

OpenGL®

Richard S. Wright, Jr., Nicholas Haemel, Graham Sellers, Benjamin Lipchak

**Dowiedz się, jak tworzyć zapierające dech w piersiach gry 3D  
i efektowne, trójwymiarowe wizualizacje!**

Jak sprawnie tworzyć podstawowe obiekty, oświetlać je i cieniować?

Jak napisać własne programy, korzystając  
z biblioteki OpenGL i języka GLSL?

Jak programować grafike na urządzenia przenośne,  
takie jak iPhone, iPod czy iPad?

Open  
GL

**OpenGL®**  
**KSIĘGA EKSPERTA**

Wydanie 5

**Helion** 

## » Idź do

- Spis treści
- Przykładowy rozdział
- Skorowidz

## » Katalog książek

- Katalog online
- Zamów drukowany katalog

## » Twój koszyk

- Dodaj do koszyka

## » Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

## » Czytelnia

- Fragmenty książek online

## » Kontakt

Helion SA  
ul. Kościuszki 1c  
44-100 Gliwice  
tel. 32 230 98 63  
e-mail: helion@helion.pl  
© Helion 1991–2011

## OpenGL. Księga eksperta. Wydanie V

Autorzy: [Richard S. Wright](#), [Nicholas Haemel](#),  
[Graham Sellers](#), [Benjamin Lipchak](#)

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-2976-3

Tytuł oryginału: [OpenGL SuperBible: Comprehensive  
Tutorial and Reference \(5th Edition\)](#)

Format: 172×245, stron: 688



### Dowiedz się, jak tworzyć zapierające dech w piersiach gry 3D i efektywne, trójwymiarowe wizualizacje!

- Jak sprawnie tworzyć podstawowe obiekty, oświetlać je i cieniować?
- Jak napisać własne programy, korzystając z biblioteki OpenGL i języka GLSL?
- Jak programować grafikę na urządzenia przenośne, takie jak iPhone, iPod czy iPad?

Po prawie dwudziestu latach na rynku biblioteka OpenGL jest dziś wiodącym API w dziedzinie programowania grafiki trójwymiarowej, gier 3D, wizualizacji, symulacji, modelowania naukowego, a nawet edytowania obrazów i filmów dwuwymiarowych. Swój sukces zawdzięcza nie tylko łatwości użycia, ale przede wszystkim kompatybilności z niemal wszystkimi platformami dostępnymi na rynku. Świetnie sprawdza się zarówno w komputerach PC z systemem Windows, jak i komputerach Mac, a także na stacjach uniksowych, w centrach rozrywki opartych o lokalizację, na najbardziej znanych konsolach do gier, w kieszonkowych grach elektronicznych, a nawet w oprzyrządowaniu lotniczym czy samochodowym. Nie bez znaczenia dla spopularyzowania tej biblioteki był także fakt, że można ją rozszerzać, dzięki czemu ma ona wszystkie zalety otwartego standardu, a dodatkowo można wzbogacać jej implementację o własne dodatki.

„OpenGL. Księga eksperta. Wydanie 5” to nowe, zaktualizowane (specyfikacja OpenGL 3.3) wydanie znanego podręcznika dla wszystkich programistów, bez względu na poziom ich zaawansowania. Książka ta stanowi wyczerpujący kurs tworzenia niesamowitych wizualizacji 3D, gier oraz wszelkiego rodzaju grafik. Dzięki niej nauczysz się pisać programy wykorzystujące bibliotekę OpenGL, konfigurować środowisko pracy do przetwarzania grafiki trójwymiarowej oraz tworzyć podstawowe obiekty, oświetlać je i cieniować. Następnie zgłębisz tajniki języka OpenGL Shading Language i zaczniesz sprawnie pisać własne programy, wprowadzać do nich rozmaite efekty wizualne oraz zwiększać ich wydajność. Poznasz wszystkie najnowsze techniki programowania przy użyciu biblioteki OpenGL, takie jak przekształcenia, nakładanie tekstur, cieniowanie, zaawansowane bufory czy zarządzanie geometrią. Przejdziesz także szczegółowy kurs programowania grafiki w urządzeniach iPhone, iPod touch oraz iPad!

**Kompletny przewodnik po najpopularniejszej na świecie bibliotece do  
programowania grafiki trójwymiarowej OpenGL 3.3!**

# Spis treści

<b>Podziękowania .....</b>	<b>17</b>
<b>O autorach .....</b>	<b>21</b>
<b>Wstęp do wydania piątego .....</b>	<b>23</b>
<b>Wstęp do wydania czwartego .....</b>	<b>25</b>
<b>Wstęp do wydania trzeciego .....</b>	<b>29</b>
<b>Wprowadzenie .....</b>	<b>31</b>
Co nowego w tym wydaniu .....	31
Struktura książki .....	32
Część I: Podstawy .....	32
Część II: Techniki średnio zaawansowane i zaawansowane .....	34
Część III: OpenGL na różnych platformach .....	34
Konwencje typograficzne .....	35
Witryna internetowa .....	35
<b>Część I   Podstawy .....</b>	<b>37</b>
<b>Rozdział 1. Wprowadzenie do grafiki trójwymiarowej i biblioteki OpenGL .....</b>	<b>39</b>
Historia grafiki komputerowej w skrócie .....	40
Elektryczność .....	40
Wejście w trzeci wymiar .....	41
Podstawowe efekty trójwymiarowe i najważniejsze pojęcia .....	44
Przekształcanie i rzutowanie .....	44
Rasteryzacja .....	45
Cieniowanie .....	45
Teksturowanie .....	46
Mieszanie kolorów .....	47
Łączenie punktów .....	47
Typowe zastosowania grafiki trójwymiarowej .....	47
Trzy wymiary w czasie rzeczywistym .....	47
Trzy wymiary bez czasu rzeczywistego .....	48
Programy do cieniowania .....	50
Podstawowe zasady programowania grafiki trójwymiarowej .....	52
To nie jest zestaw narzędzi .....	52
Układy współrzędnych .....	52
Rzutowanie z trzech w dwa wymiary .....	56
Podsumowanie .....	58

<b>Rozdział 2. Rozpoczynanie pracy .....</b>	<b>61</b>
Czym jest OpenGL .....	62
Ewolucja standardu .....	63
Mechanizm rozszerzeń .....	64
Czyje to rozszerzenie .....	66
Korzystanie z biblioteki OpenGL .....	71
Narzędzia pomocnicze .....	72
Biblioteka GLUT .....	72
Biblioteka GLEW .....	73
Biblioteka GLTools .....	73
Szczegóły interfejsu .....	74
Typy danych .....	74
Błędy OpenGL .....	76
Sprawdzanie wersji .....	77
Pobieranie wskazówek z funkcji glHint .....	77
Maszyna stanów OpenGL .....	78
Konfigurowanie środowiska programistycznego w systemie Windows .....	79
Dodawanie ścieżek .....	79
Tworzenie projektu .....	81
Dodawanie własnych plików .....	82
Konfigurowanie środowiska programistycznego w systemie Mac OS X .....	84
Niestandardowe ustawienia kompilacji .....	85
Tworzenie nowego projektu .....	85
Szkielety, nagłówki i biblioteki .....	87
Nasz pierwszy trójkąt .....	90
Układ współrzędnych .....	95
Konfigurowanie ustawień .....	98
Zabieramy się do pracy .....	100
Ożywianie sceny .....	101
Klawisze specjalne .....	101
Odświeżanie ekranu .....	103
Prosta animacja .....	103
Podsumowanie .....	104
<b>Rozdział 3. Podstawy renderowania .....</b>	<b>105</b>
Rysowanie punktów w trzech wymiarach .....	106
Podstawowy potok graficzny .....	106
Klient-serwer .....	107
Programy do cieniowania .....	108
Konfigurowanie układu współrzędnych .....	110
Rzutowanie ortogonalne .....	111
Rzutowanie perspektywiczne .....	112
Standardowe programy do cieniowania .....	112
Atrybuty .....	113
Zmienne typu uniform .....	113
Łączenie punktów .....	115
Punkty i linie .....	116

Rysowanie trójkątów w trzech wymiarach .....	120
Nawinięcie .....	121
Prosty kontener porcji danych .....	124
Niechciana geometria .....	126
Przesuwanie wielokątów .....	131
Wycinanie nożycami .....	134
Mieszanie kolorów .....	135
Łączenie kolorów .....	136
Zmiana równania mieszania .....	139
Wyglądanie .....	140
Podsumowanie .....	145

#### **Rozdział 4. Podstawy przekształceń geometrycznych.**

<b>Najważniejsze informacje o wektorach i macierzach .....</b>	<b>147</b>
Czy to jest ten straszny rozdział z matematyką? .....	148
Szybki kurs matematyki .....	149
Wektory, czyli w którym kierunku .....	149
Macierz .....	152
Przekształcenia .....	154
Współrzędne oka .....	154
Przekształcenia punktu widzenia .....	155
Przekształcenia modelowania .....	156
Dwoistość model-widok .....	157
Przekształcenia rzutowania .....	158
Przekształcenia widoku .....	159
Macierz model-widok .....	160
Konstruowanie macierzy .....	160
Łączenie przekształceń .....	164
Stosowanie macierzy model-widok .....	165
Więcej obiektów .....	167
Klasa zestawów trójkątów .....	167
Macierz rzutowania .....	171
Rzutowanie prostopadłe .....	171
Rzutowanie perspektywiczne .....	172
Macierz rzutowania model-widok .....	174
Potok przekształceń .....	178
Stos macierzy .....	179
Modyfikowanie potoku .....	180
Wersja wzbogacona .....	184
Poruszanie się przy użyciu kamer i aktorów .....	185
Układ odniesienia aktora .....	186
Kąty Eulera. „Użyj układu odniesienia, Luke!” .....	187
Obsługa kamery .....	188
Dodawanie aktorów .....	191
Oświetlenie .....	192
Podsumowanie .....	193

<b>Rozdział 5. Tekstury — podstawy .....</b>	<b>195</b>
Surowe dane obrazów .....	196
Pakowanie pikseli .....	196
Piksmapy .....	199
Upakowane formaty pikseli .....	200
Zapisywanie pikseli .....	203
Wczytywanie pikseli .....	205
Ładowanie tekstur .....	208
Wykorzystywanie bufora kolorów .....	209
Aktualizowanie tekstur .....	210
Obiekty tekstur .....	211
Nakładanie tekstur .....	212
Współrzędne tekstury .....	212
Parametry tekstur .....	214
Praktyczne zastosowanie poznanych wiadomości .....	217
Mipmapy .....	222
Filtrowanie mipmap .....	223
Generowanie poziomów mipmap .....	225
Zastosowanie mipmap .....	225
Filtrowanie anizotropowe .....	233
Kompresja tekstur .....	235
Kompresowanie tekstur .....	236
Ładowanie tekstur skompresowanych .....	238
Ostatni przykład .....	239
Podsumowanie .....	240
<b>Rozdział 6. Myślenie niekonwencjonalne — programy do cieniowania .....</b>	<b>241</b>
GLSL 101 .....	242
Zmienne i typy danych .....	243
Kwalifikatory zmiennych .....	246
Prawdziwy shader .....	248
Kompilowanie, wiązanie i konsolidowanie .....	252
Praktyczne wykorzystanie shadera .....	258
Wierzchołek prowokujący .....	259
Dane uniform shadera .....	259
Znajdowanie danych uniform .....	260
Zmienne uniform skalarne i wektorowe .....	260
Tablice uniform .....	261
Macierze uniform .....	262
Płaski shader .....	262
Funkcje standardowe .....	265
Funkcje trygonometryczne .....	265
Funkcje wykładnicze .....	266
Funkcje geometryczne .....	267
Funkcje macierzowe .....	267
Funkcje porównywania wektorów .....	267
Inne często używane funkcje .....	267

Symulowanie światła .....	272
Światło rozproszone .....	272
Shader światła rozproszonego .....	274
Model oświetlenia ADS .....	278
Cieniowanie Phong'a .....	281
Korzystanie z tekstur .....	285
Nic, tylko teksele .....	285
Oświetlanie teksele .....	287
Anulowanie przetwarzania fragmentów .....	289
Teksturowanie w stylu kreskówkowym — teksele w roli światła .....	292
Podsumowanie .....	294
<b>Rozdział 7. Tekstury — techniki zaawansowane .....</b>	<b>295</b>
Tekstury prostokątne .....	296
Wczytywanie tekstury prostokątnej .....	297
Zastosowanie tekstur prostokątnych .....	297
Tekstury sześciennie .....	300
Wczytywanie tekstur sześciennych .....	301
Tworzenie pudła nieba .....	302
Tworzenie efektu odbicia .....	304
Multiteksturowanie .....	305
Wiele współrzędnych tekstur .....	306
Przykład multiteksturowania .....	306
Teksturowanie punktów .....	309
Teksturowanie punktów .....	309
Rozmiar punktów .....	310
Podsumowanie wiadomości .....	311
Parametry punktów .....	314
Nadawanie punktom kształtów .....	314
Obracanie punktów .....	315
Tablice tekstur .....	317
Ładowanie tablicy tekstur dwuwymiarowych .....	317
Indeksowanie tablicy tekstur .....	319
Uzyskiwanie dostępu do tablic tekstur .....	320
Tekstury zastępcze .....	320
Podsumowanie .....	322
<b>Część II Techniki średnio zaawansowane i zaawansowane .....</b>	<b>323</b>
<b>Rozdział 8. Buforowanie — od tej pory przechowywanie danych zależy od Ciebie .....</b>	<b>325</b>
Bufory .....	326
Tworzenie własnych buforów .....	327
Napełnianie buforów .....	328
Obiekty bufora pikseli .....	329
Tekstury buforowe .....	336

Obiekt bufora obrazu, czyli opuszczamy okno .....	338
Sposób użycia FBO .....	339
Obiekt bufora renderowania .....	339
Bufory rysowania .....	341
Kompletność bufora obrazu .....	344
Kopiowanie danych w buforach obrazu .....	347
Praktyczny przykład wykorzystania buforów FBO .....	348
Renderowanie do tekstur .....	353
Podsumowanie .....	358

**Rozdział 9. Buforowanie — techniki zaawansowane ..... 359**

Uzyskiwanie dostępu do danych .....	360
Mapowanie buforów .....	360
Kopiowanie buforów .....	361
Wysyłanie danych z shadera pikseli i odwzorowywanie fragmentów wyjściowych .....	362
Nowe formaty dla nowej generacji urządzeń .....	364
Precyzyjne formaty zmiennooprzecinkowe .....	365
Wielopróbkowanie .....	379
Liczby całkowite .....	383
sRGB .....	384
Kompresja tekstur .....	386
Podsumowanie .....	388

**Rozdział 10. Operacje na fragmentach — koniec potoku ..... 389**

Okrawanie — przycinanie geometrii na wymiar .....	390
Wielopróbkowanie .....	391
Powierzchnia pokrycia próbki .....	391
Maska próbki .....	392
Podsumowanie dotychczasowych wiadomości .....	393
Operacje na szablonach .....	397
Testowanie głębi .....	400
Ograniczanie wartości głębi .....	400
Mieszanie kolorów .....	400
Równanie mieszania .....	401
Funkcja mieszania .....	401
Zebranie wiadomości .....	403
Rozsiewanie kolorów .....	404
Operacje logiczne .....	405
Maskowanie wyniku .....	405
Maskowanie koloru .....	406
Maskowanie głębi .....	406
Maskowanie buforów szablonu .....	407
Zastosowanie masek .....	407
Podsumowanie .....	407



<b>Rozdział 11. Programy cieniujące — techniki zaawansowane .....</b>	<b>409</b>
Zaawansowane shadery wierzchołków .....	410
Fizyczne symulacje w shaderze wierzchołków .....	410
Shadery geometrii .....	417
Shader geometrii przepuszczający dane .....	417
Zastosowanie shadera geometrii w programie .....	419
Usuwanie geometrii w shaderach geometrii .....	423
Modyfikowanie geometrii w shaderze geometrii .....	426
Generowanie geometrii w shaderze geometrii .....	427
Zmianianie typu obiektu podstawowego w shaderze geometrii .....	431
Nowe typy obiektów podstawowych w shaderach geometrii .....	433
Zaawansowane shadery fragmentów .....	436
Przetwarzanie końcowe w shaderze fragmentów — korekcja kolorów .....	438
Przetwarzanie końcowe — splot .....	439
Generowanie danych obrazu w shaderze fragmentów .....	442
Ignorowanie zadań w shaderze fragmentów .....	445
Kontrolowanie głębi poszczególnych fragmentów .....	447
Inne zaawansowane funkcje shaderów .....	448
Interpolacja i kwalifikatory pamięci .....	448
Inne zaawansowane funkcje wbudowane .....	452
Obiekty bufora bloku zmiennych jednorodnych .....	454
Tworzenie bloków zmiennych jednorodnych .....	455
Podsumowanie .....	464
<b>Rozdział 12. Zarządzanie geometrią — techniki zaawansowane .....</b>	<b>465</b>
Zbieranie informacji o potoku OpenGL — zapytania .....	466
Przygotowywanie zapytania .....	467
Wysyłanie zapytania .....	468
Pobieranie wyników zapytania .....	468
Wykorzystanie wyniku zapytania .....	469
Zmuszanie OpenGL do podejmowania decyzji .....	472
Mierzenie czasu wykonywania poleceń .....	475
Przechowywanie danych w pamięci GPU .....	477
Przechowywanie danych w buforach danych wierzchołków .....	478
Przechowywanie w buforach indeksów wierzchołków .....	482
Organizowanie buforów przy użyciu obiektów tablic wierzchołków .....	483
Optymalne rysowanie dużych ilości geometrii .....	486
Łączenie funkcji rysujących .....	486
Łączenie geometrii poprzez restart obiektów podstawowych .....	487
Rysowanie wielu egzemplarzy jednego obiektu .....	489
Automatyczne pobieranie danych .....	495
Przechowywanie przekształconych wierzchołków — przekształcenie zwrotne .....	500
Przekształcenie zwrotne .....	501
Wyłączanie rasteryzacji .....	506
Liczenie wierzchołków przy użyciu zapytań obiektów podstawowych .....	507
Wykorzystanie wyników zapytania obiektów podstawowych .....	508
Przykład zastosowania przekształcenia zwrotnego .....	509

Przycinanie i rysowanie tego, co się chce .....	518
Definiowanie własnych płaszczyzn obcinania .....	519
Synchronizacja rysowania .....	523
Podsumowanie .....	527
<b>Część III OpenGL na różnych platformach .....</b>	<b>529</b>
<b>Rozdział 13. OpenGL w systemie Windows .....</b>	<b>531</b>
Implementacje OpenGL w systemie Windows .....	532
OpenGL firmy Microsoft .....	533
Nowoczesne sterowniki graficzne .....	533
Rozszerzenia OpenGL .....	534
Rozszerzenia WGL .....	536
Podstawy renderowania w systemie Windows .....	537
Konteksty urządzenia GDI .....	538
Formaty pikseli .....	539
Kontekst renderingu OpenGL .....	547
Konsolidacja wiadomości .....	550
Tworzenie okna .....	550
Rendering pełnoekranowy .....	555
Podwójne buforowanie .....	556
Zapobieganie poszarpaniu obrazu .....	557
Podsumowanie .....	557
<b>Rozdział 14. OpenGL w systemie Mac OS X .....</b>	<b>559</b>
Cztery twarze OpenGL w systemie Mac .....	560
Biblioteka OpenGL i interfejs Cocoa .....	561
Tworzenie programu Cocoa .....	561
Składanie wszystkiego razem .....	566
Buforowanie pojedyncze czy podwójne .....	568
Program SphereWorld .....	569
Renderowanie pełnoekranowe .....	573
Renderowanie pełnoekranowe w Cocoa .....	574
CGL .....	581
Synchronizacja szybkości klatek .....	581
Przyspieszanie operacji wypełniania .....	582
Wielowątkowość w OpenGL .....	583
Podsumowanie .....	583
<b>Rozdział 15. OpenGL w Linuksie .....</b>	<b>585</b>
Wiadomości podstawowe .....	586
Rys historyczny .....	586
Co to jest X .....	586
Zaczynamy .....	587
Sprawdzanie obsługi OpenGL .....	587
Konfiguracja Mesy .....	588
Konfiguracja sterowników sprzętowych .....	589

Konfiguracja bibliotek GLUT i GLEW .....	589
Budowa aplikacji OpenGL .....	590
GLX — łączenie z X Windows .....	591
Ekran i X Windows .....	592
Zarządzanie konfiguracjami i obiektami widoku .....	592
Okna i powierzchnie renderingu .....	595
Rozszerzanie OpenGL i GLX .....	597
Zarządzanie kontekstem .....	597
Synchronizacja .....	601
Zapytania GLX .....	602
Składanie aplikacji .....	602
Podsumowanie .....	605
<b>Rozdział 16. OpenGL ES w urządzeniach przenośnych .....</b>	<b>607</b>
OpenGL na diecie .....	608
Do czego służy ES .....	608
Rys historyczny .....	609
Wybór wersji .....	611
ES 2.0 .....	611
Środowisko układów wbudowanych .....	615
Kwestie projektowe .....	616
Rozwiązywanie problemów z ograniczeniami .....	616
Działania na liczbach stałoprzecinkowych .....	617
EGL — nowe środowisko okienkowe .....	619
Ekran EGL .....	619
Tworzenie okna .....	621
Zarządzanie kontekstem .....	625
Prezentowanie buforów i synchronizacja renderowania .....	626
Jeszcze trochę o EGL .....	627
Środowiska układów wbudowanych .....	628
Popularne systemy operacyjne .....	628
Rozszerzenia producentów .....	628
Dla domowego rzemieślnika .....	628
Platformy przenośne firmy Apple .....	629
Tworzenie projektu aplikacji dla iPhone'a .....	629
Przeziadka na iPhone'a .....	633
Podsumowanie .....	640
<b>Dodatki .....</b>	<b>641</b>
<b>Dodatek A Dalsza lektura .....</b>	<b>643</b>
<b>Dodatek B Słowniczek .....</b>	<b>647</b>
<b>Skorowidz .....</b>	<b>653</b>

# **Rozdział 6. Myślenie niekonwencjonalne — programy do cieniowania**

**Autor: Richard S. Wright, Jr**



Po raz pierwszy ze shaderami spotkaliśmy się w rozdziale 3., w którym opisane zostały podstawowe techniki renderowania. Osoby, które ten rozdział pominęły, aby od razu przejść do pisania shaderów, powinny w razie potrzeby do niego wrócić i dokładnie zapoznać się z atrybutami i danymi typu `uniform` oraz sposobami ich przekazywania do shadera z kodu klienckiego. W tamtym rozdziale w kręgu naszych zainteresowań była wyłącznie strona kliencka. Użyliśmy też niektórych standardowych shaderów oraz kilku typowych funkcji i procedur renderujących. Na początku tego rozdziału można będzie pogłębić swą wiedzę na temat pracy po stronie klienckiej, a następnie pokażemy, jak pisać własne shadery, czyli serwerową część procesu renderowania. W tym celu należy zapoznać się z językiem do pisania programów do cieniowania, czyli shaderów.

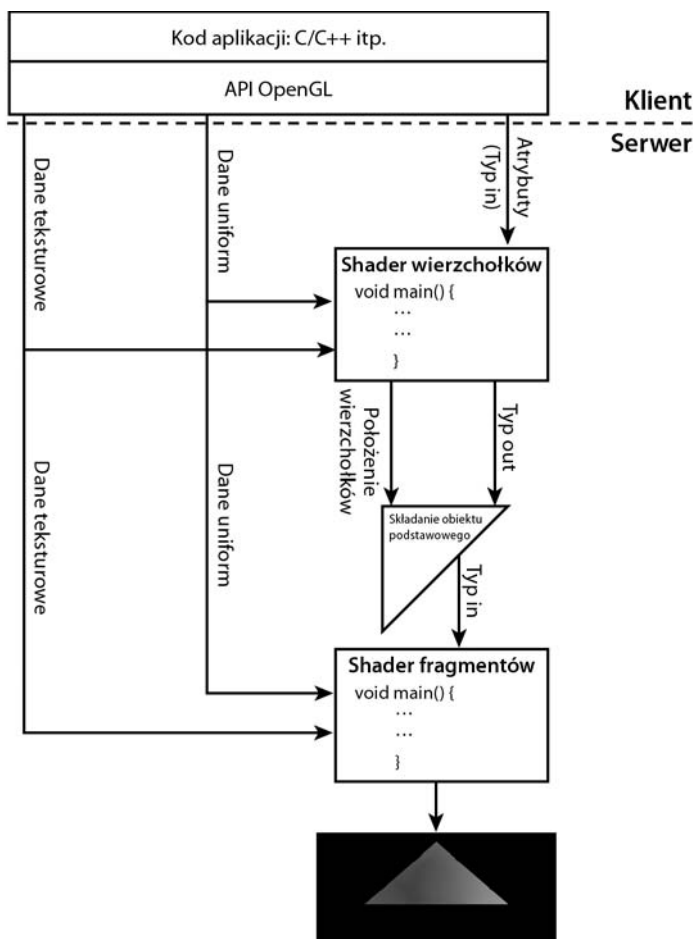
## GLSL 101

OpenGL Shading Language (GLSL) to wysokopoziomowy język programowania podobny do języka C. Napisane w nim programy podlegają kompilacji i konsolidacji przez implementację OpenGL i najczęściej w całości działają na sprzęcie graficznym. Kod shaderów wygląda podobnie do programów w języku C. Ich wykonywanie zaczyna się od funkcji `main` oraz można w nich pisać funkcje pobierające argumenty i zwracające wartości. Poniższy rysunek 6.1 to kopia rysunku 3.1 z rozdziału 3., przedstawiająca podstawowe cechy architektury shadera.

Jak wynika z powyższego schematu, zawsze potrzebne będą nam przynajmniej dwa shadery — wierzchołków i fragmentów. Trzeci rodzaj shadera, który również może być potrzebny, to tzw. shader geometrii, ale jego szczegółowy opis odłożymy do rozdziału 11. „Programy cieniujące — techniki zaawansowane”. Dane do shadera wierzchołków można przesyłać na trzy sposoby — jako atrybuty, czyli elementy danych dotyczące poszczególnych wierzchołków; dane rodzaju `uniform`, które są takie same dla całej porcji danych wierzchołkowych; oraz jako tekstury, o których szerzej pisaliśmy w rozdziale 5. Dane `uniform` i teksturowe można także przesyłać do shadera fragmentów. Nie ma natomiast sensu wysyłać do tego shadera atrybutów wierzchołków, ponieważ służy on tylko do wypełniania fragmentów (pikseli) po rasteryzacji danego obiektu podstawowego. Dane wierzchołkowe mogą natomiast być przesyłane do shadera fragmentów przez program wierzchołkowy. Wówczas jednak dane takie mogą być stałe (każdy fragment dysponuje taką samą wartością) lub wartości mogą być interpolowane na różne sposoby na powierzchni obiektu.

Kod shaderów jest bardzo podobny do kodu programów w języku C. Wykonywanie zaczyna się od funkcji `main`, obowiązują takie same konwencje stosowania komentarzy i taki sam zestaw znaków oraz używa się wielu takich samych dyrektyw preprocesora. Pełną specyfikację języka można znaleźć w dokumencie OpenGL Shading Language Specification. Wskazówki, jak znaleźć tę specyfikację, zamieszczono w dodatku A. Są tam wymienione przydatne adresy internetowe oraz inne wartościowe źródła i kursy uzupełniające. Przyjmujemy założenie, że znasz już języki C i C++, i dlatego będziemy opisywać język GLSL z perspektywy programisty tych języków.

Rysunek 6.1.  
Schemat architektury  
shadera



## Zmienne i typy danych

Naukę języka GLSL dobrze jest rozpocząć od zapoznania się z dostępnymi w nim typami danych. Są cztery typy: liczby całkowite (ze znakiem i bez znaku), liczby zmiennoprzecinkowe (od OpenGL 3.3 tylko pojedynczej precyzji) oraz wartości logiczne (bool). W GLSL nie ma wskaźników ani typów łańcuchowych i znakowych. Funkcje mogą zwracać wartości dowolnego z dostępnych typów lub nie zwracać nic, jeśli zostaną zadeklarowane jako void, ale niedozwolone jest stosowanie wskaźników void. Sposób użycia typów danych GLSL jest podobny do sposobu ich użycia w językach C i C++.

```
bool bDone = false;           // Wartość logiczna: true lub false
int iValue = 42;              // Liczba całkowita ze znakiem
uint uiValue = 3929u;        // Liczba całkowita bez znaku
float fValue = 42.0f;        // Wartość zmiennoprzecinkowa
```

## Typy wektorowe

Jedną z najbardziej interesujących rzeczy dostępnych w GLSL, a której nie ma w C i C++, są wektorowe typy danych. Wartości każdego z czterech podstawowych typów danych można przechowywać w dwu-, trzy- lub czterowymiarowych wektorach. Pełną listę wektorowych typów danych przedstawia tabela 6.1.

Tabela 6.1. Wektorowe typy danych języka GLSL

Typ	Opis
<code>vec2</code> , <code>vec3</code> , <code>vec4</code>	Dwu-, trzy- i czterokładnikowy wektor wartości zmiennoprzecinkowych
<code>ivec2</code> , <code>ivec3</code> , <code>ivec4</code>	Dwu-, trzy- i czterokładnikowy wektor wartości całkowitoliczbowych
<code>uvec2</code> , <code>uvec3</code> , <code>uvec4</code>	Dwu-, trzy- i czterokładnikowy wektor wartości całkowitoliczbowych bez znaku
<code>bvec2</code> , <code>bvec3</code> , <code>bvec4</code>	Dwu-, trzy- i czterokładnikowy wektor wartości logicznych

Zmienne typu wektorowego deklaruje się tak samo, jak wszystkie inne rodzaje zmiennych. Poniższa instrukcja deklaruje położenie wierzchołka w postaci czterokładnikowego wektora wartości zmiennoprzecinkowych:

```
vec4 vVertexPos;
```

Wektor można także zainicjalizować przy użyciu konstruktora:

```
vec4 vVertexPos = vec4(39.0f, 10.0f, 0.0f, 1.0f);
```

Nie należy jednak tego sposobu inicjalizacji mylić z konstruktorem klasy w języku C++. Wektorowe typy danych w języku GLSL nie są klasami, lecz typami wbudowanymi. Wektory można przypisywać jeden do drugiego, dodawać, skalować przez wartości skalarne (nie wektorowe) itd.

```
vVertexPos = vOldPos + vOffset;
vVertexPos = vNewPos;
vVertexPos += vec4(1.0f, 1.0f, 0.0f, 0.0f);
vVertexPos *= 5.0f;
```

Kolejną niezwykłą cechą języka GLSL jest sposób odwoływania się do poszczególnych elementów wektora. Jeśli wiesz, co to takiego konstrukcja `union` w języku C++, wektory możesz sobie wyobrazić jako unie na steroidach. W celu odwołania się do elementów wektora można użyć kropki albo dowolnego z trzech zestawów identyfikatorów: `xyzw`, `rgba` oraz `stpq`. Podczas pracy z typami wektorowymi najczęściej używamy identyfikatorów `xyzw`.

```
vVertexPos.x = 3.0f;
vVertexPos.xy = vec2(3.0f, 5.0f);
vVertexPos.xyz = vNewPos.xyz;
```

Podczas pracy z kolorami korzystamy z identyfikatorów `rgba`.

```
vOutputColor.r = 1.0f;
vOutputColor.rgba = vec4(1.0f, 1.0f, 0.5f, 1.0f);
```

Identyfikatorów `stpq` używany w pracy ze współrzędnymi tekstur.

```
vTexCoord.st = vec2(1.0f, 0.0f);
```

Dla języka GLSL nie ma znaczenia, którego zestawu identyfikatorów użyjemy, tzn. nie ma żadnego błędu w poniższej instrukcji:

```
vTexCoord.st = vVertex.st;
```

Nie można natomiast mieszać różnych zestawów w jednej operacji dostępu do wektora:

```
vTexCoord.st = vVertex.xt; // Nie można użyć jednocześnie x i t!
```

Wektorowe typy danych obsługują również technikę transformacji o nazwie **swizzling**. Polega ona na zamianie miejscami dwóch lub większej liczby elementów wektora. Może się to np. przydać do zamiany danych kolorów w formacie RGB na BGR:

```
vNewColor.bgra = vOldColor.rgba;
```

Wektorowe typy danych są typami rodzimymi nie tylko języka GLSL, lecz również sprzętu. Są szybkie, a operacje wykonywane są na wszystkich czterech składnikach naraz. Na przykład poniższa operacja:

```
vVertex.x = vOtherVertex.x + 5.0f;
vVertex.y = vOtherVertex.y + 4.0f;
vVertex.z = vOtherVertex.z + 1.0f;
```

zostałaby wykonana znacznie szybciej, gdyby użyto rodzimej notacji wektorowej:

```
vVertex.xyz = vOtherVertex.xyz + vec3(5.0f, 4.0f, 1.0f);
```

## Typy macierzowe

Oprócz typów wektorowych, język GLSL obsługuje kilka typów macierzowych. Jednak w przeciwieństwie do tych pierwszych, typy macierzowe obsługują tylko wartości zmiennoprzecinkowe. Nie można tworzyć macierzy wartości typu całkowitoliczbowego ani logicznych, ponieważ nie mają one praktycznego zastosowania. W tabeli 6.2 znajduje się lista dostępnych typów macierzowych.

Macierz w języku GLSL to w istocie tablica wektorów kolumnowych (może warto sobie w tym momencie powtórzyć wiadomości o porządku kolumnowym macierzy z rozdziału 4. „Podstawy przekształceń geometrycznych. Najważniejsze informacje o wektorach i macierzach”). Aby na przykład ustawić ostatnią kolumnę macierzy 4×4, można zastosować następujący kod:

```
mModelView[3] = vec4(0.0f, 0.0f, 0.0f, 1.0f);
```

Aby pobrać ostatnią kolumnę macierzy:

```
vec4 vTranslation = mModelView[3];
```

Jeszcze precyzyjniejsze zapytanie:

```
vec3 vTranslation = mModelView[3].xyz;
```



Tabela 6.2. Typy macierzowe języka GLSL

Typ	Opis
mat2, mat2x2	Dwie kolumny i dwa wiersze
mat3, mat3x3	Trzy kolumny i trzy wiersze
mat4, mat4x4	Cztery kolumny i cztery wiersze
mat2x3	Dwie kolumny i trzy wiersze
mat2x4	Dwie kolumny i cztery wiersze
mat3x2	Trzy kolumny i dwa wiersze
mat3x4	Trzy kolumny i cztery wiersze
mat4x2	Cztery kolumny i dwa wiersze
mat4x3	Cztery kolumny i trzy wiersze

Macierze można także mnożyć przez wektory. Działanie takie często wykonuje się w celu przekształcenia wierzchołka przez macierz rzutowania model-widok:

```
vec4 vVertex;
mat4.mvpMatrix;
...
...
vOutPos =.mvpMatrix * vVertex;
```

Typy macierzowe, podobnie jak wektory, mają również swoje konstruktory. Aby na przykład wpisać bezpośrednio do kodu macierz 4×4, można napisać poniższą instrukcję:

```
mat4 vTransform = mat4(1.0f, 0.0f, 0.0f, 0.0f,
                      0.0f, 1.0f, 0.0f, 0.0f,
                      0.0f, 0.0f, 1.0f, 0.0f,
                      0.0f, 0.0f, 0.0f, 1.0f);
```

Jako macierz przekształcenia zastosowaliśmy macierz jednostkową. Można także użyć szybszego konstruktora macierzy wypełniającego tylko przekątną jedną wartością.

```
mat4 vTransform = mat4(1.0f);
```

## Kwalifikatory zmiennych

Deklaracje zmiennych shadera można rozszerzyć o pewne kwalifikatory służące do określania ich jako zmiennych wejściowych (`in` i `uniform`), wyjściowych (`out`) lub stałych (`const`). Zmienne wejściowe odbierają dane od klienta OpenGL (atrybuty przesyłane poprzez C lub C++) lub z poprzedniego etapu pracy shadera (np. zmienne przekazywane przez shadera wierzchołków do shadera fragmentów). Zmienne wyjściowe służą do zapisywania na dowolnym etapie pracy

wartości, które chcemy udostępnić w następnych etapach, np. w celu przekazania danych z shadera wierzchołków do shadera fragmentów albo zapisania ostatecznej wartości koloru przez shader fragmentów. W tabeli 6.3 zostały zebrane główne kwalifikatory zmiennych.

Tabela 6.3. Kwalifikatory zmiennych

Kwalifikator	Opis
<brak>	Zwykła zmienna lokalna niedostępna i niewidoczna na zewnątrz
const	Stała czasu kompilacji lub parametr tylko do odczytu funkcji
in	Zmienna przekazana z poprzedniego etapu
in centroid	Zmienna przekazana z poprzedniego stanu, stosuje interpolację środkową
out	Zmienna przekazywana do następnego etapu przetwarzania lub przechowująca wartość zwrótną funkcji
out centroid	Zmienna przekazywana do następnego etapu przetwarzania, stosuje interpolację środkową
inout	Zmienna do odczytu i zapisu. Dozwolona tylko jako parametr funkcji lokalnej
uniform	Wartość przekazywana od klienta, taka sama dla wszystkich wierzchołków

Kwalifikatora `inout` można używać tylko do deklarowania parametrów funkcji. Ponieważ język GLSL nie obsługuje wskaźników (czyli referencji), kwalifikator ten stanowi jedyny sposób na przekazanie wartości do funkcji i umożliwienie jej zmodyfikowania i zwrócenia wartości zmiennej. Na przykład funkcja zadeklarowana poniżej:

```
int CalculateSomething(float fTime, float fStepSize, inout float fVariance);
```

zwróciłaby wartość całkowitoliczbową (np. znacznik powodzenia lub niepowodzenia) i dodatkowo mogłaby zmodyfikować wartość zmiennej `fVariance`, a kod wywołujący mógłby odczytać również tę jej nową wartość. Aby umożliwić modyfikowanie parametru w językach C i C++, można by było zadeklarować tę funkcję przy użyciu wskaźnika:

```
int CalculateSomething(float fTime, float fStepSize, float* fVariance);
```

Kwalifikator `centroid` działa wyłącznie w przypadku renderowania buforów wielopróbkowanych. W buforze o pojedynczym próbkowaniu interpolację wykonuje się zawsze od środka piksela. W przypadku wielopróbkowania, gdy zostanie użyty kwalifikator `centroid`, wartość interpolowana wypada w obrębie zarówno obiektu podstawowego, jak i piksela. Więcej na temat wielopróbkowania piszemy w rozdziale 9. „Buforowanie — techniki zaawansowane”.

Domyślnie parametry są interpolowane między etapami shaderów w sposób odpowiedni dla perspektywy. Można zastosować interpolację nieperspektywiczną za pomocą słowa kluczowego `noperspective`, a nawet całkiem ją wyłączyć za pomocą słowa `flat`. Można także użyć słowa kluczowego `smooth`, aby bezpośrednio zaznaczyć, że zmienna jest płynnie interpolowana w sposób perspektywiczny, ale to jest i tak działanie domyślne. Poniżej znajduje się kilka przykładowych deklaracji:

```
smooth out vec3 vSmoothValue;
flat out vec3 vFlatColor;
noperspective float vLinearlySmoothed;
```

## Prawdziwy shader

Nadszedł czas, aby w końcu przyjrzeć się prawdziwej parze shaderów, które robią coś użytecznego. W klasie `GLShaderManager` znajduje się standardowy shader nazywany shaderem jednostkowym. Nie stosuje on żadnych przekształceń geometrii i rysuje obiekty podstawowe wypełnione tylko jednym kolorem. To wydaje się nieco zbyt mało. Rozbudujemy go zatem trochę, aby zobaczyć, jak się cieniuje obiekty podstawowe, takie jak np. trójkąt, stosując inną wartość koloru dla każdego wierzchołka. Na listingu 6.1 przedstawiony jest kod shadera wierzchołków, a na listingu 6.2 — shadera fragmentów.

**Listing 6.1.** Shader wierzchołków `ShadedIdentity`

---

```
// Shader ShadedIdentity
// Shader wierzchołków
// Richard S. Wright Jr
// OpenGL. Księga eksperta
#version 330

in vec4 vVertex;           // Atrybut położenia wierzchołka
in vec4 vColor;           // Atrybut koloru wierzchołka

out vec4 vVaryingColor;   // Wartość koloru przekazywana do shadera fragmentów

void main(void)
{
    vVaryingColor = vColor; // Kopiowanie wartości koloru
    gl_Position = vVertex;  // Przekazanie dalej położenia wierzchołka
}
```

**Listing 6.2.** Shader fragmentów `ShadedIdentity`

---

```
// Shader ShadedIdentity
// Shader fragmentów
// Richard S. Wright Jr
// OpenGL. Księga eksperta
#version 330

out vec4 vFragColor;      // Kolor fragmentu do rasteryzacji
in vec4 vVaryingColor;   // Kolor przychodzący od shadera wierzchołków

void main(void)
{
    vFragColor = vVaryingColor; // Kolor interpolowany na fragment
}
```

---

## Wersje języka GLSL

Pierwszy wiersz niebędący komentarzem w każdym shaderze to określenie wersji języka:

```
#version 330
```

To oznacza, że niniejszy shader wymaga przynajmniej wersji 3.3 języka GLSL. Jeśli sterownik OpenGL jej nie obsługuje, shadera nie uda się skompilować. W OpenGL 3.2 wprowadzono język GLSL 1.5, w OpenGL 3.1 — GLSL 1.4, a w OpenGL 3.0 — GLSL 1.3. Trudno się połączyć? Nie tylko Tobie. Dlatego rada ARB zdecydowała, że od OpenGL 3.3 numer wersji języka GLSL będzie odpowiadał wersji biblioteki. Wersja 4.0 biblioteki OpenGL została opublikowana w tym samym czasie, co wersja 3.3, i odpowiadający jej język GLSL ma również numer 4.0. Konstrukcja wymagająca wersji 4.0 GLSL wyglądałaby następująco:

```
#version 400
```

Jeśli zajrzysz do kodu shaderów standardowych w bibliotece GLTools, nie znajdziesz w nich takiej informacji o wersji języka. Biblioteka ta jest przeznaczona do pracy z profilem zgodnościowym i zastosowano w niej starsze konwencje z GLSL 1.1. W istocie biblioteka ta współpracuje ze sterownikami OpenGL nawet w wersji 2.1. Pamiętajmy, że stanowi ona tylko pomoc w rozpoczęciu korzystania z biblioteki OpenGL.

## Deklaracje atrybutów

Atrybuty są określane dla poszczególnych wierzchołków przez kliencki kod C/C++. W naszym shaderze wierzchołków zostały one zadeklarowane przy użyciu specyfikatora `in`.

```
in vec4 vVertex;
in vec4 vColor;
```

Dwie powyższe instrukcje deklarują dwa atrybuty wejściowe, czteroskładnikowe położenie wierzchołka oraz czteroskładnikową wartość koloru wierzchołka. Shader ten jest wykorzystywany przez program `ShadedTriangle`. Przy użyciu klasy `GLBatch` utworzyliśmy trzy położenia wierzchołków i trzy wartości kolorów. Jak klasa `GLBatch` przekazuje te wartości do shadera, dowiesz się w podrozdziale „Kompilowanie, wiązanie i konsolidowanie”. Przypomnijmy z rozdziału 3., że w języku GLSL można mieć maksymalnie 16 atrybutów w programie wierzchołkowym. *Ponadto każdy atrybut jest zawsze czteroskładnikowym wektorem, nawet jeśli nie wszystkich składników używamy.* Gdybyśmy na przykład wewnętrznie wyznaczyli jako atrybut tylko jedną wartość typu `float`, i tak zajęłaby ona przestrzeń czterech wartości zmiennoprzecinkowych.

Dodatkową rzeczą do zapamiętania jest fakt, że zmienne oznaczone jako `in` są przeznaczone tylko do odczytu. Może się wydawać sprytnym rozwiązaniem ponowne użycie nazwy zmiennej w jakichś pośrednich obliczeniach w shaderze, ale kompilator GLSL w sterowniku zgłosiłby w takiej sytuacji błąd.

## Deklarowanie danych wyjściowych

Dalej zadeklarowaliśmy jedną zmienną wyjściową, będącą również czteroskładnikowym wektorem liczb zmiennoprzecinkowych.

```
out vec4 vVaryingColor;
```

Zmienna ta będzie określać wartość koloru wierzchołka, który ma zostać przekazany do shadera fragmentów. W shaderze fragmentów musi ona zostać zadeklarowana jako `in`, aby nie został zwrócony błąd konsolidatora podczas kompilacji i konsolidacji shaderów.

Gdy w shaderze wierzchołków zmienna zostanie zadeklarowana jako `out`, a w shaderze fragmentów jako `in`, shader fragmentów odbierze ją jako wartość interpolowaną. Domyślnie interpolacja ta jest wykonywana zgodnie z perspektywą. Aby mieć pewność, że tak się stanie, można przed zmienną wstawić dodatkowy specyfikator `smooth`. Można także zastosować specyfikator `flat`, aby wyłączyć interpolację, lub `noperspective`, aby zastosować prostą interpolację liniową między wartościami. Podczas używania słowa kluczowego `flat` trzeba wziąć pod uwagę pewne dodatkowe fakty, o których szerzej piszemy w podrozdziale „Wierzchołek prowokujący”.

## Przetwarzanie wierzchołków

Dochodzimy do głównej części naszego shadera wierzchołków, która jest wykonywana jeden raz dla każdego wierzchołka w porcji danych.

```
void main(void)
{
    vVaryingColor = vColor;
    gl_Position = vVertex;
}
```

Ten kod jest bardzo prosty. Przypisaliśmy przychodzący atrybut koloru do wychodzącej wartości interpolowanej i przypisaliśmy przychodzącą wartość wierzchołka bezpośrednio do zmiennej `gl_Position` bez transformacji. Zmienna `gl_Position` to wbudowany czteroskładnikowy wektor zawierający wymagany wynik shadera wierzchołków. Wartości z tej zmiennej są wykorzystywane na etapie składania geometrii do tworzenia obiektu podstawowego. Pamiętajmy, że ponieważ nie wykonujemy żadnych dodatkowych przekształceń, nasz wierzchołek zostanie odwzorowany na kartezjański układ współrzędnych o zakresie wartości od  $-1,0$  do  $1,0$  na wszystkich trzech osiach.

## Przetwarzanie fragmentów

Teraz przechodzimy do shadera fragmentów. W przypadku renderowania obiektu podstawowego, takiego jak trójkąt, najpierw wierzchołki są przetwarzane przez shader wierzchołków, później są one składane w trójkąt, a następnie rasteryzowane przez sprzęt. Urządzenie określa położenie poszczególnych fragmentów na ekranie (a mówiąc dokładniej — w buforze kolorów), a następnie dla każdego z nich (jeśli nie jest stosowane wielopróbkowanie, fragment odpowiada pikselowi) wywołuje egzemplarz shadera fragmentów. Kolor zwracany przez naszego shadera fragmentów to czteroskładnikowy wektor liczb zmiennoprzecinkowych, który deklarujemy następująco:

```
out vec4 vFragColor;
```

Jeśli shader fragmentów zwraca tylko jedną wartość, jest ona wewnętrznie określana jako „wartość wyjściowa zero”. Jest to pierwszy wynik shadera fragmentów, który następnie zostaje wysłany do bufora ustawionego przez funkcję `glDrawBuffers`. Domyślnym buforem jest `GL_BACK`, czyli tylny bufor koloru (oczywiście w kontekstach z podwójnym buforowaniem). Często zdarza się tak, że bufor koloru nie zawiera czterech składników zmiennoprzecinkowych i wówczas wartości wyjściowe są rzutowane na zakres bufora docelowego. W większości przypadków mogą to być np. cztery bajty bez znaku (o wartościach od 0 do 255). Moglibyśmy także zwrócić wartości całkowitoliczbowe przy użyciu typu wektorowego `ivec4` i również one zostałyby odwzorowane na zakres bufora kolorów. Możliwe jest także zwrócenie czegoś więcej niż tylko wartość koloru, jak również zapisywanie danych w kilku buforach jednocześnie. Te techniki jednak wykraczają daleko poza zakres tego wstępnego rozdziału.

Do shadera fragmentów zostaje przesłana płynnie interpolowana wartość koloru pochodząca z shadera wierzchołków. Deklaruje się ją jako zmienną z kwalifikatorem `in`:

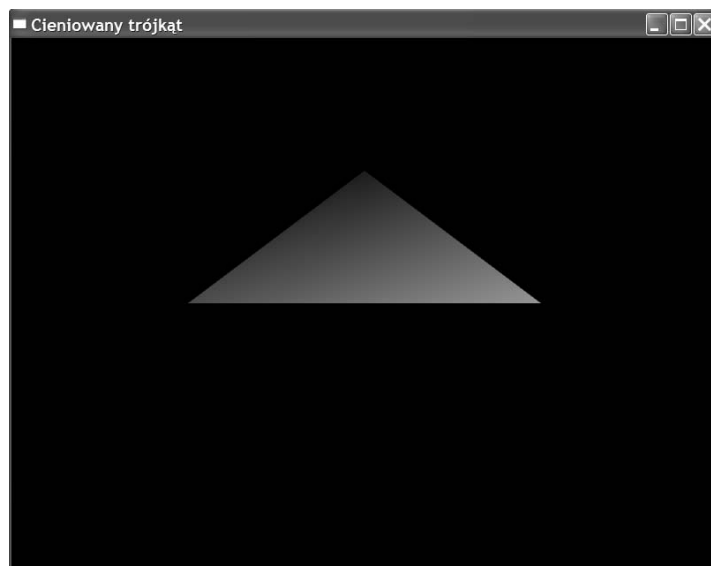
```
in vec4 vVaryingColor;
```

Główna część shadera fragmentów jest jeszcze prostsza niż shadera wierzchołków. Przypisujemy w niej otrzymaną wartość koloru bezpośrednio do koloru fragmentu.

```
void main(void)
{
    vFragColor = vVaryingColor;
}
```

Efekt działania tego shadera przedstawia rysunek 6.2.

**Rysunek 6.2.**  
Wynik działania programu `ShadedTriangle`



## Kompilowanie, wiązanie i konsolidowanie

Opisaliśmy zasadę działania prostego shadera, więc teraz powinniśmy dowiedzieć się, jak jest on kompilowany i konsolidowany w celu przygotowania do użytku w OpenGL. Kod źródłowy shadera jest przekazywany sterownikowi, kompilowany, a następnie konsolidowany, tak jak każdy typowy program w językach C i C++. Dodatkowo konieczne jest **powiązanie** nazw atrybutów z shadera z jednym z 16 gniazd atrybutów alokowanych i udostępnianych przez GLSL. Po drodze sprawdzamy błędy i nawet otrzymujemy informacje diagnostyczne od sterownika, jeśli próba skompilowania projektu się nie powiedzie.

API OpenGL nie obsługuje żadnych operacji wejścia i wyjścia na plikach. Programista musi sam zdobyć kod źródłowy swoich shaderów w najbardziej odpowiadający mu sposób. Jedną z najprostszycy metod jest zapisanie tego kodu w plikach tekstowych ASCII. Wówczas kod źródłowy można z takich plików pobrać za pomocą typowych funkcji systemu plików. Podejście to zastosowaliśmy w naszym przykładzie. Zastosowaliśmy również konwencję, według której plikom shaderów wierzchołków nadaje się rozszerzenie *.vp*, a plikom shaderów fragmentów — *.fp*. Innym rozwiązaniem jest zapisanie tekstów w postaci tablic znaków wbudowanych w kod C lub C++. Jednak tak zapisany kod trudno się modyfikuje i mimo iż cały kod znajduje się w jednym miejscu, zmienianie shaderów i eksperymentowanie z kodem źródłowym jest utrudnione. Oczywiście kod shaderów można także generować za pomocą specjalnych algorytmów, a nawet pobierać go z bazy danych albo pliku zaszyfrowanego. Te metody mogą być bardzo przydatne, gdy będziemy chcieli przesyłać gdzieś nasze aplikacje, ale w zastosowaniach edukacyjnych nic nie przebije zwykłych plików tekstowych.

Funkcja `glLoadShaderPairWithAttributes` to prawdziwy kombajn do wczytywania i inicjalizowania shaderów. Jej kod przedstawia listing 6.3. Na podstawie jego analizy przestudiujemy proces wczytywania shadera.

**Listing 6.3.** Funkcja `glLoadShaderPairWithAttributes`

```

////////////////////////////////////
// Wczytuje parę shaderów oraz je kompiluje i łączy.
// Podaj kod źródłowy każdego shadera. Następnie
// podaj nazwy tych shaderów, określ liczbę atrybutów
// oraz podaj indeks i nazwę każdego atrybutu.
GLuint glLoadShaderPairWithAttributes(const char *szVertexProg,
                                     const char *szFragmentProg, ...)
{
    // Tymczasowe obiekty shadera
    GLuint hVertexShader;
    GLuint hFragmentShader;
    GLuint hReturn = 0;
    GLint testVal;

    // Tworzenie obiektów shadera
    hVertexShader = glCreateShader(GL_VERTEX_SHADER);
    hFragmentShader = glCreateShader(GL_FRAGMENT_SHADER);

```

```
// Ładowanie obiektów. W razie niepowodzenia należy wszystko usunąć i zwrócić wartość NULL
// Shader wierzchołków
if (glLoadShaderFile(szVertexProg, hVertexShader) == false)
{
    glDeleteShader(hVertexShader);
    glDeleteShader(hFragmentShader);
    cout << "Shader " << szVertexProg
        << " nie został znaleziony.\n";
    return (GLuint)NULL;
}

// Shader fragmentów
if (glLoadShaderFile(szFragmentProg, hFragmentShader) == false)
{
    glDeleteShader(hVertexShader);
    glDeleteShader(hFragmentShader);
    cout << "Shader " << szFragmentProg
        << " nie został znaleziony.\n";
    return (GLuint)NULL;
}

// Kompilacja shaderów
glCompileShader(hVertexShader);
glCompileShader(hFragmentShader);

// Sprawdzanie błędów w shaderze wierzchołków
glGetShaderiv(hVertexShader, GL_COMPILE_STATUS, &testVal);
if (testVal == GL_FALSE)
{
    char infoLog[1024];
    glGetShaderInfoLog(hVertexShader, 1024, NULL, infoLog);
    cout << "Kompilacja shadera " << szVertexProg
        << " nie powiodła się. Błąd:\n"
        << infoLog << "\n";
    glDeleteShader(hVertexShader);
    glDeleteShader(hFragmentShader);
    return (GLuint)NULL;
}

// Sprawdzanie błędów w shaderze fragmentów
glGetShaderiv(hFragmentShader, GL_COMPILE_STATUS, &testVal);
if (testVal == GL_FALSE)
{
    char infoLog[1024];
    glGetShaderInfoLog(hFragmentShader, 1024, NULL, infoLog);
    cout << "Kompilacja shadera " << hFragmentShader
        << " nie powiodła się. Błąd:\n"
        << infoLog << "\n";
    glDeleteShader(hVertexShader);
    glDeleteShader(hFragmentShader);
    return (GLuint)NULL;
}

// Tworzenie finalnego obiektu programu i dołączenie shaderów
hReturn = glCreateProgram();
```



```

glAttachShader(hReturn, hVertexShader);
glAttachShader(hReturn, hFragmentShader);

// Teraz musimy powiązać nazwy atrybutów z odpowiadającymi im lokalizacjami
// Lista atrybutów
va_list attributeList;
va_start(attributeList, szFragmentProg);

// Iteracja przez listę argumentów
char *szNextArg;
int iArgCount = va_arg(attributeList, int); // Liczba atrybutów
for(int i = 0; i < iArgCount; i++)
{
    int index = va_arg(attributeList, int);
    szNextArg = va_arg(attributeList, char*);
    glBindAttribLocation(hReturn, index, szNextArg);
}
va_end(attributeList);

// Próba konsolidacji
glLinkProgram(hReturn);

// Te już nie są potrzebne
glDeleteShader(hVertexShader);
glDeleteShader(hFragmentShader);

// Sprawdzenie, czy konsolidacja się udała
glGetProgramiv(hReturn, GL_LINK_STATUS, &testVal);
if(testVal == GL_FALSE)
{
    char infoLog[1024];
    glGetProgramInfoLog(hReturn, 1024, NULL, infoLog);
    cout << "Konsolidacja programu " << hReturn
         << " nie powiodła się. Błąd:\n"
         << infoLog << "\n";
    glDeleteProgram(hReturn);
    return (GLuint)NULL;
}

// Wszystko zrobione. Zwrócenie shadera gotowego do użytku
return hReturn;
}

```

## Określanie atrybutów

W prototypie funkcji widać, że pobiera ona nazwę pliku shadera wierzchołków i shadera fragmentów oraz mogącą się zmieniać liczbę parametrów określających atrybuty.

```

GLuint gltLoadShaderPairWithAttributes(const char *szVertexProg,
                                       const char *szFragmentProg, ...);

```

Dla osób, które nigdy nie widziały deklaracji funkcji przyjmującej zmienną liczbę parametrów, znajdujące się na końcu listy argumentów trzy kropki mogą wyglądać jak pomyłka w druku.

Inne funkcje języka C pobierające zmienną liczbę argumentów to np. `printf` i `sprintf`. W tej funkcji pierwszy dodatkowy parametr określa liczbę atrybutów znajdujących się w shaderze wierzchołków. Po nim znajduje się wartość indeksu (liczona od zera) pierwszego atrybutu, a następnie nazwa atrybutu w postaci tablicy znaków. Numer gniazda atrybutu i nazwa atrybutu są powtarzane tyle razy, ile potrzeba. Aby na przykład wczytać shader mający atrybuty położenia wierzchołka i normalnej do powierzchni, wywołanie funkcji `glLoadShaderPairWithAttributes` mogłoby wyglądać następująco:

```
hShader = glLoadShaderPairWithAttributes("vertexProg.vp",
                                         "fragmentProg.fp", 2, 0, "vVertexPos", 1, "vNormal");
```

Wartości 0 i 1 jako lokalizacje atrybutów zostały wybrane arbitralnie. Należy tylko pamiętać, aby zawierały się one w przedziale od 0 do 15. Równie dobrze moglibyśmy użyć wartości 7 i 13. Natomiast w klasach biblioteki GLTools `GLBatch` i `GLTriangleBatch` stosowany jest spójny zestaw lokalizacji atrybutów określanych za pomocą następującej instrukcji `typedef`:

```
typedef enum GLT_SHADER_ATTRIBUTE { GLT_ATTRIBUTE_VERTEX = 0,
                                     GLT_ATTRIBUTE_COLOR, GLT_ATTRIBUTE_NORMAL,
                                     GLT_ATTRIBUTE_TEXTURE0, GLT_ATTRIBUTE_TEXTURE1,
                                     GLT_ATTRIBUTE_TEXTURE2, GLT_ATTRIBUTE_TEXTURE3,
                                     GLT_ATTRIBUTE_LAST};
```

Używając tych identyfikatorów lokalizacji atrybutów, można zacząć używać własnych shaderów obok shaderów standardowych dostarczanych w klasie `GLShaderManager`. To również oznacza, że nadal możemy przesyłać geometrię przy użyciu klas `GLBatch` i `GLTriangleBatch`, aż do rozdziału 12. „Zarządzanie geometrią — techniki zaawansowane”, w którym bardziej szczegółowo zajmiemy się technikami przesyłania atrybutów wierzchołków.

## Pobieranie kodu źródłowego

Najpierw trzeba utworzyć dwa obiekty shaderów — po jednym dla shadera wierzchołków i shadera fragmentów.

```
hVertexShader = glCreateShader(GL_VERTEX_SHADER);
hFragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
```

Przy użyciu identyfikatorów shaderów możemy wczytać ich kod źródłowy. Pominiemy szczegóły funkcji `glLoadShaderFile`, ponieważ jej głównym zadaniem jest wczytanie tekstu shadera z pliku tekstowego o podanej nazwie, zapisanego na dysku. Gdy już to zostanie zrobione, poniższy kod przesyła kod źródłowy shadera do obiektu shadera. Pamiętaj, że musimy to zrobić dwa razy — raz dla shadera wierzchołków i raz dla shadera fragmentów.

```
GLchar *fsStringPtr[1];

fsStringPtr[0] = (GLchar *)szShaderSrc;
glShaderSource(shader, 1, (const GLchar **)fsStringPtr, NULL);
```

Zmienna `szShaderSrc` to wskaźnik znakowy wskazujący cały tekst shadera. Natomiast `shader` to identyfikator obiektu shadera, który jest wczytywany.

## Kompilowanie shaderów

Kompilacja shaderów to prosta operacja polegająca na wywołaniu jednej funkcji dla każdego z nich.

```
glCompileShader(hVertexShader);
glCompileShader(hFragmentShader);
```

W każdej implementacji OpenGL znajduje się wbudowany kompilator języka GLSL dostarczony przez producenta sprzętu. Uznano bowiem, że każdy producent wie najlepiej, jak powinien działać kompilator dla jego sprzętu. Oczywiście, podobnie jak w przypadku każdego programu w językach C i C++, kompilację shadera GLSL może uniemożliwić wiele czynników, takich jak błędy składni czy błędy implementacji itp. Do sprawdzania, czy operacja się powiodła, służy funkcja `glGetShader` z argumentem `GL_COMPILE_STATUS`.

```
glGetShaderiv(hVertexShader, GL_COMPILE_STATUS, &testVal);
```

Jeśli po powrocie funkcji `testVal` ma wartość `GL_FALSE`, oznacza to, że kompilacja się nie powiodła. Gdybyśmy jednak mieli możliwość tylko dowiedzenia się, czy kompilacja została zakończona powodzeniem, czy nie, pisanie shaderów byłoby bardzo trudne. Dlatego w przypadku nieudanej operacji możemy sprawdzić w dzienniku shadera, co takiego się stało — wystarczy wyświetlić jego zawartość za pomocą funkcji `glGetShaderInfoLog`. W funkcji, którą analizujemy, komunikat o błędzie jest wyświetlany w oknie konsoli, po czym następuje usunięcie obiektów shaderów i zwrócenie wartości `NULL`.

```
if(testVal == GL_FALSE)
{
    char infoLog[1024];
    glGetShaderInfoLog(hVertexShader, 1024, NULL, infoLog);
    cout << "Kompilacja shadera " << szVertexProg
        << " nie powiodła się. Błąd:\n"
        << infoLog << "\n";
    glDeleteShader(hVertexShader);
    glDeleteShader(hFragmentShader);
    return (GLuint)NULL;
}
```

## Dołączanie i wiązanie

Skompilowanie kodu shaderów to dopiero połowa sukcesu. Zanim przejdziemy do ich konsolidacji, zdobędziemy się na małą dygresję. Najpierw musimy utworzyć finalny obiekt shadera i dołączyć do niego shadery wierzchołków i fragmentów.

```
hReturn = glCreateProgram();
glAttachShader(hReturn, hVertexShader);
glAttachShader(hReturn, hFragmentShader);
```

Teraz shader jest gotowy do konsolidacji. Zanim to jednak zrobimy, musimy zrobić jedną ważną rzecz — związać nazwy zmiennych atrybutów z określonymi numerami ich lokalizacji. Do tego służy funkcja `glBindAttribLocation`. Oto jej prototyp:

```
void glBindAttribLocation(GLuint shaderProg, GLuint attribLocation,
    const GLchar *szAttributeName);
```

Funkcja ta pobiera identyfikator interesującego nas shadera, lokalizację atrybutu, która ma zostać użyta w wiązaniu, oraz nazwę zmiennej atrybutu. Na przykład w shaderach standardowych biblioteki GLTools przyjęliśmy konwencję, że dla zmiennej atrybutu położenia wierzchołka zawsze stosujemy nazwę `vVertex`, a dla lokalizacji atrybutu wartość `GLT_ATTRIBUTE_VERTEX` (wartość 0). Również możesz zastosować taką konwencję.

```
glBindAttribLocation(hShader, GLT_ATTRIBUTE_VERTEX, "vVertex");
```

Wiązanie lokalizacji atrybutów musi zostać wykonane przed konsolidacją. W poniższym kodzie przechodzimy iteracyjnie przez listę argumentów i wywołujemy funkcję dla każdego atrybutu, który chcemy związać.

```
// Iteracja przez listę argumentów
char *szNextArg;
int iArgCount = va_arg(attributeList, int); // Liczba atrybutów
for(int i = 0; i < iArgCount; i++)
{
    int index = va_arg(attributeList, int);
    szNextArg = va_arg(attributeList, char*);
    glBindAttribLocation(hReturn, index, szNextArg);
}
va_end(attributeList);
```

## Konsolidacja shaderów

Nadszedł w końcu czas na skonsolidowanie naszych shaderów, po czym będziemy mogli usunąć reprezentujące je obiekty.

```
glLinkProgram(hReturn);

// Te już nie są potrzebne
glDeleteShader(hVertexShader);
glDeleteShader(hFragmentShader);
```

Podobnie jak w przypadku kompilacji, w pomyślnym zakończeniu konsolidacji może przeszkodzić wiele czynników. Stanie się tak na przykład wówczas, gdy w shaderze wierzchołków zadeklarujemy zmienną `out`, a nie zadeklarujemy jej odpowiednika w shaderze fragmentów. Nawet jeśli o tym nie zapomnimy, musimy jeszcze pamiętać, że zmienne te muszą być tego samego typu. Dlatego przed powrotem sprawdzamy błędy i w razie ich wystąpienia wyświetlamy stosowny komunikat diagnostyczny, podobnie jak przy kompilacji.

Teraz nasz shader jest w pełni gotowy do użycia. Powinniśmy jeszcze zaznaczyć, że jeśli utworzymy program cieniujący i zakończymy jego używanie (np. przy zamykaniu programu), powinniśmy go usunąć za pomocą poniższej funkcji.

```
void glDeleteProgram(GLuint program);
```

## Praktyczne wykorzystanie shadera

Aby użyć naszego shadera GLSL, część zrobienia tego musimy zakomunikować za pomocą funkcji `glUseProgram`:

```
glUseProgram(myShaderProgram);
```

W ten sposób aktywowaliśmy naszego shadera, dzięki czemu wszystkie nasze obiekty geometryczne będą przetwarzane przez nasze shadery wierzchołków i fragmentów. Dane uniform i teksturowe należy utworzyć przed przesłaniem atrybutów wierzchołków. Jak to zrobić, wyjaśnimy już za chwilę. Natomiast przesyłanie atrybutów wierzchołków to bardzo obszerny temat, który zasługuje na osobne omówienie w rozdziale 12. Na razie pozwolimy zarządzać naszą geometrią klasom `GLBatch` i `GLTriangleBatch`.

W pierwszym przykładowym programie, jaki przedstawiliśmy w tym rozdziale, `ShadedTriangle`, wczytaliśmy trójkąt do egzemplarza klasy `GLBatch` o nazwie `triangleBatch` przy użyciu najprostszego (nazwaliśmy go układem „jednostkowym”) układu współrzędnych:

```
// Wczytywanie trójkąta
GLfloat vVerts[] = { -0.5f, 0.0f, 0.0f,
                    0.5f, 0.0f, 0.0f,
                    0.0f, 0.5f, 0.0f };

GLfloat vColors [] = { 1.0f, 0.0f, 0.0f, 1.0f,
                     0.0f, 1.0f, 0.0f, 1.0f,
                     0.0f, 0.0f, 1.0f, 1.0f };
triangleBatch.Begin(GL_TRIANGLES, 3);
triangleBatch.CopyVertexData3f(vVerts);
triangleBatch.CopyColorData4f(vColors);
triangleBatch.End();

myIdentityShader = glLoadShaderPairWithAttributes("ShadedIdentity.vp",
                                                "ShadedIdentity.fp", 2, GLT_ATTRIBUTE_VERTEX, "vVertex",
                                                GLT_ATTRIBUTE_COLOR, "vColor");
```

Każdemu wierzchołkowi ustawiliśmy inny kolor, odpowiednio czerwony, zielony i niebieski. Na zakończenie załadowaliśmy naszą parę shaderów przy użyciu funkcji `glLoadShaderPairWithAttributes`, z którą już mogłeś się zapoznać. Zauważmy, że mamy dwa zbiory atrybutów — wartości wierzchołków i kolorów — odpowiadające zbiorom danych przesyłanym do klasy `GLBatch`.

Teraz, aby przesłać porcję danych, wystarczy wybrać shader i pozwolić klasie `GLBatch` przekazać nasze atrybuty wierzchołków:

```
glUseProgram(myIdentityShader);
triangleBatch.Draw();
```

Wynik tych wszystkich naszych działań przedstawia rysunek 6.2.

## Wierzchołek prowokujący

Program `ShadedTriangle` stanowi znakomity przykład płynnej interpolacji wierzchołków. Dla każdego wierzchołka został zdefiniowany inny kolor, w wyniku czego powstał trójkąt (widoczny na rysunku 6.2), na którym przejścia między kolorami są płynne. Świetnie, prawda? Możemy również przekazywać zmienne z jednego etapu shadera do drugiego jako typ `flat`. Jeśli chcemy, aby jakaś wartość pozostała niezmienna dla całej porcji danych, najlepiej zastosować typ `uniform`, o czym była mowa w rozdziale 3. Czasami jednak chcemy, aby jakaś wartość była niezmienna na całej powierzchni danego obiektu podstawowego, np. trójkąta, ale mogła się zmieniać między różnymi trójkątami. Przesyłanie dużej liczby trójkątów, np. po jednym trójkącie na porcję danych, tak jak w przypadku użycia zmiennych `uniform`, byłoby bardzo nieefektywnym rozwiązaniem. W takich przypadkach najlepiej użyć kwalifikatora `flat`. W shaderze `ShadedTriangle.vp` płynnie cieniowana wychodząca wartość koloru jest zadeklarowana następująco:

```
out vec4 vVaryingColor;
```

Gdybyśmy jednak do jej deklaracji dodali kwalifikator `flat` (i nie zapomnieli wprowadzić odpowiedniej poprawki w shaderze fragmentów), trójkąt miałby kolor niebieski.

```
flat out vec4 vFlatColor;
```

W przypadku, gdy dla każdego wierzchołka obiektu zdefiniowana jest inna wartość zapisana w płasko cieniowanej zmiennej, tylko jedna z nich może zostać zastosowana. Standardowo w takich przypadkach stosowana jest wartość ostatniego z wierzchołków, a więc w tym przypadku trójkąt będzie miał kolor niebieski. Konwencja ta nosi nazwę **wierzchołka prowokującego** (ang. *provoking vertex*). Poniższa funkcja pozwala zmienić ten sposób działania z ostatniego wierzchołka na pierwszy:

```
void glProvokingVertex(GLenum provokeMode);
```

Funkcja `provokeMode` przyjmuje wartości `GL_FIRST_VERTEX_CONVENTION` i `GL_LAST_VERTEX_CONVENTIONS` (domyślna).

Jak to działa, można zobaczyć w programie `ProvokingVertex`, który stanowi nieznacznie zmodyfikowaną wersję programu `ShadedTriangle`. Naciśnięciem spacji można zmienić konwencję, co z kolei spowoduje zmianę koloru trójkąta.

## Dane uniform shadera

Podczas gdy atrybuty służą do określania położenia wierzchołków, normalnych do powierzchni, współrzędnych teksturowych itp., zmienne `uniform` służą do przekazywania do shadera informacji, które powinny być stałe w całej porcji danych obiektu. Najczęściej spotykanym rodzajem danych tego typu dla shadera wierzchołków jest macierz przekształcenia. Wcześniej wszystko robiła za nas klasa `GLShaderManager` wykorzystująca wbudowane standardowe shadery i ich dane

uniform. Teraz jednak piszemy własne shadery, a więc musimy nauczyć się samodzielnie tworzyć dane uniform i to nie tylko do przechowywania macierzy. Jako uniform można zdefiniować dowolną zmienną w dowolnym z trzech etapów cieniowania (przypomnijmy, że w tym rozdziale zajmujemy się tylko shaderami wierzchołków i fragmentów). Aby to zrobić, wystarczy przed wybraną zmienną postawić słowo kluczowe `uniform`:

```
uniform float fTime;
uniform int iIndex;
uniform vec4 vColorValue;
uniform mat4.mvpMatrix;
```

Zmiennych uniform nie można deklarować jako `in` ani `out`, nie można ich interpolować między etapami cieniowania (choć można je kopiować do interpolowanych zmiennych) i nie można zmieniać ich wartości.

## Znajdowanie danych uniform

Po skompilowaniu i skonsolidowaniu shadera musimy w nim „znaleźć” lokalizację naszej zmiennej uniform. Służą do tego funkcja `glGetUniformLocation`.

```
GLint glGetUniformLocation(GLuint shaderID, const GLchar* varName);
```

Funkcja ta zwraca liczbę całkowitą ze znakiem, reprezentującą lokalizację zmiennej określonej parametrem `varName` w shaderze określonym parametrem `shaderID`. Na przykład poniższe wywołanie zwraca lokalizację zmiennej `uniform` o nazwie `vColorValue`:

```
GLint iLocation = glGetUniformLocation(myShader, "vColorValue");
```

Jeśli funkcja zwróci wartość `-1`, oznacza to, że nie udało się zlokalizować w shaderze zmiennej o podanej nazwie. Należy także pamiętać, że w nazwach zmiennych w shaderach rozpoznawana jest wielkość liter. Miejmy świadomość, że pomyślne zakończenie kompilacji shadera wcale nie oznacza, że zmienna `uniform` może nam „zniknąć”, jeśli nie będzie do niczego bezpośrednio wykorzystana. Nie musimy się martwić, że zmienne `uniform` zostaną usunięte w procesie optymalizacji kodu, ale jeśli zadeklarujemy taką zmienną i nie będziemy jej używać, kompilator ją nam usunie.

## Zmienne uniform skalarne i wektorowe

Do ustawienia wartości pojedynczego skalarnego lub wektorowego typu danych można użyć jednej z kilku wersji funkcji `glUniform`:

```
void glUniform1f(GLint location, GLfloat v0);
void glUniform2f(GLint location, GLfloat v0, GLfloat v1);
void glUniform3f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2);
void glUniform4f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2,
                 GLfloat v3);
void glUniform1i(GLint location, GLint v0);
void glUniform2i(GLint location, GLint v0, GLint v1);
```

```
void glUniform3i(GLint location, GLint v0, GLint v1, GLint v2);
void glUniform4i(GLint location, GLint v0, GLint v1, GLint v2, GLint v3);
```

Wyobraźmy sobie na przykład, że mamy w shaderze poniższe cztery deklaracje zmiennych:

```
uniform float fTime;
uniform int iIndex;
uniform vec4 vColorValue;
uniform bool bSomeFlag;
```

Kod w języku C lub C++ znajdujący je i ustawiający ich wartości mógłby wyglądać następująco:

```
GLint locTime, locIndex, locColor, locFlag;
locTime = glGetUniformLocation(myShader, "fTime");
locIndex = glGetUniformLocation(myShader, "iIndex");
locColor = glGetUniformLocation(myShader, "vColorValue");
locFlag = glGetUniformLocation(myShader, "bSomeFlag");
...
...
glUseProgram(myShader);
glUniform1f(locTime, 45.2f);
glUniform1i(locIndex, 42);
glUniform4f(locColor, 1.0f, 0.0f, 0.0f, 1.0f);
glUniform1i(locFlag, GL_FALSE);
```

Zwróćmy uwagę, że do przekazania wartości logicznej użyliśmy całkowitoliczbowej wersji funkcji `glUniform`. Wartości logiczne można również przekazywać jako wartości zmiennoprzecinkowe, gdzie 0.0 oznacza fałsz, a 1.0 — prawdę.

## Tablice uniform

Istnieją również wersje funkcji `glUniform` pobierające jako parametr wskaźnik, który może wskazywać tablicę wartości.

```
void glUniform1fv(GLint location, GLuint count, GLfloat* v);
void glUniform2fv(GLint location, GLuint count, GLfloat* v);
void glUniform3fv(GLint location, GLuint count, GLfloat* v);
void glUniform4fv(GLint location, GLuint count, GLfloat* v);

void glUniform1iv(GLint location, GLuint count, GLint* v);
void glUniform2iv(GLint location, GLuint count, GLint* v);
void glUniform3iv(GLint location, GLuint count, GLint* v);
void glUniform4iv(GLint location, GLuint count, GLint* v);
```

Parametr *count* określa liczbę elementów w tablicy zawierającej *x* składników, gdzie *x* oznacza liczbę na końcu nazwy funkcji. Jeśli np. mamy zmienną `uniform` zawierającą cztery składniki:

```
uniform vec4 vColor;
```

W C i C++ moglibyśmy ją zdefiniować jako tablicę liczb zmiennoprzecinkowych:

```
GLfloat vColor[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
```



Ponieważ jest to pojedyncza tablica czterech wartości, do przekazania jej do shadera wykorzystalibyśmy następującą konstrukcję:

```
glUniform4fv(iColorLocation, 1, vColor);
```

Gdybyśmy jednak mieli w shaderze taką tablicę wartości kolorów:

```
uniform vec4 vColors[2];
```

Wówczas w C++ dane te moglibyśmy zdefiniować i przekazać następująco:

```
GLfloat vColors[2][4] = {{ 1.0f, 1.0f, 1.0f, 1.0f },
                        { 1.0f, 0.0f, 0.0f, 1.0f }};
...
glUniform4fv(iColorLocation, 2, vColors);
```

Najprostsze ustawienie jednej zmiennoprzecinkowej wartości uniform może wyglądać tak:

```
GLfloat fValue = 45.2f;
glUniform1fv(iLocation, 1, &fValue);
```

## Macierze uniform

Na zakończenie pokażemy, jak się ustawia macierz uniform. W shaderach typy macierzowe przechowują tylko wartości zmiennoprzecinkowe, przez co mamy znacznie mniej różnych wersji. Poniższe funkcje ładują odpowiednio macierze 2×2, 3×3 oraz 4×4.

```
glUniformMatrix2fv(GLint location, GLuint count, GLboolean transpose,
                  const GLfloat *m);
glUniformMatrix3fv(GLint location, GLuint count, GLboolean transpose,
                  const GLfloat *m);
glUniformMatrix4fv(GLint location, GLuint count, GLboolean transpose,
                  const GLfloat *m);
```

Zmienna *count* określa liczbę macierzy we wskaźniku *m* (tak, można tworzyć tablice macierzy!). Znacznik logiczny *transpose* zostaje ustawiony na wartość `true`, jeśli macierz jest ustawiona w porządku kolumnowym (ustawienie to jest preferowane w OpenGL). Ustawienie tej wartości na `GL_FALSE` powoduje transpozycję macierzy podczas jej kopiowania do shadera. Może to być przydatne, gdybyśmy korzystali z biblioteki macierzowej stosującej porządek wierszowy macierzy (np. Direct3D).

## Płaski shader

Zobaczmy przykładowy shader wykorzystujący zmienne uniform. W zestawie shaderów standardowych mamy płaski shader, który tylko przekształca geometrię i ustawia jej jakiś pojedynczy kolor. Używane są w nim atrybuty położenia wierzchołków i dwie zmienne uniform — macierz przekształcenia i wartość koloru.

Program FlatShader rysuje obracający się torus i ustawia jego kolor na niebieski. Renderujemy go w trybie szkieletowym za pomocą funkcji `glPolygonMode`, aby było wyraźnie widać, że jest trójwymiarowy. Większość zastosowanego kodu klienckiego OpenGL jest już nam dobrze znana, dlatego nie zamieszczamy tu pełnego kodu programu. Na listingach 6.4 i 6.5 przedstawiony jest pełny kod obu shaderów.

**Listing 6.4.** Shader wierzchołków FlatShader

```
// Shader płaski
// Shader wierzchołków
// Richard S. Wright Jr
// OpenGL. Księga eksperta
#version 330

// Macierz przekształcenia
uniform mat4  .mvpMatrix;

// Dane wejściowe wierzchołków
in vec4 vVertex;

void main(void)
{
    // To wszystko, przekształcamy geometrię
    gl_Position = ..mvpMatrix * vVertex;
}
```

**Listing 6.5.** Shader fragmentów FlatShader

```
// Shader płaski
// Shader fragmentów
// Richard S. Wright Jr
// OpenGL. Księga eksperta
#version 130

// Tworzenie jednolitej geometrii
uniform vec4 vColorValue;

// Wyjście koloru fragmentu
out vec4 vFragColor;

void main(void)
{
    vFragColor = vColorValue;
}
```

W shaderze wierzchołków na listingu 6.4 znajduje się jedna zmienna `uniform` reprezentująca konkatenowaną macierz przekształcenia:

```
uniform mat4  .mvpMatrix;
```

Jedyną czynnością wykonywaną przez ten shader jest przekształcenie wierzchołka przy użyciu macierzy rzutowania model-widok. Jak widać, mnożenie macierzowego typu danych przez wektorowy w języku GLSL jest czymś naturalnym.

```
gl_Position =.mvpMatrix * vVertex;
```

W shaderze fragmentów na listingu 6.5 również znajduje się tylko jedna zmienna `uniform` będąca czteroskładnikową wartością koloru, który zostanie zastosowany na rasteryzowanych fragmentach.

```
uniform vec4 vColorValue;
```

Po stronie klienckiej program wczytuje oba pliki shaderów i w funkcji `SetupRC` tworzy wskaźniki do obu zmiennych `uniform`.

```
GLuint flatShader;
GLint locMP;
GLint locColor;
...
...
flatShader = gltLoadShaderPairWithAttributes("FlatShader.vp", "FlatShader.fp",
                                             1, GLT_ATTRIBUTE_VERTEX, "vVertex");

locMVP = glGetUniformLocation(flatShader, ".mvpMatrix");
locColor = glGetUniformLocation(flatShader, "vColorValue");
```

Na listingu 6.6 zamieszczony został w całości kod funkcji `RenderScene`, która renderuje obracający się torus (przypomnijmy, że tryb wielokąta ustawiliśmy na `GL_LINE`). Po wybraniu płaskiego shadera następuje ustawienie zmiennych `uniform` koloru i macierzy przekształcenia, a następnie jest wywoływana funkcja `Draw` na obiekcie torusa. Efekt tego widać na rysunku 6.3.

**Listing 6.6.** Praktyczne zastosowanie utworzonego płaskiego shadera

```
// Rysowanie sceny
void RenderScene(void)
{
    static CStopWatch rotTimer;

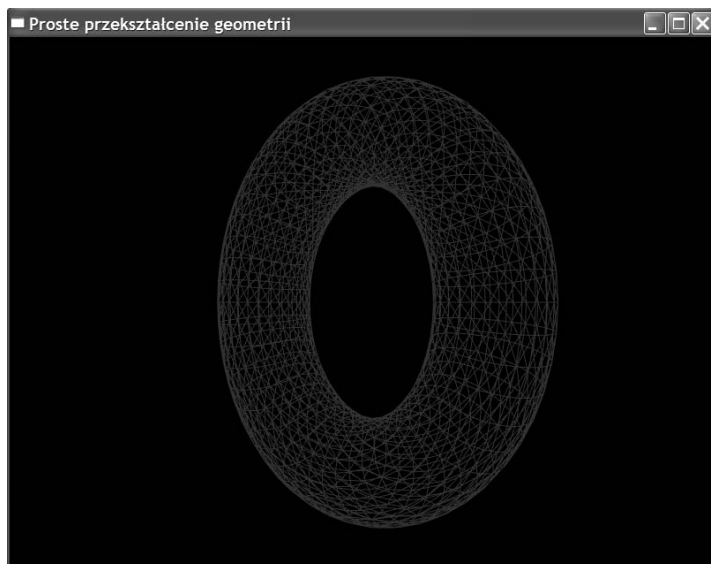
    // Wyczyszczenie okna i bufora głębi
    glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    modelViewMatrix.PushMatrix(viewFrame);
    modelViewMatrix.Rotate(rotTimer.GetElapsedSeconds() * 10.0f, 0.0f,
        ↪ 1.0f, 0.0f);

    GLfloat vColor[] = { 0.1f, 0.1f, 1.f, 1.0f };

    glUseProgram(flatShader);
    glUniform4fv(locColor, 1, vColor);
    glUniformMatrix4fv(locMVP, 1, GL_FALSE,
        ↪ transformPipeline.GetModelViewProjectionMatrix());
    torusBatch.Draw();
}
```

**Rysunek 6.3.**  
Okno programu  
FlatShader



```
modelViewMatrix.PopMatrix();

glutSwapBuffers();
glutPostRedisplay();
}
```

## Funkcje standardowe

Prawie każdy język programowania wysokiego poziomu ma zestaw standardowych funkcji. W językach C i C++ mamy do dyspozycji standardową bibliotekę wykonawczą C, standardowe funkcje wejścia i wyjścia itd. W GLSL również dostępny jest zestaw standardowych funkcji. Większość z nich służy do wykonywania działań matematycznych na wartościach skalarnych i całych wektorach. Niektóre z nich mają bardzo ogólne zastosowanie, jednak część z nich wykorzystuje się w pewnych typowych algorytmach renderowania grafiki. Funkcje zebrane w poniższych tabelach zostały prawie bez zmian skopiowane ze specyfikacji języka GLSL.

### Funkcje trygonometryczne

W tabeli 6.4 znajduje się lista funkcji trygonometrycznych udostępnianych przez język GLSL. Obsługują one typy danych float, vec2, vec3 oraz vec4. Zapis anyFloat w tabeli oznacza, że dana funkcja obsługuje każdy z wymienionych typów.

Tabela 6.4. Funkcje trygonometryczne

Funkcja	Opis
<code>anyFloat radians(anyFloat degrees)</code>	Konwertuje stopnie na radiany
<code>anyFloat degrees(anyFloat radians)</code>	Konwertuje radiany na stopnie
<code>anyFloat sin(anyFloat angle)</code>	Sinus
<code>anyFloat cos(anyFloat angle)</code>	Cosinus
<code>anyFloat tan(anyFloat angle)</code>	Tangens
<code>anyFloat asin(anyFloat x)</code>	Arcus sinus
<code>anyFloat acos(anyFloat x)</code>	Arcus cosinus
<code>anyFloat atan(anyFloat y, anyFloat x)</code>	Arcus tangens $y/x$
<code>anyFloat atan(anyFloat y_over_x)</code>	Arcus tangens $y\_over\_x$
<code>anyFloat sinh(anyFloat x)</code>	Sinus hiperboliczny
<code>anyFloat cosh(anyFloat x)</code>	Cosinus hiperboliczny
<code>anyFloat tanh(anyFloat x)</code>	Tangens hiperboliczny
<code>anyFloat asinh(anyFloat x)</code>	Arcus sinus hiperboliczny
<code>anyFloat acosh(anyFloat x)</code>	Arcus cosinus hiperboliczny
<code>anyFloat atanh(anyFloat x)</code>	Arcus tangens hiperboliczny

## Funkcje wykładnicze

Podobnie jak funkcje trygonometryczne, funkcje wykładnicze działają na zmiennoprzecinkowych typach danych (skalarnych i wektorach). Pełna lista tych funkcji znajduje się w tabeli 6.5.

Tabela 6.5. Funkcje wykładnicze

Funkcja	Opis
<code>anyFloat pow(anyFloat x, anyFloat y)</code>	$x$ podniesiony do potęgi $y$
<code>anyFloat exp(anyFloat x)</code>	Wykładnik naturalny $x$
<code>anyFloat log(anyFloat x)</code>	Logarytm naturalny $x$
<code>anyFloat exp2(anyFloat x)</code>	2 do potęgi $x$
<code>anyFloat log2(anyFloat angle)</code>	Logarytm $x$ o podstawie 2
<code>anyFloat sqrt(anyFloat x)</code>	Pierwiastek kwadratowy z $x$
<code>anyFloat inversesqrt(anyFloat x)</code>	Odwrotność pierwiastka kwadratowego z $x$

## Funkcje geometryczne

W języku GLSL dostępna jest też pewna liczba ogólnych funkcji geometrycznych. Niektóre z nich przyjmują tylko argumenty określonego typu (np. iloczyn wektorowy), a inne — każdy z wektorowych zmiennoprzecinkowych typów danych (`vec2`, `vec3` i `vec4`), które w tabeli 6.6 oznaczamy zbiorczo jako `vec`.

Tabela 6.6. Funkcje geometryczne

Funkcja	Tabela
<code>float length(vec2/vec3/vec4 x)</code>	Zwraca długość wektora <code>x</code>
<code>float distance(vec p0, vec p1)</code>	Zwraca odległość między <code>p0</code> i <code>p1</code>
<code>float dot(vec x, vec y)</code>	Zwraca iloczyn skalarny <code>x</code> i <code>y</code>
<code>vec3 cross(vec3 x, vec3 y)</code>	Zwraca iloczyn wektorowy <code>x</code> i <code>y</code>
<code>vec normalize(vec x)</code>	Zwraca wektor o długości jeden skierowany w tym samym kierunku, co <code>x</code>
<code>vec faceforward(vec N, vec I, vec nRef)</code>	Jeśli <code>dot(Nref, I) &lt; 0</code> , zwraca <code>N</code> , w przeciwnym przypadku zwraca <code>-N</code>
<code>vec reflect(vec I, vec N)</code>	Zwraca kierunek odbicia wektora padającego <code>I</code> oraz orientację płaszczyzny <code>N</code>
<code>vec refract(vec I, vec N, float eta)</code>	Zwraca wektor załamania wektora padającego <code>I</code> , orientację płaszczyzny <code>N</code> oraz współczynnik wskaźników załamania <code>eta</code>

## Funkcje macierzowe

Wiele działań na macierzach wykonuje się przy użyciu zwykłych operatorów matematycznych. W tabeli 6.7 znajduje się wykaz kilku dodatkowych funkcji macierzowych, które mogą być bardzo przydatne. Każda z nich pobiera określony typ argumentów, które zostały wyszczególnione.

## Funkcje porównywania wektorów

Wartości skalarne można porównywać za pomocą standardowych operatorów porównywania (`<`, `<=`, `>`, `>=`, `++` oraz `!=`). Do porównywania wektorów służą funkcje zebrane w tabeli 6.8. Wszystkie zwracają wektory wartości logicznych o takiej samej liczbie wymiarów, jak argumenty.

## Inne często używane funkcje

Na zakończenie w tabeli 6.9 przedstawiamy zbiór różnych funkcji ogólnego przeznaczenia. Wszystkie działają zarówno na skalarnych, jak i wektorowych typach danych oraz mogą takie typy zwracać.

Tabela 6.7. Funkcje macierzowe

Funkcja	Opis
<code>mat matrixCompMult(mat x, mat y)</code>	Mnoży dwie macierze składnik po składniku. To nie jest to samo, co mnożenie macierzy w algebrze liniowej.
<code>mat2 outerProduct(vec2 c, vec2 r)</code> <code>mat3 outerProduct(vec3 c, vec3 r)</code> <code>mat4 outerProduct(vec4 c, vec4 r)</code> <code>mat2x3 outerProduct(vec3 c, vec2 r)</code> <code>mat3x2 outerProduct(vec2 c, vec3 r)</code> <code>mat2x4 outerProduct(vec4 c, vec2 r)</code> <code>mat4x2 outerProduct(vec2 c, vec4 r)</code> <code>mat3x4 outerProduct(vec4 c, vec3 r)</code> <code>mat4x3 outerProduct(vec3 c, vec4 r)</code>	Zwraca macierz będącą iloczynem zewnętrznym dwóch podanych wektorów.
<code>mat2 transpose(mat2 m)</code> <code>mat3 transpose(mat3 m)</code> <code>mat4 transpose(mat4 m)</code> <code>mat2x3 transpose(mat3x2 m)</code> <code>mat3x2 transpose(mat2x3 m)</code> <code>mat2x4 transpose(mat4x2 m)</code> <code>mat4x2 transpose(mat2x4 m)</code> <code>mat3x4 transpose(mat4x3 m)</code> <code>mat4x3 transpose(mat3x4 m)</code>	Transponuje podaną macierz
<code>float determinant(mat2 m)</code> <code>float determinant(mat3 m)</code> <code>float determinant(mat4 m)</code>	Zwraca wyznacznik podanej macierzy
<code>mat2 inverse(mat2 m)</code> <code>mat3 inverse(mat3 m)</code> <code>mat4 inverse(mat4 m)</code>	Zwraca odwróconą wersję podanej macierzy

Tabela 6.8. Funkcje porównywania wektorów

Funkcja	Opis
<code>bvec lessThan(vec x, vec y)</code> <code>bvec lessThan(ivec x, ivec y)</code> <code>bvec lessThan(uvec x, uvec y)</code>	Zwraca wynik porównywania $x < y$ każdej pary składników

Tabela 6.8. Funkcje porównywania wektorów — *ciąg dalszy*

Funkcja	Opis
bvec lessThanEqual(vec x, vec y) bvec lessThanEqual(ivec x, ivec y) bvec lessThanEqual(uvec x, uvec y)	Zwraca wynik porównywania $x \leq y$ każdej pary składników
bvec greaterThan(vec x, vec y) bvec greaterThan(ivec x, ivec y) bvec greaterThan(uvec x, uvec y)	Zwraca wynik porównywania $x > y$ każdej pary składników
bvec greaterThanEqual(vec x, vec y) bvec greaterThanEqual(ivec x, ivec y) bvec greaterThanEqual(uvec x, uvec y)	Zwraca wynik porównywania $x \geq y$ każdej pary składników
bvec equal(vec x, vec y) bvec equal(ivec x, ivec y) bvec equal(uvec x, uvec y) bvec equal(bvec x, bvec y)	Zwraca wynik porównywania $x == y$ każdej pary składników
bvec notEqual(vec x, vec y) bvec notEqual(ivec x, ivec y) bvec notEqual(uvec x, uvec y) bvec notEqual(bvec x, bvec y)	Zwraca wynik porównywania $x != y$ każdej pary składników
bool any(bvec x)	Zwraca wartość <code>true</code> , jeśli którykolwiek składnik <code>x</code> ma wartość <code>true</code>
bool all(bvec x)	Zwraca wartość <code>true</code> , jeśli wszystkie składniki <code>x</code> mają wartość <code>true</code>
bvec not(bvec x)	Zwraca dopełnienie <code>x</code> dla każdego komponentu

Tabela 6.9. Inne często używane funkcje

Funkcja	Opis
anyFloat abs(anyFloat x) anyInt abs(anyInt x)	Zwraca wartość bezwzględną <code>x</code>
anyFloat sign(anyFloat x) anyInt sign(anyInt x)	Zwraca wartość <code>1.0</code> lub <code>-1.0</code> w zależności od znaku <code>x</code>
anyFloat floor(anyFloat x)	Zwraca najmniejszą liczbę całkowitą nie większą od <code>x</code>
anyFloat trunc(anyFloat x)	Zwraca liczbę całkowitą najbliższą, ale nie większą niż wartość bezwzględna <code>x</code>



Tabela 6.9. Inne często używane funkcje — ciąg dalszy

Funkcja	Opis
<code>anyFloat round(anyFloat x)</code>	Zwraca wartość całkowitą najbliższą wartości $x$ . Ułamek 0,5 może zostać zaokrąglony w obie strony, w zależności od implementacji
<code>anyFloat roundEven(anyFloat x)</code>	Zwraca wartość całkowitą najbliższą wartości $x$ . Ułamek 0,5 jest zaokrąglany do najbliższej parzystej liczby całkowitej
<code>anyFloat ceil(anyFloat x)</code>	Zwraca wartość najbliższej liczby całkowitej większej od $x$
<code>anyFloat fract(anyFloat x)</code>	Zwraca część ułamkową wartości $x$
<code>anyFloat mod(anyFloat x, float y)</code> <code>anyFloat mod(anyFloat x, anyFloat y)</code>	Zwraca wartość bezwzględną wyniku działania $x \bmod y$
<code>anyFloat modf(anyFloat x, out anyFloat i)</code>	Zwraca część ułamkową wartości $x$ zapisaną w $i$
<code>anyFloat min(anyFloat x, anyFloat y)</code> <code>anyFloat min(anyFloat x, float y)</code> <code>anyInt min(anyInt x, anyInt y)</code> <code>anyInt min(anyInt x, int y)</code> <code>anyUInt min(anyUInt x, anyUInt y)</code> <code>anyUInt min(anyUInt x, uint y)</code>	Zwraca mniejszą spośród wartości $x$ i $y$
<code>anyFloat max(anyFloat x, anyFloat y)</code> <code>anyFloat max(anyFloat x, float y)</code> <code>anyInt max(anyInt x, anyInt y)</code> <code>anyInt max(anyInt x, int y)</code> <code>anyUInt max(anyUInt x, anyUInt y)</code> <code>anyUInt max(anyUInt x, uint y)</code>	Zwraca większą spośród wartości $x$ i $y$
<code>anyFloat clamp(anyFloat x, anyFloat minVal, anyFloat maxVal)</code> <code>anyFloat clamp(anyFloat x, float minVal, float maxVal);</code> <code>anyInt clamp(anyInt x, anyInt minVal, anyInt maxVal)</code> <code>anyInt clamp(anyInt x, int minVal, int maxVal)</code>	Zwraca wartość $x$ przyciętą do przedziału <code>minVal-maxVal</code>

Tabela 6.9. Inne często używane funkcje — ciąg dalszy

Funkcja	Opis
<pre>anyUint clamp(anyUint x,                anyUint minVal,                anyUint maxVal); anyUint clamp(anyUint x,                uint minVal,                uint maxVal)</pre>	
<pre>anyFloat mix(anyFloat x,               anyFloat y,               anyFloat a) anyFloat mix(anyFloat x,               anyFloat y,               float a)</pre>	<p>Zwraca przejście liniowe między <math>x</math> i <math>y</math>. Wartość <math>a</math> może się zawierać w przedziale od 0 do 1</p>
<pre>anyFloat mix(anyFloat x,               anyFloat y,               anyBool a)</pre>	<p>Zwraca składniki <math>x</math>, gdy <math>a</math> ma wartość <code>false</code>, lub składniki <math>y</math>, gdy <math>a</math> ma wartość <code>true</code></p>
<pre>anyFloat step(anyFloat edge, anyFloat x) anyFloat step(float edge, anyFloat x)</pre>	<p>Zwraca wartość 0.0, jeśli wartość <math>x</math> jest mniejsza od <math>edge</math> lub 1.0 w przeciwnym przypadku</p>
<pre>anyFloat smoothstep(anyFloat edge0,                     anyFloat edge1,                     anyFloat x) anyFloat smoothStep(float edge0,                     float edge1,                     anyFloat x)</pre>	<p>Zwraca wartość 0.0, jeśli <math>x \leq edge0</math>, lub 1.0, jeśli <math>x \geq edge1</math>, oraz płynną interpolację Hermite'a dla argumentów między 0.0 i 1.0</p>
<pre>anyBool isnan(anyFloat x)</pre>	<p>Zwraca <code>true</code>, jeśli <math>x</math> jest <code>Nan</code></p>
<pre>anyBool isinf(anyFloat x)</pre>	<p>Zwraca <code>true</code>, jeśli <math>x</math> jest dodatnią lub ujemną nieskończonością</p>
<pre>anyInt floatBitsToInt(anyFloat x) anyUint floatBitsToUint(anyFloat x)</pre>	<p>Konwertuje wartości zmiennoprzecinkowe na całkowite</p>
<pre>anyFloat intBitsToFloat(anyInt x) anyFloat uintBitsToFloat(anyUint x)</pre>	<p>Konwertuje wartości całkowite na zmiennoprzecinkowe</p>

## Symulowanie światła

Znamy już solidne podstawy języka GLSL, a więc czas na rozpoczęcie pisania bardziej rozbudowanych shaderów. Jedną z fundamentalnych technik grafiki komputerowej jest symulowanie światła. Ponieważ techniki te nie są bardzo skomplikowane, doskonale nadają się do przedstawienia metod programowania shaderów. Symulowanie światła, oświetlenia i właściwości materiałów to tak obszerny temat, że można by im było poświęcić całą osobną książkę. I rzeczywiście takie książki istnieją! W tym rozdziale omówimy tylko podstawowe techniki związane z symulowaniem światła w komputerze oraz pokażemy, jak je implementować w języku GLSL. Techniki te stanowią bazę, na której opierają się techniki bardziej zaawansowane.

### Światło rozproszone

Rodzajem światła najczęściej stosowanym do oświetlenia powierzchni w grafice trójwymiarowej jest tzw. **światło rozproszone** (ang. *diffuse light*). Jest to światło skierowane odbijające się od powierzchni z natężeniem proporcjonalnym do kąta, pod jakim się od niej odbija. Dzięki temu powierzchnia obiektu jest jaśniejsza, gdy światło pada na nią pod kątem prostym, niż wówczas, gdyby padało na nią pod jakimś większym kątem. W wielu modelach światła składowa światła rozproszonego jest używana do tworzenia cieni (lub zmian kolorów) na powierzchni oświetlonych obiektów.

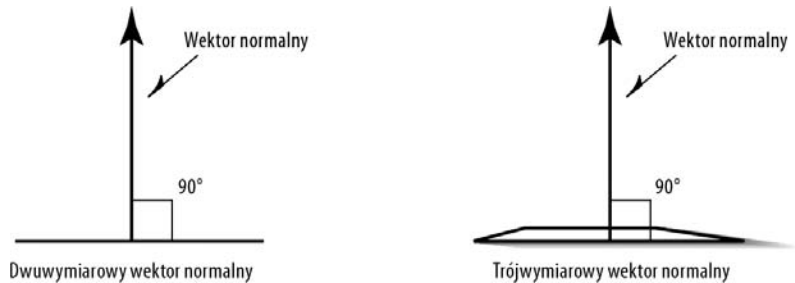
Do określenia natężenia światła w danym wierzchołku potrzebne są dwa wektory. Jeden z nich określa kierunek w stronę źródła światła. W niektórych technikach oświetleniowych dostarczany jest tylko wektor skierowany w stronę źródła światła. Takie światło nazywamy **kierunkowym** (ang. *directional*), ponieważ wszystkie wierzchołki dysponują tym samym wektorem w stronę źródła światła. Wszystko jest dobrze, jeśli źródło światła znajduje się bardzo (lub nieskończenie) daleko od oświetlanych obiektów. Wyobraźmy sobie boisko piłkarskie, na którym jest rozgrywany mecz. Kąt padania promieni słonecznych na jednym końcu boiska niewiele różni się od kąta padania na jego drugim końcu. Gdyby jednak mecz był rozgrywany w nocy, efekt istnienia pojedynczego górnego źródła światła byłby dobrze widoczny podczas przemieszczania się zawodników po boisku. Gdybyśmy do kodu symulującego światło przekazali położenie źródła światła, to aby określić wektor w stronę źródła światła, musielibyśmy w naszym shaderze odjąć przekształcone (współrzędne oka) położenie wierzchołka od położenia źródła światła.

### Normalne do powierzchni

Drugi wektor potrzebny do uzyskania światła rozproszonego (i nie tylko, o czym się niedługo przekonasz) to normalna do powierzchni. **Normalna do powierzchni (wektor normalny)** to linia mająca swój początek na płaszczyźnie, do której jest prostopadła. Nazwa ta może wydawać się

niezwykła, jakby pożyczona z jakiegoś filmu fantastycznego, ale tak naprawdę słowo „normalny” oznacza tu po prostu „prostopadły” do jakiejś realnej lub wyobrażonej płaszczyzny. Wektor to linia skierowana w określonym kierunku, a wektor normalny to linia prostopadła do płaszczyzny. Podsumowując, wektor normalny to linia ustawiona pod kątem 90 stopni do przedniej płaszczyzny naszej figury geometrycznej. Na rysunku 6.4 pokazano przykładowe wektory normalne w dwóch i trzech wymiarach.

**Rysunek 6.4.**  
Dwu- i trójwymiarowy wektor normalny

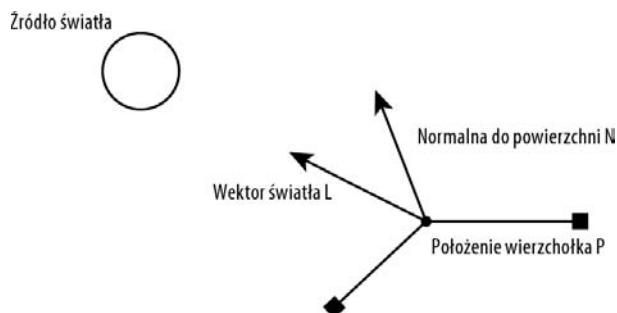


Pewnie się dziwisz, czemu musimy określić wektor normalny dla każdego wierzchołka. Dlaczego nie możemy zdefiniować jednej normalnej dla całego wielokąta i zastosować jej do wszystkich wierzchołków? Możemy, ale nie zawsze chcemy, aby normalna była prostopadła do powierzchni figury. Jak zapewne zauważyłeś, nie wszystkie powierzchnie są płaskie. Można próbować jak najwierniej je odtworzyć za pomocą płaskich wielokątów, ale efekt zawsze będzie niejednorodny lub poszarpany. Aby utworzyć powierzchnię wyglądającą na gładką, można użyć płaskich wielokątów i tak dostosować normalne do powierzchni, aby „ją optycznie wygładziły”. Na przykład w przypadku kuli normalna do powierzchni każdego wierzchołka jest prostopadła do powierzchni samej bryły, a nie do poszczególnych trójkątów, z których ta bryła została złożona.

## Oświetlanie wierzchołków

Na rysunku 6.5 przedstawiono oba wektory, którymi się zajmujemy. Natężenie światła w każdym wierzchołku określa się poprzez obliczenie iloczynu skalarnego wektora do źródła światła i wektora normalnego. Wektory te muszą mieć długość o wartości jeden, ponieważ wynikiem obliczeń może być wartość z przedziału od  $-1.0$  do  $1.0$ . Wartość  $1.0$  otrzymujemy, gdy oba wektory są skierowane w tym samym kierunku, natomiast  $-1.0$  oznacza, że wektory wskazują przeciwne kierunki. Gdy otrzymamy wartość  $0.0$ , wiemy, że wektory są ustawione względem siebie pod kątem 90 stopni. W istocie otrzymana wartość to cosinus kąta między wektorami. Jak można się domyślić, dodatnie wartości oznaczają, że światło pada na wierzchołek. Im większa wartość (czyli im bliższa wartości  $1.0$ ), tym większe natężenie światła; i odwrotnie, im mniejsza wartość (nawet poniżej zera), tym natężenie światła mniejsze.

**Rysunek 6.5.**  
Podstawowe wektory  
światła rozproszonego



Jeśli pomnożymy obliczony iloczyn skalarny przez wartość koloru wierzchołka, otrzymamy wartość koloru ze światłem o odpowiednim natężeniu. Takie płynne cieniowanie wartości kolorów między wierzchołkami czasami nazywane jest **oświetlaniem wierzchołków** (ang. *vertex lighting*) lub **cieniowaniem Gourauda** (ang. *Gouraud shading*). Obliczenie iloczynu skalarnego w języku GLSL jest łatwe. Najczęściej wykorzystuje się coś w rodzaju poniższego wywołania:

```
float intensity = dot(vSurfaceNormal, vLightDirection);
```

## Shader światła rozproszonego

Przeanalizujemy kolejny przykładowy program, o nazwie `DiffuseLight`. Posłuży nam on do zademonstrowania działania prostego shadera światła rozproszonego na niebieskiej kuli. Wykorzystaliśmy w nim także punktowe źródło światła, a więc zobaczymy również, jak uzyskuje się ten rodzaj oświetlenia w shaderze. Oczywiście użycie kierunkowego źródła światła byłoby prostsze, ponieważ już mamy ten wektor, ale to pozostawiamy jako ćwiczenie do samodzielnego wykonania. Na listingu 6.7 znajduje się kod shadera wierzchołków `DiffuseLight.vp`.

**Listing 6.7.** Shader wierzchołków światła rozproszonego

```
// Prosty shader światła rozproszonego
// Shader wierzchołków
// Richard S. Wright Jr
// OpenGL. Księga eksperta
#version 330

// Zmienne wejściowe danych wierzchołków... położenie i normalna
in vec4 vVertex;
in vec3 vNormal;

// Ustawienia dla każdej porcji danych
uniform vec4 diffuseColor;
uniform vec3 vLightPosition;
uniform mat4.mvpMatrix;
uniform mat4.mvMatrix;
uniform mat3.normalMatrix;

// Kolor
smooth out vec4 vVaryingColor;
```

```

void main(void)
{
    // Obliczenie normalnej do powierzchni we współrzędnych oka
    vec3 vEyeNormal = normalMatrix * vNormal;

    // Obliczenie położenia wierzchołka we współrzędnych oka
    vec4 vPosition4 = mvMatrix * vVertex;
    vec3 vPosition3 = vPosition4.xyz / vPosition4.w;

    // Obliczenie wektora do źródła światła
    vec3 vLightDir = normalize(vLightPosition - vPosition3);

    // Obliczenie intensywności światła przy użyciu iloczynu skalarnego
    float diff = max(0.0, dot(vEyeNormal, vLightDir));

    // Mnożenie intensywności przez kolor rozproszenia
    vVaryingColor.rgb = diff * diffuseColor.rgb;
    vVaryingColor.a = diffuseColor.a;

    // Przekształcenie geometrii
    gl_Position =.mvpMatrix * vVertex;
}

```

W shaderze zostały zdefiniowane tylko dwa atrybuty — położenie wierzchołka (`vVertex`) i normalna do powierzchni (`vNormal`). Natomiast zmiennych uniform potrzebowaliśmy aż pięciu:

```

uniform vec4    diffuseColor;
uniform vec3    vLightPosition;
uniform mat4   .mvpMatrix;
uniform mat4    mvMatrix;
uniform mat3    normalMatrix;

```

Zmienna `diffuseColor` przechowuje kolor kuli, `vLightPosition` określa położenie światła we współrzędnych oka, `mpvMatrix` zawiera macierz rzutowania model-widok, `mvMatrix` natomiast reprezentuje macierz model-widok. Wszystko to znamy z korzystania z shaderów standardowych (tyle że od strony klienta). Nowością jest natomiast macierz 3×3 o nazwie `normalMatrix`.

Normalną do powierzchni najczęściej przesyła się jako jeden z atrybutów wierzchołka. Trzeba ją tylko obrócić, aby jej kierunek znalazł się w przestrzeni oka. Nie można jednak w tym celu pomnożyć jej przez macierz model-widok, ponieważ zawiera ona dodatkowo przesunięcie, które przesunęłoby nasz wektor. Problem ten rozwiązujemy poprzez przekazanie zmiennej uniform reprezentującej **macierz normalną** (ang. *normalną matrix*) zawierającą tylko składnik obrotowy macierzy model-widok. Na szczęście w klasie `GLTransformationPipeline` dostępna jest funkcja `GetNormalMatrix`, która zwraca tę potrzebną nam macierz. Dzięki temu uzyskanie kierunku normalnej we współrzędnych oka to kwestia pomnożenia dwóch macierzy:

```
vec3 vEyeNormal = normalMatrix * vNormal;
```

Poza funkcją główną zadeklarowaliśmy płynnie cieniowaną wartość koloru o nazwie `vVaryingColor`.

```
smooth out vec4 vVaryingColor;
```

To jest jedyna — poza przekształconą geometrią — informacja wyjściowa shadera wierzchołków. Shader fragmentów jest banalnie prosty. Przypisujemy w nim przychodzącą wartość do wyjściowego koloru fragmentu.

```
vFragColor = vVaryingColor;
```

Ze względu na fakt, że przesyłamy położenie światła, a nie wektor w stronę źródła światła, położenie wierzchołka musimy przekonwertować na współrzędne oka i odjąć tę wartość od położenia światła.

```
vec4 vPosition4 = mvMatrix * vVertex;
vec3 vPosition3 = vPosition4.xyz / vPosition4.w;
// Obliczenie wektora w stronę źródła światła
vec3 vLightDir = normalize(vLightPosition - vPosition3);
```

Osobna macierz model-widok jest nam w tym shaderze potrzebna dlatego, że współrzędnych oka wierzchołka nie można pomnożyć przez macierz zawierającą rzutowanie. W tym miejscu do gry wkracza współrzędna *w*. Wykonanie tego dzielenia jest ważne na wypadek, gdyby macierz przekształcenia zawierała jakieś informacje skalowania (aby dowiedzieć się, dlaczego jest to lub nie jest ważne dla Ciebie, wróć do rozdziału 4.).

Wektory są po prostu piękne, prawda? Aby otrzymać wektor w stronę światła, odejmujemy od siebie te dwa wektory i normalizujemy wynik. Teraz możemy wykorzystać iloczyn skalarny do obliczenia intensywności światła na wierzchołku. Przy użyciu funkcji `max` języka GLSL ograniczyliśmy zakres natężenia do wartości z przedziału od zera do jeden.

```
float diff = max(0.0, dot(vEyeNormal, vLightDir));
```

Na zakończenie obliczeń światła mnożymy kolor powierzchni przez natężenie światła. W tym przypadku używamy tylko kanałów RGB, a kanał alfa zostawiamy bez zmian.

```
vVaryingColor.rgb = diff * diffuseColor.rgb;
vVaryingColor.a = diffuseColor.a;
```

Listing 6.8 przedstawia funkcje `SetupRC` i `RenderScene` z programu `DiffuseLight`.

**Listing 6.8.** Kod funkcji `SetupRC` i `RenderScene` z programu `DiffuseLight`

---

```
// Ta funkcja wykonuje wszystkie działania związane z inicjalizowaniem w kontekście renderowania.
void SetupRC(void)
{
    // Tło
    glClearColor(0.3f, 0.3f, 0.3f, 1.0f );

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);

    shaderManager.InitializeStockShaders();
    viewFrame.MoveForward(4.0f);
```

```

// Tworzenie kuli
glMakeSphere(sphereBatch, 1.0f, 26, 13);

diffuseLightShader =
↳shaderManager.LoadShaderPairWithAttributes("DiffuseLight.vp",
↳"DiffuseLight.fp", 2, GLT_ATTRIBUTE_VERTEX, "vVertex",
GLT_ATTRIBUTE_NORMAL, "vNormal");

locColor = glGetUniformLocation(diffuseLightShader, "diffuseColor");
locLight = glGetUniformLocation(diffuseLightShader, "vLightPosition");
locMVP = glGetUniformLocation(diffuseLightShader, "mvpMatrix");
locMV = glGetUniformLocation(diffuseLightShader, "mvMatrix");
locNM = glGetUniformLocation(diffuseLightShader, "normalMatrix");
}

// Rysowanie sceny
void RenderScene(void)
{
    static CStopWatch rotTimer;

    // Wyczyszczenie okna i bufora głębi
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    modelViewMatrix.PushMatrix(viewFrame);
    modelViewMatrix.Rotate(rotTimer.GetElapsedSeconds() * 10.0f, 0.0f,
↳1.0f, 0.0f);

    GLfloat vEyeLight[] = { -100.0f, 100.0f, 100.0f };
    GLfloat vDiffuseColor[] = { 0.0f, 0.0f, 1.0f, 1.0f };

    glUseProgram(diffuseLightShader);
    glUniform4fv(locColor, 1, vDiffuseColor);
    glUniform3fv(locLight, 1, vEyeLight);
    glUniformMatrix4fv(locMVP, 1, GL_FALSE,
↳transformPipeline.GetModelViewProjectionMatrix());
    glUniformMatrix4fv(locMV, 1, GL_FALSE,
↳transformPipeline.GetModelViewMatrix());
    glUniformMatrix3fv(locNM, 1, GL_FALSE,
↳transformPipeline.GetNormalMatrix());
    sphereBatch.Draw();

    modelViewMatrix.PopMatrix();

    glutSwapBuffers();
    glutPostRedisplay();
}

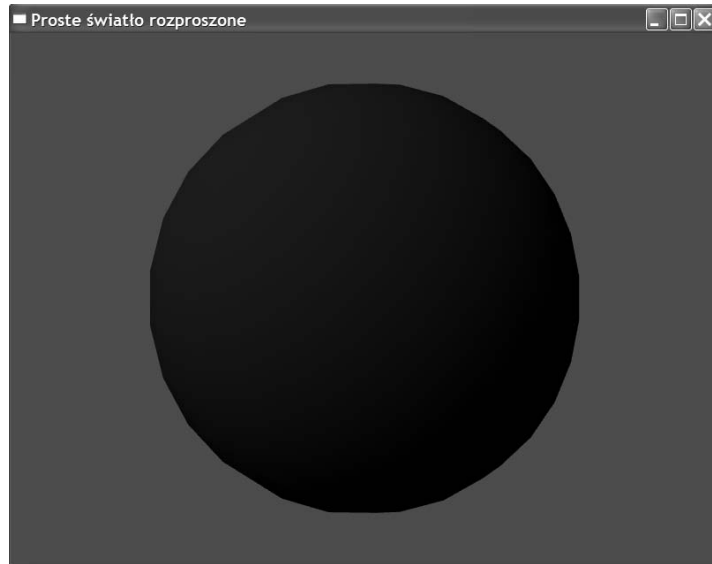
```

Jest to nasze pierwsze poważniejsze spotkanie z niestandardowym shaderem. Jak widać, aby go skonfigurować, w funkcji renderującej potrzebnych było aż pięć wywołań funkcji `glUniform`. Często spotykanym błędem, zwłaszcza popełnianym przez programistów przyzwyczajonych do starego stylu pracy, jest dalsze modyfikowanie jednego ze stosów macierzy po ustawieniu danych



uniform shadera, ale przed wyrenderowaniem geometrii. Pamiętajmy, że funkcje `glUniform` nie kopiuje do shaderów referencji do danych, lecz same dane. Stanowi to okazję do pozbycia się kilku wywołań funkcji dla wartości `uniform`, które nie zmieniają się często. Rysunek 6.6 przedstawia wynik działania naszego programu.

Rysunek 6.6.  
Program DiffuseLight



## Model oświetlenia ADS

Jednym z najczęściej wykorzystywanych modeli oświetlenia, w szczególności przez osoby zaznajomione z aktualnie wycofywanym stałym potokiem funkcji, jest tzw. model oświetlenia ADS. Akronim ten tworzą pierwsze litery angielskich wyrazów *ambient*, *diffuse* oraz *specular*, oznaczających trzy właściwości odbłaskowe materiału pokrywającego obiekty — dla światła otaczającego (ang. *ambient*), rozproszonego (ang. *diffuse*) oraz odbitego zwierciadlanie (ang. *specular*). Tym właściwościom materiału przypisuje się wartości kolorów, przy czym wyższa wartość oznacza wyższy współczynnik odbłasku. Źródła światła również mają te właściwości i im również przypisuje się wartości koloru reprezentujące jasność światła. Zatem o ostatecznym kolorze wierzchołka decyduje wypadkowa tych trzech właściwości materiału i źródła światła.

### Światło otaczające

Światło otaczające (ang. *ambient light*) nie pochodzi z żadnego konkretnego kierunku. Ma ono źródło, ale promienie światła odbijały się już po całej scenie tyle razy, że światło to całkowicie straciło jakikolwiek kierunek. Wszystkie powierzchnie obiektów znajdujących się w świetle otaczającym są oświetlone równomiernie, niezależnie od kierunku. Światło otaczające można traktować jako globalny współczynnik „rozjaśniający” stosowany dla każdego źródła światła.

Aby obliczyć udział źródła światła otaczającego w ostatecznym kolorze wierzchołka, należy przeskalować wartość właściwości odbłaskowej światła otaczającego materiału przez wartość światła otaczającego (należy pomnożyć te dwa wektory przez siebie). W języku GLSL zapisalibyśmy to następująco:

```
uniform vec3 vAmbientMaterial;
uniform vec3 vAmbientLight;
vec3 vAmbientColor = vAmbientMaterial * vAmbientLight;
```

## Światło rozproszone

Światło rozproszone to kierunkowa składowa źródła światła. Było ono w naszym centrum zainteresowania w poprzednim shaderze oświetlenia. Wartość materiału rozpraszającego należy pomnożyć przez wartość światła rozproszonego, tak jak się to robi w przypadku składowych światła otaczającego. Wartość ta jest następnie skalowana przez iloczyn skalarny normalnej do powierzchni i wektora światła (czyli natężenie światła rozproszonego). W języku shaderów można to wyrazić w następujący sposób:

```
uniform vec3 vDiffuseMaterial;
uniform vec3 vDiffuseLight;
float fDotProduct = max(0.0, dot(vNormal, vLightDir));
vec3 vDiffuseColor = vDiffuseMaterial * vDiffuseLight * fDotProduct;
```

Zauważmy, że obliczanie iloczynu skalarnego wektorów umieściliśmy w funkcji GLSL o nazwie `max`. Zrobiliśmy to dlatego, że iloczyn skalarny może mieć wartość ujemną, a przecież nie możemy zastosować ujemnego oświetlenia czy koloru. Dlatego wszystkie wartości poniżej zera zamieniamy na zero.

## Światło odbicia zwierciadlanego

Podobnie jak światło rozproszone, światło odbicia zwierciadlanego to właściwość kierunkowa, ale w odróżnieniu od niego silniej oddziałuje z powierzchnią materiału i oddziaływanie to ma ściśle określony kierunek. Mocno odbite światło najczęściej powoduje wystąpienie na powierzchni oświetlanej jasnej plamy nazywanej odbłyskiem (ang. *specular highlight*). Ze względu na dość precyzyjne ukierunkowanie odbłysk może być niewidoczny dla osoby patrzącej pod określonym kątem. Przykładami źródeł światła tworzących mocne odbłyski są reflektor i słońce, ale oczywiście warunkiem powstania tych odbłysków jest padanie promieni światła na błyszczący przedmiot.

Udział koloru w materiale połyskującym i kolorach oświetlenia skalowany jest przez pewną wartość, której otrzymanie wymaga nieco większej ilości obliczeń niż wykonywane do tej pory. Najpierw musimy znaleźć wektor odbicia światła i odwrócony wektor światła. Następnie oblicza się iloczyn skalarny tych dwóch wektorów i podnosi się go do potęgi wartości „połyskliwości”. Im większa wartość połyskliwości, tym mniejszy odbłysk. Poniżej znajduje się fragment kodu shadera wykonujący te obliczenia:

```
uniform vec3 vSpecularMaterial;
uniform vec3 vSpecularLight;
float shininess = 128.0;
vec3 vReflection = reflect(-vLightDir, vEyeNormal);
```

```
float EyeReflectionAngle = max(0.0, dot(vEyeNormal, vReflection));
fSpec = pow(EyeReflectionAngle, shininess);
vec3 vSpecularColor = vSpecularLight * vSpecularMaterial * fSpec;
```

Parametr połyskliwości można określić jako daną typu `uniform`. Tradycyjnie przypisuje się mu maksymalną wartość 128 (tradycja ta sięga jeszcze czasów stałego potoku). Zastosowanie większych wartości zwykle powoduje powstanie bardzo małych odbłyśków.

## Shader ADS

Ostateczny kolor wierzchołka można zatem, biorąc pod uwagę trzy ostatnie przykłady, obliczyć następująco:

```
vVertexColor = vAmbientColor + vDiffuseColor + vSpecularColor;
```

Implementację opisywanego rodzaju shadera zawiera program `ADSGouraud`. Zastosowaliśmy w nim jednak pewne uproszczenie. Zamiast przekazywać osobno informacje na temat kolorów i natężenia właściwości materiału i światła, przekazaliśmy po jednej wartości koloru dla materiałów światła otaczającego, rozproszonego i odbijanego w sposób zwierciadlany. To tak, jakbyśmy wcześniej pomnożyli właściwość materiału przez kolory światła. Jeśli nie planujesz zmieniać właściwości materiału w każdym wierzchołku, jest to łatwy sposób na optymalizację. W nazwie programu znalazło się słowo „Gouraud”, ponieważ wartości światła obliczamy dla każdego wierzchołka, a następnie stosujemy cieniowanie z interpolacją przestrzeni kolorów między wierzchołkami. Pełny kod shadera wierzchołków znajduje się na listingu 6.9.

**Listing 6.9.** Shader wierzchołków programu `ADSGouraud`

```
// Shader oświetlenia punkowego ADS
// Shader wierzchołków
// Richard S. Wright Jr
// OpenGL. Księga eksperta
#version 130

// Dane wejściowe wierzchołków... położenie i normalna
in vec4 vVertex;
in vec3 vNormal;

// Ustawienia dla porcji danych
uniform vec4 ambientColor;
uniform vec4 diffuseColor;
uniform vec4 specularColor;

uniform vec3 vLightPosition;
uniform mat4.mvpMatrix;
uniform mat4.mvMatrix;
uniform mat3.normalMatrix;

// Kolor do shadera fragmentów
smooth out vec4 vVaryingColor;

void main(void)
{
```

```

// Obliczanie normalnej do powierzchni we współrzędnych oka
vec3 vEyeNormal = normalMatrix * vNormal;

// Obliczenie położenia wierzchołka we współrzędnych oka
vec4 vPosition4 = mvMatrix * vVertex;
vec3 vPosition3 = vPosition4.xyz / vPosition4.w;

// Obliczenie wektora w stronę źródła światła
vec3 vLightDir = normalize(vLightPosition - vPosition3);

// Obliczenie natężenia światła rozproszonego przy użyciu iloczynu skalarnego
float diff = max(0.0, dot(vEyeNormal, vLightDir));

// Pomnożenie natężenia przez kolor rozproszenia, wartość alfa wynosi 1.0
vVaryingColor = diff * diffuseColor;

// Dodanie światła otoczenia
vVaryingColor += ambientColor;

// Światła odbicia zwierciadlanego
vec3 vReflection = normalize(reflect(-vLightDir, vEyeNormal));
float spec = max(0.0, dot(vEyeNormal, vReflection));
if(diff != 0) {
    float fSpec = pow(spec, 128.0);
    vVaryingColor.rgb += vec3(fSpec, fSpec, fSpec);
}

// Przekształcenie geometrii!
gl_Position = mvMatrix * vVertex;
}

```

Nie pokazujemy całego kodu shadera, ponieważ przypisuje on tylko przychodzącą wartość zmiennej `vVaryingColor` do koloru fragmentów:

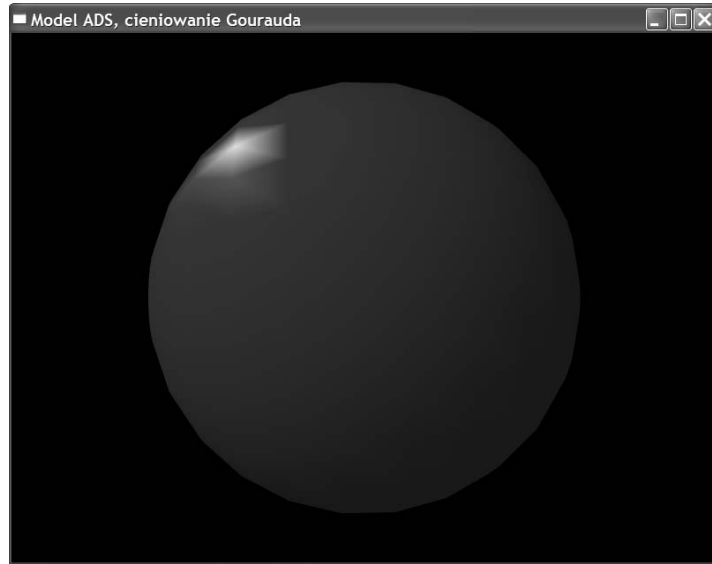
```
vFragColor = vVaryingColor;
```

Każdy trójkąt składa się z trzech wierzchołków i większej liczby wypełniających go fragmentów. Dzięki temu oświetlanie wierzchołków i technika cieniowania Gourauda są bardzo wydajne, ponieważ wszystkie obliczenia dla każdego wierzchołka są wykonywane tylko raz. Rysunek 6.7 przedstawia wynik działania programu `ADSGouraud`.

## Cieniowanie Phonga

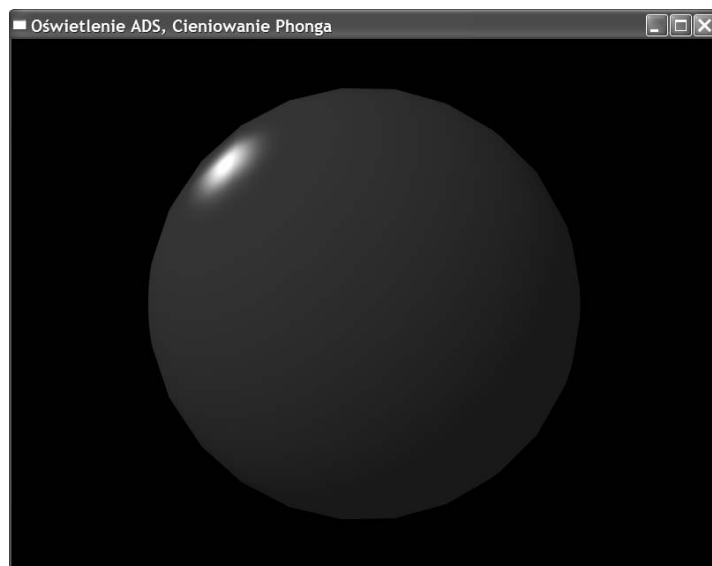
Na rysunku 6.7 widać jedną z największych wad cieniowania Gourauda — odbłysek układa się w kształt gwiazdy. W niektórych przypadkach można by było powiedzieć, że to zamierzony efekt artystyczny. Ale kula widoczna na rysunku w programie się obraca, przez co widać charakterystyczne pasma, które nie wyglądają ładnie i są nieestetyczne. Powodem tych niedoskonałości jest brak ciągłości między trójkątami wynikający z zastosowania liniowej interpolacji

**Rysunek 6.7.**  
Oświetlenie  
wierzchołkowe metodą  
cieniowania Gourauda



w przestrzeni kolorów. Te jasne linie to granice między poszczególnymi trójkątami. Problem ten można próbować wyeliminować poprzez zwiększenie liczby wierzchołków. Jednak lepszym i bardziej efektywnym sposobem jest zastosowanie techniki o nazwie **cieniowanie Phong** (ang. *Phong shading*). Zamiast wartości kolorów będziemy interpolować normalne do powierzchni między wierzchołkami. Efekt zastosowania cieniowania Phong przedstawia rysunek 6.8, na którym widać okno programu ADSPhong (rysunki 6.7 i 6.8 znajdują się również na tablicy 5 w kolorowej wkładce).

**Rysunek 6.8.**  
Oświetlenie pikselowe  
(cieniowanie Phong)



Oczywiście nie ma nic za darmo. W tej technice musimy więcej popracować w shaderze fragmentów, który będzie wywoływany o wiele częściej niż shader wierzchołków. Podstawowy kod niczym nie różni się od kodu programu ADSGouraud. Natomiast duże różnice są w kodzie shaderów. Na listingu 6.10 znajduje się kod nowego shadera wierzchołków.

**Listing 6.10.** Shader wierzchołków ADSPhong

```
// Shader oświetlenia punktowego ADS
// Shader wierzchołków
// Richard S. Wright Jr
// OpenGL. Księga eksperta
#version 330

// Dane wejściowe wierzchołków... położenie i normalna
in vec4 vVertex;
in vec3 vNormal;

uniform mat4  .mvpMatrix;
uniform mat4   mvMatrix;
uniform mat3   normalMatrix;
uniform vec3   vLightPosition;

// Kolor do shadera fragmentów
smooth out vec3 vVaryingNormal;
smooth out vec3 vVaryingLightDir;

void main(void)
{
    // Obliczanie normalnej do powierzchni we współrzędnych oka
    vVaryingNormal = normalMatrix * vNormal;

    // Obliczenie położenia wierzchołka we współrzędnych oka
    vec4 vPosition4 = mvMatrix * vVertex;
    vec3 vPosition3 = vPosition4.xyz / vPosition4.w;

    // Obliczenie wektora wskazującego kierunek w stronę źródła światła
    vVaryingLightDir = normalize(vLightPosition - vPosition3);

    // Przekształcenie geometrii!
    gl_Position =.mvpMatrix * vVertex;
}
```

We wszystkich obliczeniach oświetlenia wykorzystywana jest normalna do powierzchni i wektor kierunku światła. Wektory te przekazujemy zamiast obliczonych wartości kolorów wierzchołków (po jednej dla każdego):

```
smooth out vec3 vVaryingNormal;
smooth out vec3 vVaryingLightDir;
```

Teraz shader fragmentów ma znacznie więcej pracy, co widać na listingu 6.11.

Listing 6.11. Shader fragmentów programu ADSPhong

```

// Shader oświetlenia punktowego ADS
// Shader fragmentów
// Richard S. Wright Jr
// OpenGL. Księga eksperta
#version 330

out vec4 vFragColor;

uniform vec4    ambientColor;
uniform vec4    diffuseColor;
uniform vec4    specularColor;

smooth in vec3  vVaryingNormal;
smooth in vec3  vVaryingLightDir;

void main(void)
{
    // Obliczenie natężenia składowej światła rozproszonego poprzez obliczenie iloczynu skalarnego wektorów
    float diff = max(0.0, dot(normalize(vVaryingNormal),
    ↪normalize(vVaryingLightDir)));

    // Mnożenie natężenia przez kolor rozproszony, alfa ma wartość 1.0
    vFragColor = diff * diffuseColor;

    // Dodanie składowej światła otaczającego
    vFragColor += ambientColor;

    // Światło odbite zwierciadlanie
    vec3 vReflection = normalize(reflect(-normalize(vVaryingLightDir),
    ↪normalize(vVaryingNormal)));
    float spec = max(0.0, dot(normalize(vVaryingNormal), vReflection));
    if(diff != 0) {
        float fSpec = pow(spec, 128.0);
        vFragColor.rgb += vec3(fSpec, fSpec, fSpec);
    }
}

```

Przy obecnym rozwoju techniki wybór takich zaawansowanych metod wysokiej jakości renderowania jest często uzasadniony. Poprawa jakości obrazu jest znaczna, a utrata wydajności często zaniechywana. Jednak w niektórych sytuacjach, np. przy programowaniu mało wydajnego sprzętu (takiego jak układ wbudowany) lub dużym obciążeniu sceny innymi wysokiej jakości algorytmami, najlepszym wyborem może być cieniowanie Gourauda. Ogólna zasada optymalizacji działania shaderów głosi, aby jak najwięcej zadań wykonywać w shaderze wierzchołków, a jak najmniej w shaderze fragmentów. Chyba już wiadomo dlaczego.

## Korzystanie z tekstur

Pobranie tekstury do shadera jest bardzo łatwe. Najpierw do shadera wierzchołków przekazywane są współrzędne tekstury jako atrybuty. Następnie w shaderze fragmentów interpoluje się je płynnie między wierzchołkami. Później shader ten wykorzystuje interpolowane współrzędne do **próbkowania** (ang. *sample*) tekstury. Obiekt tekstury związany z shaderem jest już przygotowany do ewentualnego mipmapowania, ma ustawione tryby filtrowania i zawijania itd. Poddany próbkowaniu i filtrowaniu kolor tekstury wraca w postaci wartości koloru RGBA, którą można zapisać bezpośrednio we fragmencie lub połączyć z innymi obliczeniami kolorów. Pobieraniem i zwracaniem tekstur przy użyciu języka GLSL zajmiemy się bardziej szczegółowo w następnym rozdziale. Tutaj pokażemy tylko podstawowe techniki, abyśmy mogli kontynuować pracę.

### Nic, tylko teksele

Działanie najprostszego możliwego shadera wykorzystującego teksturę przedstawimy na przykładzie programu TexturedTriangle. Jego zadanie polega tylko na narysowaniu trójkąta i pokryciu go teksturą, tak jak widać na rysunku 6.9.

**Rysunek 6.9.**  
Program rysujący trójkąt z nałożoną teksturą



Kod kliencki w C/C++ renderujący trójkąt jest bardzo prosty. Także samo nałożenie tekstury na trójkąt nie jest dla nas niczym nowym, ponieważ robiliśmy to już przy użyciu shaderów standardowych. Na listingu 6.12 przedstawiamy kod shadera wierzchołków odbierającego atrybuty wierzchołków.



**Listing 6.12.** Shader wierzchołków programu *TexturedTriangle*

```
// Shader TexturedIdentity
// Shader wierzchołków
// Richard S. Wright Jr
// OpenGL SuperBible
#version 330

in vec4 vVertex;
in vec2 vTexCoords;

smooth out vec2 vVaryingTexCoords;

void main(void)
{
    vVaryingTexCoords = vTexCoords;
    gl_Position = vVertex;
}
```

Najważniejszymi elementami tego shadera są wejściowy atrybut wierzchołka o nazwie `vTexCoords` zawierający współrzędne tekstury `s` i `t` dla wierzchołka oraz zmienna wyjściowa `vVaryingTexCoords`. To wszystko, czego potrzeba do interpolowania współrzędnych tekstury na powierzchni naszego trójkąta.

Kod shadera fragmentów, przedstawiony na listingu 6.13, również jest krótki i zawiera coś nowego.

**Listing 6.13.** Shader fragmentów programu *TexturedTriangle*

```
// Shader TexturedIdentity
// Shader fragmentów
// Richard S. Wright Jr
// OpenGL. Księga eksperta
#version 330

uniform sampler2D colorMap;

out vec4 vFragColor;
smooth in vec2 vVaryingTexCoords;

void main(void)
{
    vFragColor = texture(colorMap, vVaryingTexCoords.st);
}
```

Na początku programu został użyty nowy typ zmiennej o nazwie `sampler2D`:

```
uniform sampler2D colorMap;
```

Typ `sampler` to liczba całkowita (jej wartość ustawia się za pomocą funkcji `glUniform1i`) reprezentująca jednostkę tekstury, z którą związana jest tekstura mająca zostać poddana próbkowa-

niu. Przyrostek 2D oznacza, że używana jest tekstura dwuwymiarowa. Można również korzystać z wersji 1D, 3D i innych, których szczegółowy opis znajduje się w następnym rozdziale. W rozdziale 5. opisaliśmy obiekty tekstur pozwalające zarządzać dowolną liczbą stanów tekstur, a do wybierania tych obiektów służyła nam funkcja `glBindTexture`. We wszystkich tych przypadkach wykonywaliśmy wiązanie z domyślną jednostką tekstury o numerze 0. Jednostek takich jest jednak więcej i z każdą z nich można łączyć inny obiekt tekstury. Możliwość korzystania z wielu tekstur jednocześnie pozwala uzyskać mnóstwo ciekawych efektów, ale więcej informacji na ten temat znajduje się w następnym rozdziale.

Ustawienie zmiennej `sampler`, która jest typu `uniform`, i wyrenderowanie trójkąta w kodzie klienckim jest bardzo proste.

```
glUseProgram(myTexturedIdentityShader);
glBindTexture(GL_TEXTURE_2D, textureID);
GLint iTextureUniform = glGetUniformLocation(myTexturedIdentityShader,
                                             "colorMap");

glUniform1i(iTextureUniform, 0);

triangleBatch.Draw();
```

W shaderze wywołujemy wbudowaną funkcję mapowania tekstur o nazwie `texture`. Robimy to w celu wykonania próbkowania naszej tekstury przy użyciu interpolowanych współrzędnych tekstury i przypisania wartości koloru bezpośrednio do koloru fragmentów.

```
vFragColor = texture(colorMap, vVaryingTexCoords.st);
```

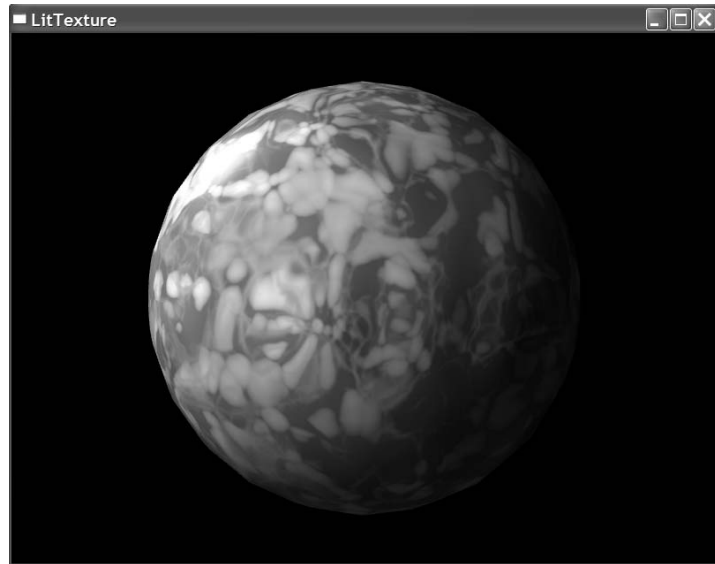
## Oświetlanie tekseleli

Wiemy już, jak próbować tekstury, a więc możemy spróbować zrobić coś ciekawszego, np. dodać teksturę do shadera `ADSPhong`. We wszystkich shaderach oświetlenia postępowaliśmy według jednego schematu — mnożyliśmy podstawowe wartości kolorów przez natężenie światła dla każdego wierzchołka lub każdego piksela. Zmodyfikowany shader `ADSPhong`, który nazwiemy `ADSTexture`, próbkuje teksturę, a następnie mnoży jej wartości kolorów przez natężenie światła. Wynik tego działania przedstawia rysunek 6.10, na którym widać okno programu `LitTexture`. Zwróć szczególną uwagę na jasny biały odbłask po lewej stronie górnej części kuli.

Ten biały rozbłysk przypomina nam o jednej ważnej rzeczy, którą musimy brać pod uwagę przy oświetlaniu teksturowanych powierzchni. Suma światła otaczającego i rozproszonego może dać światło białe, które w przestrzeni kolorów reprezentują same jedynki. Wynikiem pomnożenia koloru tekstury przez kolor biały są oryginalne, w żaden sposób niezmienione wartości kolorów tekseleli. Oznacza to, że nie da się pomnożyć koloru tekstury przez prawidłową wartość światła, aby uzyskać biały rozbłysk. Przynajmniej z założenia tak *powinno* być.

W rzeczywistości wyniki obliczeń światła, także rozbłysku, wykraczają nieco ponad 1.0 dla każdego kanału koloru. Oznacza to, że *istnieje* możliwość przesylenia kolorów i uzyskania białego

**Rysunek 6.10.**  
Połączenie światła  
z teksturą w programie  
LitTexture



rozbłysku. Prawidłowo jednak powinno się pomnożyć sumę natężeń światła otaczającego i rozproszonego przez kolor tekstury, a następnie dodać składową światła odbitego zwierciadlanie. Na listingu 6.14 znajduje się odpowiednio zmodyfikowana wersja shadera ADSPhong.

**Listing 6.14.** Shader fragmentów programu ADSTexture

```
// Shader oświetlenia punktowego ADS
// Shader fragmentów
// Richard S. Wright Jr
// OpenGL. Księga eksperta
#version 330

out vec4 vFragColor;

uniform vec4    ambientColor;
uniform vec4    diffuseColor;
uniform vec4    specularColor;
uniform sampler2D colorMap;

smooth in vec3 vVaryingNormal;
smooth in vec3 vVaryingLightDir;
smooth in vec2 vTexCoords;

void main(void)
{
    // Obliczenie natężenia składowej światła rozproszonego poprzez obliczenie iloczynu skalarnego wektorów
    float diff = max(0.0, dot(normalize(vVaryingNormal),
    ↪normalize(vVaryingLightDir)));

    // Mnożenie natężenia przez kolor rozproszony, alfa ma wartość 1.0
    vFragColor = diff * diffuseColor;
```

```

// Dodanie składowej światła otaczającego
vFragColor += ambientColor;

// Dodanie tekstury
vFragColor *= texture(colorMap, vTexCoords);

// Światło odbite zwierciadlanie
vec3 vReflection = normalize(reflect(-normalize(vVaryingLightDir),
↳normalize(vVaryingNormal)));
float spec = max(0.0, dot(normalize(vVaryingNormal), vReflection));
if(diff != 0) {
    float fSpec = pow(spec, 128.0);
    vFragColor.rgb += vec3(fSpec, fSpec, fSpec);
}
}

```

## Anulowanie przetwarzania fragmentów

Shadery fragmentów umożliwiają anulowanie przetwarzania i w konsekwencji zapisywania wartości kolorów fragmentów (a także wartości głębi i szablonów). Do zatrzymania działania shadera fragmentów służy instrukcja `discard`. Często używa się jej do wykonywania **testów alfa**. Typowa operacja mieszania składa się z odczytywania danych z bufora kolorów, wykonania przynajmniej dwóch działań mnożenia, zsumowania kolorów oraz zapisania wartości z powrotem w buforze kolorów. Jeśli kanał alfa ma wartość zero lub bardzo bliską zero, fragmentów praktycznie nie widać. Co gorsza, fragmenty te tworzą w buforze głębi niewidoczny wzór, który może zakłócić wynik testowania głębi. Testowanie alfa ma na celu znalezienie wszystkich wartości poniżej jakiegoś określonego progu i anulowanie rysowania wszystkich fragmentów, dla których kanał alfa ma taką właśnie wartość. Poniżej znajduje się przykładowy kod wyszukujący wartości alfa mniejsze od 0.1:

```

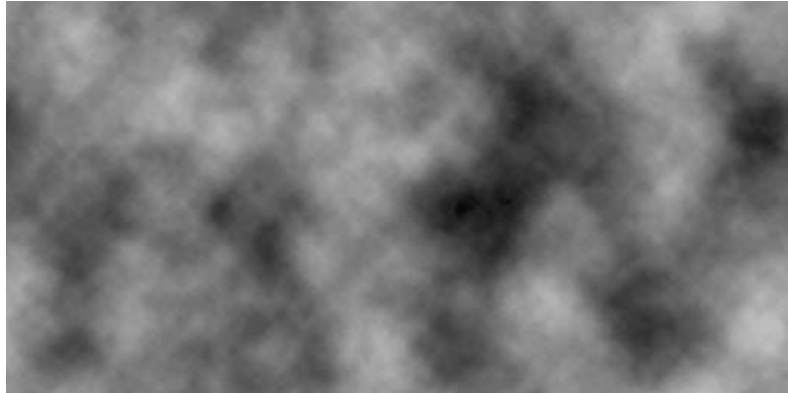
if(vColorValue.a < 0.1f)
discard;

```

Możliwość tę można wykorzystać do uzyskania ciekawego efektu, jakim jest shader erozyjny. Shader erozyjny daje złudzenie, że obiekty geometryczne w miarę upływu czasu ulegają erozji. Dzięki instrukcji `discard` można sterować wyświetlaniem fragmentów z pikselową precyzją. Przykład opisywanego efektu przedstawia program `Dissolve`. Najpierw zdobyliśmy teksturę o odpowiednim wzorze przypominającym chmury. Teksturę taką można łatwo wykonać przy użyciu większości programów do obróbki grafiki. Tekstura, z której my skorzystaliśmy, jest widoczna na rysunku 6.11.

W kodzie klienckim utworzyliśmy czasową zmienną `uniform` zmieniającą wartości w zakresie od 1.0 do 0.0 w czasie 10 sekund. Naszym celem jest to, aby nasz zielony torus w ciągu tych 10 sekund uległ całkowitemu rozkładowi. W tym celu próbujemy teksturę chmury i porównujemy jeden ze składników koloru z naszą zmienną odliczania, anulując rysowanie wszystkich fragmentów, dla których wartość koloru jest mniejsza od ustalonej minimalnej wartości. Kod źródłowy tego shadera fragmentów przedstawia listing 6.15.

**Rysunek 6.11.**  
 Tekstura z chmurą użyta  
 do zademonstrowania  
 efektu erozji



**Listing 6.15.** Shader fragmentów programu Dissolve

```

// Shader oświetlenia punktowego ADS
// Shader fragmentów
// Richard S. Wright Jr
// OpenGL. Księga eksperta
#version 330

out vec4 vFragColor;

uniform vec4    ambientColor;
uniform vec4    diffuseColor;
uniform vec4    specularColor;
uniform sampler2D cloudTexture;
uniform float    dissolveFactor;

smooth in vec3 vVaryingNormal;
smooth in vec3 vVaryingLightDir;
smooth in vec2 vVaryingTexCoord;

void main(void)
{
    vec4 vCloudSample = texture(cloudTexture, vVaryingTexCoord);

    if(vCloudSample.r < dissolveFactor)
        discard;

    // Obliczenie natężenia składowej światła rozproszonego poprzez obliczenie iloczynu skalarnego wektorów
    float diff = max(0.0, dot(normalize(vVaryingNormal),
normalize(vVaryingLightDir)));

    // Mnożenie natężenia przez kolor rozproszony, alfa ma wartość 1.0
    vFragColor = diff * diffuseColor;

    // Dodanie składowej światła otaczającego
    vFragColor += ambientColor;

```

```
// Światło odbite zwierciadlanie
vec3 vReflection = normalize(reflect(-normalize(vVaryingLightDir),
↳normalize(vVaryingNormal)));
float spec = max(0.0, dot(normalize(vVaryingNormal), vReflection));
if(diff != 0) {
    float fSpec = pow(spec, 128.0);
    vFragColor.rgb += vec3(fSpec, fSpec, fSpec);
}
}
```

Jest to kolejna modyfikacja shadera fragmentów programu ADSPhong, do którego dodaliśmy efekt rozkładania się obiektu. Najpierw utworzyliśmy zmienne uniform do przechowywania samplera teksturowego i zegara.

```
uniform sampler2D    cloudTexture;
uniform float        dissolveFactor;
```

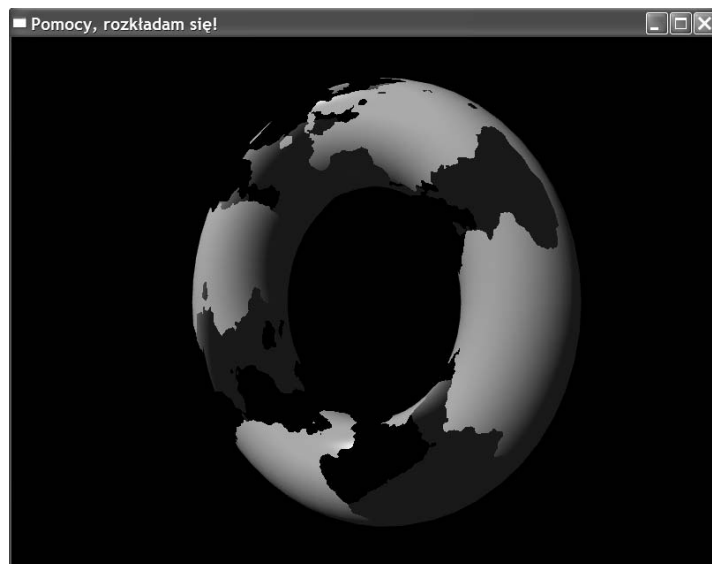
Następnie próbujemy naszą teksturę i sprawdzamy, czy wartość koloru czerwonego (biorąc pod uwagę, że obraz jest w skali szarości, wybór składowej koloru nie ma większego znaczenia) jest mniejsza od ustalonej wartości. Jeśli tak, nie rysujemy takiego fragmentu.

```
vec4 vCloudSample = texture(cloudTexture, vVaryingTexCoord);

if(vCloudSample.r < dissolveFactor)
    discard;
```

Należy również zauważyć, że działania te wykonujemy we wczesnej fazie działania shadera. Nie ma przecież sensu wykonywać czasochłonnych obliczeń pikselowych, jeśli fragment i tak nie zostanie narysowany. Rysunek 6.12 przedstawia jedną klatkę z naszej animacji.

**Rysunek 6.12.**  
Okno programu  
Dissolve



## Tekstutowanie w stylu kreskówkowym — teksele w roli światła

We wszystkich przykładach mapowania tekstur w tym i poprzednim rozdziale używaliśmy tekstur dwuwymiarowych. Są one najprostsze i najłatwiej zrozumieć sposób ich używania. Większość osób intuicyjnie rozumie proces nakładania dwuwymiarowego obrazu na płaszczyznę dwu- lub trójwymiarowego obiektu. Teraz jednak przedstawimy przykład odwzorowywania tekstury jednowymiarowej. Technika ta często wykorzystywana jest w grach komputerowych do tworzenia obrazów cieniowanych w sposób podobny do stosowanego w kreskówkach. Taki rodzaj cieniowania często nazywany jest **cieniowaniem kreskówkowym** (ang. *toon shading* albo *cel shading*). Sposób ten polega na wykorzystaniu jednowymiarowych tekstur jako tabeli wyszukiwania kolorów do wypełnienia obiektów jednolitym kolorem (w trybie `GL_NEAREST`).

Podstawą tej metody jest wykorzystanie natężenia światła rozproszonego (iloczyn skalarny normalnej do powierzchni przestrzeni oka i wektora światła padającego) jako współrzędnej określającej lokalizację koloru w jednowymiarowej teksturze stanowiącej tabelę kolorów o różnym poziomie jasności (ustawionych od najciemniejszego do najjaśniejszego). Na rysunku 6.13 przedstawiona jest tekstura jednowymiarowa składająca się z czterech czerwonych tekseli (zdefiniowanych jako składowe koloru RGB typu `unsigned byte`).

Rysunek 6.13.  
Jednowymiarowa  
tabela wyszukiwania  
kolorów



Przypomnijmy, że wartość iloczynu skalarnego światła rozproszonego mieści się w granicach od 0.0, co oznacza brak natężenia, do 1.0, co oznacza maksymalne natężenie. Ten zakres dobrze pasuje do zakresu współrzędnych jednowymiarowej tekstury. Załadowanie takiej tekstury jest bardzo łatwe, co widać poniżej:

```
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_1D, texture);
GLubyte textureData[4][3] = { 32,  0,  0,
                              64,  0,  0,
                              128, 0,  0,
                              255, 0,  0};

glTexImage1D(GL_TEXTURE_1D, 0, GL_RGB, 4, 0, GL_RGB,
             GL_UNSIGNED_BYTE, textureData);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
```

Kod ten pochodzi z programu ToonShader wyświetlającego obracający się torus renderowany techniką cieniowania kreskówkowego. Mimo iż klasa GLTriangleBatch użyta do utworzenia torusa dostarcza zestaw dwuwymiarowych współrzędnych teksturowych, my je w naszym shaderze wierzchołków ignorujemy, co widać na listingu 6.16.

**Listing 6.16.** Shader wierzchołków programu ToonShader

---

```

// Shader cieniowania kreskówkowego
// Shader wierzchołków
// Richard S. Wright Jr
// OpenGL. Księga eksperta
#version 330

// Dane wejściowe wierzchołków... położenie i normalna
in vec4 vVertex;
in vec3 vNormal;

smooth out float textureCoordinate;

uniform vec3    vLightPosition;
uniform mat4   .mvpMatrix;
uniform mat4    mvMatrix;
uniform mat3    normalMatrix;

void main(void)
{
    // Obliczanie normalnej do powierzchni we współrzędnych oka
    vec3 vEyeNormal = normalMatrix * vNormal;

    // Obliczenie położenia wierzchołka we współrzędnych oka
    vec4 vPosition4 = mvMatrix * vVertex;
    vec3 vPosition3 = vPosition4.xyz / vPosition4.w;

    // Obliczenie wektora wskazującego kierunek w stronę źródła światła
    vec3 vLightDir = normalize(vLightPosition - vPosition3);

    // Obliczenie natężenia światła rozproszonego przy użyciu iloczynu skalarnego
    textureCoordinate = max(0.0, dot(vEyeNormal, vLightDir));

    // Przekształcenie geometrii!
    gl_Position = mvMatrix * vVertex;
}

```

---

Oprócz położenia obiektów po przekształceniu shader ten zwraca jeszcze tylko interpolowaną współrzędną teksturową `textureCoordinate` zadeklarowaną jako typu `float`. Obliczenia składowej światła rozproszonego są wykonywane w sposób identyczny, jak w programie `DiffuseLight`.

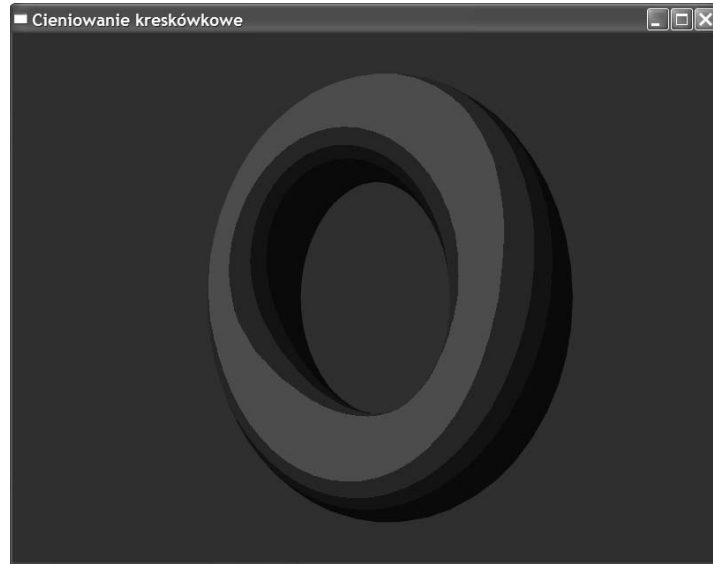
Shader fragmentów próbuje naszą jednowymiarową teksturę i zapisuje wartość we fragmencie w buforze obrazu.

```
vFragColor = texture(colorTable, textureCoordinate);
```



Wynik działania tego programu przedstawia rysunek 6.14. Kolorowy obraz tekstury jednowymiarowej i poniższy rysunek można znaleźć także w kolorowej wkładce na tablicy 6.

**Rysunek 6.14.**  
Torus cieniowany  
techniką cieniowania  
kreskówkowego



## Podsumowanie

W tym rozdziale wydostaliśmy się z ograniczeń standardowych shaderów, których używaliśmy w pięciu początkowych rozdziałach. Klasy typu `GLBatch` z biblioteki `GLTools` ułatwiają przesyłanie najbardziej typowych atrybutów. Teraz wiemy, jak podczepić do tych klas własne atrybuty shaderów. Czytelnicy zobaczyli, jakie są podobieństwa między językami C/C++ i GLSL, poznali funkcje standardowe GLSL oraz nauczyli się pisać w nim własne funkcje. Poznali też dwa modele oświetlenia i nauczyli się je implementować we własnych shaderach, poznali wady i zalety wykonywania skomplikowanych obliczeń w shaderze fragmentów i wierzchołków. Poruszony też został temat pozyskiwania danych teksturowych w shaderach.

Czytelnicy dowiedzieli się, jak odwzorowuje się tekstury dwuwymiarowe na powierzchni obiektów, oraz nauczyli się, jak można wykorzystywać tekstury jako „dane”, traktując je jako tabele wyszukiwania wykorzystywane do eliminowania elementów geometrii za pomocą instrukcji `discard` oraz jako jednowymiarowe tabele kolorów do implementowania cieniowania kreskówkowego.

W tym rozdziale przedstawiono tylko niewielką część możliwości oferowanych przez język GLSL. W dalszych rozdziałach można będzie poznać go znacznie dokładniej i nauczyć się stosować wiele innych ciekawych efektów graficznych, rozszerzając przy tym swoją wiedzę na temat API OpenGL. Po co więc zwlekać? Masz już wystarczającą wiedzę, aby zacząć eksperymentować na własną rękę. Możesz zacząć od modyfikowania przykładowych programów albo wymyślić własne!

# Skorowidz

.NET, 538  
2D, 41  
3D, 41  
3DS Max, 365

## A

abs(), 269  
acos(), 266  
acosh(), 266  
Add/Existing Framework, 87  
Add/Existing Item, 83  
AddTriangle(), 167  
ADS, 278  
    shader, 280  
ADSGouraud, 280  
ADSPhong, 283, 284, 287  
ADSTexture, 287, 288  
    shader fragmentów, 288  
AGL, 560  
aktorzy, 185, 186  
    dodawanie aktorów, 191  
aktualizacja tekstur, 210  
alfa, 98  
algorytm grupowania w stada, 511  
algorytm malarza, 126  
algorytm usuwania płaszczyzn tylnych, 424  
algorytmy antyaliasingu, 143  
algorytmy rekurencyjne, 510  
aliasing, 379  
all(), 269, 453  
alokacja obiektu VBO, 479  
alpha-to-coverage, 391  
ambient light, 278  
AMD\_, 66  
amplification, 417  
animacja, 103  
Anisotropic, 236  
antialiasing, 140  
anulowanie przetwarzania fragmentów, 289  
any(), 269, 453  
API, 52  
API Carbon, 561  
API OpenGL, 62, 74  
API WGL, 537  
API Windows, 532  
aplikacje Cocoa, 86, 561  
aplikacje dla iPhone'a, 629  
    bufor głębi, 638  
    GLTools, 632  
    język C++, 631  
    komunikaty dotykowe, 640  
    orientacja urządzenia, 638  
    projekt aplikacji, 629  
    renderowanie OpenGL ES, 635  
    SphereWorld, 633  
    tekstury, 636  
    tryb panoramiczny, 638  
    UIView, 639  
aplikacje konsolowe Win32, 81  
aplikacje OpenGL w systemie Linux, 587, 602  
ARB, 23, 64, 535  
ARB\_, 66  
ARB\_uniform\_buffer\_object, 461  
Architecture Review Board, 64, 535  
architektura klient-serwer, 107  
architektura shadera, 243  
ASCII, 41  
asin(), 266  
asinh(), 266  
asynchroniczne wywołania funkcji glReadPixels, 330  
atan(), 266  
atanh(), 266  
ATI\_, 66  
atrybuty, 109, 113  
    GLShaderManager, 113  
    GLSL, 249  
    GLT\_ATTRIBUTE\_COLOR, 114  
    GLT\_ATTRIBUTE\_NORMAL, 114, 115  
    GLT\_ATTRIBUTE\_TEXTURE0, 115  
    GLT\_ATTRIBUTE\_VERTEX, 114, 115  
    identyfikatory atrybutów, 113  
atrybuty formatów pikseli, 541  
    Cocoa, 575

atrybuty konfiguracji EGL, 621  
 atrybuty przeplatane, 480  
 attribute, 109  
 automatyczne pobieranie danych, 495  
 AUX, 72

## B

back face culling, 127  
 bajt bez znaku, 75  
 Begin(), 100, 126, 221  
 BeginMesh(), 167  
 biblioteka
 

- AUX, 72
- GLEW, 73, 537, 573
- GLTools, 65, 73
- GLUT, 72, 93, 532
- IRIS GL, 63
- Math3D, 111
- OpenGL, 31, 38, 48, 62

 Bit-Level-Image-Transfer, 347  
 bitmapy, 196  
 blending, 47, 71  
 blending equation, 136  
 blit, 347  
 BLOCK, 44  
 block\_redux, 550  
 Block-Transfer, 347  
 blok domyślny, 454  
 blok interfejsu, 421  
 blok zmiennych jednorodnych, 454, 455
 

- indeksy składowe, 456
- informacje o składowych, 456
- określanie wiązań, 463
- pobieranie indeksów składowych, 458, 460
- przypisanie punktu wiązania, 462
- tworzenie, 455
- układ standardowy, 459
- ustawianie macierzy, 459
- wartości tablicy, 458
- wartość zmiennej typu float, 458
- znajdowanie indeksu, 461

 blokada gimbała, 188  
 błędy EGL, 627  
 błędy OpenGL, 76  
 bool, 243  
 bufor danych przekształceń, 501  
 bufor głębi, 128
 

- aplikacje dla iPhone'a, 638
- maskowanie, 406

 bufor klatki, 100  
 bufor kolorów, 100, 135, 209

bufor obrazu, 326, 338
 

- FBO, 338
- kompletność, 344
- kopiowanie danych, 347
- OpenGL ES 2.0, 614
- sprawdzanie, 345
- stosowanie, 348

 bufor pikseli, 100, 331
 

- inicjalizacja, 332
- odczytywanie danych pikseli, 331
- stosowanie, 332
- tworzenie, 337

 bufor przekształcenia zwrotnego, 500, 501, 502  
 bufor rysowania, 341
 

- odwzorowywanie buforów, 343
- przekazywanie danych z shadera, 343

 bufor szablonu, 93, 397, 399, 403
 

- maskowanie, 407

 bufor wierzchołków, 478
 

- tworzenie, 478

 buforowanie, 326, 568
 

- dostęp do danych, 360
- formaty zmiennoprzecinkowe, 365
- kompresja tekstur, 386
- kopiowanie buforów, 361
- mapowanie buforów, 360
- odwzorowywanie fragmentów wyjściowych, 362
- podwójne buforowanie, 93, 539, 556
- wielopróbkowanie, 379
- wysyłanie danych z shadera pikseli, 362

 bufory, 326
 

- FBO, 348
- modele użycia, 329
- napełnianie, 328
- obiekty bufora pikseli, 329
- odczytywanie danych pikseli, 331
- odwiązanie od punktu wiązania, 328
- PBO, 329, 330
- punkty wiązania obiektów buforowych, 327
- tworzenie, 327
- usuwanie, 328

 bvec2, 244  
 bvec3, 244  
 bvec4, 244

## C

CAD, 49, 365  
 całkowitoliczbowe indeksy elementów bez znaku, 615  
 Cathode Ray Tube, 41  
 ceil(), 270

- cel shading, 292
  - central processing unit, 50
  - centroid, 247, 448, 449, 450
  - CGDisplayHideCursor(), 579
  - CGL, 560, 581
    - CGLGetCurrentContext(), 581
    - CGLSetParameter(), 582
    - czas między zamianami buforów, 582
    - funkcje, 581
    - kCGLCESurfaceBackingSize, 583
    - kCGLCPSurfaceBackingSize, 583
    - kCGLEMPEngine, 583
    - kontekst, 581
    - przyspieszanie operacji wypełniania, 582
    - synchronizacja szybkości klatek, 581
    - szarpanie obrazu, 581
    - wielowątkowość, 583
  - CGLEnable(), 583
  - CGLGetCurrentContext(), 581
  - CGLSetParameter(), 582
  - ChangeSize(), 94, 96, 175, 639
  - chmury, 367
  - ChoosePixelFormat(), 545
  - ciągi linii, 119
  - cieniowanie, 45
    - cieniowanie fragmentów, 108
    - cieniowanie Gourauda, 274, 282
    - cieniowanie kreskówkowe, 292
    - cieniowanie Phonga, 281, 282
    - cieniowanie wierzchołków, 108
  - clamp(), 270, 271, 453
  - clamped, 217
  - clipping region, 53
  - Cocoa, 561, 573
    - atrybuty formatów pikseli, 575
    - atrybuty widoku OpenGL, 567
    - buforowanie, 568
    - formaty pikseli, 574
    - GLTools, 569
    - konfiguracja właściwości widoku OpenGL, 565
    - NSOpenGLPixelFormat, 574
    - NSOpenGLView, 562, 565
    - Objective-C++, 569
    - pliki tekstur, 572
    - renderowanie pełnoekranowe, 574
    - skracanie programu SphereWorld, 570
    - SphereWorld, 569, 571
    - szkielet klasy widoku OpenGL, 567
    - tworzenie klasy OpenGL, 563
    - tworzenie programu, 561
    - widok OpenGL, 561
  - Cocoa Application, 86
  - CocoaGL, 561, 580
  - color buffer, 100
  - COLORREF, 98
  - column-major matrix ordering, 154
  - COM, 68
  - compatibility profile, 23
  - Component Object Model, 68
  - const, 246, 247
  - coordinate system, 52
  - CopyColorData4f(), 125
  - CopyNormalDataf(), 125
  - CopyTexCoordData2f(), 125
  - CopyVertexData(), 221
  - CopyVertexData3f(), 100, 102, 126
  - core profile, 23
  - cos(), 266
  - cosh(), 266
  - CPU, 50
  - Create a New Xcode Project, 85
  - crepuscular rays, 367
  - cross(), 267, 453
  - CRT, 41
  - CStopWatch, 177
  - cube map, 300
  - Cubemap, 300, 305
  - czas rzeczywisty, 41
  - czas wykonywania poleceń, 475
  - cząsteczki, 309, 511
  - częściowo przykryte wielopróbkowane piksele, 449
  - czworokąt, 436
  - czworokąt pokrywający cały ekran, 436
  - czyszczenie bufora, 100
- ## D
- dane attribute, 109
  - dane graficzne, 196
  - dane teksturowe, 110, 320
  - dane uniform, 259
  - DCE, 534
  - decaling, 132
  - definiowanie
    - płaszczyzna obcinania, 519
    - widok, 96
  - degrees(), 266
  - deklaracja
    - atrybuty, 249
    - blok zmiennych jednorodnych, 454
    - dane wyjściowe, 250
    - zmiennie, 243

Desktop Window Manager, 534  
 deprecated, 23  
 depth buffer, 128  
 depth clamping, 400  
 DEPTH\_STENCIL, 340  
 Desktop Compositing Engine, 534  
 detektor krawędzi Sobela, 443  
 determinant(), 268, 453  
 determinanta macierzy, 453  
 diffuse light, 272, 279  
 DiffuseLight, 274, 278  
 Direct3D, 68, 538  
 directional light, 272  
 DirectX, 68  
 DirectX 3D, 23  
 discard, 289, 446  
 Dissolve, 289  
     shader fragmentów, 290  
 distance(), 267, 453  
 dithering, 390, 404  
 do, 452  
 dodawanie aktorów, 191  
 dodawanie pliku źródłowego, 82  
 dokonaniu(), 478  
 dołączanie obiektów RBO, 340  
 dołączanie plików nagłówkowych, 92  
 dołączanie shadera, 256  
 domyślny shader oświetlenia, 114  
 dopasowanie tekstury do obiektu geometrycznego,  
     214  
 dostęp do tablic tekstur, 320  
 dot(), 267, 453  
 dowiązanie do stanów tekstury, 211  
 Draw(), 101, 126, 167, 183, 222  
 DWM, 534  
 dwoistość model-widok, 157  
 dwuwymiarowy układ kartezjański, 52  
 dysk, 170

## E

efekt cząsteczkowy, 309  
 efekt erozji, 290  
 efekt lustra, 354  
 efekt mieszania kolorów, 47  
 efekt odbicia, 47, 240, 304, 355  
 efekt poświaty, 376  
 efekt przestrzeni międzygwiazdnej, 312  
 efekt przezroczystości, 47  
 efekt rozmycia obiektów w ruchu, 332  
 efekt trójwymiarowy, 43, 44

EGL, 619  
     API renderingu, 620  
     atrybuty konfiguracji, 621, 624  
     błędy, 627  
     bufory, 626  
     eglBindAPI, 620  
     ekrany, 619  
     inicjalizacja, 620  
     konfiguracje ekranu, 621  
     kontekst renderingu, 625  
     łańcuchy, 627  
     pobieranie łańcuchów, 627  
     powierzchnia renderingu, 625  
     prezentacja buforów, 626  
     rozszerzanie, 627  
     synchronizacja renderowania, 626  
     tworzenie okna, 621  
     tworzenie powierzchni renderingu, 625  
     wybór konfiguracji, 622  
     zapytania o atrybuty konfiguracji, 623  
     zarządzanie kontekstem, 625  
 EGL\_ALPHA\_MASK\_SIZE, 622, 624  
 EGL\_ALPHA\_SIZE, 621, 624  
 EGL\_BAD\_ACCESS, 627  
 EGL\_BAD\_ALLOC, 627  
 EGL\_BAD\_ATTRIBUTE, 627  
 EGL\_BAD\_CONFIG, 627  
 EGL\_BAD\_CONTEXT, 627  
 EGL\_BAD\_CURRENT\_SURFACE, 627  
 EGL\_BAD\_DISPLAY, 627  
 EGL\_BAD\_MATCH, 627  
 EGL\_BAD\_NATIVE\_PIXMAP, 627  
 EGL\_BAD\_NATIVE\_WINDOW, 627  
 EGL\_BAD\_PARAMETER, 627  
 EGL\_BAD\_SURFACE, 627  
 EGL\_BIND\_TO\_TEXTURE\_RGB, 621, 624  
 EGL\_BIND\_TO\_TEXTURE\_RGBA, 621, 624  
 EGL\_BLUE\_SIZE, 621, 624  
 EGL\_BUFFER\_SIZE, 621, 624  
 EGL\_COLOR\_BUFFER\_TYPE, 622, 624  
 EGL\_CONFIG\_CAVEAT, 621, 623, 624  
 EGL\_CONFIG\_ID, 621, 624  
 EGL\_CONTEXT\_LOST, 627  
 EGL\_CORE\_NATIVE\_ENGINE, 626  
 EGL\_DEFAULT\_DISPLAY, 620  
 EGL\_DEPTH\_SIZE, 621, 624  
 EGL\_EXTENSIONS, 627  
 EGL\_FALSE, 627  
 EGL\_GREEN\_SIZE, 621, 624  
 EGL\_LEVEL, 621, 624  
 EGL\_LUMINANCE\_SIZE, 621, 624

EGL\_MAX\_SWAP\_INTERVAL, 622, 624, 626  
 EGL\_MIN\_SWAP\_INTERVAL, 622, 624, 626  
 EGL\_NATIVE\_RENDERABLE, 621, 624  
 EGL\_NATIVE\_VISUAL\_ID, 622  
 EGL\_NATIVE\_VISUAL\_TYPE, 622, 624  
 EGL\_NO\_CONTEXT, 625, 626  
 EGL\_NO\_DISPLAY, 620  
 EGL\_NO\_SURFACE, 626  
 EGL\_NONE, 623  
 EGL\_NOT\_INITIALIZED, 627  
 EGL\_OPENGL\_API, 620  
 EGL\_OPENGL\_ES\_API, 620  
 EGL\_OPENGL\_API, 620  
 EGL\_RED\_SIZE, 621, 624  
 EGL\_RENDERABLE\_TYPE, 622, 624  
 EGL\_SAMPLE\_BUFFERS, 622, 624  
 EGL\_SAMPLES, 622, 624  
 EGL\_STENCIL\_SIZE, 621, 624  
 EGL\_SUCCESS, 627  
 EGL\_SURFACE\_TYPE, 622, 624  
 EGL\_TRANSPARENT\_BLUE\_VALUE, 622, 624  
 EGL\_TRANSPARENT\_GREEN\_VALUE, 622, 624  
 EGL\_TRANSPARENT\_RED\_VALUE, 622, 624  
 EGL\_TRANSPARENT\_TYPE, 622, 624  
 EGL\_TRUE, 627  
 EGL\_VERSION, 627  
 eglBindAPI(), 620  
 eglChooseConfig(), 622  
 eglCreateContext(), 625  
 eglCreateWindowSurface(), 625  
 eglDestroyContext(), 626  
 eglDestroySurface(), 625  
 eglGetConfigAttrib(), 623  
 eglGetConfigs(), 623  
 eglGetDisplay(), 619  
 eglGetError(), 627  
 eglGetProcAddress(), 627  
 eglInitialize(), 620  
 eglMakeCurrent(), 620, 626  
 eglQueryAPI(), 620  
 eglQueryString(), 627  
 eglReleaseThread(), 620  
 eglSwapBuffers(), 626  
 eglSwapInterval(), 626  
 eglTerminate(), 620  
 eglWaitGL(), 626  
 eglWaitNative(), 626  
 ekrany EGL, 619  
 EmitPrimitive(), 423  
 EmitVertex(), 422, 423, 427

End(), 125, 126, 167  
 EndPrimitive(), 422, 423, 427  
 equal(), 269  
 ETC\_RGB8, 615  
 ETC1, 386  
 EULER, 187  
 exp(), 266  
 exp2(), 266  
 EXT\_, 66

## F

faceforward(), 267  
 FBO, 338, 350, 353  
     stosowanie, 348  
 fence sync object, 523  
 File, 82  
 fill limited, 129  
 filtr pomniejszający, 215  
 filtr powiększający, 215  
 filtr splotu, 440  
 filtr tekstur mipmapowanych, 225  
 filtrowanie tekstur, 215  
     filtrowanie anizotropowe, 234  
     filtrowanie izotropowe, 235  
     filtrowanie liniowe, 215, 216  
     filtrowanie mipmap, 224  
     filtrowanie najbliższego sąsiada, 215, 216  
 fizyczne symulacje w shaderze wierzchołków, 410  
 flat, 259, 448  
 flat shader, 114  
 FlatShader, 263  
 floatBitsToInt(), 271, 453  
 floatBitsToUInt(), 272  
 flocking algorithm, 511  
 floor(), 270  
 for, 452  
 foreshortening, 44, 158  
 format OpenEXR, 368  
 formaty całkowitoliczbowe, 384  
 formaty pikseli, 201, 539, 574  
     atrybuty, 541  
     ustawianie, 546  
     wybór, 546  
     wyliczenia, 545  
 formaty tekstur, 209  
     formaty tekstur skompresowanych, 237, 386  
     formaty zmiennoprzecinkowe bufora renderowania,  
     365, 366  
     odzorowywanie tonów, 369  
     renderowanie HDR, 366

- fract(), 270
- fragment shader, 108
- fraktale, 442
- framebuffer, 100, 338
- framebuffer object, 338
- freeglut, 72, 80, 589
- FREEGLUT\_STATIC, 92
- front face culling, 128
- frusta, 58, 112
- frusta widoku, 519
- frustum, 58
- funkcje
  - abs(), 269
  - acos(), 266
  - acosh(), 266
  - all(), 269, 453
  - any(), 269, 453
  - asin(), 266
  - asinh(), 266
  - atan(), 266
  - atanh(), 266
  - ceil(), 270
  - CGLEnable(), 583
  - ChangeSize(), 639
  - ChoosePixelFormat(), 545
  - clamp(), 270, 271, 453
  - CopyVertexData(), 221
  - cos(), 266
  - cosh(), 266
  - cross(), 267, 453
  - degrees(), 266
  - determinant(), 268, 453
  - distance(), 267, 453
  - dokonaniu(), 478
  - dot(), 267, 453
  - eglBindAPI(), 620
  - eglChooseConfig(), 622
  - eglCreateContext(), 625
  - eglCreateWindowSurface(), 625
  - eglDestroyContex(), 626
  - eglDestroySurface(), 625
  - eglGetConfigAttrib(), 623
  - eglGetConfigs(), 623
  - eglGetDisplay(), 619
  - eglGetError(), 627
  - eglGetProcAddress(), 627
  - eglInitialize(), 620
  - eglMakeCurrent(), 620, 626
  - eglQueryAPI(), 620
  - eglQueryString(), 627
  - eglReleaseThread(), 620
  - eglSwapBuffers(), 626
  - eglSwapInterval(), 626
  - eglTerminate(), 620
  - eglWaitGL(), 626
  - eglWaitNative(), 626
  - EmitPrimitive(), 423
  - EmitVertex(), 422, 423, 427
  - EndPrimitive(), 422, 423, 427
  - equal(), 269
  - exp(), 266
  - exp2(), 266
  - faceforward(), 267
  - floatBitsToInt(), 271, 453
  - floatBitsToUint(), 272
  - floor(), 270
  - fract(), 270
  - GetDC(), 554
  - GetNormalMatrix(), 275
  - glActiveTexture(), 306, 307, 337
  - glAttachShader(), 256, 419, 612
  - glBegin(), 608
  - glBeginConditionalRender(), 473
  - glBeginQuery(), 468, 469, 475, 507
  - glBeginTransformFeedback(), 505, 506
  - glBindAttribLocation(), 256, 257, 613
  - glBindBuffer(), 327, 328, 337, 479, 481, 484, 485, 503, 504
  - glBindBufferBase(), 462, 503, 504
  - glBindBufferRange(), 504
  - glBindFragDataLocation(), 363
  - glBindFragDataLocationIndexed(), 364, 402
  - glBindFramebuffer(), 339, 341
  - glBindRenderbuffer(), 340
  - glBindTexture(), 211, 219, 220, 234, 292, 297, 306, 307, 337
  - glBindVertexArray(), 478, 485
  - glBlendColor(), 140, 402
  - glBlendEquation(), 139, 401
  - glBlendEquationSeparate(), 401
  - glBlendFunc(), 137, 138, 139, 140, 401
  - glBlendFuncSeparate(), 140, 401
  - glBlitFramebuffer(), 347, 351
  - glBufferData(), 328, 331, 337, 360, 455, 478, 482, 503
  - glBufferSubData(), 328, 329, 478
  - glCheckFramebufferStatus(), 345
  - GLclampf(), 98
  - glClear(), 100, 604
  - glClearBufferiv(), 384

glClearBufferuiv(), 384  
 glClearColor(), 98, 604  
 glClearStencil(), 399  
 glClientWaitSync(), 525, 526  
 glColorMask(), 406, 470  
 glColorMaski(), 406  
 glCompileShader(), 256, 419, 612  
 glCompressedTexImage1D(), 239, 387  
 glCompressedTexImage2D(), 239, 387, 615  
 glCompressedTexImage3D(), 239, 387  
 glCompressedTexSubImage(), 239  
 glCopyBuffer(), 478  
 glCopyBufferSubData(), 362  
 glCopyTexImage1D(), 210  
 glCopyTexImage2D(), 210  
 glCopyTexImage3D(), 211  
 glCopyTexSubImage1D(), 210  
 glCopyTexSubImage2D(), 211  
 glCopyTexSubImage3D(), 211  
 glCopyTexSubImage3D(), 210  
 glCreateProgram(), 256, 612  
 glCreateShader(), 255, 419, 612  
 glCullFace(), 128  
 glDeleteBuffers(), 328  
 glDeleteFramebuffers(), 339  
 glDeleteProgram(), 257  
 glDeleteQueries(), 467  
 glDeleteSync(), 527  
 glDeleteTextures(), 212, 219  
 glDeleteVertexArrays(), 484, 485  
 glDepthMask(), 406  
 glDisable(), 78, 143, 379  
 glDrawArrays(), 415, 426, 437, 477, 484, 486, 488, 490, 608, 611  
 glDrawArraysInstanced(), 490, 491, 495  
 glDrawBuffer(), 569  
 glDrawBuffers(), 251, 343, 344, 556  
 glDrawElement(), 483  
 glDrawElements(), 415, 426, 428, 477, 482, 484, 486, 488, 490, 611  
 glDrawElementsBaseVertex(), 483  
 glDrawElementsInstanced(), 490, 491, 495  
 glDrawElementsInstancedBaseVertex(), 483  
 glDrawRangeElements(), 482, 611  
 glDrawRangeElementsBaseVertex(), 483  
 glElementPointer(), 482  
 glEnable(), 78, 117, 143, 303  
 glEnd(), 608  
 glEndConditionalRender(), 473  
 glEndQuery(), 468, 469, 475, 508  
 glEndTransformFeedback(), 506, 526  
 glewInit(), 94, 537  
 glFenceSync(), 523, 524, 526  
 glFinish(), 523  
 glFlush(), 523, 568  
 glFlushMappedBufferRange(), 361  
 glFramebufferRenderbuffer(), 341  
 glFramebufferTexture1D(), 353  
 glFramebufferTexture2D(), 353  
 glFramebufferTexture3D(), 353  
 glGenBuffers(), 327, 478  
 glGenMipmap(), 226  
 glGenFramebuffers(), 339  
 glGenQueries(), 467, 507  
 glGenRenderbuffers(), 340  
 glGenTextures(), 211, 212, 219, 233, 292  
 glGenVertexArrays(), 478, 484, 485  
 glGetActiveUniformsiv(), 456, 457  
 glGetBooleanv(), 79, 614  
 glGetBufferParameteriv(), 614  
 glGetBufferSubData(), 500  
 glGetCompressedTexImage(), 239, 329, 387  
 glGetDoublev(), 79  
 glGetError(), 76, 347, 467  
 glGetFloatv(), 79, 117, 235, 614  
 glGetFramebufferAttachmentParameteriv(), 385  
 glGetInteger64v(), 526  
 glGetIntegerv(), 79, 305, 321, 340, 344, 379, 422, 505, 548, 599, 614  
 glGetMultisamplefv(), 379, 380, 391  
 glGetQueryObjectiiv(), 468, 469, 475, 508  
 glGetShader(), 256  
 glGetShaderInfoLog(), 256  
 glGetShaderiv(), 256  
 getString(), 77, 536, 544, 548, 599  
 getStringi(), 65  
 glGetSynciv(), 524  
 glGetTexImage(), 239, 329  
 glGetTexLevelParameter(), 322  
 glGetTexParameteri(), 387  
 glGetUniformBlockIndex(), 461, 462  
 glGetUniformIndices(), 456  
 glGetUniformLocation(), 260, 261, 287  
 glHint(), 77, 238  
 glIsEnabled(), 78  
 glIsTexture(), 212, 614  
 glLineWidth(), 119  
 glLinkProgram(), 257, 363, 502, 613  
 glLogicOp(), 405  
 glMapBuffer(), 360, 361, 455, 478, 500, 615



funkcje

- glMapBufferRange(), 360
- glMultiDrawArrays(), 486, 487
- glMultiDrawElements(), 486
- glMultiDrawElementsBaseVertex(), 483
- glPixelStore(), 199
- glPixelStoref(), 199
- glPixelStorei(), 199
- glPointParameter(), 314
- glPointSize(), 116, 117, 310
- glPolygonMode(), 130, 132, 192, 263
- glPolygonOffset(), 121, 133
- glPopAttrib(), 600
- glPrimitiveRestartIndex(), 488
- glProvokingVertex(), 259
- glPushAttrib(), 600
- glQueryCounter(), 476
- glReadBuffer(), 203, 210, 331, 344, 347
- glReadPixels(), 200, 203, 327, 329, 330, 331, 332, 333
- glRenderbufferStorage(), 340, 366
- glRenderbufferStorageMultisample(), 340, 380
- glRotate(), 180
- glSampleCoverage(), 144, 392
- glSampleMaski(), 392
- glScissor(), 135, 390
- glShaderBinaryOES(), 612
- glShaderSource(), 612
- glStencilFunc(), 399
- glStencilFuncSeparate(), 397, 399
- glStencilOp(), 399
- glStencilOpSeparate(), 397, 399
- glTexBuffer(), 337, 415
- glTexImage(), 208, 210, 224, 236, 237, 239
- glTexImage1D(), 208, 387
- glTexImage2D(), 208, 301, 321, 322, 387, 637
- glTexImage2DMultisample(), 380
- glTexImage3D(), 208, 318
- glTexImage3DMultisample(), 380
- glTexParamaterf(), 214
- glTexParamaterfv(), 215
- glTexParamateri(), 214
- glTexParamateriv(), 215
- glTexParameter(), 214
- glTexParameterf(), 235
- glTexParameterfv(), 218
- glTexParameterI(), 215, 216, 224, 234, 318
- glTexSubImage(), 210, 239
- glTexSubImage1D(), 210
- glTexSubImage2D(), 210
- glTexSubImage3D(), 210
- glTexImage(), 211
- glTexImage2D(), 296, 333
- glTexParameter(), 211
- glTextSubImage(), 211
- glGetProcAddress(), 65
- glIsExtSupported(), 65, 235
- glLoadShaderFile(), 255
- glLoadShaderPairWithAttributes(), 252, 254, 255, 258
- gltMakeCube(), 302
- gltMakeCylinder(), 169
- gltMakeDisk(), 170
- gltMakeSphere(), 168
- gltMakeTorus(), 169
- glTransformFeedbackVaryings(), 501, 502, 505
- gltReadTGABits(), 205
- gltSetWorkingDirectory(), 93, 636
- gltWriteTGA(), 203
- glUniform(), 260, 261, 613
- glUniform1f(), 260, 261
- glUniform1fv(), 261, 262
- glUniform1i(), 260, 261
- glUniform1iv(), 261
- glUniform2f(), 260
- glUniform2fv(), 261
- glUniform2i(), 260
- glUniform2iv(), 261
- glUniform3f(), 260
- glUniform3fv(), 261
- glUniform3i(), 261
- glUniform3iv(), 261
- glUniform4f(), 260, 261
- glUniform4fv(), 261, 262
- glUniform4i(), 261
- glUniform4iv(), 261
- glUniformBlockBinding(), 462
- glUniformMatrix(), 613
- glUniformMatrix2fv(), 262
- glUniformMatrix3fv(), 262
- glUniformMatrix4fv(), 262
- glUnmapBuffer(), 360, 361, 615
- glupMainLoop(), 538
- glUseProgram(), 258, 613
- glutCreateWindow(), 94
- glutDisplayFunc(), 94
- glutInit(), 93
- glutInitDisplayMode(), 93, 128, 143
- glutInitWindowSize(), 94
- glutMainLoop(), 94
- glutPostRedisplay(), 103
- glutReshapeFunc(), 94, 96

glutSpecialFunc(), 101  
 glutSwapBuffers(), 101  
 glVertexAttribDivisor(), 496, 497, 498, 499  
 glVertexAttribPointer(), 479, 480, 481, 482, 483,  
     484, 485, 496, 611  
 glViewport(), 96, 352, 604  
 glWaitSync(), 525, 526, 527  
 glXChooseFBConfig(), 594, 595  
 glXChooseFBConfigs(), 595  
 glXCopyContext(), 600  
 glXCreateContextAttribsARB(), 598, 599  
 glXCreateNewContext(), 598  
 glXCreateWindow(), 596  
 glXDestroyContext(), 600  
 glXDestroyWindow(), 596  
 glXGetClientString(), 597  
 glXGetCurrentContext(), 602  
 glXGetCurrentDisplay(), 602  
 glXGetCurrentDrawable(), 602  
 glXGetCurrentReadDrawable(), 602  
 glXGetFBConfigAttrib(), 594  
 glXGetFBConfigs(), 592  
 glXGetProcAddress(), 597  
 glXGetServerString(), 597  
 glXGetVisualFromFBConfig(), 595, 603  
 glXIsDirect(), 600  
 glXMakeContextCurrent(), 601  
 glXQueryContext(), 602  
 glXQueryDrawable(), 602  
 glXQueryExtensionsString(), 597  
 glXSwapBuffers(), 601, 605  
 glXWaitGL(), 601  
 glXWaitX(), 601  
 greaterThan(), 269  
 greaterThanEqual(), 269  
 intBitsToFloat(), 272, 453  
 inverse(), 268, 453  
 inversesqrt(), 266  
 isinf(), 271  
 isnan(), 271  
 length(), 267, 453  
 lessThan(), 269  
 lessThanEqual(), 269, 453  
 lgFrontFace(), 122  
 LoadTGATexture(), 219, 220, 297  
 LoadTGATextureRect(), 297  
 log(), 266, 453  
 log2(), 266  
 m3dCrossProduct3(), 152  
 m3dDotProduct3(), 151  
 m3dGetAngleBetweenVectors3(), 151  
 m3dLoadIdentity44(), 162  
 m3dMakeOrthographicMatrix(), 298  
 m3dMatrixMultiply44(), 165, 166, 177  
 m3dRotationMatrix(), 163  
 m3dRotationMatrix44(), 163, 177  
 m3dScaleMatrix44(), 164  
 m3dTransformVector4(), 192, 193  
 m3dTranslationMatrix44(), 162, 165, 177  
 main(), 93  
 MakePyramid(), 220  
 malloc(), 205  
 matrixCompMult(), 268, 453  
 max(), 270, 276  
 min(), 270  
 mix(), 271  
 mod(), 270  
 modf(), 270  
 MultiTexCoord2f(), 221  
 normalize(), 267, 453  
 not(), 269  
 notEqual(), 269, 453  
 outerProduct(), 268, 453  
 PopMatrix(), 180  
 pow(), 266  
 PushMatrix(), 180  
 radians(), 266  
 reflect(), 267, 453  
 refract(), 267, 453  
 round(), 270  
 roundEven(), 270  
 SetPixelFormat(), 546  
 SetupRC(), 233  
 SetupWindow(), 550  
 sign(), 269, 270  
 sin(), 266  
 sinh(), 266  
 smoothstep(), 271, 453  
 SpecialKeys(), 102  
 sqrt(), 266  
 step(), 271, 453  
 SwapBuffers(), 556, 557  
 tan(), 266  
 tanh(), 266  
 texelFetch(), 380  
 transpose(), 268, 453  
 trunc(), 270  
 uintBitsToFloat(), 272  
 wglChoosePixelFormat(), 540  
 wglChoosePixelFormatARB(), 540, 545, 546, 556  
 wglCreateContext(), 549  
 wglCreateContextAttribsARB(), 547, 548, 549

funkcje

- wglDeleteContext(), 549, 554
- wglGetExtensionsStringARB(), 536
- wglGetPixelFormatAttribARB(), 544, 546
- wglGetPixelFormatAttribfvARB(), 541, 545
- wglGetPixelFormatAttribivARB(), 541, 545, 546
- wglGetPixelFormatAttributeivARB(), 544
- wglGetProcAddress(), 535, 536, 544, 627
- wglMakeCurrent(), 549, 554
- wglSwapIntervalEXT(), 557
- WinMain(), 93
- XCreateWindow(), 595, 596
- XDestroyWindow(), 597
- XOpenDisplay(), 603
- funkcje GLSL, 243, 265, 267
  - funkcje geometryczne, 267
  - funkcje macierzowe, 267, 268
  - funkcje porównywania wektorów, 267, 269
  - funkcje trygonometryczne, 265
  - funkcje wykładnicze, 266
- funkcje mieszania, 401
- funkcje shaderów, 448
- funkcje trygonometryczne, 453
- funkcje wycofywane, 69

**G**

- GDI, 54, 538
- GDI+, 538
- general-purpose computing on graphics
  - processing units, 384
- generowanie
  - dane obrazu w shaderze fragmentów, 442
  - geometria w shaderze geometrii, 427
  - grafika trójwymiarowa, 48
  - poziomy mipmap, 226
- geometry shader, 108
- GetCameraMatrix(), 189
- GetDC(), 554
- GetMatrix(), 179, 186
- GetModelViewProjectionMatrix(), 183
- GetNormalMatrix(), 275
- gimbal lock, 188
- GL Extension Wrangler, 537, 589
- GL\_ADD, 141
- GL\_ALL\_ATTRIB\_BITS, 600
- GL\_ALPHA, 209
- GL\_ALPHA\_SATURATE, 402
- GL\_ALREADY\_SIGNALED, 525
- GL\_ALWAYS, 398
- GL\_AMD\_debug\_context, 548, 549, 600

- GL\_AND, 406
- GL\_AND\_INVERTED, 406
- GL\_AND\_REVERSE, 406
- GL\_ANY\_SAMPLES\_PASSED, 472
- GL\_ARB\_compatibility, 70
- GL\_ARB\_texture\_compression, 237
- GL\_ARRAY\_BUFFER, 327, 478, 479, 481, 482, 509
- GL\_BACK, 128, 130, 251, 556
- GL\_BACK\_LEFT, 203
- GL\_BACK\_RIGHT, 203
- GL\_BGR, 201
- GL\_BGR\_INTEGER, 201
- GL\_BGRA, 201
- GL\_BGRA\_INTEGER, 201
- GL\_BLEND, 138, 400
- GL\_BLUE, 201
- GL\_BLUE\_INTETER, 201
- GL\_BUFFER\_TEXTURE, 415
- GL\_BYTE, 202
- GL\_CCW, 122
- GL\_CLAMP, 217, 218
- GL\_CLAMP\_TO\_BORDER, 217, 218
- GL\_CLAMP\_TO\_EDGE, 217, 218, 615
- GL\_CLEAR, 406
- gl\_Clip\_Distance, 519, 520
- GL\_CLIP\_DISTANCE0, 519
- GL\_COLOR\_ATTACHMENT0, 344
- GL\_COLOR\_BUFFER\_BIT, 347
- GL\_COLOR\_LOGIC\_OP, 405
- GL\_COMPILE\_STATUS, 256
- GL\_COMPRESSED\_RED, 237, 386
- GL\_COMPRESSED\_RED\_RGTC1, 386
- GL\_COMPRESSED\_RG, 237, 386
- GL\_COMPRESSED\_RG\_RGTC1, 386
- GL\_COMPRESSED\_RG\_RGTC2, 237
- GL\_COMPRESSED\_RGB, 237, 240, 386
- GL\_COMPRESSED\_RGB\_S3TC\_DXT1, 239
- GL\_COMPRESSED\_RGBA, 237, 386
- GL\_COMPRESSED\_RGBA\_S3TC\_DXT1, 239
- GL\_COMPRESSED\_RGBA\_S3TC\_DXT3, 239
- GL\_COMPRESSED\_RGBA\_S3TC\_DXT5, 239
- GL\_COMPRESSED\_SIGNED\_RED\_RGTC1, 237, 386
- GL\_COMPRESSED\_SIGNED\_RG\_RGTC1, 386
- GL\_COMPRESSED\_SIGNED\_RG\_RGTC2, 237
- GL\_COMPRESSED\_SRGB, 237, 386
- GL\_COMPRESSED\_SRGB\_ALPHA, 237, 386
- GL\_COMPRESSED\_TEXTURE\_FORMATS, 238
- GL\_CONDITION\_SATISFIED, 525
- GL\_CONSTANT\_ALPHA, 137, 140, 402
- GL\_CONSTANT\_COLOR, 137, 140, 402
- GL\_COORD\_REPLACE, 613

GL\_COPY, 406  
 GL\_COPY\_INVERTED, 406  
 GL\_COPY\_READ\_BUFFER, 327, 336, 362  
 GL\_COPY\_WRITE\_BUFFER, 327, 362  
 GL\_CULL\_FACE, 128  
 GL DECR, 398  
 GL DECR\_WRAP, 398  
 GL\_DEPTH\_ATTACHMENT, 353  
 GL\_DEPTH\_BUFFER\_BIT, 347  
 GL\_DEPTH\_CLAMP, 379, 400, 521  
 GL\_DEPTH\_COMPONENT, 200, 201  
 GL\_DEPTH\_STENCIL, 200, 201, 341  
 GL\_DEPTH\_STENCIL\_ATTACHMENT, 341  
 GL\_DEPTH\_TEST, 78, 129  
 GL\_DITHER, 404  
 GL\_DRAW\_BUFFER0, 400  
 GL\_DRAW\_BUFFER1, 400  
 GL\_DRAW\_FRAMEBUFFER, 339, 341, 347, 353  
 GL\_DST\_ALPHA, 137, 402  
 GL\_DST\_COLOR, 137, 402  
 GL\_DYNAMIC\_COPY, 329, 615  
 GL\_DYNAMIC\_DRAW, 328, 329  
 GL\_DYNAMIC\_READ, 329, 615  
 GL\_ELEMENT\_ARRAY\_BUFFER, 327, 482, 484  
 GL\_EQUAL, 398, 451  
 GL\_EQUIV, 406  
 GL\_ETC1\_RGB8\_OES, 615  
 GL\_EXT\_texture\_compression\_s3tc, 238, 239, 240  
 GL\_EXT\_texture\_filter\_anisotropic, 235  
 GL\_FALSE, 256, 406  
 GL\_FASTEST, 143  
 GL\_FILL, 130  
 GL\_FIRST\_VERTEX\_CONVENTION, 259  
 GL\_FLOAT, 202  
 GL\_FLOAT\_32\_UNSIGNED\_INT\_24\_8\_REV, 202  
 gl\_FragColor, 362  
 gl\_FragData, 362  
 gl\_FragDepth, 447  
 GL\_FRAGMENT\_SHADER, 255  
 GL\_FRAMEBUFFER\_ATTACHMENT\_COLOR\_ENCODING, 385  
 GL\_FRAMEBUFFER\_COMPLETE, 345, 346  
 GL\_FRAMEBUFFER\_INCOMPLETE\_ATTACHMENT, 346  
 GL\_FRAMEBUFFER\_INCOMPLETE\_DRAW\_BUFFER, 346  
 GL\_FRAMEBUFFER\_INCOMPLETE\_LAYER\_TARGETS, 346  
 GL\_FRAMEBUFFER\_INCOMPLETE\_MISSING\_ATTACHMENT, 346  
 GL\_FRAMEBUFFER\_INCOMPLETE\_MULTISAMPLE, 346  
 GL\_FRAMEBUFFER\_INCOMPLETE\_READ\_BUFFER, 346  
 GL\_FRAMEBUFFER\_SRGB, 385  
 GL\_FRAMEBUFFER\_UNDEFINED, 346  
 GL\_FRAMEBUFFER\_UNSUPPORTED, 346  
 GL\_FRONT, 128, 130, 556  
 GL\_FRONT\_AND\_BACK, 128, 130, 399  
 GL\_FRONT\_LEFT, 203  
 GL\_FRONT\_RIGHT, 203  
 GL\_FUNC\_ADD, 139, 401  
 GL\_FUNC\_REVERSE\_SUBTRACT, 139, 401  
 GL\_FUNC\_SUBTRACT, 139, 401  
 GL\_GEOMETRY\_SHADER, 419  
 GL\_GEQUAL, 398  
 GL\_GREATER, 398  
 GL\_GREEN, 201  
 GL\_GREEN\_INTEGER, 201  
 GL\_HALF\_FLOAT, 202  
 GL\_INCR, 398  
 GL\_INCR\_WRAP, 398  
 gl\_InstanceID, 492, 493, 495  
 GL\_INT, 202  
 GL\_INTERLEAVED\_ATTRIB, 505  
 GL\_INTERLEAVED\_ATTRIBS, 502, 503  
 GL\_INTERLEAVED\_BUFFER, 505  
 GL\_INVALID\_ENUM, 76  
 GL\_INVALID\_FRAMEBUFFER\_OPERATION, 347  
 GL\_INVALID\_OPERATION, 76  
 GL\_INVALID\_VALUE, 76  
 GL\_INVERT, 398, 406  
 GL\_KEEP, 398  
 GL\_LAST\_VERTEX\_CONVENTIONS, 259  
 GL\_LEFT, 203  
 GL\_LEQUAL, 398  
 GL\_LESS, 398  
 GL\_LINE, 130  
 GL\_LINE\_LOOP, 116, 120, 420  
 GL\_LINE\_SMOOTH, 141  
 GL\_LINE\_STRIP, 116, 119  
 GL\_LINE\_STRIP\_ADJACENCY, 433, 434  
 GL\_LINEAR, 215, 224, 225, 234, 297  
 GL\_LINEAR\_MIPMAP\_LINEAR, 225  
 GL\_LINEAR\_MIPMAP\_NEAREST, 225, 236  
 GL\_LINES, 116, 420, 506  
 GL\_LINES\_ADJACENCY, 433, 434  
 GL\_LOWER\_LEFT, 314  
 GL\_LUMINANCE, 209, 237  
 GL\_LUMINANCE\_ALPHA, 209, 237

GL\_LUMINANCE8, 207  
 GL\_MAJOR\_VERSION, 599  
 GL\_MAP\_FLUSH\_EXPLICIT\_BIT, 361  
 GL\_MAP\_INVALIDATE\_BUFFER\_BIT, 361  
 GL\_MAP\_INVALIDATE\_RANGE\_BIT, 361  
 GL\_MAP\_READ\_BIT, 361  
 GL\_MAP\_UNSYNCHRONIZED\_BIT, 361  
 GL\_MAP\_WRITE\_BIT, 361  
 GL\_MAX, 139, 401  
 GL\_MAX\_3D\_TEXTURE\_SIZE, 321  
 GL\_MAX\_CLIP\_DISTANCES, 520  
 GL\_MAX\_COLOR\_ATTACHMENTS, 340  
 GL\_MAX\_CUBE\_MAP\_TEXTURE\_SIZE, 321  
 GL\_MAX\_DRAW\_BUFFERS, 344  
 GL\_MAX\_DUAL\_SOURCE\_DRAW\_BUFFERS, 402  
 GL\_MAX\_FRAGMENT\_UNIFORM\_BUFFERS, 461  
 GL\_MAX\_GEOMETRY\_OUTPUT\_VERTICES, 422  
 GL\_MAX\_GEOMETRY\_UNIFORM\_BUFFERS, 461  
 GL\_MAX\_RENDERBUFFER\_SIZE, 340  
 GL\_MAX\_SERVER\_WAIT\_TIMEOUT, 526  
 GL\_MAX\_TEXTURE\_MAX\_ANISOTROPY\_EXT, 235  
 GL\_MAX\_TRANSFORM\_FEEDBACK\_↪  
   INTERLEAVED\_COMPONENTS, 505  
 GL\_MAX\_UNIFORM\_BUFFER\_BINDINGS, 462  
 GL\_MAX\_UNIFORM\_BUFFERS, 461  
 GL\_MAX\_VERTEX\_UNIFORM\_BUFFERS, 461  
 GL\_MIN, 139, 401  
 GL\_MINOR\_VERSION, 599  
 GL\_MULTISAMPLE, 143, 144  
 GL\_NAND, 406  
 GL\_NEAREST, 215, 217, 224, 225, 234, 297  
 GL\_NEAREST\_MIPMAP\_LINEAR, 225  
 GL\_NEAREST\_MIPMAP\_NEAREST, 225, 236  
 GL\_NEVER, 398  
 GL\_NICEST, 143  
 GL\_NO\_ERROR, 76  
 GL\_NONE, 203  
 GL\_NOOP, 406  
 GL\_NOR, 406  
 GL\_NOTEQUAL, 398  
 GL\_NUM\_COMPRESSED\_TEXTURE\_FORMATS, 238  
 GL\_ONE, 137, 402  
 GL\_ONE\_MINUS\_CONSTANT\_ALPHA, 137, 140, 402  
 GL\_ONE\_MINUS\_CONSTANT\_COLOR, 137, 140, 402  
 GL\_ONE\_MINUS\_DST\_ALPHA, 137, 402  
 GL\_ONE\_MINUS\_DST\_COLOR, 137, 402  
 GL\_ONE\_MINUS\_SRC\_ALPHA, 137, 402  
 GL\_ONE\_MINUS\_SRC\_COLOR, 137, 402  
 GL\_ONE\_MINUS\_SRC1\_ALPHA, 402  
 GL\_ONE\_MINUS\_SRC1\_COLOR, 402  
 GL\_OR, 406  
 GL\_OR\_INVERTED, 406  
 GL\_OR\_REVERSE, 406  
 GL\_OUT\_OF\_MEMORY, 76  
 GL\_PACK\_ALIGNMENT, 199  
 GL\_PACK\_IMAGE\_HEIGHT, 199  
 GL\_PACK\_LSB\_FIRST, 199  
 GL\_PACK\_ROW\_LENGTH, 199  
 GL\_PACK\_SKIP\_IMAGES, 199  
 GL\_PACK\_SKIP\_PIXELS, 199  
 GL\_PACK\_SKIP\_ROWS, 199  
 GL\_PACK\_SWAP\_BYTES, 199  
 GL\_PIXEL\_PACK\_BUFFER, 327, 329, 332, 333, 336  
 GL\_PIXEL\_UNPACK\_BUFFER, 327, 330, 333  
 GL\_POINT, 130  
 GL\_POINT\_SIZE\_GRANULARITY, 117  
 GL\_POINT\_SIZE\_RANGE, 117  
 GL\_POINT\_SMOOTH, 141  
 GL\_POINT\_SPRITE\_COORD\_ORIGIN, 314  
 GL\_POINTS, 116, 420, 506  
 GL\_POLYGON\_OFFSET\_FILL, 133  
 GL\_POLYGON\_OFFSET\_LINE, 133  
 GL\_POLYGON\_OFFSET\_POINT, 133  
 GL\_POLYGON\_SMOOTH, 141, 143  
 GL\_PRIMITIVE\_RESTART, 488  
 GL\_PRIMITIVES\_GENERATED, 507, 508  
 GL\_PROGRAM\_POINT\_SIZE, 117, 310  
 GL\_PROXY\_TEXTURE\_1D, 208, 321  
 GL\_PROXY\_TEXTURE\_2D, 208  
 GL\_PROXY\_TEXTURE\_3D, 208, 321  
 GL\_PROXY\_TEXTURE\_CUBE\_MAP, 321  
 GL\_QUERY\_NO\_WAIT, 474  
 GL\_QUERY\_RESULT\_AVAILABLE, 474  
 GL\_QUERY\_WAIT, 473  
 GL\_R11\_G11\_B10F, 366  
 GL\_R16F, 366  
 GL\_R32F, 366  
 GL\_RASTERIZER\_DISCARD, 415, 506, 507, 510  
 GL\_READ\_FRAMEBUFFER, 339, 341, 347, 353  
 GL\_RED, 201  
 GL\_RED\_INTEGER, 201  
 GL\_RENDERBUFFER, 340  
 GL\_REPEAT, 217, 218, 297  
 GL\_REPEAT\_MIRRORED, 297  
 GL\_REPLACE, 398  
 GL\_RG, 201  
 GL\_RG\_INTEGER, 201  
 GL\_RG16F, 366

GL\_RG32F, 366  
 GL\_RGB, 201, 209  
 GL\_RGB\_8, 404  
 GL\_RGB\_INTEGER, 201  
 GL\_RGB16I, 400  
 GL\_RGB32I, 400  
 GL\_RGB8, 207  
 GL\_RGB9\_E5, 387  
 GL\_RGBA, 201, 209  
 GL\_RGBA\_INTEGER, 201  
 GL\_RGBA16F, 366  
 GL\_RGBA32F, 366  
 GL\_RGBA8, 207  
 GL\_RIGHT, 203  
 GL\_SAMPLE\_ALPHA\_TO\_COVERAGE, 144, 391  
 GL\_SAMPLE\_ALPHA\_TO\_ONE, 144, 391  
 GL\_SAMPLE\_COVERAGE, 144, 392  
 gl\_SampleMask, 393  
 GL\_SAMPLES, 379  
 GL\_SAMPLES\_PASSED, 468, 472  
 GL\_SCISSOR\_TEST, 134, 390  
 GL\_SEPARATE\_ATTRIBS, 502, 504, 505  
 GL\_SET, 406  
 GL\_SHORT, 202  
 GL\_SIGNALED, 524  
 GL\_SRC\_ALPHA, 137, 402, 403  
 GL\_SRC\_ALPHA\_SATURATE, 137  
 GL\_SRC\_COLOR, 137, 402  
 GL\_SRC\_ONE\_MINUS\_ALPHA, 403  
 GL\_SRC1\_ALPHA, 402  
 GL\_SRC1\_COLOR, 402  
 GL\_SRGB8\_ALPHA8, 385  
 GL\_STATIC\_COPY, 329  
 GL\_STATIC\_DRAW, 329, 510  
 GL\_STATIC\_READ, 329, 615  
 GL\_STENCIL\_ATTACHMENT, 353  
 GL\_STENCIL\_BUFFER\_BIT, 347, 399  
 GL\_STENCIL\_INDEX, 200, 201  
 GL\_STENCIL\_TEST, 397  
 GL\_STREAM\_COPY, 329, 615  
 GL\_STREAM\_DRAW, 329, 482, 615  
 GL\_STREAM\_READ, 329, 615  
 GL\_SYNC\_FLUSH\_COMMANDS, 525  
 GL\_SYNC\_FLUSH\_COMMANDS\_BIT, 525, 526  
 GL\_SYNC\_GPU\_COMMANDS\_COMPLETE, 523, 524  
 GL\_SYNC\_STATUS, 524  
 GL\_TEXTURE\_1D, 208, 215, 226, 296  
 GL\_TEXTURE\_1D\_ARRAY, 226, 317  
 GL\_TEXTURE\_2D, 208, 215, 226, 296, 380  
 GL\_TEXTURE\_2D\_ARRAY, 226, 317, 318  
 GL\_TEXTURE\_2D\_MULTISAMPLE, 380  
 GL\_TEXTURE\_2D\_MULTISAMPLE\_ARRAY, 380  
 GL\_TEXTURE\_3D, 208, 215, 226, 296  
 GL\_TEXTURE\_BASE\_LEVEL, 224  
 GL\_TEXTURE\_BORDER\_COLOR, 218  
 GL\_TEXTURE\_BUFFER, 327, 336, 337, 362  
 GL\_TEXTURE\_COMPRESSED, 238  
 GL\_TEXTURE\_COMPRESSED\_IMAGE\_SIZE, 238  
 GL\_TEXTURE\_COMPRESSION\_HINT, 238  
 GL\_TEXTURE\_CUBE\_MAP, 215, 226  
 GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X, 301  
 GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y, 301  
 GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z, 301  
 GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X, 301  
 GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y, 301  
 GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z, 301  
 GL\_TEXTURE\_INTERNAL\_FORMAT, 238, 322  
 GL\_TEXTURE\_MAG\_FILTER, 215  
 GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT, 235  
 GL\_TEXTURE\_MAX\_LEVEL, 224  
 GL\_TEXTURE\_MAX\_LOD, 224  
 GL\_TEXTURE\_MIN\_FILTER, 215  
 GL\_TEXTURE\_MIN\_LOD, 224  
 GL\_TEXTURE\_RECTANGLE, 296, 297  
 GL\_TEXTURE\_WRAP\_R, 217  
 GL\_TEXTURE\_WRAP\_S, 217  
 GL\_TEXTURE\_WRAP\_T, 217  
 GL\_TEXTURE1, 307  
 GL\_TIME\_ELAPSED, 475  
 GL\_TIMEOUT\_EXPIRED, 525  
 GL\_TIMEOUT\_IGNORED, 525, 526  
 GL\_TIMESTAMP, 476  
 GL\_TRANSFORM\_FEEDBACK\_BUFFER, 327, 337, 503, 504, 505  
 GL\_TRANSFORM\_FEEDBACK\_PRIMITIVES\_↪WRITTEN, 507, 508, 509  
 GL\_TRIANGLE\_ADJACENCY, 417  
 GL\_TRIANGLE\_FAN, 101, 116, 123, 125, 420, 426  
 GL\_TRIANGLE\_STRIP, 116, 123, 420, 426, 437, 489  
 GL\_TRIANGLE\_STRIP\_ADJACENCY, 417, 433, 435  
 GL\_TRIANGLES, 116, 420, 426, 506  
 GL\_TRIANGLES\_ADJACENCY, 433, 435  
 GL\_UNIFORM\_ARRAY\_STRIDE, 456, 457, 458  
 GL\_UNIFORM\_BLOCK\_INDEX, 457  
 GL\_UNIFORM\_BUFFER, 327, 337, 462  
 GL\_UNIFORM\_IS\_ROW\_MAJOR, 457  
 GL\_UNIFORM\_MATRIX\_STRIDE, 456, 457  
 GL\_UNIFORM\_NAME\_LENGTH, 457

GL\_UNIFORM\_OFFSET, 456, 457, 458  
 GL\_UNIFORM\_SIZE, 457  
 GL\_UNIFORM\_TYPE, 457  
 GL\_UNPACK\_ALIGNMENT, 199  
 GL\_UNPACK\_IMAGE\_HEIGHT, 199  
 GL\_UNPACK\_LSB\_FIRST, 199  
 GL\_UNPACK\_ROW\_LENGTH, 199  
 GL\_UNPACK\_SKIP\_IMAGES, 199  
 GL\_UNPACK\_SKIP\_PIXELS, 199  
 GL\_UNPACK\_SKIP\_ROWS, 199  
 GL\_UNPACK\_SWAP\_BYTES, 199  
 GL\_UNSIGNALED, 524  
 GL\_UNSIGNED\_BYTE, 202  
 GL\_UNSIGNED\_BYTE\_2\_3\_3\_REV, 201, 202  
 GL\_UNSIGNED\_BYTE\_3\_2\_2, 202  
 GL\_UNSIGNED\_BYTE\_3\_3\_2, 201  
 GL\_UNSIGNED\_INT, 202  
 GL\_UNSIGNED\_INT\_10\_10\_10\_2, 202  
 GL\_UNSIGNED\_INT\_10F\_11F\_11F\_REV, 202  
 GL\_UNSIGNED\_INT\_2\_10\_10\_10\_REV, 202  
 GL\_UNSIGNED\_INT\_24\_8, 202  
 GL\_UNSIGNED\_INT\_8\_8\_8\_8, 202  
 GL\_UNSIGNED\_INT\_8\_8\_8\_8\_REV, 202  
 GL\_UNSIGNED\_SHORT, 202  
 GL\_UNSIGNED\_SHORT\_1\_5\_5\_5\_REV, 202  
 GL\_UNSIGNED\_SHORT\_4\_4\_4\_4, 202  
 GL\_UNSIGNED\_SHORT\_4\_4\_4\_4\_REV, 202  
 GL\_UNSIGNED\_SHORT\_5\_5\_5\_1, 202  
 GL\_UNSIGNED\_SHORT\_5\_6\_5, 202  
 GL\_UNSIGNED\_SHORT\_5\_6\_5\_REV, 202  
 GL\_UPPER\_LEFT, 314, 613  
 GL\_VERSION, 548, 599  
 GL\_VERTEX\_SHADER, 255  
 GL\_WAIT\_FAILED, 525  
 GL\_XOR, 406  
 GL\_ZERO, 137, 398, 402  
 glActiveTexture(), 306, 307, 337  
 glAttachShader(), 256, 419, 612  
 GLBatch, 100, 101, 124, 125, 221, 249, 255, 258, 298  
     Begin(), 124, 306  
     CopyColorData4f(), 125  
     CopyNormalData(), 477  
     CopyNormalDataf(), 125  
     CopyTexCoordData2f(), 125, 306  
     CopyVertexData(), 477  
     End(), 125  
     MultiTexCoord2f(), 221, 306  
     MultiTexCoord2fv(), 306  
 glBegin(), 608  
 glBeginConditionalRender(), 473  
 glBeginQuery(), 468, 469, 475, 507  
 glBeginTransformFeedback(), 505, 506  
 glBindAttribLocation(), 256, 257, 613  
 glBindBuffer(), 327, 328, 337, 479, 481, 484, 485, 503, 504  
 glBindBufferBase(), 462, 503, 504  
 glBindBufferRange(), 504  
 glBindFragDataLocation(), 363  
 glBindFragDataLocationIndexed(), 364, 402  
 glBindFramebuffer(), 339, 341  
 glBindRenderbuffer(), 340  
 glBindTexture(), 211, 219, 220, 222, 234, 292, 297, 306, 307, 337  
 glBindVertexArray(), 478, 485  
 GLbitfield, 75  
 glBlendColor(), 140, 402  
 glBlendEquation(), 139, 401  
 glBlendEquationSeparate(), 401  
 glBlendFunc(), 137, 138, 139, 140, 401  
 glBlendFuncSeparate(), 140, 401  
 glBlitFramebuffer(), 347, 351  
 GLboolean, 75  
 glBufferData(), 328, 331, 337, 360, 455, 478, 482, 503  
 glBufferSubData(), 328, 329, 478  
 GLbyte, 75  
 GLchar, 75  
 glCheckFramebufferStatus(), 345  
 GLclampd, 75  
 GLclampf, 75, 98  
 glClear(), 100, 604  
 glClearBufferiv(), 384  
 glClearBufferuiv(), 384  
 glClearColor(), 98, 604  
 glClearStencil(), 399  
 glClientWaitSync(), 525, 526  
 glColorMask(), 406, 470  
 glColorMaski(), 406  
 glCompileShader(), 256, 419, 612  
 glCompressedTexImage1D(), 239, 387  
 glCompressedTexImage2D(), 239, 387, 615  
 glCompressedTexImage3D(), 239, 387  
 glCompressedTexSubImage(), 239  
 glCopyBuffer(), 478  
 glCopyBufferSubData(), 362  
 glCopyTexImage1D(), 210  
 glCopyTexImage2D(), 210  
 glCopyTexImage3D(), 211  
 glCopyTexSubImage1D(), 210  
 glCopyTexSubImage2D(), 211  
 glCopyTexSubImage3D(), 211

glCopyTexImage3D(), 210  
 glCreateProgram(), 256, 612  
 glCreateShader(), 255, 419, 612  
 glCullFace(), 128  
 glDeleteBuffers(), 328  
 glDeleteFramebuffers(), 339  
 glDeleteProgram(), 257  
 glDeleteQueries(), 467  
 glDeleteSync(), 527  
 glDeleteTextures(), 212, 219  
 glDeleteVertexArrays(), 484, 485  
 glDepthMask(), 406  
 glDisable(), 78, 143, 379  
 GLdouble, 75, 76  
 glDrawArrays(), 415, 426, 437, 477, 484, 486, 488, 490, 608, 611  
 glDrawArraysInstanced(), 490, 491, 495  
 glDrawBuffer(), 569  
 glDrawBuffers(), 251, 343, 344, 556  
 glDrawElement(), 483  
 glDrawElements(), 415, 426, 428, 477, 482, 484, 486, 488, 490, 611  
 glDrawElementsBaseVertex(), 483  
 glDrawElementsInstanced(), 490, 491, 495  
 glDrawElementsInstancedBaseVertex(), 483  
 glDrawRangeElements(), 482, 611  
 glDrawRangeElementsBaseVertex(), 483  
 glElementPointer(), 482  
 glEnable(), 78, 117, 143, 303  
 glEnd(), 608  
 glEndConditionalRender(), 473  
 glEndQuery(), 468, 469, 475, 508  
 glEndTransformFeedback(), 506, 526  
 GLenum, 75, 205, 343  
 GLEW, 66, 73, 79, 94, 536, 537, 573, 589  
     inicjalizacja biblioteki, 94  
     Linux, 590  
 glewInit(), 94, 537  
 glext.h, 536  
 glFenceSync(), 523, 524, 526  
 glFinish(), 523  
 GLfloat, 75, 117, 261  
 glFlush(), 523, 568  
 glFlushMappedBufferRange(), 361  
 GLFrame, 180, 186, 191, 356  
 glFramebufferRenderbuffer(), 341  
 glFramebufferTexture1D(), 353  
 glFramebufferTexture2D(), 353  
 glFramebufferTexture3D(), 353  
 glFrontFace(), 122  
 GLFrustum, 111, 172, 173, 175, 181, 182  
     SetOrthographic(), 111, 172  
     SetPerspective(), 112, 173  
 glGenBuffers(), 327, 478  
 glGenMipmap(), 226  
 glGenFramebuffers(), 339  
 glGenQueries(), 467, 507  
 glGenRenderbuffers(), 340  
 glGenTextures(), 211, 212, 219, 233, 292  
 glGenVertexArrays(), 478, 484, 485  
 GLGeometryTransform, 180, 181, 183  
 glGetActiveUniformsiv(), 456, 457  
 glGetBooleanv(), 79, 614  
 glGetBufferParameteriv(), 614  
 glGetBufferSubData(), 500  
 glGetCompressedTexImage(), 239, 329, 387  
 glGetDoublev(), 79  
 glGetError(), 76, 347, 467  
 glGetFloatv(), 79, 117, 235, 614  
 glGetFramebufferAttachmentParameteriv(), 385  
 glGetInteger64v(), 526  
 glGetIntegerv(), 65, 79, 305, 321, 340, 344, 379, 422, 505, 548, 599, 614  
 glGetMultisamplefv(), 379, 380, 391  
 glGetQueryObjectuiv(), 468, 469, 475, 508  
 glShader(), 256  
 glGetShaderInfoLog(), 256  
 glGetShaderiv(), 256  
 getString(), 77, 536, 544, 548, 599  
 getStringi(), 65  
 glGetSynciv(), 524  
 glGetTexImage(), 239, 329  
 glGetTexLevelParameter(), 322  
 glGetTexParameter(), 387  
 glGetUniformLocation(), 461, 462  
 glGetUniformIndices(), 456  
 glGetUniformLocation(), 260, 261, 287  
 GLhalf, 75  
 glHint(), 77, 238  
 GLint, 65, 75  
 GLint64, 75  
 GLintptr, 75  
 glIsEnabled(), 78  
 glIsTexture(), 212, 614  
 glLineWidth(), 119  
 glLinkProgram(), 257, 363, 502, 613  
 glLogicOp(), 405  
 glMapBuffer(), 360, 361, 455, 478, 500, 615  
 glMapBufferRange(), 360



- GLMatrixStack, 179, 180, 181, 187
  - GetMatrix(), 179
  - LoadIdentity(), 179
  - LoadMatrix(), 179, 187
  - MultiMatrix(), 187
  - MultMatrix(), 179
  - PopMatrix(), 180
  - PushMatrix(), 180, 187
- glMultiDrawArrays(), 486, 487
- glMultiDrawElements(), 486
- glMultiDrawElementsBaseVertex(), 483
- glPixelStore(), 199
- glPixelStoref(), 199
- glPixelStorei(), 199
- glPointSize(), 314
- glPointSize(), 116, 117, 310
- glPolygonMode(), 130, 132, 192, 263
- glPolygonOffset(), 121, 133
- glPopAttrib(), 600
- glPrimitiveRestartIndex(), 488
- glProvokingVertex(), 259
- glPushAttrib(), 600
- glQueryCounter(), 476
- glReadBuffer(), 203, 210, 331, 344, 347
- glReadPixels(), 200, 203, 327, 329, 330, 331, 332, 333
  - asynchroniczne wywołania, 330
- glRenderbufferStorage(), 340, 366
- glRenderbufferStorageMultisample(), 340, 380
- glRotate(), 180
- glSampleCoverage(), 144, 392
- glSampleMaski(), 392
- glScissor(), 135, 390
- glShaderBinaryOES(), 612
- GLShaderManager, 99, 113, 124, 179, 248
  - UseStockShader(), 113, 114, 165
- GLShaderManager.h, 92
- glShaderSource(), 612
- GLshort, 75, 76
- GLsizei, 75
- GLsizei\_ptr, 75
- GLSL, 108, 242, 612
  - #version, 249
  - anulowanie przetwarzania fragmentów, 289
  - atrybuty, 249
  - centroid, 247
  - const, 246, 247
  - deklaracja atrybutów, 249
  - deklaracja danych wyjściowych, 250
  - dołączanie, 256
  - funkcje, 243, 265, 267
  - in, 246, 247
  - in centroid, 247
  - inout, 247
  - interpolacja nieperspektywiczna, 247
  - kompilacja, 252, 256
  - konsolidacja shaderów, 257
  - kwalifikatory zmiennych, 246, 247
  - liczby całkowite, 243
  - liczby zmiennoprzecinkowe, 243
  - macierze, 245
  - macierze uniform, 262
  - mnożenie macierzowego typu danych
    - przez wektorowy, 264
  - noperspective, 247, 250
  - odwołanie do elementów wektora, 244
  - określanie atrybutów, 254
  - out, 246, 247
  - out centroid, 247
  - płaski shader, 262
  - pobieranie kodu źródłowego, 255
  - przetwarzanie fragmentów, 250
  - przetwarzanie wierzchołków, 250
  - rgba, 244
  - shader fragmentów, 250
  - shadery, 248
  - smooth, 250
  - stpq, 244, 245
  - swizzling, 245
  - tablice uniform, 261
  - teksturowanie w stylu kreskówkowym, 292
  - tekstury, 285
  - typy danych, 243
  - typy macierzowe, 245
  - typy wbudowane, 244
  - typy wektorowe, 244
  - uniform, 246, 247, 259, 260
  - użycie shadera, 258
  - wartości logiczne, 243
  - wektory, 244
  - wersje języka, 249
  - wiązanie, 252, 256
  - wiązanie lokalizacji atrybutów, 257
  - wierzchołek prowokujący, 259
  - xyzw, 244
  - zmiennie, 243
  - zmiennie uniform, 259
  - zmiennie uniform skalarne, 260
  - zmiennie uniform wektorowe, 260
  - zmiennie wektorowe, 244
  - zmiennie wyjściowe, 246
  - znajdowanie danych uniform, 260

- GLSL 1.3, 249
- GLSL 1.4, 249
- GLSL 1.5, 249
- glStencilFunc(), 399
- glStencilFuncSeparate(), 397, 399
- glStencilOp(), 399
- glStencilOpSeparate(), 397, 399
- GLsync, 75
- GLT\_ATTRIBUTE\_COLOR, 113, 114
- GLT\_ATTRIBUTE\_NORMAL, 113, 114, 115
- GLT\_ATTRIBUTE\_TEXTURE0, 113, 115, 302
- GLT\_ATTRIBUTE\_TEXTURE1, 113
- GLT\_ATTRIBUTE\_VERTEX, 113, 114, 115, 257
- GLT\_SHADER\_ATTRIBUTE, 255
- GLT\_SHADER\_IDENTITY, 101
- GLT\_SHADER\_TEXTURE\_POINT\_LIGHT\_DIFF, 223
- glTexBuffer(), 337, 415
- glTexImage(), 208, 210, 224, 236, 237, 239
- glTexImage1D(), 208, 387
- glTexImage2D(), 208, 301, 321, 322, 387, 637
- glTexImage2DMultisample(), 380
- glTexImage3D(), 208, 318
- glTexImage3DMultisample(), 380
- glTexParameterf(), 214
- glTexParameterfv(), 215
- glTexParameteri(), 214
- glTexParameteriv(), 215
- glTexParameter(), 214
- glTexParameterf(), 235
- glTexParameterfv(), 218
- glTexParameteri(), 215, 216, 224, 234, 318
- glTexSubImage(), 210, 239
- glTexSubImage1D(), 210
- glTexSubImage2D(), 210
- glTexSubImage3D(), 210
- glTexImage(), 211
- glTexImage2D(), 296, 333
- glTexParameter(), 211
- glTextSubImage(), 211
- glGetProcAddress(), 65
- glIsExtSupported(), 65, 235
- glLoadShaderFile(), 255
- glLoadShaderPairWithAttributes(), 252, 254, 255, 258
- gltMakeCube(), 302
- gltMakeCylinder(), 169
- gltMakeDisk(), 170
- gltMakeSphere(), 168
- gltMakeTorus(), 169
- GLTools, 65, 73, 79, 80, 84, 203, 249, 569, 632
- GLTools.h, 92
- GLTransformationPipeline, 275
- glTransformFeedbackVaryings(), 501, 502, 505
- glTreadTGABits(), 205
- GLTriangleBatch, 167, 255, 258
  - AddTriangle(), 167
- gltSetWorkingDirectory(), 93, 636
- gltWriteTGA(), 203
- GLubyte, 75
- GLuint, 75, 219
- GLuint64, 75
- glUniform(), 260, 613
- glUniform1f(), 260, 261
- glUniform1fv(), 261, 262
- glUniform1i(), 260, 261
- glUniform1iv(), 261
- glUniform2f(), 260
- glUniform2fv(), 261
- glUniform2i(), 260
- glUniform2iv(), 261
- glUniform3f(), 260
- glUniform3fv(), 261
- glUniform3i(), 261
- glUniform3iv(), 261
- glUniform4f(), 260, 261
- glUniform4fv(), 261, 262
- glUniform4i(), 261
- glUniform4iv(), 261
- glUniformBlockBinding(), 462
- glUniformMatrix(), 613
- glUniformMatrix2fv(), 262
- glUniformMatrix3fv(), 262
- glUniformMatrix4fv(), 262
- glUnmapBuffer(), 360, 361, 615
- glupMainLoop(), 538
- glUseProgram(), 258, 613
- GLushort, 75
- GLUT, 72, 86, 87, 93, 532, 539, 560, 586
  - bufor szablonu, 93
  - inicjalizacja biblioteki, 93
  - klawisze specjalne, 101
  - konfiguracja ustawień, 98
  - Linux, 589
  - odświeżanie ekranu, 103
  - pętla komunikatów, 94
  - testy głębi, 93
  - tryb wyświetlania, 93
  - wymiary okna, 94

- GLUT\_DEPTH, 93, 128
- GLUT\_DOUBLE, 93, 128
- GLUT\_MULTISAMPLE, 143
- GLUT\_RGBA, 93, 128
- GLUT\_STENCIL, 93
- glutCreateWindow(), 94
- glutDisplayFunc(), 94
- glutInit(), 93
- glutInitDisplayMode(), 93, 128, 143
- glutInitWindowSize(), 94
- glutMainLoop(), 94
- glutPostRedisplay(), 103
- glutReshapeFunc(), 94, 96
- glutSpecialFunc(), 101
- glutSwapBuffers(), 101
- glVertexAttribDivisor(), 496, 497, 498, 499
- glVertexAttribPointer(), 479, 480, 481, 482, 483, 484, 485, 496, 611
- glViewport(), 96, 352, 604
- glWaitSync(), 525, 526, 527
- GLX, 587, 591
  - aplikacje, 602
  - atrybuty konfiguracji, 593
  - ekrany, 592
  - instalacja, 591
  - konteksty debugowania, 600
  - łańcuchy, 597
  - obiekty widoku, 592
  - okna, 595, 596, 603
  - powierzchnie renderingu, 595
  - rozszerzanie, 597
  - stosowanie kontekstów, 600
  - synchronizacja rysowania, 601
  - tworzenie kontekstów, 598
  - tworzenie okna, 596
  - usuwanie okna, 596
  - wersje, 603
  - zapytania, 602
  - zarządzanie konfiguracjami, 592
  - zarządzanie kontekstem, 597
- GLX\_ACCUM\_ALPHA\_SIZE, 594
- GLX\_ACCUM\_BLUE\_SIZE, 594
- GLX\_ACCUM\_GREEN\_SIZE, 594
- GLX\_ACCUM\_RED\_SIZE, 594
- GLX\_ALPHA\_SIZE, 593
- GLX\_AUX\_BUFFERS, 594
- GLX\_BAD\_ATTRIBUTE, 594
- GLX\_BLUE\_SIZE, 593
- GLX\_BUFFER\_SIZE, 593, 595
- GLX\_COLOR\_INDEX\_TYPE, 598
- GLX\_CONFIG\_CAVEAT, 593, 595
- GLX\_CONTEXT\_COMPATIBILITY\_PROFILE\_BIT\_ARB, 599
- GLX\_CONTEXT\_CORE\_PROFILE\_BIT\_ARB, 599
- GLX\_CONTEXT\_FLAGS\_ARB, 599
- GLX\_CONTEXT\_MAJOR\_VERSION\_ARB, 599
- GLX\_CONTEXT\_MINOR\_VERSION\_ARB, 599
- GLX\_CONTEXT\_PROFILE\_MASK\_ARB, 599, 600
- GLX\_DEPTH\_SIZE, 593
- GLX\_DOUBLEBUFFER, 593, 595
- GLX\_DRAWABLE\_TYPE, 593, 595
- GLX\_EXTENSIONS, 597
- GLX\_FBCONFIG\_ID, 593, 602
- GLX\_GREEN\_SIZE, 593
- GLX\_HEIGHT, 602
- GLX\_LARGEST\_PBUFFER, 602
- GLX\_LEVEL, 593
- GLX\_RESERVED\_CONTENTS, 602
- GLX\_RED\_SIZE, 593
- GLX\_RENDER\_TYPE, 593, 595, 602
- GLX\_RGBA\_TYPE, 598
- GLX\_SAMPLE\_BUFFERS, 593
- GLX\_SAMPLES, 593
- GLX\_SCREEN, 602
- GLX\_STENCIL\_SIZE, 593
- GLX\_STEREO, 593
- GLX\_TRANSPARENT\_ALPHA\_VALUE, 594
- GLX\_TRANSPARENT\_BLUE\_VALUE, 594
- GLX\_TRANSPARENT\_GREEN\_VALUE, 594
- GLX\_TRANSPARENT\_INDEX\_VALUE, 594
- GLX\_TRANSPARENT\_RED\_VALUE, 594
- GLX\_TRANSPARENT\_TYPE, 593
- GLX\_VENDOR, 597
- GLX\_VERSION, 597
- GLX\_VISUAL\_ID, 593
- GLX\_WIDTH, 602
- GLX\_WINDOW\_BIT, 595
- GLX\_X\_RENDERABLE, 593
- GLX\_X\_VISUAL\_TYPE, 593
- GLXBadMatch, 600
- GLXBadProfileARB, 600
- glXChooseFBConfig(), 594, 595
- glXChooseFBConfigs(), 595
- glXCopyContext(), 600
- glXCreateContextAttribsARB(), 598, 599
- glXCreateNewContext(), 598
- glXCreateWindow(), 596
- glXDestroyContext(), 600
- glXDestroyWindow(), 596
- glXGetClientString(), 597
- glXGetCurrentContext(), 602
- glXGetCurrentDisplay(), 602

glXGetCurrentDrawable(), 602  
 glXGetCurrentReadDrawable(), 602  
 glXGetFBConfigAttrib(), 594  
 glXGetFBConfigs(), 592  
 glXGetProcAddress(), 597  
 glXGetServerString(), 597  
 glXGetVisualFromFBConfig(), 595, 603  
 glXinfo, 587  
 glXIsDirect(), 600  
 glXMakeContextCurrent(), 601  
 glXQueryContext(), 602  
 glXQueryDrawable(), 602  
 glXQueryExtensionsString(), 597  
 glXSwapBuffers(), 601, 605  
 glXWaitGL(), 601  
 glXWaitX(), 601  
 głębia, 42, 521  
 Gouraud shading, 274  
 GPGPU, 384  
 GPU, 50, 384  
 grafika dwuwymiarowa, 41, 63  
 grafika komputerowa, 40  
 grafika komputerowa czasu rzeczywistego, 41  
 grafika trójwymiarowa, 41, 47, 48  
 grafika trójwymiarowa czasu rzeczywistego, 47  
 Graphics Device Interface, 538  
 graphics processing unit, 50  
 grawitacja, 413  
 greaterThan(), 269  
 greaterThanEqual(), 269  
 grubość linii, 119  
 grupowanie w stada, 511
 

- etapy wykonywania algorytmu, 512
- inicjalizacja atrybutów, 514
- inicjalizacja struktur danych, 513
- pętla renderująca, 515
- przekształcenie zwrotne, 514, 516
- shader wierzchołków, 516, 518

 gry OpenGL ES, 615  
 GUI, 48

## H

HDR, 366, 367, 382  
 hdr\_adaptive, 373  
 hdr\_exposure, 376  
 hdr\_imaging, 372  
 heads-up display, 475  
 hidden surface removal, 45  
 high dynamic range, 366  
 HUD, 475

## I

IBM\_, 66  
 ICD, 533  
 IDE, 84  
 identity shader, 114  
 if-else, 452  
 ignorowanie zadań w shaderze fragmentów, 445  
 iloczyn skalarny, 151, 453  
 iloczyn wektorowy, 152, 453  
 iloczyn zewnętrzny wektorów, 453  
 iluzja głębi, 42  
 immediate mode, 221  
 immersive environment, 189  
 implementacja OpenGL, 64, 67  
 implementacje OpenGL w systemie Windows, 532
 

- GLEW, 537
- ICD, 533
- OpenGL firmy Microsoft, 533
- opengl32.dll, 532
- rozszerzenia OpenGL, 534
- rozszerzenia WGL, 536
- sterowniki graficzne, 533
- Windows 7, 534
- Windows Vista, 534

 in, 110, 246, 247, 249  
 in centroid, 247  
 indeksowanie tablicy tekstur, 319  
 indeksy wierzchołków, 482  
 inicjalizacja
 

- biblioteka GLEW, 94
- biblioteka GLUT, 93
- bufor pikseli, 332
- kontekst renderingu, 554
- obiekt VBO, 479

 InitializeStockShaders(), 99, 113  
 inout, 247  
 instalacja
 

- GLUT, 590
- GLX, 591

 instalowalny sterownik klienta, 533  
 instanced array, 491, 496  
 instanced rendering, 490  
 intBitsToFloat(), 272, 453  
 interface block, 421  
 Interface Builder, 562, 563  
 interfejs Cocoa, 561  
 interleaved, 505  
 interleaved attributes, 480

interpolacja, 448  
 interpolacja bez korekty perspektywy, 247, 451  
 interpolacja liniowa, 452  
 interpolacja z korektą perspektywiczną, 452  
 inverse(), 268, 453  
 inversesqrt(), 266  
 iPad, 629  
 iPhone, 629  
 iPod Touch, 629  
 IRIS GL, 63, 586  
 isinf(), 271  
 iskrzenie, 223  
 isnan(), 271  
 ivec2, 244  
 ivec3, 244  
 ivec4, 244

**J**

jednostka grupy Khronos, 65  
 język GLSL, 108, 242  
 język OpenGL ES Shading Language, 612

**K**

kalkowanie, 132  
 kamera, 185  
 obsługa, 188  
 kanał alfa, 297  
 kartezyjański układ współrzędnych, 95  
 katalog roboczy, 93  
 kąty Eulera, 187  
 kCGLCESurfaceBackingSize, 583  
 kCGLEMPEngine, 583  
 Khronos Group, 64, 609  
 kineskop, 41  
 klasy  
 GLBatch, 101, 124, 125, 221, 249  
 GLFrame, 186, 191  
 GLFrustum, 111, 172  
 GLGeometryTransform, 180, 181  
 GLMatrixStack, 179, 180, 181, 187  
 GLShaderManager, 113, 124, 248  
 GLTransformationPipeline, 275  
 GLTriangleBatch, 167  
 klawisze specjalne, 101  
 klient, 107  
 klient-serwer, 107  
 kod shaderów, 242  
 kody błędów, 76  
 kolorowanie zbioru Julii, 445  
 kolory, 98, 99  
 kompilacja shadera, 252, 256  
 kompletność bufora obrazu, 344, 346  
 bufor odczytu obrazu, 347  
 kompletność dowiązaniowa, 344  
 ogólna kompletność bufora, 345  
 sprawdzanie bufora obrazu, 345  
 kompletność dowiązaniowa, 344  
 kompresja tekstur, 236, 386  
 ETC1, 386  
 formaty kompresji, 239, 386  
 formaty tekstur, 237  
 formaty tekstur o wspólnym wykładniku, 387  
 GL\_EXT\_texture\_compression\_s3tc, 239  
 GL\_TEXTURE\_COMPRESSED, 238  
 ładowanie tekstur, 239, 387  
 RGTC, 386  
 S3TC, 386  
 stosowanie, 387  
 komunikacja aplikacji OpenGL z X Window, 591  
 konfiguracja atrybutów egzemplarzowych, 498  
 konfiguracja Mesa3D, 588  
 konfiguracja środowiska programistycznego  
 system Mac OS X, 84  
 system Windows, 79  
 konfiguracja układu współrzędnych, 110  
 konkatencja, 165  
 konsolidacja shaderów, 257  
 kontekst bieżący, 549  
 kontekst CGL, 581  
 kontekst debugowania, 548  
 kontekst renderingu, 94, 538  
 inicjalizowanie, 554  
 WGL, 547  
 wyłączenie, 554  
 kontekst urządzenia GDI, 538, 554  
 kontekst urządzenia okna, 540  
 kontener FBO, 339  
 kontener porcji danych, 124  
 kontrola głębi poszczególnych fragmentów, 447  
 konwersja HDR na LDR, 369, 371  
 konwersja kolorów, 385  
 kopiowanie buforów, 361  
 kopiowanie danych w buforach obrazu, 347  
 blit, 347  
 korekcja kolorów, 438  
 korzystanie z biblioteki OpenGL, 71  
 kula, 168  
 kurz, 367

kwalifikatory pamięci, 448  
 centroid, 448  
 flat, 448  
 noperspective, 452  
 kwalifikatory układu shadera geometrii, 418  
 kwalifikatory zmiennych, 246  
 kwaterniony, 188

## L

LDR, 369  
 length(), 267, 453  
 lens flare, 367  
 lessThan(), 269  
 lessThanEqual(), 269, 453  
 licencjonowanie implementacji biblioteki OpenGL, 67  
 liczby  
 całkowite, 243, 383  
 zmiennoprzecinkowe, 243  
 liczenie wierzchołków przy użyciu zapytań obiektów  
 podstawowych, 507  
 light bloom, 367, 374  
 lines, 420  
 lines\_adjacency, 420  
 linie, 116, 118  
 Linux, 586  
 aplikacje OpenGL, 587, 590, 602  
 fre glut, 589  
 GLEW, 589, 590  
 GLUT, 589  
 GLX, 591  
 instalacja biblioteki GLEW, 590  
 instalacja biblioteki GLUT, 590  
 konfiguracja biblioteki GLEW, 589  
 konfiguracja biblioteki GLUT, 589  
 konfiguracja Mesa3D, 588  
 konfiguracja sterowników sprzętowych, 589  
 Mesa3D, 588  
 OpenGL, 586  
 sesje X Window, 586  
 sprawdzanie obsługi OpenGL, 587  
 X Window, 586  
 XFree86, 587  
 LitTexture, 288  
 LoadIdentity(), 179  
 LoadMatrix(), 179  
 LoadTGATexture(), 219, 220, 297  
 LoadTGATextureRect(), 297  
 log(), 266  
 log2(), 266  
 luminancja, 200  
 Lunar Lander, 41

## Ł

ładowanie  
 tablica tekstur dwuwymiarowych, 317  
 tekstury, 208  
 tekstury skompresowane, 239, 387  
 trójkąty, 100  
 łamane zamknięte, 119  
 łańcuchy EGL, 627  
 łańcuchy GLX, 597  
 łączenie funkcji rysujących, 486  
 łączenie geometrii poprzez restart  
 obiektów podstawowych, 487  
 łączenie kolorów, 136  
 łączenie przekształceń, 164  
 łączenie punktów, 47, 115  
 łąka, 492

## M

m3dCrossProduct3(), 152  
 m3dDegToRad(), 164  
 m3dDotProduct3(), 151  
 m3dGetAngleBetweenVectors3(), 151  
 m3dLoadIdentity44(), 162  
 m3dMakeOrthographicMatrix(), 298  
 M3DMatrix33f, 153  
 M3DMatrix44f, 153, 180  
 m3dMatrixMultiply44(), 165, 166, 177  
 m3dRotationMatrix(), 163  
 m3dRotationMatrix44(), 163, 177  
 m3dScaleMatrix44(), 164  
 m3dTransformVector4(), 192, 193  
 m3dTranslationMatrix44(), 162, 165, 177  
 M3DVector3f, 150  
 M3DVector4f, 150  
 Mac OS X, 84, 560  
 buforowanie, 568  
 CGL, 581  
 Cocoa, 561  
 GLTools, 569  
 GLUT, 560  
 interfejsy OpenGL, 560  
 Objective-C++, 569  
 OpenGL, 561  
 renderowanie pełnoekranowe, 573  
 SphereWorld, 569, 570  
 tworzenie aplikacji Cocoa, 561  
 macierz model-widok, 157, 160, 276  
 łączenie przekształceń, 164  
 macierz jednostkowa, 162

- macierz model-widok
  - obrót, 163
  - przesunięcie, 162
  - skalowanie, 164
  - stosowanie, 165
  - tworzenie, 160
- macierz rzutowania, 44, 166, 171
  - macierz rzutowania model-widok, 174
  - modyfikacja potoku, 180
  - potok przekształceń wierzchołków, 178
  - rzutowanie perspektywiczne, 172
  - rzutowanie prostopadłe, 171, 298
  - stos macierzy, 179
- macierze, 152
  - GLSL, 245
  - M3DMatrix33f, 153
  - M3DMatrix44f, 153
  - macierz dwuwymiarowa, 154
  - macierz jednostkowa, 162, 246
  - macierz kamery, 189
  - macierz normalna, 193, 275, 424
  - macierz obrotu, 163
  - macierz przekształcenia, 44
  - macierz przekształcenia kolorów, 439
  - macierz przesunięcia, 162
  - macierz rozmycia gaussowskiego, 375
  - macierz skalowania, 164
  - macierz uniform, 262
  - operacje, 153
  - porządek kolumnowy macierzy, 154
  - transponowanie, 160
  - typy danych, 245
- magnification filter, 215
- main(), 93
- make, 590
- MakePyramid(), 220
- maksymalny rozmiar tekstu, 321
- malloc(), 205
- mapowanie buforów, 360, 615
  - sposoby mapowania, 361
- mapowanie cieni, 213, 338
- mapy bitowe, 196
- mapy sześciennie, 300
- maska próbki, 392
- maski, 407
- maskowanie wyniku, 405
  - maskowanie buforów szablonu, 407
  - maskowanie głębi, 406
  - maskowanie koloru, 406
  - stosowanie masek, 407
- maszyna stanów OpenGL, 78
- mat2, 246
- mat2x2, 246
- mat2x3, 246
- mat2x4, 246
- mat3, 246
- mat3x2, 246
- mat3x3, 246
- mat3x4, 246
- mat4, 246
- mat4x2, 246
- mat4x3, 246
- mat4x4, 246
- matematyka, 149
- Math3D, 111
- math3d.cpp, 153
- math3d.h, 153
- matrix stack, 179
- matrixCompMult(), 268, 453
- max(), 270, 276
- Maya, 365
- mechanizm rozszerzeń, 64
- mechanizm wielopróbkowania, 143
- Mesa3D, 588
- mierzenie czasu wykonywania poleceń, 475
- mieszanie addytywne, 312
- mieszanie kolorów, 47, 71, 135, 400
  - funkcja mieszania, 401
  - łączenie kolorów, 136
  - równanie mieszania, 136, 401
  - tryby równań mieszania kolorów, 139
  - włączanie, 136
  - współczynniki mieszania, 137
  - zmiana równania mieszania, 139
- min(), 270
- minification filter, 215
- mipmapowanie, 224
- mipmapy, 223
  - filtrowanie mipmap, 224
  - generowanie poziomów mipmap, 226
  - poziomy, 223
  - stosowanie, 226
- mix(), 271
- mnożenie macierzowego typu danych przez wektorowy, 264
- mnożenie macierzy, 179
- mod(), 270
- model oświetlenia ADS, 278
  - shader ADS, 280
  - światło odbicia zwierciadlanego, 279

światło otaczające, 278  
 światło rozproszone, 279  
 modele użycia buforów, 329  
 modeling, 154  
 modelowanie, 154
 

- przekształcenia geometryczne, 156

 modelview, 157  
 ModelviewProjection, 174  
 model-widok, 157, 160  
 modf(), 270  
 modyfikacja geometrii w shaderze geometrii, 426  
 modyfikacja potoku, 180  
 MSAA, 380, 391  
 MultiMatrix(), 179  
 multisample antyaliasing, 380  
 multisampling, 143, 379  
 multiteksturowanie, 305, 306
 

- interpolacja współrzędnych tekstur, 306
- shader odbicia, 308
- wiele współrzędnych tekstur, 306

 MultiTexCoord2f(), 221  
 Multitexture, 307

## N

nachylenie, 187  
 nadawanie punktom kształtów, 314  
 nagłówki, 92  
 nakładanie tekstur, 212
 

- dopasowanie tekstury do obiektu geometrycznego, 214
- filtrowanie, 215
- parametry tekstur, 214
- współrzędne tekstury, 212
- zawijanie tekstury, 217

 nakładki HUD, 475  
 napełnianie buforów, 328  
 narzędzia pomocnicze, 72  
 nawinięcie, 121
 

- nawinięcie przeciwne do ruchu wskazówek zegara, 122
- nawinięcie zgodne z ruchem wskazówek zegara, 122

 nazwany blok zmiennych jednorodnych, 454  
 nazwy rozszerzeń OpenGL, 66  
 New Project, 85  
 New Project Assistant, 561  
 NEXTSTEP Interface Builder, 561  
 NIB, 561  
 niechciana geometria, 126
 

- algorytm malarza, 126
- bufor głębi, 128
- testowanie głębi, 128
- tryby wielokątów, 130
- usuwanie płaszczyzn, 127
- usuwanie płaszczyzn przednich, 128
- usuwanie płaszczyzn tylnych, 127

 nierównomierne skalowanie sześcianu, 164  
 niskopoziomowe API, 52  
 noperspective, 247, 250, 448, 452  
 normal matrix, 275  
 Normal3f(), 221  
 normalizacja wektorów, 150, 453  
 normalize(), 267, 453  
 normalna do powierzchni, 273, 275  
 not(), 269  
 notEqual(), 269, 453  
 NSBorderlessWindowMask, 579  
 NSOpenGL, 560  
 NSOpenGLPFAAccelerated, 576  
 NSOpenGLPFAAcceleratedCompute, 577  
 NSOpenGLPFAAccumSize, 576  
 NSOpenGLPFAAAllowOffLineRenderers, 577  
 NSOpenGLPFAAAllRenderers, 575  
 NSOpenGLPFAAAlphaSize, 575  
 NSOpenGLPFAAuxBuffers, 575  
 NSOpenGLPFAAuxDepthStencil, 576  
 NSOpenGLPFABackingStore, 576  
 NSOpenGLPFAClosestPolicy, 576  
 NSOpenGLPFAColorFloat, 576  
 NSOpenGLPFAColorSize, 575  
 NSOpenGLPFACompliant, 577  
 NSOpenGLPFADepthSize, 575  
 NSOpenGLPFADoubleBuffer, 575  
 NSOpenGLPFAScreen, 576, 579  
 NSOpenGLPFAMaximumPolicy, 576  
 NSOpenGLPFAMinimumPolicy, 576  
 NSOpenGLPFAMPSafe, 576  
 NSOpenGLPFAMultisample, 576  
 NSOpenGLPFAMultiScreen, 577  
 NSOpenGLPFANoRecovery, 576  
 NSOpenGLPFAOffScreen, 576  
 NSOpenGLPFAPixelBuffer, 577  
 NSOpenGLPFARemotePixelBuffer, 577  
 NSOpenGLPFARendererID, 576  
 NSOpenGLPFARobust, 576  
 NSOpenGLPFASampleAlpha, 576  
 NSOpenGLPFASampleBuffers, 576  
 NSOpenGLPFASamples, 576  
 NSOpenGLPFAScreenMask, 577, 579  
 NSOpenGLPFASingleRenderer, 576  
 NSOpenGLPFAStencilSize, 576  
 NSOpenGLPFAStereo, 575



NSOpenGLPFASupersample, 576  
 NSOpenGLPFAVirtualScreenCount, 577  
 NSOpenGLPFAWindow, 577  
 NSOpenGLPixelFormat, 574  
 NSOpenGLView, 562, 565, 579  
 nTextureUnit, 115  
 NV\_, 66

## O

obcinanie głębi, 521  
 obiekt bufora bloku zmiennych jednorodnych, 454  
 obiekt bufora obrazu, 338, 353
 

- sposób użycia, 339
- tworzenie, 339, 353
- usuwanie, 339

 obiekt bufora pikseli, 329  
 obiekt bufora renderowania, 339, 340, 353
 

- dołączanie obiektów, 340
- rozmiar obiektów, 341
- tworzenie, 340
- wiązanie, 340

 obiekt buforowy wierzchołków, 478  
 obiekt synchronizacji, 523
 

- limit czasu, 525
- oczekiwanie na obiekt synchronizacji, 524
- stan niezasygnalizowany, 523
- stan zasygnalizowany, 523, 524
- usuwanie, 527

 obiekt tablicy wierzchołków, 478, 483, 484  
 obiekt TBO, 337  
 obiekt tekstur, 211  
 obiekt zapytaniowy, 467  
 obiekty podstawowe, 56, 71, 106, 116
 

- ciągi linii, 119
  - GL\_LINE\_LOOP, 116, 120
  - GL\_LINE\_STRIP, 116, 119
  - GL\_LINES, 116
  - GL\_POINTS, 116
  - GL\_TRIANGLE\_FAN, 116, 123
  - GL\_TRIANGLE\_STRIP, 116, 123
  - GL\_TRIANGLES, 116
- linie, 118
  - łamane zamknięte, 119
  - punkty, 116

 Objective-C++, 569  
 obracanie punktów, 315  
 obraz komputerowy, 43  
 obrazy HDR, 367  
 obrót, 163  
 obsługa kamery, 188  
 occlusion query, 358, 466  
 oczekiwanie na obiekt synchronizacji, 524  
 odbicia światła, 367  
 odbicie, 47, 304  
 odbłysek światła od soczewek, 367  
 odbłyски, 279  
 odchylenie, 187  
 odczytywanie danych pikseli z bufora, 331  
 odświeżanie ekranu, 103  
 odwiązanie bufora od punktu wiązania, 328  
 odwrotność macierzy, 453  
 odwzorowywanie buforów, 343  
 odwzorowywanie danych wyjściowych shadera
 

- na bufory, 343

 odwzorowywanie fragmentów wyjściowych, 362  
 odwzorowywanie tekstur, 196  
 odwzorowywanie tonów, 369, 382  
 OES\_compressed\_ETC1\_RGB8\_texture, 615  
 OES\_element\_index\_uint, 615  
 OES\_fragment\_precision\_high, 615  
 OES\_mapbuffer, 615  
 OES\_texture\_3D, 615  
 OES\_texture\_float, 614  
 OES\_texture\_float\_linear, 614  
 OES\_texture\_half\_float, 614  
 OES\_texture\_half\_float\_linear, 614  
 OES\_vertex\_half\_float, 614  
 ogólna kompletność bufora, 345  
 ograniczanie wartości głębi, 400  
 ograniczanie współrzędnych, 53  
 OIT, 393  
 okna, 550
 

- GLX, 596

 okrawanie, 390  
 określanie wiązań dla bloków zmiennych
 

- jednorodnych, 463

 określanie wierzchołków, 100  
 określanie współrzędnych tekstury, 220  
 OpenEXR, 367, 368  
 OpenGL, 23, 31, 38, 40, 48, 52, 56, 62, 87, 196
 

- funkcje wycofywane, 69
- implementacje w systemie Windows, 532
- mechanizm rozszerzeń, 64
- przyszłość, 67

 OpenGL 1.1, 535  
 OpenGL 1.5, 610  
 OpenGL 2.0, 67  
 OpenGL 3.0, 70, 79  
 OpenGL 3.1, 70, 249  
 OpenGL 3.2, 70, 249  
 OpenGL 3.3, 73

- OpenGL ARB, 63, 64, 66
  - OpenGL ES, 608, 620
    - działania na liczbach stałoprzecinkowych, 617
    - kwestie projektowe, 616
    - rozszerzenia producentów, 628
    - rozwiązywanie problemów z ograniczeniami, 616
    - środowiska układów wbudowanych, 615
    - wersje, 609, 610
    - wybór wersji, 611
  - OpenGL ES 1.0, 609
  - OpenGL ES 2.0, 610, 611
    - bufory obrazu, 614
    - całkowitoliczbowe indeksy elementów
      - bez znaku, 615
    - dotatki do rdzenia, 614
    - kolorowanie wierzchołków, 611
    - mapowanie buforów, 615
    - OES\_compressed\_ETC1\_RGB8\_texture, 615
    - OES\_element\_index\_uint, 615
    - OES\_fragment\_precision\_high, 615
    - OES\_mapbuffer, 615
    - OES\_texture\_3D, 615
    - OES\_texture\_float, 614
    - OES\_texture\_half\_float, 614
    - OES\_vertex\_half\_float, 614
    - platformy przenośne firmy Apple, 629
    - przetwarzanie wierzchołków, 611
    - rasteryzacja, 613
    - shadery, 612
    - skompresowany format teksturowy Ericssona, 615
    - stan, 614
    - teksturowanie, 613
    - tekstury trójwymiarowe, 615
    - tekstury zmiennoprzecinkowe, 614
    - wartości całkowite wysokiej precyzji
      - w shaderach fragmentów, 615
    - wartości zmiennoprzecinkowe wysokiej precyzji
      - w shaderach fragmentów, 615
    - zmiennoprzecinkowy format wierzchołków
      - połowy precyzji, 614
  - OpenGL ES Application, 629
  - OpenGL ES SC 1.0, 610
  - OpenGL ES Shading Language, 612
  - OpenGL extension wrangler, 73
  - OpenGL firmy Microsoft, 533
  - OpenGL GLU, 74
  - OpenGL Shading Language, 242
  - OpenGL Shading Language Specification, 242
  - OpenGL Specification, 64
  - OpenGL Utility Library, 72
  - OpenGL Utility Toolkit, 72
  - OpenGL Working Group, 65
  - opengl32.dll, 532, 533, 534
  - OpenVG, 620
  - operacja rozwiązywania, 379
  - operacje logiczne, 405, 406
  - operacje na fragmentach, 390
    - maskowanie wyniku, 405
    - mieszanie kolorów, 400
    - operacje logiczne, 405
    - rozsiewanie kolorów, 404
    - test nożyc, 390
    - test szablonu, 397
    - testowanie głębi, 400
    - wielopróbkowanie, 391
  - operacje na pikselach, 390
  - operacje na szablonach, 397
  - operacje testu szablonu, 398
  - operatory, 453
  - oprogramowanie ogólnodostępne, 72
  - optymalizacja renderowania, 134
    - rysowanie dużych ilości geometrii, 486
  - order independent transparency, 393
  - organizowanie buforów, 483
  - Orthographic, 171
  - ortogonalne rzutowanie, 57
  - orzekanie, 473
  - osie, 53
  - oświetlenie, 192, 272
  - oświetlenie tekstei, 287
  - oświetlenie wierzchołków, 273, 274
  - out, 110, 246, 247, 365
  - out centroid, 247
  - outerProduct(), 268, 453
  - ożywianie sceny, 101
- P**
- painter's algorithm, 126
  - pakowanie pikseli, 196
  - pamięć tekstur, 320
  - parametry punktów, 314
  - parametry tekstur, 214
  - particle, 309
  - path tracing, 453
  - PBO, 329, 330, 335
  - Perspective, 171
  - perspective projection, 58
  - perspektywa, 42, 43, 158
  - pętla programu głównego, 94
  - pętle, 452
  - Phong shading, 282

- piksele, 115
- piksmapy, 199
- pipeline stall, 108
- pitch, 187
- pix\_buffs, 332
- pixel buffer, 100
- pixel buffer object, 329
- PIXELFORMATDESCRIPTOR, 544, 546
- platforma .NET, 538
- platformy przenośne firmy Apple, 629
- pliki make, 590
- plaski shader, 262
- płaszczyzna, 53
- płaszczyzna kartezjańska, 53
- płaszczyzny obcinania, 519
  - definiowanie, 519
  - gl\_Clip\_Distance, 519, 520
  - obcinanie głębi, 521
- pobieranie wskazówek, 77
- pobieranie wyników zapytania, 468
- pochylenie, 187
- początek układu kartezjańskiego, 53
- podejmowanie decyzji, 472
- podpiksele, 379
- podstawowe shadery, 113
- podwójne buforowanie, 93, 539, 556
  - Mac OS X, 568
  - WGL, 556
  - Windows, 556
- podwójny bufor, 93
- point light shader, 115
- point sprite, 309
- points, 420
- PointSprite, 311
- pojedynczy trójkąt, 120
- położenie światła, 192
- położenie w przestrzeni, 56
- pomocnicza biblioteka OpenGL, 72
- Pong, 41
- PopMatrix(), 180, 183
- poprawianie wydajności renderowania, 134
- porządek kolumnowy macierzy, 154, 160, 161
- porządek wierszowy macierzy, 160
- poświata, 367, 374, 376
- potok graficzny, 106
  - klient, 107
  - programy do cieniowania, 108
  - serwer, 107
  - zator potoku, 108
- potok przekształceń wierzchołków, 178
- pow(), 266
- powierzchnia pokrycia próbki, 391
- poziomy mipmap, 223
- prawa Newtona, 412
- prawo Hooke'a, 412, 413
- predication, 473
- primitive restart, 488
- primitives, 56, 71, 106
- proces renderowania trójkąta, 107
- procesor GPU, 384
- profil rdzenny, 23
- profil zgodnościowy, 23
- programowalne cieniowanie, 50
- programowanie grafiki trójwymiarowej, 38, 52
- programy do cieniowania, 23, 46, 50, 99, 108, 112
  - atrybuty, 109, 113
  - attribute, 109
  - domyślny shader oświetlenia, 114
  - GLSL, 242
  - in, 110
  - kod shaderów, 242
  - out, 110
  - shader cieniowany, 114
  - shader jednostkowy, 114
  - shader modulacji tekstury, 115
  - shader oświetlenia punktowego, 115
  - shader płaski, 114
  - shader punktowego oświetlenia tekstury, 115
  - shader tożsamościowy, 114
  - shader wymiany tekstury, 115
  - shadery, 242
  - tekstury, 110
  - uniform, 109, 113
- programy do cieniowania geometrii, 108
- projection, 57, 154
- projection matrix, 44, 166, 171
- projekt, 81, 85
  - aplikacje dla iPhone'a, 629
  - Xcode, 566
- promienie zmierzchu, 367
- promienistość, 338
- prostokąt okrawający, 134, 135
- provoking vertex, 259
- próbkowanie tekstury, 285
- próbkowanie z uwzględnieniem środka masy, 448
  - wykrywanie krawędzi, 450
- przechowywanie danych w buforach danych wierzchołków, 478
- przechowywanie danych w pamięci GPU, 477
- przechowywanie indeksów wierzchołków w buforach, 482

- przechowywanie przekształconych wierzchołków, 500
  - przechowywanie przeplatanych atrybutów, 480
  - przekształcanie, 44
  - przekształcenia afiniczne, 180
  - przekształcenia geometryczne, 148, 154
    - dwoistość model-widok, 157
    - macierz model-widok, 157
    - modelowanie, 154, 156
    - punkt widzenia, 154, 155
    - rzutowanie, 154, 158
    - skrót perspektywiczny, 158
    - widok, 159
    - współrzędne oka, 154
  - przekształcenia światła, 192
  - przekształcenia widoku, 159
  - przekształcenie zwrotne, 500, 501
    - algorytmy rekurencyjne, 510
    - grupowanie w stada, 511
    - stosowanie, 509
    - zapisywanie wyników pośrednich, 509
  - przelotowy shader wierzchołków, 428
  - przestrzeń kartezjańska, 96
  - przestrzeń kolorów
    - RGB, 98
    - sRGB, 384
  - przestrzeń międzygwiazdowa, 312
  - przestrzeń ograniczająca, 97
  - przestrzeń widoczna, 57
  - przesunięcie, 162
  - przesuwanie wielokątów, 131
  - przetwarzanie fragmentów, 250
  - przetwarzanie końcowe, 438
    - korekcja kolorów, 438
    - splot, 439
  - przetwarzanie wierzchołków, 250
  - przeznaczanie wartości alfa na wartość pokrycia, 391
  - przezroczystość, 138
  - przezroczystość niezależna od kolejności obiektów, 393
  - przycinanie, 518
  - przycinanie geometrii na wymiar, 390
  - przygotowywanie zapytania, 467
  - przypisanie punktu wiązania do bufora zmiennych jednorodnych, 462
  - przyspieszanie operacji wypełniania, 582
  - przyszłość OpenGL, 67
  - public domain, 72
  - puddło nieba, 189, 202
  - punkt świetlny, 192
  - punkt widzenia, 154
    - przekształcenia, 155
    - przekształcenia geometryczne, 155
  - punktowe źródło światła, 194
  - punkty, 116
  - punkty wiązania obiektów buforowych, 327
  - PushMatrix(), 180, 183
  - Pyramid, 219
- ## Q
- quad, 309, 436
  - quaternions, 188
  - query, 466
  - query object, 467
- ## R
- radians(), 266
  - radiosity, 338
  - rasterization, 45
  - rasteryzacja, 45, 97
  - ray tracer, 49
  - ray tracing, 51
  - RBO, 339, 350, 353
  - RC, 94
  - rdzeń biblioteki, 71
  - realizm, 196
  - Red-Green Texture Compression, 386
  - reflect(), 267, 453
  - refract(), 267, 453
  - region ograniczający, 53, 54
  - relacje między punktami wiązania przekształcenia zwrotnego, 504
  - renderbuffer object, 339
  - rendering context, 94
  - renderowanie, 44, 100
  - renderowanie danych wierzchołków, 330
  - renderowanie do FBO, 351
  - renderowanie do tekstur, 353
    - efekt lustra, 354
    - tworzenie obiektu FBO, 353
  - renderowanie egzemplarzowe, 498
  - renderowanie egzemplarzy obiektu, 490
  - renderowanie HDR, 366, 370
    - konwersja HDR na LDR, 369, 371
    - odwzorowywanie tonów, 369
  - OpenEXR, 367, 368
  - poświata, 374
  - RGBAInputFile, 368

- renderowanie pełnoekranowe w Mac OS X, 573
    - applicationDidFinishLaunching(), 577
    - CGDisplayHideCursor(), 579
    - Cocoa, 574
    - formaty pikseli, 574
    - klasa widoku, 580
    - NSBorderlessWindowMask, 579
    - NSOpenGLPFADepthSize, 575
    - NSOpenGLPFADoubleBuffer, 575
    - NSOpenGLPFAScreenMask, 579
    - NSOpenGLPixelFormat, 574
    - NSOpenGLView, 579
    - rdzeń aplikacji, 577
    - rozmiar pulpitu, 579
  - renderowanie pełnoekranowe w Windows, 555
    - konfiguracja pełnoekranowego okna, 555
  - renderowanie trójkąta, 107
  - renderowanie w systemie Windows, 537
  - renderowanie warunkowe, 471, 473
  - renderowanie zbioru Julii, 444
  - renderowanie zbioru Mandelbrota, 443
  - RenderRealObject(), 473
  - RenderScene(), 94, 103, 165, 191
  - RenderSimplifiedObject(), 470, 472, 473
  - resolve shader, 381
  - resolving, 379
  - restart obiektów podstawowych, 487, 488
  - RGB, 98
  - rgba, 244
  - RGBA, 93, 384
  - RGBAInputFile, 368
  - RGTC, 386
  - roll, 187
  - Rotate(), 180, 183
  - round(), 270
  - roundEven(), 270
  - rozmiar obiektów RBO, 341
  - rozmiar punktu, 117, 310
  - rozmycie gaussowskie, 375
  - rozmycie obiektów w ruchu, 332
  - rozsiewanie kolorów, 390, 404
    - zastosowanie, 404
  - rozszerzenia EGL, 627
  - rozszerzenia GLX, 597
  - rozszerzenia OpenGL, 66, 534
  - rozszerzenia WGL, 536
  - równania
    - konwersja kolorów, 385
    - mieszanie kolorów, 136, 401
  - rysowanie, 116
    - rysowanie danych zapisanych w buforze przekształcenia zwrotnego, 509
    - rysowanie dużych ilości geometrii, 486
    - rysowanie linii normalnych, 432
    - rysowanie normalnej do powierzchni, 433
    - rysowanie obiektów, 97
    - rysowanie obiektów za pomocą samych linii, 45
    - rysowanie prostokątów za pomocą prostokąta okrawającego, 135
    - rysowanie pułda nieba, 302
    - rysowanie punktów w trzech wymiarach, 106
    - rysowanie wielu egzemplarzy jednego obiektu, 489
    - rysowanie z perspektywy lustra, 356
  - rysowanie trójkątów w trzech wymiarach, 120
    - niechciana geometria, 126
    - pojedynczy trójkąt, 120
    - trójkąty sklepane, 123
    - wachlarze trójkątów, 123
  - rzutowanie, 44, 57, 154
    - przekształcenia geometryczne, 158
  - rzutowanie ortogonalne, 57, 111, 158
  - rzutowanie perspektywiczne, 58, 112, 158, 172
  - rzutowanie prostopadłe, 158, 171
  - rzutowanie współrzędnych kartezjańskich na piksele, 97
  - rzutowanie współrzędnych na rzeczywiste współrzędne ekranu, 96
  - rzutowanie współrzędnych rysowania na współrzędne okna, 54
  - rzutowanie z trzech w dwa wymiary, 56
- ## S
- S3TC, 386
  - Safety Critical, 610
  - Scale(), 180
  - schemat układu wbudowanego, 619
  - schodkowe krawędzie, 140
  - scintillation, 223
  - scissor box, 135
  - scissor rectangle, 134
  - scissor test, 390
  - Seeker, 51
  - serwer, 107
  - sesje X Window, 586
  - setFrameBuffer(), 368
  - SetMatrixStacks(), 183
  - SetOrthographic(), 111, 172

- SetPerspective(), 112, 173
- SetPixelFormat(), 546
- SetupRC(), 98, 233, 264
- SetupWindow(), 550
- SGL, 63, 586
- SGL\_, 66
- ShadedTriangle, 249, 258
- shader fragmentów, 242, 250, 436
  - ADSPhong, 284
  - anulowanie przetwarzania fragmentów, 289
  - czworokąt, 436
  - czworokąt pokrywający cały ekran, 436
  - discard, 446
  - generowanie danych obrazu, 442
  - gl\_FragDepth, 447
  - ignorowanie zadań, 445
  - kontrola głębi poszczególnych fragmentów, 447
  - korekcja kolorów, 438
  - macierze przekształcania kolorów, 439
  - przetwarzanie końcowe, 438
  - splot, 439
  - testowanie alfa, 446
- shader geometrii, 242, 417
  - algorytm usuwania płaszczyzn tylnych, 424
  - blok interfejsu, 421
  - EmitVertex(), 422
  - EndPrimitive(), 422
  - generowanie geometrii, 427
  - kwalifikatory układu, 418
  - modyfikacja geometrii, 426
  - normalna do powierzchni, 424
  - rozmiary tablic, 422
  - rozsadzanie modelu, 426
  - rysowanie linii normalnych, 432
  - rysowanie normalnej do powierzchni, 433
  - shader przepuszczający dane, 417
  - stosowanie w programie, 419
  - teselacja, 428, 430
  - tryb triangle\_strip, 418
  - tryb triangles, 418
  - tryby rysowania trybów wejściowych, 420
  - tworzenie, 419
  - typy obiektów podstawowych, 433
  - usuwanie geometrii, 423
  - warunkowe tworzenie geometrii, 425
  - wizualizacja normalnych, 432
  - wzmacnianie, 417
  - zmiana typu obiektu podstawowego, 431
- shader wierzchołków, 242, 248, 410, 428, 510
  - ADSPhong, 283
  - fizyczne symulacje, 410
  - shaders, 46, 108
  - shadery, 23, 46, 242, 248
    - architektura, 243
    - dane uniform, 259
    - kompilacja, 256
    - konsolidacja, 257
    - OpenGL ES 2.0, 612
    - shader ADS, 280
    - shader cieniowany, 114
    - shader jednostkowy, 114
    - shader modulacji tekstury, 115
    - shader oświetlenia punktowego, 115
    - shader płaski, 114, 262
    - shader punktowego oświetlenia tekstury, 115
    - shader rozwiązywania, 381
    - shader światła rozproszonego, 274
    - shader tożsamościowy, 114
    - shader wymiany tekstury, 115
    - wiązanie, 256
    - wykorzystanie, 258
  - shading, 45
  - shadow mapping, 213
  - shadow volume, 378
  - shared layout, 455
  - sign(), 269, 270
  - siła, 413
  - sin(), 266
  - sinh(), 266
  - skalar, 153
  - skalowanie, 156, 164
  - skompresowany format teksturowy Ericssona, 615
  - skrót perspektywiczny, 44, 158
  - skybox, 189, 302
  - smooth, 250
  - Smoother, 140
  - smoothstep(), 271, 453
  - Snow Leopard, 573
  - Solution Explorer, 82
  - sortowanie stanów, 144
  - SpecialKeys(), 102
  - specular highlight, 279
  - specular light, 279
  - sphere\_world\_redux, 550
  - SphereWorld, 181, 184, 240, 569, 570
    - aplikacja dla iPhone'a, 633
  - SphereWorld2, 184, 189
  - SphereWorld3, 191
  - SphereWorldFS, 581
  - splot, 439
  - sposoby mapowania buforów, 361

sposoby przechowywania danych pikselowych  
 w pamięci, 199  
 sprajty, 309  
     sprajt punktowy, 309  
     teksturowanie punktów, 309  
 sprawdzanie bufora obrazu, 345  
 sprawdzanie wartości logicznych, 79  
 sprawdzanie wersji biblioteki OpenGL, 77  
 sqrt(), 266  
 sRGB, 384, 385  
 stan OpenGL, 350  
 stan OpenGL ES 2.0, 614  
 stan potoku, 78  
 stan tekstur, 208, 211  
 standard layout, 459  
 standardowe programy do cieniowania, 112  
 step(), 271, 453  
 sterowniki graficzne, 533  
 sterowniki ICD, 533  
 stos macierzy, 179  
     ładowanie macierzy, 179  
     ładowanie macierzy jednostkowej, 179  
     mnożenie macierzy, 179  
     pobieranie macierzy, 180  
     przekształcenia afiniczne, 180  
     wstawianie macierzy, 180  
 stosowanie mipmap, 226  
 stosowanie tekstur prostokątnych, 297  
 stożek, 169  
 stpq, 244, 245  
 strumieniowe modyfikowanie tekstur, 330  
 subpixel, 379  
 surowe dane obrazów, 196  
 SwapBuffers(), 556, 557  
 swizzling, 245  
 symulacja fizyczna, 410  
 symulacja punktów połączonych sprężynami, 416  
 symulacja systemu cząsteczkowego, 510  
 symulacja światła, 272  
     cieniowanie Phong'a, 281  
     model oświetlenia ADS, 278  
     normalne do powierzchni, 273  
     oświetlenie wierzchołków, 273  
     shader światła rozproszonego, 274  
     światło rozproszone, 272  
 sync objects, 523  
 synchronizacja pionowa, 557  
 synchronizacja rysowania, 523  
     CGL, 581  
     EGL, 626

GLX, 601  
 WGL, 557  
 system cząsteczkowy, 510  
 system DCE, 534  
 szarpanie obrazu, 557, 581  
 sześcian trójwymiarowy, 42  
 szkielety, 87

## Ś

śledzenie promieni, 49, 51  
 śledzenie ścieżek promieni, 453  
 śliskie kąty, 187  
 środowiska układów wbudowanych, 615, 628  
     Apple, 629  
     rozszerzenia producentów, 628  
     system operacyjne, 628  
 środowisko programistyczne w systemie Mac OS X,  
 84  
     biblioteki, 87  
     dodawanie ścieżki GLTools do projektu, 89  
     Frameworks, 87  
     nagłówki, 87  
     szkielety, 87  
     tworzenie projektu, 85  
     ustawienia kompilacji, 85  
     Xcode, 84  
 środowisko programistyczne w systemie Windows,  
 79  
     dodawanie plików, 82  
     dodawanie ścieżek, 79  
     Solution Explorer, 82  
     Tools, 79  
     tworzenie projektu, 81  
 środowisko z zanurzeniem, 189  
 światło kierunkowe, 272  
 światło odbicia zwierciadlanego, 279  
 światło otaczające, 278  
 światło rozproszone, 272, 279  
     shader, 274

## T

tablica wierzchołków, 485  
 tablice egzemplarzowe, 491, 496  
 tablice tekstur, 317  
     dostęp do tablic tekstur, 320  
     indeksowanie, 319  
     ładowanie tablicy tekstur dwuwymiarowych, 317  
     TextureArray, 319, 320

- tablice uniform, 261
- tan(), 266
- tanh(), 266
- Targa, 198, 205
- TBO, 337, 410, 415, 512
- tearing, 557, 581
- technika HDR, 367
- technika odwzorowywania tonów, 369
- technika wielopróbkowania, 379
- teksele, 196, 285
- teksturowanie, 46
- teksturowanie kreskówkowe, 292
- teksturowanie punktów, 309
  - efekt przestrzeni międzygwiazdnej, 312
  - nadawanie punktom kształtów, 314
  - obracanie punktów, 315
  - parametry punktów, 314
  - PointSprite, 311
  - rozmiar punktów, 310
  - shader fragmentów obrotowych sprajtów punktowych, 316
  - shader wierzchołków obrotowych sprajtów punktowych, 316
- teksturowanie sprajtu punktowego, 310
- tekstury, 110, 196
  - aktualizacja tekstur, 210
  - bufor kolorów, 209
  - dowiązanie do stanów, 211
  - filtrowanie, 215
  - filtrowanie anizotropowe, 234
  - formaty tekstur, 209
  - generowanie poziomów mipmap, 226
  - GLSL, 285
  - kompresja, 236, 386
  - ładowanie tekstur, 208
  - mipmapy, 223
  - multiteksturowanie, 305
  - nakładanie, 212
  - obiekty tekstur, 211
  - oświetlenie tekstei, 287
  - parametry, 214
  - piksmapy, 199
  - próbkiowanie, 285
  - renderowanie do tekstur, 353
  - stan tekstur, 208, 211
  - szerokość obramowania, 209
  - tablice tekstur, 317
  - teksele, 196
  - teksturowanie w stylu kreskówkowym, 292
  - uchwyt obiektu tekstury, 212
  - upakowane formaty pikseli, 200
  - usuwanie obiektu tekstury, 212
  - wczytywanie pikseli, 205
  - wczytywanie tekstury, 219
  - wierzchołki, 213
  - współrzędne tekstury, 212
  - wymiary, 209
  - zapisywanie pikseli, 203
- tekstury buforowe, 336
- tekstury MSAA, 380, 381
- tekstury prostokątne, 296
  - stosowanie, 297
  - TextureRect, 298
  - wczytywanie, 297
  - współrzędne tekstury, 298
- tekstury sRGB, 385
- tekstury sześciennie, 300
  - Cubemap, 300
  - rysowanie pudła nieba, 302
  - shader fragmentów, 303
  - shader wierzchołków, 303
  - shader wierzchołków odbicia, 304
  - tworzenie efektu odbicia, 304
  - tworzenie pudła nieba, 302
  - wczytywanie, 301
- tekstury trójwymiarowe, 615
- tekstury zastępcze, 320, 321
- tekstury zmiennoprzecinkowe, 614
- teselacja, 428, 430
  - teselacja przy użyciu pasów trójkątów, 430
- test alfa, 289
- test nożyc, 390
- test okrawania, 390
- test przesłonięć, 358
- test szablonu, 397, 398
  - funkcje, 398
  - operacje, 398
- testowanie głębi, 78, 93, 128, 129, 400
  - ograniczanie wartości głębi, 400
- texel, 196
- texelFetch(), 380
- texture filtering, 215
- texture mapping, 46, 196
- texture proxy, 321
- texture replace shader, 115
- texture state, 208, 211
- texture wrapping mode, 217
- TEXTURE\_BRICK, 233
- TEXTURE\_CEILING, 233
- TEXTURE\_FLOOR, 233



- TextureArray, 319, 320
  - TexturedTriangle, 285, 286
    - shader fragmentów, 286
    - shader wierzchołków, 286
  - TextureRect, 298
    - shader fragmentów, 299
  - TGA, 198
  - tone mapping, 369
  - toon shading, 292
  - ToonShader, 293
    - shader wierzchołków, 293
  - torus, 131, 169, 183
  - transform feedback, 500
  - transform feedback buffer, 500
  - transformation matrix, 44
  - Translate(), 180, 183
  - translation matrix, 162
  - transponowanie, 160
  - transpose(), 268, 453
  - transpozycja macierzy, 453
  - trawa, 493
  - triangle\_strip, 418, 426
  - triangles, 418, 420
  - triangles\_adjacency, 420
  - trójkąty, 90, 100, 120
  - trójkąty sklejjane, 123
  - trójwymiarowy, 41
  - trójwymiarowy układ współrzędnych
    - kartezjańskich, 56
  - trunc(), 270
  - tryb kolorów RGBA, 93
  - tryb natychmiastowy, 221
  - tryb przekształcenia zwrotnego, 505
  - tryb przepłotu, 505
  - tryb szkieletowy, 263
  - tryb wyświetlania, 93
  - tryb zawijania tekstur, 217
  - tryby równań mieszania kolorów, 139
  - tryby wielokątów, 130
  - Tunnel, 226
  - tworzenie
    - aplikacje Cocoa, 561
    - aplikacje konsolowe, 81
    - blok zmiennych jednorodnych, 455
    - bufor wierzchołków, 478
    - bufory, 327
    - efekt odbicia, 304
    - frusta, 112
    - kontekst renderingu OpenGL, 547
    - kontekst urządzenia, 554
    - kontener FBO, 339
    - macierz model-widok, 160
    - obiekt bufora renderowania, 340
    - obiekt FBO, 348, 353
    - obiekt VAO, 484
    - okna, 550
    - okna EGL, 621
    - okna GLX, 596
    - okna X, 603
    - projekt, 81, 85
    - projekt aplikacji dla iPhone'a, 629
    - pudło nieba, 302
    - seria danych trójkątów, 221
    - shader geometrii, 419
    - tekstury buforowe, 337
    - tekstury MSA, 381
    - teselowane wierzchołki, 429
    - typ in, 110
    - typ out, 110
    - typ uniform, 109, 113
    - typy danych, 74
      - GLSL, 243
      - język C, 75
      - M3DMatrix33f, 153
      - M3DMatrix44f, 153
      - M3DVector3f, 150
      - M3DVector4f, 150
      - OpenGL, 75
      - typy macierzowe, 245
      - typy obiektów podstawowych, 433
- ## U
- UBO, 454
  - uchwyt obiektu tekstury, 212
  - uintBitsToFloat(), 272
  - UIView, 639
  - układ ciężarków i sprężyn, 413
  - układ kartezjański, 52
  - układ odniesienia aktora, 186
  - układ standardowy, 459
  - układ wspólny, 455
  - układ współrzędnych, 52, 95
    - konfiguracja, 110
  - układy wbudowane, 615, 619
  - uniform, 109, 113, 242, 246, 247, 259, 260, 384, 457
  - uniform buffer object, 454
  - unsigned byte, 75
  - upakowane formaty pikseli, 200
  - uruchamianie biblioteki GLUT, 93
  - urządzenia przenośne, 608

UseStockShader(), 101, 113, 114, 126, 165, 178, 183, 222

ustawianie

- format pikseli, 546
- rozmiar punktu, 117
- stan OpenGL, 350
- warstwy według głębi, 394

ustawienia kompilacji, 85

usuwanie

- bufor, 328
- geometria w shaderach geometrii, 423
- obiekt synchronizacji, 527
- obiekt tekstury, 212
- obiekt FBO, 339
- obiekt VAO, 484
- okno GLX, 596
- płaszczyzny, 127
- płaszczyzny przednie, 128
- płaszczyzny tylne, 127, 424
- powierzchnie ukryte, 45
- program cieniujący, 257

uvec2, 244

uvec3, 244

uvec4, 244

użycie shadera GLSL, 258

## V

VAO, 415, 478, 484

- liczba stanów, 485
- tworzenie obiektów, 484
- usuwanie obiektów, 485

varying, 365

VBO, 410, 415, 478

- alokacja, 479
- inicjalizacja, 479
- przechowywanie kilku atrybutów wierzchołków, 480
- przechowywanie przeplatanych atrybutów, 480
- rozmieszczenie danych w buforach, 481

vec2, 244

vec3, 244, 460

vec4, 244, 245, 365, 460

vertex, 44, 56

vertex array object, 478

vertex buffer object, 478

vertex lighting, 274

vertex shader, 108

Vertex3f(), 221

view frustum, 519

viewing, 154

viewing volume, 57

viewport, 54

viewport transformation, 159

Visual C++, 79

Visual C++ 2008 Express Edition, 79

void, 243

V-sync, 557

## W

wachlarze trójkątów, 123

walec, 169

wartości logiczne, 243

wczytywanie

- piksele, 205
- pliki Targa, 205
- tekstury, 219
- tekstury prostokątne, 297
- tekstury sześciennie, 301

wektorowe typy danych, 244, 245

wektory, 149

- GLSL, 244
- iloczyn skalarny, 151
- iloczyn wektorowy, 152
- M3DVector3f, 150
- M3DVector4f, 150
- normalizacja, 150
- typy danych, 244
- wektor jednostkowy, 150
- wektor normalny, 273

wersje języka GLSL, 249

wersor, 150

WGF, 538

WGL, 532, 536, 537

- atrybuty formatów pikseli, 541, 542
- formaty pikseli, 539
- kontekst bieżący, 549
- kontekst debugowania, 548
- kontekst renderingu OpenGL, 547
- kontekst urządzenia, 538
- PIXELFORMATDESCRIPTOR, 544
- podwójne buforowanie, 556
- rendering pełnoekranowy, 555
- rodzaje zamian buforów, 543
- synchronizacja rysowania, 557
- tworzenie kontekstu renderingu OpenGL, 547
- ustawianie formatu pikseli, 546
- wyбір formatu pikseli, 546
- wyliczenia formatów pikseli, 545
- zamiana buforów, 556
- zapobieganie poszarpaniu obrazu, 557

- WGL
  - znaczniki rodzaju wsparcia sprzętowego, 543
  - znajdowanie formatów pikseli, 539, 544
- WGL\_, 66
- WGL\_ACCELERATION\_ARB, 541, 542, 543
- WGL\_ALPHA\_BITS\_ARB, 540, 543
- WGL\_ALPHA\_SHIFT\_ARB, 543
- WGL\_ARB\_extensions\_string, 536
- WGL\_ARB\_pixel\_format, 539
- WGL\_BLUE\_BITS\_ARB, 543
- WGL\_BLUE\_SHIFT\_ARB, 543
- WGL\_COLOR\_BITS\_ARB, 540, 543
- WGL\_CONTEXT\_COMPATIBILITY\_PROFILE\_
  - ↳BIT\_ARB, 548
- WGL\_CONTEXT\_CORE\_PROFILE\_BIT\_ARB, 548
- WGL\_CONTEXT\_DEBUG\_BIT, 548
- WGL\_CONTEXT\_FLAGS\_ARB, 548
- WGL\_CONTEXT\_MAJOR\_VERSION\_ARB, 547
- WGL\_CONTEXT\_MINOR\_VERSION\_ARB, 547
- WGL\_CONTEXT\_PROFILE\_MASK\_ARB, 548
- WGL\_DEPTH\_BITS\_ARB, 542
- WGL\_DOUBLE\_BUFFER\_ARB, 543, 556
- WGL\_DRAW\_TO\_BITMAP\_ARB, 542
- WGL\_DRAW\_TO\_WINDOW\_ARB, 540, 541, 542
- WGL\_ERROR\_INVALID\_PROFILE\_ARB, 548
- WGL\_ERROR\_INVALID\_VERSION\_ARB, 548
- WGL\_EXT\_swap\_control, 557
- WGL\_FULL\_ACCELERATION\_ARB, 543
- WGL\_GENERIC\_ACCELERATION\_ARB, 543
- WGL\_GREEN\_BITS\_ARB, 543
- WGL\_GREEN\_SHIFT\_ARB, 543
- WGL\_NEED\_PALETTE\_ARB, 542
- WGL\_NEED\_SYSTEM\_PALETTE\_ARB, 542
- WGL\_NO\_ACCELERATION\_ARB, 543
- WGL\_NUMBER\_OVERLAYS\_ARB, 542
- WGL\_NUMBER\_PIXEL\_FORMATS\_ARB, 542, 545
- WGL\_NUMBER\_UNDERLAYS\_ARB, 542
- WGL\_PIXEL\_TYPE\_ARB, 543
- WGL\_RED\_BITS\_ARB, 543
- WGL\_RED\_SHIFT\_ARB, 543
- WGL\_SAMPLES\_ARB, 542
- WGL\_SHARE\_ACCUM\_ARB, 543
- WGL\_SHARE\_DEPTH\_ARB, 542
- WGL\_SHARE\_STENCIL\_ARB, 542
- WGL\_STENCIL\_BITS\_ARB, 542
- WGL\_STEREO\_ARB, 543
- WGL\_SUPPORT\_GDI\_ARB, 543
- WGL\_SUPPORT\_OPENGL\_ARB, 541, 543
- WGL\_SWAP\_COPY\_ARB, 543
- WGL\_SWAP\_EXCHANGE\_ARB, 543
- WGL\_SWAP\_LAYER\_BUFFERS\_ARB, 542
- WGL\_SWAP\_METHOD, 540
- WGL\_SWAP\_METHOD\_ARB, 542, 543
- WGL\_SWAP\_UNDEFINED\_ARB, 543
- WGL\_TRANSPARENT\_ALPHA\_VALUE\_ARB, 542
- WGL\_TRANSPARENT\_ARB, 542
- WGL\_TRANSPARENT\_BLUE\_VALUE\_ARB, 542
- WGL\_TRANSPARENT\_GREEN\_VALUE\_ARB, 542
- WGL\_TRANSPARENT\_RED\_VALUE\_ARB, 542
- wglChoosePixelFormat(), 540
- wglChoosePixelFormatARB(), 540, 545, 546, 556
- wglCreateContext(), 549
- wglCreateContextAttribsARB(), 547, 548, 549
- wglDeleteContext(), 549, 554
- wglctx.h, 536
- wglGetExtensionsStringARB(), 536
- wglGetPixelFormatAttribARB(), 544, 546
- wglGetPixelFormatAttribfvARB(), 541, 545
- wglGetPixelFormatAttribivARB(), 541, 545, 546
- wglGetPixelFormatAttributeivARB(), 544
- wglGetProcAddress(), 535, 536, 544, 627
- wglMakeCurrent(), 549, 554
- wglSwapIntervalEXT(), 557
- while, 452
- wiązanie, 252, 256
- wiązanie lokalizacji atrybutów, 257
- wiązanie RBO, 340
- widoki, 54, 96
  - przekształcenia geometryczne, 159
- widzenie w trzech wymiarach, 43
- wielokąty, 130
- wielokrotne rysowanie tej samej geometrii, 490
- wielopróbkowanie, 143, 379, 391
  - kolejność próbek, 396
  - konfiguracja stanu maski, 393
  - liczba próbek, 380
  - maska próbki, 392
  - odwzorowywanie tonów, 382
  - operacja rozwiązywania, 379
  - podpiksele, 379
  - powierzchnia pokrycia próbki, 391
  - shader rozwiązywania, 381
  - tekstury MSAA, 380, 381
  - tworzenie kontenera na wielopróbkowany obiekt RBO, 380
- wielowątkowość, 583
- wierzchołki, 44, 56, 100, 259
  - wierzchołek prowokujący, 259
- wierzchołki tekstury, 213
- winding, 122

- Windows, 79, 532
    - formaty pikseli, 539
    - GDI, 538
    - implementacje OpenGL, 532
    - inicjalizowanie kontekstu renderingu, 554
    - kontekst renderingu, 538
    - kontekst urządzenia, 538, 554
    - OpenGL firmy Microsoft, 533
    - podwójne buforowanie, 556
    - rendering pełnoekranowy, 555
    - renderowanie, 537
    - rozszerzenia OpenGL, 534
    - rozszerzenia WGL, 536
    - tworzenie okna, 550
    - wyłączanie kontekstu renderingu, 554
  - Windows BMP, 198
  - Windows Graphics Foundation, 538
  - Windows Presentation Foundation, 538
  - Windows-GL, 532
  - WinMain(), 93
  - wireframe rendering, 45
  - wizualizacja normalnych, 432
  - włączanie
    - mieszanie kolorów, 136
    - test okrawania, 390
    - testowanie głębi, 129
  - wojny API, 23
  - WPF, 538
  - wrażenie trójwymiarowości, 43
  - wskazówki, 77
  - współczynnik tłumienia, 412
  - współczynniki mieszania, 137
  - współrzędne, 95
    - współrzędne globalne położenia światła, 192
    - współrzędne oka, 154
    - współrzędne okna, 54
    - współrzędne rysowania, 54
    - współrzędne tekstury, 212, 220
  - wybór formatu pikseli, 546
  - wycinanie nożycami, 134
  - wydajność wypełniania, 582
  - wygładzanie, 140
    - algorytmy antyaliasingu, 143
    - wielopróbkowanie, 143
  - wyjście z shadera, 342
  - wykorzystanie wyniku zapytania, 469
  - wykrywanie krawędzi, 450
  - wyliczenia formatów pikseli, 545
  - wyłączanie kontekstu renderingu, 554
  - wyłączanie rasteryzacji, 506
  - wymiary obiektów, 42
  - wymiary okna, 94
  - wyniki zapytania, 468, 469
  - wyniki zapytania obiektów podstawowych, 508
  - wypełnianie, 582
  - wysyłanie danych z shadera pikseli, 362
  - wysyłanie zapytania, 468
  - wzmacnianie, 417
- X**
- X Window, 586, 591
    - ekrany, 592
  - Xcode, 84, 560, 561, 566, 629, 630
  - XCreateWindow(), 595, 596
  - XDestroyWindow(), 597
  - XFree86, 587
  - XIB, 561
  - XOpenDisplay(), 592, 603
  - xyzw, 244
- Y**
- yaw, 187
- Z**
- zaawansowane shadery fragmentów, 436
  - zakres wypełniania, 129
  - zamiana buforów, 101
  - zapisywanie pikseli, 203
  - zapobieganie poszarpaniu obrazu, 557
  - zapytania czasowe, 475
  - zapytania GLX, 602
  - zapytania obiektów podstawowych, 507
    - liczenie wierzchołków, 507
    - wyniki, 508
  - zapytanie, 466
    - błędy, 467
    - obiekt zapytaniowy, 467
    - pobieranie wyników, 468
    - przygotowywanie, 467
    - rendering warunkowy, 471
    - wykorzystanie wyniku, 469
    - wysyłanie, 468
    - zapytanie o zasłanianie, 466
    - zwracanie zasobów, 467
  - zarządzanie geometrią, 465
  - zastosowanie grafiki trójwymiarowej, 47
  - zator potoku, 108
  - zatrzymanie działania shadera fragmentów, 289

zawijanie tekstury, 217  
zbiór Julii, 442, 444  
zbiór Mandelbrota, 442  
zestaw trójkątów, 167  
z-fighting, 132  
zmiana równania mieszania, 139  
zmiana sposobu przechowywania danych  
  pikselowych w pamięci, 199  
zmiennie, 243  
  deklaracja, 243  
  kwalifikatory, 246  
  zmiennie stanu, 78  
  zmiennie uniform, 113, 259

  zmiennie uniform skalarne, 260  
  zmiennie uniform wektorowe, 260  
  zmiennie wyjściowe, 246  
zmiennoprzecinkowe bufory głębi, 378  
znaczniki błędów, 76  
znajdowanie danych uniform, 260  
znajdowanie formatów pikseli, 539, 544  
znaki ASCII, 41

## Ż

źdźbła trawy, 493

# OpenGL

Po prawie dwudziestu latach na rynku biblioteka OpenGL jest dziś wiodącym API w dziedzinie programowania grafiki trójwymiarowej, gier 3D, wizualizacji, symulacji, modelowania naukowego, a nawet edytowania obrazów i filmów dwuwymiarowych. Swoją sukces zawdzięcza nie tylko łatwości użycia, ale przede wszystkim kompatybilności z niemal wszystkimi platformami dostępnymi na rynku. Świetnie sprawdza się zarówno w komputerach PC z systemem Windows, jak i komputerach Mac, a także na stacjach uniwersalnych, w centrach rozrywki opartych na lokalizacji, na najbardziej znanych konsolach do gier, w kieszonkowych grach elektronicznych, a nawet w oprzyrządowaniu lotniczym czy samochodowym. Nie bez znaczenia dla popularyzowania tej biblioteki był także fakt, że można ją rozszerzać, dzięki czemu ma ona wszystkie zalety otwartego standardu, a dodatkowo można wzbogacać jej implementację o własne dodatki.

„OpenGL. Księga eksperta. Wydanie 5” to nowe, zaktualizowane (specyfikacja OpenGL 3.3) wydanie znanego podręcznika dla wszystkich programistów, bez względu na poziom ich zaawansowania. Książka ta stanowi wyczerpujący kurs tworzenia niesamowitych wizualizacji 3D, gier oraz wszelkiego rodzaju grafik. Dzięki niej nauczysz się pisać programy wykorzystujące bibliotekę OpenGL, skonfigurować środowisko pracy do przetwarzania grafiki trójwymiarowej oraz tworzyć podstawowe obiekty, oświetlać je i cieniować. Następnie zgłębisz tajniki języka OpenGL Shading Language i zaczniesz sprawnie pisać własne programy, wprowadzać do nich rozmaite efekty wizualne oraz zwiększać ich wydajność. Poznasz wszystkie najnowsze techniki programowania przy użyciu biblioteki OpenGL, takie jak przekształcenia, nakładanie tekstur, cieniowanie, zaawansowane buforów czy zarządzanie geometrią. Przejdziesz także szczegółowy kurs programowania grafiki w urządzeniach iPhone, iPod touch oraz iPad!

W tym wyczerpującym podręczniku znajdziesz:

- praktyczne wprowadzenie do technik programowania grafiki trójwymiarowej czasu rzeczywistego
- rdzenne techniki OpenGL 3.3 w zakresie renderowania, przekształcania i teksturowania geometrii
- wiedzę na temat pisania programów cieniujących, popartą praktycznymi przykładami
- opis technik programowania na różnych platformach — Windows (także Windows 7), Mac OS X, GNU/Linux, Unix — oraz układach wbudowanych
- informacje na temat programowania przy użyciu biblioteki OpenGL aplikacji przeznaczonych dla urządzeń iPhone, iPod touch oraz iPad (kurs prowadzony krok po kroku i ilustrowany przykładowymi programami)
- zaawansowane techniki buforowania, renderowanie w pełnej rozdzielczości przy użyciu buforów i tekstur zmiennoprzecinkowych
- możliwości przetwarzania fragmentów, czyli zarządzania końcową częścią potoku przetwarzania grafiki
- zaawansowane techniki cieniowania i zarządzania geometrią

**Kompletny przewodnik po najpopularniejszej na świecie bibliotece do programowania grafiki trójwymiarowej OpenGL 3.3!**

Nr katalogowy: 5790



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



Zamówienia telefoniczne:  
**0 801 339900**



**0 601 339900**



**Helion**

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

Helion SA  
ul. Kołczyński 1c, 44-100 Gliwice  
tel.: 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
<http://helion.pl>

**helion.pl**  
księgarnia  
internetowa

Cena: 119,00 zł

ISBN 978-83-246-2976-3



Informatyka w najlepszym wydaniu