Moja pierwsza aplikacja Windows");
- ◆ *dwStyle* — zawiera znaczniki określające sposób wyglądu i zachowania się okna; tabela 2.4 prezentuje dostępne znaczniki.

**Tabela 2.4.** Znaczniki stylu okna

Styl	Opis
WS_BORDER	Okno będzie posiadać cienką ramkę
WS_CAPTION	Okno będzie posiadać belkę tytułową i cienką linię graniczną (zawiera styl WS_BORDER)
WS_CHILD	Okno podrzędne (styl ten nie może być łączony ze stylem WS_POPUP)
WS_HSCROLL	Okno będzie posiadać pasek przewijania w poziomie
WS_ICONIC	Okno będzie po utworzeniu zwinięte do ikony (równoznaczny ze stylem WS_MINIMIZE)
WS_MAXIMIZE	Okno będzie po utworzeniu zajmować cały ekran
WS_MAXIMIZEBOX	Okno będzie zawierać przycisk umożliwiający powiększenie go na cały ekran (styl ten nie może być łączony ze stylem WS_EX_CONTEXTHELP; dla okna tego musi być także określony styl WS_SYSMENU)
WS_MINIMIZE	Okno będzie po utworzeniu zwinięte do ikony (równoznaczny ze stylem WS_ICONIC)
WS_MINIMIZEBOX	Okno będzie zawierać przycisk umożliwiający zwinięcie go do ikony (styl ten nie może być łączony ze stylem WS_EX_CONTEXTHELP; dla okna tego musi być także określony styl WS_SYSMENU)
WS_OVERLAPPED	Okno nakładające się, posiadające belkę tytułową i ramkę (równoważny stylowi WS_TILED)
WS_OVERLAPPEDWINDOW	Okno nakładające się i łączące style WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX i WS_MAXIMIZEBOX (równoważny stylowi WS_TILEDWINDOW)
WS_POPUP	Okno „wyskakujące” (styl ten nie może być łączony ze stylem WS_CHILD)
WS_POPUPWINDOW	Okno „wyskakujące” łączące style WS_BORDER, WS_POPUP i WS_SYSMENU (aby menu okna było widoczne, styl WS_POPUPWINDOW musi być wykorzystywany w połączeniu ze stylem WS_CAPTION)
WS_SIZEBOX	Okno będzie posiadać ramkę umożliwiającą zmianę jego rozmiarów (równoznaczne ze stylem WS_THICKFRAME)
WS_SYSMENU	Okno będzie posiadać menu okna otwierane na belce tytułowej (dla okna tego musi być także określony styl WS_CAPTION)
WS_VISIBLE	Okno będzie widoczne
WS_VSCROLL	Okno będzie posiadać pasek przewijania w pionie

Funkcja CreateWindowEx() zwraca wartość NULL, jeśli utworzenie okna nie powiodło się. W przeciwnym razie zwraca uchwyt utworzonego okna.

Poniżej zaprezentowany został przykład wywołania funkcji CreateWindowEx():

```
hwnd = CreateWindowEx(NULL,           // nie korzystamy z rozszerzonego stylu
                      "MojaKlasa",        // nazwa klasy okna
                      "Moja pierwsza aplikacja Windows", // tytuł okna
                      WS_OVERLAPPEDWINDOW | // styl okna
                      WS_CLIPSIBLINGS |
                      WS_CLIPCHILDREN,
                      0, 0,                // współrzędne x, y
                      200, 200)             // szerokość i wysokość okna
```

```

NULL,                                // nie istnieje okno nadzędne
NULL,                                // okno nie będzie posiadać menu
hInstance,                            // instancja aplikacji
NULL);                               // brak dodatkowych parametrów okna

```

Wywołanie to utworzy okno o rozmiarach 200 na 200 pikseli, które będzie umieszczone w górnym-lewym narożniku ekranu. Okno to nie będzie jednak widoczne zaraz po utworzeniu. Można temu zaradzić dodając do stylu okna wartość WS\_VISIBLE lub korzystając z funkcji ShowWindow() w poniższy sposób:

```
ShowWindow(hwnd, nCmdShow);
```

Zmienna nCmdShow pochodzi w tym przypadku z wywołania funkcji WinMain(). Należy jeszcze wymusić na systemie Windows odrysowanie zawartości okna poprzez wysłanie komunikatu WM\_PAINT do procedury okienkowej. W tym celu wywołuje się funkcję

```
UpdateWindow();
```

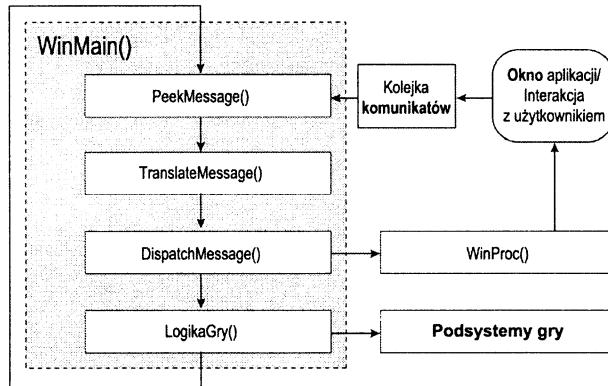
W ten sposób wyświetlane zostaje okno aplikacji. Można więc przejść do omówienia pętli przetwarzania komunikatów, która połączy omówione dotąd elementy aplikacji systemu Windows w jedną całość.

## Pętla przetwarzania komunikatów

Aplikacja komunikuje się z systemem Windows pobierając komunikaty z kolejki, które system umieszcza w niej informując o wystąpieniu zdarzeń. Komunikaty te muszą być pobierane przez aplikację w pętli przetwarzania komunikatów wykonywanej wewnątrz funkcji WinMain(). Pętla ta pobiera komunikat z kolejki, a następnie przekazuje go z powrotem do systemu, który przetwarza komunikat, zanim przekaże go procedurze okienkowej aplikacji. Jak pokazano to na rysunku 2.3, pętla przetwarzania komunikatów wykonuje zawsze dla pobranego komunikatu trzy funkcje: PeekMessage(), TranslateMessage() i DispatchMessage().

**Rysunek 2.3.**

Pętla przetwarzania komunikatów



Funkcja PeekMessage() umożliwia sprawdzenie istnienia komunikatu oczekującego w kolejce. Jej prototyp przedstawia się następująco:

```
BOOL PeekMessage(LPMSG lpMsg, HWND hWnd, UINT wMsgFilterMin, UINT wMsgFilterMax,
                  UINT wRemoveMsg);
```

Jeśli w kolejce istniał komunikat, to został z niej pobrany i przekazany za pośrednictwem parametru *lpMsg*. Przekazanie za pomocą parametru *hWnd* wartości NULL oznacza, że funkcja *PeekMessage()* powinna sprawdzić kolejkę komunikatów bieżącej aplikacji. Również parametry *wMsgFilterMin* i *wMsgFilterMax* będą miały w analizowanych zastosowaniach wartość NULL. Ostatniemu z parametrów — *wRemoveMsg* — należy nadać wartość *PM\_REMOVE*, która informuje funkcję *PeekMessage()*, że należy usunąć komunikat z kolejki po jego przetworzeniu.



Dostępna jest także funkcja *GetMessage()*, która wykonuje identyczne działania co funkcja *PeekMessage()*. Jednak wstrzymuje ona wykonanie aplikacji w przypadku, gdy kolejka komunikatów jest pusta (aż do momentu, w którym w kolejce pojawi się nowy komunikat). W analizowanych w książce przykładach wykorzystana została funkcja *PeekMessage()*, która pozwala kontynuować działanie aplikacji, gdy kolejka komunikatów jest pusta. Jest to szczególnie istotne w przypadku gier.

Funkcja *TranslateMessage()* wykorzystywana jest do tłumaczenia komunikatów o wybraniu wirtualnych klawiszy na komunikaty o znakach. Są one umieszczane z powrotem w kolejce danej aplikacji i pobierane przy następnym wywołaniu funkcji *PeekMessage()*. Prototyp funkcji *TranslateMessage()* wygląda następująco:

```
BOOL TranslateMessage(CONST MSG *lpMsg);
```

Parametr *lpMsg* reprezentuje tłumaczony komunikat.

Ostatnią z funkcji wywoływanych w pętli przetwarzania komunikatów jest *DispatchMessage()*. Powoduje ona przekazanie komunikatu, który jest jej jedynym parametrem, z powrotem do systemu Windows, który przetwarza go i przekazuje do procedury okienkowej aplikacji. Prototyp funkcji *DispatchMessage()* zaprezentowany został poniżej.

```
LONG DispatchMessage(CONST MSG *lpMsg);
```

Pętla przetwarzania komunikatów kończy swoje działanie, gdy zajdzie określony przez programistę warunek. Zwykle jest nim pobranie z kolejki komunikatu *WM\_QUIT*. Funkcja *WinMain()* kończąc swoje działanie przekazuje wtedy systemowi wartość *msg.wParam*.

A oto przykład pętli przetwarzania komunikatów:

```
// pętla przetwarzania komunikatów
// kończy działanie po otrzymaniu komunikatu WM_QUIT
while (!done)
{
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) // komunikat w kolejce?
    {
        if (msg.message == WM_QUIT)           // po otrzymaniu komunikatu WM_QUIT
        {
            done = TRUE;                   // aplikacja kończy działanie
        }
        else
        {
            // przetwarzanie
            TranslateMessage(&msg);          // tłumaczenie komunikatu
            DispatchMessage(&msg);           // przekazanie komunikatu do Windows
        }
    }
}
```

```
    }
}

return msg.wParam;                                // zwraca kod zakończenia
```

Na początku pętli wywoływana jest funkcja PeekMessage() w celu sprawdzenia obecności komunikatu w kolejce i pobrania go. Jeśli w kolejce znajduje się komunikat i nie informuje on o zakończeniu pracy aplikacji, to podejmowane jest jego przetwarzanie w pętli. Pętla przetwarzania komunikatów stanowi prosty i efektywny zarazem sposób obsługi potencjalnie dużej liczby komunikatów, które system może wysłać do aplikacji.

## Kompletna aplikacja systemu Windows

Jako że wszystkie elementy tworzące szkielet aplikacji systemu Windows są znane, można stworzyć już kompletny, działający program. Poniżej zaprezentowany został kod źródłowy aplikacji systemu Windows, która wyświetla tekst powitania niebieską czcionką na białym tle okna.

```
#define WIN32_LEAN_AND_MEAN      // "odchudza" aplikację Windows

#include <windows.h>            // standardowy plik nagłówkowy Windows

// procedura okienkowa
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT paintStruct;
    HDC hDC;                      // kontekst urządzenia
    char string[] = "Witamy!";     // wyświetlany tekst

    switch(message)
    {
        case WM_CREATE:          // okno jest tworzone
            return 0;
            break;

        case WM_CLOSE:           // okno jest zamykane
            PostQuitMessage(0);
            return 0;
            break;

        case WM_PAINT:            // zawartość okna musi być odrysowana
            hDC = BeginPaint(hwnd, &paintStruct);

            // wybiera kolor niebieski
            SetTextColor(hDC, COLORREF(0x00FF0000));

            // wyświetla tekst w środku okna
            TextOut(hDC, 150, 150, string, sizeof(string)-1);

            EndPaint(hwnd, &paintStruct);
            return 0;
            break;

        default:
            break;
    }
}
```

```
        return (DefWindowProc(hwnd, message, wParam, lParam));
    }

    // punkt rozpoczęcia wykonywania aplikacji
    int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
                        int nShowCmd)
    {
        WNDCLASSEX windowClass;           // klasa okna
        HWND hwnd;                      // uchwyty okna
        MSG msg;                        // komunikat
        bool done;                      // znacznik zakończenia aplikacji

        // tworzy strukturę definiującą klasę okna
        windowClass.cbSize = sizeof(WNDCLASSEX);
        windowClass.style = CS_HREDRAW | CS_VREDRAW;
        windowClass.lpfnWndProc = WndProc;
        windowClass.cbClsExtra = 0;
        windowClass.cbWndExtra = 0;
        windowClass.hInstance = hInstance;
        windowClass.hIcon = LoadIcon(NULL, IDI_APPLICATION);          // domyślana ikona
        windowClass.hCursor = LoadCursor(NULL, IDC_ARROW);            // domyślny kurSOR
        windowClass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH); // białe tło
        windowClass.lpszMenuName = NULL;                                // bez menu
        windowClass.lpszClassName = "MojaKlasa";
        windowClass.hIconSm = LoadIcon(NULL, IDI_WINLOGO); // logo Windows jako mała ikona

        // rejestruje klasę okna
        if (!RegisterClassEx(&windowClass))
            return 0;

        // tworzy okno
        hwnd = CreateWindowEx(NULL,
                             "MojaKlasa",                         // nazwa klasy
                             "Prawdziwa aplikacja Windows!",       // nazwa aplikacji
                             WS_OVERLAPPEDWINDOW | WS_VISIBLE | WS_SYSMENU, // styl okna
                             100, 100,                            // współrzędne x,y
                             400, 400,                            // szerokość, wysokość
                             NULL,                               // uchwyty okna nadziednego
                             NULL,                               // uchwyty menu
                             hInstance,                          // instancja aplikacji
                             NULL);                            // bez dodatkowych parametrów

        // sprawdza, czy utworzenie okna nie powiodło się (wtedy hwnd będzie NULL)
        if (!hwnd)
            return 0;

        done = false;                      // inicjuje zmienną warunku pętli

        // pętla przetwarzania komunikatów
        while (!done)
        {
            PeekMessage(&msg, hwnd, NULL, NULL, PM_REMOVE);

            if (msg.message == WM_QUIT) // czy aplikacja otrzymała komunikat WM_QUIT?
            {
                done = true;           // jeśli tak, to musi zakończyć działanie
            }
            else
        }
```

```

    {
        TranslateMessage(&msg); // przetwarza komunikat
        DispatchMessage(&msg);
    }
}

return msg.wParam;
}

```

Teraz należy przyjrzeć się bliżej kodowi aplikacji.

Pierwszy jego wiersz w postaci `#define WIN32_LEAN_AND_MEAN` zapobiega dołączeniu do kodu wynikowego aplikacji zbędnych modułów, które nie zostaną wykorzystane.

Kolejny wiersz `#include <windows.h>` jest dyrektywą kompilatora włączającą pliki nagłówkowe, które są niezbędne do poprawnej kompilacji aplikacji Windows. W innych przypadkach można też dołączyć plik nagłówkowy `<windowsx.h>`, który zawiera makrodefinicje użyteczne podczas tworzenia kodu źródłowego aplikacji systemu Windows.

Pierwszą funkcją w kodzie programu jest procedura okienkowa `WndProc()`. Dla uproszczenia wykorzystana została jej wersja, która była już wcześniej omówiona (patrz: podrozdział „Procedura okienkowa”), a teraz została rozbudowana o kilka elementów. Pierwszy z nich tworzą poniższe wiersze kodu:

```

HDC hDC; // kontekst urządzenia
char string[] = "Witamy!"; // wyświetlany tekst

```

Deklarują one kontekst urządzenia wykorzystywany do wyświetlania w obszarze okna oraz tekst wyświetlany powitanie.

### Kontekst urządzenia

Kontekst urządzenia jest strukturą danych wykorzystywaną podczas tworzenia grafiki w oknie lub na drukarce. Zawartość okna można modyfikować jedynie korzystając z kontekstu urządzenia, który można utworzyć za pomocą funkcji `GetDC()`:

```
hDC = GetDC(hwnd);
```

Wywołanie to zwraca kontekst urządzenia służący do tworzenia grafiki w oknie o uchwycie określonym przez parametr `hwnd`.

Następnie w kodzie funkcji `WndProc` pojawia się instrukcja wyboru switch rozpoznająca typ komunikatu przekazanego do procedury okienkowej. Nowym elementem jest blok obsługi komunikatu `WM_PAINT`:

```

case WM_PAINT: // zawartość okna musi być odrysowana
    hDC = BeginPaint(hwnd, &paintStruct);

    // wybiera kolor niebieski
    SetTextColor(hDC, COLORREF(0x00FF0000));

    // wyświetla tekst w środku okna
    TextOut(hDC, 150, 150, string, sizeof(string)-1);

    EndPaint(hwnd, &paintStruct);
    return 0;
}

```

Blok ten wykonywany jest, gdy zachodzi konieczność odrysowania zawartości okna na skutek jego przesunięcia, zmiany rozmiaru bądź innego zdarzenia, które miało wpływ na zawartość okna. Blok ten wykorzystuje kontekst urządzenia hDC zwrócony przez funkcję BeginPaint() dla okna o uchwycie hwnd. Korzystając z kontekstu urządzenia określa się za pomocą funkcji SetTextColor() kolor tekstu wyświetlanego później za pomocą funkcji TextOut(). Funkcje te nie zostaną teraz bliżej omówione. Jeśli czytelnik pragnie uzyskać więcej informacji na ich temat, to powinien skorzystać z dokumentacji interfejsu programowego systemu Windows, która jest dostępna w MSDN.

#### MSDrN

Microsoft Developer Network (w skrócie MSDN) zawiera dokumentację narzędzi programistycznych firmy Microsoft. Kopia MSDN dołączana jest do pakietu Visual Studio, jak i jego składowych. MSDN dostępny jest także w sieci Internet pod adresem <http://msdn.microsoft.com>.

Kolejną funkcją zdefiniowaną w kodzie źródłowym analizowanej aplikacji jest WinMain(). Stanowi ona punkt, w którym rozpoczyna się wykonywanie aplikacji. Kod jej jest dość oczywisty, ale warto zwrócić uwagę na kilka elementów. Zmienna msg wykorzystywana jest do przechowania komunikatu pobranego za pomocą funkcji PeekMessage(), który następnie przekazywany jest funkcją TranslateMessage() i DispatchMessage(). Zmienna done stanowi warunek wykonania pętli przetwarzania komunikatów i uzyskuje wartość true po otrzymaniu komunikatu WM\_QUIT.

Należy zwrócić uwagę na porządek, w jakim wykonywane są kolejne operacje niezbędne do działania aplikacji systemu Windows. Można wyróżnić wśród nich:

- ◆ definicję klasy okna;
- ◆ rejestrację klasy okna;
- ◆ utworzenie okna;
- ◆ pętlę przetwarzania komunikatów i procedurę okienkową.

W ten sposób uzyskany został szkielet aplikacji systemu Windows, który można dopasowywać do indywidualnych potrzeb. Kolejny krok jego rozwoju będzie polegał na umożliwieniu aplikacji korzystania z biblioteki OpenGL.

## Wprowadzenie do funkcji WGL

Ponieważ OpenGL jest jedynie interfejsem graficznym, to takie zadania jak na przykład interakcja z użytkownikiem czy tworzenie okien aplikacji muszą być obsługiwane przez system operacyjny. Aby ułatwić pracę programisty, dla każdego systemu operacyjnego z reguły dostarczany jest zbiór rozszerzeń umożliwiających powiązanie OpenGL z interakcją i tworzeniem okien na danej platformie. W systemie UNIX rozszerzenie takie nosi nazwę GLX. Natomiast w systemie Windows korzysta się ze zbioru funkcji WGL. Nazwy funkcji należących do tego zbioru rozpoczynają się przedrostkiem *wgl*. Również interfejs programowy systemu Windows posiada funkcje umożliwiające wykorzystanie OpenGL.

## Kontekst tworzenia grafiki

Aby zachować możliwość przenoszenia aplikacji OpenGL pomiędzy różnymi platformami, każdy system operacyjny musi dostarczyć kontekst tworzenia grafiki OpenGL. Jak pokazano wcześniej, aplikacje systemu Windows korzystają z kontekstu urządzenia, który przechowuje informacje o sposobie tworzenia grafiki, natomiast OpenGL korzysta z własnego *kontekstu tworzenia grafiki* spełniającego podobne funkcje. Kontekst urządzenia i kontekst tworzenia grafiki nie mają ze sobą nic wspólnego. Pierwszy z nich podaje się wywołując funkcje graficzne systemu Windows, a drugi jest domyślnie wykorzystywany przez komendy OpenGL. Dlatego też zanim utworzony zostanie kontekst tworzenia grafiki, trzeba najpierw określić format piksela dla kontekstu urządzenia.

Istnieje możliwość korzystania z wielu kontekstów tworzenia grafiki, jeśli tworzy się ją na przykład równocześnie w wielu oknach. Należy jedynie zawsze zapewnić to, że bieżący kontekst tworzenia grafiki będzie właściwym kontekstem dla okna, w którym tworzy się ją. Ważną cechą OpenGL jest możliwość bezpiecznego wykonywania jego poleceń przez różne wątki. Wiele wątków może tworzyć grafikę korzystając z tego samego kontekstu. Na przykład jeden z wątków może tworzyć grafikę wirtualnego świata gry, a inny elementy interfejsu użytkownika. Jeszcze inny wątek może równocześnie tworzyć grafikę pokazującą inne ujęcie świata gry w osobnym oknie. Podobnych możliwości jest bez liku.

## Korzystanie z funkcji WGL

Funkcje WGL umożliwiają działanie OpenGL w systemie Windows. Przedstawione teraz zostaną trzy najczęściej wykorzystywane funkcje WGL:

- ◆ `wglCreateContext();`
- ◆ `wglDeleteContext();`
- ◆ `wglGetCurrent();`

### Funkcja `wglCreateContext()`

Funkcja `wglCreateContext()` tworzy uchwyt kontekstu tworzenia grafiki OpenGL na podstawie przekazanego jej kontekstu urządzenia systemu Windows. Jej prototyp prezentuje się następująco:

```
HGLRC wglCreateContext(HDC hDC)
```

Funkcję tę można wywoływać dopiero po określeniu formatu pikseli dla danego kontekstu urządzenia (format pikseli zostanie omówiony niebawem.)

### Funkcja `wglDeleteContext()`

Podobnie jak w przypadku kontekstu urządzenia także i dla kontekstu tworzenia grafiki powinniśmy pamiętać o jego usunięciu, gdy przestajemy go używać. W tym celu wywołujemy funkcję `wglDeleteContext()` o przedstawionym poniżej prototypie:

```
BOOL wglDeleteContext(HGLRC hRC);
```

## Funkcja wglMakeCurrent()

Funkcja `wglMakeCurrent()` służy do wyboru bieżącego kontekstu tworzenia grafiki OpenGL. Przekazywany jej kontekst urządzenia musi posiadać taki sam format pikseli jak kontekst urządzenia, który wykorzystany został do utworzenia danego kontekstu tworzenia grafiki. W praktyce oznacza to, że kontekst urządzenia przekazywany funkcji `wglMakeCurrent()` nie musi być tym samym kontekstem urządzenia, który został przekazany funkcji `wglCreateContext()`. Prototyp funkcji `wglMakeCurrent()` przedstawia się następująco:

```
BOOL wglMakeCurrent(HDC hDC, HGLRC hRC);
```

Trzeba jednak zapewnić, że oba przekazywane jej konteksty korzystają z tego samego formatu pikseli. Jeśli konieczne okaże się zaprzestanie korzystania z bieżącego kontekstu tworzenia grafiki, należy przekazać wartość `NULL` jako parametr `hRC` funkcji `wglMakeCurrent()` lub inny kontekst tworzenia grafiki.

Funkcje `wglCreateContext()` i `wglMakeCurrent()` powinny zostać wywołane po utworzeniu okna, czyli wtedy, gdy procedura okienkowa otrzyma komunikat `WM_CREATE`. Funkcja `wglDeleteContext()` powinna natomiast zostać wywołana podczas usuwania okna (na przykład na skutek otrzymania komunikatu `WM_DESTROY`), jednak po uprzednim wywołaniu funkcji `wglMakeCurrent()` z wartością `NULL` parametru `hRC`.

Powyższe uwagi ilustruje następujący fragment kodu:

```
HRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HGLRC hRC;           // kontekst tworzenia grafiki
    static HDC hDC;             // kontekst urządzenia

    switch(message)
    {
        case WM_CREATE:         // okno jest tworzone
            hDC = GetDC(hwnd);   // pobiera kontekst urządzenia dla danego okna
            hRC = wglCreateContext(hDC); // tworzy kontekst tworzenia grafiki
            wglMakeCurrent(hDC, hRC); // określa bieżący kontekst tworzenia grafiki
            break;

        case WM_DESTROY:         // okno jest usuwane
            wglMakeCurrent(hDC, NULL); // przestaje korzystać z bieżącego kontekstu
                                         // tworzenia grafiki
            wglDeleteContext(hRC);   // usuwa kontekst tworzenia grafiki
            PostQuitMessage(0);     // wysyła komunikat WM_QUIT
            break;
    } // koniec instrukcji switch
} // koniec WndProc
```

Powyższy fragment kodu tworzy i usuwa okno grafiki OpenGL. Kontekst urządzenia i kontekst tworzenia grafiki przechowuje się za pomocą zmiennych statycznych, aby nie tworzyć ich przy każdym wywołaniu procedury okienkowej. W ten sposób czyni się jej wykonanie efektywniejszym. Pozostała część kodu jest dość oczywista i opatrzona komentarzami. Zanim jednak stworzone zostanie pierwsze okno aplikacji OpenGL, trzeba zapoznać się z formatami pikseli oraz strukturą `PIXELFORMATDESCRIPTOR`.

# Formaty pikseli

*Format pikseli* stanowi kolejne z rozszerzeń interfejsu programowego systemu Windows, które umożliwia korzystanie z OpenGL. Określając format pikseli podaje się na przykład tryb koloru, bufor głębi, liczbę bitów opisujących piksel oraz sposób buforowania zawartości okna. Format pikseli trzeba określić zawsze przed stworzeniem kontekstu tworzenia grafiki.

Aby zdefiniować charakterystykę i zachowania okna graficznego, trzeba posłużyć się strukturą `PIXELFORMATDESCRIPTOR`. Jest ona zdefiniowana następująco:

```
typedef struct tagPIXELFORMATDESCRIPTOR
{
    WORD    nSize;           // rozmiar struktury
    WORD    nVersion;        // zawsze wartość 1
    DWORD   dwFlags;         // znaczniki właściwości bufora pikseli
    BYTE    iPixelType;      // typ danych piksela
    BYTE    cColorBits;       // liczba bitów piksela
    BYTE    cRedBits;         // liczba bitów koloru czerwonego
    BYTE    cRedShift;        // przesunięcie bitów koloru czerwonego
    BYTE    cGreenBits;       // liczba bitów koloru zielonego
    BYTE    cGreenShift;      // przesunięcie bitów koloru zielonego
    BYTE    cBlueBits;         // liczba bitów koloru niebieskiego
    BYTE    cBlueShift;        // przesunięcie bitów koloru niebieskiego
    BYTE    cAlphaBits;        // liczba bitów alfa
    BYTE    cAlphaShift;       // przesunięcie bitów alfa
    BYTE    cAccumBits;        // liczba bitów bufora akumulacji
    BYTE    cAccumRedBits;     // liczba bitów akumulacji czerwieni
    BYTE    cAccumGreenBits;   // liczba bitów akumulacji zieleni
    BYTE    cAccumBlueBits;    // liczba bitów akumulacji błękitu
    BYTE    cAccumAlphaBits;   // liczba bitów akumulacji alfa
    BYTE    cDepthBits;        // liczba bitów bufora głębi
    BYTE    cStencilBits;      // liczba bitów bufora powielania
    BYTE    cAuxBuffers;       // liczba buforów pomocniczych
    BYTE    iLayerType;        // nie jest już wykorzystywany
    BYTE    bReserved;         // liczba podkładanych i nakładanych jednostek
    DWORD   dwLayerMask;      // nie jest już wykorzystywany
    DWORD   dwVisibleMask;    // indeks podłożonej płaszczyzny
    DWORD   dwDamageMask;     // nie jest już wykorzystywany
} PIXELFORMATDESCRIPTOR;
```

Teraz należy przejść do omówienia najważniejszych pól tej struktury.

## Pole `nSize`

Pole to opisuje rozmiar struktury i powinno zostać zainicjowane w poniższy sposób:

```
pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
```

Wymaganie to jest dość oczywiste i typowe w przypadku struktur przekazywanych za pośrednictwem wskaźnika. Często podczas wykonywania różnych operacji potrzebna jest możliwość określenia wielkości pamięci zajmowanej przez strukturę, a umieszczenie tej wartości w pierwszym polu umożliwia szybkie jej pobranie przez dereferencję wskaźnika.

## Pole dwFlags

Pole dwFlags określa właściwości bufora pikseli. W tabeli 2.5 zaprezentowane zostały najczęściej używane wartości znaczników określających te właściwości.

**Tabela 2.5.** Wartości pola dwFlags

Wartość	Znaczenie
PFD_DRAW_TO_WINDOW	Bufor umożliwia tworzenie grafiki w oknie
PFD_SUPPORT_OPENGL	Bufor umożliwia tworzenie grafiki OpenGL
PFD_DOUBLEBUFFER	Podwójne buforowanie (w obecnej implementacji znacznik ten oraz znacznik PFD_SUPPORT_GDI wykluczają się wzajemnie)
PFD_SWAP_LAYER_BUFFERS	Określa, że urządzenie może przełączać pojedyncze plany warstw z formatami pikseli o podwójnym buforowaniu (w przeciwnym razie wszystkie plany warstw przełączane są grupowo); jeśli znacznik jest ustawiony, to dostępna jest funkcja <code>wglSwapLayerBuffers</code>
PFD_DEPTH_DONTCARE	Dla danego formatu pikseli może być — ale nie musi — wykorzystywany bufor głębi; aby utworzyć format pikseli, który nie korzysta z bufora głębi, należy koniecznie podać ten znacznik (w przeciwnym razie będzie wykorzystywany bufor głębi)
PFD_DOUBLEBUFFER_DONTCARE	Piksele mogą być buforowane pojedynczo lub podwójnie

## Pole iPixelFormat

Pole iPixelFormat określa typ danych piksela. Może ono przyjmować jedną z poniższych wartości:

- ◆ PFD\_TYPE\_RGBA — piksele opisane w standardzie RGBA (dla każdego piksela określona są: kolor czerwony, kolor zielony, kolor niebieski i współczynnik alfa);
- ◆ PFD\_TYPE\_COLORINDEX — piksele opisane są za pomocą wartości indeksu koloru.

W analizowanych zastosowaniach nadawana będzie polu iPixelFormat zawsze wartość PFD\_TYPE\_RGBA, co pozwala skorzystać ze standardowego modelu kolorów RGB rozszerzonego o kanał alfa umożliwiający realizację efektu przezroczystości.

## Pole cColorBits

Pole cColorBits opisuje liczbę bitów określających kolor piksela. Pole to może obecnie przyjmować wartości 8, 16, 24 i 32. Jeśli karta grafiki nie umożliwia reprezentacji koloru pikseli za pomocą żądanej liczby bitów, to przyjmowana jest największa możliwa wartość. Jeśli na przykład nadana zostanie polu cColorBits wartość 24, a karta grafiki nie umożliwia opisu pikseli za pomocą 24 bitów, to przyjęta zostanie wartość 16.

Po utworzeniu struktury PIXELFORMATDESCRIPTOR należy przekazać ją funkcji `ChoosePixelFormat()`. Próbuje ona dopasować formaty pikseli dostępne dla danego kontekstu urządzenia do opisu przekazanego za pomocą struktury PIXELFORMATDESCRIPTOR. Funkcja `ChoosePixelFormat()` zwraca wartość całkowitą reprezentującą indeks dostępnego formatu pikseli. Indeks ten należy przekazać następnie funkcji `SetPixelFormat()`, która wybiera ostatecznie format pikseli dla kontekstu urządzenia.

Poniższy fragment kodu pokazuje przykład struktury PIXELFORMATDESCRIPTOR i wybór formatu pikseli:

```

int nPixelFormat;                                // indeks formatu pikseli

static PIXELFORMATDESCRIPTOR pfd = {
    sizeof(PIXELFORMATDESCRIPTOR), // rozmiar struktury
    1,                          // wersja, zawsze równa 1
    PFD_DRAW_TO_WINDOW |        // grafika w oknie
    PFD_SUPPORT_OPENGL |       // grafika OpenGL
    PFD_DOUBLEBUFFER,          // podwójne buforowanie
    PFD_TYPE_RGBA,             // tryb kolorów RGBA
    32,                        // 32-bitowy opis kolorów
    0, 0, 0, 0, 0, 0,          // nie specyfikuje bitów kolorów
    0,                         // bez bufora alfa
    0,                         // nie specyfikuje bitu przesunięcia
    0,                         // bez bufora akumulacji
    0, 0, 0, 0,                // ignoruje bity akumulacji
    16,                        // 16-bitowy bufor Z
    0,                         // bez bufora powielania
    0,                         // bez buforów pomocniczych
    PFD_MAIN_PLANE,           // główna płaszczyzna rysowania
    0,                         // zarezerwowane
    0, 0, 0 };                  // ignoruje maski warstw

// wybiera najbardziej zgodny format pikseli i zwraca jego indeks
nPixelFormat = ChoosePixelFormat(hDC, &pfd);

// określa format pikseli dla danego kontekstu urządzenia
SetPixelFormat(hDC, nPixelFormat, &pfd);

```

We fragmencie tym zwraca uwagę duża liczba zer przypisywanych różnym polom struktury PIXELFORMATDESCRIPTOR. Oznacza to, że w praktyce nie trzeba określać wartości większości pól, aby zdefiniować format pikseli. Pola te mogą okazać się potrzebne w niektórych zastosowaniach, ale zwykle nadaje się im wartość 0.

## Aplikacja OpenGL w systemie Windows

Przedstawione dotąd informacje wystarczą do utworzenia podstawowego szkieletu aplikacji systemu Windows tworzącej grafikę OpenGL. Poniżej przedstawiony został kompletny kod źródłowy programu, który wyświetla w oknie wirujący trójkąt w kolorze czerwonym na czarnym tle.

```

#define WIN32_LEAN_AND_MEAN      // odchudza aplikację Windows

////// Pliki nagłówkowe
#include <windows.h>            // standardowy plik nagłówkowy Windows
#include <gl/gl.h>               // standard plik nagłówkowy OpenGL
#include <gl/glu.h>              // plik nagłówkowy dodatkowych bibliotek OpenGL
#include <gl/glaux.h>             // funkcje pomocnicze OpenGL

////// Zmienne globalne
float angle = 0.0f;              // bieżący kąt obrotu trójkąta
HDC g_HDC;                      // kontekst urządzenia

```

```
// funkcja określająca format pikseli dla kontekstu urządzenia
void SetupPixelFormat(HDC hDC)
{
    int nPixelFormat; // indeks formatu pikseli

    static PIXELFORMATDESCRIPTOR pfd = {
        sizeof(PIXELFORMATDESCRIPTOR), // rozmiar struktury
        1, // domyślna wersja
        PFD_DRAW_TO_WINDOW | // grafika w oknie
        PFD_SUPPORT_OPENGL | // grafika OpenGL
        PFD_DOUBLEBUFFER, // podwójne buforowanie
        PFD_TYPE_RGBA, // tryb kolorów RGBA
        32, // 32-bitowy opis kolorów
        0, 0, 0, 0, 0, 0, // nie specyfikuje bitów kolorów
        0, // bez bufora alfa
        0, // ignoruje bit przesunięcia
        0, // bez bufora akumulacji
        0, 0, 0, 0, // ignoruje bity akumulacji
        16, // 16-bitowy bufor Z
        0, // bez bufora powielania
        0, // bez buforów pomocniczych
        PFD_MAIN_PLANE, // główna płaszczyzna rysowania
        0, // zarezerwowane
        0, 0, 0 }; // ignoruje maski warstw

    // wybiera najodpowiedniejszy format pikseli
    nPixelFormat = ChoosePixelFormat(hDC, &pfd);

    // określa format pikseli dla kontekstu urządzenia
    SetPixelFormat(hDC, nPixelFormat, &pfd);
}

// procedura okienkowa
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HGLRC hRC; // kontekst tworzenia grafiki
    static HDC hDC; // kontekst urządzenia
    int width, height; // szerokość i wysokość okna

    switch(message)
    {
        case WM_CREATE: // okno jest tworzone
            hDC = GetDC(hwnd); // pobiera kontekst urządzenia dla okna
            g_HDC = hDC;
            SetupPixelFormat(hDC); // wywołuje funkcję określającą format pikseli

            // tworzy kontekst tworzenia grafiki i czyni go bieżącym
            hRC = wglCreateContext(hDC);
            wglMakeCurrent(hDC, hRC);

            return 0;
            break;

        case WM_CLOSE: // okno jest zamykane
            // dezaktywuje bieżący kontekst i usuwa go
            wglMakeCurrent(hDC, NULL);
            wglDeleteContext(hRC);
    }
}
```

```
// wstawia komunikat WM_QUIT do kolejki
PostQuitMessage(0);

return 0;
break;

case WM_SIZE:
    height = HIWORD(wParam);      // pobiera nową wysokość i szerokość okna
    width = LOWORD(wParam);

    if (height==0)                // unika dzielenia przez 0
    {
        height=1;
    }

    // nadaje nowe wymiary oknu OpenGL
    glViewport(0, 0, width, height);
    // określa macierz rzutowania
    glMatrixMode(GL_PROJECTION);
    // resetuje macierz rzutowania
    glLoadIdentity();

    // wyznacza proporcje obrazu
    gluPerspective(45.0f,(GLfloat)width/(GLfloat)height,1.0f,1000.0f);

    glMatrixMode(GL_MODELVIEW); // określa macierz widoku modelu
    glLoadIdentity();           // resetuje macierz widoku modelu

return 0;
break;

default:
break;
}

return (DefWindowProc(hwnd, message, wParam, lParam));
}

// punkt, w którym rozpoczyna się wykonywanie aplikacji
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
int nShowCmd)
{
    WNDCLASSEX windowClass; // klasa okien
    HWND hwnd;              // uchwyt okna
    MSG msg;                // komunikat
    bool done;               // znacznik zakończenia aplikacji

    // definicja klasy okna
    windowClass.cbSize= sizeof(WNDCLASSEX);
    windowClass.style= CS_HREDRAW | CS_VREDRAW;
    windowClass.lpfnWndProc= WndProc;
    windowClass.cbClsExtra= 0;
    windowClass.cbWndExtra= 0;
    windowClass.hInstance= hInstance;
    windowClass.hIcon= LoadIcon(NULL, IDI_APPLICATION); // domyślna ikona
    windowClass.hCursor= LoadCursor(NULL, IDC_ARROW);   // domyślny kurSOR
    windowClass.hbrBackground= NULL;                     // bez tła
    windowClass.lpszMenuName= NULL;                     // bez menu
```

```
windowClass.lpszClassName= "MojaKlasa";
windowClass.hIconSm= LoadIcon(NULL, IDI_WINLOGO); // logo Windows

// rejestruje klasę okna
if (!RegisterClassEx(&windowClass))
    return 0;

// tworzy okno
hwnd = CreateWindowEx(NULL,
    "MojaKlasa",
    "Aplikacja OpenGL",
    WS_OVERLAPPEDWINDOW | WS_VISIBLE |
    WS_SYSMENU | WS_CLIPCHILDREN |
    WS_CLIPSIBLINGS,
    100, 100,
    400, 400,
    NULL,
    NULL,
    hInstance,
    NULL); // rozszerzony styl okna
// nazwa klasy
// nazwa aplikacji
// styl
// współrzędne x, y
// szerokość i wysokość
// uchwyt okna nadziednego
// uchwyt menu
// instancja aplikacji
// bez dodatkowych parametrów

// sprawdza, czy utworzenie okna nie powiodło się (wtedy hwnd ma wartość NULL)
if (!hwnd)
    return 0;

ShowWindow(hwnd, SW_SHOW); // wyświetla okno
UpdateWindow(hwnd); // aktualizuje okno

done = false; // inicjuje zmienną warunku wykonania pętli

// pętla przetwarzania komunikatów
while (!done)
{
    PeekMessage(&msg, hwnd, NULL, NULL, PM_REMOVE);

    if (msg.message == WM_QUIT) // otrzymano komunikat WM_QUIT ?
    {
        done = true; // jeśli tak, to aplikacja kończy działanie
    }
    else
    {
        // tworzy grafikę
        // zeruje bufor ekranu i bufor głębi
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity(); // resetuje macierz widoku modelu

        angle = angle + 0.1f; // zwiększa licznik kąta obrotu
        if (angle >= 360.0f) // jeśli pełen obrót, to resetuje licznik
            angle = 0.0f;
        glTranslatef(0.0f,0.0f,-5.0f); // przesunięcie o 5 jednostek wstecz
        glRotatef(angle, 0.0f,0.0f,1.0f); // obrót dookoła osi Z

        glColor3f(1.0f,0.0f,0.0f); // wybiera kolor czerwony
        glBegin(GL_TRIANGLES);
            glVertex3f(0.0f,0.0f,0.0f);
            glVertex3f(1.0f,0.0f,0.0f);
            glVertex3f(1.0f,1.0f,0.0f);
        glEnd();
    }
}
```

```

        SwapBuffers(g_HDC);           // przełącza bufory

        TranslateMessage(&msg);      // tłumaczy komunikat i wysyła go do systemu
        DispatchMessage(&msg);
    }
}

return msg.wParam;
}

```

Większość kodu źródłowego tej aplikacji pochodzi z przykładu aplikacji systemu Windows zamieszczonego wcześniej. Mimo że bardzo podobne, to jednak oba programy różnią się kilkoma szczegółami.

Już na początku kodu można zauważać wprowadzenie dwóch zmiennych globalnych: angle i g\_HDC.

```

////// Zmienne globalne
float angle = 0.0f;           // bieżący kąt obrotu trójkąta
HDC g_HDC;                   // kontekst urządzenia

```

Zmienna angle przechowuje bieżącą wartość kąta obrotu. Obroty omówione zostaną szczegółowo w rozdziale 7., teraz potraktowane zostaną jako przykład kodu OpenGL.

Zmienna g\_HDC przechowuje uchwyt kontekstu urządzenia wykorzystywanego do tworzenia grafiki OpenGL.

Kolejna różnica polega na wprowadzeniu definicji funkcji SetupPixelFormat(). Funkcja ta wypełnia pola struktury PIXELFORMATDESCRIPTOR i wybiera opisany w ten sposób format piksela dla przekazanego jej kontekstu urządzenia hDC. Wybór formatu pikseli odbywa się w opisany wcześniej sposób.

Po zainicjowaniu struktury PIXELFORMATDESCRIPTOR wywoływana jest funkcja ChoosePixelFormat(), która ustala najodpowiedniejszy format pikseli. Zwrócony przez nią indeks formatu pikseli przekazuje się funkcji SetPixelFormat(), która dokonuje wyboru formatu pikseli dla kontekstu urządzenia. W ten sposób wszystko jest już gotowe do tworzenia grafiki.

Teraz należy przyjrzeć się procedurze okienkowej WndProc(). Jej kod zawiera między innymi wywołania funkcji tworzących okno grafiki OpenGL. Najpierw trzeba przeanalizować blok kodu, który jest wykonywany po otrzymaniu komunikatu WM\_CREATE:

```

case WM_CREATE:                // okno jest tworzone

    hDC = GetDC(hwnd);          // pobiera kontekst urządzenia dla okna
    g_HDC = hDC;                // przechowuje kontekst za pomocą zmiennej globalnej
    SetupPixelFormat(hDC);       // wywołuje funkcję określającą format pikseli

    // tworzy kontekst tworzenia grafiki i czyni go bieżącym
    hRC = wglCreateContext(hDC);
    wglMakeCurrent(hDC, hRC);

    return 0;
break;

```

Komunikat WM\_CREATE zostaje przekazany procedurze okienkowej, gdy tworzone jest okno aplikacji. Jest to najlepszy moment, by zainicjować też okno OpenGL. W tym celu trzeba najpierw pobrać kontekst urządzenia dla tworzonego okna korzystając z funkcji GetDC(). Kontekst ten przechowuje się w zmiennej globalnej g\_HDC, ponieważ będzie jeszcze wykorzystywany do przełączania buforów okna (wybrany wcześniej format pikseli zapewnia podwójne buforowanie). Następnie wywołuje się omówioną już funkcję SetupPixelFormat().

Kolejne dwa wiersze bloku obsługi komunikatu WM\_CREATE tworzą kontekst tworzenia grafiki okna OpenGL. Funkcja wglCreateContext() tworzy kontekst, a funkcja wglMakeCurrent() wybiera go jako kontekst bieżący.

W ten sposób utworzone zostało okno grafiki OpenGL. Teraz należy przyjrzeć się blokom obsługi okna i tworzenia grafiki.

Kolejny blok kodu procedury okienkowej obsługuje komunikat WM\_SIZE:

```
case WM_SIZE:
    height = HIWORD(lParam);      // pobiera nową wysokość i szerokość okna
    width = LOWORD(lParam);

    if (height==0)                // zapobiega dzieleniu przez 0
    {
        height=1;
    }

    // nadaje nowe wymiary oknu OpenGL
    glViewport(0, 0, width, height);
    // określa macierz rzutowania
    glMatrixMode(GL_PROJECTION);
    // resetuje macierz rzutowania
    glLoadIdentity();

    // wyznacza proporcje obrazu
    gluPerspective(45.0f,(GLfloat)width/(GLfloat)height,1.0f,1000.0f);

    glMatrixMode(GL_MODELVIEW); // określa macierz widoku modelu
    glLoadIdentity();           // resetuje macierz widoku modelu

    return 0;
break;
```

Komunikat WM\_SIZE przekazywany jest procedurze okienkowej na skutek zdarzenia zmiany rozmiarów okna. Trzeba wtedy odpowiednio zmienić także rozmiary okna OpenGL. Korzystając z parametru lParam przekazanego procedurze okienkowej należy ustalić najpierw nowe rozmiary okna. Parametr ten jest typu LPARAM i posiada 32-bitową reprezentację. W przypadku komunikatu WM\_SIZE zawiera on dwie wartości 16-bitowe określające nową szerokość i wysokość okna. Do ich pobrania wykorzystuje się makrodefinicje LOWORD i HIWORD zdefiniowane jak poniżej:

```
WORD LOWORD(DWORD dwValue); // zwraca młodsze słowo 16-bitowe
WORD HIWORD(DWORD dwValue); // zwraca starsze słowo 16-bitowe
```

Młodsze słowo reprezentuje szerokość okna, a starsze jego wysokość. Po uzyskaniu tych wartości trzeba ustalić sposób przeskalowania okna OpenGL.

Najpierw należy zabezpieczyć się jednak przed możliwością wykonania dzielenia przez zero. Błąd taki mógłby wystąpić podczas określania *rzutowania perspektywy* i dlatego trzeba upewnić się, że nowa wysokość okna jest różna od zero.

Funkcja `glViewport()` przeskala okno OpenGL do nowych rozmiarów. Omówienie pozostały kodu OpenGL trzeba na razie odłożyć do rozdziału 5., w którym przedstawione zostaną macierze rzutowania. Teraz należy zadowolić się informacją, że kod ten resetuje prezentację grafiki, ponieważ zmieniły się wymiary okna.

Ostatni istotny fragment kodu wprowadzony w tej wersji aplikacji znajduje się w pętli przetwarzania komunikatów:

```
// tworzy grafikę
// zeruje bufor ekranu i bufor głębi
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity(); // resetuje macierz widoku modelu

angle = angle + 0.1f; // zwiększa licznik kąta obrotu
if (angle >= 360.0f) // jeśli pełen obrót, to resetuje licznik
    angle = 0.0f;
glTranslatef(0.0f, 0.0f, -5.0f); // przesunięcie o 5 jednostek wstecz
glRotatef(angle, 0.0f, 0.0f, 1.0f); // obrót dookoła osi Z

glColor3f(1.0f, 0.0f, 0.0f); // wybiera kolor czerwony
glBegin(GL_TRIANGLES);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(1.0f, 0.0f, 0.0f);
    glVertex3f(1.0f, 1.0f, 0.0f);
glEnd();

SwapBuffers(g_HDC); // przełączka bufory

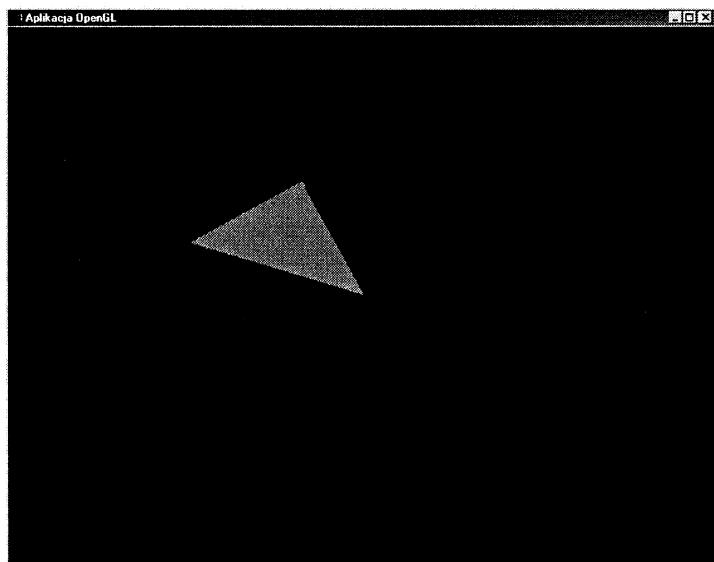
TranslateMessage(&msg); // tłumaczy komunikat i wysyła go do systemu
DispatchMessage(&msg);
```

Powyższy fragment kodu umieszczony zostaje w pętli przetwarzania komunikatów po wywołaniu funkcji `PeekMessage()`. Funkcja `glClear()` zeruje bufory kolorów i głębi, co w efekcie daje wypełnienie okna czarnym tłem. Pozostała część kodu rysuje i obraca trójkąt w kolorze czerwonym. Czytelnik powinien teraz skompilować program i uruchomić go, a następnie poeksperymentować z wartościami przekazywanymi funkcjom OpenGL i sprawdzić, jaki mają wpływ na tworzoną grafikę.

Po funkcjach OpenGL można skorzystać jeszcze ze zmiennej globalnej `g_HDC` przechowującej kontekst urządzenia. Przekazuje się ją jako parametr funkcji `SwapBuffers()`, która powoduje przełączenie buforów, zapewniając tym samym płynność animacji. Rysunek 2.4 prezentuje efekt działania programu.

I to wszystko! Zbudowana w ten sposób została pierwsza kompletna aplikacja OpenGL. Posłuży ona jako szkielet gier i programów tworzonych w kolejnych rozdziałach. Istnieje jednak jeszcze jeden aspekt tworzenia aplikacji OpenGL, który powinnien zostać omówiony w tym miejscu: grafika pełnoekranowa.

**Rysunek 2.4.**  
*Okno aplikacji  
OpenGL*



## Pełnoekranowe aplikacje OpenGL

Nie każdy użytkownik będzie zachwycony perspektywą korzystania z gry działającej w oknie. Większość gier korzystających z grafiki trójwymiarowej pracuje w trybie pełnoekranowym. Również tworzone tu programy będą posiadać taką możliwość. W tym celu czytelnik powinien stworzyć nową rozbudowaną wersję przedstawionej wcześniej aplikacji OpenGL na podstawie modyfikacji kodu, które teraz zostaną omówione.

Na początku należy utworzyć zmienną globalną o nazwie `fullScreen`, która będzie informować, czy aplikacja pracuje w trybie pełnoekranowym. Można zdefiniować ją następująco:

```
bool fullScreen = TRUE; // aplikacja rozpocznie pracę w trybie pełnoekranowym
```

Należy pamiętać, że powinna to być zmienna globalna, ponieważ będzie wykorzystywana w całym programie do sprawdzenia trybu pracy aplikacji.

Kolejna modyfikacja polegać będzie na dodaniu kodu przełączającego tryb okienkowy na pełnoekranowy. W tym celu można wykorzystać strukturę `DEVMODE`, która zawiera informacje o inicjalizacji i konfiguracji urządzenia wyjściowego. Przełączając aplikację w tryb pełnoekranowy należy pamiętać o kilku rzeczach. Należy upewnić się na przykład, czy wysokość i szerokość okna, która została podana przy jej tworzeniu jest taka sama jak wysokość i szerokość, która jest podana w strukturze `DEVMODE`. Trzeba także pamiętać, aby zmienić konfigurację urządzenia wyjściowego, *zanim* utworzone zostanie okno. Jeśli nie będzie się przestrzegać tych zaleceń, można wpaść w tarapaty. Prostym sposobem ich uniknięcia będzie wykorzystanie zmiennych definiujących rozmiary okna.

Kod przełączający aplikację na tryb pełnoekranowy jest dość prosty. Sprowadza się on do odpowiedniego wypełnienia kilku pól struktury DEVMODE i wywołania funkcji ChangeDisplaySettings():

```
DEVMODE devModeScreen; // tryb urządzenia
memset(&dmScreenSettings, 0, sizeof(dmScreenSettings)); // zeruje strukturę
devModeScreen.dmSize = sizeof(dmScreenSettings); // określa rozmiar struktury
devModeScreen.dmPelsWidth = width; // określa szerokość
devModeScreen.dmPelsHeight = height; // określa wysokość
devModeScreen.dmBitsPerPel = bits; // liczba bitów/piksel
devModeScreen.dmFields = DM_BITSPERPEL | DM_PELSWIDTH | DM_PELSHEIGHT;

if (ChangeDisplaySettings(&devModeScreen, CDS_FULLSCREEN) != DISP_CHANGE_SUCCESSFUL)
{
    // jeśli wybór trybu pełnoekranowego nie powiodł się.
    // to aplikacja znajdzie się w trybie okienkowym
    fullScreen = false;
}
```

Funkcja ChangeDisplaySettings() zmienia konfigurację domyślnego urządzenia wyświetlania na opisaną przez przekazaną jej strukturę DEVMODE. Parametr CDS\_FULLSCREEN powoduje usunięcie belki zadań z ekranu i sprawia, że system Windows przestaje zajmować się zawartością ekranu podczas zmian rozmiarów i przemieszczania okien w nowym trybie. Jeśli zmiana trybu powiedzie się, to funkcja zwróci wartość DISP\_CHANGE\_SUCCESSFUL i aplikacja znajdzie się w trybie pełnoekranowym. W przeciwnym razie trzeba będzie nadać zmiennej fullScreen wartość false oznaczającą pracę w trybie okienkowym.

Po przełączeniu aplikacji w tryb pełnoekranowy mimo wszystko należy utworzyć jej okno. Ponieważ parametry okien w trybie pełnoekranowym są inne niż w trybie okienkowym, trzeba dysponować dwoma ich zestawami. W przypadku trybu okienkowego parametry okna będą takie jak w poprzednich przykładach. Natomiast w trybie pełnoekranowym należy użyć znacznika WS\_EX\_APPWINDOW dla rozszerzonego stylu okna i znacznika WS\_POPUP dla zwykłego stylu okna. Znacznik WS\_EX\_APPWINDOW wymusza zwinięcie bieżącego okna do paska zadań po wyświetleniu okna aplikacji. Znacznik WS\_POPUP powoduje utworzenie okna bez ramki, co jest pożądane w trybie pełnoekranowym. W trybie pełnoekranowym należy także usunąć z ekranu wskaźnik myszy korzystając z funkcji ShowCursor(). Poniższy fragment kodu ilustruje sposób określenia stylu okna i ukrycia kurSORA myszy w trybie pełnoekranowym i okienkowym:

```
if (fullScreen)
{
    extendedWindowStyle = WS_EX_APPWINDOW; // zwija aktywne okno
    windowStyle = WS_POPUP; // okno bez ramki
    ShowCursor(FALSE); // chowa wskaźnik myszy
}
else
{
    extendedWindowStyle = NULL;
    windowStyle = WS_OVERLAPPEDWINDOW | WS_VISIBLE |
        WS_SYSMENU | WS_CLIPCHILDREN | WS_CLIPSIBLINGS;
}
```

Kolejna modyfikacja ma bardziej ogólny charakter i służy poprawie możliwości wyświetlania grafiki OpenGL w oknie systemu Windows. Funkcja AdjustWindowRectEx()

umożliwia wyznaczenie rozmiarów okna systemu Windows na podstawie zadanych rozmiarów prostokąta jego zawartości. Dla grafiki OpenGL oznacza to w praktyce, że granice utworzonego okna nie będą przesłaniać zewnętrznych fragmentów okna OpenGL. W ten sposób aplikacja OpenGL będzie dysponować zawsze maksymalnie dostępnym oknem graficznym. Funkcji `AdjustWindowRectEx()` należy przekazać strukturę `RECT` oraz zwykły i rozszerzony styl okna. W programie trzeba więc dodać zmienną `windowRect`. Wykorzystany w tym celu zostanie poniższy fragment kodu:

```
RECT windowRect;           // obszar okna wykorzystywany do tworzenia grafiki
windowRect.top = 0;         // górnny, lewy narożnik
windowRect.left = 0;
windowRect.bottom = screenHeight; // dolny, prawy narożnik
windowRect.right = screenWidth;

// koryguje rozmiar okna
AdjustWindowRectEx(&windowRect, windowStyle, FALSE, extendedWindowStyle);
```

Zamiast umieszczać po raz kolejny kompletny kod źródłowy zmodyfikowanej aplikacji, odsyłamy czytelnika do kodu programu o nazwie `OpenGLWindow2` zamieszczonego na dysku CD. Dodanie możliwości pracy pełnoekranowej nie wymaga zbyt wielu modyfikacji w programie. Po odpowiednim rozszerzeniu kodu można także zapytać użytkownika podczas uruchomiania aplikacji o pożądany tryb jej pracy. Wszystkie przykłady programów omawiane w dalszej części książki będą mogły pracować tak w trybie okienkowym, jak i pełnoekranowym.

## Podsumowanie

Platforma Microsoft Windows jest wielozadaniowym systemem operacyjnym. Każdemu z procesów przyznawany jest w tym systemie cyklicznie przedział czasu, w którym wykonywany jest jego kod. Procedura szeregująca procesy optymalizuje wykonanie aplikacji biorąc pod uwagę ich wymagania i priorytety.

Każdy z procesów może uruchamiać wiele niezależnie wykonywanych wątków, co sprawia, że wielozadaniowość jest możliwa także w obrębie pojedynczej aplikacji. Najnowsze wersje systemu Windows oferują możliwość wykorzystania *włókien*, które zapewniają wielozadaniowość także na poziomie poszczególnych wątków.

Główną funkcję i punkt początkowy aplikacji systemu Windows stanowi funkcja `WinMain()`. Do obsługi komunikatów wysyłanych do aplikacji przez system służy natomiast procedura okienkowa. Przekazywane jej komunikaty rozpoznawane są zwykle za pomocą instrukcji wyboru `switch` i odpowiednio obsługiwane.

Struktura `WNDCLASSEX` stanowi zbiór atrybutów wykorzystywanych do zdefiniowania klasy okna. Klasa okna musi zostać zdefiniowana i zarejestrowana, zanim utworzone zostaną okna aplikacji.

Pętla przetwarzania komunikatów stanowi część funkcji `WinMain()` i wykonuje następujące zadania:

- ◆ pobiera komunikat z kolejki za pomocą funkcji PeekMessage();
- ◆ tłumaczy komunikat
- ◆ przesyła komunikat do systemu.

W rozdziale przedstawiony został przykład kompletnej aplikacji systemu Windows ilustrujący sposób obsługi komunikatów i tworzenia okna aplikacji.

Funkcje WGL stanowią rozszerzenie interfejsu programowego systemu Windows, co umożliwia wykonywanie aplikacji OpenGL. Najważniejsze z nich związane są z tworzeniem kontekstu grafiki OpenGL, który służy do przechowywania ustawień i komend OpenGL. Można przy tym używać jednocześnie wielu kontekstów grafiki OpenGL.

W strukturze typu PIXELFORMATDESCRIPTOR umieszcza się opis kontekstu urządzenia, który będzie używany do tworzenia grafiki OpenGL. Struktura ta musi zostać zdefiniowana, zanim kod OpenGL rozpoczęcie tworzenie grafiki w oknie systemu Windows.

Większość gier stworzonych na bazie grafiki trójwymiarowej wykorzystuje pełnoekranowy tryb pracy. W rozdziale niniejszym pokazany został sposób implementacji pełnoekranowego trybu pracy związanej z potrzebami aplikacji OpenGL. Jego ilustracją jest program OpenGLWindow2, którego pełen kod źródłowy znajduje się na dysku CD.

## Rozdział 3.

# Przegląd teorii grafiki trójwymiarowej

Przedstawionych teraz zostanie kilka podstawowych zagadnień z teorii grafiki trójwymiarowej, które będą pomocne przy tworzeniu grafiki gier w kolejnych rozdziałach. Solidne podstawy w zakresie teorii grafiki trójwymiarowej nie są absolutnie konieczne, aby rozpocząć przygodę z programowaniem w OpenGL, ale z pewnością ułatwiają to zadanie. Przedstawione tu zagadnienia stanowią jedynie wierzchołek góry lodowej, nie jest bowiem możliwe wyczerpujące omówienie teorii grafiki trójwymiarowej w pojedynczym rozdziale. Więcej informacji na ten temat odnajdzie czytelnik w materiałach wymienionych w suplementie książki. W bieżącym rozdziale przedstawionych zostanie jedynie kilka podstawowych zagadnień.

Należeć do nich będą:

- ◆ skalary i wektory;
- ◆ macierze;
- ◆ przekształcenia;
- ◆ rzutowania;
- ◆ oświetlenie;
- ◆ odwzorowania tekstur.

## Skalary, punkty i wektory

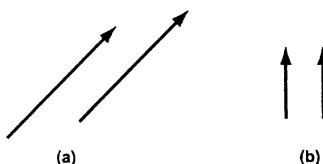
Grafika trójwymiarowa reprezentuje szereg obiektów geometrycznych tworzących trójwymiarowy świat gry. Każdy z takich światów jest inny i tworzy go kombinacja różnych obiektów, które zawsze można jednak rozłożyć na poniższe typy proste:

- ◆ **Skalary.** Chociaż skalar nie jest typem geometrycznym, to wykorzystywany jest jako jednostka miary. Skalar reprezentuje wartość pewnej wielkości fizycznej. Wartością skalarną będzie na przykład temperatura pokazywana przez termometr, długość odcinka lub natężenie oświetlenia trójwymiarowej sceny.

- ◆ **Punkty.** Punkt stanowi fundamentalne pojęcie geometrii i reprezentuje położenie w trójwymiarowej przestrzeni. W przypadku analizowanych zastosowań położenie to opisane będzie za pomocą trzech wartości skalarnych reprezentujących odległość na każdej z osi układu współrzędnych:  $x$ ,  $y$  i  $z$ .
- ◆ **Wektory.** Wektor reprezentuje wielkość fizyczną posiadającą wartość oraz kierunek. Na przykład ruch kuli toczącej się po stole bilardowym opisany jest przez określony kierunek oraz wartość reprezentującą prędkość kuli. W ten sam sposób opisać można wektory grafiki trójwymiarowej. Reprezentację wektora stanowi odcinek posiadający pewien kierunek w przestrzeni oraz wartość odpowiadającą długością odcinka. Reprezentację wektorów prezentuje rysunek 3.1. Pokazano na nim pary równych wektorów. Dwa wektory są równe, jeśli posiadają ten sam kierunek i wartość. Natomiast ich położenie w przestrzeni nie jest określone i może być różne. Analogią będzie tutaj ruch dwu różnych kul bilardowych posiadających tę samą prędkość i kierunek, ale znajdujących się w różnych miejscach stołu. W takich sytuacjach mówi się, że wektory prędkości obu kul są równe.

**Rysunek 3.1.**

*Parę równych wektorów*



## Długość wektora

Długość wektora — zwana także jego wartością bądź normą — zapisywana jest przez ujęcie nazwy wektora w parę pionowych kresek. Długość wektora  $A$  zapisze się więc jako:

$$|A|$$

Długość wektora stanowi miarę odległości, można więc wyznaczyć ją korzystając z twierdzenia Pitagorasa. Ponieważ dalsze obliczenia będą prowadzone w przestrzeni, a nie na płaszczyźnie, to twierdzenie to należy rozszerzyć o trzecią współrzędną  $z$ . W ten sposób długość wektora  $A$  wyznaczyć można będzie w następujący sposób:

$$|A| = \sqrt{A_x^2 + A_y^2 + A_z^2}$$

## Normalizacja wektora

Znając długość wektora można poddać go *normalizacji*. Normalizacja polega na zredukowaniu go do jednostkowej długości, dzięki czemu uzyskuje on szereg właściwości pożądanych przy obliczeniach dotyczących grafiki trójwymiarowej, które zostaną omówione później. Aby znormalizować wektor, należy podzielić go przez jego długość:

$$n = N / |N|$$

Znormalizowany wektor  $n$  jest więc równy wektorowi  $N$  podzielonemu przez wartość wektora  $N$ .

## Dodawanie wektorów

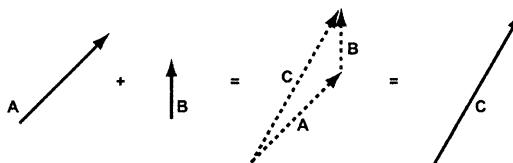
Podobnie jak w przypadku skalarów, tak i dla wektorów można wykonywać różne działania matematyczne. Podstawowymi operacjami są dodawanie i mnożenie wektora przez skalar. Jeśli wektor zapisany zostanie w postaci  $V(x,y,z)$ , to sumę dwóch wektorów  $A(x_a,y_a,z_a)$  i  $B(x_b,y_b,z_b)$  przedstawić będzie można jak poniżej:

$$A(x_a,y_a,z_a) + B(x_b,y_b,z_b) = C(x_a + x_b, y_a + y_b, z_a + z_b)$$

Ilustrację tej operacji przedstawia rysunek 3.2. Sumą wektorów  $A$  i  $B$  jest nowy wektor  $C$ .

**Rysunek 3.2.**

Dodawanie wektorów



Odejmowanie wektorów wykonuje się w ten sam sposób co ich dodawanie (z tą jednak różnicą, że odejmuje się ich składowe).

## Mnożenie wektora przez skalar

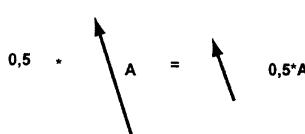
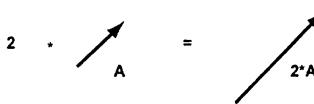
Rezultatem mnożenia wektora przez skalar jest wektor, którego długość i zwrot mogły się zmienić, ale kierunek został zachowany. Mnożąc na przykład wektor przez wartość 4 uzyskuje się nowy wektor o tym samym kierunku i zwrocie, ale o czterokrotnie większej długości. Mnożąc wektor przez wartość -1 otrzymujemy się natomiast wektor o tej samej długości i kierunku, ale o przeciwnym zwrocie. Operację mnożenia wektora przez skalar opisuje poniższe równanie:

$$s * A(x_a, y_a, z_a) = B(s * x_a, s * y_a, s * z_a)$$

Mnożenie wektora przez skalar polega więc na pomnożeniu jego składowych przez wartość skalarną. Operację tę ilustruje rysunek 3.3.

**Rysunek 3.3.**

Mnożenie wektora przez skalar



## Iloczyn skalarny wektorów

Iloczyn skalarny oraz iloczyn wektorowy stanowi przykład operacji na wektorach często wykorzystywanych w grafice trójwymiarowej. Iloczyn skalarny wektorów jest przydatny

przede wszystkim do wyznaczania kąta pomiędzy dwoma wektorami. Aby go wyznaczyć, należy dodać do siebie iloczyny odpowiednich składowych wektorów i w ten sposób otrzymać w wyniku wartość skalarną. Do wyznaczenia wartości iloczynu skalarnego dwóch wektorów  $A$  i  $B$  można wykorzystać poniższe równania:

$$\begin{aligned} A \cdot B &= A_x * B_x + A_y * B_y \\ A \cdot B &= |A| * |B| * \cos \theta \end{aligned}$$

Pierwsze z równań umożliwia wyznaczenie iloczynu skalarnego jako sumy iloczynów składowych wektorów. Przekształcając drugie z równań otrzymuje się zależność pozwalającą wyznaczyć kąt pomiędzy wektorami.

$$\cos \theta = (A \cdot B) / (|A| * |B|)$$

Jeśli wektory  $A$  i  $B$  posiadają jednostkową długość, to zależność ta uproszcza się do poniższej postaci:

$$\cos \theta = A \cdot B$$

Zależność w tej postaci jest często wykorzystywana przy wykonywaniu obliczeń związanych z tworzeniem grafiki trójwymiarowej. Iloczyn skalarny posiada także następujące właściwości:

- ◆  $A \cdot B = 0$ , jeśli kąt pomiędzy wektorami  $A$  i  $B$  wynosi  $90^\circ$ ;
- ◆  $A \cdot B > 0$ , jeśli kąt pomiędzy wektorami  $A$  i  $B$  jest mniejszy od  $90^\circ$ ;
- ◆  $A \cdot B < 0$ , jeśli kąt pomiędzy wektorami  $A$  i  $B$  jest większy od  $90^\circ$ ;
- ◆  $A \cdot B = |A|^2 = |B|^2$ , jeśli wektory  $A$  i  $B$  są równe.

## Iloczyn wektorowy

Kolejnym działaniem na wektorach często wykorzystywanym w grafice trójwymiarowej jest iloczyn wektorowy, stosowany na przykład do wykrywania zderzeń obiektów czy wyznaczania ich oświetlenia. Iloczyn dwóch wektorów  $A$  i  $B$  definiuje się następująco:

$$A \times B = |A| * |B| * \sin \theta * n$$

gdzie  $\theta$  jest kątem pomiędzy wektorami, a  $n$  jest znormalizowanym wektorem prostopadłym do wektorów  $A$  i  $B$ .

Korzystając z tego równania można wyznaczyć kąt pomiędzy wektorami  $A$  i  $B$  lub wektor  $n$ , ale iloczyn wektorowy wyznacza się zwykle w inny sposób.

Ponieważ powyższe równanie rzadko stosowane jest w praktyce, poniżej zaprezentowany został inny sposób wyznaczenia iloczynu wektorowego dwóch wektorów  $A$  i  $B$ :

$$C = A \times B = (A_y * B_z - A_z * B_y, A_z * B_x - A_x * B_z, A_x * B_y - A_y * B_x)$$

Wynikiem iloczynu wektorowego jest znormalizowany wektor. Równanie to umożliwia wyznaczenie każdej składowej tego wektora na podstawie pozostałych składowych mnożonych wektorów. Na przykład składową  $C_x$  wyznacza się na podstawie składowych  $A_y$ ,  $A_z$ ,  $B_y$  i  $B_z$ .

# Macierze

Macierz jest dwuwymiarową tablicą liczb o określonej liczbie wierszy i kolumn. Macierze mogą posiadać także trzeci wymiar, ale do opisu grafiki trójwymiarowej wystarczą macierze dwuwymiarowe. Mówiąc, że macierz posiada wymiary  $m \times n$ , należy rozumieć przez to, że macierz posiada  $m$  wierszy i  $n$  kolumn. Tak więc na przykład macierz o 3 wierszach i 2 kolumnach będzie zapisywana w skrócie jako macierz  $3 \times 2$ . Poniżej przedstawiona została macierz  $M$  o rozmiarach  $3 \times 3$ :

$$M = \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}$$

Wybrany element macierzy oznacza się podając jego wiersz i kolumnę. W powyższym przykładzie macierzy  $M$  element posiadający wartość 5 posiada pozycję (1, 1). Czytelnik może zastanawiać się, dlaczego nie jest to pozycja (2, 2). Macierze będą tu jednak implementowane w języku C lub C++ jako dwuwymiarowe tablice. Pierwszy element takiej tablicy będzie zawsze oznaczony jako (0, 0), drugi jako (0, 1) i tak dalej. Sposób oznaczenia elementów macierzy ilustruje poniższy przykład:

$$M = \begin{vmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{vmatrix}$$

Powyższy sposób opisu macierzy stosowany będzie w tekście książki i w kodzie źródłowym programów.

Teraz można już przejść do omówienia działań wykonywanych na macierzach.

## Macierz jednostkowa

W działaniach wykonywanych na macierzach pojawia się szczególnie użyteczny rodzaj macierzy: *macierz jednostkowa*. Macierz ta posiada na przekątnej elementy o wartości 1, a pozostałe elementy macierzy posiadają wartość 0. Macierze jednostkowe mogą mieć dowolne rozmiary, ale zawsze muszą być kwadratowe, czyli o rozmiarach  $m \times m$ . Poniżej znajduje się przykład macierzy jednostkowej o rozmiarach  $3 \times 3$ :

$$M = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Interesującą właściwością macierzy jednostkowej jest to, że mnożąc przez nią inną macierz otrzymuje się w wyniku tę samą macierz. Mnożenie macierzy omówione zostanie wkrótce.

## Macierz zerowa

Chociaż macierze zerowe nie będą wykorzystywane w praktyce, to należy je jednak omówić. *Macierz zerowa* posiada wszystkie elementy równe 0. Dodanie lub mnożenie

innej macierzy przez macierz zerową jest identyczne z dodawaniem lub mnożeniem przez wartość 0 w przypadku skalarów. Poniżej przedstawiamy przykład macierzy zerowej o rozmiarach  $3 \times 3$ :

$$M = \begin{vmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{vmatrix}$$

## Dodawanie i odejmowanie macierzy

Dodawanie i odejmowanie macierzy jest bardzo proste. Elementy macierzy wynikowej powstają na skutek dodania lub odjęcia odpowiednich elementów macierzy wyjściowych. Poniżej przedstawiony został przykład dodawania i odejmowania dwu macierzy  $M$  i  $N$  o rozmiarach  $3 \times 3$ .

$$M = \begin{vmatrix} 2 & 2 & 2 \\ 0 & 1 & 4 \\ 7 & 3 & 1 \end{vmatrix} \quad N = \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}$$

$$M + N = \begin{vmatrix} 2 & 2 & 2 \\ 0 & 1 & 4 \\ 7 & 3 & 1 \end{vmatrix} + \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix} = \begin{vmatrix} (2+1) & (2+2) & (2+3) \\ (0+4) & (1+5) & (4+6) \\ (7+7) & (3+8) & (1+9) \end{vmatrix} = \begin{vmatrix} 3 & 4 & 5 \\ 4 & 6 & 10 \\ 14 & 11 & 10 \end{vmatrix}$$

$$M - N = \begin{vmatrix} 2 & 2 & 2 \\ 0 & 1 & 4 \\ 7 & 3 & 1 \end{vmatrix} - \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix} = \begin{vmatrix} (2-1) & (2-2) & (2-3) \\ (0-4) & (1-5) & (4-6) \\ (7-7) & (3-8) & (1-9) \end{vmatrix} = \begin{vmatrix} 1 & 0 & -1 \\ -4 & -4 & -2 \\ 0 & -5 & -8 \end{vmatrix}$$

Jak łatwo zauważyc, element macierzy wynikowej  $M+N(0, 0)$  powstaje przez dodanie elementów  $M(0, 0)$  i  $N(0, 0)$ . W ten sam sposób powstają pozostałe elementy macierzy będącej wynikiem dodawania lub odejmowania macierzy. Działania te są wykonalne jedynie w przypadku macierzy o takich samych rozmiarach.

Dodawanie i odejmowanie macierzy jest łączne, czyli  $M+(N+\chi) = (M+N)+\chi$ . Odejmowanie macierzy nie jest jednak przemienne, ponieważ wynik operacji  $(M-N)$  nie musi być równy wynikowi operacji  $(N-M)$ .

## Mnożenie macierzy

Macierze można mnożyć przez wartość skalarną lub przez inną macierz. Pierwsza z tych operacji jest bardzo prosta i polega na pomnożeniu wszystkich elementów macierzy przez wartość skalarną. Poniżej przedstawiamy przykład mnożenia macierzy o rozmiarach  $2 \times 2$  przez wartość 2:

$$M = \begin{vmatrix} 1 & 3 \\ 2 & 0 \end{vmatrix} \quad k = 2$$

$$C = k * M = 2 * \begin{vmatrix} 1 & 3 \end{vmatrix} = \begin{vmatrix} (2*1) & (2*3) \\ (2*2) & (2*0) \end{vmatrix} = \begin{vmatrix} 2 & 6 \\ 4 & 0 \end{vmatrix}$$

Poniżej przedstawiamy ogólny zapis operacji mnożenia macierzy  $M$  o wymiarach  $3 \times 3$  przez wartość skalarną  $k$ .

$$M = \begin{vmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{vmatrix}$$

$$k * M = k * \begin{vmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{vmatrix} = \begin{vmatrix} k*m_{00} & k*m_{01} & k*m_{02} \\ k*m_{10} & k*m_{11} & k*m_{12} \\ k*m_{20} & k*m_{21} & k*m_{22} \end{vmatrix}$$

Jednym z przykładów zastosowania mnożenia macierzy przez wartość skalarną w grafice jest operacja skalowania, która zostanie omówiona w dalszej części rozdziału.

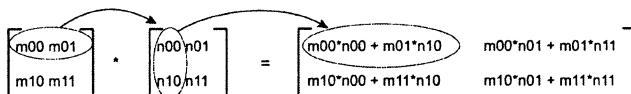
Mnożenie macierzy przez macierz jest dużo bardziej skomplikowane. Operacja ta możliwa jest tylko wtedy, gdy liczba kolumn pierwszej z macierzy równa jest liczbie wierszy drugiej macierzy. Jeśli zestawiona zostanie na przykład macierz  $A$  o rozmiarach  $3 \times 2$  z macierzą  $B$  o rozmiarach  $3 \times 3$ , to nie będzie można ich pomnożyć, ponieważ macierz  $A$  ma dwie kolumny, a macierz  $B$  trzy wiersze. Jeśli macierz  $A$  będzie na przykład macierzą o rozmiarach  $2 \times 3$ , to działanie mnożenia będzie wykonalne. W ogólnym przypadku — jeśli macierz  $A$  ma rozmiar  $m \times n$ , to macierz  $B$  musi mieć rozmiar  $n \times r$ , by mnożenie było wykonalne ( $m$  i  $r$  są dowolne).

Zanim wykonała zostanie operacja mnożenia dwu macierzy, można ustalić wymiary macierzy, która będzie wynikiem tego działania. Macierz ta będzie posiadać tyle wierszy, ile pierwsza z mnożonych macierzy, i tyle kolumn, ile ma druga z nich. Jeśli więc pomnożona na przykład zostanie macierz  $A$  o rozmiarach  $3 \times 1$  przez macierz  $B$  o rozmiarach  $1 \times 3$ , to w wyniku tego działania powstanie macierz  $C$  o rozmiarach  $3 \times 3$ . W ogólnym przypadku wynikowa macierz  $C$  będzie więc posiadać rozmiary  $m \times r$ .

A co z samym mnożeniem? Operacja mnożenia macierzy  $A$  przez macierz  $B$  polega na pomnożeniu wierszy macierzy  $A$  przez kolumny macierzy  $B$ . Każdy element wiersza macierzy  $A$  mnożony jest przez odpowiedni element kolumny macierzy  $B$ , a element macierzy wynikowej powstaje przez zsumowanie wyników poszczególnych mnożeń. Sposób mnożenia macierzy ilustruje rysunek 3.4.

**Rysunek 3.4.**

*Mnożenie macierzy*



Poniżej przedstawiony został przykład mnożenia macierzy  $A$  o rozmiarach  $2 \times 3$  przez macierz  $B$  o rozmiarach  $3 \times 3$ :

$$A = \begin{vmatrix} 1 & 3 & 4 \\ 2 & 0 & 2 \end{vmatrix} \quad B = \begin{vmatrix} 3 & 1 & 2 \\ 0 & 1 & 2 \\ 1 & 4 & 3 \end{vmatrix}$$

$$A * B = \begin{vmatrix} (1*3 + 3*0 + 4*1) & (1*1 + 3*1 + 4*4) & (1*2 + 3*2 + 4*3) \\ (2*3 + 0*0 + 2*1) & (2*1 + 0*1 + 2*4) & (2*2 + 0*2 + 2*3) \end{vmatrix}$$

$$A * B = \begin{vmatrix} 7 & 20 & 20 \\ 8 & 10 & 10 \end{vmatrix}$$

W ogólnym przypadku zapis operacji mnożenia macierzy o rozmiarach  $1 \times 3$  przez macierz o rozmiarach  $3 \times 3$  przedstawia się następująco:

$$M = \begin{vmatrix} m_{00} & m_{01} & m_{02} \\ m_{00} & m_{01} & m_{02} \end{vmatrix} \quad N = \begin{vmatrix} n_{10} & n_{11} & n_{12} \\ n_{20} & n_{21} & n_{22} \end{vmatrix}$$

$$M * N = \begin{vmatrix} (m_{00}*n_{10} + m_{01}*n_{11} + m_{02}*n_{12}) & (m_{00}*n_{10} + m_{01}*n_{11} + m_{02}*n_{12}) & (m_{00}*n_{10} + m_{01}*n_{11} + m_{02}*n_{12}) \\ (m_{00}*n_{20} + m_{01}*n_{21} + m_{02}*n_{22}) & (m_{00}*n_{20} + m_{01}*n_{21} + m_{02}*n_{22}) & (m_{00}*n_{20} + m_{01}*n_{21} + m_{02}*n_{22}) \end{vmatrix}$$

Mnożenie macierzy przez macierz nie jest działaniem przemiennym, czyli:

$$(M*N) \neq (N*M)$$

Inaczej mówiąc, mnożenie macierzy w jednym porządku spowoduje uzyskania innego wyniku niż mnożenie macierzy w porządku odwrotnym. Wyjątkiem od tej reguły jest mnożenie przez macierz jednostkową, zerową lub mnożenie dwu identycznych macierzy.

## Implementacja działań na macierzach

Macierze i wykonywane na nich działania można implementować na wiele sposobów. W tym podrozdziale stworzony zostanie osobny typ dla reprezentacji macierzy o rozmiarach  $3 \times 3$  oraz zaimplementowane zostanie każde z działań na macierzach za pomocą osobnej funkcji.

```
// typ reprezentujący macierze 3x3
typedef struct
{
    float mat[3][3];
} matrix3x3_t;
```

Typ ten wykorzystywany będzie przez funkcje implementujące działania na macierzach:

```
void MatrixAdd(matrix3x3_t* matrixA, matrix3x3_t* matrixB, matrix3x3_t* resultMatrix)
{
    // dodaje macierze matrixA i matrixB, a wynik umieszcza w resultMatrix

    // pętla przeglądająca wszystkie elementy macierzy
    for(int row=0; row < 3; row++)
    {
        for(int col=0; col < 3; col++)
        {
            // dodaje bieżące elementy macierzy
            resultMatrix->m[row][col] = matrixA->m[row][col] + matrixB->m[row][col];
        }
    }
}

void ScalarMatrixMult(float scalarValue, matrix3x3_t* matrixA, matrix3x3_t* resultMatrix)
{
    // mnoży macierz matrixA przez skalar scalarValue, a wynik umieszcza w resultMatrix

    // pętla przeglądająca wszystkie elementy macierzy
    for(int row=0; row < 3; row++)
    {
        for(int col=0; col < 3; col++)
        {
            // dodaje bieżące elementy macierzy
            resultMatrix->m[row][col] = scalarValue * matrixA->m[row][col];
        }
    }
}
```

```

        }

}

void MatrixMult(matrix3x3_t* matrixA, matrix3x3_t* matrixB, matrix3x3_t* resultMatrix)
{
    // mnoży macierz matrixA przez macierz matrixB, a rezultat umieszcza w resultMatrix

    float sum; // przechowuje sumę mnożonych elementów

    for(int row=0; row < 3; row++)
    {
        for(int col=0; col < 3; col++)
        {
            sum = 0;

            // mnoży wiersz macierzy A przez kolumnę macierzy B
            for(int k = 0; k < 3; k++)
            {
                sum += matrixA->m[row][k] * matrixA->m[k][col];
            }
            resultMatrix->m[row][col] = sum;
        }
    }
}

```

Na tym można zakończyć przedstawianie macierzy. Omówione teraz zostanie ich zastosowanie w grafice trójwymiarowej.

## Przekształcenia

Przekształcenia ożywiają trójwymiarowy świat gry. Pozwalają przemieszczać obiekty, powiększać je i pomniejszać, a nawet zmieniać ich wygląd. W rozdziale 5. pokazane zostanie to, że przekształcenia działają w tak zwanym lokalnym układzie współrzędnych. Jeśli początek tego układu znajduje się w początku ogólnego układu współrzędnych, to przekształcenia wykonywane są względem początku tego układu. Jeśli natomiast początek lokalnego układu znajduje się w na przykład w punkcie o współrzędnych (0, 10, 4), to przekształcenia będą wykonywane tak, jakby początek układu współrzędnych znajdował się w punkcie o współrzędnych (0, 10, 4).

Wykonanie przekształcenia polega w ogólnym przypadku na pomnożeniu przekształcającego punktu przez macierz przekształcenia. W rezultacie przekształcenia uzyskuje się nowy punkt. Jeśli dla punktu  $p$  wykonane zostanie przekształcenie reprezentowane przez macierz  $M$ , to uzyskany zostanie punkt  $p'$ , co można zapisać jak poniżej:

$$p' = M * p$$

Przekształcenia korzystają z tak zwanych *współrzędnych jednorodnych*. Pozwala to wykonywać przesunięcia i skalowania przekształceń, co w praktyce okazuje się przydatne. Aby skorzystać ze współrzędnych jednorodnych, należy umieścić wartość 1 jako ostatni element leżący na przekątnej macierzy przekształcenia. Wyjaśnione to zostanie bliżej przy analizie macierzy poszczególnych przekształceń.

## Przesunięcie

Przesunięcie polega na dodaniu do współrzędnych punktu odpowiednich składowych wektora przesunięcia. Macierz przesunięcia wygląda następująco:

$$MP = \begin{vmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Aby wykonać operację przesunięcia punktu korzystając z macierzy przekształcenia, trzeba zapisać współrzędne punktu w postaci następującego wektora:

$$p = \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix}$$

$$MP * p = \begin{vmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{vmatrix} * \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix} = \begin{vmatrix} (1*x) + (0*y) + (0*z) + (dx*1) \\ (0*x) + (1*y) + (0*z) + (dy*1) \\ (0*x) + (0*y) + (1*z) + (dz*1) \\ (0*x) + (0*y) + (0*z) + (1*1) \end{vmatrix} = \begin{vmatrix} x+dx \\ y+dy \\ z+dz \\ 1 \end{vmatrix}$$

W efekcie otrzyma się więc:

$$\begin{aligned} x' &= x + dx \\ y' &= y + dy \\ z' &= z + dz \end{aligned}$$

W ten sposób można przesunąć punkt lub wektor z jednego punktu trójwymiarowej przestrzeni do innego punktu. Należy zwrócić uwagę na to, że punkt opisany został za pomocą współrzędnych jednorodnych i w związku z tym na ostatniej pozycji reprezentującego go wektora pojawiła się wartość 1. Jeśli zastąpiona ona zostanie wartością 0, to wykonanie przesunięcia nie będzie możliwe, natomiast każda inna wartość spowoduje dodatkowe przeskalowanie przesunięcia.

## Obrót

Macierz obrotu wykorzystuje funkcje trygonometryczne, a obrót wokół każdej z osi wymaga osobnej macierzy o innych elementach.

Macierz obrotu wokół osi x zdefiniowana jest następująco:

$$M_{OX} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Jak łatwo zauważyc, obrót wokół osi x nie zmienia współrzędnych x. Podobnie w przypadku obrotu wokół osi y nie zmieniają się współrzędne y:

$$M_{OY} = \begin{vmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Obrót wokół osi z nie zmienia współrzędnych z:

$$M_{OZ} = \begin{vmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Aby obrócić dany punkt względem środka układu współrzędnych i wybranej osi, należy pomnożyć odpowiednią macierz obrotu przez wektor reprezentujący jednorodne współrzędne punktu. Poniżej przedstawiony został przykład tej operacji dla obrotu wokół osi z.

$$p = \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix}$$

$$\begin{aligned} M_{OZ} \cdot p &= \begin{vmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} |x| \\ |y| \\ |z| \\ |1| \end{vmatrix} = \begin{vmatrix} (x \cdot \cos \theta) - (y \cdot \sin \theta) + (0 \cdot z) + (0 \cdot 1) \\ (x \cdot \sin \theta) + (y \cdot \cos \theta) + (0 \cdot z) + (0 \cdot 1) \\ (0 \cdot x) + (0 \cdot y) + (1 \cdot z) + (0 \cdot 1) \\ (0 \cdot x) + (0 \cdot y) + (0 \cdot z) + (1 \cdot 1) \end{vmatrix} \\ &= \begin{vmatrix} (x \cdot \cos \theta) - (y \cdot \sin \theta) \\ (x \cdot \sin \theta) + (y \cdot \cos \theta) \\ z \\ 1 \end{vmatrix} \end{aligned}$$

Uzyskana jako wynik macierz kolumnowa oznacza, że:

$$\begin{aligned} x' &= (x \cdot \cos \theta) - (y \cdot \sin \theta) \\ y' &= (x \cdot \sin \theta) + (y \cdot \cos \theta) \\ z' &= z \end{aligned}$$

Z wykorzystaniem macierzy obrotów związane jest zagadnienie *składania przekształceń*. Zamiast mnożyć punkt przez wiele macierzy reprezentujących różne przekształcenia tworzy się jedną macierz reprezentującą złożenie przekształceń. Jako że mnożenie macierzy nie jest działaniem przemiennym, porządek składania przekształceń ma znaczenie. W przypadku obrotu można stworzyć macierz przekształcenia reprezentującą złożenie obrotu wokół osi z z obrotem wokół osi y i obrotem wokół osi x. Macierz  $M_O$  tego przekształcenia powstanie przez pomnożenie macierzy obrotów dookoła poszczególnych osi:

$$M_O = M_{OZ} * M_{OY} * M_{OX}$$

W wyniku tego działania otrzymany zostanie zestaw równań opisujących współrzędne nowego punktu, gdzie  $\gamma$  jest kątem obrotu wokół osi z,  $\beta$  kątem obrotu wokół osi y, a  $\alpha$  kątem obrotu wokół osi x:

$$\begin{aligned} x' &= x * (\cos \gamma * \cos \beta) + \\ &\quad y * (\cos \gamma * \sin \beta * \sin \alpha - \sin \gamma * \cos \alpha) + \\ &\quad z * (\cos \gamma * \sin \beta * \cos \alpha + \sin \gamma * \sin \alpha) \\ y' &= x * (\sin \gamma * \cos \beta) + \\ &\quad y * (\sin \gamma * \sin \beta * \sin \alpha + \cos \gamma * \cos \alpha) + \\ &\quad z * (\sin \gamma * \sin \beta * \cos \alpha - \cos \gamma * \sin \alpha) \\ z' &= -x * \sin \beta + \\ &\quad y * \cos \beta * \sin \alpha + \\ &\quad z * \cos \beta * \cos \alpha \end{aligned}$$

Równania te można zapisać w postaci macierzy reprezentującej złożenie obrotów wokół trzech osi układu współrzędnych:

$$M_O = \begin{vmatrix} x' \\ y' \\ z' \\ 1 \end{vmatrix}$$

Mnożąc tę macierz przez jednorodne współrzędne punktu można wykonać dowolny obrót punktu za pomocą pojedynczej operacji, zamiast składać kilka różnych obrotów.

## Skalowanie

Skalowanie polega na mnożeniu współrzędnych punktów przez określoną wartość współczynnika. W przypadku grafiki trójwymiarowej można skalować każdą ze współrzędnych przez inny współczynnik  $s_x$ ,  $s_y$  i  $s_z$ . Macierz skalowania wygląda następująco:

$$MS = \begin{vmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Poniżej przedstawiamy przykład skalowania punktu  $p$ :

$$p = \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix}$$

$$MS = \begin{vmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} * \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix} = \begin{vmatrix} (s_x*x) + (0*y) + (0*z) + (0*1) \\ (0*x) + (s_y*y) + (0*z) + (0*1) \\ (0*x) + (0*y) + (s_z*z) + (0*1) \\ (0*x) + (0*y) + (0*z) + (1*1) \end{vmatrix} = \begin{vmatrix} x*s_x \\ y*s_y \\ z*s_z \\ 1 \end{vmatrix}$$

Z macierzy wynikowej można odczytać równania opisujące współrzędne punktu po skalowaniu:

$$\begin{aligned} x' &= x*s_x \\ y' &= y*s_y \\ z' &= z*s_z \end{aligned}$$

Na tym można zakończyć omawianie macierzy i ich zastosowań. Tworząc grafikę OpenGL korzysta się z nich nieustannie. Jeśli czytelnik nie czuje się jeszcze pewnie w świecie macierzy i działań na nich, to dla uzupełnienia swojej wiedzy może skorzystać z jednego ze źródeł, które zostało podane w dodatku A.

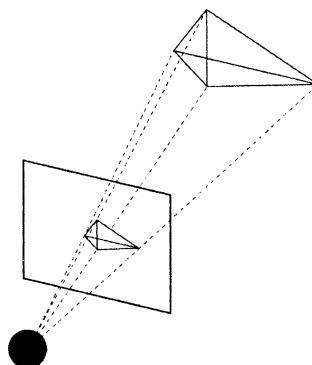
## Rzutowanie

Mimo że książka ta zajmuje się wyłącznie tworzeniem grafiki trójwymiarowej, to jednak powstaje ona na płaskim ekranie monitora. I właśnie stąd wynika koncepcja *rzutowania* przestrzennego świata gry na płaszczyznę ekranu.

Na potrzeby grafiki trójwymiarowej wyróżnia się dwa rodzaje rzutowania. *Rzutowanie izometryczne* zachowuje wymiary obiektów na płaszczyźnie rzutowania bez względu na odległość, w jakiej znajdują się w przestrzeni. *Płaszczyzna rzutowania* spełnia rolę soczewek obiektywu kamery, przez którą obserwuje się trójwymiarowy świat. Wszystkie punkty przestrzeni muszą więc być rzutowane na płaszczyznę rzutowania.

Drugi z rodzajów rzutowania — zwany *rzutowaniem perspektywicznym* — uwzględnia odległość obiektu od płaszczyzny rzutowania. Rzutowanie to można wyobrazić sobie za pomocą promieni wybiegających z punktów obiektu i zbiegających się w jednym punkcie reprezentującym oko obserwatora. Promienie te po drodze przecinają płaszczyznę rzutowania. Ilustruje to rysunek 3.5.

**Rysunek 3.5.**  
*Promienie biegące od obiektu do oka obserwatora*



Teraz należy przyjrzeć się bliżej obu rodzajom rzutowania.

## Rzutowanie izometryczne

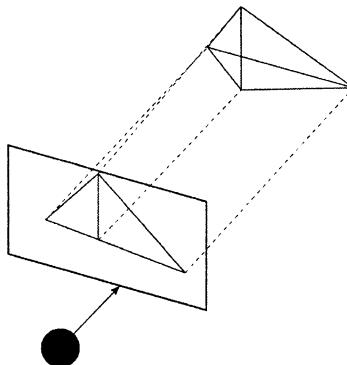
Rzutowanie izometryczne jest często stosowane przez inżynierów korzystających z komputerowych systemów wspomagania projektowania. Tworzą oni rzuty trójwymiarowego modelu pod różnymi kątami, które są zwane także *rzutami ortograficznymi*. Najczęściej wykorzystywane są rzuty ortograficzne modelu z przodu, z góry i z boku, ponieważ przekazują odbiorcy pełną informację umożliwiającą wizualizację modelowanego obiektu.

Rzutowanie izometryczne odbywa się w dwu etapach. Pierwszy etap polega na przekształceniu płaszczyzny rzutowania na płaszczyznę *xy*. Drugi etap polega na usunięciu składowej ze wszystkich punktów w przestrzeni. W ten sposób usunięta zostaje informacja o odległości punktów od płaszczyzny rzutowania, co pozwala zachować wymiary obiektów w rzucie bez względu na ich wcześniejsze położenie w przestrzeni w stosunku do obserwatora.

Chociaż tak przydatne przy modelowaniu obiektów przez inżynierów, rzutowanie izometryczne nie jest przydatne przy tworzeniu grafiki trójwymiarowej. Obrazy otrzymane na drodze rzutowania izometrycznego nie są realistyczne, ponieważ nie zawierają informacji o głębi (patrz: rysunek 3.6). Z tego powodu tworząc grafikę trójwymiarową należy korzystać głównie z rzutowania perspektywicznego.

**Rysunek 3.6.**

*Rzutowanie izometryczne*



## Rzutowanie perspektywiczne

Jak już wspomniano, rzutowanie perspektywiczne pozwala utworzyć obraz obiektów, na którym rozmiary obiektów zależą od ich odległości od obserwatora. Obiekty bardziej oddalone będą posiadać na tym obrazie mniejsze rozmiary niż obiekty położone bliżej obserwatora.

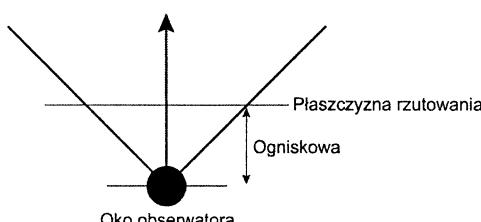
W przypadku rzutowania perspektywicznego traktuje się oko obserwatora jako pojedynczy punkt, w którym zbiegają się promienie światła odbite od obiektów. Założenie takie stanowi bardzo dobre przybliżenie rzeczywistego sposobu widzenia. Na drodze do oka obserwatora promienie światła przecinają płaszczyznę rzutowania. Zbiór punktów przecięcia tworzy płaski obraz przestrzeni na płaszczyźnie rzutowania. Pokazane to zostało na rysunku 3.5.

Rzutowanie perspektywiczne polega więc na znalezieniu na płaszczyźnie rzutowania punktów jej przecięcia przez promienie świetlne odbite od obiektów. W tym celu można skorzystać z macierzy przekształcenia perspektywicznego. Zanim zostanie ona przeanalizowana, należy omówić jeszcze pojęcie ogniskowej.

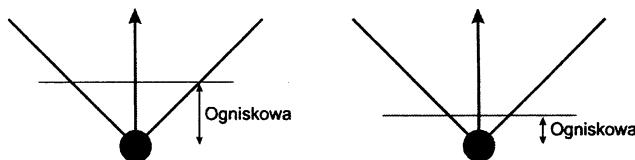
Przez ogniskową należy rozumieć w tym przypadku odległość pomiędzy okiem obserwatora a płaszczyzną rzutowania, co ilustruje rysunek 3.7. Pojęcie ogniskowej jest szczególnie przydatne podczas tworzenia grafiki trójwymiarowej i wykorzystywane jest do określenia pola widzenia, co zaprezentowano na rysunku 3.8. Zwiększenie ogniskowej powoduje ograniczenie pola widzenia. Natomiast zmniejszenie ogniskowej sprawia, że pole widzenia powiększa się. Tworząc grafikę trójwymiarową często trzeba trochę eksperymentować z wielkością ogniskowej, aby uzyskać bardziej realistyczny obraz. Powodem tego są zniekształcenia, które mogą wystąpić na skutek zastosowania przekształcenia perspektywicznego.

**Rysunek 3.7.**

*Ogniskowa jako odległość pomiędzy okiem obserwatora i płaszczyzną rzutowania*



**Rysunek 3.8.**  
Pole widzenia  
w zależności  
od ogniskowej



Teraz już można przejść do omówienia macierzy przekształcenia perspektywicznego. Aby zastosowanie tego przekształcenia było możliwe, trzeba zawsze stosować *współrzędne jednorodne*. Oznacza to, że w wektorze opisującym współrzędne punktu po składowych  $x$ ,  $y$ ,  $z$  musi zawsze występować wartość 1. Również wektor reprezentujący współrzędne punktu po przekształceniu musi zawierać na tej samej pozycji wartość 1. Jeśli raz jeszcze zostaną poddane analizie omówione dotąd przekształcenia, okaże się, że we wszystkich wykorzystywane były właśnie współrzędne jednorodne. Jeśli w wyniku przekształcenia wektor reprezentujący współrzędne punktu będzie zawierał na ostatniej pozycji wartośćną od 1, to trzeba będzie go ponownie *znormalizować*. W tym celu należy pomnożyć wektor przez wartość będącą odwrotnością tej wartości. Macierz przekształcenia perspektywicznego prezentuje się następująco:

$$M_P = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/\text{ogniskowa} \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Mnożąc tę macierz przez wektor zawierający współrzędne jednorodne punktu wykonuje się przekształcenie perspektywiczne:

$$\begin{aligned} p &= \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix} \\ M_P * p &= \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/\text{ogniskowa} \\ 0 & 0 & 0 & 1 \end{vmatrix} * \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix} = \begin{vmatrix} x \\ y \\ -1 \\ z/\text{ogniskowa} \end{vmatrix} \end{aligned}$$

Jak łatwo zauważać, ostatni element wynikowego wektora będzie zwykle różny od 1. Oznacza to, że wektor ten należy pomnożyć przez wartość będącą odwrotnością jego ostatniego elementu, czyli —  $\text{ogniskowa}/z$ .

$$\text{ogniskowa}/z * \begin{vmatrix} x \\ y \\ -1 \\ z/\text{ogniskowa} \end{vmatrix} = \begin{vmatrix} x*(\text{ogniskowa}/z) \\ y*(\text{ogniskowa}/z) \\ -\text{ogniskowa}/z \\ 1 \end{vmatrix}$$

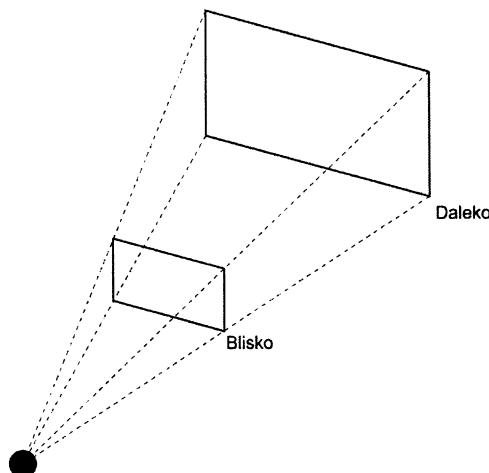
Warto przy okazji zauważać, że jeśli wartość współrzędnej  $z$  będzie równa 0, to efektem będzie błąd dzielenia przez 0. Aby zapobiec takim sytuacjom, przed wykonaniem przekształcenia perspektywicznego stosuje się obcinanie punktów, które mogłyby spowodować wystąpienie takiego błędu.

## Obcinanie

Jeśli obiekty trójwymiarowego świata nie mieszczą się na płaszczyźnie rzutowania, to zastosowanie przekształcenia perspektywicznego może prowadzić do błędów. Zwłaszcza jeśli współrzędna  $z$  w przypadku niektórych punktów tych obiektów posiada wartość 0. Gdy współrzędna  $z$  posiada wartość ujemną, to wykonywanie dla tych punktów przekształcenia perspektywicznego także nie ma sensu, ponieważ znajdują się one za obserwatorem. Ponieważ obie sytuacje mogą być przyczyną błędów, duże znaczenie ma mechanizm zapobiegania im.

Typowe rozwiązanie opisanych problemów polega na utworzeniu bryły widoku. *Bryła widoku* jest wycinkiem przestrzeni, który widziany jest przez obserwatora. Ponieważ punkty, które leżą na zewnątrz bryły widoku i tak nie są widziane przez obserwatora, nie ma sensu wykonywać dla nich przekształcenia perspektywicznego. Po zastosowaniu rzutowania perspektywicznego bryła widoku posiada kształt ostrosłupa, co przedstawia rysunek 3.9.

**Rysunek 3.9.**  
*Bryła widoku  
dla rzutowania  
perspektywicznego  
ma kształt ostrosłupa*



## Światło

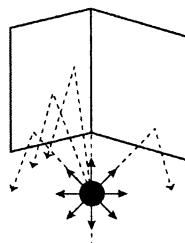
Znaczenie światła w trójwymiarowym świecie zostanie zrozumiane najlepiej, jeśli skojarzone będzie z wrażeniami powstającymi po wejściu do pokoju, w którym nie ma okien i wyłączeniu światła. Nic nie widać! Podobnie obiekty wirtualnego świata gry przestaną istnieć na ekranie bez oświetlenia. Ale nie tylko z tego powodu światło jest tak ważne. Właściwe oświetlenie podnosi realizm wirtualnego świata tworząc cienie, odbicia, kolory i inne efekty. Efektom tym zostanie poświęcona uwaga nieco później, a teraz przestawione zostaną jedynie trzy rodzaje światła: światło otoczenia, światło rozproszone i światło odbite.

## Światło otoczenia

Ten rodzaj światła wypełnia przestrzeń tak, jakby światło nie posiadało żadnego źródła i padało z równą intensywnością ze wszystkich kierunków. Oczywiście źródło światła otoczenia istnieje, ale oświetla ono obiekty ze wszystkich stron równomiernie. Gdy w po-koju zostanie umieszczona nieskończona liczba żarówek, to w jego centralnym punkcie zlokalizowane zostanie idealne światło otoczenia. Rysunek 3.10 ilustruje pojęcie światła otoczenia.

**Rysunek 3.10.**

Oświetlenie powierzchni za pomocą światła otoczenia



Źródło światła otoczenia

Ponieważ światło otoczenia oświetla równomiernie wszystkie obiekty trójwymiarowego świata, wyznaczenie jego natężenia w dowolnym punkcie nie stanowi problemu. Nie trzeba wnikać w matematyczne szczegóły wyznaczania światła i jego odbicia, ale warto już teraz wiedzieć, że wyznaczenie składowej czerwonej, zielonej i niebieskiej światła otoczenia w każdym punkcie wirtualnego świata możliwe jest za pomocą odpowiednich równań.

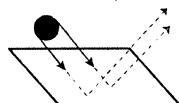
## Światło rozproszone

Ten rodzaj światła spotyka się na co dzień. Pada ono z określonego kierunku i jest równomiernie odbijane przez oświetlaną płaszczyznę. Dwie płaszczyzny mogą jednak od-biać światło rozproszone w różnym stopniu, jeśli kąt pomiędzy nimi a kierunkiem światła jest różny. Oczywiście w największym stopniu zostanie odbite światło, które oświetla płaszczyznę pod kątem prostym. Światło słoneczne stanowi dobry przykład światła rozproszonego, gdyż pada pod określonym kątem i równomiernie oświetla płaszczyznę. Je-śli obserwator światła słonecznego znajdzie się poza Ziemią, to zauważa, że promienie słoneczne odbijane są w różnym stopniu przez różne obszary naszej planety. W naj-większym stopniu odbijać je będą obszary, w przypadku których Słońce znajduje się najwyżej nad horyzontem, a w najmniejszym te, nad którymi pojawią się dopiero nad horyzontem, lub zachodzi za niego. Oczywiście różny stopień odbicia promieni sło-necznych przez różne obszary Ziemi nie kłoci się z równomiernym odbiciem światła rozproszonego przez płaszczyznę, ponieważ powierzchnię Ziemi można podzielić na wiele małych płaszczyzn. Rysunek 3.11 stanowi ilustrację światła rozproszonego.

**Rysunek 3.11.**

Oświetlenie płaszczyzny przez światło rozproszone

Źródło światła rozproszonego



## Światło odbijane

Światło odbijane pada także z określonego kierunku i jest intensywnie odbijane przez płaszczyznę jedynie w określonym kierunku. Często tworzy ono na oświetlanej powierzchni jasny punkt odbicia. Światło odbijane używane jest często do tworzenia wrażenia połyku na oświetlonych obiektach. Pojęcie światła odbijanego przybliża rysunek 3.12.

**Rysunek 3.12.**

Oświetlenie  
płaszczyzny  
przez światło  
odbijane

Źródło światła odbijanego



## Odwzorowania tekstur

Odwzorowania tekstur pozwalają przekształcić wielokąty tworzące grafikę trójwymiarową w realistyczne obrazy wirtualnego świata. Obecnie prawie wszystkie gry wykorzystują tekstury.

Tekstury mogą mieć postać prostych, geometrycznych wzorów lub mogą być stworzone przez artystę-grafika albo na drodze fotograficznej.

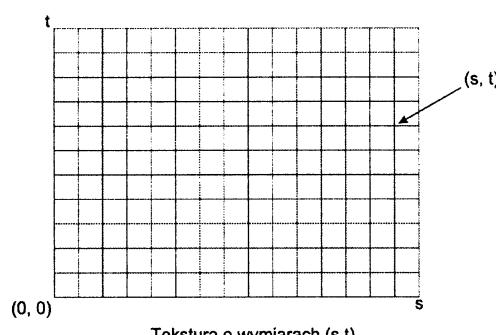
W rzeczywistym świecie rozpoznaje się obiekty na podstawie ich rozmiarów, kształtu oraz właśnie tekstur. W ten sam sposób można charakteryzować także obiekty wirtualnego świata gry — za pomocą rozmiaru, kształtu i pokrywającej je tekstury.

Przykłady analizowane w tej książce koncentrują się na wykorzystaniu tekstur dwuwymiarowych, chociaż możliwe są także tekstury jedno-, trój-, a nawet czterowymiarowe. Tekstura dwuwymiarowa ładowana jest zwykle z pliku zawierającego grafikę stworzoną przez artystę lub zeskanowaną ze zdjęcia. Coraz powszechniejsze stają się obecnie tak zwane *tekstury proceduralne*, które tworzone są w trakcie wykonywania programu.

Wyjaśnionych teraz zostanie kilka terminów związanych z teksturami. Jeśli dla tekstury dwuwymiarowej wprowadzony zostanie układ współrzędnych, to uzyska się możliwość manipulacji każdym punktem tekstuury. *Układ współrzędnych tekstuury* pokazano na rysunku 3.13. Opisuje on położenie każdego punktu za pomocą pary  $(s, t)$ . Tekstura ładowana jest z pliku do dwuwymiarowej tablicy, której każdy element nazywa się *tekselem*.

**Rysunek 3.13.**

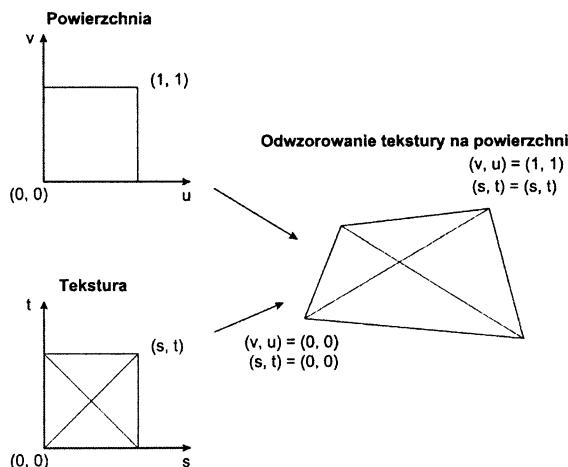
Współrzędne tekstuury



Odwzorowując teksturę na powierzchni obiektu trzeba ją najpierw odpowiednio zorientować. W tym celu korzysta się ze *współrzędnych parametrycznych* ( $u, v$ ) pokazanych na rysunku 3.14. Wartości  $u$  i  $v$  należą do przedziału od 0 do 1. Wartości te określają sposób pokrycia powierzchni za pomocą tekstuury. Sposób wykorzystania współrzędnych parametrycznych omówiony zostanie bliżej w rozdziale 8. poświęconym odwzorowaniom tekstuur.

**Rysunek 3.14.**

Współrzędne parametryczne pozwalają ustalić sposób pokrycia powierzchni przez teksturę



## Podsumowanie

Do reprezentacji obiektów trójwymiarowego świata używa się punktów, skalarów i wektorów. Skalary opisują takie wielkości jak na przykład temperatura, odległość, szerość i nie posiadają określonego kierunku ani zwrotu. Wektory opisywane są natomiast przez długość, kierunek i zwrot. Punkty reprezentują poszczególne lokalizacje trójwymiarowej przestrzeni.

Macierze wykorzystywane są do reprezentacji wektorów i operacji wykonywanych na wektorach. Do operacji tych — zwanych *przekształceniemi* — należą: przesunięcie, obrót i skalowanie. Przesunięcia te pozwalają przemieszczać obiekty w przestrzeni, obracać je wokół osi układu współrzędnych i zmieniać rozmiary obiektów za pomocą skalowania.

Rzutowanie izometryczne wykorzystywane jest przez komputerowe wspomaganie projektowania i zachowuje rozmiary rzutowanych obiektów bez względu na ich odległość od obserwatora. Rzutowanie perspektywiczne zniekształca rozmiary obiektów w taki sposób, że rozmiary obiektów znajdujących się dalej od obserwatora stają się mniejsze od rozmiarów obiektów leżących bliżej.

Bryłę widoku stanowi wycinek przestrzeni widziany przez obserwatora. Wszystkie obiekty znajdujące się poza bryłą widoku są obcinane i nie są poddawane przekształceniom. Skraca to czas tworzenia grafiki oraz pozwala uniknąć błędów związanych z wykonywaniem działania dzielenia podczas przekształceń.

Tworząc grafikę należy stosować trzy podstawowe rodzaje światła. Światło otoczenia rozchodzi się równomiernie we wszystkich kierunkach przestrzeni. Światło rozproszone pada z określonego kierunku i jest równomiernie odbijane przez płaszczyznę. Tworzenie efektów odbicia umożliwia światło odbijane jedynie w określonym kierunku.

Odwzorowania tekstur pozwalają tworzyć bardziej realistyczny, trójwymiarowy świat poprzez umieszczanie wzorów bądź obrazów na tworzących obiekty wielokątach. Podobnie jak w świecie rzeczywistym tekstury ułatwiają identyfikację obiektów.

# **Część II**

# **Korzystanie z OpenGL**



## Rozdział 4.

# **Maszyna stanów OpenGL i podstawowe elementy grafiki**

Począwszy od bieżącego rozdziału zacznie się nareszcie praktyczne korzystanie z OpenGL. Zanim jednak zostanie stworzona bardziej zaawansowana grafika, trzeba będzie najpierw poznać zasady działania maszyny stanów OpenGL.

Stan maszyny OpenGL opisany jest przez setki parametrów, które określają różne aspekty tworzenia grafiki. Ponieważ stan maszyny stanów OpenGL ma zasadniczy wpływ na powstanie tworzonyj grafiki, trzeba poznać najpierw jej domyślne wartości parametrów oraz sposób uzyskiwania i zmieniania wartości tych parametrów. Stan maszyny OpenGL może być zmieniany za pomocą szeregu funkcji, które zostaną przeanalizowane w tym rozdziale.

W bieżącym rozdziale omówione będą następujące zagadnienia:

- ◆ sposób dostępu do parametrów maszyny stanów OpenGL i metody zmiany ich wartości;
- ◆ elementy podstawowe grafiki OpenGL i ich rodzaje;
- ◆ metody zmiany sposobu obsługi i wyświetlania podstawowych elementów grafiki.

## **Funkcje stanu**

Interfejs OpenGL zaprojektowano w taki sposób, że zawiera on wiele funkcji ogólnego przeznaczenia, które mogą zmieniać wiele różnych parametrów w zależności od przekazywanych im argumentów. Rozwiążanie to zastosowano także w przypadku funkcji zmieniających parametry maszyny stanu, które przedstawione zostaną w tym podrozdziale.

Pierwszą z nich jest funkcja `glGet()`, którą stosuje się w celu uzyskania bieżących parametrów maszyny stanów. Funkcja ta posiada cztery różne wersje:

```
void glGetBooleanv(GLenum pname, GLboolean *params);
void glGetDoublev(GLenum pname, GLdouble *params);
void glGetFloatv(GLenum pname, GLfloat *params);
void getIntegerv(GLenum pname, GLint *params);
```

Parametr params jest wskaźnikiem tablicy wystarczająco pojemnej, by funkcja glGet() mogła umieścić w niej wartość parametru, którego dotyczy pytanie. Parametr pname określa natomiast parametr, którego wartość ma być poznana. Parametr pname może mieć jedną z podanych poniżej wartości. Znaczenie poszczególnych z nich omawiane będzie stopniowo, gdy pojawiać się one będą w tworzonych programach.

```
GL_ACCUM_ALPHA_BITS
GL_ACCUM_BLUE_BITS
GL_ACCUM_CLEAR_VALUE
GL_ACCUM_GREEN_BITS
GL_ACCUM_RED_BITS
GL_ALPHA_BIAS
GL_ALPHA_BITS
GL_ALPHA_SCALE
GL_ALPHA_TEST
GL_ALPHA_TEST_FUNC
GL_ALPHA_TEST_REF
GL_ATTRIB_STACK_DEPTH
GL_AUTO_NORMAL
GL_AUX_BUFFERS
GL_BLEND
GL_BLEND_DST
GL_BLEND_SRC
GL_BLUE_BIAS
GL_BLUE_BITS
GL_BLUE_SCALE
GL_CLIENT_ATTRIB_STACK_DEPTH
GL_CLIP_PLANEn (gdzie n jest wartością z przedziału od 0 do GL_MAX_CLIP_PANES - 1)
GL_COLOR_ARRAY
GL_COLOR_ARRAY_SIZE
GL_COLOR_ARRAY_STRIDE
GL_COLOR_ARRAY_TYPE
GL_COLOR_CLEAR_VALUE
GL_COLOR_LOGIC_OP
GL_COLOR_MATERIAL
GL_COLOR_MATERIAL_FACE
GL_COLOR_MATERIAL_PARAMETER
GL_COLOR_WRITEMASK
GL_CULL_FACE
GL_CULL_FACE_MODE
GL_CURRENT_COLOR
GL_CURRENT_INDEX
GL_CURRENT_NORMAL
GL_CURRENT_RASTER_COLOR
GL_CURRENT_RASTER_DISTANCE
GL_CURRENT_RASTER_INDEX
GL_CURRENT_RASTER_POSITION
GL_CURRENT_RASTER_POSITION_VALID
GL_CURRENT_RASTER_TEXTURE_COORDS
GL_CURRENT_TEXTURE_COORDS
GL_DEPTH_BIAS
GL_DEPTH_BITS
GL_DEPTH_CLEAR_VALUE
```

```
GL_DEPTH_FUNC
GL_DEPTH_RANGE
GL_DEPTH_SCALE
GL_DEPTH_TEST
GL_DEPTH_WRITEMASK
GL_DITHER
GL_DOUBLEBUFFER
GL_DRAW_BUFFER
GL_EDGE_FLAG
GL_EDGE_FLAG_ARRAY
GL_EDGE_FLAG_ARRAY_STRIDE
GL_FOG
GL_FOG_COLOR
GL_FOG_DENSITY
GL_FOG_END
GL_FOG_HINT
GL_FOG_INDEX
GL_FOG_MODE
GL_FOG_START
GL_FRONT_FACE
GL_GREEN_BIAS
GL_GREEN_BITS
GL_GREEN_SCALE
GL_INDEX_ARRAY
GL_INDEX_ARRAY_STRIDE
GL_INDEX_ARRAY_TYPE
GL_INDEX_BITS
GL_INDEX_CLEAR_VALUE
GL_INDEX_LOGIC_OP
GL_INDEX_MODE
GL_INDEX_OFFSET
GL_INDEX_SHIFT
GL_INDEX_WRITEMASK
GL_LIGHTn (gdzie n jest wartością z przedziału od 0 do GL_MAX_LIGHTS - 1)
GL_LIGHTING
GL_LIGHT_MODEL_AMBIENT
GL_LIGHT_MODEL_LOCAL_VIEWER
GL_LIGHT_MODEL_TWO_SIDE
GL_LINE_SMOOTH
GL_LINE_SMOOTH_HINT
GL_LINE_STIPPLE
GL_LINE_STIPPLE_PATTERN
GL_LINE_STIPPLE_REPEAT
GL_LINE_WIDTH
GL_LINE_WIDTH_GRANULARITY
GL_LINE_WIDTH_RANGE
GL_LIST_BASE
GL_LIST_INDEX
GL_LIST_MODE
GL_LOGIC_OP
GL_LOGIC_OP_MODE
GL_MAP1_COLOR_4
GL_MAP1_GRID_DOMAIN
GL_MAP1_GRID_SEGMENTS
GL_MAP1_INDEX
GL_MAP1_NORMAL
GL_MAP1_TEXTURE_COORD_1
GL_MAP1_TEXTURE_COORD_2
```

```
GL_MAP1_TEXTURE_COORD_3
GL_MAP1_TEXTURE_COORD_4
GL_MAP1_VERTEX_3
GL_MAP1_VERTEX_4
GL_MAP2_COLOR_4
GL_MAP2_GRID_DOMAIN
GL_MAP2_GRID_SEGMENTS
GL_MAP2_INDEX
GL_MAP2_NORMAL
GL_MAP2_TEXTURE_COORD_1
GL_MAP2_TEXTURE_COORD_2
GL_MAP2_TEXTURE_COORD_3
GL_MAP2_TEXTURE_COORD_4
GL_MAP2_VERTEX_3
GL_MAP2_VERTEX_4
GL_MAP_COLOR
GL_MAP_STENCIL
GL_MATRIX_MODE
GL_MAX_ATTRIB_STACK_DEPTH
GL_MAX_CLIENT_ATTRIB_STACK_DEPTH
GL_MAX_CLIP_PLANES
GL_MAX_EVAL_ORDER
GL_MAX_LIGHTS
GL_MAX_LIST_NESTING
GL_MAX_MODELVIEW_STACK_DEPTH
GL_MAX_NAME_STACK_DEPTH
GL_MAX_PIXEL_MAP_TABLE
GL_MAX_PROJECTION_STACK_DEPTH
GL_MAX_TEXTURE_SIZE
GL_MAX_TEXTURE_STACK_DEPTH
GL_MAX_VIEWPORT_DIMS
GL_MODELVIEW_MATRIX
GL_MODELVIEW_STACK_DEPTH
GL_NAME_STACK_DEPTH
GL_NORMAL_ARRAY
GL_NORMAL_ARRAY_STRIDE
GL_NORMALIZE
GL_PACK_ALIGNMENT
GL_PACK_LSB_FIRST
GL_PACK_ROW_LENGTH
GL_PACK_SKIP_PIXELS
GL_PACK_SKIP_ROWS
GL_PACK_SWAP_BYTES
GL_PERSPECTIVE_CORRECTION_HINT
GL_PIXEL_MAP_A_TO_A_SIZE
GL_PIXEL_MAP_B_TO_B_SIZE
GL_PIXEL_MAP_G_TO_G_SIZE
GL_PIXEL_MAP_I_TO_A_SIZE
GL_PIXEL_MAP_I_TO_B_SIZE
GL_PIXEL_MAP_I_TO_G_SIZE
GL_PIXEL_MAP_I_TO_I_SIZE
GL_PIXEL_MAP_I_TO_R_SIZE
GL_PIXEL_MAP_R_TO_R_SIZE
GL_PIXEL_MAP_S_TO_S_SIZE
GL_POINT_SIZE
GL_POINT_SIZE_GRANULARITY
GL_POINT_SIZE_RANGE
GL_POINT_SMOOTH
GL_POINT_SMOOTH_HINT
```

```
GL_POLYGON_MODE
GL_POLYGON_OFFSET_FACTOR
GL_POLYGON_OFFSET_UNITS
GL_POLYGON_OFFSET_FILL
GL_POLYGON_OFFSET_LINE
GL_POLYGON_OFFSET_POINT
GL_POLYGON_SMOOTH
GL_POLYGON_SMOOTH_HINT
GL_POLYGON_STIPPLE
GL_PROJECTION_MATRIX
GL_PROJECTION_STACK_DEPTH
GL_READ_BUFFER
GL_RED_BIAS
GL_RED_BITS
GL_RED_SCALE
GL_RENDER_MODE
GL_RGBA_MODE
GL_SCISSOR_BOX
GL_SCISSOR_TEST
GL_SHADE_MODEL
GL_STENCIL_BITS
GL_STENCIL_CLEAR_VALUE
GL_STENCIL_FAIL
GL_STENCIL_FUNC
GL_STENCIL_PASS_DEPTH_FAIL
GL_STENCIL_PASS_DEPTH_PASS
GL_STENCIL_REF
GL_STENCIL_TEST
GL_STENCIL_VALUE_MASK
GL_STENCIL_WRITEMASK
GL_STEREO
GL_SUBPIXEL_BITS
GL_TEXTURE_ID
GL_TEXTURE_2D
GL_TEXTURE_COORD_ARRAY
GL_TEXTURE_COORD_ARRAY_SIZE
GL_TEXTURE_COORD_ARRAY_STRIDE
GL_TEXTURE_COORD_ARRAY_TYPE
GL_TEXTURE_ENV_COLOR
GL_TEXTURE_ENV_MODE
GL_TEXTURE_GEN_Q
GL_TEXTURE_GEN_R
GL_TEXTURE_GEN_S
GL_TEXTURE_GEN_T
GL_TEXTURE_MATRIX
GL_TEXTURE_STACK_DEPTH
GL_UNPACK_ALIGNMENT
GL_UNPACK_LSB_FIRST
GL_UNPACK_ROW_LENGTH
GL_UNPACK_SKIP_PIXELS
GL_UNPACK_SKIP_ROWS
GL_UNPACK_SWAP_BYTES
GL_VERTEX_ARRAY
GL_VERTEX_ARRAY_SIZE
GL_VERTEX_ARRAY_STRIDE
GL_VERTEX_ARRAY_TYPE
GL_VIEWPORT
GL_ZOOM_X
GL_ZOOM_Y
```

Możliwość uzyskania informacji o stanie maszyny OpenGL jest z pewnością bardzo przydatna, jednak nie tak eksytyująca jak możliwość zmiany jej parametrów. OpenGL nie udostępnia w tym celu jednej uniwersalnej funkcji `glSet()`, lecz wiele bardziej wyspecjalizowanych funkcji, które będą przedstawiane stopniowo.

Często przy sprawdzeniu bieżących parametrów maszyny OpenGL trzeba jedynie uzyskać informację, czy dana jej właściwość została aktywowana. Można zrobić to za pomocą funkcji `glGet()`, ale wygodniej będzie zastosować funkcję `glIsEnabled()` o następującym prototypie:

```
GLboolean glIsEnabled(GLenum cap);
```

Parametr funkcji `glIsEnabled()` może mieć jedną z podanych poniżej wartości. Funkcja `glIsEnabled()` zwraca wartość `GL_TRUE`, gdy dana właściwość maszyny jest aktywna, a wartość `GL_FALSE`, gdy jest odwrotnie. Podobnie jak w przypadku funkcji `glGet()`, także i tym razem poszczególne wartości omawiane będą podczas pierwszego ich wykorzystania.

```
GL_ALPHA_TEST  
GL_AUTO_NORMAL  
GL_BLEND  
GL_CLIP_PLANEn (gdzie n jest wartością z przedziału od 0 do GL_MAX_CLIP_PANES - 1)  
GL_COLOR_ARRAY  
GL_COLOR_LOGIC_OP  
GL_COLOR_MATERIAL  
GL_CULL_FACE  
GL_DEPTH_TEST  
GL_DITHER  
GL_FOG  
GL_INDEX_ARRAY  
GL_INDEX_LOGIC_OP  
GL_LIGHTn (gdzie n jest wartością z przedziału od 0 do GL_MAX_LIGHTS - 1)  
GL_LIGHTING  
GL_LINE_SMOOTH  
GL_LINE_STIPPLE  
GL_LOGIC_OP  
GL_MAP1_COLOR_4  
GL_MAP1_INDEX  
GL_MAP1_NORMAL  
GL_MAP1_TEXTURE_COORD_1  
GL_MAP1_TEXTURE_COORD_2  
GL_MAP1_TEXTURE_COORD_3  
GL_MAP1_TEXTURE_COORD_4  
GL_MAP1_VERTEX_3  
GL_MAP1_VERTEX_4  
GL_MAP2_COLOR_4  
GL_MAP2_INDEX  
GL_MAP2_NORMAL  
GL_MAP2_TEXTURE_COORD_1  
GL_MAP2_TEXTURE_COORD_2  
GL_MAP2_TEXTURE_COORD_3  
GL_MAP2_TEXTURE_COORD_4  
GL_MAP2_VERTEX_3  
GL_MAP2_VERTEX_4  
GL_NORMAL_ARRAY  
GL_NORMALIZE  
GL_POINT_SMOOTH  
GL_POLYGON_OFFSET_FILL
```

```
GL_POLYGON_OFFSET_LINE  
GL_POLYGON_OFFSET_POINT  
GL_POLYGON_SMOOTH  
GL_POLYGON_STIPPLE  
GL_SCISSOR_TEST  
GL_STENCIL_TEST  
GL_TEXTURE_1D  
GL_TEXTURE_2D  
GL_TEXTURE_COORD_ARRAY  
GL_TEXTURE_GEN_Q  
GL_TEXTURE_GEN_R  
GL_TEXTURE_GEN_S  
GL_TEXTURE_GEN_T  
GL_VERTEX_ARRAY
```

Najczęściej będą wykorzystywane właśnie funkcje `glGet()` i `glIsEnabled()`, ale istnieje także wiele innych funkcji stanu (właściwie większość funkcji OpenGL ma pewien wpływ na stan maszyny OpenGL). W rozdziale tym przedstawionych zostanie kilka z nich, a większość pozostała w kolejnych rozdziałach.

## Podstawowe elementy grafiki

Przez podstawowe elementy grafiki należy rozumieć punkty, odcinki, trójkąty i tym podobne. Tworzona grafika będzie składać się z tysięcy takich elementów, dlatego też trzeba zapoznać się ze sposobem ich działania. Przed omówieniem poszczególnych typów elementów podstawowych przedstawić trzeba najpierw kilka funkcji OpenGL, które będą używane bardzo często. Pierwszą z nich będzie `glBegin()` posiadająca poniższy prototyp:

```
void glBegin(GLenum mode);
```

Funkcja ta przekazuje maszynie OpenGL informację o tym, że rozpoczęło się tworzenie grafiki oraz informację o typie wykorzystywanych elementów podstawowych. Typ ten określa się za pomocą parametru `mode`, który przyjmować może wartości przedstawione w tabeli 4.1.

**Tabela 4.1.** Wartości parametru funkcji `glBegin`

Wartość	Znaczenie
GL_POINTS	Pojedyncze punkty
GL_LINES	Odcinki
GL_LINE_STRIP	Sekwencja połączonych odcinków
GL_LINE_LOOP	Zamknięta sekwencja połączonych odcinków
GL_TRIANGLES	Pojedyncze trójkąty
GL_TRIANGLE_STRIP	Sekwencja połączonych trójkątów
GL_TRIANGLE_FAN	Sekwencja trójkątów posiadających jeden wspólny wierzchołek
GL_QUADS	Czworokąty
GL_QUAD_STRIP	Sekwencja połączonych czworokątów
GL_POLYGON	Wielokąty o dowolnej liczbie wierzchołków

Pozostała część tego rozdziału poświęcona będzie szczegółowo omówieniu poszczególnych typów elementów podstawowych.

Każdemu wywołaniu funkcji `glBegin()` musi towarzyszyć wywołanie funkcji `glEnd()` o następującym prototypie:

```
void glEnd();
```

Funkcja `glEnd()` nie posiada parametrów. Powiadamia ona maszynę OpenGL, że zakończone zostało tworzenie grafiki z wykorzystaniem elementów określonych za pomocą wywołania funkcji `glBegin()`. Należy pamiętać o tym, że pary wywołań funkcji `glBegin()` i `glEnd()` nie mogą być zagnieżdżane, czyli nie wolno wywoływać tych funkcji wewnętrz bloku tworzonego przez inną parę wywołań `glBegin()` i `glEnd()`.

Wewnątrz bloku tworzonego przez parę wywołań funkcji `glBegin()` i `glEnd()` można wywoływać jedynie następujące funkcje OpenGL i ich wersje: `glVertex()`, `glColor()`, `glIndex()`, `glNormal()`, `glTexCoord()`, `glEvalCoord()`, `glEvalPoint()`, `glMaterial()`, `glEdgeFlag()`, `glCallList()` i `glCallLists()`. Funkcje te omówione zostaną szczegółowo w dalszej części rozdziału. Wywołanie dowolnej innej funkcji spowoduje błąd maszyny OpenGL.

Zanim nastąpi omówienie typów elementów podstawowych, trzeba przedstawić przy najmniej funkcję, a raczej rodzinę funkcji, `glVertex()`. Funkcje te wywoływane są wewnątrz bloku wywołań funkcji `glBegin()` i `glEnd()` w celu określenia punktu w przestrzeni, którego interpretacja zależy od wartości parametrów wywołania funkcji `glBegin()`. Istnieje wiele wersji funkcji `glVertex()`, które można opisać za pomocą poniższego schematu:

```
void glVertex[2,3,4][d,f,i,s][v](...);
```

Cyfra w nazwie funkcji określa liczbę wymiarów, następującą po niej litera typ danych (`double`, `float`, `int` lub `short`). Nazwę funkcji może kończyć litera `v`, co oznacza, że parametry funkcji zostaną przekazane za pomocą tablicy (wektora) zamiast pojedynczo. Liczba i typ tych parametrów określona jest przez nazwę funkcji. Najczęściej korzysta się z wersji `glVertex3f()`, której parametrami są trzy wartości typu `float` opisujące współrzędne `x`, `y` i `z` punktu w przestrzeni.

Teraz nastąpi omówienie typów podstawowych elementów grafiki, które będą używane najczęściej.

## Tworzenie punktów w trójwymiarowej przestrzeni

Nie ma prostszego elementu grafiki niż pojedynczy punkt. Rysowanie punktu jest proste, a zarazem stwarza duże możliwości tworzenia dowolnej grafiki:

```
glBegin(GL_POINTS)
    glVertex(0.0, 0.0, 0.0);
glEnd();
```

Pierwszy wiersz powyższego fragmentu kodu informuje maszynę OpenGL, że będą rysowane punkty. Kolejny wiersz rysuje pojedynczy punkt w środku układu współrzędnych. Ostatni z wierszy informuje, że zakończone zostało rysowanie punktów. Wcięcia

wierszy kodu wewnątrz bloku wywołań funkcji `glBegin()` i `glEnd()` stanowią jedyne konwencję zapisu kodu OpenGL i nie są obowiązkowe, choć zdecydowanie poprawiają czytelność kodu.

A co w przypadku, gdy ma zostać narysowany kolejny punkt, na przykład o współrzędnych  $(0.0, 1.0, 0.0)$ ? Można oczywiście zaprogramować to jak poniżej:

```
glBegin(GL_POINTS)
    glVertex(0.0, 0.0, 0.0);
glEnd();
glBegin(GL_POINTS)
    glVertex(0.0, 1.0, 0.0);
glEnd();
```

Jednak taki sposób rysowania punktów jest wyjątkowo nieefektywny. Wewnątrz pojętynego bloku wywołań funkcji `glBegin()` i `glEnd()` można rysować wiele punktów i wobec tego kod będzie miał następującą postać:

```
glBegin(GL_POINTS)
    glVertex(0.0, 0.0, 0.0);
    glVertex(0.0, 1.0, 0.0);
glEnd();
```

Krócej, szybciej i lepiej. Wewnątrz bloku wywołań funkcji `glBegin()` i `glEnd()` można wywoływać funkcję `glVertex()` dowolną liczbę razy, a każde wywołanie spowoduje narysowanie pojedynczego punktu.

OpenGL jest zaopatrzony w funkcje umożliwiające modyfikację sposobu rysowania podstawowych elementów grafiki. W przypadku punktów można zmieniać ich wielkość oraz określać, czy podczas rysowania powinien być stosowany antialiasing.

## Zmiana rozmiaru punktów

Aby zmienić rozmiar rysowanych punktów, należy skorzystać z następującej funkcji:

```
void glPointSize(GLfloat size);
```

Domyślny rozmiar punktów posiada wartość  $1.0$ . Jeśli stosowanie antialiasingu jest wyłączone (a tak jest domyślnie), to rozmiar punktu jest zaokrąglany do najbliższej wartości całkowitej (wartości z przedziału od  $0.0$  do  $1.0$  są zawsze zaokrąglane do  $1.0$ ). Bieżącą wielkość rysowanych punktów można pobrać za pomocą funkcji `glGet()`, której przekazana zostanie wartość `GL_POINT_SIZE`.

## Antialiasing

Mimo że parametry elementów grafiki można określać z ogromną dokładnością korzystając z wartości zmiennoprzecinkowych, to jednak na ekranie pozostaje do dyspozycji jedynie skończony zbiór pikseli. W efekcie elementy tworzonej grafiki posiadają „schodkowane” krawędzie. Antialiasing umożliwia ich wygładzenie. Włącza się go przekazując wartość `GL_POINT_SMOOTH` funkcji  `glEnable()` (a wyłącza przekazując tę samą wartość funkcji `glDisable()`). O tym, czy antialiasing jest włączony w danym momencie, można przekonać się przekazując wartość `GL_POINT_SMOOTH` funkcji `glGet()` lub wywołując `glIsEnabled(GL_POINT_SMOOTH)`.

Jeśli włączy się antialiasing, to przestają być dostępne dowolne wielkości punktów. Specyfikacja OpenGL wymaga obecnie od implementacji, by antialiasing był dostępny dla punktów o wielkości 1.0. Jeśli zażąda się wielkości punktu, która nie jest dostępna w danej implementacji przy włączonym antialiasingu, to wielkość ta zostanie zaokrąglona do najbliższej wielkości, dla której możliwy jest antialiasing. Aby uzyskać informację o przedziale wielkości punktów dostępnych przy włączonym antialiasingu, należy wywołać funkcję `glGet()` przekazując jej wartość `GL_POINT_SIZE_RANGE`. Natomiast wartość `GL_POINT_SIZE_GRANULARITY` pozwoli poznać różnicę kolejnych wielkości z tego przedziału. Ilustruje to poniższy fragment kodu:

```
GLfloat sizes[2];
GLfloat granularity;

glGetFloatv(GL_POINT_SIZE_RANGE, sizes)
GLfloat minPointSize = sizes[0];
GLfloat maxPointSize = sizes[1];

glGetFloatv(GL_POINT_SIZE_GRANULARITY, &granularity);
```

Włączony antialiasing spowoduje, że współrzędne rysowanego punktu zostaną postraktowane jako środek koła, którego średnica będzie równa wielkości punktu. Sposób wypełnienia pikseli leżących na granicy koła zależeć będzie od stopnia ich pokrycia przez koło.

Wiadomo już prawie wszystko o punktach, można przejść zatem do omówienia bardziej interesujących elementów grafiki.

## Odcinki

Rysowanie odcinka nie różni się od rysowania pary punktów, poznanego już w poprzednim podrozdziale:

```
glBegin(GL_LINES)
    glVertex(-2.0, -1.0, 0.0);
    glVertex(3.0, 1.0, 0.0);
glEnd();
```

Tym razem przekazano funkcji `glBegin()` wartość `GL_LINES`, aby poinformować maszynę OpenGL, że tworzone punkty mają być interpretowane jako końce odcinków. Utworzenie drugiego z punktów powoduje wykreślenie odcinka łączącego oba punkty.

Podobnie jak w przypadku punktów wewnątrz bloku wywołań funkcji `glBegin()` i `glEnd()` można rysować wiele odcinków. Każde kolejne dwa punkty traktowane są jako końce nowego odcinka. Jeśli w bloku tym nie stworzy się parzystej liczby punktów, to ostatni z nich zostanie zignorowany.

OpenGL pozwala zmieniać szerokość odcinków, włączać i wyłączać antialiasing oraz określać wzór linii.

## Zmiana szerokości odcinka

Domyślnie szerokość odcinka posiada wartość 1.0. Można zmienić ją za pomocą funkcji `glLineWidth()` o następującym prototypie:

```
void glLineWidth(GLfloat width);
```

Informację o bieżącej szerokości linii tworzonych odcinków uzyskuje się przekazując funkcji `glGet()` wartość `GL_LINE_WIDTH`.

## Antialiasing odcinków

Antialiasing działa w przypadku odcinków tak samo jak dla punktów. Można włączać go i wyłączać przekazując wartość `GL_LINE_SMOOTH` funkcjom  `glEnable()` i  `glDisable()`, a bieżący stan określić przekazując tę wartość funkcjom  `glGet()` lub  `glIsEnabled()`. Antialiasing odcinków jest domyślnie wyłączony.

Podobnie jak w przypadku punktów specyfikacja OpenGL wymaga obecnie od implementacji, by antialiasing był dostępny dla odcinków o szerokości linii równej 1.0. Aby uzyskać informacje o rzeczywistych parametrach antialiasingu odcinków dla danej implementacji, korzysta się z wartości `GL_LINE_WIDTH_RANGE` i `GL_LINE_WIDTH_GRANULARITY`. Ilustruje to poniższy przykład:

```
GLfloat sizes[2];
GLfloat granularity;

glGetFloatv (GL_LINE_WIDTH_RANGE, sizes)
GLfloat minLineWidth = sizes[0];
GLfloat maxLineWidth = sizes[1];

glGetFloatv(GL_LINE_WIDTH_GRANULARITY, &granularity);
```

Prawda, że wygląda on prawie identycznie jak w przypadku punktów?

## Wzór linii odcinka

OpenGL pozwala określić wzór linii rysowanych odcinków. Wzór ten opisuje, które z punktów linii są rysowane i umożliwia w ten sposób tworzenie linii przerywanych. Przed użyciem wzoru linii trzeba najpierw włączyć możliwość stosowania wzorów przekazując funkcji  `glEnable()` wartość `GL_LINE_STIPPLE`. Następnie określa się wzór linii korzystając z funkcji  `glLineStipple()` posiadającej następujący prototyp:

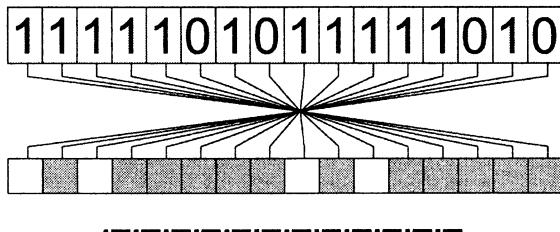
```
void glLineStipple(GLint factor, GLushort pattern);
```

Parametr `factor` posiada domyślną wartość 1 i może przyjmować wartości z przedziału od 1 do 256. Określa on, ile razy każdy bit wzorca zostanie powtórzony, zanim wykorzystany zostanie następny bit wzorca.

Parametr `pattern` stanowi wzorzec linii zapisany za pomocą 16 bitów. Bity posiadające wartość 1 określają, które piksele będą rysowane. Należy przy tym zwrócić uwagę, że bity te wykorzystywane są w odwrotnej kolejności, czyli najpierw wykorzystywane są bity mniej znaczące. Ilustruje to rysunek 4.1.

**Rysunek 4.1.**

Przykład  
interpretacji  
kolejnych bitów  
wzorca linii



Poniższy fragment kodu włącza korzystanie ze wzorców i określa wzorzec składający się na przemian z kresk i kropek:

```
glEnable(GL_LINE_STIPPLE);
Glushort stipplePattern = 0xFAFA;
glLineStipple(2, stipplePattern);
```

Bieżący wzorzec linii odcinka oraz jego wielokrotność powtórzeń można określić przekazując funkcji `glGet()` wartości `GL_LINE_STIPPLE` i `GL_LINE_STIPPLE_REPEAT`.

W kolejnym podrozdziale zostanie omówiony podstawowy element umożliwiający tworzenie wirtualnego świata gier trójwymiarowych: wielokąt.

## Wielokąty

Za pomocą punktów i odcinków można stworzyć całkiem interesującą grafikę. Jeszcze bardziej efektywne tworzenie trójwymiarowych światów umożliwiają jednak wielokąty, którym poświęcona zostanie pozostała część bieżącego rozdziału. Zanim nastąpi omówienie poszczególnych rodzajów wielokątów (to jest trójkątów, czworokątów i pozostałych wielokątów), trzeba najpierw przedstawić kilka aspektów tworzenia wielokątów.

Wielokąt tworzy się określając sekwencję jego wierzchołków w przestrzeni. Jego obszar zostaje następnie wypełniony określonym kolorem. Taki jest przynajmniej domyślny sposób tworzenia wielokątów. Oczywiście maszyna stanów OpenGL umożliwia jego modyfikację. W tym celu stosuje się następującą funkcję:

```
void glPolygonMode(GLenum face, GLenum mode);
```

OpenGL umożliwia osobne traktowanie obu stron wielokąta. Dlatego też wywołując funkcję `glPolygonMode()` trzeba określić za pomocą wartości parametru `face`, której strony dotyczą modyfikacje. Wartość `GL_FRONT` określa przednią stronę wielokąta, `GL_BACK` tylnią, a `GL_FRONT_AND_BACK` informuje, że modyfikacje dotyczyć będą obu stron wielokąta.

Parametr `mode` może natomiast przyjmować jedną z wartości przedstawionych w tabeli 4.2.

Jeśli konieczne jest na przykład to, by wypełniane były jedynie przednie powierzchnie wielokątów, a tylne posiadały jedynie krawędzie, to trzeba zastosować poniższy fragment kodu:

```
glPolygonMode(GL_FRONT, GL_FILL);
glPolygonMode(GL_BACK, GL_LINE);
```

**Tabela 4.2.** Tryby rysowania wielokątów

Wartość	Znaczenie
GL_POINT	Rysowane są jedynie wierzchołki wielokąta jako pojedyncze punkty, a sposób rysowania wierzchołków określony jest przez funkcje stanu dla rysowania punktów; tryb ten równoważny jest z wywołaniem funkcji <code>glBegin()</code> z parametrem <code>GL_POINTS</code>
GL_LINE	Rysowane są jedynie krawędzie wielokąta jako zbiór odcinków, a sposób rysowania krawędzi określony jest przez funkcje stanu dla rysowania odcinków; tryb ten równoważny jest z wywołaniem funkcji <code>glBegin()</code> z parametrem <code>GL_LOOP</code>
GL_FILL	Domyślny tryb rysowania wielokątów (wypełnia obszar wielokąta); jedynie w tym trybie ma zastosowanie wzór wielokąta i antialiasing wielokątów (patrz: kolejne podrozdziały)

Ponieważ domyślnie wielokąty są wypełniane, to pierwszy z wierszy powyższego fragmentu kodu nie jest konieczny, chyba że sposób rysowania przedniej strony wielokąta zmienił wcześniej inny fragment kodu.

Aby uzyskać informację o bieżącym trybie rysowania wielokątów, wywołuje się funkcję `glGet()` z parametrem `GL_POLYGON_MODE`.

## Ukrywanie powierzchni wielokątów

Chociaż wielokąty są nieskończenie płaskie, to jednak posiadają dwie strony, z których może oglądać je obserwator. Czasami wymagane jest, by obie strony wielokąta były prezentowane w różny sposób i właśnie dlatego wiele funkcji modyfikujących sposób wyświetlenia wielokątów wymaga określenia, której ze stron dotyczą wprowadzane zmiany. Obie strony wielokąta tworzone są zawsze oddzielnie. Czasami pojawia się sytuacja, w której wiadomo, że obserwator nie będzie mógł nigdy oglądać jednej ze stron wielokąta. Ze względu na efektywność tworzenia grafiki nie ma wtedy sensu rysowanie nie-widocznej strony wielokąta. Gdy na przykład wielokąty tworzą zamkniętą bryłę w kształcie kuli, to wnętrze jej nie będzie nigdy widoczne. Można wtedy poinformować maszynę OpenGL, że rysowanie jednej strony wielokątów nie jest konieczne. W tym celu trzeba najpierw aktywować taką możliwość przekazując wartość `GL_CULL_FACE` funkcji  `glEnable()`. Następnie należy określić, która ze stron nie powinna być rysowana. Służy do tego funkcja `glCullFace()` o następującym prototypie:

```
void glCullFace(GLenum mode);
```

Parametr `mode` może przyjmować jedną z omówionych wcześniej wartości `GL_FRONT`, `GL_BACK` lub `GL_FRONT_AND_BACK`. Wykorzystanie wartości `GL_FRONT_AND_BACK` nie ma zwykle sensu, bo w efekcie wielokąty nie są w ogóle rysowane. Domyślnie przyjmowana jest wartość `GL_BACK`.

Maszyna OpenGL musi dysponować sposobem pozwalającym określić, która ze stron wielokąta jest przednia, a która tylna. W tym celu wykorzystywany jest porządek, w jakim określone zostały wierzchołki wielokąta. Definiowanie wierzchołków wielokąta można rozpocząć od dowolnego wierzchołka, po czym podaje się kolejne wierzchołki posuwając się zgodnie z kierunkiem ruchu wskazówek zegara lub w kierunku odwrotnym. Ważne jest, aby tworząc grafikę zachować ten sam sposób definiowania wielokątów, ponieważ jedynie wtedy OpenGL będzie mógł prawidłowo rozróżnić strony wielokątów. Domyślnie przyjmuje się, że przednią stroną wielokąta jest ta strona, z której

oglądając wielokąt wierzchołki zostały zdefiniowane w kierunku przeciwnym do obrotu wskazówek zegara. Założenie to można jednak zmienić korzystając z funkcji `glFrontFace()` posiadającej poniższy prototyp:

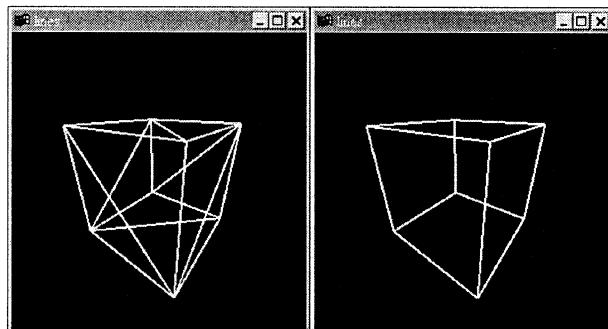
```
void glFrontFace(GLenum mode);
```

Parametr `mode` może przyjmować wartość `GL_CCW` określającą kierunek przeciwny do kierunku ruchu wskazówek zegara i wartość `GL_CW` określającą kierunek zgodny z kierunkiem ruchu wskazówek zegara.

## Ukrywanie krawędzi wielokątów

Przy tworzeniu grafiki trójwymiarowej często zachodzi potrzeba prezentacji szkieletu obiektów. W takim przypadku czasami konieczne jest to, aby niektóre z krawędzi wielokątów nie były rysowane. Na przykład przy rysowaniu kwadratu za pomocą dwu trójkątów powinna zostać ukryta przekątna kwadratu. Problem ten i jego rozwiązanie ilustruje rysunek 4.2.

**Rysunek 4.2.**  
*Ukrywanie krawędzi wielokątów*



Korzystając z funkcji `glEdgeFlag()` można poinformować maszynę OpenGL, czy dana krawędź wielokąta powinna być rysowana. Wywołanie funkcji `glEdgeFlag()` może przyjąć jedną z poniższych form:

```
void glEdgeFlag(GLboolean isEdge);
void glEdgeFlag(const GLboolean *isEdge);
```

Jedyną różnicą pomiędzy nimi jest to, że parametrem pierwszej z nich jest pojedyncza wartość logiczna, a parametrem drugiej wskaźnik tablicy zawierającej pojedynczą wartość logiczną (projektanci OpenGL musieli mieć jakiś powód, by przekazywać pojedynczą wartość za pomocą tablicy, ale nie jest on znany). Jeśli wartością tą będzie `GL_TRUE` (wartość domyślna), to krawędź zostanie narysowana. Wartość `GL_FALSE` zapobiega rysowaniu krawędzi.

## Antialiasing wielokątów

Podobnie jak w przypadku punktów i odcinków można także wygładzać krawędzie wielokątów stosując antialiasing. Antialiasing krawędzi wielokątów włącza i wyłącza się przekazując wartość `GL_POLYGON_SMOOTH` funkcjom  `glEnable()` i  `glDisable()`. Jak zwykle informację o bieżącym stanie antialiasingu można uzyskać przekazując tę wartość funkcji  `glGet()` lub  `glIsEnabled()`. Ze względów efektywnościowych antialiasing krawędzi wielokątów jest domyślnie nieaktywny.

## Wzór wypełnienia wielokątów

Ostatnią z ogólnych właściwości wielokątów jest możliwość określenia wzoru ich wypełnienia. Przypomina ona w ogólnym działaniu omówioną wcześniej możliwość określania wzoru linii odcinków. Zamiast wypełniać wielokąt jednolitym kolorem, maszyna OpenGL może użyć zdefiniowanego przez programistę wzoru. Możliwość ta jest domyślnie nieaktywna. Aby z niej skorzystać, trzeba więc ją najpierw aktywować przekazując wartość `GL_POLYGON_STIPPLE` funkcji `glEnable()`. Następnie należy przekazać maszynie wzór wypełnienia korzystając z poniższej funkcji:

```
void glPolygonStipple(const GLubyte *mask);
```

Parametr `mask` jest wskaźnikiem do tablicy zawierającej wzorzec o rozmiarach  $32 \times 32$  bity. W przeciwieństwie do bitów wzorca linii odcinka, które używane były w odwrotnej kolejności, bity wzorca wypełnienia stosowane są w naturalnej kolejności ich powstawania się. Trzeba zwrócić uwagę, że wzór wypełnienia definiowany jest na płaszczyźnie. Dlatego też obrót wielokąta nie powoduje równoczesnego obrotu wzorca jego wypełnienia.

Na tym można zakończyć omówienie wspólnych właściwości wielokątów. Przedstawione teraz zostaną poszczególne rodzaje wielokątów dostępne w OpenGL.

## Trójkąty

Trójkąty stanowią rodzaj wielokątów szczególnie zalecany przy tworzeniu grafiki trójwymiarowej. Decydują o tym następujące właściwości trójkątów:

- ◆ wierzchołki trójkąta leżą zawsze na jednej płaszczyźnie (inaczej mówiąc, trzy punkty definiują zawsze płaszczyznę);
- ◆ trójkąty nie mogą być wklęsłe;
- ◆ krawędzie jednego trójkąta nie mogą się wzajemnie przecinać.

Jeśli spróbuje się narysować wielokąt, który będzie posiadał jedną z właściwości, których nie mają trójkąty, to efekt takiej operacji okaże się trudny do przewidzenia. Stosowanie wyłącznie samych trójkątów nie ogranicza w żaden sposób możliwości tworzenia grafiki, ponieważ każdy wielokąt można rozłożyć na wiele trójkątów.

Narysowanie trójkąta wymaga utworzenia jego wierzchołków i w ogólnym zarysie przypomina tworzenie punktów czy odcinków. Trzeba jedynie zmienić wartość przekazywaną na wstępnie funkcji `glBegin()`:

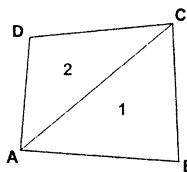
```
glBegin(GL_TRIANGLES);
    glVertex(-2.0, -1.0, 0.0);
    glVertex(3.0, 1.0, 0.0);
    glVertex(0.0, 3.0, 0.0);
glEnd();
```

Podobnie jak w przypadku punktów lub odcinków wewnętrz jednego bloku wywołań funkcji `glBegin()` i `glEnd()` można rysować wiele trójkątów. Jeśli liczba zdefiniowanych wierzchołków nie będzie dzielić się bez reszty przez 3, to dodatkowy wierzchołek lub wierzchołki zostaną zignorowane.

OpenGL dostarcza także specjalnych mechanizmów mogących poprawić efektywność tworzenia trójkątów w pewnych sytuacjach. Przykład jednej z nich prezentuje rysunek 4.3.

#### Rysunek 4.3.

Dwa trójkąty posiadające wspólne wierzchołki



Ukazane na nim trójkąty posiadają wspólne wierzchołki  $A$  i  $C$ . Rysując te trójkąty w zwykły sposób (czyli przekazując funkcji `glBegin()` wartość `GL_TRIANGLE`) trzeba określić łącznie 6 wierzchołków ( $A, B$  i  $C$  dla trójkąta 1 oraz  $A, D$  i  $C$  dla trójkąta 2). Wierzchołki  $A$  i  $C$  będą więc niepotrzebnie przetwarzane dwa razy. Takie marnotrawstwo może pojawić się na dużo większą skalę niż przedstawiono na rysunku. Skomplikowane układy trójkątów będą wtedy posiadać wiele wspólnych wierzchołków. Jeśli ograniczy się wielokrotne przetwarzanie tych samych wierzchołków, to uzyskać można poprawę efektywności tworzenia grafiki.

Jednym ze sposobów ograniczenia wielokrotnego przetwarzania wspólnych wierzchołków są łańcuchy trójkątów. Tworząc taki łańcuch trzeba na wstępnie przekazać funkcji `glBegin()` wartość `GL_TRIANGLE_STRIP`, a następnie zdefiniować sekwencję wierzchołków. OpenGL zinterpretuje ją następująco: z pierwszych trzech wierzchołków utworzy trójkąt, a każdy następny wierzchołek będzie tworzyć trójkąt z dwoma poprzednimi. W ten sposób dla każdego następnego trójkąta przetwarzany będzie tylko pojedynczy wierzchołek! W ogólnym przypadku tworząc łańcuch  $n$  trójkątów można w ten sposób ograniczyć liczbę przetwarzanych wierzchołków z  $3n$  do  $n + 2$ .

Podobną ideę wykorzystują wieńce trójkątów. Tworzą one sekwencje trójkątów wokół wspólnego centralnego wierzchołka. Rysując wieńiec trójkątów trzeba najpierw przekazać funkcji `glBegin()` wartość `GL_TRIANGLE_FAN`. Następnie definiuje się centralny wierzchołek wieńca, a po nim podaje się pary punktów tworzących z nim kolejne trójkąty. Podobnie jak w przypadku łańcuchów trójkątów, także i wieńce pozwalają narysować  $n$  trójkątów podając jedynie  $n + 2$  wierzchołków. Jednak zwykle liczba trójkątów tworzących wieńiec będzie mniejsza niż w przypadku łańcuchów, ponieważ wieńec wymaga, by wszystkie trójkąty posiadały wspólny wierzchołek.

Podstawowy problem związany ze stosowaniem łańcuchów i wieńców trójkątów polega na zidentyfikowaniu tych struktur w tworzonej grafice. Jest to dość proste dla nieskomplikowanych modeli, ale trudność tego zadania wzrasta wraz z ich złożonością. Omówienie efektywnych sposobów identyfikacji takich struktur w modelach przekracza możliwości niniejszego rozdziału. Zresztą nie będą one tak często potrzebne, ponieważ dla ograniczenia wielokrotnego przetwarzania wierzchołków i tym samym poprawienia efektywności tworzenia grafiki wprowadza się struktury zwane siatkami.

## Czworokąty

Czworokąty rysuje się za pomocą metody `glBegin()` dla wartości `GL_QUADS`, a następnie definiuje się cztery lub więcej wierzchołków. Podobnie jak w przypadku trójkątów można rysować wiele czworokątów wewnątrz jednego bloku wywołań funkcji `glBegin()` i `glEnd()`.

OpenGL umożliwia poprawę efektywności tworzenia wielu czworokątów przez zastosowanie łańcuchów. Wykorzystanie łańcuchów czworokątów sygnalizuje się za pomocą funkcji `glBegin()`, której przekazuje się wartość `GL_QUAD_STRIP`. Dzięki temu kolejne czworokąty definiuje się podając jedynie parę wierzchołków.

## Dowolne wielokąty

OpenGL umożliwia także tworzenie wielokątów o dowolnej liczbie wierzchołków. Jednak w takim przypadku wewnątrz bloku ograniczonego wywołaniami funkcji `glBegin()` i `glEnd()` można narysować tylko jeden taki wielokąt. Funkcji `glBegin()` przekazuje się wartość `GL_POLYGON`. Wywołanie funkcji `glEnd()` powoduje automatyczne zamknięcie wielokąta, czyli połączenie ostatniego wierzchołka z pierwszym. Jeśli poda się mniej niż 3 wierzchołki, to oczywiście nie zostanie narysowany żaden wielokąt.

## Przykład wykorzystania podstawowych elementów grafiki

Omówione dotąd zostały wszystkie podstawowe elementy grafiki dostępne w OpenGL. Sposoby tworzenia tych elementów oraz modyfikacji ich właściwości zilustrowane zostały fragmentami kodu. Pora zebrać te wiadomości w jedną całość i zaprezentować ich rezultaty. Kod program demonstrującego zagadnienia omówione w tym rozdziale znajduje się na dysku CD. Jeden z efektów jego działania pokazuje rysunek 4.4. Program umożliwia tworzenie wybranych za pomocą klawiatury elementów grafiki oraz modyfikację sposobu ich rysowania. Dostępne komendy podano w tabeli 4.3.

Rysunek 4.4.

Łańcuch trójkątów  
narysowany  
przez program  
demonstracyjny

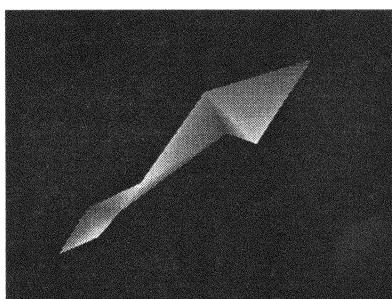


Tabela 4.3. Komendy programu demonstracyjnego

Klawisz	Akcja	Klawisz	Akcja
1	Rysuje punkty	6	Rysuje czworokąty
2	Rysuje odcinki	7	Rysuje wielokąt
3	Rysuje trójkąty	A	Włącza i wyłącza antialiasing
4	Rysuje łańcuch trójkątów	S	Włącza i wyłącza wzorce linii i wypełnienia
5	Rysuje wieniec trójkątów	P	Przełączca tryby tworzenia wielokątów

Warto dogłębiście zapoznać się z kodem źródłowym tego programu oraz spróbować zmodyfikować jego działanie. Dobre zrozumienie sposobów tworzenia podstawowych elementów grafiki będzie niezbędne podczas tworzenia każdego z kolejnych programów w tej książce.

## Podsumowanie

W rozdziale tym przekazano podstawowe informacje o maszynie stanów OpenGL. Zaprezentowano sposób pobierania wartości i jej parametrów za pomocą funkcji `glGet()` i `glIsEnabled()`. Omówiono także szereg specjalizowanych funkcji umożliwiających modyfikację wartości tych parametrów. Pozostałe aspekty maszyny stanów przedstawione będą w kolejnych rozdziałach.

W rozdziale omówiono także podstawowe elementy grafiki i sposoby modyfikacji ich właściwości. Przedstawiono sposoby tworzenia punktów, odcinków, trójkątów i innych rodzajów wielokątów.

## Rozdział 5.

# Przekształcenia układu współrzędnych i macierze OpenGL

W rozdziale tym porzucona na chwilę zostanie problematyka *tworzenia* obiektów graficznych na korzyść zagadnień związanych z ich *ruchem*. Ruch stanowi istotny składnik trójwymiarowych światów gier — bez niego byłyby one statyczne, nudne i pozbawione możliwości interakcji. OpenGL umożliwia łatwe przemieszczanie obiektów w trójwymiarowym świecie za pomocą *przekształceń układu współrzędnych*, które omówione zostaną w tym rozdziale. Przedstawione także zostaną sposoby wykorzystania własnych macierzy w OpenGL, które często stosowane są do realizacji przekształceń związanych z efektami specjalnymi w grach.

W bieżącym rozdziale przedstawione zostaną:

- ◆ podstawowe przekształcenia układu współrzędnych;
- ◆ przekształcenia kamery i widoku;
- ◆ macierze OpenGL i stosy macierzy;
- ◆ rzutowanie;
- ◆ wykorzystanie własnych macierzy w OpenGL.

## Przekształcenia układu współrzędnych

Przekształcenia pozwalają nam przemieszczać, obracać i manipulować obiektemi wirtualnego świata. Ważnym zastosowaniem przekształceń jest także rzutowanie trójwymiarowej przestrzeni na płaszczyznę ekranu. W rozdziale 3. przedstawione zostały teoretyczne podstawy przesunięcia, obrotu i skalowania. Chociaż wydaje się, że przekształcenia te modyfikują bezpośrednio dany obiekt, to jednak w rzeczywistości przekształcają jedynie jego układ współrzędnych. Gdy na przykład obróci się układ współrzędnych obiektu, to reprezentacja obiektu na ekranie także zostanie obrócona.

Podczas tworzenia trójwymiarowej grafiki wierzchołki obiektów poddawane są trzem rodzajom przekształceń, zanim zostaną narysowane na ekranie:

- ◆ **przekształceniom widoku**, które określają położenie kamery;
- ◆ **przekształceniom modelowania**, które przemieszczają obiekty na scenie;
- ◆ **przekształceniom rzutowania**, które definiują bryłę widoku oraz płaszczyzny obcięcia.

Stosowane jest także przekształcenie okienkowe, które odwzorowuje dwuwymiarowy rzut sceny w oknie na ekranie. Przekształcenie to nie dotyczy oczywiście wierzchołków trójwymiarowych obiektów, a jedynie grafiki tworzonej w oknie. W rozdziale tym omówiony zostanie jeszcze jeden rodzaj przekształcenia: przekształcenie widoku modelu. Stanowi ono kombinację przekształceń widoku i modelowania. Tabela 5.1 zawiera zestawienie omawianych przekształceń.

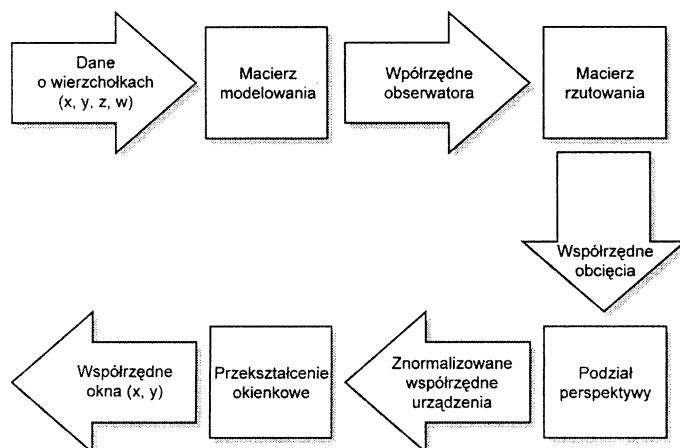
**Tabela 5.1.** Przekształcenia OpenGL

Przekształcenie	Opis
Widoku	Określa położenie kamery
Modelowania	Przemieszcza obiekty na scenie
Rzutowanie	Definiuje bryłę widoku oraz płaszczyzny obcięcia
Okienkowe	Odwzorowuje dwuwymiarowy rzut sceny w oknie
Widoku modelu	Stanowi kombinację przekształcenia widoku i przekształcenia modelowania

Przekształcenia te muszą być zawsze wykonywane w odpowiedniej kolejności. Przekształcenie widoku musi zawsze poprzedzać przekształcenie modelowania. Rzutowanie i przekształcenie okienkowe muszą natomiast poprzedzać tworzenie grafiki na ekranie. Rysunek 5.1 ilustruje kolejność wykonywania przekształceń.

**Rysunek 5.1.**

Potok przekształceń wierzchołków

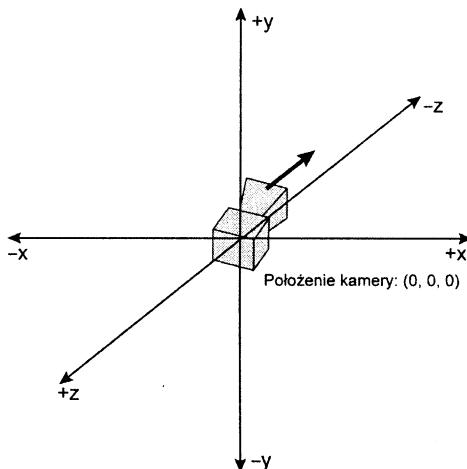


## Współrzędne kamery i obserwatora

Jedną z podstawowych koncepcji związanych z przekształceniami w OpenGL jest pojęcie *układu współrzędnych kamery* zwanego także *układem współrzędnych obserwatora*. Jest to zwykły kartezjański układ współrzędnych, w którego początku umieścił się kamera (obserwator) i skierowano ją wzdłuż ujemnej części osi z, co pokazuje rysunek 5.2.

**Rysunek 5.2.**

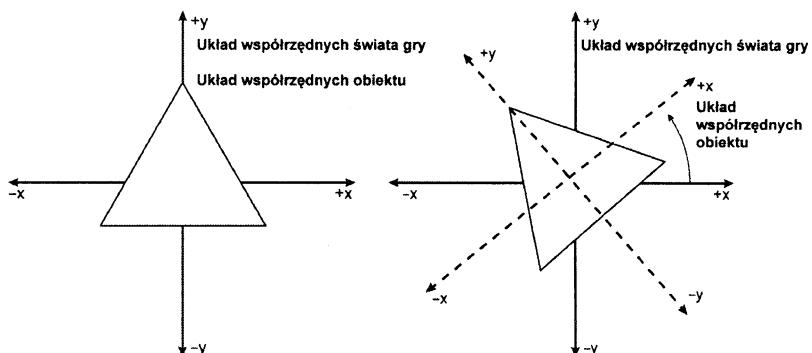
*Obserwator  
znajduje się  
w początku układu  
współrzędnych  
i spogląda wzdłuż  
ujemnej części osi z*



Układ współrzędnych obserwatora nie ulega zmianom na skutek wykonywanych przekształceń. Na przykład obracając pewien obiekt, obraca się w rzeczywistości jego układ współrzędnych względem układu współrzędnych obserwatora. Jeśli trójkąt zostanie obrócony o kąt  $45^\circ$  w kierunku przeciwnym do kierunku ruchu wskazówek zegara, to oznacza to, że obrócony w tym kierunku został układ współrzędnych tego trójkąta. Ilustruje to rysunek 5.3.

**Rysunek 5.3.**

*Obrót trójkąta  
oznacza obrót  
jego układu  
współrzędnych  
względem układu  
współrzędnych  
obserwatora*



Zrozumienie koncepcji układu współrzędnych jest kluczowe dla właściwego zrozumienia przekształceń OpenGL. W kolejnych podrozdziałach przedstawione zostaną sposoby modyfikacji układów współrzędnych, w efekcie których następują przekształcenia obiektów.

## Przekształcenia widoku

Przekształcenie widoku jest zawsze wykonywane jako pierwsze i służy do określenia położenia kamery i jej skierowania. Domyślnie kamera znajduje się zawsze w początku układu współrzędnych obserwatora i skierowana jest wzdłuż ujemnej części osi z. Położenie i orientację kamery można zmieniać za pomocą przesunięć i obrotów, które w efekcie tworzą właśnie przekształcenie widoku.

Trzeba zawsze pamiętać o tym, aby zakończyć przekształcenia widoku, zanim rozpoczęcie się wykonywanie innych przekształceń, ponieważ przekształcenie widoku przekształca bieżący układ współrzędnych względem układu współrzędnych obserwatora. Pozostałe przekształcenia odbywają się natomiast w bieżącym układzie współrzędnych.

Aby wykonać przekształcenie widoku, trzeba „wyczyścić” bieżącą macierz przekształceń za pomocą funkcji `glLoadIdentity()` o następującym prototypie:

```
void glLoadIdentity(void);
```

W wyniku jej wywołania bieżąca macierz staje się macierzą jednostkową. Jest to konieczne, ponieważ inne przekształcenia mogły zmodyfikować zawartość bieżącej macierzy.

Po zainicjowaniu bieżącej macierzy tworzy się macierz przekształcenia widoku. Istnieje kilka sposobów na jej utworzenie. Jeśli pozostawi się bieżącą macierz jako macierz jednostkową, to w rezultacie uzyskać będzie można domyślne położenie kamery w początku układu współrzędnych i domyślną jej orientację w kierunku ujemnej części osi z. Pozostałe sposoby polegają na:

- ◆ wykorzystaniu funkcji `gluLookAt()` do określenia orientacji kamery (funkcja ta hermetyzuje zbiór odpowiednich przesunięć i obrotów);
- ◆ wykorzystaniu funkcji przekształceń przesunięcia `glTranslate*`() i obrotu `glRotate*`() (funkcje te — omówione zostaną w dalszej części rozdziału — umożliwiają przemieszczenie obiektu w przestrzeni względem kamery);
- ◆ utworzeniu własnych przekształceń wykorzystujących przesunięcia i obroty we własnym układzie współrzędnych (na przykład układzie współrzędnych biegunowych w przypadku kamery okrążającej obiekt).

## Wykorzystanie funkcji `gluLookAt()`

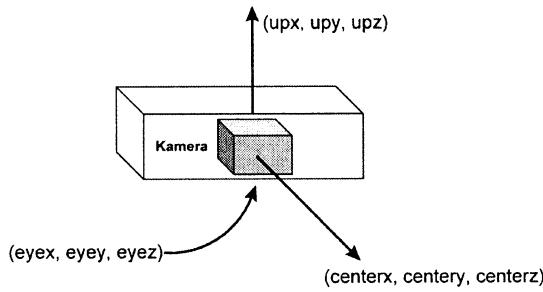
Funkcja `gluLookAt()` posiada następujący prototyp:

```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,  
               GLdouble centerx, GLdouble centery, GLdouble centerz,  
               GLdouble upx, GLdouble upy, GLdouble upz);
```

Pozwala ona określić położenie i orientację kamery. Pierwsze trzy parametry (`eyex`, `eyey`, `eyez`) definiują położenie kamery. Jeśli będą one posiadać wartość 0, to kamera zostanie umieszczona w początku układu współrzędnych. Kolejne trzy parametry (`centerx`, `centery`, `centerz`) określają punkt, w który wycelowana jest kamera czyli *kierunek widzenia* obserwatora. Zwykle punkt ten znajduje się w pobliżu środka bieżącej sceny. Ostatnie trzy parametry (`upx`, `upy`, `upz`) określają kierunek „do góry”. Ilustrację znaczenia parametrów funkcji `gluLookAt()` stanowi rysunek 5.4.

**Rysunek 5.4.**

Parametry funkcji  
*gluLookAt()*  
określają położenie  
i orientację kamery



```
gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery,
          GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);
```

Poniżej zaprezentowany został fragment kodu, który ilustruje sposób wykorzystania funkcji *gluLookAt()*. Jeśli pewne jego fragmenty wydają się niezrozumiałe, to nie ma powodu do obaw — wyjaśnia się wkrótce.

```
void DisplayScene()
{
    glClear(GL_COLOR_BUFFER_BIT); // opróżnia bufor kolorów
    glColor3f(1.0f, 0.0f, 0.0f); // wybiera kolor czerwony
    glLoadIdentity(); // czyści bieżącą macierz przekształceń

    // wykonuje przekształcenie widoku za pomocą funkcji gluLookAt()
    // umieszcza kamerę w punkcie o współrzędnych (0, 0, 10)
    // i skierowuje ją wzdłuż ujemnej części osi z podając punkt wycelowania (0, 0, -100)
    // (eyex, eyey, eyez) = (0.0, 0.0, 10.0)
    // (centerx, centery, centerz) = (0.0, 0.0, -100.0)
    // (upx, upy, upz) = (0.0, 1.0, 0.0)
    gluLookAt(0.0f, 0.0f, 10.0f, 0.0f, 0.0f, -100.0f, 0.0f, 1.0f, 0.0f);

    // rysuje trójkąt
    glBegin(GL_TRIANGLE);
        glVertex3f(10.0f, 0.0f, 0.0f);
        glVertex3f(0.0f, 10.0f, 0.0f);
        glVertex3f(-10.0f, 0.0f, 0.0f);
    glEnd();

    // opróżnia bufor
    glFlush();
}
```

Wykorzystanie funkcji *gluLookAt()* jest więc bardzo proste. Dobierając odpowiednio wartości jej parametrów można umieścić kamerę w dowolnym punkcie przestrzeni i skierować ją w wybranym kierunku.

## Wykorzystanie funkcji *glRotate\*()* i *glTranslate\*()*

Podstawową wadą funkcji *gluLookAt()* jest konieczność dołączenia do aplikacji biblioteki GLU. W przeciwnym razie trzeba skorzystać z funkcji *glRotate\*()* i *glTranslate\*()*, które reprezentują przekształcenia modelowania. Funkcje te modyfikują położenie obiektów względem stacjonarnej kamery. Zamiast więc zmieniać położenie kamery przemieszczają obiekty względem niej. Jeśli zrozumienie przekształceń modelowania sprawia czytelnikowi trudność, to przed zapoznaniem się z poniższym fragmentem kodu powinien on przeanalizować podrozdział poświęcony temu rodzajowi przekształceń.

Poniższy fragment wykorzystuje przekształcenie modelowania dla osiągnięcia takiego samego rezultatu jak przez zastosowanie funkcji gluLookAt().

```
void DisplayScene()
{
    glClear(GL_COLOR_BUFFER_BIT); // opróżnia bufor kolorów
    glColor3f(1.0f, 0.0f, 0.0f); // wybiera kolor czerwony
    glLoadIdentity(); // czyści bieżącą macierz przekształceń

    // wykonuje przekształcenie widoku za pomocą funkcji glTranslatef()
    // przesuwa bieżący układ współrzędnych do punktu (0.0, 0.0, -10.0)
    // co odpowiada umieszczeniu kamery w punkcie o współrzędnych (0.0, 0.0, 10)
    glTranslatef(0.0f, 0.0f, -10.0f);

    // rysuje trójkąt
    glBegin(GL_TRIANGLE);
        glVertex3f(10.0f, 0.0f, 0.0f);
        glVertex3f(0.0f, 10.0f, 0.0f);
        glVertex3f(-10.0f, 0.0f, 0.0f);
    glEnd();

    // opróżnia bufor
    glFlush();
}
```

Powyższy fragment kodu nie różni się istotnie od wykorzystującego funkcję gluLookAt(), ponieważ przekształcenie widoku polega w tym przypadku jedynie na przemieszczeniu kamery wzduł osi z. Jeśli przekształcenie widoku skierowałoby dodatkowo kamerę pod pewnym kątem, to należałoby także skorzystać z funkcji glRotate\*. W takim przypadku najlepiej jest skorzystać z ostatniego sposobu przekształceń widoku za pomocą własnych procedur.

## **Własne procedury przekształceń widoku**

Można założyć, że tworzony jest program symulacji lotu. W programie takim kamera umieszczona jest zwykle na miejscu pilota, a jej orientacja zgodna jest z kierunkiem ruchu samolotu. Ruch ten może odbywać się w dowolnym kierunku przestrzeni i wobec tego orientacja kamery w dowolnym momencie opisana jest przez trzy kąty obrotu względem osi układu współrzędnych. Korzystając z przekształceń modelowania można więc napisać poniższą funkcję przekształcenia widoku:

```
void PlaneView(GLfloat planeX, GLfloat planeY, GLfloat planeZ // współrzędne samolotu
                , GLfloat roll, GLfloat pitch, GLfloat yaw // kąty orientacji
                kamery
{
    // roll określa kąt obrotu względem osi z
    glRotatef(roll, 0.0f, 0.0f, 1.0f);

    // yaw określa kąt obrotu względem osi y
    glRotatef(yaw, 0.0f, 1.0f, 0.0f);

    // pitch określa kąt obrotu względem osi x
    glRotatef(roll, 1.0f, 0.0f, 0.0f);

    // zmienia położenie kamery
    glTranslatef(-planeX, -planeY, -planeZ);
}
```

Dzięki wykorzystaniu możliwości tej funkcji kamera będzie zawsze pozostawać na miejscu pilota niezależnie od ruchu samolotu. Symulacja lotu stanowi jeden z przykładów wykorzystania własnych przekształceń widoku. Inne zastosowania to na przykład ruch kamery dookoła obiektu wykorzystujący układ współrzędnych biegunkowych czy zmiany położenia i orientacji kamery za pomocą myszy i klawiatury podobne do zastosowanych w grze „Quake”.

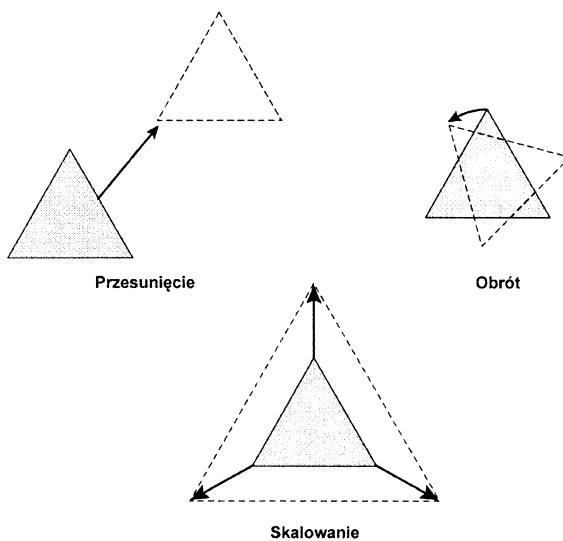
## Przekształcenia modelowania

Przekształcenia modelowania pozwalają zmieniać położenie i orientację obiektów za pomocą przesunięć, obrotów i skalowań. Wszystkie te przekształcenia można wykonać za jednym razem lub po kolejno. Rysunek 5.5 ilustruje rodzaje przekształceń modelowania:

- ◆ **przesunięcie** — polega na przemieszczeniu obiektu wzdułż pewnej osi;
- ◆ **obrót** — obiekt jest obracany o pewien kąt względem jednej z osi;
- ◆ **skalowanie** — zmienia wymiary obiektu (możliwe jest skalowanie z innym współczynnikiem dla każdej osi).

**Rysunek 5.5.**

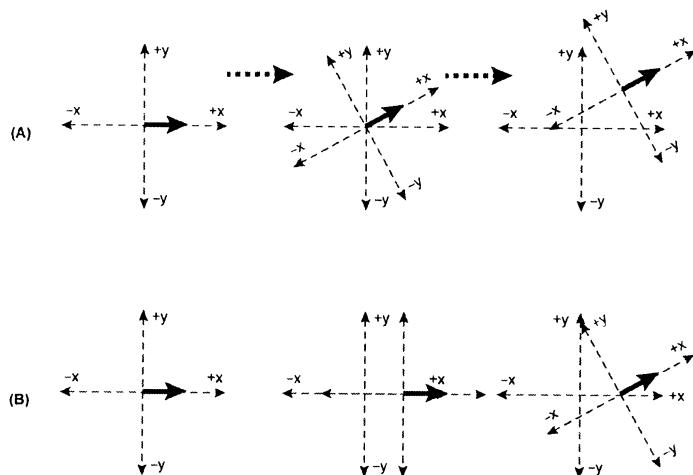
Trzy rodzaje przekształceń modelowania



Kolejność wykonywania przekształceń modelowania ma zasadnicze znaczenie dla ostatecznego wyglądu sceny. Na przykład obrót obiektu, a następnie jego przesunięcie będzie miało inny efekt niż wykonanie tych samych przekształceń w odwrotnej kolejności, co ilustruje rysunek 5.6. Pokazana na nim strzałka wychodząca ze środka układu współrzędnych w pierwszym przypadku została obrócona o kąt  $30^\circ$  dookoła osi z, a następnie przesunięta o 5 jednostek wzdułż osi x. W efekcie strzałka znalazła się ostatecznie w punkcie o współrzędnych  $(5, 4, 3)$  i skierowana jest pod kątem  $30^\circ$  względem dodatniej części osi x. Natomiast w drugim z pokazanych przypadków strzałka została najpierw przesunięta o 5 jednostek wzdułż osi x, a następnie obrócona o kąt  $30^\circ$  dookoła osi z. Choć orientacja strzałki jest taka sama jak w poprzednim przypadku, to jednak współrzędne jej początku są inne  $(5, 0)$ .

**Rysunek 5.6.**

(A) obrót poprzedzający przesunięcie,  
 (B) przesunięcie poprzedzające obrót



## Rzutowanie

Rzutowanie definiuje bryłę widoku oraz płaszczyzny obcięcia. Wykonywane jest po przekształceniach modelowania, które zostaną omówione szczegółowo w dalszej części rozdziału. Zadaniem rzutowania jest ustalenie tego, które obiekty znajdują się w bryle widoku i jak będą wyglądać na ekranie. Zastosowanie rzutowania można porównać do użycia odpowiedniego obiektywu kamery. Podobnie jak pole widzenia zależy od wyboru obiektywu kamery, tak i obraz uzyskany na ekranie zależy od wyboru przekształcenia rzutowania. Jeśli wybrany zostanie obiektyw szerokokątny, to uzyska się panoramiczny obraz sceny zawierający jednak niewiele szczegółów. Jeśli natomiast pole widzenia zostanie zauważone, co przypominać będzie zastosowanie teleobiektywu, to uzyskany zostanie efekt zbliżenia obiektów pozwalający dostrzec więcej szczegółów.

OpenGL umożliwia dwa rodzaje rzutowania:

- ◆ **rzutowanie perspektywiczne** — rzutowanie to stanowi dobre przybliżenie sposobu widzenia rzeczywistego świata (obiekty położone dalej od obserwatora wydają się mniejsze niż te, które położone są bliżej);
- ◆ **rzutowanie ortograficzne** — rzutowanie to zachowuje na ekranie rzeczywiste wymiary obiektów niezależnie od ich odległości od obserwatora; znajduje to zastosowanie w komputerowym wspomaganiu projektowania.

## Przekształcenie okienkowe

*Przekształcenie okienkowe* jest wykonywane jako ostatnie w potoku przekształceń. Umożliwia ono odwzorowanie dwuwymiarowego obrazu (uzyskanego za pomocą rzutowania perspektywicznego) na obszar okna, w którym tworzona jest grafika. Przekształcenie okienkowe określa, w jaki sposób powinny zmienić się rozmiary prezentowanego obrazu, by pasował on do rozmiarów okna grafiki.

# OpenGL i macierze

Podrozdział zajmował się będzie sposobem wykonywania przedstawionych przekształceń w OpenGL. OpenGL wykorzystuje w tym celu algebrę macierzy oraz dysponuje specjalnym *stosem macierzy* ułatwiającym tworzenie skomplikowanych modeli z wielu prostych obiektów.

## Macierz modelowania

Macierz modelowania definiuje układ współrzędnych wykorzystywany podczas tworzenia obiektów. Macierz ta posiada wymiary  $4 \times 4$  i mnożona jest o wektory reprezentujące wierzchołki oraz macierze przekształceń. W wyniku tego mnożenia powstaje macierz reprezentująca rezultat przekształceń wierzchołków.

Funkcja `glMatrixMode()` pozwala poinformować maszynę OpenGL, że będzie modyfikowana macierz modelowania. Funkcja ta posiada następujący prototyp:

```
void glMatrixMode(GLenum mode);
```

Przed wykonaniem jakiegokolwiek przekształcenia trzeba określić, czy będzie ono modyfikować macierz modelowania czy macierz rzutowania. Jeśli przekazana zostanie funkcji `glMatrixMode()` wartość `GL_MODELVIEW`, to przekształcenia będą modyfikować macierz modelowania:

```
glMatrixMode(GL_MODELVIEW);
```

Funkcji `glMatrixMode()` można także przekazać wartości `GL_PROJECTION` lub `GL_TEXTURE`. Pierwsza z nich określa macierz rzutowania, a druga macierz tekstury, której zastosowanie omówione zostanie w rozdziale 8.

W większości przypadków po wyborze macierzy modelowania należy ją zresetować za pomocą funkcji `glLoadIdentity()` omówionej już wcześniej. W wyniku jej wywołania macierz modelowania stanie się macierzą jednostkową i przywrócony zostanie wyjściowy układ współrzędnych. Ilustruje to poniższy fragment kodu:

```
// ...
glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); // resetuje macierz modelowania

// ... wykonuje przekształcenia

glBegin(GL_POINTS);
    glVertex3f(0.0f, 0.0f, 0.0f);
 glEnd();

// ... kontynuacja programu
```

## Przesunięcie

Przesunięcie pozwala przemieścić obiekt z jednego punktu przestrzeni do innego punktu. OpenGL umożliwia wykonanie przesunięć za pomocą funkcji `glTranslatef()` i `glTranslated()` zdefiniowanych jak poniżej:

```
void glTranslatef(GLfloat x, GLfloat y, GLfloat z);
void glTranslated(GLdouble x, GLdouble y, GLdouble z);
```

Jedyną różnicą pomiędzy tymi funkcjami jest typ ich parametrów. Wybór jednej z funkcji zależy więc od wymaganej dokładności przekształcenia.

Parametry  $x, y, z$  opisują wartość przesunięcia wzdłuż odpowiednich osi układu współrzędnych. Na przykład wywołanie

```
glTranslatef(3.0f, 1.0f, 8.0f);
```

spowoduje przesunięcie obiektu o trzy jednostki w dodatnim kierunku osi x, o jednostkę w dodatnim kierunku osi y i o osiem jednostek w dodatnim kierunku osi z.

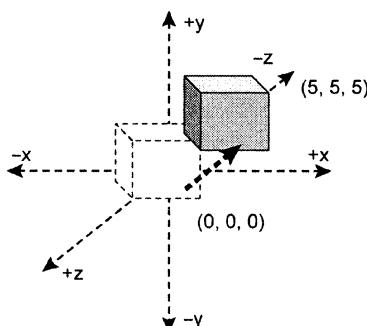
Można założyć, że należy przesunąć sześcian do punktu o współrzędnych (5, 5, 5). Najpierw trzeba wybrać macierz modelowania i zresetować ją do macierzy jednostkowej. Następnie należy przesunąć bieżącą macierz do punktu (5, 5, 5) i wywołać funkcję rysowania sześcianu `DrawCube()`. Operacje te wykonuje poniższy fragment kodu.

```
glMatrixMode(GL_MODELVIEW); // wybiera macierz modelowania
glLoadIdentity();           // resetuje macierz modelowania
glTranslatef(5.0f, 5.0f, 5.0f); // przesunięcie do punktu (5,5,5)
DrawCube();                 // rysuje sześciąn
```

Rysunek 5.7 ilustruje wykonanie tego fragmentu kodu.

**Rysunek 5.7.**

Przesunięcie  
sześcianu z początku  
układu współrzędnych  
do punktu (5, 5, 5)



## Obrót

Również funkcja umożliwiająca wykonywanie obrotów w OpenGL posiada dwie wersje różniące się typem parametrów:

```
void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);
void glRotated(GLdouble angle, GLdouble x, GLdouble y, GLdouble z);
```

Funkcje te wykonują obrót o kąt *angle* wokół wektora o składowych *x*, *y* i *z*. Wartość kąta podaje się w stopniach, w kierunku przeciwnym do kierunku ruchu wskazówek zegara.

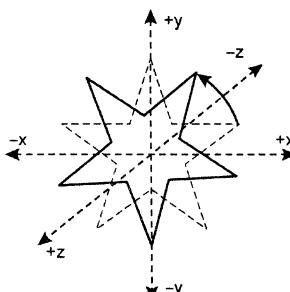
Na przykład obrót wokół osi *y* o kąt  $135^\circ$  w kierunku przeciwnym do kierunku ruchu wskazówek zegara wykonuje się w następujący sposób:

```
glRotatef(135.0f, 0.0f, 1.0f, 0.0f);
```

Wartość  $1.0f$  dla parametru *y* określa wektor jednostkowy skierowany wzdłuż osi *y*. Wykonując obroty dookoła osi układu współrzędnych wystarczy właśnie wyspecyfikować odpowiedni wektor jednostkowy. Rysunek 5.8 ilustruje wykonanie obrotu.

**Rysunek 5.8.**

*Funkcja glRotatef()*  
wykonuje obrót  
o dany kąt wokół osi  
zdefiniowanej  
za pomocą wektora  
jednostkowego



```
glRotatef(45.0f, 0.0f, 0.0f, 1.0f);
```

Obrót w kierunku zgodnym z kierunkiem ruchu wskazówek zegara wymaga podania ujemnej wartości kąta. Poniżej zaprezentowany został przykład obrotu wokół osi *y* o kąt  $135^\circ$  w kierunku zgodnym z kierunkiem ruchu wskazówek zegara:

```
glRotatef(-135.0f, 0.0f, 1.0f, 0.0f);
```

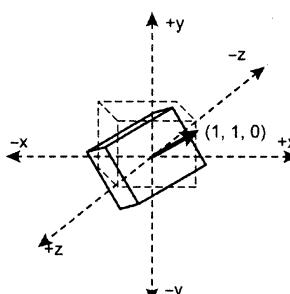
Wykonanie obrotu dookoła dowolnej osi wymaga określenia wszystkich wartości parametrów *x*, *y* i *z*. Oś obrotu można wyobrazić sobie jako odcinek łączący początek układu współrzędnych z punktem (*x*, *y*, *z*). Na przykład obrót o kąt  $90^\circ$  wokół osi określonej przez wektor  $(1, 1, 0)$  odbędzie się względem osi przechodzącej przez początek układu współrzędnych i punkt  $(1, 1, 0)$ . W kodzie programu przekształcenie to należy zapisać następująco:

```
glRotatef(90.0f, 1.0f, 1.0f, 0.0f);
```

Rysunek 5.9 ilustruje jego wykonanie.

**Rysunek 5.9.**

*Obrót wokół  
dowolnej osi*



```
glRotatef(90.0f, 1.0f, 1.0f, 0.0f);
```

Poniżej zaprezentowany został fragment kodu wykonującego obrót sześcianu o kąt  $60^\circ$  wokół osi x i kąt  $45^\circ$  wokół osi y:

```
glMatrixMode(GL_MODELVIEW);           // wybiera macierz modelowania
glLoadIdentity();                     // resetuje macierz modelowania

glRotatef(60.0f, 1.0f, 0.0f, 0.0f);   // obrót o kąt 60 stopni wokół osi x
glRotatef(45.0f, 0.0f, 1.0f, 0.0f);   // obrót o kąt 45 stopni wokół osi y
DrawCube();                          // rysuje sześciąn
```

## Skalowanie

*Skalowanie* polega na zmianie rozmiarów obiektu. W jego wyniku wierzchołki obiektu zbliżają się lub oddalają od siebie wzdłuż osi układu współrzędnych zgodnie z wartościami odpowiednich współczynników skalowania. OpenGL udostępnia funkcję skalowania `glScalef()` posiadającą podobnie jak w przypadku poprzednich przekształceń modelowania dwie wersje:

```
glScalef(GLfloat x, GLfloat y, GLfloat z);
glScaled(GLdouble x, GLdouble y, GLdouble z);
```

Parametry  $x$ ,  $y$  i  $z$  określają współczynniki skalowania wzdłuż osi układu współrzędnych. Poniższe przekształcenie zwiększa dwukrotnie rozmiary obiektu:

```
glScalef(2.0f, 2.0f, 2.0f);
```

Gdyby ktoś zechciał przekształcić sześcian w prostopadłościan zwiększając szerokość sześcianu (rozumianą jako wymiar na osi x) dwukrotnie, a pozostałe wymiary pozostawiając bez zmian, to w tym celu musiałby wykonać następujące przekształcenie:

```
glScalef(2.0f, 1.0f, 1.0f);
```

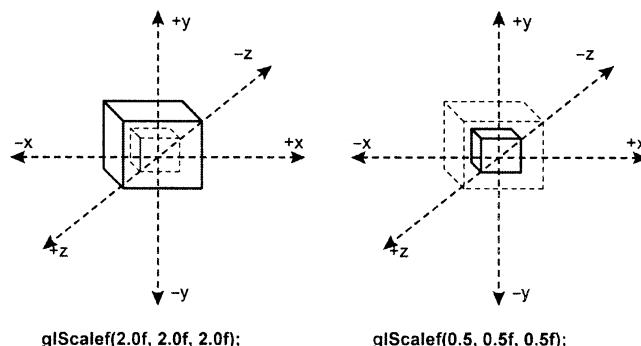
Aby zmniejszyć rozmiary obiektu, trzeba podać wartości współczynników skalowania mniejsze od jeden, na przykład:

```
glScalef(0.5f, 0.5f, 0.5f);
```

Powyższe wywołanie zmniejszy rozmiary obiektu o połowę. Współczynnik o wartości 0.2 zmniejszy rozmiar obiektu pięciokrotnie, współczynnik 0.1 dziesięciokrotnie i tak dalej. Rysunek 5.10 ilustruje działanie funkcji `glScalef()`.

**Rysunek 5.10.**

Działanie funkcji  
`glScalef()`



Poniżej zaprezentowany został fragment kodu zwiększający dwukrotnie rozmiary sześcianu:

```
glMatrixMode(GL_MODELVIEW); // wybiera macierz modelowania
glLoadIdentity();           // resetuje macierz modelowania

glScale(2.0f, 2.0f, 2.0f); // skaluje wszystkie wymiary przez współczynnik 2
DrawCube();                // rysuje sześcian
```

## Stos macierzy

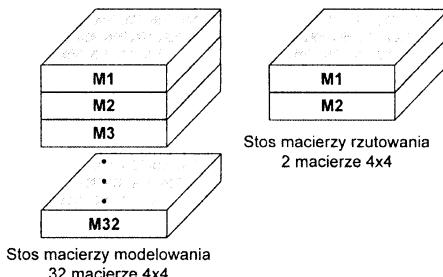
Wykorzystywana w poprzednich przykładach macierz modelowania stanowi wierzchołek stosu macierzy OpenGL. OpenGL posiada trzy typy stosów macierzy:

- ◆ stos macierzy modelowania;
- ◆ stos macierzy rzutowania;
- ◆ stos macierzy tekstur.

Macierz modelowania znajduje się na szczycie stosu macierzy modelowania, a macierz rzutowania na szczycie stosu macierzy rzutowania. Więcej informacji o tych stosach macierzy dostarcza rysunek 5.11. Stos macierzy tekstur wykorzystywany jest do przekształceń układu współrzędnych.

**Rysunek 5.11.**

*Stos macierzy modelowania może zawierać do 32 macierzy o rozmiarach 4x4, a stos macierzy rzutowania zawiera dwie takie macierze (implementacja Microsoft OpenGL)*



Stos macierzy modelowania umożliwia tworzenie skomplikowanych modeli za pomocą prostych obiektów. Zilustrujemy to na przykładzie modelu robota zbudowanego z prostopadłościanów. Podzielimy go na elementy składowe, takie jak korpus, dwoje ramion, głowę i dwie nogi. W programie elementom tym odpowiadać będą funkcje tworzące ich reprezentację graficzną. Każda z nich będzie rysować odpowiedni element wokół początku układu współrzędnych zorientowany względem osi układu.

Tworząc model takiego robota trzeba narysować najpierw jego korpus. Następnie — po przesunięciu układu współrzędnych w lewo względem korpusu — można narysować lewe ramię robota. Podobnie należy postąpić w przypadku prawego ramienia — trzeba przesunąć układ współrzędnych w prawo względem korpusu i wywołać tę samą funkcję rysowania ramienia. Również głowę i nogi należy rysować zawsze jako przesunięte względem korpusu.

Taki sposób tworzenia modeli ułatwiają właśnie stosy macierzy. Można przesunąć najpierw układ współrzędnych obiektu *A* względem obiektu *B*, a później narysować obiekt

A w początku uzyskanego układu, po czym usunąć ze stosu wykonane przekształcenie i powrócić do układu współrzędnych obiektu B. W tym celu zwykle używa się dwu funkcji umożliwiających operacje na stosie macierzy: `glPushMatrix()` i `glPopMatrix()`.

Funkcja `glPushMatrix()` tworzy kopię bieżącej macierzy i odkłada ją na szczytce stosu macierzy. W ten sposób można „zapamiętać” na stosie bieżący układ współrzędnych. Funkcja `glPushMatrix()` zdefiniowana jest następująco:

```
void glPushMatrix(void);
```

Jeśli odkładając zbyt wiele macierzy na stosie doprowadzi się do jego przepełnienia, to spowoduje to błąd `GL_STACK_OVERFLOW` maszyny OpenGL.

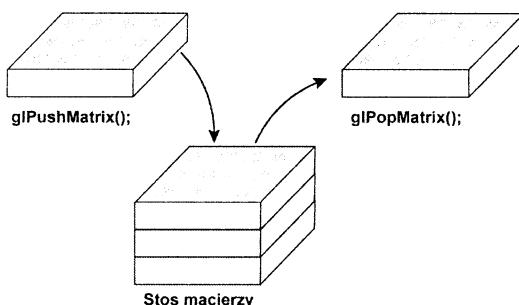
Funkcja `glPopMatrix()` usuwa macierz znajdująca się na szczytce stosu umożliwiając tym samym powrót do zapamiętanego wcześniej układu współrzędnych. Funkcja `glPopMatrix()` zdefiniowana jest następująco:

```
void glPopMatrix(void);
```

Jeśli zostanie wywołana, gdy na stosie znajdująć się będzie tylko jedna macierz, to spowoduje to błąd `GL_STACK_UNDERFLOW` maszyny OpenGL.

Rysunek 5.12 ilustruje sposób działania funkcji `glPushMatrix()` i `glPopMatrix()`.

**Rysunek 5.12.**  
*Operacje  
na stosie macierzy*



## Model robota

Poniżej zaprezentowany został kod źródłowy programu, który wykorzystuje w praktyce wszystkie informacje przedstawione w tym rozdziale. Prezentuje on model kroczącego robota okrążany przez kamerę. Model ten tworzą sześciiany skalowane do różnych rozmiarów w celu reprezentacji korpusu, głowy, ramion i nóg robota. Podczas analizowania kodu programu szczególną uwagę należy zwrócić na sposób wykorzystania funkcji `glPushMatrix()` i `glPopMatrix()` do tworzenia i poruszania modelu robota.

A oto pełen kod źródłowy programu:

```
#define WIN32_LEAN_AND_MEAN      // "odchudza" aplikację Windows

////// Pliki nagłówkowe
#include <windows.h>           // standardowy plik nagłówkowy Windows
#include <gl/gl.h>              // standardowy plik nagłówkowy OpenGL
#include <gl/glu.h>             // plik nagłówkowy biblioteki GLU
#include <gl/glaux.h>            // plik nagłówkowy pomocniczych funkcji OpenGL
```

```
////// Zmienne globalne
float angle = 0.0f;      // bieżący kąt obrotu kamery
HDC g_HDC;                // kontekst urządzenia
bool fullScreen = false;

////// Zmienne opisujące robota
float legAngle[2] = { 0.0f, 0.0f }; // bieżące kąty obrotu nóg i ramion
float armAngle[2] = { 0.0f, 0.0f };

// DrawCube
// opis: ponieważ każdy element robota powstaje przez przekształcenie sześcianu,
//       to do rysowania sześcianu w danym punkcie będziemy używać tej funkcji.

void DrawCube(float xPos, float yPos, float zPos)
{
    glPushMatrix();
    glTranslatef(xPos, yPos, zPos);
    glBegin(GL_POLYGON);
    glVertex3f(0.0f, 0.0f, 0.0f);      // górna ściana
    glVertex3f(0.0f, 0.0f, -1.0f);
    glVertex3f(-1.0f, 0.0f, -1.0f);
    glVertex3f(-1.0f, 0.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 0.0f);      // przednia ściana
    glVertex3f(-1.0f, 0.0f, 0.0f);
    glVertex3f(-1.0f, -1.0f, 0.0f);
    glVertex3f(0.0f, -1.0f, 0.0f);      // ściana po prawej
    glVertex3f(0.0f, -1.0f, -1.0f);
    glVertex3f(0.0f, 0.0f, -1.0f);      // ściana po lewej
    glVertex3f(-1.0f, 0.0f, 0.0f);
    glVertex3f(-1.0f, 0.0f, -1.0f);
    glVertex3f(-1.0f, -1.0f, -1.0f);
    glVertex3f(-1.0f, -1.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 0.0f);      // dolna ściana
    glVertex3f(0.0f, -1.0f, -1.0f);
    glVertex3f(-1.0f, -1.0f, -1.0f);
    glVertex3f(-1.0f, -1.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 0.0f);      // tylnia ściana
    glVertex3f(-1.0f, 0.0f, -1.0f);
    glVertex3f(-1.0f, -1.0f, -1.0f);
    glVertex3f(0.0f, 0.0f, -1.0f);
    glEnd();
    glPopMatrix();
}

// DrawArm
// opis: rysuje pojedyncze ramię robota
void DrawArm(float xPos, float yPos, float zPos)
{
    glPushMatrix();
    glColor3f(1.0f, 0.0f, 0.0f);      // kolor czerwony
    glTranslatef(xPos, yPos, zPos);
    glScalef(1.0f, 4.0f, 1.0f);      // ramię jest prostopadłościem 1x4x1
    DrawCube(0.0f, 0.0f, 0.0f);
    glPopMatrix();
}
```

```
// DrawHead
// opis: rysuje głowę robota
void DrawHead(float xPos, float yPos, float zPos)
{
    glPushMatrix();
    glColor3f(1.0f, 1.0f, 1.0f);      // kolor biały
    glTranslatef(xPos, yPos, zPos);
    glScalef(2.0f, 2.0f, 2.0f);      // głowa jest sześciągiem 2x2x2
    DrawCube(0.0f, 0.0f, 0.0f);
    glPopMatrix();
}

// DrawTorso
// opis: rysuje korpus robota
void DrawTorso(float xPos, float yPos, float zPos)
{
    glPushMatrix();
    glColor3f(0.0f, 0.0f, 1.0f);      // kolor niebieski
    glTranslatef(xPos, yPos, zPos);
    glScalef(3.0f, 5.0f, 2.0f);      // korpus jest prostopadłościanem 3x5x2
    DrawCube(0.0f, 0.0f, 0.0f);
    glPopMatrix();
}

// DrawLeg
// opis: rysuje pojedynczą nogę robota
void DrawLeg(float xPos, float yPos, float zPos)
{
    glPushMatrix();
    glColor3f(1.0f, 1.0f, 0.0f);      // kolor żółty
    glTranslatef(xPos, yPos, zPos);
    glScalef(1.0f, 5.0f, 1.0f);      // noga jest prostopadłościanem 1x5x1
    DrawCube(0.0f, 0.0f, 0.0f);
    glPopMatrix();
}

// DrawRobot
// opis: rysuje model robota w punkcie (xpos,ypos,zpos)
void DrawRobot(float xPos, float yPos, float zPos)
{
    static bool leg1 = true;          // zmienne określające położenie nóg i ramion
    static bool leg2 = false;         // true = przednie, false = tylnie

    static bool arm1 = true;
    static bool arm2 = false;

    glPushMatrix();

    glTranslatef(xPos, yPos, zPos);  // rysuje model robota w określonym punkcie

    // rysuje elementy robota
    DrawHead(1.0f, 2.0f, 0.0f);
    DrawTorso(1.5f, 0.0f, 0.0f);

    glPushMatrix();
    // w zależności od położenia ramienia zwiększa lub zmniejsza jego kąt
    if (arm1)
        armAngle[0] = armAngle[0] + 0.1f;
```

```
else
    armAngle[0] = armAngle[0] - 0.1f;

    // jeśli ramię osiągnęło maksymalny kąt,
    // to zmienia kierunek
    if (armAngle[0] >= 15.0f)
        arm1 = false;
    if (armAngle[0] <= -15.0f)
        arm1 = true;

    // przesuwa ramię i obraca je w celu uzyskania efektu ruchu
    glTranslatef(0.0f, -0.5f, 0.0f);
    glRotatef(armAngle[0], 1.0f, 0.0f, 0.0f);
    DrawArm(2.5f, 0.0f, -0.5f);
    glPopMatrix();

    glPushMatrix();
    // jeśli ramię porusza się do przodu, zwiększa kąt
    // w przeciwnym razie zmniejsza
    if (arm2)
        armAngle[1] = armAngle[1] + 0.1f;
    else
        armAngle[1] = armAngle[1] - 0.1f;

    // jeśli ramię osiągnęło maksymalny kąt,
    // to zmienia kierunek
    if (armAngle[1] >= 15.0f)
        arm2 = false;
    if (armAngle[1] <= -15.0f)
        arm2 = true;

    // przesuwa ramię i obraca je w celu uzyskania efektu ruchu
    glTranslatef(0.0f, -0.5f, 0.0f);
    glRotatef(armAngle[1], 1.0f, 0.0f, 0.0f);
    DrawArm(-1.5f, 0.0f, -0.5f);
    glPopMatrix();

    //DrawArm(-1.5f, 0.0f, -0.5f);

    // nogi poruszamy względem korpusu robota
    // nogą 1, czyli prawa nogą robota
    glPushMatrix();

    // w zależności od położenia nogi zwiększa lub zmniejsza kąt
    if (leg1)
        legAngle[0] = legAngle[0] + 0.1f;
    else
        legAngle[0] = legAngle[0] - 0.1f;

    // jeśli nogą osiągnęła maksymalny kąt,
    // to zmienia kierunek
    if (legAngle[0] >= 15.0f)
        leg1 = false;
    if (legAngle[0] <= -15.0f)
        leg1 = true;

    // przesuwa nogę i obraca ją w celu uzyskania efektu ruchu
    glTranslatef(0.0f, -0.5f, 0.0f);
    glRotatef(legAngle[0], 1.0f, 0.0f, 0.0f);
```

```

// rysuje nogę
DrawLeg(-0.5f, -5.0f, -0.5f);

glPopMatrix();

// to samo dla nogi 2, lewej nogi robota
glPushMatrix();

if (leg2)
    legAngle[1] = legAngle[1] + 0.1f;
else
    legAngle[1] = legAngle[1] - 0.1f;

if (legAngle[1] >= 15.0f)
    leg2 = false;
if (legAngle[1] <= -15.0f)
    leg2 = true;

glTranslatef(0.0f, -0.5f, 0.0f);
glRotatef(legAngle[1], 1.0f, 0.0f, 0.0f);
DrawLeg(1.5f, -5.0f, -0.5f);

glPopMatrix();
glPopMatrix();
}

// Render
// opis: rysuje scenę
void Render()
{
    glEnable(GL_DEPTH_TEST); // włącza testowanie głębi

    // tworzy grafikę
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // kolor czarny
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // opróżnia bufore ekranu i głębi
    glLoadIdentity(); // resetuje macierz modelowania

    angle = angle + 0.05f; // zwiększa kąt kamery
    if (angle >= 360.0f) // jeśli okrążyła obiekt, to zeruje licznik
        angle = 0.0f;

    glPushMatrix(); // odkłada bieżącą macierz na stos
    glLoadIdentity(); // resetuje macierz
    glTranslatef(0.0f, 0.0f, -30.0f); // przesunięcie do punktu (0, 0, -30)
    glRotatef(angle, 0.0f, 1.0f, 0.0f); // obrót robota względem osi y
    DrawRobot(0.0f, 0.0f, 0.0f); // rysuje robota
    glPopMatrix(); // usuwa bieżącą macierz ze stosu

    glFlush();

    SwapBuffers(g_HDC); // przełącza bufore
}

// funkcja określająca format pikseli
void SetupPixelFormat(HDC hDC)
{
    int nPixelFormat; // indeks formatu pikseli
}

```

```
static PIXELFORMATDESCRIPTOR pfd = {  
    sizeof(PIXELFORMATDESCRIPTOR), // rozmiar struktury  
    1,                          // domyślna wersja  
    PFD_DRAW_TO_WINDOW |        // grafika w oknie  
    PFD_SUPPORT_OPENGL |       // grafika OpenGL  
    PFD_DOUBLEBUFFER,          // podwójne buforowanie  
    PFD_TYPE_RGBA,             // tryb kolorów RGBA  
    32,                         // 32-bitowy opis kolorów  
    0, 0, 0, 0, 0, 0,           // nie specyfikuje bitów kolorów  
    0,                          // bez buforu alfa  
    0,                          // nie specyfikuje bitu przesunięcia  
    0,                          // bez bufora akumulacji  
    0, 0, 0, 0,                 // ignoruje bity akumulacji  
    16,                         // 16-bit bufor z  
    0,                          // bez bufora powielania  
    0,                          // bez buforów pomocniczych  
    PFD_MAIN_PLANE,            // główna płaszczyzna rysowania  
    0,                          // zarezerwowane  
    0, 0, 0 };                  // ignoruje maski warstw  
  
// wybiera najbardziej zgodny format pikseli  
nPixelFormat = ChoosePixelFormat(hDC, &pfd);  
  
// określa format pikseli dla danego kontekstu urządzenia  
SetPixelFormat(hDC, nPixelFormat, &pfd);  
}  
  
// procedura okienkowa  
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)  
{  
    static HGLRC hRC;           // kontekst tworzenia grafiki  
    static HDC hDC;             // kontekst urządzenia  
    int width, height;         // szerokość i wysokość okna  
  
    switch(message)  
    {  
        case WM_CREATE:         // okno jest tworzone  
  
            hDC = GetDC(hwnd); // pobiera kontekst urządzenia dla okna  
            g_HDC = hDC;  
            SetupPixelFormat(hDC); // wywołuje funkcję określającą format pikseli  
  
            // tworzy kontekst tworzenia grafiki i czyni go bieżącym  
            hRC = wglCreateContext(hDC);  
            wglMakeCurrent(hDC, hRC);  
  
            return 0;  
            break;  
  
        case WM_CLOSE:           // okno jest zamkane  
  
            // deaktywuje bieżący kontekst tworzenia grafiki i usuwa go  
            wglMakeCurrent(hDC, NULL);  
            wglDeleteContext(hRC);  
  
            // wstawia komunikat WM_QUIT do kolejki  
            PostQuitMessage(0);  
  
            return 0;  
            break;  
    }  
}
```

```
case WM_SIZE:
    height = HIWORD(lParam); // pobiera nowe rozmiary okna
    width = LOWORD(lParam);

    if (height==0)      // unika dzielenie przez 0
    {
        height=1;
    }

    glViewport(0, 0, width, height); // nadaje nowe wymiary okna OpenGL
    glMatrixMode(GL_PROJECTION);    // wybiera macierz rzutowania
    glLoadIdentity();              // resetuje macierz rzutowania

    // wyznacza proporcje obrazu
    gluPerspective(54.0f,(GLfloat)width/(GLfloat)height,1.0f,1000.0f);

    glMatrixMode(GL_MODELVIEW);    // wybiera macierz modelowania
    glLoadIdentity();              // resetuje macierz modelowania

    return 0;
break;

default:
    break;
}

return (DefWindowProc(hwnd, message, wParam, lParam));
}

// punkt, w którym rozpoczyna się wykonywanie aplikacji
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
int nShowCmd)
{
    WNDCLASSEX windowClass; // klasa okna
    HWND hwnd;             // uchwyt okna
    MSG msg;               // komunikat
    bool done;              // znacznik zakończenia aplikacji
    DWORD dwExStyle;       // rozszerzony styl okna
    DWORD dwStyle;          // styl okna
    RECT windowRect;

    // zmienne pomocnicze
    int width = 800;
    int height = 600;
    int bits = 32;

    //fullScreen = TRUE;

    windowRect.left=(long)0; // struktura określająca rozmiary okna
    windowRect.right=(long)width;
    windowRect.top=(long)0;
    windowRect.bottom=(long)height;

    // definicja klasy okna
    windowClass.cbSize    = sizeof(WNDCLASSEX);
    windowClass.style     = CS_HREDRAW | CS_VREDRAW;
    windowClass.lpfnWndProc = WndProc;
    windowClass.cbClsExtra = 0;
```

```
windowClass.cbWndExtra = 0;
windowClass.hInstance = hInstance;
windowClass.hIcon = LoadIcon(NULL, IDI_APPLICATION); // domyślna ikona
windowClass.hCursor = LoadCursor(NULL, IDC_ARROW); // domyślny kursor
windowClass.hbrBackground = NULL; // bez tła
windowClass.lpszMenuName = NULL; // bez menu
windowClass.lpszClassName = "MojaKlasa";
windowClass.hIconSm = LoadIcon(NULL, IDI_WINLOGO); // logo Windows

// rejestruje klasę okna
if (!RegisterClassEx(&windowClass))
    return 0;

if (fullScreen) // tryb pełnoekranowy?
{
    DEVMODE dmScreenSettings; // tryb urządzenia
    memset(&dmScreenSettings, 0, sizeof(dmScreenSettings));
    dmScreenSettings.dmSize = sizeof(dmScreenSettings);
    dmScreenSettings.dmPelsWidth = width; // szerokość ekranu
    dmScreenSettings.dmPelsHeight = height; // wysokość ekranu
    dmScreenSettings.dmBitsPerPel = bits; // bitów na piksel
    dmScreenSettings.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;

    if (ChangeDisplaySettings(&dmScreenSettings, CDS_FULLSCREEN) != DISP_CHANGE_SUCCESSFUL)
    {
        // przełączenie trybu nie powiodło się, z powrotem tryb okienkowy
        MessageBox(NULL, "Display mode failed", NULL, MB_OK);
        fullScreen=FALSE;
    }
}

if (fullScreen) // tryb pełnoekranowy?
{
    dwExStyle=WS_EX_APPWINDOW; // rozszerzony styl okna
    dwStyle=WS_POPUP; // styl okna
    ShowCursor(FALSE); // ukrywa kursor myszy
}
else
{
    dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE; // definicja klasy okna
    dwStyle=WS_OVERLAPPEDWINDOW; // styl okna
}

AdjustWindowRectEx(&windowRect, dwStyle, FALSE, dwExStyle); // koryguje rozmiar okna

// tworzy okno
hwnd = CreateWindowEx(NULL,
                      "MojaKlasa", // nazwa klasy
                      "Model robota w OpenGL", // nazwa aplikacji
                      dwStyle | WS_CLIPCHILDREN |
                      WS_CLIPSIBLINGS,
                      0, 0, // współrzędne x,y
                      windowRect.right - windowRect.left,
                      windowRect.bottom - windowRect.top, // szerokość, wysokość
                      NULL, // uchwyt okna nadziednego
```

```

        NULL,                                // uchwyt menu
        hInstance,                            // instancja aplikacji
        NULL);                               // bez dodatkowych parametrów

// sprawdza, czy utworzenie okna nie powiodło się (wtedy wartość hwnd równa NULL)
if (!hwnd)
    return 0;

ShowWindow(hwnd, SW_SHOW);   // wyświetla okno
UpdateWindow(hwnd);         // aktualizuje okno

done = false;                // inicjuje zmienną warunku pętli

// pętla przetwarzania komunikatów
while (!done)
{
    PeekMessage(&msg, hwnd, NULL, NULL, PM_REMOVE);

    if (msg.message == WM_QUIT) // aplikacja otrzymała komunikat WM_QUIT?
    {
        done = true;           // jeśli tak, to kończy działanie
    }
    else
    {
        Render();

        TranslateMessage(&msg); // tłumaczy komunikat i wysyła do systemu
        DispatchMessage(&msg);
    }
}

if (fullScreen)
{
    ChangeDisplaySettings(NULL,0);      // przywraca pulpit
    ShowCursor(TRUE);                 // i wskaźnik myszy
}
return msg.wParam;
}

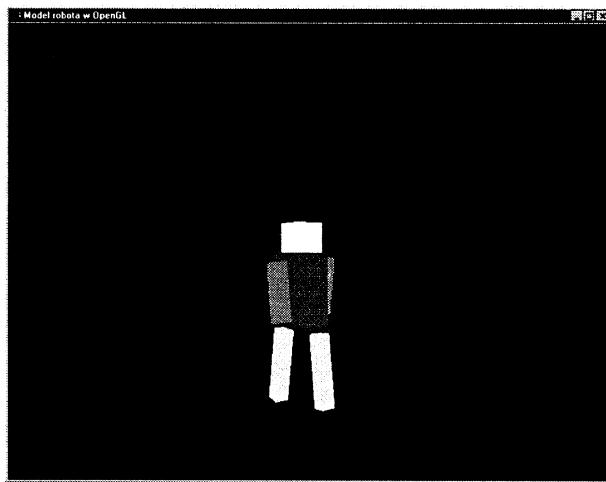
```

Całkiem długi kod, ale to dopiero początek naprawdę interesujących rzeczy. Funkcja DrawRobot() ilustruje sposób tworzenia i animacji hierarchicznego modelu. Szczególną uwagę należy zwrócić na zastosowanie funkcji `glPushMatrix()` i `glPopMatrix()` do umieszczenia poszczególnych elementów robota we właściwym miejscu lokalnego układu współrzędnych. Dobrym ćwiczeniem związanym z użyciem tych funkcji będzie rozbudowa robota o takie elementy jak dlonie czy stopy. Rysunek 5.13 prezentuje program animacji robota w działaniu.

## Rzutowanie

O rzutowaniu wspominano już w tej książce kilka razy, a nawet użyto go w przykładzie programu. Pora więc dokładniej zapoznać się z rzutowaniem w OpenGL. Jak już wspomniano OpenGL umożliwia dwa rodzaje rzutowania: ortograficzne (inaczej równoległe) oraz perspektywiczne.

**Rysunek 5.13.**  
*Program animacji  
robota w działaniu*



Rzutowanie pozwala określić bryłę widoku, która służy do osiągnięcia dwóch celów. Pierwszy z nich polega na określeniu płaszczyzn obcięcia, które umożliwiają ustalenie tego, jaka część trójwymiarowego świata jest w rzeczywistości widziana przez obserwatora. Dzięki temu obiekty znajdujące się poza bryłą widoku nie muszą być przetwarzane ani rysowane. Drugim celem określenia bryły widoku jest ustalenie sposobu rysowania obiektów. Zależy on od kształtu bryły widoku, który jest różny dla rzutowania ortograficznego i perspektywicznego.

Zanim wykonane jednak zostanie jakiekolwiek przekształcenie rzutowania należy upewnić się, że wybrany został stos macierzy rzutowania. Podobnie jak w przypadku macierzy modelowania służy do tego funkcja `glMatrixMode()`:

```
glMatrixMode(GL_PROJECTION);
```

W większości przypadków pożądane będzie także „wyczyszczenie” macierzy rzutowania za pomocą funkcji `glLoadIdentity()`, aby uniknąć akumulacji poprzednio wykonywanych przekształceń. W odróżnieniu od przekształceń modelowania, rzutowanie wykonywane jest z reguły pojedynczo.

Po wybraniu macierzy rzutowania można przystąpić do specyfikowania przekształcenia. Sposób wykonywania rzutowania omówiony zostanie najpierw dla rzutowania ortograficznego, a następnie dla rzutowania perspektywicznego, które będzie wykorzystywane najczęściej.

## Rzutowanie ortograficzne

Rzutowanie ortograficzne — zwane też równoległym — nie korzysta z perspektywy. Inaczej mówiąc, nie zmienia rozmiarów obiektów w zależności od ich odległości od kamery. Obiekty mają więc na ekranie zawsze taki sam rozmiar bez względu na to, czy znajdują się bliżej czy dalej. Chociaż rzutowanie ortograficzne nie tworzy tak realistycznego obrazu świata jak rzutowanie perspektywiczne, to jednak posiada wiele zastosowań. Tradycyjnie wykorzystywane jest w komputerowym wspomaganiu projektowania i dlatego właśnie zostało udostępnione w OpenGL. Rzutowanie ortograficzne wykorzystywane jest także w grach opartych na grafice dwuwymiarowej bądź izometrycznej.



Rzutowanie ortograficzne jest przydatne przy tworzeniu grafiki dwuwymiarowej, ale rzadko stosowane jest w grach, ponieważ tradycyjne metody tworzenia takiej grafiki umożliwiają uzyskanie większej liczby szczegółów. W przyszłości sytuacja ta może jednak ulec zmianie.

Przekształcenie ortograficzne w OpenGL wykonywane jest za pomocą funkcji `glOrtho()`:

```
glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
        GLdouble near, GLdouble far);
```

Parametry `left` i `right` definiują płaszczyznę obcięcia na osi x — `bottom` — i — `top` — na osi y, a parametry `near` i `far` określają odległość do płaszczyzn obcięcia na osi z. Parametry te określają więc bryłę widoku o kształcie prostopadłościanu.

Rzutowanie ortograficzne często używane jest podczas tworzenia grafiki dwuwymiarowej i dlatego też biblioteka GLU udostępnia dodatkową funkcję umożliwiającą rzutowanie ortograficzne w przypadku scen nie wykorzystujących współrzędnej z:

```
gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);
```

Parametry funkcji mają takie samo znaczenie jak dla funkcji `glOrtho()`. Taki sam rezultat jak za pomocą funkcji `glOrtho2D()` uzyskać można wywołując funkcję `glOrtho()`, jeśli jej parametry `near` i `far` będą miały wartości odpowiednio -1.0 i 1.0. Korzystając z funkcji `gluOrtho2D()` należy korzystać także z wersji funkcji `glVertex()` wymagającej podania jedynie dwóch parametrów określających współrzędne x i y. Zwykle w takim przypadku korzysta się także z układu współrzędnych odpowiadających wspólnym ekranowym.

## Rzutowanie perspektywiczne

Rzutowanie perspektywiczne pozwala uzyskać bardziej realistyczny obraz trójwymiarowego świata i dlatego korzysta się z niego częściej. W efekcie zastosowania przekształcenia perspektywicznego obiekty znajdujące się dalej od obserwatora są rysowane na ekranie jako mniejsze. Bryła widoku dla rzutowania perspektywicznego ma kształt ścisłego ostrosłupa skierowanego mniejszą podstawą w stronę obserwatora. Podczas rzutowania perspektywicznego OpenGL przekształca ten ostrosłup w prostopadłościan. Na skutek tego przekształcenia obiekty znajdujące się dalej od obserwatora są pomniejszane bardziej niż te, które znajdują się bliżej. Stopień pomniejszenia jest tym większy, im większa jest różnica rozmiarów obu podstaw ostrosłupa (jeśli obie podstawy są tych samych rozmiarów, to nie zostanie wprowadzona perspektywa i otrzyma się w efekcie rzutowanie ortograficzne).

Istnieje kilka sposobów określenia ostrosłupa rzutowania i tym samym rzutowania perspektywicznego. Najpierw omówiony zostanie poniższy:

```
void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
                GLdouble near, GLdouble far);
```

Parametry `left`, `right`, `bottom` i `top` wyznaczają płaszczyznę obcięcia zawierającą mniejszą z podstaw ostrosłupa, a parametry `near` i `far` określają odległość do mniejszej i większej podstawy ostrosłupa. Górnny, lewy wierzchołek mniejszej podstawy ma więc współrzędne (`left`, `top`, `-near`), a dolny, prawy (`right`, `bottom`, `-near`). Wierzchołki

większej podstawy ostrosłupa wyznacza się jako punkty przecięcia linii przechodzących przez punkt obserwatora i wierzchołki mniejszej podstawy z dalszą płaszczyzną obcięcia. Im bliżej więc znajduje się obserwator względem mniejszej z podstaw ostrosłupa, tym większa będzie druga z nich i tym samym większe wrażenie perspektywy.

Funkcja `glFrustum()` umożliwia zdefiniowanie perspektywy asymetrycznej, która użytkownika jest w pewnych, dość rzadkich przypadkach. Wyobrażanie sobie rzutowania perspektywicznego za pomocą ścieżki stożka widoku nie jest zbyt intuicyjne. Dlatego OpenGL jest zaopatrzony także w funkcję pozwalającą określić pole widzenia i na tej podstawie sam wyznacza ścieżkę ostrosłup widoku. Funkcja ta zdefiniowana jest następująco:

```
void gluPerspective(GLdouble fov, GLdouble aspect, GLdouble near, GLdouble far);
```

Parametr *fov* określa w stopniach kąt widzenia obserwatora w kierunku osi y. Parametr *aspect* specyfikuje proporcje pola widzenia, czyli stosunek szerokości do wysokości pola widzenia w kierunku osi x. Parametry *near* i *far* posiadają takie samo znaczenie jak w przypadku omówionych wcześniej funkcji rzutowania.

Jednym z problemów związanych z właściwym określeniem ścieżki ostrosłupa rzutowania perspektywicznego jest dobór szerokości jego podstaw (czyli szerokości pola widzenia). Nie ma optymalnego rozwiązania w tym zakresie, ponieważ właściwy wybór zależy od rodzaju zastosowania. Zwykle przyjmuje się za realistyczne kąty widzenia zbliżone do 90°. Jednak w praktyce dobór pola widzenia wymaga zawsze działania opartego na metodzie prób i błędów.

## Okno widoku

Niektóre z omówionych dotąd funkcji rzutowania posiadają związek z rozmiarami okna, w którym tworzona jest grafika (na przykład funkcja `gluPerspective` poprzez parametr *aspect*). Przekształcenie okienkowe wykonywane jest zawsze po rzutowaniu, dlatego też pora na jego omówienie. Macierz okienkowej nie modyfikuje się bezpośrednio, a jedynie określa rozmiary okna grafiki.

W przypadku okna grafiki istotne są jego rozmiary oraz orientacja. Określamy je za pomocą funkcji `glViewport()`:

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

Parametry *x* i *y* określają współrzędne lewego, dolnego narożnika okienka, a *width* i *height* — odpowiednio — jego szerokość i wysokość w pikselach.

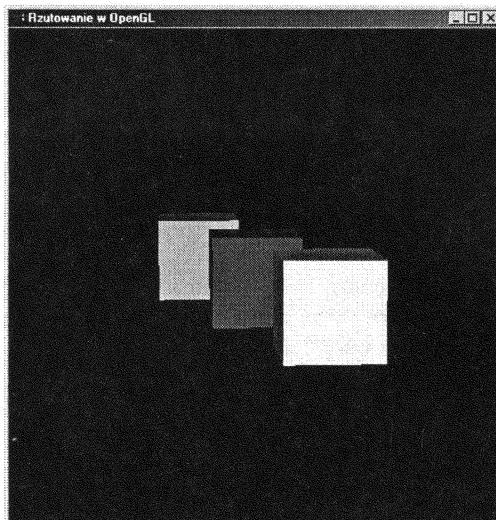
Gdy tworzy się kontekst grafiki, okienko widoku uzyskuje automatycznie rozmiary okna aplikacji, z którym związany jest kontekst. Rozmiar okna widoku musi być aktualizowany za każdym razem, gdy zmienia się rozmiar okna aplikacji. Chociaż zwykle rozmiary okna widoku odpowiadają rozmiarom okna aplikacji, to nie jest to oczywiście obowiązkowe. Grafika może być tworzona na przykład tylko w określonym regionie okna aplikacji i wtedy rozmiary okna grafiki są mniejsze.

## Przykład rzutowania

Aby ułatwić zrozumienie różnic pomiędzy omówionymi rodzajami rzutowania, przygotowany został prosty program, który pozwala oglądać tę samą scenę pokazaną z użyciem różnych rodzajów rzutowań. Po jego uruchomieniu grafika tworzona jest z wykorzystaniem rzutowania perspektywicznego. Za pomocą klawisza spacji, można jednak w każdej chwili wybierać pomiędzy rzutowaniem perspektywicznym (rysunek 5.14) i rzutowaniem ortograficznym (rysunek 5.15).

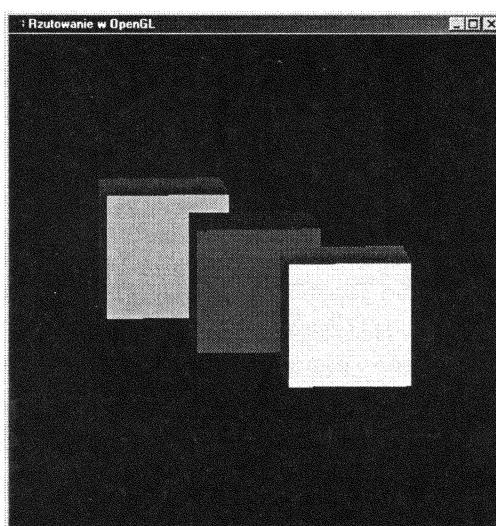
**Rysunek 5.14.**

*Rzutowanie  
perspektywiczne*



**Rysunek 5.15.**

*Rzutowanie  
ortograficzne*



Najistotniejszymi fragmentami tego programu są funkcje `ResizeScene()` i `UpdateProjection()`, których kod zaprezentowany został poniżej.

```
*****
ResizeScene()
Aktualizuje okno widoku i rzutowanie na podstawie rozmiarów okna aplikacji
*****
GLvoid ResizeScene(GLsizei width, GLsizei height)
{
    // unika dzielenia przez zero
    if (height==0)
    {
        height=1;
    }

    // aktualizuje rozmiary okna widoku
    glViewport(0, 0, width, height);

    // określa rzutowanie nie zmieniając jego rodzaju
    UpdateProjection();
} // koniec funkcji ResizeScene()
*****
UpdateProjection()

Określa rodzaj rzutowania. Jeśli toggle posiada wartość GL_TRUE, to zmieniany jest rodzaj rzutowania.
W przeciwnym razie stosuje wcześniej wybrany rodzaj rzutowania.
*****
void UpdateProjection(GLboolean toggle)
{
    static GLboolean s_usePerspective = GL_TRUE;

    // zmienia rodzaj rzutowania
    if (toggle)
        s_usePerspective = !s_usePerspective;

    // wybiera macierz rzutowania i resetuje ją
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // wybiera rodzaj rzutowania
    if (s_usePerspective)
    {
        // rzutowanie perspektywiczne
        glFrustum(-1.0, 1.0, -1.0, 1.0, 5, 100);
    }
    else
    {
        // rzutowanie ortograficzne
        glOrtho(-1.0, 1.0, -1.0, 1.0, 5, 100);
    }

    // przywraca macierz modelowania
    glMatrixMode(GL_MODELVIEW);
} // koniec funkcji UpdateProjection
```

# Wykorzystanie własnych macierzy przekształceń

Omówione dotąd funkcje przekształceń umożliwiają wykonywanie operacji na stosach macierzy bez konieczności bezpośredniego określania elementów macierzy. Pozwala to na wykonywanie wielu przekształceń bez znajomości algebraj macierzy. Jednak bardziej zaawansowane efekty wymagać będą bezpośrednich manipulacji macierzami. Przedstawiony teraz zostanie sposób ładowania elementów macierzy, mnożenia macierzy znajdującej się na szczycie stosu o inną macierz oraz przykład zastosowania własnych macierzy.

## Ładowanie elementów macierzy

Macierze OpenGL posiadają rozmiary  $4 \times 4$ , a ich elementy uporządkowane są w kolumnach, co ilustruje rysunek 5.16.

**Rysunek 5.16.**  
Macierz OpenGL

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

Macierze OpenGL można zadeklarować w programie jako dwuwymiarowe tablice, ale rozwiązanie takie posiada dość poważną wadę. W językach C i C++ elementy tablic uporządkowane są wierszami. I tak — aby odwołać się do elementu  $m_3$  macierzy pokazanej na rysunku 5.16, zamiast intuicyjnie określić go jako `matrix[3][0]`, trzeba podać `matrix[0][3]`. Takie rozwiązanie będzie przyczyną wielu błędów, dlatego lepiej jest w praktyce wykorzystywać tablice jednowymiarowe do reprezentacji macierzy OpenGL. Tablica taka będzie posiadać 16 elementów, a  $n$ -ty element będzie odpowiadać elementowi  $m_n$  macierzy z rysunku 5.16.

Aby na przykład wyspecyfikować macierz jednostkową (czego w praktyce nie trzeba nigdy robić mając do dyspozycji funkcję `glLoadIdentity`), można zdefiniować poniższą tablicę:

```
GLfloat identity[16] = { 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
    0.0, 0.0, 0.0, 1.0};
```

Po zdefiniowaniu tablicy należy załadować jej elementy do macierzy korzystając z funkcji `glLoadMatrix()` posiadającej dwie wersje:

```
void glLoadMatrixd(const GLdouble *matrix);
void glLoadMatrixf(const GLfloat *matrix);
```

Jedyną różnicą pomiędzy nimi jest typ parametru. Pierwsza z nich ładuje elementy typu `double`, a druga typu `float`. Wywołanie funkcji `glLoadMatrix()` powoduje, że elementy macierzy znajdującej się na szczycie bieżącego stosu macierzy zastępowane są elementami tablicy `matrix`. Należy więc pamiętać, że ładując elementy macierzy traci się informację znajdująjącą się dotąd w macierzy umieszczonej na szczycie stosu.

## Mnożenie macierzy

Oprócz funkcji ładowania elementów do macierzy znajdującej się na szczytce stosu dostępna jest także funkcja umożliwiająca pomnożenie jej zawartości o inną macierz. Macierz tę określa się w ten sam sposób co funkcję `glLoadMatrix()` i następnie wywołuje się jedną z wersji funkcji `glMultMatrix()`:

```
void glMultMatrixd(const GLdouble *matrix);
void glMultMatrixf(const GLfloat *matrix);
```

Parametr `matrix` jest tablicą 16 elementów. Jeśli macierz znajdująca się na szczytce stosu oznaczy się jako  $M_A$ , a macierz zdefiniowaną za pomocą tablicy `matrix` jako  $M_B$ , to na skutek wywołania funkcji `glMultMatrix()` uzyska się na szczytce bieżącego stosu macierzy wynik działania  $M_A \cdot M_B$ . Przypomnieć należy w tym miejscu, że mnożenie macierzy nie jest przemienne, a więc porządek mnożenia jest istotny i w większości przypadków wynik działania  $M_A \cdot M_B$  będzie różny od wyniku działania  $M_B \cdot M_A$ .

## Przykład zastosowania macierzy definiowanych

Program, który dołączony został na dysku CD jako ilustracja do pierwszego rozdziału książki, zawiera przykład zastosowania macierzy definiowanych bezpośrednio przez programistę. W tym przypadku służy ona do tworzenia cieni podczas animacji sześcianu. Interesujący tu fragment programu zaprezentowany został poniżej:

```
GLfloat shadowMatrix[16] = { lightPos[1], 0.0, 0.0, 0.0, -lightPos[0], 0.0,
                             -lightPos[2], -1.0, 0.0, 0.0, lightPos[1], 0.0,
                             0.0, 0.0, 0.0, lightPos[1] };

...
// rzutowanie sześcianu za pomocą macierzy cieni
glMultMatrixf(shadowMatrix);
DrawCube();
```

Macierz ta definiuje rzut wierzchołków na płaszczyznę  $y = 0$ . Jeśli wybrany zostanie w programie kolor czarny (oraz zastosowane zostanie łączenie alfa i bufor powielania, których omówienie wykracza poza tematykę tego rozdziału), to uzyska się w ten sposób możliwość tworzenia cieni rysowanych obiektów. Macierz cieni wykorzystuje się do pomnożenia o nią zawartości macierzy znajdującej się na szczytce stosu macierzy modelowania za pomocą funkcji `glMultMatrix()`. Nie stosuje się w tym przypadku funkcji `glLoadMatrix()`, by zachować przekształcenia reprezentowane na stosie macierzy modelowania używane do tworzenia sceny animacji.

Własne macierze powinno się stosować jedynie w przypadku specjalizowanych zastosowań. W pozostałych przypadkach należy zawsze korzystać z funkcji przekształceń udostępnianych przez OpenGL, ponieważ ich implementacja może korzystać ze wsparcia sprzętowego, do którego programista nie ma bezpośredniego dostępu.

## Podsumowanie

W rozdziale tym przedstawione zostały sposoby wykonywania przekształceń obiektów grafiki trójwymiarowej oraz określania widoku sceny za pomocą rzutowania. Omówione zostały macierze modelowania i rzutowania oraz działania wykonywane na tych macierzach za pomocą funkcji OpenGL i macierzy definiowanych przez programistę. Użyte w ten sposób zostały podstawowe narzędzia pozwalające tworzyć i animować obiekty trójwymiarowego świata gry.

## Rozdział 6.

# Kolory, łączenie kolorów i oświetlenie

Świat pozbawiony kolorów byłby nudny i wywoływałby depresję. Podobnie grafika tworzona na ekranie komputerów jedynie z użyciem odcienni szarości byłaby monotonna i nieciekawa. Na szczęście OpenGL dysponuje środkami umożliwiającymi tworzenie barwnych światów gry.

Kolor nie jest jedynym sposobem zwiększenia atrakcyjności tworzonej grafiki. OpenGL umożliwia manipulację oświetleniem obiektów oraz tworzenie wielu innych efektów.

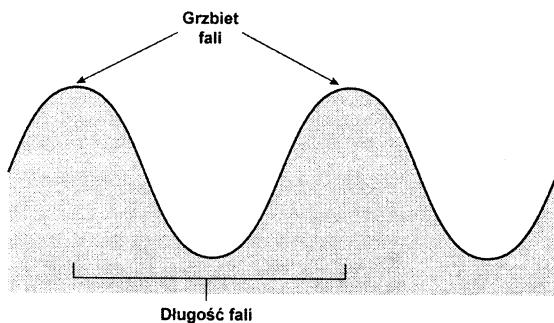
Rozdział ten rozpoczyna się omówieniem natury kolorów w świecie rzeczywistym i sposobów ich reprezentacji w grafice komputerowej. Następnie przedstawione zostaną różne rodzaje oświetlenia obiektów. Rozdział kończy omówienie łączenia kolorów i przezroczystości.

W rozdziale tym przedstawione zostaną:

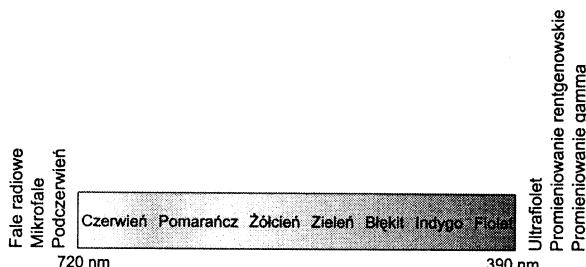
- ◆ kolory w OpenGL;
- ◆ cienie;
- ◆ oświetlenia w OpenGL;
- ◆ źródła światła;
- ◆ łączenie kolorów i przezroczystość.

## Czym są kolory?

W tym miejscu trzeba przypomnieć sobie nieco wiadomości z fizyki. Światło składa się z fotonów, cząstek poruszających się w różnych kierunkach i drgających z różną częstotliwością, dlatego też światło można interpretować zarówno jako cząstkę, jak i jako falę, co stanowi treść tak zwanego *dualizmu korpuskularno-fałowego*. Rozpatrując światło jako falę można opisać ją za pomocą długości fali, co pokazuje rysunek 6.1.

**Rysunek 6.1.***Długość fali światła*

Widzialny zakres światła tworzą fale o długości od 390 nanometrów (nm) do 720 nm. W przedziale tym mieści się pełne spektrum tęczy: począwszy od fioletu, przez błękit, zieleń, żółcień, pomarańcz aż do czerwieni. Rysunek 6.2 przedstawia widzialne spektrum światła.

**Rysunek 6.2.***Widzialne spektrum światła*

Pokazane na nim kolory stanowią jedynie fragment zbioru milionów barw i odcieni, które rozróżnia ludzkie oko. Siatkówka oka składa się z milionów specjalnych komórek, których działanie przypomina zachowanie błony fotograficznej. Podrażnione światłem o określonej długości fali wysyłają impulsy do mózgu, w którym tworzony jest obraz.

Istnieją trzy rodzaje takich komórek: reagujące na kolor zielony, czerwony i niebieski. W zależności od długości fali oglądanego światła wysyłają one do mózgu impulsy o różnym natężeniu, co pozwala rozróżniać wiele odcieni kolorów.

## Kolory w OpenGL

Opisany model widzenia kolorów można rozszerzyć także o prezentację kolorów na ekranie monitora. Kolorowe piksele powstają na nim na skutek świecenia z różną intensywnością czerwonych, zielonych i niebieskich drobin fosforu. W grafice komputerowej intensywność tych składowych opisuje się za pomocą *wartości modelu RGB*. Czasami do składowych tych dodaje się jeszcze jedną — *współczynnik alfa*, tworząc w ten sposób model *RGBA* (współczynnik alfa omówiony zostanie w dalszej części tego rozdziału przy przedstawieniu zasad łączenia kolorów). Kolory na ekranie można także opisywać za pomocą *modelu indeksowego*, w którym każdemu pikselowi na ekranie przyporządkowany jest indeks koloru w tablicy zwanej *mapą kolorów*. Z każdym z indeksów można związać odpowiednią wartość składowej czerwonej, zielonej i niebieskiej.

## Głębia koloru

*Głębia koloru* określa maksymalną liczbę kolorów, które może posiadać piksel i zależy od rozmiaru *bufora koloru*. Rozmiar ten może wynosić 4, 8, 16 i więcej bitów. 8-bitowy bufor koloru może przechowywać informację o jednym z  $2^8$ , czyli 256 kolorów. Dostępne obecnie karty graficzne dysponują zwykle buforem kolorów o rozmiarze 8, 16, 24 lub 32 bitów. Dostępne liczby kolorów dla poszczególnych rozmiarów bufora koloru przedstawia tabela 6.1.

Tabela 6.1. Najczęściej stosowane głębie kolorów

Głębia kolorów	Charakterystyka
8	256 kolorów dostępnych w palecie indeksowanej od 0 do 255
16	65 536 kolorów, składowe modelu RGB opisane są przez (odpowiednio) 5, 6 i 5 bitów; czasami wykorzystywane jest tylko 15 bitów bufora koloru i wtedy każda ze składowych modelu RGB opisywana jest za pomocą 5 bitów
24	16 777 216 kolorów, każda ze składowych modelu RGB opisana za pomocą 8 bitów
32	Tak samo jak w przypadku głębi 24-bitowej, ale poprawiona efektywność przez zastosowanie architektury 32-bitowej

## Sześciian kolorów

OpenGL opisuje kolory za pomocą intensywności ich składowej czerwonej, zielonej i niebieskiej. Intensywność poszczególnych składowych może przyjmować wartości z przedziału od 0 do 1. Rysunek 6.3 prezentuje za pomocą sześciianu kolorów to, w jaki sposób kombinacje różnej intensywności poszczególnych składowych tworzą spektrum kolorów. Na przykład wartości  $R = 1.0$ ,  $G = 0.0$ ,  $B = 0.0$  pozwalają uzyskać czerwień o największej intensywności. Wartości  $R = 1.0$ ,  $G = 1.0$ ,  $B = 0.0$  opisują natomiast największe natężenie koloru żółtego. Jeśli wszystkie składowe posiadają wartość 0.0, to otrzymujemy się kolor czarny, a jeśli wartość 1.0, to kolor biały. Jeśli wszystkie składowe mają taką samą wartość, to otrzymujemy się odcień szarości, którego intensywność zależy od tego, w jakim miejscu przedziału od 0.0 do 1.0 znajduje się ta wartość. Sześciian kolorów może być pomocny w określaniu składowych żadanego koloru. Osie układu reprezentują składową czerwoną, zieloną i niebieską. Oddalenie się od danej osi zwiększa udział pozostałych składowych w tworzonym kolorze.

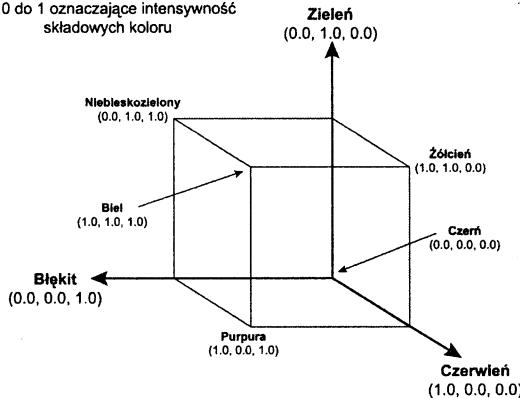
## Model RGBA w OpenGL

Specyfikując kolor w modelu RGBA przekazuje się funkcji `glColor*`() wartości składowej czerwonej, zielonej i niebieskiej. Funkcja `glColor*`() posiada kilkanaście wersji różniących się przyrostkami dodawanymi do jej nazwy. Do najczęściej używanych wersji należą:

```
void glColor3f(GLfloat r, GLfloat g, GLfloat b);
void glColor4f(GLfloat r, GLfloat g, GLfloat b, GLfloat a);
void glColor3fv(const GLfloat *v);
void glColor4fv(const GLfloat *v);
```

**Rysunek 6.3.**  
Sześcian kolorów

(R, G, B) = wartości z przedziału od 0 do 1 oznaczające intensywność składowych koloru



Pierwsza z funkcji wybiera bieżący kolor na podstawie wartości składowej czerwonej reprezentowanej przez parametr *r*, zielonej *g* i niebieskiej *b*. Druga z nich posiada dodatkowy parametr *a* reprezentujący wartość współczynnika alfa określającą sposób łączenia kolorów (omówiona zostanie w dalszej części tego rozdziału). Wartości wszystkich parametrów obu funkcji muszą należeć do przedziału od 0.0 do 1.0. Wartość 0.0 odpowiada oczywiście najmniejszej intensywności danej składowej, a wartość 1.0 największej. Kolejna para funkcji odpowiada już omówionej parze (z tą różnicą, że parametry umieszczane są w tablicy, której wskaźnik przekazywany jest funkcji). Taki sposób przekazywania parametrów jest często używany w OpenGL i będzie pojawiać się podczas omawiania kolejnych funkcji.

Aby wybrać kolor zielony za pomocą pierwszej z omówionych funkcji, należy wywołać ją w poniższy sposób:

```
void glColor3f(0.0f, 1.0f, 0.0f);
```

Aby wybrać kolor żółty, należy wywołać funkcję następująco:

```
void glColor3f(1.0f, 1.0f, 0.0f);
```

Jak łatwo zauważyc, podając wartości parametrów funkcji `glColor*`() można korzystać z pomocy w postaci sześcianu kolorów. I tak na przykład nadając wszystkim parametrom funkcji wartość 0.5 uzyskuje się kolor szary:

```
void glColor3f(0.5f, 0.5f, 0.5f);
```

Model kolorów RGBA będzie jeszcze wykorzystywany jako podstawowy model kolorów tworzący grafikę przy użyciu OpenGL. Warto jednak zapoznać się także z modelem indeksowanym.

## Model indeksowany w OpenGL

W czasach systemu operacyjnego DOS programiści zmuszeni byli do tworzenia *palety kolorów* w postaci tablicy opisującej kolory używane w programie. Analogia z paletą malarza polegała na tym, że choć dostępna była ograniczona liczba kolorów, to możliwe było jednak tworzenie nowych poprzez łączenie już istniejących. Palety kolorów używane są nadal w programach korzystających z 8-bitowej głębi koloru.

W modelu indeksowanym OpenGL tworzy *mapę kolorów*, której rozmiar określa liczbę jednocześnie dostępnych kolorów. Sposób działania tej mapy ilustruje rysunek 6.4. Rozmiar mapy kolorów jest zawsze potęgą liczby 2 i najczęściej wynosi 256 bądź 4096 (w zależności od możliwości sprzętowych). W modelu indeksowanym wielu pikselom przypisany jest kolor opisywany przez pozycję mapy o danym indeksie. Jeśli pozycja ta zostanie zmodyfikowana, to automatycznie zmieni się kolor wszystkich tych pikseli.

**Rysunek 6.4.**

Przykład  
mapy kolorów

Indeks	Intensywność		
	R	G	B
0	0	0	0
1	1	1	1
2	3	3	3
3	5	4	6
4	7	6	8
5	8	7	9
6	9	8	10
7	10	11	15
8	20	19	20
9	21	22	25
10	22	23	30
11	30	40	35
•	•	•	•
•	•	•	•
255	255	255	255

Aby wybrać określoną pozycję mapy, używa się funkcji `glIndex*`(). Jej najczęściej wykorzystywane wersje posiadają następujące prototypy:

```
void glIndexf(GLfloat c);
void glIndexfv(const GLfloat *c);
```

Obie wersje funkcji wybierają pozycję mapy o indeksie *c*. Druga z nich pobiera jednak wartość indeksu z tablicy, której wskaźnik został jej przekazany. Tablica ta zawiera tylko jedną wartość typu `GLfloat` reprezentującą wartość indeksu.

Z pomocą funkcji `glClearIndex()` można wypełnić cały bufor kolorów pozycją mapy o podanym indeksie:

```
void glClearIndex(GLfloat cindex);
```

Domyślnie bufor kolorów wypełniany jest pozycją mapy o indeksie 0.0.

## Cieniowanie

Dotychczas przy rysowaniu podstawowych elementów grafiki OpenGL wypełnienie stanowił jednolity, pojedynczy kolor. A co stanie się, jeśli podczas definiowania kolejnych wierzchołków takiego elementu wybierze się różne kolory?

Na pytanie to można odpowiedzieć posługując się przykładem odcinka łączącego dwa wierzchołki o różnych kolorach. Dla uproszczenia należy przyjąć, że pierwszy z nich jest w kolorze czarnym, a drugi jest biały. Jaki będzie kolor łączącego je odcinka? Można to ustalić posługując się modelem cieniowania.

Cieniowanie może być *płaskie* lub *gładkie*. Cieniowanie płaskie wykonywane jest za pomocą pojedynczego koloru. Zwykle jest to kolor ostatniego wierzchołka (jedyne w przypadku trybu rysowania wielokątów o dowolnej liczbie wierzchołków wybieranego za pomocą stałej GL\_POLYGON jest to kolor pierwszego z wierzchołków). Cieniowanie gładkie zwane też *cieniowaniem Gouraud* stosuje natomiast interpolację koloru pikseli odcinka.

Jeśli dla wybranego przed chwilą odcinka zastosuje się cieniowanie płaskie, to będzie on miał kolor biały, ponieważ ostatni z wierzchołków odcinka ma taki kolor. Natomiast w przypadku cieniowania gładkiego kolor pikseli będzie zmieniać się począwszy od koloru czarnego pierwszego wierzchołka poprzez coraz jaśniejsze odcienie szarości aż do koloru białego drugiego z wierzchołków. Efekt ten pokazuje rysunek 6.5.

**Rysunek 6.5.**

Gładkie cieniowanie  
odcinka łączącego  
wierzchołek  
w kolorze czarnym  
z wierzchołkiem  
w kolorze białym

Pierwszy  
wierzchołek

Drugi  
wierzchołek

Zasada działania cieniowania gładkiego, która pokazana została na przykładzie odcinka, stosowana jest także w przypadku wielokątów. Na przykład trójkąt, którego każdy z wierzchołków posiada inny kolor, będzie narysowany w ten sposób, że kolor pikseli wypełniających trójkąt będzie zmieniał się stopniowo w zależności od odległości od jego wierzchołków.

OpenGL udostępnia funkcję `glShadeMode()` pozwalającą wybierać bieżący model cieniowania. Funkcja ta posiada następujący prototyp:

```
void glShadeMode(GLenum mode);
```

Parametrowi *mode* nadaje się wartość `GL_SMOOTH`, aby wybrać cieniowanie gładkie i `GL_FLAT` w przypadku cieniowania płaskiego. Domyślnie wybrany jest model cieniowania gładkiego.

Poniżej zaprezentowany został fragment kodu rysujący trójkąt z wykorzystaniem cieniowania gładkiego:

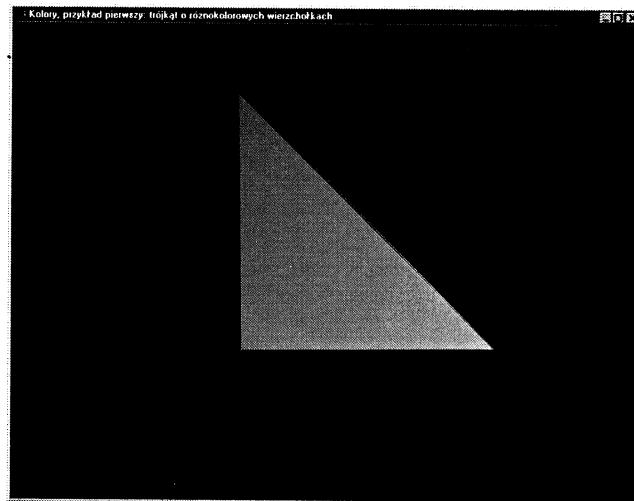
```
// wybór modelu cieniowania
glShadeMode(GL_SMOOTH);

// pewien kod ...

// rysuje trójkąt
glBegin(GL_TRIANGLES);
    glColor3f(1.0f, 0.0f, 0.0f);           // wierzchołek w kolorze czerwonym
    glVertex3f(-10.0f, -10.0f, -5.0f);
    glColor3f(0.0f, 1.0f, 0.0f);           // wierzchołek w kolorze zielonym
    glVertex3f(20.0f, -10.0f, -5.0f);
    glColor3f(0.0f, 0.0f, 1.0f);           // wierzchołek w kolorze niebieskim
    glVertex3f(-10.0f, 20.0f, -5.0f);
glEnd();
```

Rezultat jego działania pokazuje rysunek 6.6. Czerwień, zieleń i błękit każdego z wierzchołków zmieniają się stopniowo dla pikseli leżących coraz dalej od wierzchołka w danym kolorze. Piksele leżące w środku trójkąta mają kolor szary, a więc kolory poszczególnych wierzchołków posiadają tam jednakową intensywność.

**Rysunek 6.6.**  
*Trójkąt  
o wierzchołkach  
w podstawowych  
kolorach  
modelu RGB*



## Oświetlenie

W ten sposób wywód dotarł do jednego z najistotniejszych zagadnień związanych z tworzeniem trójwymiarowej grafiki — oświetlenia. Element ten odgrywa decydującą rolę w osiągnięciu realizmu wirtualnego świata gry. Dotychczas omówione zostały sposoby tworzenia obiektów, poruszania nimi, nadawania im kolorów i cieniowania. Teraz można spróbować nadać ich wyglądowi jeszcze więcej realistycznych elementów stosując różne rodzaje materiałów obiektów i ich oświetlenia.

## Oświetlenie w OpenGL i w realnym świecie

Można teraz zastanowić się nad tym, w jaki sposób oświetlenie wpływa na wygląd obiektów w rzeczywistym świecie. Obraz widziany w oku powstaje na skutek podrażnienia siatkówki przez fotony. Fotony te muszą mieć określone źródło, a do oka trafiają na skutek odbicia przez oglądane obiekty. Nie wszystkie fotony, które padają na obiekt zostają odbite, ponieważ część ich zostaje pochłonięta. Stopień odbicia padającego światła przez obiekt zależy od materiału z jakiego jest on wykonany. Obiekty, które widziane są jako połyskujące, odbijają większość światła. Natomiast obiekty, które pochłaniają znaczną część padającego na nie światła lub odbijają ją w takim kierunku, że nie trafia ono do oka, odbierane są jako ciemniejsze.

OpenGL wyznacza efekt oświetlenia obiektów w podobny sposób. Kolor obiektu zależy od stopnia odbicia przez niego składowej czerwonej, zielonej i niebieskiej padającego światła. Stopień ten OpenGL określa na podstawie rodzaju materiału, na który pada światło.

OpenGL umożliwia modelowanie oświetlenia za pomocą czterech komponentów:

- ◆ **światła otoczenia** — światło to przedstawia się tak, jakby nie posiadało źródła (zostało rozproszone w takim stopniu, że ustalenie jego źródła nie jest możliwe);
- ◆ **światła rozproszonego** — światło to pada z określonego kierunku, ale zostaje odbite przez obiekt równomiernie we wszystkich kierunkach, dzięki czemu niezależnie od położenia obserwatora obiekt wydaje się oświetlony równomiernie;
- ◆ **światła odbicia** — światło pada z określonego kierunku i odbijane jest także w jednym kierunku tworząc efekt odbicia;
- ◆ **światła emisji** — obiekty wydzielające światło emisji posiadają większą intensywność kolorów (światło emisji nie ma wpływu na wygląd pozostałych obiektów, a inne źródła światła nie mają wpływu na światło emisji danego obiektu).

## Materiały

OpenGL wyznacza kolor obiektu na podstawie stopnia odbicia przez jego materiał składowej czerwonej, zielonej i niebieskiej światła. Na przykład powierzchnia w kolorze zielonym odbija w całości składową zieloną światła, a pochłania składową czerwoną i niebieską. Jeśli więc będzie padać na nią światło posiadające wyłącznie składową czerwoną, to będzie można zobaczyć powierzchnię w kolorze czarnym, ponieważ oświetlające ją światło zostanie pochłonięte w całości. Natomiast jeśli oświetlona zostanie zielona powierzchnia białym światłem, to będzie ona miała kolor zielony, ponieważ składowa zielona zostanie odbita, a składowe niebieska i czerwona zostaną pochłonięte. Podobnie będzie się działało, jeśli oświetlona zostanie ona za pomocą źródła emitującego jedynie składową zieloną — powierzchnia będzie nadal posiadać kolor zielony, ponieważ odbije w całości składową zieloną padającego na nią światła.

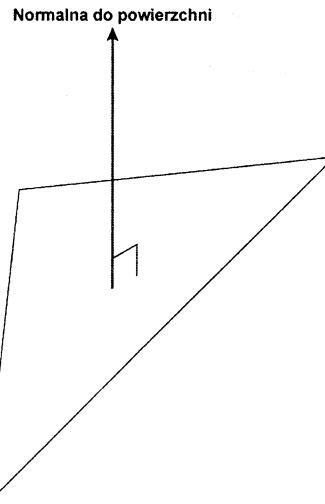
Materiały opisuje się za pomocą tych samych właściwości co światło. Właściwości te opisują stopień odbicia poszczególnych rodzajów światła przez dany materiał. Jedne materiały będą na przykład dobrze odbijać jedynie światło otoczenia, a pochłaniać światło rozproszone i światło odbicia. Inne będą odbijać światło odbicia stwarzając wrażenie połysku, a absorbować pozostałe rodzaje światła. Kolor materiału zależy od stopnia odbicia światła otoczenia i światła rozproszonego, który zwykle jest jednakowy. Aby natomiast światło odbicia zachowywało kolor źródła, przyjmuje się, że obiekty odbijają zawsze światło szare lub białe. Należy zauważyć, że przy oświetlaniu na przykład poleskującego obiektu w kolorze błękitnym, większość jego powierzchni wydawać się będzie błękitna, ale refleksy świetlne będą miały kolor biały.

## Normalne

Zanim będzie można przejść do omówienia kolejnych zagadnień związanych z oświetleniem obiektów, należy najpierw omówić *normalne do powierzchni* i sposób ich obliczania. Pojęcie normalnej ilustruje rysunek 6.7.

**Rysunek 6.7.**

*Normalna  
powierzchni*



Normalna do powierzchni jest wektorem prostopadłym do danej powierzchni. Wyznaczenie normalnej do powierzchni jest niezbędne w celu ustalenia odcienia koloru, którym wypełniony zostanie wielokąt oświetlony pod określonym kątem. Jak wiadomo, źródła światła (w zależności od ich rodzaju) promieniują w określonym kierunku lub we wszystkich kierunkach. Promienie światła padają więc na obiekt pod pewnym kątem. Znając wartość tego kąta oraz kąta odbicia światła można, biorąc pod uwagę także rodzaj materiału i rodzaj światła, wyznaczyć kolor danej powierzchni. Wyznaczenie normalnej ma więc na celu określenie kąta padania światła na daną powierzchnię.

Gdy jednak działania ograniczy się jedynie do wyznaczenia oświetlenia powierzchni na podstawie normalnej, to uzyska się mało realistyczny efekt wypełnienia powierzchni jednolitym kolorem. Dlatego też normalną, a tym samym kolor powierzchni, należy wyznaczać w odniesieniu do wszystkich wierzchołków wielokąta. Wielokąt będzie natomiast wypełniany przy użyciu modelu cieniowania gładkiego, co pozwoli uzyskać bardziej realistyczny efekt. Na początku jednak — dla uproszczenia — można ograniczyć się do wyznaczania kolorów na podstawie normalnej do powierzchni.

## Obliczanie normalnych

Do obliczania normalnych przydatne będą informacje przedstawione w rozdziale 3. Pokazano w nim sposób wyznaczania iloczynu wektorowego oraz wspomniano o jego zastosowaniu do wyznaczania zderzeń obiektów, oświetlenia obiektów i innych właściwości fizycznych. Należy więc przypomnieć jedynie o tym, że równanie pozwalające wyznaczyć iloczyn wektorów  $A$  i  $B$  wygląda następująco:

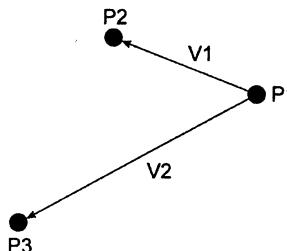
$$A * B = (A_y * B_z - A_z * B_y, A_z * B_x - A_x * B_z, A_x * B_y - A_y * B_x)$$

Aby wyznaczyć normalną jako wynik iloczynu wektorowego, niezbędne będą więc dwa wektory opisujące płaszczyznę. Skąd je wziąć? Należy przypomnieć w tym miejscu, że do wyznaczenia płaszczyzny potrzebne są trzy punkty. Dla których trzech wierzchołków można wyznaczyć płaszczyznę oraz narysować leżący na niej trójkąt.

Jeśli zatem posiada się współrzędne trzech wierzchołków  $P1$ ,  $P2$  i  $P3$ , można wyznaczyć normalną płaszczyzny, do której należą. W tym celu należy zdefiniować dwa wektory  $V1$  i  $V2$ , pierwszy wychodzący z wierzchołka  $P1$  do wierzchołka  $P2$ , a drugi z wierzchołka  $P2$  do wierzchołka  $P3$ . Ilustruje to rysunek 6.8.

#### Rysunek 6.8.

Definicja wektorów  
 $V1$  i  $V2$



Teraz można wyznaczyć już normalną do powierzchni jako iloczyn wektorów  $V1$  i  $V2$ . Obliczenie normalnej można zaimplementować za pomocą następującej funkcji:

```
void CrossProduct(float point1[3], float point2[3], float point3[3], float normal[3])
{
    float vector1[3], vector2[3];

    // wyznacza wektory potrzebne do obliczenia normalnej
    vector1[0] = point1[0] - point2[0];
    vector1[1] = point1[1] - point2[1];
    vector1[2] = point1[2] - point2[2];

    vector2[0] = point2[0] - point3[0];
    vector2[1] = point2[1] - point3[1];
    vector2[2] = point2[2] - point3[2];

    // wyznacza iloczyn wektorowy i umieszcza wynik w tablicy normal[3]
    normal[0] = vector1[1]*vector2[2] - vector1[2]*vector2[1];
    normal[1] = vector1[2]*vector2[0] - vector1[0]*vector2[2];
    normal[2] = vector1[0]*vector2[1] - vector1[1]*vector2[0];
}
```

Funkcja ta oblicza normalną do płaszczyzny wyznaczonej przez trzy wierzchołki. Wierzchołki te przekazuje się jako parametry `point1`, `point2` i `point3`, a wynik jej działania umieszczany jest w ostatnim z parametrów, czyli tablicy `normal`. Jeśli przy obliczaniu normalnej dysponuje się już wyznaczonymi wcześniej wektorami, to można skorzystać z uproszczonej wersji funkcji:

```
void CrossProduct(float vector1[3], float vector2[3], float normal[3])
{
    // wyznacza iloczyn wektorowy i umieszcza wynik w tablicy normal[3]
    normal[0] = vector1[1]*vector2[2] - vector1[2]*vector2[1];
    normal[1] = vector1[2]*vector2[0] - vector1[0]*vector2[2];
    normal[2] = vector1[0]*vector2[1] - vector1[1]*vector2[0];
}
```

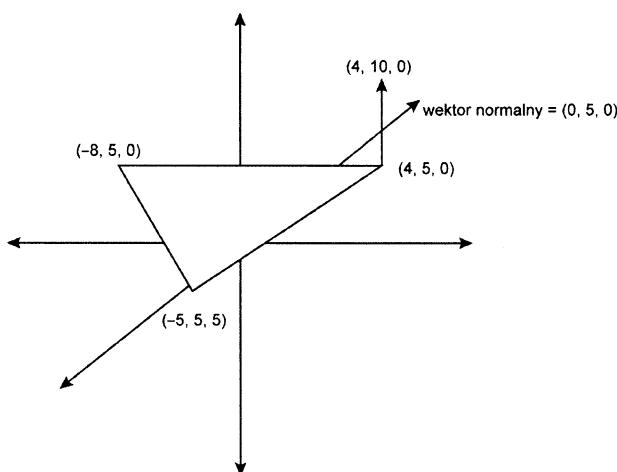
Powыższe funkcje nadają się do obliczeń normalnej w ogólnym przypadku. Jednak wyznaczanie oświetlenia wymaga dość złożonych obliczeń i dlatego pożądane są w tym przypadku wszelkiego rodzaju uproszczenia. Wkrótce omówione zostanie jedno z nich — normalne jednostkowe.

## Użycie normalnych

Normalną zapisuje się jak każdy inny wektor. Na przykład dla wektora leżącego na płaszczyźnie  $xz$  (którego wszystkie wierzchołki mają taką samą współrzędną  $y$ ) można narysować linię prostopadłą do płaszczyzny trójkąta w wierzchołku o współrzędnych  $(4, 5, 0)$ . Linia ta będzie przeходить na przykład także przez punkt o współrzędnych  $(4, 10, 0)$  i będzie wyznaczać prostopadłą i normalną trójkąta. Aby określić normalną, nie trzeba podawać obu punktów. Wystarczy odjąć od siebie ich odpowiednie współrzędne, aby uzyskać wektor normalny trójkąta  $(0, 5, 0)$ . Przykład ten ilustruje rysunek 6.9.

**Rysunek 6.9.**

Prostopadła  
wyznacza normalną  
do płaszczyzny,  
a dwa jej punkty  
pozwalały wyznaczyć  
wektor normalny



OpenGL udostępnia funkcję `glNormal3f()`, która pozwala wyznaczyć normalną w kolejnym rysowanym wierzchołku lub normalną do płaszczyzny wyznaczanej przez zbiór rysowanych wierzchołków. Jej prototyp wygląda następująco:

```
void glNormal3f(GLfloat nx, GLfloat ny, GLfloat nz);
```

Parametry funkcji reprezentują składowe wektora normalnego do płaszczyzny. Funkcję tę wywołuje się przed utworzeniem wierzchołków:

```
glBegin(GL_TRIANGLES)
    glNormal3f(0.0f, 1.0f, 0.0f);
    glVertex3f(0.0f, 5.0f, 0.0f);
    glVertex3f(4.0f, 5.0f, 0.0f);
    glVertex3f(0.0f, 5.0f, 4.0f);
glEnd();
```

Powyższy fragment kodu rysuje trójkąt znany z rysunku 6.9. Funkcja `glNormal3f()` definiuje wektor normalny o kierunku i zwrocie zgodnym z osią  $y$ , prostopadły do płaszczyzny trójkąta. Aby teraz dodać oświetlenie trójkąta, OpenGL będzie wykonywał wszystkie związane z tym obliczenia korzystając ze zdefiniowanego wektora normalnego.

Wektor normalny można zdefiniować także za pomocą tablicy zawierającej jego składowe i funkcji `glNormal3fv()`. Na przykład:

```
GLfloat normalVector = { 0.0f, 1.0f, 0.0f };

...
glBegin(GL_TRIANGLES)
    glNormal3fv(normalVector);
    glVertex3f(0.0f, 5.0f, 0.0f);
    glVertex3f(4.0f, 5.0f, 0.0f);
    glVertex3f(0.0f, 5.0f, 4.0f);
glEnd();
```

Rezultat wykonania tego fragmentu kodu będzie identyczny jak w poprzednim przypadku. Jedyna różnica polega na przekazaniu składowych wektora normalnego za pomocą tablicy zamiast każdej oddzielnie. Wykorzystanie tablicy może być szczególnie przydatne, gdy dysponuje się już zbiorami danych definiujących normalne.

## Normalne jednostkowe

Aby przeprowadzić obliczenia związane z wyznaczeniem oświetlenia obiektu, OpenGL musi przekształcić normalne przekazane za pomocą funkcji `glNormal3f()` w *normalne jednostkowe*. Normalne jednostkowe są po prostu wektorami normalnymi o długości równej 1.

W rozdziale 3. pokazano, że długość wektora  $A$  można wyznaczyć za pomocą następującego równania:

$$|A| = \sqrt{A_x^2 + A_y^2 + A_z^2}$$

W ten sposób można obliczyć oczywiście także długość dowolnego wektora normalnego. Aby w procesie *normalizacji* uzyskać wektor jednostkowy, trzeba każdą z jego składowych podzielić przez długość wektora. Uzyskany wektor posiada taki sam kierunek i zwrot jak wektor przed normalizacją, ale jednostkową długość.

Aby OpenGL używało normalnych jednostkowych, należy wywołać funkcję  `glEnable()` z parametrem `GL_NORMALIZE` lub `GL_RESCALE_NORMALIZE`:

```
	glEnable(GL_NORMALIZE);
```

Parametr `GL_RESCALE_NORMALIZE` stosuje się, gdy normalne zostaną już przeskalowane i posiadają długość równą 1. Spowoduje on przemnożenie wszystkich składowych normalnej przez tę samą wartość uzyskaną z macierzy modelowania.

Częściej stosuje się jednak parametr `GL_NORMALIZE`, który sprawia, że OpenGL wyznacza wektor jednostkowy. Ułatwia to pracę programisty, który nie musi samodzielnie wyznaczać normalnych jednostkowych. Jednak powoduje wolniejsze działanie OpenGL. Dlatego też lepiej jest zawsze samodzielnie wyznaczać normalne jednostkowe.

Poniżej zaprezentowany został przykład implementacji funkcji, której można użyć w tym celu:

```
// wyznacza i zwraca długość wektora
double VectorLength(float vector[3])
{
    return sqrt((vector[0]*vector[0]) +
                (vector[1]*vector[1]) +
                (vector[2]*vector[2]));
}
```

```
// normalizuje wektor normalVector
void Normalize(float normalVector[3])
{
    double length; // długość wektora o podwójnej precyzji

    length = VectorLength(normalVector); // oblicza długość wektora

    // dzieli składowe wektora przez długość
    for(int idx=0; idx < 3; idx++)
        normalVector[idx] /= length;
}
```

Funkcji Normalize() używa się zwykle po wyznaczeniu wektora normalnej. Można więc skorzystać z pokazanej wcześniej funkcji wyznaczania iloczynu wektorowego CrossProduct(), a następnie wywołać funkcję Normalize() uzyskując w ten sposób kod nowej funkcji obliczania normalnej jednostkowej CalculateNormal():

```
void CalculateNormal(float point1[3],
                      float point2[3],
                      float point3[3],
                      float normal[3])
{
    // oblicza iloczyn wektorów wyznaczonych przez trzy punkty
    CrossProduct(point1, point2, point3, normal);

    // normalizuje wektor normalny,
    // aby przyspieszyć obliczenia oświetlenia

    Normalize(normal);
}
```

Funkcja ta oblicza iloczyn wektorowy, który następnie normalizuje w celu uzyskania normalnej jednostkowej. Aby przyspieszyć nieco jej działanie, można umieścić kod funkcji CrossProduct() i Normalize() bezpośrednio w „ciele” funkcji CalculateNormal(). Na razie należy pozostawić jednak funkcję CalculateNormal() w obecnej postaci, aby zachować przejrzystość jej działania.

## Korzystanie z oświetlenia w OpenGL

OpenGL umożliwia umieszczenie w tworzonej scenie do ósmu źródeł światła. Liczba ta powinna być wystarczająca w zdecydowanej większości przypadków. W pozostałych przypadkach można natomiast zastosować odpowiednie przekształcenia tak, by obciążać źródła światła, które nie są widoczne. W celu dodania oświetlenia do tworzonej sceny należy wykonać podane niżej cztery polecenia (według książki *OpenGL Programming Guide* autorstwa Woo, Neidera i Davisa, a wydanej przez Addison-Wesley).

1. Należy wyznaczyć wektory normalnych w każdym wierzchołku każdego obiektu. Pozwolą one określić orientację obiektów względem źródeł światła.
2. Należy utworzyć, wybrać i umieścić źródła światła.
3. Należy utworzyć i wybrać model oświetlenia. Model ten definiuje światło otoczenia oraz położenie obserwatora, dla którego wyznaczane jest oświetlenie.
4. Należy zdefiniować właściwości materiałów obiektów znajdujących się na scenie.

Dotąd omówiony jedynie został pierwszy krok. Zanim przedstawiony będzie drugi z nich, trzeba przyjrzeć się najpierw fragmentowi kodu, który wyświetla obracający się i oświetlony sześcian:

```
////// Zmienne oświetlenia
float ambientLight[] = { 0.3f, 0.5f, 0.8f, 1.0f }; // światło otoczenia
float diffuseLight[] = { 0.25f, 0.25f, 0.25f, 1.0f }; // światło rozproszone
float lightPosition[] = { 0.0f, 0.0f, 0.0f, 1.0f }; // pozycja źródła światła
```

Zmienne te określają właściwości oświetlenia. Tablica `ambientLight` deklaruje światło otoczenia o wartościach składowej czerwonej, zielonej i niebieskiej wynoszących odpowiednio 0.3, 0.5 i 0.8. Oznacza to, że światło padające na sześcian ze wszystkich kierunków będzie miało odcień niebieski, ponieważ ta składowa ma największą intensywność. Zmienna `diffuseLight` nadaje wszystkim składowym światła rozproszonego tę samą wartość 0.25. Oznacza to, że każda ze ścian oświetlona bezpośrednio przez to światło będzie jaśniejsza od pozostałych ścian sześcianu. Ostatnia z wartości podanych dla obu rodzajów światła opisuje współczynnik alfa, który omówiony zostanie później.

Ostatnia ze zmiennych określa pozycję źródła światła. Pierwsze trzy wartości mają takie samo znaczenie jak w przypadku funkcji `glTranslate()`. W tym przykładzie źródło światła zostanie umieszczone w początku układu współrzędnych. Czwarta wartość informuje OpenGL, czy pierwsze trzy wartości reprezentują współrzędne źródła światła czy wektor. Wartość 1.0 oznacza, że reprezentują one współrzędne źródła światła.

Jeśli zmieniona zostanie ostatnia wartość na 0.0, to OpenGL zinterpretuje pierwsze trzy wartości jak składowe wektora określającego kierunek, z którego pada światło. Wektor (0, 0, 0) nie reprezentuje żadnego kierunku, wobec czego należy zmienić jego składowe. Jeśli przyjmie się założenie, że obserwator znajduje się w środku układu współrzędnych i spogląda w kierunku ujemnej części osi z, wtedy można ustalić kierunek, z którego pada światło jako dodatni kierunek osi z. Wektor taki należy zdefiniować następująco:

```
// światło padające wzduż osi z
float lightPosition[] = { 0.0f, 0.0f, 1.0f, 0.0f };
```

Teraz należy przyjrzeć się wartościom opisującym materiał sześcianu.

```
////// Zmienne opisujące materiał
float matAmbient[] = { 1.0f, 1.0f, 1.0f, 1.0f };
float matDiff[] = { 1.0f, 1.0f, 1.0f, 1.0f };
```

Pierwsza z nich, `matAmbient` definiuje stopień odbicia światła otoczenia przez dany materiał. Wartość 1.0 oznacza całkowite odbicie składowej czerwonej, zielonej i niebieskiej padającego światła. Takie same wartości zdefiniowane zostały także w tablicy `matDiff` i dlatego wszystkie składowe światła rozproszonego zostaną całkowicie odbite. Poniżej przedstawiony został kod źródłowy funkcji `Initialize()`, która wykorzystuje omówione dotąd zmienne opisujące właściwości oświetlenia i materiału:

```
void Initialize()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // kolor czarny
    glShadeModel(GL_SMOOTH); // włącza cieniowanie gładkie
    glEnable(GL_DEPTH_TEST); // usuwa ukryte powierzchnie
    glEnable(GL_CULL_FACE); // ukrywa tylną stronę wielokątów
```

```
glFrontFace(GL_CCW);           // tylna = tworzone w kierunku przeciwnym  
                                // do ruchu wskazówek zegara  
  
glEnable(GL_LIGHTING);        // aktywuje oświetlenie  
  
// Określa właściwości materiału  
glMaterialfv(GL_FRONT, GL_AMBIENT, matAmbient);  
glMaterialfv(GL_FRONT, GL_DIFFUSE, matDiff);  
  
// Określa właściwości oświetlenia  
glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight); // światło otoczenia  
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight); // światło rozproszone  
glLightfv(GL_LIGHT0, GL_POSITION, lightPosition); // położenie źródła światła  
  
// Włącza oświetlenie  
glEnable(GL_LIGHT0);  
}
```

Komentarze, którymi opatrzony został kod funkcji `Initialize()` wyjaśniają sposób jej działania. Za pomocą wartości `GL_SMOOTH` włącza się model cieniowania Gouraud, a za pomocą wartości `GL_DEPTH_TEST` aktywuje się usuwanie ukrytych powierzchni.

Wartość `GL_CULL_FACE` użyta została po raz pierwszy właśnie w kodzie funkcji `Initialize()`. Powoduje ona, że OpenGL nie prowadzi żadnych obliczeń dla tylnych stron wielokątów. Odróżnienie stron tylnych od przednich umożliwia zastosowanie wartości `GL_CCW` jako parametru funkcji `glFrontFace()`. Oznacza ona, że strony przednie posiadają uporządkowanie wierzchołków wielokątów w kierunku przeciwnym do kierunku ruchu wskazówek zegara. Trzeba pamiętać o zachowaniu tego uporządkowania podczas tworzenia wielokątów, ponieważ rysowanie ich tylnych stron wyłączone zostało za pomocą wartości `GL_CULL_FACE`.

Kolejną z użytych wartości jest `GL_LIGHTING`. Przekazana jako parametr funkcji  `glEnable()` informuje OpenGL, że będą przeprowadzane obliczenia związane z wyznaczeniem oświetlenia obiektów.

Funkcja `glMaterialfv()` określa właściwości materiału dla różnych rodzajów oświetlenia. Funkcje związane z właściwościami materiałów omówione zostaną dokładniej w dalszej części tego rozdziału. W tym przykładzie za pomocą funkcji `glMaterialfv()` określa się sposób odbicia światła otoczenia i światła rozproszonego przez przednią stronę wielokątów.

Funkcja `glLightfv()` określa właściwości źródeł światła. Trzeba w tym miejscu przypomnieć o tym, że OpenGL pozwala umieścić na tworzonej scenie do osmiu źródeł światła oznaczonych za pomocą wartości `GL_LIGHTX`, gdzie  $X$  jest liczbą z przedziału od 0 do 7. Aby więc zmodyfikować właściwości na przykład piątego źródła światła, należy podać jako parametr funkcji `glLightfv()` wartość `GL_LIGHT4`. Funkcja `glLightfv()` omówiona zostanie szczegółowo w dalszej części rozdziału.

Na końcu należy jeszcze włączyć źródło światła przekazując odpowiednią wartość `GL_LIGHTX` funkcji  `glEnable()`. W przeciwnym razie scena nie zostanie oświetlona.

Poniżej przedstawiony został kod funkcji `DrawCube()` rysującej i przekształcającej sześcian.

```
// DrawCube
// opis: rysuje sześcian na podanej pozycji
// w bieżącym układzie współrzędnych
void DrawCube(float xPos, float yPos, float zPos)
{
    glPushMatrix();
    glTranslatef(xPos, yPos, zPos);
    glBegin(GL_POLYGON);
        glNormal3f(0.0f, 1.0f, 0.0f); // górną ścianą
        glVertex3f(0.5f, 0.5f, 0.5f);
        glVertex3f(0.5f, 0.5f, -0.5f);
        glVertex3f(-0.5f, 0.5f, -0.5f);
        glVertex3f(-0.5f, 0.5f, 0.5f);
    glEnd();
    glBegin(GL_POLYGON);
        glNormal3f(0.0f, 0.0f, 1.0f); // przednią ścianą
        glVertex3f(0.5f, 0.5f, 0.5f);
        glVertex3f(-0.5f, 0.5f, 0.5f);
        glVertex3f(-0.5f, -0.5f, 0.5f);
        glVertex3f(0.5f, -0.5f, 0.5f);
    glEnd();
    glBegin(GL_POLYGON);
        glNormal3f(1.0f, 0.0f, 0.0f); // prawa ścianą
        glVertex3f(0.5f, 0.5f, 0.5f);
        glVertex3f(0.5f, -0.5f, 0.5f);
        glVertex3f(0.5f, -0.5f, -0.5f);
        glVertex3f(0.5f, 0.5f, -0.5f);
    glEnd();
    glBegin(GL_POLYGON);
        glNormal3f(-1.0f, 0.0f, 0.0f); // lewa ścianą
        glVertex3f(-0.5f, 0.5f, 0.5f);
        glVertex3f(-0.5f, 0.5f, -0.5f);
        glVertex3f(-0.5f, -0.5f, -0.5f);
        glVertex3f(-0.5f, -0.5f, 0.5f);
    glEnd();
    glBegin(GL_POLYGON);
        glNormal3f(0.0f, -1.0f, 0.0f); // dolną ścianą
        glVertex3f(-0.5f, -0.5f, 0.5f);
        glVertex3f(-0.5f, -0.5f, -0.5f);
        glVertex3f(0.5f, -0.5f, -0.5f);
        glVertex3f(0.5f, -0.5f, 0.5f);
    glEnd();
    glBegin(GL_POLYGON);
        glNormal3f(0.0f, 0.0f, -1.0f); // tylną ścianą
        glVertex3f(0.5f, -0.5f, -0.5f);
        glVertex3f(-0.5f, -0.5f, -0.5f);
        glVertex3f(-0.5f, 0.5f, -0.5f);
        glVertex3f(0.5f, 0.5f, -0.5f);
    glEnd();
    glPopMatrix();
}
```

Funkcja DrawCube() rysuje sześcian w pozycji określonej za pomocą jej parametrów względem bieżącej macierzy przekształceń. Każda z sześciu ścian rysowana jest osobno. Razem tworzą one sześcian o wymiarach  $1 \times 1 \times 1$ , którego środkiem jest początek układu współrzędnych. Najistotniejszą nowością jest tutaj użycie funkcji glNormal3f(), która definiuje normalną do każdej ze ścian sześcianu.

Przednia ściana sześcianu położona jest naprzeciw obserwatora. Jej normalna wskazuje więc w kierunku dodatniej części osi z i zdefiniowana jest przez wektor  $(0, 1, 0)$ . Normalna dolnej ściany sześcianu wskazuje w kierunku ujemnej części osi y i zdefiniowana jest przez wektor  $(0, -1, 0)$ . Normalne te zdefiniowane zostały w postaci jednostkowej po to, aby pomóc OpenGL wyznaczyć właściwe oświetlenie ścian sześcianu. W przeciwnym razie należałoby poinformować maszynę OpenGL za pomocą wartości `GL_NORMALIZE`, że sama powinna wyznaczyć normalne jednostkowe.

Poniżej zaprezentowany został kod funkcji `Render()`, która używa omówionej funkcji `DrawCube()`:

```
// Render
// opis: rysuje scenę
void Render()
{
    // opróżnia bufory ekranu i głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();           // resetuje macierz modelowania

    angle = angle + 0.1f;      // zwiększa licznik kąta obrotu
    if (angle >= 360.0f)
        angle = 0.0f;

    // wykonuje przekształcenia
    glTranslatef(0.0f, 0.0f, -3.0f);
    glRotatef(angle, 1.0f, 0.0f, 0.0f);
    glRotatef(angle, 0.0f, 1.0f, 0.0f);
    glRotatef(angle, 0.0f, 0.0f, 1.0f);

    DrawCube(0.0f, 0.0f, 0.0f); // rysuje przekształcony sześciian

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

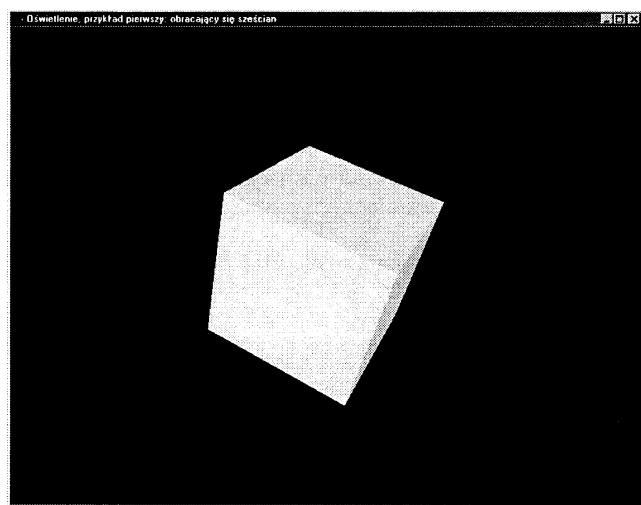
Funkcja `Render()` wywoływana jest w celu utworzenia kolejnej klatki animacji. Jej kod nie zawiera żadnych nowych elementów, które wymagałyby szerszego omówienia. Funkcja `Render()` przesywa za każdym razem bieżący układ współrzędnych do punktu  $(0, 0, -3)$ , w którym zostaje on obrócony o ten sam kąt względem wszystkich osi, a następnie rysuje sześcian. Oświetlenie sześcianu wyznaczane jest automatycznie przez OpenGL i nie wymaga jakiejkolwiek interwencji ze strony programisty.

W przykładzie tym nie jest używana utworzona wcześniej funkcja `CalculateNormal()`, ponieważ „ręczne” wyznaczenie normalnych w przypadku sześcianu nie przedstawia trudności. Podobne rozwiązanie stosowane jest często w przypadku ładowania trójwymiarowych modeli z plików. Niektóre formaty takich plików zawierają już gotowe dane określające normalne do wszystkich wielokątów modelu. W pozostałych przypadkach normalne muszą być wyznaczone podczas działania programu.

Rysunek 6.10 ilustruje działanie omówionego programu.

**Rysunek 6.10.**

*Animacja  
obracającego się  
i oświetlonego  
sześcianu*

**Tworzenie źródeł światła**

Jak pokazano w ostatnim przykładzie, dla źródła światła można zdefiniować szereg właściwości, takich jak kolor, położenie czy kierunek. Stosuje się w tym celu funkcję `glLight*` (). Poniższa wersja będzie tu najczęściej przez używana:

```
void glLightfv(GLenum light, GLenum pname, TYPE *param);
```

Funkcja ta posiada trzy parametry. Pierwszy z nich podaje źródło światła, którego właściwości są określane, drugi nazwę właściwości, a trzeci jej nową wartość. Jak już wspomniano, parametr *light* może przyjmować wartości `GL_LIGHT0`, `GL_LIGHT1` aż do `GL_LIGHT7`, co sprawia, że wybierane jest w ten sposób jedno z ośmiu dostępnych źródeł światła. Parametr *pname* określający właściwość źródła światła może być jedną z wartości przedstawionych w tabeli 6.2.

**Tabela 6.2.** Właściwości źródła światła

Właściwość	Znaczenie
<code>GL_AMBIENT</code>	Intensywność światła otoczenia
<code>GL_DIFFUSE</code>	Intensywność światła rozprozonego
<code>GL_SPECULAR</code>	Intensywność światła odbicia
<code>GL_POSITION</code>	Pozycja źródła światła (x, y, z, w)
<code>GL_SPOT_DIRECTION</code>	Wektor kierunku strumienia światła (x, y, z)
<code>GL_SPOT_EXPONENT</code>	Koncentracja strumienia światła
<code>GL_SPOT_CUTOFF</code>	Kąt rozwarcia strumienia światła
<code>GL_CONSTANT_ATTENUATION</code>	Stała wartość tłumienia
<code>GL_LINEAR_ATTENUATION</code>	Liniowa wartość tłumienia
<code>GL_QUADRATIC_ATTENUATION</code>	Kwadratowa wartość tłumienia

Dzięki zastosowaniu funkcji `glLightfv()` ostatni parametr będzie zawsze tablicą wartości typu `float`. Na przykład światło otoczenia określić można w następujący sposób:

```
float ambientLight[] = { 1.0f, 1.0f, 1.0f, 1.0f }; // światło białe, duża intensywność  
glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight); // określa światło otoczenia
```

Trzeba pamiętać o tym, że określając właściwość światła otoczenia dla danego źródła światła podaje się intensywność jego składowych w modelu RGBA, którą źródło dodaje do oświetlenia sceny. Powyższe dwa wiersze kodu informują więc OpenGL, że źródło światła `GL_LIGHT0` dodaje do oświetlenia sceny światło otoczenia w kolorze białym. Domyślnie scena nie posiada światła otoczenia, czyli jego wartość wynosi  $(0.0, 0.0, 0.0, 1.0)$ .

Znaczenie parametru `GL_DIFFUSE` jest nieco inne. Oznacza on kolor światła emitowanego przez źródło. Domyślną wartością właściwości `GL_DIFFUSE` jest  $(1.0, 1.0, 1.0, 1.0)$ , ale tylko dla źródła światła `GL_LIGHT0`. Dla pozostałych źródeł wynosi ona  $(0.0, 0.0, 0.0, 0.0)$ .

Właściwość `GL_SPECULAR` pozwala określić kolor światła odbicia. W praktyce często nadaje się jej taką samą wartość jak właściwości `GL_DIFFUSE` określającej kolor światła rozproszonego. W ten sposób uzyskuje się realistyczny efekt, ponieważ w świecie rzeczywistym obiekty często sprawiają wrażenie, że mają taki sam kolor w świetle odbitym, jak i rozproszonym. Domyślna wartość właściwości `GL_SPECULAR` wynosi  $(1.0, 1.0, 1.0, 1.0)$  dla źródła światła `GL_LIGHT0` oraz  $(0.0, 0.0, 0.0, 0.0)$  dla pozostałych źródeł.

Jak już wspomniano, podczas definiowania składowych w modelu RGBA ostatnia wartość prezentuje współczynnik alfa. Nie trzeba się nią jednak zajmować aż do momentu omówienia zagadnienia łączenia kolorów.

## Położenie źródła światła

Sposób umieszczania źródła światła omówiony został już przy okazji ostatniego programu. Teraz należy jednak przyjrzeć się temu zagadnieniu dokładniej. Położenie źródła światła definiuje się za pomocą wektora  $(x, y, z, w)$  właściwości `GL_POSITION`. Jak już poinformowano wcześniej, jeśli wartość  $w$  jest równa  $0.0$ , to  $(x, y, z)$  definiuje wektor określający kierunek, z którego pada światło. Tym samym definiowane jest *kierunkowe źródło światła*, czyli takie, którego wszystkie promienie są równoległe tak, jakby znajdowało się ono nieskończoność daleko. Najlepszym przykładem kierunkowego źródła światła jest oczywiście Słońce. Na niewielkim obszarze Ziemi jego promienie są praktycznie równoległe do siebie. Kierunkowe źródło światła może zostać zdefiniowane na przykład w poniższy sposób

```
float lightPosition[] = { 0.0f, 0.0f, 1.0f, 0.0f };  
glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
```

Powyższy fragment kodu definiuje kierunkowe źródło światła, którego promienie padają z kierunku dodatniej części osi  $z$ , czyli w domyślnym kierunku orientacji kamery.

Jeśli wartość  $w$  będzie różna od  $0.0$ , to wektor  $(x, y, z)$  zdefiniuje *pozycyjne źródło światła*. Źródło takie znajduje się w punkcie o współrzędnych  $(x, y, z)$  i promieniu we wszystkich kierunkach. Przykładem takiego źródła światła w świecie rzeczywistym jest

żarówka. Poniższy fragment kodu definiuje pozycyjne źródło światła znajdujące się w po-czętku układu współrzędnych.

```
float lightPosition[] = { 0.0f, 0.0f, 0.0f, 1.0f };
glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
```

## Tłumienie

*Tłumienie* charakteryzuje spadek intensywności światła w miarę oddalania się od jego źródła. Efekt ten najlepiej można zaobserwować na przykładzie latarni znajdującej się na zamglonej ulicy. Taki sam efekt można uzyskać w tworzonym przez programistę świecie gry, ale tylko dla pozycyjnych źródeł światła. Tłumienie nie ma sensu w przypadku kierunkowych źródeł światła, ponieważ znajdują się one w nieskończonej odległości od oświetlanych obiektów. OpenGL wyznacza tłumienie światła mnożąc jego intensywność przez współczynnik tłumienia. Tabela 6.3 prezentuje domyślne wartości właściwości źródła związanych z tłumieniem.

**Tabela 6.3.** Właściwości źródła światła określające jego tłumienie

Właściwość	Wartość domyślna
GL_CONSTANT_ATTENUATION	1.0
GL_LINEAR_ATTENUATION	0.0
GL_QUADRATIC_ATTENUATION	0.0

Wartości tych właściwości można zmienić za pomocą funkcji `glLight()`:

```
glLightfv(GL_LIGHT0, GL_ATTENUATION, 4.0f);
glLightfv(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0f);
glLightfv(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.25f);
```

Tłumienie ma wpływ na światło w całym świecie tworzonym przez OpenGL z wyjątkiem światła emisji oraz globalnego światła otoczenia. Stosowanie tłumienia może nieco spowolnić tworzenie grafiki, ponieważ wymaga dodatkowych obliczeń.

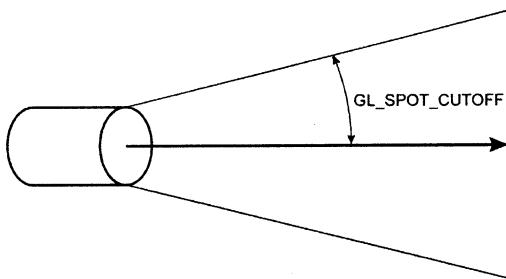
## Strumień światła

Jeśli promieniowanie pozycyjnego źródła światła, zostanie zawiązane tylko do określonego kierunku, to uzyskany zostanie w efekcie *strumień światła*. Strumień światła tworzy się w ten sam sposób co pozycyjne źródło światła (dodając jedynie kilka specyficznych parametrów, takich jak kąt rozwarcia strumienia światła, kierunek strumienia światła i jego zogniskowanie).

Przyglądając się strumieniowi światła w ciemnościach zauważać można, że tworzy on stożek świetlny w kierunku, w którym biegnie. Stożek ten definiuje się w OpenGL podając wartość kąta, jaki tworzy powierzchnia stożka z jego osią. Kąt ten stanowi wartość właściwości `GL_SPOT_CUTOFF`, co pokazuje rysunek 6.11.

**Rysunek 6.11.**

Właściwość  
`GL_SPOT_CUTOFF`  
 definiuje kąt  
 pomiędzy  
 powierzchnią i osią  
 stożka strumienia  
 światła



Jeśli właściwości `GL_SPOT_CUTOFF` nadana zostanie wartość  $180^\circ$ , to światło rozchodzić się będzie we wszystkich kierunkach i uzyskany zostanie efekt identyczny jak w przypadku zwykłego pozycyjnego źródła światła. Oprócz specjalnej wartości  $180^\circ$  OpenGL dopuszcza jedynie wartości właściwości `GL_SPOT_CUTOFF` z przedziału od 0 do  $90^\circ$ . Strumień światła o kącie rozwarcia stożka wynoszącym  $30^\circ$  trzeba zdefiniować w następujący sposób:

```
glLightfv(GL_LIGHT0, GL_SPOT_CUTOFF, 15.0f); // stożek o kącie rozwarcia 30 stopni
```

Dla strumienia światła trzeba podać także jego kierunek. Określa go właściwość `GL_SPOT_DIRECTION`, która jest wektorem postaci  $(x, y, z)$ . Domyślną wartością właściwości `GL_SPOT_DIRECTION` jest wektor  $(0.0, 0.0, -1.0)$  wskazujący ujemną część osi z. Inny kierunek strumienia światła można ustalić za pomocą funkcji `glLightfv()`:

```
float spotlightDirection[] = { 0.0, -1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spotlightDirection);
```

W tym przypadku strumień światła będzie skierowany w ujemnym kierunku osi y.

Ostatnią z właściwości strumienia światła jest jego zogniskowanie, które określa koncentrację światła na osi stożka. Oddalając się od osi stożka w kierunku jego powierzchni światło staje się coraz bardziej słumione i osiąga zerową intensywność na jego powierzchni. Zwiększąc wartość właściwości `GL_SPOT_EXPONENT` można otrzymać bardziej skoncentrowany strumień światła. Poniższy wiersz kodu nadaje tej właściwości wartość 10.0:

```
glLightfv(GL_LIGHT0, GL_SPOT_EXPONENT, 10.0f);
```

Poniżej przedstawiony został fragment programu, który kieruje strumień światła na obracającą się kulę:

```
////// Zmienne globalne
float angle = 0.0f; // bieżący kąt obrotu kuli

////// Zmienne opisujące oświetlenie
float ambientLight[] = { 0.5f, 0.5f, 0.5f, 1.0f }; // światło otoczenia
float diffuseLight[] = { 0.5f, 0.5f, 0.5f, 1.0f }; // światło rozproszone
float specularLight[] = { 1.0f, 1.0f, 1.0f, 1.0f }; // światło odbicia

// pozycja źródła strumienia światła
float spotlightPosition[] = { 6.0f, 0.5f, 0.0f, 1.0f };
float spotlightDirection[] = { -1.0f, 0.0f, -1.0f }; // kierunek strumienia światła
```

```

////// Zmienne opisujące materiał
float matAmbient[] = { 1.0f, 1.0f, 1.0f, 1.0f }; // materiał w świetle otoczenia
float matDiff[] = { 1.0f, 1.0f, 1.0f, 1.0f }; // materiał w świetle rozproszonym
float matSpecular[] = { 1.0f, 1.0f, 1.0f, 1.0f }; // materiał w świetle odbicia

// Initialize
// opis: inicjuje OpenGL
void Initialize()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // kolor czarny

    glShadeModel(GL_SMOOTH); // aktywuje cieniowanie gładkie
    glEnable(GL_DEPTH_TEST); // oraz usuwanie niewidocznych powierzchni
    glEnable(GL_CULL_FACE); // wyłącza obliczenie dla tylnych stron wielokątów
    glFrontFace(GL_CCW); // tylne = uporządkowanie wierzchołków wielokąta
                          // przeciwnie do kierunku ruchu wskazówek zegara

    glEnable(GL_LIGHTING); // aktywuje wyznaczanie oświetlenia

    // Konfiguruje źródło światła LIGHT0
    glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight); // światło otoczenia
    glLightfv(GL_LIGHT0, GL_DIFFUSE, ambientLight); // światło rozproszone
    glLightfv(GL_LIGHT0, GL_POSITION, spotlightPosition); // pozycja źródła światła

    glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 40.0f); // stożek o kącie rozwarcia 80 stopni
    glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 30.0f);
    glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spotlightDirection);

    // Włącza źródło światła
    glEnable(GL_LIGHT0);

    glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

    glMaterialfv(GL_FRONT, GL_SPECULAR, matSpecular);
    glMaterialfv(GL_FRONT, GL_SHININESS, 10.0f);
}

// Render
// opis: rysuje scenę
void Render()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // opróżnia bufory ekranu i głębi
    glLoadIdentity(); // resetuje macierz modelowania

    angle = angle + 0.1f; // zwiększa licznik kąta obrotu
    if (angle >= 360.0f)
        angle = 0.0f;

    // wykonuje przekształcenia
    // (przesunięcie o 5 jednostek w tył i obrót dookoła osi y)
    glTranslatef(0.0f, 0.0f, -5.0f);
    glRotatef(angle, 0.0f, 1.0f, 0.0f);
    glColor3f(0.0f, 0.5f, 0.5f);
    auxSolidSphere(1.0);

    glFlush();

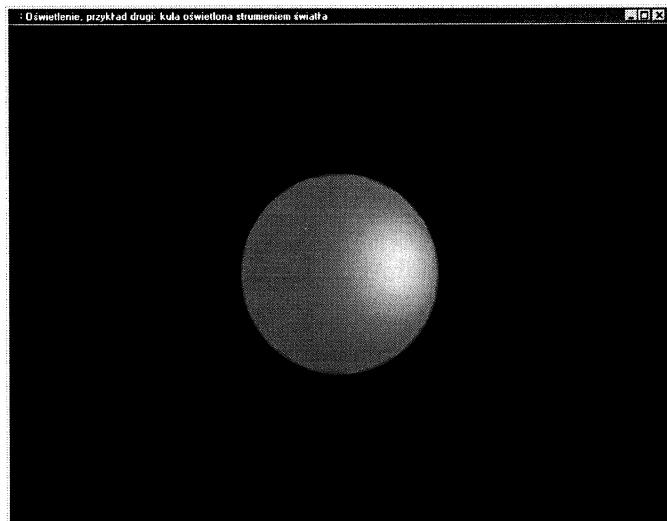
    SwapBuffers(g_HDC); // przełącza bufory
}

```

Działanie programu ilustruje rysunek 6.12. Do narysowania kuli użyta została funkcja `auxSolidSphere()` z biblioteki pomocniczej GLAUX. Funkcji tej przekazany został żądany promień kuli. Aby móc korzystać z funkcji biblioteki GLAUX, należy umieścić plik *GLAUX.LIB* wśród bibliotek dołączanych przez kompilator. Sposób postępowania podany został w rozdziale 1.

**Rysunek 6.12.**

*Strumień światła  
padający  
na wirującą kulę*



W powyższym fragmencie programu znajduje się kilka wierszy kodu, które nie zostały jeszcze omówione. Dotyczą one definiowania materiałów, do omówienia którego teraz można przejść.

## Definiowanie materiałów

Pokazano dotąd kilka przykładów definiowania właściwości materiałów tworzących obiekty, ale nie zostały jeszcze omówione służące do tego funkcje. Definiowanie materiału przypomina definiowanie źródła światła, ale oczywiście odbywa się za pomocą innych funkcji:

```
void glMaterialf(GLenum face, GLenum pname, TYPE param);
void glMaterialfv(GLenum face, GLenum pname, TYPE *param);
```

Funkcje te definiują właściwości materiałów uwzględniane podczas obliczeń oświetlenia. Parametr *face* pozwala określić, którą stronę wielokąta tworzy dany materiał. Parametr ten może mieć jedną z wartości *GL\_FRONT*, *GL\_BACK* lub *GL\_FRONT\_AND\_BACK*. Wartość *GL\_FRONT* ogranicza zastosowanie materiału tylko do przedniej strony wielokąta, a *GL\_BACK* tylko do tylnej. Wartość *GL\_FRONT\_AND\_BACK* powoduje użycie materiału dla obu stron wielokąta. Kolejny parametr, *pname*, określa właściwość materiału, która ma być zmodyfikowana. Może on przyjmować jedną z wartości wymienionych w tabeli 6.4. Ostatni z parametrów, *param*, w zależności od wersji funkcji jest albo tablicą (dla *glMaterialfv*), albo skalarem (dla *glMaterialf*).

**Tabela 6.4.** Właściwości materiałów

Właściwość	Znaczenie
GL_AMBIENT	Kolor przy oświetleniu światłem otoczenia
GL_DIFFUSE	Kolor przy oświetleniu światłem rozproszonym
GL_AMBIENT_AND_DIFFUSE	Kolor przy oświetleniu światłem otoczenia i światłem rozproszonym
GL_SPECULAR	Kolor przy oświetleniu światłem odbicia
GL_SHININESS	Współczynnik odbicia
GL_EMISSION	Kolor światła emisji

Aby określić kolor materiału dla przedniej i tylnej strony wielokąta widziany w świetle otoczenia jako czerwony, należy wykorzystać następujący fragment kodu:

```
float red[] = { 1.0f, 0.0f, 0.0f, 1.0f };
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, red);
```

Aby określić kolor materiału dla przedniej strony wielokąta widziany w świetle otoczenia i świetle rozproszonym jako biały, należy wykorzystać poniższy fragment kodu:

```
float white[] = { 1.0f, 1.0f, 1.0f, 1.0f };
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, white);
```

Trzeba pamiętać, że zdefiniowany materiał będzie określać wygląd wszystkich tworzonych wielokątów aż do momentu kolejnego wywołania funkcji `glMaterial*`.

Inny sposób określania właściwości materiałów polega na *śledzeniu kolorów*. Pozwala on określić właściwości materiałów za pomocą funkcji `glColor*`. Śledzenie kolorów musi najpierw zostać włączone poprzez wywołanie funkcji  `glEnable()` z parametrem `GL_COLOR_MATERIAL`. Następnie za pomocą funkcji  `glColorMaterial()` należy podać, które z właściwości materiałów będą określane przez wywołania funkcji  `glColor*`. Poniższy fragment kodu powoduje, że kolor przedniej strony wielokątów widziany w świetle rozproszonym będzie ustalany na drodze śledzenia kolorów:

```
glEnable(GL_COLOR_MATERIAL);           // włącza śledzenie kolorów
	glColorMaterial(GL_FRONT, GL_DIFFUSE); // strona przednia, światło rozproszone
	glColor3f(1.0f, 0.0f, 0.0f);          // kolor czerwony
 glBegin(GL_TRIANGLES);
    // tutaj rysuje trójkąty
 glEnd();
```

Jak widać, śledzenie kolorów jest wyjątkowo łatwe w konfiguracji i użyciu. Poprzedni program przykładowy wykorzystuje śledzenie kolorów do określenie właściwości materiału kuli. Definiując różne właściwości materiałów można uzyskać różne efekty związane z ich oświetleniem, które omówione zostaną niebawem.

## Modele oświetlenia

Z tworzeniem oświetlenia sceny związane jest jedno istotne zagadnienie, które należy omówić: wybór *modelu oświetlenia*. Model oświetlenia w OpenGL składa się z czterech komponentów mających wpływ na wygląd sceny:

- ◆ intensywności światła otoczenia sceny;
- ◆ położenia obserwatora (lokalne lub w nieskończoności), które ma wpływ na wyznaczanie kąta odbicia;
- ◆ oświetlenia jedno- lub dwustronnego;
- ◆ oddzielnego lub wspólnego traktowania koloru odbicia z kolorami widzianymi w świetle otoczenia i świetle rozproszonym.

Model oświetlenia definiuje się za pomocą jednej z następujących wersji funkcji `glLightModel*`:

```
void glLightModel[if](GLenum pname, TYPE param);
void glLightModel[if]v(GLenum pname, TYPE *param);
```

Pierwszy z parametrów tych funkcji, *pname*, określa definiowaną właściwość modelu oświetlenia. Drugi (w zależności od wersji funkcji) może być pojedynczą wartością typu `float` lub tablicą takich wartości. Parametr *pname* może przyjmować wartości przedstawione w tabeli 6.5.

**Tabela 6.5. Właściwości modelu oświetlenia**

Właściwość	Znaczenie
<code>GL_LIGHT_MODEL_AMBIENT</code>	Intensywność światła otaczającego sceny w modelu RGBA; domyślnie wartość (0.2, 0.2, 0.2, 1.0)
<code>GL_LIGHT_MODEL_LOCAL_VIEWER</code>	Obserwator lokalny lub w nieskończoności; domyślnie wartość <code>GL_FALSE</code> (obserwator w nieskończoności)
<code>GL_LIGHT_MODEL_TWO_SIDE</code>	Oświetlenie jedno- lub dwustronne; domyślnie wartość <code>GL_FALSE</code> (oświetlenie jednostronne)
<code>GL_LIGHT_MODEL_COLOR_CONTROL</code>	Oddzielne traktowanie koloru w świetle odbicia od kolorów w świetle otoczenia i świetle rozproszonym; domyślnie wartość <code>GL_SINGLE_COLOR</code> (łączne traktowanie)

Pierwszą z właściwości podanych w tabeli 6.5 jest `GL_LIGHT_MODEL_AMBIENT`. Jak pokazano już wcześniej, światło otoczenia może pochodzić od zdefiniowanych źródeł światła. Właściwość `GL_LIGHT_MODEL_AMBIENT` modelu oświetlenia umożliwia dodanie do sceny światła otoczenia, które nie jest związane z żadnym źródłem światła. Światło to nazywa się *globalnym światłem otoczenia*. Ponizej zaprezentowany został fragment kodu, który dodaje do sceny globalne światło otoczenia o średniej intensywności:

```
float ambientLightModel[] = { 0.5, 0.5, 0.5, 1.0 }; // średnia intensywność
glLightModel(GL_LIGHT_MODEL_AMBIENT, ambientLightModel);
```

Gdy dla obiektów znajdujących się na scenie OpenGL wyznacza także światło odbicia, to brany jest pod uwagę punkt obserwacji. Właściwość `GL_LIGHT_MODEL_LOCAL_VIEWER` pozwala określić, czy obserwator efektów odbicia jest lokalny, czy też znajduje się w nieskończonej odległości od obiektów. Lokalne punkty obserwacji zwiększą realizm tworzonej sceny, ale zmniejszą przy tym efektywność jej tworzenia, ponieważ dla każdego wierzchołka musi zostać wyznaczony kierunek obserwacji. Domyślnie jest więc wykorzystywany nieskończony odległy punkt obserwacji (wartość `GL_FALSE`). Można jednak zmienić go na lokalny punkt obserwacji w następujący sposób:

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

Kolejną z właściwości modelu oświetlenia jest `GL_LIGHT_MODEL_TWO_SIDE`. Decyduje ona o tym, czy OpenGL oblicza oświetlenie tylnych stron wielokątów. Gdyby na przykład został wykonany przekrój oświetlonego sześcianu, to okazałoby się, że wewnętrzne strony jego ścian nie są prawidłowo oświetlone. Aby to zmienić, wystarczy nadać właściwości `GL_LIGHT_MODEL_TWO_SIDE` wartość `GL_TRUE`:

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

Dzięki temu OpenGL wyznaczy dla tylnych stron wielokątów normalne jako wektory przeciwnie do podanych przez programistę dla stron przednich i prawidłowo obliczy oświetlenie tylnych stron wielokątów. Oczywiście spowoduje to spowolnienie tworzenia grafiki. Do wyznaczania oświetlenia tylko dla jednej strony wielokątów można zawsze powrócić nadając właściwości `GL_LIGHT_MODEL_TWO_SIDE` wartość `GL_FALSE`.

Ostatnią z właściwości modelu oświetlenia jest właściwość `GL_LIGHT_MODEL_COLOR_CONTROL`. Ma ona znaczenie, gdy oświetlane są obiekty pokryte teksturą. W przypadku zastosowania teksturow standardowy sposób tworzenia efektów odbicia nie sprawdza się. Dlatego też OpenGL tworzy dla wszystkich wierzchołków podwójny opis kolorów: dodatkowy w świetle odbicia i podstawowy dla pozostałych rodzajów światła. Dla tekstuury wykorzystywany jest najpierw kolor podstawowy, a następnie dodawany jest do niego kolor dodatkowy. Pozwala to uzyskać lepiej widoczne efekty odbicia na teksturowach. Poniższy wiersz kodu informuje OpenGL, że powinien stosować podwójny opis kolorów:

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
```

Aby OpenGL tworzył łączny opis kolorów dla wszystkich rodzajów światła, trzeba nadać właściwości `GL_LIGHT_MODEL_COLOR_CONTROL` wartość `GL_SINGLE_COLOR`. Właściwość `GL_LIGHT_MODEL_COLOR_CONTROL` powinno się modyfikować jedynie, gdy używana jest tekstura. W przeciwnym razie nie ma ona znaczenia.

## Efekty odbicia

Ostatni z przykładowych programów pokazał, w jaki sposób można uzyskać efekt oświetlenia obracającej się kuli za pomocą strumienia światła. Przedstawionych teraz zostanie kilka innych interesujących efektów związanych ze światłem odbicia i właściwościami materiałów. Obserwując działanie wspomnianego programu, a nawet oglądając rysunek zamieszczony w książce można zauważyc połysk oświetlanej powierzchni kuli. Powstaje on, gdy kąt padania światła jest mały i polega na odbiciu prawie całego światła.

Chociaż strumień światła utworzony przez program nie posiada składowej odbicia, to można ją łatwo dodać. Podobnie jak w przypadku innych rodzajów światła określa się ją za pomocą funkcji `glLightfv()`:

```
float specularLight[] = { 1.0f, 1.0f, 1.0f, 1.0f }; // światło odbicia
float lightPosition[] = { 0.0f, 0.0f, 0.0f, 1.0f }; // położenie źródła
...
glEnable(GL_LIGHTING); // włącza obliczenia oświetlenia
...
// konfiguruje i włącza źródło GL_LIGHT0
glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight);
glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
glEnable(GL_LIGHT0);
```

Oczywiście do źródła GL\_LIGHT0 można dodać także światło otoczenia i światło rozproszone. Jednak tymczasem należy poprzestać na zdefiniowaniu tylko światła odbicia w kolorze białym o dużej intensywności, przypominającego światło słoneczne.

Aby uzyskać efekt odbicia, trzeba zdefiniować także odpowiednio stopień odbicia dla danego materiału. Program oświetlający kulę zastosował w tym celu poniższy fragment kodu:

```
float matSpecular[] = { 1.0f, 1.0f, 1.0f, 1.0f }; // materiał w świetle odbicia  
  
// materiał o słabym połysku  
glMaterialfv(GL_FRONT, GL_SPECULAR, matSpecular);  
glMaterialf(GL_FRONT, GL_SHININESS, 10.0f);
```

Wartości (1.0, 1.0, 1.0, 1.0) umieszczone w tablicy matSpecular oznaczają, że każda powierzchnia zdefiniowana po wywołaniu funkcji glMaterialfv() będzie całkowicie odbijać światło odbicia.

Ostatni z wierszy tego fragmentu kodu nadaje wartość właściwości GL\_SHININESS, która definiuje skupienie odbitego światła. Właściwość ta może przyjmować wartości z przedziału od 1.0 do 128.0 oraz wartość 0.0 oznaczającą, że światło odbicia nie jest skupione. Jeśli nadana jej zostanie wartość 128.0, to tworzone powierzchnie będą wyraźnie polążujące.

## Ruchome źródła światła

Źródła światła można przemieszczać i obracać tak samo jak każdy inny obiekt OpenGL. Gdy wywołuje się funkcję glLight\*() w celu zdefiniowania położenia źródła światła lub kierunku, z którego pada światło, to przekazana jej informacja przekształcana jest za pomocą macierzy modelowania. W przykładach programów przedstawionych dotąd w tym rozdziale wszystkie źródła światła były statyczne, ponieważ ich położenie zostało zdefiniowane, zanim wykonane zostały jakieś kolwiek przekształcenia.

Źródło światła można przemieszczać w trójwymiarowej przestrzeni tak samo jak każdy inny obiekt. Wystarczy podać jego nowe położenie po wykonaniu przesunięcia lub obrotu. Kolejny przykład programu porusza źródło światła wokół sześcianu i pozwala użytkownikowi modyfikować wygląd sceny. Należy przyjrzeć się jego kodowi:

```
////// Pliki nagłówkowe  
#include <windows.h> // standardowy plik nagłówkowy Windows  
#include <gl/gl.h> // standardowy plik nagłówkowy OpenGL  
#include <gl/glu.h> // plik nagłówkowy biblioteki GLU  
#include <gl/glaux.h> // funkcje pomocnicze OpenGL  
  
////// Zmienne globalne  
float angle = 0.0f; // bieżący kąt obrotu źródła światła  
HDC g_HDC; // globalny kontekst urządzenia  
bool fullScreen = false; // true = tryb pełnoekranowy; false = okienkowy  
bool keyPressed[256]; // zawiera wartość true dla klawiszy,  
// które zostały naciśnięte przez użytkownika  
  
// Pozycje źródeł światła  
float lightPositionR[] = { 0.0f, 0.0f, 75.0f, 1.0f };  
float lightPositionG[] = { 0.0f, 0.0f, 75.0f, 1.0f };  
float lightPositionB[] = { 0.0f, 0.0f, 75.0f, 1.0f };
```

```
// Intensywność składowej czerwonej, zielonej i niebieskiej
// dla światła rozproszonego
float diffuseLightR[] = { 1.0f, 0.0f, 0.0f, 1.0f };
float diffuseLightG[] = { 0.0f, 1.0f, 0.0f, 1.0f };
float diffuseLightB[] = { 0.0f, 0.0f, 1.0f, 1.0f };

// Intensywność składowej czerwonej, zielonej i niebieskiej
// dla światła odbicia
float specularLightR[] = { 1.0f, 0.0f, 0.0f, 1.0f };
float specularLightG[] = { 0.0f, 1.0f, 0.0f, 1.0f };
float specularLightB[] = { 0.0f, 0.0f, 1.0f, 1.0f };

// Kierunek strumienia światła
float spotDirection[] = { 0.0f, 0.0f, -1.0f };

// Globalne właściwości oświetlenia
float diffuseLight[] = { 0.5f, 0.5f, 0.5f, 1.0f };
float specularLight[] = { 1.0f, 1.0f, 1.0f, 1.0f };
float lightPosition[] = { 0.0f, 0.0f, 100.0f, 1.0f };

float objectXRot;           // obrót sześcianu wokół osi x
float objectYRot;           // obrót sześcianu wokół osi y
float objectZRot;           // obrót sześcianu wokół osi z

float redXRot;              // obrót źródła światła czerwonego wokół osi x
float redYRot;              // obrót źródła światła czerwonego wokół osi y

int currentColor = 1;        // bieżący kolor źródła światła:
                             // 1 = czerwony, 2 = zielony, 3 = niebieski
bool spotEnabled = true;     // strumień światła true = włączony; false = wyłączony

// Initialize
// opis: inicjuje OpenGL
void Initialize()
{
    glEnable(GL_SMOOTH); // aktywuje cieniowanie gładkie
    glEnable(GL_DEPTH_TEST); // oraz usuwanie niewidocznych powierzchni
    glEnable(GL_CULL_FACE); // wyłącza obliczenie dla tylnych stron wielokątów
    glFrontFace(GL_CCW); // tylne = uporządkowanie wierzchołków wielokąta
                          // przeciwnie do kierunku ruchu wskazówek zegara

    glEnable(GL_LIGHTING); // aktywuje wyznaczanie oświetlenia

    // LIGHT0 reprezentuje strumień światła,
    // którego źródło znajduje się w punkcie (0.0, 0.0, 100.0)
    // i skierowane jest w kierunku ujemnej części osi z.
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
    glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight);
    glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
    glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 40.0f);
    glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 80.0f);

    // LIGHT1 reprezentuje poruszające się źródło światła.
    // Początkowo ma ono kolor czerwony.
    glLightfv(GL_LIGHT1, GL_DIFFUSE, diffuseLightR);
    glLightfv(GL_LIGHT1, GL_SPECULAR, specularLightR);
    glLightfv(GL_LIGHT1, GL_POSITION, lightPositionR);
```

```
// włącza źródła światła
glEnable(GL_LIGHT0);
glEnable(GL_LIGHT1);

// włącza śledzenie kolorów
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

// efekt silnego odbicia
glMaterialfv(GL_FRONT, GL_SPECULAR, specularLight);
glMateriali(GL_FRONT, GL_SHININESS, 128);

// czarny kolor tła
glClearColor(0.0f, 0.0f, 0.0f);
}

// Render
// opis: rysuje scenę
void Render()
{
    // opróżnia bufory ekranu i głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // resetuje macierz modelowania
    glLoadIdentity();

    // przesunięcie do punktu (0, 0, -150)
    glTranslatef(0.0f, 0.0f, -150.0f);

    glPushMatrix();
    // obrót wzgldem osi x i y
    glRotatef(redYRot, 0.0f, 1.0f, 0.0f);
    glRotatef(redXRot, 1.0f, 0.0f, 0.0f);

    // umieszcza źródło światła
    glLightfv(GL_LIGHT1, GL_POSITION, lightPositionR);

    switch (currentColor)
    {
        case 1:           // światło czerwone
        {
            glLightfv(GL_LIGHT1, GL_DIFFUSE, diffuseLightR);
            glLightfv(GL_LIGHT1, GL_POSITION, lightPositionR);
            glLightfv(GL_LIGHT1, GL_SPECULAR, specularLightR);

            // przesuwa źródło światła
            glTranslatef(lightPositionR[0], lightPositionR[1], lightPositionR[2]);
            glColor3f(1.0f, 0.0f, 0.0f);
            break;
        }
        case 2:           // światło zielone
        {
            glLightfv(GL_LIGHT1, GL_DIFFUSE, diffuseLightG);
            glLightfv(GL_LIGHT1, GL_POSITION, lightPositionG);
            glLightfv(GL_LIGHT1, GL_SPECULAR, specularLightG);

            // przesuwa źródło światła
            glTranslatef(lightPositionG[0], lightPositionG[1], lightPositionG[2]);
        }
    }
}
```

```

        glColor3f(0.0f, 1.0f, 0.0f);
        break;
    }
    case 3:           // światło niebieskie
    {
        glLightfv(GL_LIGHT1, GL_DIFFUSE, diffuseLightB);
        glLightfv(GL_LIGHT1, GL_POSITION, lightPositionB);
        glLightfv(GL_LIGHT1, GL_SPECULAR, specularLightB);

        // przesuwa źródło światła
        glTranslatef(lightPositionB[0], lightPositionB[1], lightPositionB[2]);
        glColor3f(0.0f, 0.0f, 1.0f);
        break;
    }
}

// przechowuje atrybuty oświetlenia
glPushAttrib(GL_LIGHTING_BIT);
    glDisable(GL_LIGHTING); // wyłącza oświetlenie podczas rysowania kuli
                            // reprezentującej poruszające się źródło światła
    auxSolidSphere(2.5f);
    glEnable(GL_LIGHTING);
    glPopAttrib();          // odtwarza atrybuty oświetlenia
    glPopMatrix();

// Rysuje obracający się sześciian
glPushMatrix();
    glColor3f(1.0f, 1.0f, 1.0f);
    glRotatef(objectXRot, 1.0f, 0.0f, 0.0f);
    glRotatef(objectYRot, 0.0f, 1.0f, 0.0f);
    glRotatef(objectZRot, 0.0f, 0.0f, 1.0f);
    auxSolidCube(70.0f);
    glPopMatrix();

glFlush();
SwapBuffers(g_HDC);      // przełącza bufory

// zwiększa liczniki obrotu
objectXRot += 0.01f;
objectYRot += 0.02f;
objectZRot += 0.01f;

redXRot += 0.3f;
redYRot += 0.1f;
}

```

Kod używanej dotąd wersji procedury okienkowej WndProc() wzbogacony został tym razem o następujące wiersze:

```

case WM_KEYDOWN:           // klawisz naciśnięty
    keyPressed[wParam] = true;
    return 0;
    break;

case WM_KEYUP:             // klawisz zwolniony
    keyPressed[wParam] = false;
    return 0;
    break;

```

Poniżej przedstawiona jest pętla przetwarzania komunikatów stanowiąca część funkcji WinMain():

```
while (!done)
{
    PeekMessage(&msg, hwnd, NULL, NULL, PM_REMOVE);

    if (msg.message == WM_QUIT) // nadszedł komunikat WM_QUIT?
    {
        done = true;           // jeśli tak, to aplikacja kończy działanie
    }
    else
    {
        if (keyPressed[VK_ESCAPE])
            done = true;
        else
        {
            if (keyPressed['R'])
                currentColor = 1;

            if (keyPressed['G'])
                currentColor = 2;

            if (keyPressed['B'])
                currentColor = 3;

            if ((keyPressed['S']) && (spotEnabled))
            {
                spotEnabled = false;
                glDisable(GL_LIGHT0);
            }
            else if ((keyPressed['S']) && (!spotEnabled))
            {
                spotEnabled = true;
                glEnable(GL_LIGHT0);
            }
        }

        Render();

        // tłumaczy komunikat i umieszcza go w kolejce
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
} // koniec pętli while
```

Zaprezentowany kod nie został zoptymalizowany pod kątem efektywności wykonania. Wystarczy zmienić jedynie kilka wierszy programu, aby znacznie przyspieszyć jego działanie. Zadanie to pozostawione zostanie jako ćwiczenie do wykonania. Koniecznymi modyfikacjami będą na pewno zmiany, które należy wprowadzić wewnątrz instrukcji switch funkcji Render(). Określenie koloru nie musi być wykonywane w niej za każdym razem, gdy rysowana jest cena. Aby natomiast uzyskać wrażenie ruchu źródła światła, jego pozycja musi być zmieniana za każdym razem za pomocą odpowiednich przekształceń, co ilustruje poniższy fragment kodu:

```
// obrót względem osi x i y
glRotatef(redYRot, 0.0f, 1.0f, 0.0f);
glRotatef(redXRot, 1.0f, 0.0f, 0.0f);
```

```
// umieszcza źródło światła
glLightfv(GL_LIGHT1, GL_POSITION, lightPositionR);
```

Gdyby źródło to tworzyło strumień światła, to konieczne byłoby także określenie jego kierunku za pomocą właściwości `GL_SPOT_DIRECTION`. Ponieważ jednak w tym przykładzie używa się zwykłego, pozycyjnego źródła światła, to wystarczy jedynie określić jego położenie za pomocą właściwości `GL_POSITION`.

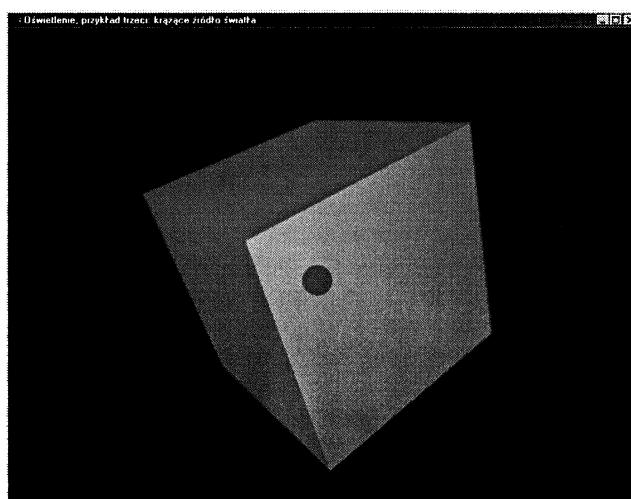
Podczas rysowania kuli reprezentującej poruszające się źródło światła należy skorzystać z funkcji `glPushAttrib()`. Nie trzeba tu szczegółowo omawiać jej działania. Podobnie jak dla przekształceń modelowania, rzutowania i odwzorowań tekstur, także i dla tworzenia grafiki OpenGL posiada odpowiedni stos pozwalający przechować bieżący stan maszyny OpenGL. Przekazując funkcji `glPushAttrib()` parametr `GL_LIGHTING_BIT` używa się przechowywanie na tym stosie wszystkich parametrów związanych z oświetleniem tworzonej sceny. Dzięki temu można wyłączyć na chwilę oświetlenie i narysować kulę bez jakichkolwiek efektów świetlnych. Następnie włącza się oświetlenie i przywraca jego charakterystykę pobierając zachowane na stanie atrybuty za pomocą funkcji `glPopAttrib()`.

Program ten pokazuje także sposób obsługi poleceń wydawanych przez użytkownika za pomocą klawiatury. Definiuje on w tym celu tablicę wartości logicznych o rozmiarze 255 odpowiadającym liczbie wszystkich kodów ASCII, które można wprowadzić za pomocą klawiatury. Jeśli użytkownik naciśnie klawisz *A*, to kod obsługi komunikatu `WM_KEYDOWN` umieszczony w funkcji `WndProc()` nada elementowi `keyPressed[65]` wartość `true`, ponieważ 65 jest kodem ASCII litery *A*.

Rozwiązanie to umożliwia sprawdzenie wewnątrz pętli komunikatów o tym, które klawisze zostały naciśnięte i wykonanie odpowiedniego kodu. Omawiany program pozwala w ten sposób użytkownikowi zmieniać kolor światła emitowanego przez poruszające się źródło. Naciskając klawisz *S* można wyłączyć strumień światła padający na sześcian, a za pomocą klawiszy *R*, *G* i *B* zmieniać kolor światła wysyłanego przez poruszające się źródło na czerwony, zielony lub niebieski. Rysunek 6.13 ilustruje działanie programu.

**Rysunek 6.13.**

Poruszające się źródło światła oświetlające obracający się sześcian



Efektem często stosowanym we współczesnych grach są flesze oświetlające obiekty, na które spogląda obserwator. Ich implementacja polega na umieszczeniu źródła światła na stałej pozycji względem obserwatora. W tym celu pozycję tę należy określić w układzie współrzędnych obserwatora. Najpierw należy załadować macierz jednostkową do macierzy modelowania, a następnie umieścić źródło światła w początku układu współrzędnych. Jeśli nie zostanie określona przy tym orientacja tego źródła, to uzyskany będzie efekt latarni oświetlającej scenę z pozycji kamery. Lepszy efekt uzyska się kierując źródło światła wzdłuż ujemnej części osi z. Ponieważ pozycja tego źródła jest stała, to wystarczy zdefiniować ją raz podczas inicjacji grafiki OpenGL. Dzięki temu podczas tworzenia kolejnych scen nie trzeba już więcej zajmować się tworzeniem efektu flesza.

## Łączenie kolorów

Łączenie kolorów pozwala uzyskać efekt przezroczystości. Dzięki temu grafika OpenGL może symulować obiekty świata rzeczywistego, przez które można widzieć inne obiekty — na przykład szkło lub wodę.

Przy łączaniu kolorów znajduje zastosowanie wartość współczynnika alfa, który pojawił się wiele razy przy omawianiu funkcji OpenGL. Łączenie kolorów polega na utworzeniu nowego koloru na podstawie kolorów obiektu przesłanianego i przesłaniającego. Wykonywane jest zwykle na składowych modelu RGB przy wykorzystaniu współczynnika alfa reprezentującego przesłanianie. Im mniejsza jego wartość, tym większe będzie wrażenie przezroczystości obiektu przesłaniającego. W dalszej części rozdziału obiekt przesłaniający będzie nazywany *źródłem*, a obiekt przesłany *celem*.

Aby korzystać z łączenia kolorów w OpenGL, należy je uaktywnić przekazując funkcji  `glEnable()` parametr `GL_BLEND`. Następnie wywołuje się funkcję `glBlendFunc()`, której trzeba przekazać definicję funkcji łączenia źródła z celem. Domyślne funkcje łączenia odpowiadają wywołaniu funkcji `glBlendFunc()` z parametrem `GL_ONE` dla źródła i `GL_ZERO` dla celu. Tabela 6.6 przedstawia dostępne funkcje łączenia źródła, a tabela 6.7 funkcję łączenia celu.

Tabela 6.6. Funkcje łączenia źródła

Funkcja	Opis
<code>GL_ZERO</code>	Kolorem źródła jest $(0, 0, 0, 0)$
<code>GL_ONE</code>	Wykorzystuje bieżący kolor źródła
<code>GL_DST_COLOR</code>	Mnoży kolor źródła przez kolor celu
<code>GL_ONE_MINUS_DST_COLOR</code>	Mnoży kolor źródła przez dopełnienie koloru celu, czyli przez $(1, 1, 1, 1) - \text{kolor celu}$
<code>GL_SRC_ALPHA</code>	Mnoży kolor źródła przez wartość alfa źródła
<code>GL_ONE_MINUS_SRC_ALPHA</code>	Mnoży kolor źródła przez dopełnienie wartości alfa źródła, czyli przez $(1 - \text{wartość alfa źródła})$
<code>GL_DST_ALPHA</code>	Mnoży kolor źródła przez wartość alfa celu
<code>GL_ONE_MINUS_DST_ALPHA</code>	Mnoży kolor źródła przez dopełnienie wartości alfa celu, czyli przez $(1 - \text{wartość alfa celu})$
<code>GL_SRC_ALPHA_SATURATE</code>	Mnoży kolor źródła przez mniejszą z wartości: alfa źródła lub dopełnienie alfa celu

**Tabela 6.7.** Funkcje łączenia celu

Funkcja	Opis
GL_ZERO	Kolorem celu jest $(0, 0, 0, 0)$
GL_ONE	Wykorzystuje bieżący kolor celu
GL_SRC_COLOR	Mnoży kolor celu przez kolor źródła
GL_ONE_MINUS_SRC_COLOR	Mnoży kolor celu przez dopełnienie koloru źródła, czyli przez $[(1, 1, 1, 1) - \text{kolor źródła}]$
GL_SRC_ALPHA	Mnoży kolor celu przez wartość alfa źródła
GL_ONE_MINUS_SRC_ALPHA	Mnoży kolor celu przez dopełnienie wartości alfa źródła, czyli przez $(1 - \text{wartość alfa źródła})$
GL_DST_ALPHA	Mnoży kolor celu przez wartość alfa celu
GL_ONE_MINUS_DST_ALPHA	Mnoży kolor celu przez dopełnienie wartości alfa celu, czyli przez $(1 - \text{wartość alfa celu})$
GL_SRC_ALPHA_SATURATE	Mnoży kolor celu przez mniejszą z wartości: alfa źródła lub dopełnienia alfa celu

W praktyce aplikacje używają niewielu z wymienionych w tabelach funkcji. Dla uzyskania efektu przezroczystości stosuje się funkcję GL\_SRC\_ALPHA dla źródła i funkcję GL\_ONE\_MINUS\_SRC\_ALPHA dla celu. Z pozostałymi funkcjami można poeksperymentować w celu stworzenia innych efektów.

## Przezroczystość

Efekt przezroczystości uzyskuje się stosując kombinację funkcji źródła GL\_SRC\_ALPHA z funkcją celu GL\_ONE\_MINUS\_SRC\_ALPHA. Poniższy fragment kodu konfiguruje sposób łączenia kolorów tak, by można było uzyskać efekt przezroczystości:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Przy tak określonych funkcjach łączenia kolor źródła nakładany jest w stopniu określonym przez współczynnik alfa na kolor pikseli celu.

Teraz należy przyjrzeć się przykładowi zastosowania przezroczystości. Poniższy fragment kodu wyświetla dwa nakładające się wielokąty dla wartości współczynnika alfa równej 0.6. Jak widać, kolejność nakładania wielokątów ma znaczenie dla końcowego efektu:

```
bool leftFirst = true; // true = rysuje najpierw lewy wielokąt
// false = rysuje najpierw prawy wielokąt

// Initialize
// opis: inicjuje OpenGL
void Initialize()
{
    glEnable(GL_BLEND); // aktywuje łączenie kolorów

    // wybiera funkcje dające efekt przezroczystości
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

```
glShadeModel(GL_SMOOTH);           // wybiera model cieniowania gładkiego

// tło w kolorze czarnym
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
}

// DrawLeftPoly()
// opis: rysuje lewy wielokąt
void DrawLeftPoly()
{
    glColor4f(0.8f, 0.9f, 0.7f, 0.6); // wartość współczynnika alfa = 0.6
    glBegin(GL_QUADS);
        glVertex3f(-10.0f, -10.0f, 0.0f);
        glVertex3f(0.0f, -10.0f, 0.0f);
        glVertex3f(0.0f, 10.0f, 0.0f);
        glVertex3f(-10.0f, 10.0f, 0.0f);
    glEnd();
}

// DrawRightPoly()
// opis: rysuje prawy wielokąt
void DrawRightPoly()
{
    glColor4f(0.0f, 0.5f, 0.5f, 0.6); // wartość współczynnika alfa = 0.6
    glBegin(GL_QUADS);
        glVertex3f(0.0f, -10.0f, 0.0f);
        glVertex3f(10.0f, -10.0f, 0.0f);
        glVertex3f(10.0f, 10.0f, 0.0f);
        glVertex3f(0.0f, 10.0f, 0.0f);
    glEnd();
}

// Render
// opis: rysuje scenę
void Render()
{
    // opróżnia bufor ekranu i głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    if (angle >= 359.9f)
        angle = 0.0f;

    angle += 0.1;                  // zwiększa kąt obrotu

    glTranslatef(0.0f, 0.0f, -40.0f);

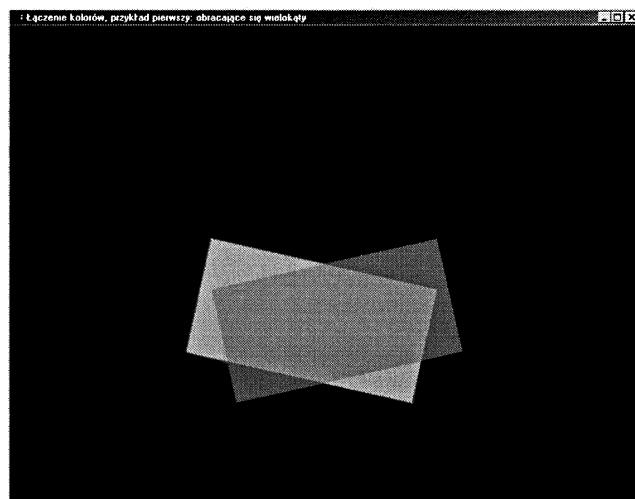
    // obraca i rysuje wielokąt przesłany (cel)
    glPushMatrix();
        glRotatef(angle, 0.0f, 0.0f, 1.0f);
        if (leftFirst)
            DrawLeftPoly();
        else
            DrawRightPoly();
    glPopMatrix();
}
```

```
// obraca i rysuje wielokąt przesłaniający (źródło)
glPushMatrix();
    glRotatef(angle, 0.0f, 0.0f, -1.0f);
    if (leftFirst)
        DrawRightPoly();
    else
        DrawLeftPoly();
glPopMatrix();

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

Działanie powyższego fragmentu kodu ilustruje rysunek 6.14. Przykład ten pokazuje, że łączenie kolorów w przypadku obiektów znajdujących się w takiej samej odległości od obserwatora jest stosunkowo proste.

**Rysunek 6.14.**  
Łączenie kolorów  
dwóch wielokątów



Jednak podczas tworzenia prawdziwie trójwymiarowej grafiki zadanie łączenia kolorów nieco się komplikuje. Najważniejsza różnica polega na konieczności włączenia testowania głębi za pomocą funkcji glEnable(), której przekazuje się wartość GL\_DEPTH\_TEST:

```
glEnable(GL_DEPTH_TEST);
```

Testowanie głębi jest konieczne, by obiekty znajdujące się bliżej przesłaniały obiekty położone dalej od obserwatora. Bufor głębi służy do śledzenia odległości pomiędzy obserwatorem i obiektem, do którego należy dany piksel na ekranie. Jeśli pojawi się inny obiekt znajdujący się bliżej obserwatora, to kolor piksela ulegnie zmianie, a w buforze głębi zostanie umieszczona nowa wartość opisująca odległość obiektu, do którego należy teraz piksel. Takie rozwiązanie pozwala maszynie OpenGL ukrywać dalsze obiekty za nieprzezroczystymi obiekttami, które znajdują się bliżej.

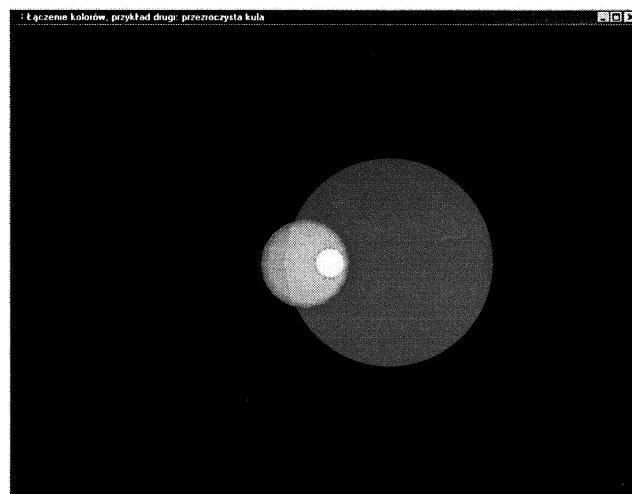
Aby właściwie wykonać operację połączenia kolorów, trzeba włączać i wyłączać bufor głębi podczas rysowania sceny. Najpierw rysuje się wszystkie nieprzezroczyste obiekty przy włączonym buforze głębi. Następnie przełącza się bufor głębi za pomocą funkcji

`glDepthMask()` w tryb, w którym możliwy jest tylko odczyt bufora. W ten sposób można zapobiec modyfikacji zawartości bufora głębi uzyskanej podczas rysowania nieprzezroczystych obiektów. Można teraz rysować obiekty przezroczyste bez obaw, że zmodyfikuje to dotychczasową zawartość bufora. Podczas rysowania obiektów przezroczystych nadal ich odległość od obserwatora porównywana jest z zawartością bufora głębi. Dzięki temu obiekty przezroczyste, które znajdują się za obiektami nieprzezroczystymi nie są rysowane. Jeśli natomiast obiekty przezroczyste znajdują się przed obiektami nieprzezroczystymi, to wykonywane będzie łączenie kolorów. Aby ustawić bufor głębi w tryb, w którym możliwy będzie tylko jego odczyt, należy przekazać funkcji `glDepthMask()` wartość `GL_FALSE`. Aby przywrócić normalny tryb działania bufora, należy przekazać funkcji `glDepthMask()` wartość `GL_TRUE`.

Poniższy program stanowi ilustrację omówionego sposobu działania. Rysunek 6.15 prezentuje efekt działania programu, który wyświetla dwie kule — przezroczystą i nieprzezroczystą — krążące wokół jeszcze jednej, mniejszej kuli. Ta ostatnia kula reprezentuje położenie pozycyjnego źródła światła, które odbijane jest przez krążące kule.

**Rysunek 6.15.**

Efekt przezroczystości kuli uzyskany przy zastosowaniu funkcji `glDepthMask()`



```

float lightPosition[] = { 0.0f, 0.0f, 1.0f, 0.0f }; // kierunek globalnego oświetlenia

float diffuseLight[] = { 1.0f, 1.0f, 1.0f, 1.0f }; // światło rozproszone
float diffuseMat[] = { 1.0f, 1.0f, 1.0f, 1.0f }; // materiał w świetle rozproszonym

float ballDiffuse[] = { 0.5f, 0.5f, 0.0f, 1.0f }; // światło rozproszone źródła-kuli
float ballSpecular[] = { 1.0f, 1.0f, 1.0f, 1.0f }; // światło odbicia źródła-kuli
float ballPosition[] = { 0.0f, 0.0f, 0.0f, 1.0f }; // położenie źródła-kuli

// Initialize
// opis: inicjuje OpenGL
void Initialize()
{
    // włącza oświetlenie, bufor głębi oraz ukrywanie tylnych stron wielokątów
    glEnable(GL_LIGHTING);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
}

```

```
// określa właściwości globalnego światła rozproszonego
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
glEnable(GL_LIGHT0);

// określa właściwości światła źródła-kuli
glLightfv(GL_LIGHT1, GL_DIFFUSE, ballDiffuse);
glLightfv(GL_LIGHT1, GL_SPECULAR, ballSpecular);
glEnable(GL_LIGHT1);

// określa wygląd materiału obiektów w świetle rozproszonym
glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuseMat);

// włącza śledzenie kolorów
glEnable(GL_COLOR_MATERIAL);

// wybiera model cieniowania gładkiego
glShadeModel(GL_SMOOTH);

// tło w kolorze czarnym
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
}

// Render
// opis: rysuje scenę
void Render()
{
    // opróżnia bufor ekranu i bufor głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    // obie kule krążą z tą samą prędkością
    if (angle >= 359.9f)
        angle = 0.0f;
    angle += 0.1;

    // przesunięcie do tyłu o 15 jednostek
    glTranslatef(0.0f, 0.0f, -15.0f);

    // określa pozycję centralnej kuli reprezentującej źródło światła
    glLightfv(GL_LIGHT1, GL_POSITION, ballPosition);

    // rysuje centralną kulę
    glPushMatrix();
    glColor3f(1.0f, 1.0f, 0.0f);
    glTranslatef(ballPosition[0], ballPosition[1], ballPosition[2]);
    auxSolidSphere(0.5f);
    glPopMatrix();

    // rysuje nieprzezroczystą kulę
    glPushMatrix();
    glRotatef(angle, 0.0f, 1.0f, 0.0f);
    glTranslatef(0.0f, 0.0f, 6.0f);
    glColor4f(1.0f, 0.2f, 0.2f, 1.0f);
    auxSolidSphere(2.0f);
    glPopMatrix();
```

```
// aktywuje łączenie kolorów
glEnable(GL_BLEND);

// włącza tryb "tylko-do-odczytu" bufora głębi
glDepthMask(GL_FALSE);

// wybiera funkcje łączenia by uzyskać efekt przezroczystości
glBlendFunc(GL_SRC_ALPHA, GL_ONE);

// rysuje przezroczystą kulę
glPushMatrix();
    glRotatef(angle, 0.0f, 1.0f, 0.0f);
    glTranslatef(0.0f, 0.0f, -6.0f);
    glColor4f(0.0f, 0.5f, 0.5f, 0.3f);
    auxSolidSphere(2.0f);
glPopMatrix();

// przełącza bufor głębi z powrotem do normalnego trybu pracy
glDepthMask(GL_TRUE);

// wyłącza łączenie kolorów
glDisable(GL_BLEND);

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

Jak łatwo zauważyc program rysuje wszystkie nieprzezroczyste obiekty, zanim włączy łączenie kolorów. Następnie przełącza bufor w tryb „tylko-do-odczytu” i rysuje przezroczyste obiekty. Program ten stosuje także kilka efektów związanych z oświetleniem, aby pokazać ich wpływ na łączenie kolorów i efekt przezroczystości.

Jako że wartość współczynnika alfa dla przezroczystej kuli wynosi 0.3, przenika ją większa część światła. Gdyby nie źródło światła w postaci centralnej kuli, to przezroczysta kula byłaby bardzo ciemna. Trzeba przypomnieć tu o tym, że im bardziej współczynnik alfa zbliża się do wartości 1.0, tym mniej przezroczysty staje się obiekt. Gdy zmniejszy się wartość współczynnika alfa, obiekt zyska na przezroczystości. Wartość 0.0 opisuje obiekt doskonale przezroczysty, a wartość 1.0 obiekt zupełnie nieprzezroczysty.

## Podsumowanie

Światło posiada jednocześnie naturę cząstki i fali. Siatkówka oka odbiera światło jako kombinację składowej czerwonej, zielonej i niebieskiej. Podobnie drobiny luminoforów na ekranie monitora emitują światło w kolorze czerwonym, zielonym i niebieskim.

Głębia koloru określa liczbę kolorów, które są do dyspozycji. Zależy ona od rozmiaru bufora koloru. 8-bitowy bufor koloru umożliwia rozróżnienie  $2^8$ , czyli 256 kolorów. Obecnie dostępne karty grafiki umożliwiają zwykle pracę z 8-, 16-, 24- lub 32-bitowym buforem koloru.

OpenGL definiuje kolory za pomocą intensywności składowej czerwonej, zielonej i niebieskiej. Intensywność ta może przyjmować wartości z przedziału od 0.0 (brak składowej) do 1.0 (pełna intensywność).

Cieniowanie może być *płaskie* lub *gładkie*. Płaskie cieniowanie polega na wypełnieniu wielokąta jednym kolorem, który najczęściej jest kolorem ostatniego z wierzchołków. Bardziej realistyczny efekt oferuje cieniowanie gładkie zwane też *cieniowaniem Gouraud*, które interpoluje kolor wypełnienia pomiędzy wierzchołkami wielokąta.

OpenGL modeluje oświetlenie za pomocą czterech komponentów: światła otoczenia, światła rozproszonego, światła odbicia i światła emisji. OpenGL wyznacza kolor materiału na podstawie stopnia odbicia składowej czerwonej, zielonej i niebieskiej światła.

Normalne używane są przez OpenGL do wyznaczania oświetlenia płaszczyzny. Ze względu na efektywność obliczeń maszynie OpenGL przekazuje się normalne jednostkowe.

*OpenGL Programming Guide* wyróżnia cztery etapy tworzenia oświetlenia w OpenGL.

1. Wyznaczenie wektorów normalnych we wszystkich wierzchołkach każdego obiektu. Normalne te określają orientację obiektów względem źródeł światła.
2. Utworzenie, wybranie i określenie pozycji źródła światła.
3. Utworzenie i wybranie modelu oświetlenia. Model ten definiuje światło otoczenia oraz położenie obserwatora uwzględniane w obliczeniach oświetlenia.
4. Zdefiniowanie właściwości materiałów obiektów znajdujących się na scenie.

Źródła światła można przesuwać i obracać w przestrzeni tak samo jak inne obiekty OpenGL. Informacja, którą przekazuje się funkcji `glLight*`() definiując położenie źródła lub kierunek światła, jest następnie przekształcana za pomocą bieżącej macierzy modelowania.

Łączenie kolorów pozwala uzyskać efekt przezroczystości. Efekt ten otrzymuje się tworząc kombinację funkcji łączenia źródła `GL_SRC_ALPHA` z funkcją łączenia celu `GL_ONE_MINUS_SRC_ALPHA`.

Za pomocą funkcji `glDepthMask()` przełącza się bufor głębi w tryb „tylko-do-odczytu”, aby rysować przezroczyste obiekty.

## Rozdział 7.

# Mapy bitowe i obrazy w OpenGL

W rozdziale tym na chwilę będzie trzeba porzucić trójwymiarowy świat, aby zapoznać się z *grafiką rastrową* stosującą dwuwymiarowe tablice pikseli. Trzeba bowiem zapoznać się z funkcjami OpenGL działającymi na mapach bitowych i obrazach. Przedstawione więc zostaną sposoby odczytu i zapisu plików graficznych w dwóch formatach: stosowanym w systemie Windows formacie *BMP* (*.bmp*) oraz formacie *Targa* (*.tga*).

W rozdziale tym omówione zostaną:

- ◆ mapy bitowe w OpenGL;
- ◆ mapy bitowe systemu Windows;
- ◆ pliki graficzne formatu *Targa*.

## Mapy bitowe w OpenGL

OpenGL definiuje *mapę bitową* jako dwuwymiarową tablicę pikseli, w której każdy piksel opisany jest za pomocą pojedynczego bitu. Mapy bitowe wykorzystywane są jako maski podczas rysowania prostokątnych obszarów ekranu oraz w celu definiowania znaków czcionek ekranowych. Można przyjąć na przykład, że zdefiniowana została mapa bitowa o rozmiarach  $16 \times 16$  pokazana na rysunku 7.1. Podczas jej rysowania piksel uzyska bieżący kolor określony wcześniej za pomocą funkcji `glColor3f()` pod warunkiem, że odpowiadająca mu pozycja mapy posiada wartość 1. W przypadku wartości 0 piksel pozostawiany jest bez zmian.

Czcionki ekranowe omówione zostaną dokładniej w rozdziale 11. Teraz wystarczy jedynie pokazanie tego, w jaki sposób stosując funkcje `glBitmap*`() i `glRasterPos*`() można wyświetlić pojedynczy znak czcionki.

**Rysunek 7.1.***Mapa bitowa**o rozmiarach 16×16*

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	
0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	
0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	
0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	
0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	
0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	
0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	
0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	
0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

## Umieszczanie map bitowych

Funkcja `glRasterPos*`() pozwala określić położenie rysowanej mapy bitowej na ekranie. Przekazywane funkcji współrzędne wyznaczają położenie lewego, dolnego narożnika mapy bitowej. Poprzez przekazanie funkcji `glRasterPos*`() współrzędnych (30, 10) można umieścić dolny, lewy narożnik rysowanej mapy bitowej w punkcie o współrzędnych (30, 10). Funkcja ta posiada następujące wersje:

```
void glRasterPos[234][sifd](TYPE x, TYPE y, TYPE z, TYPE w);
void glRasterPos[234][sifd]v(TYPE *coords);
```

Dla tego przykładu wywołanie jej będzie mieć postać:

```
glRasterPos2i (30, 10);
```

Jeśli dla grafiki dwuwymiarowej nie zostanie wyspecyfikowana macierz modelowania i macierz rzutowania, to współrzędne przekazywane funkcji `glRasterPos*`() zamieniane będą na współrzędne ekranowe. Okno grafiki dwuwymiarowej definiuje się podobnie jak w przypadku grafiki trójwymiarowej. Zamiast użycia funkcji `gluProjection()` stosuje się jednak funkcje `glOrtho()` lub `gluOrtho2D()`. Poniżej przedstawiony został przykład definicji okna grafiki za pomocą funkcji `glOrtho()`, gdzie zmienna `width` określa szerokość okna, a zmienna `height` jego wysokość:

```
glViewport(0, 0, width, height); // określa nowe rozmiary okna
glMatrixMode(GL_PROJECTION); // wybiera macierz rzutowania
glLoadIdentity(); // resetuje macierz rzutowania

// definiuje okno grafiki dwuwymiarowej
glOrtho(0.0f, width - 1.0, 0.0, height - 1.0, -1.0, 1.0);

glMatrixMode(GL_MODELVIEW); // wybiera macierz modelowania
glLoadIdentity(); // resetuje macierz modelowania
```

Przekazując funkcji `glGetBooleanv()` wartość `GL_CURRENT_RASTER_POSITION_VALID` można dowiedzieć się, czy wybrana została dozwolona pozycja rysowania mapy bitowej. Jeśli funkcja `glGetBooleanv()` zwróci wartość `false`, to będzie znaczyć, że wybrana pozycja nie jest dozwolona.

## Rysowanie mapy bitowej

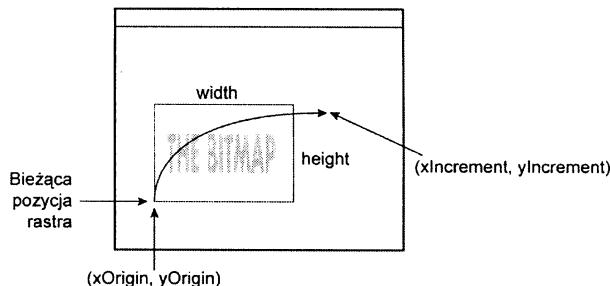
Po wybraniu pozycji mapę bitową rysuje się za pomocą funkcji `glBitmap()` zdefiniowanej następująco:

```
void glBitmap(GLsizei width, GLsizei height, GLfloat xOrigin, GLfloat yOrigin, GLfloat xIncrement, GLfloat yIncrement, const GLubyte *bitmap);
```

Funkcja ta rysuje mapę o podanej szerokości i wysokości w punkcie o współrzędnych ( $xOrigin$ ,  $yOrigin$ ) w bieżącym układzie współrzędnych rastra. Parametry  $xIncrement$  i  $yIncrement$  określają wartości, o które zwiększone zostaną bieżące współrzędne rastra po narysowaniu mapy bitowej. Znaczenie parametrów funkcji `glBitmap()` ilustruje rysunek 7.2.

**Rysunek 7.2.**

Znaczenie parametrów funkcji `glBitmap()`



Wadą map bitowych w OpenGL jest to, że nie mogą one być ani obracane, ani powiększane. Operacje te w OpenGL można wykonywać natomiast w przypadku map pikseli oraz obrazów. Przedstawione one zostaną w dalszej części rozdziału.

## Przykład zastosowania mapy bitowej

Omówiony teraz zostanie przykład programu, który tworząc kolejne ramki grafiki rysować będzie mapę bitową o rozmiarach  $16 \times 16$  w 50 losowo wybranych pozycjach.

Mapa ta definiować będzie literę *A* za pomocą poniższej tablicy:

```
unsigned char letterA[] = {  
    0xC0, 0x03,  
    0xC0, 0x03,  
    0xC0, 0x03,  
    0xC0, 0x03,  
    0xC0, 0x03,  
    0xC0, 0x03,  
    0xDF, 0xFB,  
    0x7F, 0xFE,  
    0x60, 0x06,  
    0x30, 0x0C,  
    0x30, 0x0C,  
    0x18, 0x18,  
    0x18, 0x18,  
    0x0C, 0x30,  
    0x0C, 0x30,  
    0x07, 0xE0,  
    0x07, 0xE0  
};
```

Wygląd zdefiniowanej mapy bitowej pokazuje rysunek 7.3. Należy zwrócić uwagę na to, że mapę bitową definiuje się w tablicy „do góry nogami” w stosunku do sposobu, w jaki zostanie ona przedstawiona na ekranie.

### Rysunek 7.3.

*Definicja  
mapy bitowej  
reprezentującej  
literę A*

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
2	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0
4	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0
5	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0
6	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
7	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0
8	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1
9	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
10	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
11	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
12	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
13	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
14	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
15	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Grafika będzie jak zwykle tworzona za pomocą funkcji Render(), która tym razem zostanie zdefiniowana następująco:

```
void Render()
{
    // opróżnia bufory ekranu i głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // wybiera wyrównanie do pełnych bajtów
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    // wybiera kolor biały
    glColor3f(1.0f, 1.0f, 1.0f);

    // rysuje 50 razy mapę bitową 16 x 16 litery A
    // w losowo wybranych pozycjach okna 800 x 600
    for (int numA = 0; numA < 70; numA++)
    {
        glRasterPos2i(rand() % 800, rand() % 600);
        glBitmap(16, 16, 0.0, 0.0, 0.0, 0.0, letterA);
    }

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

Przed rozpoczęciem rysowania funkcja Render() wybiera kolor biały jako bieżący. Pozycja, na której rysowana jest mapa bitowa za pomocą funkcji glBitmap(), odpowiada pozycji określonej za pomocą funkcji glRasterPos2i(). Po narysowaniu kolejnej mapy bitowej funkcja glBitmap() nie zmienia bieżącej pozycji rastra, ponieważ jej parametry xIncrement i yIncrement mają wartość 0.

Okno grafiki dwuwymiarowej definiuje się za pomocą funkcji glOrtho():

```
glViewport(0, 0, width, height); // określa nowe wymiary okna grafiki
glMatrixMode(GL_PROJECTION);    // wybiera macierz rzutowania
glLoadIdentity();                // i resetuje ją

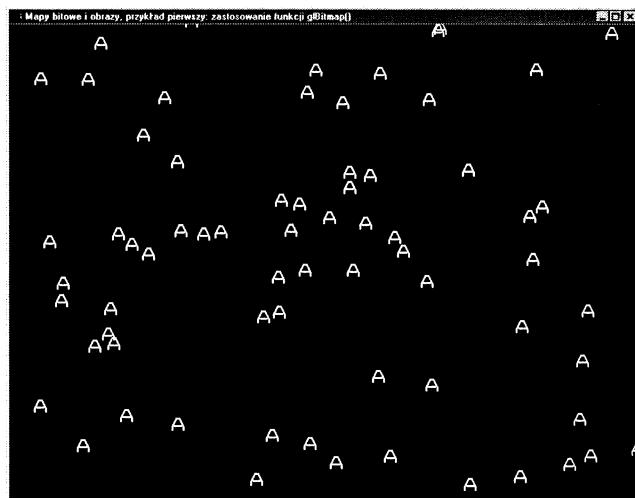
// rzutowanie ortograficzne
glOrtho(0.0f, width - 1.0, 0.0, height - 1.0, -1.0, 1.0);

glMatrixMode(GL_MODELVIEW);      // wybiera macierz modelowania
glLoadIdentity();                // i resetuje ją
```

Wynik działania tego prostego przykładu zastosowania mapy bitowej ilustruje rysunek 7.4.

#### Rysunek 7.4.

Przykład  
zastosowania  
funkcji `glBitmap()`



## Wykorzystanie obrazów graficznych

Przy tworzeniu grafiki rastrowej w OpenGL częściej korzysta się z *obrazów graficznych* niż z map bitowych. Od tych ostatnich różni je przede wszystkim bogatszy opis pikseli za pomocą wartości modelu kolorów RGB.

Jako że w OpenGL można manipulować pojedynczymi pikselami obrazów graficznych, często stosuje się w ich przypadku także nazwę *mapy pikseli*. W rozdziale tym przedstawiony będzie sposób wyświetlania map pikseli na ekranie, natomiast w rozdziale 8. omówione zostaną możliwości ich wykorzystania do tworzenia tekstur pokrywających wielokąty grafiki trójwymiarowej.

## Rysowanie obrazów graficznych

Przy założeniu, że już załadowana została do pamięci mapa pikseli, można użyć funkcji `glDrawPixels()`, aby wyświetlić ją w określonym miejscu okna grafiki. Podobnie jak w przypadku funkcji `glBitmap()` położenie mapy pikseli określa się za pomocą funkcji `glRasterPos*`(). Funkcja `glDrawPixels()` zdefiniowana jest w następujący sposób:

```
void glDrawPixels(GLsizei width, GLsizei height, GLenum format, GLenum type, const GLvoid *pixels);
```

Funkcji tej przekazuje się (kolejno) następujące parametry: szerokość i wysokość obrazu graficznego, format i typ pikseli, dane opisujące piksele. Dopuszczalne formaty pikseli przedstawia tabela 7.1. Najczęściej używa się formatu GL\_RGB, który określa dla każdego piksela wartość składowej czerwonej, zielonej i niebieskiej.

**Tabela 7.1.** Formaty pikseli

Format pikseli	Opis
GL_ALPHA	Piksel opisany przez współczynnik alfa
GL_BGR	Piksel opisany przez składową niebieską, zieloną i czerwoną
GL_BGRA	Piksel opisany przez składową niebieską, zieloną, czerwoną i współczynnik alfa
GL_BLUE	Piksel opisany przez składową niebieską
GL_COLOR_INDEX	Piksel opisany przez indeksowy model kolorów
GL_GREEN	Piksel opisany przez składową zieloną
GL_RED	Piksel opisany przez składową czerwoną
GL_RGB	Piksel opisany przez składową czerwoną, zieloną i niebieską
GL_RGBA	Piksel opisany przez składową czerwoną, zieloną, niebieską i współczynnik alfa

Typy pikseli prezentuje tabela 7.2. Typ piksela definiuje typ danych stosowany do opisu piksela.

**Tabela 7.2.** Typy pikseli

Typ pikseli	Opis
GL_BITMAP	Pojedynczy bit (0 lub 1)
GL_BYTE	Liczba całkowita ze znakiem, 8 bitów
GL_UNSIGNED_BYTE	Liczba całkowita bez znaku, 8 bitów
GL_SHORT	Liczba całkowita ze znakiem, 16 bitów
GL_UNSIGNED_SHORT	Liczba całkowita bez znaku, 16 bitów
GL_INT	Liczba całkowita ze znakiem, 32 bity
GL_UNSIGNED_INT	Liczba całkowita bez znaku, 32 bity

Poniżej zaprezentowany został fragment kodu rysujący obraz graficzny za pomocą funkcji `glDrawPixels()` na pozycji o współrzędnych (300, 300) zdefiniowany w buforze wskazywanym przez zmienną `imageData`:

```
unsigned char *imageData;
int imageWidth, imageHeight;
...
glRasterPos2i(300, 300);
glDrawPixels(imageWidth, imageHeight, GL_RGB, GL_UNSIGNED_BYTE, imageData);
```

## Odczytywanie obrazu z ekranu

Często w programach zachodzi konieczność odczytania bieżącej zawartości ekranu i zapamiętania jej na dysku bądź przetworzenia w celu uzyskania efektów specjalnych. W OpenGL zadanie to wykonuje się za pomocą funkcji `glReadPixels()` o następującym prototypie:

```
void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum format,  
GLenum type, GLvoid *pixels);
```

Parametry tej funkcji mają takie samo znaczenie jak w przypadku funkcji `glDrawPixels()`. Uzupełnione zostały o parę współrzędnych ( $x, y$ ) określających lewy, dolny narożnik fragmentu ekranu, który zostanie zapamiętany w buforze `pixels`. Parametry `width` i `height` określają rozmiary tego fragmentu. Parametry określające format i typ pikseli mogą przyjmować te same wartości co w przypadku funkcji `glDrawPixels()` (wymienione w tabelach 7.1 i 7.2).

Poniższy fragment kodu prezentuje sposób umieszczenia w buforze zawartości górnej połowy okna grafiki za pomocą funkcji `glReadPixels()`:

```
void *imageData;  
int screenWidth, screenHeight;  
...  
glReadPixels(0, screenHeight/2, screenWidth, screenHeight/2, GL_RGB, GL_UNSIGNED_BYTE,  
imageData);
```

## Kopiowanie danych ekranu

OpenGL umożliwia także kopiowanie pikseli pomiędzy różnymi obszarami ekranu za pomocą funkcji `glCopyPixels()` zdefiniowanej następująco:

```
glCopyPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum buffer);
```

Funkcja ta kopiuje fragment ekranu w kształcie prostokąta o szerokości `width` i wysokości `height`, którego lewy, dolny wierzchołek posiada współrzędne ( $x, y$ ) i umieszcza go na bieżącej pozycji rastra. Parametr `buffer` może przyjmować jedną z wartości przedstawionych w tabeli 7.3.

**Tabela 7.3.** Rodzaje bufora dla funkcji `glCopyPixels()`

Wartość parametru <code>buffer</code>	Opis
<code>GL_COLOR</code>	Kopiuje zawartość bufora kolorów
<code>GL_DEPTH</code>	Kopiuje zawartość bufora głębi
<code>GL_STENCIL</code>	Kopiuje zawartość bufora powielania

Jednym z typowych zastosowań funkcji `glCopyPixels()` w grach jest wprowadzenie szkła powiększającego lub celownika optycznego broni. Kopując pewien obszar ekranu i powiększając go za pomocą funkcji `glPixelZoom()` uzyskać można właśnie taki efekt.

## Powiększanie, pomniejszanie i tworzenie odbić

OpenGL umożliwia powiększanie i pomniejszanie fragmentów obrazu oraz tworzenie ich odbić. Służy do tego funkcja `glPixelZoom()` posiadająca następujący prototyp:

```
void glPixelZoom(GLfloat xZoom, GLfloat yZoom);
```

Domyślna wartość obu parametrów wynosi 1.0, co oznacza normalny obraz. Wartości z przedziału od 0.0 do 1.0 powodują pomniejszenie obrazu, a większe od 1.0 jego powiększenie. Jeśli dodatkowo wartość taka będzie ujemna, to spowoduje ona odbicie obrazu względem bieżącej pozycji rastra. Poniżej zaprezentowanych zostało kilka przykładów użycia funkcji `glPixelZoom()`:

```
glPixelZoom(-1.0f, -1.0f); // tworzy odbicie w pionie i w poziomie  
glPixelZoom(0.5f, 0.5f); // zmniejsza rozmiary obrazu o połowę  
glPixelZoom(5.0f, 5.0f); // powiększa obraz 5-krotnie w każdym z wymiarów
```

## Upakowanie danych mapy pikseli

Po uruchomieniu tego samego programu tworzącego grafikę OpenGL na różnych komputerach można zaobserwować różną szybkość jego działania. Może to być spowodowane szeregiem różnych czynników. Jednym z nich jest różna prędkość kopiowania danych w zależności od ich wyrównania do granic słów 2-, 4- i 8-bajtowych. Wyrównanie danych opisujących piksele można kontrolować za pomocą funkcji `glPixelStorei()` zdefiniowanej następująco:

```
void glPixelStorei(GLenum pname, TYPE param);
```

W przypadku aktualnych zastosowań parametr *pname* będzie przyjmować wartość `GL_PACK_ALIGNMENT` lub `GL_UNPACK_ALIGNMENT`, natomiast parametr *param* wartości 1, 2, 4 lub 8. Podając `GL_PACK_ALIGNMENT` jako wartość parametru *pname* określa się sposób, w jaki dane opisujące piksele wyrównywane są podczas umieszczania ich w buforze. Natomiast parametr `GL_UNPACK_ALIGNMENT` pozwala określić wyrównanie danych niezbędne do ich właściwego odczytania. Domyślną wartością parametru *param* jest w obu przypadkach wartość 4.

Poniższy przykład informuje OpenGL, że dane umieszczone zostały w kolejnych bajtach:

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

## Mapy bitowe systemu Windows

Jako że opanowana już została umiejętność wykonywania podstawowych operacji na obrazach graficznych, można pokusić się o próbę zastosowania ich do prawdziwych danych. Jako pierwszy omówiony zostanie format plików graficznych *BMP* używany w systemie Windows.

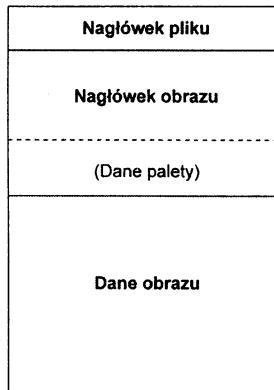
Wielką zaletą plików *BMP* jest to, że mogą być łatwo tworzone, edytowane i przeglądane przez każdego użytkownika systemów rodziny Windows. Ich wadą jest natomiast brak kompresji, który sprawia, że posiadać mogą całkiem spore rozmiary. Jednak brak kompresji oznacza także, że operacje odczytu i zapisu takich plików są bardzo proste.

## Format plików BMP

Jak pokazane zostało na rysunku 7.5, w strukturze pliku *BMP* można wyróżnić trzy części: nagłówek pliku, nagłówek mapy bitowej oraz właściwe dane mapy bitowej.

Rysunek 7.5.

Struktura pliku BMP



Do pobrania nagłówka pliku *BMP* wykorzystać można strukturę BITMAPFILEHEADER zdefiniowaną jak poniżej:

```
typedef struct tagBITMAPFILEHEADER {  
    WORD    bfType;  
    DWORD   bfSize;  
    WORD    bfReserved1;  
    WORD    bfReserved2;  
    DWORD   bfOffBits;  
} BITMAPFILEHEADER;
```

Po wczytaniu nagłówka pliku *BMP* należy koniecznie sprawdzić, czy pole *bfType* struktury BITMAPFILEHEADER posiada wartość 0x4D42. W ten sposób weryfikuje się to, czy dany plik rzeczywiście jest plikiem formatu *BMP*.

Następna część pliku *BMP* zawiera informacje dotyczące mapy bitowej. Informacje te umieszczone są w jednej lub dwóch strukturach w zależności od tego, czy mapa bitowa opisuje piksele za pomocą 8-bitów i posiada paletę kolorów. Ponieważ w tym przykładzie nie jest konieczne korzystanie z plików *BMP* dysponujących własną paletą, wystarczy jedynie przeczytać dane umieszczone w strukturze BITMAPINFOHEADER:

```
typedef struct tagBITMAPINFOHEADER{  
    DWORD    biSize;  
    LONG     biWidth;  
    LONG     biHeight;  
    WORD     biPlanes;  
    WORD     biBitCount;  
    DWORD   biCompression;
```

```
    DWORD      biSizeImage;
    LONG       biXPelsPerMeter;
    LONG       biYPelsPerMeter;
    DWORD      biClrUsed;
    DWORD      biClrImportant;
} BITMAPINFOHEADER;
```

Zawartość tej struktury jest właściwie oczywista. Kłopotów może przysporzyć jedynie pole *biCompression*. Chociaż używane tu pliki *BMP* nie stosują kompresji, to istnieją jednak wersje formatu *BMP*, które mogą stosować kompresję za pomocą algorytmu RLE. Omówienie tego algorytmu wykracza poza tematykę tej książki, więc można ograniczyć się jedynie do sprawdzenia, czy pole *biCompression* posiada wartość *BI\_RGB*, co oznacza brak kompresji.

Ostatnią część struktury pliku *BMP* stanowią właściwe dane obrazu graficznego. Dane te opisują kolejne piksele obrazu w formacie 1-, 4-, 8-, 16- lub 24-bitowym.

## Ładowanie plików BMP

Aby załadować zawartość pliku *BMP* do pamięci, należy odczytać po kolei wszystkie trzy jego części. Po odczytaniu struktury *BITMAPINFOHEADER* można wyznaczyć wielkość bufora pamięci potrzebnego do przechowania obrazu zapisanego w pliku. Po wczytaniu danych do bufora należy dla wszystkich pikseli zamienić miejscami wartości składowej czerwonej ze składową niebieską, aby OpenGL właściwie wyświetlał kolory. Ponizej przedstawiony został przykład implementacji funkcji ładującej plik *BMP* stosujący 24-bitowy opis pikseli:

```
unsigned char *LoadBitmapFile(char *filename, BITMAPINFOHEADER *bitmapInfoHeader)
{
    FILE *filePtr;                      // wskaźnik pliku
    BITMAPFILEHEADER bitmapFileHeader;   // nagłówek pliku
    unsigned char*bitmapImage;          // bufor obrazu
    intimageIdx = 0;                    // licznik bajtów obrazu
    unsigned char tempRGB;              // zmienna zamiany składowych

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

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

```
// ustawia wskaźnik pliku na początku danych opisujących obraz
fseek(filePtr, bitmapFileHeader.bfOffBits, SEEK_SET);

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

// sprawdza, czy pamięć została przydzielona
if (!bitmapImage)
{
    free(bitmapImage);
    fclose(filePtr);
    return NULL;
}

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

// sprawdza, czy operacja powiodła się
if (bitmapImage == NULL)
{
    fclose(filePtr);
    return NULL;
}

// zamienia składowe R i B, aby uzyskać format RGB używany przez OpenGL
for (imageIdx = 0; imageIdx < bitmapInfoHeader->biSizeImage; imageIdx+=3)
{
    tempRGB = bitmapImage[imageIdx];
    bitmapImage[imageIdx] = bitmapImage[imageIdx + 2];
    bitmapImage[imageIdx + 2] = tempRGB;
}

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

Jak pokazuje powyższy przykład załadowanie obrazu graficznego z pliku *BMP* nie jest skomplikowane. Poniżej zaprezentowany został sposób wykorzystania funkcji `LoadBitmapFile()`:

```
BITMAPINFOHEADER bitmapInfoHeader;      // nagłówek obrazu
unsigned char* bitmapData;               // bufor obrazu
...
bitmapData = LoadBitmapFile("test.bmp", &bitmapInfoHeader);
glPixelStorei(GL_UNPACK_ALIGNMENT, 4); // określa sposób wyrównania danych
glRasterPos2i(100,100);                // określa bieżącą pozycję rastra
glDrawPixels(bitmapInfoHeader.biWidth, bitmapInfoHeader.biHeight, GL_RGB,
             GL_UNSIGNED_BYTE, bitmapImage); // wyświetla obraz
```

## Zapis obrazu w pliku BMP

A co w przypadku, kiedy obraz uzyskany za pomocą funkcji `glReadpixels()` będzie trzeba zapisać w pliku *BMP*? W takim przypadku należy indywidualnie wypełnić struktury `BITMAPFILEHEADER` i `BITMAPINFOHEADER`.

Poniżej prezentujemy przykład implementacji funkcji zapisu obrazu graficznego o szerokości i wysokości określonych przez parametry width i height:

```
int WriteBitmapFile(char *filename, int width, int height, unsigned char *imageData)
{
    FILE *filePtr;                      // wskaźnik pliku
    BITMAPFILEHEADER bitmapFileHeader;   // nagłówek pliku
    BITMAPINFOHEADER bitmapInfoHeader;   // nagłówek obrazu
    int imageIdx;                       // indeks obrazu
    unsigned char tempRGB;              // zmieniona zamiana składowych

    // otwiera plik do zapisu w trybie "writing binary"
    filePtr = fopen(filename, "wb");
    if (!filePtr)
        return 0;

    // definiuje nagłówek pliku
    bitmapFileHeader.bfSize = sizeof(BITMAPFILEHEADER);
    bitmapFileHeader.bfType = 0x4D42;
    bitmapFileHeader.bfReserved1 = 0;
    bitmapFileHeader.bfReserved2 = 0;
    bitmapFileHeader.bfOffBits = sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER);

    // definiuje nagłówek obrazu
    bitmapInfoHeader.biSize = sizeof(BITMAPINFOHEADER);
    bitmapInfoHeader.biPlanes = 1;
    bitmapInfoHeader.biBitCount = 24;      // 24-bity
    bitmapInfoHeader.biCompression = BI_RGB; // bez kompresji
    // szerokość * wysokość * (abajty opisu RGB)
    bitmapInfoHeader.biSizeImage = width * abs(height) * 3;
    bitmapInfoHeader.biXPelsPerMeter = 0;
    bitmapInfoHeader.biYPelsPerMeter = 0;
    bitmapInfoHeader.biClrUsed = 0;
    bitmapInfoHeader.biClrImportant = 0;
    bitmapInfoHeader.biWidth = width;       // szerokość
    bitmapInfoHeader.biHeight = height;     // wysokość

    // zamienia składowe z formatu RGB na GBR
    for (imageIdx = 0; imageIdx < bitmapInfoHeader.biSizeImage; imageIdx+=3)
    {
        tempRGB = imageData[imageIdx];
        imageData[imageIdx] = imageData[imageIdx + 2];
        imageData[imageIdx + 2] = tempRGB;
    }

    // zapisuje nagłówek pliku
    fwrite(&bitmapFileHeader, 1, sizeof(BITMAPFILEHEADER), filePtr);

    // zapisuje nagłówek obrazu
    fwrite(&bitmapInfoHeader, 1, sizeof(BITMAPINFOHEADER), filePtr);

    // zapisuje dane obrazu
    fwrite(imageData, 1, bitmapInfoHeader.biSizeImage, filePtr);

    // zamyka plik
    fclose(filePtr);

    return 1;
}
```

Jeśli funkcja ta będzie zastosowana razem z funkcją `glReadPixels()`, to uzyskana zostanie możliwość zapisu zawartości okna grafiki w pliku *BMP*. Poniżej przedstawiony został kod funkcji implementującej takie rozwiązanie:

```
unsigned char *imageData;  
...  
void SaveScreenshot ( int winWidth, int winHeight )  
{  
    imageData = malloc(800*600*3); // przydziela pamięć bufora imageData  
    memset(imageData, 0, 800*600*3); // opróżnia bufor  
  
    // wczytuje obraz okna do bufora  
    glReadPixels(0, 0, 799, 599, GL_RGB, GL_UNSIGNED_BYTE, imageData);  
  
    // zapisuje bufor do pliku  
    WriteBitmapFile("writeout.bmp", 800, 600, (unsigned char*)imageData);  
  
    // zwalnia bufor  
    free(imageData);  
}
```

## Pliki graficzne Targa

Kolejnym omówionym formatem plików graficznych będzie format *Targa*. Format ten nie jest szczególnie skomplikowany. Jego podstawową zaletą w stosunku do formatu *BMP* jest zastosowanie kanału alfa, który umożliwia wykorzystanie plików *Targa* do realizacji odwzorowań tekstur.

### Format plików Targa

Struktura plików *Targa* podzielona została na dwie części: nagłówek i dane. Nagłówek składa się z 12 pól opisanych przez poniższą strukturę:

```
typedef struct tagTARGAFILEHEADER  
{  
    unsigned char imageIDLength; // liczba znaków w polu identyfikacji  
                                // 0 oznacza brak pola identyfikacji  
    unsigned char colorMapType; // typ mapy kolorów, zawsze wartość 0  
    unsigned char imageTypeCode; // wartość 2 = RGB bez kompresji  
                                // wartość 3 = obraz bez kompresji, odcienie szarości  
    short int colorMapOrigin; // początek mapy kolorów, zawsze wartość 0  
    short int colorMapLength; // długość mapy kolorów, zawsze wartość 0  
    short int colorMapEntrySize; // rozmiar elementów mapy kolorów, zawsze wartość 0  
    short int imageXOrigin; // współrzędna x lewego, dolnego narożnika obrazu,  
                            // zawsze wartość 0  
    short int imageYOrigin; // współrzędna y lewego, dolnego narożnika obrazu,  
                            // zawsze wartość 0  
    short int imageWidth; // szerokość obrazu w pikselach  
                          // najpierw bajt mniej znaczący  
    short int imageHeight; // wysokość obrazu w pikselach  
                          // najpierw bajt mniej znaczący  
    unsigned char bitCount; // liczba bitów: 16, 24 lub 32  
    unsigned char imageDescriptor; // wartość 0 x 00 = 24 bity, wartość 0 x 08 = 32-bity  
} TARGAFILEHEADER;
```

Dane obrazu w formacie *Targa* zaczynają się zaraz za nagłówkiem pliku. Pierwszym polem nagłówka, którego zawartością trzeba się zainteresować, jest *imageTypeCode*. Określa ono typ pliku *Targa*. Niektóre z typów plików *Targa* prezentuje tabela 7.4. W podanych tu przykładach wykorzystane zostaną jedynie typy reprezentowane przez wartości 2 i 3.

**Tabela 7.4.** Typy plików *Targa*

Kod	Opis
2	Kolorowy obraz RGB bez kompresji
3	Obraz biało-czarny bez kompresji
10	Skompresowany obraz RGB (algorytm RLE)
11	Skompresowany obraz biało-czarny

Ostatnie cztery pola nagłówka pliku *Targa* pozwalają ustalić rozmiary bufora potrzebnego do wczytania obrazu zapisanego w pliku oraz sposób jego odczytu. Podobnie jak w przypadku plików *BMP* format *Targa* zawiera opis pikseli obrazu w formacie *BGR* lub *BGRA* (odpowiednio dla opisu 24- i 32-bitowego).

## Ładowanie zawartości pliku *Targa*

Trzeba przyjrzeć się zatem bliżej sposobowi, w jaki należy załadować zawartość pliku *Targa*. Poniższą strukturę definiuje się tak, aby można było przechować w niej wszystkie istotne informacje o obrazie zawartym w pliku tego typu:

```
typedef struct
{
    unsigned char imageTypeCode;
    short int     imageWidth;
    short int     imageHeight;
    unsigned char bitCount;
    unsigned char *imageData;
} TGAFILE;
```

Poniżej przedstawiamy przykład implementacji funkcji ładującej obraz graficzny zapisany w pliku *Targa*:

```
int LoadTGAFfile(char *filename, TGAFILE *tgaFile)
{
    FILE *filePtr;
    unsigned char ucharBad;           // nieużywane dane typu unsigned char
    short int   sintBad;             // niewykorzystywane dane typu short int
    long       imageSize;            // rozmiar obrazu TGA
    int        colorMode;            // 4 = RGBA lub 3 = RGB
    long       imageIdx;             // licznik
    unsigned char colorSwap;         // zmieniąca składowych koloru

    // otwiera plik
    filePtr = fopen(filename, "rb");
    if (!filePtr)
        return 0;

    // wczytuje dwa pierwsze nieużywane bajty
    fread(&ucharBad, sizeof(unsigned char), 1, filePtr);
    fread(&ucharBad, sizeof(unsigned char), 1, filePtr);
```

```
// wczytuje typ pliku
fread(&tgaFile->imageTypeCode, sizeof(unsigned char), 1, filePtr);

// używa się jedynie typu 2 (kolorowy) lub 3 (biało-czarny)
if ((tgaFile->imageTypeCode != 2) && (tgaFile->imageTypeCode != 3))
{
    fclose(filePtr);
    return 0;
}

// wczytuje kolejnych 13 nieużywanych bajtów
fread(&sintBad, sizeof(short int), 1, filePtr);
fread(&sintBad, sizeof(short int), 1, filePtr);
fread(&ucharBad, sizeof(unsigned char), 1, filePtr);
fread(&sintBad, sizeof(short int), 1, filePtr);
fread(&sintBad, sizeof(short int), 1, filePtr);

// wczytuje rozmiary obrazu
fread(&tgaFile->imageWidth, sizeof(short int), 1, filePtr);
fread(&tgaFile->imageHeight, sizeof(short int), 1, filePtr);

// wczytuje głębię kolorów
fread(&tgaFile->bitCount, sizeof(unsigned char), 1, filePtr);

// wczytuje 1 bajt nieużywanych danych
fread(&ucharBad, sizeof(unsigned char), 1, filePtr);

// tryb kolorów, 3 = BGR, 4 = BGRA
colorMode = tgaFile->bitCount / 8;
imageSize = tgaFile->imageWidth * tgaFile->imageHeight * colorMode;

// przydziela pamięć na bufor obrazu
tgaFile->imageData = (unsigned char*)malloc(sizeof(unsigned char)*imageSize);

// wczytuje dane obrazu
fread(tgaFile->imageData, sizeof(unsigned char), imageSize, filePtr);

// zmienia opis kolorów z BGR na RGB
// aby zostały prawidłowo zinterpretowane przez OpenGL
for (imageIdx = 0; imageIdx < imageSize; imageIdx += colorMode)
{
    colorSwap = tgaFile->imageData[imageIdx];
    tgaFile->imageData[imageIdx] = tgaFile->imageData[imageIdx + 2];
    tgaFile->imageData[imageIdx + 2] = colorSwap;
}

// zamyka plik
fclose(filePtr);

return 1;
}
```

Funkcję tę stosować można w poniższy sposób:

```
TGAFILE *myTGA;
...
myTGA = (TGAFILE*)malloc(sizeof(TGAFILE));
LoadTGAFfile("test.tga", myTGA);
```

A następnie — podobnie jak w przypadku pliku *BMP* — wyświetla się załadowany obraz za pomocą funkcji `glDrawPixels()`:

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
glRasterPos2i(200,200);
glDrawPixels(myTGA->imageWidth, myTGA->imageHeight, GL_RGB, GL_UNSIGNED_BYTE,
myTGA->imageData);
```

## Zapis obrazów w plikach Targa

Zapis informacji w pliku *Targa* jest równie prosty jak jej odczyt. Wystarczy jedynie wyspecyfikować typ pliku, głębię kolorów i tryb ich opisu. W pozostałych polach nagłówka pliku należy umieścić wartość 0. Zapisując obraz OpenGL w pliku *Targa* trzeba pamiętać o zmianie modelu kolorów z RGB(A) na BGR(A). Należy przyjrzeć się zatem przykładowej implementacji funkcji zapisu obrazu w pliku *Targa*:

```
int WriteTGAFile(char *filename, short int width, short int height, unsigned char*
imageData)
{
    unsigned char byteSkip;           // zmienne do wypełniania nieużywanych pól
    short int    shortSkip;
    unsigned char imageType;          // typ zapisywanej obrazu
    int          colorMode;
    unsigned char colorSwap;
    int          imageIdx;
    unsigned char bitDepth;
    long         imageSize;
    FILE *filePtr;

    // otwiera plik do zapisu
    filePtr = fopen(filename, "wb");
    if (!filePtr)
    {
        fclose(filePtr);
        return 0;
    }

    imageType = 2;                   // RGB bez kompresji
    bitDepth = 24;                  // 24-bitowa głębia kolorów
    colorMode = 3;                  // tryb RGB

    byteSkip = 0;
    shortSkip = 0;

    // zapisuje dwa bajty nieużywanych danych
    fwrite(&byteSkip, sizeof(unsigned char), 1, filePtr);
    fwrite(&byteSkip, sizeof(unsigned char), 1, filePtr);

    // zapisuje imageType
    fwrite(&imageType, sizeof(unsigned char), 1, filePtr);

    fwrite(&shortSkip, sizeof(short int), 1, filePtr);
    fwrite(&shortSkip, sizeof(short int), 1, filePtr);
    fwrite(&byteSkip, sizeof(unsigned char), 1, filePtr);
    fwrite(&shortSkip, sizeof(short int), 1, filePtr);
    fwrite(&shortSkip, sizeof(short int), 1, filePtr);
```

```
// zapisuje rozmiary obrazu
fwrite(&width, sizeof(short int), 1, filePtr);
fwrite(&height, sizeof(short int), 1, filePtr);
fwrite(&bitDepth, sizeof(unsigned char), 1, filePtr);

// zapisuje jeden bajt nieużywanych danych
fwrite(&byteSkip, sizeof(unsigned char), 1, filePtr);

// oblicza rozmiar obrazu
imageSize = width * height * colorMode;

// zmienia opis kolorów z RGB na BGR
for (imageIdx = 0; imageIdx < imageSize ; imageIdx += colorMode)
{
    colorSwap = imageData[imageIdx];
    imageData[imageIdx] = imageData[imageIdx + 2];
    imageData[imageIdx + 2] = colorSwap;
}

// zapisuje dane obrazu
fwrite(imageData, sizeof(unsigned char), imageSize, filePtr);

// zamyka plik
fclose(filePtr);

return 1;
}
```

Podobnie jak w przypadku plików *BMP* funkcję tę można stosować w połączeniu z funkcją `glReadPixels()` do zapisu zawartości okna grafiki OpenGL w pliku *Targa*:

```
void SaveScreenshot()
{
    imageData = malloc(800*600*3); // przydziela pamięć buforowi imageData
    memset(imageData, 0, 800*600*3); // opróżnia zawartość bufora imageData

    // wczytuje dane obrazu w oknie grafiki
    glReadPixels(0, 0, 799, 599, GL_RGB, GL_UNSIGNED_BYTE, imageData);

    // zapisuje obraz w pliku
    WriteTGAFile("writeout.tga", 800, 600, (unsigned char*)imageData);

    // zwalnia pamięć bufora
    free(imageData);
}
```

## Podsumowanie

Mapy bitowe w OpenGL definiowane są za pomocą dwuwymiarowych tablic, w których każdy piksel opisany jest za pomocą pojedynczego bitu. Mapy bitowe wykorzystywane są jako maski prostokątnych fragmentów ekranu oraz do reprezentacji znaków ekranowych.

Funkcja `glRasterPos*`() pozwala określić położenie rysowanej mapy bitowej lub obrazu na ekranie za pomocą współrzędnych lewego, dolnego narożnika.

Po określeniu pozycji mapę bitową rysuje się za pomocą funkcji `glBitmap()`.

Po załadowaniu obrazu do bufora pamięci można wyświetlić go stosując funkcję `glDrawPixels()`. Pozycję wyświetlonego obrazu określa się za pomocą funkcji `glRasterPos*`().

Zawartość ekranu można odczytać korzystając z funkcji `glReadPixels()`, która następnie powinna zostać przetworzona w celu uzyskania efektów specjalnych lub zapisana w pliku.

Funkcja `glCopyPixels()` umożliwia przenoszenie fragmentów obrazu na ekranie.

OpenGL pozwala powiększać i zmniejszać mapy bitowe, a także tworzyć ich odbicia. Służy do tego funkcja `glPixelZoom()`.

Formaty Windows *BMP* i Truevision *Targa* umożliwiają łatwy zapis i odczyt obrazów z plików.

## Rozdział 8.

# Odwzorowania tekstur

Żaden z omówionych dotychczas elementów grafiki OpenGL nie podnosi realizmu tworzonej grafiki w takim stopniu jak tekstury. Zamiast tworzyć obiekty wirtualnego świata z kolorowych wielokątów, które udawać będą jedynie kształt prawdziwych obiektów, można pokryć je dodatkowo teksturami, które pozwalają uzyskać obraz obiektów przypominający realizmem fotografię. Omówione więc zostanie pojęcie tekstury i metody jej implementacji w celu podniesienia realizmu tworzonej grafiki.

W rozdziale tym przedstawione zostaną:

- ◆ podstawy odwzorowań tekstur;
- ◆ obiekty tekstur;
- ◆ powielanie i nakładanie tekstur;
- ◆ mipmapy i poziomy szczegółowości;
- ◆ dwa przykłady zastosowań tekstur.

## Odwzorowania tekstur

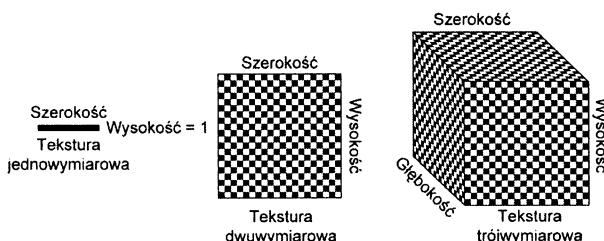
*Odwzorowania tekstur* pozwalają pokrywać wielokąt za pomocą realistycznych obrazów. Na przykład prostokąt można pokryć obrazkiem przedstawiającym okładkę tej książki i uzyskać w ten sposób reprezentację okładki w trójwymiarowym świecie. Podobnie kulę można pokryć mapą powierzchni Ziemi uzyskując w ten sposób trójwymiarową reprezentację naszej planety. Tekstury są powszechnie wykorzystywane w grafice trójwymiarowej. Zwłaszcza w grach umożliwiają uzyskanie tak pożądanego realizmu tworzonych scen.

Mapy tekstur są dwuwymiarowymi tablicami zawierającymi elementy nazywane *tekstelami*. Mimo że reprezentują zawsze obszar prostokąta, to mogą być odwzorowywane na dowolnych trójwymiarowych obiektach, takich jak kule, walce i tym podobne.

Zwykle stosowane są tekstury dwuwymiarowe, ale używa się także tekstur jedno- i trójwymiarowych. Tekstura dwuwymiarowa posiada szerokość i wysokość, co pokazano na rysunku 8.1. Tekstury wymiarowe są charakteryzowane jedynie przez szerokość, ponieważ ich wysokość wynosi zawsze jeden piksel. Tekstury trójwymiarowe posiadają szerokość, wysokość oraz głębokość i czasami nazywane są *teksturami bryłowymi*. W bieżącym rozdziale przedstawiane będą głównie tekstury dwuwymiarowe.

**Rysunek 8.1.**

*Różnice  
pomiędzy teksturami  
jedno-, dwu-  
i trójwymiarowymi*



Po wykonaniu odwzorowania tekstuury na wielokąt będzie ona przekształcana razem z wielokątem. W przypadku wspomnianej już na przykład reprezentacji Ziemi tekstuра będzie obracać się razem z kulą, na której została umieszczona stwarzając w ten sposób wrażenie obrotu Ziemi. Także w przypadku, gdy do jej animacji dodany zostanie kolejny obrót i przesunięcie na orbicie wokół innej kuli reprezentującej Słońce, tekstuра pozostanie na powierzchni kuli. Odwzorowanie tekstuury można wyobrazić sobie jako zmianę sposobu wyglądu powierzchni obiektu lub wielokąta. Niezależnie od wykonywanych przekształceń pokrywać będzie ona zawsze jego powierzchnię. Przed omówieniem szczegółów należy przyjrzeć się najpierw przykładowi zastosowania tekstuury.

**Przykład zastosowania tekstuury**

W przykładzie tym wykorzystana zostanie znana z poprzednich rozdziałów animacja obracającego się sześcianu, ale tym razem będzie on pokryty tekstuру o wzorze szachownicy przedstawioną na rysunku 8.2.

**Rysunek 8.2.**

*Tekstura  
szachownicy*



Ponieważ tekstuра ta umieszczona została w pliku *BMP*, to do jej załadowania wykorzystać można funkcję *LoadBitmapFile()* zaimplementowaną w poprzednim rozdziale. Tekstury w OpenGL muszą posiadać zawsze rozmiary będące potęgą liczby 2, na przykład  $64 \times 64$  lub  $256 \times 256$ . W tym przykładzie wykorzystana zostanie tekstuра o rozmiarach  $64 \times 64$ .

Pierwszy fragment kodu tego przykładu przedstawia pliki nagłówkowe i zmienne globalne:

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

////// Pliki nagłówkowe
#include <windows.h> // standardowy plik nagłówkowy Windows
#include <stdio.h> // plik nagłówkowy operacji wejścia i wyjścia
#include <stdlib.h> // standardowy plik nagłówkowy OpenGL
#include <gl/gl.h> // funkcje pomocnicze OpenGL
#include <gl/glu.h>

////// 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
```

```
///// Opis tekstu
BITMAPINFOHEADERbitmapInfoHeader; // nagłówek pliku
unsigned char*bitmapData; // dane tekstu
unsigned int texture; // obiekt tekstu
```

Większość zaprezentowanych deklaracji jest już znana z poprzednich przykładów. Jedyną nowością jest zmienna texture. Używana będzie jako identyfikator tekstu po jej załadowaniu do pamięci.

Kolejnym fragmentem kodu jest funkcja LoadBitmapFile(), której działanie omówione zostało w poprzednim rozdziale. Parametrem funkcji LoadBitmapFile() jest nazwa pliku oraz zmienna typu BITMAPINFOHEADER. Funkcja zwraca wskaźnik do bufora, w którym umieszczone są dane obrazu wczytane z pliku. Przechowywane są one jako ciąg bajtów, czyli wartości typu unsigned char.

```
unsigned char *LoadBitmapFile(char *filename, BITMAPINFOHEADER *bitmapInfoHeader)
{
    FILE *filePtr; // wskaźnik pozycji pliku
    BITMAPFILEHEADERbitmapFileHeader; // nagłówek pliku
    unsigned char*bitmapImage; // dane obrazu
    intimageIdx = 0; // licznik pikseli
    unsigned chartempRGB; // zmienna zamiany składowych

    // 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);

    // 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;
}

// zamienia miejscami składowe R i B
for (imageIdx = 0; imageIdx < bitmapInfoHeader->biSizeImage; imageIdx+=3)
{
    tempRGB = bitmapImage[imageIdx];
    bitmapImage[imageIdx] = bitmapImage[imageIdx + 2];
    bitmapImage[imageIdx + 2] = tempRGB;
}

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

```

Kolejna funkcja rysuje sześcian. Od poprzednich wersji tej funkcji odróżnia ją to, że dla każdego z wierzchołków sześcianu definiuje współrzędne tekstury. Współrzędne tekstur omówione zostaną wkrótce, teraz należy przyjrzeć się implementacji funkcji DrawTextureCube():

```

void DrawTextureCube(float xPos, float yPos, float zPos)
{
    glPushMatrix();
    glTranslatef(xPos, yPos, zPos);

    glBegin(GL_QUADS);           // górná ściana
    glVertex3f(-0.5f, 0.5f, 0.5f);
    glVertex3f(0.5f, 0.5f, 0.5f);
    glVertex3f(0.5f, 0.5f, -0.5f);
    glVertex3f(-0.5f, 0.5f, -0.5f);
    glEnd();

    glBegin(GL_QUADS);           // przednia ściana
    glVertex3f(0.5f, -0.5f, 0.5f);
    glVertex3f(0.5f, 0.5f, 0.5f);
    glVertex3f(-0.5f, 0.5f, 0.5f);
    glVertex3f(-0.5f, -0.5f, 0.5f);
    glEnd();

    glBegin(GL_QUADS);           // prawa ściana
    glVertex3f(0.5f, 0.5f, -0.5f);
    glVertex3f(0.5f, 0.5f, 0.5f);
    glVertex3f(0.5f, -0.5f, 0.5f);
    glVertex3f(0.5f, -0.5f, -0.5f);
    glEnd();

    glBegin(GL_QUADS);           // lewa ściana
    glVertex3f(-0.5f, 0.5f, 0.5f);
    glVertex3f(-0.5f, 0.5f, -0.5f);
    glVertex3f(-0.5f, -0.5f, 0.5f);
    glVertex3f(-0.5f, -0.5f, -0.5f);
    glEnd();
}

```

```
glBegin(GL_QUADS);           // dolna 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();

glBegin(GL_QUADS);           // 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();
}
```

Zadaniem następnej funkcji jest inicjalizacja grafiki OpenGL oraz obsługi tekstur. Poniżej przedstawiony został pełen kod źródłowy funkcji Initialize():

```
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 wskaźówek zegara

    glEnable(GL_TEXTURE_2D); // włącza tekstury dwuwymiarowe

    // ładuje obraz tekstury
    bitmapData = LoadBitmapFile("checker.bmp", &bitmapInfoHeader);

    glGenTextures(1, &texture);           // tworzy obiekt tekstury
    glBindTexture(GL_TEXTURE_2D, texture); // aktywuje obiekt tekstury

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

    // tworzy obraz tekstury
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, bitmapInfoHeader.biWidth,
                 bitmapInfoHeader.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, bitmapData);
}
```

Pierwszym wierszem kodu funkcji Initialize(), na który należy zwrócić uwagę, jest wiersz zawierający wywołanie funkcji glEnable() z parametrem GL\_TEXTURE\_2D. Informuje się w ten sposób maszynę OpenGL, że tworzone wielokąty będą pokrywane wzorem tekstury. Stosowanie tekstur można wyłączyć wywołując funkcję glDisable() z parametrem GL\_TEXTURE\_2D.

Po załadowaniu obrazu tekstury z pliku *BMP* stosuje się funkcję glGenTextures(), aby nadać nazwę obiektowi tekstury. Funkcja glBindTexture() wiąże obiekt tekstury z bieżącą teksturą, która będzie wykorzystywana podczas rysowania wielokątów. Aby skorzystać w programie z wielu różnych tekstur, trzeba przy zmianie bieżącej tekstury za każdym razem wywoływać funkcję glBindTexture().

Funkcja `glTexParameter()` określa sposób filtrowania tekstury. Filtrowanie tekstur zostańione omówione w dalszej części bieżącego rozdziału.

Wzór tekstury reprezentowany za pomocą wczytanego wcześniej obrazu określa się za pomocą funkcji `glTexImage2D()`. Za pomocą jej parametrów można określić także rozmiary, typ, położenie i format danych.

Funkcja `Render()` będzie jak zwykle tworzyć kolejną klatkę animacji sześcianu wykorzystując odpowiednie przekształcenia i wywołując funkcję rysowania sześcianu `DrawTextureCube()`:

```
void Render()
{
    // opróżnia bufor ekranu i bufor głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glLoadIdentity();

    glTranslatef(0.0f, 0.0f, -3.0f);      // wykonuje przekształcenia
    glRotatef(angle, 1.0f, 0.0f, 0.0f); // odsuwa sześciyan o 3 jednostki i obraca go
    glRotatef(angle, 0.0f, 1.0f, 0.0f);
    glRotatef(angle, 0.0f, 0.0f, 1.0f);

    DrawTextureCube(0.0f, 0.0f, 0.0f); // rysuje sześciyan pokryty teksturą

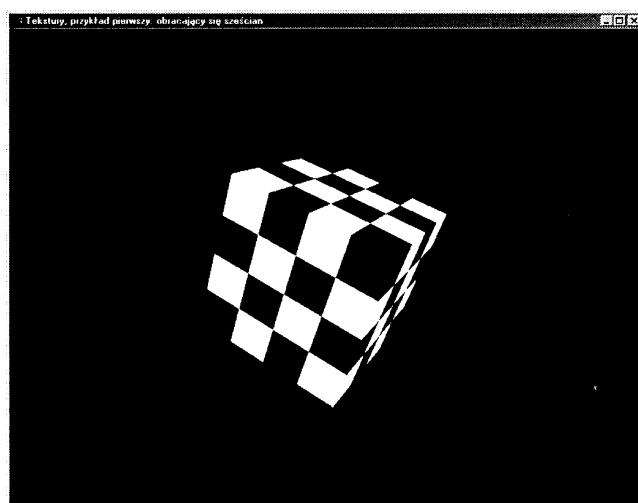
    if (angle >= 360.0f)
        angle = 0.0f;
    angle+=0.2f;

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

I to wszystko! Po umieszczeniu przedstawionych fragmentów kodu w wykorzystywanym doąd szkieletie aplikacji systemu Windows uzyska się działającą animację sześcianu pokrytego teksturą. Pokazuje ją rysunek 8.3.

**Rysunek 8.3.**

Animacja sześcianu pokrytego teksturą



Po zapoznaniu się ze sposobem stosowania tekstur można przejść do bardziej szczegółowego omówienia działania tekstur OpenGL.

## Mapy teksturowe

Po załadowaniu z pliku obrazu tekstyury należy zadeklarować go w OpenGL jako mapę tekstyury. W zależności od liczby wymiarów tekstyury używa się w tym celu innej funkcji. Dla mapy tekstyury dwuwymiarowej będzie to funkcja `glTexImage2D()`. Dla map teksturowych jednowymiarowych użyć trzeba funkcji `glTexImage1D()`, a dla map teksturowych trójwymiarowych funkcji `glTexImage3D()`. W kolejnych podrozdziałach omówiony zostanie sposób użycia tych funkcji.

### Tekstyury dwuwymiarowe

Aby zdefiniować tekstyurę dwuwymiarową użyć można funkcji `glTexImage2D()` zdefiniowanej następująco:

```
void glTexImage2D(GLenum target, GLint level, GLint internalFormat, GLsizei width,
    GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid* texels);
```

Parametr `target` może przyjmować wartość `GL_TEXTURE_2D` lub `GL_PROXY_TEXTURE_2D`. W aktualnych zastosowaniach będzie posiadać on zawsze wartość `GL_TEXTURE_2D`. Parametr `level` określa rozdzielczość mapy tekstyury. Ponieważ tekstyury o różnych rozdzielczościach omówione zostaną później, to na razie parametrowi temu trzeba nadawać wartość 0, co oznacza tylko jedną rozdzielczość.

Parametr `internalFrame` opisuje format teksceli, które przekazywane będą funkcji. Może przyjmować on wartości z przedziału od 1 do 4 bądź jednej z 32 stałych zdefiniowanych przez OpenGL. Zamiast przedstawiać wszystkie te stałe wymienić wystarczy tylko te, które będą najbardziej przydatne: `GL_LUMINANCE`, `GL_LUMINANCE_ALPHA`, `GL_RGB` i `GL_RGBA`.

Parametry `width` i `height` definiują szerokość i wysokość mapy tekstyury. Ich wartości muszą być potęgą liczby 2. Parametr `border` określa, czy dookoła tekstyury znajduje się ramka. Przyjmuje wtedy wartość 1, natomiast wartość 0 oznacza brak ramki.

Parametr `format` definiuje format danych obrazu tekstyury. Może on przyjmować jedną z następujących wartości: `GL_COLOR_INDEX`, `GL_RGB`, `GL_RGBA`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_LUMINANCE` lub `GL_LUMINANCE_ALPHA`.

Parametr `type` definiuje typ danych obrazu tekstyury. Może on przyjmować jedną z następujących wartości: `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_FLOAT` lub `GL_BITMAP`.

Ostatni z parametrów funkcji, `texels`, jest wskaźnikiem właściwych danych obrazu tekstyury, które zostały wygenerowane za pomocą pewnego algorytmu lub załadowane z pliku.

Można przypuścić, że załadowany został obraz formatu *RGBA* o szerokości *textureWidth* i wysokości *textureHeight* do bufora wskazywanego przez zmienną *textureData*. Funkcję `glTexImage2D()` wywoływać się będzie wtedy w następujący sposób:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, textureWidth, textureHeight, 0, GL_RGBA,
GL_UNSIGNED_BYTE, textureData);
```

Po wywołaniu funkcji `glTexImage2D()` tekstura jest gotowa do użycia.

## Tekstury jednowymiarowe

Tekstury jednowymiarowe odpowiadają teksturom dwuwymiarowym, których wysokość jest równa 1. Tekstury jednowymiarowe wykorzystywane są często do tworzenia kolorowych obwódek bądź przez algorytmy cieniowania, które wymagają tworzenia zbyt dużej liczby wielokątów. Teksturę jednowymiarową tworzy się za pomocą funkcji `glTexImage1D()` posiadającej następujący prototyp:

```
void glTexImage1D(GLenum target, GLint level, GLint internalFormat, GLsizei width,
GLint border, GLenum format, GLenum type, const GLvoid* texels);
```

Wszystkie parametry funkcji `glTexImage1D()` są takie same jak w przypadku funkcji `glTexImage2D()`. Jedyna różnica polega na braku parametru reprezentującego wysokość tekstury. Tablica *texels* jest w przypadku funkcji `glTexImage1D()` jednowymiarowa, a nie dwuwymiarowa jak dla funkcji `glTexImage2D()`. Parametr *target* posiadać będzie w analizowanych zastosowaniach zawsze wartość `GL_TEXTURE_1D`.

Poniżej zaprezentowany został krótki fragment kodu ilustrujący sposób wykorzystania funkcji `glTexImage1D()`:

```
unsigned char imageData[128];
...
glTexImage1D(GL_TEXTURE_1D, 0, GL_RGBA, 32, 0, GL_RGBA, GL_UNSIGNED_BYTE, imageData);
```

## Tekstury trójwymiarowe

Tekstury trójwymiarowe pozwalają uzyskać niezwykłe efekty, jednak ich zastosowanie pochłania ogromne ilości pamięci nawet dla najmniejszych tekstur. Dlatego też dotychczas tekstury trójwymiarowe wykorzystywane są szerzej jedynie w zastosowaniach medycznych, takich jak na przykład rezonans magnetyczny. Powszechnie zastosowanie tekstur trójwymiarowych w grach jest jeszcze kwestią przyszłości.

Do tworzenia tekstur trójwymiarowych używa się funkcji `glTexImage3D()` o następującym prototypie:

```
glTexImage3D(GLenum target, GLint level, GLint internalFormat, GLsizei width, GLsizei height,
GLsizei depth, GLint border, GLenum format, GLenum type, const GLvoid* texels);
```

Jej parametry są takie same jak w przypadku funkcji `glTexImage1D()` i `glTexImage2D()`. Jedyna różnica polega na dodaniu parametru *depth* reprezentującego trzeci wymiar tekstury. Oczywiście tym razem parametr *texels* będzie wskazywał tablicę trójwymiarową.

Poniżej przedstawiony został krótki fragment kodu ilustrujący sposób korzystania z funkcji `glTexImage3D()` w programach:

```
unsigned char imageData[16][16][16][3];
...
glTexImage3D(GL_TEXTURE_3D, 0, GL_RGB, 16, 16, 16, 0, GL_RGB, GL_UNSIGNED_BYTE, imageData);
```

## Obiekty tekstur

*Obiekty tekstur* służą do przechowywania gotowych do użycia tekstur. Umożliwiają załadowanie do pamięci wielu obrazów tekstur, a następnie odwoływanie się do nich podczas rysowania sceny. Rozwiążanie takie zwiększa efektywność tworzenia grafiki, ponieważ tekstury nie muszą być ładowane za każdym razem, gdy konieczne jest ich użycie.

### Tworzenie nazwy tekstury

Zanim użyty zostanie obiekt tekstury, trzeba najpierw utworzyć jego nazwę. Nazwy tekstur są liczbami dodatnimi całkowitymi różnymi od zera. Użycie funkcji `glGenTextures()` do tworzenia nazw pozwala zachować pewność, że istniejąca już nazwa nie zostanie zduplikowana:

```
void glGenTextures(GLsizei n, GLuint *textureNames);
```

Parametr `n` określa liczbę nazw tekstur, które powinny zostać utworzone i umieszczone w tablicy `textureNames`. Na przykład poniższy fragment kodu utworzy trzy nowe nazwy tekstur:

```
unsigned int textureNames[3];
...
glGenTextures(3, textureNames);
```

### Tworzenie i stosowanie obiektów tekstur

Po utworzeniu nazwy tekstury należy związać ją z danymi opisującymi teksturę. Służy do tego funkcja `glBindTexture()`:

```
void glBindTexture(GLenum target, GLuint textureName);
```

Pierwsze wywołanie tej funkcji powoduje utworzenie nowego obiektu tekstury o domyślnych właściwościach. Właściwości te można zmienić wywołując odpowiednie funkcje OpenGL związane z teksturami. Parametr `target` funkcji `glBindTexture()` może przyjmować wartości `GL_TEXTURE_1D`, `GL_TEXTURE_2D` lub `GL_TEXTURE_3D`.

Po związaniu obiektu tekstury z danymi można ponownie użyć funkcji `glBindTexture()` do zmiany właściwości obiektu tekstury. Można założyć, że w programie utworzono wiele obiektów tekstu i powiązano je z danymi tekstur. Przy tworzeniu grafiki informuje się maszynę OpenGL za pomocą funkcji `glBindTexture()`, której tekstury należy użyć. Poniższy fragment kodu informuje OpenGL, że należy użyć drugiej ze zdefiniowanych tekstur:

```

unsigned int textureNames[3];
...
glGenTextures(3, textureNames);
...
glBindTexture(GL_TEXTURE_2D, textureNames[1]);
...
// kod określający dane i właściwości tekstury
...
glBindTexture(GL_TEXTURE_2D, textureNames[1]);
...
// rysuje obiekty

```

## Filtrowanie tekstur

Ponieważ obraz tekstury ulega naturalnemu zniekształceniu podczas wykonywania przekształceń na wielokątach, to w efekcie pojedynczy piksel może reprezentować na ekranie wiele teksceli mapy tekstury. *Filtrowanie tekstur* umożliwia określenie sposobu, w jaki OpenGL wyznacza w takim przypadku reprezentację teksceli.

Termin powiększenie stosowany w odniesieniu do filtrowania tekstur oznacza sytuację, w której piksel reprezentuje jedynie część pojedynczego tekscela. Natomiast pomniejszenie oznacza, że pojedynczy piksel reprezentuje wiele teksceli. Sposób, w jaki OpenGL obsługuje obie sytuacje ustala się korzystając z funkcji `glTexParameter()`:

```
void glTexParameter(GLenum target, GLenum pname, GLint param);
```

Parametr *param* reprezentuje rodzaj używanych tekstur i może przyjmować wartości `GL_TEXTURE_1D`, `GL_TEXTURE_2D` lub `GL_TEXTURE_3D`. Parametr *pname* pozwala określić, czy definiuje się obsługę powiększania czy pomniejszania i może przyjmować wartości odpowiednio `GL_TEXTURE_MAG_FILTER` lub `GL_TEXTURE_MIN_FILTER`. Tabela 8.1 prezentuje możliwe wartości parametru *param*.

**Tabela 8.1.** Wartości określające sposób filtrowania tekstur

Wartość	Opis
<code>GL_NEAREST</code>	Używa tekscela położonego najbliżej środka rysowanego piksela
<code>GL_LINEAR</code>	Stosuje liniową interpolację (średnią ważoną) czterech teksceli położonych najbliżej środka rysowanego piksela
<code>GL_NEAREST_MIPMAP_NEAREST</code>	Używa obrazu o najbardziej zbliżonej rozdzielczości do rozdzielczości wielokąta oraz filtr <code>GL_NEAREST</code>
<code>GL_NEAREST_MIPMAP_LINEAR</code>	Używa obrazu o najbardziej zbliżonej rozdzielczości do rozdzielczości wielokąta oraz filtr <code>GL_LINEAR</code>
<code>GL_LINEAR_MIPMAP_NEAREST</code>	Stosuje liniową interpolację dwóch mipmap o najbardziej zbliżonej rozdzielczości do rozdzielczości wielokąta oraz używa filtr <code>GL_NEAREST</code>
<code>GL_LINEAR_MIPMAP_LINEAR</code>	Stosuje liniową interpolację dwóch mipmap o najbardziej zbliżonej rozdzielczości do rozdzielczości wielokąta oraz używa filtr <code>GL_LINEAR</code>

Filtry wykorzystujące mipmapy można stosować jedynie dla parametru `GL_TEXTURE_MIN_FILTER`. Mipmapy oraz poziomy szczegółowości omówione zostaną wkrótce.

Poniżej przedstawiony został przykład konfiguracji filtrów powiększania i pomniejszania za pomocą funkcji `glTexParameteri()`:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

## Funkcje tekstur

Podobnie jak w przypadku łączenia kolorów OpenGL pozwala także definiować sposób zachowania kolorów tekstury za pomocą funkcji tekstur. Dla każdej tekstury można wybrać jedną z czterech dostępnych funkcji wywołując funkcję `glTexEnv()`:

```
void glTexEnv(GLenum target, GLenum pname, GLint param);
```

Parametr *target* musi posiadać wartość `GL_TEXTURE_ENV`, a parametr *pname* wartość `GL_TEXTURE_ENV`, która informuje maszynę OpenGL, że określany jest sposób łączenia kolorów tekstury z pikselami istniejącymi w buforze ekranu. Parametr *param* może przyjmować jedną z wartości podanych w tabeli 8.2.

Tabela 8.2. Funkcje tekstur

Funkcja	Opis
<code>GL_BLEND</code>	Kolor tekstury mnożony jest przez kolor piksela iłączony ze stałym kolorem
<code>GL_DECAL</code>	Kolor tekstury zastępuje kolor piksela
<code>GL_MODULATE</code>	Kolor tekstury mnożony jest przez kolor piksela

Domyślną funkcją jest `GL_MODULATE`. Poniższe wywołanie funkcji `glGetEnv()` zastępuje ją funkcją `GL_DECAL`:

```
glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
```

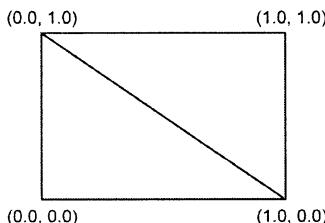
## Współrzędne tekstury

Tworząc scenę należy określić współrzędne tekstury dla wszystkich wierzchołków. Współrzędne tekstury pozwalają ustalić położenie teksceli na rysowanym obiekcie. Współrzędne  $(0, 0)$  określają lewy, dolny narożnik tekstury, a współrzędne  $(1, 1)$  prawy, górny narożnik.

Podczas rysowania wielokąta trzeba określić współrzędne tekstury dla każdego z jego wierzchołków. W przypadku tekstur dwuwymiarowych mają one postać  $(s, t)$ , gdzie  $s$  i  $t$  przyjmują wartości z przedziału od 0 do 1. Rysunek 8.4 pokazuje współrzędne tekstury dla wszystkich wierzchołków wielokąta. Współrzędne te muszą być zawsze określone przed narysowaniem danego wierzchołka.

**Rysunek 8.4.**

Współrzędne tekstury dla wierzchołków wielokąta



Sposób interpolacji współrzędnych tekstury przez OpenGL przedstawiony zostanie na przykładzie hipotetycznego wierzchołka umieszczonego dokładnie w środku wielokąta. Ponieważ współrzędne lewego, dolnego wierzchołka wynoszą  $(0, 0)$ , a prawego, górnego  $(1, 1)$ , to współrzędne hipotetycznego wierzchołka OpenGL określi jako  $(0.5, 0.5)$ . OpenGL wyznacza automatycznie współrzędne tekstury wewnętrznych wielokąta, a programista musi określić je jedynie dla wierzchołków.

Współrzędne tekstury określa się za pomocą funkcji `glTexCoord2f()`. Posiada ona szereg wersji, ale w analizowanych zastosowaniach przydatne będą jedynie:

```
void glTexCoord2f(float s, float d);
void glTexCoord2fv(float *coords);
```

Funkcje te pozwalają określić bieżące współrzędne tekstury. Przy tworzeniu wierzchołka za pomocą funkcji `glVertex*` otrzymuje on współrzędne tekstury ustalone za pomocą funkcji `glTexCoord2f()`.

W ostatnim z przykładów programów funkcja `glTexCoord2f()` użyta została w następujący sposób:

```
glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-0.5f, 0.5f, 0.5f); // lewy, dolny
    glTexCoord2f(1.0f, 0.0f); glVertex3f(0.5f, 0.5f, 0.5f); // prawy, dolny
    glTexCoord2f(1.0f, 1.0f); glVertex3f(0.5f, 0.5f, -0.5f); // prawy, górny
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-0.5f, 0.5f, -0.5f); // lewy górny
glEnd();
```

Choć zwykle współrzędne tekstury przyjmują wartości z przedziału od 0 do 1, to mogą także otrzymywać wartości spoza tego przedziału. W rezultacie otrzymuje się efekty powtarzania lub rozciągania tekstury. Należy przyjrzeć się im bliżej.

## Powtarzanie i rozciąganie tekstur

Jeśli współrzędne tekstury będą posiadać wartości spoza przedziału od 0 do 1, to OpenGL może powtarzać lub rozciągać teksturę. *Powtarzanie* tekstury określane jest czasami także mianem *kafelkowania*. Jeśli współrzędne tekstury będą posiadać wartość  $(2.0, 2.0)$ , to tekstura zostanie powtórzona czterokrotnie, co pokazuje rysunek 8.5. Jeśli współrzędne tekstury będą posiadać wartość  $(1.0, 2.0)$ , to tekstura zostanie powtórzona dwukrotnie tylko w kierunku  $t$ .

**Rysunek 8.5.**

Powtórzenie tekstury



Rozciąganie tekstury polega na nadaniu współrzędnym tekstury większym od 1.0 wartości 1.0, a współrzędnym mniejszym od 0.0 wartości 0.0. Rysunek 8.6 pokazuje wynik rozciągnięcia tekstury o współrzędnych (2.0, 2.0). W rezultacie obraz tekstury pojawia się w lewym, dolnym narożniku wielokąta.

**Rysunek 8.6.**

Rozciąganie tekstury



Powtarzanie i rozciąganie tekstur wykonuje się w OpenGL za pomocą funkcji `glTexParameter()`, która została omówiona przy okazji filtrowania tekstur. Tym razem jednak parametry target nadana została wartość `GL_TEXTURE_WRAP_S` i `GL_TEXTURE_WRAP_T` w celu wyboru współrzędnych  $s$  i  $t$ . Dla danej współrzędnej określa się operację powtarzania lub rozciągania tekstury za pomocą wartości `GL_REPEAT`, `GL_CLAMP` lub `GL_CLAMP_TO_EDGE` (wartość `GL_CLAMP_TO_EDGE` stosowana jest jedynie w przypadku, gdy tekstura posiada ramkę, a programista chce, aby OpenGL zignorował ją podczas rozciągania tekstury).

Poniżej zaprezentowany został fragment kodu informującego OpenGL, aby powtórzył teksturę w obu kierunkach.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

OpenGL może równocześnie powtarzać teksturę w jednym kierunku i rozciągać ją w drugim kierunku. Oto fragment kodu, który rozciąga teksturę w kierunku s i powtarza w kierunku t:

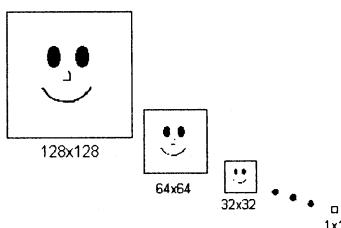
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

## Mipmapy i poziomy szczegółowości

Podczas rysowania obiektów pokrytych teksturami można zauważać pewne obszary, w których pojawiają się nieoczekiwane wzory, zwłaszcza przy oddalaniu się lub zbliżaniu obiektów. Są one rezultatem filtrowania zastosowanego przez OpenGL w celu dopasowania tekstury do rozmiarów obiektu. Niepożądane efekty można wyeliminować za pomocą *mipmap*, które pozwalają kontrolować poziom szczegółowości tekstury (patrz: rysunek 8.7).

**Rysunek 8.7.**

*Mipmapy pozwalają kontrolować poziom szczegółowości tekstury*



OpenGL dobiera odpowiednią mipmapę na podstawie bieżącej rozdzielczości obiektu. Gdy obiekt znajduje się bliżej obserwatora, stosowana jest mipmapa zawierająca więcej szczegółów. Gdy obiekt oddala się od obserwatora wybierane są mipmapy o coraz mniejszym poziomie szczegółowości.

Mipmapy definiuje się w ten sam sposób co tekstury, czyli za pomocą funkcji `glTexImage2D()`. Jedyna różnica polega na określeniu stopnia szczegółowości mipmapy za pomocą parametru `level`. Definiując zwykłą teksturę parametrowi `level` nadaje się wartość 0. W przypadku mipmap wartość 0 oznacza mipmapę o największym poziomie szczegółowości. Wartość 1 oznacza rozdzielcość dwukrotnie mniejszą niż dla poziomu 0, wartość 2 czterokrotnie mniejszą i tak dalej.

Mipmapy muszą posiadać rozmiary będące potęgą liczby 2 (począwszy od największej mipmapy aż do mipmapy o rozmiarach  $1 \times 1$ ). Jeśli na przykład mipmapa o największej rozdzielczości posiada rozmiary  $64 \times 64$ , to trzeba utworzyć także mipmapy  $32 \times 32$ ,  $16 \times 16$ ,  $8 \times 8$ ,  $4 \times 4$ ,  $2 \times 2$  i  $1 \times 1$ . Definiując mipmapę  $64 \times 64$  za pomocą funkcji `glTexImage2D()` parametrowi `level` nadaje się wartość 0, a definiując mipmapę  $32 \times 32$  wartość 1 i tak dalej. Ilustruje to poniższy fragment kodu:

```

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 64, 64, 0, GL_RGB, GL_UNSIGNED_BYTE, texImage0);
glTexImage2D(GL_TEXTURE_2D, 1, GL_RGB, 32, 32, 0, GL_RGB, GL_UNSIGNED_BYTE, texImage1);
glTexImage2D(GL_TEXTURE_2D, 2, GL_RGB, 16, 16, 0, GL_RGB, GL_UNSIGNED_BYTE, texImage2);
glTexImage2D(GL_TEXTURE_2D, 3, GL_RGB, 8, 8, 0, GL_RGB, GL_UNSIGNED_BYTE, texImage3);
glTexImage2D(GL_TEXTURE_2D, 4, GL_RGB, 4, 4, 0, GL_RGB, GL_UNSIGNED_BYTE, texImage4);
glTexImage2D(GL_TEXTURE_2D, 5, GL_RGB, 2, 2, 0, GL_RGB, GL_UNSIGNED_BYTE, texImage5);
glTexImage2D(GL_TEXTURE_2D, 6, GL_RGB, 1, 1, 0, GL_RGB, GL_UNSIGNED_BYTE, texImage6);

```

Mipmapy można stosować jedynie pod warunkiem, że wybrany został odpowiedni filtr pomniejszania (prezentuje je tabela 8.3).

**Tabela 8.3.** Filtry wykorzystujące mipmapy

Filtr	Opis
GL_NEAREST_MIPMAP_NEAREST	Używa obrazu o najbardziej zbliżonej rozdzielcości do rozdzielcości wielokąta oraz filtr GL_NEAREST
GL_NEAREST_MIPMAP_LINEAR	Używa obrazu o najbardziej zbliżonej rozdzielcości do rozdzielcości wielokąta oraz filtr GL_LINEAR
GL_LINEAR_MIPMAP_NEAREST	Stosuje liniową interpolację dwóch mipmap o najbardziej zbliżonej rozdzielcości do rozdzielcości wielokąta oraz używa filtra GL_NEAREST
GL_LINEAR_MIPMAP_LINEAR	Stosuje liniową interpolację dwóch mipmap o najbardziej zbliżonej rozdzielcości do rozdzielcości wielokąta oraz używa filtra GL_LINEAR.

## Automatyczne tworzenie mipmap

Biblioteka GLU udostępnia zestaw funkcji pozwalający zautomatyzować tworzenie mipmap. W przypadku tekstur dwuwymiarowych wiele wywołań funkcji glTexImage2D() można zastąpić pojedynczym wywołaniem funkcji gluBuild2dMipmaps() o następującym prototypie:

```
int gluBuild2dMipmaps(GLenum target, GLint internalFormat, GLint width, GLint height,
                      GLenum format, GLenum type, void *texels);
```

Funkcja ta tworzy sekwencję mipmap, a sama wywołuje funkcję glTexImage2D(). Trzeba przekazać jej jedynie mipmapę o największej rozdzielcości, a pozostałe zostaną wygenerowane automatycznie.

Poprzedni przykład po zastosowaniu funkcji gluBuild2dMipmaps() będzie wyglądał następująco:

```

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_LINEAR);
glBuild2dMipmaps(GL_TEXTURE_2D, GL_RGB, 64, 64, GL_RGB, GL_UNSIGNED_BYTE, texImage0);

```

## Animacja powiewającej flagi

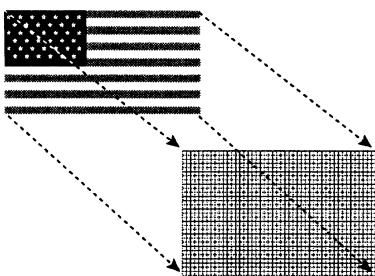
W podrozdziale tym omówiony zostanie krok po kroku przykład programu wyświetlającego animację powiewającej flagi Stanów Zjednoczonych.

## Objaśnienia

W przykładzie tym wykorzystana zostanie tekstuura reprezentująca flagę Stanów Zjednoczonych. Tekstura ta posiada rozmiary 128×64 i zapisana jest w pliku *BMP* stosującym 24-bitowy opis kolorów. Zostanie ona nałożona na zbiór czworokątów tworzących prostokątną strukturę o wymiarach 35×20 czworokątów. Takie rozmiary pozwolą uzyskać realistyczny wizerunek flagi. Aby stworzyć efekt powiewania flagi, trzeba będzie zmieniać współrzędne tekstury dla wszystkich wierzchołków tworzących strukturę flagi pokazaną na rysunku 8.8.

**Rysunek 8.8.**

Tekstura flagi pokrywa siatkę czworokątów o wymiarach 35×20

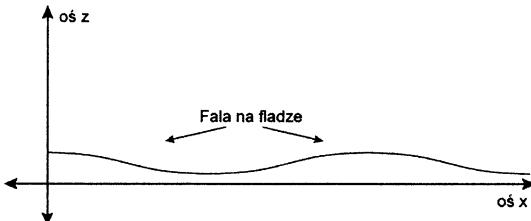


W jaki sposób uzyskać efekt powiewania? Wykorzystać należy funkcję trygonometryczną  $\sin()$  do zainicjowania pozycji wierzchołków na powiewającej fladze. Tworząc kolejne klatki, należy przemieszczać powstałą „falę” wzdłuż flagi.

Flagę trzeba umieścić na płaszczyźnie *xy*, wobec czego zmiany współrzędnych wierzchołków będą dotyczyć jedynie wspólrzędnej *z*. Rysunek 8.9 ilustruje działanie efektu powiewania.

**Rysunek 8.9.**

Flaga powiewająca wzdłuż osi *z*



## Implementacja

Zamiast prezentować w tym miejscu kompletny, obszerny kod źródłowy programu omówione zostaną krok po kroku jego najważniejsze fragmenty.

```
#define BITMAP_ID 0x4D42           // identyfikator formatu BMP

////// Pliki nagłówkowe
#include <windows.h>                // standardowy plik nagłówkowy Windows
#include <stdio.h>                   // plik nagłówkowy operacji wejścia i wyjścia
#include <stdlib.h>
#include <math.h>
```

```
#include <gl/gl.h>           // standardowy plik nagłówkowy OpenGL
#include <gl/glu.h>           // funkcje pomocnicze OpenGL

##### Zmienne globalne
HDC g_HDC;                  // globalny kontekst urządzenia
bool fullScreen = false;     // true = tryb pełnoekranowy;
                             // false = tryb okienkowy
bool keyPressed[256];        // tablica przyciśnięć klawiszy

##### Opis tekstury
BITMAPINFOHEADER bitmapInfoHeader; // nagłówek pliku
unsigned char* bitmapData;         // dane tekstury
unsigned int texture;             // obiekt tekstury
##### Opis flagi
float flagPoints[36][20][3];     // flaga 36x20 (w punktach)
float wrapValue;                // przenosi falę z końca na początek flagi
```

Powyzszy fragment prezentuje wykorzystywane pliki nagłówkowe i zmienne globalne. Większość z nich znanych jest już z poprzednich przykładów. Dodatkowo dołączony został plik nagłówkowy *math.h* ze względu na użycie funkcji *sin()*.

Dane flagi można przechowywać w dwuwymiarowej tablicy punktów o rozmiarach 36×20. Punkty te tworzyć będą 35 czworokątów w kierunku osi x i 19 czworokątów w kierunku osi y. Każdy z nich otrzyma odpowiednie współrzędne tekstury, aby flaga była wyświetlana we właściwy sposób.

Zmiennej *wrapValue* należy używać do przeniesienia danych opisujących falę na fladze z końca na początek flagi. Dzięki temu w trakcie trwania animacji nie będzie konieczne wielokrotne obliczanie danych fali.

Kolejny fragment kodu prezentuje funkcję *InitializeFlag()*, która inicjuje punkty flagi za pomocą funkcji *sin()*:

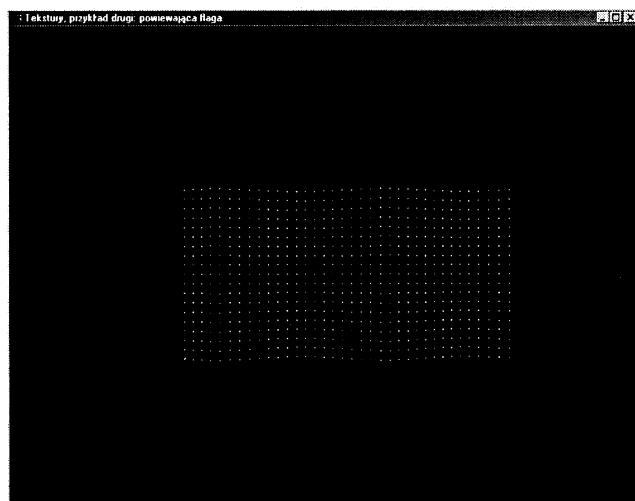
```
void InitializeFlag()
{
    int xIdx;          // licznik w płaszczyźnie x
    int yIdx;          // licznik w płaszczyźnie y
    float sinTemp;

    // Przegląda wszystkie punkty flagi w pętli
    // i wyznacza wartość funkcji sin dla współrzędnej z.
    // Współrzędne x i y otrzymują wartość odpowiedniego licznika pętli.
    for (xIdx = 0; xIdx < 36; xIdx++)
    {
        for (yIdx = 0; yIdx < 20; yIdx++)
        {
            flagPoints[xIdx][yIdx][0] = (float)xIdx;
            flagPoints[xIdx][yIdx][1] = (float)yIdx;
            sinTemp = ((float)xIdx*20.0f) / 360.0f * 2.0f * PI;
            flagPoints[xIdx][yIdx][2] = (float)sin(sinTemp);
        }
    }
}
```

Sposób działania tej funkcji jest bardzo prosty. Przegląda ona tablicę punktów tworzących flagę i wyznacza wartość funkcji `sin()` dla współrzędnej  $x$  każdego punktu. Kształt powiewającej flagi można zmienić modyfikując sposób wyliczania wartości zmiennej `sinTemp`. Po wykonaniu funkcji uzyskuje się zbiór punktów wyglądający jak na rysunku 8.10.

**Rysunek 8.10.**

Wykonanie funkcji  
`InitializeFlag()`  
 powoduje  
 ułożenie punktów  
 we wzór fali  
 rozchodzącej się  
 wzdłuż osi  $x$



Funkcja `Initialize()` służy jak zwykle do zainicjowania grafiki OpenGL:

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

    glEnable(GL_TEXTURE_2D);          // włącza tekstury dwuwymiarowe

    // ładuje obraz tekstury
    bitmapData = LoadBitmapFile("usflag.bmp", &bitmapInfoHeader);

    glGenTextures(1, &texture);        // tworzy obiekt tekstury
    glBindTexture(GL_TEXTURE_2D, texture); // aktywuje obiekt tekstury

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

    // tworzy obraz tekstury
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, bitmapInfoHeader.biWidth,
    bitmapInfoHeader.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, bitmapData);
    InitializeFlag();
}
```

Funkcja ta wygląda znajomo. Do załadowania obrazu tekstury z pliku *BMP* wykorzystuje się funkcję `LoadBitmapFile()`, która została zaimplementowana w poprzednim rozdziale.

Po załadowaniu zawartości pliku *BMP* tworzy się obiekt tekstury za pomocą funkcji `glGenTextures()` i aktywuje się go wywołując funkcję `glBindTexture()`. Dane opisujące teksturę dostarczyć można obiekowi tekstury za pośrednictwem funkcji `glTexImage2D()`.

Kolejna funkcja, `DrawFlag()`, tworzy obraz flagi dla pojedynczej klatki animacji:

```
void DrawFlag()
{
    int xIdx;                                // licznik w kierunku osi x
    int yIdx;                                // licznik w kierunku osi y
    float texLeft;                            // lewa współrzędna tekstury
    float texBottom;                           // dolna współrzędna tekstury
    float texTop;                             // górna współrzędna tekstury
    float texRight;                           // prawa współrzędna tekstury

    glPushMatrix();
    glBindTexture(GL_TEXTURE_2D, texture); // wybiera obiekt tekstury
    glBegin(GL_QUADS);                      // rysuje czworokąty

    // przegląda punkty w pętli z wyjątkiem dwóch ostatnich w każdym kierunku,
    // ponieważ używane są jedynie do rysowania ostatnich czworokątów
    for (xIdx = 0; xIdx < 36; xIdx++)        // kierunek osi x
    {
        for (yIdx = 0; yIdx < 18; yIdx++)    // kierunek osi y
        {
            // wyznacza współrzędne tekstury dla bieżącego czworokąta
            texLeft = float(xIdx) / 35.0f;      // lewa współrzędna tekstury
            texBottom = float(yIdx) / 18.0f;     // dolna współrzędna tekstury
            texRight = float(xIdx+1) / 35.0f;   // prawa współrzędna tekstury
            texTop = float(yIdx+1) / 18.0f;     // górna współrzędna tekstury

            // lewy dolny wierzchołek
            glTexCoord2f( texLeft, texBottom );
            glVertex3f( flagPoints[xIdx][yIdx][0], flagPoints[xIdx][yIdx][1],
                         flagPoints[xIdx][yIdx][2] );

            // prawy dolny wierzchołek
            glTexCoord2f( texRight, texBottom );
            glVertex3f( flagPoints[xIdx+1][yIdx][0], flagPoints[xIdx+1][yIdx][1],
                         flagPoints[xIdx+1][yIdx][2] );

            // prawy górny wierzchołek
            glTexCoord2f( texRight, texTop );
            glVertex3f( flagPoints[xIdx+1][yIdx+1][0], flagPoints[xIdx+1][yIdx+1][1],
                         flagPoints[xIdx+1][yIdx+1][2] );

            // lewy górny wierzchołek
            glTexCoord2f( texLeft, texTop );
            glVertex3f( flagPoints[xIdx][yIdx+1][0], flagPoints[xIdx][yIdx+1][1],
                         flagPoints[xIdx][yIdx+1][2] );
        }
    }
    glEnd();

/*
Udaje ruch flagi:
w każdym wierszu przesuwa współrzędne z o jeden punkt w prawo.
*/
}
```

```

    for( yIdx = 0; yIdx < 19; yIdx++ )           // pętla w płaszczyźnie y
    {
        // zapamiętuje współrzędną z skrajnego, prawego punktu flagi
        wrapValue = flagPoints[35][yIdx][2];

        for( xIdx = 35; xIdx >= 0; xIdx--)// pętla w płaszczyźnie y
        {
            // bieżący punkt otrzymuje współrzędną z poprzedniego punktu
            flagPoints[xIdx][yIdx][2] = flagPoints[xIdx-1][yIdx][2];
        }

        // skrajny, lewy punkt otrzymuje wartość przechowaną w wrapValue
        flagPoints[0][yIdx][2] = wrapValue;
    }

    glPopMatrix();
}

```

Komentarze w kodzie funkcji DrawFlag() pomagają zrozumieć jej działanie. Pierwszą operacją wykonywaną przez funkcję jest wybranie tekstury flagi jako bieżącej tekstury, która używana będzie podczas tworzenia grafiki. Następnie za pomocą wartości GL\_QUADS informuje się maszynę OpenGL, że będą rysowane czworokąty.

Rysowanie flagi rozpoczyna się od jej lewego, dolnego narożnika i przegląda wierzchołki w taki sposób, by możliwe było narysowanie kolejnego czworokąta. Zanim jednak narysowany zostanie czworokąt, trzeba najpierw wyznaczyć współrzędne tekstury dla jego wierzchołków w odniesieniu do całości flagi tak, by została ona prawidłowo pokryta tekstem. Współrzędne tekstury oblicza się dzieląc współrzędne *x* i *y* danego wierzchołka przez liczbę punktów flagi wzdłuż odpowiedniej osi.

Po wyznaczeniu współrzędnych tekstury można narysować czworokąt za pomocą funkcji glTexCoord2f() i glVertex3f(). Wierzchołki czworokąta definiuje się w porządku przeciwnym do kierunku ruchu wskazówek zegara.

Po narysowaniu całej flagi trzeba zmodyfikować jej dane, aby uzyskać efekt fali bieżącej wzdłuż flagi. W tym celu przegląda się punkty flagi wierszami i przesuwa współrzędne z każdego punktu do jego sąsiada po prawej. Najpierw zapamiętuje się jednak w zmiennej *wrapValue* wartość współrzędnej *y* dla skrajnego punktu po prawej stronie flagi. Wartość tę otrzymuje następnie skrajny punkt po lewej stronie flagi, co pozwala uzyskać efekt powtarzającej się fali.

Funkcja Render() jest bardzo podobna do wersji, które używane były w poprzednich programach. Wywołuje ona funkcję DrawFlag() dla każdej klatki animacji po uprzednim przesunięciu układu współrzędnych tak, aby flaga była lepiej widoczna:

```

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

    glTranslatef(-15.0f, -10.0f, -50.0f);    // wykonuje przesunięcie

    DrawFlag(); // rysuje flagę
}

```

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

Poniżej zaprezentowana została pozostała część kodu aplikacji, którego zrozumienie nie powinno przedstawać większych trudności:

```
// funkcja określająca format pikseli
void SetupPixelFormat(HDC hDC)
{
    int nPixelFormat;           // indeks formatu pikseli

    static PIXELFORMATDESCRIPTOR pfd = {
        sizeof(PIXELFORMATDESCRIPTOR),          // rozmiar struktury
        1,                                     // domyślna wersja
        PFD_DRAW_TO_WINDOW |                  // grafika w oknie
        PFD_SUPPORT_OPENGL |                 // grafika OpenGL
        PFD_DOUBLEBUFFER,                   // podwójne buforowanie
        PFD_TYPE_RGBA,                      // tryb kolorów RGBA
        32,                                    // 32-bitowy opis kolorów
        0, 0, 0, 0, 0, 0,                    // nie specyfikuje bitów kolorów
        0,                                     // bez buforu alfa
        0,                                     // nie specyfikuje bitu przesunięcia
        0,                                     // bez bufora akumulacji
        0, 0, 0, 0,                            // ignoruje bity akumulacji
        16,                                    // 16-bit bufor z
        0,                                     // bez bufora powielania
        0,                                     // bez buforów pomocniczych
        PFD_MAIN_PLANE,                     // główna płaszczyzna rysowania
        0,                                     // zarezerwowane
        0, 0, 0 };                           // ignoruje maski warstw

    nPixelFormat = ChoosePixelFormat(hDC, &pfd); // wybiera odpowiedni format pikseli

    // określa format pikseli dla danego kontekstu urządzenia
    SetPixelFormat(hDC, nPixelFormat, &pfd);
}

// procedura okienkowa
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HGLRC hRC;                  // kontekst tworzenia grafiki
    static HDC hDC;                   // kontekst urządzenia
    int width, height;               // szerokość i wysokość okna

    switch(message)
    {
        case WM_CREATE:             // okno jest tworzone
            hDC = GetDC(hwnd);      // pobiera kontekst urządzenia dla okna
            g_HDC = hDC;
            SetupPixelFormat(hDC);   // wywołuje funkcję określającą format pikseli

            // tworzy kontekst tworzenia grafiki i czyni go bieżącym
            hRC = wglCreateContext(hDC);
            wglMakeCurrent(hDC, hRC);
    }
}
```

```
        return 0;
        break;

    case WM_CLOSE:           // okno jest zamykane
        // deaktywuje bieżący kontekst tworzenia grafiki i usuwa go
        wglMakeCurrent(hDC, NULL);
        wglDeleteContext(hRC);

        // wstawia komunikat WM_QUIT do kolejki
        PostQuitMessage(0);

        return 0;
        break;

    case WM_SIZE:
        height = HIWORD(lParam);      // pobiera nowe rozmiary okna
        width = LOWORD(lParam);

        if (height==0)               // zapobiega dzieleniu przez 0
        {
            height=1;
        }

        glViewport(0, 0, width, height); // nadaje nowe wymiary oknu OpenGL
        glMatrixMode(GL_PROJECTION);   // wybiera macierz rzutowania
        glLoadIdentity();             // resetuje macierz rzutowania

        // wyznacza proporcje obrazu
        gluPerspective(54.0f,(GLfloat)width/(GLfloat)height,1.0f,1000.0f);

        glMatrixMode(GL_MODELVIEW);   // wybiera macierz modelowania
        glLoadIdentity();            // resetuje macierz modelowania

        return 0;
        break;

    case WM_KEYDOWN:          // użytkownik naciął klawisz?
        keyPressed[wParam] = true;
        return 0;
        break;

    case WM_KEYUP:
        keyPressed[wParam] = false;
        return 0;
        break;

    default:
        break;
    }

    return (DefWindowProc(hwnd, message, wParam, lParam));
}

// punkt, w którym rozpoczyna się wykonywanie programu
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
int nShowCmd)
{

```

```
WNDCLASSEX windowClass;           // klasa okna
HWND hwnd;                      // uchwyt okna
MSG msg;                        // komunikat
bool done;                      // znacznik zakończenia aplikacji
DWORD dwExStyle;                // rozszerzony styl okna
DWORD dwStyle;                  // styl okna
RECT windowRect;

// zmienne pomocnicze
int width = 800;
int height = 600;
int bits = 32;

windowRect.left=(long)0;          // struktura określająca rozmiary okna
windowRect.right=(long)width;
windowRect.top=(long)0;
windowRect.bottom=(long)height;

// definicja klasy okna
windowClass.cbSize= sizeof(WNDCLASSEX);
windowClass.style= CS_HREDRAW | CS_VREDRAW;
windowClass.lpfnWndProc= WndProc;
windowClass.cbClsExtra= 0;
windowClass.cbWndExtra= 0;
windowClass.hInstance= hInstance;
windowClass.hIcon= LoadIcon(NULL, IDI_APPLICATION); // domyślna ikona
windowClass.hCursor= LoadCursor(NULL, IDC_ARROW);   // domyślny kurSOR
windowClass.hbrBackground= NULL;                   // bez tła
windowClass.lpszMenuName= NULL;                   // bez menu
windowClass.lpszClassName= "MojaKlasa";
windowClass.hIconSm= LoadIcon(NULL, IDI_WINLOGO); // logo Windows

// rejestruje klasę okna
if (!RegisterClassEx(&windowClass))
    return 0;

if (fullScreen)                  // tryb pełnoekranowy?
{
    DEVMODE dmScreenSettings;      // tryb urządzenia
    memset(&dmScreenSettings,0,sizeof(dmScreenSettings));
    dmScreenSettings.dmSize = sizeof(dmScreenSettings);
    dmScreenSettings.dmPelsWidth = width;        // szerokość ekranu
    dmScreenSettings.dmPelsHeight = height;       // wysokość ekranu
    dmScreenSettings.dmBitsPerPel = bits;         // bitów na piksel
    dmScreenSettings.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;

    if (ChangeDisplaySettings(&dmScreenSettings, CDS_FULLSCREEN)
        != DISP_CHANGE_SUCCESSFUL)
    {
        // przełączenie trybu nie powiodło się, z powrotem tryb okienkowy
        MessageBox(NULL, "Display mode failed", NULL, MB_OK);
        fullScreen=FALSE;
    }
}

if (fullScreen)                  // tryb pełnoekranowy?
{
```

```

dwExStyle=WS_EX_APPWINDOW;           // rozszerzony styl okna
dwStyle=WS_POPUP;                  // styl okna
ShowCursor(FALSE);                 // ukrywa kursor myszy
}
else
{
    dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE; // definicja klasy okna
    dwStyle=WS_OVERLAPPEDWINDOW;                   // styl okna
}

AdjustWindowRectEx(&windowRect, dwStyle, FALSE, dwExStyle); // koryguje rozmiar okna

// tworzy okno
hwnd = CreateWindowEx(NULL,           // rozszerzony styl okna
                      "MojaKlasa", // nazwa klasy
                      "Tekstury, przykład drugi: powiewająca flaga", // nazwa aplikacji
                      dwStyle | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
                      0, 0,          // współrzędne x,y
                      windowRect.right - windowRect.left,
                      windowRect.bottom - windowRect.top, // szerokość, wysokość
                      NULL,          // uchwyty okna nadziednego
                      NULL,          // uchwyty menu
                      hInstance,     // instancja aplikacji
                      NULL);        // bez dodatkowych parametrów

// sprawdza, czy utworzenie okna nie powiodło się
// (wtedy wartość hwnd równa NULL)

if (!hwnd)
    return 0;

ShowWindow(hwnd, SW_SHOW); // wyświetla okno
UpdateWindow(hwnd); // aktualizuje okno

done = false; // inicjuje zmienną warunku pętli
Initialize(); // inicjuje OpenGL

// pętla przetwarzania komunikatów
while (!done)
{
    PeekMessage(&msg, hwnd, NULL, NULL, PM_REMOVE);

    if (msg.message == WM_QUIT)      // aplikacja otrzymała komunikat WM_QUIT?
    {
        done = true;                // jeśli tak, to kończy działanie
    }
    else
    {
        if (keyPressed[VK_ESCAPE])
            done = true;
        else
        {
            Render();

            TranslateMessage(&msg); // tłumaczy komunikat i wysyła do systemu
            DispatchMessage(&msg);
        }
    }
}

```

```
}

free(bitmapData);

if (fullScreen)
{
    ChangeDisplaySettings(NULL,0); // przywraca pulpit
    ShowCursor(TRUE);           // i wskaźnik myszy
}
```

Rezultat działania programu pokazuje rysunek 8.11.

#### Rysunek 8.11.

Animacja powiewającej flagi



## Uksztaltonanie terenu

Kolejny program tworzyć będzie grafikę naśladującą naturalne ukształtowanie *pagórkowatego terenu* za pomocą siatki różnych wysokości.

Przedstawiona tutaj metoda jest tylko jednym ze sposobów imitacji naturalnego terenu. Istnieje szereg innych metod, które umożliwiają uzyskanie bardziej efektywnego lub realistycznego efektu. W tym przykładzie zastosowany zostanie także prosty sposób ruchu kamery, który umożliwi obserwację terenu z różnych stron.

Teren pokryty zostanie tekstonią trawy. Inna tekstonia wykorzystana zostanie do uzyskania realistycznego efektu powierzchni wody wypełniającej zagłębiania terenu.

## Objaśnienia

W celu reprezentacji ukształtowania terenu utworzona zostanie regularna siatka wierzchołków w równych odstępach, dla których określi się różną wysokość terenu.

Wysokość każdego z wierzchołków będzie określana na podstawie mapy bitowej o rozmiarach  $32 \times 32$ . Mapa ta będzie zawierać jednakowe wartości składowych kolorów dla poszczególnych bitów (odcienie szarości). W formacie 24-bitowym każda z wartości składowych opisana jest za pomocą 8-bitów, wobec czego będą one reprezentować wysokości od 0 do 255.

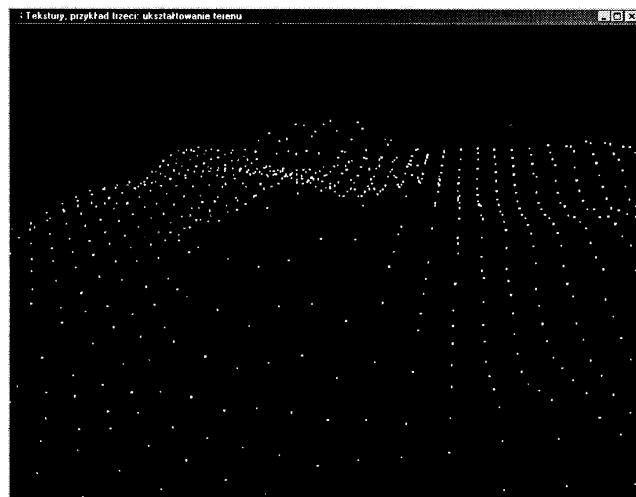
Po załadowaniu tej mapy dysponuje się zbiorem punktów reprezentujących wysokość terenu w pewnych odstępach. Odstępy pomiędzy wierzchołkami ustala się posługując się *skalą mapy*. Skalowanie pozwala zwiększać lub zmniejszać rozmiary obiektów w sposób określony przez współczynnik skalowania. Koncepcję tę wykorzystać można w tym przykładzie zwiększając odstęp pomiędzy wierzchołkami za pomocą współczynnika skali mapy.

Ustalając współrzędne wierzchołka odpowiadającego węźlowi siatki mnoży się indeks węzła siatki przez współczynnik skali mapy. Na przykład współrzędną  $x$  wierzchołka zdefiniuje się jako iloczyn jego indeksu na osi  $x$  siatki i współczynnika skali mapy.

Ponieważ mapa terenu ma postać siatki wysokości, to w programie reprezentować będzie ją można za pomocą dwuwymiarowej tablicy współrzędnych wierzchołków. Siatka ta leżeć będzie w płaszczyźnie  $xz$ , a wysokość terenu mierzoną będzie wzdłuż osi  $y$ .

Po załadowaniu siatki uzyskany zostanie obraz terenu przedstawiony na rysunku 8.12.

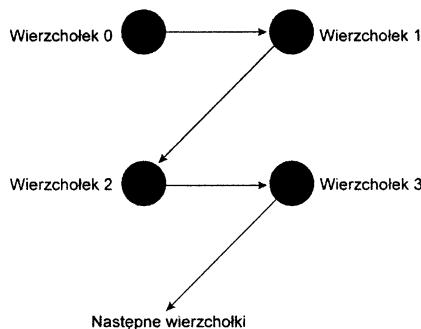
**Rysunek 8.12.**  
Zbiór punktów siatki



Mapę ukształtowania terenu rysować można tworząc łańcuchy trójkątów GL\_TRIANGLE\_STRIP rzędami wzdłuż osi  $z$ . W tym celu trzeba jednak tworzyć kolejne wierzchołki w specjalnym porządku. Rozpoczynając tworzenie kolejnego łańcucha należy posuwać się wzdłuż osi  $x$  podając kolejne wierzchołki według wzoru przypominającego literę  $Z$ , co ilustruje rysunek 8.13. Aby uzyskać właściwe odwzorowanie tekstury, każde kolejne cztery wierzchołki będą przekazywane maszynie OpenGL. W ten sam sposób należy postępować też w odniesieniu do kolejnych wierszy siatki.

**Rysunek 8.13.**

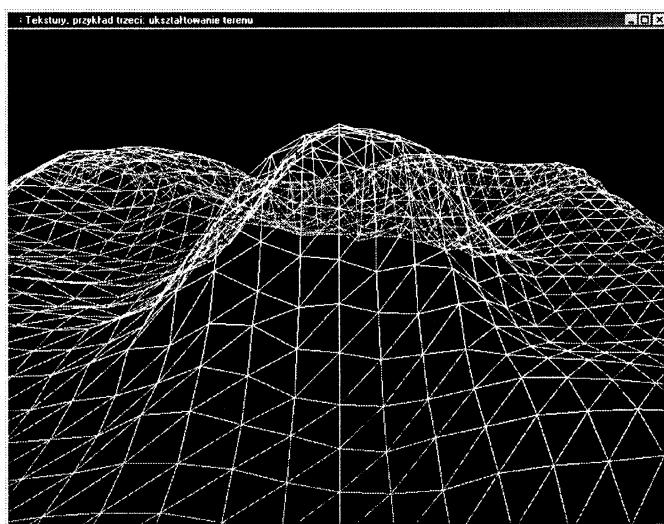
*Porządek tworzenia wierzchołków pojedynczego wiersza siatki*



Rysunek 8.14 przedstawia efekt rysowania wierzchołków za pomocą wzoru przypominającego literę Z zastosowanego do zbioru punktów pokazanych na rysunku 8.12.

**Rysunek 8.14.**

*Mapa terenu uzyskana po narysowaniu łańcuchów trójkątów*



Jeśli trójkąty wypełniony zostanie odcieniami szarości określonymi przez wartości składowych kolorów dla poszczególnych wierzchołków mapy oraz modelem cieniowania gładkiego, to uzyskany zostanie obraz terenu przedstawiony na rysunku 8.15. Jaśniejsze odcienie szarości reprezentują obszary położone wyżej, a ciemniejsze niżej. Efekt ten wzbogacony następnie zostanie teksturą dla podniesienia realizmu.

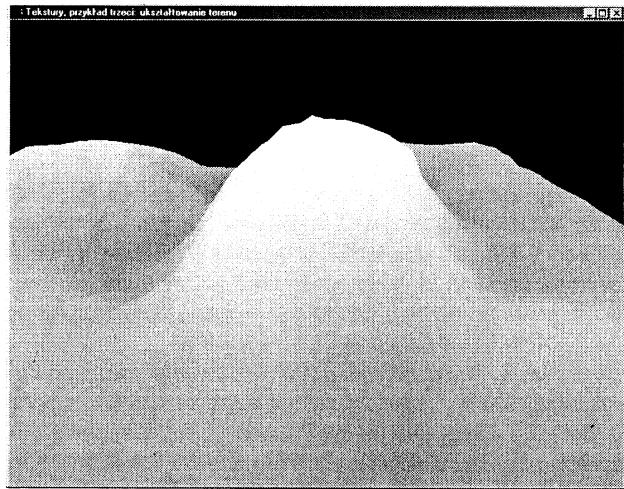
Tekstura zostanie zastosowana do każdego kolejnego zestawu czterech wierzchołków z osobna. Sposób definicji współrzędnych tekstuury dla poszczególnych wierzchołków przedstawia rysunek 8.16.

W wyniku pokrycia czworokątów teksturową przypominającą murawę uzyskuje się realistyczny obraz terenu przedstawiony na rysunku 8.17.

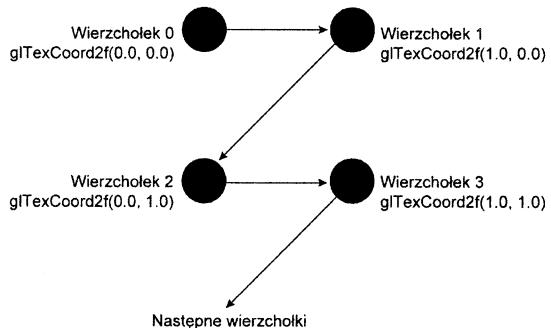
Wygląda całkiem dobrze, prawda?

**Rysunek 8.15.**

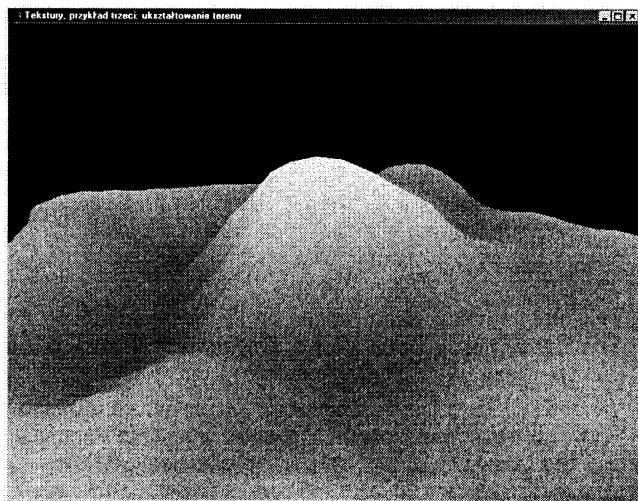
Mapa terenu uzyskana na podstawie odcieni szarości poszczególnych wierzchołków

**Rysunek 8.16.**

Współrzędne tekstury wierzchołków mapy

**Rysunek 8.17.**

Mapa terenu uzyskana przez zastosowanie tekstury

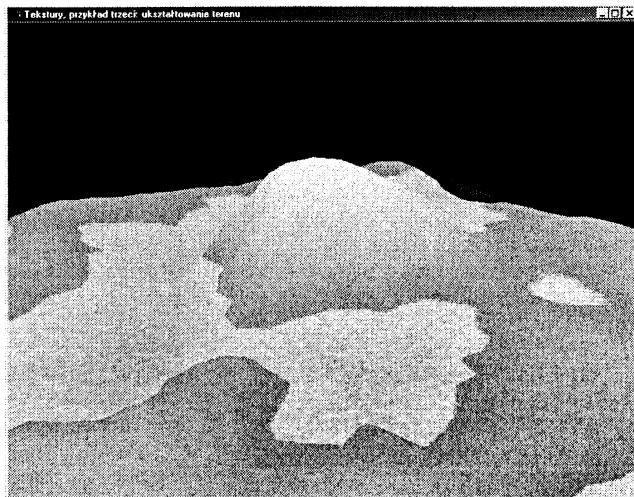


Uzyskany obraz terenu wzbogacony zostanie dodatkowo o efekt wody. W tym celu należy zdefiniować poziom morza jako wysokość terenu, poniżej której będzie on zalany wodą. Na poziomie morza trzeba umieścić pojedynczy prostokąt wypełniony przezroczystym

kolorem niebieskim reprezentującym powierzchnię wody. Dla uzyskania bardziej realistycznego wyglądu można pokryć ją odpowiednią tekstrurą. Zmieniając cyklicznie w kolejnych ujęciach wysokość poziomu wody łatwo jest uzyskać jeszcze jeden efekt przypominający zalewanie brzegu przez fale. Uzyskany w ten sposób obraz terenu prezentuje rysunek 8.18.

**Rysunek 8.18.**

Obraz terenu  
wzbogacony o wodę



Mimo wysiłku włożonego w uatrakcyjnienie obrazu terenu jego statyczna obserwacja szybko się znudzi. Dlatego też umożliwić należy poruszanie kamerą za pomocą myszy. Ruchy myszą w lewo i prawo powodować będą okrążanie terenu przez kamerę, a ruchy w górę i w dół zbliżanie się kamery do terenu i jej oddalenie. Niezależnie od zmiany położenia kamera będzie zawsze wycelowana w ten sam punkt, który określony będzie w kodzie programu. Położenie kamery zmieniać można obsługując komunikaty WM\_MOUSEMOVE w funkcji okienkowej WndProc(). Dla każdego komunikatu trzeba ustalić wielkość przesunięcia myszy i na jego podstawie wyznaczyć nowe położenie kamery korzystając z funkcji trygonometrycznych.

## Implementacja

Kod programu omówiony zostanie fragmentami. Pierwszy z nich definiuje stałe i deklaruje zmienne:

```
////// Definicje
#define BITMAP_ID 0x4D42          // identyfikator formatu BMP
#define MAP_X32                     // rozmiar mapy wzduż osi x
#define MAP_Z32                     // rozmiar mapy wzduż osi y
#define MAP_SCALE20.0f               // skala mapy
#define PI3.14159

////// Pliki nagłówkowe
#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>                                // funkcje pomocnicze OpenGL

////// Zmienne globalne
HDC g_HDC;                                         // globalny kontekst urządzenia
bool fullScreen = false;                           // true = tryb pełnoekranowy;
                                                    // false = tryb okienkowy
bool keyPressed[256];                             // tablica przycisków klawiszy

float angle = 0.0f;                                 // kąt kamery
float radians = 0.0f;                               // kąt kamery w radianach
float waterHeight = 154.0f;                         // poziom morza
bool waterDir = true;                             // animacja falowania wody;
                                                    // true = poziom w góre, false = poziom w dół

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

////// Opis tekstury
BITMAPINFOHEADER bitmapInfoHeader;              // pomocniczy nagłówek obrazu
BITMAPINFOHEADER landInfo;                       // nagłówek obrazu tekstury lądu
BITMAPINFOHEADER waterInfo;                      // nagłówek obrazu tekstury wody

unsigned char* imageData;                        // dane mapy
unsigned char* landTexture;                     // dane tekstury lądu
unsigned char* waterTexture;                    // dane tekstury wody
unsigned int land;                             // obiekt tekstury lądu
unsigned int water;                            // obiekt tekstury wody

////// Opis terenu
float terrain[MAP_X][MAP_Z][3];                // wysokość terenu w punktach siatki (0-255)

```

Tym razem pierwszy fragment kodu jest nieco dłuższy. Blok definicji stałych określa rozmiary mapy wzdłuż osi x i z oraz jej skalę. Definiuje także wartość liczby  $\pi$  wykorzystywaną podczas obliczania położenia kamery.

Zadanie większości z zadeklarowanych zmiennych globalnych jest oczywiste. Zmienna `waterHeight` definiuje poziom morza, na którym rysuje się czworokąt reprezentujący powierzchnię wody. Zmienna `waterDir` określa kierunek ruchu poziomu wody, która może podnosić się lub opadać. Kolejny blok zmiennych służy do przechowania informacji o położeniu myszy oraz pozycji i orientacji kamery. Zmienna `imageData` wskazująca będzie dane mapy terenu o rozmiarach  $32 \times 32$  załadowane z pliku *BMP*. Zmienne `landTexture` i `waterTexture` wskazują będące dane tekstur reprezentujących powierzchnię lądu i wody. Aby efektywnie korzystać z tych tekstur, trzeba utworzyć dla nich osobne obiekty tekstur, których nazwy przechowywane będą za pomocą zmiennych `land` i `water`.

Ostatnią zmienną globalną zadeklarowaną w powyższym fragmencie kodu jest tablica przechowująca opis terenu w postaci współrzędnych wierzchołków dla poszczególnych punktów siatki. Rozmiary terenu określone są przez stałe `MAP_X` i `MAP_Z`.

Kod programu zawiera także znaną z poprzednich przykładów implementację funkcji LoadBitmapFile() służącą do załadowania zawartości pliku *BMP* stosującego 24-bitowy opis kolorów. Funkcja ta nie będzie tutaj omawiana.

Kolejny blok kodu stanowi funkcja InitializeTerrain():

```
void InitializeTerrain()
{
    // oblicza współrzędne wierzchołków
    // dla wszystkich punktów siatki
    for (int z = 0; z < MAP_Z; z++)
    {
        for (int x = 0; x < MAP_X; x++)
        {
            terrain[x][z][0] = float(x)*MAP_SCALE;
            terrain[x][z][1] = (float)imageData[(z*MAP_Z+x)*3];
            terrain[x][z][2] = -float(z)*MAP_SCALE;
        }
    }
}
```

Funkcja ta wyznacza współrzędne wierzchołków dla wszystkich punktów siatki terenu. Wyznaczenie współrzędnych *x* i *z* polega na pomnożeniu indeksu punktu siatki przez współczynnik skali mapy. Wartość zmiennej *z* jest ujemna, ponieważ teren leży wzduż ujemnej części osi *z*. Oczywiście wybór położenia terenu może być zupełnie inny, ale wtedy należy pamiętać także o zmianie sposobu ruchu kamery i jej orientacji.

Sposób wyznaczenia współrzędnej *y* jest najciekawszym fragmentem kodu funkcji InitializeTerrain(). Polega on na pobraniu wartości składowej koloru dla odpowiedniego piksela mapy bitowej opisującej teren. Nieco kłopotu może sprawić jedynie znalezienie piksela odpowiadającego bieżącemu punktowi siatki. Korzysta się w tym celu z następującego równania:

$$(z*MAP_Z+x)*3$$

Analizując to równanie należy pamiętać, że dane mapy bitowej znajdują się w jednowymiarowym buforze i zawierają takie same wartości składowej czerwonej, zielonej i niebieskiej dla każdego piksela.

Kolejna funkcja, LoadTextures(), służy do załadowania obrazów tekstur i utworzenia obiektów tekstur:

```
bool LoadTextures()
{
    // ładuje obraz tekstury lądu
    landTexture = LoadBitmapFile("green.bmp", &landInfo);
    if (!landTexture)
        return false;

    // ładuje obraz tekstury wody
    waterTexture = LoadBitmapFile("water.bmp", &waterInfo);
    if (!waterTexture)
        return false;
```

```

// tworzy mipmapy lądu
glGenTextures(1, &land);
 glBindTexture(GL_TEXTURE_2D, land);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
 gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, landInfo.biWidth, landInfo.biHeight, GL_RGB,
 GL_UNSIGNED_BYTE, landTexture);

// tworzy mipmapy wody
glGenTextures(1, &water);
 glBindTexture(GL_TEXTURE_2D, water);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
 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, waterInfo.biWidth, waterInfo.biHeight,
 GL_RGB, GL_UNSIGNED_BYTE, waterTexture);

return true;
}

```

Po załadowaniu obrazów tekstur za pomocą funkcji `LoadBitmapFile()` najpierw trzeba utworzyć obiekt tekstury lądu i określić sposób filtrowania tekstury. Następnie tworzy się automatycznie mipmapy korzystając z funkcji `gluBuild2DMipmaps()`. Podobnie pośtuje się w przypadku tekstury wody, ale dodatkowo definiuje się dla niej właściwości powtarzania w obu kierunkach, z których korzystać można przy rysowaniu wody:

```

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

    // wyznacza położenie kamery
    cameraX = lookX + sin(radians)*mouseY; // mnożenie przez mouseY
    cameraZ = lookZ + cos(radians)*mouseY; // powoduje zbliżenie bądź oddalenie kamery
    cameraY = lookY + mouseY / 2.0f;

    // wycelowuje kamerę w środek terenu
    lookX = (MAP_X*MAP_SCALE)/2.0f;
    lookY = 150.0f;
    lookZ = -(MAP_Z*MAP_SCALE)/2.0f;

    // opróżnia bufory 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);

    // wybiera teksturę lądu
    glBindTexture(GL_TEXTURE_2D, land);

    // przegląda rzędami wszystkie punkty siatki,
    // rysując dla każdego pojedynczy łańcuch trójkątów wzdłuż osi z.
    for (int z = 0; z < MAP_Z-1; z++)
    {
        glBegin(GL_TRIANGLE_STRIP);
        for (int x = 0; x < MAP_X-1; x++)

```

```
{  
    // dla każdego wierzchołka: obliczamy składowe odcienia szarości  
    // wyznaczamy współrzędne tekstury i rysujemy go.  
    /*  
        wierzchołki rysowane są w następującej kolejności:  
  
        0 ---> 1  
            /  
            /  
            ||  
        2 ---> 3  
    */  
  
    // wierzchołek 0  
    glColor3f(terrain[x][z][1]/255.0f, terrain[x][z][1]/255.0f,  
              terrain[x][z][1]/255.0f);  
    glTexCoord2f(0.0f, 0.0f);  
    glVertex3f(terrain[x][z][0], terrain[x][z][1], terrain[x][z][2]);  
  
    // wierzchołek 1  
    glTexCoord2f(1.0f, 0.0f);  
    glColor3f(terrain[x+1][z][1]/255.0f, terrain[x+1][z][1]/255.0f,  
              terrain[x+1][z][1]/255.0f);  
    glVertex3f(terrain[x+1][z][0], terrain[x+1][z][1], terrain[x+1][z][2]);  
  
    // wierzchołek 2  
    glTexCoord2f(0.0f, 1.0f);  
    glColor3f(terrain[x][z+1][1]/255.0f, terrain[x][z+1][1]/255.0f,  
              terrain[x][z+1][1]/255.0f);  
    glVertex3f(terrain[x][z+1][0], terrain[x][z+1][1], terrain[x][z+1][2]);  
  
    //wierzchołek 3  
    glTexCoord2f(1.0f, 1.0f);  
    glColor3f(terrain[x+1][z+1][1]/255.0f, terrain[x+1][z+1][1]/255.0f,  
              terrain[x+1][z+1][1]/255.0f);  
    glVertex3f(terrain[x+1][z+1][0], terrain[x+1][z+1][1], terrain[x+1][z+1][2]);  
}  
glEnd();  
}  
  
// włącza łączenie kolorów  
 glEnable(GL_BLEND);  
  
// przełączca bufor głębi w tryb "tylko-do-odczytu"  
 glEnable(GL_DEPTHMASK);  
  
// określa funkcje łączenia dla efektu przejrzystości  
 glEnable(GL_SRC_ALPHA, GL_ONE);  
  
 glColor4f(0.5f, 0.5f, 1.0f, 0.7f); // kolor niebieski, przezroczysty  
 glBindTexture(GL_TEXTURE_2D, water); // wybiera teksturę wody  
  
// rysuje powierzchnię wody jako jeden duży czworokąt  
 glBegin(GL_QUADS);  
    glTexCoord2f(0.0f, 0.0f); // lewy, dolny wierzchołek  
    glVertex3f(terrain[0][0][0], waterHeight, terrain[0][0][2]);
```

```

glTexCoord2f(10.0f, 0.0f);           // prawy, dolny wierzchołek
glVertex3f(terrain[MAP_X-1][0][0], waterHeight, terrain[MAP_X-1][0][2]);

glTexCoord2f(10.0f, 10.0f);          // prawy, górny wierzchołek
glVertex3f(terrain[MAP_X-1][MAP_Z-1][0], waterHeight, terrain[MAP_X-1][MAP_Z-1][2]);

glTexCoord2f(0.0f, 10.0f);           // lewy, górny wierzchołek
glVertex3f(terrain[0][MAP_Z-1][0], waterHeight, terrain[0][MAP_Z-1][2]);
glEnd();

// przywraca zwykły tryb pracy bufora głębi
glDepthMask(GL_TRUE);

// wyłącza łączenie kolorów
glDisable(GL_BLEND);

// tworzy animację powierzchni wody
if (waterHeight > 155.0f)
    waterDir = false;
else if (waterHeight < 154.0f)
    waterDir = true;

if (waterDir)
    waterHeight += 0.01f;
else
    waterHeight -= 0.01f;

glFlush();
SwapBuffers(g_HDC);                // przełącza bufory
}

```

Funkcja Render() rozpoczyna swoje działanie od wyznaczenia kąta obrotu kamery wyrażonego w radianach. Korzystając z tej wartości oblicza się nowe położenia kamery — współrzędną *x* za pomocą funkcji *sin()* tego kąta, a współrzędną *z* za pomocą funkcji *cos()*. Następnie oblicza się współrzędne środka terenu, w który wycelowana będzie kamera.

Kamerę umieszcza się i wycelowuje za pomocą funkcji *gluLookAt()*.

Przed rysowaniem terenu trzeba poinformować maszynę OpenGL wywołując funkcję *glBindTexture()*, że będzie wykorzystywana tekstura lądu. Dla każdego rzędu punktów siatki wzdłuż osi *z* rysuje się łańcuch trójkątów. Dla każdego wierzchołka określa się składowe odcienia szarości (dzieląc wysokość terenu przez 255) oraz współrzędne tekstury. W celu wyznaczenia kolejności tworzenia wierzchołków trzeba posłużyć się omówionym już wzorem przypominającym literę *Z*.

Ostatni fragment funkcji Render() rysuje powierzchnię wody. Można rozpoznać w nim kod służący do uzyskania efektu przezroczystości, który omówiony został w rozdziale 7. Dodatkowo za pomocą funkcji *glBindTexture()* tworzy on teksturę powierzchni wody. Wierzchołki czworokąta reprezentującego powierzchnię wody zdefiniowane są w porządku przeciwnym do kierunku ruchu wskazówek zegara. Po narysowaniu czworokąta wyłączany jest efekt przezroczystości i wykonywany kod animacji powierzchni wody.

Funkcja inicjacji OpenGL i danych programu wygląda tym razem następująco:

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

    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

    imageData = LoadBitmapFile("terrain2.bmp", &bitmapInfoHeader);

    // inicjuje dane terenu i ładuje tekstury
    InitializeTerrain();
    LoadTextures();
}
```

Po zainicjowaniu odpowiednich parametrów maszyny OpenGL funkcja `Initialize()` ładuje dane terenu za pomocą funkcji `LoadBitmapFile()`. Następnie wywołuje funkcje służące do inicjacji danych terenu i załadowania tekstur.

Ze względu na możliwość poruszania kamerą za pomocą myszy trzeba wprowadzić modyfikacje w kodzie funkcji okienkowej:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HGLRC hRC;                  // kontekst tworzenia grafiki
    static HDC hDC;                    // kontekst urządzenia
    int width, height;                // szerokość i wysokość okna
    int oldMouseX, oldMouseY;          // współrzędne poprzedniego położenia myszy

    switch(message)
    {
        case WM_CREATE:             // okno jest tworzone
            hDC = GetDC(hwnd);       // pobiera kontekst urządzenia dla okna
            g_HDC = hDC;
            SetupPixelFormat(hDC);   // wywołuje funkcję określającą format pikseli

            // tworzy kontekst tworzenia grafiki i czyni go bieżącym
            hRC = wglCreateContext(hDC);
            wglMakeCurrent(hDC, hRC);

            return 0;
            break;

        case WM_CLOSE:              // okno jest zamykane
            // deaktywuje bieżący kontekst tworzenia grafiki i usuwa go
            wglMakeCurrent(hDC, NULL);
            wglDeleteContext(hRC);

            // wstawia komunikat WM_QUIT do kolejki
            PostQuitMessage(0);

            return 0;
            break;
    }
}
```

```
case WM_SIZE:
    height = HIWORD(lParam);           // pobiera nowe rozmiary okna
    width = LOWORD(lParam);

    if (height==0)                   // unika dzielenie przez 0
    {
        height=1;
    }

    glViewport(0, 0, width, height); // nadaje nowe wymiary oknu OpenGL
    glMatrixMode(GL_PROJECTION);     // wybiera macierz rzutowania
    glLoadIdentity();               // resetuje macierz rzutowania

    // wyznacza proporcje obrazu
    gluPerspective(54.0f,(GLfloat)width/(GLfloat)height,1.0f,1000.0f);

    glMatrixMode(GL_MODELVIEW);      // wybiera macierz modelowania
    glLoadIdentity();               // resetuje macierz modelowania

    return 0;
break;

case WM_KEYDOWN:                  // użytkownik naciął klawisz?
keyPressed[wParam] = true;
return 0;
break;

case WM_KEYUP:
keyPressed[wParam] = false;
return 0;
break;

case WM_MOUSEMOVE:
// zapamiętuje współrzędne myszy
oldMouseX = mouseX;
oldMouseY = mouseY;

// pobiera nowe współrzędne myszy
mouseX = LOWORD(lParam);
mouseY = HIWORD(lParam);

// ograniczenia ruchu kamery
if (mouseY < 200)
    mouseY = 200;
if (mouseY > 450)
    mouseY = 450;

if ((mouseX - oldMouseX) > 0)    // mysz przesunięta w prawo
    angle += 3.0f;
else if ((mouseX - oldMouseX) < 0) // mysz przesunięta w lewo
    angle -= 3.0f;

return 0;
break;

default:
break;
}

return (DefWindowProc(hwnd, message, wParam, lParam));
```

Modyfikacja polega na dodaniu obsługi komunikatu WM\_MOUSEMOVE. Należy zapamiętać poprzednie współrzędne myszy, aby ustalić, w jakim kierunku została przesunięta. Następnie pobiera się nowe współrzędne myszy korzystając z wartości parametru lParam. Aby kontrolować sposób obserwacji grafiki przez użytkownika, wprowadza się ograniczenia ruchu kamery. Na podstawie ruchu myszy zwiększa się lub zmniejsza kąt obrotu kamery.

I to wszystko! Przykład ten pokazuje, jak łatwo można uzyskać zaawansowaną grafikę OpenGL, jeśli tylko dokona się właściwej analizy i rozkładu problemu.

## Podsumowanie

*Odwzorowania tekstur* polegają na umieszczeniu obrazów na powierzchniach wielokątów i służą do zwiększenia realizmu tworzonej grafiki.

Po załadowaniu obrazu tekstury z pliku należy zdefiniować go jako dane tekstury OpenGL. W zależności od liczby wymiarów tekstury używa się w tym celu funkcji glTexImage1D(), glTexImage2D() lub glTexImage3D().

Obiekty tekstur ułatwiają korzystanie z tekstur OpenGL. Pozwalają załadować wiele tekstur i używać ich podczas tworzenia grafiki.

Podczas tworzenia obiektów pokrytych teksturami można zaobserwować na ich powierzchni niepożądane efekty związane z koniecznością dopasowania rozmiarów tekstur do zmieniających się rozmiarów obiektów. Efekty te można wyeliminować kontrolując poziom szczegółowości tekstur za pomocą mipmap.



## 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 doskonałszego wyglądu.

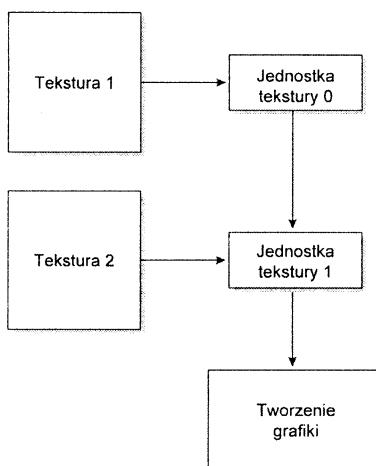
## Tekstury wielokrotne

Dotychczas wykorzystywana była pojedyncza textura 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 zatwierdzane 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 teksturopy*. Jednostki teksturopy omówione zostaną szczegółowo w dalszej części rozdziału. Obecnie wystarczy jedynie informacja dotycząca tego, że każda jednostka teksturopy reprezentuje pojedynczą teksturę, a także to, że podczas tworzenia teksturopy wielokrotnej każda jednostka teksturopy przekazuje wynik jej zastosowania kolejnej jednostce teksturopy aż do utworzenia końcowej teksturopy wielokrotniej. Rysunek 9.1 ilustruje proces tworzenia teksturopy 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 teksturow wielokrotnych

Przed rozpoczęciem stosowania tekstur wielokrotnych trzeba najpierw sprawdzić, czy udostępnia je używana implementacja OpenGL. Tekstury wielokrotnie 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 teksturow 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 współzędnych 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_TEXTUREi_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, oznaczać 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 związać 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);
glTexEnvi(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);
glTexEnvi(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);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, checkerTex->texID);
glTexEnvi(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 tekstrą, 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 tekstuury tylko dla pierwszej jednostki tekstuury. 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 sześciian



A oto pierwszy fragment kodu programu:

```
////// Definicje
#define BITMAP_ID 0x4042           // 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 "glext.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 przycisków 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

// Tekstury
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 *glext.h* należy dodawać 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.

Powyzszy 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. Upraszczą to proces ładowania i stosowania tekstur w OpenGL.

Omawiany program będzie umożliwiał 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 oddalenie 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 = strcspn(str, " ");

        // sprawdza, czy str 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;
}

```

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);
```

```

glTexEnv(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);
glTexEnv(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 miejscu 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);

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 bufora 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);

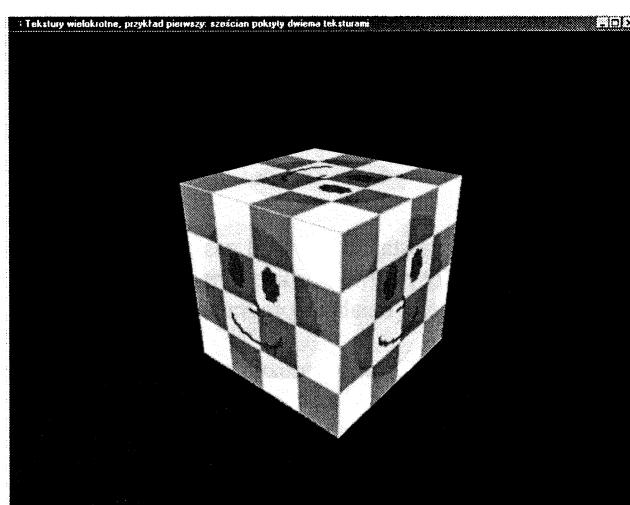
    // rysuje sześcian o środku w punkcie (0,0,0)
    DrawCube(0.0f, 0.0f, 0.0f);

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

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

**Rysunek 9.3.**

Przykład  
zastosowania  
tekstyury 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:

```
glTexGen(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGen(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.

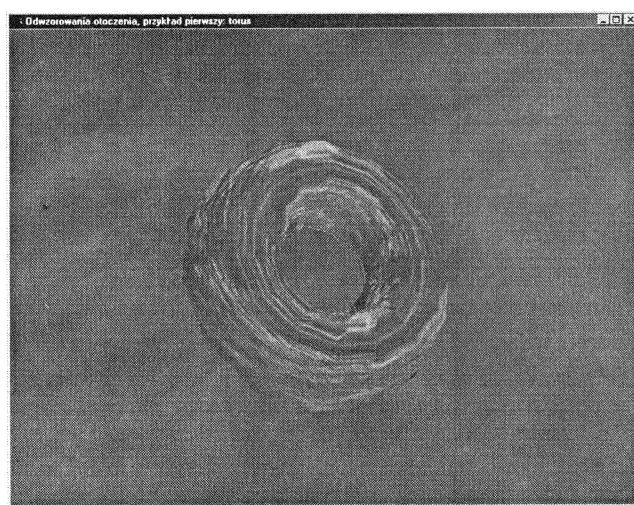
## Torus na niebie

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

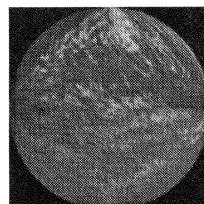
Tekstonią otoczenia utworzyć można na podstawie tekstonii nieba, do której zastosowany zostanie efekt rybiego oka za pomocą dowolnego edytora obrazów dysponującego taką możliwością. Rysunek 9.5 przedstawia otrzymaną tekstonią otoczenia.

**Rysunek 9.4.**

Torus pokryty teksturą otoczenia

**Rysunek 9.5.**

Tekstura otoczenia uzyskana na podstawie tekstuury 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 bufora 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
    glTexGen(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
    glTexGen(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 bufory  
}
```

Jak łatwo zauważyc, 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 niezwykłych efektów, jakie można osiągnąć poprzez manipulacje 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 bufory 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 tekstonią
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 tekstuury jako bieżącą macierz. Następnie ładuje do niej macierz jednostkową, zanim zastosuje do bieżącej macierzy funkcję obrotu `glRotatef()`. Na skutek jej użycia tekstonia zostaje obrócona o pewien kąt względem osi z. Po wykonaniu obrotu wybiera macierz modelowania jako bieżącą macierz i rysuje czworokąt pokryty obróconą tekstonią. Proste, prawda?

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

```
// opróżnia buforey ekranu i głębię
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
// wybiera tryb operacji na macierzy tekstuury
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glRotatef(angle, 0.0f, 0.0f, 1.0f); // obraca tekstonię
// przywraca tryb operacji na macierzy modelowania
glMatrixMode(GL_MODELVIEW);

glBindTexture(GL_TEXTURE_2D, texID); // wybiera tekstonię

// rysuje czworokąt pokryty tekstonią
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 tekstuury dla następnych obiektów
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glMatrixMode(GL_MODELVIEW);

// rysuje następne obiekty
...
```

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

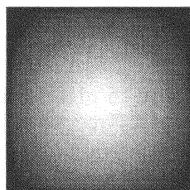
## Mapy oświetlenia

Zadaniem *map oświetlenia* jest symulacja statycznego oświetlenia powierzchni. Mapa oświetlenia jest tekstonią, która opisuje sposób oświetlenia powierzchni. Mapy oświetlenia znajdują coraz powszechniejsze zastosowanie dzięki wprowadzeniu sprzętowej obsługi tekstuur 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



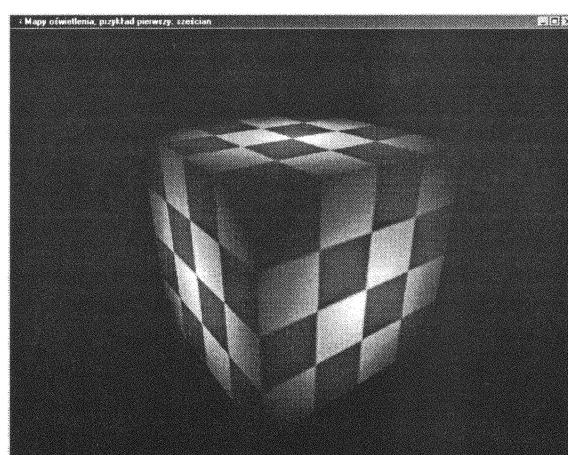
Mapy oświetlenia są teksturami wykorzystującymi jedynie odcienie szarości. Aby uzyskać efekt oświetlenia, mapę taką należy połączyć z tekstonią pokrywającą wielokąt.

## Stosowanie map oświetlenia

Mapę oświetlenia stosuje się na przykład do uzyskania efektu oświetlenia sześcianu pokrytego tekstonią 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() ładowającej 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, lighmapData);
// lub pojedyncza tekstura
glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, width, height, 0, GL_LUMINANCE,
    GL_UNSIGNED_BYTE, lighmapData);
```

Aby zastosować mapę oświetlenia, należy posłużyć się mechanizmem tekstur wielokrotnych i włączyć tryb nakładania tekstury `GL_MODULATE`. Trzeba zatem przyjrzeć się najistotniejszym 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 *lighmapTex; // 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 podłączonym 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;   // zmenna 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);
    glTexEnvi(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 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);
    glTexEnvi(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);

    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);           // góra 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 bufore ekranu i głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    // umieszcza kamere
    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 bufore
}
```

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 wielokąta 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 powołanie 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.



## Rozdział 10.

# Listy wyświetlania i tablice wierzchołków

W przypadku wielu aplikacji graficznych i praktycznie wszystkich gier szczególne znanie ma utrzymanie stałego tempa tworzenia kolejnych klatek animacji. Olbrzymi postęp, jaki dokonał się w zakresie sprzętowych układów graficznych, nie zmusza już programistów do optymalizacji każdego wiersza tworzonego kodu. Nie oznacza to jednak, że programiści zostali zwolnieni z obowiązku tworzenia efektywnego kodu. Efektywne wykorzystanie sprzętowych układów grafiki możliwe jest przez umiejętne zastosowanie następujących mechanizmów udostępnianych przez OpenGL:

- ◆ **list wyświetlania** — umożliwiają one prekompilację często używanych zestawów poleceń OpenGL, a dzięki temu efektywniejsze ich wykonanie;
- ◆ **tablic wierzchołków** — pozwalają efektywniej przechowywać dane o wierzchołkach i szybko wykonywać operacje zmiany współrzędnych i kolorów wierzchołków.

## Listy wyświetlania

W programach przykładowych zaprezentowanych w poprzednich rozdziałach często powtarzały się fragmenty kodu OpenGL konfiguruujące maszynę stanów w ten sam sposób. Koncepcja list wyświetlania polega na tym, że fragmenty takie są wstępnie przetwarzane, dzięki czemu później mogą zostać wykonane dużo mniejszym kosztem.

Zaletą list wyświetlania jest łatwość użycia. Jedyna trudność polega na ustaleniu, czy w danym przypadku spowodują one rzeczywiście szybsze wykonanie kodu. W praktyce okazuje się też, że końcowy rezultat może zależeć od konkretnej implementacji OpenGL. Nigdy jednak zastosowanie list wyświetlania nie powinno spowodować pogorszenia efektywności działania programu.

Sposób tworzenia i stosowania list wyświetlania przedstawiony zostanie na przykładzie programu, który rysować będzie zbiór piramid za pomocą czterech trójkątów (podstawa ostrosłupa piramidy nie będzie rysowana, ponieważ nie będzie widoczna). Pojedynczą piramidę można narysować wywołując funkcję, która narysuje wieniec trójkątów tworzących ściany ostrosłupa:

```

void DrawPyramid()
{
    glBegin(GL_TRIANGLE_FAN);
    glVertex3f(0.0, 1.0, 0.0);
    glVertex3f(-1.0, 0.0, 1.0);
    glVertex3f(1.0, 0.0, 1.0);
    glVertex3f(1.0, 0.0, -1.0);
    glVertex3f(-1.0, 1.0, -1.0);
    glEnd();
}

```

Ponieważ funkcja ta będzie wywoływana wielokrotnie, to jest kandydatem do zastosowania list wyświetlania. W rzeczywistości użycie list wyświetlania nie przyniesie istotnej poprawy efektywności działania tej funkcji, ponieważ nie wykonuje ona złożonych operacji. W tym przypadku jednak służy ona do zilustrowania zastosowania list wyświetlania na prostym przykładzie.

## Tworzenie listy wyświetlanego

Zanim umieszczony zostanie pierwszy element na liście wyświetlania, trzeba najpierw uzyskać jej nazwę podobnie jak w przypadku obiektów tekstur omówionych w rozdziale 8. Nazwę taką można pobrać za pomocą funkcji `glGenLists()`:

```
GLuint glGenLists(GLsizei range);
```

Parametr `range` umożliwia określenie liczby list wyświetlania, które mają być utworzone. Funkcja `glGenLists()` zwraca dodatnią liczbę całkowitą będącą identyfikatorem pierwszej z utworzonych list. Po zwiększeniu go o jeden uzyskać można identyfikator kolejnej listy i tak dalej. Identyfikatory te stosuje się do poinformowania maszyny OpenGL, z której listy należy skorzystać.

Należy jeszcze sprawdzić, czy wartość zwrocona przez funkcję `glGenLists()` nie jest równa 0. Wartość 0 nie jest identyfikatorem listy, lecz informuje o tym, że utworzenie list wyświetlania nie powiodło się. Przyczyna takiej sytuacji może być na przykład brak ciągłej przestrzeni identyfikatorów o wymaganej przez programistę wielkości. W dowolnym momencie można także sprawdzić, czy dany identyfikator określa dostępną listę wyświetlania. W tym celu należy wywołać funkcję `glIsList()` o następującym prototypie:

```
GLboolean glIsList(GLuint listName);
```

Funkcja ta zwraca wartość `GL_TRUE`, jeśli wartość `listName` reprezentuje identyfikator dostępnej listy wyświetlania i wartość `GL_FALSE` w przeciwnym razie.

## Umieszczanie poleceń na liście wyświetlanego

Po uzyskaniu identyfikatora listy wyświetlania możemy już umieszczać na niej polecenia. Odbywa się to w sposób przypominający użycie pary funkcji `glBegin()` i `glEnd()` otaczających wywołania funkcji tworzących podstawowe elementy grafiki. Najpierw wywołujemy funkcję, która określa listę wyświetlania, na której chcemy umieszczać polecenia. Po umieszczeniu poleceń wywołujemy kolejną funkcję, która kończy proces tworzenia listy. Funkcje te, o nazwach `glNewList()` i `glEndList()` zdefiniowane są następująco:

```
void glNewList(GLuint listName, GLenum mode);  
void glEndList();
```

Parametr *listName* jest identyfikatorem listy, na której umieszcza się polecenia. Może to być identyfikator nowej listy, która właśnie została utworzona za pomocą funkcji `glGenLists()` lub listy, która była używana już w innych celach. W tym drugim przypadku ponowne użycie listy powoduje usunięcie jej dotychczasowej zawartości i umieszczenie nowych poleceń. Parametr *mode* określa tryb komplikacji poleceń umieszczanych na liście i może przyjmować wartość `GL_COMPILE` lub `GL_COMPILE_AND_EXECUTE`. Pierwsza z wymienionych wartości powoduje jedynie komplikację poleceń, a druga dodatkowo wykonuje je. W większości przypadków umieszczając polecenia na liście wystarczy jedynie zainicjować listę i wobec tego najczęściej używa się wartości `GL_COMPILE` parametru *mode*.

Mimo że w bloku kodu ograniczonym wywołaniami funkcji `glNewList()` i `glEndList()` można wywoływać dowolne funkcje OpenGL, to jednak nie wszystkie mogą zostać skompilowane i umieszczone na liście wyświetlanego. Funkcje te zamiast tego zostają natychmiast wykonane. Należą do nich:

```
glColorPointer  
glDeleteLists  
glDisableClientState  
glEdgeFlagPointer  
glEnableClientState  
glFeedbackBuffer  
glFinish  
glFlush  
glGenLists  
glIndexPointer  
glInterleavedArrays  
glIsEnabled  
glIsList  
glNormalPointer  
glPopClientAttrib  
glPixelStore  
glPushClientAttrib  
glReadPixels  
glRenderMode  
glSelectBuffer  
glTexCoordPointer  
glVertexPointer
```

Natychmiast wykonane zostaną także wszystkie polecenia `glGet()` oraz wywołanie funkcji `glTexImage()`, jeśli tworzy ono teksturę proxy (jeśli nie używa się tekstur proxy, to można bezpiecznie umieścić polecenie `glTexImage()` na liście wyświetlanego, chociaż istnieją efektywniejsze polecenia obsługi tekstu, które omówione zostaną wkrótce).

## Wykonywanie list wyświetlania

Po utworzeniu listy wyświetlania i umieszczeniu na niej poleceń można zastosować listę w dowolnym miejscu programu, w którym zwykle wykonywane są te polecenia. W tym celu należy wywołać funkcję

```
void glCallList(GLuint listName);
```

Wykonuje ona polecenia znajdujące się na liście wyświetlania o identyfikatorze *listName*.

Możliwe jest także wykonanie wielu list wyświetlania za pomocą pojedynczego wywołania przedstawionej poniżej funkcji:

```
void glCallLists(GLsizei num, GLenum type, const GLvoid *lists);
```

Parametr *num* określa liczbę list wyświetlania, które mają być wykonane, a parametr *lists* wskazuje tablicę ich identyfikatorów. Choć wartość zwracana przez funkcję `glGenLists()` jest typu `unsigned int` i typ ten używany jest także przez inne funkcje związane z listami wyświetlania, to jednak w praktyce można rzutować go na inny typ, który w danej sytuacji jest wygodniejszy. Właśnie dlatego parametr *lists* wskazuje typ `void`, a parametr *type* umożliwia określenie typu danych znajdujących się w tablicy. Dozwolone typy prezentuje tabela 10.1.

**Tabela 10.1.** Wartości parametru *type* funkcji `glCallLists()`

Wartość	Typ
<code>GL_BYTE</code>	Liczba całkowita ze znakiem reprezentowana za pomocą jednego bajta
<code>GL_UNSIGNED_BYTE</code>	Liczba całkowita bez znaku reprezentowana za pomocą jednego bajta
<code>GL_SHORT</code>	Liczba całkowita ze znakiem reprezentowana za pomocą dwóch bajtów
<code>GL_UNSIGNED_SHORT</code>	Liczba całkowita bez znaku reprezentowana za pomocą dwóch bajtów
<code>GL_INT</code>	Liczba całkowita ze znakiem reprezentowana za pomocą czterech bajtów
<code>GL_UNSIGNED_INT</code>	Liczba całkowita bez znaku reprezentowana za pomocą czterech bajtów
<code>GL_FLOAT</code>	Liczba zmienoprzecinkowa reprezentowana za pomocą czterech bajtów
<code>GL_2_BYTES</code>	Obszar wskazywany przez parametr <i>lists</i> traktowany jest jak tablica bajtów, których każda kolejna para reprezentuje identyfikator listy; wartość tego identyfikatora uzyskiwana jest przez pomnożenie wartości pierwszego bajtu pary (bez znaku) przez wartość $2^8$ i dodanie do niej wartości drugiego bajtu pary (również bez znaku)
<code>GL_3_BYTES</code>	Obszar wskazywany przez parametr <i>lists</i> traktowany jest jak tablica bajtów, których każda kolejna trójka reprezentuje identyfikator listy; wartość tego identyfikatora uzyskiwana jest przez zsumowanie wartości pierwszego bajtu trójki (bez znaku) pomnożonego przez wartość $2^{16}$ z wartością drugiego bajtu trójki (bez znaku) pomnożoną przez wartość $2^8$ i z wartością trzeciego bajtu trójki (również bez znaku)
<code>GL_4_BYTES</code>	Obszar wskazywany przez parametr <i>lists</i> traktowany jest jak tablica bajtów, których każda kolejna czwórka reprezentuje identyfikator listy; wartość tego identyfikatora uzyskiwana jest przez zsumowanie wartości pierwszego bajtu czwórki (bez znaku) pomnożonego przez wartość $2^{32}$ z wartością drugiego bajtu czwórki (bez znaku) pomnożoną przez wartość $2^{16}$ z wartością trzeciego bajtu czwórki (bez znaku) pomnożoną przez wartość $2^8$ z wartością czwartego bajtu czwórki (również bez znaku)

Funkcja `glCallLists()` wykonuje po kolejci listy wyświetlenia, których identyfikatory znajdują się w tablicy *lists* (począwszy od indeksu 0, a skończywszy na *num*-1). Jeśli któryś z elementów tablicy nie jest identyfikatorem dostępnej listy wyświetlania, to jest po prostu ignorowany.

Może zdarzyć się (zwłaszcza w przypadku używania list wyświetlania do tworzenia tekstów), że wykonywanie list, których nazwy znajdują się w tablicy będziemy trzeba rozpocząć od pewnej wartości jej indeksu. Wartość tę można określić za pomocą funkcji

```
void glListName(GLuint offset);
```

Wykonywanie list rozpocznie się wtedy od identyfikatora o indeksie *offset* i zakończy na identyfikatorze o indeksie *offset + num - 1*. Domyślnie wartość *offset* wynosi 0. Należy pamiętać, że parametr *offset* opisuje stan maszyny OpenGL i gdy zmieniona zostanie jego wartość, to pozostała ona aktualna aż do następnego wywołania funkcji *glListName()*. Bieżącą wartość parametru *offset* można odczytać za pomocą funkcji *glGet()* przekazując jej wartość *GL\_LIST\_BASE*.

## Uwagi dotyczące list wyświetlania

Należy zwrócić uwagę na kilka istotnych aspektów korzystania z list wyświetlania. Polecenia *glCallLists()* lub *glCallList()* można umieszczać na listach wyświetlania. Aby zapobiec możliwości wystąpienia nieskończonej rekursji wzajemnych wywołań dwóch list wyświetlania, polecenia umieszczone na liście wyświetlania wykonywanej przez funkcję *glCallList()* nie stają się częścią nowej listy wyświetlania.

Listy wyświetlania mogą zawierać polecenia zmieniające stan maszyny OpenGL. Stan maszyny OpenGL nie jest automatycznie zachowywany przed wykonaniem listy wyświetlania i przywracany po wykonaniu listy. Dlatego trzeba pamiętać o zapamiętywaniu i odwracaniu stanu maszyny OpenGL za pomocą funkcji *glPushMatrix()* i *glPopMatrix()* oraz *glPushAttrib()* i *glPopAttrib()*.

## Usuwanie list wyświetlania

Utworzenie listy wyświetlania wymaga przydzielenia przez OpenGL obszaru pamięci, w którym przechowywane są polecenia. Po zakończeniu używania listy wyświetlania należy ją usunąć, aby maszyna OpenGL mogła zwolnić zajmowaną przez listę pamięć. Listę wyświetlania usuwa się za pomocą funkcji *glDeleteLists()*:

```
void glDeleteLists(GLuint listname, GLsizei range);
```

Wywołanie tej funkcji spowoduje zwolnienie pamięci zajmowanej przez listy o identyfikatorach z przedziału od *listName* do *listName + range - 1*. Jeśli którykolwiek z tych identyfikatorów odnosi się do nieistniejącej listy, to zostanie po prostu zignorowany. Gdy parametr *range* będzie miał wartość 0, to wywołanie funkcji nie spowoduje żadnej akcji, a jeśli wartość ujemną, to wystąpi błąd.

Jako że opanowana została już umiejętność tworzenia, wypełniania, wykonywania i usuwania listy wyświetlania, można zastosować ją we wspomnianym już programie rysującym zbiór piramid. Najpierw utworzona zostanie oczywiście lista wyświetlania:

```
GLuint pyramidList;
```

```
pyramidList = glGenLists(1);
```

Następnie trzeba będzie wypełnić ją poleceniami. W tym celu należy zmienić postać zaprezentowanej wcześniej funkcji DrawPyramid(). Ponieważ polecenia tworzenia wierzchołków umieszczone zostaną na liście, to funkcja ta będzie wywoływana tylko jeden raz na początku programu i wobec tego należy zmienić jej nazwę na bardziej odpowiednią InitializePyramid(). Wewnątrz tej samej funkcji utworzona zostanie lista wyświetlnia. Parametrem funkcji InitializePyramid() będzie referencia identyfikatora listy:

```
void InitializePyramid(GLuint &pyramidList)
{
    pyramidList = glGenLists(1);

    glNewList(pyramidList, GL_COMPILE);
    glBegin(GL_TRIANGLE_FAN);
    glVertex3f(0.0, 1.0, 0.0);
    glVertex3f(-1.0, 0.0, 1.0);
    glVertex3f(1.0, 0.0, 1.0);
    glVertex3f(1.0, 0.0, -1.0);
    glVertex3f(-1.0, 1.0, -1.0);
    glEnd();
    glEndList();
}
```

Odtąd aby narysować piramidę, trzeba będzie wykonać najpierw odpowiednie przekształcenia, a następnie wywołać funkcję

```
glCallList(pyramidList);
```

Po zakończeniu korzystania z listy wyświetlnia (w tym przypadku prawdopodobnie związane to będzie z zakończeniem pracy programu), trzeba usunąć listę w następujący sposób:

```
glDeleteLists(pyramidList, 1);
```

Chociaż w przypadku omówionego programu zastosowanie listy wyświetlnia nie spowoduje istotnej poprawy efektywności jego wykonania, to jednak stanowi on dobrą ilustrację sposobu posługiwania się listami wyświetlnia.

## Listy wyświetlnia i tekstury

Na listach wyświetlnia można umieszczać dowolne funkcje OpenGL związane z teksturami. Może to stanowić zachętą do umieszczenia na liście wyświetlnia polecień związanych z tworzeniem i konfiguracją tekstury. Rozwiążanie takie byłoby z pewnością optymalne, gdyby nie istniały obiekty tekstur. Obiekty tekstur nie tylko ułatwiają zastosowanie tekstur w programie, ale także optymalizują ich użycie dając w efekcie poprawę efektywności większą niż możliwa by uzyskać stosując w tym celu listy wyświetlnia. Dlatego też najlepiej jest tworzyć tekstury w prezentowany dotąd sposób — przez wiązanie ich z obiektami tekstur. Natomiast wywołania funkcji glBindTexture(), glTexCoord() i nawet glTexEnv() można później umieszczać na listach wyświetlnia, ponieważ funkcje te związane są z zastosowaniem tekstur, a nie ich tworzeniem.

## Przykład: animacja robota z użyciem list wyświetlania

Zastosowanie list wyświetlania zademonstrowane zostanie teraz na przykładzie nowej wersji programu animacji robota przedstawionego w rozdziale 5. Zamiast tworząc grafikę kolejnej klatki animacji rysować poszczególne elementy robota, można umieścić je na osobnych listach wyświetlania. Na listach tych umieścić można także te z przekształceń, które nie są związane z ruchem robota. Nowa wersja programu tworzy najpierw listy wyświetlania elementów robota, a następnie wykonuje je tworząc w ten sposób kolejne klatki animacji robota. Poniżej zaprezentowany został kod funkcji stosującej listy wyświetlania. Należy zwrócić uwagę na to, że w przykładzie tym stosuje się zagnieżdzone listy wyświetlania. Jedna z list wyświetla sześcian i wykorzystywana jest przez listy wyświetlania poszczególnych elementów robota, które najpierw wykonują odpowiednie przekształcenia przesunięcia i skalowania.

```
void InitializeLists()
{
    // tworzy 5 list
    g_cube = glGenLists(5);

    glNewList(g_cube, GL_COMPILE);
    glBegin(GL_POLYGON);
        glVertex3f(0.0f, 0.0f, 0.0f);      // górná ściana
        glVertex3f(0.0f, 0.0f, -1.0f);
        glVertex3f(-1.0f, 0.0f, -1.0f);
        glVertex3f(-1.0f, 0.0f, 0.0f);
        glVertex3f(0.0f, 0.0f, 0.0f);      // przednia ściana
        glVertex3f(-1.0f, 0.0f, 0.0f);
        glVertex3f(-1.0f, -1.0f, 0.0f);
        glVertex3f(0.0f, -1.0f, 0.0f);
        glVertex3f(0.0f, 0.0f, 0.0f);      // prawa ściana
        glVertex3f(0.0f, -1.0f, 0.0f);
        glVertex3f(0.0f, -1.0f, -1.0f);
        glVertex3f(0.0f, 0.0f, -1.0f);
        glVertex3f(-1.0f, 0.0f, 0.0f);     // lewa ściana
        glVertex3f(-1.0f, 0.0f, -1.0f);
        glVertex3f(-1.0f, -1.0f, -1.0f);
        glVertex3f(-1.0f, -1.0f, 0.0f);
        glVertex3f(0.0f, 0.0f, 0.0f);      // dolna ściana
        glVertex3f(0.0f, -1.0f, -1.0f);
        glVertex3f(-1.0f, -1.0f, -1.0f);
        glVertex3f(-1.0f, 0.0f, 0.0f);     // tylna ściana
        glVertex3f(-1.0f, 0.0f, -1.0f);
        glVertex3f(-1.0f, -1.0f, -1.0f);
        glVertex3f(0.0f, -1.0f, -1.0f);
    glEnd();
    glEndList();

    g_head = g_cube + 1;
    glNewList(g_cube, GL_COMPILE);
    glPushMatrix();
        glColor3f(1.0f, 1.0f, 1.0f);    // kolor biały
        glTranslate(1.0f, 2.0f, 0.0f);
        glScalef(2.0f, 2.0f, 2.0f);    // głowa robota jest sześcianem 2x2x2
    glCallList(g_cube);
```

```
    glPopMatrix();
    glEndList();

    g_torso = g_cube + 2;
    glNewList(g_torso, GL_COMPILE);
    glPushMatrix();
        glColor3f(0.0f, 0.0f, 1.0f); // kolor niebieski
        glTranslate(1.5f, 0.0f, 0.0f);
        glScalef(3.0f, 5.0f, 2.0f); // korpus robota jest prostopadłościanem 3x5x2
        glCallList(g_cube);
    glPopMatrix();
    glEndList();

    g_arm = g_cube + 3;
    glNewList(g_arm, GL_COMPILE);
    glPushMatrix();
        glColor3f(1.0f, 0.0f, 0.0f); // kolor czerwony
        glScalef(1.0f, 4.0f, 1.0f); // ramię robota jest prostopadłościanem 1x4x1
        glCallList(g_cube);
    glPopMatrix();
    glEndList();

    g_leg = g_cube + 4;
    glNewList(g_leg, GL_COMPILE);
    glPushMatrix();
        glColor3f(1.0f, 1.0f, 1.0f); // kolor żółty
        glScalef(1.0f, 5.0f, 1.0f); // nogą robota jest prostopadłościanem 1x5x1
        glCallList(g_cube);
    glPopMatrix();
    glEndList();
} // koniec funkcji InitializeLists()
```

Ponieważ program ten jest bardzo prosty, to zastosowanie list wyświetlania nie poprawia znacząco efektywności jego działania. Na niektórych maszynach udało się jednak zaobserwować wzrost szybkości tworzenia klatek animacji o 5 do 10%, ale na szybszych komputerach efektywność działania obu wersji programu była już praktycznie taka sama.

## Tablice wierzchołków

W przykładzie programu rysującego piramidy zastosowanie funkcji OpenGL ograniczało się do tworzenia wierzchołków. W praktyce często zdarza się, że programy tworzące grafikę trójwymiarową przetwarzają dużą liczbę wierzchołków lub związanych z nimi danych. Programy zaprezentowane dotąd w tej książce tworzyły dość proste obiekty i dlatego wierzchołki tych obiektów mogły być definiowane bezpośrednio w kodzie programu. Jednak typowa gra korzysta zwykle z modeli składających się z setek, a nawet tysięcy wielokątów. Oczywiście tworzenie tak skomplikowanych modeli za pomocą definicji wierzchołków umieszczonych bezpośrednio w kodzie jest wyjątkowo niepraktyczne. Zamiast tego stosuje się zwykle jedno z przedstawionych niżej rozwiązań.

- ◆ **Proceduralne tworzenie modelu.** Niektóre z tworzonych obiektów można opisać za pomocą równań lub tworzyć ich losowe wartości właściwości w trakcie działania programu. Dobrym przykładem takich obiektów są frakty. Ich dane tworzone są poprzez wywołanie procedury, która tworzy za każdym razem takie same wartości.
- ◆ **Załadowanie modelu z pliku.** Obecnie dostępnych jest wiele doskonałych pakietów umożliwiających wizualne tworzenie modeli i zapisanie danych opisujących model w pliku. Dane te mogą być następnie czytane przez program. Rozwiązanie takie oferuje zdecydowanie największą elastyczność tak z punktu widzenia tworzenia modeli, jak i zarządzania ich danymi. Ładowanie modeli omówione zostanie szerzej w dalszej części książki.

Niezależnie od tego, które z powyższych rozwiązań zostanie wybrane, to z pewnością przy tworzeniu kolejnej klatki grafiki nie będzie trzeba tworzyć całego modelu od nowa. Dane dostarczone przez jedno z tych rozwiązań będzie można umieścić w buforze i przetwarzać. Na tym właśnie opiera się koncepcja tablic wierzchołków.

W procesie tworzenia modelu można wyróżnić przedstawione niżej etapy:

1. Tworzenie danych modelu na drodze proceduralnej bądź przez pobranie ich z pliku.
2. Dane modelu umieszcza się w tablicy lub zbiorze tablic (na przykład współrzędne wierzchołków można umieścić w jednej tablicy, dane opisujące ich kolor w innej, a składowe normalne w jeszcze innej i tak dalej).
3. Przy tworzeniu grafiki OpenGL pobiera się kolejne elementy tablicy (lub tablic) i wywołuje dla każdego z nich odpowiednie funkcje OpenGL. Alternatywnie można też korzystać z pewnych podzbiorów danych umieszczonych w tablicach.

Do realizacji opisanego sposobu działania w praktyce wystarczy znajomość programowania pętli w języku C i przedstawionych dotąd poleceń języka OpenGL. Jednak po nieważ rozwiązanie takie stosowane jest powszechnie, to OpenGL posiada wbudowaną obsługę tablic wierzchołków, która optymalizuje efektywność ich wykorzystania.

## Obsługa tablic wierzchołków w OpenGL

Podobnie jak w przypadku innych mechanizmów udostępnianych przez OpenGL przed skorzystaniem z obsługi tablic wierzchołków trzeba najpierw ją aktywować. Choć można spodziewać się w tym przypadku kolejnego zastosowania funkcji `glEnable()`, to jednak OpenGL dostarcza osobnej pary funkcji umożliwiających zarządzanie obsługą tablic wierzchołków:

```
void glEnableClientState(GLenum array);
void glDisableClientState(GLenum array);
```

Parametr `array` jest znacznikiem określającym rodzaje tablic, których obsługę włącza się (lub wyłącza). Dlatego też dla każdego rodzaju danych związanych z opisem wierzchołków (na przykład współrzędne, kolor, normalne) należy utworzyć osobne tablice i osobno aktywować obsługę wykorzystywanych typów tablic. Wartości znacznika `array` dla różnych typów tablic prezentuje tabela 10.2.

**Tabela 10.2.** Znaczniki typów tablic

Znacznik	Znaczenie
GL_COLOR_ARRAY	Włącza obsługę tablic opisujących kolor każdego wierzchołka
GL_EDGE_FLAG_ARRAY	Włącza obsługę tablic zawierających znaczniki krawędzi dla każdego wierzchołka
GL_INDEX_ARRAY	Włącza obsługę tablic zawierających indeksy koloru dla każdego wierzchołka
GL_NORMAL_ARRAY	Włącza obsługę tablic zawierających składowe normalnej każdego wierzchołka
GL_TEXTURE_COORD_ARRAY	Włącza obsługę tablic zawierających współrzędne tekstury dla każdego wierzchołka
GL_VERTEX_ARRAY	Włącza obsługę tablic zawierających współrzędne każdego wierzchołka

Dokumentacja OpenGL określa wszystkie typy tablic wspólnym mianem *tablic wierzchołków*, ponieważ wszystkie zawierają dane związane z wierzchołkami. Jest to nieco mylące, ponieważ jedna z tych tablic opisuje współrzędne wierzchołków i w skrócie także nazywana jest *tablicą wierzchołków*. Nazwy znaczników określających typ tablicy zachowują na szczęście podobieństwo do nazw funkcji OpenGL, które wywoływać można za pomocą danych umieszczonych w konkretnej tablicy (na przykład `glVertex()` dla tablicy wierzchołków, `glColor()` dla tablicy kolorów, `glTexCoord()` dla współrzędnych tekstury i tak dalej).

## Stosowanie tablic wierzchołków

Kolejny etap po aktywowaniu wykorzystywanych typów tablic polega na dostarczeniu maszynie OpenGL tablic wypełnionych danymi. Zadaniem programisty jest utworzenie odpowiednich tablic i wypełnienie ich danymi (utworzonymi na drodze proceduralnej lub wczytanymi z pliku), a następnie powiadomienie maszyny OpenGL o istnieniu tych tablic. W tym celu trzeba wywoływać różne funkcje w zależności od typu wykorzystywanych tablic. Przyjrzymy się im zatem bliżej.

Funkcja `glColorPointer()` definiuje tablicę kolorów i posiada następujący prototyp:

```
void glColorPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *array);
```

Parametr *size* określa liczbę składowych opisujących kolor wierzchołka i posiada wartość 3 lub 4. Parametr *type* reprezentuje typ elementów tablicy czyli typ, za pomocą którego reprezentowane są składowe kolorów. Może przyjmować on jedną z wartości `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_FLOAT` lub `GL_DOUBLE`. Parametr *stride* określa liczbę bajtów, które oddzielają opis kolorów kolejnych wierzchołków. Jeśli pomiędzy bajtami opisu kolorów kolejnych wierzchołków nie występują dodatkowe bajty, to wartość parametru *stride* wynosi 0. Oznacza to, że parametr *array* jest wskaźnikiem tablicy kolorów (a dokładniej rzecz biorąc pierwszego elementu tablicy kolorów).

Funkcja `glEdgeFlagPointer()` definiuje tablicę znaczników krawędzi. Poniżej zaprezentowany został jej prototyp:

```
void glEdgeFlagPointer(GLsizei stride, const GLboolean *array);
```

Znaczniki krawędzi wykorzystywane są podczas wyświetlania samych krawędzi wielokątów. Tablica znaczników krawędzi pozwala określić, które z nich mają być rysowane. Podobnie jak w przypadku poprzedniej funkcji parametr *stride* określa liczbę bajtów oddzielających kolejne pozycje tablicy wartości logicznych wskazywanej przez parametr *array*.

Funkcja `glIndexPointer()` definiuje tablicę indeksów kolorów dla wierzchołków w przypadku, gdy wykorzystywany jest tryb wyświetlnia stosujący paletę kolorów. Prototyp funkcji zdefiniowany jest następująco:

```
void glIndexPointer(GLenum type, GLsizei stride, const GLvoid *array);
```

Parametr *type* reprezentuje typ elementów tablicy indeksów i może przyjmować wartości `GL_SHORT`, `GL_INT` lub `GL_DOUBLE`. Parametry *stride* i *array* posiadają takie samo znaczenie jak w przypadku poprzednich funkcji.

Funkcja `glNormalPointer()` definiuje tablicę wektorów normalnych dla poszczególnych wierzchołków i posiada następujący prototyp:

```
void glNormalPointer(GLenum type, GLsizei stride, const GLvoid *array);
```

Każdy wektor normalny opisany jest za pomocą trzech składowych, których typ opisuje parametr *type*. Może on przyjmować wartości `GL_BYTE`, `GL_SHORT`, `GL_INT`, `GL_FLOAT` lub `GL_DOUBLE`. Parametry *stride* i *array* posiadają takie samo znaczenie jak w przypadku poprzednich funkcji.

Funkcja `glTexCoordPointer()` definiuje tablicę współrzędnych tekstury dla wszystkich wierzchołków. Jej prototyp zdefiniowany jest następująco:

```
glTexCoordPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *array);
```

Parametr *size* określa liczbę współrzędnych tekstury i może przyjmować wartości 1, 2, 3 lub 4. Parametr *type* określa typ danych reprezentujących współrzędne tekstury i może przyjmować wartości `GL_SHORT`, `GL_INT`, `GL_FLOAT` lub `GL_DOUBLE`. Parametry *stride* i *array* posiadają takie samo znaczenie jak w przypadku poprzednich funkcji.

Funkcja `glVertexPointer()` definiuje tablicę współrzędnych wierzchołków. Funkcja ta zdefiniowana jest następująco:

```
void glVertexPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *array);
```

Parametr *size* określa liczbę współrzędnych wierzchołka i może przyjmować wartości 2, 3 lub 4. Parametr *type* określa typ danych reprezentujących współrzędne wierzchołków i może przyjmować wartości `GL_SHORT`, `GL_INT`, `GL_FLOAT` lub `GL_DOUBLE`. Parametry *stride* i *array* posiadają takie samo znaczenie jak w przypadku poprzednich funkcji.

Po zdefiniowaniu różnych typów tablic wierzchołków można skorzystać z zawartych w nich danych za pomocą odpowiednich funkcji OpenGL. A teraz należy przejść do szczegółowego omówienia tych funkcji.



W danym momencie można posiadać tylko jedną tablicę danego typu. Oznacza to, że jeśli tablice wierzchołków programista zamierza wykorzystać w celu reprezentacji wielu modelów, to albo musi umieścić razem opisujące je dane w jednym i tym samy zestawie tablic, albo przełączać osobne zestawy tablic dla poszczególnych obiektów za pomocą funkcji `gl*Pointer()`. Choć pierwsza metoda jest nieco szybsza, ponieważ nie wymaga zmiany stanu maszyny OpenGL, to jednak druga metoda oferuje lepsze zarządzanie danymi.

## glDrawArrays()

Wywołanie tej funkcji powoduje, że OpenGL korzysta ze wszystkich aktywnych typów tablic wierzchołków i tworzy podstawowe elementy grafiki na podstawie zawartych w nich danych. Aby zrozumieć dokładnie, w jaki sposób działa ta funkcja, należy przyjrzeć się jej prototypowi:

```
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

Parametr `mode` ma takie samo znaczenie jak parametr funkcji `glBegin()`: określa typ podstawowych elementów grafiki tworzonych na podstawie danych wierzchołków. Może przyjmować wartości `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_TRIANGLES`, `GL_QUAD_STRIP`, `GL_QUADS` lub `GL_POLYGON`. Parametr `first` określa indeks elementu tablicy, od którego OpenGL powinien rozpoczęć działanie, a parametr `count` specyfikuje zakres przetwarzanych elementów tablicy. Należy zwrócić uwagę, że po wywołaniu funkcji `glDrawArrays()` bieżący stan maszyny związany z typem przetwarzanych tablic nie jest zdefiniowany. Na przykład przetwarzanie przez funkcję `glDrawArrays()` tablicy normalnych powoduje, że bieżąca normalna nie jest zdefiniowana.

## glDrawElements()

Funkcja ta jest bardzo podobna do funkcji `glDrawArrays()`, ale posiada nieco większe możliwości. Za pomocą funkcji `glDrawArrays()` można przejrzeć ciągły zakres elementów zgodnie z ich uporządkowaniem w tablicy. Funkcja `glDrawElements()` pozwala natomiast określić dowolny porządek przetwarzania wierzchołków. Jej prototyp wygląda następująco:

```
glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid *indices);
```

Parametr `mode` posiada takie samo znaczenie jak dla funkcji `glDrawArrays()`, podobnie parametr `count`. Parametr `type` określa typ elementów wskazywanych przez parametr `indices` i może przyjmować jedną z wartości `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT` lub `GL_UNSIGNED_INT`. Parametr `indices` reprezentuje tablicę zawierającą wierzchołki, które mają być narysowane.

Aby zrozumieć, na czym polega zaleta tej funkcji należy zaznaczyć, że pozwala ona nie tylko przetwarzać wierzchołki w dowolnym porządku, ale także przetwarzać ten sam wierzchołek wielokrotnie. W przypadku obiektów używanych w grach większość wierzchołków należy do więcej niż jednego wielokąta. Przechowując opis wierzchołka należącego do wielu wielokątów tylko jeden raz można zaoszczędzić sporo pamięci. Dodatkowo dobre implementacje OpenGL wykonują operacje na takim wierzchołku tylko raz,

co zwiększa w oczywisty sposób efektywność wykonywania programu. Więcej informacji na ten temat zawiera podrozdział „Blokowanie tablic” wierzchołków znajdujący się w dalszej części bieżącego rozdziału.

## glDrawRangeElements()

Funkcję tę wprowadzono dopiero w wersji 1.2 specyfikacji OpenGL, nie jest więc ona dostępna w starszych implementacjach. Używamy jej podobnie do funkcji `glDrawElements()`. Różnica polega na ograniczeniu zakresu przetwarzanych elementów tablicy wierzchołków. Na przykład rysując obiekt, który zdefiniowany jest przez pierwszych sto wierzchołków tablicy składającej się z tysiąca elementów, można poinformować o tym OpenGL właśnie za pomocą funkcji `glDrawRangeElements()`. Może to spowodować optymalizację procesu tworzenia grafiki, jeśli OpenGL przeniesie dane stu wierzchołków do pamięci o szybszym dostępie. Prototyp funkcji `glDrawRangeElements()` wygląda następująco:

```
void glDrawRangeElements(GLenum mode, GLuint start, GLuint end, GLsizei count,
                           GLenum type, const GLvoid *indices);
```

Parametry `mode`, `type` i `count` posiadają takie samo znaczenie jak w przypadku funkcji `glDrawElements()`. Parametry `start` i `end` określają dolną i górną granicę przetwarzanego zakresu indeksów tablicy wskazywanej przez `indices`.

## glArrayElement()

Funkcja ta stanowi najmniej efektywny sposób przetwarzania tablic wierzchołków. Nie działa ona w odniesieniu do pewnego zakresu wierzchołków, lecz do pojedynczego elementu tablicy wierzchołków. Funkcja `glArrayElement()` posiada następujący prototyp:

```
void glArrayElement(GLint index);
```

Parametr `index` określa oczywiście wartość indeksu wierzchołka, który ma być przetwarzany.

Dla lepszego zrozumienia sposobów korzystania z tablic wierzchołków należy podsumować uzyskane dotąd informacje. Tablice wierzchołków trzeba najpierw wypełnić danymi, które tworzy się za pomocą pewnego algorytmu lub wczytuje z pliku. Dane te opisują zbiór wierzchołków tworzących model pewnego obiektu. Informacja dotycząca każdego wierzchołka może opisywać jego współrzędne, kolor, współrzędne tekstury, znacznik krawędzi i wektor normalny. Każda z tych informacji umieszczona jest w tablicy innego typu. Jeśli korzysta się z tablicy danego typu, to trzeba aktywować jej przetwarzanie w OpenGL. Poszczególne tablice wybiera się następnie za pomocą funkcji `gl*Pointer()`.

Aby skorzystać z danych umieszczonych w tablicach wierzchołków i narysować obiekt, należy wywołać jedną z omówionych wyżej funkcji. Dla każdego wierzchołka OpenGL pobierze dane związane z opisem danego wierzchołka i wywoła dla nich odpowiednią funkcję. Na przykład dla danych opisujących kolor wierzchołka będzie to funkcja `glColor()`, dla danych reprezentujących składowe wektora normalnego funkcja `glNormal()` i tak dalej. Oczywiście działanie OpenGL nie polega w tych przypadkach na wywołaniu wymienionych funkcji (wtedy można by zrobić to samodzielnie i w ogóle pominąć konsepcję tablic wierzchołków), ale efekt jest taki sam.

## Tablice wierzchołków i tekstury wielokrotne

Stosowanie tekstur wielokrotnych omówionych w rozdziale 9. w połączeniu z tablicami wierzchołków wymaga pewnych dodatkowych zabiegów. Każda jednostka tekstury posiada własny zbiór stanów i dlatego tablice wierzchołków muszą być aktywowane dla każdej jednostki tekstur osobno. Każda z jednostek tekstur dysponuje też własnym wskaźnikiem tablicy wierzchołków przechowującej współrzędne tekstury.

Implementacje umożliwiające stosowanie tekstur wielokrotnych przyjmują, że domyślnie aktywna jest pierwsza jednostka tekstury. Wywołania funkcji `glTexCoordPointer()` oraz `glEnableClientState()` i `glDisableClientState()` z parametrem `GL_TEXTURE_COORD_ARRAY` mają wpływ jedynie na aktywną jednostkę tekstury. Aby więc użyć tablic wierzchołków także w przypadku pozostałych jednostek tekstur, trzeba aktywować je za pomocą poniższej funkcji:

```
void glClientActivateTextureARB(enum texture);
```

Parametr `texture` określa jednostkę tekstury, która jest aktywowana i ma postać `GL_TEXTUREi_ARB`, gdzie *i* jest wartością z przedziału od 0 do `GL_MAX_TEXTURE_UNITS_ARB` - 1.

Po aktywowaniu wybranej jednostki tekstury można wywołać funkcję `glEnableClientState()` lub `glDisableClientState()`, aby włączyć lub wyłączyć dla niej mechanizm tablic wierzchołków oraz funkcję `glTexCoordPointer()` w celu zdefiniowania tablicy współrzędnych tekstury. Domyślnie mechanizm tablic wierzchołków jest wyłączony dla wszystkich jednostek tekstur. Poniższy fragment kodu przedstawia sposób aktywacji tablic wierzchołków dla pierwszych dwóch jednostek tekstur:

```
// aktywuje tablice wierzchołków dla jednostki tekstury o indeksie 0  
glEnableClientState(GL_TEXTURE_COORD_ARRAY);  
  
// definiuje tablicę współrzędnych tekstury dla jednostki tekstury o indeksie 0  
glTexCoordPointer(2, GL_FLOAT, 0, (GLvoid *)texUnit0Vertices);  
  
// wybiera jednostkę tekstury o indeksie 1  
glClientActiveTextureARB(GL_TEXTURE1_ARB);  
  
// aktywuje tablice wierzchołków dla jednostki tekstury o indeksie 1  
glEnableClientState(GL_TEXTURE_COORD_ARRAY);  
  
// definiuje tablicę współrzędnych tekstury dla jednostki tekstury o indeksie 1  
glTexCoordPointer(2, GL_FLOAT, 0, (GLvoid *)texUnit1Vertices);
```

Po aktywowaniu i zdefiniowaniu tablicy wierzchołków dla każdej z używanych jednostek tekstur można już korzystać z funkcji `glDrawArrays()` i `glDrawElements()` w normalny sposób.

## Blokowanie tablic wierzchołków

Wiele implementacji OpenGL udostępnia rozszerzenie pozwalające zakładać i zdejmować blokady tablic wierzchołków. Zablokowanie tablic wierzchołków informuje maszynę OpenGL, że do momentu zdjęcia blokady nie będą modyfikowane znajdujące się w nich dane. Dzięki temu OpenGL zamiast wielokrotnie wykonywać przekształcenia

wierzchołków wykonuje je tylko raz, a wynik przechowuje w buforze. W rezultacie takiego działania można uzyskać znaczne przyspieszenie tworzenia grafiki (zwłaszcza, jeśli zawiera ona wiele wspólnych wierzchołków lub wykonywane jest wiele przebiegów dla tych samych danych). Ponieważ informacja o wierzchołkach zostaje w wyniku zablokowania tablic skompilowana, to nazwą tego rozszerzenia jest `GL_EXT_compiled_vertex_array`. Z rozszerzeniem tym związane są następujące funkcje:

```
void glLockArraysEXT(GLint first, GLsizei count);
void glUnlockArraysEXT();
```

Parametr `first` jest indeksem pierwszego wierzchołka, który ma być zablokowany, a parametr `count` określa liczbę blokowanych wierzchołków.

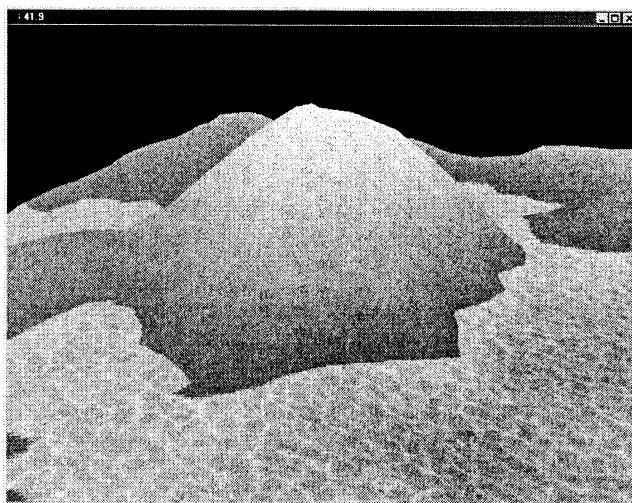
Przykład kodu sprawdzającego dostępność i używającego mechanizmu blokowania tablic zawiera kod programu, który omawiany jest w następnym podrozdziale.

## Przykład: ukształtowanie terenu po raz drugi

W rozdziale 8. omówiony został przykład programu prezentującego ukształtowanie terenu na podstawie mapy wysokości. Teraz zmodyfikowany zostanie jego kod tak, by korzystał z tablic wierzchołków. Kompletny kod nowej wersji programu znajduje się na dysku CD, a tutaj przedstawione są jedynie fragmenty związane z zastosowaniem tablic wierzchołków. Jak pokazują rysunki 10.1 i 10.2, jedyna różnica w działaniu obu wersji programu polega na szybkości tworzenia grafiki.

**Rysunek 10.1.**

Mapa ukształtowania terenu prezentowana przez program z rozdziału 8.

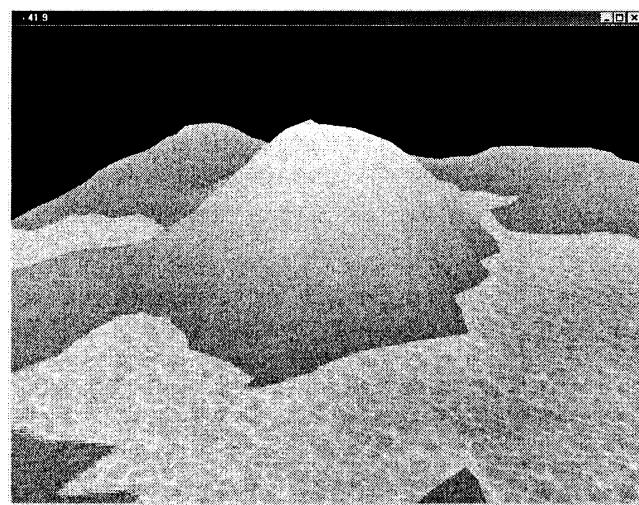


Na początku trzeba oczywiście zadeklarować wykorzystywane tablice:

```
GLuint g_indexArray[MAP_X * MAP_Z * 6]; // tablica indeksów wierzchołków
float g_terrain[MAP_X * MAP_Z][3]; // mapa wysokości terenu 256x256
float g_colorArray[MAP_X * MAP_Z][3]; // tablica kolorów
float g_texcoordArray[MAP_X * MAP_Z][2]; // tablica współrzędnych tekstury
```

**Rysunek 10.2.**

Mapa ukształtowania terenu tworzona przez wersję programu stosującą tablice wierzchołków



Tablica g\_terrain przechowuje będzie dane o położeniu wierzchołków, tablica g\_colorArray dane o ich kolorze, tablica gTexCoordArray współrzędne tekstury. Tablica g\_indexArray zawierać będzie indeksy wierzchołków i używana będzie przez funkcję glDrawElements().

Funkcja InitializeArrays() umieszcza dane w każdej z wymienionych tablic i aktywuje mechanizm tablic wierzchołków:

```
void InitializeArrays()
{
    // zmienne wskazujące bieżącą pozycję tablicy indeksów
    int index = 0;
    int currentVertex;

    // przegląda w pętli wszystkie wierzchołki mapy terenu
    for (int z = 0; z < MAP_Z; z++)
    {
        for (int x = 0; x < MAP_X; x++)
        {
            // wierzchołki uporządkowane są od lewej do prawej, z góry w dół
            currentVertex = z * MAP_X + x;

            // umieszcza wartości w tablicy kolorów
            g_colorArray[currentVertex][0] = g_colorArray[currentVertex][1] =
                g_colorArray[currentVertex][2] = g_terrain[x + MAP_X * z][1]/255.0f;

            // umieszcza wartości w tablicy współrzędnych tekstury
            g_texcoordArray[currentVertex][0] = (float) x;
            g_texcoordArray[currentVertex][1] = (float) z;
        }
    }

    for (z = 0; z < MAP_Z - 1; z++)
    {
        for (int x = 0; x < MAP_X; x++)
        {
            // umieszcza wartości w tablicy indeksów
            g_indexArray[index] = currentVertex;
            index++;
        }
    }
}
```

```

    {
        currentVertex = z * MAP_X + x;
        g_indexArray[index++] = currentVertex + MAP_X;
        g_indexArray[index++] = currentVertex;
    }
}

// aktywuje typy używanych tablic
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);

// definiuje tablice wierzchołków
glVertexPointer(3, GL_FLOAT, 0, g_terrain);
glColorPointer(3, GL_FLOAT, 0, g_colorArray);
glTexCoordPointer(2, GL_FLOAT, 0, g_texcoordArray);
} // koniec funkcji InitializeArrays()

```

Funkcja `InitializeArrays()` przygotowuje tablice wierzchołków do użycia. Jako że analizowany przykład korzystać będzie z rozszerzenia `GL_EXT_compiled_vertex_array`, trzeba sprawdzić, czy jest ono dostępne. Służy do tego poniższy fragment kodu, który po ustaleniu, że mechanizm blokowania tablic wierzchołków jest dostępny, pobiera wskaźniki funkcji `glEXTLockArrays()` i `glEXTUnlockArrays()`.

```

// sprawdza dostępność blokowania tablic wierzchołków
char *extList = (char *) glGetString(GL_EXTENSIONS);

if (extList && strstr(extList, "GL_EXT_compiled_vertex_array"))
{
    // pobiera wskaźniki funkcji rozszerzeń
    glLockArraysEXT = (PFNGLLOCKARRAYSEXTPROC)
        wgGetProcAddress("glLockArraysEXT");
    glUnlockArraysEXT = (PFNGLUNLOCKARRAYSEXTPROC)
        wgGetProcAddress("glUnlockArraysEXT");
}

```

Stosowanie blokad tablic wierzchołków jest całkiem proste. Ponieważ w tym programie nie będzie zmieniany wyglądu terenu, po zainicjowaniu tablic można je od razu zablokować:

```

// blokuje tablice wierzchołków, jeśli mechanizm blokowania jest dostępny
if (glLockArraysEXT)
    glLockArraysEXT(0, MAP_X * MAP_Z);

```

Blokadę zwalnia się kończąc działanie programu:

```

// zwalnia blokadę tablic, jeśli mechanizm blokowania jest dostępny
if (glUnlockArraysEXT)
    glUnlockArraysEXT();

```

Dane wyświetlane są za pomocą funkcji `glDrawElements()`. Ponieważ ze względu na efektywność tworzenia grafiki dane o wierzchołkach zapisane są tak, by tworzyły łańcuchy trójkątów, to koniecznych jest wiele wywołań funkcji `glDrawElements()`. Każde takie wywołanie przetwarza inny zakres elementów tablicy wierzchołków:

```
// rysuje w pętli kolejne łańcuchy trójkątów
for (int z = 0; z < MAP_Z-1; z++)
{
    // rysuje trójkąty łańcucha
    glDrawElements(GL_TRIANGLE_STRIP, MAP_X * 2, GL_UNSIGNED_INT,
        &g_indexArray[z * MAP_X * 2]);
}
```

Stworzenie wersji programu używającej tablice wierzchołków zajęło niecałą godzinę. Nowa wersja działa o około 10 do 15% szybciej od dotychczasowej. W praktyce oznacza to tworzenie od 5 do 10 dodatkowych klatek grafiki na sekundę. Jest to istotna poprawa działania uzyskana przy stosunkowo małym nakładzie pracy.

## Podsumowanie

W rozdziale tym przedstawione zostały dwa wydajne sposoby zwiększenia efektywności grafiki OpenGL. Ponieważ ich możliwości pokrywają się częściowo, zasadne jest pytanie o to, którą z nich należy preferować. Jeśli program rysuje wiele wierzchołków, które nie zmieniają często swego stanu, to należy skorzystać raczej z tablic wierzchołków. Gdy stan grafiki ulega częstym zmianom, lepszym rozwiązaniem będą listy wyświetlanie. Różnicę w efektywności obu metod można ustalić w konkretnym przypadku doświadczalnie. Niezależnie jednak od tego, która z metod zostanie wybrana zawsze uzyskuje się wzrost efektywności tworzenia grafiki (nawet, jeśli będzie on niewielki). Nabierając stopniowo doświadczenia w korzystaniu z obu mechanizmów będzie można tworzyć coraz bardziej efektywne rozwiązania.

## Rozdział 11.

# Wyświetlanie tekstów

Każda aplikacja graficzna wyświetla pewne teksty na ekranie. Mogą to być na przykład pozycje menu, teksty komunikatów, napisy prezentowane przez wygaszacz ekranu bądź używane do uzyskania efektów specjalnych. W rozdziale tym przedstawione zostaną najczęściej stosowane techniki tworzenia tekstów w OpenGL. Obejmować one będą tworzenie i korzystanie z czcionek rastrowych, konturowych i pokrytych tekstrurą.

W rozdziale tym omawiamy:

- ◆ czcionki rastrowe;
- ◆ czcionki konturowe;
- ◆ czcionki pokryte tekstrurą.

## Czcionki rastrowe

Czcionki rastrowe są najczęściej wykorzystywany sposobem wyświetlania tekstów w OpenGL i pozwalają na uzyskanie najlepszego efektu. Jednocześnie tworzenie ich jest bardzo proste i odbywa się za pomocą funkcji `wglUseFontBitmaps()`. Tworzy ona czcionki rastrowe na podstawie czcionek dostępnych w systemie operacyjnym.

Aby zastosować czcionki rastrowe do wyświetlania tekstów, trzeba najpierw utworzyć 96 list wyświetlania, które rysować będą poszczególne znaki czcionki. W tym celu należy skorzystać z funkcji `glGenLists()` w następujący sposób:

```
unsigned int base;  
  
base = glGenLists(96);
```

Następnie tworzymy czcionkę systemu Windows korzystając z funkcji `CreateFont()` zdefiniowanej w następujący sposób:

```
HFONT CreateFont(  
    int nHeight,           // wysokość czcionki  
    int nWidth,            // średnia szerokość znaku  
    int nEscapement,       // kąt pochylenia  
    int nOrientation,      // kąt orientacji względem linii bazowej  
    int fnWeight,          // waga czcionki
```

```

    DWORD fdwItalic,           // znacznik kursywy
    DWORD fdwUnderline,        // znacznik podkreślenia
    DWORD fdwStrikeOut,        // znacznik wytłuszczenia
    DWORD fdwCharSet,          // identyfikator zbioru znaków
    DWORD fdwOutputPrecision,  // dokładność prezentacji
    DWORD fdwClipPrecision,   // dokładność obcinania
    DWORD fdwQuality,          // jakość prezentacji
    DWORD fdwPitchAndFamily,   // rodzina czcionek
    LPCTSTR lpszFace          // nazwa czcionki
);

```

Funkcja ta zwraca uchwyty czcionki systemu Windows. Dla czcionki tej należy wybrać kontekst urządzenia, który następnie przekazuje się jako parametr funkcji `wglUseFontBitmaps()`:

```

HFONT hFont;           // uchwyty czcionki systemu Windows

// tworzy czcionkę Courier o wielkości 14 punktów
hFont = CreateFont(fontSize, 0, 0, FW_BOLD, FALSE, FALSE, ANSI_CHARSET,
                    OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS, ANTIALIASED_QUALITY,
                    FF_DONTCARE | DEFAULT_PITCH, "Courier");

// sprawdza, czy utworzenie czcionki powiodło się
if (!hFont)
    return 0;

// wybiera kontekst urządzenia
SelectObject(g_HDC, hFont);

// przygotowuje czcionkę do użycia w OpenGL
wglUseFontBitmaps(g_HDC, 32, 96, base);

```

Powyższy fragment kodu tworzy listy wyświetlanego dla wytłuszczonej czcionki Courier o wielkości 14 punktów.

Wyświetlanie czcionki jest prostsze niż jej tworzenie. W tym celu należy wywołać funkcje `glListBase()` i `glCallLists()` w pokazany niżej sposób:

```

char *str;

glPushAttrib(GL_LIST_BIT);
glListBase(base - 32);
glCallLists(strlen(str), GL_UNSIGNED_BYTE, str);
glPopAttrib();

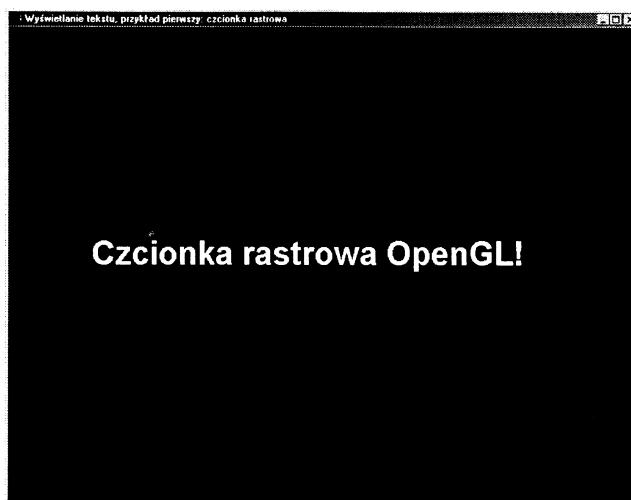
```

Funkcja `glBaseList()` definiuje najpierw bazową listę wyświetlaną, a funkcja `glCallLists()` wykonuje odpowiednie listy wyświetlania znaków tworzących przekazany jej łańcuch.

Korzystając z zaprezentowanych fragmentów kodu można stworzyć zestaw funkcji ułatwiających korzystanie z czcionek rastrowych w OpenGL. Używać ich można za pomocą prostego programu, którego efekt działania przedstawia rysunek 11.1.

**Rysunek 11.1.**

*Program  
wyświetlający tekst  
za pomocą  
czcionki rastrowej*



W celu zapamiętania identyfikatora listy bazowej trzeba zadeklarować zmienną `listBase`:

```
unsigned int listBase;
```

Funkcja `CreateBitmapFont()` utworzy czcionkę rastrową OpenGL korzystając z funkcji `CreateFont()` systemu Windows:

```
unsigned int CreateBitmapFont(char *fontName, int fontSize)
{
    HFONT hFont;           // uchwyt czcionki systemu Windows
    unsigned int base;

    base = glGenLists(96);      // listy wyświetlania dla 96 znaków

    if (strcmp(fontName, "symbol") == 0)
    {
        hFont = CreateFont(fontSize, 0, 0, 0, FW_BOLD, FALSE, FALSE, FALSE,
                            SYMBOL_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
                            ANTIALIASED_QUALITY, FF_DONTCARE | DEFAULT_PITCH,
                            fontName);
    }
    else
    {
        hFont = CreateFont(fontSize, 0, 0, 0, FW_BOLD, FALSE, FALSE, FALSE,
                            ANSI_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
                            ANTIALIASED_QUALITY, FF_DONTCARE | DEFAULT_PITCH,
                            fontName);
    }

    if (!hFont)
        return 0;

    SelectObject(g_HDC, hFont);
    wglUseFontBitmaps(g_HDC, 32, 96, base);

    return base;
}
```

Funkcja `CreateBitmapFont()` tworzy najpierw listy wyświetlania dla 96 znaków czcionki. Następnie sprawdza, czy nazwa wybranej czcionki jest łańcuch "symbol". Jeśli tak jest, wywołuje funkcję `CreateFont()` przekazując jej stałą `SYMBOL_CHARSET` jako wartość parametru `fdwCharSet`. W przeciwnym razie wybiera zestaw znaków `ANSI_CHARSET`. Po przygotowaniu czcionki rastrowej za pomocą funkcji `wglUseBitmapFonts()` funkcja `CreateBitmapFont()` zwraca identyfikator bazowej listy wyświetlania.

Kolejna funkcja, `PrintString()`, wyświetla łańcuch znaków za pomocą bieżącej czcionki rastrowej na bieżącej pozycji rastra:

```
void PrintString(unsigned int base, char *str)
{
    if ((base == 0) || (str == NULL))
        return;

    glPushAttrib(GL_LIST_BIT);
    glListBase(base - 32);
    glCallLists(strlen(str), GL_UNSIGNED_BYTE, str);
    glPopAttrib();
}
```

Przed zakończeniem działania programu należy zwolnić wszystkie listy wyświetlania. Zadanie to wykonuje funkcja `ClearFont()`:

```
void ClearFont(unsigned int base)
{
    if (base != 0)
        glDeleteLists(base, 96);
}
```

Wywołuje ona funkcję `glDeleteLists()` usuwając 96 list wyświetlania (począwszy od indeksu listy bazowej przekazanego za pośrednictwem parametru `base`).

W tym przykładzie zastosowana zostanie czcionka Arial o wielkości 48 punktów. Wyboru czcionki dokonuje się wewnątrz funkcji `Initialize()`:

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

    glShadeModel(GL_SMOOTH); // cieniowanie gładkie
    glEnable(GL_DEPTH_TEST); // usuwanie przesłoniętych powierzchni

    listBase = CreateBitmapFont("Arial", 48); // tworzy czcionkę rastrową
}
```

Ostatnia z funkcji, `Render()`, wyświetla tekst w środku okna grafiki:

```
void Render()
{
    // opróżnia bufore ekranu i głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    // odsuwa płaszczyznę napisu o jednostkę
    glTranslatef(0.0f, 0.0f, -1.0f);
```

```
// wybiera kolor biały  
glColor3f(1.0f, 1.0f, 1.0f);  
  
// wybiera pozycję rastra  
glRasterPos2f(-0.5f, 0.0f);  
  
// wyświetla tekst  
PrintString(listBase, "Czcionka rastrowa OpenGL!");  
  
glFlush();  
SwapBuffers(g_HDC); // przełącza bufory  
}
```

Funkcja Render() używa funkcji glRasterPos2f() do określenia położenia tekstu w oknie i funkcji glColor2f() do wyboru koloru tekstu. Ponieważ grafika tworzona jest w trybie perspektywy, to wykonywane przekształcenia mają wpływ na współrzędne rastra. Na skutek przesunięcia rastra o jednostkę wzduł osi z, współrzędne okna na osi x wynoszą od około -0.6 do 0.6. Tekst wyświetlany więc będzie od punktu, którego współrzędna x ma wartość 0.5, co w efekcie daje w przybliżeniu wycentrowane położenie tekstu.

Na tym można zakończyć omówienie czcionek rastrowych. Następny podrozdział poświęcony będzie czcionkom konturowym.

## Czcionki konturowe

Czcionki konturowe są podobne do omówionych czcionek rastrowych, jednak umożliwiają uzyskanie efektów trójwymiarowych. Nadając czcionkom konturowym pewną grubość uzyskać można trójwymiarowe czcionki, które można poddawać przekształceniom jak inne obiekty OpenGL.

Aby używać czcionek konturowych, trzeba najpierw zadeklarować w programie tablicę 256 zmiennych typu GLYPHMETRICSFLOAT, które przechowują informacje o położeniu i orientacji poszczególnych znaków. Typ GLYPHMETRICSFLOAT reprezentuje strukturę wykorzystywaną do tworzenia tekstów w OpenGL. Zdefiniowano ją w następujący sposób:

```
typedef struct _GLYPHMETRICSFLOAT {  
    FLOAT      gmfBlackBoxX;  
    FLOAT      gmfBlackBoxY;  
    POINTFLOAT gmfptGlyphOrigin;  
    FLOAT      gmfCellIncX;  
    FLOAT      gmfCellIncY;  
} GLYPHMETRICSFLOAT;
```

Zmienne typu GLYPHMETRICSFLOAT przekazywać należy funkcji wg1UseFontOutlines(). Funkcja ta tworzy zestaw list wyświetlania (po jednej dla każdego znaku czcionki), które wykorzystuje się następnie do rysowania tekstu w oknie grafiki. Funkcja wg1UseFontOutlines() zdefiniowana jest następująco:

```
BOOL wg1UseFontOutlinesA(  
    HDC hdc, // kontekst urządzenia dla czcionki  
    DWORD first, // pierwszy znak czcionki,  
                  // dla którego należy utworzyć listę wyświetlania
```

```

DWORD count,           // liczba znaków czcionki,
                     // dla których należy utworzyć listy wyświetlania
DWORD listBase,       // bazowa lista wyświetlania
FLOAT deviation,      // dopuszczalne zbiorowe odchylenie od obrysów
FLOAT extrusion,     // wartość w kierunku osi z
int format,           // listy wyświetlania rysują odcinki lub wielokąty
LPGLYPHMETRICSFLOAT lpgmf // adres bufora, w którym umieszczone zostaną dane znaku
);

```

Tworzenie czcionek konturowych przypomina więc tworzenie czcionek rastrowych. Można zauważyć to porównując przedstawioną wcześniej funkcję tworzenia czcionek rastrowych `CreateBitmapFont()` z funkcją tworzenia czcionek konturowych:

```

unsigned int CreateOutlineFont(char *fontName, int fontSize, float depth)
{
    HFONT hFont;           // uchwyt czcionki systemu Windows
    unsigned int base;

    base = glGenLists(256); // listy wyświetlania dla 256 znaków

    if (strcmp(fontName, "symbol") == 0)
    {
        hFont = CreateFont(fontSize, 0, 0, 0, FW_BOLD, FALSE, FALSE, FALSE,
                           SYMBOL_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
                           ANTIALIASED_QUALITY, FF_DONTCARE | DEFAULT_PITCH,
                           fontName);
    }
    else
    {
        hFont = CreateFont(fontSize, 0, 0, 0, FW_BOLD, FALSE, FALSE, FALSE,
                           ANSI_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
                           ANTIALIASED_QUALITY, FF_DONTCARE | DEFAULT_PITCH,
                           fontName);
    }

    if (!hFont)
        return 0;

    SelectObject(g_HDC, hFont);
    wglUseFontOutlines(g_HDC, 0, 255, base, 0.0f, depth, WGL_FONT_POLYGONS, gmf);

    return base;
}

```

Można jednak zauważyc kilka różnic pojawiających się w kodach obydwu funkcji. Funkcja `CreateOutlineFont()` posiada dodatkowy parametr `depth` określający głębokość konturu czcionek wzduż osi z. Kolejną różnicą jest utworzenie 256 list dla wszystkich znaków kodu ASCII. Czcionkę konturową przygotowuje się do użycia za pomocą funkcji `wglUseFontOutlines()`.

Wyświetlanie tekstu za pomocą czcionek konturowych odbywa się dokładnie w ten sam sposób jak za pomocą czcionek rastrowych:

```

glPushAttrib(GL_LIST_BIT);
glListBase(base);
glCallLists(strlen(str), GL_UNSIGNED_BYTE, str);
glPopAttrib();

```

Teraz omówionych zostanie kilka funkcji przykładowego programu, który rysować będzie obracający się tekst (jak robią to programy wygaszacz ekranu). Program ten korzysta z omówionej wcześniej funkcji `CreateOutlineFont()`, dlatego nie trzeba teraz przedstawiać jej kodu ani kodu szkieletu aplikacji systemu Windows.

Na początku należy zadeklarować następujące zmienne:

```
HDC g_HDC;           // globalny kontekst urządzenia  
float angle = 0.0f;  
unsigned int listBase; // pierwsza lista wyświetlanego  
GLYPHMETRICSFLOAT gmf[256]; // informacje o położeniu i orientacji znaków
```

Kolejna funkcja, `ClearFont()`, działa tak samo jak funkcja `ClearFont()` z poprzedniego przykładu, ale zwalnia 256 list wyświetlania:

```
void ClearFont(unsigned int base)  
{  
    glDeleteLists(base, 256);  
}
```

W „ciele” funkcji `PrintString()` umieszczony tym razem został fragment kodu, który centruje wyświetlany tekst względem podanego punktu przestrzeni. Przegląda on w pętli znaki wyświetlanego tekstu sumując ich szerokość na podstawie opisu zawartego w strukturze `GLYPHMETRICSFLOAT`, a następnie przesuwa układ współrzędnych w ujemnym kierunku osi x o wartość równą połowie sumy szerokości znaków.

```
void PrintString(unsigned int base, char *str)  
{  
    float length = 0;  
  
    if ((str == NULL))  
        return;  
  
    // centruje tekst  
    for (unsigned int loop=0;loop<(strlen(str));loop++) // określa długość tekstu  
    {  
        length+=gmf[str[loop]].gmfCellIncX; // sumuje szerokość znaków  
    }  
    glTranslatef(-length/2,0.0f,0.0f); // przesuwa układ współrzędnych  
    // o połowę uzyskanej sumy  
  
    // rysuje tekst  
    glPushAttrib(GL_LIST_BIT);  
    glListBase(base);  
    glCallLists(strlen(str), GL_UNSIGNED_BYTE, str);  
    glPopAttrib();  
}
```

Aby uzyskać właściwy obraz czcionek, należy włączyć oświetlenie sceny. Wywołując funkcję `glEnable(GL_COLOR_MATERIAL)` można przyjąć też dla uproszczenia, że kolor określa równocześnie materiał tekstu. Operacje te wykonuje funkcja `Initialize()`:

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

```

glShadeModel(GL_SMOOTH);                                // cieniowanie gładkie
glEnable(GL_DEPTH_TEST);                               // usuwa przesłonięte powierzchnie
glEnable(GL_LIGHT0);                                  // włącza źródło światła light0
glEnable(GL_LIGHTING);                                // włącza oświetlenie
glEnable(GL_COLOR_MATERIAL);                          // kolor jako materiał

listBase = CreateOutlineFont("Arial", 10, 0.25f); // czcionka Arial, 10 pkt.
}

```

Funkcja Render() ilustruje największą zaletę czcionek konturowych, którą jest możliwość dokonywania ich przekształceń w trójwymiarowej przestrzeni. Czcionki konturowe traktuje ona tak samo jak każdy inny obiekt i wykonuje przekształcenie przesunięcia i trzy obroty względem osi układu współrzędnych:

```

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

    // przesuwa układ o 10 jednostek "w głąb" ekranu
    // i wykonuje obroty względem osi układu
    glTranslatef(0.0f, 0.0f, -15.0f);
    glRotatef(angle*0.9f, 1.0f, 0.0f, 0.0f);
    glRotatef(angle*1.5f, 0.0f, 1.0f, 0.0f);
    glRotatef(angle, 0.0f, 0.0f, 1.0f);

    // wybiera kolor
    glColor3f(0.0f, 1.0f, 1.0f);

    // wyświetla tekst
    PrintString(listBase, "Czcionka konturowa OpenGL!");

    glColor3f(1.0f, 0.0f, 0.0f);
    glTranslatef(-2.0f, 0.0f, -2.0f);
    PrintString(listBase, "Drugi tekst!");

    angle += 0.1f;

    glFlush();
    SwapBuffers(g_HDC);                                // przełącza bufory
}

```

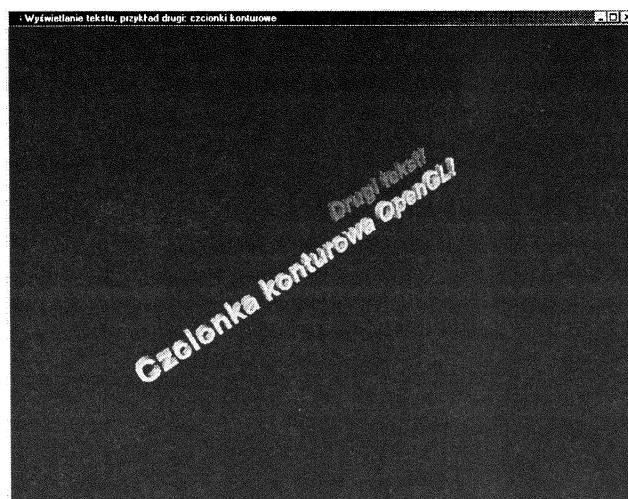
Po umieszczeniu omówionych funkcji w szkielecie aplikacji systemu Windows uzyskać można program, którego efekt działania pokazuje rysunek 11.2.

## Czcionki pokryte tekstem

Jako że czcionki konturowe składają się z wielokątów i traktowane są przez OpenGL jak zwykłe obiekty, ich wygląd wzbogacić można teksturami. Nie trzeba przy tym zajmować się wyznaczaniem współrzędnych tekstur dla wierzchołków wielokątów tworzących czcionki, ponieważ OpenGL wykonuje to zadanie automatycznie.

**Rysunek 11.2.**

*Przykład zastosowania czcionki konturowej*



Zastosowanie czcionek pokrytych teksturą zaprezentowane zostanie na przykładzie programu wyświetlającego tekst obracający się na ekranie. Dodatkowo program będzie posiadać możliwość oddalania i zbliżania tekstu za pomocą klawiszy *A* i *Z*.

A oto jego kod źródłowy:

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

////// Pliki nagłówkowe
#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 <gl/glaux.h>

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

////// Zmienne globalne
HDC g_HDC; // globalny kontekst urządzenia
bool fullScreen = false; // true = tryb pełnoekranowy;
bool keyPressed[256]; // false = tryb okienkowy
                       // tablica przyciśnięć klawiszy

float angle = 0.0f; // pierwsza lista wyświetlania
unsigned int listBase; // dane o położeniu i orientacji znaków
GLYPHMETRICSFLOAT .gmf[256]; // wykorzystywane przez listy wyświetlania

float zDepth = -10.0f; // bieżąca pozycja na osi z
```

```
////// Zmienne tekstury
texture_t *texture;

/*
Kod funkcji LoadBitmapFile() został pominięty
*/

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

    // przydziela pamięć dla struktury opisującej teksturę
    thisTexture = (texture_t*)malloc(sizeof(texture_t));
    if (thisTexture == NULL)
        return NULL;

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

    // określa rozmiary tekstury
    thisTexture->width = texInfo.biWidth;
    thisTexture->height = texInfo.biHeight;

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

    return thisTexture;
}

bool LoadAllTextures()
{
    // ładuje teksturę wody
    texture = LoadTextureFile("water.bmp");
    if (texture == NULL)
        return false;

    // konfiguruje teksturę
    glBindTexture(GL_TEXTURE_2D, texture->texID);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_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, texture->width, texture->height,
                      GL_RGB, GL_UNSIGNED_BYTE, texture->data);

    // OpenGL automatycznie generuje współrzędne tekstury dla znaków czcionki
    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
    glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);
```

```
        return true;
    }

    unsigned int CreateOutlineFont(char *fontName, int fontSize, float depth)
    {
        HFONT hFont;                      // uchwyt czcionki systemu Windows
        unsigned int base;

        base = glGenLists(256);           // listy wyświetlania dla 256 znaków czcionki

        if (strcmp(fontName, "symbol") == 0)
        {
            hFont = CreateFont(fontSize, 0, 0, 0, FW_BOLD, FALSE, FALSE, FALSE,
                                SYMBOL_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
                                ANTIALIASED_QUALITY, FF_DONTCARE | DEFAULT_PITCH,
                                fontName);
        }
        else
        {
            hFont = CreateFont(fontSize, 0, 0, 0, FW_BOLD, FALSE, FALSE, FALSE,
                                ANSI_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
                                ANTIALIASED_QUALITY, FF_DONTCARE | DEFAULT_PITCH,
                                fontName);
        }

        if (!hFont)
            return 0;

        SelectObject(g_HDC, hFont);
        wglUseFontOutlines(g_HDC, 0, 255, base, 0.0f, depth, WGL_FONT_POLYGONS, gmf);

        return base;
    }

    void ClearFont(unsigned int base)
    {
        glDeleteLists(base, 256);
    }

    void PrintString(unsigned int base, char *str)
    {
        float length = 0;

        if ((str == NULL))
            return;

        // centruje tekst
        for (unsigned int loop=0;loop<(strlen(str));loop++) // długość tekstu
        {
            length+=gmf[str[loop]].gmfCellIncX;           // sumuje szerokość znaków
        }
        glTranslatef(-length/2.0,0.0f,0.0f);             // przesuwa do środka tekstu

        // wyświetla tekst
        glPushAttrib(GL_LIST_BIT);
        glListBase(base);
        glCallLists(strlen(str), GL_UNSIGNED_BYTE, str);
        glPopAttrib();
    }
}
```

```
void CleanUp()
{
    ClearFont(listBase);
    free(texture);
}

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

    glShadeModel(GL_SMOOTH);                      // cieniowanie gładkie
    glEnable(GL_DEPTH_TEST);                      // usuwa przesłonięte powierzchnie
    glEnable(GL_LIGHT0);                          // włącza źródło światła light0
    glEnable(GL_LIGHTING);                        // włącza oświetlenie
    glEnable(GL_COLOR_MATERIAL);                  // kolor jako materiał
    glEnable(GL_TEXTURE_2D);                      // włącza odwzorowanie tekstur

    listBase = CreateOutlineFont("Arial", 14, 0.1f); // ładuje czcionkę Arial

    LoadAllTextures();

    glBindTexture(GL_TEXTURE_2D, texture->texID);
}

// Render
// opis: rysuje scenę
void Render()
{
    // opróżnia bufory ekranu i głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    // zbliża lub oddala napis
    // i obraca go względem osi układu
    glTranslatef(0.0f, 0.0f, zDepth);
    glRotatef(angle*0.9f, 1.0f, 0.0f, 0.0f);
    glRotatef(angle*1.5f, 0.0f, 1.0f, 0.0f);
    glRotatef(angle, 0.0f, 0.0f, 1.0f);

    // wyświetla napis
    PrintString(listBase, "TEKSTURA NA CZCIONCE!");

    angle += 0.1f;

    glFlush();
    SwapBuffers(g_HDC);                         // przełącza bufory
}

// poniższy kod stanowi fragment procedury okienkowej WndProc()

switch(message)
{
    ...
    case WM_SIZE:
        height = HIWORD(lParam);            // określa rozmiary okna
        width = LOWORD(lParam);
}
```

```
if (height==0)                                // zapobiega dzieleniu przez 0
{
    height=1;
}

glViewport(0, 0, width, height);   // nadaje oknu grafiki nowe rozmiary
glMatrixMode(GL_PROJECTION);        // wybiera macierz rzutowania
glLoadIdentity();                  // i resetuje ją

// wyznacza proporcje okna
gluPerspective(54.0f,(GLfloat)width/(GLfloat)height,1.0f,1000.0f);

glMatrixMode(GL_MODELVIEW);        // wybiera macierz modelowania
glLoadIdentity();                  // i resetuje ją

return 0;
...
}

...

// poniższy kod stanowi fragment funkcji WinMain()
done = false;
Initialize();

while (!done)
{
    PeekMessage(&msg, hwnd, NULL, NULL, PM_REMOVE);

    if (msg.message == WM_QUIT)          // komunikat WM_QUIT?
    {
        done = true;                  // jeśli tak, to aplikacja kończy działanie
    }
    else
    {
        if (keyPressed[VK_ESCAPE])
            done = true;
        else
        {
            if (keyPressed['A'])
                zDepth += 0.1f;
            if (keyPressed['Z'])
                zDepth -= 0.1f;

        Render();

        TranslateMessage(&msg);        // tłumaczy komunikat i wysyła do systemu
        DispatchMessage(&msg);
    }
}
}

CleanUp();
```

Jak łatwo zauważyc, zasadnicza różnica pomiędzy programem stosującym zwykłe czcionki konturowe i programem rysującym czcionki konturowe pokryte teksturą polega na tym,

że w drugim z tych przypadków trzeba poinformować maszynę OpenGL o konieczności automatycznego wyznaczania współrzędnych tekstury. Służy temu poniższy fragment kodu:

```
// OpenGL automatycznie generuje współrzędne tekstury dla znaków czcionki
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
 glEnable(GL_TEXTURE_GEN_S);
 glEnable(GL_TEXTURE_GEN_T);
```

Należy zwrócić uwagę na zastosowanie funkcji `glTexGeni()`. Po raz pierwszy zastosowana została w przykładach omawiających odwzorowanie otoczenia. Teraz trzeba przyznać się jej dokładniej. Pierwszy parametr funkcji `glTexGeni()` informuje maszynę OpenGL o tym, która ze współrzędnych tekstury będzie generowana automatycznie. Parametr ten może przyjmować wartości `GL_S` lub `GL_T` w przypadku tekstur dwuwymiarowych.

Drugi parametr funkcji `glTexGeni()` określa tryb odwzorowania tekstury używany podczas automatycznej generacji współrzędnej. Może on przyjmować wartości przedstawione w tabeli 11.1.

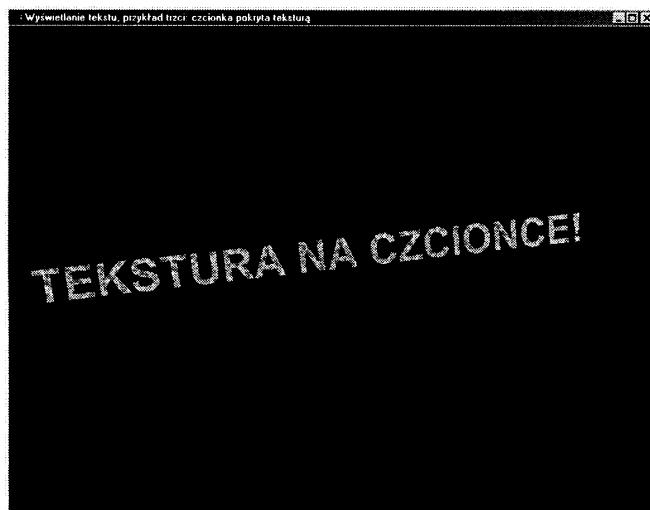
**Tabela 11.1.** Tryb odwzorowania tekstury

Wartość	Opis
<code>GL_EYE_LINEAR</code>	Tekstura umieszczana jest w stałym miejscu ekranu. Obiekty, których obraz pojawia się w tym miejscu, są pokrywane tekstrą.
<code>GL_OBJECT_LINEAR</code>	Tekstura umieszczana jest na obiekcie.
<code>GL_SPHERE_MAP</code>	Tekstura reprezentuje odwzorowanie otoczenia (mapę sferyczną).

Tekst napisu można wyświetlić po związaniu tekstu z obiektem tekstu. W przykładzie tym do pokrycia czcionki użыта została tekstura imitująca powierzchnię wody, a uzyskany rezultat przedstawia rysunek 11.3.

**Rysunek 11.3.**

Przykład  
czcionki konturowej  
pokryte tekstrą



## Podsumowanie

Czcionki rastrowe stanowią najprostszy sposób wyświetlania tekstu w OpenGL. Czcionki rastrowe przygotowuje się do użycia za pomocą funkcji `wglUseFontBitmaps()`, która tworzy je na podstawie czcionek dostępnych w systemie Windows.

W podobny sposób — przez wywołanie funkcji `wglUseOutlineFonts()` — powstają czcionki konturowe. Odróżnia je jednak możliwość określenia ich grubości i przetwarzania w taki sposób, w jaki przetwarza się każdy obiekt trójwymiarowy. Dzięki temu można je obracać, skalować i przesuwać w trójwymiarowej przestrzeni. Wywołanie funkcji `wglUseOutlineFonts()` tworzy reprezentację czcionek konturowych, która wykorzystuje wielokąty OpenGL.

Ponieważ czcionki konturowe reprezentowane są za pomocą wielokątów i traktowane są przez OpenGL jak zwykłe obiekty trójwymiarowe, można wzbogacić ich wygląd pokrywając je teksturami. Współrzędne tekstury dla czcionek są automatycznie generowane przez maszynę OpenGL.



# Rozdział 12.

# Bufory OpenGL

Bufory OpenGL wykorzystywane były w każdym z przedstawionych programów, ale jak dotąd nie zostały omówione szczegółowo. Praktycznie każdy program używał buforów koloru i głębi w celu podwójnego buforowania grafiki i usuwania przesłoniętych powierzchni. W bieżącym rozdziale przedstawione zostaną także inne zastosowania buforów, a ponadto omówione zostaną bufory powielania i akumulacji.

W rozdziale tym przedstawione będą:

- ◆ bufory OpenGL;
- ◆ zastosowania bufora kolorów;
- ◆ zastosowania bufora głębi;
- ◆ zastosowania bufora powielania;
- ◆ zastosowania bufora akumulacji.

## Bufory OpenGL — wprowadzenie

OpenGL udostępnia kilka różnych typów buforów. Wszystkie bufory używane są w procesie tworzenia obrazu na ekranie monitora. Obraz powstający na ekranie reprezentowany jest za pomocą dwuwymiarowej tablicy pikseli, czyli *bufora*, w którym przechowywany jest opis kolorów każdego piksela. Bufor jest więc zbiorem danych opisujących piksele. Na przykład bufor kolorów OpenGL przechowuje dane o kolorach pikseli zapisane zgodnie z modelem RGBA lub modelem indeksowym.

Zawartość *bufor obrazu* powstaje na podstawie zawartości wszystkich buforów dostępnych w systemie. W OpenGL bufor obrazu jest więc kombinacją bufora koloru, bufora głębi, bufora powielania i bufora akumulacji. Przy zmianie zawartości któregokolwiek z wymienionych buforów zmienia się w efekcie zawartość bufora obrazu.

### Konfiguracja formatu pikseli

Bufory OpenGL stosowane były od początku poznawania OpenGL, a mimo to nie doczekały się specjalnego omówienia. Większość przykładowych programów używała funkcji `SetupPixelFormat()` w celu skonfigurowania kontekstu urządzenia dla okna grafiki

OpenGL. Kontekst ten określał stosowany format pikseli i tym samym format przechowującego je bufora.. Wystarczy przypomnieć kod funkcji SetupPixelFormat():

```
void SetupPixelFormat(HDC hDC)
{
    int nPixelFormat;           // indeks formatu pikseli

    static PIXELFORMATDESCRIPTOR pfd = {
        sizeof(PIXELFORMATDESCRIPTOR), // rozmiar struktury
        1,                          // domyślna wersja
        PFD_DRAW_TO_WINDOW |       // grafika w oknie
        PFD_SUPPORT_OPENGL |      // grafika OpenGL
        PFD_DOUBLEBUFFER,         // podwójne buforowanie
        PFD_TYPE_RGBA,            // tryb kolorów RGBA
        32,                        // 32-bitowy opis kolorów
        0, 0, 0, 0, 0, 0,          // nie specyfikuje bitów kolorów
        0,                         // bez bufora alfa
        0,                         // nie specyfikuje bitu przesunięcia
        0,                         // bez bufora akumulacji
        0, 0, 0, 0,                // ignoruje bity akumulacji
        16,                        // 16-bitowy bufor z
        0,                         // bez bufora powielania
        0,                         // bez buforów pomocniczych
        PFD_MAIN_PLANE,           // główna płaszczyzna rysowania
        0,                         // zarezerwowane
        0, 0, 0 };                 // ignoruje maski warstw

    // wybiera najbardziej zgodny format pikseli
    nPixelFormat = ChoosePixelFormat(hDC, &pfd);

    // określa format pikseli dla danego kontekstu urządzenia
    SetPixelFormat(hDC, nPixelFormat, &pfd);
}
```

W celu określenia formatu pikseli można posłużyć się strukturą PIXELFORMATDESCRIPTOR. Oto jej definicja:

```
typedef struct tagPIXELFORMATDESCRIPTOR
{
    WORD nSize;                  // rozmiar struktury
    WORD nVersion;               // wersja struktury
    DWORD dwFlags;               // właściwości bufora pikseli
    BYTE iPixelType;              // typ danych pikseli
    BYTE cColorBits;              // rozmiar bufora koloru
    BYTE cRedBits;                // liczba bitów składowej czerwonej
    BYTE cRedShift;               // przesunięcie bitów składowej czerwonej
    BYTE cGreenBits;              // liczba bitów składowej zielonej
    BYTE cGreenShift;             // przesunięcie bitów składowej
    BYTE cBlueBits;                // liczba bitów składowej niebieskiej
    BYTE cBlueShift;               // przesunięcie bitów składowej
    BYTE cAlphaBits;               // liczba bitów współczynnika alfa
    BYTE cAlphaShift;              // przesunięcie bitów współczynnika alfa
    BYTE cAccumBits;               // liczba bitów bufora akumulacji
    BYTE cAccumRedBits;             // liczba bitów składowej czerwonej w b. akumulacji
    BYTE cAccumGreenBits;           // liczba bitów składowej zielonej w b. akumulacji
    BYTE cAccumBlueBits;             // liczba bitów składowej niebieskiej w b. akumulacji
    BYTE cAccumAlphaBits;           // liczba bitów współczynnika alfa w b. akumulacji
    BYTE cDepthBits;                // liczba bitów bufora głębi
```

```

BYTE cStencilBits;           // liczba bitów bufora powielania
BYTE cAuxBuffers;          // liczba buforów pomocniczych (MS OpenGL nie używa)
BYTE iLayerType;           // pole nie jest już używane, a wartość ignorowana
BYTE bReserved;             // liczba warstw grafiki
DWORD dwLayerMask;         // pole nie jest już używane, a wartość ignorowana
DWORD dwVisibleMask;       // kolor przezroczysty lub indeks koloru warstwy pod spodem
DWORD dwDamageMask;        // pole nie jest już używane, a wartość ignorowana
} PIXELFORMATDESCRIPTOR;

```

Pola tej struktury inicjuje się wewnątrz funkcji `SetupPixelFormat()`. Omówione teraz zostanie znaczenie najważniejszych pól struktury. Pole `dwFlags` określa właściwości bufora pikseli. Tabela 12.1 prezentuje znaczniki używane do określenia wartości tego pola.

**Tabela 12.1.** Znaczniki pola `dwFlags`

Znacznik	Opis
<code>PFD_DRAW_TO_WINDOW</code>	Bufor może być wykorzystywany do tworzenia grafiki w oknie
<code>PFD_DRAW_TO_BITMAP</code>	Bufor może być wykorzystywany do tworzenia mapy bitowej w pamięci
<code>PFD_SUPPORT_GDI</code>	Bufor GDI
<code>PFD_SUPPORT_OPENGL</code>	Bufor OpenGL
<code>PFD_DOUBLEBUFFER</code>	Podwójne buforowanie
<code>PFD_STEREO</code>	Bufor stereoskopowy
<code>PFD_DEPTH_DONTCARE</code>	Żadany format pikseli może — ale nie musi — posiadać bufora głębi; znacznik wykorzystywany jedynie podczas wywoływania funkcji <code>ChoosePixelFormat()</code>
<code>PFD_DOUBLEBUFFER_DONTCARE</code>	Żadany format pikseli może — ale nie musi — być podwójnie buforowany; znacznik wykorzystywany jedynie podczas wywoływania funkcji <code>ChoosePixelFormat()</code>
<code>PFD_STEREO_DONTCARE</code>	Żadny format pikseli może — ale nie musi — dysponować buforem stereoskopowym; znacznik wykorzystywany jedynie podczas wywoływania funkcji <code>ChoosePixelFormat()</code>

Pole `iPixelType` wykorzystywane jest do określenia typu danych opisujących piksele. Zwykle posiada wartość `PFD_TYPE_RGBA`. Dopuszczalne wartości pola prezentuje tabela 12.2.

**Tabela 12.2.** Wartości pola `iPixelType`

Wartość	Opis
<code>PFD_TYPE_RGBA</code>	Każdy piksel opisany jest za pomocą czterech składowych występujących w następującym porządku: składowa czerwona, składowa zielona, składowa niebieska i współczynnik alfa
<code>PFD_TYPE_COLORINDEX</code>	Każdy piksel opisany jest za pomocą indeksu koloru

Pola `cColorBits`, `cDepthBits`, `cAccumBits` i `cStencilBits` definiują rozmiary buforów koloru, głębi, akumulacji i powielania. Funkcja `SetupPixelFormat()` nadaje polom `cAccumBits` i `cStencilBits` wartości 0. Nadanie zerowego rozmiaru dowolnemu buforowi jest równoznaczne z jego wyłączeniem.

Pole `iLayerType` może przyjmować wartości `PFD_MAIN_PLANE`, `PFD_OVERLAY_PLANE` lub `PFD_UNDERLAY_PLANE`. Najczęściej nadaje się mu wartość `PFD_MAIN_PLANE` określającą główną warstwę rysowania grafiki.

Po nadaniu wartości wszystkim polom struktury `PIXELFORMATDESCRIPTOR` należy przekazać ją jako parametr funkcji `ChoosePixelFormat()`, która próbuje dopasować najbardziej zgodny z formatów pikseli obsługiwanych przez kontekst urządzenia do opisu formatu zawartego w strukturze `PIXELFORMATDESCRIPTOR`. Funkcja `ChoosePixelFormat()` zwraca indeks formatu pikseli, który następnie należy przekazać funkcji `SetPixelFormat()` wybierającej format pikseli dla danego kontekstu urządzenia. Ilustruje to poniższy fragment kodu:

```
...  
// wypełnienie struktury PIXELFORMATDESCRIPTOR  
  
// wybiera najbardziej zgodny format pikseli  
nPixelFormat = ChoosePixelFormat(hDC, &pfds);  
  
// określa format pikseli dla danego kontekstu urządzenia  
SetPixelFormat(hDC, nPixelFormat, &pfds);  
}
```

I tak opanowana została umiejętność konfigurowania buforów OpenGL, więc należy przyjrzeć się sposobom ich wykorzystania.

## Opróżnianie buforów

Zanim przedstawione zostanie szczegółowo zastosowanie każdego z buforów omówić trzeba najpierw sposoby ich opróżniania. OpenGL udostępnia dla każdego typu bufora osobną funkcję pozwalającą zdefiniować wartość, którą wypełniony zostanie bufor podczas opróżniania. Opróżniając bufor za pomocą funkcji `glClear()` powoduje się jego wypełnienie zdefiniowaną wcześniej wartością. Funkcje te posiadają następujące prototypy:

```
void ClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);  
void glClearDepth(GLclampd depth);  
void glClearStencil(GLint s);  
void glClearAccum(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);
```

Wymienione funkcje pozwalają określić wartość używaną podczas opróżniania buforów koloru, głębi, powielania i akumulacji. Zmienne typów `GLclampf` i `GLclampd` muszą posiadać wartość z przedziału od 0.0 do 1.0. Domyślnie bufore są wypełniane są wartością 0.0 (z wyjątkiem bufora głębi, który wypełniany jest wartością 1.0).

W celu opróżnienia bufora należy wywołać funkcję `glClear()` zdefiniowaną w następujący sposób:

```
void glClear(GLbitfield mask);
```

Parametr `mask` stanowić może kombinację znaczników wymienionych w tabeli 12.3.

**Tabela 12.3.** Znaczniki używane przez funkcję `glClear()`

Znacznik	Opis
<code>GL_COLOR_BUFFER_BIT</code>	Bufor koloru w modelu RGBA
<code>GL_DEPTH_BUFFER_BIT</code>	Bufor głębi
<code>GL_STENCIL_BUFFER_BIT</code>	Bufor powielania
<code>GL_ACCUM_BUFFER_BIT</code>	Bufor akumulacji

## Bufor koloru

Bufor koloru przechowuje opis koloru pikseli w modelu RGBA lub indeksowym. Najczęściej na ekranie rysuje się właśnie zawartość tego bufora. Umożliwia on dodatkowo zastosowanie mechanizmu podwójnego buforowania oraz uzyskanie obrazu stereoskopowego.

### Podwójne buforowanie

*Podwójne buforowanie* wykorzystywane jest podczas animacji obiektów. Podczas gdy zawartość jednego bufora wyświetlana jest na ekranie, kolejna klatka animacji tworzona jest w drugim buforze. Po narysowaniu kompletnej sceny kolejnej klatki bufore są przełączane, co pozwala zachować płynność animowanej grafiki. Tworzenie grafiki w buforze, którego zawartość nie jest wyświetlana, pozwala uniknąć migotania, które pojawia się podczas tworzenia grafiki w buforze, którego zawartość jest prezentowana na ekranie. Podwójne buforowanie dostępne jest tylko dla bufora kolorów. Mechanizmu tego nie można więc wykorzystywać w przypadku buforów głębi, akumulacji i powielania.

Jeśli wybrany format pikseli umożliwia podwójne buforowanie, to OpenGL domyślnie tworzy grafikę w buforze, którego zawartość nie jest wyświetlana. Za pomocą funkcji `glDrawBuffer()` można określić bufor, w którym rysowana jest grafika. Funkcja ta zdefiniowana jest następująco:

```
void glDrawBuffer(GLenum mode);
```

Parametr `mode` może przyjmować wartości przedstawione w tabeli 12.4.

**Tabela 12.4.** Tryby rysowania grafiki określone za pomocą funkcji `glDrawBuffer()` dla podwójnego buforowania

Tryb	Opis
<code>GL_FRONT</code>	Grafika tworzona jest w buforze koloru wyświetlonym na ekranie (domyślny sposób działania, gdy mechanizm podwójnego buforowania nie jest dostępny)
<code>GL_BACK</code>	Grafika tworzona jest w buforze koloru, którego zawartość nie jest wyświetlana na ekranie (domyślny sposób działania, gdy mechanizm podwójnego buforowania jest dostępny)
<code>GL_FRONT_AND_BACK</code>	Grafika tworzona jest w obu buforach koloru
<code>GL_NONE</code>	Grafika nie jest rysowana w żadnym z buforów koloru

W przypadku stosowania mechanizmu podwójnego buforowania domyślną wartością parametru jest `GL_BACK`, a w przeciwnym przypadku `GL_FRONT`.

Za pomocą funkcji `glReadBuffer()` można określić, z którego z buforów koloru pobierany jest opis pikseli podczas wykonywania funkcji `glReadPixels()`, `glCopyPixels()`, `glCopyTexImage*`() i `glCopyTexSubImage*`() . Funkcja ta posiada następujący prototyp:

```
void glReadBuffer(GLenum mode);
```

Parametr `mode` może przyjmować te same wartości co w przypadku funkcji `glDrawBuffer()`.

## Buforowanie stereoskopowe

*Buforowanie stereoskopowe* wykorzystuje osobne bufore dla obrazu widzianego lewym i prawym okiem, co umożliwia uzyskanie pełnego wrażenia trójwymiarowości prezentowanych obiektów. Podobnie jak w przypadku podwójnego buforowania bufor, w którym tworzona jest grafika, określa się za pomocą funkcji `glDrawBuffer()`. Jej parametr może przyjmować wartości przedstawione w tabeli 12.5. Wartości tych nie należy łączyć z wartościami przedstawionymi wcześniej w tabeli 12.4.

**Tabela 12.5.** Tryby rysowania grafiki określone za pomocą funkcji `glDrawBuffer()` dla buforowania stereoskopowego

Tryb	Opis
<code>GL_LEFT</code>	Grafika tworzona jest w obu buforach (wyświetlanym i niewyświetlanym) dla lewego oka
<code>GL_RIGHT</code>	Grafika tworzona jest w obu buforach (wyświetlanym i niewyświetlanym) dla prawego oka
<code>GL_FRONT_LEFT</code>	Grafika tworzona jest w wyświetlanym buforze dla lewego oka
<code>GL_FRONT_RIGHT</code>	Grafika tworzona jest w wyświetlanym buforze dla prawego oka
<code>GL_BACK_LEFT</code>	Grafika tworzona jest dla lewego oka w buforze, którego zawartość nie jest wyświetlana
<code>GL_BACK_RIGHT</code>	Grafika tworzona jest dla prawego oka w buforze, którego zawartość nie jest wyświetlana

## Bufor głębi

Typowym zastosowaniem *bufora głębi* jest zapobieganie rysowaniu powierzchni obiektów, która nie jest w danej scenie widoczna. Bufor głębi przechowuje dla każdego piksela odległość pomiędzy obserwatorem i obiektem, do reprezentacji którego należy dany piksel. Podczas rysowania obiektu, który przesłania obserwatorowi inny obiekt, modyfikowana jest zawartość bufora głębi w zależności od wybranej funkcji porównania głębi.

## Funkcje porównania głębi

Podczas rysowania grafiki OpenGL współrzędna z punktu, którego reprezentacją będzie dany piksel, porównywana jest zawsze z wartością współrzędnej z zapisaną dla danego piksela w buforze głębi. Sposób porównania określany jest za pomocą funkcji `glDepthFunc()` o następującym prototypie:

```
void glDepthFunc(GLenum func);
```

Parametr `func` może przyjmować wartości przedstawione w tabeli 12.6.

**Tabela 12.6.** Funkcje porównania głębi

Funkcja	Opis
<code>GL_NEVER</code>	Rezultat porównania jest zawsze negatywny
<code>GL_LESS</code>	Rezultat porównania jest pozytywny, jeśli nowa wartość współrzędnej z jest mniejsza od wartości współrzędnej z zapisanej w buforze (domyślana funkcja porównania głębi)
<code>GL_EQUAL</code>	Rezultat porównania jest pozytywny, jeśli nowa wartość współrzędnej z jest równa wartości współrzędnej z zapisanej w buforze
<code>GL_LEQUAL</code>	Rezultat porównania jest pozytywny, jeśli nowa wartość współrzędnej z jest mniejsza lub równa wartości współrzędnej z zapisanej w buforze
<code>GL_GREATER</code>	Rezultat porównania jest pozytywny, jeśli nowa wartość współrzędnej z jest większa od wartości współrzędnej z zapisanej w buforze
<code>GL_NOTEQUAL</code>	Rezultat porównania jest pozytywny, jeśli nowa wartość współrzędnej z jest różna od wartości współrzędnej z zapisanej w buforze
<code>GL_GEQUAL</code>	Rezultat porównania jest pozytywny, jeśli nowa wartość współrzędnej z jest większa lub równa wartości współrzędnej z zapisanej w buforze
<code>GL_ALWAYS</code>	Rezultat porównania jest zawsze pozytywny

Jeśli wynik porównania za pomocą wybranej funkcji jest pozytywny, to OpenGL umieszcza piksel w buforze koloru. Domyślną funkcją porównania głębi jest `GL_LESS`, która powoduje narysowanie piksela, jeśli współrzędna z punktu, który on reprezentuje na ekranie, jest mniejsza od wartości współrzędnej z zapisanej w buforze głębi.

## Zastosowania bufora głębi

Najczęstszym zastosowaniem bufora głębi jest zapobieganie rysowaniu niewidocznych powierzchni. Bufor głębi można także wykorzystać w celu uzyskania przekroju po to, by zaprezentować wewnętrzną budowę złożonego obiektu (na przykład komputera czy silnika).

Prezentację rozmaitych zastosowań bufora głębi należy rozpocząć od prostego programu, który umożliwiać będzie użytkownikowi zmianę funkcji porównania głębi z `GL_LESS` na `GL_ALWAYS` i z powrotem poprzez naciśnięcie klawisza spacji. Program rozpocznie działanie korzystając z funkcji `GL_LESS`, dzięki czemu tworzona przez niego scena będzie posiadać normalny wygląd, gdyż przesłonięte powierzchnie nie będą rysowane. Po naciśnięciu klawisza spacji funkcja porównania głębi zostanie zmieniona na `GL_ALWAYS`, a obiekty sceny widoczne będą w porządku rysowania (a nie przesłaniania). W tym przypadku —

jeśli na przykład sześciian przesłania kulę — aby uzyskać prawidłowy wygląd sceny, należy najpierw narysować kulę, a dopiero potem sześciian. Jeśli najpierw zostanie narysowany sześciian, to kula zostanie narysowana tak, jakby znajdowała się przed sześciadem, co oczywiście nie jest prawdą. W przypadku funkcji GL\_LESS nie trzeba zwracać uwagi na kolejność rysowania obiektów, ponieważ OpenGL automatycznie zapobiega rysowaniu przesłoniętych powierzchni korzystając z bufora głębi. Poniżej zaprezentowany został kod źródłowy omawianego programu.

```
////// Pliki nagłówkowe
#include <windows.h> // standardowy plik nagłówkowy Windows
#include <math.h>
#include <gl/gl.h> // standardowy plik nagłówkowy OpenGL
#include <gl/glu.h> // plik nagłówkowy biblioteki GLU
#include <gl/glaux.h> // funkcje pomocnicze OpenGL

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

bool depthLess = true; // true: funkcja GL_LESS
                      // false: funkcja GL_ALWAYS

// Initialize
// opis: inicjuje OpenGL
void Initialize()
{
    glEnable(GL_DEPTH_TEST); // włącza bufor głębi
    glEnable(GL_CULL_FACE); // wyłącza rysowanie tylnych stron wielokątów
    glEnable(GL_LIGHTING); // włącza oświetlenie
    glEnable(GL_LIGHT0); // włącza źródło światła light0
    glEnable(GL_COLOR_MATERIAL); // kolor jako materiał
    glShadeModel(GL_SMOOTH); // cieniowanie gładkie
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // tło w kolorze czarnym
}

// Render
// opis: rysuje scenę
void Render()
{
    // opróżnia bufore ekranu i głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    // wybiera funkcję porównania głębi
    if (depthLess)
        glDepthFunc(GL_LESS);
    else
        glDepthFunc(GL_ALWAYS);

    angle += 0.3; // zwiększa kąt obrotu

    // odsuwa scenę o 100 jednostek
    glTranslatef(0.0f, 0.0f, -100.0f);
```

```
// najpierw rysuje nieprzezroczysty sześcian
glPushMatrix();
glRotatef(angle, 1.0f, 0.0f, 0.0f);
glRotatef(angle*0.5f, 0.0f, 1.0f, 0.0f);
glRotatef(angle*1.2f, 0.0f, 0.0f, 1.0f);
glColor3f(0.2f, 0.4f, 0.6f);
auxSolidCube(30.0f);
glPopMatrix();

// a następnie schowaną za nim nieprzezroczystą kulę
glPushMatrix();
glTranslatef(0.0f, 0.0f, -50.0f);
glRotatef(angle, 0.0f, 1.0f, 0.0f);
glColor3f(1.0f, 1.0f, 1.0f);
auxSolidSphere(30.0f);
glPopMatrix();

glFlush();
SwapBuffers(g_HDC); // przełącza bufory
}

// funkcja określająca format pikseli
void SetupPixelFormat(HDC hDC)
{
    int nPixelFormat; // indeks formatu pikseli

    static PIXELFORMATDESCRIPTOR pfd = {
        sizeof(PIXELFORMATDESCRIPTOR), // rozmiar struktury
        1, // domyślna wersja
        PFD_DRAW_TO_WINDOW | // grafika w oknie
        PFD_SUPPORT_OPENGL | // grafika OpenGL
        PFD_DOUBLEBUFFER, // podwójne buforowanie
        PFD_TYPE_RGBA, // tryb kolorów RGBA
        32, // 32-bitowy opis kolorów
        0, 0, 0, 0, 0, 0, // nie specyfikuje bitów kolorów
        0, // bez bufora alfa
        0, // nie specyfikuje bitu przesunięcia
        0, // bez bufora akumulacji
        0, 0, 0, 0, // ignoruje bity akumulacji
        16, // 16-bitowy bufor z
        0, // bez bufora powielania
        0, // bez buforów pomocniczych
        PFD_MAIN_PLANE, // główna płaszczyzna rysowania
        0, // zarezerwowane
        0, 0, 0 }; // ignoruje maski warstw

    // wybiera najbardziej zgodny format pikseli
    nPixelFormat = ChoosePixelFormat(hDC, &pfd);

    // określa format pikseli dla danego kontekstu urządzenia
    SetPixelFormat(hDC, nPixelFormat, &pfd);
}

// procedura okienkowa
HRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HGLRC hRC;// kontekst tworzenia grafiki
    static HDC hDC;// kontekst urządzenia
    int width, height;// szerokość i wysokość okna
```

```
switch(message)
{
    case WM_CREATE:           // okno jest tworzone
        hDC = GetDC(hwnd);      // pobiera kontekst urządzenia dla okna
        g_HDC = hDC;
        SetupPixelFormat(hDC);   // wywołuje funkcję określającą format pikseli

        // tworzy kontekst tworzenia grafiki i czyni go bieżącym
        hRC = wglCreateContext(hDC);
        wglMakeCurrent(hDC, hRC);

        return 0;
        break;

    case WM_CLOSE:            // okno jest zamykane
        // dezaktywuje bieżący kontekst tworzenia grafiki i usuwa go
        wglMakeCurrent(hDC, NULL);
        wglDeleteContext(hRC);

        // wstawia komunikat WM_QUIT do kolejki
        PostQuitMessage(0);

        return 0;
        break;

    case WM_SIZE:
        height = HIWORD(lParam);    // pobiera nowe rozmiary okna
        width = LOWORD(lParam);

        if (height==0)             // unika dzielenia przez 0
        {
            height=1;
        }

        glViewport(0, 0, width, height); // nadaje nowe wymiary oknu OpenGL
        glMatrixMode(GL_PROJECTION);   // wybiera macierz rzutowania
        glLoadIdentity();             // resetuje macierz rzutowania

        // wyznacza proporcje obrazu
        gluPerspective(54.0f,(GLfloat)width/(GLfloat)height,1.0f,1000.0f);

        glMatrixMode(GL_MODELVIEW);   // wybiera macierz modelowania
        glLoadIdentity();             // resetuje macierz modelowania

        return 0;
        break;

    case WM_KEYDOWN:           // został naciśnięty klawisz?
        keyPressed[wParam] = true;
        return 0;
        break;

    case WM_KEYUP:
        keyPressed[wParam] = false;
        return 0;
        break;
}
```

```
    default:
        break;
    }

    return (DefWindowProc(hwnd, message, wParam, lParam));
}

// punkt, w którym rozpoczyna się wykonywanie aplikacji
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
                    int nShowCmd)
{
    WNDCLASSEX windowClass;           // klasa okna
    HWND hwnd;                      // uchwyty okna
    MSG msg;                        // komunikaty
    bool done;                      // znacznik zakończenia aplikacji
    DWORD dwExStyle;                // rozszerzony styl okna
    DWORD dwStyle;                  // styl okna
    RECT windowRect;

    // zmienne pomocnicze
    int width = 800;
    int height = 600;
    int bits = 32;

    fullScreen = FALSE

    windowRect.left=(long)0;          // struktura określająca rozmiary okna
    windowRect.right=(long)width;
    windowRect.top=(long)0;
    windowRect.bottom=(long)height;

    // definicja klasy okna
    windowClass.cbSize= sizeof(WNDCLASSEX);
    windowClass.style= CS_HREDRAW | CS_VREDRAW;
    windowClass.lpfnWndProc= WndProc;
    windowClass.cbClsExtra= 0;
    windowClass.cbWndExtra= 0;
    windowClass.hInstance= hInstance;
    windowClass.hIcon= LoadIcon(NULL, IDI_APPLICATION); // domyślna ikona
    windowClass.hCursor= LoadCursor(NULL, IDC_ARROW);   // domyślny kurSOR
    windowClass.hbrBackground= NULL;                     // bez tła
    windowClass.lpszMenuName= NULL;                     // bez menu
    windowClass.lpszClassName= "MojaKlasa";
    windowClass.hIconSm= LoadIcon(NULL, IDI_WINLOGO);  // logo Windows

    // rejestruje klasę okna
    if (!RegisterClassEx(&windowClass))
        return 0;

    if (fullScreen)                  // tryb pełnoekranowy?
    {
        DEVMODE dmScreenSettings;    // tryb urządzenia
        memset(&dmScreenSettings, 0, sizeof(dmScreenSettings));
        dmScreenSettings.dmSize = sizeof(dmScreenSettings);
        dmScreenSettings.dmPelsWidth = width;      // szerokość ekranu
        dmScreenSettings.dmPelsHeight = height;     // wysokość ekranu
        dmScreenSettings.dmBitsPerPel = bits;       // liczba bitów na piksel
        dmScreenSettings.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;
    }
}
```

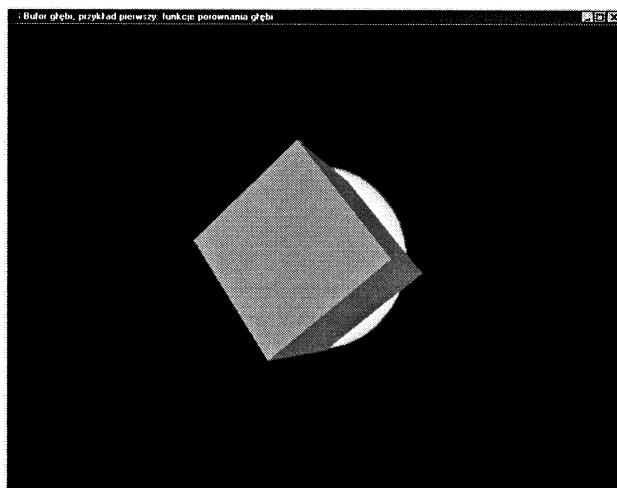
```
if (ChangeDisplaySettings(&dmScreenSettings, CDS_FULLSCREEN) !=  
    DISP_CHANGE_SUCCESSFUL)  
{  
    // przełączenie trybu nie powiodło się, z powrotem tryb okienkowy  
    MessageBox(NULL, "Przełączenie nie powiodło się.", NULL, MB_OK);  
    fullScreen=FALSE;  
}  
}  
  
if (fullScreen)           // tryb pełnoekranowy?  
{  
    dwExStyle=WS_EX_APPWINDOW;      // rozszerzony styl okna  
    dwStyle=WS_POPUP;               // styl okna  
    ShowCursor(FALSE);             // ukrywa kursor myszy  
}  
else  
{  
    dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE; // definicja klasy okna  
    dwStyle=WS_OVERLAPPEDWINDOW;        // styl okna  
}  
  
AdjustWindowRectEx(&windowRect, dwStyle, FALSE, dwExStyle); // koryguje rozmiar okna  
  
// tworzy okno  
hwnd = CreateWindowEx(  
    NULL,                                // rozszerzony styl okna  
    "MojaKlasa",                          // nazwa klasy  
    "Bufor głębi, przykład pierwszy: funkcje porównania głębi", // nazwa aplikacji  
    dwStyle | WS_CLIPCHILDREN |  
    WS_CLIPSIBLINGS,  
    0, 0,                                // współrzędne x,y  
    windowRect.right - windowRect.left,  
    windowRect.bottom - windowRect.top,    // szerokość, wysokość  
    NULL,                                 // uchwyty okna nadzawanego  
    NULL,                                 // uchwyty menu  
    hInstance,                            // instancja aplikacji  
    NULL);                               // bez dodatkowych parametrów  
  
// sprawdza, czy utworzenie okna nie powiodło się (wtedy wartość hwnd równa NULL)  
if (!hwnd)  
    return 0;  
  
ShowWindow(hwnd, SW_SHOW);          // wyświetla okno  
UpdateWindow(hwnd);                // aktualizuje okno  
  
done = false;                      // inicjuje zmienną warunku pętli  
Initialize();                     // inicjuje OpenGL  
  
// pętla przetwarzania komunikatów  
while (!done)  
{  
    PeekMessage(&msg, hwnd, NULL, NULL, PM_REMOVE);  
  
    if (msg.message == WM_QUIT)        // aplikacja otrzymała komunikat WM_QUIT?  
    {  
        done = true;                 // jeśli tak, to kończy działanie  
    }  
    else
```

```
{  
    if (keyPressed[VK_ESCAPE])  
        done = true;  
    else  
    {  
        if (keyPressed[VK_SPACE])  
            depthLess = !depthLess;  
  
        Render();  
  
        TranslateMessage(&msg); // tłumaczy komunikat i wysyła do systemu  
        DispatchMessage(&msg);  
    }  
}  
  
if (fullScreen)  
{  
    ChangeDisplaySettings(NULL,0); // przywraca pulpit  
    ShowCursor(TRUE); // i wskaźnik myszy  
}  
  
return msg.wParam;  
}
```

Program ten stanowi ilustrację omówionego wcześniej przykładu sześciangu przesłaniającego kulę. Rysunek 12.1 przedstawia działanie programu w sytuacji, gdy używana jest funkcja porównania głębi GL\_LESS. Efekt zmiany funkcji porównania na GL\_ALWAYS prezentuje rysunek 12.2.

### Rysunek 12.1.

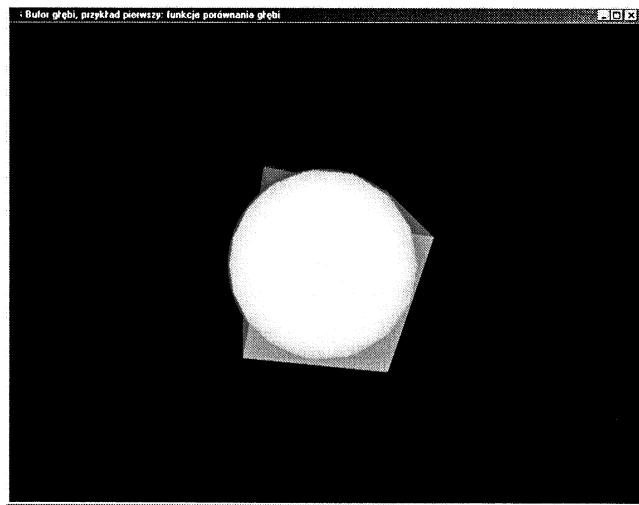
Przykład  
zastosowania  
funkcji GL\_LESS



Kolejnym przykładem zastosowania bufora głębi jest przekrój sceny. Rysując na przykład silnik samochodu można także pokazać jego wewnętrzną konstrukcję. Wykorzystuje się w tym celu bufor głębi i rysuje płaszczyznę przekroju. Płaszczyznę przekroju rysuje się w następujący sposób: trzeba wyłączyć tworzenie grafiki w buforze koloru za pomocą funkcji glDrawBuffer(), a następnie narysować płaszczyznę przekroju i włączyć z powrotem rysowanie w buforze koloru wywołując ponownie funkcję glDrawBuffer(). Ilustruje to poniższy fragment kodu:

**Rysunek 12.2.**

Przykład  
zastosowania  
funkcji *GL\_ALWAYS*



```
glDrawBuffer(GL_NONE); // wyłącza tworzenie grafiki w buforze koloru
// w tym miejscu rysuje płaszczyznę przekroju
glDrawBuffer(GL_BACK);
```

Przykład programu, który zaprezentowany zostanie poniżej, nie będzie rysował silnika samochodu, lecz stworzy grafikę złożoną z kuli umieszczonej wewnątrz nieprzezroczystego sześcianu. Rozpoczynając działanie programu pokaże użytkownikowi jedynie sześcian. Po wybraniu klawisza spacji na jednej ze ścian sześcianu zostanie narysowana płaszczyzna przekroju, która pozwoli dostrzec kulę umieszczoną wewnątrz sześcianu.

```
////// Pliki nagłówkowe
#include <windows.h> // standardowy plik nagłówkowy Windows
#include <math.h>
#include <gl/gl.h> // standardowy plik nagłówkowy OpenGL
#include <gl/glu.h> // plik nagłówkowy biblioteki GLU
#include <gl/glaux.h> // funkcje pomocnicze OpenGL

////// Zmienne globalne
float angle = 0.0f; // bieżący kąt obrotu
HDC g_HDC; // globalny kontekst urządzenia
bool fullScreen = false; // true = tryb pełnoekranowy;
bool keyPressed[256]; // false = tryb okienkowy
// tablica przyciśnięć klawiszy
bool cuttingPlane = true; // true: rysowana jest płaszczyzna przekroju

// Initialize
// opis: inicjuje OpenGL
void Initialize()
{
    glEnable(GL_DEPTH_TEST); // włącza bufor głębi
    glEnable(GL_CULL_FACE); // wyłącza rysowanie tylnych stron wielokątów
    glEnable(GL_LIGHTING); // włącza oświetlenie
    glEnable(GL_LIGHT0); // włącza źródło światła light0
    glEnable(GL_COLOR_MATERIAL); // kolor jako materiał
    glShadeModel(GL_SMOOTH); // cieniowanie gładkie
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // tło w kolorze czarnym
}
```

```
// Render
// opis: rysuje scenę
void Render()
{
    // opróżnia bufory ekranu i głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    angle += 0.5;                      // zwiększa kąt obrotu

    // odsuwa scenę o 100 jednostek
    glTranslatef(0.0f, 0.0f, -100.0f);

    // najpierw rysuje kulę wewnątrz sześcianu
    glPushMatrix();
    glRotatef(angle, 0.0f, 1.0f, 0.0f);
    glColor3f(1.0f, 1.0f, 0.0f);
    auxSolidSphere(5.0f);
    glPopMatrix();

    // a następnie nieprzezroczysty sześcian
    glPushMatrix();
    glRotatef(angle, 1.0f, 0.0f, 0.0f);
    glRotatef(angle*0.5f, 0.0f, 1.0f, 0.0f);
    glRotatef(angle*1.2f, 0.0f, 0.0f, 1.0f);

    // rysuje płaszczyznę przekroju
    // na jednej ze ścian sześcianu
    if (cuttingPlane)
    {
        glDrawBuffer(GL_NONE);

        glBegin(GL_QUADS);
        glVertex3f(-10.0f, -10.0f, 15.1f);
        glVertex3f(10.0f, -10.0f, 15.1f);
        glVertex3f(10.0f, 10.0f, 15.1f);
        glVertex3f(-10.0f, 10.0f, 15.1f);
        glEnd();

        glDrawBuffer(GL_BACK);
    }

    glColor3f(0.2f, 0.4f, 0.6f);
    auxSolidCube(30.0f);
    glPopMatrix();

    glFlush();
    SwapBuffers(g_HDC);                  // przełącza bufory
}

void SetupPixelFormat(HDC hDC)
{
    /* ... */
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
```

```

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
                    int nShowCmd)
{
    /* ... */
    // pętla przetwarzania komunikatów
    while (!done)
    {
        PeekMessage(&msg, hwnd, NULL, NULL, PM_REMOVE);

        if (msg.message == WM_QUIT)      // aplikacja otrzymała komunikat WM_QUIT?
        {
            done = true;              // jeśli tak, to kończy działanie
        }
        else
        {
            if (keyPressed[VK_ESCAPE])
                done = true;
            else
            {
                if (keyPressed[VK_SPACE])
                    cuttingPlane = !cuttingPlane;

                Render();

                TranslateMessage(&msg);    // tłumaczy komunikat i wysyła do systemu
                DispatchMessage(&msg);
            }
        }
    }

    if (fullScreen)
    {
        ChangeDisplaySettings(NULL,0); // przywraca pulpit
        ShowCursor(TRUE);           // i wskaźnik myszy
    }

    return msg.wParam;
}

```

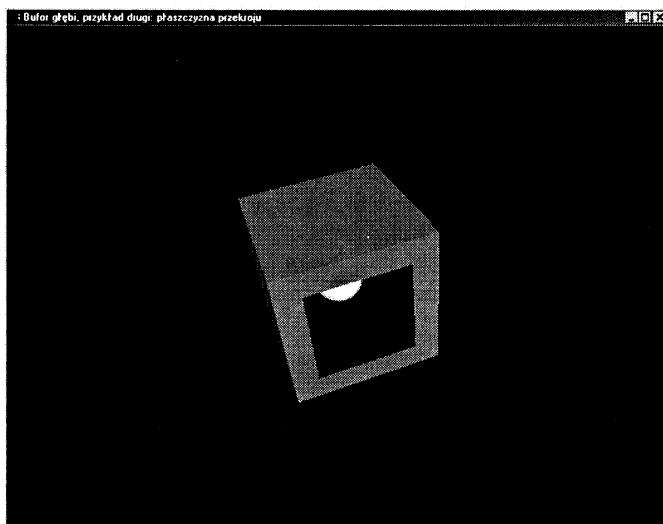
Program rysuje najpierw kulę, ponieważ będzie ona widoczna na płaszczyźnie przekroju. Następnie rysuje płaszczyznę przekroju, a dopiero potem sześcian. W ten sposób na jednej ze ścian sześcianu pojawia się otwór, przez który widoczna jest kula umieszczona we wnętrzu sześcianu. Działanie programu ilustruje rysunek 12.3.

## Bufor powielania

*Bufor powielania* stosuje się — podobnie do bufora głębi — aby zablokować tworzenie fragmentów grafiki. Jednak bufor powielania umożliwia przy tym osiągnięcie dodatkowych efektów, dla których możliwości bufora głębi nie są wystarczające. Przykładem zastosowania bufora powielania może być okno budynku, które umożliwia obserwację znajdujących się za nim obiektów.

**Rysunek 12.3.**

Przykład  
zastosowania  
płaszczyzny  
przekroju



Aby korzystać z bufora powielania, trzeba nadać polu cStencilBits struktury PIXELFORMAT\_DESCRIPTOR odpowiednią wartość, na przykład:

```
pfd.cStencilBits = 16;
```

Powyższy wiersz kodu konfiguruje 16-bitowy bufor powielania OpenGL. Przed rozpoczęciem jego używania trzeba jeszcze aktywować go za pomocą funkcji glEnable(), której należy przekazać wartość GL\_STENCIL\_TEST:

```
glEnable(GL_STENCIL_TEST);
```

Ale to jeszcze nie wszystko. W przypadku bufora powielania należy określić jeszcze sposób jego wykorzystania przez maszynę OpenGL. Jeśli to nie zostanie zrobione, bufor powielania nie będzie używany w procesie tworzenia grafiki. Sposób wykorzystania bufora powielania określa się za pomocą funkcji glStencilFunc() i glStencilOp(). Funkcja glStencilFunc() zdefiniowana jest następująco:

```
void glStencilFunc(GLenum func, GLint ref, GLuint mask);
```

Jej parametrami są (kolejno): funkcja porównania, wartość referencji i maska powielania. Tabela 12.7 prezentuje dostępne funkcje porównania.

Poniżej zaprezentowany został przykład wywołania funkcji glStencilFunc(), w efekcie którego wybrana zostaje funkcja porównania dającą zawsze rezultat pozytywny, wartość referencji równą 1 i maskę równą 1:

```
glStencilFunc(GL_ALWAYS, 1, 1);
```

Funkcja glStencilOp() pozwala określić rodzaj operacji powielania. Posiada ona następujący prototyp:

```
void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);
```

**Tabela 12.7.** Funkcje porównania dla bufora powielania

Funkcja	Opis
GL_NEVER	Rezultat porównania jest zawsze negatywny
GL_LESS	Rezultat porównania jest pozytywny, jeśli nowa wartość współrzędnej z jest mniejsza od wartości współrzędnej zapisanej w buforze (domyślna funkcja porównania głębi)
GL_EQUAL	Rezultat porównania jest pozytywny, jeśli nowa wartość współrzędnej z jest równa wartości współrzędnej zapisanej w buforze
GL_LEQUAL	Rezultat porównania jest pozytywny, jeśli nowa wartość współrzędnej z jest mniejsza lub równa wartości współrzędnej zapisanej w buforze
GL_GREATER	Rezultat porównania jest pozytywny, jeśli nowa wartość współrzędnej z jest większa od wartości współrzędnej zapisanej w buforze
GL_NOTEQUAL	Rezultat porównania jest pozytywny, jeśli nowa wartość współrzędnej z jest różna od wartości współrzędnej zapisanej w buforze
GL_GEQUAL	Rezultat porównania jest pozytywny, jeśli nowa wartość współrzędnej z jest większa lub równa wartości współrzędnej zapisanej w buforze
GL_ALWAYS	Rezultat porównania jest zawsze pozytywny

Jej parametry określają kolejno: rodzaj akcji podejmowanej, gdy funkcja porównania dla bufora powielania da rezultat negatywny (`fail`), funkcja porównania dla bufora powielania da rezultat pozytywny, ale negatywny dla bufora głębi (`zfail`), a także wtedy, gdy funkcja porównania da w obu przypadkach rezultat pozytywny (`zpass`). Parametry te mogą przyjmować wartości przedstawione w tabeli 12.8.

**Tabela 12.8.** Operacje bufora powielania

Operacja	Opis
GL_KEEP	Zachowuje bieżącą wartość
GL_ZERO	Nadaje wartość 0
GL_REPLACE	Nadaje wartość referencji <code>ref</code> określoną za pomocą funkcji <code>glStencilFunc()</code>
GL_INCR	Zwiększa bieżącą wartość o 1
GL_DECR	Zmniejsza bieżącą wartość o 1
GL_INVERT	Zmienia wartości wszystkich bitów na przeciwnie

Poniższe wywołanie informuje OpenGL, że za każdym razem, gdy tworzona jest grafika w buforze powielania, powinien wypełnić go wartością `ref`, która zdefiniowana została za pomocą funkcji `glStencilFunc()`:

```
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
```

W kolejnym z omawianych przykładów programów wykorzystany zostanie poniższy fragment kodu do narysowania posadzki w buforze powielania:

```
// konfiguruje bufor powielania - wypełnianie wartością referencji
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
glStencilFunc(GL_ALWAYS, 1, 1);

// rysuje posadzkę w buforze powielania
DrawFloor();
```

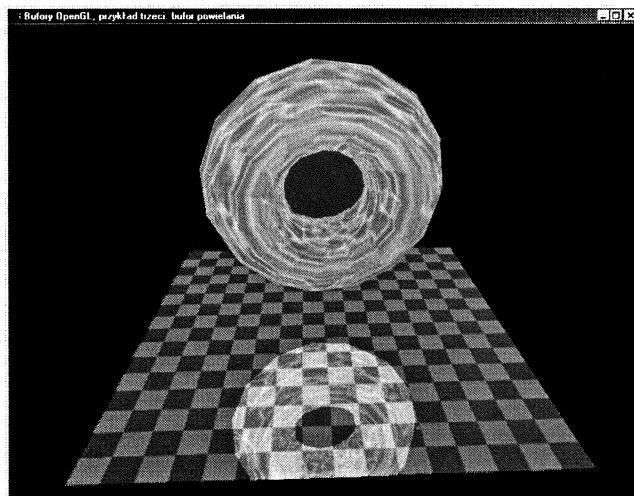
Posadzki tej nie można zobaczyć, ale stworzony został w ten sposób w buforze powielania rodzaj otworu, który będzie obcinął fragmenty obiektów znajdujące się poza nim. Czas przeanalizować kolejny przykład.

## Przykład zastosowania bufora powielania

Przykład, który został omówiony w niniejszym podrozdziale, prezentuje sposób, w jaki można tworzyć odbicia i obcinać je za pomocą krawędzi odbijającego obiektu, którym będzie w tym przypadku posadzka. Rysunek 12.4 przedstawia efekt działania prezentowanego programu.

Rysunek 12.4.

Przykład  
zastosowania  
bufora powielania



Tworząc kolejną klatkę animacji program wyłącza najpierw testowanie głębi, później możliwość modyfikacji bufora koloru. Następnie włącza bufor powielania i tworzy w nim obraz podłogi. Po ponownym włączeniu bufora głębi i modyfikacji kolorów konfiguruje bufor powielania w taki sposób, że tworzenie grafiki możliwe jest tylko w przypadku, gdy bufor powielania zawiera wartość 1, czyli wartość referencji. Dzięki temu OpenGL nie będzie tworzyć grafiki poza obszarem wyznaczonym przez wartości 1 umieszczone w buforze powielania. W tym momencie może przystąpić do rysowania odbicia, które nie będzie pojawiać się poza obszarem posadzki. W tym celu skaluje układ współrzędnych tak, by uzyskać jego przeciwną orientację, a później rysuje torus pod płaszczyzną posadzki.

Po narysowaniu odbicia program wyłącza bufor powielania i tworzy pozostałą część sceny w zwykły sposób. Najpierw rysuje przezroczystą posadzkę, a następnie prawdziwy torus.

Przyjrzyjmy się zatem kodowi programu:

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

```

////// Pliki nagłówkowe
#include <windows.h>                                // standardowy plik nagłówkowy Windows
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glaux.h>                                // standardowy plik nagłówkowy OpenGL
                                                       // plik nagłówkowy biblioteki GLU
                                                       // funkcje pomocnicze 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 objectAngle = 0.0f;                           // kąt obrotu obiektu
float angle = 0.0f;                                 // kąt kamery
float radians = 0.0f;                               // kąt kamery w radianach

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

////// Opis tekstury
texture_t *envTex;                                // tekstura otoczenia
texture_t *floorTex;                             // tekstura posadzki

// wierzchołki posadzki
float floorData[4][3] = { { -5.0, 0.0, 5.0 }, { 5.0, 0.0, 5.0 },
                         { 5.0, 0.0, -5.0 }, { -5.0, 0.0, -5.0 } };

/* Kod funkcji LoadBitmapFile() został pominięty */
/* Kod funkcji LoadTextureFile() został pominięty */

bool LoadAllTextures()
{
    // ładuje teksturę otoczenia
    envTex = LoadTextureFile("waterenv.bmp");
    if (envTex == NULL)
        return false;

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

    // konfiguruje teksturę 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);
glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, envTex->width, envTex->height,
                  GL_RGB, GL_UNSIGNED_BYTE, envTex->data);

// konfiguruje teksturę posadzki
glBindTexture(GL_TEXTURE_2D, floorTex->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);
glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, floorTex->width, floorTex->height,
                  GL_RGB, GL_UNSIGNED_BYTE, floorTex->data);

    return true;
}

void CleanUp()
{
    free(envTex);
    free(floorTex);
}

// Initialize
// opis: inicjuje OpenGL
void Initialize()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // tło w kolorze czarnym

    glShadeModel(GL_SMOOTH);           // cieniowanie gładkie
    glEnable(GL_CULL_FACE);           // wyłącza rysowanie tylnych stron wielokątów
    glFrontFace(GL_CCW);              // tylnych = o uporządkowaniu wierzchołków
                                      // w kierunku przeciwnym do kierunku
                                      // ruchu wskaźówek zegara
    glEnable(GL_TEXTURE_2D);           // włącza tekstury dwuwymiarowe

    LoadAllTextures();
}

// DrawFloor()
// opis: rysuje posadzkę (jako pojedynczy czworokąt)
void DrawFloor()
{
    glBindTexture(GL_TEXTURE_2D, floorTex->texID);
    glBegin(GL_QUADS);
        glTexCoord2f(0.0, 0.0); glVertex3fv(floorData[0]);
        glTexCoord2f(0.0, 4.0); glVertex3fv(floorData[1]);
        glTexCoord2f(4.0, 4.0); glVertex3fv(floorData[2]);
        glTexCoord2f(4.0, 0.0); glVertex3fv(floorData[3]);
    glEnd();
}

// DrawTorus()
// opis: rysuje torus pokryty tekstrurą
```

```
void DrawTorus()
{
    glPushMatrix();
    // konfiguruje odwzorowania otoczenia
    glTexGen(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
    glTexGen(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);

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

    // przesuwa i obraca
    glTranslatef(0.0f, 4.0f, 0.0f);
    glRotatef(objectAngle, 1.0f, 1.0f, 0.0f);
    glRotatef(objectAngle, 0.0f, 1.0f, 0.0f);
    glRotatef(objectAngle, 0.0f, 0.0f, 1.0f);

    // rysuje torus
    auxSolidTorus(1.0f, 2.0f);

    // wyłącza tworzenie współrzędnych tekstury
    glDisable(GL_TEXTURE_GEN_T);
    glDisable(GL_TEXTURE_GEN_S);
    glPopMatrix();
}

// Render
// opis: rysuje scenę
void Render()
{
    objectAngle += 0.2f;// zwiększa kąt obrotu
    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 (0,2,0)
    lookX = 0.0f;
    lookY = 2.0f;
    lookZ = 0.0f;

    // opróżnia bufory koloru, gębi i powielania
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
    glLoadIdentity();

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

    // wyłącza bufor gębi
    glDisable(GL_DEPTH_TEST);

    // zabrania modyfikacji wszystkich składowych koloru
    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
```

```
// włącza bufor powielania
glEnable(GL_STENCIL_TEST);

// określa wartość referencji dla bufora powielania
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
glStencilFunc(GL_ALWAYS, 1, 1);

// rysuje posadzkę; nadaje wszystkim pikselom w buforze powielania wartość 1,
// która została zdefiniowana jako wartość maski za pomocą funkcji glStencilFunc()
DrawFloor();

// włącza możliwość modyfikacji składowych koloru
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);

// włącza bufor głębi
glEnable(GL_DEPTH_TEST);

// rysowanie możliwe tylko tam, gdzie bufor powielania zawiera 1
glStencilFunc(GL_EQUAL, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

// rysuje "odbicie"
glPushMatrix();

// tworzy odbicie torusa
glScalef(1.0, -1.0, 1.0);

// zabrania rysowania przednich stron wielokątów
glCullFace(GL_FRONT);

// rysuje odbicie torusa
DrawTorus();

// ponownie zabrania rysowania tylnych stron wielokątów
glCullFace(GL_BACK);

glPopMatrix();

// wyłącza bufor powielania
glDisable(GL_STENCIL_TEST);

// rysuje przezroczystą podłogę
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glColor4f(1.0f, 1.0f, 1.0f, 0.4f);
DrawFloor();
glDisable(GL_BLEND);

// rysuje "prawdziwy" torus
DrawTorus();

glFlush();
SwapBuffers(g_HDC); // przełącza bufory
}

// funkcja określająca format pikseli
void SetupPixelFormat(HDC hDC)
{
```

```

int nPixelFormat; // indeks formatu pikseli

static PIXELFORMATDESCRIPTOR pfd = {
    sizeof(PIXELFORMATDESCRIPTOR), // rozmiar struktury
    1, // domyślna wersja
    PFD_DRAW_TO_WINDOW | // grafika w oknie
    PFD_SUPPORT_OPENGL | // grafika OpenGL
    PFD_DOUBLEBUFFER, // podwójne buforowanie
    PFD_TYPE_RGBA, // tryb kolorów RGBA
    32, // 32-bitowy opis kolorów
    0, 0, 0, 0, 0, 0, // nie specyfikuje bitów kolorów
    0, // bez bufora alfa
    0, // nie specyfikuje bitu przesunięcia
    0, // bez bufora akumulacji
    0, 0, 0, 0, // ignoruje bity akumulacji
    16, // 16-bitowy bufor z
    16, // 16-bitowy bufor powielania
    0, // bez buforów pomocniczych
    PFD_MAIN_PLANE, // główna płaszczyzna rysowania
    0, // zarezerwowane
    0, 0, 0 }; // ignoruje maski warstw

// wybiera najbardziej zgodny format pikseli
nPixelFormat = ChoosePixelFormat(hDC, &pfd);

// określa format pikseli dla danego kontekstu urządzenia
SetPixelFormat(hDC, nPixelFormat, &pfd);
}

/* Kod funkcji WndProc() został pominięty */
/* Kod funkcji WinMain() został pominięty */

```

## Bufor akumulacji

Trzeba powiedzieć to otwarcie: przy pisaniu tej książki implementacja bufora akumulacji działała wolno. Tak długo jak nie będą dostępne sprzętowe rozwiązania obsługi bufora akumulacji, jego zastosowanie w grach nie będzie możliwe. Dlatego też bufor akumulacji nie będzie omawiany szczegółowo, a jedynie przedstawiona zostanie jego podstawowa funkcjonalność.

Koncepcja bufora akumulacji polega na *akumulacji* wielu kolejnych obrazów tworzonych w buforze koloru. Obraz zakumulowany w buforze akumulacji można następnie umieścić w buforze koloru i uzyskać w ten sposób wiele ciekawych efektów, takich jak na przykład rozmycie obrazu wskutek ruchu obiektów, antialiasing czy cienie.

Z działaniem bufora akumulacji związana jest tylko jedna funkcja OpenGL:

```
void glAccum(GLenum op, GLfloat value);
```

Parametr *op* określa wykonywaną operację, a parametr *value* wartość używaną przez tę operację. Tabela 12.9 przedstawia dostępne operacje.

**Tabela 12.9.** Operacje bufora akumulacji

Operacja	Opis
GL_ACCUM	Do zawartości bufora akumulacji dodawane są wartości RGBA pobierane z bieżącego bufora
GL_LOAD	Tak jak w przypadku GL_ACCUM, ale pobierane wartości zastępują wartości w buforze akumulacji
GL_ADD	Dodaje wartość <i>value</i> do wartości każdego piksela w buforze akumulacji
GL_MULT	Mnoży wartość każdego piksela w buforze akumulacji razy wartość <i>value</i>
GL_RETURN	Mnoży wartość każdego piksela w buforze akumulacji razy wartość <i>value</i> , a wynik umieszcza w buforze koloru

Aby uzyskać efekt rozmycia obiektu na skutek jego ruchu, należy umieścić kilka kolejnych obrazów reprezentujących „ogon” obiektu w buforze akumulacji za pomocą funkcji `glAccum()`:

```
glAccum(GL_ACCUM, 0.1);
```

W przypadku zastosowania bufora akumulacji w celu uzyskania efektu rozmycia parametr *value* traktowany jest jak współczynnik zaniku kolejnych obrazów. Za każdym razem, gdy wywołana zostanie powyższa funkcja, obraz umieszczany w buforze akumulacji będzie słabszy od poprzedniego. Po umieszczeniu wszystkich obrazów w buforze akumulacji należy wywołać ponownie funkcję `glAccum()` (tym razem w celu wykonania operacji GL\_RETURN):

```
glAccum(GL_RETURN, 1.0);
```

Poniższy fragment kodu ilustruje sposób zastosowania obu operacji bufora akumulacji w celu utworzenia pojedynczego obrazu zawierającego efekt rozmycia obiektu na skutek jego ruchu:

```
int idx;

// najpierw tworzy pełen obraz obiektu
DrawObject();

// konfiguruje bufor akumulacji tak, by zawierał 70% obrazu obiektu
glAccum(GL_LOAD, 0.7);

// rysuje w pętli cztery obrazy obiektu tworzące rozmyty "ogon"
for(idx = 1; idx < 5; idx++)
{
    // obraca obiekt wokół osi y i rysuje go
    glRotatef(objectAngle - (float)idx, 0.0, 1.0, 0.0);
    DrawObject();

    // umieszcza 20% narysowanego obrazu w buforze akumulacji
    glAccum(GL_ACCUM, 0.2);
}

// umieszcza zawartość bufora akumulacji w buforze koloru
glAccum(GL_RETURN, 1.0);
```

Przykład ten kończy krótkie omówienie bufora akumulacji. Chociaż jest on bardzo przydatny do tworzenia ciekawych efektów graficznych, to jednak jego niska efektywność nie pozwala stosować go w grach. Ponieważ możliwości sprzętu graficznego rosną w zawrotnym tempie, być może wkrótce pojawi się także możliwość sprzętowej obsługi bufora akumulacji.

## Podsumowanie

Obszary ekranu reprezentować można za pomocą dwuwymiarowych tablic opisujących kolor pikseli, czyli *buforów*. W OpenGL bufory przechowują dla każdego piksela składowe koloru w modelu RGBA lub indeks koloru w modelu indeksowym. Bufor ekranu stanowi kompozycję wszystkich pozostałych buforów systemu.

OpenGL wymaga określenia formatu pikseli za pomocą struktury `PIXELFORMATDESCRIPTOR`.

OpenGL udostępnia osobną funkcję dla każdego typu bufora pozwalającą określić wartość, którą będzie wypełniany bufor podczas kasowania jego zawartości: `glClearColor()`, `glClearDepth()`, `glClearStencil()` i `glClearAccum()`.

*Podwójne buforowanie* polega na zastosowaniu dwóch buforów podczas tworzenia klatek animacji. Gdy wyświetlana jest zawartość jednego bufora, w drugim tworzona jest kolejna klatka animacji. Po jej ukończeniu bufory zostają przełączone. Rozwiążanie takie pozwala uzyskać efekt płynnej animacji i zapobiega migotaniu ekranu. Podwójne buforowanie dostępne jest jedynie w przypadku bufora koloru. Nie jest używane w przypadku buforów głębi, akumulacji i powielania.

*Buforowanie stereoskopowe* korzysta z osobnych buforów dla lewego i prawego oka, co pozwala osiągnąć w ten sposób efekt pełnej trójwymiarowości obrazu.

*Bufor głębi* zapobiega rysowaniu przesłoniętych obiektów. Przechowuje on dla każdego piksela odległość pomiędzy obserwatorem i punktem obiektu reprezentowanym przez dany piksel. Podczas rysowania obiektu przesłaniającego innego obiektu zawartość bufora głębi modyfikowana jest na podstawie funkcji porównania głębi.

Również *bufor powielania* zapobiega rysowaniu fragmentów tworzonej sceny. Umożliwia jednak uzyskanie efektów, dla których bufor głębi jest niewystarczający. Przykładem zastosowania bufora powielania mogą być okna budynków pozwalające obserwować obiekty znajdujące się na zewnątrz.

Koncepcja bufora akumulacji polega na *akumulacji* wielu obrazów tworzonych w buforze koloru. Zakumulowany obraz można następnie przenieść do bufora koloru i uzyskać w ten sposób efekty, takie jak rozmycie obiektów na skutek ich ruchu, antialiasing czy cienie.

## Rozdział 13.

# Powierzchnie drugiego stopnia

W rozdziale 4. omówione zostały szczegółowo podstawowe elementy grafiki, takie jak punkty czy trójkąty. Z elementów tych tworzy się większość obiektów grafiki trójwymiarowej. Czasami jednak wygodniej jest skorzystać z gotowych obiektów reprezentujących typowe bryły — na przykład kule bądź walce. Oczywiście można stworzyć własną bibliotekę takich obiektów, ale zamiast ponownie wymyślać koło, w większości przypadków lepiej skorzystać z obiektów udostępnianych przez bibliotekę GLU.

Biblioteka GLU umożliwia rysowanie różnych *powierzchni drugiego stopnia*, takich jak dyski, walce, stożki i kule. Zanim omówione zostaną sposoby tworzenia poszczególnych rodzajów powierzchni, najpierw przedstawić trzeba ogólne zasady posługiwania się nimi.

W rozdziale tym przedstawione zostaną następujące zagadnienia:

- ◆ powierzchnie drugiego stopnia i sposób ich tworzenia;
- ◆ zmiana właściwości powierzchni drugiego stopnia posiadająca wpływ na ich wygląd;
- ◆ rodzaje powierzchni drugiego stopnia udostępniane przez bibliotekę GLU.

## Powierzchnie drugiego stopnia w OpenGL

Aby właściwie zrozumieć zasady posługiwania się *powierzchniami drugiego stopnia* w OpenGL, najlepiej jest przyjąć, że termin ten odnosi się nie tyle do powierzchni w sensie konkretnego obiektu graficznego, ale raczej do obiektu określającego sposób rysowania tej powierzchni. Dlatego też zanim narysuje się jakąkolwiek powierzchnię drugiego stopnia, trzeba najpierw stworzyć odpowiedni obiekt.

Obiekt powierzchni drugiego stopnia można stworzyć wywołując funkcję `gluNewQuadric()`:

```
GLUquadricObj *gluNewQuadric();
```

Funkcja ta zwraca wskaźnik nowego obiektu powierzchni drugiego stopnia (bądź wartość NULL, jeśli obiekt nie został utworzony).

Po utworzeniu obiektu można użyć go do rysowania wielu powierzchni. Za pomocą jednego obiektu powierzchni drugiego stopnia można rysować dyski, walce, stożki i kule. W tym momencie można oczywiście zapytać, po co tworzyć więcej niż jeden obiekt powierzchni drugiego stopnia? I skoro wystarczy jeden taki obiekt do rysowania dowolnych powierzchni, to czy nie może być on tworzony automatycznie przez maszynę OpenGL?

Rzeczywiście, w przypadku niektórych zastosowań wystarczający okaże się pojedynczy obiekt. Jednak trzeba pamiętać, że każdy obiekt powierzchni drugiego stopnia posiada pewien stan określający sposób rysowania powierzchni. Dysponując pojedynczym obiektem można rysować albo wiele takich samych powierzchni, albo za każdym razem, gdy trzeba narysować inną powierzchnię, zmieniać stan obiektu. W tym drugim przypadku okaże się, że bardziej efektywnym rozwiązaniem jest posiadanie wielu obiektów o różnych stanach niż częste zmiany stanu pojedynczego obiektu.

Stan obiektu powierzchni drugiego stopnia określa:

- ◆ styl rysowanej powierzchni;
- ◆ jej normalne;
- ◆ orientację powierzchni;
- ◆ współrzędne tekstury.

## Styl powierzchni

Styl powierzchni określa, czy tworzące ją wielokąty będą wypełniane podczas rysowania, czy rysowane będą tylko ich krawędzie lub same wierzchołki. Dostępny jest także styl sylwetki, który podobny jest do stylu, w którym rysowane są krawędzie wielokątów, ale różni się tym, że nie są rysowane krawędzie łączące wielokąty leżące w jednej płaszczyźnie. Styl powierzchni określać można za pomocą funkcji

```
void gluQuadricDrawStyle(GLUquadricObj *quadObj, GLenum style);
```

Parametr *quadObj* jest wskaźnikiem obiektu powierzchni, którego stan należy zmienić, a parametr *style* może przyjmować wartość GLU\_FILL, GLU\_LINE, GLU\_POINT lub GLU\_SILHOUETTE. Domyślnym stylem rysowania powierzchni jest GLU\_FILL.

## Wektory normalne

Stan obiektu powierzchni drugiego stopnia określa także sposób tworzenia wektorów normalnych do rysowanej powierzchni. Wektory te mogą być tworzone dla każdego wielokąta (w rezultacie otrzymuje się cieniowanie płaskie) bądź generowane dla każdego wierzchołka (cieniowanie gładkie). Sposób tworzenia wektorów normalnych określa się za pomocą funkcji

```
void gluQuadricNormals(GLUquadricObj *quadObj, GLenum normalMode);
```

Parametr *quadObj* jest wskaźnikiem obiektu powierzchni drugiego stopnia, a parametr *normalMode* może przyjmować wartość *GLU\_NONE*, *GLU\_FLAT* lub *GLU\_SMOOTH*.

## Orientacja

Precyzyjne wytłumaczenie znaczenia orientacji zależy w pewnym stopniu od rodzaju powierzchni drugiego stopnia. W ogólnym przypadku orientacja określa zwrot wektorów normalnych do powierzchni. Określa się ją za pomocą funkcji

```
void gluQuadricOrientation(GLUquadricObj *quadObj, GLenum orientation);
```

Parametr *orientation* może przyjmować domyślną wartość *GLU\_OUTSIDE* lub wartość *GLU\_INSIDE*. Wartość *GLU\_OUTSIDE* sprawia, że wektory normalne skierowane są na zewnątrz powierzchni, a *GLU\_INSIDE* do wewnętrz. O ile pojęcie wnętrza jest oczywiste w przypadku stożka, walca czy kuli, to w przypadku dysku wymaga ono dodatkowej definicji. Przez zewnętrzna stronę dysku rozumieć należy stronę zwróconą w kierunku dodatniej części osi z.

## Współrzędne tekstury

Obiekt powierzchni drugiego stopnia określa też, czy dla rysowanej powierzchni tworzone są współrzędne tekstury. Dokładny sposób tworzenia współrzędnych tekstury zależy od rodzaju powierzchni. Tworzenie współrzędnych tekstury można kontrolować za pomocą funkcji

```
void gluQuadricTexture(GLUquadricObj *quadObj, GLboolean useTextureCoords);
```

Jeśli parametr *useTextureCoords* posiada wartość *GL\_TRUE*, to współrzędne tekstury są tworzone. Domyślnie generowanie współrzędnych tekstury jest wyłączone.

## Usuwanie obiektów powierzchni

Obiekty powierzchni drugiego stopnia zajmują pamięć i dlatego należy je usuwać, gdy nie są już potrzebne. Służy do tego funkcja *gluDeleteQuadric()*:

```
void gluDeleteQuadric(GLUquadricObj *quadObj);
```

Jej wywołanie usuwa obiekt wskazywany przez parametr *quadObj* i zwalnia zajmowaną przez niego pamięć.

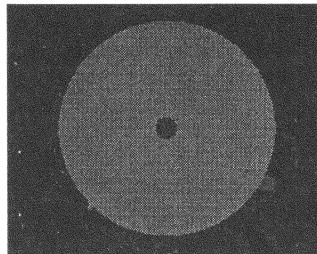
W ten sposób poznane zostały zasady tworzenia i usuwania obiektów powierzchni drugiego stopnia, a także możliwości zmiany ich stanu. Najwyższa więc pora, by wykorzystać je do rysowania powierzchni.

## Dyski

*Dyskiem*, w sensie biblioteki GLU, jest płaskie koło, które może posiadać w środku otwór, co ilustruje rysunek 13.1. Dysk rysowany jest wokół środka układu współrzędnych w płaszczystie z = 0. Dowolne położenie można mu nadać za pomocą przesunięć i obrótów.

**Rysunek 13.1.**

Dysk jako  
powierzchnia  
drugiego stopnia



Dysk można narysować wywołując poniższą funkcję:

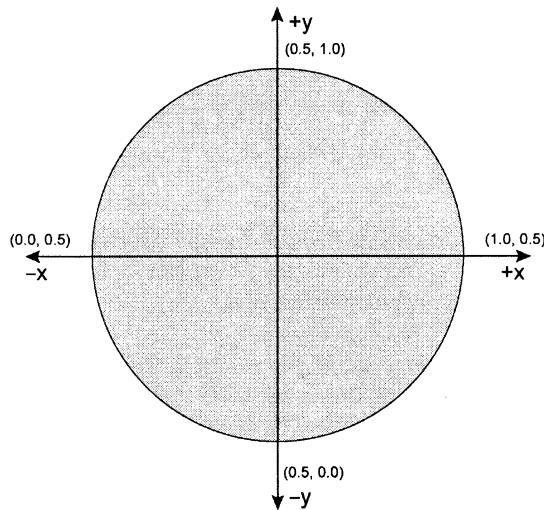
```
void gluDisk(GLUquadricObj *quadObj,
GLdouble innerRadius,
GLdouble outerRadius,
GLint slices,
GLint loops);
```

Parametr *quadObj* jest wskaźnikiem obiektu powierzchni. Promień dysku definiuje parametr *outerRadius*. Jeśli wartość parametru *innerRadius* określającego wewnętrzny promień dysku jest większa od 0, to dysk posiadać będzie w środku otwór. Parametry *slices* i *loops* kontrolują liczbę wielokątów tworzących dysk. Parametr *slices* definiuje liczbę wielokątów wokół osi z, a parametr *loops* liczbę pierścieni wielokątów tworzących dysk. Im większa jest wartość obu parametrów, tym lepszy będzie wygląd dysku, ale i dłuższy okaże się czas jego rysowania. Trzeba więc znaleźć rozsądny kompromis pomiędzy jakością i efektywnością. Aby dysk nie posiadał nadmiaru kantów, parametr *slices* powinien mieć wartość co najmniej 20. W większości przypadków wystarczy, że parametr *loops* posiadał będzie wartość 2. Należy ją zwiększyć, jeśli planowane jest stworzenie na powierzchni dysku efektu odbicia.

Jeśli dla dysku tworzone są współrzędne tekstury, to ich wartości generowane są liniowo dookoła dysku, co ilustruje rysunek 13.2.

**Rysunek 13.2.**

Współrzędne  
tekstury dysku



Oprócz pełnych dysków biblioteka GLU umożliwia także tworzenie wycinków dysku za pomocą funkcji `gluPartialDisk()` o następującym prototypie:

```
void gluPartialDisk(GLUquadricObj *quadObj,  
GLdouble innerRadius,  
GLdouble outerRadius,  
GLint slices,  
GLint loops,  
GLdouble startAngle,  
GLdouble sweepAngle);
```

Pierwsze pięć parametrów funkcji `gluPartialDisk()` posiada dokładnie takie samo znaczenie jak w przypadku funkcji `gluDisk()`. Parametr `startAngle` określa początkową wartość kąta wycinka dysku. Kąty mierzone są w stopniach (od dodatniej części osi y w kierunku zgodnym z kierunkiem ruchu wskazówek zegara). Parametr `sweepAngle` określa kąt rozwarcia wycinka dysku. Tak więc końcowa wartość kąta wycinka wynosi `startAngle + sweepAngle`. Jeśli na przykład parametry `startAngle` i `sweepAngle` będą miały wartość  $90^\circ$ , to wycinek dysku zostanie narysowany pomiędzy dodatnią częścią osi x i ujemną częścią osi y.

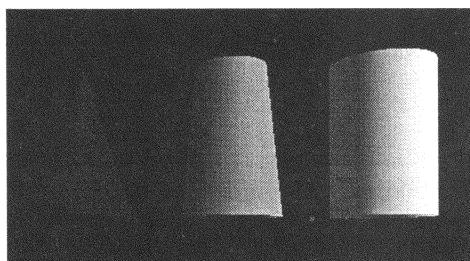
Orientacja i współrzędne tekstury wycinka dysku zdefiniowane są tak samo jak w przypadku pełnych dysków.

## Walce

Walce i stożki rysowane są za pomocą tej samej funkcji. Dlatego też określać je można wspólnym mianem walców (stożek należy uważać za specjalny przypadek walca). Kilka różnych walców pokazuje rysunek 13.3. Walce rysowane są wokół dodatniej części osi z, a ich podstawa leży w płaszczyźnie  $z = 0$ . Oczywiście za pomocą odpowiednich przekształceń można umieścić je w dowolnym punkcie przestrzeni.

Rysunek 13.3.

Walce



Walce udostępniane przez bibliotekę GLU posiadają jedynie powierzchnię boczną. Ich podstawy nie są rysowane. Można je uzyskać rysując dyski o takiej samej wartości parametru `slices` jak dla powierzchni walca.

Walce rysuje się za pomocą funkcji `gluCylinder()` zdefiniowanej jak poniżej:

```
void gluCylinder(GLUquadricObj *quadObj,  
GLdouble baseRadius,  
GLdouble topRadius,
```

```
GLdouble height,  
GLint slices,  
GLint stacks);
```

Parametr *quadObj* wskazuje oczywiście obiekt powierzchni. Parametr *baseRadius* określa promień dolnej podstawy walca (dla której współrzędna z posiada wartość 0). Parametr *topRadius* określa promień górnej podstawy walca (dla której współrzędna z posiada wartość *height* definiującą wysokość walca). Parametry *slices* i *stacks* wyznaczają liczbę wielokątów tworzących powierzchnię walca. Parametr *slices* określa liczbę wielokątów wokół osi z i powinien mieć wartość co najmniej 20, natomiast parametr *stacks* określa liczbę wielokątów wzdłuż osi z. Zwykle wystarczy, jeśli parametr *stacks* posiada wartość z przedziału od 2 do 4. Większe wartości parametru *stacks* są przydatne przy tworzeniu efektów odbicia na powierzchni walca.

Dolna i górnna podstawa walca mogą mieć różne średnice. Jeśli średnica jednej z nich równa jest 0, w efekcie otrzymany zostaje stożek.

Jeśli włączone zostanie generowanie współrzędnych tekstury, to będą one tworzone liniowo. Współrzędna t zmieniać się będzie wzdłuż osi walca i ostatecznie przyjmie wartość 0.0 dla wierzchołków, których współrzędna z równa jest 0 oraz wartość 1.0 dla wierzchołków, których współrzędna z posiada wartość *height*. Współrzędna s zmienia się od wartości 0.0 na dodatniej części osi y, przez wartość 0.25 na dodatniej części osi x, wartość 0.5 na ujemnej części osi y, wartość 0.75 na ujemnej części osi x, do wartości 1.0 z powrotem na dodatniej części osi y.

## Kule

Kule, a precyzyjniej rzecz ujmując, ich powierzchnie stanowią najłatwiejszy w użyciu rodzaj powierzchni udostępnianych przez bibliotekę GLU. Rysuje się je za pomocą funkcji *gluSphere()* zdefiniowanej następująco:

```
void gluSphere(GLUquadricObj *quadObj,  
               GLdouble radius,  
               GLint slices,  
               GLint stacks);
```

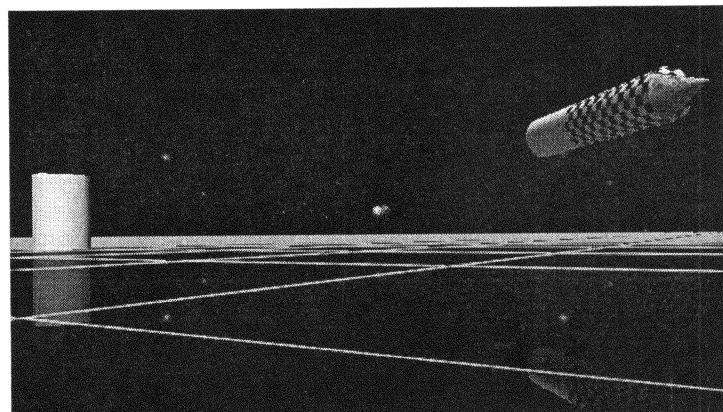
Parametr *radius* definiuje promień powierzchni kuli, a parametry *slices* i *stacks* liczbę wielokątów odpowiednio dookoła i wzdłuż osi z. Jako że oba parametry mają równy wpływ na wygląd powierzchni kuli, ich wartość powinna wynosić co najmniej 20.

Współrzędne tekstury dla powierzchni kuli tworzone są w ten sposób, że wartość współrzędnej t zmienia się liniowo wzdłuż osi z od 0.0 dla wierzchołków, których współrzędna z posiada wartość *-radius*, do wartości 1.0 dla wierzchołków, dla których współrzędna z posiada wartość *radius*. Współrzędna s zmienia się w taki sam sposób jak w przypadku walców.

# Przykład użycia powierzchni drugiego stopnia

Przedstawiony teraz zostanie przykład programu ilustrującego użycie powierzchni drugiego stopnia. Rysunek 13.4 prezentuje trójwymiarowy świat, w którym program umieścił różne rodzaje powierzchni drugiego stopnia. Zastosowane zostały przy tym różne style powierzchni i różne sposoby generowania wektorów normalnych tworzące różne efekty cieniowania. W wirtualnym świecie można poruszać się korzystając z klawiszy ze strzałkami. Pełen kod źródłowy programu został umieszczony na dysku CD. Tutaj omówione zostaną jego fragmenty związane z użyciem powierzchni drugiego stopnia.

**Rysunek 13.4.**  
*Wirtualny świat wypełniony powierzchniami drugiego stopnia*



Zanim narysowane zostaną powierzchnie drugiego stopnia, trzeba najpierw utworzyć odpowiednie obiekty i określić ich stan. Jest to zadanie funkcji InitializeScene():

```
GLUquadricObj *g_normalObject = NULL;
GLUquadricObj *g_wireframeObject = NULL;
GLUquadricObj *g_texturedObject = NULL;
GLUquadricObj *g_flatshadedObject = NULL;

...
BOOL InitializeScene()
{
    ...
    // tworzy "zwykły" obiekt powierzchni (domyślne wartości stanu)
    g_normalObject = gluNewQuadric();

    // tworzy obiekt powierzchni o stylu "szkieletowym"
    g_wireframeObject = gluNewQuadric();
    gluQuadricDrawStyle(g_wireframeObject, GLU_LINE);

    // tworzy obiekt powierzchni generujący współrzędne tekstury
    g_texturedObject = gluNewQuadric();
    gluQuadricTexture(g_texturedObject, GL_TRUE);
```

```
// tworzy obiekt powierzchni stosujący cieniowanie płaskie
g_flatshadedObject = gluNewQuadric();
gluQuadricNormals(g_flatshadedObject, GLU_FLAT);

...}
```

Funkcja InitializeScene() tworzy cztery obiekty powierzchni za pomocą funkcji gluNewQuadric(). Obiekt wskazywany przez zmienną g\_normalObject należy pozostawić w domyślnym stanie. Dla obiektu wskazywanego przez g\_wireframeObject trzeba wywołać funkcję gluQuadricDrawStyle() w celu poinformowania OpenGL, że rysowane będą tylko krawędzie wielokątów tworzących powierzchnię. Trzeci z obiektów, wskazywany przez zmienną g\_texturedObject, będzie generował współrzędne tekstury, ponieważ wywołana została dla niego funkcja gluQuadricTexture(). I wreszcie ostatni z obiektów, g\_flatshadedObject, będzie generował wektory normalne do wielokątów uzyskując w efekcie cieniowanie gładkie.

Funkcja DisplayScene() wywołuje kilka różnych funkcji, które rysują powierzchnie drugiego stopnia za pomocą omówionych obiektów. Ponieważ funkcje te działają w podobny sposób, przedstawiony tutaj zostanie kod tylko jednej z nich:

```
GLvoid DrawWireframeObjects(GLfloat rotation)
{
    // inicjuje generator pseudolosowy
    srand(300);

    // przechowuje właściwości kolorów
    glPushAttrib(GL_CURRENT_BIT);

    // włącza łączenie kolorów wymagane przez antialiasing
    glEnable(GL_BLEND);

    // rysuje parę "goniących się" kul
    glPushMatrix();
    glMateriali(GL_FRONT_AND_BACK, GL_SHININESS, rand() % 128);
    glColor3f(FRAND, FRAND, FRAND);
    glTranslatef(-20.0f, 2.0f, -20.0f);
    glRotatef(rotation * 2.0f, 1.0f, 0.0f, 0.0f);
    glRotatef(rotation * 2.0f, 0.0f, 1.0f, 0.0f);
    glRotatef(rotation * 2.0f, 0.0f, 0.0f, 1.0f);
    glTranslatef(-0.4f, 0.0f, 0.0f);
    gluSphere(g_wireframeObject, 0.3f, 16, 10);
    glPopMatrix();

    glPushMatrix();
    glMateriali(GL_FRONT_AND_BACK, GL_SHININESS, rand() % 128);
    glColor3f(FRAND, FRAND, FRAND);
    glTranslatef(-20.0f, 2.0f, -20.0f);
    glRotatef(-rotation * 2.0f, 1.0f, 0.0f, 0.0f);
    glRotatef(-rotation * 2.0f, 0.0f, 1.0f, 0.0f);
    glRotatef(-rotation * 2.0f, 0.0f, 0.0f, 1.0f);
    glTranslatef(0.4f, 0.0f, 0.0f);
    gluSphere(g_wireframeObject, 0.3f, 16, 10);
    glPopMatrix();
```

```
// rysuje odwrócony stożek
glPushMatrix();
glMateriali(GL_FRONT_AND_BACK, GL_SHININESS, rand() % 128);
glColor3f(FRAND, FRAND, FRAND);
glTranslatef(-150.0, 0.5, 0.0);
glRotatef(-90, 1.0, 0.0, 0.0);
gluCylinder(g_wireframeObject, 0.0, 0.5, 3.0, 32, 4);
glPopMatrix();

// rysuje wypełniony dysk
glPushMatrix();
glMateriali(GL_FRONT_AND_BACK, GL_SHININESS, rand() % 128);
glColor3f(FRAND, FRAND, FRAND);
glTranslatef(-40.0, 0.5, 20.0);
glRotatef(45.0, 1.0, 0.0, 0.0);
gluDisk(g_wireframeObject, 0.0, 0.5, 32, 4);
glPopMatrix();

glDisable(GL_BLEND);

// przywraca poprzednie kolory
glPopAttrib();
} // DrawWireframeObjects()
```

Jako że obiekty powierzchni zostały już odpowiednio skonfigurowane, na tym etapie wystarczy już tylko określić materiał i kolor powierzchni, jej położenie (za pomocą przesunięć i obrotów) i narysować ją wywołując funkcję gluDisk(), gluCylinder() lub gluSphere().

Gdy program kończy swoje działanie, należy pamiętać o usunięciu obiektów powierzchni:

```
BOOL Cleanup()
{
    // usuwa istniejące obiekty powierzchni
    if (g_normalObject)
        gluDeleteQuadric(g_normalObject);
    if (g_wireframeObject)
        gluDeleteQuadric(g_wireframeObject);
    if (g_texturedObject)
        gluDeleteQuadric(g_texturedObject);
    if (g_flatshadedObject)
        gluDeleteQuadric(g_flatshadedObject);

    return TRUE;
} // Cleanup()
```

Jeśli tylko wskaźniki obiektów są różne od NULL, to zawsze można bezpiecznie usuwać obiekty powierzchni korzystając z funkcji gluDeleteQuadric().

## Podsumowanie

W rozdziale tym przedstawiony został sposób tworzenia, konfigurowania i usuwania obiektów powierzchni drugiego stopnia. Przede wszystkim omówione jednak zostało za- stosowanie tych obiektów do rysowania różnego rodzaju powierzchni.

Powierzchnie, które można rysować za pomocą biblioteki GLU są z pewnością bardzo przydatne, ale nie wyczerpują zagadnienia. Inne biblioteki udostępniają także inne rodzaje powierzchni, które mogą okazać się użyteczne w pewnych zastosowaniach. Na przykład biblioteka GLUT dysponuje funkcjami umożliwiającymi rysowanie sześcianu, torusa, a nawet czajnika! Dlatego też tworząc aplikacje używające obiektów o określonym kształcie, warto sprawdzić, czy nie są one dostępne w gotowych bibliotekach, zanim podejmie się decyzję o ich samodzielnej implementacji.

## Rozdział 14.

# Krzywe i powierzchnie

Wszystkie prezentowane dotąd programy tworzyły grafikę za pomocą wierzchołków, odcinków i wielokątów. Zastosowanie tych podstawowych elementów grafiki jest bardzo proste i przydatne, ale posiada także ograniczenia. Na przykład uzyskanie doskonale gładkiej zakrzywionej powierzchni złożonej z wielokątów nie jest możliwe. Powierzchnię taką można jedynie aproksymować stosując dużą liczbę niewielkich wielokątów. Dlatego też OpenGL umożliwia tworzenie krzywych i powierzchni w inny sposób — za pomocą punktów kontrolnych, evaluatorów i obiektów powierzchni B-sklejanych.

W bieżącym rozdziale przedstawione zostaną:

- ◆ podstawowe pojęcia związane z krzywymi i powierzchniami;
- ◆ evaluatory;
- ◆ pokrywanie powierzchni teksturą.

## Reprezentacja krzywych i powierzchni

*Krzywa* może przybierać dowolne kształty w trójwymiarowej przestrzeni. Posiada ona określony początek i koniec oraz długość. *Powierzchnia* charakteryzuje się dodatkowo szerokością i składa się z wielu krzywych.

Punkty, odcinki i wielokąty posiadają pewną reprezentację matematyczną. Podobnie krzywe i powierzchnie. Zamiast korzystać z tradycyjnych równań, które w przypadku prostej mają postać

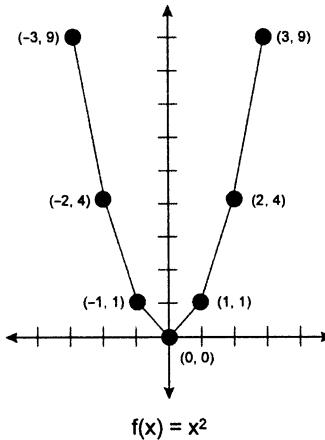
$$y = m \cdot x + b$$

można używać równań *parametrycznych*. Przed analizą równań parametrycznych trzeba przypomnieć sobie najpierw, w jaki sposób należy posługiwać się tradycyjnymi równaniami krzywych.

Aby uzyskać wykres paraboli pokazany na rysunku 14.1, należy użyć równania  $f(x) = x^2$  i wykreślić punkty w pewnym przedziale wartości współrzędnej  $x$ . Równanie to oznacza, że funkcja  $x$  równa jest  $x^2$ . Jeśli więc  $x$  posiada na przykład wartość 2, to wartość funkcji (czyli inaczej  $y$ ) wyniesie 4. Jeśli  $x$  równe jest -5, to wartość funkcji wynosi 25 i tak dalej.

**Rysunek 14.1.**

Wykres funkcji  
 $f(x) = x^2$



## Równania parametryczne

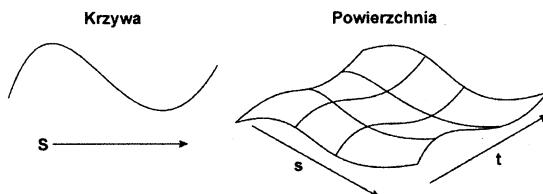
Zamiast wyrażać równanie krzywej za pomocą równania opisującego współrzędną  $y$  jako funkcję współrzędnej  $x$ , można przedstawić obie współrzędne  $x$  i  $y$  jako funkcje innej zmiennej. Na przykład w fizyce często wyraża się współrzędne  $x$  i  $y$  na płaszczyźnie w funkcji czasu. Również przy tworzeniu grafiki trójwymiarowej można zdefiniować współrzędne cząstki poruszającej się w przestrzeni jako funkcje wirtualnego czasu. Współrzędne  $x$ ,  $y$  i  $z$  cząstki będzie się wtedy obliczać za pomocą równania parametrycznego dla bieżącej wartości czasu. Równanie parametryczne może mieć następującą postać:

$$Q(t) = \{ x(t), y(t) \}$$

Zastosowanie parametrycznego równania krzywej nie powinno sprawić kłopotu. Rysowanie krzywej rozpocząć należy od jej punktu początkowego i kontynuować zwiększać wartość parametru aż do osiągnięcia punktu końcowego krzywej. Parametr ten można nazwać  $s$ , a jego dziedziną powinien stać się przedział wartości przyjmowanych dla punktu początkowego i końcowego. W przypadku powierzchni używać należy dwóch parametrów  $s$  i  $t$ , które zmieniać będą się w kierunkach, w których rozciągać się będzie powierzchnia, ale nie będą bezpośrednio reprezentować jej współrzędnych. Rysunek 14.2 ilustruje zastosowanie parametrów  $s$  i  $t$  w celu definicji krzywych i powierzchni.

**Rysunek 14.2.**

Parametryczna  
 reprezentacja  
 krzywych  
 i powierzchni

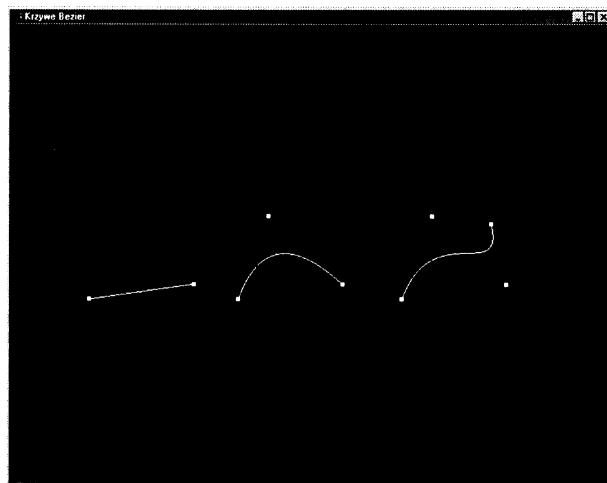


## Punkty kontrolne i pojęcie ciągłości

*Punkty kontrolne* określają kształt krzywej. Można wyobrazić je sobie jako magnesy przyciągające krzywą. Pierwszy i ostatni punkt kontrolny krzywej definiują zawsze jej końce. Pozostałe punkty kontrolne pozwalają zmieniać kształt krzywej. Rysunek 14.3

**Rysunek 14.3.**

Punkty kontrolne pozwalają zmieniać kształt krzywej



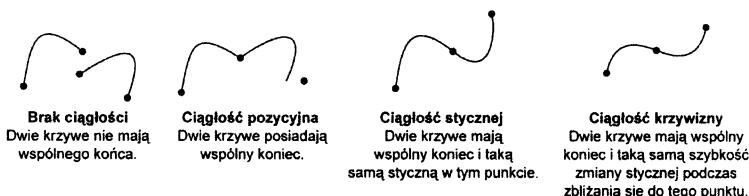
przedstawia trzy krzywe, z których każda posiada inną liczbę punktów kontrolnych. Należy zwrócić uwagę na to, że za każdym razem, gdy dodaje się nowy punkt kontrolny, kształt krzywej ulega zmianie tak, jakby była ona przyciągana przez ten punkt.

Termin ciągłość opisuje sposób połączenia dwóch krzywych, które posiadają wspólny punkt kontrolny definiujący koniec każdej krzywej. Wyróżnia się cztery kategorie ciągłości krzywych (patrz: rysunek 14.4):

- ◆ **brak ciągłości** — dwie krzywe nie mają wspólnego końca (kategoria  $C^0$ );
- ◆ **ciągłość pozycyjna** — dwie krzywe mają wspólny koniec (kategoria  $C^1$ );
- ◆ **ciągłość stycznej** — dwie krzywe mają wspólny koniec i taką samą styczną w tym punkcie (kategoria  $C^2$ );
- ◆ **ciągłość krzywizny** — dwie krzywe mają wspólny koniec i taką samą szybkość zmiany stycznej podczas zbliżania się do tego punktu (kategoria  $C^3$ ).

**Rysunek 14.4.**

Kategorie ciągłości krzywych



## Ewaluator

*Ewaluator* razem z punktami kontrolnymi definiują krzywą lub powierzchnię. Używane są do definiowania krzywych i powierzchni Béziera. Zanim użyty zostanie ewaluator, trzeba najpierw sprawdzić, czy dana krzywa bądź powierzchnia jest krzywą bądź powierzchnią Béziera. Jeśli nie, to najpierw należy poddać ją konwersji. Krzywą Béziera definiuje następujące funkcja parametryczna:

$$C(u) = [ x(u) \ y(u) ]$$

Parametr  $u$  przybiera zwykle wartości z pewnej dziedziny, zwykle z przedziału od 0 do 1. Rozszerzając tę funkcję o parametr  $v$  i zmienną  $z$  uzyskuje się reprezentację powierzchni Béziera:

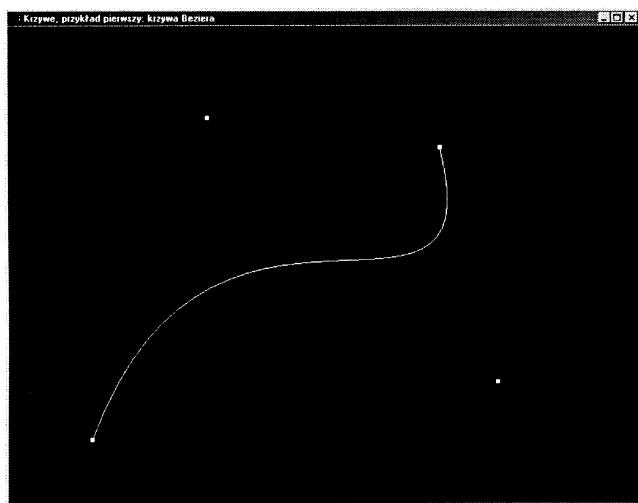
$$S(u, v) = [ x(u, v) \ y(u, v) \ z(u, v) ]$$

Powierzchnie Béziera staną się przedmiotem analizy nieco później, a teraz należy skoncentrować się na parametrze  $u$ . Podczas obliczania wartości  $C(u)$  parametr  $u$  definiuje interwał kolejnych punktów krzywej. Interwał ten określa rozdzielczość krzywej. Na przykład interwał 1/30 sprawi, że uzyskana zostanie niższa jakość krzywej niż w przypadku interwału 1/100.

Należy uważnie przyjrzeć się fragmentowi kodu, który rysuje krzywą pokazaną na rysunku 14.5.

**Rysunek 14.5.**

Przykład krzywej  
Béziera



Najpierw trzeba zdefiniować punkty kontrolne krzywej:

```
float control[4][3] = { { 1.0, -3.0, 0.0 }, { 3.0, 2.5, 0.0 },
{ 8.0, -2.0, 0.0 }, { 7.0, 2.0, 0.0 } };
```

Pierwszy i ostatni punkt kontrolny, czyli (odpowiednio)  $control[0]$  i  $control[3]$  reprezentują końce krzywej. Pozostałe punkty kontrolne,  $control[1]$  i  $control[2]$ , umożliwiają zmianę kształtu krzywej.

Funkcja `Initialize()` wybiera czarny kolor tła oraz cieniowanie płaskie.

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

    glShadeModel(GL_FLAT); // cieniowanie płaskie
}
```

Funkcja Render() rysuje krzywą:

```
void Render()
{
    // opróżnia bufory ekranu i głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    // centruje położenie krzywej
    glTranslatef(-5.0f, 0.0f, -10.0f);

    // tworzy krzywą Béziera
    glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &control[0][0]);
    glEnable(GL_MAP1_VERTEX_3);
    // wybiera kolor biały
    glColor3f(1.0, 1.0, 1.0);
    // rysuje odcinki krzywej korzystając z ewaluatora
    glBegin(GL_LINE_STRIP);
    for (int i = 0; i <= 100; i++)
        glVertex3fv((float)i/100.0f);
    glEnd();
    // rysuje punkty kontrolne krzywej
    glPointSize(3.0);
    glColor3f(1.0, 1.0, 1.0);
    glBegin(GL_POINTS);
    for (i = 0; i < 4; i++)
        glVertex3fv(&control[i][0]);
    glEnd();

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

Najpierw należy skonfigurować dla krzywej Béziera jednowymiarowy ewaluator:

```
glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &control[0][0]);
```

Funkcja glMap1f() definiuje jednowymiarowy ewaluator i posiada następujący prototyp:

```
void glMap1f(GLenum target, float u1, float u2, int stride, int order,
             const float *points);
```

Parametry *u1* i *u2* definiują przedział wartości parametru *u*. Parametr *stride* określa odległość pomiędzy punktami krzywej, a parametr *order* liczbę punktów kontrolnych. Parametr *points* wskazuje dane punktów kontrolnych.

Parametr *target* może przyjmować wartości przedstawione w tabeli 14.1. Określa on format punktów kontrolnych. W tym przykładzie wybrana została wartość *GL\_MAP1\_VERTEX\_3*, która oznacza format opisu punktów kontrolnych za pomocą współrzędnych wierzchołka.

Wybrany format punktów kontrolnych należy dodatkowo aktywować:

```
glEnable(GL_MAP1_VERTEX_3);
```

**Tabela 14.1.** Format punktów kontrolnych

Wartość	Opis formatu
GL_MAP1_VERTEX_3	Współrzędne wierzchołka (x, y, z)
GL_MAP1_VERTEX_4	Współrzędne wierzchołka (x, y, z, w)
GL_MAP1_INDEX	Indeks koloru
GL_MAP1_COLOR_4	Składowe koloru w modelu RGBA
GL_MAP1_NORMAL	Współrzędne wektora normalnego
GL_MAP1_TEXTURE_COORD_1	Współrzędne tekstury jednowymiarowej (s)
GL_MAP1_TEXTURE_COORD_2	Współrzędne tekstury dwuwymiarowej (s, t)
GL_MAP1_TEXTURE_COORD_3	Współrzędne tekstury trójwymiarowej (s, t, r)
GL_MAP1_TEXTURE_COORD_4	Współrzędne tekstury czterowymiarowej (s, t, r, q)

Teraz można już narysować krzywą:

```
glBegin(GL_LINE_STRIP);
    for (int i = 0; i <= 100; i++)
        glEvalCoord1f((float)i/100.0f);
glEnd();
```

Powstanie w ten sposób krzywa Béziera złożona ze 100 odcinków. Kolejne jej punkty wyznacza się w pętli wywołując evaluator. Funkcja `glEvalCoord1f()` zdefiniowana jest następująco:

```
void glEvalCoord1f(float u);
```

Funkcja ta wykorzystuje bieżącą mapę punktów kontrolnych krzywej określona wcześniej za pomocą funkcji `glMap1f()` i aktywowaną za pomocą funkcji `glEnable()`. Parametr `u` przyjmuje kolejne wartości z przedziału od 0.0 do 0.1 rozdzielone ustalonym interwałem.

Po narysowaniu krzywej można jeszcze zaprezentować jej punkty kontrolne. W tym celu definiuje się ją jako wierzchołki za pomocą funkcji `glVertex3f()` oraz nadaje się im wielkość 3.0 dla lepszej widoczności:

```
glPointSize(3.0);
	glColor3f(1.0, 1.0, 1.0);
	glBegin(GL_POINTS);
    for (i = 0; i < 4; i++)
        glVertex3fv(&control[i][0]);
glEnd();
```

I to wszystko! Wykonując ten kod uzyskać można efekt zbliżony do przedstawionego na rysunku 14.5.

## Siatka równoodległa

Rysowanie krzywej można uprościć przez zastosowanie siatki równoodległej. Siatkę taką definiuje funkcja `glMapGrid1f()` o następującym prototypie:

```
void glMapGrid1f(int n, float u1, float u2);
```

Parametr `n` definiuje liczbę oczek siatki, a parametry `u1` i `u2` jej zakres.

Po zdefiniowaniu siatki można narysować krzywą składającą się z punktów lub odcinków za pomocą funkcji `glEvalMesh1()`. Funkcja ta zdefiniowana jest następująco:

```
void glEvalMesh1(GLenum mode, int p1, int p2);
```

W zależności od tego, czy krzywa składać ma się z punktów czy odcinków, parametr `mode` przyjmuje wartość `GL_POINT` lub `GL_LINE`. Parametry `p1` i `p2` definiują zakres rysowania krzywej.

Zobaczmy więc, w jaki sposób funkcje te upraszczają kod poprzedniego przykładu:

```
glMapGrid1f(100, 0.0, 100.0);
glEvalMesh1(GL_LINE, 0, 100);
```

Powyższe dwa wiersze kodu zastępują następujący fragment kodu:

```
glBegin(GL_LINE_STRIP);
for (int i = 0; i <= 100; i++)
    glEvalCoord1f((float)i/100.0f);
glEnd();
```

Zastosowanie tych funkcji upraszcza rysowanie krzywych. Jednak swoją efektywność ujawniają one dopiero w przypadku powierzchni.

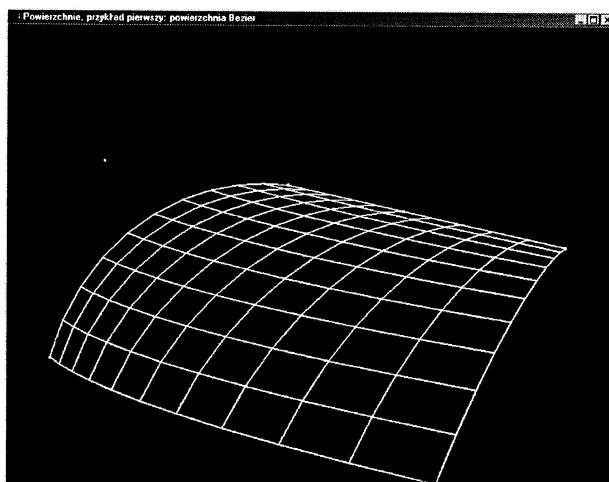
## Powierzchnie

Różnica pomiędzy trójwymiarową powierzchnią i dwuwymiarową krzywą polega na dodaniu parametru  $v$ . W rezultacie zamiast funkcji krzywej postaci  $C(u)$  otrzymuje się funkcję powierzchni w postaci  $S(u, v)$ :

$$S(u, v) = [ x(u, v) \ y(u, v) \ z(u, v) ]$$

Tworzenie powierzchni odbywa się w taki sam sposób jak tworzenie krzywej (z tą różnicą, że istnieje dodatkowy parametr  $v$  i współrzędna z punktów kontrolnych). Rysunek 14.6 przedstawia przykład powierzchni Béziera.

**Rysunek 14.6.**  
Przykład powierzchni  
Béziera



Poniższy fragment kodu ilustruje sposób rysowania powierzchni Béziera:

```

float cSurface[3][3][3] = { { { -200.0, 40.0, 200.0 }, { -100.0, 100.0, 200.0 },
                            { 200.0, 0.0, 200.0 } },
                           { { -240.0, 0.0, 0.0 }, { -150.0, 100.0, 0.0 },
                            { 200.0, 0.0, 0.0 } },
                           { { -200.0, -80.0, -200.0 }, { -100.0, 100.0, -200.0 },
                            { 200.0, 0.0, -200.0 } } };

void Render()
{
    // pominięto fragment kodu określający położenie i orientację kamery

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

    // pominięto fragment kodu określający położenie i orientację kamery

    glColor3f(1.0, 1.0, 1.0);

    // tworzy powierzchnię Beziera i aktywuje ją
    glMap2f(GL_MAP2_VERTEX_3, 0.0, 10.0, 3, 3, 0.0, 10.0, 9, 3, &cSurface[0][0][0]);
    glEnable(GL_MAP2_VERTEX_3);

    // tworzy siatkę równoodległą
    glMapGrid2f(10, 0.0f, 10.0f, 10, 0.0f, 10.0f);
    // i wyznacza z jej pomocą powierzchnię
    glEvalMesh2(GL_LINE, 0, 10, 0, 10);

    // rysuje punkty kontrolne w kolorze żółtym
    glPointSize(3.0);
    glColor3f(1.0, 1.0, 0.0);
    glBegin(GL_POINTS);
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            glVertex3fv(&cSurface[1][j][0]);
    glEnd();
    glPointSize(1.0);

    glFlush();
    SwapBuffers(g_HDC);           // przełącza bufory
}

```

Funkcja `Render()` definiuje najpierw trzy zbiory punktów kontrolnych, z których każdy zawiera trzy punkty. W istocie definiują one trzy krzywe Béziera zależne dla parametru `u` połączone parametrem `v` w celu uzyskania powierzchni.

Funkcja `Render()` wywołuje funkcję `glMap2f()` zdefiniowaną w następujący sposób:

```
void glMap2f(GLenum target, float u1, float u2, int ustride, int uorder,
             float v1, float v2, int vstride, int vorder, float points);
```

Funkcja ta stanowi wersję omówionej wcześniej funkcji `glMap1f()` rozszerzoną o parametr `v`. Parametr `target` może przyjmować wartości przedstawione w tabeli 14.1 (z tą różnicą, że ich nazwa zawiera ciąg MAP2, a nie MAP1).

Po utworzeniu powierzchni Béziera za pomocą funkcji `glMap2f()` należy ją aktywować za pomocą funkcji  `glEnable()`:

```
 glEnable(GL_MAP2_VERTEX_3);
```

Funkcja `glMapGrid2()` pozwala zdefiniować dwuwymiarową siatkę wykorzystywaną przez ewaluatory. Podobnie jak w przypadku funkcji `glMap2f()`, także funkcja `glMapGrid2()` stanowi rozszerzoną wersję funkcji `glMapGrid1f()`:

```
 void glMapGrid2f(int nu, float u1, float u2, int nv, float v1, float v2);
```

Za pomocą funkcji `glEvalMesh2()` wywołuje się ewaluatory dla zdefiniowanej wcześniej siatki. Funkcja `glEvalMesh2()` posiada następujący prototyp:

```
 void glEvalMesh2(GLenum mode, int i1, int i2, int j1, int j2);
```

Parametr `mode` może przyjmować wartość `GL_POINT`, `GL_LINE` lub `GL_FILL`.

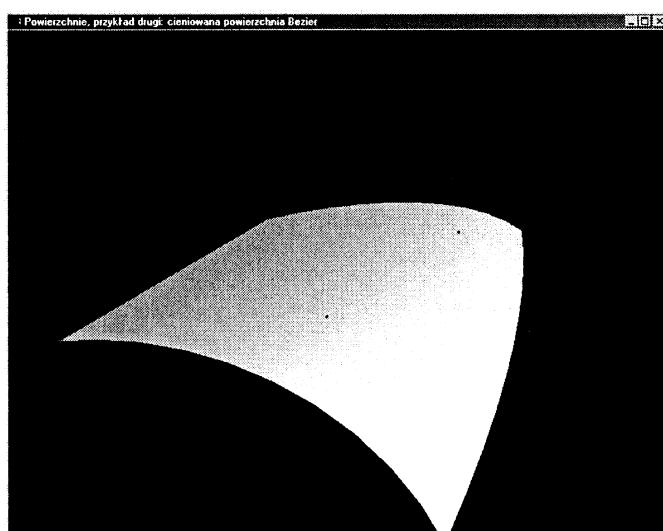
Jeśli parametrowi `mode` nadana zostanie wartość `GL_FILL`, to OpenGL wypełni powierzchnię pojedynczym kolorem. W celu uzyskania bardziej realistycznego wyglądu powierzchni można wykorzystać oświetlenie i cieniowanie. Aby OpenGL mógł prawidłowo obliczyć oświetlenie powierzchni niezbędne są wektory normalne do tej powierzchni. Z pomocą przychodzi tu następujące wywołanie funkcji  `glEnable()`:

```
 glEnable(GL_AUTO_NORMAL);
```

Po skonfigurowaniu oświetlenia powierzchni wystarczy w programie umieścić powyższy wiersz i automatycznie uzyskane zostanie właściwe oświetlenie i cieniowanie powierzchni. Przykład oświetlonej i cieniowanej powierzchni narysowanej za pomocą funkcji `glEvalMesh2()` prezentuje rysunek 14.7.

**Rysunek 14.7.**

Przykład oświetlonej  
i cieniowanej  
powierzchni Béziera



## Pokrywanie powierzchni teksturami

Ewaluatora można wykorzystać także podczas pokrywania powierzchni teksturami. W tym celu wywołując funkcję `glMap2f()` z parametrem `GL_MAP2_TEXTURE_COORD_2` tworzy się ewaluator wyznaczający współrzędne tekstury dla danej powierzchni. Poniższy fragment kodu pokrywa teksturą szachownicy powierzchnię znaną z poprzednich przykładów:

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

////// Pliki nagłówkowe ...

////// 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
float angle = 0.0f;             // kąt kamery
float radians = 0.0f;            // kąt kamery w radianach

float cSurface[3][3][3] = { { { -200.0, 40.0, 200.0 }, { -100.0, 100.0, 200.0 },
                            { 200.0, 0.0, 200.0 } },
                           { { -240.0, 0.0, 0.0 }, { -150.0, 100.0, 0.0 },
                             { 200.0, 0.0, 0.0 } },
                           { { -200.0, -80.0, -200.0 }, { -100.0, 100.0, -200.0 },
                             { 200.0, 0.0, -200.0 } } };

float sTexCoords[2][2][2] = { {{0.0, 0.0}, {0.0, 1.0}}, {{1.0, 0.0}, {1.0, 1.0}} };

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

////// Tekstury
texture_t *surfaceTex;

// LoadBitmapFile
// ...

// LoadTextureFile()
// opis: ładuje teksturę z pliku
texture_t *LoadTextureFile(char *filename)
{
    BITMAPINFOHEADER texInfo;
    texture_t *thisTexture;

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

```
// ładuje dane tekstury i sprawdza poprawność operacji
thisTexture->data = LoadBitmapFile(filename, &texInfo);
if (thisTexture->data == NULL)
{
    free(thisTexture);
    return NULL;
}

// określa rozmiary tekstury
thisTexture->width = texInfo.biWidth;
thisTexture->height = texInfo.biHeight;

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

return thisTexture;
}

// LoadAllTextures()
// opis: ładuje tekstury używane w programie
bool LoadAllTextures()
{
    // ładuje teksturę szachownicy
    surfaceTex = LoadTextureFile("chess.bmp");
    if (surfaceTex == NULL)
        return false;

    // konfiguruje teksturę szachownicy
    glBindTexture(GL_TEXTURE_2D, surfaceTex->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, surfaceTex->width, surfaceTex->height,
                      GL_RGB, GL_UNSIGNED_BYTE, surfaceTex->data);

    return true;
}

// CleanUp()
// opis: zwalnia obiekty
void CleanUp()
{
    free(surfaceTex);
}

// Initialize
// opis: inicjuje OpenGL
void Initialize()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // tło w kolorze czarnym
    glShadeModel(GL_FLAT); // cieniowanie gładkie
    glEnable(GL_DEPTH_TEST); // usuwanie przesłoniętych powierzchni
    glEnable(GL_TEXTURE_2D); // włącza tekstury dwuwymiarowe

    LoadAllTextures(); // ładuje tekstury
}
```

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

    // wyznacza współrzędne kamery
    cameraX = lookX + (float)sin(radians)*mouseY;
    cameraZ = lookZ + (float)cos(radians)*mouseY;
    cameraY = lookY + mouseY / 2.0f + 30.0f;

    // punkt wycelowania kamery
    lookX = -20.0f;
    lookY = 20.0f;
    lookZ = 0.0f;

    // opróżnia bufory 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);

    // tworzy evaluator punktów kontrolnych powierzchni
    glMap2f(GL_MAP2_VERTEX_3, 0.0, 1.0, 3, 3, 0.0, 1.0, 9, 3, &cSurface[0][0][0]);
    // tworzy evaluator współrzędnych tekstury
    glMap2f(GL_MAP2_TEXTURE_COORD_2, 0, 1, 2, 2, 0, 1, 4, 2, &sTexCoords[0][0][0]);
    // aktywuje utworzone evaluatory
    glEnable(GL_MAP2_TEXTURE_COORD_2);
    glEnable(GL_MAP2_VERTEX_3);

    // tworzy siatkę powierzchni
    glMapGrid2f(10, 0.0f, 1.0f, 10, 0.0f, 1.0f);
    glEvalMesh2(GL_FILL, 0, 10, 0, 10);

    // wyłącza tekstury podczas rysowania punktów kontrolnych
    glDisable(GL_TEXTURE_2D);

    // rysuje punkty kontrolne w kolorze żółtym i rozmiarze 4
    glPointSize(4.0);
    glColor3f(1.0, 1.0, 0.0);
    glBegin(GL_POINTS);
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                glVertex3fv(&cSurface[i][j][0]);
    glEnd();

    // przywraca domyślne rozmiary punktów i włącza tekstury
    glPointSize(1.0);
    glEnable(GL_TEXTURE_2D);

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

Szczególnie interesująca jest dla nas poniższa część kodu:

```
// tworzy evaluator punktów kontrolnych powierzchni  
glMap2f(GL_MAP2_VERTEX_3, 0.0, 1.0, 3, 3, 0.0, 1.0, 9, 3, &cSurface[0][0][0]);  
  
// tworzy evaluator współrzędnych tekstuury  
glMap2f(GL_MAP2_TEXTURE_COORD_2, 0, 1, 2, 2, 0, 1, 4, 2, &sTexCoords[0][0][0]);  
  
// aktywuje utworzone ewaluatory  
glEnable(GL_MAP2_TEXTURE_COORD_2);  
glEnable(GL_MAP2_VERTEX_3);  
  
// tworzy siatkę powierzchni  
glMapGrid2f(10, 0.0f, 1.0f, 10, 0.0f, 1.0f);  
glEvalMesh2(GL_FILL, 0, 10, 0, 10);
```

Najpierw — za pomocą funkcji `glMap2f()` — tworzy on ewaluatory punktów kontrolnych powierzchni oraz współrzędnych tekstuury. Współrzędne tekstuury określone są za pomocą tablicy `sTexCoords`:

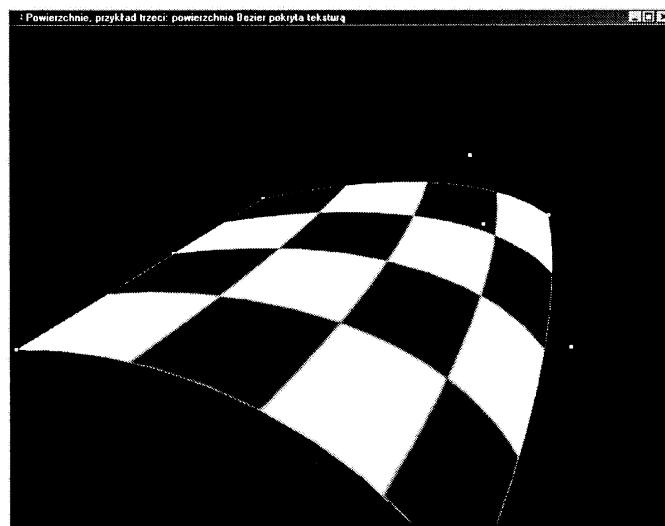
```
float sTexCoords[2][2][2] = {{{0.0, 0.0}, {0.0, 1.0}}, {{1.0, 0.0}, {1.0, 1.0}}};
```

Definiuje ona kwadrat o wierzchołkach w punktach  $(0, 0)$ ,  $(0, 1)$ ,  $(1, 0)$ ,  $(1, 1)$ . Punkty te odpowiadają bezpośrednio narożnikom tekstuury, ponieważ tak zostały skonfigurowane za pomocą funkcji `glMap2f()`. Za każdym razem, gdy przy rysowaniu powierzchni wywołana zostanie funkcja `glEvalMesh2()`, OpenGL automatycznie wyznaczy też współrzędne tekstuury.

Rysunek 14.8 przedstawia tworzoną przez powyższy kod powierzchnię pokrytą teksturą.

**Rysunek 14.8.**

Powierzchnia pokryta teksturą



# Powierzchnie B-sklejane

Wraz ze wzrostem liczby punktów kontrolnych krzywej Béziera wzrasta także trudność związana z uzyskaniem ciągły i gładkiej krzywej. Krzywe Béziera klasyfikowane są na podstawie liczby punktów kontrolnych. Na przykład krzywe posiadające trzy punkty kontrolne nazywane są krzywymi drugiego stopnia, a krzywe o czterech punktach kontrolnych krzywymi trzeciego stopnia. Krzywe posiadające pięć, sześć, siedem i więcej punktów kontrolnych zaczynają tracić swoją gładkość na skutek oddziaływania tak wielu punktów kontrolnych.

Problem ten rozwiązują krzywe B-sklejane nazywane także w terminologii OpenGL krzywymi NURBS (*non-uniform rational B-splines*). Krzywe B-sklejane zachowują się tak samo jak krzywe Béziera. Ich przewaga polega na tym, że skomplikowana krzywa jest dzielona na segmenty, które posiadają cztery punkty kontrolne i są odpowiednikiem krzywych Béziera trzeciego stopnia.

Nie trzeba tutaj omawiać szczegółowo teorii krzywych B-sklejanych, gdyż przedstawione zostaną funkcje, za pomocą których można je rysować.

Węzeł krzywej B-sklejanej opisany jest za pomocą sekwencji wartości określających jego wpływ na segmenty krzywej. Jest to zasadniczy element, który odróżnia krzywe B-sklejane od zwykłych krzywych Béziera.

Każdy punkt kontrolny krzywej B-sklejanej dysponuje dwoma węzłami, które mogą przybierać dowolne wartości z dziedzin parametrów  $u$  i  $v$ . Dla segmentu krzywej posiadającego cztery punkty kontrolne istnieć więc będzie osiem węzłów. Ilustruje to poniższy fragment programu:

```
GLUnurbsObj *myNurb;  
float knots[8] = { 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0 };  
float nurb[4][4][3];  
  
void CleanUp()  
{  
    gluDeleteNurbsRenderer(myNurb);  
}  
  
// Initialize  
// opis: inicjuje OpenGL  
void Initialize()  
{  
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // tło w kolorze czarnym  
  
    glEnable(GL_DEPTH_TEST); // usuwanie przesłoniętych powierzchni  
    glEnable(GL_LIGHTING); // włącza oświetlenie  
    glEnable(GL_LIGHT0); // włącza źródło światła light0  
    glEnable(GL_COLOR_MATERIAL); // kolor jako materiał  
    glEnable(GL_AUTO_NORMAL); // automatyczne tworzenie wektorów normalnych  
    glEnable(GL_NORMALIZE);  
  
    // konfiguruje powierzchnię o kształcie wypukłym  
    int u, v;
```

```
for (u = 0; u < 4; u++)
{
    for (v = 0; v < 4; v++)
    {
        nurb[u][v][0] = 3.0*((float)u - 1.5);
        nurb[u][v][1] = 2.0*((float)v - 1.5);

        if ((u == 1 || u == 2) && (v == 1 || v == 2))
            nurb[u][v][2] = 3.0;
        else
            nurb[u][v][2] = -1.0;
    }
}

// inicjuje obiekt krzywej B-sklejanej
myNurb = gluNewNurbsRenderer();

// określa maksymalną długość w przypadku użycia wielokątów
gluNurbsProperty(myNurb, GLU_SAMPLING_TOLERANCE, 50.0);

// rysuje powierzchnię za pomocą wielokątów
gluNurbsProperty(myNurb, GLU_DISPLAY_MODE, GLU_FILL);
}

// Render
// opis: rysuje scenę
void Render()
{
    // opróżnia bufory ekranu i głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    // przesuwa i obraca powierzchnię, aby była lepiej widoczna
    glTranslatef(0.0, 0.0, -10.0);
    glRotatef(290.0, 1.0, 0.0, 0.0);

    glPushMatrix();
    glRotatef(angle, 0.0, 0.0, 1.0);      // obraca powierzchnię

    glColor3f(0.2, 0.5, 0.8);

    // rozpoczyna rysowanie powierzchni B-sklejanej
    gluBeginSurface(myNurb);

    // rysuje powierzchnię B-sklejaną
    gluNurbsSurface(myNurb, 8, knots, 8, knots, 4*3, 3, &nurb[0][0][0],
                    4, 4, GL_MAP2_VERTEX_3);

    // kończy rysowanie powierzchni B-sklejanej
    gluEndSurface(myNurb);

    // rysuje punkty kontrolne
    glPointSize(6.0);
    glColor3f(1.0, 1.0, 0.0);
    glBegin(GL_POINTS);
        for (int i = 0; i < 4; i++)
            for (int j = 0; j < 4; j++)

```

```

glVertex3fv(&nurb[i][j][0]);
glEnd();
glPointSize(1.0);

glPopMatrix();

glFlush();
SwapBuffers(g_HDC); // przełącza bufory

angle+=0.3f; // zwiększa kąt obrotu
}

```

Jak pokazuje powyższy przykład, narysowanie powierzchni B-sklejanej nie wymaga zbyt długiego kodu. Biblioteka GLU udostępnia odpowiedni zestaw funkcji wygodniejszy nawet w użyciu niż przedstawione wcześniej funkcje rysowania powierzchni Béziera.

Najpierw należy zadeklarować wskaźnik obiektu powierzchni B-sklejanej, którym będzie można się posługiwać podczas jej definiowania i rysowania:

```
GLUnurbsObj *myNurb; // obiekt powierzchni B-sklejanej
```

Trzeba także zdefiniować węzły punktów kontrolnych oraz tablicę, w której umieszczone zostaną punkty kontrolne:

```
float knots[8] = { 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0 };
float nurb[4][4][3];
```

Funkcja Initialize() włącza oświetlenie i instruuje maszynę OpenGL, by wyznaczała wektory normalne. Punkty kontrolne powierzchni należy zadeklarować przed rozpoczęciem jej definiowania:

```
// inicjuje obiekt krzywej B-sklejanej
myNurb = gluNewNurbsRenderer();
// określa maksymalną długość w przypadku użycia wielokątów
gluNurbsProperty(myNurb, GLU_SAMPLING_TOLERANCE, 50.0);
// rysuje powierzchnię za pomocą wielokątów
gluNurbsProperty(myNurb, GLU_DISPLAY_MODE, GLU_FILL);
```

Funkcja gluNewNurbsRenderer() tworzy obiekt powierzchni B-sklejanej.

Funkcję gluNurbsProperty() wykorzystuje się w celu określenia tolerancji próbkowania oraz sposobu rysowania powierzchni B-sklejanej. Parametr GLU\_SAMPLING\_TOLERANCE określa maksymalną długość w przypadku wielokątów. W tym przykładzie jest to 50 jednostek. Parametr GLU\_DISPLAY\_MODE określa sposób rysowania powierzchni B-sklejanej i może przyjmować jedną z trzech wartości:

- ◆ GLU\_FILL — powierzchnia będzie rysowana za pomocą wielokątów;
- ◆ GLU\_OUTLINE\_POLYGON;
- ◆ GLU\_OUTLINE\_PATCH.

Poniżej zaprezentowane zostały prototypy obu omówionych funkcji:

```
GLUnurbsObj* gluNewNurbsRenderer(void);
void gluNurbsProperty(GLUnurbsObj *nobj, GLenum property, float value);
```

Aby narysować powierzchnię B-sklejaną wystarczą jedynie trzy wywołania funkcji:

```
// rozpoczyna rysowanie powierzchni B-sklejanej  
gluBeginSurface(myNurb);  
  
// rysuje powierzchnię B-sklejaną  
gluNurbsSurface(myNurb, 8, knots, 8, knots, 4*3, 3, &nurb[0][0][0], 4, 4,  
GL_MAP2_VERTEX_3);  
  
// kończy rysowanie powierzchni B-sklejanej  
gluEndSurface(myNurb);
```

Funkcja `gluBeginSurface()` zapowiada rysowanie powierzchni B-sklejanej za pomocą przekazanego jej obiektu powierzchni. Funkcja ta posiada następujący prototyp:

```
void gluBeginSurface(GLUnurbsObj *nobj);
```

Funkcja `gluNurbsSurface()` wyznacza wierzchołki powierzchni niezbędne do jej prawidłowego narysowania. Zdefiniowana jest w następujący sposób:

```
void gluNurbsSurface(GLUnurbsObj *nobj, int uknot_count, float *uknot,  
int vknot_count, float *vknot, int u_stride, int v_stride,  
float *ctlarray, int uorder, int vorder, GLenum type);
```

Pierwszy z parametrów funkcji, `nobj`, wskazuje obiekt powierzchni B-sklejanej. Parametry `uknot_count` i `uknot` określają liczbę węzłów i ich dane w kierunku u. Podobnie parametry `vknot_count` i `vknot` określają liczbę węzłów i ich dane w kierunku v. Parametry `u_stride` i `v_stride` określają odległości pomiędzy punktami kontrolnymi w kierunku u i v. Dane punktów kontrolnych są przekazywane w tablicy wskazywanej przez parametr `ctlarray`. Parametry `uorder` i `vorder` określają stopień wielomianu płaszczyzny w kierunkach u i v, który zwykle równy jest liczbie punktów kontrolnych w danym kierunku. Parametr `type` specyfikuje typ rysowanej powierzchni B-sklejanej. W tym przykładzie przyjmuje on wartość `GL_MAP2_VERTEX_3`.

Po wyznaczeniu wierzchołków powierzchni wywołanie funkcji `gluEndSurface()` informuje OpenGL, że proces rysowania powierzchni został zakończony. Funkcji tej przekazuje się obiekt narysowanej powierzchni. Funkcja `gluEndSurface()` posiada więc następujący prototyp:

```
void gluEndSurface(GLUnurbsObj *nobj);
```

Kończąc działanie programu należy zawsze usunąć wykorzystywane obiekty powierzchni B-sklejanych. Służy do tego funkcja `gluDeleteNurbsRenderer()` zdefiniowana w przedstawiony poniżej sposób:

```
gluDeleteNurbsRenderer(GLUnurbsObj *nobj);
```

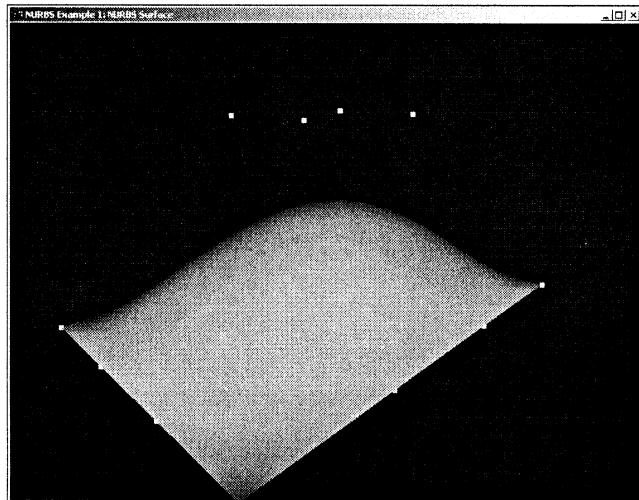
W tym przykładzie wywołuje się ją w następujący sposób:

```
gluDeleteNurbsRenderer(myNurb);
```

Rezultat działania programu przedstawia rysunek 14.9. Na tym można zakończyć omówienie podstaw korzystania z powierzchni B-sklejanych.

**Rysunek 14.9.**

Przykład powierzchni  
B-sklejanej



## Podsumowanie

Krzywa może przyjmować dowolne kształty w przestrzeni. Podobnie jak odcinek prostej posiada on początek i koniec oraz długość. Powierzchnia posiada dodatkowo szerokość i składa się z krzywych.

Równania parametryczne wyrażają współrzędne  $x$  i  $y$  za pomocą funkcji pewnej zmiennej. W fizyce często wyraża się współrzędne punktu na płaszczyźnie w funkcji czasu. Przy tworzeniu grafiki trójwymiarowej można zastosować równania parametryczne do określenia współrzędnych cząstki w funkcji wirtualnego czasu.

Punkty kontrolne określają kształt krzywej. W swoim działaniu przypominają magnesy przyciągające fragmenty krzywej. Pierwszy i ostatni punkt kontrolny określają zawsze początek i koniec krzywej. Pozostałe punkty kontrolne umożliwiają zmianę kształtu krzywej.

Krzywe B-sklejane są podobne do krzywych Béziera. Krzywe B-sklejane składają się z segmentów, z których każdy posiada cztery punkty kontrolne. Podział na segmenty, które są w istocie krzywymi Béziera trzeciego stopnia, umożliwia uzyskanie gładkiej krzywej o skomplikowanych kształtach.

## Rozdział 15.

# Efekty specjalne

W poprzednich rozdziałach omówiono szereg podstawowych technik graficznych umożliwiających stworzenie wirtualnego świata gry. W bieżącym rozdziale przedstawionych zostanie kilka sposobów jego uatrakcyjnienia za pomocą efektów specjalnych.

Efekty specjalne są dość rozległym zagadnieniem i dlatego trzeba ograniczyć się do omówienia jedynie najczęściej stosowanych ich rodzajów. W rozdziale tym przedstawione zostanie:

- ◆ plakatowanie jako sposób zmniejszenia liczby wielokątów;
- ◆ tworzenie systemów cząstek w celu uzyskania wielu różnych efektów począwszy od fontann, a skończywszy na eksplozjach;
- ◆ zastosowanie mgły jako efektu specjalnego i sposobu pozwalającego na zmniejszenie złożoności sceny;
- ◆ tworzenie odbić na płaskich powierzchniach;
- ◆ kilka technik tworzenia cieni.

Większość efektów specjalnych można osiągnąć na wiele sposobów różniących się efektywnością oraz jakością uzyskanego efektu. Ponieważ książka poświęcona jest programowaniu gier, to w rozdziale tym skoncentrować się będzie trzeba na technikach umożliwiających uzyskanie odpowiedniej szybkości tworzenia efektów specjalnych.

## Plakatowanie

Plakatowanie nie jest szczególnie widowiskowym efektem specjalnym, ale stosowane bywa w połączeniu z systemami cząstek i dlatego należy omówić je jako pierwsze. Podstawowym zadaniem *plakatowania* jest zapewnienie, że wielokąt będzie zawsze zwrócony w kierunku obserwatora. Zastosowanie plakatowania omówione zostanie na przykładzie.

Można założyć, że scena tworzona przez gry zawiera drzewa, których model ładowany jest z pliku lub tworzony proceduralnie. Gdy jednak obserwator znajdzie się w bardzo dużej odległości od drzew, to aby przedstawić każde z nich, przetwarzać będzie trzeba nadal dziesiątki, jeśli nie setki wielokątów. Tymczasem ich reprezentację na ekranie stanowić

będzie tylko kilka pikseli. Rozwiążanie tego problemu polegać może na stosowaniu modeli, które wraz ze wzrostem odległości od obserwatora zawierać będą coraz mniej wielokątów. Inny często stosowany sposób polega na zastąpieniu trójwymiarowego modelu odległego obiektu za pomocą dwuwymiarowego obrazu umieszczonego w formie tekstury na pojedynczym wielokącie. Gdy obserwator zbliży się do takiego obiektu, dwuwymiarowy obraz może z powrotem zostać zastąpiony trójwymiarowym modelem.

Można założyć teraz, że obserwator spogląda w kierunku północnym na dwuwymiarowy obraz obiektu umieszczony na wielokącie ustawionym w płaszczyźnie wschód-zachód. Jeśli obserwator zacznie okrązać ten obiekt na przykład w kierunku wschodnim, to kąt obserwacji zacznie się zmniejszać, a obraz obiektu zwężać. Gdy obserwator osiągnie kierunek na wschód od obiektu, to jego obraz przestanie być widoczny.

Sytuacja taka zdarza się w praktyce bardzo często, gdy używa się obiektów dwuwymiarowych w celu reprezentacji obiektów trójwymiarowych. Systemy częstek, które omówione zostaną wkrótce, wykorzystują pojedyncze czworokąty pokryte tekstem do reprezentacji trójwymiarowych częstek. Również starsze gry tworzące trójwymiarową grafikę, jak na przykład *Doom*, *Duke Nukem' 3D* czy *Daggerfall*, często korzystały z dwuwymiarowych obrazów w celu reprezentacji postaci gry. Aby zachować wrażenie ich trójwymiarowości stosuje się właśnie plakatowanie, które zapewnia, że wielokąt pokryty tekstem jest zawsze skierowany w stronę obserwatora.

Teraz należy zastanowić się nad sposobem implementacji plakatowania. Istnieje wiele sposobów na uzyskanie tego efektu. Wybrany tu został jeden z prostszych, który wymaga tylko kilku wierszy kodu, a przede wszystkim najlepiej nadaje się do zastosowania w systemach częstek, które wkrótce będą tworzone. Wielokąt plakatu umieścić należy w taki sposób, aby jego wektor normalny posiadał kierunek zgodny z kierunkiem widzenia obserwatora, ale przeciwny zwrot. W tym celu wystarczy jedynie odwrócić przekształcenia opisane przez macierz modelowania.

Najpierw trzeba więc pobrać zawartość macierzy modelowania:

```
GLfloat viewMatrix[16];  
glGetFloatv(GL_MODELVIEW_MATRIX, viewMatrix);
```

Następnie powinno się wyznaczyć macierz odwrotną do macierzy modelowania. Jak się jednak okazuje, potrzebne informacje można uzyskać szybciej.

Wystarczy zauważyc, że interesująca jest dla programisty jedynie część macierzy o wymiarach  $3 \times 3$ . Jako że macierz  $3 \times 3$  powstała przez odrzucenie ostatniej kolumny i ostatniego wiersza macierzy modelowania, jest także macierzą ortogonalną, więc wyznaczenie jej macierzy odwrotnej sprowadza się do transpozycji. Po wykonaniu transpozycji dwa pierwsze wiersze reprezentować będą wektory ortogonalne do kierunku widzenia obserwatora zwrócone w góre i w prawo. Ponieważ wiersze macierzy odwrotnej są równoważne z kolumnami macierzy przed operacją transpozycji, to operację tę można pominać. Ilustruje to rysunek 15.1.

**Rysunek 15.1.**  
*Transpozycja  
 macierzy  
 modelowania*

$$\begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix}
 \begin{array}{l} \text{wektor right} \rightarrow \\ \text{wektor up} \rightarrow \end{array}
 \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}
 \begin{array}{l} \\ \\ \text{macierz } M^T \\ \text{(transpozycja macierzy } M) \end{array}$$

Wspomniane wektory pobiera się więc bezpośrednio z macierzy modelowania:

```
vector3_t right(viewMatrix[0], viewMatrix[4], viewMatrix[8]);
vector3_t up(viewMatrix[1], viewMatrix[5], viewMatrix[9]);
```

Typ `vector3_t` jest prostą klasą wektorów dysponującą przeciążonymi operatorami dla większości operacji na wektorach i zdefiniowaną w pliku nagłówkowym `vectorlib.h` umieszczonym na płycie CD.

Plakat będzie tworzony wokół określonego punktu przestrzeni. Punkt ten będzie środkiem czworokąta pokrytego teksturą. Wierzchołki czworokąta wyznaczyć można skalując odpowiednio oba wektory, a następnie dodając je. Ilustruje to poniższa formuła:

```
newPoint = centerPoint + up * heightScale + right * widthScale;
```

Dla punktów leżących na lewo od środka czworokąta współczynnik `widthScale` powinien być ujemny, podobnie współczynnik `heightScale` dla punktów leżących poniżej środka. Jeśli rysowany czworokąt jest — tak jak w tym przykładzie — kwadratem, to wystarczy tylko jedna wartość współczynnika skalowania. Nadal jednak trzeba pamiętać, aby wartość współczynnika była ujemna dla punktów leżących na lewo lub poniżej środka wielokąta.

Poniżej przedstawiony został fragment kodu tworzący kwadrat plakatu. Zmienna `point` reprezentuje środek kwadratu, a `size` połowę długości jego boku:

```
// dolny, lewy wierzchołek
glTexCoord2f(0.0, 0.0); glVertex3fv((point + (right + up) * -size).v);
// dolny, prawy wierzchołek
glTexCoord2f(1.0, 0.0); glVertex3fv((point + (right - up) * size).v);
// górny, prawy wierzchołek
glTexCoord2f(1.0, 1.0); glVertex3fv((point + (right + up) * size).v);
// górny, lewy wierzchołek
glTexCoord2f(0.0, 1.0); glVertex3fv((point + (up - right) * size).v);
```

## Przykład: kaktusy na pustyni

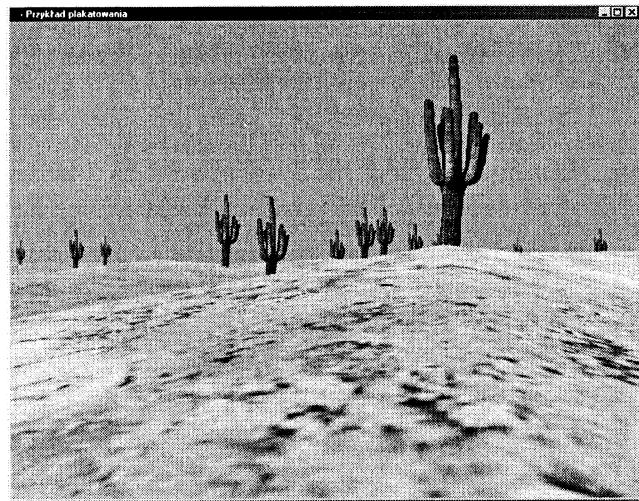
Ponieważ typowym przykładem zastosowania plakatowania jest teren porośnięty drzewami, można pokusić się o oryginalność i stworzyć scenę przedstawiającą kaktusy rosnące na pustyni, co pokazuje rysunek 15.2.

Najważniejszy fragment tworzącego scenę programu to funkcja `DrawCacti()`:

```
void DrawCacti()
{
    // inicjuje generator pseudolosowy
    srand(100);
```

**Rysunek 15.2.**

*Kaktusy na pustyni  
jako przykład  
techniki  
plakatowania*



```
// przezroczysta część tekstury nie będzie rysowana
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_ALPHA_TEST);
glAlphaFunc(GL_GREATER, 0);

// pobiera macierz modelowania
float mat[16];
glGetFloatv(GL_MODELVIEW_MATRIX, mat);

// tworzy wektory ortogonalne
vector3_t right(mat[0], mat[4], mat[8]);
vector3_t up(mat[1], mat[5], mat[9]);

// wybiera teksturę kaktusa
glBindTexture(GL_TEXTURE_2D, g_cactus);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

// rysuje wszystkie kaktusy
glBegin(GL_QUADS);
for (int n = 0; n < NUM_CACTI; n++)
{
    // losowa wielkość kaktusa
    float size = 5.0f + FRAND + 3.0f;

    // losowe położenie na mapie
    vector3_t pos(RAND_COORD((MAP_X - 1) * MAP_SCALE), 0.0,
                  -RAND_COORD((MAP_Z - 1) * MAP_SCALE));
    pos.y = GetHeight(pos.x, pos.z) + size - 0.5f;

    // dolny, lewy wierzchołek
    glTexCoord2f(0.0, 0.0); glVertex3fv((pos + (right + up) * -size).v);
    // dolny, prawy wierzchołek
    glTexCoord2f(1.0, 0.0); glVertex3fv((pos + (right - up) * size).v);
    // górnny, prawy wierzchołek
    glTexCoord2f(1.0, 1.0); glVertex3fv((pos + (right + up) * size).v);
```

```
// górnny, lewy wierzchołek
glTexCoord2f(0.0, 1.0); glVertex3fv((pos + (up - right) * size).v);
}
glEnd();
glDisable(GL_ALPHA);
glDisable(GL_BLEND);
} // DrawCacti()
```

Program umożliwia poruszanie się po krajobrazie za pomocą klawiszy ze strzałkami. Przytrzymanie klawisza *Shift* powoduje zwiększenie prędkości poruszania się.

## Zastosowania systemów cząstek

Zanim będzie można przejść do omówienia szczegółów zastosowań systemów cząstek, trzeba najpierw zdefiniować to pojęcie. Przez *system cząstek* należy rozumieć zbiór pojedynczych elementów zwanych *cząstkami*. Każda z cząstek posiadać może wartości atrybutów odróżniające ją od innych cząstek. Do atrybutów tych należą na przykład prędkość, kolor, czas życia i tak dalej. Każda z cząstek działa autonomicznie, niezależnie od innych cząstek. Cząstki danego systemu posiadają jednak pewien zbiór wspólnych atrybutów. Dzięki temu — mimo że działają niezależnie — tworzą jednak wspólnie jeden efekt. Najprostszy przykład zastosowania systemu cząstek to snop iskier, z których każda reprezentowana jest przez jedną cząstkę. Łącząc zastosowanie systemu cząstek z możliwościami tekstur można tworzyć także realistyczne efekty przedstawiające ogień, dym, eksplozję, przepływ cieczy, opady atmosferyczne, opary, rozgwieźdzone niebo i wiele innych.

Ze względu na samodzielna naturę cząstki stanowią wyjątkowo wdzięczny przedmiot zastosowania programowania obiektowego. Klasy języka C++ modelujące system cząstek można tworzyć na dwa sposoby. Pierwszy polega na stworzeniu klasy bazowej systemu cząstek, z której dziedziczyć będą one specjalizowane klasy reprezentujące różne rodzaje systemów cząstek (na przykład osobne klasy dla symulacji dymu, inne dla ognia i tak dalej). Drugi sposób polega na utworzeniu ogólnej klasy reprezentującej system cząstek. W zależności od zastosowania będzie można zmieniać zestaw atrybutów tej klasy tak, by uzyskać pożądane właściwości systemu cząstek. Wybór sposobu implementacji systemu cząstek zależy od osobistych preferencji programisty. Tu wybrany został pierwszy sposób, ponieważ wydaje się bardziej uniwersalny i przejrzysty.

Zanim wniknie się w szczegóły implementacji systemu cząstek, należy przyjrzeć się najpierw jego elementom i ich atrybutom.

### Cząstki

Omówienie elementów systemu cząstek należy rozpocząć od najbardziej podstawowego elementu, czyli cząstki. Na początku trzeba zdecydować o tym, jakie atrybuty będą charakteryzować cząstkę. Mogą być wśród nich:

- ◆ położenie;
- ◆ prędkość;

- ◆ czas życia;
- ◆ rozmiar;
- ◆ masa;
- ◆ reprezentacja;
- ◆ kolor;
- ◆ przynależność.



Cząstki mogą posiadać także inne atrybuty. Zaprezentowana tutaj lista powinna okazać się wystarczająca w przypadku większości zastosowań.

## Położenie

Aby narysować cząstkę, trzeba znać jej położenie w przestrzeni. Położenie cząstki jest z pewnością jej atrybutem, a nie atrybutem systemu cząstek. Można także przechowywać informację o poprzednich położeniacach cząstki, jeśli zamierza się rysować jej ślad. Położenie cząstki zależy od jej prędkości.



Choć niektóre z atrybutów wydają się przynależeć do cząstek, to będą jednak przechowywane przez system cząstek, ponieważ wszystkie cząstki danego systemu współpracują tworząc jeden, wspólny efekt. Jeśli na przykład wszystkie cząstki danego systemu będą posiadać taką samą masę, to nie ma sensu tworzenie setek kopii tej samej wartości przechowywanych przez poszczególne cząstki. Wystarczy, że wartość tę będzie przechowywał system cząstek.

## Prędkość

W większości zastosowań cząstki poruszają się, dlatego też trzeba określić ich prędkość. Najwygodniej przechować jest informację o prędkości razem z wektorem określającym kierunek ruchu. Dane te posłużą do wyznaczenia nowego położenia cząstki.

Prędkość cząstek może ulegać zmianie na skutek oddziaływania grawitacji bądź oporu powietrza. Wpływ tych czynników omówiony zostanie w dalszej części rozdziału. Ruch cząstki może także charakteryzować się pewnym przyspieszeniem. Zwykle jednak przyczyny zmiany prędkości cząstki będą miały charakter zewnętrzny.

## Czas życia

Cząstki emitowane są przez źródło i po pewnym czasie przestają istnieć. Dlatego też dla każdej cząstki trzeba przechowywać informację o tym, jak długo istnieje dana cząstka, ile jeszcze czasu może istnieć. Czas życia cząstki może mieć wpływ na inne atrybuty, takie jak jej rozmiar, kolor, które mogą zmieniać się z upływem czasu.

## Rozmiar

Rozmiar nie musi być koniecznie atrybutem poszczególnych cząstek. Jeśli rozmiar cząstki nie zmienia się w ciągu jej życia lub rozmiary cząstek emitowanych przez źródło nie różnią się, to nie ma takiej potrzeby. W praktyce jednak sytuacja, w której rozmiar cząstki zmienia się z czasem występuje dość często. W takim wypadku może okazać się także potrzebny dodatkowy atrybut określający szybkość zmiany rozmiaru cząstki.

## Masa

Podobne uwagi jak te, które są związane z rozmiarem cząstki, można przywołać w stosunku do jej masy. *Masa* cząstki nie jest może zbyt precyzyjnym określeniem tego atrybutu, ale w przybliżeniu oddaje jego znaczenie. Atrybut ten określa wpływ różnych czynników zewnętrznych na zachowanie cząstki.

## Reprezentacja

Aby cząstki tworzyły pewien efekt wizualny, muszą posiadać swoją reprezentację na ekranie. Najczęściej używane są trzy sposoby prezentacji cząstek:

- ◆ **punkty** — reprezentacja cząstek za pomocą punktów okazuje się wystarczająca w przypadku wielu efektów (zwłaszcza, gdy są one obserwowane z większej odległości);
- ◆ **odcinki** — odcinki używane są do reprezentacji cząstek, gdy konieczne jest uzyskanie efektu śladu cząstki (odcinek łączy zwykle bieżącą pozycję cząstki z pozycją poprzednią);
- ◆ **czworokąty pokryte tekstem** — sposób ten oferuje największe możliwości i dlatego stosowany jest najczęściej; cząstka reprezentowana jest za pomocą czworokąta pokrytego tekstem, zwykle o wartości współczynnika alfa mniejszej od 1 (przykładem zastosowania tej metody może być cząstka imitująca iskrę reprezentowana za pomocą czworokąta pokrytego tekstem wyobrażającą iskrę).

W większości przypadków wszystkie cząstki danego systemu posiadają taką samą reprezentację i, jeśli reprezentowane są za pomocą czworokątów, pokryte są taką samą tekstem. Dlatego też najczęściej reprezentacja jest atrybutem systemu cząstek, a nie pojedynczych cząstek.



W przypadku reprezentacji cząstek za pomocą czworokątów zwykle stosuje się plakietowanie.

## Kolor

Jeśli cząstki reprezentowane są za pomocą punktów lub odcinków, to trzeba określić także ich kolor. Także stosując reprezentację cząstek w postaci czworokątów pokrytych tekstem, w niektórych zastosowaniach warto połączyć teksturę z różnymi kolorami cząstek (na przykład snop różnokolorowych iskier). Kolor cząstek może zmieniać się z czasem i wtedy konieczny jest dodatkowy atrybut opisujący szybkość tych zmian.

## Przynależność

Cząstka powinna znać swoją przynależność do określonego systemu cząstek, jeśli korzysta z metod klasy reprezentującej system cząstek.

## Metody

Oprócz atrybutów cząstki mogą posiadać metody. Jak zostanie to pokazane, wystarczy, że klasa cząstki posiada jedną metodę umożliwiającą zmianę atrybutów cząstki. Parametrem tej metody będzie okres czasu, który upłynął od jej poprzedniego wywołania. Pozostałe operacje na cząstkach (takie jak na przykład inicjacja cząstki lub jej usunięcie) wykonywane będą bezpośrednio przez system cząstek.

## Systemy cząstek

Każdy system cząstek dysponuje własnym zbiorem autonomicznych cząstek, które posiadają jednak pewne wspólne atrybuty. Zadaniem systemu jest tworzenie tych cząstek i nadanie im odpowiednich atrybutów w taki sposób, by uzyskać wymagany efekt. System cząstek określa zwykle:

- ◆ listę cząstek;
- ◆ swoje położenie;
- ◆częstość emisji cząstek;
- ◆ oddziaływanie cząstek;
- ◆ zakresy wartości atrybutów cząstek i ich wartości domyślne;
- ◆ własny, bieżący stan;
- ◆ sposób łączenia kolorów cząstek;
- ◆ reprezentację cząstek.



Jeśli system cząstek porusza się, to zwykle posiadać będzie także wektor prędkości. Jest to lepsze rozwiązanie niż arbitralna zmiana położenia systemu cząstek przez aplikację.

## Lista cząstek

Aby system mógł zarządzać cząstkami, musi posiadać ich listę. Dla danego systemu należy także określić największą dozwoloną liczbę posiadanych cząstek.

## Położenie

System cząstek musi posiadać określone położenie w przestrzeni, aby możliwe było wyznaczenie początkowego położenia cząstek. Zwykle położenie systemu cząstek modelowane jest za pomocą pojedynczego punktu w przestrzeni. Jednak nic nie stoi na przeszkodzie, aby system cząstek reprezentowany był na przykład za pomocą prostokąta, a cząstki emitowane były z losowo wybranych punktów tego prostokąta. Takie położenie systemu cząstek będzie przydatne w uzyskaniu efektu śniegu. Prostokąt systemu cząstek umieścić należy wtedy w określonym obszarze nieba, z którego następować będzie opad.

## Częstość emisji

Atrybut ten określa to, jak często system cząstek tworzy nowe cząstki. Jeśli system regularnie emituje cząstki, to musi przechowywać informację dotyczącą czasu, który upłynął od momentu utworzenia poprzedniej cząstki i zerować go podczas emisji kolejnej cząstki.



Częstość emisji powiązana jest z maksymalną liczbą cząstek systemu i czasem ich życia. Jeśli cząstki systemu żyją zbyt długo, to system może zbyt szybko osiągnąć maksymalną liczbę cząstek. W tej sytuacji emisja kolejnej cząstki nie jest możliwa przez pewien okres czasu, co powoduje skokowe działanie efektu. Jeśli nie było ono założone, to należy lepiej dobrą częstotliwość emisji i długość życia cząstek bądź usuwać najdłużej istniejącą cząstkę, gdy zachodzi konieczność utworzenia nowej.



Nie wszystkie systemy cząstek muszą posiadać określona częstotliwość emisji. Na przykład systemy cząstek używane do uzyskania efektu wybuchu emitują wszystkie cząstki za jednym razem.

## Oddziaływanie

Jeśli cząstki będą oddziaływać z otoczeniem na różne sposoby, podniesie to realizm tworzonego przez nie efektu. Wektor reprezentujący oddziaływanie cząstek powinien stanowić raczej atrybut systemu cząstek niż być arbitralnie określany przez aplikację dla wszystkich systemów cząstek.

## Atrybuty cząstek, zakresy ich wartości i wartości domyślne

System cząstek nadaje wartości atrybutom cząstki w momencie jej tworzenia. Wartości atrybutów niektórych cząstek mogą zmieniać się w czasie ich istnienia. System cząstek przechowuje więc wartości domyślne atrybutów cząstek oraz zakresy ich wartości.

Często tworząc cząstki trzeba będzie uzyskać pewne ich zróżnicowanie, aby wszystkie nie wyglądały dokładnie tak samo. W tym celu system cząstek powinien określać dopuszczalne odchylenie wartości atrybutu od wartości domyślnej. Przy tworzeniu cząstki pomnoży się wtedy wartość tego odchylenia przez wartość losową z przedziału od -1 do 1, a wynik doda do domyślnej wartości atrybutu.



Tworząc efekty specjalne nie trzeba niewolniczo modelować rzeczywistego świata. Wystarczy, że efekt będzie prezentował się realistycznie. Dlatego też podobnie jak atrybut masy posiada jedynie umowne znaczenie i nie reprezentuje masy rzeczywistej cząstki, tak i oddziaływanie nie muszą reprezentować wiernie swoich odpowiedników w świecie rzeczywistym. Gdy na przykład modeluje się grawitację, wartość przyspieszenia nie musi wynosić  $9,8 \text{ m/s}^2$ . Ważne, by wpływ przyciągania ziemskiego na cząstki był wystarczająco realistyczny. Rozwijając przykład grawitacji można by zwiększyć przyciąganie ziemskie dla kuli stalowej, a zmniejszyć dla piórka i uzyskać w ten sposób efekt szybszego opadania kuli zgodny z rzeczywistym doświadczeniem. Oczywiście wiadomo, że siła przyciągania nie zależy od rodzaju materiału, z którego wykonany jest obiekt. Symulując oddziaływanie grawitacji połączono w jedną całość dodatkowe czynniki, takie jak na przykład opór powietrza. Jeśli uzyskany model dobrze symuluje rzeczywisty efekt i jest w dodatku prosty, to należy go zastosować.



Niektóre z domyślnych wartości atrybutów cząstek przechowywanych przez system cząstek nie muszą przekładać się wprost na wartości atrybutów poszczególnych cząstek. System cząstek może na przykład przechowywać wektor prędkości cząstek, a każda z cząstek jedynie odchylenie swojej prędkości od domyślnej wartości.

## Stan bieżący

System cząstek może zmieniać swoje działanie z upływem czasu. W systemach cząstek, które będą tworzone w tej książce, zmiana ta polegać będzie na wyłączeniu lub wyłączeniu emisji cząstek. Istnieje wiele sytuacji, w których będzie trzeba wyłączyć emisję cząstek przez system. Typowym przykładem są systemy cząstek symulujące efekt wybuchu, które po krótkiej chwili przestają emitować cząstki. Inne systemy cząstek będą wyłączone najczęściej w sytuacji, gdy znajdą się poza obszarem widzenia obserwatora.

Wyłączając system cząstek trzeba zdecydować także o losie wyemitowanych przez niego cząstek, które jeszcze istnieją. Cząstki te można natychmiast usunąć bądź zezwolić im, by kontynuowały swoje istnienie. Stan systemu cząstek będzie można przechowywać za pomocą odpowiedniej zmiennej.

## Łączenie kolorów

Większość systemów cząstek korzysta z możliwości łączenia kolorów. Sposób zastosowania łączenia kolorów zależy od konkretnego systemu.

## Reprezentacja

W większości przypadków wszystkie cząstki danego systemu będą reprezentowane w ten sam sposób, czyli za pomocą punktów, odcinków bądź czworokątów pokrytych teksturą. Informacje o sposobie reprezentacji cząstek przechowywać będzie system cząstek. Jeśli na przykład cząstki reprezentowane będą za pomocą czworokątów, to system cząstek będzie przechowywać nazwę tekstury.



W tym miejscu trzeba podjąć ważną decyzję związaną z projektem klas reprezentujących cząstki. Można zdecydować się na utworzenie ogólnej klasy cząstek, która umożliwiać będzie tworzenie cząstek reprezentowanych za pomocą punktów, odcinków bądź czworokątów. Można też utworzyć bazową klasę cząstek, na podstawie której będzie można tworzyć wyspecjalizowane klasy cząstek różniące się sposobem ich reprezentacji. Wybór jednej z tych możliwości zależy od preferencji i potrzeb programisty (na przykład od tego, czy zamierza w ogóle korzystać ze wszystkich sposobów reprezentacji, czy może tylko z jednego). W przykładach prezentowanych w tej książce używana będzie bazowa klasa cząstek, a konkretne klasy cząstek tworzone będą za pomocą dziedziczenia. Rozwiążanie takie jest bardziej uniwersalne, gdyż umożliwia tworzenie dodatkowych, alternatywnych sposobów reprezentacji cząstek.

## Metody

System cząstek będą wykonywać różne operacje dostępne w postaci metod klasy systemu cząstek.

- ◆ **Inicjacja.** Aby system cząstek stworzył pożądany efekt wizualny, trzeba go odpowiednio skonfigurować, czyli nadać jego atrybutom właściwe wartości. Ponieważ atrybutów tych może być sporo, to zwykle zamiast przekazywać ich wartości metodzie inicjacji osobno, można umieścić je w pojedynczej strukturze.
- ◆ **Aktualizacja.** Metoda ta, wywoływana w regularnych odstępach czasu, będzie aktualizować cząstki danego systemu oraz decydować, czy należy wyemitować kolejne cząstki.
- ◆ **Rysowanie.** Metoda ta rysować będzie wszystkie istniejące cząstki systemu korzystając z atrybutów poszczególnych cząstek i atrybutów określonych przez system. System cząstek określa też sposób graficznej reprezentacji cząstek.
- ◆ **Ruch.** Jeśli będzie trzeba zmienić położenie systemu cząstek, to potrzebna będzie do tego odpowiednia metoda. W takim przypadku przydatna może okazać się także metoda zwracająca bieżące położenie systemu cząstek.
- ◆ **Zmiana stanu.** Metoda ta zmieniać będzie stan systemu cząstek w opisanych wcześniej sytuacjach. Pomocna może okazać się także metoda pozwalająca określić bieżący stan systemu cząstek.
- ◆ **Oddziaływania.** Oddziaływanie mające wpływ na zachowanie cząstek systemu mogą zmieniać się w czasie i wtedy niezbędną okaże się metoda umożliwiająca zmianę odpowiedniego atrybutu systemu. Także cząstki mogą korzystać z osobnej metody w celu pobrania informacji o oddziaływaniach.

Zestaw metod może oczywiście zmieniać się w zależności od konkretnych potrzeb i zastosowań.

## Menedżer systemów cząstek

Systemy cząstek zarządzają autonomicznymi cząstkami — emitują je, aktualizują i usuwają. Jeśli używa się wielu systemów cząstek, co w praktyce zdarza się bardzo często, to celowe jest rozszerzenie tej struktury o kolejny poziom — menedżera systemów cząstek. Zadanie menedżera systemów cząstek może polegać na zmianie położenia systemów cząstek, modyfikacji oddziaływań na cząstki systemów na skutek zmiany pewnych globalnych czynników (na przykład wiatru), a także na tworzeniu i usuwaniu systemów cząstek na skutek zdarzeń zachodzących w grze bądź znalezienia się poza obszarem widzianym przez obserwatora.

Działanie menedżer systemów cząstek zależy w znacznym stopniu od specyfiki aplikacji i dlatego nie będzie analizowana tutaj jego implementacja.

## Implementacja

Przeanalizowano dotąd zagadnienia związane z projektowaniem systemu cząstek. Są one niezwykle ważne, jeśli zamierza się uzyskać uniwersalne systemy cząstek o dużych możliwościach zastosowań. Pozwalają one stworzyć zestaw bazowych klas implementujących wspólne cechy wszystkich systemów cząstek. Klas tych nie będzie się wykorzystywać do tworzenia efektów wizualnych w grach. Posługują one do stworzenia nowych klas implementujących specyficzne efekty.

Klasa reprezentująca pojedyncze cząstki nie będzie posiadać metod, dlatego można zdefiniować ją za pomocą zwykłej struktury:

```
struct particle_t
{
    vector3_t m_pos;           // bieżące położenie cząstki
    vector3_t m_prevPos;       // poprzednie położenie cząstki
    vector3_t m_velocity;      // kierunek i prędkość ruchu
    vector3_t m_acceleration; // przyspieszenie

    float m_energy;           // czas istnienia cząstki

    float m_size;              // rozmiar cząstki
    float m_sizeDelta;         // zmiana rozmiaru cząstki

    float m_weight;            // wpływ grawitacji na ruch cząstki
    float m_weightDelta;        // zmiana wpływu grawitacji

    float m_color[4];          // bieżący kolor cząstki
    float m_colorDelta[4];      // zmiana koloru cząstki
};
```

Jak łatwo zauważyc, struktura ta nie zawiera wielu omówionych tu atrybutów. Ponieważ będzie ona klasą bazową, to zawiera jedynie te atrybuty, które używane są w większości zastosowań systemów cząstek. Oczywiście można by umieścić w strukturze wszystkie atrybuty, jakie kiedykolwiek będą wykorzystywane, a wtedy nie trzeba byłoby tworzyć nowych klas cząstek. Jednak w praktyce okazałoby się, że w różnych zastosowaniach wiele z tych atrybutów jest niewykorzystywanych. Jeśli pomnoży się niewykorzystane pola struktury przez liczbę systemów i ich cząstek, to uzyska się całkiem sporą ilość zmarnowanej pamięci.

Ponieważ cząstki reprezentuje zwykła struktura, to nie posiadają one własnych metod. I nie ma takiej potrzeby. Atrybuty cząstek będą inicjowane przez system cząstek. Zmiana atrybutów cząstek związana z upływem czasu może być wykonywana przez metodę cząstek, ale rozwiązanie takie będzie mało efektywne. Narzutu związanego z wywoływaniem takiej metody dla setek czy tysiący cząstek można uniknąć, jeśli system cząstek będzie dokonywał bezpośredniej modyfikacji atrybutów cząstek.

A oto implementacja systemu cząstek:

```
class CParticleSystem
{
public:
    CParticleSystem(int maxParticles, vector3_t origin);

    // funkcje abstrakcyjne
    virtual void Update(float elapsedTime) = 0;
    virtual void Render() = 0;

    virtual int Emit(int numParticles);

    virtual void InitializeSystem();
    virtual void KillSystem();
```

```
protected:  
    virtual void InitializeParticle(int index) = 0;  
    particle_t *m_particleList; // lista cząstek należących do systemu  
    int m_maxParticles; // maksymalna liczba cząstek  
    int m_numParticles; // liczba istniejących cząstek  
    vector3_t m_origin; // położenie systemu cząstek  
  
    float m_accumulatedTime; // okres czasu od wyemitowania poprzedniej cząstki  
  
    vector3_t m_force; // oddziaływanie (grawitacja, wiatr, etc.)  
};  
  
// Particles.cpp  
//*********************************************************************  
CParticleSystem::Konstruktor  
  
Inicjuje atrybuty systemu.  
//*********************************************************************  
CParticleSystem::CParticleSystem(int maxParticles, vector3_t origin)  
{  
    m_maxParticles = maxParticles;  
    m_origin = origin;  
    m_particleList = NULL;  
} // CParticleSystem::Konstruktor  
  
//*********************************************************************  
CParticleSystem::Emit()  
  
Tworzy nowe cząstki. Ich liczba określona jest przez parametr metody.  
Atrybuty nowych cząstek uzyskują domyślną wartość zmodyfikowaną o losową  
wielkość odchyłki. Jedyne wartości początkowe atrybutów posiadają losową  
odchyłkę. Wartości końcowe są zawsze takie same, co może wymagać zmiany.  
//*********************************************************************  
int CParticleSystem::Emit(int numParticles)  
{  
    // tworzy numParticles nowych cząstek (jeśli jest miejsce)  
    while (numParticles && (m_numParticles < m_maxParticles))  
    {  
        // inicjuje bieżącą cząstke i zwiększa licznik cząstek  
        InitializeParticle(m_numParticles++);  
        -numParticles;  
    }  
    return numParticles;  
} // CParticleSystem::Emit  
  
//*********************************************************************  
CParticleSystem::InitializeSystem()  
  
Przydziela pamięć dla maksymalnej liczby cząstek systemu  
//*********************************************************************  
void CParticleSystem::InitializeSystem()  
{  
    // jeśli metodę wywołano w celu ponownej inicjacji systemu  
    if (m_particleList)
```

```
{  
    delete[] m_particleList;  
    m_particleList = NULL;  
}  
  
// przydziela pamięć maksymalnej liczbie cząstek  
m_particleList = new particle_t[m_maxParticles];  
  
// zeruje liczbę cząstek  
// i czas, który upłynął od utworzenia poprzedniej cząstki  
m_numParticles = 0;  
m_accumulatedTime = 0.0f;  
} // CParticleSystem::InitializeSystem  
  
*****  
CParticleSystem::KillSystem()  
  
Zatrzymuje emisję cząstek. Jeśli parametr metody ma wartość true,  
to wszystkie istniejące cząstki są usuwane.  
*****  
void CParticleSystem::KillSystem()  
{  
    if (m_particleList)  
    {  
        delete[] m_particleList;  
        m_particleList = NULL;  
    }  
  
    m_numParticles = 0;  
} // CParticleSystem::KillSystem
```

Zaprezentowana klasa jest abstrakcyjna, nie może być więc bezpośrednio użyta do tworzenia systemów cząstek. Stanowi jednak szkielet, który można rozszerzyć na drodze dziedziczenia i stworzyć nowe klasy systemów cząstek. Klasa CParticleSystem() posiada metody umożliwiające inicjację lub usunięcie systemu cząstek, które zarządzają przydziałem pamięci dla maksymalnej liczby cząstek. Cząstki te umieszczone są w tablicy, której elementy o indeksach od 0 do (`m_numParticles - 1`) reprezentują cząstki aktywne, a elementy o indeksach od `m_numParticles` do (`m_maxParticles - 1`) wykorzystywane są przez metodę `Emit()` podczas tworzenia nowych cząstek.

## Tworzenie efektów za pomocą systemów cząstek

Klasę bazową systemu cząstek wykorzystuje się do tworzenia specjalizowanych klas pochodnych używanych już bezpośrednio w celu uzyskania wizualnych efektów specjalnych. Osiagnięcie realistycznie wyglądających efektów wymaga właściwego doboru wartości atrybutów systemu cząstek i sposobu jego działania. Nie istnieje gotowa recepta działania, a osiągnięcie zadowalającego rezultatu zawsze wymaga pewnego nakładu pracy, w tym doboru niektórych atrybutów metodą prób i błędów. Poniżej znajduje się kilka ogólnych wskazówek, które mogą okazać się pomocne podczas tworzenia systemów cząstek.

- ◆ **Analiza efektu.** Większość atrybutów systemu cząstek musi reprezentować w pewien sposób właściwości rzeczywistych zjawisk. Dlatego też nawet pobienna analiza pozwala rozpocząć implementację odpowiedniego systemu cząstek. Na przykład słup dymu wznosi się zwykle pionowo i może być odchylany przez wiatr. Oznacza to, że początkowy wektor prędkości powinien być skierowany pionowo w górę, a wektor oddziaływania powinien być do niego prostopadły. Ponieważ wznosząc się dym staje się coraz rzadszy należy wziąć także pod uwagę zmianę rozmiaru i koloru cząstek z upływem czasu.
- ◆ **Zastosowanie praw fizyki.** Omawiając zachowanie cząstek brano dotychczas pod uwagę najbardziej podstawowe zasady fizyki. Jeśli cząstki mają wiernie symulować zachowanie się ich rzeczywistych odpowiedników, to należy zastosować bardziej złożone modele, co wiąże się z większym nakładem pracy oraz wolniejszym działaniem systemu cząstek. Właściwy model fizyczny systemu cząstek to jednak dopiero połowa sukcesu. Równie doskonały musi być sposób jego wizualnej prezentacji.
- ◆ **Korzystanie z doświadczeń innych.** W sieci Internet można znaleźć szereg przykładów implementacji systemów cząstek. Dla wielu z nich udostępniony został także kod źródłowy. Przy małej ilości czasu przeznaczonej na opracowanie charakterystyki systemu cząstek od podstaw można wzorować się na efektach uzyskanych przez innych. W dodatku A podano kilka adresów stron internetowych zawierających przykłady zastosowania systemów cząstek w celu uzyskania różnych efektów.
- ◆ **Eksperyment.** Na płycie CD dołączonej do tej książki umieszczony został program „Particle Sim” autorstwa Briana Tischlera (używający Direct3D według projektu Richa Bensonia). Umożliwia on obserwację (na bieżąco!) wpływu zmian atrybutów systemu cząstek na tworzony efekt wizualny.

## Przykład: śnieżyca

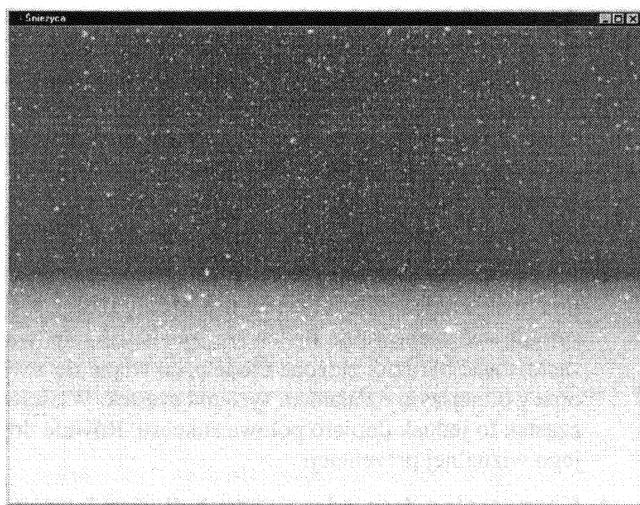
Omówienie systemów cząstek nie byłoby kompletne bez przykładu programu demonstrującego ich zastosowanie. Pełen kod źródłowy takiego programu umieszczony został na płycie CD. Kilku systemów cząstek używa także program omawiany w rozdziale 21.

Przykład, który omówiony zostanie w tym podrozdziale, stosuje system cząstek w celu uzyskania efektu śnieżycy pokazanego na rysunku 15.3. Klasa tego systemu cząstek, CSnowstorm, reprezentuje płatki śniegu za pomocą czworokątów pokrytych tekstyurą i emitowanych z pewnego obszaru nieba.

Poniżej przedstawiona została implementacja klasy CSnowstorm, a pełen kod programu znajduje się na dysku CD.

```
const vector3_t SNOWFLAKE_VELOCITY (0.0f, -3.0f, 0.0f);
const vector3_t VELOCITY_VARIATION (0.2f, 0.5f, 0.2f);
const float     SNOWFLAKE_SIZE      = 0.02f;
const float     SNOWFLAKES_PER_SEC = 2000;
```

**Rysunek 15.3.**  
Śnieżycą nad terenem pokrytym mgłą



```
***** Struktury danych *****
class CSnowstorm : public CParticleSystem
{
public:
    CSnowstorm(int maxParticles, vector3_t origin, float height, float width,
               float depth);

    void Update(float elapsedTime);
    void Render();

    void InitializeSystem();
    void KillSystem();

protected:
    void InitializeParticle(int index);
    float m_height;
    float m_width;
    float m_depth;

    GLuint m_texture; // tekstura płatka śniegu
};

// snowstorm.cpp

*****
CSnowstorm::Konstruktor

Nie wykonyuje żadnych działań.
*****
CSnowstorm::CSnowstorm(int numParticles, vector3_t origin, float height, float width,
                       float depth)
: m_height(height), m_width(width), m_depth(depth),
  CParticleSystem(numParticles, origin)
{
} // CSnowstorm::Konstruktor
```

```
*****
CSnowstorm::InitializeParticle()

Określa początkowe wartości atrybutów cząstek
*****
void CSnowstorm::InitializeParticle(int index)
{
    // umieszcza cząstkę w losowo wybranym punkcie strefy emisji
    m_particleList[index].m_pos.y = m_height;
    m_particleList[index].m_pos.x = m_origin.x + FRAND * m_width;
    m_particleList[index].m_pos.z = m_origin.z + FRAND * m_depth;

    // określa rozmiar cząstki
    m_particleList[index].m_size = SNOWFLAKE_SIZE;

    // i nadaje jej prędkości losowe odchylenie
    m_particleList[index].m_velocity.x = SNOWFLAKE_VELOCITY.x
        + FRAND * VELOCITY_VARIATION.x;
    m_particleList[index].m_velocity.y = SNOWFLAKE_VELOCITY.y
        + FRAND * VELOCITY_VARIATION.y;
    m_particleList[index].m_velocity.z = SNOWFLAKE_VELOCITY.z
        + FRAND * VELOCITY_VARIATION.z;
} // CSnowstorm::InitializeParticle

*****
CSnowstorm::Update

Aktualizuje atrybuty cząstek, usuwa je i tworzy nowe
*****
void CSnowstorm::Update(float elapsedTime)
{
    for (int i = 0; i < m_numParticles; )
    {
        // aktualizuje położenie cząstki na podstawie jej prędkości i czasu, który upłynął
        m_particleList[i].m_pos = m_particleList[i].m_pos
            + m_particleList[i].m_velocity * elapsedTime;

        // jeśli cząstka znalazła się w płaszczyźnie gruntu, usuwa ją
        if (m_particleList[i].m_pos.y <= m_origin.y)
        {
            // przenosi ostatnią z cząstek na miejsce usuniętej i zmniejsza licznik cząstek
            m_particleList[i] = m_particleList[--m_numParticles];
        }
        else
        {
            ++i;
        }
    }

    // zwiększa licznik czasu
    m_accumulatedTime += elapsedTime;

    // określa liczbę nowych cząstek
    int newParticles = SNOWFLAKES_PER_SEC * m_accumulatedTime;
```

```

// zmniejsza licznik czasu po utworzeniu nowych cząstek
m_accumulatedTime -= 1.0f/(float)SNOWFLAKES_PER_SEC * newParticles;

Emit(newParticles);
} // CSnowstorm::Update()

//****************************************************************************
CSnowstorm::Render()

Rysuje płatki śniegu jako czworokąty pokryte teksturową
*****
void CSnowstorm::Render()
{
    // włącza łączenie kolorów i tekstury dwuwymiarowe
    glEnable(GL_BLEND);
    glEnable(GL_TEXTURE_2D);

    // określa tryb łączenia kolorów
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);

    // wybiera teksturę płatka śniegu
    glBindTexture(GL_TEXTURE_2D, m_texture);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

    // zapobiega wielokrotnemu indeksowaniu tablicy
    vector3_t partPos;
    float size;

    // rysuje czworokąty
    glBegin(GL_QUADS);
    for (int i = 0; i < m_numParticles; ++i)
    {
        partPos = m_particleList[i].m_pos;
        size = m_particleList[i].m_size;
        glTexCoord2f(0.0, 1.0);
        glVertex3f(partPos.x, partPos.y, partPos.z);
        glTexCoord2f(1.0, 1.0);
        glVertex3f(partPos.x + size, partPos.y, partPos.z);
        glTexCoord2f(1.0, 0.0);
        glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
        glTexCoord2f(0.0, 0.0);
        glVertex3f(partPos.x, partPos.y - size, partPos.z);
    }
    glEnd();

    glDisable(GL_BLEND);
    glDisable(GL_TEXTURE_2D);
} // CSnowstorm::Update()

//****************************************************************************
CSnowstorm::InitializeSystem

Ładuje teksturę płatka śniegu
*****
void CSnowstorm::InitializeSystem()

```

```
{  
    // tworzy obiekt tekstury  
    glGenTextures(1, &m_texture);  
    glBindTexture(GL_TEXTURE_2D, m_texture);  
  
    // ładuje obraz tekstury  
    BITMAPINFOHEADER bitmapInfoHeader;  
    unsigned char *buffer = LoadBitmapFileWithAlpha("snowstorm.bmp", &bitmapInfoHeader);  
  
    // konfiguruje teksturę  
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);  
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);  
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);  
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);  
    glTexImage2D(GL_TEXTURE_2D, 0, 4, bitmapInfoHeader.biWidth,  
    bitmapInfoHeader.biHeight,  
    0, GL_RGBA, GL_UNSIGNED_BYTE, buffer);  
    gluBuild2DMipmaps(GL_TEXTURE_2D, 4, bitmapInfoHeader.biWidth,  
    bitmapInfoHeader.biHeight, GL_RGBA, GL_UNSIGNED_BYTE, buffer);  
  
    // zwalnia bufor  
    free(buffer);  
  
    // wywołuje metodę klasy bazowej  
    CParticleSystem::InitializeSystem();  
} // CSnowstorm::InitializeSystem  
  
*****  
CSnowstorm::KillSystem  
  
Zwala obiekt tekstury  
*****  
void CSnowstorm::KillSystem()  
{  
    // jeśli obiekt tekstury istnieje, to zwalnia go  
    if (glIsTexture(m_texture))  
    {  
        glDeleteTextures(1, &m_texture);  
    }  
  
    // wywołuje metodę klasy bazowej  
    CParticleSystem::KillSystem();  
} // CSnowstorm::KillSystem
```

## Mgła

Efekt mgły może posiadać różne zastosowania. Oprócz naśladowania rzeczywistego zjawiska może dodatkowo służyć do przesłonięcia obiektów znajdujących się w większej odległości od obserwatora. Pozwala to zmniejszyć złożoność grafiki i tym samym zwiększyć efektywność tworzonej sceny. Zastosowanie mgły zapobiega też nagletemu pojawianiu się obiektów w polu widzenia obserwatora.

Istnieje szereg sposobów na stworzenie implementacji efektu mgły. OpenGL udostępnia gotowe funkcje umożliwiające tworzenie mgły.

## Mgła w OpenGL

Rozwiążanie zastosowane w OpenGL polega na łączeniu każdego piksela obrazu z kolorem mgły na podstawie współczynnika łączenia zależnego od odległości od obserwatora, gęstości mgły i bieżącego trybu tworzenia mgły. Aby skorzystać z mechanizmu tworzenia mgły udostępnianego w OpenGL, należy najpierw aktywować go za pomocą funkcji `glEnable()`:

```
glEnable(GL_FOG);
```

Właściwości mgły określa się wywołując następujące wersje funkcji `glFog()`:

```
glFogf(GLenum pname, GLfloat param);
glFogi(GLenum pname, GLint param);
glFogfv(GLenum pname, GLfloat *params);
glFogiv(GLenum pname, GLint *params);
```

Parametr `param` reprezentuje pojedynczą wartość właściwości, a parametr `params` tablicę jednej lub więcej wartości. Znaczenie tych wartości zależy od parametru `pname`, który może przyjmować wartości przedstawione w tabeli 15.1.

**Tabela 15.1.** Właściwości mgły

Właściwość	Opis
<code>GL_FOG_MODE</code>	Właściwość ta może posiadać wartość <code>GL_LINEAR</code> , <code>GL_EXP</code> lub <code>GL_EXP2</code> określającą, które z równań używane jest do obliczania współczynnika łączenia (domyślną wartością jest <code>GL_EXP</code> )
<code>GL_FOG_DENSITY</code>	Właściwość ta określa gęstość mgły, która używana jest w równaniach współczynnika łączenia; jej wartość musi być dodatnia (domyślną wartością jest 1.0)
<code>GL_FOG_START</code>	Właściwość ta określa początek mgły i wykorzystywana jest w równaniach współczynnika łączenia
<code>GL_FOG_END</code>	Właściwość ta określa koniec mgły i wykorzystywana jest w równaniach współczynnika łączenia
<code>GL_FOG_INDEX</code>	Właściwość określająca indeks koloru mgły
<code>GL_FOG_COLOUR</code>	Właściwość określająca kolor mgły; jej wartość przekazuje się za pomocą tablicy zawierającej składowe koloru (domyślnym kolorem jest czarny)

Współczynnik łączenia używany podczas łączenia koloru mgły z kolorem pikseli wyznaczany jest za pomocą jednego z trzech równań wybranego przez właściwość `GL_FOG_MODE`. Równania te mają następującą postać:

```
GL_LINEAR
współczynnik = (koniec - z) / (koniec - początek)
GL_EXP
współczynnik = e ^ (-gęstość*głębokość)
GL_EXP2
współczynnik = e ^ (-gęstość*głębokość)^2
```

Wystarczy pamiętać o tym, które z właściwości mgły wykorzystują poszczególne równania. Właściwości określające początek i koniec mgły wykorzystywane są jedynie przez równanie używane w trybie GL\_LINEAR, a gęstość w trybach GL\_EXP lub GL\_EXP2.

I to wszystko! Wykorzystanie mechanizmu tworzenia efektu mgły udostępnianego przez OpenGL jest więc niezwykle proste. Osiągnięcie odpowiedniego efektu może na początku wymagać nieco eksperymentów związanych z właściwościami mgły, ale później mechanizm mgły w OpenGL okaże się niezwykle użyteczny.



Efekt mgły można włączać i wyłączać w dowolnych momentach tworzenia grafiki OpenGL. Dzięki temu efekt mgły może mieć wpływ na wygląd wybranych obiektów.

## Mgła objętościowa

Chociaż mechanizm tworzenia mgły w OpenGL jest łatwy w użyciu i daje dobre efekty w większości zastosowań, to posiada on także pewne ograniczenia. Najważniejszą jego wadą jest obliczanie koloru mgły dla danego wierzchołka na podstawie jego odległości od obserwatora na osi z. Uwidacznia się ona szczególnie w przypadku obiektów znajdujących się na skraju pola widzenia obserwatora, które są położone blisko obserwatora na osi z i znajdują się w większej odległości w pozostałych wymiarach. Obiekty te nie są wystarczająco przesłonięte mgłą.

Lepszym rozwiązańem od wyznaczenia efektu mgły jedynie na podstawie odległości jest jego reprezentacja za pomocą oddzielnego bytu w wirtualnym świecie. Istnieje kilka sposobów takiej reprezentacji mgły wykorzystujących sfery, cząstki lub mapy mgły (konsepcja map mgły przypomina omówione już mapy oświetlenia). Wszystkie te sposoby określane są wspólnym mianem *mgły objętościowej*, ponieważ definiują objętość mgły wypełniającej wirtualny świat. Mgła ta nie posiada bezpośredniej reprezentacji graficznej, ale jej objętość uwzględniana podczas rysowania przesłanianych przez nią obiektów.

W książce tej nie będą omawiane techniczne szczegóły implementacji mgły objętościowej. W sieci Internet dostępnych jest kilka artykułów opisujących szczegółowo to zagadnienie. Adresy tych artykułów zamieszczone zostały w dodatku A.

## Odbicia

W rozdziale 9. omówione zostały mapy otoczenia umożliwiające uzyskanie efektu odbicia otoczenia na powierzchni obiektu. Istnieją także inne sposoby symulacji odbić, które są stosunkowo łatwe w użyciu. W podrozdziale tym omówiony zostanie jeden z takich sposobów polegający na rzutowaniu odbić na płaszczyznę.

Technikę tę zastosowano już w jednym z przykładowych programów zamieszczonych w tej książce. Jednak wtedy uproszczony został nieco sposób tworzenia odbić, co było możliwe ze względu na specyfikę sceny.

Omówienie sposobu tworzenia odbić rozpocznie się więc od bardzo prostego przykładu, który pozwoli zrozumieć ideę jego działania, a następnie przedstawione zostaną dodatkowe czynniki, które trzeba będzie wziąć pod uwagę w ogólnym przypadku. Można więc wyobrazić sobie nieskończoną płaszczyznę o współrzędnej  $y = 0$ , nad którą unosi się pewien obiekt. W jaki sposób uzyskać można jego odbicie na płaszczyźnie? Wystarczy najpierw narysować obiekt i płaszczyznę, a następnie narysować powtórnie obiekt zmieniając znak jego współrzędnych na osi  $y$ :

```
DrawGround();
DrawObject();
glScale(1.0, -1.0, 1.0);
DrawObject();
```

## Odbicia światła

Aby jednak utworzone odbicie wyglądało prawidłowo, trzeba wziąć pod uwagę wiele dodatkowych czynników. Na przykład jeśli wspomniana scena jest oświetlona, to odbicie obiektu będzie nieprawidłowo oświetlone, jeśli rysując je nie zmieni się położenia źródła światła. Zakładając, że źródła światła umieszczane są na scenie za pomocą funkcji `PositionLights()`, kod tworzący scenę powinien mieć następującą postać:

```
PositionLights();
DrawGround();
DrawObject();
glScale(1.0, -1.0, 1.0);
PositionLights();
DrawObject();
```

## Obsługa bufora głębi

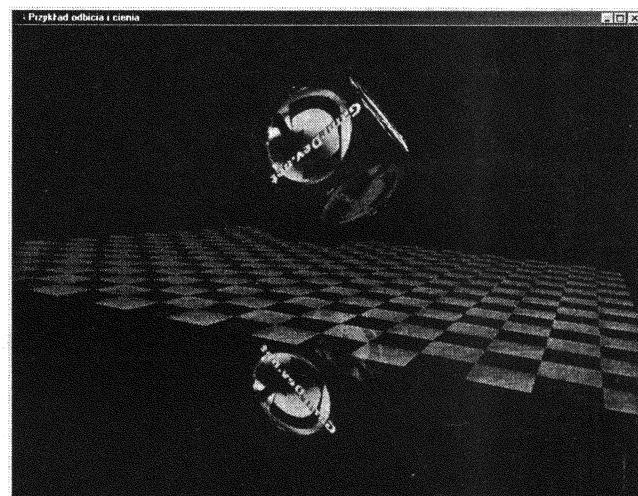
Uzyskanie prawidłowego odbicia wymaga rozwiązania jeszcze wielu innych problemów. Na przykład jeśli podczas tworzenia sceny wykorzystuje się bufor głębi, to odbicie nie będzie w ogóle widoczne, ponieważ płaszczyzna znajdująca się będzie bliżej obserwatora. Oczywiście wystarczy wyłączyć bufor głębi podczas rysowania samego odbicia, by uzyskać prawidłowy obraz, ale tylko w przypadku tej prostej sceny. Jednak jeśli będą rysowane odbicia przesłaniających się obiektów, to z pewnością będzie trzeba skorzystać z bufora głębi. Rozwiązanie tego problemu polega na wyłączeniu zapisu do bufora głębi jedynie podczas rysowania płaszczyzny odbicia. Bufor głębi wykorzystuje się podczas rysowania całej sceny uzyskując właściwe przesłanianie obiektów. Pozostaje on włączony także podczas rysowania płaszczyzny odbicia, dzięki czemu nie przesłania ona obiektów znajdujących się bliżej obserwatora. Jednocześnie brak możliwości zapisu do bufora głębi podczas rysowania płaszczyzny powoduje, że nie przysłania ona znajdujących się pod nią obiektów reprezentujących odbicia. Kod tworzący scenę będzie miał wtedy następującą postać:

```
// rysuje oryginalne obiekty
PositionLights();
DrawObject();
glDepthMask(GL_FALSE);
DrawGround();
// oraz ich odbicia
glScale(1.0, -1.0, 1.0);
PositionLights();
DrawObject();
```

## Obsługa skończonych płaszczyzn za pomocą bufora powielania

Dotąd zakładano na potrzeby opisywanej sceny, że płaszczyzna odbicia jest nieskończona. W praktyce jednak wszystkie płaszczyzny odbić będą skończone. Stwarza to niebezpieczeństwo, że obserwator dostrzeże obiekty reprezentujące odbicia poza obszarem płaszczyzny odbicia (patrz: rysunek 15.4). Podobna sytuacja pojawia się, gdy płaszczyzna odbicia otoczona jest przez inne płaszczyzny, na których nie powinny pojawić się odbicia. Na przykład odbicia nie powinny pojawiać się na ścianie, na której wisi lustro.

**Rysunek 15.4.**  
*Koniec iluzji odbicia!*



Opisane problemy nie występowały w dotychczasowych przykładach, ponieważ obserwator nie mógł znaleźć się w położeniu, które ujawniałoby wadę efektu odbicia. Teraz poszukiwać należy jednak ogólnego rozwiązania tych problemów, które sprawdzi się w każdej sytuacji. W tym celu trzeba poinformować maszynę OpenGL, aby nie tworzyła rysunku odbić poza płaszczyzną odbicia. Z pomocą przyjdzie tutaj bufor powielania. Trzeba narysować w nim płaszczyznę odbicia wyznaczając w ten sposób obszar, w którym rysowane będą obiekty reprezentujące odbicie. Oryginalne obiekty i płaszczyznę można narysować później w normalnym trybie. Nowa wersja kodu rysującego odbicie będzie teraz wyglądać następująco:

```
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);
 glEnable(GL_STENCIL_TEST);
 glStencilFunc(GL_ALWAYS, 1, 0xFFFFFFFF);
 glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
 DrawSurface();

// włącza bufore koloru i głębi
 glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
 glDepthMask(GL_TRUE);
 glStencilFunc(GL_EQUAL, 1, 0xFFFFFFFF);
 glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
```

```

glPushMatrix();
glScalef(1.0, -1.0, 1.0);
glLightfv(GL_LIGHT0, GL_POSITION, g_lightPos);
DrawObject();
glPopMatrix();

PositionLights();
DrawSurface();
DrawObject();

```

## Tworzenie nieregularnych odbić

Opisana metoda tworzenia odbić zakłada użycie doskonale odbijającej powierzchni, dzięki czemu powstaje wierne odbicie oryginalnego obiektu. Założenie takie jest uzasadnione w przypadku luster. W przypadku mniej doskonałych powierzchni odbijających, takich jak na przykład powierzchnia przedmiotów metalowych, rysunek odbicia powinien zostać połączony z rysunkiem powierzchni. Oczywiście można w tym celu posłużyć się funkcjami łączenia kolorów dostępnymi w OpenGL. Najpierw należy narysować obiekt reprezentujący odbicie, a następnie połączyć z nim płaszczyznę odbicia. Jedyną zmianą w ostatnim przykładzie kodu będzie włączenie łączenia kolorów przed drugim wywołaniem funkcji DrawSurface().

## Odbicia na dowolnie zorientowanych płaszczyznach

Dotychczas używane były płaszczyzny xz jako płaszczyzny odbicia. Założenie takie pozwoliło uprościć przekształcenia związane z tworzeniem odbicia. W praktyce jednak tworzy się odbicia na dowolnie zorientowanych płaszczyznach. W tym celu niezbędne są dodatkowe przekształcenia.

1. Konieczne jest przesunięcie i obrót powierzchni odbicia i odbijanych obiektów w taki sposób, by powierzchnia odbicia była położona w płaszczyźnie xz, a jej środek pokrywał się z początkiem układu współrzędnych.
2. Konieczne są przekształcenia odbicia przez odwrócenie współrzędnej względem płaszczyzny odbicia (czyli współrzędnej y dla płaszczyzny xz, współrzędnej z dla płaszczyzny xy i współrzędnej x dla płaszczyzny yz).
3. Konieczne jest przekształcenie odwrotne do przekształcenia z punktu pierwszego przywracające wyjściowe położenie płaszczyzny odbicia i obiektów.



Zamiast płaszczyzny xz w punkcie 1. można równie dobrze korzystać z płaszczyzny xy bądź yz.

Zakładając, że w punkcie 1. przekształca się powierzchnię odbicia do płaszczyzny xz i zmienna mat reprezentuje macierz przekształcenia wykonanego w punkcie 1., a zmienna matInverse macierz odwrotną do macierzy mat, to kompletne przekształcenie odbicia będzie wyglądać jak poniżej:

```

glMultMatrixf(matInverse);
glScale(1.0, -1.0, 1.0);
glMultMatrixf(mat);

```

Opisane przekształcenia są wystarczające do uzyskania realistycznie wyglądających odbić na dowolnych płaszczyznach. Powtarzając przekształcenia dla płaszczyzn kolejnych wielokątów tworzących pewną powierzchnię można uzyskać też odbicia na dowolnych powierzchniach. Jako że oznacza to wykonywanie przekształceń wszystkich odbijanych obiektów dla każdego wielokąta, to należy korzystać z tej metody ostrożnie. Zwłaszcza w przypadku gier można spowodować w ten sposób zbyt duże spowolnienie ich działania.

Zwykle w tym miejscu ilustrowano omówiony materiał przykładem programu. Jako że odbicia były już tworzone przez programy z poprzednich rozdziałów, można tym razem połączyć program ilustrujący tworzenie odbić z programem tworzącym cienie, który omówiony zostanie w następnym podrozdziale.

## Cienie

Ponieważ cienie występują powszechnie w oglądanym przez nas na co dzień świecie, to ich dodanie do obiektów świata wirtualnego znacznie zwiększa jego realizm. Dodatkowo zwiększały one także wrażenie trójwymiarowości, gdyż pomagają uwidoczyć wzajemne położenie obiektów. Dlatego też na przestrzeni ostatnich lat opracowano wiele różnych technik tworzenia cieni.

Narysowanie cienia kuli zawieszonej nad płaszczyzną i oświetlonej pojedynczym źródłem światła nie przedstawia większego problemu. Wystarczy w tym celu określić obszar płaszczyzny, który przesłania kula, patrząc z perspektywy źródła światła. Sprawa jednak komplikuje się w przypadku, gdy zacieniona powierzchnia nie jest płaska, a obiekt rzucający cień ma skomplikowany, nieregularny kształt i oświetlony jest przez wiele źródeł światła. Wprowadzenie wielu źródeł światła czyni też możliwą sytuację, w której wiele obiektów zacieniać się będzie nawzajem. Dodatkowo różne rodzaje oświetlenia tworzą różne cienie. Jedynie oświetlenie za pomocą pojedynczego, skupionego strumienia światła tworzy cienie o wyraźnych granicach i równomiernym wypełnieniu. Na skutek oświetlenia obiektów światłem otoczenia bądź wieloma źródłami światła powstają cienie o rozmytych granicach i wypełnieniu zmieniającym się w kierunku środka cienia. Symulacja tego rodzaju cieni wymaga zdecydowanie większej pracy.

Modelowanie cieni jest więc dość złożonym zadaniem i dlatego opracowano wiele różnych sposobów jego realizacji. Większość metod tworzenia cieni dotyczy pojedynczych aspektów tego zagadnienia. Jedynie kilka metod bierze pod uwagę większość omówionych zagadnień. W kolejnych podrozdziałach omówione zostaną jedynie te metody, które są na tyle efektywne, by mogły zostać użyte w grach.

## Cienie statyczne

Metoda cieni statycznych polega na jednorazowym przygotowaniu cieni sceny jeszcze przed uruchomieniem aplikacji graficznej. W tym celu można wykorzystać dowolnie skomplikowany algorytm o dużym stopniu realizmu, ponieważ obliczenia nie są już wykonywane podczas działania programu. Cienie prezentowane są za pomocą teksturow umieszczonych na stałe na obiektach. Podstawową zaletą tej metody jest oczywiście

brak dodatkowego przetwarzania związanego z obsługą cieni, natomiast wadą — niski stopień realizmu uzyskanego efektu. Choć cienie przemieszczają się razem z obiektami względem źródeł światła, to ich kształt nie ulega zmianie. Efekt taki jest do przyjęcia jedynie w grach używających grafiki dwuwymiarowej. Ponieważ metoda ta sprowadza się jedynie przy rozszerzaniu wyglądu obiektów o odpowiednie tekstury, nie będzie omawiana szczegółowo.

## Rzutowanie cieni

Rzutowanie cieni jest metodą stosowaną od dłuższego czasu w wielu grach wykorzystujących grafikę trójwymiarową. Metodę tę zastosować można także w przykładzie programu, który został zamieszczony przy końcu tego podrozdziału. Rzutowanie cieni jest dość efektywną metodą tworzenia realistycznych cieni. Jej idea polega na rzutowaniu obiektów na płaszczyznę z perspektywy oświetlającego je źródła światła. Uzyskuje się w ten sposób cień będący płaskim obrazem obiektu na płaszczyźnie (w podobny sposób jak płaskie obrazy obiektów widziane na ekranie). Wystarczy jedynie wypełnić obraz obiektu czarnym kolorem i cień jest gotowy!

## Macierz rzutowania cieni

Teraz należy przyjrzeć się szczegółom implementacji tej metody tworzenia cieni. Aby wykonać rzutowanie, trzeba zdefiniować odpowiednią macierz. Jej zawartość zależeć będzie od położenia źródła światła i płaszczyzny, na którą rzutuje się obiekt. Nie trzeba wyprowadzać tej macierzy od podstaw, wystarczy zaprezentować jej gotową postać (patrz: rysunek 15.5). Zmienna `dot` reprezentuje iloczyn skalarny zmiennej `lightPos` i `plane`. Zmienna `lightPos` jest tablicą, której pierwsze trzy elementy reprezentują współrzędne  $x$ ,  $y$  i  $z$  określające położenie źródła światła, a czwarty element przyjmuje wartość 0 lub 1 w zależności od tego, czy źródło to jest kierunkowe czy pozycyjne. Zmienna `plane` jest strukturą, której pola `a`, `b`, `c` i `d` reprezentują współczynniki równania płaszczyzny  $ax + by + cz + d = 0$ . Aby uzyskać wartości `a`, `b`, `c` i `d` dla danej płaszczyzny, należy wyznaczyć wektor normalny płaszczyzny oraz określić współrzędne dowolnego punktu płaszczyzny. Wartości `a`, `b`, `c` i `d` odpowiadają wtedy składowym  $x$ ,  $y$  i  $z$  wektora normalnego, a wartość `d` uzyskuje się wstawiając współrzędne punktu płaszczyzny do jej równania. Sposób wyznaczenia równania płaszczyzny omawiany jest szczegółowo w rozdziale 19.

**Rysunek 15.5.**

Macierz  
rzutowania cieni

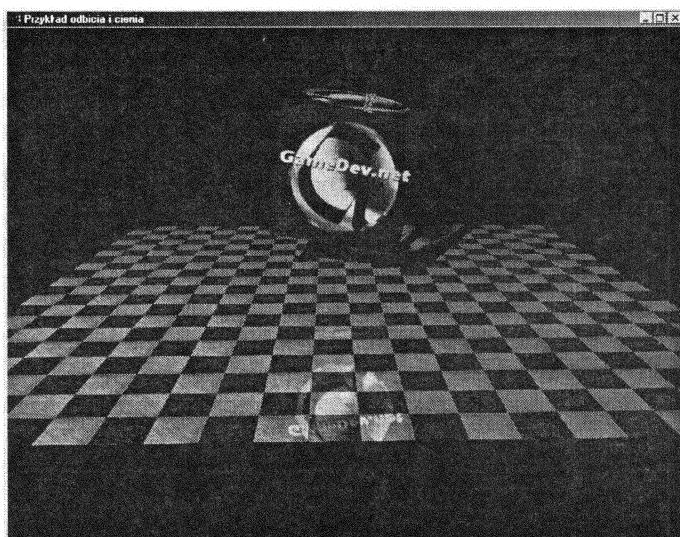
$$\begin{bmatrix} \text{dot} - \text{lightPos}[0] * \text{plane.a} & -\text{lightPos}[1] * \text{plane.a} & -\text{lightPos}[2] * \text{plane.a} & -\text{lightPos}[3] * \text{plane.a} \\ -\text{lightPos}[0] * \text{plane.b} & \text{dot} - \text{lightPos}[1] * \text{plane.b} & -\text{lightPos}[2] * \text{plane.b} & -\text{lightPos}[3] * \text{plane.b} \\ -\text{lightPos}[0] * \text{plane.c} & -\text{lightPos}[1] * \text{plane.c} & \text{dot} - \text{lightPos}[2] * \text{plane.c} & -\text{lightPos}[3] * \text{plane.c} \\ -\text{lightPos}[0] * \text{plane.d} & -\text{lightPos}[1] * \text{plane.d} & -\text{lightPos}[2] * \text{plane.d} & \text{dot} - \text{lightPos}[3] * \text{plane.d} \end{bmatrix}$$

Po utworzeniu macierzy rzutowania cieni można przekształcić z jej pomocą wierzchołki obiektu i uzyskać jego spłaszczony obraz, co pokazuje rysunek 15.6. Aby wyglądał on jak cień obiektu, trzeba podjąć jeszcze dodatkowe działania.

Wystarczy jedynie przed narysowaniem obrazu cienia wyłączyć oświetlenie sceny i mechanizm tekstur oraz wybrać czerń jako bieżący kolor rysowania. Dla uzyskania bardziej realistycznego cienia wskazane jest też połączenie jego kolorów z kolorami zacienionej płaszczyzny.

**Rysunek 15.6.**

Obraz obiektu na płaszczyźnie powstały na skutek rzutowania



## Problemy z buforem głębi

Z metodą tworzenia cieni za pomocą rzutowania obiektów związany jest następujący problem. Wykonywane przekształcenia wierzchołków obiektu mają doprowadzić do powstania obrazu obiektu na zacienianej płaszczyźnie. Ponieważ jednak obliczenia wykonywane podczas przekształceń posiadają skończoną dokładność, to odległość wizerunku cienia wyznaczana dla potrzeb bufora głębi będzie zawsze różna od odległości zacienianej płaszczyzny — raz mniejsza, a innym razem większa. W efekcie tworzone cienie mogą nie być pełne, gdyż zostaną częściowo przesłonięte przez płaszczyznę. Istnieje kilka rozwiązań tego problemu. W przykładach programów zamieszczonych w tej książce będzie się wyłączać bufor głębi podczas rysowania cieni. Trzeba jednak pamiętać wtedy, aby cienie zawsze rysować przed obiektem, który je przesłaniają. W przeciwnym razie cienie pojawią się na obiektach, na których nie powinno się znaleźć.

## Ograniczanie obszaru cieni za pomocą bufora powielania

Ponieważ płaszczyzna, na której umieszczone są cienie posiada skończone rozmiary, trzeba zapobiec rysowaniu cieni poza jej obszarem. Podobnie jak w przypadku tworzenia odbić, wykorzystuje się w tym celu bufor powielania. Bufor powielania pozwala się uporać także z innym problemem pojawiającym się podczas łączenia kolorów: jeśli obiekt rzucający cień posiada skomplikowany kształt, to może zdarzyć się, że niektóre z punktów płaszczyzny zostaną połączone z punktami obrazu obiektu więcej niż raz. W efekcie punkty te będą intensywniej zacienione od pozostałych punktów. Aby zapobiec temu niepożdanemu efektowi, trzeba zmieniać zawartość bufora powielania dla każdego z rysowanych pikseli.

Proces zastosowania bufora powielania podczas tworzenia cieni można ująć w następujących punktach.

1. Zacienianą powierzchnię rysuje się w buforze powielania, aby ograniczyć rysowanie cieni tylko do jej obszaru.
2. Tworzy się macierz rzutowania cieni na podstawie położenia źródła światła i zacienianej płaszczyzny, a następnie mnoży ją przez macierz modelowania.
3. Wyłącza się oświetlenie sceny i mechanizm tekstur. Wybiera się czerń jako bieżący kolor rysowania.
4. Wyłącza się bufor głębi.
5. Włącza się łączenie kolorów.
6. Rysuje się obiekt rzucający cień.

Po wykonaniu tych operacji trzeba pamiętać o przywróceniu oświetlenia sceny, włączeniu odwzorowań tekstur i bufora głębi. Przykładowy program zamieszczony na końcu bieżącego podrozdziału przedstawia kod odpowiadający wymienionym operacjom.

## **Obsługa wielu źródeł światła i wielu zacienianych powierzchni**

Omawiając dotąd metodę tworzenia cieni zakładano istnienie pojedynczego źródła światła i pojedynczej, zacienianej płaszczyzny. Opisaną metodę można rozszerzyć o przypadki, w których występuje wiele źródeł światła przez powtórzenie procesu tworzenia cieni dla każdego źródła. Podobnie można postąpić w przypadku rzucania cieni na wiele płaszczyzn. Podobnie jak w przypadku odbić, płaszczyzny te mogą być wielokątami tworzącymi dowolnie złożone powierzchnie. Jednak koszt utworzenia cieni na powierzchniach złożonych z wielu wielokątów będzie zbyt duży, by można było ponieść go w przypadku interaktywnych z natury gier.

## **Problemy związane z rzutowaniem cieni**

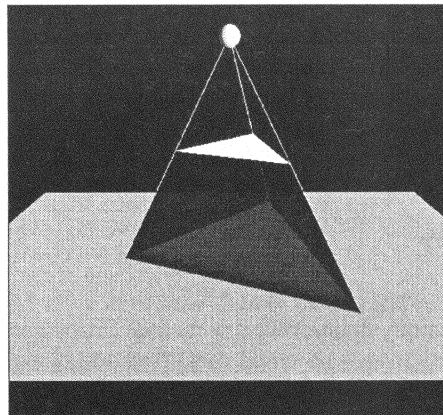
Rzutowanie cieni umożliwia uzyskanie dobrych efektów, ale nie jest metodą pozbawioną wad i ograniczeń. Po pierwsze — za pomocą tej metody nie można uzyskać cieni rzucanych przez część obiektu na inną część tego samego obiektu. Metoda ta okaże się też nieprzydatna w przypadku istnienia wielu źródeł światła i wzajemnie zacieniających się obiektów. Rozwiążaniem w obu przypadkach byłoby potraktowanie każdego wielokąta jako zacienianej płaszczyzny. Jednak w praktyce rozwiązanie takie okaże się mało wygodne i zbyt czasochłonne. Po drugie — rzutowanie cieni tworzy wizerunki cieni o ostrych granicach i jednolitym wypełnieniu. Uzyskanie bardziej realistycznych cieni o rozmytych granicach i zmieniającym się wypełnieniu jest możliwe za pomocą metody rzutowania cieni w wielu przebiegach. Podczas kolejnego przebiegu zmienia się nieco położenie źródła światła, a cienie uzyskane w poszczególnych przebiegach łączą się razem, co pozwala uzyskać stopniową zmianę ich jasności. Jednak złożoność takiego rozwiązania skutecznie zapobiega jego stosowaniu w grach. I wreszcie — po trzecie — łączenie kolorów wizerunku cienia z kolorami zacienianej płaszczyzny powoduje, że zachowane zostaną na niej wszystkie istniejące dotąd efekty światłowe. Jednak pomimo swoich ograniczeń metoda rzutowania cieni sprawdza się w praktyce w bardzo wielu sytuacjach. Przykładowy program pokaże też, że tworzone z jej pomocą efekty cieni są zupełnie zadowalające.

## Bryły cieni w buforze powielania

Metoda wykorzystująca bryły cieni opracowana została jakiś czas temu, ale nie była powszechnie stosowana ze względu na stosunkowo dużą złożoność obliczeniową. Obecnie dzięki znacznemu wzrostowi możliwości przetwarzania sprzętu komputerowego (a w szczególności kart graficznych) możliwe staje się zastosowanie tej metody nawet w grach. Metoda brył cieni umożliwia tworzenie prawidłowych wizerunków cieni na dowolnych powierzchniach (w tym także wzajemne zacienianie się różnych obiektów oraz tworzenie cieni jednego fragmentu obiektu na innej części tego samego obiektu). Zastosowanie tej metody pozwala także uniknąć efektów świetlnych w obszarze cienia. Omówiony tutaj zostanie algorytm metody bryły cieni (bez analizy szczegółów jej implementacji).

Ogólna koncepcja tej metody polega na poprowadzeniu linii od źródła światła przez krawędzie obiektów i zdefiniowanie w ten sposób brył cieni, co ilustruje rysunek 15.7. Po zdefiniowaniu brył cieni należy ustalić, które obiekty lub ich fragmenty znajdują się wewnętrz danej bryły cienia. Metoda bryły cieni składa się więc z dwóch zasadniczych operacji: wyznaczenia bryły cieni i określenia, czy dany punkt obiektu należy do bryły cienia.

**Rysunek 15.7.**  
*Bryła cienia*



Najwięcej pracy wymaga wyznaczenie brył cieni. W tym celu trzeba wyznaczyć sylwetkę każdego obiektu z perspektywy źródła światła. Sylwetka ta składa się z tych wszystkich krawędzi obiektu, które są wspólne dla jednego wielokąta obróconego przednią stroną w kierunku źródła światła i drugiego obróconego stroną tylną. Nie będą tutaj omawiane szczegóły implementacji operacji określania sylwetki. Ograniczyć się można jedynie do zwrócenia uwagi, że wymaga ona przejrzenia wszystkich krawędzi wszystkich obiektów tworzących scenę. Po znalezieniu sylwetki obiektu tworzy się bryłę cienia przeprowadzając ze źródła światła linie przechodzące przez wierzchołki należące do sylwetki.

W celu ustalenia tego, czy dany punkt leży wewnętrz bryły cienia, trzeba połączyć go linią z obserwatorem i policzyć, ile razy wnika ona w bryłę cienia, a także — ile razy ją opuszcza. Wnikanie linii do wnętrza bryły cienia rozpoznaje się po tym, że przecinany przez nią wielokąt skierowany jest przodem do obserwatora. Gdy natomiast wielokąt będzie skierowany tyłem do obserwatora, będzie to oznaczać opuszczenie bryły cienia.

Do zliczania opisanych sytuacji wykorzystać można bufor powielania w przedstawiony niżej sposób.

1. Należy narysować scenę przy włączonym świetle otoczenia i emisji.
2. W drugim przebiegu tworzenia grafiki trzeba narysować bryły cienia. Ponieważ nie powinny być one widoczne, to wyłączyć trzeba najpierw bufor koloru i głębi, a bryły cienia utworzyć jedynie w buforze powielania. W przebiegu tym zlicza się także liczbę razy, którą wnikają linie proste do wnętrza brył cienia. W tym celu należy zwiększać wartość bufora powielania dla każdego punktu za każdym razem, gdy prosta łącząca dany punkt z obserwatorem przetnie wielokąt zwrócony przednią stroną w kierunku obserwatora.
3. Kolejny przebieg wykonać trzeba podobnie jak opisany w punkcie 2., ale z tą różnicą, że tym razem będzie trzeba zmniejszać wartość bufora powielania dla każdego punktu za każdym razem, gdy prosta łącząca dany punkt z obserwatorem przetnie wielokąt zwrócony tylną stroną w kierunku obserwatora.
4. Kolejny przebieg rysowania sceny wykonać należy przy włączonym świetle rozproszonym i świetle odbicia jedynie dla tych punktów, dla których bufor powielania zawiera wartość 0.

Opisana metoda posiada także pewne wady. Wymaga czterokrotnego tworzenia grafiki, co jednak wobec szybkiego rozwoju sprzętu graficznego nie jest już takim problemem. Gorzej, że opisana metoda zakłada, że obserwator znajduje się poza bryłami cienia, co nie zawsze musi mieć miejsce. Istnieją sposoby poradzenia sobie z tym problemem, ale komplikują one dodatkowo algorytm. Mimo to metoda brył cieni wkracza coraz śmielej w dziedzinę gier, dlatego warto o niej wiedzieć.

## Inne metody

Istnieje szereg innych metod tworzenia, które nie będą tutaj omawiane, takich jak mapy cieni, metody hybrydowe czy metoda bilansu energetycznego. Więcej informacji na temat, jak również na temat metod przedstawionych w tym rozdziale, zawierają źródła wymienione w dodatku A.

## Przykład: odbicia i cienie

Metody tworzenia odbić i cieni za pomocą rzutowania zilustrowane zostaną na przykładzie animacji obracającego się sześcianu zaprezentowanej już w rozdziale 1. Program ten został tym razem wzbogacony o tworzenie odbicia i cienia sześcianu. Użycie bufora powielania zapobiega tworzeniu odbicia i cienia poza obszarem płaszczyzny. Pełen kod źródłowy programu umieszczony został na dysku CD. Poniżej przedstawione zostały jego najważniejsze fragmenty związane z tworzeniem odbicia i cienia.

Na początku tworzy się macierz rzutowania cieni:

```
float plane[4] = { 0.0, 1.0, 0.0, 0.0 };
SetShadowMatrix(g_shadowMatrix, g_lightPos, plane);
```

Zmienna g\_shadowMatrix jest globalnie dostępna tablicą szesnastu wartości typu float, a g\_lightPos globalną tablicą określającą położenie źródła światła. Funkcja SetShadowMatrix() tworzy wspomnianą macierz w następujący sposób:

```
void SetShadowMatrix(GLfloat destMat[16], float lightPos[4], float plane[4])
{
    GLfloat dot;

    // iloczyn skalarny płaszczyzny i położenia światła
    dot = plane[0] * lightPos[0] + plane[1] * lightPos[1] + plane[2] * lightPos[2] + plane[3] * lightPos[3];

    // pierwsza kolumna macierzy
    destMat[0] = dot - lightPos[0] * plane[0];
    destMat[4] = 0.0f - lightPos[0] * plane[1];
    destMat[8] = 0.0f - lightPos[0] * plane[2];
    destMat[12] = 0.0f - lightPos[0] * plane[3];

    // druga kolumna macierzy
    destMat[1] = 0.0f - lightPos[1] * plane[0];
    destMat[5] = dot - lightPos[1] * plane[1];
    destMat[9] = 0.0f - lightPos[1] * plane[2];
    destMat[13] = 0.0f - lightPos[1] * plane[3];

    // trzecia kolumna macierzy
    destMat[2] = 0.0f - lightPos[2] * plane[0];
    destMat[6] = 0.0f - lightPos[2] * plane[1];
    destMat[10] = dot - lightPos[2] * plane[2];
    destMat[14] = 0.0f - lightPos[2] * plane[3];

    // czwarta kolumna macierzy
    destMat[3] = 0.0f - lightPos[3] * plane[0];
    destMat[7] = 0.0f - lightPos[3] * plane[1];
    destMat[11] = 0.0f - lightPos[3] * plane[2];
    destMat[15] = dot - lightPos[3] * plane[3];
}
```

Scena rysowana jest przez funkcję DisplayScene():

```
BOOL DisplayScene()
{
    // określa kierunek widzenia
    glLoadIdentity();
    gluLookAt(0.0, 3.0, 10.0,
              0.0, 0.0, 0.0,
              0.0, 1.0, 0.0);

    // opróżnia zawartość ekranu
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

    // obraca scenę
    g_rotationAngle += (DEGREES_PER_SECOND * g_timer.GetElapsedSeconds());
    glRotated(-g_rotationAngle/8.0, 0.0, 1.0, 0.0);
    glRotated(10.0 * sin(g_rotationAngle/45.0), 1.0, 0.0, 0.0);

    // przygotowuje się do zapisu do bufora powielania
    // wyłączając buforey koloru i głębi
```

```
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);

// konfiguruje bufor powielania tak,
// by zawierał wartość 1 tam, gdzie będziemy tworzyć grafikę
 glEnable(GL_STENCIL_TEST);
 glStencilFunc(GL_ALWAYS, 1, 0xFFFFFFFF);
 glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);

// rysuje płaszczyznę w buforze powielania
DrawSurface();

// włącza bufory koloru i głębi
 glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);

// grafika będzie tworzony tylko tam,
// gdzie bufor powielania zawiera wartość 1
 glStencilFunc(GL_EQUAL, 1, 0xFFFFFFFF);

// zapobiega modyfikacji bufora powielania
 glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

// rysuje odbicie sześcianu
glPushMatrix();
 glScalef(1.0, -1.0, 1.0);
 glLightfv(GL_LIGHT0, GL_POSITION, g_lightPos);
 DrawCube();
glPopMatrix();

// rysuje płaszczyznę łącząc ją z obrazem odbicia
glLightfv(GL_LIGHT0, GL_POSITION, g_lightPos);
 glEnable(GL_BLEND);
DrawSurface();
 glDisable(GL_BLEND);

// rysuje cień
glPushMatrix();
 // wybiera kolor czarny, łączy wizerunek cienia z płaszczyzną, bez oświetlenia
 // i bez testowania głębi
 glDisable(GL_TEXTURE_2D);
 glDisable(GL_LIGHTING);
 glDisable(GL_DEPTH_TEST);
 glEnable(GL_BLEND);
 // zapobiega wielokrotnemu rysowaniu cienia
 glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);
 glColor4f(0.0, 0.0, 0.0, 0.5f);

// rzutuje sześcian za pomocą macierzy cienia
glMultMatrixf(g_shadowMatrix);
DrawCube();

 glEnable(GL_TEXTURE_2D);
 glEnable(GL_DEPTH_TEST);
 glDisable(GL_BLEND);
 glEnable(GL_LIGHTING);
glPopMatrix();
 glDisable(GL_STENCIL_TEST);
```

```
// rysuje sześciian  
glPushMatrix();  
    DrawCube();  
glPopMatrix();  
  
return TRUE;  
} // DisplayScene()
```

Uzyskany efekt przedstawia rysunek 15.8.

#### Rysunek 15.8.

Odbicie  
i cień sześcianu



## Podsumowanie

W rozdziale tym przedstawione zostały sposoby tworzenia kilku często używanych efektów specjalnych. Omówiono plakatowanie, które jest sposobem efektywnego tworzenia pseudotrójwymiarowych scen za pomocą dwuwymiarowych reprezentacji obiektów. Systemy cząstek pozwalają tworzyć wiele różnych efektów. Zaprezentowano klasę bazową reprezentującą system cząstek, która posłużyć może do tworzenia własnych klas realizujących różne efekty wizualne. Mgła jest jednym z efektów specjalnych, ale służy także do zmniejszenia stopnia złożoności tworzonej grafiki przez przesłanianie oddalających się obiektów. Rozdział kończy omówienie efektywnych technik tworzenia odbić i cieni.

Nie każda gra musi używać wszystkich efektów omówionych w tym rozdziale. Na realizm gry wpływ ma nie tylko jakość tworzonych efektów, ale także ich właściwy dobór.



# **Część III**

# **Tworzymy grę**



## Rozdział 16.

# DirectX: DirectInput

Większość zaprezentowanych dotąd programów używała danych wejściowych wprowadzanych przez użytkownika. Interakcja z użytkownikiem jest podstawowym elementem każdej gry. W rozdziale tym omówione zostaną sposoby obsługi urządzeń wejścia w systemie Windows za pomocą interfejsu programowego DirectInput będącego częścią DirectX:

- ◆ przedstawione zostanie alternatywne rozwiązanie w stosunku do DirectInput korzystające bezpośrednio z komunikatów systemu Windows i interfejsu programowego Win32, a następnie omówione będą zalety i wady każdego z rozwiązań;
- ◆ omówiony zostanie sposób korzystania z interfejsu programowego DirectInput;
- ◆ utworzone zostaną obiekty DirectInputDevice reprezentujące klawiaturę, mysz i inne urządzenia wejścia;
- ◆ omówione zostaną różne sposoby uzyskiwania danych z urządzeń wejścia;
- ◆ stworzony zostanie prosty system wejścia wykorzystujący interfejs DirectInput, łatwy do zastosowania w istniejących, jak i w nowych projektach, a także przykładowy program ilustrujący sposób korzystania z tego systemu.

## Dlaczego DirectInput?

Żaden z przedstawionych dotąd programów nie korzystał z możliwości interfejsu programowego DirectInput. Jednocześnie większość z tych programów odczytywała polecenia wprowadzane przez użytkownika za pomocą urządzeń wejścia. Może więc zrozić się pytanie, do czego potrzebny będzie dodatkowy interfejs programowy DirectInput. Aby zrozumieć zalety posługiwania się interfejsem DirectInput, trzeba najpierw przedstawić alternatywny sposób korzystania z urządzeń wejścia i ujawnić jego ograniczenia.

## Komunikaty systemu Windows

System Windows wysyła nieustannie do aplikacji komunikaty zawierające informacje o zachodzących zdarzeniach. Wśród komunikatów tych znajdują się także komunikaty generowane przez urządzenia wejścia. Za każdym razem, gdy użytkownik naciska klawisz, przesuwa mysz bądź naciska jej przycisk, system Windows automatycznie generuje

odpowiedni komunikat i wysyła go do procedury okienkowej aplikacji. Istnieje wiele typów komunikatów związanych z urządzeniami wejścia. W tabeli 16.1 przedstawione zostały te z nich, które będą potrzebne do stworzenia interaktywnych aplikacji.

**Tabela 16.1. Komunikaty urządzeń wejścia**

Komunikat	Opis
WM_CHAR	Komunikat ten generowany jest za każdym razem, gdy funkcja TranslateMessage() przetwarza komunikat WM_KEYDOWN. Pole <i>wParam</i> zawiera kod znaku przypisanego naciśniętemu klawiszowi. Pole <i>lParam</i> zawiera dodatkowe informacje, takie jak liczba powtórzeń, poprzedni stan klawisza, informacja o tym, czy był naciśnięty równocześnie klawisz Alt i tak dalej. Komunikat ten jest szczególnie przydatny, gdy obsługiwany tekst wprowadzany jest przez użytkownika za pomocą klawiatury. Nie jest jednak przydatny w grach i dlatego nie będzie dokładniej omawiany.
WM_KEYDOWN	Komunikat ten generowany jest za każdym razem, gdy naciśnięty zostanie dowolny klawisz (ale nie w połączeniu z klawiszem Alt). Pole <i>wParam</i> zawiera wirtualny kod naciśniętego klawisza (kody te reprezentowane są za pomocą stałych postaci VK_nazwaklawisza; kompletną listę kodów zawiera dokumentacja interfejsu programowego Win32). Pole <i>lParam</i> zawiera dodatkowe informacje o stanie klawiatury. Zaletą tego komunikatu jest to, że przekazuje on informację o naciśnięciu klawisza. Wirtualne kody klawisza nie są tłumaczone na kody znaków i nie rozróżniają przypadków wybrania małej i dużej litery. Dzięki temu komunikat ten jest bardziej przydatny w grach niż komunikat WM_CHAR.
WM_KEYUP	Komunikat ten spełnia taką samą rolę jak komunikat WM_KEYDOWN. Różnica polega na tym, że generowany jest podczas zwolnienia klawisza.
WM_LBUTTONDOWN	Komunikat ten generowany jest na skutek naciśnięcia lewego przycisku myszy. Pole <i>wParam</i> zawiera informację o tym, które z klawiszy i przycisków były także przyciśnięte w tym samym czasie. Jego wartość może być kombinacją następujących stałych:  MK_CONTROL oznacza, że naciśnięty był także klawisz Ctrl; MK_SHIFT oznacza, że naciśnięty był także klawisz Shift; MK_LBUTTON oznacza, że naciśnięty był także lewy przycisk myszy; MK_MBUTTON oznacza, że naciśnięty był także środkowy przycisk myszy; MK_RBUTTON oznacza, że naciśnięty był także prawy przycisk myszy.  Bardziej znaczące słowo pola <i>lParam</i> reprezentuje współrzędną y kurSORA myszy w momencie naciśnięcia jej przycisku, a słowo mniej znaczące współrzędną x. Współrzędne określone są względem lewego, górnego narożnika obszaru okna (lub ekranu w trybie pełnoekranowym). Wartości współrzędnych zapisane w polu <i>lParam</i> można pobrać za pomocą makrodefinicji GET_X_LPARAM i GET_Y_LPARAM lub też makra MAKEPOINTS, które wypełni strukturę typu POINT.
WM_MBUTTONDOWN	Komunikat ten generowany jest na skutek naciśnięcia środkowego przycisku myszy (obecnie najczęściej w postaci kółka). Pole <i>wParam</i> zawiera te same informacje co w przypadku komunikatu WM_LBUTTONDOWN.
WM_RBUTTONDOWN	Komunikat ten generowany jest na skutek naciśnięcia prawego przycisku myszy. Pole <i>wParam</i> zawiera te same informacje co w przypadku komunikatu WM_LBUTTONDOWN.
WM_LBUTTONUP	Komunikat ten generowany jest na skutek zwolnienia lewego przycisku myszy. Pole <i>wParam</i> zawiera te same informacje co w przypadku komunikatu WM_LBUTTONDOWN.
WM_MBUTTONUP	Komunikat ten generowany jest na skutek zwolnienia środkowego przycisku myszy (obecnie najczęściej w postaci kółka). Pole <i>wParam</i> zawiera te same informacje co w przypadku komunikatu WM_LBUTTONDOWN.

**Tabela 16.1.** Komunikaty urządzeń wejścia — ciąg dalszy

Komunikat	Opis
WM_RBUTTONDOWN	Komunikat ten generowany jest na skutek zwolnienia prawego przycisku myszy. Pole <i>wparam</i> zawiera te same informacje co w przypadku komunikatu WM_LBUTTONDOWN.
WM_MOUSEMOVE	Komunikat ten generowany jest za każdym razem, gdy kursor myszy zmienia położenie. Pole <i>wparam</i> zawiera te same informacje co w przypadku komunikatu WM_LBUTTONDOWN. Pole <i>lParam</i> zawiera informacje o nowym położeniu kurSORA myszy zapisaną w taki sam sposób jak w przypadku innych komunikatów myszy.
WM_MOUSEWHEEL	Komunikat ten generowany jest za każdym razem, gdy obracane jest kółko myszy. Pole <i>wparam</i> zawiera te same informacje co w przypadku komunikatu WM_LBUTTONDOWN oraz dodatkowo — w starszym słowie — drogę, jaką przebyło kółko wyrażoną jako wielokrotność stałej WHEEL_DATA (posiadającej wartość 120). Jeśli odległość ta ma znak dodatni, to kółko zostaje obrócone w kierunku „od” użytkownika. Jeśli jest ujemna, to kółko zostaje obrócone w kierunku „do” użytkownika.

Komunikaty te można wykorzystywać jako źródło danych wejściowych w grach, ale wymaga to użycia kilku sztuczek.

Zwykle gra udostępnia użytkownikowi wiele poleceń wykonywanych przez naciśnięcie pojedynczego klawisza. Na przykład użytkownik może nacisnąć klawisz numeryczny w celu zmiany rodzaju broni, klawisz funkcyjny, aby uzyskać informacje o bieżącej konfiguracji gry i tak dalej. W takich przypadkach komunikaty systemu Windows okazują się zupełnie dobrym źródłem informacji. Obsługując dany komunikat można wywołać funkcję inicjującą pewną animację, zmieniającą konfigurację gry lub wykonującą jeszcze inną operację.

Teraz należy rozważyć sytuację, która wymaga od użytkownika przytrzymania klawisza przez pewien okres czasu, na przykład w celu zmiany sposobu poruszania się (przytrzymując klawisz *Shift* użytkownik może poruszać się biegiem). W takim przypadku komunikaty systemu Windows okazują się mało wygodnym rozwiązaniem. Po naciśnięciu klawisza system Windows wysyła komunikat do aplikacji. Następnie po krótkiej przerwie powtarza okresowo wysyłanie komunikatu. O ile obsługa cyklicznie nadchodzących komunikatów nie powinna sprawić większego kłopotu, to wspomniana przerwa po pierwszym komunikacie powoduje zwykle zakłócenie płynności tworzenia grafiki za każdym razem, gdy użytkownik zaczyna się poruszać lub zmienia kierunek ruchu. Korzystanie z komunikatów w opisanej sytuacji jest też mało intuicyjne. Ponieważ użytkownik gry porusza się przez większość czasu, to wygodniejsza będzie możliwość sprawdzania stanu klawiatury za każdym razem, gdy tworzona jest kolejna klatka animacji. Ta sama uwaga dotyczy też sposobu korzystania z informacji o ruchu i przyciskach myszy.

Aby informację zawartą w komunikatach systemu Windows można było właściwie wykorzystać, trzeba przechowywać ją tak, by była dostępna podczas tworzenia każdej klatki animacji, a nie jedynie w pętli przetwarzania komunikatów. W celu reprezentacji stanu klawiatury można utworzyć tablicę złożoną z 256 elementów o początkowej wartości false. Obsługując komunikat WM\_KEYDOWN będzie można zmieniać wartość odpowiedniego elementu tablicy na true. Po otrzymaniu komunikatu WM\_KEYUP zmieni się ją z powrotem na false. Przy tworzeniu grafiki będzie można w dowolnym momencie sprawdzić w tej tablicy, czy dany klawisz jest naciśnięty. Wadą tego rozwiązania jest to, że jeśli użytkownik naciśnie i zwolni klawisz odpowiednio szybko (czyli, gdy komunikaty WM\_KEYDOWN i WM\_KEYUP

pojawią się podczas tego samego przebiegu pętli tworzącej klatki animacji), to aplikacja może nie zauważyc tego zdarzenia. Zakładając jednak, że tempo tworzenia animacji wynosi co najmniej 20 klatek na sekundę, wydaje się, że sytuacja taka jest mało prawdopodobna.

Mimo że komunikaty systemu Windows sprawdzają się jako źródło informacji wejściowej w wielu aplikacjach, to w przypadku gier okazuje się, że posiadają zasadniczą wadę — opóźnienie. Wada ta dotyczy nie tylko komunikatów związanych z urządzeniami wejścia, ale komunikatów systemu w ogóle. Przy dużym obciążeniu systemu lub znacznej liczbie komunikatów oczekujących w kolejce okres czasu upływający od naciśnięcia klawisza do momentu jego obsługi może okazać się zbyt długi. Powoduje to utratę płynności gry. Wystąpieniu takiej sytuacji trzeba zapobiec.

## Interfejs programowy Win32

Jak już wspomniano, w wielu przypadkach wygodniej jest samodzielnie sprawdzać stan urządzenia wejścia, niż polegać jedynie na powiadomieniach urządzenia występujących wtedy, gdy użytkownik wykona jakieś działanie. Opanowana przecież została umiejętność obsługi urządzenia wejścia z wykorzystaniem komunikatów systemu Windows. Okazuje się jednak, że często można sprawdzać stan urządzenia bezpośrednio, czyli z pominięciem systemu komunikatów. Interfejs programowy Win32 udostępnia funkcje umożliwiające sprawdzanie stanu klawiatury oraz manipulatorów. Istnieje także szereg funkcji sprawdzania stanu myszy, ale akurat w jej przypadku łatwiej jednak jest korzystać z komunikatów systemu Windows. Należy przyjrzeć się zatem sposobowi obsługi klawiatury i manipulatorów za pomocą funkcji interfejsu Win32.

### Win32 i obsługa klawiatury

Interfejs Win32 posiada wiele funkcji umożliwiających odczyt danych wprowadzanych za pomocą klawiatury. Dla omawianych zastosowań interesująca będzie tylko jedna z nich:

```
SHORT GetAsyncKeyState(int vKey);
```

Wartość zwracana przez funkcję GetAsyncKeyState() zawiera dwie informacje. Jeśli ustawiony jest jej najbardziej znaczący bit, oznacza to, że klawisz vKey jest naciśnięty. Jeśli natomiast ustawiony jest najmniej znaczący bit zwracanej wartości, oznacza to, że klawisz vKey nie był naciśnięty podczas ostatniego wywołania funkcji GetAsyncKeyState(). Ponieważ najbardziej interesująca jest informacja o tym, czy klawisz jest naciśnięty, to wygodnie będzie posłużyć się następującymi makrodefinicjami:

```
#define KEY_DOWN(vKey) (GetAsyncKeyState(vKey) & 0x8000) ? true : false  
#define KEY_UP(vKey) (GetAsyncKeyState(vKey) & 0x8000) ? false : true
```

Parametr przekazywany funkcji GetAsyncKeyState() (oraz makrom) reprezentuje jedną z 256 wartości wirtualnych kodów klawiszy (w rzeczywistości jest ich mniej niż 256, ponieważ niektóre wartości nie są używane). Najczęściej używane kody przedstawione zostały w tabeli 16.2. Należy zwrócić uwagę na to, że w tabeli nie zostały wymienione wartości alfanumeryczne (czyli cyfry i litery), ponieważ w ich przypadku wirtualne kody odpowiadają ich kodom ASCII i wobec tego można używać samych znaków zamiast wirtualnych kodów (czyli 1, 2, 3 ..., i A, B, C ...). Jednak w przypadku liter alfabetu jest to prawda tylko w przypadku dużych liter.

**Tabela 16.2.** Najczęściej używane kody wirtualnych klawiszy

Kod	Klawisz
VK_BACK	Cofania
VK_TAB	Tabulacji
VK_CLEAR	Usuwanie
VK_RETURN	<i>Enter</i>
VK_SHIFT	<i>Shift</i>
VK_CONTROL	<i>Ctrl</i> (oba klawisze)
VK_MENU	<i>Alt</i> (oba klawisze)
VK_PAUSE	<i>Pause</i>
VK_CAPITAL	<i>Caps Lock</i>
VK_ESCAPE	<i>Esc</i>
VK_SPACE	<i>Spacja</i>
VK_PRIOR	<i>Page Up</i>
VK_NEXT	<i>Page Down</i>
VK_END	<i>End</i>
VK_HOME	<i>Home</i>
VK_LEFT	Strzałka w lewo
VK_UP	Strzałka w górę
VK_RIGHT	Strzałka w prawo
VK_DOWN	Strzałka w dół
VK_SNAPSHOT	<i>Print Screen</i>
VK_INSERT	<i>Insert</i>
VK_DELETE	<i>Delete</i>
VK_HELP	<i>Help</i>
VK_LWIN	Klawisz odpowiadający naciśnięciu lewego przycisku myszy
VK_RWIN	Klawisz odpowiadający naciśnięciu prawego przycisku myszy
VK_APPS	Klawisz aplikacji
VK_NUMPAD0	Klawisz numeryczny 0
VK_NUMPAD1	Klawisz numeryczny 1
VK_NUMPAD2	Klawisz numeryczny 2
VK_NUMPAD3	Klawisz numeryczny 3
VK_NUMPAD4	Klawisz numeryczny 4
VK_NUMPAD5	Klawisz numeryczny 5
VK_NUMPAD6	Klawisz numeryczny 6
VK_NUMPAD7	Klawisz numeryczny 7
VK_NUMPAD8	Klawisz numeryczny 8
VK_NUMPAD9	Klawisz numeryczny 9
VK_MULTIPLY	Klawisz mnożenia

**Tabela 16.2.** Najczęściej używane kody wirtualnych klawiszy — ciąg dalszy

Kod	Klawisz
VK_ADD	Klawisz dodawania
VK_SEPARATOR	Klawisz separatora
VK_SUBTRACT	Klawisz odejmowania
VK_DECIMAL	Klawisz kropki dziesiętnej
VK_DIVIDE	Klawisz dzielenia
VK_Fxx	Klawisz funkcyjny Fxx ( <i>F1</i> , <i>F2</i> i tak dalej)
VK_NUMLOCK	<i>Num Lock</i>
VK_SCROLL	<i>Scroll Lock</i>
VK_LSHIFT	Lewy klawisz <i>Shift</i>
VK_RSHIFT	Prawy klawisz <i>Shift</i>
VK_LCONTROL	Lewy klawisz <i>Ctrl</i>
VK_RCONTROL	Prawy klawisz <i>Ctrl</i>

Funkcja `GetKeyState()` umożliwia realizację tego samego zadania na wyższym poziomie przetwarzania. Jednak tworzy ona informację o stanie klawiszy na podstawie komunikatów systemu Windows i wobec tego także może być źródłem niepożądanych opóźnień w interakcji z użytkownikiem.

Funkcja `GetAsyncKeyState()` umożliwia sprawdzanie stanu klawiatury w dowolnym momencie. Jeśli będzie sprawdzany stan klawiatury za każdym razem, gdy tworzona będzie kolejna klatka animacji, to pozwoli to na błyskawiczną reakcję na działania użytkownika.

## Win32 i obsługa manipulatorów

Interfejs Win32 udostępnia niewielki zestaw funkcji umożliwiających obsługę manipulatorów (jednego lub dwóch). Aby z nich skorzystać, należy dołączyć do programu źródłowego plik nagłówkowy `mmsystem.h` oraz do programu wykonywalnego bibliotekę `winmm.lib`. Korzystając z tych funkcji aplikacja może bezpośrednio sprawdzać stan manipulatora, a także być zawiadamiana za pomocą komunikatów. Podstawową wadą funkcji obsługi manipulatora udostępnianych przez Win32 jest ich niska efektywność oraz brak obsługi dodatkowych możliwości obecnie oferowanych urządzeń. Z tego powodu przydatność tych funkcji w grach jest bardzo ograniczona. Dlatego też zamiast omawiać szczegółowo wszystkie dostępne funkcje, można poprzestać na zaprezentowaniu możliwości sprawdzania stanu manipulatora.

Bieżący stan manipulatora można sprawdzić za pomocą funkcji:

```
MMRESULT joyGetPos(UNIT joyID, LPJOYINFO pji);
```

Jeśli jej wykonanie przebiegnie pomyślnie, to zwróci ona wartość `JOYERR_NOERROR`. W przeciwnym razie zwróci kod błędu. Parametr `joyID` określa wybrany manipulator i może przyjmować wartość `JOYSTICKID1` lub `JOYSTICKID2`. Parametr `pji` wskazuje strukturę typu `JOYINFO` zdefiniowanego w następujący sposób:

```
typedef struct
{
    UINT wXpos;      // współrzędna x
    UINT wYpos;      // współrzędna y
    UINT wZpos;      // współrzędna z
    UINT wButtons;   // znaczniki określające stan przycisków manipulatora
} JOYINFO;
```

Pole `wButtons` zawiera kombinację następujących znaczników reprezentujących naciśnięcie poszczególnych przycisków manipulatora: `JOY_BUTTON1`, `JOY_BUTTON2`, `JOY_BUTTON3` lub `JOY_BUTTON4`. Informacja ta jest wystarczająca jedynie w przypadku korzystania z podstawowych funkcji manipulatorów i nie uwzględnia dodatkowych możliwości dostępnych obecnie urządzeń. Interfejs Win32 posiada funkcję `joyGetPosEx()`, która umożliwia uzyskanie dodatkowych informacji o stanie bardziej zaawansowanych manipulatorów. Jednak interfejs DirectInput dysponuje bardziej uniwersalnymi rozwiązaniami.

## DirectInput

Przedstawione dotąd metody uzyskiwania informacji wejścia są łatwe w użyciu i sprawdzają się w wielu zastosowaniach. Jednak wykorzystanie interfejsu programowego DirectInput ma w stosunku do nich kilka istotnych zalet:

- ◆ umożliwia pełen dostęp do wszystkich możliwości zaawansowanych urządzeń wejścia, takich jak na przykład siłowe sprzężenie zwrotne;
- ◆ zapewnia bezpośredni dostęp do sprzętu wejścia umożliwiając tym samym szybką jego obsługę oraz pełną kontrolę urządzenia (efektywny dostęp do urządzeń wejścia umożliwia ich obsługę przez aplikację nawet, gdy wykonywana jest ona w tle);
- ◆ w wersji DirectX 8 aplikacja nie musi nawet znać typu obsługiwanej urządzenia, aby prawidłowo pobierać z niego dane, co pozwala uniknąć tworzenia rozbudowanego kodu dla obsługi różnych urządzeń.

Interfejs DirectInput jest wyjątkowo rozbudowany, ponieważ posiada wiele zaawansowanych możliwości i stanowi próbę zapewnienia obsługi dowolnego urządzenia wejścia. Z tego też względu nie trzeba omawiać go w całości. Zamiast przedstawić wyspecjalizowane możliwości, większości których i tak prawdopodobnie nigdy się nie będzie wykorzystywać, można skoncentrować się na omówieniu najbardziej przydatnych aspektów interfejsu. Również większość tabel zamieszczonych w tym rozdziale przedstawiąć będzie jedynie wybrane znaczniki i opcje. Informację o pozostałych zawiera dokumentacja pakietu SDK.

## Inicjacja interfejsu DirectInput

DirectInput, podobnie jak pozostałe składniki DirectX, oparty jest na modelu obiektowym COM. Model COM (*Common Object Model*) jest złożonym modelem komunikacji i komponentów lansowanym przez firmę Microsoft. Jego omówieniu można by poświęcić osobną książkę, ale na szczęście jego znajomość nie jest potrzebna do posługiwania się

interfejsem DirectX. Świadomość faktu, że DirectX używa modelu COM przydaje się o tyle, że pozwala uniknąć zaskoczenia metodami i terminologią stosowaną przez DirectX.

Jak wspomniano na początku tej książki, DirectX składa się z szeregu komponentów. Większość z komponentów wysokiego poziomu odpowiada różnym rodzajom urządzeń. I tak na przykład Direct3D związane jest z obsługą karty graficznej, DirectSound służy do wykorzystania możliwości karty dźwiękowej. Wyjątkiem od tej reguły jest interfejs DirectInput, ponieważ nie jest on związany z obsługą konkretnego rodzaju sprzętu. Interfejs ten umożliwia dopiero tworzenie obiektów DirectInputDevice reprezentujących różne urządzenia wejścia, takie jak na przykład klawiatura, mysz czy manipulatory. Zanim jednak utworzone zostaną obiekty DirectInputDevice, trzeba najpierw utworzyć i zainicjować obiekt DirectInput.

Aby korzystać z interfejsu DirectInput, należy dołączyć do tekstu źródłowego programu plik nagłówkowy *dinput.h*, a do programu wykonywalnego biblioteki *dinput8.lib* oraz *dGUID.lib* (ta ostatnia używana jest przez wszystkie komponenty DirectX).

Aby zainicjować obiekt DirectInput, trzeba najpierw uzyskać jego interfejs korzystając z funkcji *DirectInput8Create()*:

```
HRESULT WINAPI DirectInput8Create(
    HINSTANCE hinst,
    DWORD dwVersion,
    REFIID riidIf,
    LPVOID *ppv0ut,
    LPUNKNOWN punkOuter);
```

Niestety, interfejsy rodziny DirectX są mało eleganckie i słabo czytelne w porównaniu z interfejsem OpenGL. W obecnej wersji interfejsu DirectX wprowadzono poprawki w stosunku do wersji poprzednich.

Parametr *hInst* stanowi uchwyt aplikacji (lub biblioteki DLL). Parametr *dwVersion* reprezentuje wykorzystywaną wersję interfejsu DirectInput. Standardowo nadaje mu się wartość *DIRECTINPUT\_VERSION*, która zdefiniowana jest w pliku nagłówkowym *dinput.h*. Jeśli programista zamierza korzystać ze starszej wersji, to może podać inną wartość parametru. Jednak tworząc nowe programy z reguły wykorzystywana będzie bieżąca wersja interfejsu. Parametr *riidIf* jest unikatowym identyfikatorem określającym interfejs DirectInput, który będzie wykorzystywany. W naszym przypadku będzie miał on wartość *IID\_IDirectInput8*. Parametr *ppv0ut* jest wskaźnikiem zmiennej, która będzie wskazywać pożądany interfejs, jeśli wywołanie funkcji *DirectInput8Create()* zakończy się pomyślnie. W przypadku 8. wersji interfejsu DirectInput typem tej zmiennej będzie *LPDIRECTINPUT8* i będzie ona wskazywać interfejs wykorzystywany do tworzenia obiektów DirectInputDevice. Ostatni z parametrów funkcji, *punkOuter*, wykorzystywany jest w przypadku agregacji. Ponieważ nie będzie ona stosowana, to parametrowi *punkOuter* nadać można zawsze wartość *NULL*.

## Wartości zwracane przez funkcje DirectInput

Większość funkcji DirectInput omawianych w tym rozdziale może zwracać takie same wartości, dlatego też zamiast omawiać je przy okazji przedstawiania kolejnych funkcji, można zrobić to teraz.

Jeśli wykonania funkcji DirectInput przebiegło pomyślnie, to zwraca ona wartość DI\_OK. W przeciwnym razie zwraca ona jeden z kilkunastu możliwych kodów błędu opisanych w dokumentacji pakietu SDK DirectX. Wielu programistów korzystających z DirectX nie sprawdza rodzaju kodu błędu, lecz korzysta z makra FAILED. Zwraca ono wartość true, jeśli funkcja DirectX zwróciła kod błędów i wartość false w przypadku, gdy zwróciła ona wartość ID\_OK. Fragment kodu służący do uzyskania interfejsu DirectInput będzie wtedy wyglądać następująco:

```
HRESULT result;
LPDIRECTINPUT8 pDirectInput;

if (FAILED(result = DirectInput8Create(g_hInstance, DIRECTINPUT_VERSION,
                                         IID_IDirectInput8,
                                         (void**)&pDirectInput, NULL)))
{
    // obsługa błędów
}
```

Rzutowanie wskaźnika pDirectInput na typ (void\*\*) jest konieczne także w przypadku kilku innych funkcji DirectX. Funkcja DirectInput8Create() używa podwójnego wskaźnika typu void w celu zachowania zgodności z poprzednimi wersjami DirectX. Jeśli nie zostanie wykonane rzutowanie wskaźnika typu LPDIRECTINPUT8 do typu void \*\*, to komplikacja zakończy się błędem.

Wywołanie funkcji DirectInput8Create() udostępnia interfejs obiektu DirectInput. Obiekt ten nie oferuje żadnej funkcjonalności związanej z obsługą urządzeń wejścia, ale jest niezbędny do tworzenia obiektów DirectInputDevice reprezentujących te urządzenia. Obiekty te pozwalają pobierać dane z takich urządzeń jak klawiatura, mysz czy manipulatory.

## Korzystanie z DirectInput

Dysponując interfejsem obiektu DirectInput można przystąpić do tworzenia obiektów reprezentujących urządzenie wejścia. Dla każdego urządzenia wykorzystywanego w programie trzeba utworzyć osobny obiekt DirectInputDevice.

### Dodawanie urządzeń

Obiekty DirectInputDevice są abstrakcją rzeczywistych urządzeń wejścia i zaopatrują w metody umożliwiające interakcję z tymi urządzeniami i pobieranie z nich danych. Istotna zaleta zastosowania obiektów DirectInputDevice polega na tym, że jeśli stworzony zostanie na przykład obiekt reprezentujący manipulator, to nie trzeba będzie interesować się jego typem i tworzyć procedur obsług różnych manipulatorów. Interfejs obiektu umożliwia uzyskanie informacji o liczbie przycisków manipulatora, o tym, czy używa on siłowego sprzężenia zwrotnego oraz o innych jego możliwościach.

Obiekt DirectInputDevice tworzy się w kilku etapach. Niektóre z nich są dość zawiłe i wymagają szerszego omówienia. Poniżej przedstawiamy ich listę:

- ◆ stworzenie wyliczenia dostępnych urządzeń (opcjonalnie);
- ◆ stworzenie obiektu reprezentującego urządzenie;
- ◆ sprawdzenie możliwości urządzenia (opcjonalnie);
- ◆ stworzenie wyliczenia obiektów (opcjonalnie);
- ◆ określenie formatu danych urządzenia;
- ◆ określenie poziomu współpracy;
- ◆ zmodyfikowanie właściwości urządzenia (opcjonalnie);
- ◆ zajęcie urządzenia.

## Tworzenie urządzeń

Urządzenia tworzy się za pomocą następującej metody interfejsu obiektu DirectInput:

```
HRESULT IDirectInput8::CreateDevice(  
    REFGUID rguid,  
    LPDIRECTINPUTDEVICE *ppDirectInputDevice,  
    LPUNKNOWN punkOuter);
```

Parametr *rguid* jest unikatowym identyfikatorem urządzenia, dla którego tworzy się obiekt. Parametr *ppDirectInputDevice* wskazuje wskaźnik interfejsu urządzenia, jeśli wykonanie metody *CreateDevice()* zakończyło się pomyślnie. Jak już wcześniej wspomniano, parametrowi *punkOuter* będzie zawsze nadawana wartość NULL.

Pierwszy z parametrów metody *CreateDevice()* wymaga szerszego omówienia. Aby uzyskać interfejs konkretnego urządzenia, trzeba posiadać jego identyfikator przekazywany właśnie za pomocą pierwszego parametru metody. DirectInput definiuje następujące identyfikatory urządzeń:

- ◆ GUID\_SysKeyboard identyfikujący domyślną klawiaturę systemu;
- ◆ GUID\_SysMouse identyfikujący domyślną mysz systemu.

Jeśli system posiada więcej niż jedną klawiaturę albo mysz, to będą one reprezentowane za pomocą jednego urządzenia systemowego.

Identyfikatory pozostałych typów urządzeń uzyskuje się po pobraniu listy urządzeń dostępnych w systemie.

## Tworzenie wyliczenia urządzeń

Interfejs DirectInput posiada następującą metodę umożliwiającą tworzenie wyliczeń urządzeń:

```
HRESULT IDirectInput8::EnumDevices(  
    DWORD dwDevType,  
    PDIENUMCALLBACK lpCallback,  
    LPVOID pvRef,  
    DWORD dwFlags);
```

Metoda ta wprowadza wiele nowych zagadnień i dlatego jej omówienie zajmie sporo miejsca.

## **dwDevType**

Pierwszy parametr metody, *dwDevType*, pozwala określić typ urządzeń, które ma zawierać wyliczenie. Urządzenia o innym typie nie zostaną wtedy uwzględnione. Najczęściej używane typy i podtypy urządzeń przedstawia tabela 16.3 (kompletną listę typów urządzeń podaje dokumentacja DirectX SDK).

**Tabela 16.3.** Typy urządzeń *DirectInput*

Typ	Opis
DI8DEVTYPE_1STPERSON	Urządzenia nadające się do gier akcji. Dostępne są następujące podtypy: DI8DEVTYPE1STPERSON_LIMITED: nie posiada wystarczającej liczby obiektów urządzenia dla odwzorowania akcji; DI8DEVTYPE1STPERSON_SHOOTER: urządzenie umożliwiające strzelanie; DI8DEVTYPE1STPERSON_SIXDOF: urządzenia dysponujące sześcioma stopniami swobody; DI8DEVTYPE1STPERSON_UNKNOWN: typ nieznany.
DI8DEVTYPE_DEVICE	Urządzenia, które nie należą do innych typów.
DI8DEVTYPE_DRIVING	Urządzenia sterowania pojazdem. Dostępne są następujące podtypy: DI8DEVTYPEPEDRIVING_COMBINEDPEDALS: wspólny pedał hamulca i gazu; DI8DEVTYPEPEDRIVING_DUALPEDALS: osobne педаły hamulca i gazu; DI8DEVTYPEPEDRIVING_HANDHELD: ręczne sterowanie pojazdem; DI8DEVTYPEPEDRIVING_LIMITED: nie posiada wystarczającej liczby obiektów urządzenia dla odwzorowania akcji; DI8DEVTYPEPEDRIVING_THREEPEDALS: osobne педалы hamulca, sprzęgła i gazu.
DI8DEVTYPE_FLIGHT	Urządzenia używane przez symulatory lotu. Dostępne są następujące podtypy: DI8DEVTYPEFLIGHT_LIMITED: nie posiada wystarczającej liczby obiektów urządzenia dla odwzorowania akcji; DI8DEVTYPEFLIGHT_RC: urządzenie przypominające w działaniu aparaturę zdalnego sterowania modelem; DI8DEVTYPEFLIGHT_STICK: drążek sterowy; DI8DEVTYPEFLIGHT_YOKE: wolant.
DI8DEVTYPE_GAMEPAD	Tabliczka. Dostępne są następujące podtypy: DI8DEVTYPEGAMEPAD_LIMITED: nie posiada wystarczającej liczby obiektów urządzenia dla odwzorowania akcji; DI8DEVTYPEGAMEPAD_STANDARD: tabliczka o wystarczającej liczbie obiektów urządzenia dla odwzorowania akcji; DI8DEVTYPEGAMEPAD_TILT: tabliczka zwracająca podczas przechylania współrzędne na osiach x i y.
DI8DEVTYPE_JOYSTICK	Manipulator. Dostępne są następujące podtypy: DI8DEVTYPEJOYSTICK_LIMITED: nie posiada wystarczającej liczby obiektów urządzenia dla odwzorowania akcji; DI8DEVTYPEJOYSTICK_STANDARD: zwykły manipulator o wystarczającej liczbie obiektów urządzenia dla odwzorowania akcji.

**Tabela 16.3.** Typy urządzeń DirectX — ciąg dalszy

Typ	Opis
DI8DEVTYPE_KEYBOARD	Klawiatura. Dostępne są następujące podtypy: DI8DEVTYPEKEYBOARD_UNKNOWN: klawiatura nieznanego typu; DI8DEVTYPEKEYBOARD_PCXT: klawiatura PC/XT o 83 klawiszach; DI8DEVTYPEKEYBOARD_OLIVETTI: klawiatura Olivetti o 102 klawiszach; DI8DEVTYPEKEYBOARD_PCAT: klawiatura PC/AT o 84 klawiszach; DI8DEVTYPEKEYBOARD_PCNH: klawiatura o 101 lub 102 klawiszach; DI8DEVTYPEKEYBOARD_NOKIA1050: klawiatura Nokia 1050; DI8DEVTYPEKEYBOARD_NOKIA9140: klawiatura Nokia 9149.
DI8DEVTYPE_MOUSE	Mysz lub podobne urządzenie: DI8DEVTYPEMONOUSE_ABSOLUTE: mysz zwracająca współrzędne bezwzględne; DI8DEVTYPEMONOUSE_FINGERSTICK: wodzik; DI8DEVTYPEMONOUSE_TOUCHPAD: tabliczka dotykowa; DI8DEVTYPEMONOUSE_TRACKBALL: kot (manipulator kulowy); DI8DEVTYPEMONOUSE_TRADITIONAL: mysz tradycyjna; DI8DEVTYPEMONOUSE_UNKNOWN: podtyp nieokreślony.
DI8DEVTYPE_SCREENPOINTER	Urządzenie umożliwiające wskazywanie na ekranie: DI8DEVTYPESCREENPTR_UNKNOWN: podtyp nieokreślony; DI8DEVTYPESCREENPTR_LIGHTGUN: pistolet świetlny; DI8DEVTYPESCREENPTR_LIGHTPEN: pióro świetlne; DI8DEVTYPESCREENPTR_TOUCH: ekran dotykowy.
DI8DEVTYPE_SUPPLEMENTAL	Urządzenie dodatkowe przewidziane do wykorzystania w połączeniu z zasadniczym urządzeniem wejścia: DI8DEVTYPESUPPLEMENTAL_2NDHANDCONTROLLER: pomocnicze urządzenie ręcznego sterowania; DI8DEVTYPESUPPLEMENTAL_COMBINEDPEDALS: pojedynczy pedał gazu i hamulca; DI8DEVTYPESUPPLEMENTAL_DUALPEDALS: osobny pedał gazu i hamulca; DI8DEVTYPESUPPLEMENTAL_HANDTRACKER: urządzenie śledzące ruchy ręki; DI8DEVTYPESUPPLEMENTAL_HEADTRACKER: urządzenie śledzące ruchy głowy; DI8DEVTYPESUPPLEMENTAL_RUDDERPEDALS: pedały steru; DI8DEVTYPESUPPLEMENTAL_SHIFTER: zmiana biegów; DI8DEVTYPESUPPLEMENTAL_SHIFTSTICKGATE: zmiana biegów za pomocą przycisków; DI8DEVTYPESUPPLEMENTAL_SPLITTHROTTLE: urządzenie raportujące dwie lub więcej wartości otwarcia przepustnicy; DI8DEVTYPESUPPLEMENTAL_THREEPEDALS: osobne pedały, gazu, hamulca i sprzęgła; DI8DEVTYPESUPPLEMENTAL_THROTTLE: otwarcie przepustnicy; DI8DEVTYPESUPPLEMENTAL_UNKNOWN: podtyp nieokreślony.

Niektóre z wymienionych w tabeli wartości mogą być niezrozumiałe, ale ich przeznaczenie wyjaśni się, gdy zaczną być używane. Oprócz wymienionych typów można używać także wartości obejmujących wiele typów:

- ◆ DI8DEVCLASS\_ALL — wszystkie typy urządzeń;
- ◆ DI8DEVCLASS\_DEVICE — urządzenia, które nie należą do typów określonych przez wymienione niżej wartości;
- ◆ DI8DEVCLASS\_GAMECTRL — urządzenia wejścia wykorzystywane w grach;
- ◆ DI8DEVCLASS\_KEYBOARD — klawiatury; ekwiwalent typu DI8DEVTYP\_KEYBOARD;
- ◆ DI8DEVCLASS\_POINTER — myszy i urządzenia wskazujące (na przykład typy DI8DEVTYP\_MOUSE i DI8DEVTYP\_SCREENPOINTER).

### **IpCallBack**

Drugi z parametrów metody EnumDevices() jest wskaźnikiem funkcji wywoływanej zwrotnie zdefiniowanej przez programistę. Funkcja ta będzie wywoływana jeden raz dla każdego urządzenia wejścia znalezionego w systemie, którego typ zgodny będzie z typem określonym za pomocą parametru *dwDevType*. Zwykle funkcja będzie umieszczać dane o kolejnych urządzeniach w pewnej strukturze danych. Po zakończeniu tego procesu program lub użytkownik wybierze jedno z urządzeń (więcej na ten temat za chwilę). Funkcja zwrotna musi posiadać następującą postać:

```
BOOL CALLBACK MyDIDevicesCallback(LPCDIDEVICEINSTANCE lpddi, LPVOID pvRef);
```

Parametr *lpddi* wskazuje strukturę typu DIDEVICEINSTANCE, która zawiera informację o urządzeniu, w tym jego identyfikator. Parametr *pvRef* posiada taką samą wartość jak parametr *pvRef*, który przekazany został metodzie EnumDevices() (zwykle jest to wskaźnik do struktury danych, w której umieszczono informacje o kolejnych urządzeniach). Funkcja ta powinna zwrócić wartość DIENUM\_CONTINUE, jeśli proces wyliczania urządzeń ma być kontynuowany. Wartość DIENUM\_STOP powoduje przerwanie procesu wyliczania (na przykład w przypadku, gdy znaleziono już poszukiwane urządzenie).

### **dwFlags**

Ostatni z parametrów metody EnumDevices() reprezentuje zbiór znaczników, które służą do ograniczenia zakresu poszukiwanych urządzeń wejścia. Może być on kombinacją przykładowych wartości przedstawionych w tabeli 16.4.

**Tabela 16.4.** Znaczniki wyliczenia urządzeń

Wartość	Znaczenie
DIEDFL_ALLDEVICES	Wyliczenie wszystkich urządzeń (wartość domyślna)
DIEDFL_ATTACHEDONLY	Wyliczenie tylko tych urządzeń, które są podłączone i zainstalowane
DIEDFL_FORCEFEEDBACK	Wyliczenie tylko tych urządzeń, które dysponują siłowym sprzężeniem zwrotnym

Za chwilę przeanalizowany zostanie przykład kodu korzystającego z wyliczeń urządzeń. Przedstawione dotychczas informacje powinny jednak dać już ogólny pogląd na sposób działania wyliczeń. Wykorzystywane są one w celu uzyskania identyfikatorów urządzeń innych niż standardowa mysz i klawiatura systemu.

## Sprawdzanie możliwości urządzenia

Typ urządzenia, który podaje się przy wywołaniu metody `EnumDevices()`, pozwala określić ogólną charakterystykę interesujących programistę urządzeń. Czasami jednak zaczodzi konieczność dokładniejszego sprawdzenia możliwości urządzenia, zwłaszcza w przypadku, gdy określa się ogólny typ urządzenia. Po utworzeniu urządzenia można zweryfikować to, czy posiada ono wymagane możliwości, a także to, czy jest w danym momencie podłączone do komputera. Służy do tego metoda `IDirectInputDevice8::GetCapabilities()` zdefiniowana jak poniżej:

```
HRESULT IDirectInputDevice8::GetCapabilities(LPDIDEVCAPS pDIDevCaps);
```

Parametr `pDIDevCaps` wskazuje strukturę typu `DIDEVCAPS`, która zostanie wypełniona informacjami o możliwościach urządzenia, jeśli wywołanie metody `GetCapabilities()` zakończy się pomyślnie. Struktura `DIDEVCAPS` posiada następujące pola:

- ◆ `dwSize` — rozmiar struktury `DIDEVCAPS` w bajtach (polu temu trzeba nadać wartość przed wywołaniem metody `GetCapabilities()`);
- ◆ `dwFlags` — kombinacja znaczników (najważniejsze z nich przedstawia tabela 16.5);
- ◆ `dwDevType` — typ urządzenia (dowolna z wartości wymienionych w tabeli 16.3);
- ◆ `dwAxes` — liczba osi urządzenia;
- ◆ `dwButtons` — liczba przycisków urządzenia;
- ◆ `dwPovs` — liczba kontrolek punktu widzenia;
- ◆ `dwFFSamplePeriod` — w przypadku urządzeń wykorzystujących siłowe sprzężenie zwrotne reprezentuje najmniejszą dopuszczalną liczbę mikrosekund pomiędzy kolejnymi poleceniami określającymi wielkość siły;
- ◆ `dwFFMinTimeResolution` — w przypadku urządzeń wykorzystujących siłowe sprzężenie zwrotne reprezentuje rozdzielcość licznika czasu w mikrosekundach;
- ◆ `dwFirmwareRevision`, `dwHardwareRevision`, `dwFFDriverRevision` — numery wersji oprogramowania, urządzenia i jego sterownika.

**Tabela 16.5.** Znaczniki możliwości urządzenia

Znacznik	Znaczenie
<code>DIDC_ALIAS</code>	Urządzenie jest w rzeczywistości synonimem innego urządzenia
<code>DIDC_ATTACHED</code>	Urządzenie jest podłączone do komputera
<code>DIDC_FORCEFEEDBACK</code>	Urządzenie wykorzystuje siłowe sprzężenie zwrotne (dla urządzeń tych istnieje szereg dodatkowych znaczników, które nie będą szczegółowo omawiane, warto jednak zaznaczyć, że są to: <code>DIDC_DEADBAND</code> , <code>DIDC_FFADE</code> , <code>DIDC_FFATTACK</code> , <code>DIDC_POSNEGCOEFFICIENTS</code> , <code>DIDC_POSNEGSATURATION</code> , <code>DIDC_SATURATION</code> i <code>DIDC_STARTDELAY</code> )
<code>DIDC_POLLEDDATAFORMAT</code>	W celu uzyskania danych należy „odpytywać” jeden lub więcej obiektów w bieżącym formacie
<code>DIDC_POLLEDDEVICE</code>	W celu uzyskania danych należy „odpytywać” jeden lub więcej obiektów urządzeń

Informacje, które zawiera struktura DIDEVCAPS, po wykonaniu metody GetCapabilities() wykorzystuje się w celu sprawdzenia możliwości urządzenia wejścia.

## Wyliczenia obiektów

Elementy funkcjonalne urządzeń wejścia, takie jak przyciski, klawisze, suwaki i tym podobne, reprezentowane są za pomocą osobnych obiektów. Obiekty te nie są potrzebne w przypadku standardowych urządzeń wejścia, takich jak klawiatura czy mysz. Jednak konfiguracja manipulatorów i innych urządzeń wejścia może być bardzo zróżnicowana. Aby sprawdzić, jakie elementy są rzeczywiście dostępne korzysta się z metody EnumObjects():

```
HRESULT IDirectInputDevice8::EnumObjects(LPDIDENUMDEVICEOBJECTSCALLBACK lpCallback,
                                         LPVOID pvRef,
                                         DWORD dwFlags);
```

Parametr *lpCallback* jest wskaźnikiem funkcji zwrotnej. Parametr *pvRef* jest wskaźnikiem obiektu, który zostanie przekazany funkcji zwrotnej. Parametr *dwFlags* służy do określenia typów obiektów, które zawierać będzie wyliczenie i stanowi kombinację znaczników. Najważniejsze z nich zaprezentowane zostały w tabeli 16.6.

**Tabela 16.6.** Znaczniki typu obiektów

Znacznik	Znaczenie
DIDFT_ABSAXIS	Oś bezwzględna
DIDFT_ALL	Wszystkie typy obiektów
DIDFT_AXIS	Dowolna oś (względna lub bezwzględna)
DIDFT_BUTTON	Przyciski (naciskane lub przełączniki)
DIDFT_NODATA	Obiekty, które nie generują danych
DIDFT_OUTPUT	Obiekty, do których można wysyłać dane (za pomocą metody SendDeviceData())
DIDFT_POV	Kontrolka punktu widzenia
DIDFT_PSHBUTTON	Przycisk
DIDFT_RELAXIS	Oś względna
DIDFT_TGLBUTTON	Przycisk typu przełącznik (zmienia stan po każdym naciśnięciu)

Prototyp funkcji zwrotnej wygląda następująco:

```
BOOL CALLBACK DIEnumDeviceObjectsCallback(LPCDIDEVICEOBJECTINSTANCE lpddoi,
                                         LPVOID pvRef);
```

Parametr *lpddoi* jest wskaźnikiem struktury typu DIDEVICEOBJECTINSTANCE (patrz: dokumentacja SDK). Parametr *pvRef* posiada wartość, która przekazana została wcześniej metodzie EnumObjects().

## Określanie formatu danych urządzenia

Zanim urządzenie zacznie być używane, wymagane jest określenie jego formatu danych (o ile nie korzysta się z map akcji). DirectInput umożliwia zastosowanie różnych формataw danych pobieranych z urządzenia wejścia. Aby określić format danych urządzenia, trzeba wywołać następującą funkcję:

```
HRESULT IDirectInputDevice::SetDataFormat(LPCDIDATAFORMAT lpdf);
```

Parametr *lpdf* jest wskaźnikiem struktury typu DIDATAFORMAT definiującej format danych, który będzie używany. DirectX dostarcza kilku gotowych zmiennych globalnych określających format danych dla najczęściej spotykanych urządzeń wejścia:

- ◆ *c\_dfDIKeyboard* — tablica 256 znaków reprezentujących klawisze (jeśli najbardziej znaczący bit znaku jest ustawiony, oznacza to, że klawisz jest naciśnięty); DirectInput definiuje zestaw stałych reprezentujących klawisze, ich identyfikatory mają postać DIK\_ *klawisz* (patrz: dokumentacja DirectX SDK);
- ◆ *c\_dfDIMouse* — mysz o trzech osiach i czterech przyciskach; jej stan reprezentowany jest za pomocą struktury typu DIMOUSESTATE:

```
typedef struct DIMOUSESTATE
{
    LONG lX;           // położenie osi X (względne)
    LONG lY;           // położenie osi Y (względne)
    LONG lZ;           // położenie osi Z (względne)
    BYTE rgButtons[4]; // przyciski; najwyższy bit ustawiony, gdy przycisk naciśnięty
} DIMOUSESTATE, *LPDIMOUSESTATE;
```

- ◆ *c\_dfDIMouse2* — mysz o trzech osiach i ośmiu przyciskach; jej stan reprezentuje strukturę DIMOUSESTATE2, która jest zdefiniowana analogicznie jak struktura DIMOUSESTATE, jedynie tablica rgButtons posiada osiem elementów zamiast czterech;
- ◆ *c\_dfDIJoystick* — manipulator o trzech osiach pozycji i trzech osiach obrotu, dwóch suwakach, dwóch kontrolerach punktu widzenia i 32 przyciskach opisany za pomocą struktury DIJOystate zdefiniowanej jak poniżej:

```
typedef struct DIJOystate
{
    LONG lX;           // ós X (zwykle lewo-prawo)
    LONG lX;           // ós X (zwykle lewo-prawo)
    LONG lX;           // ós X (zwykle lewo-prawo)
    LONG lRx;          // obrót wokół osi x
    LONG lRx;          // obrót wokół osi x
    LONG lRx;          // obrót wokół osi x
    LONG rgSlider[2]; // osie u i v
    DWORD rgdwPOV[4]; // kontrolery kierunkowe
    BYTE rgButtons[32]; // przyciski; najwyższy bit ustawiony, gdy przycisk naciśnięty
} DIJOystate, *LPDIJOystate;
```

- ◆ *c\_dfDIJoystick2* — opisany za pomocą struktury DIJOystate2 (patrz: dokumentacja DirectX SDK).

Zdecydowana większość gier korzysta właśnie z tych gotowych formatów i dlatego nie trzeba omawiać szczegółowo struktury DIDATAFORMAT. Jej dokumentację zawiera pakiet DirectX SDK.

## Określanie poziomu współpracy

Ze względu na wielozadaniowość systemu Windows urządzenia wejściowe mogą być współużytkowane przez wiele aplikacji. Sposób współużytkowania urządzeń wejścia z innymi aplikacjami przez tworzoną tu grę określa się za pomocą poziomu współpracy korzystając z poniższej metody:

```
HRESULT IDirectInputDevice::SetCooperativeLevel(HWND hwnd, DWORD dwFlags);
```

Parametr *hwnd* reprezentuje uchwyty głównego okna programu. Parametr *dwFlags* stanowi kombinację znaczników określających poziom współpracy. Znaczniki te przedstawione zostały w tabeli 16.7.

Tabela 16.7. Znaczniki poziomu współpracy

Znacznik	Znaczenie
DISCL_BACKGROUND	Aplikacja używa dane urządzenie, gdy działa w tle
DISCL_EXCLUSIVE	Aplikacja rezerwuje dostęp do urządzenia na zasadach wyłączności, co oznacza, że w tym czasie żadna inna aplikacja nie może uzyskać dostępu do urządzenia na zasadach wyłączności (możliwy jest inny rodzaj dostępu)
DISCL_FOREGROUND	Aplikacja używa dane urządzenie, gdy działa na pierwszym planie (jeśli jej okno przestanie być aktywne, to aplikacja zwalnia urządzenie)
DISCL_NONEXCLUSIVE	Aplikacja używa dane urządzenie, jeśli nie koliduje to z innymi aplikacjami
DISCL_NOWINKEY	Wyłącza klawisz opatrzony logo systemu Windows (jeśli taki istnieje) zapobiegając niezamierzonymu przerwaniu wykonywania aplikacji

Aplikacja musi wybrać znacznik DISCL\_BACKGROUND lub DISCL\_FOREGROUND oraz DISCL\_EXCLUSIVE lub DISCL\_NONEXCLUSIVE. W większości przypadków używa się znaczników DISCL\_FOREGROUND i DISCL\_EXCLUSIVE. Uzupełnienie ich znacznikiem DISCL\_NOWINKEY jest dobrym rozwiązaniem zwłaszcza w przypadku gier.

## Modyfikacja właściwości urządzenia

DirectInput umożliwia zmianę właściwości urządzenia, takich jak zakres działania, precyza, rozmiary buforów i tym podobne. Modyfikacje te odbywają się za pośrednictwem metody IDirectInputDevice::SetProperty() zdefiniowanej w następujący sposób:

```
HRESULT IDirectInputDevice::SetProperty(REFGUID rguidProp, LPDIRECTPROPHDR pdiph);
```

Parametr *rguidProp* reprezentuje identyfikator właściwości, która ma być zmodyfikowana. W aktualnych zastosowaniach będzie wykorzystywana jedynie możliwość zmiany rozmiaru bufora wejściowego. Właściwość ta identyfikowana jest za pomocą stałej DIPROP\_BUFFERSIZE. Domyślny rozmiar bufora wejścia wynosi 0 i będzie trzeba zmienić go, aby móc korzystać z buforowania danych wejścia (więcej na ten temat w dalszej części rozdziału). Pozostałe właściwości urządzeń wejścia, które można modyfikować, dotyczą głównie manipulatorów. Pełną ich listę zawiera dokumentacja DirectX SDK. Parametr *pdiph* jest wskaźnikiem struktury DIPROPHDR, która zostanie omówiona szczegółowo w dalszej części rozdziału.

## Zajmowanie urządzenia

Aby korzystać z urządzenia, trzeba najpierw uzyskać do niego dostęp, pamiętając o tym, że większość urządzeń wejścia jest już współużytkowana przez działające aplikacje. Dostęp do urządzenia uzyskuje się za pomocą metody `Acquire()`:

```
HRESULT IDirectInputDevice8::Acquire();
```

Metoda ta zwraca wartość `DI_OK`, jeśli uzyskany został dostęp do urządzenia. Jeśli próbowano uzyskać dostęp do urządzenia, z którego korzysta już dana aplikacja, to metoda `Acquire()` zwróci wartość `S_FALSE`. W pozostałych przypadkach zwracany jest jeden z poniższych kodów błędów:

- ◆ `DIERR_INVALIDPARAM` — przed wywołaniem metody `Acquire()` nie została wywołana metoda `SetDataFormat()` lub `SetActionMap()`;
- ◆ `DIERR_NOTINITIALIZED` — przed wywołaniem metody `Acquire()` nie została wykonana inicjalizacja;
- ◆ `DIERR_OTHERAPPHASPRIO` — błąd ten pojawia się, gdy aplikacja posiada prawo korzystania z urządzenia w pierwszym planie i próbuje uzyskać dostęp do urządzenia działając w tle.

Należy zwrócić uwagę, że jeśli aplikacja używa urządzenia wejścia, gdy wykonywana jest na pierwszym planie, to przechodząc do tła zwolni urządzenie. Dlatego też program korzystający z urządzeń wejścia powinien obsługiwać komunikat `WM_ACTIVATE` i ponownie zajmować urządzenie, jeśli jest ono jeszcze potrzebne.

Po zajęciu urządzenie można zacząć pobierać od niego dane.

## Pobieranie danych wejściowych

Dane pobierane od urządzeń wejściowych można pobierać bezpośrednio bądź za pośrednictwem bufora.

### Bezpośredni dostęp do danych

Bezpośredni dostęp do danych umożliwia określenie stanu urządzenia wejścia w interesującym programistę momencie. Jeśli na przykład użytkownik naciska klawisz označający ruch do przodu, to można „przybliżyć” do niego wizerunek wirtualnego świata. Nie jest interesuje przy tym to, jakie klawisze zostały naciśnięte wcześniej. W celu bezpośredniego pobrania danych można skorzystać z metody `GetDeviceState()`:

```
HRESULT IDirectInputDevice8::GetDeviceState(DWORD cbData, LPVOID lpvData);
```

Parametr `cbData` określa liczbę bajtów zajmowanych przez strukturę wskazywaną przez parametr `lpvData`. W strukturze tej zostaną umieszczone dane opisujące bieżący stan urządzenia. Typ tej struktury zależy od parametrów, które zostały przekazane wcześniej metodzie `SetDataFormat()`.

## Buforowanie danych

Bezpośredni dostęp do danych nie jest właściwym rozwiązyaniem w każdej sytuacji. Jego podstawową wadą jest możliwość utraty danych. Jeśli na przykład użytkownik naciśnie klawisz i zwolni go pomiędzy kolejnymi wywołaniami metody GetDeviceState(), to nigdy się o tym nie dowie programista. Oczywiście można zminimalizować prawdopodobieństwo wystąpienia takiej sytuacji, jeśli sprawdza się stan klawiatury odpowiednio często. Alternatywnym rozwiązyaniem jest buforowanie danych. Bufor umożliwia przechowanie kolejnych stanów urządzenia wejścia dając jednocześnie absolutną pewność, że żadne dane nie zostaną utracone. Dane z bufora pobiera się za pomocą metody GetDeviceData() posiadającej następujący prototyp:

```
HRESULT IDirectInputDevice8::GetDeviceData(DWORD cbObjectData,
                                             LPDIDEVICEOBJECTDATA rgdod,
                                             LPDWORD pdwInOut,
                                             DWORD dwFlags);
```

Parametr *cbObjectData* określa liczbę bajtów zajmowanych przez strukturę DIDEVICEOBJECTDATA (patrz: dokumentacja DirectX SDK). Parametr *rgdod* jest wskaźnikiem tablicy takich struktur. Parametr *pdwInOut* ma podwójne zadanie. Wywołując metodę GetDeviceData() należy nadać mu wartość równą liczbie elementów w tablicy *rgdod*. Natomiast po wykonaniu funkcji będzie on zawierać liczbę elementów rzeczywiście umieszczonych w tablicy *rgdod*. Parametr *dwFlags* może posiadać wartość 0, co spowoduje usunięcie pobranych danych z bufora lub wartość DIGDD\_PEEK oznaczającą, że dane pozostaną w buforze. Zwykle korzysta się z pierwszej możliwości.

Wybór bezpośredniego dostępu do danych lub za pośrednictwem bufora zależy wyłącznie od aktualnych potrzeb. W większości przypadków dostęp bezpośredni okazuje się wystarczający.

## „Odpytywanie” urządzeń

Oba omówione sposoby pobierania danych z urządzeń wejścia zwracają dane, które zostały przekazane przez urządzenie do DirectInput. Większość urządzeń przekazuje dane automatycznie, gdy tylko się pojawią. Jednak niektóre urządzenia mogą wymagać „odpytania”, zanim możliwe będzie pobranie danych. Urządzenia takie „odpytuje się” za pomocą metody Poll() o następującym prototypie:

```
HRESULT IDirectInputDevice8::Poll();
```

„Odpytanie” urządzenia powoduje aktualizację jego danych przez DirectInput. Jeśli urządzenie nie wymaga „odpytywania”, to wywołanie metody Poll() nie powoduje żadnego efektu.

## Kończenie pracy z urządzeniem

Poprawne zakończenie pracy programu używającego DirectInput wymaga wykonania kilku operacji. Przede wszystkim należy zwolnić wszystkie urządzenia za pomocą poniższej metody:

```
HRESULT IDirectInputDevice8::Unacquire();
```

Wywołanie tej metody dla urządzenia, które nie zostało zajęte, nie wywołuje żadnego efektu. Można więc wywołać tę metodę kolejno dla wszystkich urządzeń. Z metody tej korzysta się nie tylko przy zakończeniu pracy programu, ale także w sytuacji, gdy aplikacja przechodzi w stan nieaktywny.

Następnie należy zwolnić interfejs każdego z urządzeń. Służy do tego następująca metoda:

```
HRESULT IDirectInputDevice8::Release();
```

Na końcu trzeba jeszcze zwolnić obiekt DirectInput za pomocą poniższej metody:

```
HRESULT IDirectInput8::Release();
```

## Odwzorowania akcji

Omawiając sposób korzystania z DirectInput napisano, że dla każdego typu urządzenia niezbędny jest osobny obiekt DirectInputDevice. W przypadku wersji DirectX 8 nie musi to być prawdą, ponieważ dodano w niej do DirectInput nową warstwę abstrakcji urządzeń wejścia wykorzystującą technologię odwzorowań akcji.

Korzystając z odwzorowań akcji definiuje się zbiór elementów kontrolnych, które może używać dana gra. Na przykład przy tworzeniu programu symulacji lotu będą to elementy kontrolujące lot samolotu, rodzaj używanej broni i tym podobne. Następnie trzeba poinformować DirectInput, że dana aplikacja należy do kategorii symulatorów lotu i zarejestrować dla niej odpowiednie elementy kontrolne. Po uruchomieniu gry DirectInput wykrywa wszystkie podłączone urządzenia wejścia i pozwala użytkownikowi wybrać i skonfigurować te, których chce używać. Na końcu tworzy odwzorowanie wybranych urządzeń na zdefiniowanych przez grę elementach kontrolnych. Zaletą takiego rozwiązania jest to, że nie trzeba pisać procedur obsługi różnych typów urządzeń, a użytkownik może wybrać i używać dowolnego z nich.

Słabym punktem odwzorowań akcji jest złożony proces konfigurowania tego mechanizmu, który nie będzie szczegółowo omawiany. W przykładach DirectInput będzie wykorzystywany w zwykły sposób. Odwzorowania akcji opisane są szczegółowo w dokumentacji pakietu DirectX SDK, a także w artykułach udostępnianych przez firmę Microsoft (patrz: dodatek A).

## Tworzenie podsystemu wejścia

W podrozdziale tym przeanalizowany zostanie sposób tworzenia podsystemu wejścia działającego w oparciu o DirectInput. Podsystem ten będzie wykorzystywany później w tworzonych grach. Będzie się on składać z warstwy zarządzającej oraz obiektów obsługujących klawiaturę i mysz. Większość gier działających na komputerach PC wykorzystuje klawiaturę i mysz i dlatego rozbudowanie podsystemu wejścia o obsługę manipulatorów nie jest konieczne na obecnym etapie.

Warstwa zarządzająca będzie tworzyć minimalną klasę obudowującą interfejs DirectInput i zarządzać obiektami urządzeń, dla których utworzone zostaną osobne klasy. Gry nie wymagają szczególnie złożonych podsystemów wejścia, wobec czego przy projektowaniu istotna będzie jego prostota. Jeśli zajdzie taka potrzeba, to zaprojektowane klasy powinny zapewnić możliwość jego rozbudowy.

Zaproponowane tu rozwiązanie podsystemu wejścia umożliwiać będzie pełną obsługę danych wejścia z możliwością uzyskania bezpośredniego dostępu do obiektów reprezentujących klawiaturę i mysz. Aby skorzystać w aplikacji z tego podsystemu, należy zadeklarować w programie jako obiekt klasy `CInputSystem`, a następnie wywołać metodę `Initialize()` w celu jego inicjacji.



Inicjacji podsystemu wejścia dokonywać będzie metoda `Initialize()`, a nie konstruktor klasy `CInputSystem`. Rozwiązanie takie jest wskazane, ponieważ podsystem wejścia może zostać wtedy zadeklarowany jako obiekt globalny. Konstruktory obiektów globalnych wywoływanie są, zanim wykonana zostanie jakakolwiek inna funkcja, co może doprowadzić do korzystania z niezainicjowanego interfejsu DirectInput.

Po zainicjowaniu podsystemu wejścia przy tworzeniu kolejnej klatki animacji będzie wywoływana za każdym razem metoda aktualizacji `Update()`, a następnie pobierane będą dane za pomocą metod `KeyUp()`, `KeyDown()`, `ButtonUp()`, `GetMouseMovement()`. Gdy okno aplikacji przestanie być wykonywane na pierwszym planie, należy wywołać metodę `UnacquireAll()` zwalniając w ten sposób zajęte urządzenia. Gdy aplikacja powróci do pierwszego planu, zostanie ono zajęte ponownie po wywołaniu metody `AcquireAll()`. Na zakończenie działania programu zwolnić trzeba wszystkie interfejsy i zajmowaną pamięć korzystając z metody `Shutdown()`.

Teraz należy przyjrzeć się implementacji klasy podsystemu wejścia:

```
// w pliku InputSystem.h
#define IS_USEKEYBOARD 1
#define IS_USEMOUSE    2

class CInputSystem
{
public:
    bool Initialize(HWND hwnd, HINSTANCE appInstance, bool isExclusive, DWORD flags = 0);
    bool Shutdown();

    void AcquireAll();
    void UnacquireAll();

    CKeyboard *GetKeyboard() { return m_pKeyboard; }
    CMouse   *GetMouse()   { return m_pMouse; }

    bool Update();

    bool KeyDown(int key) { return (m_pKeyboard && m_pKeyboard->KeyDown(key)); }
    bool KeyUp(int key)  { return (m_pKeyboard && m_pKeyboard->KeyUp(key)); }

    bool ButtonDown(int button) { return (m_pMouse && m_pMouse->ButtonDown(button)); }
    bool ButtonUp(int button) { return (m_pMouse && m_pMouse->ButtonUp(button)); }
```

```

void GetMouseMovement(int &dx, int &dy) { if (m_pMouse) m_pMouse->GetMovement(dx, dy); }
int GetMouseWheelMovement() { return (m_pMouse) ? m_pMouse->GetWheelMovement() : 0; }

private:
    CKeyboard *m_pKeyboard;
    CMouse     *m_pMouse;

    LPDIRECTINPUT8 m_pDI;
};

// w pliku InputSystem.cpp
//****************************************************************************
CInputSystem::Initialize()

Inicjuje system wejścia. Parametr isExclusive powinien mieć wartość true
dla wyłącznego dostępu do myszy. Parametr flags jest kombinacją znaczników
IS_USEKEYBOARD i/lub IS_USEMOUSE.
*****
bool CInputSystem::Initialize(HWND hwnd, HINSTANCE appInstance, bool isExclusive,
DWORD flags)
{
    // tworzy obiekt DirectInput
    if (FAILED(DirectInput8Create(appInstance,
                                  DIRECTINPUT_VERSION,
                                  IID_IDirectInput8,
                                  (void **)&m_pDI,
                                  NULL)))
        return false;

    if (flags & IS_USEKEYBOARD)
    {
        m_pKeyboard = new CKeyboard(m_pDI, hwnd);
        if (m_pKeyboard == NULL)
            return false;
    }
    if (flags & IS_USEMOUSE)
    {
        m_pMouse = new CMouse(m_pDI, hwnd, isExclusive);
        if (m_pMouse == NULL)
            return false;
    }

    return true;
} // CInputSystem::Initialize()

//****************************************************************************
CInputSystem::Shutdown()

Zwala obiekty i pamięć.
*****
bool CInputSystem::Shutdown()
{
    UnacquireAll();
    if (m_pKeyboard)
    {
        delete m_pKeyboard;
        m_pKeyboard = NULL;
    }
}

```

```
if (m_pKeyboard)
{
    delete m_pMouse;
    m_pMouse = NULL;
}

if (FAILED(m_pDI->Release()))
    return false;

return true;
} // CInputSystem::Shutdown()

/***** CInputSystem::Update() *****/
Odpytuje wszystkie urządzenia.
bool CInputSystem::Update()
{
    if (m_pKeyboard)
        m_pKeyboard->Update();
    if (m_pMouse)
        m_pMouse->Update();

    return true;
} // CInputSystem::Update()

/***** CInputSystem::AcquireAll() *****/
Zajmuje wszystkie urządzenia.
void CInputSystem::AcquireAll()
{
    if (m_pKeyboard)
        m_pKeyboard->Acquire();
    if (m_pMouse)
        m_pMouse->Acquire();
} // CInputSystem::AcquireAll()

/***** CInputSystem::UnacquireAll() *****/
Zwala wszystkie urządzenia.
void CInputSystem::UnacquireAll()
{
    if (m_pKeyboard)
        m_pKeyboard->Unacquire();
    if (m_pMouse)
        m_pMouse->Unacquire();
} // CInputSystem::UnacquireAll()
```

Klasy reprezentujące klawiaturę i mysz obudowują odpowiednie obiekty urządzeń DirectInput. Ichinicjacja odbywa się w konstruktorach i obejmuje tworzenie obiektów urządzeń, określenie ich formatu danych i poziomu współpracy oraz zajęcie urządzenia. Każda z klas przechowuje też ostatnio pobieraną informację o stanie urządzenia. Dane te aktualizowane są za pomocą metody Update(), którą wywołuje system wejścia. Metodę Update() można też wywoływać indywidualnie, jeśli korzysta się bezpośrednio z obiektów urządzeń. Obiekt klawiatury pozwala sprawdzać stan wszystkich klawiszy, a obiekt myszy dostarcza informacji o względnych zmianach jej położenia i położenia kółka myszy oraz o stanie jej przycisków (do czterech). Poniżej przedstawiona została implementacja klas klawiatury i myszy:

```
class CKeyboard
{
public:
    CKeyboard(LPDIRECTINPUT8 pDI, HWND hwnd);
    ~CKeyboard();

    bool KeyDown(int key) { return (_m_keys[key] & 0x80) ? true : false; }
    bool KeyUp(int key) { return (_m_keys[key] & 0x80) ? false : true; }

    bool Update();

    void Clear() { ZeroMemory(_m_keys, 256 * sizeof(char)); }

    bool Acquire();
    bool Unacquire();

private:
    LPDIRECTINPUTDEVICE8 _m_pDIDev;
    int _m_keys[256];
};

//*********************************************************************
CKeyboard::Constructor

Inicjuje urządzenie DirectX
*****
CKeyboard::CKeyboard(LPDIRECTINPUT8 pDI, HWND hwnd)
{
    if (FAILED(pDI->CreateDevice(GUID_SysKeyboard, &_m_pDIDev, NULL)))
    {
        // obsługa błędów
    }

    if (FAILED(_m_pDIDev->SetDataFormat(&c_dfDIKeyboard)))
    {
        // obsługa błędów
    }

    if (FAILED(_m_pDIDev->SetCooperativeLevel(hwnd, DISCL_FOREGROUND | DISCL_NONEXCLUSIVE)))
    {
        // obsługa błędów
    }

    if (FAILED(_m_pDIDev->Acquire()))
    {
        // obsługa błędów
    }
}
```

```
    Clear();
} // CKeyboard::Constructor

/***** CKeyboard::Destructor *****/
Zwalnia urządzenie DirectX
***** CKeyboard::~CKeyboard()
{
    if (m_pDIDev)
    {
        m_pDIDev->Unacquire();
        m_pDIDev->Release();
    }
} // CKeyboard::Destructor

/***** CKeyboard::Update() *****/
Odpytuje klawiaturę i przechowuje informację o jej stanie.
***** CKeyboard::Update()
bool CKeyboard::Update()
{
    if (FAILED(m_pDIDev->GetDeviceState(sizeof(m_keys), (LPVOID)m_keys)))
    {
        if (FAILED(m_pDIDev->Acquire()))
        {
            return false;
        }
        if (FAILED(m_pDIDev->GetDeviceState(sizeof(m_keys), (LPVOID)m_keys)))
        {
            return false;
        }
    }
    return true;
} // CKeyboard::Update()

/***** CKeyboard::Acquire() *****/
Zajmuje klawiaturę.
***** CKeyboard::Acquire()
bool CKeyboard::Acquire()
{
    Clear();
    return (!FAILED(m_pDIDev->Acquire()));
} // CKeyboard::Acquire()

/***** CKeyboard::Unacquire() *****/
Zwalnia klawiaturę.
***** CKeyboard::Unacquire()
```

```
bool CKeyboard::Unacquire()
{
    Clear();
    return (!FAILED(m_pDIDev->Unacquire()));
} // CKeyboard::Unacquire()

class CMouse
{
public:
    CMouse(LPDIRECTINPUT8 pDI, HWND hwnd, bool isExclusive = true);
    ~CMouse();

    bool ButtonDown(int button) { return (m_state.rgbButtons[button] & 0x80) ? true : false; }
    bool ButtonUp(int button) { return (m_state.rgbButtons[button] & 0x80) ? false : true; }
    int GetWheelMovement() { return m_state.lZ; }
    void GetMovement(int &dx, int &dy) { dx = m_state.lX; dy = m_state.lY; }

    bool Update();

    bool Acquire();
    bool Unacquire();

private:
    LPDIRECTINPUTDEVICE8 m_pDIDev;
    DIMOUSESTATE         m_state;
};

//*********************************************************************
CMouse::Constructor

Inicjuje urządzenie DirectInput.
//*********************************************************************
CMouse::CMouse(LPDIRECTINPUT8 pDI, HWND hwnd, bool isExclusive)
{
    if (FAILED(pDI->CreateDevice(GUID_SysMouse, &m_pDIDev, NULL)))
    {
        // obsługa błędów
    }

    if (FAILED(m_pDIDev->SetDataFormat(&c_dfDIMouse)))
    {
        // obsługa błędów
    }

    DWORD flags;
    if (isExclusive)
        flags = DISCL_FOREGROUND | DISCL_EXCLUSIVE | DISCL_NOWINKEY;
    else
        flags = DISCL_FOREGROUND | DISCL_NONEXCLUSIVE;

    if (FAILED(m_pDIDev->SetCooperativeLevel(hwnd, flags)))
    {
        // obsługa błędów
    }
}
```

```
if (FAILED(m_pDIDev->Acquire()))
{
    // obsługa błędów
}

if (FAILED(m_pDIDev->GetDeviceState(sizeof(DIMOUSESTATE), &m_state)))
{
    // obsługa błędów
}
} // CMouse::Constructor

//*********************************************************************
CMouse::~CMouse()
{
    if (m_pDIDev)
    {
        m_pDIDev->Unacquire();
        m_pDIDev->Release();
    }
} // CMouse::Destructor

//*********************************************************************
CMouse::Update()
{
    Odpytuje mysz i przechowuje informacje o jej stanie.
    ****
    bool CMouse::Update()
    {
        if (FAILED(m_pDIDev->GetDeviceState(sizeof(DIMOUSESTATE), &m_state)))
        {
            if (FAILED(m_pDIDev->Acquire()))
            {
                return false;
            }
            if (FAILED(m_pDIDev->GetDeviceState(sizeof(DIMOUSESTATE), &m_state)))
            {
                return false;
            }
        }

        return true;
    } // CMouse::Update()

    ****
    CMouse::Acquire()
    Zajmuje mysz.
    ****
```

```

bool CMouse::Acquire()
{
    return (!FAILED(m_pIDev->Acquire()));
} // CMouse::Acquire

//*********************************************************************
CMouse::Unacquire()

Zwalnia mysz
//********************************************************************/
bool CMouse::Unacquire()
{
    return (!FAILED(m_pIDev->Unacquire()));
} // CMouse::Unacquire()

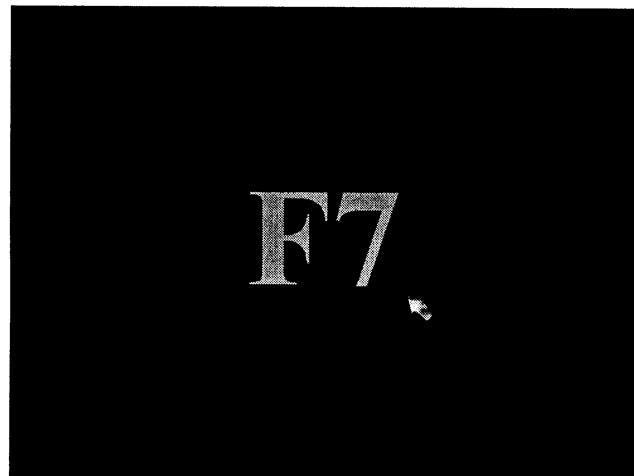
```

## Przykład zastosowania systemu wejścia

Zastosowanie podsystemu wejścia zilustrowane zostanie przykładem prostego programu, którego efekt działania przedstawia rysunek 16.1. Program śledzi ruch myszy i wyświetla jej wskaźnik na ekranie. Naciśnięcie przycisku myszy lub ruch jej kółka powodują zmianę kształtu kurSORA myszy. Program monitoruje także stan klawiszy funkcyjnych klawiatury i wyświetla nazwę ostatnio naciśniętego klawisz funkcyjnego na środku ekranu.

**Rysunek 16.1.**

*Przykład  
zastosowania  
systemu wejścia*



Pełen kod źródłowy przykładowego programu znajduje się na dysku CD. Tutaj zamieszczony został jedynie kod funkcji ProcessInput(). Należy zwrócić uwagę, że aby zmienić kształt kurSORA dokładnie raz przy każdym naciśnięciu przycisku myszy, nie wystarczy jedynie wywołanie metody ButtonDown(). Ponieważ w każdej sekundzie tworzy się grafikę co najmniej kilkanaście razy, a przycisk myszy jest naciśnięty nieco dłużej niż ułamek sekundy, to metoda ButtonDown() zwraca wartość true więcej niż jeden raz dla pojedynczego naciśnięcia przycisku. Należy więc obsłużyć moment naciśnięcia przycisku, a następnie nie wykonywać żadnych działań do momentu jego zwolnienia.

```
void ProcessInput()
{
    static bool leftButtonDown = false;
    static bool rightButtonDown = false;

    // zmienia kursor myszy na skutek naciśnięcia jej przycisku lub obrotu kółka
    if (g_input.ButtonDown(0))
        leftButtonDown = true;

    if (g_input.ButtonDown(1))
        rightButtonDown = true;

    if (g_input.GetMouseWheelMovement() < 0 || (leftButtonDown && g_input.ButtonUp(0)))
    {
        leftButtonDown = false;
        g_textureIndex--;
        if (g_textureIndex < 0)
            g_textureIndex = NUM_TEXTURES - 1;
    }

    if (g_input.GetMouseWheelMovement() > 0 || (rightButtonDown && g_input.ButtonUp(1)))
    {
        rightButtonDown = false;
        g_textureIndex++;
        if (g_textureIndex == NUM_TEXTURES)
            g_textureIndex = 0;
    }

    // aktualizuje położenie myszy
    int dx, dy;
    g_input.GetMouseMovement(dx, dy);

    // zapobiega opuszczeniu ekranu przez kursor
    g_mouseX += dx;
    if (g_mouseX >= gScreenWidth)
        g_mouseX = gScreenWidth - 1;
    if (g_mouseX < 0)
        g_mouseX = 0;
    g_mouseY -= dy;
    if (g_mouseY >= gScreenHeight)
        g_mouseY = gScreenHeight - 1;
    if (g_mouseY < 0)
        g_mouseY = 0;

    // sprawdza, czy naciśnięty został jeden z klawiszy funkcyjnych
    if (g_input.KeyDown(DIK_F1))
        strcpy(g_lastKey, "F1");
    if (g_input.KeyDown(DIK_F2))
        strcpy(g_lastKey, "F2");
    if (g_input.KeyDown(DIK_F3))
        strcpy(g_lastKey, "F3");
    if (g_input.KeyDown(DIK_F4))
        strcpy(g_lastKey, "F4");
    if (g_input.KeyDown(DIK_F5))
        strcpy(g_lastKey, "F5");
    if (g_input.KeyDown(DIK_F6))
        strcpy(g_lastKey, "F6");
    if (g_input.KeyDown(DIK_F7))
```

```
strcpy(g_lastKey, "F7");
if (g_input.KeyDown(DIK_F8))
    strcpy(g_lastKey, "F8");
if (g_input.KeyDown(DIK_F9))
    strcpy(g_lastKey, "F9");
if (g_input.KeyDown(DIK_F10))
    strcpy(g_lastKey, "F10");
if (g_input.KeyDown(DIK_F11))
    strcpy(g_lastKey, "F11");
if (g_input.KeyDown(DIK_F12))
    strcpy(g_lastKey, "F12");

// sprawdza, czy naciśnięty klawisz zakończenia ekranu
if (g_input.KeyDown(DIK_ESCAPE))
    PostQuitMessage(0);

} // ProcessInput ()
```

## Podsumowanie

W systemie Windows dostępnych jest kilka sposobów pobierania informacji wejściowej wprowadzanej przez użytkownika. Komunikaty systemu Windows i interfejs programowy Win32 stanowią najprostsze rozwiązanie, którego efektywność nie jest jednak wystarczająca w przypadku gier. DirectInput stanowi alternatywne rozwiązanie, którego zaletą jest duża szybkość. W rozdziale przedstawiono jedynie część możliwości DirectInput, jednak w stopniu wystarczającym do zastosowania w grach.

## Rozdział 17.

# Zastosowania DirectX Audio

W czasach systemu operacyjnego DOS tworzenie podsystemu dźwiękowego gry uważane było za szczególnie trudne zadanie ze względu na ogólną liczbę różnych kart dźwiękowych dostępnych na rynku. Sytuacji tej nie poprawiło wprowadzenie systemu Windows, który choć przesłaniał te różnice sprzętowe, to jednak nie nadawał się na docelową platformę dla tworzonych gier ze względu na niską efektywność. Obecnie DirectX Audio oferuje programistom nie tylko doskonałą efektywność i obsługę praktycznie każdej karty dźwiękowej, ale przede wszystkim bogatą funkcjonalność pozwalającą zrealizować najbardziej zaawansowane zadania. W rozdziale tym pokazane zostanie to, w jaki sposób wykorzystać DirectX Audio do stworzenia podsystemu dźwiękowego gry.

W rozdziale tym przedstawione zostaną:

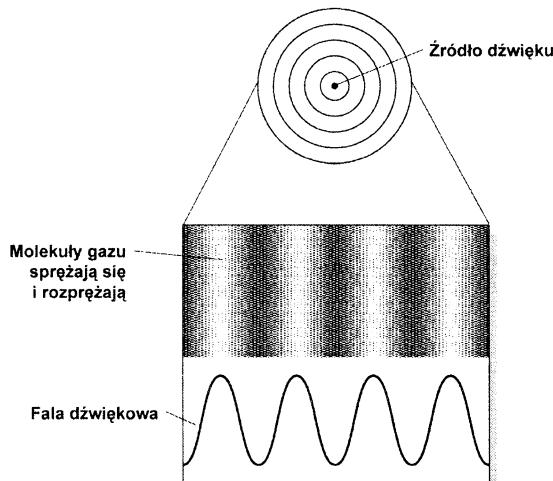
- ◆ podstawowe informacje na temat dźwięku;
- ◆ podstawy DirectX Audio;
- ◆ ładowanie i odtwarzanie dźwięku za pomocą DirectMusic;
- ◆ ścieżki dźwiękowe DirectX Audio;
- ◆ przestrzenne efekty dźwiękowe DirectX Audio.

## Dźwięk

Fizyka mówi, że dźwięk jest falą mechaniczną emitowaną przez źródło i rozchodząjącą się w pewnym kierunku, co ilustruje rysunek 17.1. W przypadku atmosfery ziemskiej zasady te określają molekuły gazów. Natomiast obserwując wybuch w przestrzeni kosmicznej nie można by usłyszeć żadnych efektów dźwiękowych, ponieważ dźwięk nie może rozchodzić się w próżni. Fala dźwiękowa może natomiast przemieszczać się w innych ośrodkach, na przykład wodzie i to z prędkością większą niż w powietrzu.

**Rysunek 17.1.**

Dźwięk jest falą mechaniczną emitowaną przez źródło



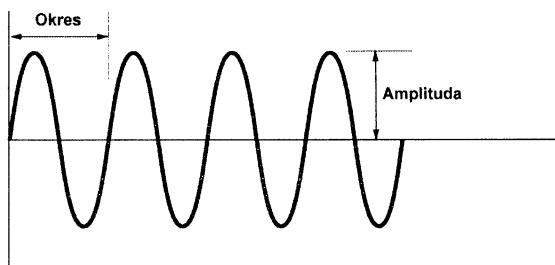
Mówiąc, że dźwięk jest falą mechaniczną opisuje się w rzeczywistości ruch cząstek ośrodka, w którym rozchodzi się dźwięk. Aby się o tym przekonać, można na przykład zbliżyć dłoń do głośnika basów zestawu muzycznego odtwarzającego głośno muzykę. Poczuje się falę powietrza tworzoną przez drgającą membranę głośnika. Energia membrany przekazywana jest otaczającym ją cząstkom powietrza, które następnie przekazują ją kolejnym cząstkom i tak dalej aż do momentu, gdy drgania te dotrą do ucha i mózg przetworzy je na dźwięk.

Dźwięk rozprzestrzenia się więc w powietrzu dzięki zderzeniom molekuł, które w ten sposób przekazują sobie energię mechaniczną. Taki sposób rozchodzenia się fali dźwiękowej sprawia, że prędkość dźwięku jest stosunkowo niewielka (zwłaszcza, jeśli porównać ją z prędkością światła). Na przykład obserwując wystrzał działa z odległości 1000 metrów ujrzy się natychmiast towarzyszący mu błysk i dym, ale huk wystrzału usłyszy się ponad dwie sekundy później. Dźwięk rozchodzi się bowiem w powietrzu z prędkością około 344 metrów na sekundę.

Fał dźwiękową charakteryzuje:

- ◆ **amplituda** — amplitudę można przedstawić jako objętość ośrodka przemieszczanego przez falę dźwiękową; dysponując potężnymi głośnikami zyskuje się możliwość przemieszczania znacznych mas powietrza i uzyskania dużej amplitudy fali (na wykresie fali pokazanym na rysunku 17.2 amplituda jest reprezentowana przez wysokość grzbietu fali);
- ◆ **częstotliwość** — częstotliwość jest liczbą pełnych okresów drgań wykonywanych przez źródło fali w jednostce czasu (*okres* fali reprezentowany jest na jej wykresie przez odległość pomiędzy dwoma grzbietami fali, co pokazuje rysunek 17.2); jednostką częstotliwości jest herc (Hz); częstotliwość dźwięku odbiera się jako jego wysokość (na przykład głośnik basu emiteme dźwięki o niższej częstotliwości niż głośnik wysokotonowy).

**Rysunek 17.2.**  
*Wykres fali*



## Dźwięk i komputery

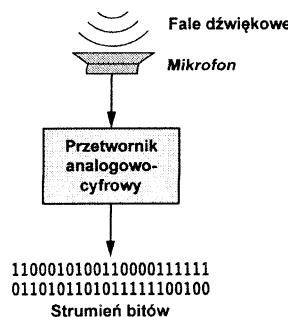
Oczywiście komputery nie przechowują ani nie tworzą dźwięku jako mechanicznych fal rozchodzących się w ośrodku złożonym z molekuł. Nie jest to po prostu technologicznie możliwe.

Komputery mogą natomiast przechowywać i odtwarzać dźwięk zapisany cyfrowo, a także uzyskiwany na drodze syntezy. Źródłem dźwięku zapisanego w postaci cyfrowej jest zwykły mikrofon. Dźwięk zapisany cyfrowo używany jest w grach przede wszystkim do uzyskania efektów dźwiękowych, takich jak eksplozje, odgłosy ruchu czy mowa. Natomiast dźwięk tworzony przez syntezator stanowi najczęściej podkład muzyczny gry.

## Cyfrowy zapis dźwięku

Aby zapisać dźwięk w postaci cyfrowej, trzeba skorzystać z przetwornika analogowo-cyfrowego, który zamienia analogowy sygnał mikrofonu na cyfrowe wartości odpowiadające zmianom amplitudy dźwięku w regularnych odstępach czasu. Rysunek 17.3 prezentuje ten proces.

**Rysunek 17.3.**  
*Cyfrowy zapis dźwięku*



Dysponując cyfrowym zapisem dźwięku można go odtworzyć za pomocą przetwornika cyfrowo-analogowego, który wytwarza analogowy sygnał wysyłany do głośników.

Cyfrowy zapis dźwięku charakteryzuje następujące parametry:

- ♦ **częstotliwość próbkowania** — określa ona liczbę próbek dźwięku zapisywanych w jednostce czasu i musi być co najmniej dwa razy większa od częstotliwości dźwięku (na przykład zapisując dźwięk o częstotliwości 10 000 Hz trzeba próbować go o częstotliwością co najmniej 20 000 Hz);

- ◆ **rozdzielcość próbkowania amplitudy** — rozdzielcość ta określa liczbę rozróżnianych wartości amplitudy (jeśli dysponuje się 8-bitowym przetwornikiem analogowo-cyfrowym, to można zapisać jedynie 256 różnych wartości amplitudy; dobrą jakość dźwięku zapewniają przetworniki 16-bitowe umożliwiające rozróżnienie 65 536 wartości amplitudy).

Dźwięk zapisany w postaci cyfrowej używany jest w grach głównie w celu uzyskania krótkich efektów dźwiękowych. Możliwe jest również odtwarzanie ścieżek muzycznych zapisanych cyfrowo, ale w tym przypadku zasadniczy problem stanowią duże rozmiary plików dźwiękowych. Dopiero wprowadzenie technologii MP3 sprawiło, że zastosowanie muzyki zapisanej cyfrowo w grach stało się o wiele bardziej realne. Jest to o tyle istotne, że muzyka zapisana cyfrowo posiada zdecydowanie lepszą jakość od efektów muzycznych tworzonych na drodze syntezy.

## Synteza dźwięku

Ponieważ falę można opisać za pomocą równania matematycznego, to cyfrowy dźwięk można tworzyć także za pomocą odpowiednich algorytmów, a nie tylko odtwarzać gotowy, wcześniej zapisany w postaci cyfrowej. Na przykład wiedząc, że nucie A odpowiada dźwiękowi o częstotliwości 440 Hz wystarczy sprawić, aby syntezator dźwięku wygenerował falę o takiej częstotliwości i w rezultacie otrzyma się ton odpowiadający nucie A.

Kiedy komputery zaczęły tworzyć dźwięki zachwycały się melodiami tworzonymi za pomocą generowanych po kolej dźwięków o różnej częstotliwości. Takie możliwości jednak szybko okazały się niewystarczające i w rezultacie powstały syntezatory wielokanałowe, które mogły generować jednocześnie wiele różnych tonów. Uzyskany efekt był już dużo lepszy, ale ciągle charakteryzował się zbyt syntetycznym, nienaturalnym brzmieniem. Przyczyną tego problemu jest to, że tony emitowane przez tradycyjne instrumenty składają się z wielu tak zwanych częstotliwości harmonicznych, a nie pojedynczej częstotliwości emitowanej przez syntezator.

W odpowiedzi skonstruowano więc syntezatory z modulacją fazy (FM), które mają możliwość zmiany amplitudy i częstotliwości dźwięku w danym kanale i tym samym możliwość tworzenia naturalniej brzmiących dźwięków przez przesunięcie ich fazy i wprowadzenie wielu częstotliwości harmonicznych.

Obecnie stosowane są jeszcze dwie inne metody syntezy dźwięku wykorzystujące:

- ◆ **tablice wzorców dźwięku** — metoda ta stanowi połączenie syntezy dźwięku i jego cyfrowego zapisu; tablica wzorców dźwięku zawiera próbki naturalnych dźwięków przetworzone przez procesor sygnałowy (DSP), na ich podstawie procesor DSP potrafi wygenerować dźwięk o wybranej amplitudzie i częstotliwości, a uzyskany w ten sposób efekt jest zbliżony jakością do zapisanego cyfrowo dźwięku i wymaga jedynie przechowywania próbek dźwięku;
- ◆ **obwiednię dźwięku** — synteza dźwięku wykorzystująca informacje o jego obwiedni umożliwia stworzenie przez procesor sygnałowy modelu dźwięku wybranego instrumentu; model ten jest na tyle wierny, że wygenerowane na jego podstawie dźwięki są praktycznie nie do odróżnienia od dźwięków prawdziwego instrumentu.

- ◆ Chociaż każda z tych metod umożliwia uzyskanie niezwykłych efektów, to jednak tworząc z ich pomocą podkład muzyczny trzeba przyjąć dość duże ograniczenie związane z formatem plików *MIDI* (*Musical Instrument Digital Interface*). Pliki *MIDI* umożliwiają zapis kompozycji muzycznych za pomocą instrumentów i dźwięków w osobnych kanałach. Dysponując na przykład czterema kanałami można za pomocą pliku *MIDI* stworzyć kwartet (na przykład fortepian, flet, gitara i saksofon), w którym każdy instrument będzie grał własną sekwencję nut w osobnym kanale. Problem z plikami *MIDI* polega na tym, że odtwarzając je na różnych komputerach można uzyskać różne efekty. Przyczyną jest to, że pliki *MIDI* zawierają jedynie zapis nutowy pozostawiając kwestie tworzenia dźwięku konkretnemu sprzętowi. Zaletą takiego rozwiązania jest natomiast niewielki rozmiar plików *MIDI*. Zapis kilkunastu minut muzyki w pliku *MIDI* zajmuje jedynie kilkaset kilobajtów, podczas gdy zapis cyfrowy tego samego fragmentu liczony jest w dziesiątkach megabajtów.

## Czym jest DirectX Audio?

DirectX Audio jest komponentem DirectX odpowiedzialnym za tworzenie ścieżki dźwiękowej z wykorzystaniem istniejących możliwości sprzętowych. W poprzednich wersjach DirectX odtwarzaniu dźwięków służyły komponenty DirectSound i DirectMusic. W wersji 8. DirectX połączono je w komponent DirectX Audio. DirectSound służy do cyfrowego zapisu dźwięku i jego odtwarzania, a DirectMusic stanowi zasadniczy mechanizm tworzenia dowolnych dźwięków.

W wersji 8. DirectX komponenty DirectSound i DirectMusic stanowią interfejsy DirectX Audio. Z ich pomocą można:

- ◆ ładować i odtwarzać dźwięki z plików w formatach *MIDI*, *WAV* lub w formacie DirectMusic Producer;
- ◆ odtwarzać jednocześnie dźwięk z wielu źródeł;
- ◆ precyzyjnie planować w czasie zdarzenia związane z tworzeniem dźwięków;
- ◆ używać dźwięków ładowalnych DLS (*downloadable sounds*), dzięki którym efekty dźwiękowe będą brzmieć tak samo na różnych komputerach;
- ◆ stosować przestrzenne efekty dźwiękowe;
- ◆ stosować zmiany rozdzielczości, pogłos, zniekształcenia i inne efekty;
- ◆ zapisywać dane *MIDI* lub przesyłać je pomiędzy portami;
- ◆ zapisywać pliki *WAV* na podstawie dźwięku uzyskanego za pośrednictwem mikrofonu lub innych źródeł;
- ◆ odtwarzać muzykę, której brzmienie ulega z każdym powtórzeniem nieznacznym zmianom;
- ◆ tworzyć dynamicznie połączenia istniejących fragmentów muzyki.

Powyższa lista prezentuje jedynie ułamek bogatych możliwości DirectX Audio. Zanim z nich zacznie się korzystać, trzeba najpierw zapoznać się z podstawowymi informacjami dotyczącymi DirectX Audio.

## Charakterystyka DirectX Audio

Zanim zacznie się korzystać z możliwości DirectX Audio, trzeba poznać najpierw podstawowe funkcje:

- ◆ obiekty ładujące;
- ◆ segmenty i stany segmentów;
- ◆ wykonanie;
- ◆ komunikaty;
- ◆ kanały wykonania;
- ◆ syntezator DLS;
- ◆ instrumenty i ładowanie ich dźwięków;
- ◆ ścieżki dźwiękowe i bufory.

Należy przyjrzeć się bliżej każdemu z nich.

### Obiekty ładujące

Obiekty te służą do ładowania innych obiektów (w tym także plików segmentowych DirectMusic, kolekcji DLS, plików *MIDI*, plików *WAV* mono- i stereofonicznych). Dlatego też *obiekty ładujące* tworzy się zwykle jako pierwsze spośród obiektów DirectX Audio.

### Segmenty i stany segmentów

Przez *segment* rozumieć należy obiekt obudowujący dane reprezentujące pewien dźwięk lub fragment muzyki odtwarzany jako całość. Dane te mogą być zapisane w formacie *MIDI*, *WAV* lub reprezentować fragment muzyki tworzony podczas wykonania programu lub kolekcję informacji pliku segmentowego DirectMusic Producer.

Segment może być odtwarzany jako segment zasadniczy lub segment wtórny. W danym momencie może być odtwarzany tylko jeden segment zasadniczy, ale w tym samym czasie może być odtwarzanych wiele segmentów wtórnego, które zwykle reprezentują krótkie motywy muzyczne lub efekty dźwiękowe.

Segmenty mogą zawierać różne rodzaje danych, takie jak dźwięk zapisany cyfrowo, informacje o zmianie akordów i instrumentów oraz o zmianie tempa. Mogą także przechowywać informacje o ścieżkach dźwięku (które omówione zostaną wkrótce), za pomocą których są odtwarzane (w tym także przy użyciu efektów specjalnych).

## Wykonanie

*Obiekt wykonania* obsługuje przepływ danych ze źródła do syntezytora. Określa on chro- nometraż wykonania, przyporządkowanie kanałów do ścieżek dźwiękowych, trasowanie komunikatów, sposób zarządzania narzędziami, powiadomienia i wiele innych. Zwykle wystarcza pojedynczy obiekt wykonania.

## Komunikaty

Dane, których przepływ obsługuje obiekt wykonania, mają postać komunikatów. *Ko- munikat* może zawierać informację o pojedynczym tonie, dźwięku, zmianie kontrolera lub nawet tekst piosenki wyświetlany podczas jej odtwarzania. Zwykle nie trzeba bezpośrednio obsługiwać komunikatów w programie, ponieważ są one generowane podczas odtwarzania segmentu. Jeśli zajdzie taka potrzeba, to można wstawić komunikat do wykonania bądź przejmować komunikaty generowane automatycznie. Komunikaty można także stosować do powiadamiania (na przykład o tym, że osiągnięty został pewien punkt wykonania).

## Kanały wykonania

*Kanał wykonania* łączy partię ze ścieżką dźwięku. *Partia* może reprezentować kanał MIDI, partię w segmencie DirectMusic Producer lub dźwięk. Każdy odtwarzany dźwięk składa się z jednej lub więcej partii, które zwykle reprezentują instrumenty muzyczne. Kanały wykonania przypominają kanały MIDI, jednak ich liczba jest praktycznie nieograniczona (zwykle może istnieć do 16 kanałów MIDI).

## Syntezator DSL

Jeśli dane opisujące dźwięk nie określają jeszcze bezpośrednio jego częstotliwości, jak jest to na przykład w przypadku zapisu nuty w pliku *MIDI*, to zanim zostaną przekazane do karty dźwiękowej muszą zostać przetworzone przez syntezytator. Większość syntezytatorów wyposażona jest w implementację ładowania dźwięków DLS Level 2. W przypadku braku sprzętowego syntezytatora dźwięk syntezytowany jest wyłącznie na drodze programowej. Syntezytator DSL generuje dźwięk na podstawie wzorca, co umożliwia osiągnięcie nawet bardzo złożonych brzmień dowolnych dźwięków.

## Instrumenty i ładowanie dźwięków

Aby naśladować prawdziwy instrument, syntezytator musi posiadać opis jego brzmienia. Opis ten ma postać próbki dźwięku i danych dotyczących ich artykulacji. Tworzą one kolekcje dźwięków ładowalnych DLS, które należy załadować do syntezytatora.

Zwykle brzmienia poszczególnych nut dla danego instrumentu tworzone są na podstawie jednej próbki, dla której zmieniana jest odpowiednio częstotliwość. DLS Level 2 umożliwia już reprezentowanie każdej nuty za pomocą osobnej próbki lub kombinacji próbek. Nawet prędkość odtwarzania nut może powodować wybór różnych próbek.

## Ścieżki dźwięku i bufory

**Ścieżka dźwięku** wyznacza drogę jaką przebywa segment DirectMusic od wykonania do syntezatora, następnie do buforów DirectSound, gdzie tworzone są efekty dźwiękowe i do zasadniczego bufora, gdzie następuje miksuwanie ostatecznego dźwięku.

Można utworzyć wiele ścieżek dźwięku i odtwarzać z nich pomocą segmentów. Na przykład dla odtwarzania plików *MIDI* z efektem pogłosu stworzy się jedną ścieżką, a inną ścieżkę dla odtwarzania plików *WAV* z przestrzennym efektem dźwięku. Ścieżkę dźwięku można sobie wyobrażać jako łańcuch obiektów, które przetwarzają dane dźwięku.

## Przepływ danych dźwięku

DirectX Audio używa się najczęściej w celu załadowania muzyki lub efektów dźwiękowych z plików *MIDI*, *WAV* lub plików segmentów DirectMusic Producer. Po załadowaniu dane te zostają obudowane w obiekty segmentów, z których każdy reprezentuje dane z pojedynczego źródła. Obiekt wykonania może następnie odtwarzać jeden z tych segmentów jako segment zasadniczy oraz dowolną liczbę segmentów wtórych.

Segment zawiera jeden lub więcej śladów. Każdy ślad zawiera synchronizowane dane opisujące nuty i zmiany tempa. Podczas odtwarzania śladu przez obiekt wykonania powstają komunikaty oznaczone etykietą czasu.

Obiekt wykonania rozsyła komunikaty do narzędzi pogrupowanych w następujące grafy:

- ◆ **graf narzędzi segmentów** — akceptuje komunikaty pochodzące z wybranych segmentów;
- ◆ **graf narzędzi ścieżki dźwięku** — akceptuje komunikaty pochodzące ze wszystkich segmentów ścieżki dźwięku;
- ◆ **graf narzędzi wykonania** — akceptuje komunikaty pochodzące ze wszystkich segmentów. Narzędzia mogą modyfikować, przekazywać, usuwać bądź wysyłać nowe komunikaty.

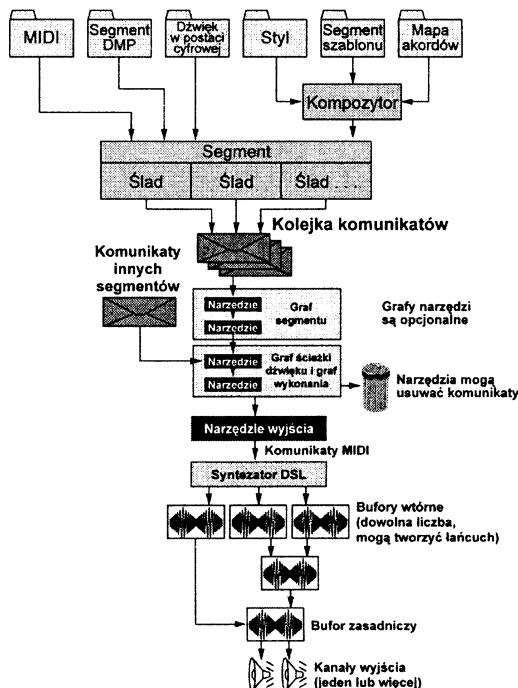
Na końcu komunikaty trafiają do narzędzia wyjścia, które przetwarzają dane na format *MIDI* i przekazują je do syntezatora. Syntezator tworzy dźwięki i przekazuje je do ujścia, które zarządza ich dystrybucją poprzez magistrale do jednego z trzech typów buforów DirectSound:

- ◆ **buforów ujścia** — są to bufore wtórne, które przekształcają dane na format bufora zasadniczego i umożliwiają zmianę natężenia dźwięku, jego położenia w przestrzeni i innych właściwości; bufore te mogą przekazywać swoje dane do modułów efektów, takich jak na przykład pogłos, zniekształcenie czy echo; uzyskany rezultat przekazywany jest albo wprost do bufora zasadniczego albo do jednego lub więcej buforów miksuowania;
- ◆ **bbuforów miksuowania** — bufore te otrzymują dane z innych buforów, stosując odpowiednie efekty i miksuje uzyskany rezultat; globalne efekty wykonywane są najczęściej właśnie za pomocą buforów miksuowania;
- ◆ **bbufora zasadniczego** — wykonuje on ostateczne miksuwanie wszystkich danych i wysyła rezultat do kanałów wyjścia.

Rysunek 17.4 ilustruje przepływ danych od plików do głośników.

**Rysunek 17.4.**

Przepływ danych dźwięku w DirectX Audio



## Ładowanie i odtwarzanie dźwięku w DirectMusic

DirectMusic jest komponentem DirectX bazującym na modelu COM. W praktyce oznacza to, że DirectMusic nie posiada funkcji pomocniczych tworzących obiekty COM i wobec tego trzeba tworzyć je indywidualnie korzystając z wywołań funkcji biblioteki COM. Korzystanie z DirectMusic nie wymaga dołączenia do programu wykonywalnego dodatkowej biblioteki, a jedynie włączenia plików nagłówkowych *dmusiccc.h* i *dmusici.h* do programu źródłowego.

Proces odtworzenia dźwięku za pomocą DirectMusic składa się z sześciu podstawowych etapów.

1. **Inicjowanie COM.** Ponieważ nie istnieją funkcje pomocnicze tworzące obiekty DirectMusic, trzeba samodzielnie zainicjować COM za pomocą funkcji `CoInitialize()`.
2. **Tworzenie i inicjowanie obiektu wykonania.** Za pomocą funkcji `CoCreateInstance()` tworzy się pojedynczy obiekt wykonania i uzyskuje interfejs klasy `IDirectMusicPerformance8`. Następnie wywołuje się metodę `IDirectMusicPerformance8::InitAudio()` w celu określenia domyślnej ścieżki dźwięku.

- 3. Tworzenie obiektu ładującego.** Ponownie trzeba skorzystać z funkcji CoCreateInstance(), tym razem w celu uzyskania interfejsu IDirectMusicLoader8.
- 4. Ładowanie segmentu.** Wywołuje się metodę IDirectMusicLoader8::SetSearchDirectory() w celu określenia położenia plików danych. Za pomocą metody IDirectMusicLoader8::GetObject() ładuje się segment z pliku i uzyskuje jego interfejs IDirectMusicSegment8.
- 5. Ładuje się brzmienia instrumentów.** Za pomocą metody IDirectMusicSegment8::Download() ładuje się dane DLS.
- 6. Odtwarzanie segmentu.** Wskaźnik segmentu przekazuje się metodzie IDirectMusicPerformance8::PlaySegmentEx().

Teraz można już przejść do szczegółowego omówienia każdego z tych etapów.

## Iinicjacja COM

Jest to z pewnością najprostszy etap w procesie odtwarzania dźwięków. Funkcję CoInitialize() należy wywołać na początku programu (przed wywołaniem innych funkcji COM):

```
// inicjacja COM
CoInitialize(NULL);
```

## Tworzenie i inicjacja obiektu wykonania

Interfejs obiektu jest zasadniczym interfejsem komponentu DirectX Audio. Aby go utworzyć, należy wywołać funkcję CoCreateInstance() przekazując jej identyfikator interfejsu wykonania, identyfikator klasy obiektu oraz zmienną, za pomocą której zostanie zwrócony wskaźnik interfejsu. Najpierw trzeba jednak zdefiniować zmienną typu IDirectMusicPerformance8:

```
IDirectMusicPerformance8* dmusicPerformance = NULL; // obiekt wykonania
```

Następnie tworzy się obiekt wykonania:

```
// tworzymy wykonanie
CoCreateInstance(CLSID_IDirectMusicPerformance, NULL, CLSCTX_INPROC,
    IID_IDirectMusicPerformance8, (void**)&dmusicPerformance);
```

Teraz można wywoływać już metody interfejsu wykonania. Jako pierwszą należy wywołać metodę IDirectMusicPerformance8::InitAudio(), która inicjuje wykonanie oraz umożliwia (opcjonalnie) określenie domyślnej ścieżki dźwięku. Metoda ta zdefiniowana jest następująco:

```
HRESULT InitAudio(
    IDirectMusic** ppDirectMusic, // wskaźnik interfejsu obiektu DirectMusic
    IDirectSound** ppDirectSound, // wskaźnik interfejsu obiektu DirectSound
    HWND hWnd, // uchwyt okna
    DWORD dwDefaultPathType, // typ domyślnej ścieżki dźwięku
    DWORD dwPChannelCount, // liczba kanałów wykonania dla ścieżki dźwięku
    DWORD dwFlags, // znaczniki właściwości syntezatora
    DMUS_AUDIOPARAMS pParams); // parametry syntezatora
```

A oto przykład wywołania metody InitAudio():

```
dmusicPerformance->InitAudio(
    NULL,                                // interfejs IDirectMusic (zbędny)
    NULL,                                // interfejs IDirectSound (zbędny)
    NULL,                                // uchwyt okna
    DMUS_APATH_SHARED_STEREOPLUSREVERB, // domyślna ścieżka
    64,                                   // liczba kanałów wykonania
    DMUS_AUDIOF_ALL,                     // właściwość syntezatora
    NULL);                               // domyślne parametry syntezatora
```

Powyzsze wywołanie metody InitAudio() skonfiguruje interfejsy DirectMusic i DirectSound tak, by korzystały z 64 kanałów wykonania przydzielonych domyślnej ścieżce dźwięku oraz wszystkich możliwości syntezatora.

## Tworzenie obiektu ładującego

Interfejs IDirectMusicLoader8 można uzyskać za pomocą funkcji CoCreateInstance(), ale najpierw trzeba zdefiniować odpowiednią zmienną:

```
IDirectMusicLoader8* dmusicLoader = NULL; // obiekt ładujący

// tworzy obiekt ładujący
CoCreateInstance(CLSID_IDirectMusicPerformance, NULL, CLSCTX_INPROC,
IID_IDirectMusicLoader8, (void**)&dmusicLoader);
```

## Załadowanie segmentu

Zanim załaduję się segment, trzeba poinformować najpierw obiekt ładujący, gdzie może odnaleźć pliki zawierające dane dźwięku. Najlepiej zdefiniować w tym celu domyślny katalog takich plików, choć istnieje także możliwość określania pełnej ścieżki dostępu za każdym razem, gdy ładowany będzie plik. Domyślny katalog plików dźwiękowych konfiguruje się za pomocą metody IDirectMusicLoader8::SetSearchDirectory() zdefiniowanej w następujący sposób:

```
HRESULT SetSearchDirectory(
    REFGUID rguidClass, // referencja identyfikatora klasy obiektu,
                        // którego dotyczy to wywołanie
    WCHAR* pwszPath,   // ścieżka dostępu do katalogu
    BOOL fClear);     // jeśli wartość TRUE, to usuwa wszelkie informacje
                      // o obiektach przed ustaleniem ścieżki dostępu
```

A oto przykład użycia tej metody:

```
WCHAR searchPath[MAX_PATH]; // ścieżka dostępu do plików dźwiękowych

// konwersja CHAR na WCHAR
MultiByteToWideChar(CP_ACP, 0, "c:\\\\music", -1, searchPath, MAX_PATH);

// konfiguruje ścieżkę dostępu
dmusicLoader->SetSearchDirectory(GUID_DirectMusicAllTypes, searchPath, FALSE);
```

Teraz można już załadować segment z pliku korzystając z jednej z metod: IDirectMusicLoader8::LoadObjectFromFile() lub IDirectMusicLoader8::GetObject(). Zwykle stosuje się metodę LoadObjectFromFile(), która zdefiniowana jest jak poniżej:

```
HRESULT LoadObjectFromFile(
    REFGUID rguidClassID, // unikalny identyfikator klasy obiektu
    REFIID iidInterfaceID, // unikalny identyfikator interfejsu
    WCHAR *pwzFilePath, // nazwa pliku zawierającego obiekt
    void **ppObject); // wskaźnik wymaganego interfejsu obiektu
```

Przy korzystaniu z tej metody należy przekazać jej nazwę pliku zawierającego ładowany segment oraz obiekt segmentu. Obiekt segmentu definiuje się w poniższy sposób:

```
IDirectMusicSegment8* dmusicSegment = NULL; // obiekt segmentu
```

A oto fragment kodu ładujący segment z pliku:

```
// nazwa pliku zawierającego segment
WCHAR filename[MAX_PATH] = L"testsound.wav";

// ładuje segment z pliku
if (FAILED(dmusicLoader->LoadObjectFromFile(CLSID_DirectMusicSegment,
                                                IID_IDirectMusicSegment8,
                                                filename,
                                                (void**)&dmusicSegment)))
{
    MessageBox(hwnd, "Nie znaleziono pliku audio!", "Błąd!", MB_OK);
    return 0;
}
```

## Ładowanie instrumentów

Zanim przystąpi się do odtworzenia załadowanego segmentu, trzeba najpierw załadować jeszcze dane odpowiednich instrumentów do syntezatora. Wywołuje się w tym celu metodę IDirectMusicSegment8::Download() o poniższym prototypie:

```
HRESULT Download(
    IUnknown *pAudioPath // wskaźnik interfejsu IUnknown obiektu wykonania
); // lub ścieżki dźwięku, do której trafią ładowane dane
```

Aby załadować segment do wykonania, należy posłużyć się poniższym wywołaniem metody Download():

```
dmusicSegment->Download(dmusicPerformance); // ładuje segment do wykonania
```

## Odtwarzanie segmentu

Zawartość pliku dźwiękowego odtwarza się poprzez przekazanie segmentu metodzie IDirectMusicPerformance8::PlaySegmentEx() o następującym prototypie:

```
HRESULT PlaySegmentEx(
    IUnknown* pSource, // wskaźnik interfejsu IUnknown odtwarzanego obiektu
    WCHAR pwzSegmentName, // zawsze wartość NULL (nie używany w DirectX 8)
    IUnknown* pTransition, // szablonu segmentu używany podczas przejścia do segmentu
    DWORD dwFlags, // znaczniki określające zachowanie metody
```

```

    int64 i64StartTime, // czas rozpoczęcia odtwarzania
    IDirectMusicSegmentState** ppSegmentState, // stan segmentu
    IUnknown* pFrom, // obiekt, którego odtwarzanie należy zatrzymać, zanim
    IUnknown* pAudioPath // ścieżka dźwięku używana do odtwarzania (domyślnie NULL)
);

```

Metoda ta pozwala określić precyzyjnie sposób odtwarzania segmentu. Aby odtworzyć go natychmiast korzystając z domyślnej ścieżki dźwięku, wystarczy nadać wszystkim parametrom metody, oprócz pierwszego, wartość NULL lub 0:

```

dmusicPerformance->PlaySegmentEx(
    dmSeg, // odtwarzany segment
    NULL, // parametr nie jest wykorzystywany
    NULL, // przejście pomiędzy segmentami
    0, // znaczniki
    0, // czas rozpoczęcia (wartość 0 oznacza natychmiast)
    NULL, // stan segmentu
    NULL, // zatrzymywany obiekt
    NULL); // domyślna ścieżka dźwięku

```

Po wywołaniu tej metody program będzie kontynuował swoje wykonanie, a załadowany plik dźwiękowy będzie odtwarzany aż do momentu jego zatrzymania.

## Zatrzymanie odtwarzania segmentu

Odtwarzanie segmentu można zatrzymać za pomocą metody `IDirectMusicPerformance8::Stop()` o następującym prototypie:

```

HRESULT Stop(
    IDirectMusicSegment* pSegment, // zatrzymywany segment
    IDirectMusicSegmentState* pSegmentState, // stan zatrzymywaneego segmentu
    MUSIC_TIME mtTime, // czas zatrzymania segmentu
    DWORD dwFlags // kiedy powinno nastąpić zatrzymanie segmentu
);

```

Z metody tej można skorzystać na przykład w taki sposób:

```

dmusicPerformance->Stop(
    dmusicSegment, // zatrzymywany segment (wartość NULL oznacza wszystkie segmenty)
    NULL, // stan zatrzymywaneego segmentu
    0, // czas zatrzymania segmentu (0 oznacza natychmiast)
    0); // kiedy zatrzymać segment (0 oznacza natychmiast)

```

Odtwarzanie segmentu można zatrzymać także za pomocą metody `IDirectMusicPerformance8::StopEx()` zdefiniowanej jak poniżej:

```

HRESULT StopEx(
    IUnknown *pObjectToStop, // interfejs zatrzymywanej obiektu
    int64 i64StopTime, // czas zatrzymania
    DWORD dwFlags // kiedy zatrzymać odtwarzanie
);

```

Metoda ta umożliwia zatrzymanie odtwarzania segmentu lub ścieżki dźwięku. Będzie ona używana w następujący sposób:

```
dmusicPerformance->StopEx(
    dmusicSegment, // zatrzymywany interfejs
    0,             // czas zatrzymania
    0);            // kiedy zatrzymać odtwarzanie
```

## Kontrola odtwarzania segmentu

Za pomocą metody `IDirectMusicPerformance::IsPlaying()` można sprawdzić, czy odtwarzany jest segment bądź stan segmentu. Metoda ta zdefiniowana jest następująco:

```
HRESULT IsPlaying(
    IDirectMusicSegment* pSegment,
    IDirectMusicSegmentState* pSetState
);
```

Aby sprawdzić, czy dany segment jest odtwarzany, należy przekazać jego wskaźnik metodzie `IsPlaying()` jako wartość parametru `pSegment`, a parametrowi `pSetState` nadać wartość `NULL`:

```
if (dmusicPerformance->IsPlaying(dmusicSegment, NULL) == S_OK)
{
    // segment jest nadal odtwarzany
}
else
{
    // segment nie jest już odtwarzany
}
```

## Określanie liczby odtworzeń segmentu

Liczبę odtworzeń segmentu można określić za pomocą metody `IDirectMusicSegment8::SetRepeats()` zdefiniowanej następująco:

```
HRESULT SetRepeats(
    DWORD dwRepeats // liczba odtworzeń segmentu
);
```

Metoda określa liczbę odtworzeń segmentu. Domyslnie dotyczy to całego segmentu, ale za pomocą metody `IDirectMusicSegment8::SetLoopPoints()` można zdefiniować odtwarzany fragment segmentu. Jeśli parametr `dwRepeats` otrzyma wartość `DMUS_SEG_REPEAT_INFINITE`, to segment będzie odtwarzany w sposób ciągły, aż do momentu, gdy jawnie zatrzymane zostanie jego odtwarzanie. Jeśli natomiast parametr `dwRepeats` otrzyma wartość 0, to zostanie odtworzony tylko raz.

Metoda `IDirectMusicSegment8::SetLoopPoints()` pozwala określić początek i koniec odtwarzanego fragmentu segmentu i zdefiniowana jest w następujący sposób:

```
HRESULT SetLoopPoints(
    MUSIC_TIME mtStart, // początek odtwarzanego fragmentu
    MUSIC_TIME mtEnd   // koniec odtwarzanego fragmentu
);
```

Po jej wywołaniu segment zostanie odtworzony za pierwszym razem od początku aż do punktu `mtEnd`, a następne odtworzenia będą rozpoczynać się od punktu `mtStart` i kończyć

w punkcie *mtEnd*. Takich odtworzeń będzie tyle, ile zostało określone za pomocą metody `IDirectMusicSegment8::SetRepeats()`.

## **Kończenie odtwarzania dźwięku**

Gdy zakończone zostanie już odtwarzanie dźwięku i będzie można zakończyć wykonanie aplikacji, należy zwolnić obiekty COM. Oczywiście najpierw trzeba zakończyć odtwarzanie segmentów za pomocą metody `IDirectMusicPerformance8::Stop()`. Następnie można zakończyć pracę obiektu wykonania posługując się metodą `IDirectMusicPerformance8::CloseDown()` zdefiniowaną jak poniżej:

HRESUIT CloseDown();

Wywołuje się ją w następujący sposób:

```
dmusicPerformance->CloseDown();
```

Teraz można już zwolnić wszystkie wykorzystywane interfejsy, w tym interfejsy obiektu wykonania, obiektu ładującego i segmentu:

```
dmusicLoader->Release();  
dmusicPerformance->Release();  
dmusicSegment->Release();
```

Korzystanie z modelu COM kończy wywołanie funkcji CoUninitialize():

Collnitialize()

## **Prosty przykład**

Poniższy przykład ilustruje zastosowanie omówionych dotąd sposobów ładowania plików dźwiękowych i ich odtwarzania. Program ten ładuje plik *MIDI* i odtwarza go podczas rysowania prostej animacji za pomocą OpenGL. Zrozumienie jego działania nie powinno sprawić kłopotu — tekst źródłowy programu został opatrzony odpowiednimi komentarzami.

```
define WIN32_LEAN_AND_MEAN // "odchudza" aplikację Windows
#define INITGUID // używa identyfikatorów GUID w DirectMusic

// Pliki nagłówkowe
#include <windows.h> // standardowy plik nagłówkowy Windows
#include <dmusicc.h> // pliki nagłówkowe DirectMusic
#include <dmusici.h>
#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

// Zmienne globalne
HDC g_HDC; // globalny kontekst urządzenia
bool fullScreen = false; // true = tryb pełnoekranowy;
// false = tryb okienkowy
```

```
bool keyPressed[256];           // tablica przyciśnięcie klawiszy

float angle = 0.0f;
unsigned int listBase;          // początkowa lista wyświetlania
GLYPHMETRICSFLOAT gmf[256];    // dane o orientacji i położeniu znaków
                                // używane przez listy wyświetlania

////// Zmienne DirectMusic
IDirectMusicLoader8 *dmusicLoader = NULL;           // obiekt ładujący
IDirectMusicPerformance8 *dmusicPerformance = NULL; // obiekt wykonania
IDirectMusicSegment8 *dmusicSegment = NULL;         // segment

//*****************************************************************************
* Interfejsy DirectMusic
*****//*************************************************************************/\n\n

// InitDirectXAudio()
// opis: inicjuje komponent DirectX Audio
bool InitDirectXAudio(HWND hwnd)
{
    char pathStr[MAX_PATH];// ścieżka dostępu plików dźwiękowych
    WCHAR wcharStr[MAX_PATH];

    // tworzy obiekt ładujący
    if (FAILED(CoCreateInstance(CLSID_DirectMusicLoader, NULL, CLSCTX_INPROC,
                                IID_IDirectMusicLoader8, (void**)&dmusicLoader)))
    {
        MessageBox(hwnd, "Nie można utworzyć obiektu IDirectMusicLoader8!",
                   "Błąd!", MB_OK);
        return false;
    }

    // tworzy obiekt wykonania
    if (FAILED(CoCreateInstance(CLSID_DirectMusicPerformance, NULL, CLSCTX_INPROC,
                               IID_IDirectMusicPerformance8,
                               (void**)&dmusicPerformance)))
    {
        MessageBox(hwnd, "Nie można utworzyć obiektu IDirectMusicPerformance8!",
                   "Błąd!", MB_OK);
        return false;
    }

    // inicjuje obiekt wykonania za pomocą domyślnej ścieżki dźwięku
    dmusicPerformance->InitAudio(NULL, NULL, hwnd,
                                  DMUS_APATH_SHARED_STEREOPLUSREVERB, 64,
                                  DMUS_AUDIOIF_ALL, NULL);

    // pobiera bieżący katalog
    GetCurrentDirectory(MAX_PATH, pathStr);

    // zamienia jego nazwę na Unicode
    MultiByteToWideChar(CP_ACP, 0, pathStr, -1, wcharStr, MAX_PATH);

    // określa ścieżkę dostępu do plików dźwiękowych
    dmusicLoader->SetSearchDirectory(GUID_DirectMusicAllTypes, wcharStr, FALSE);

    return true;
}
```

```
// LoadSegment()
// opis: ładuje segment z pliku
bool LoadSegment(HWND hwnd, char *filename)
{
    WCHAR wcharStr[MAX_PATH];
    .
    // zamienia nazwę pliku na Unicode
    MultiByteToWideChar(CP_ACP, 0, filename, -1, wcharStr, MAX_PATH);

    // ładuje segment z pliku
    if (FAILED(dmusicLoader->LoadObjectFromFile(CLSID_DirectMusicSegment,
        IID_IDirectMusicSegment8,
        wcharStr,
        (void**)&dmusicSegment)))
    {
        MessageBox(hwnd, "Nie znaleziono pliku audio!",
            "Błąd!", MB_OK);

        return false;
    }

    // ładuje instrumenty do syntezytora
    dmusicSegment->Download(dmusicPerformance);

    return true;
}

// PlaySegment()
// opis: rozpoczyna odtwarzanie segmentu
void PlaySegment(IDirectMusicPerformance8* dmPerf, IDirectMusicSegment8* dmSeg)
{
    // odtwarza segment od następnego taktu
    dmPerf->PlaySegmentEx(dmSeg, NULL, NULL, 0, 0, NULL, NULL);
}

// StopSegment()
// opis: zatrzymuje odtwarzanie segmentu
void StopSegment(IDirectMusicPerformance8* dmPerf, IDirectMusicSegment8* dmSeg)
{
    // zatrzymuje odtwarzanie segmentu dmSeg
    dmPerf->StopEx(dmSeg, 0, 0);
}

// CloseDown()
// opis: kończy wykonanie
void CloseDown(IDirectMusicPerformance8* dmPerf)
{
    // zatrzymuje odtwarzanie
    dmPerf->Stop(NULL, NULL, 0, 0);

    // kończy pracę DirectMusic
    dmPerf->CloseDown();
}

/****************************************
*Interfejsy OpenGL
****************************************/
```

```
// CreateOutlineFont()
// opis: tworzy czcionkę obrysową za pomocą funkcji CreateFont()
unsigned int CreateOutlineFont(char *fontName, int fontSize, float depth)
{
    HFONT hFont;           // uchwyty czcionki systemu Windows
    unsigned int base;

    base = glGenLists(256);      // listy wyświetlania dla 96 znaków

    if (strcmp(fontName, "symbol") == 0)
    {
        hFont = CreateFont(fontSize, 0, 0, 0, FW_BOLD, FALSE, FALSE, FALSE,
                            SYMBOL_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
                            ANTIALIASED_QUALITY, FF_DONTCARE | DEFAULT_PITCH,
                            fontName);
    }
    else
    {
        hFont = CreateFont(fontSize, 0, 0, 0, FW_BOLD, FALSE, FALSE, FALSE,
                            ANSI_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
                            ANTIALIASED_QUALITY, FF_DONTCARE | DEFAULT_PITCH,
                            fontName);
    }

    if (!hFont)
        return 0;

    SelectObject(g_HDC, hFont);
    wglUseFontOutlines(g_HDC, 0, 255, base, 0.0f, depth, WGL_FONT_POLYGONS, gmf);

    return base;
}

// ClearFont()
// opis: usuwa listy wyświetlania czcionki
void ClearFont(unsigned int base)
{
    glDeleteLists(base, 256);
}

// PrintString()
// opis: wyświetla tekst wskazywany przez str
// za pomocą czcionki o początkowej liście wyświetlania base
void PrintString(unsigned int base, char *str)
{
    float length = 0;

    if ((str == NULL))
        return;

    // centruje tekst
    for (unsigned int loop=0;loop<(strlen(str));loop++)// długość tekstu w znakach
    {
        length+=gmf[str[loop]].gmfCellIncX; // zwiększa długość tekstu w pikselach
        // o szerokość znaku
    }
    glTranslatef(-length/2,0.0f,0.0f);      // centruje tekst
```

```
// rysuje tekst
glPushAttrib(GL_LIST_BIT);
glListBase(base);
glCallLists(strlen(str), GL_UNSIGNED_BYTE, str);
glPopAttrib();
}

// CleanUp()
// opis: zwalnia zasoby aplikacji
void CleanUp()
{
    ClearFont(listBase);
    dmusicLoader->Release();
    dmusicPerformance->Release();
    dmusicSegment->Release();
}

// Initialize
// opis: inicjuje OpenGL
void Initialize()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // tło w kolorze czarnym

    glShadeModel(GL_SMOOTH);           // cieniowanie gładkie
    glEnable(GL_DEPTH_TEST);          // usuwanie przesłoniętych powierzchni
    glEnable(GL_LIGHT0);              // włącza źródło światła light0
    glEnable(GL_LIGHTING);            // włącza oświetlenie
    glEnable(GL_COLOR_MATERIAL);      // kolor jako materiał

    listBase = CreateOutlineFont("Arial", 10, 0.25f); // ładuje czcionkę Arial 10 pkt.
}

// Render
// opis: rysuje scenę
void Render()
{
    // opróżnia bufory ekranu i głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    // odsuwa o 10 jednostek i obraca wokół wszystkich osi
    glTranslatef(0.0f, 0.0f, -15.0f);
    glRotatef(angle*0.9f, 1.0f, 0.0f, 0.0f);
    glRotatef(angle*1.5f, 0.0f, 1.0f, 0.0f);
    glRotatef(angle, 0.0f, 0.0f, 1.0f);

    // kolor niebieskawy
    glColor3f(0.3f, 0.4f, 0.8f);

    // wyświetla tekst
    PrintString(listBase, "DirectX Audio!");

    // kolor żółto-zielony
    glColor3f(0.6f, 0.8f, 0.5f);

    // wyświetla tekst
    glPushMatrix();
    glTranslatef(-3.0f, -1.0f, 0.0f);
```

```

    PrintString(listBase, "O - Odtwarzanie");
    glPopMatrix();
    glPushMatrix();
    glTranslatef(-3.0f, -2.0f, 0.0f);
    PrintString(listBase, "Z - Zatrzymanie");
    glPopMatrix();
    glPushMatrix();
    glTranslatef(-3.0f, -3.0f, 0.0f);
    PrintString(listBase, "ESC - Koniec");
    glPopMatrix();

    angle += 0.4f;

    SwapBuffers(g_HDC);           // przełącza bufory
}

// funkcja określająca format pikseli
void SetupPixelFormat(HDC hDC)
{
    int nPixelFormat;           // indeks formatu pikseli

    static PIXELFORMATDESCRIPTOR pfd = {
        sizeof(PIXELFORMATDESCRIPTOR), // rozmiar struktury
        1,                          // domyślna wersja
        PFD_DRAW_TO_WINDOW |        // grafika w oknie
        PFD_SUPPORT_OPENGL |       // grafika OpenGL
        PFD_DOUBLEBUFFER,          // podwójne buforowanie
        PFD_TYPE_RGBA,             // tryb kolorów RGBA
        32,                         // 32-bitowy opis kolorów
        0, 0, 0, 0, 0, 0,          // nie specyfikuje bitów kolorów
        0,                          // bez buforu alfa
        0,                          // nie specyfikuje bitu przesunięcia
        0,                          // bez bufora akumulacji
        0, 0, 0, 0,                // ignoruje bity akumulacji
        16,                        // 16-bit bufor z
        0,                          // bez bufora powielania
        0,                          // bez buforów pomocniczych
        PFD_MAIN_PLANE,           // główna płaszczyzna rysowania
        0,                          // zarezerwowane
        0, 0, 0 };                 // ignoruje maski warstw

    // wybiera najbardziej zgodny format pikseli
    nPixelFormat = ChoosePixelFormat(hDC, &pfd);

    // określa format pikseli dla danego kontekstu urządzenia
    SetPixelFormat(hDC, nPixelFormat, &pfd);
}

// procedura okienkowa
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HGLRC hRC;           // kontekst tworzenia grafiki
    static HDC hDC;              // kontekst urządzenia
    int width, height;          // szerokość i wysokość okna

    switch(message)
    {
        case WM_CREATE:         // okno jest tworzone

```

```
hDC = GetDC(hwnd);           // pobiera kontekst urządzenia dla okna
g_HDC = hDC;
SetupPixelFormat(hDC);      // wywołuje funkcję określającą format pikseli

// tworzy kontekst tworzenia grafiki i czyni go bieżącym
hRC = wglCreateContext(hDC);
wglMakeCurrent(hDC, hRC);

return 0;
break;

case WM_CLOSE:             // okno jest zamykane

// deaktywuje bieżący kontekst tworzenia grafiki i usuwa go
wglMakeCurrent(hDC, NULL);
wglDeleteContext(hRC);

// wstawia komunikat WM_QUIT do kolejki
PostQuitMessage(0);

return 0;
break;

case WM_SIZE:
height = HIWORD(lParam);   // pobiera nowe rozmiary okna
width = LOWORD(lParam);

if (height==0)             // unika dzielenia przez 0
{
    height=1;
}

glViewport(0, 0, width, height); // nadaje nowe wymiary oknu OpenGL
glMatrixMode(GL_PROJECTION);   // wybiera macierz rzutowania
glLoadIdentity();             // resetuje macierz rzutowania

// wyznacza proporcje obrazu
gluPerspective(54.0f,(GLfloat)width/(GLfloat)height,1.0f,1000.0f);

glMatrixMode(GL_MODELVIEW);   // wybiera macierz modelowania
glLoadIdentity();             // resetuje macierz modelowania

return 0;
break;

case WM_KEYDOWN:            // użytkownik naciągnął klawisz?
keyPressed[wParam] = true;
return 0;
break;

case WM_KEYUP:
keyPressed[wParam] = false;
return 0;
break;

default:
break;
}
```

```
    return (DefWindowProc(hwnd, message, wParam, lParam));
}

// punkt, w którym rozpoczyna się wykonywanie aplikacji
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
    WNDCLASSEX windowClass;           // klasa okna
    HWND    hwnd;                   // uchwyty okna
    MSG     msg;                   // komunikaty
    bool   done;                   // znacznik zakończenia aplikacji
    DWORD  dwExStyle;             // rozszerzony styl okna
    DWORD  dwStyle;               // styl okna
    RECT   windowRect;

    // zmienne pomocnicze
    int width = 800;
    int height = 600;
    int bits = 16;

    //fullScreen = TRUE;

    windowRect.left=(long)0;          // struktura określająca rozmiary okna
    windowRect.right=(long)width;
    windowRect.top=(long)0;
    windowRect.bottom=(long)height;

    // definicja klasy okna
    windowClass.cbSize= sizeof(WNDCLASSEX);
    windowClass.style= CS_HREDRAW | CS_VREDRAW;
    windowClass.lpfnWndProc= WndProc;
    windowClass.cbClsExtra= 0;
    windowClass.cbWndExtra= 0;
    windowClass.hInstance= hInstance;
    windowClass.hIcon= LoadIcon(NULL, IDI_APPLICATION); // domyślna ikona
    windowClass.hCursor= LoadCursor(NULL, IDC_ARROW);   // domyślny kursor
    windowClass.hbrBackground= NULL;                    // bez tła
    windowClass.lpszMenuName= NULL;                    // bez menu
    windowClass.lpszClassName= "MojaKlasa";
    windowClass.hIconSm= LoadIcon(NULL, IDI_WINLOGO); // logo Windows

    // rejestruje klasę okna
    if (!RegisterClassEx(&windowClass))
        return 0;

    if (fullScreen)                         // tryb pełnoekranowy?
    {
        DEVMODE dmScreenSettings;           // tryb urządzenia
        memset(&dmScreenSettings,0,sizeof(dmScreenSettings));
        dmScreenSettings.dmSize = sizeof(dmScreenSettings);
        dmScreenSettings.dmPelsWidth = width; // szerokość ekranu
        dmScreenSettings.dmPelsHeight = height; // wysokość ekranu
        dmScreenSettings.dmBitsPerPel = bits; // bitów na piksel
        dmScreenSettings.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;

        //
        if (ChangeDisplaySettings(&dmScreenSettings, CDS_FULLSCREEN) !=
            DISP_CHANGE_SUCCESSFUL)
```

```
{  
    // przełączenie trybu nie powiodło się, z powrotem tryb okienkowy  
    MessageBox(NULL, "Display mode failed", NULL, MB_OK);  
    fullScreen=FALSE;  
}  
}  
  
if (fullScreen)                                // tryb pełnoekranowy?  
{  
    dwExStyle=WS_EX_APPWINDOW;                  // rozszerzony styl okna  
    dwStyle=WS_POPUP;                          // styl okna  
    ShowCursor(FALSE);                        // ukrywa kurSOR myszy  
}  
else  
{  
    dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE; // rozszerzony styl okna  
    dwStyle=WS_OVERLAPPEDWINDOW;                // styl okna  
}  
  
AdjustWindowRectEx(&windowRect, dwStyle, FALSE, dwExStyle); // koryguje rozmiar okna  
  
// tworzy okno  
hwnd = CreateWindowEx(  
    NULL,                                         // rozszerzony styl okna  
    "MojaKlasa",                                  // nazwa klasy  
    "DirectX Audio, przykład pierwszy: odtwarzanie", // nazwa aplikacji  
    dwStyle | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,  
    0, 0,                                         // współrzędne x,y  
    windowRect.right - windowRect.left,           // szerokość, wysokość  
    windowRect.bottom - windowRect.top,            // uchwyty okna nadzawanego  
    NULL,                                         // uchwyty menu  
    NULL,                                         // instancja aplikacji  
    hInstance,                                     // bez dodatkowych parametrów  
    NULL);  
  
// sprawdza, czy utworzenie okna nie powiodło się (wtedy wartość hwnd równa NULL)  
if (!hwnd)  
    return 0;  
  
// inicjuje COM  
if (FAILED(CoInitialize(NULL)))  
    return 0;                                       // jeśli inicjacja COM nie powiodła się.  
                                                // program kończy działanie  
  
// inicjuje DirectX Audio  
if (!InitDirectXAudio(hwnd))  
    return 0;  
  
// ładuje segment  
if (!LoadSegment(hwnd, "canyon.mid"))  
    return 0;  
  
// odtwarza segment  
PlaySegment(dmusicPerformance, dmusicSegment);  
  
ShowWindow(hwnd, SW_SHOW);                      // wyświetla okno  
UpdateWindow(hwnd);                            // aktualizuje okno
```

```
done = false;           // inicjuje zmienną warunku pętli
Initialize();          // inicjuje OpenGL

// pętla przetwarzania komunikatów
while (!done)
{
    PeekMessage(&msg, hwnd, NULL, NULL, PM_REMOVE);

    if (msg.message == WM_QUIT)    // aplikacja otrzymała komunikat WM_QUIT?
    {
        done = true;            // jeśli tak, to kończy działanie
    }
    else
    {
        // odtwarza segment
        if (keyPressed['0'] || keyPressed['o'])
        {
            if (dmusicPerformance->IsPlaying(dmusicSegment, NULL) != S_OK)
                PlaySegment(dmusicPerformance, dmusicSegment);
        }

        // zatrzymuje odtwarzanie
        if (keyPressed['Z'] || keyPressed['z'])
        {
            if (dmusicPerformance->IsPlaying(dmusicSegment, NULL) == S_OK)
                StopSegment(dmusicPerformance, dmusicSegment);
        }

        // kończy działanie programu
        if (keyPressed[VK_ESCAPE])
            done = true;
        else
        {
            Render();             // rysuje grafikę

            TranslateMessage(&msg); // tłumaczy komunikat i wysyła do systemu
            DispatchMessage(&msg);
        }
    }
}

// kończy wykonanie
CloseDown(dmusicPerformance);

// zwalnia zasoby
CleanUp();

// kończy pracę COM
CoUninitialize();

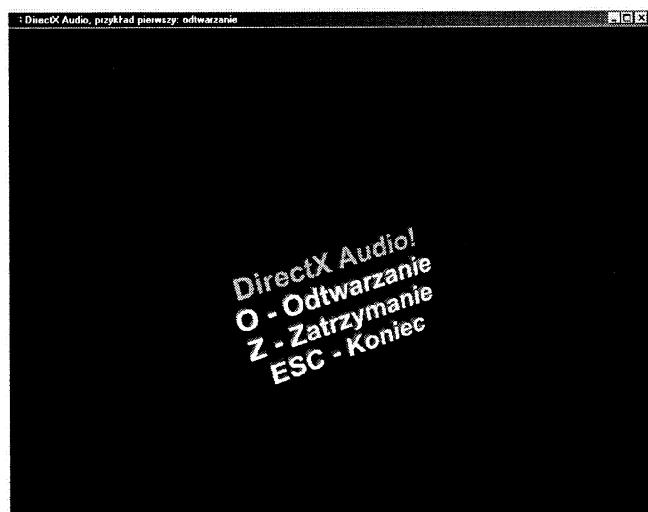
if (fullScreen)
{
    ChangeDisplaySettings(NULL,0); // przywraca pulpit
    ShowCursor(TRUE);           // i wskaźnik myszy
}

return msg.wParam;
}
```

Program ten wyświetla obracający się tekst pokazany na rysunku 17.5 odtwarzając w tle plik *canyon.mid*. Naciśnięcie klawisza *Z* powoduje zatrzymanie jego odtwarzania, a klawisza *O* wznowienie odtwarzania. Program kończy pracę po naciśnięciu klawisza *Esc*.

**Rysunek 17.5.**

Przykład  
zastosowania  
DirectX Audio



## Zastosowania ścieżek dźwięku

Ścieżki dźwięku zarządzają przepływem danych dźwięku poprzez różne obiekty od momentu załadowania tych danych z pliku aż do ich wysłania do głośników. Ścieżka dźwięku obejmować może obiekt wykonania, segment, graf narzędzi, syntezator i bufory DirectSound. Jak już pokazano, jeśli nie korzysta się z przestrzennych efektów dźwiękowych i odtwarza na przykład pliki *MIDI*, można skorzystać z domyślnej ścieżki dźwięku. Aby wykorzystać pełne możliwości DirectX Audio, trzeba stosować własne ścieżki dźwięku.

Ścieżkę dźwięku można skonfigurować na cztery różne sposoby:

- ◆ utworzyć jedną lub więcej standardowych ścieżek dźwięku za pomocą metody `IDirectMusicPerformance8::CreateStandardAudioPath()`;
- ◆ utworzyć domyślną, standardową ścieżkę dźwięku za pomocą metody `IDirectMusicPerformance8::InitAudio()`;
- ◆ uzyskać konfigurację ścieżki dźwięku z pliku stworzonego przez DirectMusic Producer i przekazać obiekt konfiguracji metodzie `IDirectMusicPerformance8::CreateAudioPath()`;
- ◆ za pomocą DirectMusic utworzyć nową ścieżkę dźwięku ze ścieżki dźwięku odtwarzanego segmentu.

Posługując się metodami `IDirectMusicLoader8::GetObject()` lub `IDirectMusicLoader8::LoadObjectFromFile()` można załadować obiekt konfiguracji ścieżki dźwięku w taki sam sposób, jak każdy inny obiekt DirectX Audio. Konfigurację ścieżki dźwięku dla danego segmentu pobiera się za pomocą metody `IDirectMusicSegment8::GetAudioPathConfig()`.

Konfiguracje ścieżek dźwięku nie posiadają własnego interfejsu, a tym samym własnych metod. Dlatego też obiektu konfiguracji nie można zmodyfikować, a jedynie przekazać metodzie `IDirectMusicPerformance8::CreateAudioPath()`.

## Domyślna ścieżka dźwięku

Domyślna ścieżka dźwięku używana jest podczas odtwarzania segmentu za pomocą metody `IDirectMusicPerformance8::PlaySegment()` lub `IDirectMusicPerformance8::PlaySegmentEx()`. Aby utworzyć nową ścieżkę dźwięku i uczynić ją ścieżką domyślną, trzeba określić typ standardowej ścieżki dźwięku za pomocą parametru `dwDefaultType` metody `IDirectMusicPerformance8::InitAudio()`.

Domyślną ścieżkę dźwięku określa się wywołując metodę `IDirectMusicPerformance8::SetDefaultAudioPath()` zdefiniowaną jak poniżej:

```
HRESULT SetDefaultAudioPath(
    IDirectMusicAudioPath *pAudioPath    // interfejs domyślnej ścieżki dźwięku
);
```

Domyślną ścieżkę dźwięku pobiera się za pomocą metody `IDirectMusicPerformance8::GetDefaultAudioPath()` zdefiniowanej następująco:

```
HRESULT GetDefaultAudioPath(
    IDirectMusicAudioPath **ppAudioPath // domyślna ścieżka dźwięku
);
```

## Standardowe ścieżki dźwięku

Pomijając przypadki, w których korzysta się wyłącznie ze ścieżek tworzonych na podstawie obiektów konfiguracji ścieżek dźwięku, zwykle w programie tworzy się jedną lub więcej standardowych ścieżek dźwięku za pomocą metody `IDirectMusicPerformance8::CreateStandardAudioPath()`:

```
HRESULT CreateStandardAudioPath(
    DWORD dwType,                // typ ścieżki
    DWORD dwChannelCount,        // liczba kanałów wykonania
    BOOL fActive,                // true = ścieżka aktywna zaraz po utworzeniu
    IDirectMusicAudioPath **ppNewPath // adres wskaźnika ścieżki dźwięku
);
```

Standardową ścieżkę dźwięku można także utworzyć korzystając z parametru `dwDefaultPathType` metody `IDirectMusicPerformance8::InitAudio()`.

Parametr `dwType` metody `CreateStandardAudioPath()` identyfikuje typ tworzonej ścieżki dźwięku. Tabela 17.1 przedstawia typy ścieżek dźwięku i używane przez nie buforów. Niektóre z tych buforów mogą być współużytkowane przez wiele ścieżek dźwięku.

**Tabela 17.1.** Typy standardowych ścieżek dźwięku

<b>Typ ścieżki dźwięku</b>	<b>Standardowe bufory</b>	<b>Współużytkowanie buforów</b>
DMUS_APATH_DYNAMIC_3D	Efekt przestrzenny	nie
DMUS_APATH_DYNAMIC_MONO	Monofoniczny	nie
DMUS_APATH_DYNAMIC_STEREO	Stereofoniczny	nie
DMUS_APATH_SHARED_STEREOPLUSREVERB	Stereofoniczny, pogłos	tak, tak

Poniższy fragment kodu tworzy ścieżkę dźwięku umożliwiającą realizację efektu przestrzennego:

```

IDirectMusicAudioPath8 *dmusic3DAudioPath = NULL; // interfejs ścieżki dźwięku

if (FAILED(dmusicPerformance->CreateStandardAudioPath(DMUS_APATH_DYNAMIC_3D,
                                                       128, TRUE,
                                                       &dmusic3DAudioPath)))
{
    // ścieżka dźwięku została utworzona
}
else
    // ścieżka dźwięku nie została utworzona

```

### **Odtwarzanie za pomocą ścieżki dźwięku**

Dotychczas segment odtwarzany był za pomocą domyślnej ścieżki dźwięku i wobec tego parametrowi *pAudioPath* metody `IDirectMusicPerformance8::PlaySegmentEx()` trzeba było nadać wartość `NULL`. Aby skorzystać z innej ścieżki dźwięku, można nadal używać metody `IDirectMusicPerformance8::PlaySegmentEx()`, ale ścieżkę dźwięku należy określić za pomocą jednego z poniższych sposobów:

- ◆ dostarczając wskaźnik obiektu interfejsu ścieżki dźwiękowej jako wartość parametru *pAudioPath*;
  - ◆ umieszczając znacznik **DMUS\_SEGF\_USE\_AUDIOPATH** jako element parametru *dwFlags*, co spowoduje, że segment utworzy ścieżkę dźwięku na podstawie konfiguracji zawartej w obiekcie segmentu.

Tabela 17.2 przedstawia dopuszczalne wartości znaczników przekazywanych za pośrednictwem parametru *dwFlags* funkcji `PlaySegmentEx()`.

Aby odtworzyć segment za pomocą ścieżki dźwiękowej (nie korzystając przy tym ze znaczników), należy wywołać metodę `PlaySegmentEx()` w następujący sposób:

```
dmusicPerformance->PlaySegmentEx(dmusicSegment, NULL, NULL, 0, 0, NULL,  
    dmusic3DAudioPath);
```

Dotąd pomijana była kwestia odtwarzania efektów dźwiękowych. Odtwarza się je jako segmenty wtórne za pomocą znacznika **DMUS SEGFI SECONDARY**:

```
dmusicPerformance->PlaySegmentEx(dmusicSegment, NULL, NULL,  
    DMUS_SEGF_DEFAULT | DMUS_SEGF_SECONDARY, 0, NULL,  
    dmusic3DAudioPath);
```

**Tabela 17.2.** Znaczniki metody *PlaySegmentEx()*

Znacznik	Znaczenie
DMUS_SEGF_REFTIME	Parametry czasowe podawane są jako wartości względne
DMUS_SEGF_SECONDARY	Segment wtórny
DMUS_SEGF_QUEUE	Umieszcza segment na końcu kolejki segmentów zasadniczych (dotyczy tylko segmentów zasadniczych)
DMUS_SEGF_CONTROL	Odtwarza jako segment kontrolny (dotyczy tylko segmentów wtórnego)
DMUS_SEGF_AFTERPREPARETIME	Odtwarza po czasie przygotowania
DMUS_SEGF_GRID	Odtwarza na granicy siatki
DMUS_SEGF_BEAT	Odtwarza na granicy akordu
DMUS_SEGF_MEASURE	Odtwarza na granicy miary
DMUS_SEGF_DEFAULT	Używa domyślnej granicy segmentu
DMUS_SEGF_NOINVALIDATE	Nowy segment nie powoduje przerwania odtwarzania segmentu bieżącego
DMUS_SEGF_ALIGN	Początek segmentu może zostać wyrównany do granicy, która już minęła
DMUS_SEGF_VALID_START_BEAT	Odtwarzanie może rozpocząć się od dowolnego akordu
DMUS_SEGF_VALID_START_GRID	Odtwarzanie może rozpoczęć się od dowolnego punktu siatki
DMUS_SEGF_VALID_START_TICK	Odtwarzanie może rozpoczęć się w dowolnym czasie
DMUS_SEGF_AUTOTRANSITION	Tworzy i odtwarza segment przejściowy na podstawie szablonu
DMUS_SEGF_AFTERQUEUEUTIME	Odtwarza po czasie kolejkowania (domyślnie dla wszystkich segmentów zasadniczych)
DMUS_SEGF_AFTERLATENCYTIME	Odtwarza po czasie opóźnienia (domyślnie dla wszystkich segmentów)
DMUS_SEGF_SEGMENTEND	Odtwarza po segmencie, który odtwarzany jest od podanego czasu rozpoczęcia odtwarzania (inne segmenty znajdujące się w kolejce są usuwane)
DMUS_SEGF_MARKER	Odtwarza po kolejnym markerze segmentu zasadniczego
DMUS_SEGF_TIMESIG_ALWAYS	Wyrównuje czas rozpoczęcia odtwarzania do bieżącej sygnatury czasu (nawet w przypadku, gdy nie istnieje segment zasadniczy)
DMUS_SEGF_USE_AUDIOPATH	Korzysta ze ścieżki dźwięku zawartej w segmencie (aby segment ten został prawidłowo odtworzony, musi być aktywne automatyczne ładowanie instrumentów)
DMUS_SEGF_VALID_START_MEASURE	Odtwarzanie może rozpoczęć się od początku miary

W przypadku odtwarzania segmentów wtórnego nie trzeba określić ścieżki dźwięku. W powyższym wywołaniu wartość ostatniego parametru mogłaby więc równie dobrze wynosić NULL.

Głośność kanałów wykonania podczas odtwarzania dźwięku za pomocą ścieżki dźwięku kontroluje się za pomocą metody `IDirectMusicAudioPath8::SetVolume()` zdefiniowanej w następujący sposób:

```
HRESULT SetVolume(  
    long lVolume, // określa tłumienie wyrażone w setkach decybeli  
    // przyjmując wartości z przedziału od -9600 do 0.  
    // gdzie 0 oznacza pełną głośność  
    DWORD dwDuration // określa czas w milisekundach, w którym odbywa się zmiana głośności  
    // wartość 0 oznacza największą sprawność zmiany głośności  
);
```

Aby uzyskać pełną głośność dźwięku w ciągu pół sekundy, metodę SetVolume() należy wywołać jak poniżej:

```
dmusic3DAudioPath->SetVolume(0, 500);
```

Głośność określana za pomocą metody SetVolume() nie jest głośnością globalną, lecz raczej głośnością kanałów wykonania używanych podczas odtwarzania za pomocą ścieżki dźwięku.

## Pobieranie obiektów należących do ścieżek dźwięku

W praktyce zdarza się, że trzeba nierzaz uzyskać interfejs jednego z obiektów należących do ścieżki dźwięku. Na przykład w sytuacji, gdy należy zmienić właściwości przestrzennego efektu.

Aby pobrać obiekt ze ścieżki dźwięku, należy wywołać metodę IDirectMusicAudioPath8::GetObjectInPath() zdefiniowaną jak poniżej:

```
HRESULT GetObjectInPath(  
    DWORD dwPChannel, // liczba przeszukiwanych kanałów wykonania  
    // wartość DMUS_PCHANNEL_ALL oznacza wszystkie  
    DWORD dwStage, // etap ścieżki  
    DWORD dwBuffer, // indeks bufora, gdy etap ścieżki równy jest DMO  
    // lub gdy jest to bufor miksujący  
    REFGUID guidObject, // identyfikator klasy  
    DWORD dwIndex, // indeks obiektu na liście dopasowania; 0 dla pierwszego obiektu  
    REGUID iidInterface, // identyfikator interfejsu, na przykład IID_IDirectMusicGraph  
    void **ppObject // adres zmiennej, która zawierać będzie wskaźnik interfejsu  
);
```

Poniższy fragment kodu ilustruje sposób dostępu do bufora standardowej ścieżki dźwięku:

```
IDirectMusicAudioPath *dmusicAudioPath;  
IDirectSoundBuffer8 *dmusicSoundBuff;  
  
// tworzy standardową ścieżkę dźwięku zawierającą bufor pogłosu  
// ścieżka ta nie jest aktywna  
dmusicPerformance->CreateStandardAudioPath(DMUSIC_APATH_DYNAMIC_3D, 64,  
    FALSE, &dmusicAudioPath);  
  
// pobiera wskaźnik obiektu bufora  
dmusicAudioPath->GetObjectInPath(DMUS_PCHANNEL_ALL, DMUS_PATH_BUFFER, 0,  
    GUID_NULL, 0, IID_IDirectSoundBuffer8,  
    (void **)&dmusicSoundBuff);
```

# Dźwięk przestrzenny

Efekt przestrzennego dźwięku można łatwo zastosować do bufora DirectSound, który pobrać należy ze ścieżki dźwięku. Trójwymiarowość dźwięku ma szczególne znaczenie w grach, w przypadku których może zrobić na użytkowniku nawet większe wrażenie niż najdoskonalsza grafika.

## Współrzędne przestrzeni dźwięku

Zanim zacznie się korzystać z efektu dźwięku przestrzennego, trzeba najpierw poznać sposób określania współrzędnych przestrzeni dźwięku stosowany w DirectX Audio. Położenie, prędkość i orientacja źródła dźwięku definiowane są w kartezjańskim układzie współrzędnych na osiach x, y i z. Osie te są położone w taki sam sposób względem obserwatora jak osie układu współrzędnych używanego przez OpenGL z jednym wyjątkiem: oś z posiada przeciwny zwrot. Inaczej mówiąc, ujemna część osi zamiast znajdować się „za” płaszczyzną ekranu, znajduje się tym razem „przed” ekranem. Dodatnia część osi z będzie wobec tego znajdować się „za” ekranem.

Położenie źródła dźwięku określane jest w metrach. Można to zmienić stosując współczynnik odległości określający liczbę metrów przypadających na jednostkę odległości zdefiniowaną przez aplikację. Prędkość mierzona jest w jednostkach aplikacji na sekundę.

Orientacja źródła dźwięku określana jest w jednostkach aplikacji względem orientacji świata. Na przykład jeśli świat zorientowany jest w kierunku ujemnej części osi z (co odpowiada kierunkowi południowemu w DirectX Audio), a orientacja odbiorcy dźwięku określona jest przez wektor  $(1, 0, 0)$ , to będzie on zorientowany w kierunku zachodnim.

Wektory używane przez DirectX Audio pochodzą z komponentu DirectX odpowiedzialnego za tworzenie grafiki trójwymiarowej, czyli Direct3D. Wektor zdefiniowany jest w tym przypadku w następujący sposób:

```
typedef struct {
    float x;
    float y;
    float z;
} D3DXVECTOR;
```

## Percepcja położenia źródła dźwięku

Sposób, w jaki odbierany jest dźwięk pochodzący od źródła znajdującego się w pewnym punkcie przestrzeni, zależy od wielu czynników (nie wyłączając wzroku)! Do czynników związanych bezpośrednio z dźwiękiem należą:

- ◆ **ogólna głośność** — gdy źródło dźwięku oddala się od obserwatora, to natężenie dźwięku maleje;
- ◆ **różnica natężenia dźwięku dobiegającego do obu uszu** — dźwięk pochodzący ze źródła znajdującego się po prawej stronie obserwatora jest odbierany jako głośniejszy prawym uchem niż lewym;

- ◆ **różnica czasu, w którym dźwięk dociera do obu uszu** — dźwięk pochodzący ze źródła znajdującego się po prawej stronie obserwatora dociera nieco wcześniej do ucha prawnego niż do ucha lewego (różnica ta jest rzędu pojedynczych milisekund);
- ◆ **tlumienie** — kształt i orientacja uszu powodują, że dźwięki docierające do nas z tyłu są bardziej tłumione niż dźwięki docierające z przodu; również dźwięk pochodzący ze źródła znajdującego się po prawej stronie obserwatora dociera bardziej stłumiony do ucha lewego niż do prawnego.

Wymienione powyżej czynniki nie wyczerpują tematu, ale posiadają zasadnicze znaczenie dla odbioru położenia źródła dźwięku i zostały zaimplementowane w DirectX Audio.

## Bufor efektu przestrzennego w DirectSound

Każde źródło dźwięku reprezentowane jest przez interfejs IDirectSound3DBuffer8. Źródło takie musi być ze swojej natury monofoniczne, czyli jego dźwięk będzie docierać do odbiorcy jednym kanałem. Przy próbie uzyskania efektu przestrzennego dla źródła stereofonicznego uzyskany zostanie błąd.

Aby stworzyć przestrzenne źródło dźwięku, trzeba najpierw uzyskać interfejs IDirectSound3DBuffer8. Jak już pokazano to omawiając ścieżki dźwięku, interfejs taki można pobrać za pomocą metody IDirectMusicPerformance8::GetObjectInPath() w następujący sposób:

```
IDirectMusicAudioPath *dmusicAudioPath;  
IDirectSound3DBuffer8 *dmusic3DSoundBuff;  
  
// tworzy ścieżkę dźwięku z buforem efektu przestrzennego  
// (operacja zbędna, jeśli już taką posiadam)  
if (FAILED(dmusicPerformance->CreateStandardAudioPath(DMUS_APATH_DYNAMIC_3D,  
64, TRUE,  
&dmusic3DAudioPath)))  
    return 0;  
  
// pobiera bufor efektu przestrzennego  
if (FAILED(dmusic3DAudioPath->GetObjectInPath(0, DMUS_PATH_BUFFER, 0, GUID_NULL, 0,  
IID_IDirectSound3DBuffer,  
(void**)&ds3DBuffer)))  
    return 0;
```

## Parametry przestrzennego źródła dźwięku

Parametry te można konfigurować pojedynczo lub wsadowo. W pierwszym przypadku trzeba skorzystać z odpowiedniej metody interfejsu IDirectSound3DBuffer8, które przedstawia tabela 17.3.

Wszystkie te metody opisane są szczegółowo w dokumentacji DirectX 8.

**Tabela 17.3.** Metody określania pojedynczych parametrów przestrzennego źródła dźwięku

Kategoria parametru	Metody
Odległość	GetMaxDistance GetMinDistance SetMaxDistance SetMinDistance
Tryb działania	GetMode SetMode
Pozycja	GetPosition SetPosition
Stożek rozchodzenia się dźwięku	GetConeAngles GetConeOrientation GetConeOutsideVolume SetConeAngles SetConeOrientation SetConeOutsideVolume
Prędkość ruchu źródła	GetVelocity SetVelocity

Zwykle jednak trzeba nadawać lub pobierać wartość wszystkich parametrów za jednym razem. Służą temu metody `IDirectSound3DBuffer8::SetAllParameters()` i `IDirectSound3DBuffer8::GetAllParameters()`. Metoda `SetAllParameters()` posiada następującą definicję:

```
HRESULT SetAllParameters(
    LPCDS3DBUFFER pcDs3DBuffer, // struktura DS3DBUFFER opisująca charakterystykę źródła
    DWORD dwApply           // określa, kiedy zastosować nowe parametry
                           // zwykle wartość DS3D_IMMEDIATE (natychmiast)
);
```

Jak łatwo zauważyc, aby korzystać z tej metody, trzeba utworzyć i wypełnić strukturę typu DS3DBUFFER. Zdefiniowano ją w następujący sposób:

```
typedef struct _DS3DBUFFER
{
    DWORD      dwSize;          // rozmiar struktury DS3DBUFFER
    D3DVECTOR vPosition;       // położenie źródła dźwięku
    D3DVECTOR vVelocity;       // prędkość źródła dźwięku
    DWORD      dwInsideConeAngle; // kąt rozwarcia wewnętrznego stożka dźwięku
    DWORD      dwOutsideConeAngle; // kąt rozwarcia zewnętrznego stożka dźwięku
    D3DVECTOR vConeOrientation; // orientacja stożków dźwięku
    LONG       lConeOutsideVolume; // głośność na zewnątrz stożka dźwięku
    D3DVALUE   fMinDistance;    // minimalna odległość
    D3DVALUE   fMaxDistance;    // maksymalna odległość
    DWORD      dwMode;          // tryb działania
} DS3DBUFFER, *LPDS3DBUFFER;

typedef const DS3DBUFFER *LPCDS3BUFFER;
```

Pole *dwMode* struktury DS3DBUFFER może przyjmować następujące wartości:

- ◆ DS3DMODE\_DISABLE — efekt przestrzenny nie jest wykorzystywany;
- ◆ DS3DMODE\_HEADRELATIVE — parametry źródła dźwięku określone są względem obserwatora (jeśli położenie obserwatora zmienia się, to bezwzględne wartości parametrów źródła dźwięku są aktualizowane tak, by pozostawały one stałe względem obserwatora);
- ◆ DS3DMODE\_NORMAL — zwykły, domyślny tryb pracy.

Teraz należy przyjrzeć się po kolei polom struktury DS3DBUFFER.

## Odległość minimalna i maksymalna

Przez odległość minimalną należy rozumieć odległość źródła od obserwatora, przy której słyszy on maksymalne natężenie dźwięku. Gdy obserwator zbliża się do źródła dźwięku, jego natężenie wzrasta. W połowie drogi będzie dwukrotnie większe niż na początku, ale po osiągnięciu odległości minimalnej przestanie dalej wzrastać.

Odległość minimalna jest szczególnie przydatna, gdy trzeba uzyskać różnicę głośności dwóch różnych źródeł dźwięku. Najlepiej będzie rozważyć to na przykładzie silnika odrzutowca i agregatu klimatyzacji. Chociaż pierwsze z tych źródeł generuje dźwięk o dużo większym natężeniu, to w praktyce oba będą trzeba zapisać z tym samym poziomem głośności. Aby uzyskać realistyczny efekt podczas ich odtwarzania, należy skonfigurować minimalną odległość dla źródła dźwięku reprezentującego odrzutowiec tak, by wynosiła 100 metrów, a dla klimatyzacji tylko 1 metr. Tak więc w przypadku, gdy obserwator znajdzie się w odległości 200 metrów od odrzutowca natężenie dźwięku zmniejsza się dwukrotnie. Podobnie w przypadku odległości 2 metrów od agregatu klimatyzacji. Domyślna wartością parametru odległości minimalnej jest DS3D\_DEFAULTMININSTANCE i wynosi on 1 jednostkę (czyli 1 metr przy domyślnej wartości współczynnika odległości). Jeśli nie zostanie zmieniona odległość minimalna, to maksymalne natężenie dźwięku będzie w odległości 1 metra od źródła, dwukrotnie mniejsze w odległości 2 metrów, czterokrotnie w odległości 4 metrów i tak dalej.

Odległość maksymalna określa natomiast odległość od źródła dźwięku, po przekroczeniu której natężenie dźwięku przestaje maleć. Domyślną wartością tego parametru jest DS3D\_DEFAULTMAXDISTANCE (wynosi miliard metrów). W praktyce oznacza to, że głośność dźwięku wyznaczana jest nawet przy odległościach, z których w rzeczywistości nie jest on słyszalny.

Różnica pomiędzy odlegością maksymalną i minimalną ma wpływ na efekt tłumienia dźwięku. Jeśli jest ona duża to natężenie dźwięku nie zmienia się tak gwałtownie, jak w przypadku małej różnicy.

## Tryb działania

Dostępne są trzy tryby działania przestrzennego efektu dźwięku:

- ◆ zwykły — w trybie tym parametry źródła dźwięku określają jego bezwzględne położenie i charakterystykę;

- ◆ **względem obserwatora** — zmiana położenia i orientacji obserwatora powoduje automatyczną zmianę parametrów źródła dźwięku.
- ◆ **wyłączony** — efekt przestrzennego źródła dźwięku nie jest wykorzystywany.

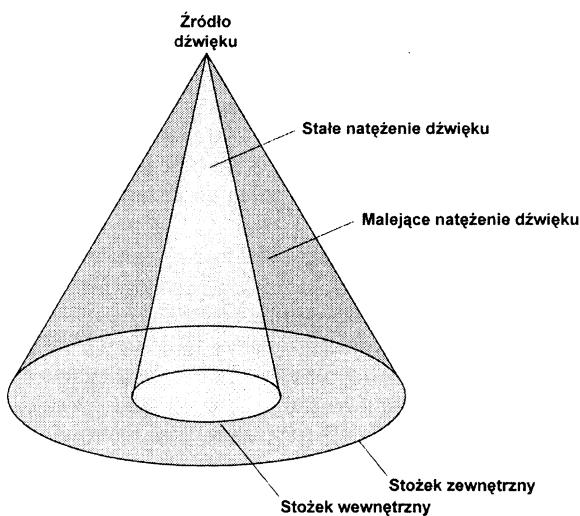
## Położenie i prędkość

W zależności od trybu działania efektu przestrzennego położenie źródła dźwięku reprezentowane jest za pomocą współrzędnych świata bądź względem obserwatora. Prędkość źródła dźwięku wykorzystywana jest do wyznaczenia efektu Dopplera.

## Stożki dźwięku

Stożki dźwięku pozwalają zdefiniować kierunkową głośność dźwięku. Zwykłe źródło emisji dźwięku o takiej samej amplitudzie we wszystkich kierunkach. Stożek dźwięku składa się z części zewnętrznej i wewnętrznej, co ilustruje rysunek 17.6. Wewnątrz stożka zewnętrznego natężenie dźwięku jest takie samo jak w przypadku, gdy nie używa się stożków. Na zewnątrz stożka zewnętrznego dźwięk podlega tłumieniu, które można konfigurować. Wartość natężenia dźwięku na zewnątrz tego stożka jest ujemna i wyrażona w setkach decybeli, ponieważ reprezentuje tłumienie względem domyślnej wartości głośności równej 0.

**Rysunek 17.6.**  
Stożek dźwięku



Pomiędzy obydwooma stożkami znajduje się strefa przejściowa, w której natężenie dźwięku maleje ze wzrostem kąta rozwarcia stożka.

Jednym z zastosowań stożków dźwięku jest symulacja dźwięku dobiegającego z pomieszczenia poprzez otwarte drzwi. Źródło dźwięku należy umieścić w pomieszczeniu i zorientować je w kierunku drzwi w taki sposób, że wewnętrzny stożek będzie wypełniał szerokość otworu drzwi, a stożek zewnętrzny będzie nieco większy. Obserwator przechodząc koło otworu drzwiowego usłyszy dobiegający z pomieszczenia dźwięk, którego natężenie będzie wygasac po minięciu drzwi.

## Odbiorca efektu dźwięku przestrzennego

Efekt dźwięku przestrzennego nie zależy wyłącznie od parametrów jego źródła, ale także od położenia, orientacji i prędkości odbiorcy dźwięku (obserwatora). Domyślnie odbiorca dźwięku znajduje się w środku układu współrzędnych, nie porusza się i spogląda w kierunku dodatniej części osi z. Zmieniając jego położenie, orientację i prędkość można odpowiednio dopasować efekt dźwięku przestrzennego korzystając z efektu Dopplera i zmieniając głośność dźwięku wraz ze zmianą odległości od jego źródła. W przeciwieństwie do źródeł dźwięku, których może być wiele, istnieje tylko jeden odbiorca dźwięku.

Orientacja odbiorcy zdefiniowana jest za pomocą dwóch wektorów: wskazującego kierunek „do przodu” i „w górę”. Jeśli wektory te nie będą tworzyć odpowiedniego kąta, to DirectX Audio zmodyfikuje składowe wektora wskazującego kierunek „do przodu”.

Aby pobrać obiekt odbiorcy dźwięku korzysta się z metody `IDirectMusicAudioPath8::GetObjectInPath()` w podobny sposób jak podczas pobierania obiektu bufora:

```
IDirectSound3DListener8 *dmusicListener;
// pobiera obiekt odbiorcy dźwięku ze ścieżki dźwięku
if (FAILED(dmusic3DAudioPath->GetObjectInPath(DMUS_PCHANNEL_ALL,
DMUS_PATH_PRIMARY_BUFFER,
0, GUID_NULL, 0,
IID_IDirectSound3DListener,
(void**)&dmusicListener)))
    return 0;
```

Parametry pobranego obiektu określa się w podobny sposób jak parametry źródła dźwięku. Można konfigurować je pojedynczo za pomocą metod wymienionych w tabeli 17.4 lub skorzystać z metod `IDirectSound3DListener8::SetAllParameters()` i `IDirectSound3DListener8::GetAllParameters()`.

**Tabela 17.4.** Metody interfejsu `IDirectSound3DListener8`

Kategoria	Metody
Współczynnik odległości	<code>GetDistanceFactor</code> <code>SetDistanceFactor</code>
Współczynnik efektu Dopplera	<code>GetDopplerFactor</code> <code>SetDopplerFactor</code>
Orientacja	<code>GetOrientation</code> <code>SetOrientation</code>
Położenie	<code>GetPosition</code> <code>SetPosition</code>
Współczynnik tłumienia	<code>GetRolloffFactor</code> <code>SetRolloffFactor</code>
Prędkość	<code>GetVelocity</code> <code>SetVelocity</code>

Jak pokazuje poniższa definicja metody SetAllParameters() parametry odbiorcy dźwięku określa się przy użyciu struktury typu DS3DLISTENER:

```
HRESULT SetAllParameters(
    DS3DLISTENER pcListener, // parametry odbiorcy dźwięku
    DWORD dwApply           // kiedy należy je zastosować
);
```

Struktura DS3DLISTENER zdefiniowana jest następująco:

```
typedef struct
{
    DWORD     dwSize;          // rozmiar struktury DS3DLISTENER w bajtach
    D3DVECTOR vPosition;      // położenie odbiorcy dźwięku
    D3DVECTOR vVelocity;      // prędkość odbiorcy dźwięku
    D3DVECTOR vOrientationFront; // wektor wskazujący kierunek "do przodu"
    D3DVECTOR vOrientationTop; // wektor wskazujący kierunek "w górę"
    D3DVALUE fDistanceFactor; // współczynnik odległości
    D3DVALUE fRollOffFactor;  // współczynnik tłumienia
    D3DVALUE fDopplerFactor;  // współczynnik efektu Dopplera
} DS3DLISTENER, *LPDS3DLISTENER;

typedef const DS3DLISTENER *LPCDS3DLISTENER;
```

Aby określić wartości parametrów, trzeba nadać je polom struktury DS3DLISTENER i wywołać metodę IDirectSound3DListener8::SetAllParameters().

## Przykład zastosowania efektu przestrzennego

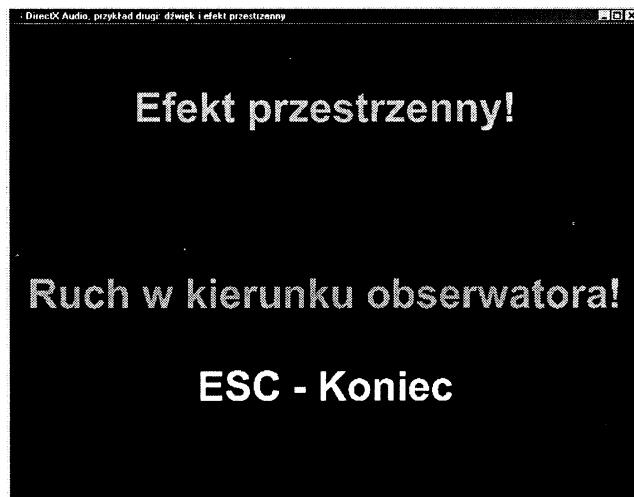
Zamieszczony poniżej program ilustruje sposób użycia efektu przestrzennego w DirectX Audio. Jego część graficzna rysuje tekst w ten sam sposób co w poprzednim przykładzie. Tym razem jednak tekst ten nie obraca się, lecz oddala się i zbliża do obserwatora wydając przy tym dźwięk „kliknięcia”. Zmieniając położenie źródła dźwięku wyposażonego w bufor IDirectSound3DBuffer wraz z położeniem tekstu uzyskuje się efekt malejącej głośności dźwięku, gdy tekst oddala się od obserwatora, a rosnącej, gdy się zbliża i stłumionej po jego minięciu. Odbiorcę dźwięku należy umieścić w tym samym punkcie, co obserwatora. Wyświetlany tekst zmienia się w zależności od tego, czy zbliża się, czy oddala od obserwatora. Działanie programu ilustruje rysunek 17.7. Poniżej zamieszczony został pełen kod źródłowy programu opatrzony komentarzami ułatwiającymi jego zrozumienie.

```
#define WIN32_LEAN_AND_MEAN // "odchudza" aplikację Windows
#define INITGUID               // używa identyfikatorów GUID w DirectMusic

////// Pliki nagłówkowe
#include <windows.h>           // standardowy plik nagłówkowy Windows
#include <dmusicc.h>             // pliki nagłówkowe DirectMusic
#include <dmusici.h>
#include <d3d8types.h>            // zawiera definicję typu D3DVECTOR
#include <cguid.h>                // zawiera definicję GUID_NULL
#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
```

**Rysunek 17.7.**

*Przykład  
zastosowania efektu  
przestrzennego*



```
////// 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;
unsigned int listBase; // początkowa lista wyświetlanego
GLYPHMETRICSFLOAT gmf[256]; // dane o orientacji i położeniu znaków
                             // używane przez listy wyświetlanego

////// Zmienne DirectMusic
IDirectMusicLoader8 *dmusicLoader = NULL; // obiekt ładowający
IDirectMusicPerformance8 *dmusicPerformance = NULL; // obiekt wykonania
IDirectMusicSegment8 *dmusicSegment = NULL; // segment
IDirectMusicAudioPath *dmusic3DAudioPath = NULL; // ścieżka dźwięku
IDirectSound3DBuffer *ds3DBuffer = NULL; // bufor efektu przestrzennego
IDirectSound3DListener *ds3DListener = NULL; // odbiorca dźwięku

DS3DBUFFER dsBufferParams; // parametry bufora efektu przestrzennego
DS3DLISTENER dsListenerParams; // parametry odbiorcy dźwięku

/*********************************************
*   Interfejsy DirectMusic
********************************************/

// InitDirectXAudio()
// opis: inicjuje komponent DirectX Audio
bool InitDirectXAudio(HWND hwnd)
{
    char pathStr[MAX_PATH];// ścieżka dostępu plików dźwiękowych
    WCHAR wcharStr[MAX_PATH];

    // tworzy obiekt ładowający
    if (FAILED(CoCreateInstance(CLSID_DirectMusicLoader, NULL, CLSCTX_INPROC,
        IID_IDirectMusicLoader8, (void**)&dmusicLoader)))

```



```
{  
    MessageBox(hwnd, "Nie można pobrać obiektu odbiorcy!",  
              "Błąd", MB_OK);  
    return false;  
}  
  
// pobiera parametry odbiorcy  
dsListenerParams.dwSize = sizeof(DS3DLISTENER);  
ds3DListener->GetAllParameters(&dsListenerParams);  
  
// określa położenie odbiorcy  
dsListenerParams.vPosition.x = 0.0f;  
dsListenerParams.vPosition.y = 0.0f;  
dsListenerParams.vPosition.z = 0.0f;  
ds3DListener->SetAllParameters(&dsListenerParams, DS3D_IMMEDIATE);  
  
// pobiera bieżący katalog  
GetCurrentDirectory(MAX_PATH, pathStr);  
  
// zamienia jego nazwę na Unicode  
MultiByteToWideChar(CP_ACP, 0, pathStr, -1, wcharStr, MAX_PATH);  
  
// określa ścieżkę dostępu do plików dźwiękowych  
dmusicLoader->SetSearchDirectory(GUID_DirectMusicAllTypes, wcharStr, FALSE);  
  
return true;  
}  
  
// LoadSegment()  
// opis: ładuje segment z pliku  
bool LoadSegment(HWND hwnd, char *filename)  
{  
    WCHAR wcharStr[MAX_PATH];  
  
    // zamienia nazwę pliku na Unicode  
    MultiByteToWideChar(CP_ACP, 0, filename, -1, wcharStr, MAX_PATH);  
  
    // ładuje segment z pliku  
    if (FAILED(dmusicLoader->LoadObjectFromFile(CLSID_DirectMusicSegment,  
                                                IID_IDirectMusicSegment8,  
                                                wcharStr,  
                                                (void**)&dmusicSegment)))  
    {  
        MessageBox(hwnd, "Nie znaleziono pliku audio!",  
                  "Błąd!", MB_OK);  
  
        return false;  
    }  
  
    // konfiguruje nieskończoną liczbę powtórzeń odtwarzanego segmentu  
    dmusicSegment->SetRepeats(DMUS_SEG_REPEAT_INFINITE);  
  
    // ładuje instrumenty do ścieżki  
    dmusicSegment->Download(dmusic3DAudioPath);  
  
    return true;  
}
```

```
// PlaySegment()
// opis: odtwarza segment
void PlaySegment(IDirectMusicPerformance8* dmPerf, IDirectMusicSegment8* dmSeg)
{
    // odtwarza segment od następnego taktu
    dmPerf->PlaySegmentEx(dmSeg, NULL, NULL, DMUS_SEGF_DEFAULT, 0,
                           NULL, NULL, dmusic3DAudioPath);
}

// StopSegment()
// opis: zatrzymuje odtwarzanie segmentu
void StopSegment(IDirectMusicPerformance8* dmPerf, IDirectMusicSegment8* dmSeg)
{
    // zatrzymuje odtwarzanie dmSeg
    dmPerf->StopEx(dmSeg, 0, 0);
}

// CloseDown()
// opis: kończy wykonanie
void CloseDown(IDirectMusicPerformance8* dmPerf)
{
    // zatrzymuje odtwarzanie
    dmPerf->Stop(NULL, NULL, 0, 0);

    // kończy pracę DirectMusic
    dmPerf->CloseDown();
}

// Set3DSoundParams()
// opis: konfiguruje parametry bufora efektu przestrzennego
void Set3DSoundParams(float doppler, float rolloff, float minDist, float maxDist)
{
    // parametry efektu Dopplera i tłumienia
    dsListenerParams.flDopplerFactor = doppler;
    dsListenerParams.flRolloffFactor = rolloff;

    if (ds3DListener)
        ds3DListener->SetAllParameters(&dsListenerParams, DS3D_IMMEDIATE);

    // odległość minimalna i maksymalna
    dsBufferParams.flMinDistance = minDist;
    dsBufferParams.flMaxDistance = maxDist;

    if (ds3DBuffer)
        ds3DBuffer->SetAllParameters(&dsBufferParams, DS3D_IMMEDIATE);
}

// Set3DSoundPos()
// opis: aktualizuje położenie źródła dźwięku (akceptuje współrzędne OpenGL)
void Set3DSoundPos(IDirectSound3DBuffer* dsBuff, float x, float y, float z)
{
    // używa -z ponieważ oś z a przeciwny zwrot w DirectX w stosunku do OpenGL
    if (dsBuff != NULL)
    {
        dsBuff->SetPosition(x, y, -z, DS3D_IMMEDIATE);
    }
}
```



```
glTranslatef(-length/2, 0.0f, 0.0f);           // centruje tekst

// rysuje tekst
glPushAttrib(GL_LIST_BIT);
    glListBase(base);
    glCallLists(strlen(str), GL_UNSIGNED_BYTE, str);
glPopAttrib();
}

// CleanUp()
// opis: zwalnia zasoby aplikacji
void CleanUp()
{
    ClearFont(listBase);

    dmusic3DAudioPath->Release();
    dmusicLoader->Release();
    dmusicPerformance->Release();
    dmusicSegment->Release();
}

// Initialize
// opis: inicjuje OpenGL
void Initialize()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // tło w kolorze czarnym

    glShadeModel(GL_SMOOTH);           // cieniowanie gładkie
    glEnable(GL_DEPTH_TEST);          // usuwanie przesłoniętych powierzchni
    glEnable(GL_LIGHT0);              // włącza źródło światła light0
    glEnable(GL_LIGHTING);            // włącza oświetlenie
    glEnable(GL_COLOR_MATERIAL);      // kolor jako materiał

    listBase = CreateOutlineFont("Arial", 10, 0.25f); // ładuje czcionkę Arial 10 pkt.
}

// Render
// opis: rysuje scenę
void Render()
{
    static float zpos = 0.0f;        // położenie na osi z
    static bool zDir = false;         // kierunek ruchu false = ujemny, true = dodatni

    // na przemian odsuwa i zbliża tekst
    if (zDir)
        zpos += 0.08f;
    else
        zpos -= 0.08f;

    if (zpos > 30.0f)
        zDir = false;
    if (zpos < -30.0f)
        zDir = true;

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



```
0,                                // bez buforów pomocniczych
PFD_MAIN_PLANE,                  // główna płaszczyzna rysowania
0,                                // zarezerwowane
0, 0, 0 }:                         // ignoruje maski warstw

// wybiera najbardziej zgodny format pikseli
nPixelFormat = ChoosePixelFormat(hDC, &pf);

// określa format pikseli dla danego kontekstu urządzenia
SetPixelFormat(hDC, nPixelFormat, &pf); }

// procedura okienkowa
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HGLRC hRC;                // kontekst tworzenia grafiki
    static HDC hDC;                  // kontekst urządzenia
    int width, height;              // szerokość i wysokość okna

    switch(message)
    {
        case WM_CREATE:           // okno jest tworzone

            hDC = GetDC(hwnd);      // pobiera kontekst urządzenia dla okna
            g_HDC = hDC;
            SetupPixelFormat(hDC); // wywołuje funkcję określającą format pikseli

            // tworzy kontekst tworzenia grafiki i czyni go bieżącym
            hRC = wglCreateContext(hDC);
            wglMakeCurrent(hDC, hRC);

            return 0;
            break;

        case WM_CLOSE:             // okno jest zamykane

            // deaktywuje bieżący kontekst tworzenia grafiki i usuwa go
            wglMakeCurrent(hDC, NULL);
            wglDeleteContext(hRC);

            // wstawia komunikat WM_QUIT do kolejki
            PostQuitMessage(0);

            return 0;
            break;

        case WM_SIZE:
            height = HIWORD(lParam); // pobiera nowe rozmiary okna
            width = LOWORD(lParam);

            if (height==0)          // unika dzielenia przez 0
            {
                height=1;
            }

            glViewport(0, 0, width, height); // nadaje nowe wymiary oknu OpenGL
            glMatrixMode(GL_PROJECTION);   // wybiera macierz rzutowania
            glLoadIdentity();             // resetuje macierz rzutowania
    }
}
```

```
// wyznacza proporcje obrazu
gluPerspective(54.0f,(GLfloat)width/(GLfloat)height,1.0f,1000.0f);

glMatrixMode(GL_MODELVIEW);           // wybiera macierz modelowania
glLoadIdentity();                   // resetuje macierz modelowania

return 0;
break;

case WM_KEYDOWN:                  // użytkownik nacisnął klawisz?
keyPressed[wParam] = true;
return 0;
break;

case WM_KEYUP:
keyPressed[wParam] = false;
return 0;
break;

default:
break;
}

return (DefWindowProc(hwnd, message, wParam, lParam));
}

// punkt, w którym rozpoczyna się wykonywanie aplikacji
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpcCmdLine, int nShowCmd)
{
    WNDCLASSEX windowClass;          // klasa okna
    HWND hwnd;                      // uchwyt okna
    MSG msg;                        // komunikat
    bool done;                      // znacznik zakończenia aplikacji
    DWORD dwExStyle;                // rozszerzony styl okna
    DWORD dwStyle;                  // styl okna
    RECT windowRect;                // zmienne pomocnicze

    int width = 800;
    int height = 600;
    int bits = 16;

    //fullScreen = TRUE;

    windowRect.left=(long)0;          // struktura określająca rozmiary okna
    windowRect.right=(long)width;
    windowRect.top=(long)0;
    windowRect.bottom=(long)height;

    // definicja klasy okna
    windowClass.cbSize              = sizeof(WNDCLASSEX);
    windowClass.style               = CS_HREDRAW | CS_VREDRAW;
    windowClass.lpfnWndProc         = WndProc;
    windowClass.cbClsExtra          = 0;
    windowClass.cbWndExtra          = 0;
    windowClass.hInstance            = hInstance;
    windowClass.hIcon               = LoadIcon(NULL, IDI_APPLICATION); // domyślna ikona
```



```
// sprawdza, czy utworzenie okna nie powiodło się (wtedy wartość hwnd równa NULL)
if (!hwnd)
    return 0;

// inicjuje COM
if (FAILED(CoInitialize(NULL)))
return 0; // jeśli inicjacja COM nie powiodła się,
// program kończy działanie

// inicjuje DirectX Audio
if (!InitDirectXAudio(hwnd))
    return 0;

// ładuje segment
if (!LoadSegment(hwnd, "start.wav"))
    return 0;

// konfiguruje parametry efektu przestrzennego
Set3DSoundParams(0.0, 0.1f, 1.0f, 100.0f);

// odtwarza segment
PlaySegment(dmusicPerformance, dmusicSegment);

ShowWindow(hwnd, SW_SHOW);           // wyświetla okno
UpdateWindow(hwnd);                 // aktualizuje okno

done = false;                         // inicjuje zmienną warunku pętli
Initialize();                         // inicjuje OpenGL

// pętla przetwarzania komunikatów
while (!done)
{
    PeekMessage(&msg, hwnd, NULL, NULL, PM_REMOVE);

    if (msg.message == WM_QUIT)      // aplikacja otrzymała komunikat WM_QUIT?
    {
        done = true;                // jeśli tak, to kończy działanie
    }
    else
    {
        // kończy działanie programu
        if (keyPressed[VK_ESCAPE])
            done = true;
        else
        {
            Render();                  // rysuje grafikę

            TranslateMessage(&msg);   // tłumaczy komunikat i wysyła do systemu
            DispatchMessage(&msg);
        }
    }
}

// kończy wykonanie
CloseDown(dmusicPerformance);

// zwalnia zasoby
CleanUp();
```

```
// kończy pracę COM  
CoUninitialize();  
  
if (fullScreen)  
{  
    ChangeDisplaySettings(NULL,0); // przywraca pulpit  
    ShowCursor(TRUE); // i wskaźnik myszy  
}  
  
return msg.wParam;  
}
```

## Podsumowanie

Na tym można zakończyć omówienie możliwości DirectX Audio. W rozdziale tym nie zostały przedstawione niektóre możliwości DirectX Audio, takie jak dynamiczne komponowanie muzyki za pomocą DirectMusic oraz zastosowania efektów pogłosu, echa, chóru i zniekształceń dźwięku. Pominięte zostało także zagadnienie odtwarzania dźwięku w formacie *MP3*. Kompletny przykład implementacji odtwarzacza MP3 zawiera dokumentacja DirectShow.

Komputery mogą przechowywać i odtwarzać dźwięk zapisany cyfrowo oraz uzyskiwany na drodze syntezy. Źródłem dźwięku zapisanego w postaci cyfrowej jest zwykły mikrofon. Dźwięk zapisany cyfrowo używany jest w grach przede wszystkim do uzyskania efektów dźwiękowych, takich jak eksplozie, odgłosy ruchu czy mowa. Natomiast dźwięk tworzony przez syntezator stanowi najczęściej podkład muzyczny gry.

Komponent DirectX Audio umożliwia tworzenie dynamicznych ścieżek dźwiękowych wykorzystujących w pełni możliwości sprzętowe, w tym także efekt przestrzenny.

Odtwarzanie dźwięku za pomocą DirectMusic wymaga zainicjowania modelu COM, utworzenia i zainicjowania obiektu wykonania, utworzenia obiektu ładowającego, załadowania segmentu oraz instrumentów i odtworzenia segmentu.

Ścieżki dźwięku zarządzają przepływem danych dźwięku poprzez różne obiekty na drodze od pliku do kanałów wyjściowych. Ścieżka dźwięku może zawierać obiekt wykonania, segment, grafy narzędzi, syntezator i bufore DirectSound.

Efekt przestrzenny można łatwo uzyskać pobierając bufor DirectSound ścieżki dźwięku. Zastosowanie efektu przestrzennego jest jednym z ważniejszych sposobów podnoszenia realizmu gry.

## Rozdział 18.

# Modele trójwymiarowe

Praktycznie każda współczesna gra przechowuje opis modeli trójwymiarowych postaci i innych obiektów w plikach o określonym formacie. Większość z tych gier umożliwia także użytkownikowi modyfikację wyglądu istniejących i tworzenie nowych postaci. W rozdziale tym przedstawione zostaną popularne formaty plików modeli trójwymiarowych *MD2*, które zostały zastosowane w grze „Quake 2”. Przedstawiony będzie także sposób ładowania takich modeli, wyświetlania ich i animowania we własnych grach.

W rozdziale tym omówione zostaną następujące zagadnienia:

- ◆ formaty plików modeli trójwymiarowych;
- ◆ format *MD2*;
- ◆ ładowanie teksturowe w formacie *PCX*.

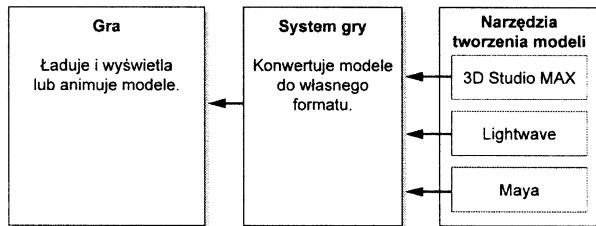
## Formaty plików modeli trójwymiarowych

Umieszczanie definicji modeli trójwymiarowych bezpośrednio w kodzie źródłowym programu nie jest dobrym rozwiązaniem. Ręczne wpisywanie danych modelu jest pracochłonne i łatwo przy tym o pomyłkę. Modyfikowanie modelu postaci lub dodanie nowej postaci wymaga za każdym razem ingerencji w kod źródłowy programu. Dlatego też lepiej jest tworzyć modele trójwymiarowe za pomocą wyspecjalizowanych narzędzi (lub w najprostszym przypadku umieścić dane modelu w pliku tekstowym), następnie zapisać je w pliku o określonym formacie, który będzie odczytywać później grę.

Większość dostępnych na rynku gier korzysta z własnych, specjalizowanych formatów plików do przechowywania modeli obiektów. Proces ich powstawania ilustruje rysunek 18.1. Firmy produkujące gry zatrudniają własnych grafików, którzy tworzą modele postaci występujących w grach za pomocą takich narzędzi jak *3D Studio Max* firmy Discreet, *Lightwave* firmy NewTek, *trueSpace* firmy Caligari czy *Maya* firmy Alias/Wavefront. Modele te następnie zapisywane są w specjalizowanym formacie używanym przez grę.

**Rysunek 18.1.**

**Obiekty gry**  
 ładowane są  
 z plików modeli  
 przygotowanych  
 za pomocą  
 odpowiednich  
 narzędzi



Książka ta nie zajmuje się problematyką tworzenia modeli. Bardziej interesujące jest ładowanie, wyświetlanie i animacja gotowych modeli zapisanych w plikach. Formaty tych plików mogą być bardzo proste i sprowadzać się do opisu wierzchołków modelu w pliku tekstowym, a także bardzo skomplikowane i przyjmować postać plików binarnych zawierających informacje o wierzchołkach, teksturach i innych właściwościach modelu. Czas więc zapoznać się z formatem *MD2* stosowanym w grze „Quake 2” firmy id Software. Format ten posiada duże możliwości i jednocześnie jego użycie nie jest skomplikowane. Można go polecić jako wzór każdemu programiście, który zamierza opracować własny format modeli.

## Format MD2

Strukturę plików formatu *MD2* tworzą dwie części: nagłówek i właściwe dane. Nagłówek zawiera podstawowe informacje o rozmiarach modelu, takie jak na przykład liczba wierzchołków i trójkątów tworzących model oraz położenie opisujących je danych w pliku. Poniżej przedstawiona została struktura nagłówka:

```

typedef struct
{
    int ident;          // "IDP2" oznacza format MD2
    int version;        // obecnie wersja 8
    int skinwidth;      // szerokość tekstury
    int skinheight;     // wysokość tekstury
    int framesize;      // rozmiar klatki (w bajtach)
    int numSkins;       // liczba tekstur
    int numXYZ;         // liczba punktów
    int numST;          // liczba współrzędnych tekstury
    int numTris;         // liczba trójkątów
    int numGLcmds;       // liczba komend OpenGL
    int numFrames;       // całkowita liczba klatek
    int offsetSkins;     // położenie (w pliku) nazw tekstur (każda po 64 bajty)
    int offsetST;        // położenie (w pliku) współrzędnych tekstury
    int offsetTris;      // położenie (w pliku) siatki trójkątów
    int offsetFrames;    // położenie (w pliku) danych klatki (punktów)
    int offsetGLcmds;    // typ używanych poleceń OpenGL
    int offsetEnd;        // koniec pliku
} modelHeader_t;
  
```

Teraz należy przyjrzeć się bliżej polom tej struktury. Pierwsze z nich jest identyfikatorem formatu pliku. Ładując model z pliku należy więc sprawdzić, czy pole *ident* posiada wartość IDP2. Jeśli tak, to można kontynuować ładowanie modelu. W przeciwnym razie należy utworzyć komunikat o błędzie i poinformować użytkownika, że dane modelu znajdują się w pliku nieznanego formatu.

Pola *skinwidth* i *skinheight* określają szerokość i wysokość tekstur, którymi będzie pokrywany model. Pozwalają także wyznaczyć rozmiar tekstury w bajtach.

Pole *framesize* informuje o rozmiarze klatki wyrażonym w bajtach. *Klatki* używane przez format *MD2* reprezentują model w określonym momencie ruchu i stosowane są w celu uzyskania płynnego efektu animacji postaci. Animacje modeli formatu *MD2* omówione zostaną wkrótce dokładniej, a na razie wystarczy zapamiętać, że pole *framesize* pozwala ustalić wielkość pamięci potrzebnej do przechowania danych klatki.

Pole *numSkins* określa liczbę tekstur dostępnych dla danego modelu. Zwykle modele postaci w grze „Quake 2” przygotowywane są z zestawem tekstur, z których wybiera użytkownik. Pole to informuje o tym, ile tekstur zostanie wymienionych w pliku począwszy od pozycji *offsetSkins*. Chociaż sam pomysł wart jest uwagi, to jednak lądując modele *MD2* nie należy korzystać z informacji zawartej w polu *numSkins*, ponieważ nazwy tekstu zawierają ścieżki plików przydatne jedynie po zainstalowaniu gry „Quake 2”.

Pole *numXYZ* specyfikuje całkowitą liczbę wierzchołków modelu, która stanowi sumę wierzchołków dla poszczególnych klatek. Gdy na przykład definicja modelu zawierać będzie 100 klatek po 100 wierzchołków każda, to pole *numXYZ* będzie miało wartość 10 000.

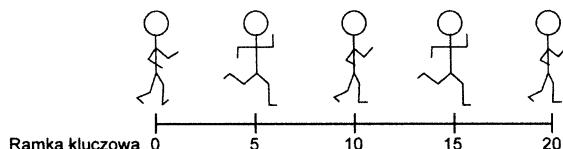
Kolejne pole nagłówka, *numST*, zawiera liczbę współrzędnych tekstury, co pozwala określić wielkość pamięci potrzebnej do przechowania współrzędnych tekstury.

Pole *numTris* specyfikuje liczbę trójkątów, z których składa się model. Im większa liczba trójkątów, tym bardziej szczegółowy model, ale i mniejsza efektywność tworzenia grafiki. Wartość ta musi więc być wynikiem kompromisu pomiędzy szczegółowością grafiki, a minimalną wydajnością systemu.

Pole *numGLcmds* określa liczbę komend OpenGL przechowywanych w pliku począwszy od pozycji *offsetGLcmds*. Przez komendy należy rozumieć tutaj liczby całkowite określające sposób rysowania modelu. Na przykład dane pierwszych dziewięciu wierzchołków modelu mogą zostać wykorzystane do rysowania łańcucha trójkątów, jeśli komendą OpenGL będzie `GL_TRIANGLE_STRIP`. Kolejna z wartości umieszczonych w buforze *offsetGLcmds* może reprezentować komendę `GL_TRIANGLE_FAN`, wobec czego kolejnych 20 wierzchołków będzie tworzyć wieniec trójkątów. Zastosowanie takiego sposobu tworzenia grafiki nie jest obligatoryjne, ale w pewnych przypadkach bardzo przydatne.

Ostatnim z pól nagłówka określających rozmiary danych jest pole *numFrames*. Specyfikuje ono liczbę klatek animacji, którymi dysponuje model. Są to tak zwane kluczowe ramki animacji, które określają wygląd modelu w określonych momentach animacji, co ilustruje rysunek 18.2.

**Rysunek 18.2.**  
Kluczowe  
klatki animacji



Zastosowanie kluczowych klatek animacji pozwala istotnie zmniejszyć rozmiary pliku modelu, ponieważ nie musi on wtedy zawierać wszystkich klatek animacji. Aby uzyskać efekt płynnej animacji, trzeba jednak samodzielnie obliczyć kolejne pozycje modelu pomiędzy kluczowymi klatkami animacji. Sposób wyznaczania tych pozycji omówimy wkrótce.

Ostatni blok pół nagłówka zawiera pola określające położenie w pliku różnych danych opisujących model. Pierwsze z nich, *offsetSkins*, wskazuje fragment pliku zawierający nazwy tekstur. Każda z nich zajmuje 64 bajty. Jeśli więc w pliku umieszczonych jest pięć tekstur, to pole *offsetSkins* wskazuje fragment pliku o rozmiarze  $64 * 5$  bajtów, czyli 320 bajtów.

Pole *offsetST* określa położenie danych opisujących współrzędne tekstury. Ładując dane modelu trzeba odczytać *numST* współrzędnych tekstury.

Pole *offsetTris* wskazuje dane siatki trójkątów tworzących model, a pole *offsetFrames* położenie danych wierzchołków dla wszystkich klatek animacji.

Ostatnie dwa pola, *offsetGLcmds* i *offsetEnd*, wskazują (odpowiednio) położenie komend OpenGL i końca pliku modelu.

## Implementacja formatu MD2

Znając strukturę pliku w formacie *MD2* można przystąpić do jego implementacji. Na początku trzeba utworzyć reprezentację podstawowego elementu opisu każdego modelu: wektora. W rzeczywistości przy tworzeniu gry dysponuje się już na tym etapie odpowiednią reprezentacją wektora, ale tym razem przedmiotem zainteresowania jest stworzenie jedynie przykładu ilustrującego posługiwanie się formatem *MD2*. Poniższy fragment kodu definiuje typ *vector\_t* wraz z kilkoma niezbędnymi operacjami na wektorach:

```
// pojedynczy wektor
typedef struct
{
    float point[3];
} vector_t;

// odejmowanie wektorów
vector_t operator-(vector_t a, vector_t b)
{
    vector_t c;

    c.point[0] = a.point[0] - b.point[0];
    c.point[1] = a.point[1] - b.point[1];
    c.point[2] = a.point[2] - b.point[2];

    return c;
}

// mnożenie wektorów
vector_t operator*(float f, vector_t b)
{
    vector_t c;
```

```
c.point[0] = f * b.point[0];
c.point[1] = f * b.point[1];
c.point[2] = f * b.point[2];

return c;
}

// dzielenie wektorów
vector_t operator/(vector_t a, vector_t b)
{
    vector_t c;

    c.point[0] = a.point[0] / b.point[0];
    c.point[1] = a.point[1] / b.point[1];
    c.point[2] = a.point[2] / b.point[2];

    return c;
}

// dodawanie wektorów
vector_t operator+(vector_t a, vector_t b)
{
    vector_t c;

    c.point[0] = a.point[0] + b.point[0];
    c.point[1] = a.point[1] + b.point[1];
    c.point[2] = a.point[2] + b.point[2];

    return c;
}
```

Następnie należy zdefiniować strukturę do przechowywania pojedynczej współrzędnej tekstury oraz strukturę, która przechowuje będzie wartość indeksu współrzędnej tekstury w tablicy współrzędnych teksturow:

```
// współrzędna tekstury
typedef struct .
{
    float s;           // współrzędna s
    float t;           // współrzędna t
} texCoord_t;

// indeks współrzędnej tekstury
typedef struct
{
    short s;
    short t;
} stIndex_t;
```

Kolejna z definiowanych struktur umożliwiać będzie przechowanie informacji o pojedynczym punkcie klatki animacji. Będzie ona zawierać wartość indeksu tablicy wektorów normalnych oświetlenia. Indeks ten nie będzie na razie wykorzystywany, ale musi być uwzględniony, aby poprawnie załadować model zapisany w formacie *MD2*.

```
// pojedynczy punkt klatki animacji
typedef struct
{
    unsigned char v[3];           // informacja o punkcie
    unsigned char normalIndex;   // pole to nie będzie używane
} framePoint_t;
```

Struktura `framePoint_t` będzie używana jako elementu struktury przechowującej informacje o pojedynczej klatce animacji:

```
// informacja o pojedynczej klatce animacji
typedef struct
{
    float scale[3];              // skalowanie wierzchołków klatki
    float translate[3];          // przesunięcie wierzchołków klatki
    char name[16];               // nazwa modelu
    framePoint_t fp[1];          // początek listy wierzchołków klatki
} frame_t;
```

Ladując model z pliku w formacie *MD2* trzeba wyznaczyć położenie danych opisujących wierzchołki klatki korzystając z informacji zawartej w polach *scale* i *translate* struktury `frame_t`. Pole *fp* określa natomiast położenie bufora zawierającego opis wszystkich wierzchołków.

Ponieważ siatka modelu w formacie *MD2* zbudowana jest z samych trójkątów, logicznym rozwiązaniem będzie utworzenie listy trójkątów modelu. Dysponując taką listą można następnie określić strukturę modelu za pomocą listy indeksów trójkątów. Poniższa struktura przechowujeć będzie indeksy trzech wierzchołków trójkąta (*meshIndex*) oraz indeksy trzech współrzędnych tekstury pokrywającej trójkąt (*stIndex*):

```
// dane opisujące pojedynczy trójkąt
typedef struct
{
    unsigned short meshIndex[3]; // indeksy wierzchołków
    unsigned short stIndex[3];  // indeksy współrzędnych tekstury
} mesh_t;
```

Wszystkie zdefiniowane dotąd struktury łączy się w jedną całość za pomocą struktury definiującej model:

```
// dane modelu
typedef struct
{
    int numFrames;                // liczba klatek
    int numPoints;                // liczba punktów
    int numTriangles;             // liczba trójkątów
    int numST;                    // liczba tekstur
    int frameSize;                // rozmiar klatki w bajtach
    int texWidth, texHeight;      // szerokość i wysokość tekstury
    int currentFrame;             // bieżąca klatka animacji
    int nextFrame;                // kolejna klatka animacji
    float interpol;               // współczynnik interpolacji
    mesh_t *triIndex;             // lista trójkątów
    texCoord_t *st;               // lista współrzędnych tekstury
    vector_t *pointList;          // lista wierzchołków
    texture_t *modelTex;          // dane tekstury
} modelData_t;
```

Struktura ta zawiera kilka pól odpowiadających bezpośrednio polom nagłówka pliku w formacie *MD2*, takim jak na przykład liczba klatek, wierzchołków, współrzędnych tekstury i trójkątów oraz rozmiar klatki w bajtach. Przechowuje także wartości wykorzystywane w procesie animacji, takie jak identyfikator bieżącej klatki kluczowej, identyfikator kolejnej klatki kluczowej oraz sposób interpolacji pomiędzy tymi klatkami. Ostatnie pola struktury *modelData\_t* są wskaźnikami list trójkątów, współrzędnych tekstury, wierzchołków oraz danych bieżącej tekstury modelu. Dane tekstury przechowywane są za pomocą struktury *texture\_t* zdefiniowanej jak poniżej:

```
enum texTypes_t
{
    PCX,
    BMP,
    TGA
};

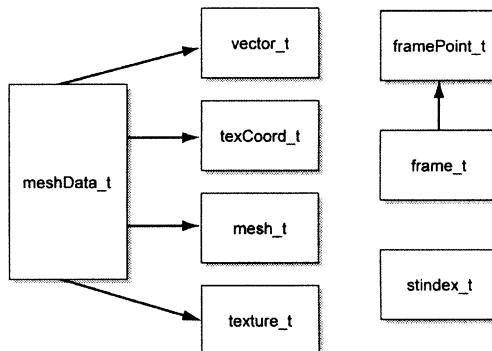
typedef struct
{
    texTypes_t textureType; // typ pliku tekstury

    int width;           // szerokość tekstury
    int height;          // wysokość tekstury
    long int scaledWidth;
    long int scaledHeight;

    unsigned int texID; // identyfikator obiektu tekstury
    unsigned char *data; // dane tekstury
    unsigned char *palette; // paleta tekstury (jeśli istnieje)
} texture_t;
```

Rysunek 18.3 ilustruje ogólną architekturę reprezentacji modelu *MD2*.

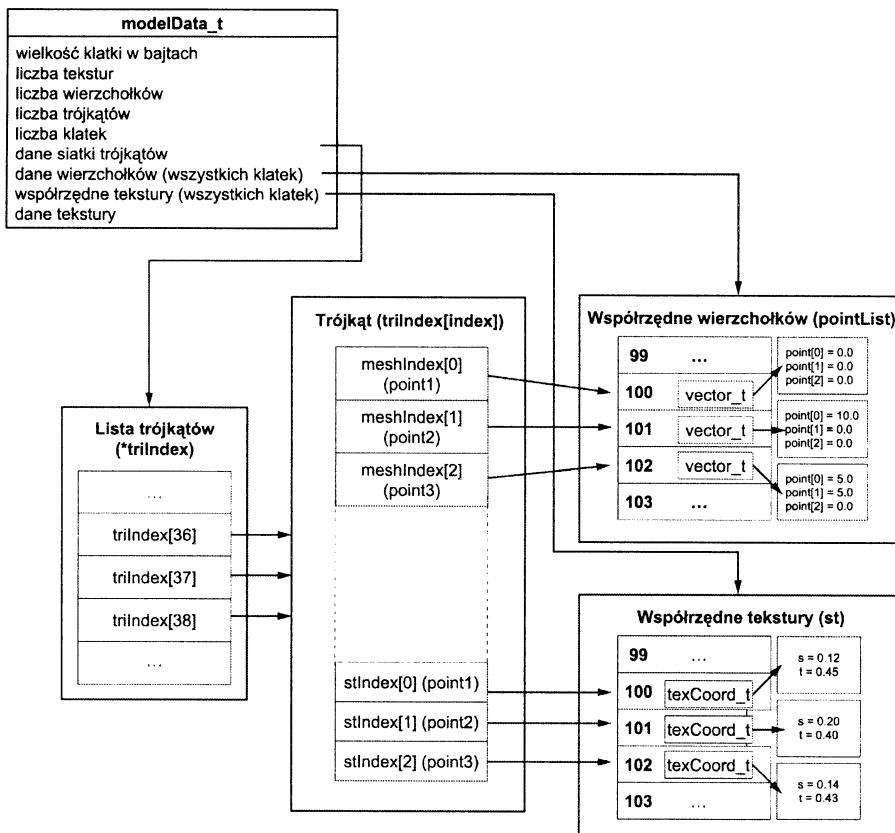
**Rysunek 18.3.**  
Struktury  
reprezentujące  
model *MD2*



Rysunek 18.4 pokazuje sposób korzystania ze struktury *modelData\_t* w celu uzyskania współrzędnych wierzchołków i współrzędnych tekstury.

Dysponując strukturą *modelData\_t* posiada się:

- ◆ listę trójkątów wskazywaną przez pole *triIndex*;
- ◆ listę współrzędnych wierzchołków wskazywaną przez pole *pointList*;
- ◆ listę współrzędnych tekstury wskazywaną przez pole *st*.



Rysunek 18.4. Sposób korzystania z reprezentacji modelu MD2

Przeglądając listę trójkątów można uzyskać dla każdego trójkąta wartości indeksów współrzędnych trzech wierzchołków i ich współrzędnych tekstuury. Indeksy te umożliwiają odszukanie współrzędnych wierzchołków na liście `pointList` i współrzędnych tekstuury na liście `st`. Informacje te umożliwiają narysowanie bieżącego trójkąta. Rysując w ten sposób wszystkie trójkąty z listy `triIndex` uzyskuje się pełen model.

Ponieważ znany już jest sposób korzystania ze struktur reprezentujących model, można zająć się teraz implementacją działających na nich funkcji.

## Ładowanie modelu MD2

Oczywiście pierwszą operacją, którą trzeba zaimplementować, aby móc korzystać z modelu MD2, jest jego załadowanie z pliku. Poniżej zaprezentowany został kod funkcji `LoadMD2Model()`, której przekazane zostały nazwy plików zawierających model i teksturę. Funkcja `LoadMD2Model()` utworzy struktury reprezentujące model:

```
modelData_t *LoadMD2Model(char *filename, char *textureName)
{
    FILE *filePtr; // wskaźnik pliku
    int fileLen; // długość pliku
    char *buffer; // bufor pliku
```

```

modelData_t *model;           // model
modelHeader_t *modelHeader;  // nagłówek modelu
texture_t *md2Texture;      // tekstura modelu

stIndex_t *stPtr;            // dane tekstury
frame_t *frame;              // dane klatek
vector_t *pointListPtr;      // zmienna indeksu
mesh_t *triIndex, *bufIndexPtr; // zmienne indeksu
int i, j;                   // zmienne indeksu

```

Większość z pokazanych wyżej zmiennych lokalnych służy do przechowania danych, zanim zostaną umieszczone w strukturze model zwracanej przez funkcję.

Następnie funkcja otwiera plik i ustala jego długość, co pozwala przydzielić wystarczający obszar pamięci do załadowania całej zawartości pliku:

```

// otwiera plik modelu
filePtr = fopen(filename, "rb");
if (filePtr == NULL)
    return NULL;

// ustala długość pliku
fseek(filePtr, 0, SEEK_END);
fileLen = ftell(filePtr);
fseek(filePtr, 0, SEEK_SET);

// wczytuje cały plik do bufora
buffer = (char*)malloc(fileLen + 1);
fread(buffer, sizeof(char), fileLen, filePtr);

```

Wyodrębnienie w buforze fragmentu reprezentującego nagłówek pliku pozwala określić atrybuty modelu. Funkcja przydziela także pamięć na dane modelu:

```

// wyodrębnia nagłówek
modelHeader = (modelHeader_t*)buffer;

// przydziela pamięć na dane modelu
model = (modelData_t*)malloc(sizeof(modelData_t));
if (model == NULL)
    return NULL;

```

Korzystając z informacji zawartych w nagłówku funkcja wyznacza wielkość obszaru pamięci potrzebnego do przechowania informacji o wierzchołkach modelu. W tym celu mnoży liczbę wierzchołków modelu o liczbę klatek animacji. Pole pointList będzie wskazywać obszar pamięci, w którym zostanie umieszczona informacja o wszystkich wierzchołkach modelu.

Po przydzieleniu pamięci na listę wierzchołków zapamiętywana jest liczba wierzchołków, klatek i wielkość klatki:

```

// przydziela pamięć dla wszystkich wierzchołków wszystkich klatek animacji
model->pointList = (vector_t*)malloc(sizeof(vector_t)*modelHeader->numXYZ *
                                         modelHeader->numFrames);

// przechowuje podstawowe dane o modelu
model->numPoints = modelHeader->numXYZ; // liczba wierzchołków
model->numFrames = modelHeader->numFrames; // liczba klatek
model->frameSize = modelHeader->framesize; // wielkość klatki

```

Następnie funkcja LoadMD2Model() ładuje wierzchołki modelu do pamięci przeglądając w pętli kolejne klatki animacji, wyznaczając położenie danych wierzchołków i określając położenie każdego wierzchołka korzystając przy tym z wartości współczynników skalowania i przesunięcia dla każdej klatki:

```
// przegląda klatki animacji
for(j = 0; j < modelHeader->numFrames; j++)
{
    // początek opisu punktów danej klatki
    frame = (frame_t*)&buffer[modelHeader->offsetFrames +
        modelHeader->framesize * j];

    // wyznacza położenie opisu kolejnych punktów
    pointListPtr = (vector_t*)&model->pointList[modelHeader->numXYZ * j];
    for(i = 0; i < modelHeader->numXYZ; i++)
    {
        pointListPtr[i].point[0] = frame->scale[0] * frame->fp[i].v[0] +
            frame->translate[0];
        pointListPtr[i].point[1] = frame->scale[1] * frame->fp[i].v[1] +
            frame->translate[1];
        pointListPtr[i].point[2] = frame->scale[2] * frame->fp[i].v[2] +
            frame->translate[2];
    }
}
```

Po załadowaniu współrzędnych wierzchołków wszystkich klatek funkcja ładuje tekstury modelu do bufora md2Texture. Po sprawdzeniu, czy plik zawierający teksturę został odnaleziony, a jego zawartość poprawnie załadowana, funkcja umieszcza w polu modelTex adres bufora md2Texture.

```
// ładuje teksturę modelu
md2Texture = LoadTexture(textureName);
if (md2Texture != NULL)
{
    // konfiguruje teksturę OpenGL
    SetupMD2Texture(md2Texture);
    model->modelTex = md2Texture;
}
else
    return NULL;
```

Następnie przydziela pamięć niezbędną do przechowania współrzędnych tekstury i zapamiętuje ich liczbę:

```
// przydziela pamięć na współrzędne tekstury
model->st = (texCoord_t*)malloc(sizeof(texCoord_t)*modelHeader->numST);

// zapamiętuje liczbę współrzędnych tekstury
model->numST = modelHeader->numST;
```

Współrzędne tekstury pobierane są w pętli i umieszczane w strukturach modelu:

```
// wskaźnik bufora współrzędnych tekstury
stPtr = (stIndex_t*)&buffer[modelHeader->offsetST];

// zapamiętuje współrzędne tekstury
for (i = 0; i < modelHeader->numST; i++)
```

```
{
    model->st[i].s = (float)stPtr[i].s / (float)md2Texture->width;
    model->st[i].t = (float)stPtr[i].t / (float)md2Texture->height;
}
```

Kolejną operacją jest przydzielenie obszaru pamięci na dane trójkątów, zapamiętanie ich całkowitej liczby i zainicjowanie tymczasowego wskaźnika bufora trójkątów:

```
// przydziela pamięć liście trójkątów
triIndex = (mesh_t*)malloc(sizeof(mesh_t) * modelHeader->numTris);

// zapamiętuje liczbę trójkątów
model->numTriangles = modelHeader->numTris;
model->triIndex = triIndex;

// tymczasowy wskaźnik bufora trójkątów
bufIndexPtr = (mesh_t*)&buffer[modelHeader->offsetTris];
```

Po przydzieleniu pamięci funkcja ładuje do niej dane trójkątów z bufora pliku przeglądając w pętli trójkąty wszystkich klatek animacji i umieszczając indeksy ich wierzchołków i indeksy współrzędnych tekstury w strukturach modelu:

```
// wypełnia listę trójkątów
for (j = 0; j < model->numFrames; j++)
{
    // dla wszystkich trójkątów danej klatki
    for(i = 0; i < modelHeader->numTris; i++)
    {
        triIndex[i].meshIndex[0] = bufIndexPtr[i].meshIndex[0];
        triIndex[i].meshIndex[1] = bufIndexPtr[i].meshIndex[1];
        triIndex[i].meshIndex[2] = bufIndexPtr[i].meshIndex[2];
        triIndex[i].stIndex[0] = bufIndexPtr[i].stIndex[0];
        triIndex[i].stIndex[1] = bufIndexPtr[i].stIndex[1];
        triIndex[i].stIndex[2] = bufIndexPtr[i].stIndex[2];
    }
}
```

W ten sposób funkcja `LoadMD2Model()` załadowała wszystkie dane modelu. Może teraz zamknąć plik i zwolnić bufor, w którym umieściła jego zawartość, a także zainicjować zmienne modelu wykorzystywane podczas animacji:

```
// zamyka plik i zwalnia jego bufor
fclose(filePtr);
free(buffer);
// inicjuje zmienne animacji
model->currentFrame = 0;
model->nextFrame = 1;
model->interpol = 0.0;

return model;
}
```

W jaki sposób można używać tej funkcji w opracowywanych właśnie programach? Trzeba najpierw zadeklarować wskaźnik modelu:

```
modelData_t *myModel;
```

Zakładając, że model znajduje się w pliku o nazwie *mymodel.md2*, jego tekstura w pliku *mymodel.pcx*, a oba pliki znajdują się w katalogu bieżącym, wystarczy w celu załadowania modelu wywołać funkcję *LoadMD2Model()* w następujący sposób:

```
myModel = LoadMD2Model("mymodel.md2", "mymodel.pcx");
```

W ten sposób model zostaje załadowany i jest gotowy do użycia.

## Wyświetlanie modelu MD2

Po załadowaniu modelu można wykorzystać go w różny sposób. Na razie analizie będzie poddane jedynie jego wyświetlenie, a animacja zostanie pozostawiona na później.

Wyświetlenie modelu wymaga wybrania konkretnej klatki animacji, a następnie przejścia w pętli wszystkich jej trójkątów i wykorzystania zawartych w nich indeksów wierzchołków. Zadanie to wykonuje funkcja *DisplayMD2()*:

```
void DisplayMD2(modelData_t *model, int frameNum)
{
    vector_t *pointList; // lista wierzchołków danej klatki
    int i;               // zmieniąca indeksu

    // inicjuje wskaźnik listy wierzchołków
    pointList = &model->pointList[model->numPoints * frameNum];

    // wybiera teksturę
    glBindTexture(GL_TEXTURE_2D, model->modelTex->texID);

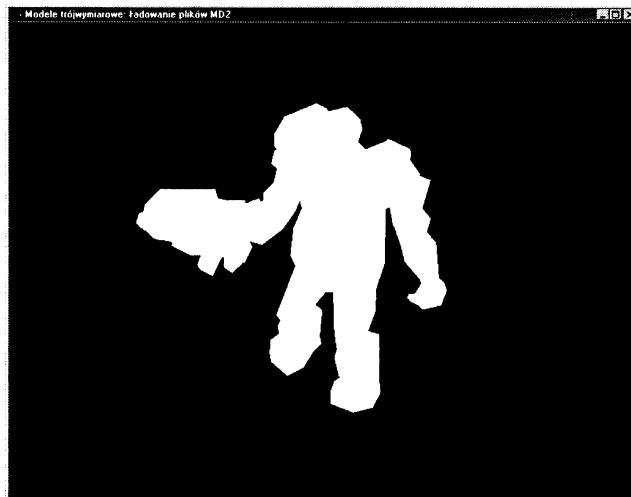
    // wyświetla model za pomocą trójkątów wypełnionych jednym kolorem
    glBegin(GL_TRIANGLES);
    for(i = 0; i < model->numTriangles; i++)
    {
        glVertex3fv(pointList[model->triIndex[i].meshIndex[0]].point);
        glVertex3fv(pointList[model->triIndex[i].meshIndex[2]].point);
        glVertex3fv(pointList[model->triIndex[i].meshIndex[1]].point);
    }
    glEnd();
}
```

Parametrami tej funkcji jest model i identyfikator tej jego klatki, która ma być narysowana. Korzystając z parametru *frameNum* funkcja wyznacza położenie wierzchołków danej klatki na liście wierzchołków. Powyższa wersja funkcji *DisplayMD2()* wyświetla model jedynie za pomocą trójkątów wypełnionych pojedynczym kolorem. W pętli przegląda kolejne trójkąty i na podstawie zawartych w nich indeksów wierzchołków pobiera odpowiednie wierzchołki i rysuje je. Poniższe wywołanie funkcji *DisplayMD2()* powoduje narysowanie pierwszej klatki modelu *myModel*:

```
DisplayMD2(myModel, 0);
```

Rysunek 18.5 ilustruje efekt jej wykonania dla modelu jednej z postaci gry Quake 2.

**Rysunek 18.5.**  
Efekt wywołania  
pierwszej wersji  
funkcji  
*DisplayMD2()*



Oczywiście rezultat pokazany na rysunku 18.5 nie jest zadawalający. Aby uzyskać bardziej realistyczny wygląd modelu, należy wzbogacić go o oświetlenie. W tym celu należy wyznaczyć wektor normalny do każdego z trójkątów modelu. Wykorzystać można do tego funkcję *CalculateNormal()* zdefiniowaną jak poniżej:

```
void CalculateNormal( float *p1, float *p2, float *p3 )
{
    float a[3], b[3], result[3];
    float length;

    a[0] = p1[0] - p2[0];
    a[1] = p1[1] - p2[1];
    a[2] = p1[2] - p2[2];

    b[0] = p1[0] - p3[0];
    b[1] = p1[1] - p3[1];
    b[2] = p1[2] - p3[2];

    result[0] = a[1] * b[2] - b[1] * a[2];
    result[1] = b[0] * a[2] - a[0] * b[2];
    result[2] = a[0] * b[1] - b[0] * a[1];

    // wyznacza długość wektora normalnego
    length = (float)sqrt(result[0]*result[0] + result[1]*result[1] +
        result[2]*result[2]);

    // i normalizuje go
    glNormal3f(result[0]/length, result[1]/length, result[2]/length);
}
```

Funkcję tę wykorzystać można także w kolejnej wersji funkcji *DisplayMD2()* do wyznaczenia wektora normalnego do trójkąta przed narysowaniem jego wierzchołków:

```
.....
glBegin(GL_TRIANGLES);
for(i = 0; i < model->numTriangles; i++)
```

```

    {
        CalculateNormal(pointList[model]->triIndex[i].meshIndex[0]).point,
        pointList[model]->triIndex[i].meshIndex[2].point,
        pointList[model]->triIndex[i].meshIndex[1].point);
        glVertex3fv(pointList[model]->triIndex[i].meshIndex[0].point);
        glVertex3fv(pointList[model]->triIndex[i].meshIndex[2].point);
        glVertex3fv(pointList[model]->triIndex[i].meshIndex[1].point);
    }
    glEnd();
    ...
}

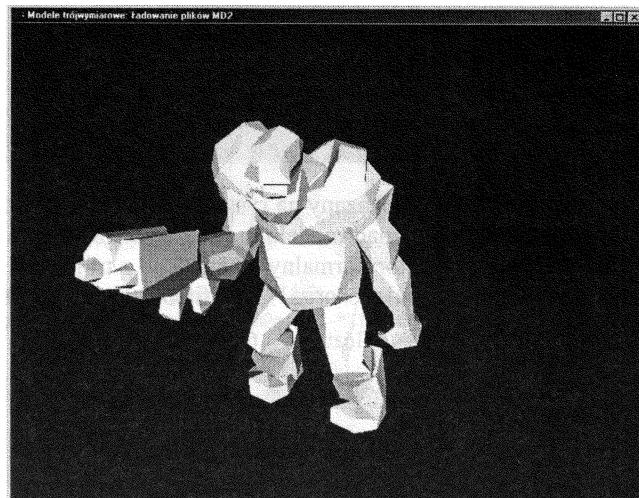
```

Teraz model prezentuje się o wiele lepiej, co pokazuje rysunek 18.6.

#### Rysunek 18.6.

*Model MD2*

*z oświetleniem  
i cieniowaniem*



Aby jeszcze podnieść jego realizm, w kolejnej wersji funkcji `DisplayMD2()` wzbogacony zostanie o teksturę.

## Pokrywanie modelu teksturową

Model *MD2* pokrywa się teksturową w taki sam sposób jak każdy inny obiekt. Najpierw ładuje się teksturę, konfiguruje ją w OpenGL i wybiera obiekt tekstuury przed okresem współrzędnych tekstuury dla każdego z rysowanych wierzchołków. Współrzędne tekstuury można pobrać wykorzystując w tym celu indeksy zawarte w trójkątach modelu (podobnie jak w przypadku współrzędnych wierzchołków).

Większość modeli dostarczanych wraz z grami posiada tekstyury umieszczone w plikach formatu *PCX*. Nie trzeba tutaj omawiać szczegółowo formatu plików *PCX*, ale można przedstawić fragmenty kodu służące ładowaniu ich zawartości jako tekstuur OpenGL. Przy tworzeniu własnych modeli można korzystać oczywiście z innych formatów plików zawierających tekstuury. Mogą to być na przykład pliki formatów *BMP* lub *TGA*, które zostały omówione szczegółowo we wcześniejszych rozdziałach.

Jeśli założy się, że tekstura została już załadowana z pliku i jej dane umieszczone są w strukturach reprezentujących model, to aby z niej skorzystać, trzeba zmodyfikować

funkcję `DisplayMD2()`. Należy wybrać obiekt tekstury i zdefiniować współrzędne tekstury dla rysowanych wierzchołków:

```
void DisplayMD2(modelData_t *model, int frameNum)
{
    vector_t *pointList; // lista wierzchołków danej klatki
    int i; // zmieniąca indeksu

    // inicjuje wskaźnik listy wierzchołków
    pointList = &model->pointList[model->numPoints * frameNum];

    // wybiera teksturę
    glBindTexture(GL_TEXTURE_2D, model->modelTex->texID);

    // wyświetla model pokryty teksturą i oświetlony
    glBegin(GL_TRIANGLES);
    for(i = 0; i < model->numTriangles; i++)
    {
        CalculateNormal(pointList[model->triIndex[i].meshIndex[0]].point,
                        pointList[model->triIndex[i].meshIndex[2]].point,
                        pointList[model->triIndex[i].meshIndex[1]].point);

        // definiuje współrzędne tekstury i rysuje wierzchołki
        glTexCoord2f(model->st[model->triIndex[i].stIndex[0]].s,
                     model->st[model->triIndex[i].stIndex[0]].t);
        glVertex3fv(pointList[model->triIndex[i].meshIndex[0]].point);

        glTexCoord2f(model->st[model->triIndex[i].stIndex[2]].s,
                     model->st[model->triIndex[i].stIndex[2]].t);
        glVertex3fv(pointList[model->triIndex[i].meshIndex[2]].point);

        glTexCoord2f(model->st[model->triIndex[i].stIndex[1]].s,
                     model->st[model->triIndex[i].stIndex[1]].t);
        glVertex3fv(pointList[model->triIndex[i].meshIndex[1]].point);
    }
    glEnd();
}
```

Jak łatwo zauważyc, podobnie jak było w przypadku współrzędnych wierzchołków, także i teraz współrzędne tekstury uzyskuje się za pomocą wartości indeksów przechowywanych w strukturach opisujących trójkąty (patrz: rysunek 18.4). Rezultat uzyskany w wyniku wywołania nowej wersji funkcji `DisplayMD2()` prezentuje rysunek 18.7.

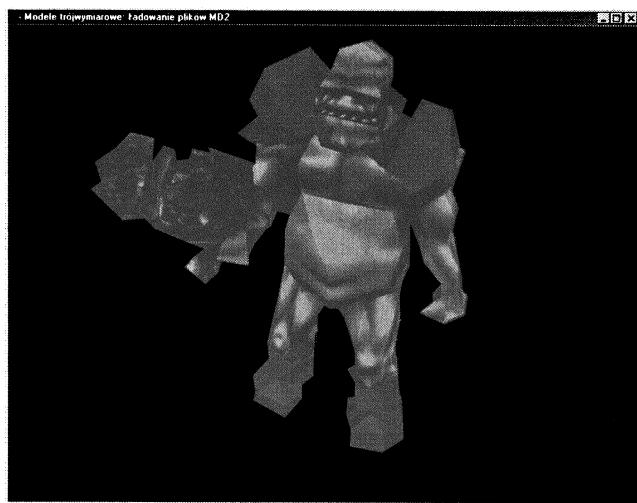
W ten sposób uzyskany został kompletny wizerunek modelu, pokryty tekstonią i oświetlony. Ma on jednak dość poważną wadę: jest statyczny. Aby tchnąć w niego życie, trzeba wykorzystać metodę *interpolacji kluczowych klatek animacji*.

## Animacja modelu

Przy definiowaniu struktury `modelData_t` reprezentującej model umieszczone w niej zostały pola `currentFrame`, `nextFrame` i `interpol`, które wykorzystane zostaną podczas animacji modelu. Animacja modelu z pewnością jest najważniejszym ze środków podnoszących realizm postaci.

**Rysunek 18.7.**

*Model MD2  
pokryty tekstuurą*

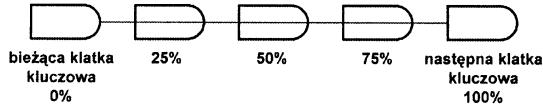


W podrozdziale tym omówiona zostanie metoda wykorzystująca interpolację klatek. Ponieważ model w formacie *MD2* zawiera jedynie kluczowe klatki animacji, to aby uzyskać efekt płynnej animacji, konieczna jest *interpolacja* wyglądu modelu pomiędzy tymi klatkami. Metoda ta znana jest także pod nazwą *morfingu*.

Załóżmy, że model zawiera klatki animacji pocisku, który przemieszcza się z punktu  $(x_0, y_0, z_0)$  do  $(x_1, y_1, z_1)$ . Twórca modelu mógł nawet umieścić w nim tylko dwie kluczowe klatki animacji: jedną przedstawiającą pocisk w punkcie  $(x_0, y_0, z_0)$  i drugą w punkcie  $(x_1, y_1, z_1)$ . Kolejnym zadaniem będzie więc wyznaczenie kilku położen pocisku na drodze z punktu  $(x_0, y_0, z_0)$  do  $(x_1, y_1, z_1)$  w taki sposób, aby można było uzyskać wrażenie płynnej animacji. Sposób wyznaczania tych położzeń prezentuje rysunek 18.8.

**Rysunek 18.8.**

*Zastosowanie  
interpolacji  
do wyznaczenia  
klatek animacji  
pomiędzy klatkami  
kluczowymi*



Metoda interpolacji klatek polega na wyznaczeniu położenia wierzchołków klatki pośredniej na podstawie bieżącej i docelowej klatki kluczowej oraz współczynnika określającego w procentach położenie klatki pośredniej na drodze pomiędzy klatkami kluczowymi. Wszystkie informacje potrzebne do wykonania tej operacji umieszczone są już w strukturze `modelData_t` i zapisane w polach `currentFrame`, `nextFrame` i `interpol`.

Pola `currentFrame` i `nextFrame` określają klatki kluczowe, pomiędzy którymi dokonuje się interpolacji. Pole `interpol` określa w procentach położenie klatki pośredniej na drodze pomiędzy tymi klatkami. Jeśli na przykład wierzchołek bieżącej klatki kluczowej posiada współrzędną  $x$  równą 0.0, pole `interpol` posiada wartość 0.5, a ten sam wierzchołek modelu w kolejnej klatce kluczowej posiada współrzędną  $x$  równą 10.0, to jego współrzędna  $x$  w klatce pośredniej uzyskana na drodze interpolacji wynosić będzie 5.0.

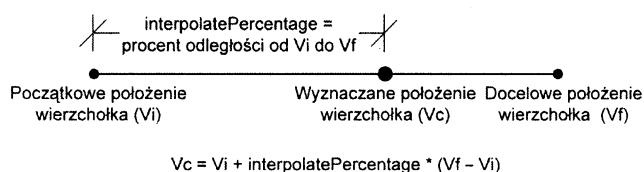
Aby wyznaczyć współrzędne wierzchołków w metodzie interpolacji klatek, można posłużyć się więc następującym wzorem:

$$X_i + \text{interpolatePercentage} * (X_f - X_i)$$

$X_i$  reprezentuje w nim położenie wierzchołka w bieżącej klatce kluczowej, a  $X_f$  w kolejnej takiej klatce. Parametr `interpolatePercentage` określa położenie rysowanej klatki pośredniej pomiędzy obiema klatkami kluczowymi. Rysunek 18.9 ilustruje znaczenie wszystkich parametrów tego wzoru.

**Rysunek 18.9.**

Interpolacja  
położenia  
wierzchołka



Przedstawiona poniżej funkcja `DisplayMD2Interpolate()` umożliwiać będzie płynną animację modelu na podstawie jego klatek kluczowych:

```
void DisplayMD2Interpolate(modelData_t *model)
{
    vector_t *pointList;      // wierzchołki bieżącej klatki
    vector_t *nextPointList; // wierzchołki następnej klatki
    int i;                  // zmenna indeksu
    float x1, y1, z1;        // współrzędne wierzchołka bieżącej klatki
    float x2, y2, z2;        // współrzędne wierzchołka następnej klatki

    vector_t vertex[3];      // zmieniąca pomocnicza

    if (model == NULL)
        return;

    // po osiągnięciu kolejnej klatki kluczowej
    // należy zmienić bieżącą i następną klatkę kluczową
    if (model->interpol >= 1.0)
    {
        model->interpol = 0.0f;           // zeruje współczynnik interpolacji
        model->currentFrame++;          // zwiększa numer bieżącej klatki
        if (model->currentFrame >= model->numFrames)
            model->currentFrame = 0;

        model->nextFrame = model->currentFrame + 1;

        if (model->nextFrame >= model->numFrames)
            model->nextFrame = 0;
    }

    // lista wierzchołków bieżącej klatki
    pointList = &model->pointList[model->numPoints*model->currentFrame];

    // lista wierzchołków następnej klatki
    nextPointList = &model->pointList[model->numPoints*model->nextFrame];

    // konfiguruje teksturę i rozpoczyna rysowanie trójkątów
    glBindTexture(GL_TEXTURE_2D, model->modelTex->texID);
    glBegin(GL_TRIANGLES);
    for (i = 0; i < model->numTriangles; i++)

```

```

{
    // współrzędne pierwszego wierzchołka trójkąta w obu kluczowych klatkach
    x1 = pointList[model->triIndex[i].meshIndex[0]].point[0];
    y1 = pointList[model->triIndex[i].meshIndex[0]].point[1];
    z1 = pointList[model->triIndex[i].meshIndex[0]].point[2];
    x2 = nextPointList[model->triIndex[i].meshIndex[0]].point[0];
    y2 = nextPointList[model->triIndex[i].meshIndex[0]].point[1];
    z2 = nextPointList[model->triIndex[i].meshIndex[0]].point[2];

    // xi + percentage * (xf - xi)
    // interpolowane współrzędne pierwszego wierzchołka
    vertex[0].point[0] = x1 + model->interpol * (x2 - x1);
    vertex[0].point[1] = y1 + model->interpol * (y2 - y1);
    vertex[0].point[2] = z1 + model->interpol * (z2 - z1);

    // współrzędne drugiego wierzchołka trójkąta
    x1 = pointList[model->triIndex[i].meshIndex[2]].point[0];
    y1 = pointList[model->triIndex[i].meshIndex[2]].point[1];
    z1 = pointList[model->triIndex[i].meshIndex[2]].point[2];
    x2 = nextPointList[model->triIndex[i].meshIndex[2]].point[0];
    y2 = nextPointList[model->triIndex[i].meshIndex[2]].point[1];
    z2 = nextPointList[model->triIndex[i].meshIndex[2]].point[2];

    // interpolowane współrzędne drugiego wierzchołka
    vertex[2].point[0] = x1 + model->interpol * (x2 - x1);
    vertex[2].point[1] = y1 + model->interpol * (y2 - y1);
    vertex[2].point[2] = z1 + model->interpol * (z2 - z1);

    // współrzędne trzeciego wierzchołka trójkąta
    x1 = pointList[model->triIndex[i].meshIndex[1]].point[0];
    y1 = pointList[model->triIndex[i].meshIndex[1]].point[1];
    z1 = pointList[model->triIndex[i].meshIndex[1]].point[2];
    x2 = nextPointList[model->triIndex[i].meshIndex[1]].point[0];
    y2 = nextPointList[model->triIndex[i].meshIndex[1]].point[1];
    z2 = nextPointList[model->triIndex[i].meshIndex[1]].point[2];

    // interpolowane współrzędne trzeciego wierzchołka
    vertex[1].point[0] = x1 + model->interpol * (x2 - x1);
    vertex[1].point[1] = y1 + model->interpol * (y2 - y1);
    vertex[1].point[2] = z1 + model->interpol * (z2 - z1);

    // wektor normalny do trójkąta
    CalculateNormal(vertex[0].point, vertex[2].point, vertex[1].point);

    // rysuje trójkąt pokryty teksturą
    glTexCoord2f(model->st[model->triIndex[i].stIndex[0]].s,
                 model->st[model->triIndex[i].stIndex[0]].t);
    glVertex3fv(vertex[0].point);

    glTexCoord2f(model->st[model->triIndex[i].stIndex[2]].s,
                 model->st[model->triIndex[i].stIndex[2]].t);
    glVertex3fv(vertex[2].point);

    glTexCoord2f(model->st[model->triIndex[i].stIndex[1]].s,
                 model->st[model->triIndex[i].stIndex[1]].t);
    glVertex3fv(vertex[1].point);
}

glEnd();

model->interpol += 0.05f;      // zwiększa współczynnik interpolacji
}

```

Funkcja `DisplayMD2Interpolate()` interpoluje współrzędne kolejnych trzech wierzchołków modelu, a następnie rysuje je jako trójkąt pokryty tekstem. Operacja ta wykonywana jest dla wszystkich wierzchołków modelu. Funkcja ta zwiększa stopniowo współczynnik interpolacji, a po osiągnięciu wartości 1.0 dokonuje zmiany klatek kluczowych, względem których wykonuje interpolację.

Chociaż zaprezentowana wyżej funkcja stanowi dobrą ilustrację metody interpolacji klatek, to jednak nie nadaje się do zastosowania w grze. Podstawową jej wadą jest brak możliwości wskazania klatki, od której rozpoczyna się animacja oraz określenia liczby klatek pośrednich. Dlatego też utworzona zostanie jej zmodyfikowana wersja poprzez dodanie parametrów pozwalających określić pierwszą i ostatnią klatkę animacji oraz wartość, o którą zwiększać się będzie współczynnik interpolacji:

```
void DisplayMD2Interpolate(modelData_t *model, int startFrame, int endFrame,
                            float percent)
{
    vector_t *pointList;      // wierzchołki bieżącej klatki
    vector_t *nextPointList; // wierzchołki następnej klatki
    int i;                   // zmieniona indeksu
    float x1, y1, z1;        // współrzędne wierzchołka bieżącej klatki
    float x2, y2, z2;        // współrzędne wierzchołka następnej klatki

    vector_t vertex[3];      // zmieniona pomocnicza

    if (model == NULL)
        return;
    if ( (startFrame > currentFrame) )
        currentFrame = startFrame;

    // weryfikacja wartości parametrów
    if ( (startFrame < 0) || (endFrame < 0) )
        return;

    // weryfikacja wartości parametrów
    if ( (startFrame >= model->numFrames) || (endFrame >= model->numFrames) )
        return;

    // po osiągnięciu kolejnej klatki kluczowej
    // należy zmienić bieżącą i następną klatkę kluczową
    if (model->interpol >= 1.0)
    {
        model->interpol = 0.0f;          // zeruje współczynnik interpolacji
        model->currentFrame++;         // zwiększa numer bieżącej klatki
        if (model->currentFrame >= endFrame)
            model->currentFrame = startFrame;

        model->nextFrame = model->currentFrame + 1;

        if (model->nextFrame >= endFrame)
            model->nextFrame = startFrame;
    }

    // lista wierzchołków bieżącej klatki
    pointList = &model->pointList[model->numPoints*model->currentFrame];
```

```

// lista wierzchołków następnej klatki
nextPointList = &model->pointList[model->numPoints*model->nextFrame];

// konfiguruje teksturę i rozpoczyna rysowanie trójkątów
glBindTexture(GL_TEXTURE_2D, model->modelTex->texID);
glBegin(GL_TRIANGLES);
for (i = 0; i < model->numTriangles; i++)
{
    // wyznacza interpolowane współrzędne i rysuje trójkąty
    ...
}
glEnd();

model->interpol += percent;// zwiększa współczynnik interpolacji
}

```

Tę wersję funkcji `DisplayMD2Interpolate()` można wywołać na przykład w następujący sposób:

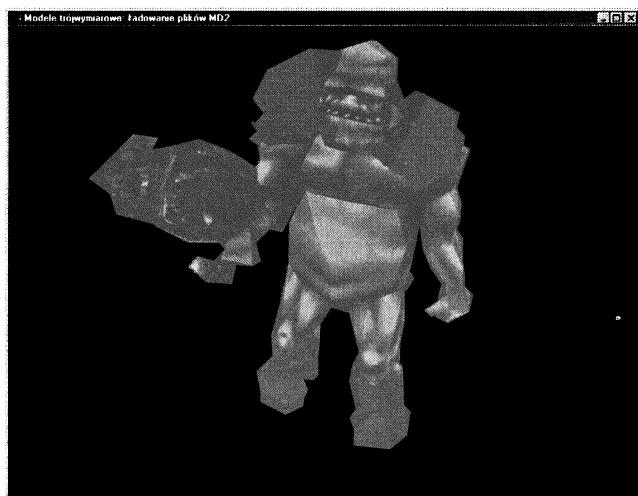
```
DisplayMD2Interpolate(myModel, 0, 40, 0.005);
```

Nowa wersja funkcji `DisplayMD2Interpolate()` umożliwia nie tylko określenie początkowej i końcowej klatki animacji, ale także sposobu, w jaki wzrasta współczynnik interpolacji. Możliwość ta okazuje się szczególnie przydatna, gdy zachodzi potrzeba określania liczby klatek animacji, by zmieściła się ona w określonym przedziale czasu (na przykład w ciągu 1 sekundy).

Chociaż ilustracja w książce nie oddaje ruchu animowanej postaci, to jednak rysunek 18.10 przedstawia efekt użycia nowej wersji funkcji `DisplayMD2Interpolate()`.

**Rysunek 18.10.**

*Model animowany za pomocą funkcji `DisplayMD2Interpolate()`*



## Klasa **CMD2Model**

I tak możliwe już jest załadowanie, wyświetlenie, a nawet animowanie modelu w formacie *MD2*. Dostępny jest także odpowiedni zestaw struktur reprezentujących taki model.

Jednak dołączanie opracowanych dotąd struktur danych i funkcji do kodu przygotowywanego gry jest nieco kłopotliwe.

Wygodniejszym rozwiązaniem będzie stworzenie klasy języka C++ reprezentującej model w formacie *MD2*. Klasa ta umożliwiać będzie łatwe rozszerzanie reprezentacji modelu, co nie było możliwe w przypadku reprezentacji modelu przedstawionej w pośrednim podrozdziale. A oto definicja klasy *CMD2Model*, która wykorzystuje opracowane wcześniej struktury:

```
class CMD2Model
{
    private:

        int numFrames;           // liczba klatek animacji
        int numVertices;         // liczba wierzchołków modelu
        int numTriangles;        // liczba trójkątów modelu
        int numST;               // liczba tekstur modelu
        int frameSize;           // rozmiar klatki w bajtach
        int currentFrame;        // bieżąca klatka animacji
        int nextFrame;            // kolejna klatka animacji
        float interpol;           // współczynnik interpolacji
        mesh_t *triIndex;         // lista trójkątów
        texCoord_t *st;           // lista współrzędnych tekstury
        vector_t *vertexList;      // lista wierzchołków
        texture_t *modelTex;       // dane tekstury

        void SetupSkin(texture_t *thisTexture);

    public:

        CMD2Model();           // konstruktor
        ~CMD2Model();          // destruktor

        // ładuje model i jego tekstury
        int Load(char *modelFile, char *skinFile);

        // ładuje tylko model
        int LoadModel(char *modelFile);

        // ładuje tylko tekstury
        int LoadSkin(char *skinFile);

        // określa teksturę modelu
        int SetTexture(texture_t *texture);

        // animuje model metodą interpolacji klatek kluczowych
        int Animate(int startFrame, int endFrame, float percent);

        // wyświetla wybraną klatkę modelu
        int RenderFrame(int keyFrame);

        // zwalnia pamięć wykorzystywaną przez model
        int Unload();
};
```

Klasa `CMD2Model` hermetyzuje opracowaną dotąd funkcjonalność modelu. Składowe o dostępie prywatnym są na tym etapie już dobrze znane. Metoda `SetupSkin()` stanowi odpowiednik wykorzystywanej poprzednio funkcji `SetupMD2Texture()`.

Składowe o dostępie publicznym to kilka metod, których funkcjonalność odpowiada w zasadzie funkcjom, które zaimplementowane zostały w poprzednim podrozdziale. Na przykład metoda `Load()` ładuje kompletny model wraz z tekstem tak, by był on gotowy do użycia. Metody `LoadSkin()`, `LoadModel()` i `SetTexture()` dokonują jedynie częściowej inicjalizacji modelu. Metoda `LoadSkin()` ładuje jedynie teksturę modelu z podanego pliku, a metoda `LoadModel()` model pozbawiony tekstury. Metoda `SetTexture()` wybiera załadowaną wcześniej teksturę dla danego modelu.

Metoda `Animate()` rysuje animowany model posługując się klatkami kluczowymi od `startFrame` do `endFrame` i metodą interpolacji tych klatek. Metoda `RenderFrame()` rysuje pojedynczą, nieruchomą klatkę. Ostatnia z metod, `UnLoad()`, zwalnia pamięć wykorzystywaną przez model. Hermetyzacja modelu za pomocą klasy języka C++ nie zmieniła więc jego funkcjonalności, lecz ułatwiła wykorzystanie go w programach.

Ponieważ omówiony już został sposób działania każdej z funkcji modelu *MD2*, nie będzie omawiana ponownie implementacja odpowiadających im metod w wersji obiektowej. Warto jednak zwrócić uwagę na niewielkie zmiany, których wymagały opracowane wcześniej funkcje po to, aby mogły stać się metodami klasy `CMD2Model`. A oto ich implementacja:

```
CMD2Model::CMD2Model()
{
    numVertices = 0;      // wierzchołki
    numTriangles = 0;     // trójkąty
    numFrames = 0;        // klatki
    numST = 0;            // współrzędne tekstury
    frameSize = 0;         // rozmiar klatki
    currentFrame = 0;      // bieżąca klatka
    nextFrame = 1;          // następna klatka
    interpol = 0.0;        // współczynnik interpolacji
    triIndex = NULL;       // indeksy trójkątów
    st = NULL;             // indeksy współrzędnych tekstury
    vertexList = NULL;     // lista wierzchołków
    modelTex = NULL;       // tekstura
    ModelState = MODEL_IDLE;
}

CMD2Model::~CMD2Model()
{
}

void CMD2Model::SetupSkin(texture_t *thisTexture)
{
    // konfiguruje teksturę modelu MD2
    glGenTextures(1, &thisTexture->texID);
    glBindTexture(GL_TEXTURE_2D, thisTexture->texID);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
}
```

```
switch (thisTexture->textureType)
{
    case BMP:
        gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, thisTexture->width,
                           thisTexture->height, GL_RGB, GL_UNSIGNED_BYTE,
                           thisTexture->data);
        break;

    case PCX:
        gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, thisTexture->width,
                           thisTexture->height, GL_RGBA, GL_UNSIGNED_BYTE,
                           thisTexture->data);
        break;

    case TGA:
        break;

    default:
        break;
}

int CMD2Model::Load(char *modelFile, char *skinFile)
{
    FILE *filePtr;           // wskaźnik pliku
    int fileLen;             // długość pliku
    char *buffer;            // bufor pliku

    modelHeader_t *modelHeader; // nagłówek modelu

    stIndex_t *stPtr;         // dane tekstury
    frame_t *frame;           // dane klatki
    vector_t *vertexListPtr;   // zmienna indeksu
    mesh_t *bufIndexPtr;      // zmienna indeksu
    int i, j;                 // zmienne indeksu

    // otwiera plik modelu
    filePtr = fopen(modelFile, "rb");
    if (filePtr == NULL)
        return FALSE;

    // ustala długość pliku
    fseek(filePtr, 0, SEEK_END);
    fileLen = ftell(filePtr);
    fseek(filePtr, 0, SEEK_SET);

    // wczytuje cały plik do bufora
    buffer = new char [fileLen+1];
    fread(buffer, sizeof(char), fileLen, filePtr);

    // wyodrębnia nagłówek
    modelHeader = (modelHeader_t*)buffer;

    vertexList = new vector_t [modelHeader->numXYZ * modelHeader->numFrames];

    numVertices = modelHeader->numXYZ;
    numFrames = modelHeader->numFrames;
    frameSize = modelHeader->framesize;
```

```
for (j = 0; j < numFrames; j++)
{
    frame = (frame_t*)&buffer[modelHeader->offsetFrames + frameSize * j];

    vertexListPtr = (vector_t*)&vertexList[numVertices * j];
    for (i = 0; i < numVertices; i++)
    {
        vertexListPtr[i].point[0] = frame->scale[0] * frame->fp[i].v[0] +
            frame->translate[0];
        vertexListPtr[i].point[1] = frame->scale[1] * frame->fp[i].v[1] +
            frame->translate[1];
        vertexListPtr[i].point[2] = frame->scale[2] * frame->fp[i].v[2] +
            frame->translate[2];
    }
}

modelTex = LoadTexture(skinFile);
if (modelTex != NULL)
    SetupSkin(modelTex);
else
    return FALSE;

numST = modelHeader->numST;
st = new texCoord_t [numST];

stPtr = (stIndex_t*)&buffer[modelHeader->offsetST];
for (i = 0; i < numST; i++)
{
    st[i].s = (float)stPtr[i].s / (float)modelTex->width;
    st[i].t = (float)stPtr[i].t / (float)modelTex->height;
}

numTriangles = modelHeader->numTris;
triIndex = new mesh_t [numTriangles];

// tymczasowy wskaźnik bufora trójkątów
bufIndexPtr = (mesh_t*)&buffer[modelHeader->offsetTris];

// wypełnia listę trójkątów
for (j = 0; j < numFrames; j++)
{
    // dla wszystkich trójkątów danej klatki
    for(i = 0; i < numTriangles; i++)
    {
        triIndex[i].meshIndex[0] = bufIndexPtr[i].meshIndex[0];
        triIndex[i].meshIndex[1] = bufIndexPtr[i].meshIndex[1];
        triIndex[i].meshIndex[2] = bufIndexPtr[i].meshIndex[2];
        triIndex[i].stIndex[0] = bufIndexPtr[i].stIndex[0];
        triIndex[i].stIndex[1] = bufIndexPtr[i].stIndex[1];
        triIndex[i].stIndex[2] = bufIndexPtr[i].stIndex[2];
    }
}

// zamyka plik i zwalnia jego bufor
fclose(filePtr);
free(buffer);
```

```
currentFrame = 0;
nextFrame = 1;
interpol = 0.0;

    return TRUE;
}

int CMD2Model::LoadModel(char *modelFile)
{
    FILE *filePtr;                      // wskaźnik pliku
    int fileLen;                        // długość pliku
    char *buffer;                       // bufor pliku

    modelHeader_t *modelHeader;          // nagłówek modelu

    stIndex_t *stPtr;                   // dane tekstury
    frame_t *frame;                    // dane klatek
    vector_t *vertexListPtr;           // zmienna indeksu
    mesh_t *triIndex, *bufIndexPtr;    // zmienne indeksu
    int i, j;                          // zmienne indeksu

    // otwiera plik modelu
    filePtr = fopen(modelFile, "rb");
    if (filePtr == NULL)
        return FALSE;

    // ustala długość pliku
    fseek(filePtr, 0, SEEK_END);
    fileLen = ftell(filePtr);
    fseek(filePtr, 0, SEEK_SET);

    // wczytuje cały plik do bufora
    buffer = new char [fileLen+1];
    fread(buffer, sizeof(char), fileLen, filePtr);

    // wyodrębnia nagłówek
    modelHeader = (modelHeader_t*)buffer;

    // przydziela pamięć na listę wierzchołków
    vertexList = new vector_t [modelHeader->numXYZ * modelHeader->numFrames];

    numVertices = modelHeader->numXYZ;
    numFrames = modelHeader->numFrames;
    frameSize = modelHeader->framesize;

    for (j = 0; j < numFrames; j++)
    {
        frame = (frame_t*)&buffer[modelHeader->offsetFrames + frameSize * j];

        vertexListPtr = (vector_t*)&vertexList[numVertices * j];
        for (i = 0; i < numVertices; i++)
        {
            vertexListPtr[i].point[0] = frame->scale[0] * frame->fp[i].v[0] +
                frame->translate[0];
            vertexListPtr[i].point[1] = frame->scale[1] * frame->fp[i].v[1] +
                frame->translate[1];
        }
    }
}
```

```

        vertexListPtr[i].point[2] = frame->scale[2] * frame->fp[i].v[2] +
                                    frame->translate[2];
    }
}

numST = modelHeader->numST;

st = new texCoord_t [numST];

stPtr = (stIndex_t*)&buffer[modelHeader->offsetST];
for (i = 0; i < numST; i++)
{
    st[i].s = 0.0;
    st[i].t = 0.0;
}

numTriangles = modelHeader->numTris;
triIndex = new mesh_t [numTriangles];

// tymczasowy wskaźnik bufora trójkątów
bufIndexPtr = (mesh_t*)&buffer[modelHeader->offsetTris];

// wypełnia listę trójkątów
for (j = 0; j < numFrames; j++)
{
    // dla wszystkich trójkątów danej klatki
    for(i = 0; i < numTriangles; i++)
    {
        triIndex[i].meshIndex[0] = bufIndexPtr[i].meshIndex[0];
        triIndex[i].meshIndex[1] = bufIndexPtr[i].meshIndex[1];
        triIndex[i].meshIndex[2] = bufIndexPtr[i].meshIndex[2];
        triIndex[i].stIndex[0] = bufIndexPtr[i].stIndex[0];
        triIndex[i].stIndex[1] = bufIndexPtr[i].stIndex[1];
        triIndex[i].stIndex[2] = bufIndexPtr[i].stIndex[2];
    }
}

// zamyka plik
fclose(filePtr);

modelTex = NULL;
currentFrame = 0;
nextFrame = 1;
interpol = 0.0;

return 0;
}

int CMD2Model::LoadSkin(char *skinFile)
{
    int i;

    modelTex = LoadTexture(skinFile);

    if (modelTex != NULL)
        SetupSkin(modelTex);
    else
        return -1;
}

```

```
for (i = 0; i < numST; i++)
{
    st[i].s /= (float)modelTex->width;
    st[i].t /= (float)modelTex->height;
}

return 0;
}

int CMD2Model::SetTexture(texture_t *texture)
{
    int i;

    if (texture != NULL)
    {
        free(modelTex);
        modelTex = texture;
    }
    else
        return -1;

    SetupSkin(modelTex);

    for (i = 0; i < numST; i++)
    {
        st[i].s /= (float)modelTex->width;
        st[i].t /= (float)modelTex->height;
    }

    return 0;
}

int CMD2Model::Animate(int startFrame, int endFrame, float percent)
{
    vector_t *vList;           // wierzchołki bieżącej klatki
    vector_t *nextVList;       // wierzchołki następnej klatki
    int i;                     // zmienna indeksu
    float x1, y1, z1;          // współrzędne wierzchołka bieżącej klatki
    float x2, y2, z2;          // współrzędne wierzchołka następnej klatki

    vector_t vertex[3];

    if ((startFrame > currentFrame))
        currentFrame = startFrame;

    if ((startFrame < 0) || (endFrame < 0))
        return -1;

    if ((startFrame >= numFrames) || (endFrame >= numFrames))
        return -1;

    if (interpol >= 1.0)
    {
        interpol = 0.0f;
        currentFrame++;
        if (currentFrame >= endFrame)
            currentFrame = startFrame;
```

```
nextFrame = currentFrame + 1;

if (nextFrame >= endFrame)
    nextFrame = startFrame;

}

vList = &vertexList[numVertices*currentFrame];
nextVList = &vertexList[numVertices*nextFrame];

glBindTexture(GL_TEXTURE_2D, modelTex->texID);
glBegin(GL_TRIANGLES);
for (i = 0; i < numTriangles; i++)
{
    // współrzędne pierwszego wierzchołka trójkąta w obu kluczowych klatkach
    x1 = vList[triIndex[i].meshIndex[0]].point[0];
    y1 = vList[triIndex[i].meshIndex[0]].point[1];
    z1 = vList[triIndex[i].meshIndex[0]].point[2];
    x2 = nextVList[triIndex[i].meshIndex[0]].point[0];
    y2 = nextVList[triIndex[i].meshIndex[0]].point[1];
    z2 = nextVList[triIndex[i].meshIndex[0]].point[2];

    // interpolowane współrzędne pierwszego wierzchołka
    vertex[0].point[0] = x1 + interpol * (x2 - x1);
    vertex[0].point[1] = y1 + interpol * (y2 - y1);
    vertex[0].point[2] = z1 + interpol * (z2 - z1);

    // współrzędne drugiego wierzchołka trójkąta
    x1 = vList[triIndex[i].meshIndex[2]].point[0];
    y1 = vList[triIndex[i].meshIndex[2]].point[1];
    z1 = vList[triIndex[i].meshIndex[2]].point[2];
    x2 = nextVList[triIndex[i].meshIndex[2]].point[0];
    y2 = nextVList[triIndex[i].meshIndex[2]].point[1];
    z2 = nextVList[triIndex[i].meshIndex[2]].point[2];

    // interpolowane współrzędne drugiego wierzchołka
    vertex[2].point[0] = x1 + interpol * (x2 - x1);
    vertex[2].point[1] = y1 + interpol * (y2 - y1);
    vertex[2].point[2] = z1 + interpol * (z2 - z1);

    // współrzędne trzeciego wierzchołka trójkąta
    x1 = vList[triIndex[i].meshIndex[1]].point[0];
    y1 = vList[triIndex[i].meshIndex[1]].point[1];
    z1 = vList[triIndex[i].meshIndex[1]].point[2];
    x2 = nextVList[triIndex[i].meshIndex[1]].point[0];
    y2 = nextVList[triIndex[i].meshIndex[1]].point[1];
    z2 = nextVList[triIndex[i].meshIndex[1]].point[2];

    // interpolowane współrzędne trzeciego wierzchołka
    vertex[1].point[0] = x1 + interpol * (x2 - x1);
    vertex[1].point[1] = y1 + interpol * (y2 - y1);
    vertex[1].point[2] = z1 + interpol * (z2 - z1);

    // wektor normalny do trójkąta
    CalculateNormal(vertex[0].point, vertex[2].point, vertex[1].point);
}
```

```
// rysuje trójkąt pokryty teksturową
glTexCoord2f(st[triIndex[i].stIndex[0]].s,
             st[triIndex[i].stIndex[0]].t);
glVertex3fv(vertex[0].point);

glTexCoord2f(st[triIndex[i].stIndex[2]].s ,
             st[triIndex[i].stIndex[2]].t);
glVertex3fv(vertex[2].point);

glTexCoord2f(st[triIndex[i].stIndex[1]].s,
             st[triIndex[i].stIndex[1]].t);
glVertex3fv(vertex[1].point);
}

g1End();

interpol += percent; // zwiększa współczynnik interpolacji

return 0;
}

int CMD2Model::RenderFrame(int keyFrame)
{
    vector_t *vList;
    int i;

    // wskaźnik wyświetlanej klatki
    vList = &vertexList[numVertices * keyFrame];

    // wybiera teksturę
    glBindTexture(GL_TEXTURE_2D, modelTex->texID);

    // wyświetla klatkę modelu
    glBegin(GL_TRIANGLES);
    for(i = 0; i < numTriangles; i++)
    {
        CalculateNormal(vList[triIndex[i].meshIndex[0]].point,
                         vList[triIndex[i].meshIndex[2]].point,
                         vList[triIndex[i].meshIndex[1]].point);

        glTexCoord2f(st[triIndex[i].stIndex[0]].s,
                     st[triIndex[i].stIndex[0]].t);
        glVertex3fv(vList[triIndex[i].meshIndex[0]].point);

        glTexCoord2f(st[triIndex[i].stIndex[2]].s ,
                     st[triIndex[i].stIndex[2]].t);
        glVertex3fv(vList[triIndex[i].meshIndex[2]].point);

        glTexCoord2f(st[triIndex[i].stIndex[1]].s,
                     st[triIndex[i].stIndex[1]].t);
        glVertex3fv(vList[triIndex[i].meshIndex[1]].point);
    }
    g1End();

    return 0;
}

int CMD2Model::Unload()
{
```

```

    if (triIndex != NULL)
        free(triIndex);
    if (vertexList != NULL)
        free(vertexList);
    if (st != NULL)
        free(st);

    return 0;
}

```

Obiektową wersję modelu *MD2* można łatwo wykorzystać w tworzonych programach. Aby na przykład załadować model, wyświetlić jego animację, a następnie zwolnić pamięć zajmowaną przez model, można posłużyć się następującymi wierszami kodu:

```

CMD2Model *myModel; // instancja modelu
...
myModel = new CMD2Model; // tworzy obiekt
...
myModel->Load("mymodel.md2", "mymodel.pcx");
...
myModel->Animate(0, 40, percentage);
...
myModel->UnLoad();
...
delete myModel;

```

## Sterowanie animacją modelu

Mimo że opanowana została umiejętność odtwarzania animacji modelu, gry wymagają ponadto dostosowania sposobu animacji postaci do zmieniającej się sytuacji. Zwykle więc animowany obiekt musi posiadać odpowiedni stan określający sposób animacji. Na przykład stan bezczynności oznacza, że model nie wykonuje żadnych akcji. Inne typowe stany modeli postaci występujących w grze będą reprezentować chód, bieg, skoki, kucanie i agonię.

Stany, w których może znajdować się model, zależą od rodzaju gry, dla której został on opracowany. Tabela 18.1 przedstawia klatki modelu w formacie *MD2* odpowiadające różnym jego stanom.

Jeśli na przykład model dysponować ma czterema stanami reprezentującymi bezczynność, bieg, skok i kucanie, to można utworzyć dla nich następujący typ wyliczenia:

```

enum ModelState_t
{
    MODEL_IDLE,      // bezczynność
    MODEL_CROUCH,   // kucanie
    MODEL_RUN,       // bieg
    MODEL_JUMP       // skok
};

```

Następnie definicję klasy *CMD2Model* rozszerzyć należy o dwie metody służące zmianie i pobraniu bieżącego stanu modelu:

**Tabela 18.1.** Stany animacji modelu MD2

Numer klatki	Stan animacji modelu
0 – 39	Bezczynność
40 – 46	Bieg
47 – 60	Postrzał
61 – 66	Postrzał w plecy
67 – 73	Skok
74 – 95	Bezczynność
96 – 112	Postrzał i upadek
113 – 122	Bezczynność
123 – 135	Bezczynność
136 – 154	Kucanie
155 – 161	Czołganie
162 – 169	Bezczynność w pozycji przykućniętej
170 – 177	Agonia na kolanach
178 – 185	Upadek na plecy i agonia
186 – 190	Upadek na brzuch i agonia
191 – 198	Osunięcie się i agonia

```
// zmienia stan animacji modelu
int SetState(modelState_t state);

// pobiera stan animacji modelu
modelState_t GetState();
```

Także składowe o dostępie prywatnym trzeba uzupełnić o zmienną, która przechowysywać będzie bieżący stan animacji modelu:

```
modelState_t modelState; // bieżący stan animacji modelu
```

Najczęściej stan animacji modelu ulegać będzie zmianie na skutek akcji wykonywanych przez użytkownika, na przykład za pomocą klawiatury. Najprostszym rozwiązaniem będzie więc zmiana stanu animacji w odpowiedzi na naciśnięcie przez użytkownika określonego klawisza. Dla uproszczenia można założyć, że funkcja WinMain() będzie przechowywać nowy stan animacji za pomocą poniższej zmiennej globalnej:

```
modelState_t modelState; // globalna zmienna stanu
```

A oto zmodyfikowana pętla przetwarzania komunikatów funkcji WinMain():

```
// pętla przetwarzania komunikatów
while (!done)
{
    PeekMessage(&msg, hwnd, NULL, NULL, PM_REMOVE);

    if (msg.message == WM_QUIT) // aplikacja odebrała komunikat WM_QUIT?
    {
        done = true;           // jeśli tak, to aplikacja kończy działanie
```

```
        }
    else
    {
        if (keyPressed[VK_ESCAPE])
            done = true;
        else
        {
            if (keyPressed[VK_UP])
                modelState = MODEL_RUN;
            else if (keyPressed[VK_CONTROL])
                modelState = MODEL_CROUCH;
            else if (keyPressed[VK_SHIFT])
                modelState = MODEL_JUMP;
            else
                modelState = MODEL_IDLE;

        Render();

        TranslateMessage(&msg); // tłumaczy i wysyła komunikat
        DispatchMessage(&msg);
    }
}
```

Zmiana stanu dokonana przez funkcję WinMain() wykorzystywana będzie przez funkcję Render(), która wyświetlać będzie odpowiednią animację modelu:

```
void Render()
{
    float percent;
    ...
    percent = 0.07f;
    ...

    glPushMatrix();
    glRotatef(90.0f, -1.0f, 0.0f, 0.0f);
    glColor3f(1.0, 1.0, 1.0);
    // zmienia bieżący stan animacji modelu
    myModel->SetState(modelState);
    gunModel->SetState(modelState);

    // wyświetla animację odpowiednią do stanu modelu
    // UWAGA: poniższy sposób nie jest wzorcowy!!!
    switch (myModel->GetState())
    {
        case MODEL_IDLE:
            myModel->Animate(0, 39, percent);
            gunModel->Animate(0, 39, percent);
            break;
        case MODEL_RUN:
            myModel->Animate(40, 46, percent);
            gunModel->Animate(40, 46, percent);
            break;
        case MODEL_CROUCH:
            myModel->Animate(136, 154, percent);
            gunModel->Animate(136, 154, percent);
            break;
    }
}
```

```
    case MODEL_JUMP:
        myModel->Animate(67, 73, percent);
        gunModel->Animate(67, 73, percent);
        break;
    default:
        break;
    }
    glPopMatrix();
    ...
}
```

I to wszystko! Gdy użytkownik nie naciśnie żadnego klawisza, to model znajdować się będzie w stanie bezczynności. Wybranie klawisza oznaczonego strzałką w góre spowoduje wyświetlenie animacji biegu, a klawiszy *Ctrl* i *Shift* (odpowiednio) animacji kucania i skoku.

Na tym można zakończyć omówienie formatu *MD2*. Stanowi on dobry przykład podstawowego formatu modeli trójwymiarowych, ale nie musi być najważniejszym formatem we wszystkich zastosowaniach. Dlatego warto poeksperymentować z własnymi formatami, a także zapoznać się z formatami modeli stosowanymi w innych grach.

## Ładowanie plików PCX

Ponieważ większość modeli używanych w grze *Quake 2* posiada tekstury zapisane w plikach formatu *PCX*, poniżej zaprezentowany został kod funkcji umożliwiającej załadowanie tekstuury z takiego pliku. Nie będzie tutaj omawiana szczegółowo implementacja tej funkcji, gdyż analizę jej kodu pozostawiamy czytelnikowi.

```
// LoadPCXFile()
// opis: ładuje zawartość pliku PCX
unsigned char *LoadPCXFile(char *filename, PCXHEADER *pcxHeader)
{
    int idx = 0;                      // licznik indeksu
    int c;                            // pobiera znak z pliku
    int i;                            // licznik indeksu
    int numRepeat;                   // wskaźnik pliku
    FILE *filePtr;                   // szerokość obrazka PCX
    int width;                        // wysokość obrazka PCX
    int height;                       // dane obrazka PCX
    unsigned char *pixelData;         // dane palety PCX
    unsigned char *paletteData;       // otwiera plik PCX
    filePtr = fopen(filename, "rb");
    if (filePtr == NULL)
        return NULL;

    // pobiera pierwszy znak z pliku, który powinien mieć wartość 10
    c = getc(filePtr);
    if (c != 10)
    {
        fclose(filePtr);
        return NULL;
    }
```

```
// pobiera następny znak, który powinien mieć wartość 5
c = getc(filePtr);
if (c != 5)
{
    fclose(filePtr);
    return NULL;
}

// ustawia wskaźnik pliku na jego początek
rewind(filePtr);

// pomija pierwsze 4 znaki
fgetc(filePtr);
fgetc(filePtr);
fgetc(filePtr);
fgetc(filePtr);

// pobiera współrzędną x lewej krawędzi obrazka PCX
pcxHeader->xMin = fgetc(filePtr);           // mniej znaczące słowo
pcxHeader->xMin |= fgetc(filePtr) << 8; // bardziej znaczące słowo

// pobiera współrzędną y dolnej krawędzi obrazka PCX
pcxHeader->yMin = fgetc(filePtr);           // mniej znaczące słwo
pcxHeader->yMin |= fgetc(filePtr) << 8; // bardziej znaczące słwo

// pobiera współrzędną x prawej krawędzi obrazka PCX
pcxHeader->xMax = fgetc(filePtr);           // mniej znaczące słwo
pcxHeader->xMax |= fgetc(filePtr) << 8; // bardziej znaczące słwo

// pobiera współrzędną y górnej krawędzi obrazka PCX
pcxHeader->yMax = fgetc(filePtr);           // mniej znaczące słwo
pcxHeader->yMax |= fgetc(filePtr) << 8; // bardziej znaczące słwo

// oblicza szerokość i wysokość obrazka PCX
width = pcxHeader->xMax - pcxHeader->xMin + 1;
height = pcxHeader->yMax - pcxHeader->yMin + 1;

// przydziela pamięć na dane obrazka PCX
pixelData = (unsigned char*)malloc(width*height);

// ustawia wskaźnik pliku na 128. bajt, gdzie zaczynają się dane obrazka
fseek(filePtr, 128, SEEK_SET);

// dekoduje i przechowuje piksele obrazka
while (idx < (width*height))
{
    c = getc(filePtr);
    if (c > 0xbf)
    {
        numRepeat = 0x3f & c;
        c = getc(filePtr);

        for (i = 0; i < numRepeat; i++)
        {
            pixelData[idx++] = c;
        }
    }
}
```

```
        else
            pixelData[idx++] = c;

        fflush(stdout);
    }

    // przydziela pamięć na paletę obrazka PCX
    paletteData = (unsigned char*)malloc(768);

    // paleta zajmuje 769 końcowych bajtów pliku PCX
    fseek(filePtr, -769, SEEK_END);

    // weryfikuje paletę; pierwszy znak powinien mieć wartość 12
    c = getc(filePtr);
    if (c != 12)
    {
        fclose(filePtr);
        return NULL;
    }

    // wczytuje paletę
    for (i = 0; i < 768; i++)
    {
        c = getc(filePtr);
        paletteData[i] = c;
    }

    // zamyka plik i umieszcza wskaźnik palety w nagłówku
    fclose(filePtr);
    pcxHeader->palette = paletteData;

    // zwraca wskaźnik do danych obrazka
    return pixelData;
}
```

## Podsumowanie

Format pliku *MD2* składa się z dwóch części: nagłówka i właściwych danych modelu. Nagłówek zawiera podstawowe informacje dotyczące rozmiarów modelu, takie jak na przykład liczba wierzchołków i trójkątów, a także informacje o położeniu danych opisujących model w pliku.

*Klatki kluczowe* reprezentują wygląd animowanego modelu w pewnych, stałych odstępach czasu.

*Interpolacja klatek kluczowych* pozwala wyznaczyć klatki pośrednie niezbędne do uzyskania efektu płynnej animacji. Modele w formacie *MD2* przechowują jedynie kluczowe klatki animacji, co pozwala zmniejszyć rozmiar plików.

Jedną z metod sterowania animacją modelu, z pewnością nie najdoskonalszą, stanowią stany animacji modelu. Zmiana stanu animacji modelu w odpowiedzi na akcję użytkownika lub inne zdarzenie pozwala uzyskać bardziej realistyczne modele postaci.



## Rozdział 19.

# Modelowanie fizycznych właściwości świata

O atrakcyjności współczesnych gier decyduje przede wszystkim niezwykle realistyczne przedstawienie wirtualnego świata gry. Nawet najdoskonalsza grafika nie zrobi na graczu odpowiedniego wrażenia, jeśli prezentowane przez nią obiekty nie będą zachowywać się w wystarczająco realistyczny sposób. Czasy, w których projektanci tworzyli gry podporządkowując ich sposób działania skromnym możliwościom sprzętowym, minęły bezpowrotnie. Obecnie obiekty prezentowane w grach muszą zachowywać się zgodnie z doświadczeniami ze świata rzeczywistego.

W rozdziale tym przedstawione więc zostaną sposoby symulacji rzeczywistego świata wykorzystujące modele fizyczne oparte na zasadach dynamiki Newtona oraz metody wykrywania zderzeń.

## Powtórka z fizyki

Przed omówieniem sposobów symulacji właściwości rzeczywistego świata, trzeba najpierw upewnić się, że posiada się solidną wiedzę z zakresu podstaw fizyki. Nie trzeba tu przeprowadzić pełnego kursu podstaw fizyki, a jedynie ograniczyć się można do omówienia podstawowych zagadnień, których zrozumienie niezbędne jest do tworzenia realistycznych światów gier.

### Czas

Mimo że pojęcia czasu nie można łatwo zdefiniować, wszyscy dobrze wiedzą, czym jest czas. Działanie cywilizacji opiera się na wykorzystaniu kalendarza i zegarów. Używa się ich, aby odmierzać czas w sekundach, minutach, godzinach, dniach i latach. Różne dziny nauki i techniki wymagają bardziej dokładnych pomiarów czasu w milisekundach ( $10^{-3}$ ), mikrosekundach ( $10^{-6}$ ) i nanosekundach ( $10^{-9}$ ).

Fizyka używa pojęcia czasu dla określenia zmiany położenia obiektów w przestrzeni. Na przykład mówiąc, że obiekt porusza się z prędkością *40 kilometrów na godzinę* ma się na myśli to, że w okresie jednej godziny pokonuje on dystans czterdziestu kilometrów.

Czas można stosunkowo łatwo symulować w tworzonych grach. Można na przykład zdefiniować *wirtualny czas gry*, którego jedna sekunda równa jest okresowi potrzebnemu na stworzenie jednej klatki obrazu. Aby jednak rozwiązywanie takie właściwie modelowało wpływ fizycznego czasu, niezbędne jest zachowanie stałej prędkości tworzenia klatek. Można założyć na przykład, że początkowo tworzy się 30 klatek na sekundę i stała prędkość pewnego obiektu w wirtualnym świecie wynosi 30 metrów na sekundę. Jeśli następnie prędkość tworzenia klatek spadnie do 20 na sekundę, to również odpowiednio zwolni swój ruch wspomniany obiekt, mimo że nie zmieniono jego prędkości.

Innym rozwiązyaniem problemu symulacji czasu w grach będzie zastosowanie czasu rzeczywistego. Pozwala ono na doskonalsze modelowanie rzeczywistego świata, gdyż nie posiada opisanych ograniczeń poprzedniego rozwiązania. Prędkość ruchu obiektów nie będzie tu zależeć już od prędkości tworzenia klatek obrazu. Ponieważ celem twórców gier jest osiągnięcie możliwie najdoskonalszego poziomu realizmu wirtualnego świata, to w tworzonych programach będzie się więc korzystać z czasu rzeczywistego.

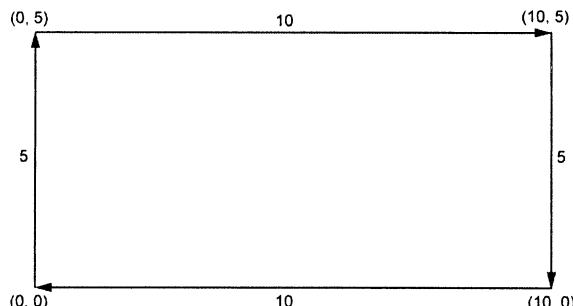
## Odległość, przemieszczenie i położenie

Odległość i przemieszczenie, choć pozornie wydają się oznaczać to samo, stanowią jednak dwie różne wielkości o różnej definicji i znaczeniu. *Odległość* jest wielkością skalarną definiującą długość odcinka łączącego punkty, pomiędzy którymi przemieścił się obiekt. *Przemieszczenie* jest natomiast wektorem definiującym zmianę położenia obiektu. Aby lepiej zrozumieć różnice pomiędzy tymi pojęciami, należy posłużyć się przykładem.

Można przyjąć, że obiekt porusza się po drodze w kształcie prostokąta pokazanej na rysunku 19.1. Ruch swój rozpoczyna w punkcie  $(0, 0)$  i udaje się na północ do punktu  $(0, 5)$ , następnie na wschód — do punktu  $(10, 5)$ , po czym w kierunku południowym — przez punkt  $(10, 0)$  — powraca do punktu wyjścia poruszając się na zachód.

**Rysunek 19.1.**

Obiekt obiegający obwód prostokąta pokonuje pewną odległość, ale nie przemieszcza się



Obiekt ten przebywa następującą odległość:

$$\begin{aligned} \text{odległość} &= \text{odległość w kierunku północnym} + \text{odległość w kierunku wschodnim} + \\ &\quad \text{odległość w kierunku południowym} + \text{odległość w kierunku zachodnim} \\ \text{odległość} &= 5 + 10 + 5 + 10 \\ \text{odległość} &= 30 \end{aligned}$$

Jeśli jednak przemieszczenie obiektu wyrazi się za pomocą wektorów, to okaże się, że nie zostało ono dokonane:

$$\begin{aligned} \text{przemieszczenie} &= \text{wektor w kierunku północnym} + \text{wektor w kierunku wschodnim} + \\ &\quad \text{wektor w kierunku południowym} + \text{wektor w kierunku zachodnim} \\ \text{przemieszczenie} &= (0, 5) + (10, 0) + (0, -5) + (-10, 0) \\ \text{przemieszczenie} &= (0 + 10 + 0 - 10, 5 + 0 - 5 + 0) \\ \text{przemieszczenie} &= (0, 0) \end{aligned}$$

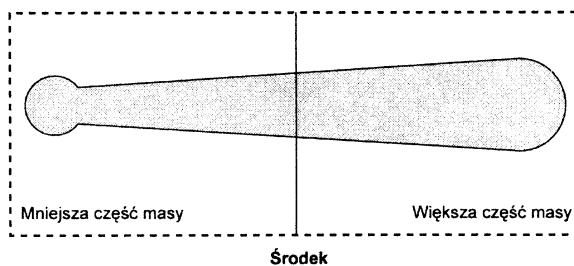
Mimo że obiekt przebył pewną odległość, to jednak nie przemieścił się.

Aby określić odległość przebytą przez obiekt lub wyznaczyć jego przemieszczenie, trzeba posiadać informację o położeniu obiektu. W przestrzeni położenie obiektu opisują trzy współrzędne ( $x, y, z$ ), a na płaszczyźnie tylko dwie ( $x, y$ ). Dla wielu obiektów, na przykład piłki, wystarczy jedynie informacja o położeniu ich geometrycznego środka. Sytuacja komplikuje się w przypadku obiektów o nieregularnych kształtach, takich jak na przykład kij do bejsbolu.

Spoglądając na rysunek 19.2 łatwo zauważyc, że masa kija jest istotnie większa na jednym z jego końców. Zamiast więc reprezentować położenie takiego kija za pomocą położenia jego środka geometrycznego, należy posłużyć się w tym celu jego *środkiem masy*, co ilustruje rysunek 19.3.

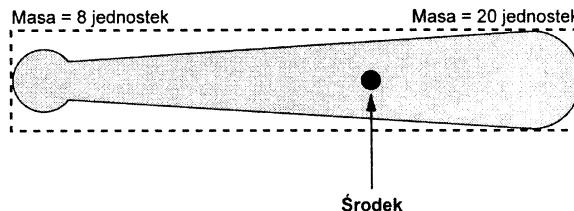
**Rysunek 19.2.**

Geometryczny  
środek kija



**Rysunek 19.3.**

Położenie środka  
masy kija może  
reprezentować  
położenie  
całego obiektu



Posługując się środkiem masy obiektu uzyskuje się w wielu przypadkach doskonalszą reprezentację jego fizycznych właściwości. Nie znaczy to jednak, że zawsze trzeba używać w grach środka masy obiektów. Jest to wskazane jedynie wtedy, gdy przewiduje się dość zaawansowane odwzorowanie fizyki rzeczywistego świata.

W jaki sposób można wyznaczyć środek masy obiektu? Ponieważ obiekty wirtualnego świata konstruowane są za pomocą wierzchołków, to z każdym wierzchołkiem można związać pewną wagę określającą wielkość masy dla tego wierzchołka. Aby wyznaczyć środek masy, sumuje się najpierw iloczyny współrzędnych wierzchołków i wagi wierzchołków, a następnie dzieli uzyskane sumy przez sumę wag wszystkich wierzchołków. Metodę tę ilustruje poniższy fragment kodu:

```

float centerOfMass[3];           // (x, y, z)
float object[NUM_VERTICES][3];   // wierzchołki obiektu
float objectMass[NUM_VERTICES];  // wagi wierzchołków
int i;                          // zmienna indeksu
float totalMass = 0.0;
...
centerOfMass[0] = 0.0;
centerOfMass[1] = 0.0;
centerOfMass[2] = 0.0;

// wyznacza sumę wag
for(i = 0; i < NUM_VERTICES; i++)
    totalMass += objectMass[i];

// wyznacza sumy iloczynów wag i poszczególnych współrzędnych
for(i = 0; i < NUM_VERTICES; i++)
{
    centerOfMass[0] += object[i][0] * objectMass[i];
    centerOfMass[1] += object[i][1] * objectMass[i];
    centerOfMass[2] += object[i][2] * objectMass[i];
}

// dzieli uzyskane sumy przez sumę wag wszystkich wierzchołków
centerOfMass[0] /= totalMass;
centerOfMass[1] /= totalMass;
centerOfMass[2] /= totalMass;

// środek masy opisany jest za pomocą współrzędnych (x, y, z):
// (centerOfMass[0], centerOfMass[1], centerOfMass[2])

```

Jak już wspomniano, korzystanie ze środka masy obiektów nie jest obowiązkowe i ma sens jedynie wtedy, gdy trzeba dodatkowo zwiększyć realizm odwzorowania rzeczywistego świata. Dla uproszczenia w omawianych przykładach wykorzystywane będą więc geometryczne środki obiektów.

## Prędkość

Prędkość określa odległość przebytą w jednostce czasu. Rozróżnia się dwa rodzaje prędkości:

- ◆ chwilową;
- ◆ średnią.

Prędkość chwilowa określa prędkość obiektu w danym momencie. Na przykład spoglądając na prędkościomierz samochodu jadącego z prędkością 20 kilometrów na godzinę, można powiedzieć, że tyle właśnie wynosi jego prędkość chwilowa.

Prędkość średnią wyznacza się w pewnym okresie czasu. W tym celu dzieli się odległość przebytą przez obiekt przez czas, którego potrzebował on na przebycie tej drogi. Jeśli drogę oznaczy się przez s, a czas przez t, to średnią prędkość v wyznaczyć będzie można za pomocą następującej zależności:

$$v = \Delta s / \Delta t = (s_k - s_p) / (t_k - t_p)$$

Jeśli więc obiekt przebył drogę 10 metrów w czasie jednej sekundy, to jego prędkość wynosić będzie:

$$v = \Delta s / \Delta t = 10 / 1 = 10 \text{ m/s}$$

W jaki sposób wykorzystać prędkość obiektów w grach? Niezależnie od tego, jak skomplikowane obliczenia fizyczne wykonuje się, wszystko sprowadza się do wyznaczenia położenia obiektu w kolejnej klatce. Znając prędkość obiektu wyznacza się je na podstawie wzoru:

$$x_k = x_p + v * \Delta t$$

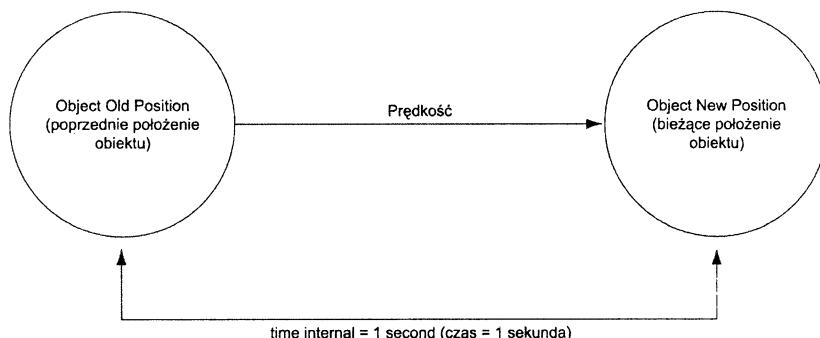
co oznacza, że

$$\text{nowe położenie} = \text{poprzednie położenie} + (\text{prędkość} * \text{zmiana czasu})$$

Obiekt rozpoczynający swój ruch w położeniu  $x_p$  z prędkością  $v$  znajdzie się po czasie  $\Delta t$  w położeniu  $x_k$ . Sposób wyznaczenia nowego położenia obiektu ilustruje szczegółowo rysunek 19.4.

**Rysunek 19.4.**

Wyznaczanie nowego położenia obiektu poruszającego się ze stałą prędkością



## Przyspieszenie

Omawiając ruch obiektów zakładano dotąd, że poruszają się one ze stałą prędkością. W świecie rzeczywistym ruch takiego praktycznie nie występuje. Przez *przyspieszenie* obiektu należy rozumieć zmianę jego prędkości w czasie.

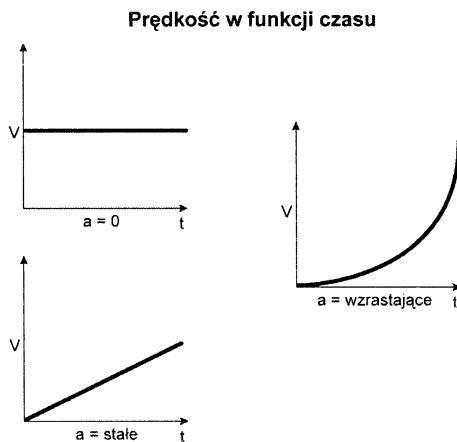
Pojęcie przyspieszenia ilustrują wykresy widoczne na rysunku 19.5. Pierwszy z nich opisuje ruch obiektu z zerowym przyspieszeniem, ponieważ prędkość ruchu nie zmienia się w czasie. Kolejny z wykresów reprezentuje ruch ze stałym przyspieszeniem, którego wartość reprezentuje nachylenie wykresu. Ostatni z wykresów prezentuje ruch obiektu, którego przyspieszenie ciągle wzrasta. Wartość przyspieszenia w każdym punkcie tego wykresu reprezentuje nachylenie stycznej do krzywej wykresu.

Przyspieszenie definiuje się jako wielkość zmiany prędkości w pewnym okresie czasu:

$$a = \Delta v / \Delta t = (v_k - v_p) / (t_k - t_p)$$

**Rysunek 19.5.**

*Wykresy zmiany prędkości obiektu w czasie dla różnych przypadków przyspieszenia*



Jednostką przyspieszenia jest metr na sekundę kwadrat ( $\text{m/s}^2$ ). Jeśli obiekt rozpoczyna ruch z prędkością  $v_0$ , a jego stałe przyspieszenie wynosi  $a$ , to po czasie  $\Delta t$  prędkość obiektu  $v_k$  wynosić będzie:

$$v_k = v_0 + a \cdot \Delta t$$

co znaczy, że

$$\text{nowa prędkość} = \text{poprzednia prędkość} + (\text{przyspieszenie} * \text{zmiana czasu})$$

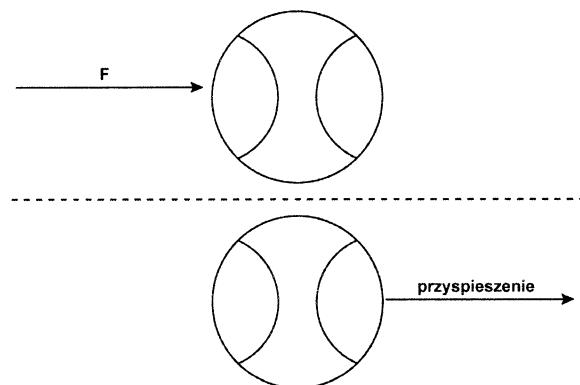
Nową wartość prędkości można podstawić do równania opisującego nowe położenie obiektu. W ten sposób uzyskać można podstawowy model ruchu obiektów, który można użyć w grze. Zanim to jednak nastąpi, trzeba poznać jeszcze kilka innych zagadnień.

**Siła**

Siła stanowi jedno z najważniejszych pojęć w fizyce. *Siła* nadaje przyspieszenie obiektem. Rzucając piłkę oddziałuje się na nią siłą, co pokazuje rysunek 19.6.

**Rysunek 19.6.**

*Piłka uzyskuje przyspieszenia na skutek oddziaływania siły*



Działanie sił opisują trzy zasady dynamiki Newtona stanowiące podstawę klasycznej fizyki.

## Pierwsza zasada dynamiki

Pierwsza zasada dynamiki brzmi następująco: „każde ciało znajduje się w ruchu jednostajnym, jeżeli nie oddziałuje na nie żadna siła”. Innymi słowy — obiekt pozostaje w stanie spoczynku lub porusza się ze stałą prędkością tak dugo, jak długo nie oddziałuje na niego żadna siła. Zasada ta nosi także nazwę *zasady bezwładności*.

Gdy umieści się poruszającą się piłkę w przestrzeni kosmicznej, czyli w idealnej próżni, to poruszać się ona będzie w nieskończoność ze stałą prędkością. W rzeczywistym świecie piłka taka zatrzyma się po pewnym czasie na skutek tarcia, oporu powietrza bądź oddziaływania grawitacji.

## Druga zasada dynamiki

Druga zasada dynamiki mówi, że „związek pomiędzy siłą, przyspieszeniem obiektu i jego masą wyraża się zależnością  $F = m * a$ ”. Równanie  $F = m * a$  informuje w istocie o tym, że na skutek działania siły zmienia się prędkość ruchu ciała.

Masa jest wielkością skalarną, a przyspieszenie wektorem, wobec czego także siła jest wektorem, którego zwrot i kierunek są takie same jak zwrot i kierunek wektora przyspieszenia. Jednostką siły jest  $\text{kg} \cdot \text{m/s}^2$ , czyli newton (N).

## Trzecia zasada dynamiki

Trzecia zasada dynamiki mówi, że „każdej akcji odpowiada reakcja o takiej samej wielkości i kierunku, lecz przeciwnym zwrocie”. Potwierdzenie tej zasady można łatwo zaobserwować w otaczającym świecie. Na przykład próbując wyważyć drzwi wywiera się na nie nacisk, a drzwi oddziałują z tą samą siłą. Podobnie jest, gdy wysiada się z łódki — następuje wtedy wzajemne oddziaływanie pasażera i łódki, na skutek czego oddala się ona od brzegu.

## Pęd

*Pędem* obiektu nazywa się iloczyn jego prędkości i masy:

$$\text{moment} = \text{masa} * \text{prędkość}$$

lub

$$p = m * v$$

Ponieważ równania opisujące siłę i pęd zawierają masę, to można je ze sobą powiązać. Z równania pędu otrzymujemy się to, że masa wyraża się przez poniższą zależność:

$$m = p / v$$

Podstawiając tę zależność do równania opisującego siłę otrzymujemy się:

$$F = m * a$$

$$F = (p / v) * a$$

$$F = (p * a) / v$$

Po zastąpieniu przyspieszenia a przez  $\Delta v / \Delta t$ , okaże się, że siła równa jest zmianie pędu w czasie:

$$F = \Delta p / \Delta t$$

Zmiana siły oddziałującej na obiekt zmienia więc także jego pęd. Można założyć na przykład, że piłka o masie 1 kilograma porusza się z prędkością 3 metrów na sekundę. Pęd jej jest więc równy:

$$p = m * v = 3 * 1 = 3 \text{ kg m/s}$$

Jeśli piłka ta zderzy się w czasie 1 sekundy ze ścianą, to zakładając, że jej prędkość, a tym samym i pęd po zderzeniu równy jest zero, to siłę oddziaływania piłki na ścianę (i zgodnie z trzecią zasadą dynamiki także siłę oddziaływania ściany na piłkę) wyznaczyć można w poniższy sposób:

$$F = \Delta p / \Delta t$$

$$F = (3 - 0) / (1 - 0)$$

$$F = 3 \text{ N}$$

## Zasada zachowania pędu

Przewaga opisu ruchu obiektów za pomocą momentu polega na zastosowaniu zasady zachowania pędu, która pozwala wyznaczać prędkość obiektów po ich zderzeniach.

Przy omawianiu zderzeń obiektów będzie się tu zakładać, że są one doskonale sprężyste. Na skutek doskonale sprężystego zderzenia piłki ze ścianą jej prędkość po zderzeniu będzie taka sama jak przed zderzeniem. Założenie takie stanowi uproszczenie zderzeń zachodzących w rzeczywistym świecie, które nie są doskonale sprężyste. Piłka po odbiciu od ściany w rzeczywistości będzie miała mniejszą prędkość niż przed zderzeniem na skutek utraty części energii przekształconej w ciepło na skutek tarcia bądź odkształcenia.

Zasada zachowania pędu omówiona zostanie na przykładzie zderzeń doskonale sprężystych zachodzących w jednym wymiarze.

Można przyjąć założenie, że zderzają się dwa obiekty A i B o różnej prędkości i masie. Ich prędkość po zderzeniu można wyznaczyć posługując się poniższym równaniem zachowania pędu:

$$m_a * v_{ap} + m_b * v_{bp} = m_a * v_{ak} + m_b * v_{bk}$$

Jak łatwo zauważyc, równanie to posiada dwie niewiadome:  $v_{ak}$  i  $v_{bk}$ . Dlatego, aby obliczyć prędkość po zderzeniu, trzeba zastosować jeszcze zasadę zachowania energii. Energię kinetyczną obiektu wyznacza się za pomocą poniższego równania:

$$E = [1/2] * m * v^2$$

Jednostką energii jest  $\text{kg} * \text{m}^2 / \text{s}^2$ , czyli dżul [J]. Sumaryczna energia kinetyczna obiektów uczestniczących w doskonale sprężystym zderzeniu zostaje zachowana:

$$E_{ap} + E_{bp} = E_{ak} + E_{bk}$$

$$[1/2] * m_a * v_{ap}^2 + [1/2] * m_b * v_{bp}^2 = [1/2] * m_a * v_{ak}^2 + [1/2] * m_b * v_{bk}^2$$

Łącząc zasadę zachowania pędu z zasadą zachowania energii otrzymuje się po prostych przekształceniach zależności opisujące prędkości obu obiektów po zderzeniu:

$$V_{ak} = (2*m_b * V_{bp} + v_{ap} * (m_a - m_b)) / (m_a + m_b)$$

$$V_{bk} = (2*m_a * V_{ap} + V_{bp} * (m_a - m_b)) / (m_a + m_b)$$

Zależności te przydadzą się później podczas tworzenia symulacji zderzeń. Teraz omówiona zostanie siła tarcia.

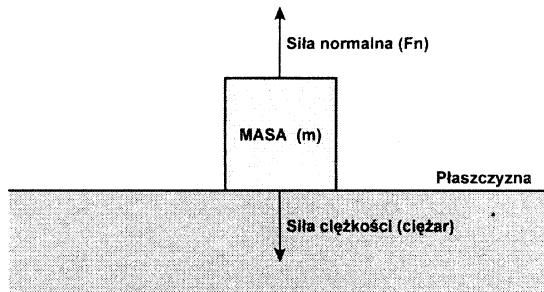
## Tarcie

*Tarcie* jest najpowszechniejszą przyczyną utraty energii przez obiekty w otaczającym świecie. Gdyby jednak nie było tarcia, nie można by nawet utrzymać w ręce tej książki!

Tarcie przedstawia się jako siłę działającą na obiekt w kierunku przeciwnym do jego ruchu. Rysunek 19.7 przedstawia podstawowy model tarcia na płaskiej powierzchni.

Rysunek 19.7.

Podstawowy  
model tarcia



Rozróżnia się dwa podstawowe rodzaje tarcia:

- ◆ **tarcie statyczne** — oddziałuje ono na obiekt, który się nie porusza; wielkość tarcia statycznego informuje o sile, którą trzeba przezwyciężyć, aby wprawić obiekt w ruch;
- ◆ **tarcie kinetyczne** — oddziałuje ono na obiekt, który się porusza; wielkość tarcia kinetycznego informuje o sile, którą trzeba przezwyciężyć, aby obiekt nadal się poruszał.

Zanim najdziejdzie czas analizy tarcia oddziałującego na obiekty na płaszczyźnie, trzeba jeszcze wprowadzić pojęcie *siły normalnej* do powierzchni. Siła ta oddziałuje na każde ciało spoczywające na powierzchni i jest zawsze prostopadła do jej płaszczyzny. Siłę tę wyznacza się posługując się równaniem:

$$F_N = -m * g$$

gdzie m jest masą obiektu, g przyspieszeniem ziemskim równym  $9,81 \text{ m/s}^2$ . Iloczyn m \* g nazywa się *cięzarem* obiektu.

## Tarcie na płaszczyźnie

Na obiekt poruszający się po płaszczyźnie oddziałuje tarcie, które w końcu powoduje zatrzymanie obiektu. W przypadku obiektów nieruchomych tarcie informuje o sile, którą

trzeba przewyściężyć, aby wprawić obiekt w ruch. Siłę tarcia  $F_t$  wyznacza się w tym przypadku za pomocą poniższego równania:

$$F_t = \mu_s * F_N$$

gdzie  $F_N$  jest siłą normalną do płaszczyzny, a  $\mu_s$  jest statycznym współczynnikiem tarcia.

Współczynnik ten zależy od rodzaju powierzchni i może wynosić od 0,001 dla bardzo gładkich powierzchni do 0,9 dla powierzchni chropowatych.

Siłę tarcia oddziałującego na poruszający się obiekt wyznacza się za pomocą poniższego równania:

$$F_t = \mu_k * F_N$$

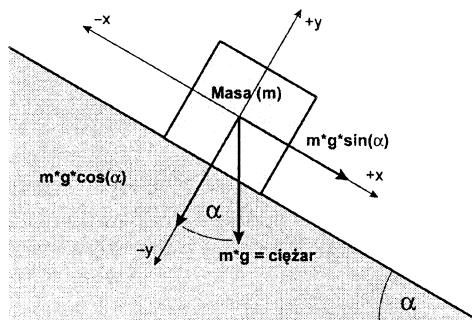
Jak widać, jedyną różnicą jest obecność współczynnika tarcia kinetycznego zamiast współczynnika tarcia statycznego. Dla tej samej powierzchni wartość współczynnika tarcia kinetycznego jest nieco mniejsza niż wartość współczynnika tarcia statycznego.

## Tarcie na równi pochyłej

W przypadku ciała poruszającego się po równi pochyłej wyznaczenie siły tarcia nieco się komplikuje. Sytuację tę ilustruje rysunek 19.8.

**Rysunek 19.8.**

Tarcie  
na równi pochyłej



Siła grawitacji oddziaływająca na obiekt powoduje jego ruch w dół równi pochyłej. Powierzchnia równi posiada określone wartości współczynników tarcia statycznego  $\mu_s$  i kinetycznego  $\mu_k$ . Na obiekt oddziałuje też siła normalna do powierzchni równi pochyłej.

Opisując ruch obiektu po równi pochyłej zmienić trzeba nieco orientację układu współrzędnych tak, by osь x była równoległa do równi pochyłej.

Aby wyznaczyć siłę tarcia oddziałującą na obiekt na równi pochyłej, trzeba najpierw wyznaczyć sumy sił oddziałujących na obiekt wzdłuż każdej z osi układu współrzędnych. Wzdłuż osi x na obiekt działa składowa siły grawitacji równa

$$F_g = m * g * \sin(\alpha)$$

Wzdłuż tej samej osi — ale w przeciwnym kierunku — działa siła tarcia statycznego równa

$$F_t = - F_N * \mu_s$$

W stanie równowagi suma sił działających wzdłuż osi x równa jest zero:

$$\sigma F_x = m * g * \sin(\alpha) - F_N * \mu_s = 0$$

W podobny sposób wyznaczyć można siły działające wzdłuż osi y. Składową grawitacji działającą wzdłuż tej osi przedstawia poniższe równanie:

$$F_g = m * g * \cos(\alpha)$$

W kierunku przeciwnym działa siła normalna do równi pochyłej:

$$F_t = F_N$$

Suma sił działających wzdłuż osi y wynosi więc:

$$\sigma F_y = F_N - m * g * \cos(\alpha)$$

Ponieważ suma ta musi być równa zero, to siła normalna do równi musi być równa składowej grawitacji działającej wzdłuż osi y:

$$F_N = m * g * \cos(\alpha)$$

Stąd otrzymuje się zależność:

$$m * g * \cos(\alpha) * \mu_s = m * g * \sin(\alpha)$$

A po uproszczeniu

$$\mu_s = \sin(\alpha) / \cos(\alpha) = \tan(\alpha)$$

Po dalszych przekształceniach uzyskuje się wyrażenie na krytyczną wartość kąta  $\alpha$ :

$$\alpha_k = \arctan(\mu_s)$$

Wartość ta określa kąt nachylenia równi, przy którym obiekt zacznie się z niej zsuwać.

Do czego mogą przydać się te równanie podczas tworzenia gry? Można na przykład znaleźć za ich pomocą wypadkową siłę działającą na obiekt zsuwający się po równi pochylnej:

$$F = m * g * \sin(\alpha) - m * g * \cos(\alpha) * \mu_k$$

A następnie wyznaczyć wartość przyspieszenia:

$$a = g * (\sin(\alpha) - \mu_k * \cos(\alpha))$$

Należy pamiętać, że przyspieszenie to wyznaczone jest w układzie współrzędnych, w którym osz x jest równoległa do równi. Aby uzyskać składowe przyspieszenia w tradycyjnym układzie współrzędnych, trzeba pomnożyć powyższe równanie przez odpowiednie funkcje trygonometryczne kąta nachylenia równi:

$$a_x = \cos(\alpha) * g * (\sin(\alpha) - \mu_k * \cos(\alpha))$$

$$a_y = \sin(\alpha) * g * (\sin(\alpha) - \mu_k * \cos(\alpha))$$

Powyższe równania można użyć do wyznaczenia prędkości obiektu, a następnie jego położenia w kolejnych klatkach grafiki.

# Modelowanie świata rzeczywistego

Pozostała część rozdziału poświęcona zostanie przedstawieniu jednego ze sposobów modelowania świata rzeczywistego w tworzonych grach.

Zanim będzie można przystąpić do omawiania szczegółów, trzeba poinformować o przyjętych tu założeniach dotyczących sposobu prezentacji tego materiału. Do modelowania świata rzeczywistego zastosowane zostaną techniki projektowania obiektowego. Zakłada się więc znajomość języka programowania C++ oraz programowania obiektowego.

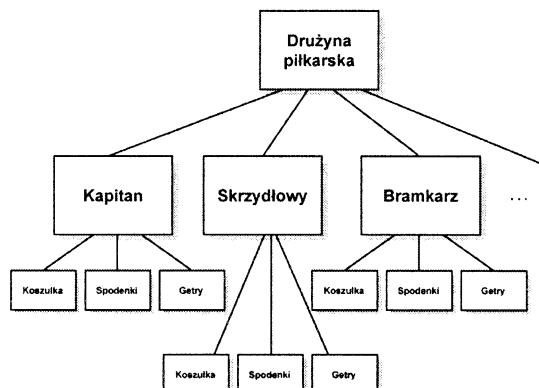
W prezentowanym materiale pojawi się sporo informacji z dziedziny matematyki, których nie będzie się tu szczegółowo wyjaśniać. Jeśli zrozumienie tych fragmentów okaże się zbyt trudne, to konieczne będzie zapoznanie się z informacjami o stronach WWW zamieszczonymi w dodatku A.

Zaprezentowane poniżej rozwiązania nie zawsze są optymalne, ale głównym celem było tu przedstawienie takich technik, które są jednocześnie proste i stosunkowo uniwersalne.

## Rozkład problemu

Projektując pewien system zawsze trzeba najpierw dokonać jego rozkładu na możliwie niewielkie komponenty funkcjonalne i określić sposób ich współpracy. Zagadnienie to zilustrowane zostanie na przykładzie drużyny piłkarskiej. W jej skład wchodzą zawodnicy, którzy wykonują na boisku różne zadania. Kapitan, skrzydłowy, bramkarz, skrzydłowy i tak dalej. Każdy z zawodników posiada strój składający się z pewnych elementów, takich jak koszulka, spodenki czy getry. Drużynę piłkarską można więc modelować za pomocą hierarchii obiektów przedstawionej na rysunku 19.9.

**Rysunek 19.9.**  
Modelowanie  
drużyny piłkarskiej  
metodą rozkładu  
na komponenty



Tworząc model świata gry trzeba dokonać jego rozkładu na prostsze elementy, które z kolei rozkłada się na jeszcze prostsze składniki aż do momentu uzyskania podstawowych obiektów. Każdy z tych podstawowych składników grafiki można następnie wyrazić za pomocą wektorów, płaszczyzn i obiektów. Zanim omówione zostaną one szczegółowo, trzeba jednak najpierw przedstawić niezwykle istotny składnik wirtualnego świata, jakim jest czas.

## Czas

Ponieważ czas jest jednym z najważniejszych atrybutów opisujących rzeczywistość, to do jego modelowania powinno się przyłożyć dużą uwagę. Wspomniano już o dwóch sposobach modelowania czasu: odmierzaniu czasu za pomocą kolejnych klatek grafiki bądź wykorzystaniu czasu rzeczywistego. Rozwiązanie przedstawione w tym rozdziale używać będzie czasu rzeczywistego.

Czas odmierzać się będzie za pomocą *licznika czasu*. Umożliwi to realistyczne modelowanie fizyki obiektów, a także uniezależni model od prędkości tworzenia klatek grafiki.

Do wyboru są dwa typy takich liczników: licznik o dużej dokładności oraz licznik systemu Windows wysyłający komunikaty WM\_TIMER. Zastosowany tu zostanie pierwszy z nich, ponieważ jest on dokładniejszy i szybszy w działaniu, co w przypadku gier nie jest bez znaczenia.

Korzystając z licznika o dużej dokładności trzeba najpierw ustalić jego rozdzielczość, a następnie pobrać bieżący czas i przechować go w zmiennej jako czas rozpoczęcia działania aplikacji. Zadania te wykonać można posługując się funkcjami QueryPerformanceFrequency() oraz QueryPerformanceCounter() zdefiniowanymi w następujący sposób:

```
BOOL QueryPerformanceFrequency(LARGE_INTEGER *lpFrequency);
BOOL QueryPerformanceCounter(LARGE_INTEGER *lpPerformanceCount);
```

Funkcja QueryPerformanceFrequency() umożliwia określenie rozdzielczości licznika czasu w taktach na sekundę, a funkcja QueryPerformanceCounter() jego bieżącej wartości.

Typem parametru obu funkcji jest LARGE\_INTEGER. Typ ten zdefiniowany jest za pomocą struktury umożliwiającej przechowywanie 64-bitowych liczb całkowitych:

```
typedef union _LARGE_INTEGER
{
    struct
    {
        DWORD LowPart;
        DWORD HighPart;
    };
    LONGLONG QuadPart;
} LARGE_INTEGER;
```

Struktura ta jest w rzeczywistości unią. Jeśli kompilator nie umożliwia reprezentacji 64-bitowych liczb całkowitych, to trzeba wtedy używać jej pól LowPart i HighPart. W przeciwnym razie korzysta się z pola QuadPart typu LONGLONG (64-bity ze znakiem).

Zadeklarowane teraz zostaną dwie zmienne typu LARGE\_INTEGER, które posłużą do pobrania rozdzielczości i bieżącej wartości licznika czasu:

```
LARGE_INTEGER ticksPerSecond;
LARGE_INTEGER startTime;
```

Zmienne te przekazać można jako parametry wspomnianych funkcji:

```
if (!QueryPerformanceFrequency(&m_ticksPerSecond))
{
    // licznik czasu o dużej rozdzielczości nie jest dostępny
    return false;
}
else
{
    QueryPerformanceCounter(&m_startTime);
    return true;
}
```

Oprócz odmierzania czasu gry osobny licznik czasu zastosuje się także w celu określania tempa tworzenia klatek grafiki. Do tworzenia liczników czasu posłuży klasa CHiResTimer, którą zdefiniować można w pliku nagłówkowym *HiResTimer.h* w następujący sposób:

```
#ifndef __TIMER_H_INCLUDED__
#define __TIMER_H_INCLUDED__

#include <windows.h>

class CHiResTimer
{
public:
    CHiResTimer() {}
    ~CHiResTimer() {}

/*****************/
Init()

Jeśli dostępny jest licznik czasu o dużej dokładności, to funkcja Init() zapamiętuje jego rozdzielczość i zwraca wartość true. W przeciwnym razie zwraca wartość false i wtedy licznik nie może być używany.
*****************/
bool Init()
{
    if (!QueryPerformanceFrequency(&m_ticksPerSecond))
    {
        // system nie posiada licznika czasu o dużej dokładności
        return false;
    }
    else
    {
        QueryPerformanceCounter(&m_startTime);
        return true;
    }
} // Init()

float GetElapsedSeconds(unsigned long elapsedFrames = 1)
{
    static LARGE_INTEGER s_lastTime = m_startTime;
    LARGE_INTEGER currentTime;
    QueryPerformanceCounter(&currentTime);
```

```
float seconds = ((float)currentTime.QuadPart - (float)s_lastTime.QuadPart) /  
    (float)m_ticksPerSecond.QuadPart;  
  
// "zeruje" licznik czasu  
s_lastTime = currentTime;  
  
return seconds;  
} // GetElapsedSeconds()
```

```
*****  
GetFPS()
```

Zwraca średnią liczbę klatek w okresie czasu elapsedFrames (domyślnie równym 1). Jeśli metoda nie jest wywoływana podczas tworzenia każdej ramki, to należy samemu zliczać liczbę ramek oraz zerować ją po wywołaniu metody.

```
*****  
float GetFPS(unsigned long elapsedFrames = 1)  
{  
    static LARGE_INTEGER s_lastTime = m_startTime;  
    LARGE_INTEGER currentTime;  
  
    QueryPerformanceCounter(&currentTime);  
  
    float fps = (float)elapsedFrames * (float)m_ticksPerSecond.QuadPart /  
        ((float)currentTime.QuadPart -  
         (float)s_lastTime.QuadPart);  
  
    // "zeruje" licznik czasu  
    s_lastTime = currentTime;  
  
    return fps;  
} // GetFPS
```

```
*****  
LockFPS()
```

Blokuje wykonanie programu tak, by prędkość tworzenia klatek nie wzrosła ponad wartość określona przez parametr metody. Metoda ta nie zapewnia utrzymania stałej prędkości tworzenia klatek, a jedynie zapobiega jej wzrostowi ponad podaną wartość. Natomiast prędkość ta w praktyce może spaść poniżej tej wartości. Zakłada się, że metoda ta wywoływana jest podczas tworzenia każdej klatki. Metoda zwraca chwilową wartość prędkości tworzenie klatek (w klatkach na sekundę) <= targetFPS.

```
*****  
float LockFPS(unsigned char targetFPS)  
{  
    if (targetFPS == 0)  
        targetFPS = 1;  
  
    static LARGE_INTEGER s_lastTime = m_startTime;  
    LARGE_INTEGER currentTime;  
    float    fps;
```

```

// oczekuje w pętli okres czasu, by osiągnąć zadaną prędkość
do {
    QueryPerformanceCounter(&currentTime);
    fps = (float)m_ticksPerSecond.QuadPart/
        ((float)(currentTime.QuadPart - s_lastTime.QuadPart));
} while (fps > (float)targetFPS);

// "zeruje" licznik czasu
s_lastTime = m_startTime;

return fps;
} // LockFPS()

private:
    LARGE_INTEGER    m_startTime;
    LARGE_INTEGER    m_ticksPerSecond;
};

#endif // __TIMER_H_INCLUDED__

```

Metoda `Init()` inicjuje licznik czasu, a metoda `GetElapsedSeconds()` zwraca liczbę sekund, które upłynęły od poprzedniego jej wywołania. Za pomocą metody `GetFPS()` używa się bieżącą prędkość tworzenia klatek grafiki, a metoda `LockFPS()` umożliwia ograniczenie tej prędkości „z góry”.

W jaki sposób należy posługiwać się zdefiniowaną klasą? Najpierw trzeba zadeklarować odpowiednią zmienną instancji:

```

#include "HiResTimer.h"

CHiResTimer *timer = NULL;

```

Następnie podczas inicjacji gry trzeba będzie utworzyć obiekt licznika czasu i zainicjować go za pomocą metody `CHiResTimer::Init()`:

```

timer = new CHiResTimer();
if (!timer->Init())
{
    // inicjacja licznika czasu nie powiodła się
}
else
{
    // licznik został prawidłowo zainicjowany
}

```

Pozostałe metody można stosować na wiele różnych sposobów. Tutaj należy przyjrzeć się bliżej jedynie sposobowi korzystania z metody `CHiResTimer::GetElapsedTime()`, która umożliwia odmierzanie czasu gry.

Metodę tę będzie się wywoływać tworząc każdą klatkę grafiki i wyznaczając nowe położenie rysowanych obiektów, co ilustruje rysunek 19.10.

**Rysunek 19.10.**  
Główna pętla gry  
wykorzystująca  
licznik czasu



Wywołując metodę CHiResTimer::GetElapsedTime() przekazuje się jej liczbę klatek, które zostały utworzone od poprzedniego jej wywołania, a zwrotną wartość przechowuje się w zmiennej o nazwie `timeElapsed`:

```

float timeElapsed; // czas, który upłynął od poprzedniego wywołania (poprzedniej klatki)
...
timeElapsed = timer->GetElapsedSeconds(1); // utworzono 1 klatkę
// obliczenie położenia obiektu wynikające ze sposobu jego ruchu
// np. DoPhysics(timeElapsed);
// narysowanie obiektu
  
```

## Wektor

Kolejną niezbędną klasą będzie klasa reprezentująca wektor. O wektorach pisano już dość szczegółowo omawiając podstawy grafiki trójwymiarowej. Teraz należy rozszerzyć to omówienie o kolejne operacje na wektorach i nowe sposoby wykorzystania wektorów do modelowania rzeczywistego świata.

Przypomnieć należy, że wektor posiada długość, kierunek oraz zwrot. Jego reprezentacji można użyć jako typ danych dla różnych wielkości fizycznych. Za pomocą wektora łączącego dany punkt z początkiem układu współrzędnych można także definiować położenie tego punktu w przestrzeni, co ilustruje rysunek 19.11.

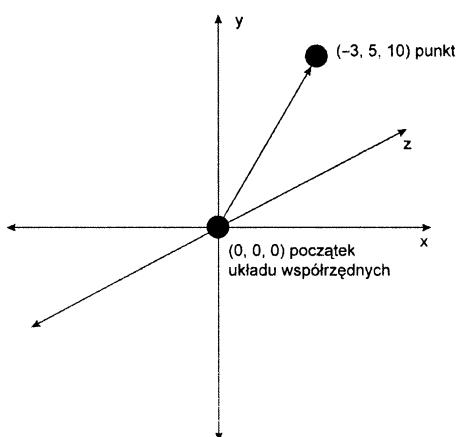
Operacje wykonywane za pomocą wektorów mogą obejmować dodawanie i odejmowanie wektorów, mnożenie i dzielenie wektora przez skalar, normalizację wektora, iloczyn skalarny i wektorowy, obliczanie długości wektora oraz tworzenie odbicia wektora względem normalnej. Klasa CVector, która zostanie zaimplementowana, będzie posiadać metody umożliwiające wykonywanie wymienionych oraz innych operacji na wektorach.

Najpierw jednak zdefiniować trzeba typ `scalar_t`, który posłuży do reprezentacji skalarów:

```
typedef float scalar_t;
```

**Rysunek 19.11.**

Punkt przestrzeni  
można określić  
za pomocą wektora



Typ ten wykorzystywany jest przez definicję klasy CVector, która została umieszczona w pliku *vector.h*:

```
#ifndef __VECTOR_H
#define __VECTOR_H

#include <math.h>

typedef float scalar_t; // skalar

// klasa CVector
// specjalne podziękowania dla Basa Kuenena za koncepcje wykorzystania operatorów

class CVector
{
public:
    scalar_t x;
    scalar_t y;
    scalar_t z; // składowe x,y,z

public:
    // konstruktor
    CVector(scalar_t a = 0, scalar_t b = 0, scalar_t c = 0) : x(a), y(b), z(c) {}
    CVector(const CVector &vec) : x(vec.x), y(vec.y), z(vec.z) {}

    // podstawienie wektora
    const CVector &operator=(const CVector &vec)
    {
        x = vec.x;
        y = vec.y;
        z = vec.z;

        return *this;
    }

    // równość wektorów
    const bool operator==(const CVector &vec) const
    {
        return (x == vec.x) && (y == vec.y) && (z == vec.z);
    }
}
```

```
{  
    return ((x == vec.x) && (y == vec.y) && (z == vec.z));  
}  
  
// nierówność wektorów  
const bool operator!=(const CVector &vec) const  
{  
    return !(*this == vec);  
}  
  
// dodawanie wektorów  
const CVector operator+(const CVector &vec) const  
{  
    return CVector(x + vec.x, y + vec.y, z + vec.z);  
}  
  
// dodawanie wektorów  
const CVector operator+() const  
{  
    return CVector(*this);  
}  
  
// dodawanie wektorów  
const CVector& operator+=(const CVector& vec)  
{  
    x += vec.x;  
    y += vec.y;  
    z += vec.z;  
    return *this;  
}  
  
// odejmowanie wektorów  
const CVector operator-(const CVector& vec) const  
{  
    return CVector(x - vec.x, y - vec.y, z - vec.z);  
}  
  
// odejmowanie wektorów  
const CVector operator-() const  
{  
    return CVector(-x, -y, -z);  
}  
  
// odejmowanie wektorów  
const CVector &operator-=(const CVector& vec)  
{  
    x -= vec.x;  
    y -= vec.y;  
    z -= vec.z;  
  
    return *this;  
}  
  
// mnożenie wektora przez skalar  
const CVector &operator*=(const scalar_t &s)  
{  
    x *= s;  
    y *= s;  
    z *= s;
```

```
        return *this;
    }

    // dzielenie wektora przez skalar
    const CVector &operator/(const scalar_t &s)
    {
        const float recip = 1/s; // dla efektywności

        x *= recip;
        y *= recip;
        z *= recip;

        return *this;
    }

    // mnożenie wektora przez skalar
    const CVector operator*(const scalar_t &s) const
    {
        return CVector(x*s, y*s, z*s);
    }

    // mnożenie wektora przez skalar
    friend inline const CVector operator*(const scalar_t &s, const CVector &vec)
    {
        return vec*s;
    }

/* friend inline const CVector operator*(const CVector &vec, const scalar_t &s)
{
    return CVector(vec.x*s, vec.y*s, vec.z*s);
}

*/ // dzielenie wektora przez skalar
const CVector operator/(scalar_t s) const
{
    s = 1/s;

    return CVector(s*x, s*y, s*z);
}

// iloczyn wektorowy
const CVector CrossProduct(const CVector &vec) const
{
    return CVector(y*vec.z - z*vec.y, z*vec.x - x*vec.z, x*vec.y - y*vec.x);
}

// iloczyn wektorowy
const CVector operator^(const CVector &vec) const
{
    return CVector(y*vec.z - z*vec.y, z*vec.x - x*vec.z, x*vec.y - y*vec.x);
}

// iloczyn skalarny
const scalar_t DotProduct(const CVector &vec) const
{
    return x*vec.x + y*vec.y + z*vec.z;
}
```

```
// iloczyn skalarny
const scalar_t operator%(const CVector &vec) const
{
    return x*vec.x + y*vec.y + z*vec.z;
}

// długość wektora
const scalar_t Length() const
{
    return (scalar_t)sqrt((double)(x*x + y*y + z*z));
}

// wektor jednostkowy
const CVector UnitVector() const
{
    return (*this) / Length();
}

// normalizacja wektora
void Normalize()
{
    (*this) /= Length();
}

// operator modułu (długości) wektora
const scalar_t operator!() const
{
    return sqrtf(x*x + y*y + z*z);
}

// zmienia długość wektora
const CVector operator | (const scalar_t length) const
{
    return *this * (length / !(*this));
}

// zmienia długość wektora
const CVector& operator |= (const float length)
{
    return *this = *this | length;
}

// zwraca kąt, który tworzą wektory
const float inline Angle(const CVector& normal) const
{
    return acosf(*this % normal);
}

// tworzy odbicie wektora względem powierzchni zdefiniowanej przez normalną
const CVector inline Reflection(const CVector& normal) const
{
    const CVector vec(*this | 1);      // normalizuje wektor
    return (vec - normal * 2.0 * (vec % normal)) * !*this;
}

};

#endif
```

Korzystając z możliwości przeciążenia operatorów języka C++ można uczynić zapis operacji na wektorach bardziej naturalnym i czytelnym. Na przykład zamiast dodawać wektory w następujący sposób:

```
CVector a, b;
CVector result;
...
result->x = a->x + b->x; // dodawanie składowej x
result->y = a->y + b->y; // dodawanie składowej y
result->z = a->z + b->z; // dodawanie składowej z
```

operacje tę można wykonać w poniższy sposób:

```
CVector a, b;
CVector result;
...
result = a + b; // dodawanie dwóch wektorów
```

A oto przykłady wykorzystania innych metod klasy CVector:

```
CVector a, b;
CVector result;
result = a ^ b; // iloczyn wektorowy
result = a->CrossProduct(b); // jak wyżej
result = a % b; // iloczyn skalarny
a |= 10; // długość wektora a będzie wynosić 10
b->Normalize(); // normalizuje wektor b
result = a * 2.0; // skala wektor
```

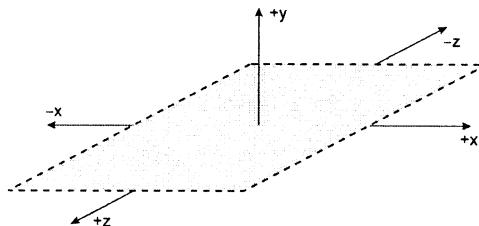
Klasa CVector będzie wykorzystywana w obliczeniach kinematyki obiektów do opisu ich położenia i wykrywania zderzeń.

## Płaszczyzna

Z lekcji geometrii wszyscy pamiętają, że płaszczyzna rozciąga się w nieskończoność w dwóch wymiarach, co ilustruje rysunek 19.12. W grach płaszczyzna może reprezentować granicę pewnego obszaru, być częścią obiektu lub opisywać wielokąt.

**Rysunek 19.12.**

Płaszczyzna jest nieskończona w swoich dwóch wymiarach



Płaszczyznę można opisać przez następujące równanie:

$$A*x + B*y + C*z - D = 0$$

Współczynniki ( $A, B, C$ ) definiują wektor normalny do płaszczyzny, a zmienne ( $x, y, z$ ) punkt na płaszczyźnie. Wartość  $D$  jest skalarem zwany *stałą przesunięcia płaszczyzny* i określa odległość płaszczyzny od początku układu współrzędnych. Równanie to

właściwie wyczerpuje potrzebną tu wiedzę o płaszczyznach, ale warto dokładniej zrozumieć jego sens i pochodzenie.

Można przekształcić je zatem do następującej postaci:

$$A*x + B*y + C*z = D$$

Czy lewa strona równania nie wygląda znajomo? Jeśli jeszcze nie, to pomocne będzie poniższe równanie:

$$A \cdot B = A.x*B.x + A.y*B.y + A.z*B.z$$

Jak łatwo teraz zauważać, równanie płaszczyzny stanowi iloczyn skalarny wektora normalnego do płaszczyzny i dowolnego punktu na płaszczyźnie, który można przedstawić za pomocą wektora łączącego początek układu współrzędnych z tym punktem. Równanie płaszczyzny można więc zapisać także w następujący sposób:

$$A \cdot N = (A_x, A_y, A_z) \cdot (N_x, N_y, N_z)$$

$$A \cdot N = A_x * N_x + A_y * N_y + A_z * N_z$$

Porównując te równania z początkowym równaniem płaszczyzny otrzymuje się następującą odpowiedniość:

$$(A, B, C) = (N_x, N_y, N_z)$$

$$(x, y, z) = (A_x, A_y, A_z)$$

Po podstawieniu uzyskać można:

$$A*x + B*y + C*z$$

Obliczając iloczyn wektora normalnego do płaszczyzny i dowolnego punktu tej płaszczyzny uzyskuje się stałą przesunięcia płaszczyzny:

$$A*x + B*y + C*z = D$$

i w rezultacie równanie płaszczyzny:

$$A*x + B*y + C*z - D = 0$$

Podobnie jak w przypadku wektorów dla płaszczyzn zaimplementować trzeba odpowiednią klasę o nazwie `CPlane`. Klasa ta zawierać będzie szereg metod pozwalających na przykład sprawdzić to, czy dany punkt należy do płaszczyzny, wyznaczyć odległość punktu od płaszczyzny, znaleźć punkt przecięcia promienia z płaszczyzną.

Aby obliczyć odległość punktu od płaszczyzny, trzeba zsumować wynik iloczynu skalarnego wektora normalnego do płaszczyzny i wektora punktu ze stałą przesunięcia płaszczyzny. Jeśli okaże się, że odległość ta równa jest zeru, oznacza to, że punkt leży na płaszczyźnie. Aby znaleźć punkt przecięcia płaszczyzny przez promień, trzeba znać jego początek i kierunek. Następnie wyznacza się iloczyn skalarny wektora normalnego do płaszczyzny i wektora promienia. Jeśli w wyniku otrzymane zostanie zero, to oznacza to, że promień jest równoległy do płaszczyzny i punkt przecięcia nie istnieje. W pozostałych przypadkach oblicza się odległość punktu początkowego promienia od płaszczyzny. Odległość tę dzieli się przez iloczyn skalarny wektora normalnego do płaszczyzny i wektora promienia. Uzyskany wynik mnoży się przez wektor promienia, a wynik mnożenia odejmuje się od wektora punktu początkowego promienia. A oto implementacja klasy `CPlane`, która została umieszczona w pliku nagłówkowym `plane.h`:

```
#ifndef __PLANE_H
#define __PLANE_H

#include "vector.h"

class CPlane
{
public:
    CVector N;          // wektor normalny do płaszczyzny
    double D;           // stała przesunięcia płaszczyzny

    // konstruktory

    // Ax + By + Cz - D = 0
    CPlane(double a = 1, double b = 0, double c = 0, double d = 0)
    {
        N = CVector(a, b, c);
        D = d;
    }

    // tworzy obiekt płaszczyzny na podstawie wektora normalnego
    // i stałej przesunięcia płaszczyzny D=d
    CPlane(const CVector& normal, double d = 0)
    {
        N = normal;
        D = d;
    }

    // tworzy kopię obiektu płaszczyzny plane
    CPlane(const CPlane& plane)
    {
        N = plane.N;
        D = plane.D;
    }

    // tworzy obiekt płaszczyzny na podstawie współrzędnych trzech jej punktów
    CPlane(const CVector& vertexA, const CVector& vertexB, const CVector& vertexC)
    {
        CVector normalA((vertexC - vertexA) | 1.0); // wektor normalny jednostkowy C - A
        CVector normalB((vertexC - vertexB) | 1.0); // wektor normalny jednostkowy C - B

        N = (normalA ^ normalB) | 1.0;                // normalizuje iloczyn skalarny
        D = -vertexA % N;                            // wyznacza odległość
    }

    // operator podstawienia
    const CPlane& operator=(const CPlane& plane)
    {
        N = plane.N;
        D = plane.D;

        return *this;
    }

    // operator porównania
    const bool operator==(const CPlane& plane) const
    {
        return N == plane.N && D == plane.D;
    }
}
```

```

// operator porównania
const bool operator!=(const CPlane& plane) const
{
    return !(*this == plane);
}

// sprawdza, czy punkt point leży na płaszczyźnie
const bool inline PointOnPlane(const CVector& point) const
{
    return DistanceToPlane(point) == 0;
}

// zwraca odległość punktu od płaszczyzny
const double inline DistanceToPlane(const CVector& point) const
{
    return N % point + D;
}

// zwraca punkt przecięcia płaszczyzny przez promień
const CVector inline RayIntersection(const CVector& rayPos,
                                      const CVector& rayDir) const
{
    const double a = N % rayDir;
    if (a == 0)
        return rayPos; // promień jest równoległy do płaszczyzny
    return rayPos - rayDir * (DistanceToPlane(rayPos) / a);
};

#endif // __PLANE_H

```

A oto kilka przykładów użycia klasy CPlane:

```

#include "plane.h"
...
CPlane *plane1 = NULL;
CPlane *plane2 = NULL;
...
plane1 = new CPlane(0.0, 1.0, 0.0, 5.0);
plane2 = new CPlane(plane1); // kopia płaszczyzny plane1
plane1 = plane2; // podstawienie wskaźnika
if (plane1->PointOnPlane(CVector(0.0, 5.0, 0.0))) // sprawdza, czy punkt (0,5,0) należy
do płaszczyzny
{
    // punkt leży na płaszczyźnie

```

Zastosowania klasy CPlane omówione zostaną szerzej przy przedstawieniu sposobów wykrywania zderzeń obiektów.

## Obiekt

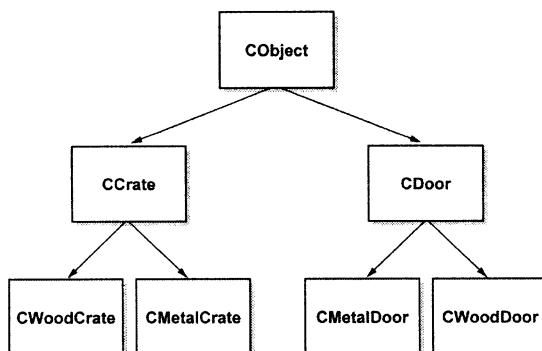
Najbardziej podstawowym bytem modelowanego świata jest obiekt. Obiektami są ściany, pomieszczenia, gracze, postacie, kamery, źródła światła i wszystko inne, co tylko można rozpoznać jako samodzielny byt świata gry. Obiekty mogą się przemieszczać, można na nie oddziaływać, a one mogą oddziaływać także na siebie wzajemnie. Wszystko, co można zaobserwować jest obiektem.

Klasyfikując obiekty dostrzec można, że posiadają one zestaw wspólnych atrybutów. Liczba i rodzaj tych atrybutów zależy od obiektów występujących w grze, ale zwykle zawierać będą one atrybuty opisujące ruch obiektów i sposoby oddziaływanego pomiędzy nimi. Tworząc podstawową klasę CObject umieścić w niej trzeba atrybuty opisujące położenie, prędkość i przyspieszenie obiektu. Można także uzupełnić je o promień sfery ograniczającej obiekt i służącej do wykrywania zderzeń z innymi obiektami (więcej na ten temat wkrótce). Atrybuty podstawowej klasy obiektów nie umożliwiają pełnego opisu dowolnego obiektu. Jeśli programista będzie próbował reprezentować za pomocą obiektu klasy CObject na przykład skrzynię, to zabraknie mu atrybutów opisujących jej rozmiary. Oczywiście prawidłowe rozwiązanie tego problemu nie będzie polegać na umieszczeniu w klasie CObject atrybutów pozwalających opisać dowolny obiekt.

Klasa CObject zostanie klasą bazową hierarchii klas opisujących różne kategorie obiektów. Korzystając z dziedziczenia w języku C++ utworzyć można na przykład jej klasy pochodne CCrate i CDoor reprezentujące (odpowiednio) skrzynie i drzwi, co ilustruje rysunek 19.13. Klasy pochodne będą zawierać wszystkie atrybuty klasy CObject oraz atrybuty specyficzne dla danej klasy. W przypadku klasy reprezentującej drzwi może to być na przykład położenie klamki, kierunek otwierania skrzydła, a w przypadku skrzyni jej rozmiary i tekstura. Rozwijając to podejście można utworzyć kolejny poziom klas pochodnych opisujących różne rodzaje skrzyni i drzwi.

**Rysunek 19.13.**

Zastosowanie dziedziczenia pozwala utworzyć klasy pochodne klasy CObject



Implementacja klasy CObject zawierać będzie atrybuty opisujące położenie, prędkość i przyspieszenie obiektu oraz rozmiar sfery otaczającej obiekt. Posiadać będzie także metody umożliwiające załadowanie obiektu do pamięci, zwolnienie pamięci zajmowanej przez obiekt, narysowanie obiektu i jego animację. Poniżej przedstawiona została definicja klasy CObject (umieszczona w pliku *object.h*):

```

#ifndef __OBJECT_H
#define __OBJECT_H

#include "vector.h"

class CObject
{
public:
    CVector position;           // położenie obiektu
    CVector velocity;          // prędkość obiektu
    CVector acceleration;      // przyspieszenie obiektu
    scalar_t size;              // promień sfery
}
  
```

```
public:  
    CObject() {};           // konstruktor  
    ~CObject() {};         // destruktor  
  
    virtual void Load() = 0; // ładuje obiekt do pamięci  
    virtual void Unload() = 0; // zwalnia pamięć zajmowaną przez obiekt  
    virtual void Draw() = 0; // rysuje obiekt  
  
    // animuje obiekt (korzystając z praw fizyki)  
    virtual void Animate(scalar_t deltaTime) = 0;  
};  
  
#endif
```

Metody Load(), UnLoad(), Draw() i Animate() zostały zadeklarowane jako metody wirtualne, wobec czego każda klasa pochodna musi dostarczyć ich implementacji. W przeciwnym razie nastąpi błąd komilacji.

A oto przykład definicji klasy pochodnej klasy CObject:

```
#include "object.h"  
  
class CCrate  
{  
public:  
    CCrate() {};  
    ~CCrate() {};  
  
    void Load()  
    {  
        // ... ładuje dane opisujące skrzynię  
    }  
  
    void Unload  
    {  
        // zwalnia pamięć zajmowaną przez dane skrzyni  
    }  
  
    void Draw()  
    {  
        // rysuje skrzynię  
    }  
  
    void Animate(scalar_t deltaTime)  
    {  
        // tworzy klatkę animacji  
    }  
};
```

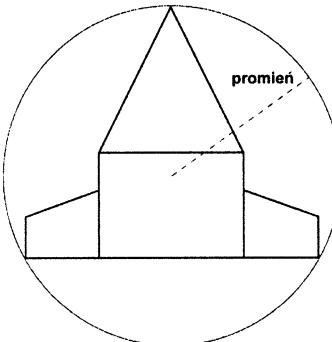
## Zderzenia obiektów

*Wykrywanie zderzeń obiektów* oraz ich obsługa są konieczne dla prawidłowej prezentacji oddziaływań pomiędzy obiektami gry. Pozwalają na przykład ustalić, czy wystrzelona kula trafiła przeciwnika czy też znajdująca się zanim ścianę, a także utrzymać postacie gry wewnątrz ścian pomieszczeń.

## Sfery ograniczeń

Jeden z najprostszych sposobów wykrywania zderzeń polega na zastosowaniu *sfer ograniczeń obiektów*. Przyjmuje się, że każdy obiekt ogranicza sfera o promieniu równym odległości od środka obiektu do najbardziej oddalonego jego wierzchołka, co ilustruje rysunek 19.14.

**Rysunek 19.14.**  
*Sfera ograniczająca obiekt*



W zależności od regularności kształtu obiektu może zajść konieczność wyznaczenia promienia ograniczającej go sfery. W tym celu trzeba przejrzeć w pętli wszystkie wierzchołki obiektu i znaleźć największą odległość wierzchołka od środka. W każdym przebiegu pętli sprawdzić należy oczywiście tylko to, czy odległość bieżącego wierzchołka jest większa od bieżącej odległości maksymalnej. Jeśli jest, to zastąpić trzeba bieżącą odległość przez odległość bieżącego wierzchołka. A oto przykład implementacji funkcji znajdującej promień sfery ograniczającej obiekt:

```
float FindBoundingSphereRadius(CVector *points, CVector center, int numPoints)
{
    float currDist = 0.0; // bieżąca odległość
    float maxDist = 0.0; // największa odległość

    // sprawdza, czy lista wierzchołków nie jest pusta
    if (points == NULL)
        return -1.0;

    // przegląda w pętli wierzchołki obiektu
    for(int idx = 0; idx < numPoints; idx++)
    {
        // wyznacza kwadrat odległości wierzchołka od środka obiektu
        // korzystając z operatorów i metod klasy CVector
        // poniższą operację możemy też zapisać jako
        // currDist = (points[idx] - center).Length()
        currDist = ((points[idx]->x - center.x)*( points[idx]->x - center.x)) +
                   ((points[idx]->y - center.y)*( points[idx]->y - center.y)) +
                   ((points[idx]->z - center.z)*( points[idx]->z - center.z));

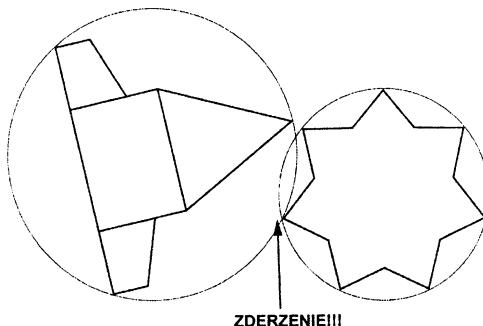
        // porównuje bieżącą odległość z odlegością maksymalną
        if (currDist > maxDist)
            maxDist = currDist;
    }

    // zwraca promień sfery
    return sqrt(maxDist);
}
```

Wykrywając zderzenia obiektów za pomocą sfer ograniczeń sprawdza się, czy odległość pomiędzy obiektami jest mniejsza niż suma promieni ich sfer ograniczeń. Można założyć na przykład, że do statku kosmicznego zbliża się asteroid. Sytuację tę ilustruje rysunek 19.15. Metoda sfer ograniczających obiekty wykryje ich kolizję, jeśli odległość pomiędzy środkami asteroidu i statku będzie mniejsza lub równa sumie promieni ograniczających je sfer.

**Rysunek 19.15.**

*Wykrywanie  
zderzenia metodą  
sfer ograniczających*

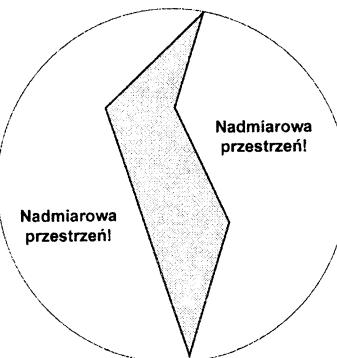


ZDERZENIE!!!

Metoda sfer ograniczających posiada pewne wady. Wystarczy przyjrzeć się na przykład obiekowi przedstawionemu na rysunku 19.16. W przypadku obiektów o tak nieregularnym kształcie sfery ograniczające okazują się mało dokładnym rozwiązaniem problemu wykrywania zderzeń.

**Rysunek 19.16.**

*Zastosowanie sfery  
ograniczającej  
w przypadku obiektu  
o nieregularnych  
ksztaltach*

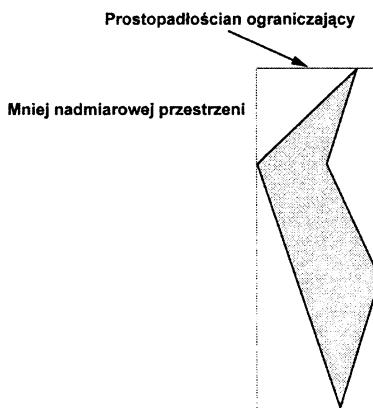


## Prostopadłościany ograniczenie

Dokładniejszą detekcję zderzeń umożliwia metoda wykorzystująca *prostopadłościany ograniczające obiekty*. Zamiast sfer ograniczeń wykorzystuje ona prostopadłościany ograniczające najodleglejsze wierzchołki obiektu na każdej z osi układu współrzędnych. Przykładowy prostopadłościan ograniczeń przedstawia rysunek 19.17.

Prostopadłościan ograniczeń tworzy się w podobny sposób jak sferę ograniczeń, a jedną różnicą jest to, że zamiast promienia sfery trzeba wyznaczyć ściany prostopadłościacanu. W tym celu należy przeglądać listę wierzchołków obiektu i poszukać wierzchołków o najmniejszej i największej wartości danej współrzędnej. Implementacja funkcji tworzącej prostopadłościan może wyglądać następująco:

**Rysunek 19.17.**  
Prostopadłościan  
ograniczeń  
otaczający obiekt



```
void CalcBoundingBox(CVector *points, int numPoints, CVector *min, CVector *max)
{
    // sprawdza, czy lista wierzchołków nie jest pusta
    if (points == NULL)
        return;

    // zeruje najmniejszą i największą wartość współrzędnych
    min->x = min->y = min->z = 0.0;
    max->x = max->y = max->z = 0.0;

    // przegląda w pętli listę wierzchołków
    for(int idx = 0; idx < numPoints; idx++)
    {
        if (points[idx]->x > max->x)          // max x
            max->x = points[idx]->x;
        if (points[idx]->x < min->x)          // min x
            min->x = points[idx]->x;
        if (points[idx]->y > max->y)          // max y
            max->y = points[idx]->y;
        if (points[idx]->y < min->y)          // min y
            min->y = points[idx]->y;
        if (points[idx]->z > max->z)          // max z
            max->z = points[idx]->z;
        if (points[idx]->z < min->z)          // min z
            min->z = points[idx]->z;
    }
}
```

Funkcja ta utworzy prostopadłościan, którego ściany będą równoległe do osi układu współrzędnych. Wykrywanie zderzeń za pomocą takich prostopadłościanów jest szybkie i wymaga jedynie wykonania prostego porównania:

```
if ( (object->x >= box.min->x && object->x <= box.max->x) &&
    (object->y >= box.min->y && object->y <= box.max->y) &&
    (object->z >= box.min->z && object->z <= box.max->z) )
{
    // nastąpiło zderzenie
}
```

Prostopadłościany ograniczeń, chociaż dokładniejsze od sfer ograniczeń, posiadają tę samą wadę — nie są wystarczająco dokładne w wielu przypadkach. W zależności od rodzaju tworzonej gry można jednak wstępnie zastosować sfery lub prostopadłościany ograniczeń, aby zaobserwować, czy zderzenia obiektów w ogóle zachodzą. Po określaniu natury zderzeń można udoskonalić metodę wykrywania zderzeń przez zastosowanie hierarchii mniejszych sfer lub prostopadłościanów ograniczających obiekt.

Dobrym przykładem zastosowania takiego rozwiązania są gry dotyczące sportów i walki, w których należy rozpoznawać uderzenie przeciwnika w różne części ciała. Hierarchia sfer bądź prostopadłościanów ograniczających postać przeciwnika będzie składać się zasadniczej sfery lub prostopadłościanu służącego do wykrywania uderzenia, a następnie serii mniejszych ograniczeń pozwalających zidentyfikować miejsce uderzenia.

## Zderzenia płaszczyzn

W przypadkach, gdy metody wykorzystujące sfery bądź prostopadłościany zderzeń okazują się niewystarczające, trzeba sięgnąć po inne rozwiązanie. Nie są to szczególnie rzadkie sytuacje, gdyż na przykład zapobieganie przenikaniu postaci gry przez ściany pomieszczeń wymaga precyzyjnego wykrywania kolizji.

Aby nie dopuścić do przenikania obiektów przez ściany, trzeba wykrywać moment ich kontaktu ze ścianą i zapobiegać dalszemu ruchowi w tym kierunku. W tym celu wystarczy sprawdzić, czy wystąpi przecięcie obiektu z płaszczyzną wielokąta tworzącego ścianę. Jeśli przecięcie takie będzie miało miejsce, to będzie trzeba jeszcze sprawdzić, czy zachodzi ono w obrębie wielokąta.

Metoda ta może służyć nie tylko do wykrywania zderzeń obiektów ze ścianami, ale także kolizji z dowolnymi wielokątami. Można wykorzystać ją na przykład do wyznaczenia punktu trafienia promienia lasera w konkretny wielokąt pewnego obiektu. Należy przyjrzeć się zatem bliżej sposobowi jej implementacji.

Płaszczyzny i ich równania omówione zostały już wcześniej w bieżącym rozdziale. Dla reprezentacji płaszczyzn utworzono klasę CPlane. Dwie jej metody wymagają w tym miejscu szerszego omówienia.

```
// fragment klasy CPlane

// sprawdza, czy punkt point leży na płaszczyźnie
const bool inline PointOnPlane(const CVector& point) const
{
    return DistanceToPlane(point) == 0;
}

// zwraca odległość punktu od płaszczyzny
const double inline DistanceToPlane(const CVector& point) const
{
    return N % point + D;
```

Najpierw trzeba przyjrzeć się metodzie DistanceToPlane(). Zwraca ona odległość punktu przekazanego jej jako parametr od płaszczyzny reprezentowanej przez obiekt klasy CPlane. Aby wyznaczyć wartość tej odległości do stałej przesunięcia płaszczyzny, dodaje się wynik

iloczynu skalarnego wektora normalnego do płaszczyzny i wektora reprezentującego dany punkt. Jeśli uzyskany wynik jest ujemny, oznacza to, że punkt znajduje się „za” płaszczyzną (czyli po przeciwej stronie płaszczyzny niż wektor normalny). Jeśli wynik jest dodatni, to punkt znajduje się po tej samej stronie płaszczyzny co wektor normalny. Gdy odległość punktu od płaszczyzny równa jest zero, oznacza to, że punkt leży na płaszczyźnie, co wykorzystuje metoda `PointOnPlane()`.

Zastosowanie tych metod do wykrywania zderzeń wymaga jedynie pewnej zmiany kontekstu sytuacji. Można założyć na przykład, że badany punkt jest w rzeczywistości docelowym punktem obiektu w tworzonej klatce grafiki. Najpierw wywołuje się wtedy metodę `DistanceToPlane()`, a uzyskany rezultat porównuje z bieżącą odlegością obiektu od płaszczyzny. Kolizja z płaszczyzną następuje wtedy, gdy odległości te posiadać będą przeciwnie znaki lub punkt docelowy będzie leżał na płaszczyźnie. Jeśli na przykład obiekt znajduje się 5 jednostek od płaszczyzny, a metoda `DistanceToPlane()` zwróci wartość -0.05, to wykryte zostanie zderzenie. Gdy natomiast odległość obiektu od płaszczyzny wynosi 3 jednostki, a metoda `DistanceToPlane()` zwróci wartość 0.01, to zderzenie nie nastąpi.

Sam fakt zderzenia obiektu z płaszczyzną nie oznacza jeszcze, że miało miejsce zderzenie z wielokątem. Dotychczas ustalono jedynie, że punkt obiektu poruszając się przecina płaszczyznę wielokąta. Trzeba więc sprawdzić jeszcze, czy punkt przecięcia leży w obrębie wielokąta. W tym celu używa się metody `RayIntersection()` klasy `CPlane`:

```
const CVector inline RayIntersection(const CVector& rayPos,
                                      const CVector& rayDir) const
{
    const double a = N % rayDir;
    if (a == 0)
        return rayPos; // promień jest równoległy do płaszczyzny

    return rayPos - rayDir * (DistanceToPlane(rayPos) / a);
}
```

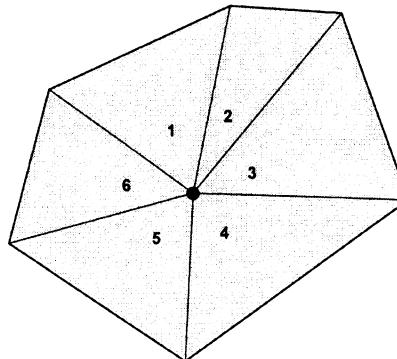
Metoda ta zwraca punkt na płaszczyźnie, w którym przecina ją promień o początku w punkcie `rayPos` i kierunku `rayDir`. W omawianym przypadku jej zastosowania początkiem promienia będzie bieżące położenie punktu, a kierunek promienia może być reprezentowany przez wektor prędkości obiektu. Po ustaleniu punktu kolizji trzeba sprawdzić, czy leży on wewnątrz wielokąta.

Istnieje wiele metod umożliwiających ustalenie, czy punkt leży w obszarze wielokąta. Jedna z nich polega na utworzeniu płaszczyzn dla każdego boku wielokąta tak, by wektory normalne do tych płaszczyzn były skierowane do środka wielokąta. Następnie wyznacza się odległość punktu od każdej z tych płaszczyzn. Jeśli wszystkie odległości są większe od zera, to punkt należy do wielokąta. Wadą tej metody jest jej niska efektywność.

Inna metoda polega na sumowaniu kątów, które tworzą odcinki łączące dany punkt z wierzchołkami wielokąta. Jeśli punkt leży wewnątrz wielokąta, to suma tych kątów powinna być równa (lub bardzo bliska) wartości  $2\pi$  (w radianach) lub  $360$  (w stopniach), co pokazuje rysunek 19.18.

**Rysunek 19.18.**

*Sumowanie kątów pozwala ustalić, czy punkt leży wewnątrz wielokąta*



A oto kod funkcji, która sprawdza, czy punkt p należy do wielokąta zdefiniowanego przez wierzchołki points:

```
#define TWOPI 6.283185307179586476925287
#define RADTODEG 57.2957795
#define TOLERANCE 0.0000001

// PointInPolygon
// opis: zwraca wartość logiczną true, gdy punkt p należy
// do wielokąta zdefiniowanego przez wierzchołki points
bool PointInPolygon(CVector p, CVector *points, int numPoints)
{
    CVector segment1, segment2; // wektory łączące punkt z wierzchołkami
    double length1, length2; // długości tych wektorów
    double sumAngles = 0.0; // suma kątów pomiędzy wektorami
    double cosAngle = 0.0; // cosinus kąta pomiędzy wektorami

    // przegląda w pętli wierzchołki wielokąta
    for(int idx = 0; idx < numPoints; idx++)
    {
        // oblicza długości dwóch wektorów
        // łączących punkt z wierzchołkami wielokąta
        segment1 = points[idx] - p;
        segment2 = points[(idx + 1) % numPoints] - p;

        // sprawdza, czy punkt leży na granicy
        if (segment1.Length() * segment2.Length() <= TOLERANCE)
        {
            sumAngles = TWOPI; // przyjmujemy, że punkty graniczne należą do wielokąta
            break;
        }

        // wyznacza funkcję cosinus kąta między wektorami
        cosAngle = (segment1 % segment2) / (segment1.Length() * segment2.Length());

        // dodaje kąt do sumy kątów
        sumAngles += cos(cosAngle);
    }
}
```

```

// jeśli suma kątów równa jest 2*PI (uwzględniając tolerancję),
// to punkt należy do wielokąta
if ((sumAngles <= (TWOPI + TOLERANCE)) && (sumAngles >= (TWOPI - TOLERANCE)))
    return true;
else
    return false;
}

```

Jeśli funkcja PointInPolygon() zwróci wartość true, oznacza to, że nastąpi zderzenie z wielokątem.

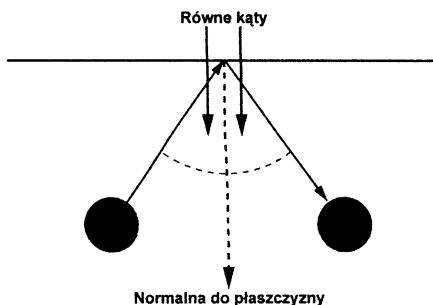
## Obsługa zderzenia

Po wykryciu zderzenia trzeba je odpowiednio obsłużyć. Sposób obsługi zależy w dużej mierze od rodzaju zderzających się obiektów, ale istnieje ogólna zasada, która okazuje się przydatna w większości przypadków.

Najłatwiej omówić ją na przykładzie zderzeń kul bilardowych. Kule te uderzając w bandę stołu pod pewnym kątem odbijają się następnie pod takim samym kątem, co pokazano na rysunku 19.19. Kąt ten nazywany jest w fizyce *kątem padania* (tworzy go wektor prędkości obiektu i wektor normalny do płaszczyzny odbicia).

**Rysunek 19.19.**

Kąt odbicia  
jest równy  
kątowi padania



Rozważając zderzenia kuli nie bierze się pod uwagę ani ich ruchu obrotowego, ani tarcia pomiędzy bandą i kulą. Interesujący jest tu jedynie kierunek, w którym kula będzie się poruszać po zderzeniu, ponieważ zgodnie z zasadami zachowania pędu i energii prędkość kuli po zderzeniu pozostanie taka sama.

Dysponując wektorem prędkości kuli i wektorem normalnym do płaszczyzny bandy można łatwo obliczyć kierunek ruchu kuli po zderzeniu. W tym celu można skorzystać z zaimplementowanej wcześniej metody Reflection() klasy CVector:

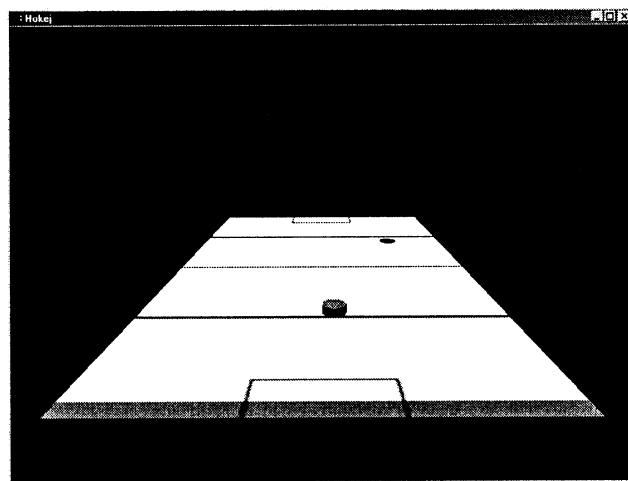
```
reflectedVelocity = velocity.Reflection(plane->normal);
```

Uzyskanie nowego wektora prędkości obiektu pozwala obsłużyć jego zderzenie z przeszkołą.

## Przykład: hokej

Omówione dotąd zagadnienia zilustrowane zostaną przykładem kodu prostej gry. Rysunek 19.20 przedstawia widok okna programu, który zaimplementowany zostanie w tym

**Rysunek 19.20.**  
Program ilustrujący  
zasady fizyki  
na przykładzie hokeja

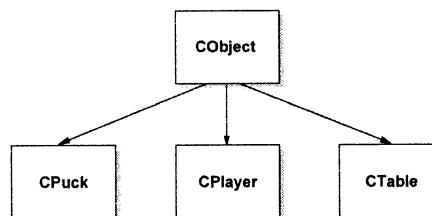


podrozdziale. Chociaż nie jest to typowa gra, to program ten ilustruje podstawowe zagadnienia fizyki, które omówione zostały w bieżącym rozdziale, takie jak prędkość, przyspieszenie, tarcie, wykrywanie zderzeń, a także wykorzystanie modelowania obiektowego do tworzenia wirtualnego świata.

## Świat hokeja

Zanim można będzie przejść do szczegółów związanych z implementacją hokeja, trzeba zastanowić się nad ogólnym sposobem, w jaki zaprojektowany zostanie szkielet gry. Zgodnie z zasadami programowania obiektowego należy najpierw zidentyfikować obiekty tworzące świat gry, a następnie zaproponować struktury, za pomocą których będzie można je modelować. W przypadku tej gry obiektami tymi będą: lodowisko, krążek i gracz. Dla ich reprezentacji utworzone zostaną osobne klasy jako pochodne klasy bazowej CObject. Rysunek 19.21 przedstawia uzyskaną hierarchię klas.

**Rysunek 19.21.**  
Struktura klas  
gry w hokeja



## Lodowisko

Pierwszą z implementowanych klas gry będzie klasa reprezentująca lodowisko. Będzie ona ograniczać ruch krążka i gracza do powierzchni o rozmiarach  $300 \times 500$  za pomocą czterech płaszczyzn tworzących bandy. Aby uczynić lodowisko bardziej realistycznym, można pokryć je odpowiednią tekstonią. A oto definicja klasy CTable, która została umieszczona w pliku *table.h*:

```

#ifndef __TABLE_H
#define __TABLE_H

#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glaux.h>
#include <math.h>
#include "vector.h"
#include "object.h"
#include "texture.h"
#include "plane.h"

/*
CTable (klasa pochodna klasy CObject)

Opis: reprezentuje lodowisko. Przechowuje współrzędne jego narożników,
      płaszczyzny band i teksturę lodowiska.

*/
class CTable : public CObject
{
private:
    CTexture iceTex;           // tekstura lodowiska

public:
    float tableCorners[4][3];   // współrzędne narożników
    CPlane tableWalls[4];       // bandy

    CTable()
    {
        position = CVector(0.0, 0.0, 0.0);
    }

    ~CTable()
    {}

    void Load();
    void Unload();

    void Draw();
    void Animate(scalar_t deltaTime);
    void SetupTexture();
};

#endif

```

Klasa CTable przechowuje współrzędne narożników lodowiska, obiekty klasy CPlane reprezentujące jego bandy oraz obiekt tekstury klasy CTexture. Ponieważ klasa CTable jest pochodną klasy CObject, to posiada także atrybuty opisujące położenie, prędkość i przyspieszenie. Można się skorzystać jedynie z atrybutu określającego położenie obiektu, aby umieścić lodowisko w początku układu współrzędnych.

Pierwszą z omawianych metod klasy CTable będzie metoda SetupTexture(), która konfiguruje teksturę OpenGL. Zakłada ona, że tekstura została już wcześniej załadowana za pomocą metody CTexture::LoadTexture() (która wywołuje się wewnątrz metody CTable::Load()).

```
// SetupTexture()
// opis: inicjuje teksturę OpenGL
void CTable::SetupTexture()
{
    // konfiguruje teksturę
    glGenTextures(1, &iceTex.texID);
    glBindTexture(GL_TEXTURE_2D, iceTex.texID);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    switch (iceTex.textureType)
    {
        case BMP:
            gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, iceTex.width, iceTex.height,
                               GL_RGB, GL_UNSIGNED_BYTE, iceTex.data);
            break;
        case PCX:
            gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, iceTex.width, iceTex.height,
                               GL_RGBA, GL_UNSIGNED_BYTE, iceTex.data);
        case TGA:      // nie obsługuje formatu TGA
            break;
        default:
            break;
    }
}
```

Sposób działania metody `SetupTexture()` jest dość oczywisty. Tworzy ona jedynie obiekt tekstury i konfiguruje jej parametry korzystając z mipmap (patrz: rozdział 8.). Mipmapy tworzone są na podstawie obrazu zawartego w pliku w formacie *BMP* lub *PCX*.

Kolejna z metod klasy `CTable`, `Load()`, inicjuje atrybuty obiektu:

```
// Load()
// opis: inicjuje atrybuty obiektu
void CTable::Load()
{
    // inicjuje współrzędne narożników (300x500)
    tableCorners[0][0] = 0.0;
    tableCorners[0][1] = 0.0;
    tableCorners[0][2] = 0.0;

    tableCorners[1][0] = 0.0;
    tableCorners[1][1] = 0.0;
    tableCorners[1][2] = -500.0;

    tableCorners[2][0] = 300.0;
    tableCorners[2][1] = 0.0;
    tableCorners[2][2] = -500.0;

    tableCorners[3][0] = 300.0;
    tableCorners[3][1] = 0.0;
    tableCorners[3][2] = 0.0;

    // inicjuje bandy (płaszczyzny)
    // 0 = lewa, 2 = na przeciw gracza, 1 = prawa, 3 = od strony gracza
```

```

// narożniki 1,0 lewa banda
// narożniki 3,2 prawa banda
// narożniki 2,1 banda na przeciwny gracza
// narożniki 3,0 banda od strony gracza
tableWalls[0] = CPlane(
    CVector(tableCorners[0][0], tableCorners[0][1], tableCorners[0][2]),
    CVector(tableCorners[1][0], tableCorners[1][1], tableCorners[1][2]),
    CVector(tableCorners[1][0], 10.0, tableCorners[1][2])
);

tableWalls[1] = CPlane(
    CVector(tableCorners[2][0], tableCorners[2][1], tableCorners[2][2]),
    CVector(tableCorners[3][0], tableCorners[3][1], tableCorners[3][2]),
    CVector(tableCorners[3][0], 10.0, tableCorners[3][2])
);

tableWalls[2] = CPlane(
    CVector(tableCorners[1][0], tableCorners[1][1], tableCorners[1][2]),
    CVector(tableCorners[2][0], tableCorners[2][1], tableCorners[2][2]),
    CVector(tableCorners[2][0], 10.0, tableCorners[2][2])
);

tableWalls[3] = CPlane(
    CVector(tableCorners[3][0], tableCorners[3][1], tableCorners[3][2]),
    CVector(tableCorners[0][0], tableCorners[0][1], tableCorners[0][2]),
    CVector(tableCorners[0][0], 10.0, tableCorners[0][2])
);

// ładuje i inicjuje tekstury
iceTex.LoadTexture("table.bmp");
SetupTexture();
}

```

Metoda Load() definiuje narożniki lodowiska i na podstawie ich współrzędnych tworzy płaszczyznę band. Bandy otrzymują wysokość 10 jednostek. Na końcu metoda konfiguruje teksturę posługując się metodą SetupTexture().

Kolejna z metod klasy CTable, Draw(), rysuje lodowisko:

```

// Draw()
// opis: rysuje lodowisko
void CTable::Draw()
{
    // rysuje lodowisko
    glPushMatrix();
    glTranslatef(position.x, position.y, position.z);

    // rysuje taflę
    glColor3f(1.0, 1.0, 1.0);
    glBindTexture(GL_TEXTURE_2D, iceTex.texID);
    glBegin(GL_TRIANGLE_STRIP);
    glTexCoord2f(1.0, 1.0);
    glVertex3fv(&tableCorners[3][0]);
    glTexCoord2f(0.0, 1.0);
    glVertex3fv(&tableCorners[2][0]);
    glTexCoord2f(1.0, 0.0);
    glVertex3fv(&tableCorners[0][0]);
}

```

```
    glTexCoord2f(0.0, 0.0);
    glVertex3fv(&tableCorners[1][0]);
    glEnd();

    // włącza łączenie kolorów dla band
    glEnable(GL_BLEND);
    glDepthMask(GL_FALSE);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);

    glColor4f(0.0, 0.1, 0.3, 1.0);
    // lewa banda
    glBegin(GL_TRIANGLE_STRIP);
        glVertex3fv(&tableCorners[1][0]);
        glVertex3f(tableCorners[1][0], 10.0, tableCorners[1][2]);
        glVertex3fv(&tableCorners[0][0]);
        glVertex3f(tableCorners[0][0], 10.0, tableCorners[0][2]);
    glEnd();

    // prawa banda
    glBegin(GL_TRIANGLE_STRIP);
        glVertex3fv(&tableCorners[3][0]);
        glVertex3f(tableCorners[3][0], 10.0, tableCorners[3][2]);
        glVertex3fv(&tableCorners[2][0]);
        glVertex3f(tableCorners[2][0], 10.0, tableCorners[2][2]);
    glEnd();

    // przeciwnielega banda
    glBegin(GL_TRIANGLE_STRIP);
        glVertex3fv(&tableCorners[2][0]);
        glVertex3f(tableCorners[2][0], 10.0, tableCorners[2][2]);
        glVertex3fv(&tableCorners[1][0]);
        glVertex3f(tableCorners[1][0], 10.0, tableCorners[1][2]);
    glEnd();

    // banda od strony gracza (przezroczysta)
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glColor4f(0.0, 0.1, 0.3, 0.6);
    glBegin(GL_TRIANGLE_STRIP);
        glVertex3fv(&tableCorners[3][0]);
        glVertex3f(tableCorners[3][0], 10.0, tableCorners[3][2]);
        glVertex3fv(&tableCorners[0][0]);
        glVertex3f(tableCorners[0][0], 10.0, tableCorners[0][2]);
    glEnd();

    // przywraca testowanie głębi, wyłącza łączenie kolorów
    glDepthMask(GL_TRUE);
    glDisable(GL_BLEND);
    glPopMatrix();
}
```

Metoda ta rysuje pokrytą teksturą tafłę lodowiska oraz bandy. Rysując bandę leżącą najbliżej gracza włącza łączenie kolorów tak, by banda nie przesłaniała odbijającego się od niej krążka.

Ostatnia z metod, `Unload()`, wywołuje jedynie metodę `CTexture::Unload()` w celu zwolenia pamięci zajmowanej przez teksturę.

```
// Unload
// opis: zwalnia pamięć zajmowaną przez teksturę
void CTable::Unload()
{
    iceTex.Unload();
}
```

## Krażek i zderzenia

Zająć się teraz należy kluczowym obiektem z punktu widzenia programu gry w hokeja: krażkiem. Krażek reprezentowany będzie przez walec unoszący się nad powierzchnią lodowiska. Jego ruch ograniczony będzie bandami. Na kierunek i przyspieszenie ruchu krażka będą wpływać: graczy oraz tarcie, które spowoduje zatrzymanie krażka po pewnym czasie.

Oprócz atrybutów, które krażek odziedziczy po klasie `CObject` (położenie, prędkość, przyspieszenie) klasa `CPuck` przechowywać będzie promień krażka używany podczas jego rysowania i wykrywania zderzeń. Dodatkowo poza klasą `CPuck` należy przechowywać współrzędne punktów opisujących dolną i górną podstawę krażka (koła o promieniu jednej jednostki leżące w odległości 0,6 jednostki od siebie). Krażek można narysować jako walec wypełniony jednolitym kolorem. Punkty opisujące podstawy krażka oraz definicję klasy `CPuck` umieszczone zostały w pliku *puck.h*:

```
#ifndef __PUCK_H
#define __PUCK_H

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glaux.h>
#include "object.h"
#include "vector.h"
#include "table.h"

// punkty opisujące podstawy krażka
static float puckData[32][3] =
    // podstawa dolna
    { { 0.0, 0.01, 1.0 }, { 0.3827, 0.01, 0.9239 },
    { 0.7071, 0.01, 0.7071 }, { 0.9239, 0.01, 0.3827 }, { 1.0, 0.01, 0.0 },
    { 0.9239, 0.01, -0.3827 }, { 0.7071, 0.01, -0.7071 }, { 0.3827, 0.01, -0.9239 },
    { 0.0, 0.01, -1.0 }, { -0.3827, 0.01, -0.9239 }, { -0.7071, 0.01, -0.7071 },
    { -0.9239, 0.01, -0.3827 }, { -1.0, 0.01, 0.0 }, { -0.9237, 0.01, 0.3827 },
    { -0.7071, 0.01, 0.7071 }, { -0.3827, 0.01, 0.9239 },
    // podstawa górska
    { 0.0, 0.07, 1.0 }, { 0.3827, 0.07, 0.9239 },
    { 0.7071, 0.07, 0.7071 }, { 0.9239, 0.07, 0.3827 }, { 1.0, 0.07, 0.0 },
    { 0.9239, 0.07, -0.3827 }, { 0.7071, 0.07, -0.7071 }, { 0.3827, 0.07, -0.9239 },
    { 0.0, 0.07, -1.0 }, { -0.3827, 0.07, -0.9239 }, { -0.7071, 0.07, -0.7071 },
    { -0.9239, 0.07, -0.3827 }, { -1.0, 0.07, 0.0 }, { -0.9237, 0.07, 0.3827 },
    { -0.7071, 0.07, 0.7071 }, { -0.3827, 0.07, 0.9239 } };
```

```
class CPuck : public CObject
{
private:
    float radius; // promień krążka

public:
    CPuck()
    {
        acceleration = CVector(0.0, 0.0, 0.0);
        velocity = CVector(50.0, 0.0, 100.0);
        position = CVector(150.0, 0.0, -200.0);
        radius = 10.0; size = radius;
    }

    CPuck(float r)
    {
        radius = r; size = r;
        acceleration = CVector(0.0, 0.0, 0.0);
        velocity = CVector(50.0, 0.0, 100.0);
        position = CVector(150.0, 0.5, -200.0);
    }

    ~CPuck() {}

    void Load();
    void Unload();

    void Draw();
    void Animate(scalar_t deltaTime);
    void Animate(scalar_t deltaTime, CTable *table);
};

#endif
```

Należy zauważyć, że definicja klasy CPuck zawiera dwie wersje metody Animate(). Druga z nich posiada dodatkowy parametr, table, który pozwala określić lodowisko, po którym porusza się krążek i używany jest do wykrywania zderzeń.

Ponieważ w rzeczywistości klasa CPuck implementuje jedynie metody Draw() i Animate() klasy bazowej, to poniżej przedstawiony został pełen kod implementacji metod klasy (znajdujący się w pliku *puck.cpp*):

```
#include "puck.h"

// Load()
// opis: nie jest używana
void CPuck::Load()
{
    // metoda wirtualna musi być jednak zastąpiona
}

// Unload()
// opis: nie jest używana
void CPuck::Unload()
{
    // metoda wirtualna musi być jednak zastąpiona
}
```

```
// Draw()
// opis: rysuje krążek jako walec wypełniony jednym kolorem
void CPuck::Draw()
{
    glPushMatrix();
    glTranslatef(position.x, position.y, position.z);
    glScalef(radius, radius, radius);
    glColor3f(0.0, 0.0, 0.0); // kolor czarny

    // rysuje pierścień krążka
    glBegin(GL_TRIANGLE_STRIP);
    for (int i = 0; i < 16; i++)
    {
        glVertex3fv(&puckData[i+16][0]); // dolne punkty
        glVertex3fv(&puckData[i][0]); // górne punkty
    }
    glVertex3fv(&puckData[16][0]);
    glVertex3fv(&puckData[0][0]);
    glEnd();

    // rysuje podstawy krążka
    glBegin(GL_TRIANGLE_FAN);
    glVertex3f(0.0, puckData[0][1], 0.0);
    for (i = 0; i < 16; i++)
        glVertex3fv(&puckData[i][0]);
    glVertex3fv(&puckData[0][0]);
    glEnd();
    glBegin(GL_TRIANGLE_FAN);
    glVertex3f(0.0, puckData[31][1], 0.0);
    for (i = 16; i < 32; i++)
        glVertex3fv(&puckData[i][0]);
    glVertex3fv(&puckData[16][0]);
    glEnd();

    glPopMatrix();
}

// Animate()
// opis: ta wersja nie jest używana
void CPuck::Animate(scalar_t deltaTime)
{
    // metoda wirtualna musi być jednak zastąpiona
}

// Animate()
// opis: obsługuje ruch krążka i wykrywa zderzenia
void CPuck::Animate(scalar_t deltaTime, CTable *table)
{
    if (deltaTime <= 0) return;

    double fastestTime = deltaTime;
    CPlane *planeCollision = NULL;
    for (int idx = 0; idx < 4; idx++)
    {
        CPlane *plane = &table->tableWalls[idx];
        double collisionTime;
        double a, b, c;
```

```

// iloczyn skalarnej
a = plane->N % (acceleration * 0.5);
b = plane->N % velocity;
c = plane->N % position + plane->D - radius;

if (a == 0)                                // równanie pierwszego stopnia
{
    if (b != 0 && c != 0)      // musi mieć pewną prędkość
    {
        collisionTime = -c/b;
        if (collisionTime >= 0 && collisionTime < fastestTime)
        {
            fastestTime = collisionTime;
            planeCollision = plane;
        }
    }
}
else
{
    // równanie drugiego stopnia
    double D = b*b - 4*a*c; // wyznacznik delta
    if (D >= 0)
    {
        // rozwiązania równania (wartości ujemne są odrzucane)
        collisionTime = (- b - sqrt(D)) / (2*a);
        if (collisionTime >= 0 && collisionTime < fastestTime)
        {
            fastestTime = collisionTime;
            planeCollision = plane;
        }
    }
}
}

// tarcie (współczynnik = 0.2)
if (velocity.Length() > 0.0)
    acceleration = -velocity * 0.2;

// wyznacza prędkość w punkcie zderzenia
position += velocity * fastestTime + acceleration * (fastestTime*fastestTime*0.5);
velocity += acceleration * fastestTime;

// największa dozwolona prędkość 800
if (velocity.Length() > 800.0)
    velocity |= 800.0; // prędkość maksymalna 800

if (planeCollision)
{
    // prędkość po odbiciu od bandy
    velocity = velocity.Reflection(planeCollision->N);
}

// wywołanie rekurencyjne
Animate(deltaTime - fastestTime, table);
}

```

Teraz trzeba przyjrzeć się dokładnie temu, w jaki sposób metoda `Animate()` wykrywa zderzenia krążka. Zastosowane w tym przypadku rozwiązywanie różni się od omawianych wcześniej sposobów wykrywania kolizji. Tym razem wykorzystane zostanie równanie drugiego stopnia opisujące położenie obiektu w funkcji czasu:

$$x_f = x_0 + v_0 \cdot t + [1/2] \cdot a \cdot t^2$$

Równanie to jest w rzeczywistości dobrze znanym równaniem funkcji kwadratowej postaci:

$$a \cdot x^2 + b \cdot x + c = 0$$

Wartości współczynników  $a$ ,  $b$ , i  $c$  wynoszą w tym przypadku:

$$a = [1/2] \cdot a$$

$$b = v_0$$

$$c = x_0 - x_f$$

Równanie to trzeba jednak rozszerzyć o kolejne wymiary, aby mogło opisywać ruch obiektu w przestrzeni:

$$x_t = C_x + b_x \cdot t + a_x \cdot t^2$$

$$y_t = C_y + b_y \cdot t + a_y \cdot t^2$$

$$z_t = C_z + b_z \cdot t + a_z \cdot t^2$$

Podstawiając te wartości do równania płaszczyzny otrzymuje się:

$$A \cdot x_t + B \cdot y_t + C \cdot z_t + D = 0$$

Powyzsze równanie informuje o współrzędnych punktu zderzenia ( $x_t$ ,  $y_t$ ,  $z_t$ ) obiektu z płaszczyzną. Podstawiając do niego odpowiednie równania i wykonując kilka przekształceń uzyskuje się postać:

$$(A \cdot a_x + B \cdot a_y + C \cdot a_z) \cdot t^2 + (A \cdot b_x + B \cdot b_y + C \cdot b_z) \cdot t + (A \cdot C_x + B \cdot C_y + C \cdot C_z) + D = 0$$

Jest to zwykłe równanie kwadratowe typu  $a \cdot x^2 + b \cdot x + c$ , którego współczynniki stanowią wynik iloczynu skalarnego, na przykład:

$$a \cdot t^2 = (A \cdot a_x + B \cdot a_y + C \cdot a_z) \cdot t^2 = N \cdot ([1/2] \cdot A)$$

Współczynniki równania można więc zapisać korzystając z iloczynu skalarnego wektorów:

$$a = N \cdot (A \cdot [1/2])$$

$$b = N \cdot V$$

$$c = N \cdot X + D$$

Używając wartości tych współczynników można wyznaczyć czas zderzenia. W tym celu trzeba rozróżnić dwa przypadki ruchu obiektu: z przyspieszeniem i bez przyspieszenia.

Jeśli współczynnik  $a$  równania kwadratowego równy jest zero, to obiekt nie posiada przyspieszenia. W tym przypadku czas zderzenia oblicza się dzieląc wartość współczynnika  $c$  przez współczynnik prędkości  $b$ , co ilustruje poniższy fragment kodu metody `Animate()`:

```
if (a == 0)
{
    if (b != 0 && c != 0) // musi mieć pewną prędkość
    {
        collisionTime = -c/b;
        if (collisionTime >= 0 && collisionTime < fastestTime)
```

```
{
    fastestTime = collisionTime;
    planeCollision = plane;
}
}
```

Wyliczając w nim wartość czasu zderzenia trzeba zmienić jej znak, ponieważ współczynnik c ma wartość ujemną ze względu na stałą przesunięcia płaszczyzny. Następnie trzeba sprawdzić, czy zderzenie zachodzi w bieżącym przedziale czasu i zapamiętać czas zderzenia oraz płaszczyznę, z którą zderzył się obiekt.

W przypadku, gdy obiekt porusza się ruchem przyspieszonym, stosuje się wzór na ogólne rozwiązanie równania drugiego stopnia:

$$x_{1,2} = (-b \pm \sqrt{b^2 - 4 * a * c}) / 2 * a$$

Zanim skorzysta się z powyższego równania, trzeba najpierw sprawdzić, czy poszukiwane rozwiązanie w ogóle istnieje. W tym celu oblicza się wartość wyznacznika:

$$D = b^2 - 4*a*c$$

Jeśli jego wartość jest mniejsza od zera, to równanie kwadratowe nie ma rozwiązania. Jeśli wartość ta równa jest zeru, istnieje dokładnie jedno rozwiązanie, a gdy jest większa, to dwa rozwiązania. Ponieważ czas zderzenia może mieć tylko dodatnią wartość, to ujemne rozwiązanie będzie zawsze odrzucane. Wobec tego rozwiązanie równania będzie należało wyznaczać zawsze za pomocą tego samego równania, co pokazuje poniższy fragment kodu metody Animate():

```
double D = b*b - 4*a*c;           // wyznacznik delta
if (D >= 0)
{
    // rozwiązania równania (wartości ujemne są odrzucane)
    collisionTime = (- b - sqrt(D)) / (2*a);
    if (collisionTime >= 0 && collisionTime < fastestTime)
    {
        fastestTime = collisionTime;
        planeCollision = plane;
    }
}
```

Po ustaleniu czasu zderzenia metoda wprowadza opóźnienie ruchu krążka na skutek tarcia. Następnie korzystając ze zwykłych równań prędkości i położenia wyznacza nowe położenie i nową prędkość krążka. Aby krążek nie poruszał się zbyt szybko, wprowadza także dodatkowe ograniczenie jego prędkości.

Łatwo zauważyc też analizując kod metody Animate() do końca, że jest ona rekurencyjna. Celem takiego sposobu jej implementacji jest dokładne odnalezienie momentu zderzenia. W sytuacji, gdy zderzenie następuje w czasie pomiędzy kolejnymi klatkami grafiki, wyznaczenie położenia obiektu po zderzeniu staje się możliwe właśnie na skutek rekurencyjnego wywołania metody Animate().

Jeśli rekurencyjny sposób działania metody Animate() wydaje się zawiły, to warto dokładnie, krok po kroku prześledzić działanie jej kodu. Mimo że wykrywanie zderzeń w ten sposób jest dość skomplikowane, to w praktyce jego zastosowanie jest dość proste i bardzo przydatne.

Zwrócić także trzeba uwagę na to, że w przypadku ruchu krążka na lodowisku traktuje się bandy jak nieskończone płaszczyzny, a nie wielokąty, ponieważ nie sprawdza się, czy zderzenie miało miejsce w określonych granicach (jak dla wielokątów). Nie trzeba tego robić, ponieważ bandy lodowiska nakładają na siebie nawzajem ograniczenia ruchu krążka. Wystarczy więc sprawdzić, czy krążek uderzył w ogóle w bandę, a nie w którym miejscu to nastąpiło.

## Gracz

Obiekt gracza zapewnia programowi interaktywność. Gracz reprezentowany jest za pomocą walca, który można przemieszczać w obrębie lodowiska za pomocą myszy. Podobnie jak było w przypadku krążka, tak i teraz klasa reprezentująca gracza będzie posiadać atrybut określający promień walca, który będzie wykorzystywany podczas jego rysowania. Dodatkowo przechowywać będzie także współrzędne poprzedniej pozycji myszy, aby wyznaczyć ruch gracza na lodowisku. A oto definicja klasy `CPlayer`, która została umieszczona w pliku `player.h`:

```
#ifndef __PLAYER_H
#define __PLAYER_H

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glaux.h>
#include "object.h"
#include "vector.h"
#include "table.h"
#include "puck.h"

static float playerData[32][3] =
    // punkty dolnej podstawy
    { { 0.0, 0.01, 1.0 }, { 0.3827, 0.01, 0.9239 },
      { 0.7071, 0.01, 0.7071 }, { 0.9239, 0.01, 0.3827 }, { 1.0, 0.01, 0.0 },
      { 0.9239, 0.01, -0.3827 }, { 0.7071, 0.01, -0.7071 }, { 0.3827, 0.01, -0.9239 },
      { 0.0, 0.01, -1.0 }, { -0.3827, 0.01, -0.9239 }, { -0.7071, 0.01, -0.7071 },
      { -0.9239, 0.01, -0.3827 }, { -1.0, 0.01, 0.0 }, { -0.9237, 0.01, 0.3827 },
      { -0.7071, 0.01, 0.7071 }, { -0.3827, 0.01, 0.9239 },
    // punkty górnej podstawy
      { 0.0, 0.8, 1.0 }, { 0.3827, 0.8, 0.9239 },
      { 0.7071, 0.8, 0.7071 }, { 0.9239, 0.8, 0.3827 }, { 1.0, 0.8, 0.0 },
      { 0.9239, 0.8, -0.3827 }, { 0.7071, 0.8, -0.7071 }, { 0.3827, 0.8, -0.9239 },
      { 0.0, 0.8, -1.0 }, { -0.3827, 0.8, -0.9239 }, { -0.7071, 0.8, -0.7071 },
      { -0.9239, 0.8, -0.3827 }, { -1.0, 0.8, 0.0 }, { -0.9237, 0.8, 0.3827 },
      { -0.7071, 0.8, 0.7071 }, { -0.3827, 0.8, 0.9239 } };

class CPlayer : public CObject
{
private:
    float radius;
    int oldMouseX;
    int oldMouseY;
```

```
public:  
    CPlayer()  
    {  
        acceleration = CVector(0.0, 0.0, 0.0);  
        velocity = CVector(0.0, 0.0, 0.0);  
        position = CVector(150.0, 0.0, -110.0);  
        oldMouseX = 0;  
        oldMouseY = 0;  
        radius = size = 10.0;  
    }  
  
    ~CPlayer() {}  
  
    void Draw();  
    void Load();  
    void Unload();  
    void Animate(scalar_t deltaTime);  
  
    void Move(scalar_t deltaTime, int mouseX, int mouseY, CTable *table, CPuck *puck);  
};  
  
#endif
```

Tym razem zamiast metody `Animate()` umieszczona została w klasie deklaracja metody `Move()`, która animuje ruch gracza na podstawie ruchów myszy i wykrywa jego zderzenia z bandami i krążkiem.

Implementacja pozostałych metod klasy `CPlayer` przypomina implementację metod klasy `CPuck`. Poniżej zaprezentowany został kod źródłowy implementacji klasy `CPlayer` znajdujący się w pliku *player.cpp*:

```
#include "player.h"  
  
// Load()  
// opis: nie jest używana  
void CPlayer::Load()  
{  
  
// Unload()  
// opis: nie jest używana  
void CPlayer::Unload()  
{  
  
// Draw()  
// opis: rysuje reprezentację gracza na bieżącej pozycji  
void CPlayer::Draw()  
{  
    glPushMatrix();  
    glTranslatef(position.x, position.y, position.z);  
    glScalef(radius, radius, radius);  
    glColor3f(0.5, 0.0, 0.0);  
  
    // rysuje pierścień walca  
    glBegin(GL_TRIANGLE_STRIP);  
    for (int i = 0; i < 16; i++)
```

```
{  
    glVertex3fv(&playerData[i+16][0]); // wierzchołki górnej podstawy  
    glVertex3fv(&playerData[i][0]); // wierzchołki dolnej podstawy  
}  
glVertex3fv(&playerData[16][0]);  
glVertex3fv(&playerData[0][0]);  
glEnd();  
  
glColor3f(0.5, 0.5, 0.5);  
// rysuje górną i dolną podstawę  
glBegin(GL_TRIANGLE_FAN);  
    glVertex3f(0.0, playerData[0][1], 0.0);  
    for (i = 0; i < 16; i++)  
        glVertex3fv(&playerData[i][0]);  
    glVertex3fv(&playerData[0][0]);  
glEnd();  
glBegin(GL_TRIANGLE_FAN);  
    glVertex3f(0.0, playerData[31][1], 0.0);  
    for (i = 16; i < 32; i++)  
        glVertex3fv(&playerData[i][0]);  
    glVertex3fv(&playerData[16][0]);  
glEnd();  
glPopMatrix();  
}  
  
// Animate()  
// opis: nie jest używana  
void CPlayer::Animate(scalar_t deltaTime)  
{  
  
// Move()  
// opis: zmienia położenie gracza w zależności od ruchu krążka  
//       wykrywa zderzenia z krajkiem  
void CPlayer::Move(scalar_t deltaTime, int mouseX, int mouseY, CTable *table,  
                  CPuck *puck)  
{  
    int xDiff, yDiff; // odległość ruchu myszy w jednostkach myszy  
  
    // wyznacza odległość ruchu myszy  
    xDiff = (mouseX - oldMouseX)*4.0;  
    yDiff = (mouseY - oldMouseY)*4.0;  
  
    // zapamiętuje położenie myszy  
    oldMouseX = mouseX;  
    oldMouseY = mouseY;  
  
    // oblicza prędkość gracza na podstawie ruchu myszy  
    velocity = CVector(xDiff, 0.0, yDiff) * 10.0;  
  
    // oblicza nowe położenie gracza  
    position = position + (velocity * deltaTime);  
  
    // wykrywa zderzenia gracza z bandami  
    if (position.x - radius < table->tableCorners[0][0])  
        position.x = table->tableCorners[0][0] + radius;  
    if (position.x + radius > table->tableCorners[3][0])  
        position.x = table->tableCorners[3][0] - radius;
```

```
if (position.z - radius < table->tableCorners[1][2])
    position.z = table->tableCorners[1][2] + radius;
if (position.z + radius > table->tableCorners[3][2])
    position.z = table->tableCorners[3][2] - radius;

// wykrywa zderzenie gracza z krążkiem
if ((puck->position - position).Length() <= (puck->size + radius))
{
    // zderzenie, gdy gracz nie porusza się
    if (velocity.Length() == 0)
    {
        puck->velocity = -puck->velocity;
    }
    else      // zderzenie gracza w ruchu
    {
        // nowa prędkość krążka równa jest odbiciu iloczynu wektorowego
        // prędkości gracza i prędkości krążka powiększonego o prędkość gracza
        puck->velocity = puck->velocity.Reflection(puck->velocity ^ velocity);
    }
}
```

Metoda `Move()` oblicza najpierw wielkość przesunięcia myszy od momentu utworzenia poprzedniej klatki grafiki i na jej podstawie wyznacza prędkość ruchu gracza oraz jego położenie. Następnie wykrywa w uproszczony sposób zderzenia gracza z bandami oraz zderzenia z krążkiem.

W celu wykrycia zderzenia krążka i gracza porównywana jest ich wzajemna odległość z sumą promieni tych obiektów. Podobnie jak w przypadku kolizji ze ścianami metoda ta jest uproszczona i nie umożliwia perfekcyjnej symulacji zderzeń. Jej udoskonalenie możliwe jest na wzór metody, która została zastosowana do wykrywania zderzeń krążka z bandami.

## Kompletowanie programu

Utworzone w ten sposób zostały klasy reprezentujące wszystkie obiekty gry w hokeja. Teraz trzeba złożyć z nich jedną całość. Zanim to nastąpi, trzeba jednak dodać pewien brakujący element do tworzonego systemu: czas. Należy przypomnieć, że metoda `Animate()` dziedziczona po klasie `CObject` posiada jako parametr wartość zmiany czasu. Wszystkie obliczenia kinematyki ruchu obiektów przeprowadzane są w funkcji czasu. Do odmierzania czasu rzeczywistego wykorzystany więc zostanie licznik opracowanej wcześniej klasy `CHiResTimer`. Jej metoda `GetElapsedTime()` zwróci okres czasu, który upłynął od poprzedniego wywołania tej metody.

Połączenie czterech obiektów: lodowiska, krążka, gracza i licznika czasu w jedną całość nie powinno sprawić większego problemu, ponieważ prawidłowo zaprojektowana została struktura tego programu.

Na początku programu, którego kod źródłowy znajduje się w pliku `main.cpp`, umieszczona została deklaracja zmiennych globalnych:

```

#define WIN32_LEAN_AND_MEAN           // "odchudza" aplikację Windows
#define WIN32_EXTRALEAN

##### Pliki nagłówkowe
#include <windows.h>                // standardowy plik nagłówkowy Windows
#include <gl/gl.h>                   // standardowy plik nagłówkowy OpenGL
#include <gl/glu.h>                  // plik nagłówkowy biblioteki OpenGL
#include <gl/glaux.h>                // funkcje pomocnicze OpenGL

#include "HiResTimer.h"              // licznik czasu o dużej rozdzielczości
#include "vector.h"                 // algebra wektorów
#include "object.h"                  // klasa bazowa
#include "table.h"                   // lodowisko
#include "puck.h"                    // krążek
#include "player.h"                 // gracz

##### Zmienne globalne
HDC g_HDC;                         // globalny kontekst urządzenia
int mouseX, mouseY; // współrzędne myszy

##### Zmienne oświetlenia
float ambientLight[] = { 0.5f, 0.5f, 0.5f, 1.0f }; // światło otoczenia
float diffuseLight[] = { 0.5f, 0.5f, 0.5f, 1.0f }; // światło rozproszone
float lightPosition[] = { 0.0f, -1.0f, 0.0f, 0.0f }; // położenie źródła światła

##### Obiekty gry w hokeja
CTable *myTable = NULL;             // lodowisko
CPuck *myPuck = NULL;              // krążek
CHiResTimer *timer = NULL;          // licznik czasu
CPlayer *player = NULL;             // gracz

```

Funkcja Initialize() wykonuje zadanie inicjacji OpenGL i utworzenia obiektów gry:

```

// Initialize()
// opis: inicjuje OpenGL i tworzy obiekty gry
void Initialize()
{
    glClearColor(0.0, 0.0, 0.0, 0.0); // włącza bufor głębi
    glEnable(GL_DEPTH_TEST);          // cieniowanie gładkie
    glShadeModel(GL_SMOOTH);
    glDepthFunc(GL_EQUAL);
    glEnable(GL_CULL_FACE);
    glFrontFace(GL_CCW);

    // konfiguruje źródło światła LIGHT0
    glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight); // światło otoczenia
    glLightfv(GL_LIGHT0, GL_DIFFUSE, ambientLight); // światło rozproszone
    glLightfv(GL_LIGHT0, GL_POSITION, lightPosition); // położenie źródła światła

    // włącza oświetlenie
    glEnable(GL_LIGHT0);
    glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

    // włącza obsługę tekstur
    glEnable(GL_TEXTURE_2D);
}

```

```
// tworzy lodowisko i ładuje jego tekstury  
myTable = new CTable;  
myTable->Load();  
  
// tworzy krążek o promieniu 10 jednostek  
myPuck = new CPuck(10.0);  
  
// tworzy licznik czasu i inicjuje go  
timer = new CHiResTimer;  
timer->Init();  
  
// tworzy obiekt gracza  
player = new CPlayer;  
}
```

Kod kolejnej funkcji, Render(), zamieszczony jest poniżej:

```
// Render()  
// opis: przeprowadza obliczenia ruchu obiektów i rysuje je  
void Render()  
{  
  
    // ustala okres czasu, który upłynął  
    float elapsedSec = timer->GetElapsedSeconds(1);  
  
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
    glLoadIdentity();  
  
    // umieszcza kamerę w punkcie o współrzędnych (150, 150, 200)  
    // i skierowuje ją na środek lodowiska  
    gluLookAt(150.0, 150.0, 200.0, 150.0, 0.0, -300.0, 0.0, 1.0, 0.0);  
  
    // oblicza nowe położenia krążka i gracza  
    player->Move(elapsedSec, mouseX, mouseY, myTable, myPuck);  
    myPuck->Animate(elapsedSec, myTable);  
  
    // rysuje obiekty  
    player->Draw();  
    myTable->Draw();  
    myPuck->Draw();  
  
    glFlush();  
    SwapBuffers(g_HDC);  
}
```

Funkcja ta ustala najpierw okres czasu, który upłynął od utworzenia poprzedniej klatki grafiki. Czas ten wykorzystywany jest do ustalenia bieżącego położenia krążka i gracza.

Funkcja CleanUp() zwalnia pamięć zajmowaną przez obiekty gry. Wywoływana jest, gdy program kończy swoje działanie.

```
// CleanUp()  
// opis: usuwa obiekty  
void CleanUp()  
{  
    // lodowisko  
    myTable->Unload();
```

```

delete myTable;
myTable = NULL;

// krążek
delete myPuck;
myPuck = NULL;

// licznik czasu
delete timer;
timer = NULL;

// gracz
delete player;
player = NULL;
}

```

Na końcu zaprezentowany został kod trzech funkcji związanych z działaniem aplikacji w systemie Windows: `SetupPixelFormat()`, `WndProc()` i `WinMain()`:

```

// SetupPixelFormat()
// funkcja określająca format pikseli
void SetupPixelFormat(HDC hDC)
{
    int nPixelFormat;// indeks formatu pikseli

    static PIXELFORMATDESCRIPTOR pfd = {
        sizeof(PIXELFORMATDESCRIPTOR), // rozmiar struktury
        1,// domyślna wersja
        PFD_DRAW_TO_WINDOW |          // grafika w oknie
        PFD_SUPPORT_OPENGL |         // grafika OpenGL
        PFD_DOUBLEBUFFER,            // podwójne buforowanie
        PFD_TYPE_RGBA,               // tryb kolorów RGBA
        32,                          // 32-bitowy opis kolorów
        0, 0, 0, 0, 0, 0,           // nie specyfikuje bitów kolorów
        0,                           // bez buforu alfa
        0,                           // nie specyfikuje bitu przesunięcia
        0,                           // bez bufora akumulacji
        0, 0, 0, 0,                 // ignoruje bity akumulacji
        16,                          // 16-bitowy bufor z
        0,                           // bez bufora powielania
        0,                           // bez buforów pomocniczych
        PFD_MAIN_PLANE,             // główna płaszczyzna rysowania
        0,                           // zarezerwowane
        0, 0, 0 };                  // ignoruje maski warstw

    // wybiera najbardziej zgodny format pikseli
    nPixelFormat = ChoosePixelFormat(hDC, &pfd);

    // określa format pikseli dla danego kontekstu urządzenia
    SetPixelFormat(hDC, nPixelFormat, &pfd);
}

// WndProc()
// procedura okienkowa
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)

```

```
{  
    static HGLRC hRC;// kontekst tworzenia grafiki  
    static HDC hDC;// kontekst urządzenia  
    int width, height;// szerokość i wysokość okna  
    int oldMouseX, oldMouseY;  
  
    switch(message)  
    {  
        case WM_CREATE:           // okno jest tworzone  
  
            hDC = GetDC(hwnd);      // pobiera kontekst urządzenia dla okna  
            g_hDC = hDC;  
            SetupPixelFormat(hDC); // wywołuje funkcję określającą format pikseli  
  
            // tworzy kontekst tworzenia grafiki i czyni go bieżącym  
            hRC = wglCreateContext(hDC);  
            wglMakeCurrent(hDC, hRC);  
  
            return 0;  
            break;  
  
        case WM_CLOSE:            // okno jest zamykane  
  
            // deaktywuje bieżący kontekst tworzenia grafiki i usuwa go  
            wglMakeCurrent(hDC, NULL);  
            wglDeleteContext(hRC);  
  
            // wstawia komunikat WM_QUIT do kolejki  
            PostQuitMessage(0);  
  
            return 0;  
            break;  
  
        case WM_SIZE:  
            height = HIWORD(lParam); // pobiera nowe rozmiary okna  
            width = LOWORD(lParam);  
  
            if (height==0)          // unika dzielenia przez 0  
            {  
                height=1;  
            }  
  
            glViewport(0, 0, width, height); // nadaje nowe wymiary oknu OpenGL  
            glMatrixMode(GL_PROJECTION);   // wybiera macierz rzutowania  
            glLoadIdentity();             // resetuje macierz rzutowania  
  
            // wyznacza proporcje obrazu  
            gluPerspective(54.0f,(GLfloat)width/(GLfloat)height,1.0f,1000.0f);  
  
            glMatrixMode(GL_MODELVIEW);    // wybiera macierz modelowania  
            glLoadIdentity();             // resetuje macierz modelowania  
  
            return 0;  
            break;  
  
        case WM_MOUSEMOVE:  
            // zapamiętuje położenie myszy  
            oldMouseX = mouseX;  
            oldMouseY = mouseY;
```

```
// pobiera nowe współrzędne myszy
mouseX = LOWORD(lParam);
mouseY = HIWORD(lParam);

break;

default:
    break;
}

return (DefWindowProc(hwnd, message, wParam, lParam));
}

// WinMain()
// punkt, w którym rozpoczyna się wykonywanie aplikacji
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
    WNDCLASSEX windowClass;// klasa okna
    HWND hwnd;// uchwyty okna
    MSG msg;// komunikat
    bool done;           // znacznik zakończenia działania aplikacji

    // definicja klasy okna
    windowClass.cbSize= sizeof(WNDCLASSEX);
    windowClass.style= CS_HREDRAW | CS_VREDRAW;
    windowClass.lpfnWndProc= WndProc;
    windowClass.cbClsExtra= 0;
    windowClass.cbWndExtra= 0;
    windowClass.hInstance= hInstance;
    windowClass.hIcon= LoadIcon(NULL, IDI_APPLICATION); // domyślna ikona
    windowClass.hCursor= LoadCursor(NULL, IDC_ARROW); // domyślny kursor
    windowClass.hbrBackground= NULL; // bez tła
    windowClass.lpszMenuName= NULL; // bez menu
    windowClass.lpszClassName= "MojaKlasa";
    windowClass.hIconSm= LoadIcon(NULL, IDI_WINLOGO); // logo Windows

    // rejestruje klasę okna
    if (!RegisterClassEx(&windowClass))
        return 0;

    // tworzy okno
    hwnd = CreateWindowEx(NULL,
        "MojaKlasa",                  // nazwa klasy
        "Hokej",                      // nazwa aplikacji
        WS_OVERLAPPEDWINDOW | WS_VISIBLE | // styl okna
        WS_SYSMENU | WS_CLIPCHILDREN | // 
        WS_CLIPSIBLINGS,
        100, 100,                      // współrzędne x,y
        800, 600,                      // szerokość, wysokość
        NULL,                          // uchwyty okna nadziednego
        NULL,                          // uchwyty menu
        hInstance,                     // instancja aplikacji
        NULL);                         // bez dodatkowych parametrów

    // sprawdza, czy utworzenie okna nie powiodło się (wtedy wartość hwnd równa NULL)
    if (!hwnd)
        return 0;
```

```
ShowWindow(hwnd, SW_SHOW); // wyświetla okno
UpdateWindow(hwnd); // aktualizuje okno

done = false; // inicjuje zmienną warunku pętli
Initialize();
ShowCursor(FALSE);

// pętla przetwarzania komunikatów
while (!done)
{
    PeekMessage(&msg, hwnd, NULL, NULL, PM_REMOVE);

    if (msg.message == WM_QUIT) // aplikacja otrzymała komunikat WM_QUIT?
    {
        done = true; // jeśli tak, to kończy działanie
    }
    else
    {
        Render(); // rysuje grafikę

        TranslateMessage(&msg); // tłumaczy komunikat i wysyła do systemu
        DispatchMessage(&msg);
    }
}

CleanUp();
ShowCursor(TRUE);

return msg.wParam;
}
```

Nowym elementem w powyższym kodzie jest wywołanie funkcji `ShowCursor()` przez funkcję `WinMain()`. Przekazując funkcji `ShowCursor()` wartość `FALSE` informuje się system Windows, że podczas działania tego programu kursor myszy nie powinien być widoczny. Jako że za pomocą myszy kontroluje się ruch obiektu reprezentującego gracza, kursor myszy nie jest tu do niczego potrzebny. Kończąc działanie program przywraca kursor myszy przekazując funkcji `ShowCursor()` wartość `TRUE`. Jeśli programista o tym zapomni, to pozbawi interfejs użytkownika kursora myszy!

## Podsumowanie

Modelowanie zasad fizyki i wykrywanie zderzeń obiektów (oraz ich obsługa) stanowią rozległą problematykę, którą trudno wyczerpać w ramach jednego rozdziału. Starano się tu przedstawić jedynie podstawowe modele rzeczywistego świata stosowane w grach.

Omówiony w rozdziale program demonstracyjny gry w hokeja należy traktować jako punkt wyjściowy do opracowania kompletnej gry. Trzeba też pamiętać o konieczności opracowania doskonalszego sposobu wykrywania zderzeń gracza i krążka, co pozostało jako ćwiczenie do wykonania.



## Rozdział 20.

# Tworzenie szkieletu gry

Stworzenie dobrego szkieletu gry nie jest łatwym zadaniem. Popularne szkielety gier — takie jak szkielet gry „Quake 3” firmy id Software czy szkielet „Unreal” firmy Epic Games — posiadają niezwykle złożoną architekturę i są rezultatem prac prowadzonych przez kilka lat. W bieżącym rozdziale zbudowany zostanie dość prosty szkielet gry o nazwie *SimpEngine*. Pełen jego kod źródłowy dostępny jest na dysku CD dołączonym do książki.

W rozdziale tym przedstawiona zostanie:

- ◆ ogólna architektura szkieletu SimpEngine;
- ◆ obsługa komunikatów systemu Windows i wejścia;
- ◆ główna pętla gry;
- ◆ kamera i świat gry;
- ◆ problematyka bytów i modeli;
- ◆ problematyka systemu dźwięku;
- ◆ problematyka systemu cząstek.

## Architektura szkieletu SimpEngine

SimpEngine będzie obiektowym szkieletem stosowanym do tworzenia gier. Jako że będzie on stosunkowo nieskomplikowany, służyć będzie do tworzenia niewielkich gier (za to w łatwy i szybki sposób).

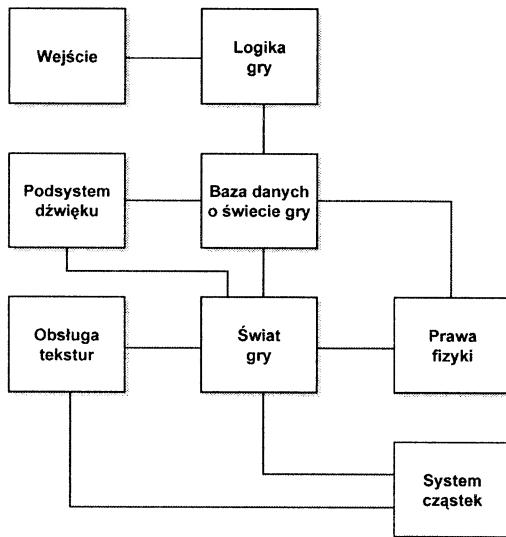
Szkielet SimpEngine składać się będzie z podsystemów przedstawionych na rysunku 20.1.

Podsystem wejścia przesyłać będzie komunikat podsystemowi logiki, który go obsłuży wykonyując główną pętlę gry. W pojedynczym przebiegu tej pętli podsystem logiki gry obsługuje wejście gry, wykonuje obliczenia związane z zastosowaniem zasad fizyki dla obiektów gry, wykrywa i obsługuje zderzenia obiektów, ładuje i usuwa obiekty, zmienia położenie kamery i zajmuje się odtwarzaniem dźwięków.

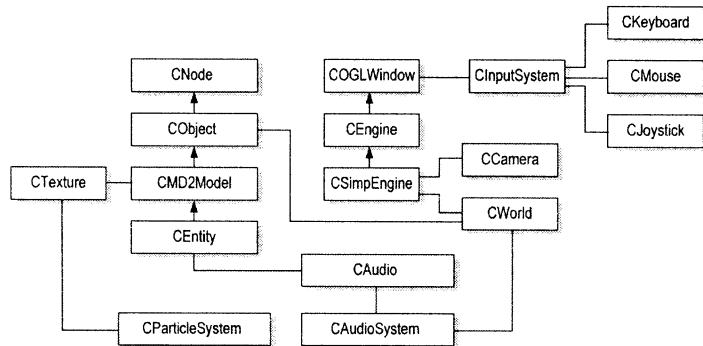
Architekturę tę implementować będą klasy przedstawione na rysunku 20.2.

**Rysunek 20.1.**

Diagram podsystemów szkieletu SimpEngine i ich interakcji

**Rysunek 20.2.**

Klasy szkieletu SimpEngine



Podstawowe klasy szkieletu SimpEngine to klasy COGLWindow i CNode. Klasa COGLWindow obsługuje komunikaty systemu Windows, w podstawowym zakresie wejście programu, a ponadto tworzy okno aplikacji OpenGL. Jest też klasą bazową dla klas CEngine i CSimpEngine. Jak zostanie to pokazane w następnym podrozdziale, klasa CNode reprezentuje węzły drzewa list cyklicznych. Drzewo to reprezentować będzie hierarchię obiektów świata gry i umożliwiać będzie zarządzanie nimi. Przy wykorzystaniu klas COGLWindow i CNode stworzona zostanie cała hierarchia klas, które reprezentować będą obiekty, przeciwników, kontenery obiektów, kamery i inne elementy, o których w przyszłości będzie rozbudowywana gra.

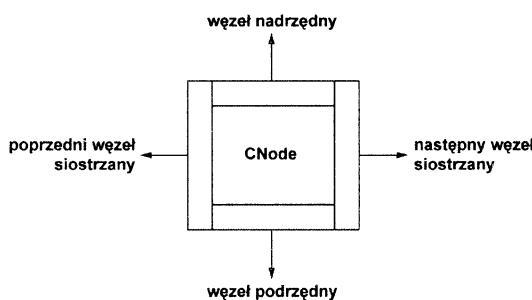
Omówienie tych klas należy rozpocząć od poznania sposobu, w jaki szkielet gry obsługuje obiekty klas COBJekt i CNode.

## Zarządzanie danymi za pomocą obiektów klasy CNode

Klasa CNode reprezentuje węzły drzewa list cyklicznych. Więcej informacji na temat list cyklicznych można odnaleźć w źródłach wymienionych w dodatku A. Na rysunku 20.3

**Rysunek 20.3.**

Połączenia obiektu klasy *CNode*

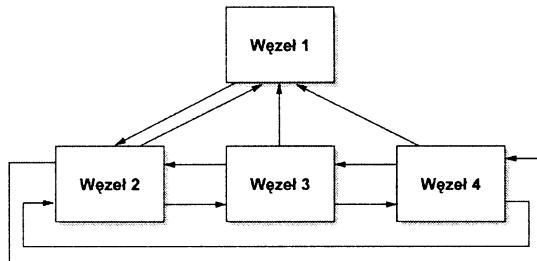


pokazano, że każdy obiekt klasy CNode posiada połączenia z węzłem nadrzędnym, podrzędnym oraz poprzednim i następnym węzłem tego samego poziomu drzewa. Czasami połączenia te mogą tworzyć pętle do tego samego węzła lub posiadać wartość NULL oznaczającą, że połączenie nie istnieje.

Po połączeniu kilku obiektów klasy CNode można uzyskać strukturę podobną do przedstawionej na rysunku 20.4. W przedstawionej na nim hierarchii węzeł 1 stanowi korzeń drzewa i jest węzłem nadrzędnym dla węzła 2, węzła 3 i węzła 4. Węzeł 2 jest także węzłem podrzędnym węzła 1, ale węzeł 3 i węzeł 4 nie są węzłami podrzędnymi. Są one jedynie węzłami siostrzanymi węzła 2. Każdy węzeł nadrzędny (na przykład węzeł 1) może posiadać co najwyżej jeden węzeł podrzędny. Węzeł podrzędny stanowi początek listy cyklicznej węzłów siostrzanych danego poziomu drzewa. Aby przejrzeć wszystkie węzły danego poziomu drzewa, trzeba najpierw znaleźć węzeł podrzędny, a następnie przejrzeć węzły siostrzane będące elementami listy cyklicznej.

**Rysunek 20.4.**

Kolekcja obiektów klasy *CNode*



Poniżej zaprezentowana została definicja klasy CNode, która została umieszczona w pliku *tree.h*:

```
class CNode
{
public:
    // dane
    CNode *parentNode; // węzeł nadrzędny
    CNode *childNode; // węzeł podrzędny
    CNode *prevNode; // poprzedni węzeł siostrzany
    CNode *nextNode; // następny węzeł siostrzany

    bool HasParent() { return (parentNode != NULL); } // czy węzeł posiada węzeł nadrzędny?
    bool HasChild() { return (childNode != NULL); } // czy węzeł posiada węzeł podrzędny?
```

```
// czy węzeł stanowi początek listy węzłów siostrzanych?  
bool IsFirstChild()  
{  
    if (parentNode)  
        return (parentNode->childNode == this);  
    else  
        return false;  
}  
  
// czy węzeł stanowi koniec listy węzłów siostrzanych?  
bool IsLastChild()  
{  
    if (parentNode)  
        return (parentNode->childNode->prevNode == this);  
    else  
        return false;  
}  
  
// umieszcza węzeł na liście cyklicznej danego węzła nadzielnego  
void AttachTo(CNode *newParent)  
{  
    // jeśli węzeł należy już do innej listy, to usuwa go z niej  
    if (parentNode)  
        Detach();  
  
    parentNode = newParent;  
  
    if (parentNode->childNode)  
    {  
        prevNode = parentNode->childNode->prevNode;  
        nextNode = parentNode->childNode;  
        parentNode->childNode->prevNode->nextNode = this;  
        parentNode->childNode->prevNode = this;  
    }  
    else  
    {  
        parentNode->childNode = this; // pierwszy węzeł = węzeł podzienny  
    }  
}  
  
// dodaje węzeł podzienny  
void Attach(CNode *newChild)  
{  
    // jeśli węzeł należy już innej listy, to usuwa go z niej  
    if (newChild->HasParent())  
        newChild->Detach();  
  
    newChild->parentNode = this;  
  
    if (childNode)  
    {  
        newChild->prevNode = childNode->prevNode;  
        newChild->nextNode = childNode;  
        childNode->prevNode->nextNode = newChild;  
        childNode->prevNode = newChild;  
    }  
    else  
        childNode = newChild;  
}
```

```
// usuwa węzeł z listy
void Detach()
{
    // jeśli węzeł ten jest pierwszym węzłem listy,
    // to kolejny węzeł tej listy staje się węzłem podrzednym
    if (parentNode && parentNode->childNode == this)
    {
        if (nextNode != this)
            parentNode->childNode = nextNode;
        else
            parentNode->childNode = NULL;      // brak węzła podrzednego
    }

    // usuwa połączenia
    prevNode->nextNode = nextNode;
    nextNode->prevNode = prevNode;

    // węzeł nie należy już do listy
    prevNode = this;
    nextNode = this;
}

// zlicza węzły
int CountNodes()
{
    if (childNode)
        return childNode->CountNodes() + 1;
    else
        return 1;
}

// konstruktor
CNode()      // tworzy węzeł
{
    parentNode = childNode = NULL;
    prevNode = nextNode = this;
}

// konstruktor
CNode(CNode *node)
{
    // tworzy węzeł i przyłącza go do parametru konstruktora
    parentNode = childNode = NULL;
    prevNode = nextNode = this;
    AttachTo(node);
}

// destruktor
virtual ~CNode()
{
    Detach();          // odłącza węzeł

    while (childNode) // usuwa węzły podrzedne
    {
        delete childNode;
    }
};

};
```

Klasa CNode posiada metody umożliwiające łączenie i odłączanie węzłów, sprawdzanie tego, czy dany węzeł posiada węzeł nadzędny lub podrzędny i taki dalej.

Już na tym etapie łatwo ocenić przydatność klasy CNode, ale swoją użyteczność ujawnia ona w pełni dopiero jako klasa pochodna CObject.

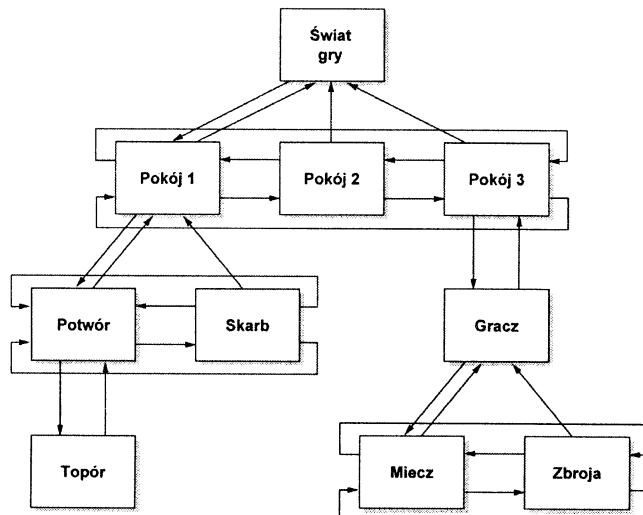
## Zarządzanie obiekty: klasa CObject

Klasa CObject reprezentuje najbardziej podstawowy typ obsługiwany przez szkielet gry i jest klasą pochodną klasy CNode. Dzięki temu obiekty klasy CObject można ze sobą łączyć w hierarchię, którą łatwiej jest zarządzać.

Rysunek 20.5 przedstawia przykładową hierarchię obiektów hipotetycznej gry. Jej korzeń reprezentuje cały świat gry. Na liście jego węzłów podrzędnych umieszczone zostały pomieszczenia gry. W każdym z tych pomieszczeń może znajdować się dowolna liczba skarbów, potworów, graczy i innych obiektów. Poszczególne postacie gry mogą być też wyposażone w różną broń, która tworzy kolejny poziom drzewa.

**Rysunek 20.5.**

Przykładowa  
hierarchia obiektów  
hipotetycznej gry



Prawdziwą potęgę klasa CObject ujawnia jednak dopiero przy rysowaniu lub przemieszczaniu obiektów, a także wykrywaniu ich zderzenia. Przed omówieniem tych operacji należy jednak przedstawić definicję klasy CObject umieszczoną w pliku *object.h*:

```

class CObject : public CNode
{
protected:

    // wyrzuca położenie i prędkość obiektu
    virtual void OnAnimate(scalar_t deltaTime)
    {
        position += velocity * deltaTime;
        velocity += acceleration * deltaTime;
    }
}
  
```

```
// rysuje obiekt dla danego położenia kamery
virtual void OnDraw(CCamera *camera) {}

// wykrywa zderzenia
virtual void OnCollision(CObject *collisionObject) {}

// obsługuje zderzenia
virtual void OnPrepare()
{
    ProcessCollisions(FindRoot()); // wykrywa zderzenia
    // począwszy od obiektu w korzeniu drzewa
}

public:
    CVector position;           // położenie obiektu
    CVector velocity;          // prędkość obiektu
    CVector acceleration;      // przyspieszenie
    scalar_t size;             // promień sfery ograniczeń

CObject() { isDead = false; }
~CObject() {}

virtual void Load() {}
virtual void Unload() {}

// rysuje obiekt
void Draw(CCamera *camera)
{
    // umieszcza macierz modelowania na stosie
    glPushMatrix();
    OnDraw(camera);           // rysuje obiekt
    if (HasChild())           // rysuje obiekty podrzędne
        ((CObject*)childNode)->Draw(camera);
    glPopMatrix();

    // rysuje obiekty siostrzane
    if (HasParent() && !IsLastChild())
        ((CObject*)nextNode)->Draw(camera);
}

// animuje obiekt
void Animate(scalar_t deltaTime)
{
    OnAnimate(deltaTime);     // animuje dany obiekt

    // animuje obiekty podrzędne
    if (HasChild())
        ((CObject*)childNode)->Animate(deltaTime);

    // animuje obiekty siostrzane
    if (HasParent() && !IsLastChild())
        ((CObject*)nextNode)->Animate(deltaTime);

    if (isDead)
        delete this;
}
```

```

// wykrywa zderzenia
void ProcessCollisions(CObject *obj)
{
    // jeśli sfera ograniczeń tego obiektu koliduje ze sferą obiektu obj
    // i obiekt obj nie jest tym samym obiektem
    if (((obj->position - position).Length() <= (obj->size + size)) &&
        (obj != ((CObject*)this)))
    {
        OnCollision(obj);      // obsługuje zderzenie z obiektem obj

        // wykrywa zderzenia obiektu obj z obiektami podlegającymi
        if (HasChild())
            ((CObject*)childNode)->ProcessCollisions(obj);

        // wykrywa zderzenia obiektu obj z obiektami siostrzonymi
        if (HasParent() && !IsLastChild())
            ((CObject*)nextNode)->ProcessCollisions(obj);
    }

    // jeśli obiekt obj posiada obiekty podlegające, należy sprawdzić zderzenia z nimi
    if (obj->HasChild())
        ProcessCollisions((CObject*)(obj->childNode));

    // jeśli obiekt obj posiada obiekty siostrzane, należy sprawdzić zderzenia z nimi
    if (obj->HasParent() && !obj->IsLastChild())
        ProcessCollisions((CObject*)(obj->nextNode));
}

// przygotowuje obiekt
void Prepare()
{
    OnPrepare();                // przygotowuje dany obiekt

    if (HasChild())            // i jego obiekty podlegające
        ((CObject*)childNode)->Prepare();

    if (HasParent() && !IsLastChild()) // oraz siostrzane
        ((CObject*)nextNode)->Prepare();
}

// znajduje korzeń drzewa list cyklicznych
CObject *FindRoot()
{
    // jeśli dany obiekt posiada węzeł nadążający,
    // to zwraca jego korzeń
    if (parentNode)
        return ((CObject*)parentNode)->FindRoot();

    return this;
}:

```

W dostępnej publicznie części klasy umieszczone zostały metody `Prepare()`, `Animate()`, `ProcessCollisions()` i `Draw()`. Każda z nich wykonuje odpowiednie operacje na danym obiekcie, a następnie na jego obiektach podlegających. Dobrym tego przykładem jest kod źródłowy metody `CObject::Animate()`:

```

// animuje obiekt
void Animate(scalar_t deltaTime)
{
    OnAnimate(deltaTime);           // animuje dany obiekt

    // animuje obiekty podrzędne
    if (HasChild())
        ((CObject*)childNode)->Animate(deltaTime);

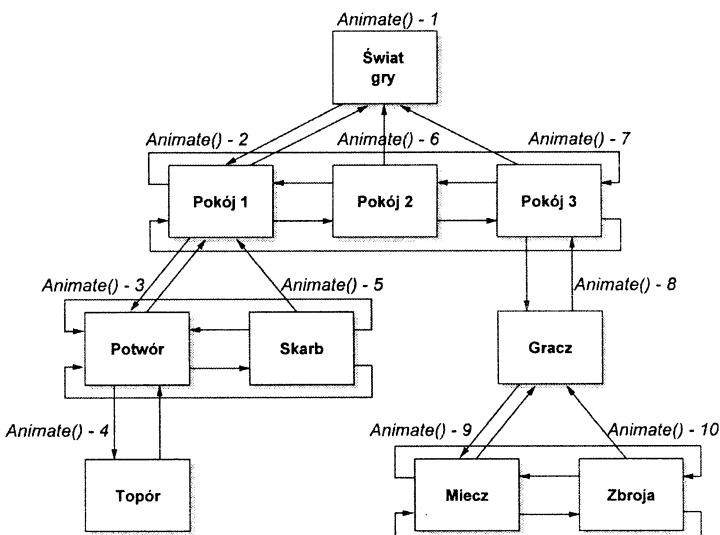
    // animuje obiekty siostrzane
    if (HasParent() && !IsLastChild())
        ((CObject*)nextNode)->Animate(deltaTime);

    if (isDead)
        delete this;
}

```

Każdy z obiektów klasy CObject i jej pochodnych posiada zestaw własnych metod wirtualnych o dostępie chronionym i nazwach On\*(), takich jak OnPrepare(), OnAnimate(), OnCollision() i OnDraw(). Za każdym razem, gdy na rzecz obiektu wywoływana jest jedna z metod o dostępie publicznym (jak na przykład Animate()), to odpowiednia metoda On\*() umożliwia wykonanie specjalizowanego kodu dla danej klasy obiektu. W przypadku przedstawionej wyżej metody Animate() najpierw wywoływana jest metoda OnAnimate(), która wykonuje rzeczywistą animację tego obiektu, a następnie rekurencyjnie wywoływana jest metoda Animate() dla jego obiektów podrzędnych i siostrzanych. Ostatni blok kodu w metodzie Animate() pozwala usunąć obiekt z hierarchii, jeśli zachodzi taka konieczność. Rysunek 20.6 ilustruje wywołanie metody Animate() na rzecz obiektu znajdującego się w korzeniu drzewa hipotetycznej hierarchii.

**Rysunek 20.6.**  
Przykład działania  
metody Animate()  
wywołanej dla  
korzenia drzewa



Metodę Animate() stosuje się zwykle po to, aby wyznaczyć położenie obiektu w kolejnej klatce animacji, a metodę Prepare(), aby zmienić stan obiektu. Zderzenia obiektu wykrywa metoda ProcessCollisions(), a metoda Draw() służy do narysowania reprezentacji graficznej obiektu.

Ponieważ metody `OnPrepare()`, `OnAnimate()` i `OnDraw()` są wirtualne, to każda klasa pochodna klasy `CObject` musi dostarczyć własnej ich implementacji. W ten sposób tworząc nowe klasy pochodne klasy `CObject` można uzyskać prosty i efektywny sposób rozbudowywanego szkieletu gry o nowe obiekty.

Należy też zwrócić uwagę na to, że metoda `Draw()` ujmuje wywołania metody `Draw()` dla obiektów podlegających w blok pary wywołań funkcji `glPushMatrix()` i `glPopMatrix()`, których zastosowanie omówione zostało w rozdziale 5. Wspomniano tam o możliwości tworzenia hierarchii obiektów, w których współrzędne obiektów podlegających określone są względem obiektów nadległych. Na przykład głowa robota może być obiektem podlegającym jego korpusowi i posiadać współrzędne  $(0.0, 5.0, 0.0)$  względem niego.

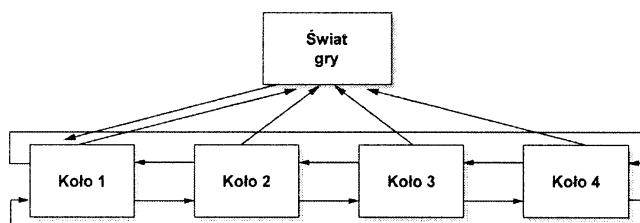
Można teraz założyć, że rysuje się samochód i jego cztery koła. Koła dołączyć należy jako obiekty podlegające do nadwozia samochodu. Wystarczy więc określić ich współrzędne w stosunku do reszty samochodu — na przykład: lewe, przednie koło  $(2.0, 0.0, 2.0)$ , prawe, przednie koło  $(-2.0, 0.0, 2.0)$  i tak dalej — a hierarchia obiektów klasy `CObject` sama zadba o ich właściwe narysowanie.

Rozwiązanie takie ilustruje poniższy fragment kodu:

```
CObject *body;
CObject *tire1, *tire2, *tire3, *tire4;
...
tire1->AttachTo(body); // lub: body->Attach(tire1)
tire2->AttachTo(body);
tire3->AttachTo(body);
tire4->AttachTo(body);
...
body->Draw(...);
```

Kod ten utworzy hierarchię obiektów pokazaną na rysunku 20.7.

**Rysunek 20.7.**  
Hierarchia obiektów  
reprezentująca  
samochód i jego koła



## Trzon szkieletu SimpEngine

Zasadniczą część szkieletu stanowi klasa `COGLWindows`, która tworzy okno grafiki OpenGL i obsługuje komunikaty wysypane do aplikacji przez system Windows. Dla każdego z obsługiwanych komunikatów klasa `COGLWindow` definiuje osobną metodę wirtualną. Na przykład komunikatowi `WM_SIZE` odpowiada metoda wirtualna `OnSize()`. Metody te wywoływane są przez procedurę okienkową `WndProcOGL`, gdy otrzyma ona odpowiedni rodzaj komunikatu. A oto definicja klasy `COGLWindow`:

```
class COGLWindow
{
protected:
    HWND hWnd;           // uchwyt okna
    HDC hDC;             // kontekst urządzenia
    HPALETTE hPalette;   // paleta
    HGLRC hGLRC;         // kontekst tworzenia grafiki

private:
    // procedura okienkowa
    friend LRESULT APIENTRY WndProcOGL(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

    void SetupPixelFormat();           // konfiguruje format pikseli
    void SetupPalette();              // konfiguruje paletę

    // metody obsługiwane komunikatów systemu Windows
    bool Create();                  // WM_CREATE
    void Destroy();                 // WM_DESTROY
    void PaletteChanged(WPARAM wParam); // WM_PALETTECHANGED
    BOOL QueryNewPalette();         // WM_QUERYNEWPALETTE
    void Paint();                   // WM_PAINT
    void Size();                    // WM_SIZE
    void MouseMove(int x, int y);   // WM_MOUSEMOVE

    int GetMouseX(LPARAM lParam);   // zwracają współrzędne myszy
    int GetMouseY(LPARAM lParam);   // zwracają znormalizowane współrzędne myszy

    float GetNormalizedPosX(LPARAM lParam); // zwracają znormalizowane współrzędne myszy
    float GetNormalizedPosY(LPARAM lParam); // z przedziału od -1.0 do 1.0

    int iPrevWidth;
    int iPrevHeight;
    void BeginFullScreen(int w, int h, int b);
    void EndFullScreen();

public:
    int width;                  // rozmiary okna
    int height;                 // rozmiary okna
    int centerX;                // współrzędne środka okna
    int centerY;                // współrzędne środka okna
    int bits;                   // liczba bitów na piksel
    int aspect;                 // współczynnik proporcji okna
    int mouseX;                 // pozycja myszy
    int mouseY;                 // pozycja myszy
    bool fullscreen;             // tryb pełnoekranowy?

    float mouseSensitivity;     // czułość myszy

    bool useDInput;             // wartość true, jeśli korzysta z dinput
    CInputSystem *inputSystem;   // system wejścia

protected:
    virtual bool OnCreate() { return true; }
    virtual bool OnClose() { return true; }
    virtual void OnSize() { }
    virtual void OnMouseDownL(float x, float y) { }
    virtual void OnMouseDownR(float x, float y) { }
```

```

virtual void OnMouseUpL() { }
virtual void OnMouseUpR() { }
virtual void OnMouseMove(int x, int y, int centerX, int centerY) { }
virtual void OnMouseMove(int deltaX, int deltaY) { }
virtual void OnMouseDragL(int x, int y, int dx, int dy) { }
virtual void OnMouseDragR(int x, int y, int dx, int dy) { }
virtual void OnCommand(WORD wNotifyCode, WORD wID, HWND hWndCtrl) { }
virtual void OnContextMenu(HWND hWnd, int x, int y) { }
virtual void OnKeyUp(int nVirtKey) { }
virtual void OnInitMenu(HMENU hMenu) { }
virtual void OnKeyDown(int nVirtKey) { }
virtual void OnChar(char c) { }

public:
COGLWindow() {}
COGLWindow(const char *szName, bool fscreen, int w, int h, int b);
virtual ~COGLWindow();

// poniższa metoda musi być wywołana, zanim użyta zostanie jakakolwiek inna
static bool RegisterWindow(HINSTANCE hInst);
};


```

## System wejścia

Klasa COGLWindow przechowuje także referencję obiektu klasy CInputSystem, który reprezentuje podsystem wejścia wykorzystujący interfejs DirectInput do odczytu bieżącego stanu klawiatury, myszy i manipulatora. Klasa systemu wejścia będzie zawierać metody pozwalające sprawdzić, czy klawisz został naciśnięty lub zwolniony, jak zmieniła swoją pozycję mysz, a także to, jaki jest stan jej przycisków, o ile przesunął się drążek manipulatora i to, czy użytkownik skorzystał z jego przycisków. Obiektem klasy CInputSystem będzie zarządzać wyłącznie klasa COGLWindows (tworzyć go i usuwać). Poniżej przedstawiona została definicja klasy CInputSystem:

```

class CInputSystem
{
public:
    CInputSystem() { m_pKeyboard = NULL; m_pMouse = NULL; m_pJoystick = NULL; }
    ~CInputSystem() {}
    bool Initialize(HWND hwnd, HINSTANCE appInstance, bool isExclusive, DWORD flags = 0);
    bool Shutdown();

    void AcquireAll();
    void UnacquireAll();

    CKeyboard *GetKeyboard() { return m_pKeyboard; }
    CMouse    *GetMouse()   { return m_pMouse; }
    CJoystick *GetJoystick() { return m_pJoystick; }

    bool Update(); // aktualizuje stan urządzeń ...

    bool KeyDown(char key) { return (m_pKeyboard && m_pKeyboard->KeyDown(key)); }
    bool KeyUp(char key) { return (m_pKeyboard && m_pKeyboard->KeyUp(key)); } ...

    bool ButtonDown(int button) { return (m_pMouse && m_pMouse->ButtonDown(button)); }
    bool ButtonUp(int button) { return (m_pMouse && m_pMouse->ButtonUp(button)); }
};


```

```
void GetMouseMovement(int &dx, int &dy)
{ if (m_pMouse) m_pMouse->GetMovement(dx, dy); }

private:
CKeyboard *m_pKeyboard; // obiekt klawiatury
CMouse   *m_pMouse;    // obiekt myszy
CJoystick *m_pJoystick; // obiekt manipulatora

LPDIRECTINPUT8 m_pDI;
};
```

## Klasa CEngine

Klasa CEngine jest klasą pochodną klasy COGLWindow i dostarcza implementację pętli przetwarzania komunikatów systemu Windows, główną pętlę gry oraz metodę obsługi informacji wejścia otrzymywanych z podsystemu wejścia. A oto definicja klasy CEngine:

```
class CEngine : public COGLWindow
{
private:

protected:
CHiResTimer *timer;// licznik czasu o dużej rozdzielczości

virtual void GameCycle(float deltaTime); // główna pętla gry

virtual void OnPrepare() {} // konfiguruje OpenGL

// klasy pochodne muszą dostarczać własną implementację tej metody
virtual CCamera *OnGetCamera() { return NULL; }

// klasy pochodne muszą dostarczać własną implementację tej metody
virtual CWorld *OnGetWorld() { return NULL; }

virtual void CheckInput(float deltaTime); // pozyskuje dane wejściowe

public:
CEngine() {}
CEngine(const char *szName, bool fscreen, int w, int h, int b) :
COGLWindow(szName, fscreen, w, h, b) :
~CEngine() {}
LRESULT EnterMessageLoop();
};
```

Metoda EnterMessageLoop() zawiera typową pętlę przetwarzania komunikatów systemu Windows i posługuje się licznikiem czasu klasy CHiResTimer. Poniżej zaprezentowany został kod metody CEngine::EnterMessageLoop():

```
LRESULT CEngine::EnterMessageLoop()
{
MSG msg; // odebrany komunikat
timer = new CHiResTimer; // licznik czasu

timer->Init(); // inicjuje licznik czasu
```

```

for (;;)
{
    // wykonuje pojedynczy cykl gry
    GameCycle(timer->GetElapsedSeconds(1));
    while (PeekMessage (&msg, NULL, 0, 0, PM_NOREMOVE))
    {
        // sprawdza obecność komunikatów
        if (!GetMessage (&msg, NULL, 0, 0))
        {
            delete timer;           // usuwa licznik
            return msg.wParam;     // przekazuje sterowanie do systemu
        }

        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }

    delete timer;           // usuwa licznik

    return msg.wParam;     // przekazuje sterowanie do systemu
}

```

W każdym przebiegu pętli metoda EnterMessageLoop() wykonuje pojedynczy cykl gry i sprawdza, czy aplikacja otrzymała komunikat o zakończeniu pracy. Metoda cyklu gry, GameCycle(), otrzymuje jako parametr czas, który upłynął od poprzedniego cyklu. Czas ten wykorzystywany jest do przeprowadzenia obliczeń z wykorzystaniem zasad fizyki oraz wykrywania zderzeń.

## Cykl gry

Na typowy cykl gry składają się następujące etapy:

- ◆ pobranie informacji wejściowej;
- ◆ zmiana położenia gracza;
- ◆ zastosowanie sztucznej inteligencji;
- ◆ obliczenia z zastosowaniem zasad fizyki;
- ◆ odtwarzanie dźwięku;
- ◆ narysowanie grafiki.

W przypadku szkieletu SimpEngine cykl gry nie będzie aż tak przejrzysty ze względu na istniejącą hierarchię obiektów i konieczność wywołania metod Prepare(), Animate() i Draw() klasy CObject. Ilustruje to poniższy kod metody CEngine::GameCycle():

```

void CEngine::GameCycle(float deltaTime)
{
    CCamera *camera = OnGetCamera(); // pobiera kamerę
    CWorld *world = OnGetWorld();   // i korzeń hierarchii

    // sprawdza informację wejścia
    CheckInput(deltaTime);

```

```
// konfiguruje OpenGL  
OnPrepare();  
  
// przygotowuje obiekty i obsługuje zderzenia  
world->Prepare();  
  
// zmienia położenie i orientację kamery  
camera->Animate(deltaTime);  
  
// zmienia położenie i orientację obiektów  
world->Animate(deltaTime);  
  
// rysuje obiekty  
world->Draw(camera);  
  
// przełącza bufory  
SwapBuffers(hDC);  
}
```

Etapy związane z obsługą kamery i obiektów gry omówione zostaną wkrótce. Teraz wystarczy, jeśli zapamiętana zostanie kolejność etapów tworzących cykl gry.

## Obsługa wejścia

Jak łatwo zauważycią analizując kod metody GameCycle(), stan urządzeń wejścia określany jest za pomocą metody CheckInput(), która używa obiektu klasy CInputSystem zarządzanego przez klasę COGLWindow. Zamieszczony poniżej kod metody CheckInput() sprawdza, czy nastąpiło zdarzenie związane z urządzeniami wejścia (na przykład naciśnięcie klawisza *Esc*) i wywołuje odpowiednią metodę jego obsługi:

```
void CEngine::CheckInput(float deltaTime)  
{  
    static float buttonDelta = 0.0f; // okres czasu, który musi upływać, zanim może być  
                                   // wykryte następne naciśnięcie przycisku myszy  
  
    int mouseDeltaX, mouseDeltaY; // zmiana położenia myszy  
  
    // zmniejsza czas do następnego wykrywalnego naciśnięcia przycisku myszy  
    buttonDelta -= deltaTime;  
  
    if (buttonDelta < 0.0f)  
        buttonDelta = 0.0f;  
  
    // aktualizuje urządzenia  
    inputSystem->Update();  
  
    // pobiera informacje o ruchu myszy  
    inputSystem->GetMouseMovement(mouseDeltaX, mouseDeltaY);  
    OnMouseMove(mouseDeltaX, mouseDeltaY);  
  
    // naciśnięcie klawisza W  
    if (inputSystem->KeyDown(DIK_W))  
        OnKeyDown(VK_UP);  
    // naciśnięcie klawisza S  
    if (inputSystem->KeyDown(DIK_S))  
        OnKeyDown(VK_DOWN);
```

```

// naciśnięcie klawisza A
if (inputSystem->KeyDown(DIK_A))
    OnKeyDown(VK_LEFT);
// naciśnięcie klawisza D
if (inputSystem->KeyDown(DIK_D))
    OnKeyDown(VK_RIGHT);
// naciśnięcie klawisza +
if (inputSystem->KeyDown(DIK_ADD))
    OnKeyDown(VK_ADD);
// naciśnięcie klawisza -
if (inputSystem->KeyDown(DIK_SUBTRACT))
    OnKeyDown(VK_SUBTRACT);
// naciśnięcie klawisza Esc
if (inputSystem->KeyDown(DIK_ESCAPE))
    OnKeyDown(VK_ESCAPE);

// naciśnięcie lewego przycisku myszy
if (inputSystem->ButtonDown(0))
{
    // jeśli naciśnięcie może być już rozróżnione
    if (buttonDelta == 0.0f)
    {
        OnMouseDownL(0,0);
        buttonDelta = 0.5f;    // pół sekundy do następnego naciśnięcia
    }
}
}

```

Metoda ta wywołuje najpierw metodę `CInputSystem::Update()` w celu aktualizacji informacji o stanie urządzeń wejściowych używanych przez aplikację (na przykład klawiatury i myszy). Następnie pobiera informację o zmianie położenia myszy i przekazuje ją metodzie obsługi `OnMouseMove()`. Sprawdza też, czy zostały naciśnięte wybrane klawisze przekazując odpowiednie stałe `DIK_` metodzie `CInputSystem::KeyDown()`. A także — czy wybrany został jeden lub więcej przycisków myszy za pomocą metody `CInputSystem::ButtonDown()`. W przypadku wykrycia naciśnięcia jednego z interesujących klawiszy wywołuje metodę `OnKeyDown()` przekazując jej jako parametr wirtualny kod klawisza. Gdy naciśnięty został lewy przycisk myszy, wywołuje metodę `OnMouseButton()`.

## Klasa CSimpEngine

Klasa `CSimpEngine` stanowi klasę pochodną klasy `CEngine` i reprezentuje kluczową część szkieletu `SimpEngine`. Pokazana poniżej definicja klasy pokazuje, że zawiera ona metodę obsługi wejścia `OnKeyDown()`, która może być modyfikowana w zależności od potrzeb tworzonej aplikacji. Przechowuje także obiekty klas `CCamera` i `CWorld` reprezentujące odpowiednio kamerę i świat gry.

```

class CSimpEngine : public CEngine
{
private:
    CCamera *gameCamera; // kamera
    CWorld *gameWorld; // świat gry

protected:
    CCamera *OnGetCamera() { return gameCamera; }
    CWorld *OnGetWorld() { return gameWorld; }
}

```

```
void OnPrepare();
void OnMouseDownL(float x, float y);
void OnMouseMove(int deltaX, int deltaY);
void OnMouseMove(int x, int y, int centerX, int centerY);
void OnKeyDown(int nVirtKey);

public:
CSimpEngine()
{
gameCamera = new CCamera;
gameWorld = new CWorld;
}

CSimpEngine(const char *szName, bool fscreen, int w, int h, int b) :
CEngine(szName, fscreen, w, h, b)
{
gameCamera = new CCamera;
gameWorld = new CWorld(gameCamera);
gameCamera->centerX = centerX;
gameCamera->centerY = centerY;
gameWorld->SetScreen(w,h);
}

~CSimpEngine()
{
delete gameWorld;
delete gameCamera;
gameWorld = NULL;
gameCamera = NULL;
}:
```

Obiekty reprezentujące kamerę i świat gry pojawiały się już wcześniej podczas omawiania szkieletu SimpEngine. Najwyższa więc pora, aby przyjrzeć im się bliżej.

## Kamera

Klasa CCamera wykorzystywana jest przez szkielet gry do określenia sposobu widzenia świata gry przez jej użytkownika. Zdefiniowana jest w następujący sposób:

```
class CCamera
{
private:
    // poniższe składowe określają położenie i orientację kamery
    // i wykorzystywane są przez metody MoveTo i LookTo
    CVector initPosition, finalPosition;
    CVector initLookAt, finalLookAt;

    CVector lookAtVel;           // prędkość orientacji kamery
    CVector lookAtAccel;         // przyspieszenie orientacji kamery

    void UpdateLookAt();
    void UpdateMoveTo();
```

```

public:
    CVector position;           // położenie kamery
    CVector velocity;          // prędkość kamery
    CVector acceleration;      // przyspieszenie kamery
    CVector lookAt;            // wektor orientacji

    // wektory kierunków: w góre, na przód i w prawo
    CVector up;
    CVector forward;
    CVector right;

    // kąty pochylenia kamery względem osi y i x
    float yaw;
    float pitch;

    int screenWidth, screenHeight;
    int centerX, centerY;

    CCamera();
    CCamera(int width, int height) {}
    CCamera(CVector *look);
    CCamera(CVector *pos, CVector *look);
    ~CCamera();

    void LookAt(CObject *object);           // skierowanie kamery na obiekt
    void LookAtNow(CObject *object);        // natychmiastowe skierowanie kamery na obiekt
    void MoveTo(CObject *object);           // zbliżenie kamery na obiekt
    void MoveToNow(CObject *object);        // natychmiastowe zbliżenie kamery na obiekt
    // natychmiastowa zmiana położenia kamery
    // do punktu o podanych współrzędnych
    void MoveToNow(scalar_t x, scalar_t y, scalar_t z);

    void RotateYaw(scalar_t radians);       // obrót kamery względem osi y
    void RotatePitch(scalar_t radians);     // obrót kamery względem osi x
    void RotateRoll(scalar_t radians);      // obrót kamery względem osi z

    // równania ruchu kamery
    void Animate(scalar_t deltaTime);
};


```

Posługiwanie się obiektem kamery jest bardzo proste, ponieważ odbywa się na podstawie danych odbieranych przez system wejścia. Najważniejszą metodą klasy CCamerę jest metoda `CCamera::Animate()`, która działa podobnie do metody `CObject::Animate()`, gdyż służy do przemieszczania i orientowania kamery. Ruch kamery opisany jest za pomocą prędkości i przyspieszenia, natomiast jej orientacja za pomocą kątów pochylenia względem osi y i osi x. Wektor `lookAt` definiuje punkt, w który skierowana jest kamera. Jeśli kamera będzie skierowana wzdułż ujemnej części osi z, to wektor `lookAt` będzie równy (0.0, 0.0, -1.0). Poniżej zaprezentowany został kod źródłowy metody `CCamera::Animate()`:

```

void CCamerę::Animate(scalar_t deltaTime)
{
    if ((yaw >= 360.0f) || (yaw <= -360.0f))
        yaw = 0.0f;

```

```
if (pitch > 60.0f) // granice pochylenia kamery względem osi x
    pitch = 60.0f;
if (pitch < -60.0f)
    pitch = -60.0f;

float cosYaw = (scalar_t)cos(DEG2RAD(yaw));
float sinYaw = (scalar_t)sin(DEG2RAD(yaw));
float sinPitch = (scalar_t)sin(DEG2RAD(pitch));

float speed = velocity.z * deltaTime;           // prędkość naprzód lub do tyłu
float strafeSpeed = velocity.x * deltaTime; // prędkość w lewo lub prawo

if (speed > 15.0)                                // ograniczenie prędkości
    speed = 15.0;                                 // naprzód
if (strafeSpeed > 15.0)                         // w prawo
    strafeSpeed = 15.0;                           // do tyłu
if (speed < -15.0)                               // do tyłu
    speed = -15.0;                             // w lewo
if (strafeSpeed < -15.0)
    strafeSpeed = -15.0;                         // w lewo

if (velocity.Length() > 0.0)                     // wpływ tarcia
    acceleration = -velocity * 1.5f;

velocity += acceleration*deltaTime;

// oblicza nowe położenie kamery
// na podstawie prędkości ruchu wzduż osi z i osi x
position.x += float(cos(DEG2RAD(yaw + 90.0)))*strafeSpeed;
position.z += float(sin(DEG2RAD(yaw + 90.0)))*strafeSpeed;
position.x += float(cosYaw)*speed;
position.z += float(sinYaw)*speed;

// wyznacza nowy wektor orientacji kamery
lookAt.x = float(position.x + cosYaw);
lookAt.y = float(position.y + sinPitch);
lookAt.z = float(position.z + sinYaw);

// używa funkcji gluLookAt do umieszczenia kamery w nowym położeniu
// i jej zorientowania
gluLookAt(position.x, position.y, position.z,
           lookAt.x, lookAt.y, lookAt.z,
           0.0, 1.0, 0.0);
}
```

Kończąc swoje działanie metoda `Animate()` określa nową pozycję i orientację kamery za pomocą funkcji `gluLookAt()`. Jeśli weźmie się pod uwagę cykl gry, można zauważyć, że kamera jest pierwszym obiektem, dla którego wykonywana jest metoda `Animate()`. Powodem tego jest właśnie wywołanie funkcji `gluLookAt()`. Zanim narysowane zostaną obiekty świata gry, musi najpierw zostać skonfigurowana kamera, aby OpenGL mógł ustalić prawidłowy sposób prezentacji grafiki.

## Świat gry

Klasa CWorld definiuje świat gry prezentowany przez szkielet SimpEngine. Jak już pokazano, metoda CEngine::GameCycle() korzysta z klasy CWorld, aby wywołać metody Prepare(), Animate() i Draw() dla wszystkich obiektów świata gry. Zwykle obiekt klasy CWorld reprezentuje określony poziom gry. Jeśli gra dysponuje wieloma poziomami, to obiekty klasy CWorld można dodatkowo umieścić w pewnym kontenerze i w ten sposób rozbudować hierarchię obiektów o kolejny szczebel. Klasa CWorld przechowuje także obiekt podsystemu dźwięku należący do klasy CAudioSystem, którego zadaniem jest załadowanie i odtwarzanie odpowiednich dźwięków i podkładu muzycznego. A oto definicja klasy CWorld:

```
class CWorld
{
protected:
    void OnAnimate(float deltaTime);
    void OnDraw(CCamera *camera);
    void OnPrepare();

public:
    HWND hwnd;

    CTerrain *terrain;           // teren
    CCamera *camera;             // kamera
    CAudioSystem *audioSystem;   // podsystem dźwięku

    CWorld();
    CWorld(CCamera *c);
    ~CWorld();

    void LoadWorld();            // ładuje obiekty świata gry
    void UnloadWorld();          // usuwa obiekty świata gry

    // stosuje zasady fizyki dla wszystkich obiektów świata gry
    void Animate(float deltaTime);

    // rysuje wszystkie obiekty świata gry
    void Draw(CCamera *camera);

    void Prepare() { OnPrepare(); }
};
```

Na zasadzie przykładu umieszczona została w definicji klasy CWorld referencja obiektu klasy CTerrain reprezentującego teren, po którym porusza się gracz i często stanowi korzeń hierarchii obiektów.

## Obsługa modeli

Szkielet SimpEngine uzupełniony zostanie jeszcze o możliwość dodawania modeli w formacie *MD2* za pomocą obiektów klasy CMD2Model. Klasa ta będzie klasą pochodną klasy CObject, ponieważ modele *MD2* należy traktować tak samo jak inne obiekty świata.

Ponieważ pełen kod klasy `CMD2Model` przedstawiony został w rozdziale 18., nie będzie tutaj powtórzony. Należy pamiętać jedynie, aby zmienić wiersz

```
class CMD2Model  
na  
class CMD2Model : public CObject
```

W ten sposób klasa `CMD2Model` zostanie włączona do hierarchii dziedziczenia klasy bazowej `CObject`.

Klasa `CMD2Model` stanowi jedynie punkt wyjściowy, ponieważ szkielet gry będzie wykorzystywać raczej obiekty klasy `CEntity`. Klasa ta będzie reprezentować animowane modele *MD2* i stanowić klasę pochodną klasy `CObject`.

Klasa `CEntity` przechowuje dodatkowo informację o orientacji modelu (obrót względem osi y), dźwięki, które może on wydawać w postaci obiektów `CAudio` (klasę tę omówimy wkrótce), bieżącą klatkę początkową i końcową animacji oraz prędkość animacji. Klasa `CEntity` posiada następującą definicję:

```
class CEntity : public CMD2Model  
{  
protected:  
    void OnAnimate(float deltaTime);  
    void OnDraw(CCamera *camera);  
    void OnCollision(CObject *collisionObject);  
    void OnPrepare();  
  
public:  
    float direction;           // orientacja modelu (kąt obrotu względem osi y w radianach)  
    CAudio *entitySound;      // dźwięki wydawane przez model  
                            // w obecnej wersji tylko jeden dźwięk  
  
    CEntity();  
    ~CEntity();  
  
    int stateStart, stateEnd; // klatka początkowa i końcowa animacji  
    float deltaT;            // używana podczas interpolacji klatek  
    float animSpeed;          // tempo animacji  
  
    void LoadAudio(CAudioSystem *audioSystem, char *filename, bool is3DSound);  
};
```

## System dźwięku

Zadaniem klasy `CAudioSystem` jest tworzenie, odtwarzanie i zarządzanie dźwiękami i muzyką za pomocą interfejsu DirectX Audio i obiektów klasy `CAudio` reprezentujących pojedyncze efekty dźwiękowe lub motywy muzyczne.

Definicja klasy `CAudioSystem` przedstawia się następująco:

```

class CAudio
{
private:
    IDirectMusicSegment8 *dmusicSegment;      // segment

    // Bufor efektu przestrzennego

    IDirectSound3DBuffer *ds3DBuffer;

    bool is3DSound;           // wartość true, jeśli używany jest efekt przestrzenny

protected:

public:
    CAudio() { dmusicSegment = NULL; ds3DBuffer = NULL; is3DSound = false; }
    ~CAudio()
    {
        if (dmusicSegment != NULL)
        {
            dmusicSegment->Release();
            dmusicSegment = NULL;
        }

        if (ds3DBuffer != NULL)
        {
            ds3DBuffer->Release();
            ds3DBuffer = NULL;
        }
    }

    void SetSegment(IDirectMusicSegment8 *seg) { dmusicSegment = seg; }
    IDirectMusicSegment8 *GetSegment() { return dmusicSegment; }

    void Set3DBuffer(IDirectSound3DBuffer *dsBuff);
    IDirectSound3DBuffer *Get3DBuffer() { return ds3DBuffer; }

    bool Is3DSound() { return is3DSound; }
    void Set3DSound(bool b) { is3DSound = b; }

    void Set3DParams(float minDistance, float maxDistance);
    void Set3DPos(float x, float y, float z);
};

```

Klasa CAudio korzysta zarówno z interfejsu IDirectMusicSegment8, jak i IDirectSound3DBuffer8. Jeśli jej obiekt reprezentuje tylko podkład muzyczny, to wystarczy jedynie załadować odpowiedni segment, a bufor efektu przestrzennego nie jest wykorzystywany.

Klasa CAudioSystem zarządza obiektami klasy CAudio. Oto jej definicja:

```

class CAudioSystem
{
private:
    IDirectMusicLoader8 *dmusicLoader;          // obiekt ładujący
    IDirectMusicPerformance8 *dmusicPerformance; // obiekt wykonania
    IDirectMusicAudioPath8 *dmusic3DAudioPath;   // ścieżka dźwięku
    IDirectSound3DListener8 *ds3DListener;       // obiekt słuchacza

    DS3DLISTENER dsListenerParams;             // właściwości słuchacza

```

```
public:  
    CAudioSystem();  
    ~CAudioSystem();  
  
    bool InitDirectXAudio(HWND hwnd);  
    IDirectSound3DBuffer8 *Create3DBuffer();  
    CAudio *Create(char *filename, bool is3DSound);  
    IDirectMusicSegment8 *CreateSegment(char *filename, bool is3DSound);  
  
    void Play(CAudio *audio, DWORD numRepeats);  
    void Stop(CAudio *audio);  
  
    void PlaySegment(IDirectMusicSegment8 *dmSeg, bool is3DSound, DWORD numRepeats);  
    void StopSegment(IDirectMusicSegment8 *dmSeg);  
    void Shutdown();  
  
    void SetListenerPos(float cameraX, float cameraY, float cameraZ);  
    void SetListenerRolloff(float rolloff);  
    void SetListenerOrientation(float forwardX, float forwardY, float forwardZ,  
        float topX, float topY, float topZ)  
    {  
        ds3DListener->SetOrientation(forwardX, forwardY, -forwardZ, topX, topY, topZ,  
            DS3D_IMMEDIATE);  
    }  
  
    IDirectMusicPerformance8 *GetPerformance() { return dmusicPerformance; }  
};
```

## System cząstek

Ostatnią z omawianych klas szkieletu SimpEngine jest klasa `CParticleSystem` reprezentująca system cząstek. W praktyce będą tworzone jej klasy pochodne po to, aby można było uzyskać efekty specjalne takie jak wybuchy, dym, ogień czy opady atmosferyczne. Sposoby korzystania z systemów cząstek omówione zostały w rozdziale 15. i dlatego nie będą omówione tutaj szczegółowo zastosowania klasy `CParticleSystem`.

Należy jednak przypomnieć, że najpierw należy utworzyć klasę pochodną klasy `CParticleSystem`, która będzie reprezentować wybrany rodzaj efektu specjalnego. Następnie trzeba dostarczyć implementację jej metod `Update()`, `Render()` i `InitializeParticle()`, które opisywać będą zachowanie się cząstek tworzących efekt. Korzystając z systemu cząstek będzie można wywoływać jego metodę `Update()` z metody `OnAnimate()` odpowiedniego obiektu oraz metodę `Render()` z odpowiedniej metody `OnDraw()`.

## Podsumowanie

W rozdziale tym przedstawiony został przykład szkieletu SimpEngine, który wykorzystany zostanie w następnym rozdziale do stworzenia gry.

Klasa `CNode` jest klasą bazową klasy `CObject`, którą wykorzystuje się do tworzenia hierarchii obiektów wykorzystywanej przez szkielet SimpEngine.

Zasadniczą część szkieletu SimpEngine stanowi klasa COGLWindow, która tworzy okno grafiki OpenGL i obsługuje komunikaty wysyłane do tego okna przez system Windows. Dla każdego z obsługiwanych komunikatów klasa COGLWindow definiuje osobną, wirtualną metodę obsługi. Klasa COGLWindow przechowuje także obiekt klasy CInputSystem reprezentujący podsystem wejścia, który korzystając z interfejsu DirectInput pobiera informacje o stanie klawiatury, myszy i manipulatora. System wejścia posiada metody umożliwiające sprawdzenie tego, czy poszczególne klawisze zostały naciśnięte, o ile przemieściła się mysz, jak zmienił się stan manipulatora. Klasa CEngine stanowi klasę pochodną klasy COGLWindow i zawiera metody obsługujące główną pętlę przetwarzania komunikatów systemu Windows, cykl gry oraz informacje wejściowe uzyskane za pośrednictwem systemu wejścia.

Na typowy cykl gry składają się etapy związane z pobraniem informacji wejściowej, zmianą położenia gracza, zastosowaniem sztucznej inteligencji, wyznaczeniem równań ruchu obiektów na podstawie praw fizyki, odtworzeniem dźwięków i narysowaniem grafiki. Cykl gry zastosowany w szkielecie SimpEngine nie stanowi dokładnego odbicia wymienionych etapów, ponieważ musi wywoływać metody Prepare(), Animate() i Draw() dla wszystkich obiektów hierarchii klasy CObject.

Sposób użycia urządzeń wejścia przez gracza (naciśnięcie klawisza, przesunięcie myszy, naciśnięcie przycisku manipulatora i tak dalej) pozwala określić metoda CEngine::CheckInput(), która wykorzystuje w tym celu obiekt klasy CInputSystem zdefiniowany w klasie COGLWindow. Klasa CSimpEngine stanowi klasę pochodną klasy CEngine i reprezentuje część szkieletu, która zyskuje możliwość sterowania po uruchomieniu gry.

Klasa CCamera definiuje sposób widzenia świata gry przez gracza i tym samym sposób tworzenia jego graficznej reprezentacji. Klasa CWorld definiuje świat gry, którego obiekty obsługują szkielet SimpEngine. Metoda CEngine::GameCycle() wykorzystuje klasę CWorld, aby wywołać metody Prepare(), Animate() i Draw() dla wszystkich obiektów świata.

Dzięki temu, że klasa CMD2Model jest także klasą pochodną klasy CObject, możliwe staje się włączenie modeli w formacie MD2 do hierarchii obiektów gry. Klasa CEntity jest klasą pochodną klasy CMD2Model i hermetyzuje funkcjonalność pozwalającą korzystać z modeli MD2 w ten sam sposób co z innych obiektów gry.

Klasa CAudioSystem zarządza tworzeniem i odtwarzaniem efektów dźwiękowych i fragmentów muzycznych reprezentowanych przez obiekty klasy CAudio. Klasy pochodne klasy CParticleSystem umożliwiają tworzenie dowolnych efektów specjalnych wykorzystujących systemy cząstek.

## Rozdział 21.

# Piszemy grę: „Czas zabijania”

Uwieńczeniem przedstawionych w tej książce zagadnień będzie napisanie własnej gry. Jej ukończenie zająć może około tygodnia przy zastosowaniu szkieletu SimpEngine przedstawionego w rozdziale 20. Gra wykorzystuje efekty eksplozji uzyskiwane przez wykorzystanie systemów częstek, animowane modele w formacie *MD2*, wykrywanie zderzeń za pomocą sfer ograniczeń, sterowanie kamerą, efekty dźwiękowe, sztuczną inteligencję (w podstawowym zakresie) oraz tworzenie terenu i rysowanie jego reprezentacji graficznej. Gra jest bardzo prosta — głównie dlatego, że na jej opracowanie poświęcono tak niewiele czasu. Na zakończenie lektury książki wskazana jest więc jej rozbudowa lub nawet stworzenie zupełnie nowej, lepszej gry!

W rozdziale tym omówione zostaną następujące zagadnienia:

- ◆ wstępny projekt gry „Czas zabijania”;
- ◆ tworzenie świata gry, a zwłaszcza jej terenu;
- ◆ postacie gry i ich sztuczna inteligencja;
- ◆ symulacje rakiet i efekty wybuchów;
- ◆ graficzny interfejs użytkownika;
- ◆ korzystanie z gry;
- ◆ komplikacja i utworzenie programu wykonywalnego.

## Wstępny projekt

Trzeba przyznać, że jako autorzy na początku nie mieliśmy pojęcia, jak dokładnie będzie wyglądać gra omówiona w tym rozdziale. Chcieliśmy, aby gracz poruszał się po losowo generowanym terenie i mógł widowiskowo niszczyć wrogów. Żadnej treści. Żadnej fabuły. Żadnych nagród.

Z czasem wpadliśmy na pomysł, aby ustawić dla gracza limit czasu, w którym powinien wyeliminować wszystkich przeciwników. W ten sposób określony został cel postawiony przed graczem. Zdecydowaliśmy także, że przeciwnicy nie będą strzelać do

gracza. Łatwe zadanie? Nie całkiem, ponieważ wrogowie będą uciekać przed graczem, a nie dążyć do konfrontacji jak w przypadku większości gier. Jeśli gracz wyeliminuje wszystkich przeciwników w określonym czasie, to zostanie zwycięzcą, a w przeciwnym razie przegra. W ten sposób ustaliliśmy zasady gry „Czas zabijania” i rozpoczęliśmy pracę nad implementacją gry.

## Świat gry

Pierwszym elementem gry, który został opracowany był generowany losowo teren gry. Reprezentuje go klasa CTerrain będąca klasą pochodną klasy CObject. Teren zdefiniowany jest za pomocą szeregu wartości z przedziału od 0 do 1, które następnie mnożone są przez odpowiedni współczynnik w zależności od szerokości terenu. Aby uzyskać bardziej stromą rzeźbę terenu, można zwiększyć wartość tego współczynnika, a żeby wygładzić jego rzeźbę, należy zmniejszyć współczynnik. Poniżej przedstawiona została definicja klasy CTerrain:

```
class CTerrain : public CObject
{
private:
    int width;                                // kwadratowy teren o wymiarach width na width
                                                // najlepiej takich, że  $2^n = width$ 
    float terrainMul;                         // współczynnik skalowania rozmiarów terenu
    float heightMul;                           // współczynnik skalowania wysokości terenu
    float scanDepth;                          // głębokość widocznej części terenu
    float textureMul;                          // współczynnik skalowania tekstuury terenu

    float RangedRandom(float v1, float v2);
    void NormalizeTerrain(float field[], int size);
    void FilterHeightBand(float *band, int stride, int count, float filter);
    void FilterHeightField(float field[], int size, float filter);
    void MakeTerrainPlasma(float field[], int size, float rough);

protected:
    // teren nie porusza się, równania ruchu zbędne
    void OnAnimate(scalar_t deltaTime) {}

    void OnDraw(CCamera *camera);           // rysowanie terenu
    void OnCollision(CObject *collisionObject); // zderzenia terenu

public:
    float *heightMap;                      // dynamiczna mapa wysokości
    CTerrainTexture terrainTex[5];          // wiele tekstur terenu
    float fogColor[4];                      // kolor mgły lub nieba

    CTerrain();
    CTerrain(int width, float rFactor);
    ~CTerrain() { delete [] heightMap; }

    void Load() {}
    void Unload() {}

    void BuildTerrain(int width, float rFactor); // generuje teren
```

```
float GetWidth() { return width; }           // zwraca szerokość terenu
float GetMul() { return terrainMul; }         // zwraca współczynnik skalowania
float GetScanDepth() { return scanDepth; }     // zwraca głębie skanowania

    float GetHeight(double x, double z);        // wysokość terenu w punkcie (x, z)
};
```

Jak łatwo zauważyc, obiekt klasy CTerrain wymaga jedynie wykrywania zderzeń i prawidłowego tworzenia reprezentującej go grafiki. Metoda CTerrain::OnDraw() rysuje teren korzystając ze zmiennej składowej scanDepth, która pozwala jej ustalić, jaki obszar terenu jest widoczny (w dal). Parametrem tej metody jest obiekt klasy CCamera, który z kolei pozwala określić, jaki wycinek terenu jest widoczny dla gracza. A oto implementacja metody CTerrain::OnDraw():

```
void CTerrain::OnDraw(CCamera *camera)
{
    int z, x;                                // zmienne indeksu

    glEnable(GL_DEPTH_TEST);                  // włącza bufor głębi

    glFogi(GL_FOG_MODE, GL_LINEAR);
    glFogfv(GL_FOG_COLOR, fogColor);
    glFogf(GL_FOG_START, scanDepth * 0.2f);
    glFogf(GL_FOG_END, scanDepth * 2.5);
    glHint(GL_FOG_HINT, GL_FASTEST);
    glEnable(GL_FOG);                        // włącza mgłę

    // konfiguruje i włącza łączenie kolorów
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_BLEND);
    glEnable(GL_ALPHA_TEST);
    glAlphaFunc(GL_GREATER, 0.0);
    glDisable(GL_ALPHA_TEST);

    // włącza obsługę tekstur i wybiera podstawową teksturę
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, terrainTex[0].texID);
    glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    glColor3f(1.0, 1.0, 1.0);

    // rysuje łańcuchy trójkątów posuwając się wzduż osi z
    for (z = (int)(camera->position.z / terrainMul - scanDepth), z=z<0?0:z;
         (z < camera->position.z / terrainMul + scanDepth) && z < width-1;
         z++)
    {
        glBegin(GL_TRIANGLE_STRIP);
        for (x = (int)(camera->position.x / terrainMul - scanDepth), x=x<0?0:x;
             (x < camera->position.x / terrainMul + scanDepth) && x < width-1;
             x++)
        {
            glTexCoord2f(textureMul * x, textureMul * z);
            glVertex3f((float)x*terrainMul,
                       (float)heightMap[x + z*width]*heightMul,
                       (float)z*terrainMul);
        }
    }
}
```

```

glTexCoord2f(textureMul * (x+1), textureMul * z);
glVertex3f((float)(x+1)*terrainMul,
           (float)heightMap[x+1 + z*width]*heightMul,
           (float)z*terrainMul);

glTexCoord2f(textureMul * x, textureMul * (z+1));
glVertex3f((float)x*terrainMul,
           (float)heightMap[x + (z+1)*width]*heightMul,
           (float)(z+1)*terrainMul);

glTexCoord2f(textureMul * (x+1), textureMul * (z+1));
glVertex3f((float)(x+1)*terrainMul,
           (float)heightMap[x+1 + (z+1)*width]*heightMul,
           (float)(z+1)*terrainMul);
}

glEnd();
}
}
}

```

Należy zwrócić uwagę na to, że na wstępnie metoda CTerrain::OnDraw() konfiguruje mgłę o liniowej gęstości i umiarkowanej intensywności, a także włącza łączenie kolorów, dzięki czemu efekt znikania terenu we mgle jest doskonalszy. W ten sposób mgła ukrywa przed graczem tę część terenu, która nie jest rysowana, o czym decyduje wartość zmiennej scanDepth.

Metoda CTerrain::GetHeight() pozwala ustalić wysokość terenu w dowolnym jego punkcie, nawet pomiędzy wierzchołkami. W tym celu znajdowane są cztery wierzchołki najbliższe danego punktu ( $x, z$ ) i wysokość terenu wyznaczana jest na drodze interpolacji wysokości w tych czterech wierzchołkach. Metoda CTerrain::GetHeight() wykorzystywana jest przez metody wszystkich obiektów świata gry podczas wykrywania zderzeń z terenem.

## Przeciwnicy

„Czas zabijania” dysponuje dwoma rodzajami przeciwników — Ogro i Sod. Nazwy te pochodzą od wykorzystanych tu modeli w formacie *MD2*. Przeciwników można również łatwo wyeliminować, ale Ogro jest wolniejszy i mniej „inteligentny” od Soda.

Dla reprezentacji tych postaci utworzyć trzeba najpierw klasę CEnemy jako klasę pochodną klasy CEntity.

Oto jej definicja:

```

class CEnemy : public CEntity
{
private:
protected:
    float distFromPlayer;      // odległość przeciwnika od gracza
    float runSpeed;            // prędkość ucieczki
    AIState_t aiState;         // stan przeciwnika
}

```

```
// metodę tę zastępujemy implementując nowy rodzaj przeciwnika
virtual void OnProcessAI() {}

// obsługa zderzeń z innymi obiektami
void OnCollision(CObject *collisionObject);

public:
    CPlayer *player;

    CEnemy()
    {
        isDead = false;           // przeciwnik wchodzi do gry żywą
        velocity = CVector(0.0, 0.0, 0.0); // wektor ruchu przeciwnika
        runSpeed = velocity.z;   // prędkość ucieczki
        SetState(MODEL_IDLE);    // stan bezczynności
        direction = 0.0f;         // zwrotny na północ
        player = NULL;           // gracz jeszcze nie istnieje
    }

    ~CEnemy() {}

    void ProcessAI() { OnProcessAI(); }          // sztuczna inteligencja
    void SetPlayer(CPlayer *p) { player = p; } // referencja do obiektu gracza
};
```

Klasa CEnemy posiada dwie metody wirtualne, OnProcessAI() i OnCollision(), które należy zastąpić tworząc własne klasy przeciwników. Metoda OnProcessAI() określa sposób „myślenia” przeciwnika. Stan działania przeciwnika przechowuje zmienna aiState o dość chronionym.

## Sztuczna inteligencja przeciwnika

Sposób działania przeciwnika określony jest przez maszynę stanów. Stany tej maszyny zdefiniowane są za pomocą typu wyliczeniowego AIState\_t:

```
enum AIState_t
{
    AI_UNCARING,    // przeciwnik nie jest zaniepokojony
    AI_SCARED,      // przeciwnik jest przestraszony i ucieka
    AI_DEAD         // przeciwnik jest martwy
};
```

„Czas zabijania” wyróżnia tylko kilka podstawowych stanów, w których może znajdować się przeciwnik: jeśli jeszcze żyje, to albo nie jest zaniepokojony, albo został przestraszony i ucieka przed graczem. Gdy przeciwnik nie został jeszcze przestraszony, to pozostaje bez ruchu lub porusza się w dowolnym kierunku. Tabela 21.1 przedstawia związek pomiędzy stanami przeciwnika i stanami modelu MD2 zastosowany w tej grze.

Jak łatwo zauważyc, stanowi AI\_UNCARING reprezentującemu nieprzestraszonego przeciwnika odpowiadają dwa stany modelu: MODEL\_IDLE lub MODEL\_RUN. To, w którym z nich w danym momencie znajduje się przeciwnik, zależy wyłączenie od jego sztucznej inteligencji zaimplementowanej przez metodę wirtualną OnProcessAI().

**Tabela 21.1.** Stany przeciwnika

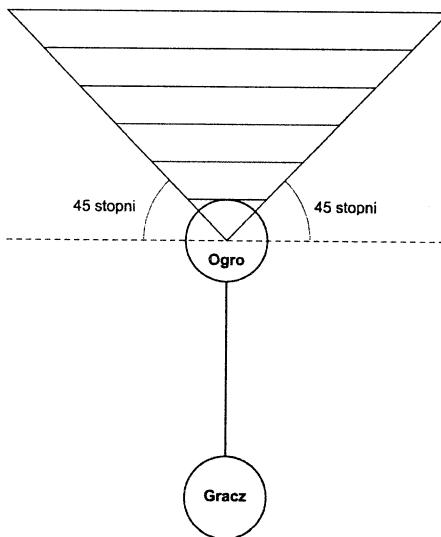
Stan przeciwnika	Stan modelu
AI_UNCARING	MODEL_IDLE
	MODEL_RUN
AI_SCARED	MODEL_RUN
AI_DEAD	MODEL_DIE

## Ogro

Stwórz ten porusza się powoli i ucieka przed graczem w mało inteligentny sposób. Gdy pozostawiony zostanie w spokoju (stan AI\_UNCARING), to prawdopodobieństwo, że będzie się poruszał wynosi tylko 25%. W pozostałych przypadkach pozostanie bezczynny (75%). Gdy gracz znajdzie się od niego w odległości mniejszej niż 100 jednostek, spłoszy się (stan AI\_SCARED) i będzie uciekał pod kątem 45° (w jedną lub drugą stronę). Ilustruje to rysunek 21.1.

**Rysunek 21.1.**

Ogro ucieka  
przed graczem  
pod kątem 45°



Jeśli podczas ucieczki Ogro zderzy się z innym przedstawicielem swojego gatunku lub z Sodem, to powróci do stanu AI\_UNCARING i przestanie się poruszać. Ogro będzie także zderzać się z terenem, po którym się porusza. W takiej sytuacji będzie trzeba pobrać wysokość terenu w punkcie, w którym znalazły się Ogro, i „podnieść” jego pozycję na tę wysokość. Należy także sprawdzać współrzędne x i z pozycji Ogro, aby nie „uciekł” poza granice terenu. Jeśli Ogro zostanie trafiony rakietą, przejdzie do stanu AI\_DEAD.

Poniżej przedstawiona została implementacja metody sztucznej inteligencji Ogra: COgroEnemy::OnProcessAI():

```
void COgroEnemy::OnProcessAI()
{
    // oblicza odległość gracza
    CVector diff = player->position - position;
```

```
if (aiState != AI_DEAD)
{
    // jeśli gracz znajduje się dostatecznie blisko, rozpoczyna ucieczkę
    distFromPlayer = sqrt(diff.x*diff.x + diff.y*diff.y + diff.z*diff.z);
    if (distFromPlayer < 100.0)
        aiState = AI_SCARED;
    else
        aiState = AI_UNCARING;
}
}
```

Metoda C0groEnemy::OnPrepare() zmienia stan modelu *MD2* na podstawie stanu, w którym znajduje się Ogró:

```
void C0groEnemy::OnPrepare()
{
    float dirToPlayer; // kąt pomiędzy wektorami ruchu Ogra i gracza

    CVector diff; // wektor od Ogra do gracza
    diff.x = position.x - player->position.x;
    diff.z = position.z - player->position.z;
    diff.Normalize();

    // znajduje kąt pomiędzy wektorami ruchu Ogra i gracza
    // w odniesieniu do ujemnej części osi z
    dirToPlayer = RAD2DEG(diff.Angle(CVector(0.0,-1)));

    // inicjuje generator liczb pseudolosowych
    srand((unsigned int)time(NULL));

    // wywołuje metodę sztucznej inteligencji
    ProcessAI();

    // zmienia ModelState na podstawie AIstate
    switch (aiState)
    {
        case AI_SCARED:
            // określa kierunek ucieczki Ogra
            direction = (dirToPlayer - 90) + ((rand()%90)-45);

            ModelState = MODEL_RUN;
            velocity = CVector(0.0, 0.0, 15.0);
            break;
        case AI_UNCARING:
            direction = float(rand() % 360); // zwrócony w dowolnym kierunku
            if ((rand() % 4) != 0) // bezczynny 75% czasu
            {
                ModelState = MODEL_IDLE;
                velocity = CVector(0.0, 0.0, 0.0);
            }
            else
            {
                velocity = CVector(0.0, 0.0, 15.0);
                ModelState = MODEL_RUN;
            }
            break;
        case AI_DEAD:
            ModelState = MODEL_DIE;
```

```

velocity = CVector(0.0, 0.0, 0.0);
if (nextFrame == stateStart)      // usuwa Ogra ze świata gry
{
    // usuwa Ogra
    isDead = true;
}
break;
default:
    break;
}

// wywołuje metodę dla stanów modelu MD2
CEntity::OnPrepare();
}

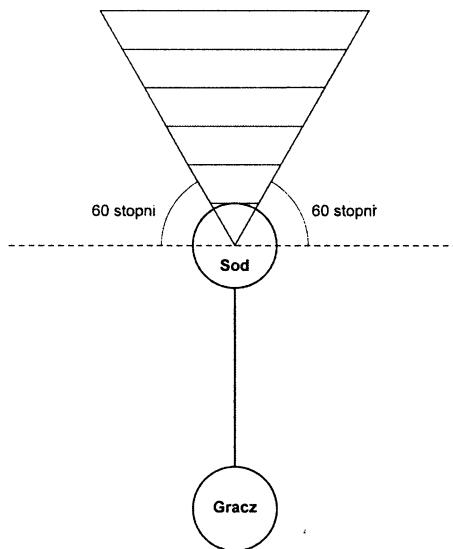
```

## Sod

Przeciwnik ten przypomina w zachowaniu Ogra, ale porusza się szybciej i nieco „inteligentniej” — zaczyna uciekać już wtedy, gdy gracz znajdzie się w odległości 125 jednostek w kierunkach pokazanych na rysunku 21.2. Kod implementujący jego zachowanie jest praktycznie identyczny z kodem Ogra, a jedyny wyjątek stanowi obliczenie kąta ucieczki. Nie będzie więc tutaj przedstawiony.

**Rysunek 21.2.**

*Sod ucieka  
przed graczem  
pod kątem 60°*



## Rakiety i eksplozje

Gracz może odpalać w kierunku przeciwników rakiety posługując się lewym przyciskiem myszy. Rakiety reprezentowane są przez obiekty klasy CRocket, która ładuje model rakiety w formacie *MD2*, rysuje go i wyznacza trajektorie jego ruchu. Rakiety poruszają się z prędkością 120 jednostek na sekundę. Klasa CRocket jest oczywiście klasą pochodną klasy CEntity. Poniżej przedstawiono jej definicję:

```
class CRocket : public CEntity
{
private:
    void SetupExplosionTexture(); // konfiguruje teksturę OpenGL

protected:
    void OnAnimate(scalar_t deltaTime); // wyznacza trajektorię rakiety
    void OnCollision(CObject *collisionObject); // wykrywa jej zderzenie
    void OnDraw(CCamera *camera); // rysuje rakietę
    void OnPrepare();

public:
    float distanceTravel; // odległość, którą przebyła rakieta
    CVector forward; // kierunek ruchu rakiety
    bool isExplosion; // wartość true, gdy rakieta wybuchła

    CTexure *explosionTex; // tekstura wybuchu
    CExplosion *explosion; // wybuch symulowany przez system cząstek

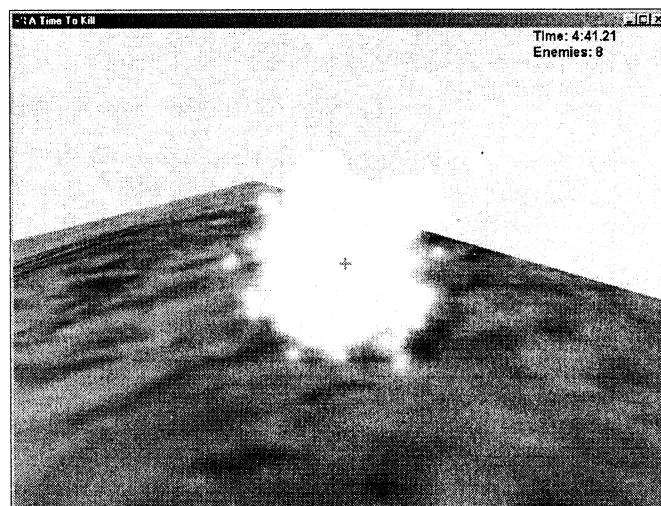
    CRocket();
    ~CRocket();

    void Load();
    void Unload();
}
```

Uderzając w obiekt rakieta eksploduje. Efekt wybuchu tworzony jest przez obiekt klasy CExplosion, która jest klasą pochodną klasy CParticleSystem. Zdarzenie to powoduje zmianę wartości składowej isExplosion klasy CRocket. Klasa CExplosion rysując wybuch wykorzystuje teksturę explosionTex określoną przez klasę CRocket. Rysunek 21.3 przedstawia efekt wybuchu rakiety.

### Rysunek 21.3.

*Wybuch rakiety.  
Uwagi zwracają  
czworokąty pokryte  
teksturą, do których  
zastosowano  
łączenie kolorów*



# Interfejs użytkownika gry

Klasa CGUI reprezentuje graficzny interfejs użytkownika gry „Czas zabijania”. Przedstawia on graczy upływający czas i liczbę przeciwników, którzy pozostali do wyeliminowania. Wyświetla napisy: „Wygrałeś!” i „Przegrałeś！”, a także celownik na środku okna programu. A oto definicja klasy CGUI:

```
class CGUI
{
private:
    // czas pozostały graczy w bieżącej rozgrywce
    int minutesLeft, secondsLeft, millisecondsLeft;
    int enemiesLeft; // liczba przeciwników do wyeliminowania

    CFont *font; // czcionka wykorzystywana do napisów
    CFont *crosshair; // czcionka celownika
    CFont *endText; // czcionka tekstu informującego o wygranej

public:
    CGUI();
    ~CGUI();

    void SetCurrentTime(float timeLeft); // zmienia czas pozostały graczy
    void SetEnemiesLeft(int eLeft); // zmienia liczbę przeciwników do wyeliminowania
    void Draw(); // rysuje interfejs użytkownika
    void Animate(float deltaTime); // nieużywana

    void DrawWinner();
    void DrawLoser();
}:
```

Bardzo prosta, prawda? Warto jeszcze sprawdzić, w jaki sposób klasa CGUI jest wykorzystywana w programie zapoznając się z implementacją klasy CWorld umieszczoną w pliku *world.cpp*.

# Korzystanie z gry

Po załadowaniu programu zegar rozpoczyna odmierzanie limitu czasu, a gracz musi rozpocząć poszukiwanie przeciwników. Liczba przeciwników pozostających do wyeliminowania prezentowana jest w prawym górnym rogu ekranu. Za każdym razem, gdy uda się graczowi ustrzelić Ogra lub Soda, liczba ta będzie zmniejszać się. Gracz zostanie zwycięzcą, jeśli zdoła zniszczyć wszystkich przeciwników przed upływem zadanego czasu.

Sposób sterowanie grą przedstawia tabela 21.2.

**Tabela 21.2.** Sterowanie gry

Klawisz lub przycisk	Funkcja
<i>W</i>	Ruch do przodu
<i>S</i>	Ruch do tyłu
<i>A</i>	Ruch w lewo
<i>D</i>	Ruch w prawo
Lewy przycisk myszy	Odpalenie rakiety
Ruch myszy	Spojrzenie w wybranym kierunku
Klawisz + (w bloku numerycznym)	Zwiększa czułość myszy
Klawisz – (w bloku numerycznym)	Zmniejsza czułość myszy
<i>Esc</i>	Koniec

## Kompilacja gry

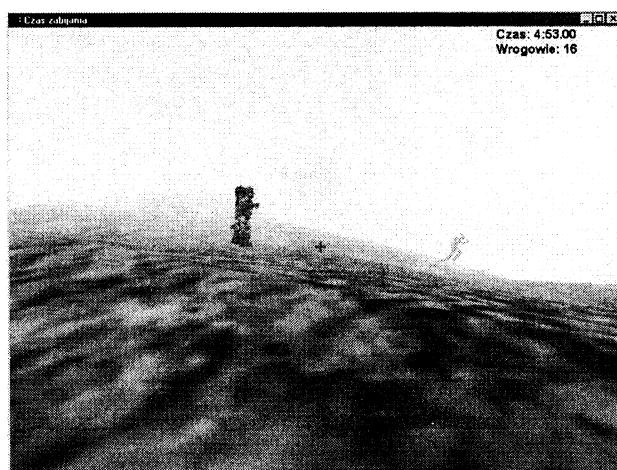
Aby skompilować kod źródłowy gry i utworzyć plik wykonywalny, niezbędne są następujące biblioteki:

- ◆ *opengl32.lib* — podstawowa biblioteka OpenGL;
- ◆ *glu32.lib* — biblioteka GLU;
- ◆ *dxguid.lib* — biblioteka DirectX GUID;
- ◆ *winmm.lib* — biblioteka multimedialnych systemu Windows;
- ◆ *dinput8.lib* — biblioteka DirectInput.

Używając kompilatora Microsoft Visual C++ należy rozwinąć menu *Project*, wybrać pozycję *Setting*, następnie zakładkę *Link* i wpisać podane wyżej nazwy plików bibliotek w wierszu *Object/Library Modules*.

Rysunek 21.4 przedstawia scenę z gry „Czas zabijania”.

**Rysunek 21.4.**  
„Czas zabijania”  
— Ogro po lewej  
i Sod po prawej



## Podsumowanie

W rozdziale tym przedstawione zostały tylko podstawowe informacje dotyczące gry „Czas zabijania”, które pozwolą rozpoczęć pracę z jej pełnym kodem. Kod ten umieszczony został na dysku CD dołączonym do książki.

Omówiony został sposób tworzenia świata gry, jej postaci i ich sztucznej inteligencji, rakiet i wybuchów, prosty interfejs użytkownika, sposób korzystania z gry i komplikacji jej kodu.

Z pewnością zaprezentowana gra może zostać istotnie ulepszona i rozbudowana. Przykładowe modyfikacje mogą polegać na dodaniu warkoczy dymu do lecących rakiet, wydawaniu dźwięków przez przeciwników, wprowadzeniu radaru do wykrywania Ogrów i Sodów, szybszej grafiki i nowych obiektów świata gry. Możliwości są nieograniczone i z pewnością czytelnicy będą potrafili zaskoczyć autorów niejednym pomysłem.

Dysk CD dołączony do książki zawiera o wiele więcej materiału niż tylko kody źródłowe programów omawianych w książce. Specjalne podziękowania należą się Jeffowi „NeHe” Molofee za wybranie i umieszczenie na dysku CD ciekawych programów demonstracyjnych wykorzystujących OpenGL. Na podziękowania takie zasłużył także Bas Kuenen, przede wszystkim za pomysły i pomoc w zaprojektowaniu szkieletu SimpEngine.

# **Dodatki**



## Dodatek A

# Zasoby sieci Internet

Istnieje wiele ciekawych zagadnień związanych z programowaniem gier, które chcieliśmy przedstawić w tej książce, lecz uniemożliwiły nam to ramy pojedynczego tomu. Na szczęście w sieci Internet dostępnych jest wiele informacji nadających się do wykorzystania podczas tworzenia gier. Poniżej przedstawiamy wybór tych, które mogą okazać się najbardziej przydatne.

W sieci Internet stworzyliśmy też specjalną stronę poświęconą niniejszej książce. Znajduje się ona pod adresem <http://glbook.gamedev.net> i zawiera erratę, odpowiedzi na najczęściej zadawane pytania i inne dodatkowe informacje.

## Programowanie gier

W sieci Internet istnieją setki stron poświęconych temu zagadnieniu. Najlepsze z nich podajemy poniżej. Strony te dotyczą programowania gier w ogóle, a nie programowania z wykorzystaniem OpenGL.

### GameDev.net

<http://www.gamedev.net>

Wiodąca strona dla programistów gier o różnym zaawansowaniu. Zawiera ponad 1000 artykułów dotyczących wszystkich aspektów programowania gier, a w tym zastosowań OpenGL, DirectX, sieci komputerowych, dźwięku, grafiki, sztucznej inteligencji i wielu innych. Oprócz tego znaleźć tam można nowości z branży, przykłady kodów źródłowych, słownik terminologii związanej z programowaniem gier oraz największe i najaktywniejsze forum gromadzące programistów gier z całego świata. Całość okraszona jest ciekawą oprawą graficzną.

## **Wyszukiwarka informacji związanych z programowaniem gier**

*<http://www.gdse.com>*

Serwis GDSE (*Game Development Search Engine*) umożliwia przeszukiwanie setek stron poświęconych programowaniu gier za pomocą specjalizowanej wyszukiwarki. Dzięki niemu można łatwo dotrzeć do interesującej odwiedzającego stronę informacji rozproszonej na dziesiątkach stron w Internecie.

## **flipCode**

*<http://www.flipcode.com>*

Strona ta publikuje nowości oraz artykuły z zakresu programowania gier komputerowych. Szczególnie interesujące są artykuły uznanych programistów, w których dzielą się z odbiorcami swoimi osiągnięciami i wiedzą.

## **Gamasutra**

*<http://www.gamasutra.com>*

Strona ta jest własnością firmy Gama Network, która wydaje także Game Developer Magazine oraz organizuje konferencję Game Developers Conference. Na stronie tej znajdują się materiały publikowane wcześniej we wspomnianym czasopiśmie, prezentowane na konferencjach, a także oryginalne, dotąd niepublikowane teksty.

## **OpenGL**

Programiści OpenGL tworzą w Internecie aktywną społeczność. Już pierwszy próba wyszukania jakiejkolwiek informacji na ten temat przyniesie w efekcie dziesiątki — jeśli nie setki — stron poświęconych programowaniu w OpenGL. Poniżej przedstawiamy najbardziej interesujące.

## **NeHe Productions**

*<http://nehe.gamedev.net>*

NeHe Productions autorstwa Jeffa Molofee jest jedną z czołowych stron poświęconych technologii OpenGL. Obok ponad 40 oryginalnych kursów OpenGL zawiera także imponujący zestaw programów demonstrujących możliwości OpenGL. Autor ciągle wzmacnia także kolekcję referencji do innych stron poświęconych OpenGL.

## OpenGL.org

<http://www.opengl.org>

*OpenGL.org* jest oficjalną stroną rady ARB. Regularnie publikuje ona wiadomości dotyczące rozwoju specyfikacji oraz zawiera szereg przydatnych informacji.

## Inne strony poświęcone OpenGL

<http://reality.sgi.com/mjk/tips/>

zaawansowane techniki OpenGL

<http://glvelocity.gamedevnet.net>

glVelocity

<http://romka.demonews.com>

Romka Graphics

<http://nate.scuzzy.net/>

strona Nate'a Millera

<http://www.gamedev.net/opengl>

OpenGL Journal

<http://reality.sgi.cim/blythe/sig99/>

notatki z kursu Siggraph '99

## DirectX

W sieci Internet istnieje także wiele stron poświęconych DirectX. Kilka, naszym zdaniem najbardziej przydatnych, wymieniamy niżej.

## DirectX Developer Center

<http://msdn.microsoft.com/directx>

Tutaj możemy załadować najnowszą wersję pakietu DirectX SDK, poznać nowości i zapoznać się z ważniejszymi wydarzeniami związanymi z DirectX, a także przeczytać artykuły, których autorami są członkowie zespołu pracującego nad DirectX.

## Lista mailingowa DirectX

Archiwum i uczestnictwo: <http://discuss.microsoft.com/archives/directxdev.html>

Najczęściej zadawane pytania: <http://msdn.microsoft.com/library/techart/dxfaq2.htm>

Lista prowadzona jest przez firmę Microsoft i stanowi jedno z najlepszych źródeł informacji o DirectX. Zabierają tu głos członkowie zespołu DirectX, programiści wiodących producentów sprzętu oraz uznani programiści gier. Przed zarejestrowaniem się — a zwłaszcza przez zabranie głosu w dyskusji — zalecamy zapoznanie się z archiwami listy i najczęściej zadawanymi pytaniami.

## Inne zasoby

W Internecie znajdują się także omówienia zagadnień zbyt wyspecjalizowanych lub zaawansowanych, by mogły znaleźć się w książkach. Poniżej prezentujemy kilka stron, które uzupełniają materiał przedstawiony w naszej książce.

### **ParticleSystems.com**

*<http://particlesystems.com>*

Strona poświęcona systemom cząstek. Zawiera doskonały interfejs programowy Particle Systems API.

### **Real-Time Rendering**

*<http://www.realtime rendering.com>*

Oficjalna strona doskonałej książki „Real-Time Rendering”. Zawiera także inne artykuły jej autorów oraz zbiór odnośników do interesujących informacji, które można znaleźć w Sieci. Skupia ona dużo wartościowych informacji, których eksploracji warto poświęcić więcej czasu.

### **Informacja techniczna dla programistów**

NVIDIA: *<http://www.nvidia.com/developer.nsf>*

Intel: *<http://cedar.intel.com/cgi-bin/ids.dll/main.jsp>*

ATI: *[http://www.ati.com/na/pages/resource\\_centre/dev\\_rel/devrel.htm](http://www.ati.com/na/pages/resource_centre/dev_rel/devrel.htm)*

Producenci sprzętu, tacy jak NVIDIA, Intel czy ATI, posiadają strony zawierające zaawansowane informacje związane z tworzeniem oprogramowania graficznego wykorzystującego ich produkty.

### **Artykuły dotyczące efektu mgły**

*<http://www.gamedev.net/opengl/volfog.html>*

*<http://www.gamedev.net/reference/articles/article677.asp>*

*<http://www.gamedev.net/reference/articles/article672.asp>*

W rozdziale 15. obiecaliśmy, że wspomnimy o artykułach poświęconych efektowi mgły objętościowej, więc dotrzymujemy danego słowa.

## Dodatek B

# Dysk CD

Dysk CD dołączony do tej książki zawiera kody źródłowe i pliki wykonywalne programów omówionych w książce, a także wybór programów narzędziowych, demonstracyjnych i gier OpenGL.

## Struktura plików na dysku CD

Na dysku CD znajduje się pięć głównych katalogów zawierających:

- ◆ **kod źródłowy** — teksty źródłowe wszystkich przykładów programów omawianych w kolejnych rozdziałach tej książki;
- ◆ **gry** — dodatkowe przykłady gier opracowane i dostarczone przez utalentowanych autorów;
- ◆ **programy demonstracyjne OpenGL** — przykłady zastosowań OpenGL;
- ◆ **programy i biblioteki** — graficzne, multimedialne i narzędziowe;
- ◆ **pakiety SDK** — pakiety niezbędne do tworzenia aplikacji OpenGL i DirectX.

## Wymagania sprzętowe

Poniżej przedstawiamy minimalne oraz zalecane parametry systemu używanego do przeglądania dysku CD:

- ◆ napęd CD-ROM, DVD, CD-R lub CD-RW;
- ◆ przeglądarka stron internetowych: Internet Explorer 4.0 lub nowsza wersja, Netscape Navigator w wersji 4.0 lub nowszej;
- ◆ pamięć: minimum 32 MB, zalecane 64 MB lub 128 MB;
- ◆ karta grafiki: szybka karta ze sprzętową obsługą OpenGL, na przykład NVIDIA GeForce czy Voodoo3 lub lepsza;
- ◆ procesor: przynajmniej Pentium pracujący z częstotliwością 450 MHz (im szybszy, tym lepiej).

## Instalacja

Po włożeniu dysku CD powinno się ukazać automatycznie menu dysku, jeśli korzystamy z systemu Windows 9x/Me/XP i jest włączona opcja automatycznego uruchamiania dysku. W przeciwnym razie należy ręcznie wybrać w folderze *Mój Komputer* napęd, w którym znajduje się dysk CD (najczęściej *D:*) i kliknąć dwukrotnie plik *start.exe*.

## Typowe problemy i ich rozwiązywanie

Niezależnie od tego, ile wysiłku zostanie włożone w przetestowanie oprogramowania, i tak zawsze trzeba liczyć się z tym, że pojawią się jakieś problemy.

Przed umieszczeniem na dysku wszystkie programy zostały sprawdzone przez aktualną wersję skanera antywirusowego. Dysk CD przeszedł pomyślnie tę weryfikację i przyjęliśmy, że jest wolny od tego rodzaju zagrożeń.

Wszystkie pliki umieszczone na dysku CD posiadają atrybut „tylko-do-odczytu”. Po ich przeniesieniu na dysk twardy należy pamiętać o usunięciu tego atrybutu, ponieważ w przeciwnym razie komplikacja programów nie będzie możliwa.

Większość przykładów umieszczonych na dysku CD zakłada znajomość języka C++ oraz umiejętność radzenia sobie ze zwykłymi problemami, które napotyka każdy programista. Typowym problemem gnębiącym początkujących programistów jest ustawiczne zapominanie o konieczności dołączania odpowiednich bibliotek w celu uzyskania wykonywalnego programu. Gwarantujemy, że wszystkie programy zamieszczone na dysku komplikują się poprawnie i dają w efekcie poprawne programy wykonywalne, gdy dołączone zostaną odpowiednie biblioteki. Jeśli podczas komplikacji pojawią się jakieś problemy, może to wiązać się jedynie z nieodpowiednią wersją kompilatora.

Mimo że programy demonstracyjne zostały także przetestowane, to nie można wykluczyć, że niektóre z nich w szczególnych warunkach mogą doprowadzić nawet do zawieszenia się systemu operacyjnego. W takim przypadku należy uruchomić system od nowa i nie uaktywniać więcej programu, który doprowadził do jego zawieszenia. Zawieszenie się systemu operacyjnego nie powoduje jego uszkodzenia.

Najczęstszym problemem, który można napotkać przy uruchamianiu wersji demonstracyjnych programów graficznych, jest pojawienie się komunikatu: „Failed to create rendering context”. Oznacza on niemożność utworzenia właściwego kontekstu tworzenia grafiki. W takim przypadku pomaga zwykle zmiana konfiguracji pulpitu prowadząca do tego, by używał on 16-bitowego kodowania kolorów.

Jeśli działanie niektórych programów demonstracyjnych wydaje się zbyt wolne, to warto sprawdzić, czy karta graficzna rzeczywiście posiada sprzętową obsługę OpenGL i czy nie jest dostępna nowsza wersja jej sterownika.

Milę eksploracji dysku CD!

# **Skorowidz**

## **A**

AdjustWindowRectEx(), 69  
akumulacja obrazów, 324  
alfa, 173  
algorytm  
    cieniowania, 206  
    metody bryły cieni, 383  
    RLE, 190  
animacje, 471  
modele MD2, 471  
powiewająca flaga, 213  
sterowanie modelem, 498  
ANSI\_CHARSET, 288  
antialiasing, 38, 101  
    odcinków, 103  
    wielokątów, 106  
aplikacje, 40  
multimedialne, 36  
OpenGL, 60, 67  
pełnoekranowe OpenGL, 67  
sieciowe, 30, 37  
    Windows, 41, 52, 570  
ARB, 30, 38, 237  
Architectural Review Board, 237  
architektura  
    DirectX, 35  
    gry, 27, 28  
    OpenGL, 31  
    otwarta, 29  
        szkieletu SimpEngine, 561  
ASCII, 172, 290  
automatyczne tworzenie mipmap, 213  
AVI, 36

## **B**

BeginPaint(), 44  
bezpośredni dostęp do danych, 408  
BGRA, 194  
BI\_RGB, 190  
biblioteki  
    dinput8.lib, 398  
    DirectInput, 595  
    DirectX, 21  
    DLL, 398  
    dxguid.lib, 398  
    GLAUX, 163  
    GLU, 30, 31, 327  
    GLUT, 33  
    graficzne, 19  
    multimedialnych, 595  
bieżąca macierz przekształceń, 114  
BITMAPFILEHEADER, 189, 191  
BITMAPINFOHEADER, 189, 190, 191, 201  
blokowanie tablic wierzchołków, 280  
błędy  
    GL\_STACK\_OVERFLOW, 124  
    GL\_STACK\_UNDERFLOW, 124  
BMP, 181, 188, 189, 541  
    wczytywanie plików, 190  
    zapis obrazu, 191  
bryła  
    cienia, 383  
    widoku, 86, 118  
bufor, 301, 326  
    akumulacji, 38, 301, 305, 324, 326  
    DirectSound, 450  
    efektu przestrzennego, 451  
    GDI, 303  
    geometrii, 38

bufor  
 głębi, 58, 177, 180, 187, 261, 301, 305, 306, 307,  
 326, 376, 381  
 IDirectSound3DBuffer, 456  
 koloru, 143, 179, 187, 301, 305  
 konfiguracja formatu pikseli, 301  
 obrazu, 301  
 OpenGL, 303  
 opróżnianie, 304  
 powielania, 187, 301, 305, 316, 318, 319, 326,  
 377, 381  
 przelaczanie, 66  
 stereoskopowy, 303  
 Z niezależny od sprzętu, 38  
 zerowanie, 66  
 buforowanie, 59  
 danych, 409  
 podwójne, 65, 305  
 stereoskopowe, 306, 326

**C**

c\_dfDIJoystick2, 406  
 c\_dfDIKeyboard, 406  
 c\_dfDIMouse, 406  
 CalcBoundingBox(), 534  
 CALLBACK, 42, 129, 403  
 CAudio, 581, 582  
 CAudioSystem, 580, 581, 582  
 CCamera, 576, 577  
 Animate(), 578  
 CD\_HREDRAW, 45  
 CEnemy, 588  
 OnCollision(), 589  
 OnProcessAI(), 589  
 CEngine, 573  
 EnterMessageLoop(), 573  
 GameCycle(), 574, 580  
 CEntity, 581, 588  
 CGUI, 594  
 ChangeDisplaySettings(), 68  
 CHiResTimer, 518, 520, 553, 573  
 ChoosePixelFormat(), 59, 64, 302, 304  
 ciągłość, 339  
 krzywizny, 339  
 pozycyjna, 339  
 styczna, 339  
 cienie, 379, 384  
 bryła, 383  
 bufor głębi, 381  
 bufor powielania, 381  
 kuli, 379  
 macierz rzutowania, 380  
 ograniczanie obszaru, 381

rzutowanie, 380, 382  
 statyczne, 379  
 wiele zacienianych powierzchni, 382  
 wiele źródeł światła, 382  
 cieniowanie, 145, 180, 206  
 gladkie, 146  
 Gouraud, 146  
 płaskie, 146  
 CInputSystem, 411, 572, 575  
 AcquireAll(), 413  
 ButtonDown(), 576  
 Initialize(), 412  
 Shutdown(), 412  
 UnacquireAll(), 413  
 Update(), 413, 576  
 CKeyboard, 414  
 Acquire(), 415  
 Constructor, 414  
 Unacquire(), 415  
 Update(), 415  
 ClearColor(), 304  
 ClearFont(), 288, 291  
 CloseDown(), 435  
 CMD2Model, 488, 490, 498, 580, 584  
 Animate(), 490, 495  
 GetState(), 499  
 Load(), 490, 491  
 LoadModel(), 490, 493  
 LoadSkin(), 490, 494  
 RenderFrame(), 497  
 SetState(), 499  
 SetTexture(), 490, 495  
 SetupMD2Texture(), 490  
 SetupSkin(), 490  
 Unload(), 497  
 CMouse, 416  
 Acquire(), 418  
 Unacquire(), 418  
 Update(), 417  
 CNode, 562, 563  
 Attach(), 564  
 AttachTo(), 564  
 CountNodes(), 565  
 Detach(), 565  
 IsFirstChild(), 564  
 IsLastChild(), 564  
 COObject, 530, 531, 540, 544, 566, 569, 574, 584  
 Animate(), 567, 568, 569  
 Draw(), 567, 568  
 OnAnimate(), 566, 569, 570  
 OnCollision(), 567, 569  
 OnDraw(), 567, 569  
 OnPrepare(), 567, 569, 570  
 Prepare(), 568, 569  
 ProcessCollisions(), 568, 569

- CoCreateInstance(), 429, 430  
COGLWindow, 562, 570, 575, 584  
COgroEnemy  
    OnPrepare(), 591  
    OnProcessAI(), 590  
CoInitialize(), 429  
COM, 38, 397, 468  
Common Object Model, 397  
CParticleSystem, 366, 583, 584, 593  
    Emit(), 367  
    InitializeSystem(), 367  
    KillSystem(), 368  
CPlane, 527  
    DistanceToPlane(), 529, 535, 536  
    PointOnPlane(), 529, 536  
    RayIntersection(), 529  
CPlayer, 550  
    Animate(), 551  
    Move(), 551  
CPuck, 544  
    Animate(), 545, 547, 549  
    Draw(), 545  
CreateAudioPath(), 446  
CreateBitmapFont(), 287, 288, 290  
CreateDevice(), 400  
CreateFont(), 285, 287  
CreateOutlineFont(), 290, 291  
CreateStandardAudioPath(), 445, 446  
CreateWindow(), 48  
CreateWindowEx(), 48, 49  
CRocket, 592  
CS\_CLASSDC, 45  
CS\_DBCLKS, 45  
CS\_GLOBALCLASS, 45  
CS\_NOCLOSE, 45  
CS\_OWNDC, 45  
CS\_PARENTDC, 45  
CS\_SAVEBITS, 45  
CS\_VREDRAW, 45  
CSimpEngine, 576  
CSnowstorm, 369  
    InitializeParticle(), 371  
    InitializeSystem(), 372  
    KillSystem(), 373  
    Konstruktor, 370  
    Render(), 372  
    Update, 371  
CTable, 539, 540  
    Draw(), 542  
    Load(), 540, 541, 542  
    SetupTexture(), 540, 541, 542  
CTerrain, 586  
    GetHeight(), 588  
    OnDraw(), 587  
CTexture, 540  
    LoadTexture(), 540  
CVector, 521  
    CrossProduct(), 524  
    DotProduct(), 524  
    Length(), 525  
    Normalize(), 525  
    Reflection(), 538  
    UnitVector(), 525  
CWorld, 576, 580, 594  
cykl gry, 574, 584  
czas, 505, 517  
częstki, 359  
czcionki  
    Arial, 288  
konturowe, 289  
pokryte teksturą, 292  
przekształcenia w przestrzeni trójwymiarowej, 292  
rastrowe, 285  
systemu Windows, 285  
trójwymiarowe, 289  
czworokąty, 108

**D**

- D3DX, 34  
DEVMODE, 67, 68  
DI\_OK, 399  
DI8DEVCLASS\_ALL, 403  
DI8DEVCLASS\_DEVICE, 403  
DI8DEVCLASS\_GAMECTRL, 403  
DI8DEVCLASS\_KEYBOARD, 403  
DI8DEVCLASS\_POINTER, 403  
DI8DEVTYPE\_1STPERSON, 401  
DI8DEVTYPE\_DEVICE, 401  
DI8DEVTYPE\_DRIVING, 401  
DI8DEVTYPE\_FLIGHT, 401  
DI8DEVTYPE\_GAMEPAD, 401  
DI8DEVTYPE\_JOYSTICK, 401  
DI8DEVTYPE\_KEYBOARD, 402  
DI8DEVTYPE\_MOUSE, 402, 403  
DI8DEVTYPE\_SCREENPOINTER, 402, 403  
DI8DEVTYPE\_SUPPLEMENTAL, 402  
DIDATAFORMAT, 406  
DIDC\_ALIAS, 404  
DIDC\_ATTACHED, 404  
DIDC\_FORCEFEEDBACK, 404  
DIDC\_POLLEDDATAFORMAT, 404  
DIDC\_POLLEDDEVICE, 404  
DIDEVCAPS, 404  
DIDEVICEINSTANCE, 403  
DIDEVICEOBJECTDATA, 409  
DIDEVICEOBJECTINSTANCE, 405  
DIDFT\_ABSAXIS, 405

- DIDFT\_ALL, 405  
DIDFT\_AXIS, 405  
DIDFT\_BUTTON, 405  
DIDFT\_NODATA, 405  
DIDFT\_OUTPUT, 405  
DIDFT\_POV, 405  
DIDFT\_PSHBUTTON, 405  
DIDFT\_RELAXIS, 405  
DIDFT\_TGLBUTTON, 405  
DIEDFL\_ALLDEVICES, 403  
DIEDFL\_ATTACHEDONLY, 403  
DIEDFL\_FORCEFEEDBACK, 403  
DIENUM\_CONTINUE, 403  
DIENUM\_STOP, 403  
DIEnumDeviceObjectsCallback(), 405  
DIERR\_INVALIDPARAM, 408  
DIERR\_NOTINITIALIZED, 408  
DIERR\_OTHERAPPHASPRIO, 408  
dinput.h, 398  
dinput8.lib, 398, 595  
DIPROP\_BUFFERSIZE, 407  
DIPROPHEADER, 407  
Direct Audio, 36  
Direct3D, 34, 36, 398  
DirectDraw, 34, 36  
DirectInput, 34, 36, 37, 391, 397, 572  
    bezpośredni dostęp do danych, 408  
    buforowanie danych, 409  
    DI\_OK, 399  
    DirectInputDevice, 398  
        dodawanie urządzeń, 399  
        format danych urządzenia, 405  
        inicjacja interfejsu, 397  
        klawiatura, 400  
        kończenie pracy z urządzeniem, 409  
        modyfikacja właściwości urządzenia, 407  
        mysz, 400  
        obiekty DirectInputDevice, 399  
        odpytywanie urządzeń, 409  
        odwzorowania akcji, 410  
        pobieranie danych wejściowych, 408  
        poziom współpracy, 407  
        sprawdzanie możliwości urządzenia, 404  
        tworzenie obiektów DirectInputDevice, 398  
    tworzenie urządzeń, 400  
    urządzenia, 401  
    urządzenia wejściowe, 391  
    wartości zwracane przez funkcje, 398  
    wyliczenia obiektów, 405  
    wyliczenia urządzeń, 400  
    zajmowanie urządzenia, 408  
    znaczniki możliwości urządzenia, 404  
    znaczniki poziomu współpracy, 407  
    znaczniki typu obiektów, 405  
    znaczniki wyliczenia urządzeń, 403  
    DIRECTINPUT\_VERSION, 398  
    DirectInput8Create(), 398, 399  
    DirectInputDevice, 398  
    DirectMusic, 34, 425, 428, 468  
    DirectPlay, 34, 36  
    DirectSetup, 37  
    DirectShow, 34, 36, 468  
    DirectSound, 34, 425, 445  
        brzmienia instrumentów, 430  
        bufor efektu przestrzennego, 451  
        bufor mikowania, 428  
        bufor ujścia, 428  
        bufor zasadniczy, 428  
        inicjacja COM, 429, 430  
        inicjacja obiektu wykonania, 429, 430  
        kontrola odtwarzania, 434  
        kończenie odtwarzania, 435  
        liczba odtworzeń segmentu, 434  
        ładowanie dźwięku, 429  
        ładowanie instrumentów, 432  
        ładowanie segmentu, 430, 431  
        odległość maksymalna, 453  
        odległość minimalna, 453  
        odtwarzanie dźwięku, 429  
        odtwarzanie segmentu, 430, 432  
        parametry przestrzennego źródła, 451  
        tworzenie obiektu ładującego, 430, 431  
        tworzenie obiektu wykonania, 429, 430  
        zatrzymanie odtwarzania, 433  
    DirectX, 21, 25, 28, 30, 33, 37, 398, 601  
        architektura, 35  
        DirectInput, 36, 391  
        DirectPlay, 36  
        DirectSetup, 37  
        DirectShow, 36  
        DirectX Audio, 36  
        DirectX Graphics, 36  
        HAL, 35  
        HEL, 35  
        historia, 34  
        lista mailingowa, 601  
    DirectX 8, 397, 451  
    DirectX 8.0 SDK, 29  
    DirectX Audio, 36, 37, 421, 425, 426, 445, 446, 450, 468  
        bufory, 428  
        DirectMusic, 425  
        DirectSound, 425  
        efekt przestrzenny, 456  
        graf narzędzi segmentów, 428  
        graf narzędzi ścieżki dźwięku, 428  
        graf narzędzi wykonania, 428  
        instrumenty, 427  
        kanaly wykonania, 427  
        komunikaty, 427

- ładowanie dźwięków, 427  
obiekty lądujące, 426  
przepływ danych dźwięku, 428  
segmenty, 426  
stany segmentów, 426  
syntezator DSL, 427  
ścieżki dźwięku, 428  
wykonanie, 427  
DirectX Developer Center, 601  
DirectX Graphics, 36  
DirectX GUID, 595  
DirectX Media Objects, 36  
DirectX SDK, 406  
DISCL\_BACKGROUND, 407  
DISCL\_EXCLUSIVE, 407  
DISCL\_FOREGROUND, 407  
DISCL\_NONEXCLUSIVE, 407  
DISCL\_NOWINKEY, 407  
DispatchMessage(), 50, 51  
DisplayMD2(), 480  
DisplayMD2Interpolate(), 485, 487, 488  
DisplayScene(), 116  
DLL, 398  
DLS, 36  
DLS Level 2, 427  
długość wektora, 72  
DMO, 36  
DMUS\_APATH\_DYNAMIC\_3D, 447  
DMUS\_APATH\_DYNAMIC\_MONO, 447  
DMUS\_APATH\_DYNAMIC\_STEREO, 447  
DMUS\_APATH\_SHARED\_  
STEREOPLUSREVERB, 447  
DMUS\_SEG\_REPEAT\_INFINITE, 434  
DMUS\_SEGF\_AFTERLATENCYTIME, 448  
DMUS\_SEGF\_AFTERPREPARETIME, 448  
DMUS\_SEGF\_AFTERQUEUEUETIME, 448  
DMUS\_SEGF\_ALIGN, 448  
DMUS\_SEGF\_AUTOTRANSITION, 448  
DMUS\_SEGF\_BEAT, 448  
DMUS\_SEGF\_CONTROL, 448  
DMUS\_SEGF\_DEFAULT, 448  
DMUS\_SEGF\_GRID, 448  
DMUS\_SEGF\_MARKER, 448  
DMUS\_SEGF\_MEASURE, 448  
DMUS\_SEGF\_NOINVALIDATE, 448  
DMUS\_SEGF\_QUEUE, 448  
DMUS\_SEGF\_REFTIME, 448  
DMUS\_SEGF\_SECONDARY, 448  
DMUS\_SEGF\_SEGMENTEND, 448  
DMUS\_SEGF\_TIMESIG\_ALWAYS, 448  
DMUS\_SEGF\_USE\_AUDIOPATH, 447, 448  
DMUS\_SEGF\_VALID\_START\_BEAT, 448  
DMUS\_SEGF\_VALID\_START\_GRID, 448  
DMUS\_SEGF\_VALID\_START\_MEASURE, 448  
DMUS\_SEGF\_VALID\_START\_TICK, 448  
Doom, 26  
Doom 3, 20  
DOS, 34, 41  
dostępność tekstur wielokrotnych, 238  
downloadable sounds, 425  
DrawCube(), 120, 155  
DrawSurface(), 378  
DrawTextureCube(), 202  
druga zasada dynamiki, 511  
drzewa, 355  
DS3D\_DEFAULTMAXDISTANCE, 453  
DS3D\_DEFAULTMINDISTANCE, 453  
DS3DBUFFER, 452, 453  
DS3DLISTENER, 456  
DS3DMODE\_DISABLE, 453  
DS3DMODE\_HEADRELATIVE, 453  
DS3DMODE\_NORMAL, 453  
DSL, 427  
DSP, 424  
dualizm korpuskularno-falowy, 141  
dwuwymiarowe tablice, 138  
dxguid.lib, 398, 595  
dynamika Newtona, 510  
dyski, 329  
współrzędne tekstury, 330  
wycinki, 331  
dziedziczenie, 530  
dźwięki, 36, 421  
amplituda, 422  
cyfrowy zapis, 423  
częstotliwość, 422  
częstotliwość próbkowania, 423  
DLS, 36, 425  
efekt Dopplera, 454  
głośność, 450  
komputery, 423  
kończenie odtwarzania, 435  
MIDI, 425  
odbiorca, 455  
odtwarzanie, 468  
percepcja położenia źródła, 450  
położenie źródła, 450  
procesor sygnałowy, 424  
przepływ danych, 428  
przestrzenne, 450  
rozdzielcość próbkowania amplitudy, 424  
syntezator, 424  
ścieżki, 445  
tłumienie, 451  
współrzędne przestrzeni, 450  
źródło, 423

**E**

efekty  
 cienie, 379  
 Doplerta, 454  
 dźwiękowe, 27, 581  
 dźwięku przestrzennego, 450  
 flesza, 173  
 latarni, 173  
 mgła, 373, 375  
 odbicia, 166, 375  
 oświetlenia, 255  
 plakatowanie, 355  
 przezroczystości, 173, 174  
 specjalne, 187, 355  
 system częstek, 359  
 tekstury wielokrotnej, 261  
 wybuchu, 593  
 ekran, 301  
 kopiowanie danych, 187  
 eksplozje, 592  
 EndPaint(), 44  
 EnumDevices(), 400, 403, 404  
 EnumObjects(), 405  
 ewaluatory, 339

**F**

FAILED, 399  
 fclose(), 190, 191  
 filtrowanie, 212, 230  
 GL\_LINEAR, 208  
 GL\_LINEAR\_MIPMAP\_LINEAR, 208  
 GL\_LINEAR\_MIPMAP\_NEAREST, 208  
 GL\_NEAREST, 208  
 GL\_NEAREST\_MIPMAP\_LINEAR, 208  
 GL\_NEAREST\_MIPMAP\_NEAREST, 208  
 mipmapy, 213  
 tekstur, 204, 208  
 FindBoundingSphereRadius(), 532  
 flesze, 173  
 fopen(), 190  
 format pikseli, 58, 301  
 fov, 135  
 fread(), 190, 191  
 free(), 191  
 fseek(), 191  
 funkcje  
 AdjustWindowRectEx(), 69  
 BeginPaint(), 44  
 CalcBoundingBox(), 534  
 ChangeDisplaySettings(), 68  
 ChoosePixelFormat(), 59, 64, 302, 304  
 ClearColor(), 304

ClearColor(), 288, 291  
 CoCreateInstance(), 429, 430  
 CoInitialize(), 429  
 CreateBitmapFont(), 287, 288, 290  
 CreateFont(), 285, 287  
 CreateOutlineFont(), 290, 291  
 CreateWindow(), 48  
 CreateWindowEx(), 48, 49  
 DirectInput8Create(), 398, 399  
 DispatchMessage(), 50, 51  
 DisplayMD2(), 480  
 DisplayMD2Interpolate(), 485, 487, 488  
 DisplayScene(), 116  
 DrawCube(), 120, 155  
 DrawSurface(), 378  
 DrawTextureCube(), 202  
 EndPaint(), 44  
 EnumDevices(), 400, 403  
 fclose(), 190, 191  
 FindBoundingSphereRadius(), 532  
 fopen(), 190  
 fread(), 190, 191  
 free(), 191  
 fseek(), 191  
 fwrite(), 192  
 GetAsyncKeyState(), 394, 396  
 GetDC(), 54, 65  
 GetElapsedSeconds(), 520  
 GetFPS(), 519, 520  
 GetMessage(), 51  
 glAccum(), 324, 325  
 glActiveTextureARB(), 239, 240  
 glArrayElement(), 279  
 glBaseList(), 286  
 glBegin(), 99, 100, 101, 102  
 glBindTexture(), 203, 207, 217, 232, 240, 272  
 glBitmap(), 183, 184, 185  
 glBlendFunc(), 173  
 glBuild2DMipmaps(), 213  
 glCallList(), 100, 269, 271, 272  
 glCallLists(), 100, 270, 271, 286  
 glClear(), 66, 116, 128, 162, 304  
 glClearAccum(), 304, 326  
 glClearColor(), 128, 326, 340  
 glClearDepth(), 304  
 glClearIndex(), 145  
 glClearStencil(), 304, 326  
 glClientActivateTextureARB(), 239, 280  
 glColor(), 100  
 glColor2f(), 289  
 glColor3f(), 116, 143, 144, 164, 170, 181  
 glColor3fv(), 143  
 glColor4f(), 143  
 glColor4fv(), 143

glColorMask(), 377  
glColorMaterial(), 164  
glColorPointer(), 276  
glCopyPixels(), 187, 198, 306  
glCullFace(), 105  
glDeleteLists(), 271, 272, 288  
glDepthFunc(), 307  
glDepthMask(), 177, 179, 180, 377  
glDisable(), 101, 103, 170  
glDisableClientState(), 275, 280  
glDrawArrays(), 278, 280  
glDrawBuffer(), 306, 313, 314  
glDrawElements(), 278, 280, 282, 283  
glDrawPixels(), 185, 186, 187, 196, 198  
glDrawRangeElements(), 279  
glEdgeFlag(), 100, 106  
glEdgeFlagPointer(), 276  
glEnable(), 101, 103, 128, 152, 164, 173, 176,  
    178, 275, 317, 374  
glEnableClientState(), 275, 280  
glEnd(), 100, 101  
glEndList(), 268, 269  
glEvalCoord(), 100  
glEvalCoord1f(), 342  
glEvalMesh1(), 343  
glEvalMesh2(), 345, 349  
glEvalPoint(), 100  
glEXTLockArrays(), 283  
glEXTUnlockArrays(), 283  
glFlush(), 116, 162  
glFrontFace(), 106, 155, 162  
glFrustum(), 134, 135  
glGenLists(), 268, 270, 271, 285  
glGenTextures(), 203, 207  
glGet(), 93, 94, 98, 99, 101, 105, 110, 269  
glGetBooleanv(), 182  
glGetFloatv(), 102, 356  
glGetIntegerv(), 240  
glGetString(), 238  
glIndex(), 100  
glIndexf(), 145  
glIndexfv(), 145  
glIndexPointer(), 277  
glIsEnabled(), 98, 99, 101, 110  
glLightfv(), 155, 158, 159, 160, 161, 162, 166,  
    168, 172, 178  
glLightModel(), 165  
glLightModeli(), 165, 166  
glLineStipple(), 103  
glLineWidth(), 103  
glListBase(), 286  
glListName(), 271  
glLoadIdentity(), 114, 116, 119, 128, 130, 133,  
    138, 182  
glLoadMatrix(), 138, 139  
glLockArraysEXT(), 281  
glMap1f(), 341, 342, 344  
glMap2f(), 344, 345, 346, 349  
glMapGrid1f(), 342, 343, 345  
glMapGrid2(), 345  
glMaterial(), 100  
glMaterialf(), 163  
glMaterialfv(), 155, 164, 167  
glMateriali(), 169  
glMatrixMode(), 119, 123, 130, 133, 182, 253  
glMultiTexCoord2f(), 242, 246  
glMultiTexCoord2fARB(), 240, 241  
glMultiTexCoord3dARB(), 241  
glMultiTexCoord1fARB(), 239  
glMultMatrix(), 139, 253  
glMultMatrixf(), 378  
glNewList(), 268, 269  
glNormal(), 100, 279  
glNormal3f(), 151, 152  
glNormalPointer(), 277  
glOrtho(), 134, 182  
glPixelStorei(), 188, 196  
glPixelZoom(), 187, 188, 198  
glPointSize(), 101  
glPolygonMode(), 104  
glPolygonStipple(), 107  
glPopAttrib(), 271  
glPopMatrix(), 124, 128, 132, 253, 271, 570  
glPushAttrib(), 170, 172, 271  
glPushMatrix(), 124, 132, 169, 178, 253, 271,  
    378, 570  
glQuadricTexture(), 334  
glRasterPos(), 198  
glRasterPos2f(), 289  
glRasterPos2i(), 184, 186  
glReadBuffer(), 306  
glReadPixels(), 187, 191, 193, 198, 306  
glRotated(), 120  
glRotatef(), 116, 120, 121, 171, 253, 254  
glScale(), 123, 376, 378  
glScaled(), 122  
glScalef(), 122  
glSet(), 98  
glShadeModel(), 146, 162, 168, 340  
glStencilFunc(), 317, 318, 377  
glStencilOp(), 317, 377  
glTexCoord(), 100, 272  
glTexCoord2f(), 210, 218, 241  
glTexCoord2fv(), 210  
glTexCoordPointer(), 277, 280  
glTexEnv(), 240, 272  
glTexEnvi(), 209, 241  
glTexGen(), 240

funkcje  
 glTexGeni(), 298  
 glTexImage(), 240, 269  
 glTexImage1D(), 205, 206, 235  
 glTexImage2D(), 204, 205, 206, 212, 213, 217,  
   235  
 glTexImage3D(), 205, 206, 207, 235  
 glTexParameter(), 240  
 glTexParameter(), 209, 211, 212, 241  
 glTranslate(), 154  
 glTranslated(), 120  
 glTranslatef(), 116, 120, 169, 178  
 gluBeginSurface(), 353  
 gluBuild2DMipmaps(), 213, 230, 241  
 gluCheckExtension(), 238  
 gluCylinder(), 331  
 gluDeleteNurbsRenderer(), 353  
 gluDeleteQuadric(), 329, 335  
 gluDisk(), 330, 331  
 gluEndSurface(), 353  
 gluLookAt(), 114, 116, 232, 579  
 gluNewNurbsRenderer(), 352  
 gluNewQuadric(), 327, 334  
 glUnlockArraysEXT(), 281  
 gluNurbsProperty(), 352  
 gluNurbsSurface(), 353  
 gluNutbsProperty(), 352  
 gluOrtho2D(), 134, 182  
 gluPartialDisk(), 331  
 gluPerspective(), 135  
 gluProjection(), 182  
 gluQuadricDrawStyle(), 328, 334  
 gluQuadricNormals(), 328  
 gluQuadricTexture(), 329  
 gluSphere(), 332  
 glVertex(), 100, 101, 134  
 glVertex3f(), 100, 218, 268, 342  
 glVertexPointer(), 277  
 glViewport(), 66, 135, 182  
 graficzne, 56  
 InitMultiTex(), 244  
 InStr(), 243  
 konwencja wywołania, 42  
 LoadBitmapFile(), 190, 191, 200, 201, 216, 230,  
   233, 256  
 LoadCursor(), 47  
 LoadGrayBitmap(), 259  
 LoadIcon(), 46  
 LoadLightmap(), 259  
 LoadMD2Model(), 476, 478  
 LoadTextureFile(), 245, 259  
 LoadTGAFile(), 194  
 LockFPS(), 519, 520  
 main(), 42  
 MatrixAdd(), 78  
 MatrixMult(), 79  
 PeekMessage(), 50, 51, 55, 66, 70  
 PlaneView(), 116  
 PointInPolygon(), 538  
 PositionLights(), 376  
 PostQuitMessage(), 44  
 PrintString(), 288, 291  
 QueryPerformanceCounter(), 517  
 QueryPerformanceFrequency(), 517  
 rastra(), 184  
 RegisterClassEx(), 48  
 Render(), 184  
 rozszerzeń, 239  
 ScalarMatrixMult(), 78  
 SetPixelFormat(), 59, 64, 129, 302, 304  
 SetShadowMatrix(), 385  
 SetTextColor(), 55  
 SetupPixelFormat(), 64, 65, 301, 303  
 ShowCursor(), 68, 222, 559  
 ShowWindow(), 50, 222  
 sin(), 215  
 stanu, 93  
 SwapBuffers(), 66, 157, 179, 184  
 tekstu, 209  
 TranslateMessage(), 50, 51, 55  
 trygonometryczne, 80  
 UpdateWindow(), 50  
 WGL, 55, 56  
 wglCreateContext(), 56, 57, 65  
 wglDeleteContext(), 56  
 wglGetProcAddress(), 239  
 wglMakeCurrent(), 57, 65  
 wglSwapLayerBuffers(), 59  
 wglUseBitmapFonts(), 288  
 wglUseFontBitmaps(), 285, 286  
 wglUseFontOutlines(), 289  
 wglUseOutlineFonts(), 299  
 WinMain(), 42, 45, 50, 55, 69, 130, 171  
 WndProc(), 54, 64, 227  
 WriteBitmapFile(), 192  
 WriteTGAFile(), 196  
 fwrite(), 192

**G**

g\_normalObject, 334  
 g\_texturedObject, 334  
 g\_wireframeObject, 334  
 GDI, 31, 303  
 GeForce, 19  
 GetAsyncKeyState(), 394, 396  
 GetAudioPathConfig(), 446  
 GetCapabilities(), 405

GetDC(), 54, 65  
GetDefaultAudioPath(), 446  
GetDeviceData(), 409  
GetDeviceState(), 408  
GetElapsedSeconds(), 520  
GetElapsedTime(), 521  
GetFPS(), 519, 520  
GetMessage(), 51  
GetObject(), 446  
GetObjectInPath(), 449  
GL, 19  
GL\_ACCUM, 325  
GL\_ACCUM\_BUFFER\_BIT, 305  
GL\_ADD, 325  
GL\_ALPHA, 186, 205  
GL\_ALWAYS, 307, 313, 314, 318  
GL\_AMBIENT, 158, 164  
GL\_AMBIENT\_AND\_DIFFUSE, 164  
GL\_ARB\_multitexture, 238, 244  
GL\_AUTO\_NORMAL, 345  
GL\_BACK, 104, 105, 163, 305, 306  
GL\_BACK\_LEFT, 306  
GL\_BACK\_RIGHT, 306  
GL\_BGR, 186  
GL\_BGRA, 186  
GL\_BITMAP, 186, 205  
GL\_BLEND, 173, 209  
GL\_BLUE, 186, 205  
GL\_BYTE, 186, 205  
GL\_CCW, 106, 155  
GL\_CLAMP\_TO\_EDGE, 211  
GL\_COLOR, 187  
GL\_COLOR\_ARRAY, 276  
GL\_COLOR\_BUFFER\_BIT, 305  
GL\_COLOR\_INDEX, 186, 205  
GL\_COLOR\_MATERIAL, 164, 291  
GL\_COMPILE, 269  
GL\_COMPILE\_AND\_EXECUTE, 269  
GL\_CONSTANT\_ATTENUATION, 158, 160  
GL\_CULL\_FACE, 105, 155  
GL\_CURRENT\_RASTER\_POSITION\_VALID, 182  
GL\_CW, 106  
GL\_DECAL, 209  
GL\_DECR, 318  
GL\_DEPTH, 187  
GL\_DEPTH\_BUFFER\_BIT, 305  
GL\_DEPTH\_TEST, 155, 176  
GL\_DIFFUSE, 158, 159, 164  
GL\_DOUBLE, 277  
GL\_DST\_ALPHA, 173, 174  
GL\_DST\_COLOR, 173  
GL\_EDGE\_FLAG\_ARRAY, 276  
GL\_EMISSION, 164  
GL\_EQUAL, 307, 318  
GL\_EXP, 374  
GL\_EXP2, 374  
GL\_EXT\_compiled\_vertex\_array, 281, 283  
GL\_EXTENSIONS, 238  
GL\_EYE\_LINEAR, 298  
GL\_FALSE, 106, 166, 177, 268  
GL\_FILL, 105, 345  
GL\_FLAT, 146  
GL\_FLOAT, 205  
GL\_FOG, 374  
GL\_FOG\_COLOUR, 374  
GL\_FOG\_DENSITY, 374  
GL\_FOG\_END, 374  
GL\_FOG\_INDEX, 374  
GL\_FOG\_MODE, 374  
GL\_FOG\_START, 374  
GL\_FRONT, 105, 163, 305, 306  
GL\_FRONT\_AND\_BACK, 104, 105, 163, 305  
GL\_FRONT\_LEFT, 306  
GL\_FRONT\_RIGHT, 306  
GL\_GEQUAL, 307, 318  
GL\_GREATER, 307, 318  
GL\_GREEN, 186, 205  
GL\_INCR, 318  
GL\_INDEX\_ARRAY, 276  
GL\_INT, 186, 205, 277  
GL\_INVERT, 318  
GL\_KEEP, 318  
GL\_LEFT, 306  
GL\_LESS, 307, 313, 318  
GL\_LIGHT\_MODEL\_AMBIENT, 165  
GL\_LIGHT\_MODEL\_COLOR\_CONTROL, 165, 166  
GL\_LIGHT\_MODEL\_LOCAL\_VIEWER, 165  
GL\_LIGHT\_MODEL\_TWO\_SIDE, 165, 166  
GL\_LIGHT0, 158, 159, 167  
GL\_LIGHT1, 158  
GL\_LIGHT4, 155  
GL\_LIGHT7, 158  
GL\_LIGHTING, 155  
GL\_LIGHTING\_BIT, 172  
GL\_LINE, 105, 343, 345  
GL\_LINE\_LOOP, 99, 278  
GL\_LINE\_SMOOTH, 103  
GL\_LINE\_STIPPLE, 103, 104  
GL\_LINE\_STIPPLE\_REPEAT, 104  
GL\_LINE\_STRIP, 99, 278  
GL\_LINE\_WIDTH, 103  
GL\_LINE\_WIDTH\_GRANULARITY, 103  
GL\_LINE\_WIDTH\_RANGE, 103  
GL\_LINEAR, 208, 374  
GL\_LINEAR\_ATTENUATION, 158, 160  
GL\_LINEAR\_MIPMAP\_LINEAR, 208, 213

GL\_LINEAR\_MIPMAP\_NEAREST, 208, 213  
GL\_LINES, 99, 102, 278  
GL\_LIST\_BASE, 271  
GL\_LOAD, 325  
GL\_LUMINANCE, 205, 257, 260  
GL\_LUMINANCE\_ALPHA, 205  
GL\_MAP1\_COLOR\_4, 342  
GL\_MAP1\_INDEX, 342  
GL\_MAP1\_NORMAL, 342  
GL\_MAP1\_TEXTURE\_COORD\_1, 342  
GL\_MAP1\_TEXTURE\_COORD\_2, 342  
GL\_MAP1\_TEXTURE\_COORD\_3, 342  
GL\_MAP1\_TEXTURE\_COORD\_4, 342  
GL\_MAP1\_VERTEX\_3, 341, 342  
GL\_MAP1\_VERTEX\_4, 342  
GL\_MAP2\_TEXTURE\_COORD\_2, 346  
GL\_MAP2\_VERTEX\_3, 353  
GL\_MAX\_TEXTURE\_UNITS\_ARB, 280  
GL\_MODELVIEW, 119  
GL\_MODULATE, 209, 257, 260  
GL\_MULT, 325  
GL\_NEAREST, 208  
GL\_NEAREST\_MIPMAP\_LINEAR, 208, 213  
GL\_NEAREST\_MIPMAP\_NEAREST, 208, 213  
GL\_NEVER, 307, 318  
GL\_NONE, 305  
GL\_NORMAL\_ARRAY, 276  
GL\_NORMALIZE, 152, 157  
GL\_NOTEQUAL, 307, 318  
GL\_OBJECT\_LINEAR, 298  
GL\_ONE, 173, 174  
GL\_ONE\_MINUS\_DST\_ALPHA, 173, 174  
GL\_ONE\_MINUS\_DST\_COLOR, 173  
GL\_ONE\_MINUS\_SRC\_ALPHA, 174, 180  
GL\_ONE\_MINUS\_SRC\_COLOR, 174  
GL\_PACK\_ALIGNMENT, 188  
GL\_POINT, 105, 343, 345  
GL\_POINT\_SIZE, 101  
GL\_POINT\_SIZE\_GRANULARITY, 102  
GL\_POINT\_SIZE\_RANGE, 102  
GL\_POINT\_SMOOTH, 101  
GL\_POINTS, 99, 100, 278  
GL\_POLYGON, 99, 109, 146, 278  
GL\_POLYGON\_MODE, 105  
GL\_POLYGON\_SMOOTH, 106  
GL\_POLYGON\_STIPPLE, 107  
GL\_POSITION, 158, 172  
GL\_PROJECTION, 119  
GL\_PROXY\_TEXTURE\_2D, 205  
GL\_QUAD\_STRIP, 99, 278  
GL\_QUADRATIC\_ATTENUATION, 158, 160  
GL\_QUADS, 99, 108, 218, 278  
GL\_RED, 186, 205  
GL\_REPEAT, 211  
GL\_REPLACE, 318  
GL\_RESCALE\_NORMALIZE, 152  
GL\_RETURN, 325  
GL\_RGB, 186, 205, 257  
GL\_RGBA, 186, 205  
GL\_RIGHT, 306  
GL\_SHININESS, 164, 167  
GL\_SHORT, 186, 205, 277  
GL\_SINGLE\_COLOR, 166  
GL\_SMOOTH, 146, 155  
GL\_SPECULAR, 158, 159, 164  
GL\_SPHERE\_MAP, 298  
GL\_SPOT\_CUTOFF, 158, 160, 161  
GL\_SPOT\_DIRECTION, 158, 172  
GL\_SPOT\_EXPONENT, 158, 161  
GL\_SRC\_ALPHA, 173, 174, 180  
GL\_SRC\_ALPHA\_SATURATE, 173, 174  
GL\_SRC\_COLOR, 174  
GL\_STACK\_OVERFLOW, 124  
GL\_STACK\_UNDERFLOW, 124  
GL\_STENCIL, 187  
GL\_STENCIL\_BUFFER\_BIT, 305  
GL\_STENCIL\_TEST, 317  
GL\_TEXTURE, 119  
GL\_TEXTURE\_1D, 206, 207, 208  
GL\_TEXTURE\_2D, 203, 205, 207, 208  
GL\_TEXTURE\_3D, 207, 208  
GL\_TEXTURE\_COORD\_ARRAY, 276, 280  
GL\_TEXTURE\_ENV, 209  
GL\_TEXTURE\_MAG\_FILTER, 208  
GL\_TEXTURE\_MIN\_FILTER, 208  
GL\_TEXTURE\_WRAP\_S, 211  
GL\_TEXTURE\_WRAP\_T, 211  
GL\_TEXTURE0\_ARB, 240  
GL\_TEXTUREi\_ARB, 280  
GL\_TRIANGLE, 108  
GL\_TRIANGLE\_FAN, 99, 108, 278, 471  
GL\_TRIANGLE\_STRIP, 99, 108, 224, 278, 471  
GL\_TRIANGLES, 99, 107, 278  
GL\_TRUE, 268  
GL\_UNPACK\_ALIGNMENT, 188  
GL\_UNSIGNED\_BYTE, 186, 205  
GL\_UNSIGNED\_INT, 186, 205  
GL\_UNSIGNED\_SHORT, 186, 205  
GL\_VERTEX\_ARRAY, 276  
GL\_ZERO, 173, 174, 318  
glAccum(), 324, 325  
glActiveTextureARB(), 239, 240  
glArrayElement(), 279  
GLAUX, 163  
glaux.lib, 28  
glBaseList(), 286  
glBegin(), 99, 100, 101, 102  
glBindTexture(), 203, 207, 217, 232, 240, 272

glBitmap(), 183, 184, 185  
glBlendFunc(), 173  
glBuild2DMipmaps(), 213  
glCallList(), 100, 269, 271, 272  
glCallLists(), 100, 270, 271, 286  
GLclampd, 304  
GLclampf, 304  
glClear(), 66, 116, 128, 162, 304  
glClearAccum(), 304, 326  
glClearColor(), 128, 326, 340  
glClearDepth(), 304  
glClearIndex(), 145  
glClearStencil(), 304, 326  
glClientActivateTextureARB(), 239, 280  
glColor(), 100  
glColor2f(), 289  
glColor3f(), 116, 143, 144, 164, 170, 181  
glColor3fv(), 143  
glColor4f(), 143  
glColor4fv(), 143  
glColorMask(), 377  
glColorMaterial(), 164  
glColorPointer(), 276  
glCopyPixels(), 187, 198, 306  
glCullFace(), 105  
glDeleteLists(), 271, 272, 288  
glDepthFunc(), 307  
glDepthMask(), 177, 179, 180, 377  
glDisable(), 101, 103, 170  
glDisableClientState(), 275, 280  
glDrawArrays(), 278, 280  
glDrawBuffer(), 306, 313, 314  
glDrawElements(), 278, 280, 282, 283  
glDrawPixels(), 185, 186, 187, 196, 198  
glDrawRangeElements(), 279  
glEdgeFlag(), 100, 106  
glEdgeFlagPointer(), 276  
glEnable(), 101, 103, 128, 152, 164, 173, 176, 178, 275, 317, 374  
glEnableClientState(), 275, 280  
glEnd(), 100, 101  
glEndList(), 268, 269  
glEvalCoord(), 100  
glEvalCoord1f(), 342  
glEvalMesh1(), 343  
glEvalMesh2(), 345, 349  
glEvalPoint(), 100  
glEXTLockArrays(), 283  
glEXTUnlockArrays(), 283  
glFlush(), 116, 162  
glFrontFace(), 106, 155, 162  
glFrustum(), 134, 135  
glGenLists(), 268, 270, 271, 285  
glGenTextures(), 203, 207  
glGet(), 93, 94, 98, 99, 101, 105, 110, 269  
glGetBooleanv(), 182  
glGetFloatv(), 102, 356  
glGetIntegerv(), 240  
glGetString(), 238  
glIndex(), 100  
glIndexf(), 145  
glIndexfv(), 145  
glIndexPointer(), 277  
glIsEnabled(), 98, 99, 101, 110  
glLightfv(), 155, 158, 159, 160, 161, 162, 166, 168, 172, 178  
glLightModel(), 165  
glLightModeli(), 165, 166  
glLineStipple(), 103  
glLineWidth(), 103  
glListBase(), 286  
glListName(), 271  
glLoadIdentity(), 114, 116, 119, 128, 130, 133, 138, 182, 220  
glLoadMatrix(), 138, 139  
glLockArraysEXT(), 281  
glMap1f(), 341, 342, 344  
glMap2f(), 344, 345, 346, 349  
glMapGrid1f(), 342, 343, 345  
glMapGrid2(), 345  
glMaterial(), 100  
glMaterialf(), 163  
glMaterialfv(), 155, 164, 167  
glMateriali(), 169  
glMatrixMode(), 119, 123, 130, 133, 182, 220, 253  
glMultiTexCoord2f(), 242, 246  
glMultiTexCoord2fARB(), 240, 241  
glMultiTexCoord3dARB(), 241  
glMultiTexCoord1fARB(), 239  
glMultMatrix(), 139, 253  
glMultMatrixf(), 378  
glNewList(), 268, 269  
glNormal(), 100, 279  
glNormal3f(), 151, 152  
glNormalPointer(), 277  
globalne światło otoczenia, 165  
glOrtho(), 134, 182  
glPixelStorei(), 188, 196  
glPixelZoom(), 187, 188, 198  
glPointSize(), 101  
glPolygonMode(), 104  
glPolygonStipple(), 107  
glPopAttrib(), 271  
glPopMatrix(), 124, 128, 132, 253, 271, 570  
glPushAttrib(), 170, 172, 271  
glPushMatrix(), 124, 132, 169, 178, 253, 271, 378, 570  
glQuadraticTexture(), 334

glRasterPos(), 198  
glRasterPos2f(), 289  
glRasterPos2i(), 182, 184, 186  
glReadBuffer(), 306  
glReadPixels(), 187, 191, 193, 198, 306  
glRotated(), 120  
glRotatef(), 116, 120, 121, 171, 253, 254  
glScale(), 123, 376, 378  
glScaled(), 122  
glScalef(), 122  
glSet(), 98  
glShadeModel(), 146, 162, 168, 340  
glStencilFunc(), 317, 318, 377  
glStencilOp(), 317, 377  
glTexCoord(), 100, 272  
glTexCoord2f, 210  
glTexCoord2f(), 210, 218, 241  
glTexCoord2fv(), 210  
glTexCoordPointer(), 277, 280  
glTexEnv(), 240, 272  
glTexEnvi(), 209, 241  
glTexGen(), 240  
glTexGeni(), 298  
glTexImage(), 240, 269  
glTexImage1D(), 205, 206, 235  
glTexImage2D(), 204, 205, 206, 212, 213, 217, 235  
glTexImage3D(), 205, 206, 207, 235  
glTexParameter(), 240  
glTexParameter(), 209, 211, 212, 241  
glTranslate(), 154  
glTranslated(), 120  
glTranslatef(), 116, 120, 169, 178  
GLU, 30, 31, 115, 134, 327, 595  
GLU\_DISPLAY\_MODE, 352  
GLU\_FILL, 328, 352  
GLU\_FLAT, 329  
GLU\_INSIDE, 329  
GLU\_LINE, 328  
GLU\_NONE, 329  
GLU\_OUTLINE\_PATCH, 352  
GLU\_OUTLINE\_POLYGON, 352  
GLU\_OUTSIDE, 329  
GLU\_POINT, 328  
GLU\_SILHOUETTE, 328  
GLU\_SMOOTH, 329  
glu32.lib, 28, 595  
gluBeginSurface(), 353  
gluBuild2DMipmaps(), 213, 230, 241  
gluCheckExtension(), 238  
gluCylinder(), 331  
gluDeleteNurbsRenderer(), 353  
gluDeleteQuadric(), 329, 335  
gluDisk(), 330, 331  
gluEndSurface(), 353  
gluLookAt(), 114, 116, 232, 579  
gluNewNurbsRenderer(), 352  
gluNewQuadric(), 327, 334  
glUnlockArraysEXT(), 281  
gluNurbsProperty(), 352  
gluNurbsSurface(), 353  
gluNutbsProperty(), 352  
gluOrtho2D(), 134, 182  
gluPartialDisk(), 331  
gluPerspective(), 135  
gluProjection(), 182  
gluQuadricDrawStyle(), 328, 334  
gluQuadricNormals(), 328  
gluQuadricTexture(), 329  
gluSphere(), 332  
GLUT, 32  
glVertex(), 100, 101, 134  
glVertex3f(), 100, 210, 218, 268, 342  
glVertexPointer(), 277  
glViewport(), 66, 135, 182, 220  
GLYPHMETRICSFLOAT, 289, 291  
głębia, 306  
    koloru, 143  
    porównywanie, 307  
Gouraud, 146, 180  
grafika, 25, 99  
    antialiasing, 101  
    cieniowanie, 145  
    czworokąty, 108  
    dwuwymiarowa, 134, 182  
    odcinki, 99, 102  
    OpenGL, 60, 70  
    potok tworzenia, 32  
    punkty, 99, 100  
    rastrowa, 181  
    trójkąty, 99, 107  
    wielokąty, 104  
grafika trójwymiarowa, 19, 67, 71, 237  
    iloczyn wektorowy, 74  
    macierze, 75  
    obcinanie, 86  
    obroty, 80  
    odwzorowania tekstur, 88  
    przekształcenia, 79  
    przesunięcie, 80  
    punkty, 71  
    rzutowanie, 82  
    skalary, 71  
    skalowanie, 82  
    światło, 86  
    wektory, 71  
Graphics Device Interface, 31  
graphics library, 19  
grawitacja, 511

gry, 26  
architektura, 28  
cykl, 574  
czas zabijania, 585  
Doom, 26  
Doom 3, 20  
hokej, 538  
interakcja, 391  
komputerowe, 19  
podsystemy, 27  
projekt, 585  
Quake, 20, 26  
Quake 2, 470  
Quake 3, 30  
sieciowe, 34  
sterowanie, 391  
Unreal Tournament, 26  
Wolfenstein, 26  
GUID\_SysKeyboard, 400  
GUID\_SysMouse, 400

## H

HAL, 35, 37  
Hardware Abstraction Layer, 35  
Hardware Emulation Layer, 35  
HCURSOR, 47  
hDC, 64  
HEL, 35, 37  
hermetyzacja modelu, 490  
HICON, 46  
hierarchiczne tworzenie grafiki, 253  
hInst, 398  
hInstance, 42  
HIWORD, 65  
hokej, 538  
    gracz, 550  
    krążek, 544  
    lodowisko, 539  
    zderzenia, 544  
hPrevInstance, 42  
hwnd, 43

## I

IDC\_APPSTARTING, 47  
IDC\_ARROW, 47  
IDC\_CROSS, 47  
IDC\_HELP, 47  
IDC\_IBEAM, 47  
IDC\_NO, 47  
IDC\_SIZEALL, 47  
IDC\_SIZENESW, 47  
IDC\_SIZENS, 47

IDC\_SIZENWSE, 47  
IDC\_SIZEWE, 47  
IDC\_UPARROW, 47  
IDC\_WAIT, 47  
IDI\_APPLICATION, 46  
IDI\_ASTERISK, 46  
IDI\_ERROR, 46  
IDI\_EXCLAMATION, 46  
IDI\_HAND, 46  
IDI\_INFORMATION, 46  
IDI\_QUESTION, 46  
IDI\_WARNING, 46  
IDI\_WINLOGO, 46  
IDirectInputDevice  
    SetCooperativeLevel(), 407  
    SetProperty(), 407  
IDirectInputDevice8  
    Acquire(), 408  
    CreateDevice(), 400  
    EnumDevices(), 400, 404  
    EnumObjects(), 405  
    GetCapabilities(), 404  
    GetDeviceData(), 409  
    GetDeviceState(), 408  
    Poll(), 409  
    Release(), 410  
    Unacquire(), 409  
IDirectMusicAudioPath8  
    GetObjectInPath(), 449, 455  
    SetVolume(), 448  
IDirectMusicLoader8, 430  
    GetObject(), 430, 432, 446  
    LoadObjectFromFile(), 432, 446  
    SetSearchDirectory(), 430, 431  
IDirectMusicPerformance8, 429, 430  
    CloseDown(), 435  
    CreateAudioPath(), 445, 446  
    CreateStandardAudioPath(), 445, 446  
    GetDefaultAudioPath(), 446  
    GetObjectInPath(), 451  
    InitAudio(), 429, 445, 446  
    IsPlaying(), 434  
    PlaySegment(), 446  
    PlaySegmentEx(), 430, 432, 446, 447  
    SetDefaultAudioPath(), 446  
    Stop(), 433, 435  
IDirectMusicSegment8, 430, 582  
    Download(), 430, 432  
    GetAudioPathConfig(), 446  
    SetLoopPoints(), 434  
    SetRepeats(), 434, 435  
IDirectSound3DBuffer, 456  
IDirectSound3DBuffer8, 451, 582  
    GetAllParameters(), 452  
    SetAllParameters(), 452

IDirectSound3DListener8, 455  
 GetAllParameters(), 455  
 GetDistanceFactor(), 455  
 GetDopplerFactor(), 455  
 GetOrientation(), 455  
 GetPosition(), 455  
 GetRolloffFactor(), 455  
 GetVelocity(), 455  
 SetAllParameters(), 455  
 SetDistanceFactor(), 455  
 SetDopplerFactor(), 455  
 SetOrientation(), 455  
 SetPosition(), 455  
 SetRolloffFactor(), 455  
 SetVelocity(), 455  
 ikony, 46  
 iloczyn wektorowy, 74  
 inicjowanie grafiki, 216  
 InitAudio(), 429, 446  
 InitMultiTex(), 244  
 instalacja, 37  
 InStr(), 243  
 int, 42  
 interaktywność, 391, 550  
 interfejs  
     DirectInput, 398  
     DirectX, 33  
     GDI, 31  
 IDirectSound3DBuffer8, 451  
 IDirectSound3DListener8, 455  
 IRIS GL, 19  
 OpenGL, 19  
 użytkownika gry, 594  
 Internet, 26, 36  
 interpolacja, 484  
 IRIS GL, 19  
 IsPlaying(), 434

**J**

jednostki tekstur, 237, 240  
 język  
     C, 41  
     C++, 28, 41, 526  
 JOYERR\_NOERROR, 396  
 JOYSTICKID1, 396  
 JOYSTICKID2, 396

**K**

kafelkowanie, 32, 210  
 kamera, 112, 243, 576, 577  
     orientacja, 115  
     położenie, 114  
     stacjonarna, 115

kanał  
     alfa, 193  
     wykonania, 427  
 karty dźwiękowe, 34, 36, 421  
 karty graficzne, 398  
     VGA, 34  
 katalogi, 46  
 kąt  
     odbicia, 538  
     padania, 538  
     widzenia obserwatora, 135  
 kierunek widzenia obserwatora, 114  
 kinematyka, 526  
 klasy  
     CAudioSystem, 580, 581  
     CCamera, 577  
     CEnemy, 588  
     CEngine, 573  
     CEntity, 581, 588  
     CGUI, 594  
     CHiResTimer, 518, 553  
     CInputSystem, 411, 572  
     CKeyboard, 414  
     CMD2Model, 488, 580  
     CMouse, 416  
     CNode, 562, 563  
     COBJECT, 530, 531, 566, 569  
     COGLWindow, 562, 570  
     COgroEnemy, 590  
     CParticleSystem, 366, 583, 593  
     CPlane, 527  
     CPlayer, 550  
     CPuck, 544  
     CRocket, 592  
     CSimpEngine, 576  
     CSnowstorm, 369  
     CTable, 539  
     CTerrain, 586  
     CTexture, 540  
     CVector, 521  
     CWorld, 580  
 IDirectMusicPerformance8, 429  
 okien, 44  
 klatki animacji, 157  
 klawiatura, 172, 394, 572  
     obsługa Win32, 394  
 klawisze, 51, 392, 393, 406  
     wirtualne kody, 395  
 kod  
     ASCII, 290  
     SCII, 172  
     wirtualne klawisze, 395  
 kolejka zdarzeń, 40

kolory, 55, 58, 141  
alfa, 173  
bufor, 143  
głębia, 143  
intensywność, 143  
łączenie, 173  
mapa, 142  
model indeksowy, 142, 144  
model RGB, 142  
model RGBA, 142, 143, 144  
model widzenia, 142  
OpenGL, 142  
paleta, 144  
pixela, 59  
składowe, 143  
spektrum, 143  
sześciian, 143  
wierzchołków, 276  
współczynnik alfa, 142  
kompilacja, 595  
kompilator, 28  
kompresja BMP, 190  
komunikacja, 36  
komunikaty, 42  
obsługa, 43  
pętla przetwarzania, 50  
urządzeń wejścia, 392  
Windows, 391, 394  
WM\_CHAR, 392  
WM\_CLOSE, 44  
WM\_CREATE, 57, 64  
WM\_DESTROY, 57  
WM\_KEYDOWN, 172, 392, 393  
WM\_KEYUP, 392, 393  
WM\_LBUTTONDOWN, 392  
WM\_LBUTTONUP, 392  
WM\_MBUTTONDOWN, 392  
WM\_MBUTTONUP, 392  
WM\_MOUSEMOVE, 393  
WM\_MOUSEWHEEL, 393  
WM\_PAINT, 44  
WM\_QUIT, 44, 55  
WM\_RBUTTONDOWN, 392  
WM\_RBUTTONUP, 393  
WM\_SIZE, 65  
WM\_TIMER, 517  
kontekst  
  grafiki, 56, 135  
  urządzenia, 54, 70  
  urządzenia hDC, 64  
  wybór, 65  
kopiowanie danych ekranu, 187  
krawędzie, 106  
krzywe, 38, 337

Béziera, 339, 340, 341, 342  
B-sklejane, 350, 354  
ciągłość, 338, 339  
dwuwymiarowe, 343  
ewaluatory, 339  
kształt, 339  
NURBS, 350  
punkty kontrolne, 338, 340, 342  
równania, 337, 338  
siatka równoodległa, 342  
kształty, 354  
kule, 332  
kursor myszy, 46, 47

## L

LARGE\_INTEGER, 517  
licznik  
  czasu, 517  
  systemu Windows, 517  
Lightwave, 469  
linie, 37  
liniowa interpolacja, 208  
Linux, 20  
listy cykliczne, 563  
listy wyświetlania, 267, 271  
  identyfikatory, 272  
rekursja, 271  
tekstyry, 272  
tworzenie, 267, 268  
usuwanie, 271  
wstawianie poleceń, 268  
wykonywanie, 269  
LoadBitmapFile(), 190, 191, 200, 201, 216, 230,  
  233, 256  
LoadCursor(), 47  
LoadGrayBitmap(), 259  
LoadIcon(), 46  
LoadLightmap(), 259  
LoadMD2Model(), 476, 478  
LoadObjectFromFile(), 432, 446  
LoadTextureFile(), 245, 259  
LoadTGAFile(), 194  
LockFPS(), 519, 520  
LONGLONG, 517  
WORD, 65  
LPARAM, 65  
lpCmdLine, 42  
LPDIRECTINPUT8, 398, 399  
LRESULT, 42  
lustra, 378

**Ł**

ładowanie

- elementów macierzy, 138
  - ikon, 46
  - modelu z pliku, 275
  - plików BMP, 190
  - plików PCX, 501
  - plików Targa, 194
- łączenie
- kolorów, 173, 176
  - wierzchołków, 38

**M**

macierze, 32, 75, 111, 119

- definiowane, 139
- dodawanie, 76
- jednostkowe, 75, 120
- kolumnowe, 81
- ładowanie elementów, 138
- mnożenie, 76, 139
- modelowania, 119, 120
- obrotu, 80
- odejmowanie, 76
- odwrotne, 356
- przekształceń, 138
- przekształceń perspektywicznych, 84
- rzutowania, 119, 133
- rzutowania cieni, 380
- stos, 119, 123
- tekstur, 119, 253
- transpozycja, 357
- zerowe, 75

main(), 42

manipulator, 36, 394, 572

obsługa Win32, 396

mapy

- kolorów, 142, 145
- oświetlenia, 255, 265, 375
- pikseli, 185
- sferyczne, 298
- tekstur, 205
- terenu, 228

mapy bitowe, 45, 181, 183, 197

OpenGL, 181

pakowanie danych, 188

położenie, 182

rysowanie, 183

umieszczanie, 182

Windows, 188

masa obiektu, 507

maska powielania, 317

maszyna stanów, 93, 271

elementy grafiki, 99

parametry, 93

punkty, 100

zakończenie tworzenia grafiki, 100

zmiana parametrów, 98

ateriały, 148, 155

definiowanie, 163

śledzenie kolorów, 164

MatrixAdd(), 78

MatrixMult(), 79

Maya, 469

MD2, 470, 472, 584

animacja modelu, 471, 483

ładowanie modelu, 476

nagłówek, 470

pokrywanie modelu tekturem, 482

siatka modelu, 474

stany animacji, 499

sterowanie animacją modelu, 498

struktury, 475

wyświetlanie modelu, 480

mechanika obiektów, 27

mechanizm rozszerzeń, 38

menedżer systemów cząstek, 365

metody wirtualne, 531

mgła, 370, 373, 602

gęstość, 374

GL\_EXP, 374, 375

GL\_EXP2, 374, 375

GL\_LINEAR, 374, 375

indeks koloru, 374

koniec, 374

objętościowa, 375

OpenGL, 374

tryb tworzenia, 374

właściwości, 374

Microsoft DDK, 38

Microsoft DirectX, 34

Microsoft Visual C++, 28

MIDI, 425, 428

miksowanie dźwięku, 34

mipmapy, 32, 208, 212, 235, 541

automatyczne tworzenie, 213

MMRESULT, 396

mmsystem.h, 396

model

cieniowania, 145

obiektowy COM, 397

oświetlenia, 164

model kolorów, 59

indeksowy, 142

RGB, 142, 185

RGBA, 142, 144, 159

- modele, 123  
ładowanie z pliku, 275  
robota, 124  
modele trójwymiarowe, 469  
3D Studio Max, 469  
animacja, 483  
format MD2, 470  
formaty plików, 469  
Lightwave, 469  
ładowanie plików PCX, 501  
Maya, 469  
pokrywanie teksturową, 482  
sterowanie animacją, 498  
tworzenie, 469  
modelowanie, 112  
cieni, 379  
oświetlenia, 148  
światła, 505  
światła rzeczywistego, 363, 516  
morphing, 484  
MP3, 468  
MPG, 36  
MSDN, 55  
Musical Instrument Digital Interface, 425  
mysz, 392, 550, 572
- N**
- narzędzia, 28  
non-uniform rational B-splines, 350  
normalizacja, 152  
wektora, 72  
normalne, 148, 151  
jednostkowe, 152  
obliczanie, 149  
powierzchni, 148  
nShowCmd, 42  
NULL, 42  
NURBS, 32, 350
- O**
- obcinanie, 86  
obiekty, 529  
DMO, 36  
dwuwymiarowe, 356  
flesze, 173  
ładujące, 426  
materiały, 148, 163  
odwzorowanie otoczenia, 250  
powierzchni drugiego stopnia, 328  
prostopadłościany ograniczeń, 533  
przeslonięte, 326  
ruch, 111
- sfery ograniczeń, 532  
tekstur, 200, 207  
trójwymiarowe, 86, 199  
wykonania, 427  
wyznaczanie oświetlenia, 155
- obrazy  
dwuwymiarowe, 32  
graficzne, 185  
odbicia, 188  
odeczytywanie obrazu z ekranu, 187  
pomniejszanie, 188  
powiększanie, 188  
stereoskopowe, 38  
upakowanie danych mapy pikseli, 188
- obroty, 64, 80, 117, 120  
kierunek, 121  
oś, 121
- obserwator, 113  
kąt widzenia, 135  
kierunek widzenia, 114
- obsługa  
komunikatów, 43  
zdarzeń, 42
- odbicia, 188, 375, 384  
bufor głębi, 376  
bufor powielania, 377  
nieregularne, 378  
płaszczyzna dowolnie zorientowana, 378  
płaszczyzna odbicia, 376  
przedmioty metalowe, 378  
światła, 376
- odbiórca dźwięku, 456
- odcinki, 99, 102  
antialiasing, 103  
szerokość, 103  
wzór linii, 103
- odległość, 506
- odpytywanie urządzeń, 409
- odrysowanie okna, 44
- odtwarzanie dźwięku, 429, 580  
głośność kanałów, 448  
ścieżki dźwięku, 447
- odtwarzanie plików, 36
- odwzorowanie  
akcji, 410  
otoczenia, 250, 265  
tekstur, 88, 199, 235, 237
- ogniskowa, 84
- okna, 43  
atrybuty klasy, 45  
grafiki OpenGL, 57  
klasa, 44  
odrysowanie zawartości, 44
- OpenGL, 66

- okna  
 podrzędne, 49  
 rejestracja klasy, 48  
 style, 45, 49  
 tworzenie, 48  
 tworzenie grafiki, 54  
 widoku, 135  
 wyskakujące, 49
- OpenGL, 19, 21, 25, 28, 29, 30, 37, 55, 56, 68, 176, 250, 398, 595  
 aplikacje pełnoekranowe, 67  
 ARB, 30  
 architektura, 31  
 funkcje rozszerzeń, 239  
 historia, 30  
 jądro, 31  
 kolory, 142  
 macierze, 119  
 maszyna stanów, 93  
 okno grafiki, 57  
 przekształcenia, 112  
 rozszerzenia specyfikacji, 31  
 specyfikacja, 30  
 stany, 93  
 Windows, 39  
 zasoby Internetowe, 600
- OpenGL 1.2, 31  
 OpenGL Architecture Review Board, 30  
 OpenGL Utility Library, 30, 31  
 OpenGL Utility Toolkit, 32  
 opengl32.lib, 28, 595  
 operacje graficzne niskiego poziomu, 30  
 opróżnianie buforów, 304  
 oświetlenie, 86, 147, 255, 481  
 dwustronne, 38  
 efekt odbicia, 166  
 globalne światło otoczenia, 165  
 materiały, 148, 163  
 modele, 164, 165  
 modelowanie, 148  
 normalne, 148  
 obiekty teksturowane, 166  
 OpenGL, 147, 153  
 realny świat, 147  
 ruchome źródła światła, 167  
 światło, 154  
 źródła światła, 153, 158  
 otoczenie, 250
- P**
- pakiet GLUT, 32  
 paleta kolorów, 144  
 Particle Systems API, 602
- pasek przewijania, 49  
 PCX, 482, 541  
 PeekMessage(), 50, 51, 55, 66, 70  
 pełnoekranowe aplikacje OpenGL, 67  
 perspektywa, 135  
 asymetryczna, 135  
 pęd, 511  
 pętla przetwarzania komunikatów, 50, 66, 573  
 PFD\_DEPTH\_DONTCARE, 303  
 PFD\_DOUBLEBUFFER, 129, 219, 303  
 PFD\_DOUBLEBUFFER\_DONTCARE, 303  
 PFD\_DRAW\_TO\_BITMAP, 303  
 PFD\_DRAW\_TO\_WINDOW, 129, 219, 303  
 PFD\_MAIN\_PLANE, 129, 219, 304  
 PFD\_OVERLAY\_PLANE, 304  
 PFD\_STEREO, 303  
 PFD\_STEREO\_DONTCARE, 303  
 PFD\_SUPPORT\_GDI, 59, 303  
 PFD\_SUPPORT\_OPENGL, 129, 219, 303  
 PFD\_TYPE\_COLORINDEX, 303  
 PFD\_TYPE\_RGBA, 129, 219, 303  
 PFD\_UNDERLAY\_PLANE, 304  
 picking, 38  
 pierwsza zasada dynamiki, 511  
 piksel, 56, 59, 181  
 format, 58, 186  
 konfiguracja formatu, 301  
 kopiowanie, 187  
 typy, 186
- PIXELFORMATDESCRIPTOR, 57, 58, 64, 70, 302, 304, 317, 326  
 cColorBits, 59  
 dwFlags, 59  
 iPixelType, 59  
 nSize, 58
- plakat, 357  
 plakatowanie, 355, 357  
 macierz modelowania, 356
- PlaneView(), 116
- PlaySegmentEx(), 446, 447, 448
- pliki, 46  
 AVI, 36  
 BMP, 181, 189, 482  
 graficzne, 181  
 MIDI, 425, 435  
 MP3, 468  
 MPG, 36  
 multimedialne, 34, 36  
 nagłówkowe, 28  
 PCX, 482  
 Targa, 193  
 TGA, 181, 482  
 WAV, 428

- Plug and Play, 34  
płaszczyzna, 526  
obcięcia, 112, 118  
odbicia, 377  
rzutowania, 83  
wektor normalny, 151  
płynność animacji, 66  
podsystem wejścia, 410, 418, 572  
podwójne buforowanie, 65, 303, 305  
PointInPolygon(), 538  
Poll(), 409  
położenie, 506  
kamery, 112  
pomniejszanie, 188  
PositionLights(), 376  
PostQuitMessage(), 44  
potok  
przekształceń wierzchołków, 112  
tworzenia grafiki, 32  
potokowe tworzenie grafiki, 37  
powielanie, 317  
powierzchnia, 337, 343  
    Béziera, 339, 343, 345  
    B-sklejana, 350, 352  
    cieniowanie, 345  
    NURBS, 32  
    oświetlenie, 345  
    parametryczna, 38  
    pokrywanie powierzchni teksturami, 346  
    punkty kontrolne, 353, 354  
    rysowanie, 353  
    siatka, 345  
    tworzenie, 343  
    wyznaczanie wierzchołków, 353  
powierzchnie drugiego stopnia, 327, 333  
cieniowanie, 328  
dyski, 329  
kule, 332  
obiekty, 328  
OpenGL, 327  
orientacja, 329  
stożki, 331  
style, 328  
tworzenie obiektów, 327  
usuwanie obiektów, 329  
walce, 331  
wektory normalne, 328  
współrzędne tekstury, 329  
powiększanie, 188  
powtarzanie tekstury, 210  
poziomy szczegółowości, 212  
pozycjonowanie źródeł dźwięku, 36  
prawa fizyki, 369  
prędkość, 508  
    chwilowa, 508  
    średnia, 508  
PrintString(), 288, 291  
priorytety procesów, 40  
proceduralne tworzenie modelu, 275  
procedury  
    obsługi zdarzeń, 42  
    okienkowe, 40, 42  
    przekształceń widoku, 116  
procesor  
    graficzny, 19  
    sygnałowy, 424  
procesy, 40  
program, 33  
programowanie  
    gier, 21, 599  
    obiektowe, 516  
    sterowanie za pomocą zdarzeń, 40  
    system Windows, 39, 41  
projektowanie obiektowe, 516  
prostopadłościany ograniczeń, 533  
proxy, 269  
przeciążanie operatorów, 526  
przedział czasu, 40  
przekształcenia, 32, 79, 112  
    bieżąca macierz, 114  
    modelowania, 112, 115, 117  
    obroty, 80, 117  
    okienkowe, 112, 118  
    potok, 112  
    przesunięcie, 80, 117  
    rzutowanie, 82, 112, 118  
    skalowanie, 82, 117  
    składania, 81  
    układu współrzędnych, 111  
    widoku, 112, 114, 116  
    wierzchołków, 119  
    współrzędne jednorodne, 79  
przełączanie buforów, 66  
przemieszczanie, 506  
przesłonięte obiekty, 326  
przestrzenny dźwięk, 450  
    bufor efektu, 451  
    odbiorca efektu, 455  
    odległość źródła, 453  
    parametry, 451  
    położenie, 454  
    położenie źródła, 450  
    prędkość, 454  
    stożki dźwięku, 454  
    tryb działania, 453  
    współrzędne, 450  
    względem obserwatora, 454  
    zwykły, 453

przesunięcie, 80, 117, 120  
 przetwarzanie  
     komunikatów, 42  
     tablic wierzchołków, 279  
 przezroczystość, 173, 174  
 przyspieszenie, 509  
 pulpit, 46  
 punkty, 37, 71, 72, 99  
     kontrolne, 354  
     rozmiary, 101

**Q**

Quake, 20, 26, 117  
 Quake 2, 470  
 Quake 3, 30, 561  
 QueryPerformanceCounter(), 517  
 QueryPerformanceFrequency(), 517

**R**

rada ARB, 237  
 raster, 184  
 RegisterClassEx(), 48  
 rekursja nieskończona, 271  
 Render(), 184  
 RGB, 59, 185  
 RGBA, 59, 142, 143, 144, 159, 206  
 RLE, 190  
 rozciąganie tekstur, 210  
 rozszerzenia OpenGL, 237  
     GL\_ARB\_multitexture, 244  
     GL\_EXT\_compiled\_vertex\_array, 281  
     sprawdzanie dostępności, 238  
 równania parametryczne, 337, 338  
 ruch, 111, 506  
     pęd, 511  
     prędkość, 508  
     przyspieszenie, 509  
     tarcie, 513  
         zasada zachowania pędu, 512  
         zderzenia, 513  
 ruchome źródła światła, 167  
 rysowanie  
     czworokątów, 108  
     krawędzie, 106  
     mapy bitowej, 183  
     obrazów graficznych, 185  
     odcinków, 102  
     powierzchni Béziera, 344  
     powierzchni B-sklejanej, 353  
     punktów, 101  
     trójkątów, 107  
     tryby, 305

tylnych stron, 155 \*  
 wielokątów, 104  
 rzeźba terenu, 243  
 rzut ortograficzny, 83  
 rzutowanie, 82, 118, 132, 136  
     cieni, 380, 382  
     izometryczne, 83  
     okno widoku, 135  
     ortograficzne, 118, 132, 133, 136  
     perspektywiczne, 66, 83, 84, 118, 132, 134, 136  
     płaszczyzna, 83  
     równolegle, 133

**S**

ScalarMatrixMult(), 78  
 scena, 117, 307, 313, 376  
 SDK, 397  
 SDK DirectX, 399  
 segment, 426  
 SetCooperativeLevel(), 407  
 SetDefaultAudioPath(), 446  
 SetPixelFormat(), 59, 64, 129, 302, 304  
 SetProperty(), 407  
 SetShadowMatrix(), 385  
 SetTextColor(), 55  
 SetupPixelFormat(), 64, 65, 301, 303  
 SetVolume(), 448  
 sfery ograniczeń obiektów, 532  
 SGI, 30  
 ShowCursor(), 68, 222, 559  
 ShowWindow(), 50, 222  
 siatka  
     równonodległa, 342  
     wierzchołków, 223  
 sieć, 36  
     Internet, 26  
     lokalna, 26  
 Silicon Graphics, 19, 29  
 siła, 510  
     druga zasada dynamiki, 511  
     grawitacji, 511  
     pierwsza zasada dynamiki, 511  
     tarcia, 514  
     trzecia zasada dynamiki, 511  
 siłowe sprężenie zwrotne, 36  
 SimpEngine, 561, 585  
     architektura szkieletu, 561  
     CAudioSystem, 580, 581  
     CCamera, 577  
     CEngine, 573  
     CInputSystem, 572  
     CNode, 562, 563  
     COBJECT, 566

- COGLWindow, 562, 570  
CSimpEngine, 576  
cykl gry, 574  
diagram podsystemów, 562  
kamera, 576, 577  
klasy, 562  
obsługa modeli, 580  
obsługa wejścia, 575  
system cząstek, 583  
system dźwięku, 581  
system wejścia, 572  
świat gry, 576, 580  
zarządzanie danymi, 562  
zarządzanie obiektami, 566  
 $\sin()$ , 215  
skala mapy, 224  
skalary, 71  
skalowanie, 82, 117, 122, 224  
    współczynniki, 122  
składania przekształceń, 81  
specyfikacja, 38  
spektrum kolorów, 143  
sprzętowe przetwarzanie grafiki, 26  
stacja graficzna, 19  
stała przesunięcia płaszczyzny, 526  
sterowanie, 391  
    animacją modelu, 498  
    za pomocą zdarzeń, 40  
sterowniki OpenGL, 29  
stos macierzy, 119, 123  
    modelowania, 123  
    rzutowania, 123  
    tekstur, 123, 253, 265  
stożki, 331  
    dźwięku, 454  
strumień światła, 160  
style okna, 45  
SwapBuffers(), 66, 157, 179, 184  
switch, 43, 54  
SYMBOL\_CHARSET, 288  
symulacja zderzeń, 513  
syntezator dźwięku, 424  
syntezator DSL, 427  
system  
    DOS, 34  
    dźwięku, 581  
    OS X, 20  
    Unix, 19, 33  
    UNIX, 55  
    wejścia, 418, 572  
    Windows, 28  
    Windows 95, 19  
    Windows NT, 19  
system cząstek, 359, 362, 583  
    aktualizacja, 365  
    atrybuty cząstek, 363  
    CParticleSystem, 366  
    czas życia cząstki, 360  
    cząstki, 359  
    częstość emisji, 363  
    inicjacja, 365  
    kolory, 361  
    lista cząstek, 362  
    łączenie kolorów, 364  
    masa cząstki, 361  
    menedżer, 365  
    oddziaływanie, 363, 365  
    położenie, 362  
    położenie cząstki, 360  
    prawa fizyki, 369  
    prędkość cząstki, 360  
    projektowanie, 365  
    przynależność cząstek, 362  
    reprezentacja, 364  
    reprezentacja cząstki, 361  
    rozmiar cząstki, 361  
    ruch, 365  
    rysowanie, 365  
    stan bieżący, 364  
    śnieżyca, 369  
    tworzenie efektów, 368  
    zmiana stanu, 365  
sześciian kolorów, 143  
szkielet aplikacji Windows, 52  
    OpenGL, 60  
szkielet gry, 561  
    Quake 3, 561  
    SimpEngine, 561  
    Unreal, 561  
szkło, 187  
sztuczna inteligencja, 25  
przeciwnik, 589

## Ś

- ścieżki dźwięku, 445, 468  
domyślne, 446  
konfiguracja, 445  
odtwarzanie dźwięku, 447  
pobieranie obiektów, 449  
rodzaje, 447  
    standardowe, 446  
śledzenie kolorów, 164  
śnieżyca, 369  
środek masy obiektu, 507  
świat gry, 586

światło, 86, 141  
długość fali, 142  
emisji, 148  
odbicia, 88, 148  
otoczenia, 87, 148  
rozproszone, 87, 148  
stopień odbicia, 147  
strumień, 160  
widzialne spektrum, 142

## T

tablice, 75  
tablice wierzchołków, 267, 274, 276  
blokowanie, 280  
GL\_COLOR\_ARRAY, 276  
GL\_EDGE\_FLAG\_ARRAY, 276  
GL\_INDEX\_ARRAY, 276  
GL\_NORMAL\_ARRAY, 276  
GL\_TEXTURE\_COORD\_ARRAY, 276  
GL\_VERTEX\_ARRAY, 276  
kolory, 276  
ograniczanie zakresu przetwarzanych elementów, 279  
OpenGL, 275  
przetwarzanie, 279  
tekstury wielokrotne, 280  
uporządkowanie elementów, 278  
wektory normalne, 277  
współrzędne tekstur, 277  
współrzędne wierzchołków, 277  
zarządzanie, 275  
znaczniki typów, 276  
tarcie, 513  
  płaszczyzna, 513  
  równia pochyla, 514  
Targa, 181  
  format plików, 193  
  ładowanie plików, 194  
  zapis obrazów, 196  
technologia  
  DirectX, 35  
  Plug and Play, 34  
teksel, 88, 199  
tekstury, 185, 199, 200, 501  
  brylowe, 199  
  dwuwymiarowe, 199, 205  
  filtrowanie, 204, 208, 212  
  format teksceli, 205  
  GL\_BLEND, 209  
  GL\_DECAL, 209  
  GL\_MODULATE, 209, 257  
  interpolacja współrzędnych, 210  
  jednostki, 237

  jednowymiarowe, 206  
  kafelkowanie, 210  
  listy wyświetlania, 272  
  łączenie kolorów, 209  
  macierze, 253  
  mapy, 205  
  mapy oświetlenia, 255  
  mipmapy, 212  
  obiekty, 207  
  odwzorowania, 88, 199  
  odwzorowanie otoczenia, 250  
  otoczenia, 250  
  pokrywanie powierzchni, 346  
  położenie, 204  
  powtarzanie, 210  
  poziomy szczegółowości, 212  
  proceduralne, 88  
  proxy, 269  
  przestrzenne, 38  
  rozciąganie, 210, 211  
  rozmiary, 204  
  teksel, 88, 199  
  trójwymiarowe, 200, 206  
tryb nakładania, 257  
tryb odwzorowania, 298  
tworzenie, 207  
typ danych, 205  
układ współrzędnych, 88  
wiązanie obiektu, 203  
współrzędne, 88, 209, 215  
tekstury wielokrotne, 237, 242, 255, 265  
  jednostki tekstury, 237, 240  
  sprawdzanie dostępności, 238  
tablice wierzchołków, 280  
tworzenie, 237, 240  
wieloprzebiegowe, 261  
współrzędne, 241  
teksty  
  bazowa lista wyświetlania, 286  
  czcionki, 285  
  czcionki konturowe, 289  
  czcionki pokryte tekstuą, 292  
  czcionki rastrowe, 287  
  efekty trójwymiarowe, 289  
  wyświetlanie, 285  
teoria grafiki trójwymiarowej, 71  
teren, 223  
testowanie głębi, 319  
torus, 250  
TranslateMessage(), 50, 51, 55  
transpozycja, 356  
trójkąty, 37, 99, 107  
  krawędzie, 107  
  wierńce, 108  
wierzchołki, 107, 108

trójwymiarowa grafika, 19  
trójwymiarowe bryły, 32  
trójwymiarowy świat, 86, 134  
tryb  
    okienkowy, 67  
    pełnoekranowy, 67  
    rysowania grafiki, 305  
    rysowania wielokątów, 105  
trzecia zasada dynamiki, 511  
tworzenie  
    aplikacji Windows, 42  
    cieni, 380  
    czcionek konturowych, 290  
    czcionek systemu Windows, 285  
    efektów dźwiękowych, 37  
    gier, 25  
    grafiki, 100  
    grafiki dwuwymiarowej, 134  
    grafiki OpenGL, 64  
    grafiki trójwymiarowej, 25, 106  
jednostek tekstyury, 240  
listy wyświetlania, 268  
materiałów, 163  
mgły, 374  
modelu, 275  
nieregularnych odbić, 378  
obiektów DirectInputDevice, 398  
obiektów graficznych, 111  
obiektów ładujących, 430, 431  
obiektów tekstu, 207  
obiektów wykonania, 429  
obrazu, 301  
odbić, 376  
okna, 44, 48  
palety kolorów, 144  
podsystemu wejścia, 410  
potokowe grafiki, 37  
powierzchni, 343  
programów systemu Windows, 39  
punktów, 100  
szkieletu gry, 561  
ścieżki dźwiękowej, 425  
tekstu, 289  
tekstur, 207  
tekstur wielokrotnych, 237, 240  
wektora normalnej, 153  
wierzchołków, 274  
wycinków dysku, 331  
wyliczeń urządzeń, 400  
źródeł światła, 158

**U**  
uchwyt  
    aplikacji, 398  
    czcionki, 286  
    kontekstu tworzenia grafiki, 56  
    kontekstu urządzenia, 64  
układ współrzędnych, 81, 100  
    biegunowych, 117  
    kamery, 113  
    obserwatora, 113  
    przekształcenia, 111  
    tekstyury, 88  
ukrywanie  
    krawędzi wielokątów, 106  
    powierzchni wielokątów, 105  
ukształtowanie terenu, 223  
    tablice wierzchołków, 281  
Unacquire(), 409  
Unix, 19, 55  
Unreal, 561  
Unreal Tournament, 26  
upakowanie danych mapy pikseli, 188  
UpdateWindow(), 50  
urządzenia  
    wejściowe, 36, 391, 397  
    wyjściowe, 67

**V**  
VGA, 34  
Visual C++, 28  
Visual Studio, 55  
vKey, 394

**W**  
walce, 331  
warstwa  
    abstrakcji sprzętu HAL, 35  
    emulacji sprzętu HEL, 35  
WAV, 428  
wątki, 40  
wektory, 71, 72, 521  
    długość, 72  
    dodawanie, 73  
    iloczyn skalarny, 73  
    jednostkowe, 121, 152  
    mnożenie przez skalar, 73  
    norma, 72  
    normalizacja, 72  
    normalne, 481  
    ortogonalne, 356

WGL, 55, 70  
wglCreateContext(), 56, 57, 65  
wglDeleteContext(), 56  
wglGetProcAddress(), 239  
wglMakeCurrent(), 57, 65  
wglSwapLayerBuffers(), 59  
wglUseBitmapFonts(), 288  
wglUseFontBitmaps(), 285, 286  
wglUseFontOutlines(), 289  
wglUseOutlineFonts(), 299  
WHEEL\_DATA, 393  
widok, 112  
    modelu, 112  
wielokąty, 99, 104, 109  
    antialiasing, 106  
    strony, 104  
    tryby rysowania, 105  
ukrywanie krawędzi, 106  
ukrywanie powierzchni, 105  
wierzchołki, 104  
    wypełnienia, 107  
    wzór wypełnienia, 107  
wieloprzebiegowe tekstury wielokrotne, 261  
wielowątkowość, 40  
wielozadaniowość, 40, 407  
wieńce trójkątów, 108  
wierzchołki, 104, 122, 274  
    ograniczanie wielokrotnego przetwarzania, 108  
    wielokąta, 105  
Win32, 392, 394  
    obsługa klawiatury, 394  
    obsługa manipulatorów, 396  
WINAPI, 42  
Windows, 28, 39  
    aplikacje, 40, 41  
    aplikacje OpenGL, 60  
    funkcje WGL, 55  
ikony, 46  
kolejka zdarzeń, 40  
komunikaty, 42  
kontekst urządzenia, 54  
kursor myszy, 46  
obsługa komunikatów, 43  
pętla przetwarzania komunikatów, 50  
procedury okienkowe, 40, 42  
programowanie, 41  
tworzenia grafiki w oknie, 54  
wątki, 40  
wielowątkowość, 40  
WinMain(), 42  
    wykonywanie procesów, 40  
Windows 3.1, 34  
Windows 95, 19  
Windows NT, 19  
windowsx.h, 54  
WinMain(), 42, 45, 50, 55, 69, 130, 171  
winmm.lib, 595  
wirtualny  
    czas, 506  
    świat, 505  
włókna, 40  
WM\_CHAR, 392  
WM\_CLOSE, 44  
WM\_CREATE, 42, 44, 57, 64  
WM\_DESTROY, 57  
WM\_KEYDOWN, 170, 172, 392, 393  
WM\_KEYUP, 170, 392, 393  
WM\_LBUTTONDOWN, 392  
WM\_LBUTTONUP, 392  
WM\_MBUTTONDOWN, 392  
WM\_MBUTTONUP, 392  
WM\_MOUSEMOVE, 227, 393  
WM\_MOUSEWHEEL, 393  
WM\_MOVE, 42  
WM\_PAINT, 44, 50, 54  
WM\_QUIT, 44, 55  
WM\_RBUTTONDOWN, 392  
WM\_RBUTTONUP, 393  
WM\_SIZE, 42, 65, 570  
WM\_TIMER, 517  
WNDCLASS, 44, 48  
WNDCLASSEX, 44, 45, 48, 69  
WndProc(), 54, 64, 227  
Wolfenstein, 26  
WriteBitmapFile(), 192  
WriteTGAFile(), 196  
WS\_BORDER, 49  
WS\_CAPTION, 49  
WS\_CHILD, 49  
WS\_EX\_APPWINDOWS, 68  
WS\_HSCROLL, 49  
WS\_ICONIC, 49  
WS\_MAXIMIZE, 49  
WS\_MAXIMIZEBOX, 49  
WS\_MINIMIZE, 49  
WS\_MINIMIZEBOX, 49  
WS\_OVERLAPPED, 49  
WS\_OVERLAPPEDWINDOW, 49  
WS\_POPUP, 49, 68  
WS\_POPUPWINDOW, 49  
WS\_SIZEBOX, 49  
WS\_SYSMENU, 49  
WS\_VISIBLE, 49, 50  
WS\_VSCROLL, 49  
współczynnik alfa, 142, 173  
współrzędne, 82  
    jednorodne, 79, 85  
    kamery, 113  
obserwatora, 113

parametryczne, 89  
przestrzeni dźwięku, 450  
tekstury, 209, 241  
wygładzanie krawędzi, 101  
wykrywanie zderzeń, 526, 533  
wyświetlanie tekstów, 285  
wzory  
linii, 103  
wypełnienia wielokątów, 107

**Z**

zapis obrazu, 191  
zarządcy procesów, 40  
zarządzanie danymi, 562  
zasada  
bezwładności, 511  
zachowania energii, 513  
zachowania pędu, 512  
zderzenia, 513, 544, 566  
kąt odbicia, 538  
kąt padania, 538  
obiektów, 531  
obiektu z płaszczyzną, 536

obsługa, 538  
płaszczyzn, 535  
prostopadłościany ograniczeń, 533  
sfery ograniczeń, 532  
wykrywanie, 531  
zmienne  
globalne, 64, 200  
GLYPHMETRICSFLOAT, 289  
znaki, 288

**Ž**

źródła dźwięku, 453  
źródła światła, 153, 180  
intensywność odbicia, 158  
intensywność otoczenia, 158  
kierunkowe, 159  
koncentracja strumienia, 158  
położenie, 159  
pozycyjne, 159  
ruchome, 167  
strumień, 160  
tłumienie, 158, 160





# **CD-ROM dołączony do książki zawiera między innym:**

- ◆ Pakiety SDK (DirectX 8.0, OpenGL)
- ◆ Przykłady wykorzystane w książce
- ◆ Dodatkowe biblioteki
- ◆ Programy demonstracyjne
- ◆ Gry