helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Język Cg. Programowanie grafiki w czasie rzeczywistym

Autorzy: Randima Fernando, Mark J. Kilgard
Tłumaczenie: Rafał Jońca
ISBN: 83-7361-241-6
Tytuł oryginału: The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics
Format: B5, stron: 308

Cg to kompletne środowisko programistyczne do szybkiego tworzenia efektów specjalnych i grafiki o kinowej jakości w czasie rzeczywistym dla wielu platform. Ponieważ język jest niezależny od sprzętu, programiści mogą pisać kod dla interfejsów OpenGL, DirectX oraz systemów Windows, Linux, Mac OS X, a także platform konsolowych, (Xbox) bez potrzeby korzystania z języka asemblerowego. Język Cg powstał w firmie NVIDIA Corporation przy bliskiej współpracy z firmą Microsoft® Corporation i jest kompatybilny z OpenGL API oraz językiem HLSL dla biblioteki DirectX 9.

Książka jest podręcznikiem przeznaczonym dla średnio zaawansowanych programistów. Opisuje ona zarówno sam język programowania Cg, jak i metody wielu składników nowoczesnych aplikacji bazujących na grafice trójwymiarowej.

Prezentowane w książce zagadnienia to m.in.:

- Historia języka Cg
- Środowisko programistyczne Cg
- Składnia Cg i słowa kluczowe
- Przekształcenia w przestrzeni trójwymiarowej
- Oświetlenie bazujące na wierzchołkach i pikselach
- Interpolacja ujęć kluczowych i system kości
- Mapowanie środowiska
- Mapowanie nierówności
- Mgła, światła reflektorowe, cienie
- Zwiększanie wydajności

„Książka ważna i na czasie: tworzenie tekstur proceduralnych na poziomie pikseli — animowanych chmur, ognia, wody i wielu innych sztuczek — nareszcie z ekranów kin przechodzi pod strzechy. Cała moc jest dostępna dzięki językowi przypominającemu język C, co otwiera nowy rozdział w grafice komputerowej”.
— Ken Perlin, Uniwersytet w Nowym Jorku

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Przedmowa ... 13

Wstęp ... 15

Rozdział 1. Wprowadzenie ... 21

1.1. Czym jest Cg? ... 21
1.1.1. Języki dla programowalnego sprzętu graficznego ... 22
1.1.2. Model przepływu danych w Cg ... 22
1.1.3. Specjalizacja a generalizacja procesorów graficznych ... 23
1.1.4. Wydajność języka Cg ... 24
1.1.5. Współdziałanie z konwencjonalnymi językami ... 24
1.1.6. Inne aspekty języka Cg ... 26
1.1.7. Ograniczone środowisko wykonywania programów Cg ... 27
1.2. Wierzchołki, fragmenty i potok grafiki ... 28
1.2.1. Ewolucja sprzętu graficznego ... 28
1.2.2. Cztery generacje sprzętu graficznego ... 29
1.2.3. Sprzętowy potok graficzny ... 33
1.2.4. Programowalny potok graficzny ... 37
1.2.5. Język Cg zapewnia możliwość programowania jednostek wierzchołków i fragmentów ... 40
1.3. Historia powstania Cg ... 40
1.3.1. Współpraca firm NVIDIA i Microsoft w celu określenia języków Cg i HLSL ... 42
1.3.2. Nieinteraktywne języki cieniowania ... 42
1.3.3. Interfejsy programistyczne w grafice trójwymiarowej ... 45
1.4. Środowisko Cg ... 45
1.4.1. Standardowe interfejsy programistyczne 3D: OpenGL i Direct3D ... 45
1.4.2. Kompilator i biblioteka wykonywania Cg ... 47
1.4.3. Narzędzia CgFX i format pliku ... 49
1.5. Ćwiczenia ... 53

Rozdział 2. Najprostsze programy 55
2.1. Prosty program wierzchołków 55
2.1.1. Struktura wyjścia 56
2.1.2. Identyfikatory 57
2.1.3. Elementy struktur 58
2.1.4. Wektory 58
2.1.5. Macierze 58
2.1.6. Semantyka 59
2.1.7. Funkcje 60
2.1.8. Różnice w semantyce wejścia i wyjścia 61
2.1.9. Ciało funkcji 62
2.2. Kompilacja przykładu 64
2.2.1. Profile programu wierzchołków 64
2.2.2. Klasy błędów kompilacji programów Cg 66
2.2.3. Błędy wynikające ze złego profilu 66
2.2.4. Norma — kilka funkcji wejścia 68
2.2.5. Pobieranie i konfiguracja programów wierzchołków i fragmentów 68
2.3. Prosty program fragmentów 70
2.3.1. Profile dla programów fragmentów 71
2.4. Rendering przykładowych programów wierzchołków i fragmentów 72
2.4.1. Rendering trójkąta w OpenGL 73
2.4.2. Rendering trójkąta w Direct3D 74
2.4.3. Uzyskanie tych samych wyników 74
2.5. Ćwiczenia 76

Rozdział 3. Parametry, tekstury i wyrażenia 77
3.1. Parametry 77
3.1.1. Parametry jednolite 77
3.1.2. Kwalifikator typu const 80
3.1.3. Różnorodność parametrów 80
3.2. Próbkowanie tekstur 82
3.2.1. Obiekty próbek 82
3.2.2. Próbki tekstur 83
3.2.3. Wysyłanie współrzędnych tekstury w trakcie próbkowania tekstury 84
3.3. Wyrażenia matematyczne 85
3.3.1. Operatory 85
3.3.2. Typy danych uzależnione od profilu 86
3.3.3. Funkcje wbudowane w standardową bibliotekę Cg 90
3.3.4. Skręcanie w dwuwymiarze 93
3.3.5. Efekt podwójnego widzenia 96
3.4. Ćwiczenia 100

Rozdział 4. Przekształcenia 101
4.1. Układy współrzędnych 101
4.1.1. Przestrzeń obiektu 102
4.1.2. Współrzędne homogeniczne 103
4.1.3. Przestrzeń świata 103
4.1.4. Przekształcenie modelu 104
4.1.5. Przestrzeń oka 105
4.1.6. Przekształcenie widoku 105
4.1.7. Przestrzeń przycięcia 106
4.1.8. Przekształcenie rzutowania 106
4.1.9. Znormalizowane współrzędne urządzenia 107
4.1.10. Współrzędne okna 108
4.2. Zastosowanie teorii 108
4.3. Ćwiczenia 109

Rozdział 5. Oświetlenie 111
5.1. Oświetlenie i związane z nim modele 111
5.2. Implementacja podstawowego modelu oświetlenia opartego na wierzchołkach 113
5.2.1. Podstawowy model oświetlenia 113
5.2.2. Program wierzchołków dla prostego oświetlenia opartego na wierzchołkach 119
5.2.3. Program fragmentów dla modelu oświetlenia wykorzystującego wierzchołki 128
5.2.4. Efekt modelu oświetlenia opartego na wierzchołkach 128
5.3. Model oświetlenia oparty na fragmentach 129
5.3.1. Implementacja modelu oświetlenia opartego na fragmentach 130
5.3.2. Program wierzchołków dla modelu oświetlenia opartego na fragmentach 131
5.3.3. Program fragmentów dla modelu oświetlenia opartego na fragmentach 131
5.4. Tworzenie funkcji modelu oświetlenia 133
5.4.1. Deklarowanie funkcji 133
5.4.2. Funkcja oświetlenia 134
5.4.3. Struktury 135
5.4.4. Tablice 136
5.4.5. Sterowanie wykonywaniem programu 137
5.4.6. Obliczenie modelu oświetlenia rozproszenia i rozbłysku 138
5.5. Rozszerzenie modelu podstawowego 138
5.5.1. Zanik światła wraz z odległością 139
5.5.2. Dodanie efektu reflektora 140
5.5.3. Światła kierunkowe 145
5.6. Ćwiczenia 145

Rozdział 6. Animacja 147
6.1. Ruch w czasie 147
6.2. Pulsujący obiekt 148
6.2.1. Program wierzchołków 149
6.2.2. Obliczanie przemieszczenia 150

6.3. Systemy cząsteczek ... 152
6.3.1. Warunki początkowe ... 153
6.3.2. Wektoryzacja obliczeń ... 153
6.3.3. Parametry systemu cząsteczek ... 154
6.3.4. Program wierzchołków ... 154
6.3.5. Ubieramy system cząsteczek ... 156
6.4. Interpolacja ujęć kluczowych ... 157
6.4.1. Teoria ujęć kluczowych ... 157
6.4.2. Rodzaje interpolacji ... 160
6.4.3. Prosta interpolacja ujęć kluczowych ... 160
6.4.4. Interpolacja ujęć kluczowych z oświetleniem ... 162
6.5. System skóry dla wierzchołków ... 163
6.5.1. Teoria systemu skóry dla wierzchołków ... 163
6.5.2. System skóry w programie wierzchołków ... 166
6.6. Ćwiczenia ... 167

Rozdział 7. Mapowanie środowiska ... 169
7.1. Mapowanie środowiska ... 169
7.1.1. Tekstury map sześciennych ... 170
7.1.2. Generowanie map sześciennych ... 171
7.1.3. Koncepcja mapowania środowiska ... 171
7.1.4. Obliczenie wektorów odbicia ... 172
7.1.5. Założenia mapowania środowiska ... 173
7.2. Mapowanie odbić ... 174
7.2.1. Parametry określane przez aplikację ... 175
7.2.2. Program wierzchołków ... 175
7.2.3. Program fragmentów ... 179
7.2.4. Mapy sterujące ... 180
7.2.5. Program wierzchołków a program fragmentów ... 180
7.3. Mapowanie załamań ... 181
7.3.1. Zjawisko załamania światła ... 182
7.3.2. Program wierzchołków ... 184
7.3.3. Program fragmentów ... 186
7.4. Efekt Fresnela i rozszczepienie chromatyczne ... 187
7.4.1. Efekt Fresnela ... 187
7.4.2. Rozszczepienie chromatyczne ... 188
7.4.3. Parametry zależne od aplikacji ... 189
7.4.4. Program wierzchołków ... 190
7.4.5. Program fragmentów ... 191
7.5. Ćwiczenia ... 193

Rozdział 8. Mapowanie nierówności ... 195
8.1. Mapowanie nierówności ceglanej ściany ... 195
8.1.1. Mapa normalnych ceglanej ściany ... 196
8.1.2. Przechowywanie map nierówności jako map normalnych ... 197
8.1.3. Proste mapowanie nierówności dla ceglanego muru ... 200
8.1.4. Mapowanie nierówności dla rozbłysku ... 203
8.1.5. Mapowanie nierówności na innej geometrii ... 206
8.2. Mapowanie nierówności ceglanej podłogi ... 208
8.2.1. Program wierzchołków dla renderingu obrazu ceglanej podłogi ... 210
8.3. Mapowanie nierówności dla torusa ... 213
8.3.1. Matematyka dotycząca torusa ... 213
8.3.2. Program wierzchołków dla torusa z mapowaniem nierówności ... 216
8.4. Mapowanie nierówności dla teksturowanych siatek wielokątnych ... 218
8.4.1. Algorytm dla pojedynczego trójkąta ... 218
8.4.2. Możliwe problemy ... 220
8.4.3. Uogólnienie do siatek z wielokątów ... 222
8.5. Połączenie mapowania nierówności z innymi efektami ... 223
8.5.1. Standardowe tekstury ... 223
8.5.2. Mapy połysku ... 223
8.5.3. Rzucanie cieni na samego siebie ... 224
8.6. Ćwiczenia ... 225

Rozdział 9. Zagadnienia zaawansowane ... 227
9.1. Mgła ... 227
9.1.1. Mgła jednorodna ... 228
9.1.2. Atrybuty mgły ... 229
9.1.3. Matematyka mgły ... 229
9.1.4. Dostosowanie równań do zachowania zgodnego z intuicją ... 232
9.1.5. Tworzenie jednorodnej mgły w programie Cg ... 233
9.2. Rendering nierealistyczny ... 235
9.2.1. Cieniowanie jak w kreskówkach ... 235
9.2.2. Implementacja cieniowania kreskówkowego ... 236
9.2.3. Łączymy wszystko razem ... 239
9.2.4. Problemy związane z tym rozwiązaniem ... 241
9.3. Rzutowanie tekstur ... 241
9.3.1. W jaki sposób działa rzutowanie tekstur? ... 242
9.3.2. Implementacja rzutowania tekstury ... 244
9.3.3. Kod rzutowania tekstury ... 245
9.4. Mapowanie cieni ... 248
9.5. Łączenie ... 250
9.5.1. Mapowanie pikseli z wejścia na wyjście ... 251
9.5.2. Podstawowe operacje dotyczące łączenia ... 252
9.6. Ćwiczenia ... 254

Rozdział 10. Profile i wydajność ... 257
10.1. Opis profili ... 257
10.1.1. Profil shadera wierzchołków dla DirectX 8 ... 257
10.1.2. Podstawowy profil programu wierzchołków dla kart NVIDIA i OpenGL ... 258
10.1.3. Profil programu wierzchołków ARB dla OpenGL ... 259
10.1.4. Profil shadera wierzchołków dla DirectX 9 ... 259
10.1.5. Zaawansowany profil programu wierzchołków dla kart NVIDIA i OpenGL ... 259
10.1.6. Profile shadera pikseli dla DirectX 8 ... 260
10.1.7. Podstawowy profil programu fragmentów NVIDIA dla OpenGL ... 261
10.1.8. Profile shadera pikseli dla DirectX9 ... 261
10.1.9. Profil programu fragmentów ARB dla OpenGL ... 262
10.1.10. Zaawansowany profil programu fragmentów NVIDIA dla OpenGL ... 262
10.2. Wydajność ... 263
10.2.1. Korzystanie ze standardowej biblioteki Cg ... 263
10.2.2. Zalety parametrów jednorodnych ... 264
10.2.3. Program fragmentów a program wierzchołków ... 264
10.2.4. Typy danych i ich wpływ na wydajność ... 265
10.2.5. Wykorzystanie zalet wektoryzacji ... 265
10.2.6. Kodowanie funkcji w teksturach ... 266
10.2.7. Intensywnie wykorzystanie przemieszania i negacji ... 267
10.2.8. Cieniujemy tylko te fragmenty, które musimy ... 267
10.2.9. Krótszy kod asemblerowy nie zawsze jest szybszy ... 268
10.3. Ćwiczenia ... 268

Dodatek A Narzędzia Cg ... 269
A.1. Pobieranie przykładów prezentowanych w niniejszej książce ... 269
A.2. Pobieranie narzędzia Cg Toolkit ... 269

Dodatek B Biblioteka wykonywania Cg ... 271
B.1. Czym jest biblioteka wykonywania Cg? ... 271
B.2. Dlaczego warto używać biblioteki wykonywania Cg? ... 271
B.2.1. Dostosowanie do nowszych procesorów graficznych ... 271
B.2.2. Brak problemów z zależnościami ... 272
B.2.3. Zarządzanie parametrami wejściowymi ... 272
B.3. W jaki sposób działa biblioteka wykonywania Cg? ... 273
B.3.1. Pliki nagłówkowe ... 274
B.3.2. Tworzenie kontekstu ... 274
B.3.3. Kompilacja programu ... 274
B.3.4. Wczytanie programu ... 275
B.3.5. Modyfikacja parametrów programu ... 276
B.3.6. Wykonanie programu ... 276
B.3.7. Zwalnianie zasobów ... 277
B.3.8. Obsługa błędów ... 277
B.4. Dodatkowe informacje ... 278

Dodatek C Format pliku CgFX 279
C.1. Czym jest CgFX? 279
C.2. Opis formatu 280
C.2.1. Techniki 280
C.2.2. Przebiegi 281
C.2.3. Stany renderingu 281
C.2.4. Zmienne i semantyka 282
C.2.5. Przypisy 282
C.2.6. Przykładowy plik CgFX 283
C.3. Moduły Cg obsługujące format CgFX 284
C.4. Dodatkowe informacje o CgFX 285

Dodatek D Słowa kluczowe języka Cg 287
D.1. Lista słów kluczowych języka Cg 287

Dodatek E Funkcje standardowej biblioteki Cg 289
E.1. Funkcje matematyczne 290
E.2. Funkcje geometryczne 293
E.3. Funkcje mapowania tekstur 294
E.4. Funkcje pochodnych 295
E.5. Funkcja testowania 296

Skorowidz 297

Rozdział 5. Oświetlenie

W tym rozdziale opiszemy, w jaki sposób symulować oświetlenie obiektów na scenie za pomocą źródeł światła. Zaczniemy od utworzenia uproszczonej wersji powszechnie stosowanego modelu oświetlenia. Następnie stopniowo będziemy dodawali coraz to nowe funkcje do modelu podstawowego, aby był bardziej użyteczny. Niniejszy rozdział składa się z pięciu podrozdziałów.

- 5.1. *Oświetlenie i związane z nim modele* — wyjaśnia znaczenie oświetlenia oraz opisuje koncepcję modeli oświetlenia.
- 5.2. *Implementacja podstawowego modelu oświetlenia opartego na wierzchołkach* — przedstawia uproszczoną wersję modelu oświetlenia używanego w OpenGL i Direct3D. Opisuje także krok po kroku wykonanie tego modelu w programie wierzchołków.
- 5.3. *Oświetlenie oparte na fragmentach* — omawia różnicę między oświetleniem opartym na wierzchołkach i fragmentach oraz przedstawia sposób implementacji oświetlenia dla fragmentów.
- 5.4. *Tworzenie funkcji oświetlenia* — wyjaśnia sposób tworzenia własnej funkcji modelowania oświetlenia.
- 5.5. *Rozszerzenie modelu podstawowego* — wprowadza kilka udogodnień do podstawowego modelu oświetlenia: teksturowanie, zanik i efekty świateł reflektorowych. Przy okazji wprowadzimy kilka kluczowych koncepcji języka Cg, na przykład tworzenie funkcji, tablic i struktur.

5.1. Oświetlenie i związane z nim modele

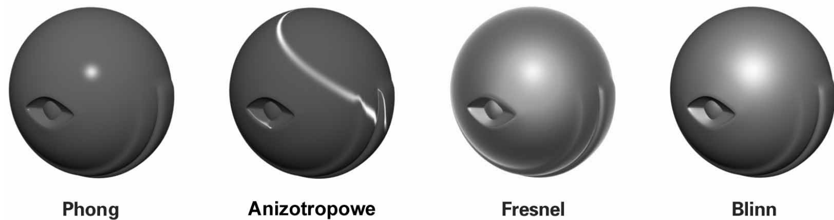
Do tej pory omawiane przykłady były proste i dotyczyły podstawowych koncepcji potrzebnych do napisania programu. W kilku następnych rozdziałach przedstawimy kilka interesujących efektów. W tym rozdziale zajmiemy się modelowaniem oświetlenia.

Dodanie do sceny oświetlenia pozwala na uzyskanie zróżnicowanego cieniowania a tym samym bardziej interesujących obrazów. Właśnie z tego powodu reżyserzy zwracają dużą uwagę na oświetlenie — wpływa ono na sposób odbierania opowiadanej historii. Ciemne obszary sceny wzmagają uczucie tajemniczości i stopniują napięcie (niestety w grafice komputerowej cieni nie dostaje się „za darmo", gdy tylko doda się oświetlenie. W rozdziale 9. dokładnie opisujemy tworzenie cieni).

Oświetlenie i właściwości użytego materiału definiują wygląd obiektu. Model oświetlenia definiuje sposób, w jaki światło wchodzi w interakcję z obiektem. Wykorzystywana jest przy tym charakterystyka światła i materiału obiektu. W ciągu ostatnich lat powstało wiele różnych modeli oświetlenia, od prostych aproksymacji po bardzo dokładne symulacje.

Na rysunku 5.1 przedstawiono obiekty zrederowane za pomocą różnych modeli oświetlenia. Warto zauważyć, w jaki sposób modele symulują materiały z rzeczywistego świata.

Rysunek 5.1. *Różne modele oświetlenia*



W przeszłości potok graficzny z na stałe ustalonymi funkcjami był ograniczony do jednego modelu cieniowania. Model ten jest nazywany *modelem oświetlenia o stałej funkcji*. Model ten bazuje na modelu Phong, ale posiada kilka modyfikacji i dodatków. Model oświetlenia o stałej funkcji ma kilka zalet: wygląda zadowalająco, nie jest kosztowny obliczeniowo oraz udostępnia kilka parametrów, które można wykorzystać do sterowania wyglądem. Problem polega na tym, że model ten wygląda odpowiednio tylko dla ograniczonej liczby materiałów. Obiekty wydają się być wykonane z plastiku lub gumy, więc obrazy komputerowe nie wyglądają zbyt realistycznie.

Aby obejść ograniczenia modelu oświetlenia stałej funkcji, programiści grafiki zaczęli wykorzystywać inne cechy potoku graficznego. Na przykład sprytnie napisane programy używały odpowiednich tekstur, aby lepiej symulować niektóre materiały.

Dzięki językowi Cg i programowalnym jednostkom graficznym można napisać własny, złożony model cieniowania w języku wysokiego poziomu. Nie musimy już konfigurować ograniczonego zbioru stanów potoku graficznego lub

programować w niewygodnym języku asemblerowym. Najważniejsze jest jednak to, że nie jesteśmy ograniczeni do jednego, stałego modelu oświetlenia. Możemy napisać własny model, który zostanie wykonany w procesorze graficznym.

5.2. Implementacja podstawowego modelu oświetlenia opartego na wierzchołkach

W tym podrozdziale opiszemy, w jaki sposób zaimplementować uproszczoną wersję modelu cieniowania stałej funkcji za pomocą programu wierzchołków. Popularność i prostota tego modelu powodują, że idealnie nadaje się on do rozpoczęcia nauki opisu oświetlenia. Najpierw zajmiemy się opisem samego modelu. Jeśli Czytelnik dobrze zna ten model, może przejść do podrozdziału 5.2.2.

5.2.1. Podstawowy model oświetlenia

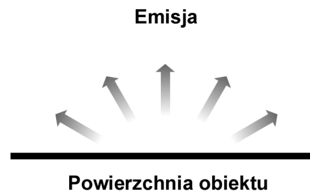
OpenGL i Direct3D stosują prawie identyczny model oświetlenia o stałej funkcji. W naszym przykładzie zastosujemy wersję uproszczoną, którą będziemy nazywać modelem podstawowym. Model podstawowy, podobnie jak modele OpenGL i Direct3D, modyfikują i rozszerzają klasyczny model Phong. W modelu podstawowym kolor powierzchni jest sumą współczynników oświetlenia: emisyjnego, otoczenia, rozproszenia i rozbłysku. Każdy z współczynników zależy od kombinacji właściwości materiału obiektu (na przykład połyskliwości i koloru materiału) i właściwości światła (na przykład położenie i kolor światła). Każdy ze współczynników stanowi wektor `float3` zawierający komponenty koloru czerwonego, zielonego i niebieskiego.

Ogólne równanie opisujące ten model można napisać następująco.

kolor powierzchni = emisja + otoczenie + dyfuzja + rozbłysk

Współczynnik emisji

Współczynnik emisji określa światło emitowane lub oddawane przez powierzchnię i jest niezależny od wszystkich źródeł światła. Współczynnikiem emisji jest wartość RGB wskazująca kolor emitowanego światła. Jeśli oglądamy materiał emitujący światło w ciemnym pokoju, zobaczymy właśnie ten kolor. Współczynnik emisji umożliwia symulację świecenia. Na rysunku 5.2 przedstawiono koncepcję współczynnika emisji a na rysunku 5.3 — rendering obiektu z uwzględnieniem



Rysunek 5.2.
Współczynnik emisji

Rysunek 5.3.
Rendering obiektu z uwzględnieniem współczynnika emisji

tylko współczynnika emisji. Rendering jest nieciekawy, ponieważ cały obiekt pokrywa jeden kolor. W odróżnieniu od rzeczywistego świata, obiekty emitujące światło na scenie nie oświetlają pobliskich obiektów. Taki obiekt nie jest źródłem światła — niczego nie oświetla i nie rzuca cieni. Współczynnik emisji można traktować jako kolor dodawany po obliczeniu wszystkich innych współczynników oświetlenia. Bardziej zaawansowane modele oświetlenia ogólnego symulują sposób, w jaki wyemitowane światło wpływa na resztę sceny, ale tymi modelami nie będziemy zajmowali się w tej książce.

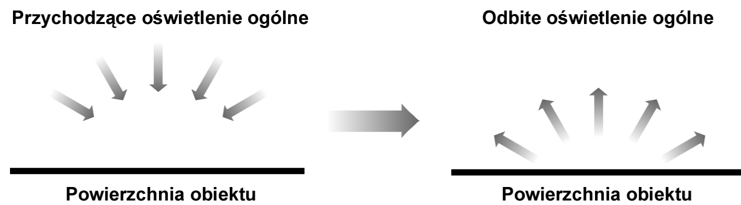
Oto wzór matematyczny wykorzystywany do obliczania współczynnika emisji (*emissive*).

$$emissive = K_e$$

gdzie K_e to kolor emisji dla materiału.

Współczynnik otoczenia

Współczynnik otoczenia dotyczy światła, które jest tak rozproszone w scenie, że wydaje się, iż pochodzi ze wszystkich stron. Oświetlenie otoczenia nie ma jakiegoś określonego kierunku, wydaje się pochodzić ze wszystkich kierunków. Oznacza to, że współczynnik ten nie zależy od położenia światła. Rysunek 5.4 obrazuje koncepcję a na rysunku 5.5 przedstawiono rendering obiektu, który otrzymuje tylko światło otoczenia. Współczynnik otoczenia zależy od współczynnika odbicia materiału obiektu a także koloru światła rzucanego na materiał. Podobnie jak w przypadku współczynnika emisji, współczynnik otoczenia to jeden stały kolor. Różnica polega na tym, że współczynnik otoczenia jest modyfikowany przez globalną wartość oświetlenia ogólnego.



Rysunek 5.4.
Współczynnik otoczenia

Rysunek 5.5.
Rendering obiektu z uwzględnieniem współczynnika otoczenia

Oto wzór matematyczny dla współczynnika otoczenia (*ambient*)

$$ambient = K_a \times globalAmbient$$

gdzie:

- K_a to współczynnik odbicia materiału,
- *globalAmbient* to kolor oświetlenia ogólnego.

Współczynnik rozproszenia

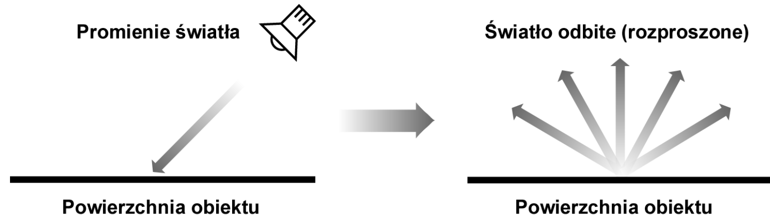
Współczynnik rozproszenia dotyczy promienia światła odbijanego przez powierzchnię w równym stopniu dla wszystkich kierunków. Powierzchnie, dla których stosuje się współczynnik rozproszenia są chropowate w skali mikroskopijnej, więc odbijają światło we wszystkich kierunkach w równym stopniu. Gdy promień światła dochodzi do zakamarków powierzchni, odbija się we wszystkich możliwych kierunkach (patrz rysunek 5.6).

Rysunek 5.6.
Rozproszenie światła

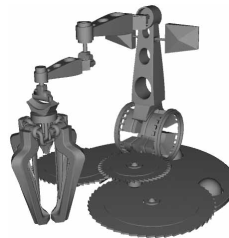
Natężenie światła odbitego od obiektu jest proporcjonalne do kąta padania światła na powierzchnię. Powierzchnie niewygładzone nazywane są często powierzchniami rozpraszającymi światło. Współczynnik rozproszenia dla każdego

punktu powierzchni jest taki sam, niezależnie od tego, gdzie znajduje się punkt widzenia. Rysunek 5.7 ilustruje znaczenie współczynnika rozproszenia a rysunek 5.8 — rendering obiektu rozpraszającego światło.

Rysunek 5.7. *Współczynnik rozproszenia*



Rysunek 5.8. *Rendering obiektu z uwzględnieniem współczynnika rozproszenia*



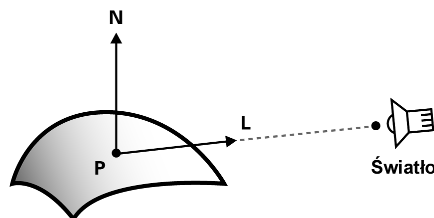
Oto wzór matematyczny używany do obliczenia współczynnika rozproszenia (*diffuse*) — patrz rysunek 5.9.

$$diffuse = K_d \times lightColor \times \max(N L, 0)$$

gdzie:

- K_d to kolor rozproszenia materiału,
- *lightColor* to kolor padającego światła,
- N to znormalizowana normalna powierzchni,
- L to znormalizowany wektor skierowany w stronę źródła światła,
- P to cieniowany punkt.

Rysunek 5.9. *Obliczanie natężenia światła rozproszonego*

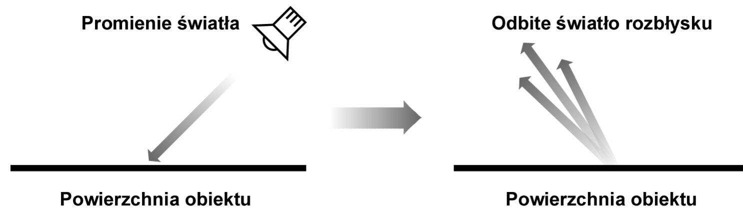


Iloczyn skalarny znormalizowanych wektorów N i L jest miarą kąta między tymi wektorami. Im mniejszy kąt między wektorami, tym większa będzie wartość iloczynu skalarnego a tym samym także ilość odbijanego światła będzie większa. Powierzchnia, której normalna jest zwrócona w tym samym kierunku,

co wektor światła, spowoduje powstanie ujemnej wartości iloczynu skalarnego, więc max($N L$, 0) z równania zapewnia, że dla tej powierzchni nie pojawi się kolor rozproszenia.

Współczynnik rozbłysku

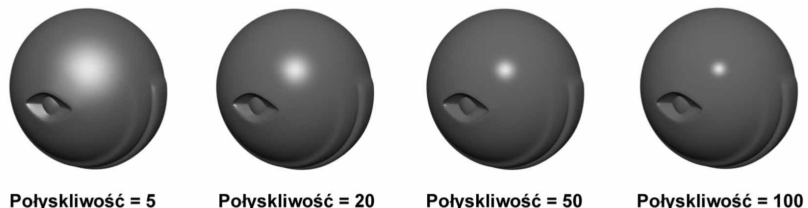
Współczynnik rozbłysku reprezentuje światło odbite od powierzchni w podobny sposób, jak to się dzieje w przypadku lustra. Współczynnik ten ma duże znaczenie dla reprezentacji gładkich i lśniących powierzchni, na przykład wypolerowanego metalu. Rysunek 5.10 ilustruje koncepcję współczynnika rozbłysku a rysunek 5.11 — rendering obiektu z rozbłyskiem.



Rysunek 5.10. *Współczynnik rozbłysku*

Rysunek 5.11. *Rendering obiektu z uwzględnieniem współczynnika rozbłysku*

W odróżnieniu od współczynników emisji, otoczenia i rozproszenia, współczynnik rozbłysku zależy od punktu widzenia obserwatora. Jeśli patrzący nie znajduje się w położeniu, które otrzymuje odbite promienie, nie zauważy rozbłysku na powierzchni. Na współczynnik rozbłysku wpływa nie tylko kolor powierzchni i źródła światła, ale także ustawienie połyskliwości powierzchni. Bardziej błyszczące obiekty posiadają mniejszy i węższy rozbłysk, natomiast materiały o mniejszej połyskliwości mają większy, łagodniejszy rozbłysk. Na rysunku 5.12 przedstawiono ten sam obiekt z różnymi ustawieniami połyskliwości.



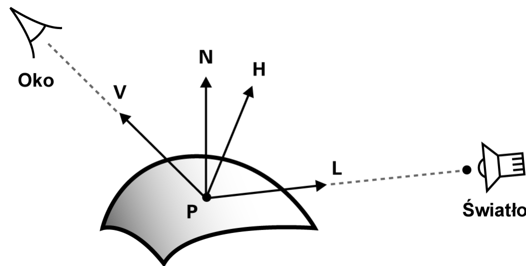
Rysunek 5.12. *Przykłady różnych wartości połyskliwości*

Oto wzór matematyczny, którego używamy do obliczenia współczynnika rozbłysku (*specular*) — ilustracja wzoru na rysunku 5.13).

$$specular = K_s \times lightColor \times facing \times (\max(N \cdot H, 0))^{shininess}$$

gdzie:

- K_s to kolor rozbłysku dla materiału,
- *lightColor* to kolor promieni świetlnych,
- N to znormalizowana normalna powierzchni,
- V to znormalizowany wektor zwrócony w stronę widza,
- L to znormalizowany wektor zwrócony w stronę źródła światła,
- H to znormalizowany wektor w połowie między V i L,
- P to analizowany punkt powierzchni,
- *facing* (skierowanie) jest równe 1, jeśli $N \cdot L$ jest większe od zera, w przeciwnym razie wynosi 0.



Rysunek 5.13. *Obliczanie współczynnika rozbłysku*

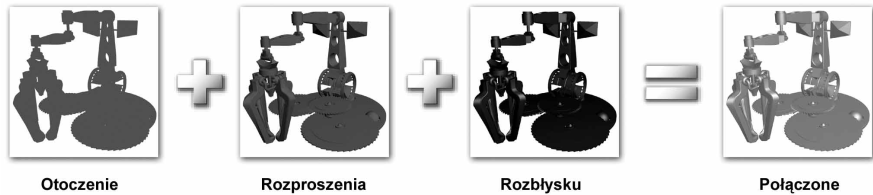
Gdy kąt między wektorem widoku V i wektorem połowy kąta H jest niewielki, na powierzchni obiektu pojawia się rozbłysk. Obliczanie wykładnika z iloczynu skalarnego N i H zapewnia szybki zanik rozbłysku, gdy wektory N i H zaczynają się rozchodzić.

Dodatkowo wymuszamy wyzerowanie współczynnika, jeśli współczynnik rozproszenia jest równy zero z powodu ujemnej wartości iloczynu skalarnego N i L. W ten sposób mamy pewność, że rozbłysk nie pojawi się na powierzchni, której normalna jest odwrócona tyłem do źródła światła.

Dodanie współczynników do siebie

Połączenie współczynników otoczenia, rozproszenia i rozbłysku daje nam wynikowy model oświetlenia, co przedstawiono na rysunku 5.14. Nie zastosowaliśmy współczynnika emisji, ponieważ jest on stosowany w zasadzie tylko w efektach specjalnych.

Rysunek 5.14. *Łączenie współczynników*



Uproszczenia

Czytelnik, który zna model stosowany w interfejsach OpenGL i Direct3D, zapewne zauważył wiele uproszczeń podstawowego modelu oświetlenia. Używamy globalnego modelu oświetlenia otoczenia zamiast osobnego modelu dla każdego źródła światła. Stosujemy także tę samą wartość dla kolorów rozproszenia i rozbłysku, choć powinno to być rozdzielone. Poza tym nie bierzemy pod uwagę zaniku i efektów świateł reflektorowych.

5.2.2. Program wierzchołków dla prostego oświetlenia opartego na wierzchołkach

W tym podrozdziale omówimy program Cg dla wierzchołków, który implementuje podstawowy model oświetlenia opisany w podrozdziale 5.2.1.

Program wierzchołków `C5E1v_basicLight` z przykładu 5.1 wykonuje dwa zadania:

- przekształca położenie wierzchołków z przestrzeni obiektu do przestrzeni przycięcia;
- oblicza kolor wierzchołka, używając współczynników emisji, otoczenia, rozproszenia i rozbłysku dla jednego źródła światła.

Przykład 5.1. *Program wierzchołków C5E1v_basicLight*

```
void C5E1v_basicLight(float4 position  : POSITION,
                      float3 normal    : NORMAL,

                  out float4 oPosition : POSITION,
                  out float4 color     : COLOR,

              uniform float4x4 modelViewProj,
              uniform float3 globalAmbient,
              uniform float3 lightColor,
              uniform float3 lightPosition,
              uniform float3 eyePosition,
              uniform float3 Ke,
              uniform float3 Ka,
```

```
               uniform float3 Kd,
               uniform float3 Ks,
               uniform float  shininess)
{
  oPosition = mul(modelViewProj, position);

  float3 P = position.xyz;
  float3 N = normal;

  // Obliczanie współczynnika emisji
  float3 emissive = Ke;

  // Obliczanie współczynnika otoczenia
  float3 ambient = Ka * globalAmbient;

  // Obliczanie współczynnika rozproszenia
  float3 L = normalize(lightPosition - P);
  float diffuseLight = max(dot(N, L), 0);
  float3 diffuse = Kd * lightColor * diffuseLight;

  // Obliczanie współczynnika rozbłysku
  float3 V = normalize(eyePosition - P);
  float3 H = normalize(L + V);
  float specularLight = pow(max(dot(N, H), 0), shininess);
  if (diffuseLight <= 0) specularLight = 0;
  float3 specular = Ks * lightColor * specularLight;

  color.xyz = emissive + ambient + diffuse + specular;
  color.w = 1;
}
```

W tym przykładzie wykonujemy obliczenia oświetlenia w przestrzeni obiektu. Można je także wykonać w innych przestrzeniach, jeśli przekształcimy do niej wszystkie potrzebne układy współrzędnych. Na przykład interfejsy OpenGL i Direct3D wykonują obliczenia w przestrzeni oka. Przestrzeń oka jest bardziej użyteczna w przypadku wielu źródeł światła, ale przestrzeń obiektu jest łatwiejsza w implementacji.

Ćwiczenia na końcu tego rozdziału wyjaśniają wady i zalety obliczania oświetlenia w przestrzeni oka i przestrzeni obiektu.

Dane dostarczane przez aplikacje

W tabeli 5.1 zamieszczono listę danych, które aplikacja musi dostarczyć do potoku graficznego. Dana oznaczona jest jako *zmienna*, jeśli zmienia się dla każdego wierzchołka i jako *jednorodna*, jeśli zmienia się rzadko (na przykład raz na obiekt).

Tabela 5.1. *Dane przekazywane do potoku graficznego przez aplikację*

Parametr	Nazwa zmiennej	Typ	Kategoria
PARAMETRY GEOMETRYCZNE			
położenie wierzchołka w przestrzeni obiektu	`position`	`float4`	zmienna
normalna wierzchołka w przestrzeni obiektu	`normal`	`float3`	zmienna
połączone macierze model-widok i perspektywy	`modelViewProj`	`float4x4`	jednorodna
położenie światła w przestrzeni obiektu	`lightPosition`	`float3`	jednorodna
położenie oka w przestrzeni obiektu	`eyePosition`	`float3`	jednorodna
PARAMETRY ŚWIATŁA			
kolor światła	`lightColor`	`float3`	jednorodna
globalny kolor otoczenia	`globalAmbient`	`float3`	jednorodna
PARAMETRY MATERIAŁU			
wartość emisji	`Ke`	`float3`	jednorodna
wartość otoczenia	`Ka`	`float3`	jednorodna
wartość rozproszenia	`Kd`	`float3`	jednorodna
wartość rozbłysku	`Ks`	`float3`	jednorodna
połyskliwość	`shininess`	`float`	jednorodna

Wskazówka dotycząca testowania

Łatwo zauważyć, że kod obliczający model oświetlenia jest bardziej skomplikowany od wszystkich poprzednich programów opisanych w niniejszej książce. Gdy pracujemy nad złożonym programem, warto tworzyć go fragment po fragmencie. Sprawdzamy program po dodaniu każdej nowej funkcji lub elementu, by przekonać się, że działa tak, jak tego oczekujemy. Podejście polegające na napisaniu całego kodu i dopiero późniejszym jego testowaniu nie jest zbyt rozsądne. W przypadku popełnienia błędu jego odnalezienie będzie znacznie łatwiejsze, gdy zna się poprzednią poprawną wersję programu.

Wskazówka ta w szczególności dotyczy kodu oświetlenia, ponieważ obliczanie modelu oświetlenia można podzielić na kilka etapów (emisja, otoczenie, rozproszenie, rozbłysk). Z tego powodu warto najpierw obliczyć współczynnik `emissive` a następnie ustawić `color` na `emissive`. Następnie obliczyć `ambient` i ustawić `color` na `ambient` plus `emissive`. Tworząc program Cg w ten sposób można uniknąć wielu błędów i problemów.

Kod programu wierzchołków

Obliczanie położenia w przestrzeni przycięcia

Zaczynamy od obliczenia położenia wierzchołka w przestrzeni przycięcia w celu przekazania jej do rasteryzera (opisywaliśmy to zadanie w rozdziale 4.).

```
oPosition = mul(modelViewProj, position);
```

Następnie tworzymy kopię zmiennej, aby zapamiętać położenie wierzchołka w przestrzeni obiektu, ponieważ ta informacja będzie nam potrzebna w przyszłości. Stosujemy zmienną tymczasową typu `float3`, ponieważ wszystkie pozostałe wektory oświetlenia (normalna powierzchni, położenie światła i położenie oka) także są typu `float3`.

```
float3 P = position.xyz;
```

Warto zauważyć specjalny rodzaj składni: `position.xyz`. To pierwsza wzmianka w tej książce o cesze języka Cg zwanej *przemieszaniem*.

Przemieszanie

Przemieszanie umożliwia zmianę kolejności komponentów i utworzenie nowego wektora zawierającego komponenty w takiej kolejności, jaką określi programista. W przypadku przemieszania używa się tego samego operatora kropki, co w przypadku dostępu do elementów struktury oraz informacji o nowej kolejności komponentów. Po znaku kropki może się znaleźć dowolna kombinacja liter `x`, `y`, `z` i `w`. W przypadku kolorów RGB można też zastosować litery `r`, `g`, `b` i `a`. Litery te wskazują, które komponenty oryginalnego wektora posłużą do utworzenia nowego wektora. Litery `x` i `r` odpowiadają pierwszemu komponentowi, `y` i `b` drugiemu itd. W poprzednim przykładzie `position` była zmienna typu `float4`. Zastosowanie `.xyz` powoduje wydobycie komponentów x, y i z ze zmiennej `position` i umieszczenie ich w nowym wektorze. Nowy wektor przypisywany jest do zmiennej `P` typu `float3`.

Języki C i C++ nie obsługują przemieszania, ponieważ nie zawierają wbudowanej obsługi typów wektorowych. Przemieszanie jest ważnym elementem języka Cg, gdyż zwiększa wydajność kodu.

Oto kilka innych przykładów zastosowania przemieszania.

```
float vec1 = float4(4.0, -2.0, 5.0, 3.0);
float2 vec2 = vec1.yx;     // vec2 = (-2.0, 4.0)
float scalar = vec1.w;     // scalar = 3.0
float3 vec3 = scalar.xxx;  // vec3 = (3.0, 3.0, 3.0)
```

Warto dokładnie przyjrzeć się tym czterem wierszom kodu. Pierwszy wiersz deklaruje zmienną vec1 typu float4. Drugi wiersz przypisuje komponenty y i x z vec1 do nowego wektora typu float2. Wektor jest następnie przypisany do vec2. W trzecim wierszu komponent w z vec1 jest przypisywany do typu float (zmiennej o nazwie scalar). W ostatnim wierszu tworzymy wektor float3, trzykrotnie kopiując wartość zmiennej scalar. Jest to nazywane rozmazywaniem i pozwala zauważyć, że Cg traktuje wartości skalarne jak wektory jednokomponentowe (wartości skalarne odczytujemy końcówką .x).

Możliwe jest także przemieszczanie macierzy w celu tworzenia wektorów bazujących na ciągu elementów macierzy. W tym celu używamy notacji ._m<wiersz><kolumna>. Można połączyć ze sobą kilka przemieszczeń macierzy, aby otrzymać wektor o odpowiednim rozmiarze. Oto przykład.

```
float4x4 myMatrix;
float myFloatScalar;
float4 myFloatVec4;

// Ustawia myFloatScalar na myMatrix[3][2]
myFloatScalar = myMatrix._32;

// Przypisuje główną przekątną macierzy myMatrix do myFloatVec4
myFloatVec4 = myMatrix._m00_m11_m22_m33;
```

Dodatkowo można uzyskać dostęp do poszczególnych wierszy macierzy za pomocą operatora tablicy []. Używając zmiennej zadeklarowanej w poprzednim kodzie, możemy napisać.

```
// Ustawiamy myFloatVector na pierwszy wiersz myMatrix
myFloatVec4 = myMatrix[0];
```

Maskowanie zapisu

Język Cg obsługuje także inną operację, związaną z przemieszaniem, nazywaną *maskowaniem zapisu*, która umożliwia aktualizację tylko niektórych komponentów wektora w trakcie zapisu. Możemy na przykład zapisać wartości tylko do komponentów *x* i *w* wektora float4, używając wektora float2.

```
// Zakładamy, że na początku wektory mają wartości:
// vec1 = (4.0, -2.0, 5.0, 3.0)
// vec2 = (-2.0, 4.0);
vec1.xw = vec2;   // Teraz vec1 = (-2.0, -2.0, 5.0, 4.0)
```

Operator maskowania zapisu może zawierać komponenty x, y, z i w (lub r, g, b i a) w dowolnej kolejności. Każda z liter może wystąpić co najwyżej raz w danym maskowaniu zapisu. Nie można mieszać liter xyzw i rgba w jednym operatorze.

W większości nowoczesnych procesorów graficznych operacje przemieszania i maskowania nie wpływają na wydajność wykonywanych działań. Warto więc z nich korzystać, gdy tylko pozwalają zwiększyć szybkość działania i czytelność kodu.

Obliczanie emitowanego światła

W przypadku emitowanego światła w zasadzie nie wykonujemy żadnych obliczeń. Aby zwiększyć czytelność kodu, tworzymy zmienną o nazwie `emissive` zawierającą współczynnik emisji światła.

```
// Obliczenie współczynnika emisji
float3 emissive = Ke;
```

Gdy kompilator Cg przekształca kod do postaci wykonywalnej, optymalizuje go, aby nie pojawił się spadek wydajności wynikający z korzystania ze zmiennych tymczasowych, na przykład `emissive`. Zastosowanie tej zmiennej czyni kod czytelniejszym, zatem warto stosować kopie niektórych zmiennych. Nie będzie to miało żadnego wpływu na wydajność kodu wykonywalnego.

Współczynnik oświetlenia otoczenia

Czytelnik zapewne przypomina sobie, że w przypadku potrzeby obliczenia współczynnika oświetlenia należy pomnożyć kolor otoczenia materiału (`Ka`) przez globalny kolor otoczenia. Jest to wymnażanie poszczególnych elementów wektora, czyli poszczególne komponenty koloru `Ka` mnożymy przez odpowiednie komponenty globalnego oświetlenia. Zadanie to wykona poniższy kod, który korzysta z maskowania i przemieszania.

```
// Mało wydajny sposób obliczenia współczynnika otoczenia
float3 ambient;
ambient.x = Ka.x * globalAmbient.x;
ambient.y = Ka.y * globalAmbient.y;
ambient.z = Ka.z * globalAmbient.z;
```

Przedstawiony kod jest poprawny, ale nie jest elegancki ani wydajny. Język Cg obsługuje operacje na wektorach, zatem możemy tego rodzaju operację wyrazić w spójniejszy sposób. Poniżej przedstawiamy bardziej elegancki sposób skalowania wektora przez wektor.

```
// Obliczenie współczynnika otoczenia
float3 ambient = Ka * globalAmbient;
```

Proste, prawda? Wbudowana w język Cg obsługa podstawowych operacji na wektorach i macierzach jest bardzo pomocna.

Współczynnik światła rozproszenia

Powoli przechodzimy do coraz bardziej interesujących obliczeń przydatnych w opisywaniu modelu oświetlenia. W przypadku koloru rozproszenia potrzebujemy wektora skierowanego od wierzchołka do źródła światła. Aby zdefiniować wektor, od punktu końcowego odejmuje się jego początek. W tym przypadku wektor kończy się w położeniu `lightPosition` a zaczyna w `P`.

```
// Obliczanie wektora światła
float3 L = normalize(lightPosition - P);
```

Jesteśmy zainteresowani tylko kierunkiem a nie wartością, więc normalizujemy wektor. Standardowa biblioteka Cg zawiera funkcję `normalize`, która zwraca znormalizowany wektor. Jeśli nie znormalizujemy wektora, otrzymane oświetlenie obiektu będzie zbyt jasne lub zbyt ciemne.

`normalize(v)`	zwraca znormalizowaną wersję wektora `v`

Następnie wykonuje się rzeczywiste obliczenia oświetlenia. Jest to złożone równanie, więc warto przeanalizować je fragment po fragmencie. Najpierw pojawia się iloczyn skalarny. Przypomnijmy, że iloczyn skalarny to prosta funkcja matematyczna, która oblicza jedną wartość reprezentującą cosinus kąta między dwoma wektorami jednostkowymi. W języku Cg do obliczania iloczynu skalarnego używamy funkcji `dot`.

`dot(a, b)`	zwraca iloczyn skalarny wektorów `a` i `b`

Z tego powodu fragment kodu, który znajduje iloczyn skalarny między `N` i `L` ma postać.

```
dot(N, L);
```

Pojawia się jednak pewien problem. Powierzchnia ustawiona tyłem do światła otrzyma oświetlenie ujemne, ponieważ iloczyn skalarny zwróci wartość ujemną, gdy normalna jest odwrócona od źródła światła. Ujemne wartości oświetlenia nie mają żadnej podstawy fizycznej i spowodują powstanie błędów w równaniu oświetlenia. Aby uniknąć problemów, musimy wartości ujemne zamienić na zero, czyli dla wartości ujemnych iloczynu skalarnego do dalszych obliczeń przekazać wartość zero. Operację przycięcia łatwo przeprowadzić, używając funkcji `max`.

`max(a, b)`	zwraca maksimum z `a` i `b`

Połączenie dwóch poprzednich operacji spowoduje powstanie następującego kodu.

```
max(dot(N, L), 0);
```

Cały kod obliczający światło rozproszenia ma postać.

```
float diffuseLight = max(dot(N, L), 0);
```

Następnie musimy jeszcze pomnożyć ze sobą kolor rozproszenia materiału (`Kd`) z kolorem rozproszenia światła (`lightColor`). Obliczona wartość `diffuseLight` to wartość skalarna. Warto zapamiętać, że w języku Cg można pomnożyć wektor przez skalar. Spowoduje to przeskalowanie wszystkich elementów wektora przez skalar. Możemy więc połączyć obliczenia dla wszystkich kolorów w dwóch operacjach mnożenia.

```
float3 diffuse = Kd * lightColor * diffuseLight;
```

Współczynnik światła rozbłysku

Obliczenie współczynnika światła rozbłysku wymaga największej liczby operacji. Warto przyjrzeć się jeszcze raz rysunkowi 5.13, który przedstawia różne potrzebne wektory. Wektor `L` obliczyliśmy już wcześniej dla koloru rozproszenia, ale potrzebujemy jeszcze wektorów `V` i `H`. Uzyskanie ich nie jest trudne, ponieważ znamy położenie oka (`eyePosition`) i położenie wierzchołka (`P`).

Zaczynamy od znalezienia wektora od wierzchołka do położenia oka. Wektor ten typowo nazywany jest *wektorem widoku* (w naszym przykładzie jako zmienna `V`). Jako że interesuje nas tylko kierunek, powinniśmy znormalizować wektor. Zadanie to wykonuje poniższy kod.

```
float3 V = normalize(eyePosition - P);
```

Następnie obliczamy `H`, czyli wektor znajdujący się w połowie między wektorem światła `L` i wektorem widoku `V`. Wektor ten nazywany jest *wektorem połowy kąta*. Musimy znormalizować `H`, gdyż interesuje nas tylko kierunek.

```
// Mało wydajny sposób obliczenia H
float3 H = normalize(0.5 * L + 0.5 * V);
```

Skoro wykonujemy operację normalizacji, skalowanie `V` i `L` przez 0.5 nie ma żadnego sensu (normalizacja i tak zniesie te mnożenie). Z tego powodu możemy skrócić kod do następującego.

```
float3 H = normalize(L + V);
```

Teraz można już przystąpić do obliczeń współczynnika rozbłysku. Podobnie jak w przypadku współczynnika rozproszenia, będziemy budowali wyrażenie krok po kroku. Zaczniemy od iloczynu skalarnego `H` i `N`.

```
dot(H, N)
```

Musimy zastosować ograniczenie od dołu do zera, podobnie jak podczas obliczania współczynnika oświetlenia rozproszonego.

```
max(dot(H, N), 0)
```

Wynik musimy podnieść do potęgi określonej parametrem `shininess`. Powoduje to zmniejszanie się rozbłysku wraz ze zwiększaniem wartości połyskliwości (`shininess`). Aby podnieć wartość do potęgi, stosujemy funkcję `pow`.

`pow(x, y)`	zwraca x^y

Po dodaniu funkcji potęgowania uzyskujemy następujące wyrażenie.

```
pow(max(dot(H, N), 0),shininess)
```

Po połączeniu wszystkiego razem otrzymujemy:

```
float specularLight = pow(max(dot(N, H), 0), shininess);
```

Musimy jeszcze zapewnić, by rozbłysk nie pojawiał się wtedy, gdy oświetlenie rozproszenia wynosi 0 (wtedy powierzchnia jest ustawiona tyłem do źródła światła). Innymi słowy, gdy oświetlenie rozproszenia jest równe 0, oświetlenie rozbłysku także ustawiamy na zero. W tym celu musimy wykorzystać instrukcje warunkowe języka Cg.

Instrukcje warunkowe

Podobnie jak w języku C, w języku Cg można wykorzystywać instrukcje warunkowe, słowa kluczowe `if` oraz `else`. Oto przykład.

```
if (value == 1) {
  color = float4(1.0, 0.0, 0.0, 1.0);   // kolor czerwony
} else {
  color = float4(0.0, 1.0, 0.0, 1.0);   // kolor zielony
}
```

Podobnie jak w języku C możemy stosować notację `?:`, aby w zwięzły sposób zaimplementować instrukcje warunkowe.

```
(sprawdzane wyrażenie) ? (instrukcje, jeśli prawda)
                       : (instrukcje, jeśli fałsz)
```

Poprzedni przykład można więc zapisać następująco.

```
color = (value == 1) ? float4(1.0, 0.0, 0.0, 1.0)
                     : float4(0.0, 1.0, 0.0, 1.0);
```

Powracamy do przykładu, by napisać kod sprawdzający warunek dla obliczania współczynnika światła rozproszonego.

```
float specularLight = pow(max(dot(N, H), 0), shininess);
```

```
if (diffuseLight <= 0) specularLight = 0;
```

Podobnie, jak w przypadku obliczeń współczynnika światła rozproszonego, należy dokonać wymnożenia przez kolor rozbłysku materiału (`Ks`) i kolor światła (`lightColor`). Stosowanie dwóch kolorów do sterowania rozbłyskiem może się początkowo wydawać dziwne, ale taka elastyczność jest użyteczna, ponieważ pewne materiały (na przykład metale) mają rozbłysk w kolorze materiału, ale inne materiały (na przykład plastiki) mają rozbłysk w kolorze białym. Oba rozbłyski są następnie modyfikowane przez kolor światła. Zmienne `Ks` i `lightColor` umożliwiają modyfikację modelu w celu uzyskania odpowiedniego wyglądu obiektu.

Komponent rozbłysku obliczamy w następujący sposób.

```
float3 specular = Ks * lightColor * specularLight;
```

Łączymy wszystko razem

Na końcu łączymy współczynniki emisji, otoczenia, rozproszenia i rozbłysku, aby uzyskać wynikowy kolor wierzchołka. Kolor ten przypisujemy do parametru wyjściowego o nazwie `color`.

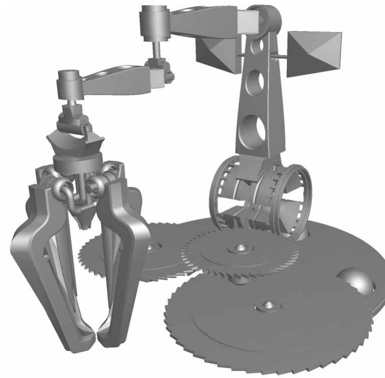
```
color.xyz = emissive + ambient + diffuse + specular;
```

5.2.3. Program fragmentów dla modelu oświetlenia wykorzystującego wierzchołki

Obliczenia oświetlenia są wykonywane w programie wierzchołków, zatem program fragmentów musi jedynie interpolować kolor i przekazać go do bufora ramki. W tym celu korzystamy z programu `C2E2f_passthrough`.

5.2.4. Efekt modelu oświetlenia opartego na wierzchołkach

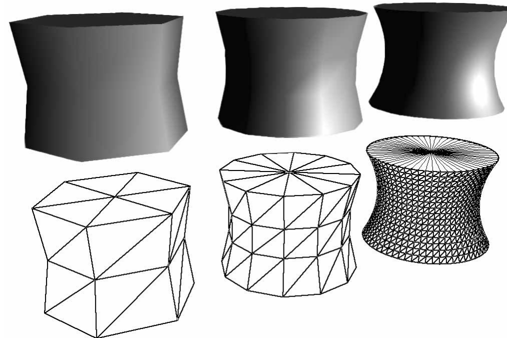
Na rysunku 5.15 przedstawiono przykładowy rendering, który wykorzystuje program obliczający model oświetlenia dla wierzchołków.



Rysunek 5.15.
Efekt modelu oświetlenia opartego na wierzchołkach

5.3. Model oświetlenia oparty na fragmentach

Zapewne każdy zauważy, że wyniki zastosowania modelu oświetlenia wykorzystującego wierzchołki nie są idealne. Cieniowanie wygląda na trójkątne, czyli można rozpoznać strukturę siatki, jeśli jest bardzo prosta. Jeśli obiekt zawiera niewiele wierzchołków, oparty na nich model oświetlenia da niedokładny obraz. Wystarczy jednak zwiększyć szczegółowość siatki, aby zauważyć znaczącą poprawę (patrz rysunek 5.16). Na rysunku przedstawiono trzy walce o różnej złożoności siatki. Poniżej wersji uwzględniającej oświetlenie znajduje się odpowiedni model siatki. Złożoność siatki rośnie od lewej do prawej — jakość modelu oświetlenia ulega znacznej poprawie.



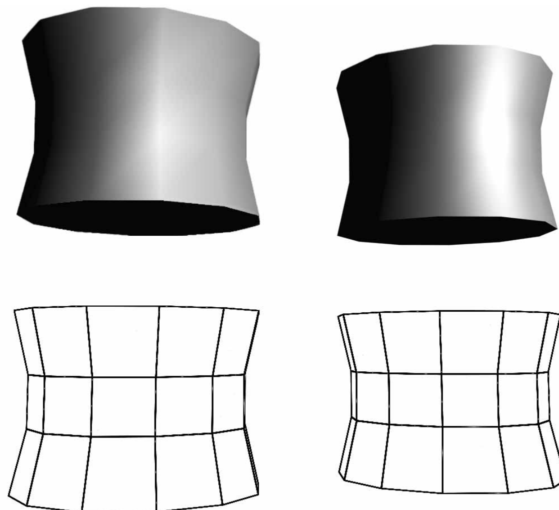
Rysunek 5.16.
Wpływ złożoności siatki na efekt oświetlenia

Prostsze modele wyglądają nieciekawie w przypadku zastosowania modelu oświetlenia opartego na wierzchołkach z powodu niedoskonałej interpolacji danych. W tego rodzaju modelu oświetlenia wartości obliczane są tylko dla wierzchołków. Następnie w każdym z trójkątów zachodzi interpolacja oświetlenia dla konkretnych fragmentów. Tego rodzaju podejście nazywane jest płynną interpolacją koloru lub *cieniowaniem Gourauda*. Brak w nim szczegółowości, ponieważ

równanie oświetlenia nie jest obliczane dla każdego fragmentu. Na przykład rozbłysk, który nie znajduje się w żadnym z wierzchołków trójkąta (ale na przykład w jego środku) nie będzie widoczny.

Dokładnie taka sytuacja występuje w przedstawionym przed chwilą przykładzie modelu oświetlenia dla wierzchołków — program wierzchołków obliczył oświetlenie a następnie jednostka rasteryzacji dokonała interpolacji kolorów dla fragmentów.

Aby uzyskać bardziej dokładne wyniki, musimy dokonywać obliczeń modelu oświetlenia dla każdego fragmentu (a nie wierzchołka). Zamiast interpolować wynikowy kolor, interpolujemy normalne powierzchni. Następnie program fragmentów wykorzystuje normalne, aby obliczyć oświetlenie dla każdego piksela. Technikę tę nazywamy *cieniowaniem Phong* (nie należy jej mylić z modelem oświetlenia Phong, które dotyczy aproksymacji rozbłysków z modelu podstawowego) lub bardziej ogólnie *oświetleniem opartym na fragmentach*. Tego rodzaju model oświetlenia umożliwia osiągnięcie lepszych wyników, ponieważ całe równanie oświetlenia jest obliczane dla każdego fragmentu każdego trójkąta (patrz rysunek 5.17). Z lewej strony rysunku znajduje się walec renderowany za pomocą oświetlenia dla wierzchołków a po prawej za pomocą oświetlenia dla fragmentów. Oba walce cechuje ten sam poziom złożoności siatki. Warto zauważyć, że rozbłyski na lewym walcu są bardziej rozmyte niż na prawym.



Rysunek 5.17. *Porównanie modelu oświetlenia dla wierzchołków i modelu oświetlenia dla fragmentów*

5.3.1. Implementacja modelu oświetlenia opartego na fragmentach

CineFX® Program fragmentów z tego przykładu wymaga procesora graficznego czwartej generacji, czyli GeForce FX firmy NVIDIA lub Radeon 9700 firmy ATI.

W tym przykładzie przeniesiemy wszystkie obliczenia oświetlenia z programu wierzchołków do programu fragmentów. Program wierzchołków posłuży tylko do przygotowania i przesłonienia niektórych parametrów do programu fragmentów. Wzorzec ten stosuje wiele zaawansowanych technik, ponieważ programy fragmentów dają lepszą kontrolę nad końcowym obrazem niż programy wierzchołków. Oczywiście z wyborem programu wierzchołków lub fragmentów związane są też inne kwestie, na przykład wydajność. Dokładniej ten temat omawiamy w rozdziale 10.

Czytelnik przekona się, że program fragmentów wykonujący obliczenia oświetlenia jest bardzo podobny do programu wierzchołków wykonującego to samo zadanie. Wynika to z faktu, że język Cg umożliwia stosowanie tej samej składni w programach wierzchołków i fragmentów. Nie musimy więc po raz drugi tłumaczyć tych samych wierszy kodu. Podobnie jak poprzednio wszystkie obliczenia wykonujemy w przestrzeni obiektu, gdyż taka implementacja jest prostsza.

5.3.2. Program wierzchołków dla modelu oświetlenia opartego na fragmentach

Program wierzchołków w tym przykładzie to tylko łącznik: wykonuje minimalne obliczenia i przekazuje wszystkie potrzebne dane do kolejnych etapów potoku graficznego, aby program fragmentów mógł wykonać niezbędne obliczenia. Po zapisaniu położenia homogenicznego, program wierzchołków przekazuje położenie i normalną w przestrzeni obiektu jako współrzędne tekstury jednostek teksturowania 0 i 1.

Przykład 5.2 zawiera pełny kod źródłowy programu wierzchołków. Nie stosujemy w nim żadnych nowych technik, więc wszystkie wiersze kodu powinny być zrozumiałe.

Przykład 5.2. *Program wierzchołków C5E2v_fragmentLighting*

```
void C5E2v_fragmentLighting(float4 position : POSITION,
                            float3 normal   : NORMAL,

                        out float4 oPosition : POSITION,
```

```
                         out float3 objectPos : TEXCOORD0,
                         out float3 oNormal   : TEXCOORD1,

                     uniform float4x4 modelViewProj)
{
  oPosition = mul(modelViewProj, position);
  objectPos = position.xyz;
  oNormal = normal;
}
```

5.3.3. Program fragmentów dla modelu oświetlenia opartego na fragmentach

Program `C5E3f_basicLight` jest prawie taki sam jak program wierzchołków z poprzedniego przykładu oświetlenia, więc nie będziemy zagłębiali się w szczegóły. Przykład 5.3 przedstawia kod źródłowy dla programu modelu oświetlenia opartego na fragmentach.

Przykład 5.3. *Program fragmentów C5E3f_basicLight*

```
void C5E3f_basicLight(float4 position  : TEXCOORD0,
                      float3 normal    : TEXCOORD1,

                  out float4 color     : COLOR,

              uniform float3 globalAmbient,
              uniform float3 lightColor,
              uniform float3 lightPosition,
              uniform float3 eyePosition,
              uniform float3 Ke,
              uniform float3 Ka,
              uniform float3 Kd,
              uniform float3 Ks,
              uniform float  shininess)
{
  float3 P = position.xyz;
  float3 N = normal;

  // Obliczenie współczynnika emisji
  float3 emissive = Ke;

  // Obliczenie współczynnika otoczenia
  float3 ambient = Ka * globalAmbient;

  // Obliczenie współczynnika rozproszenia
  float3 L = normalize(lightPosition - P);
  float diffuseLight = max(dot(L, N), 0);
  float3 diffuse = Kd * lightColor * diffuseLight;

  // Obliczenie współczynnika rozbłysku
  float3 V = normalize(eyePosition - P);
  float3 H = normalize(L + V);
  float specularLight = pow(max(dot(H, N), 0), shininess);
```

```
    if (diffuseLight <= 0) specularLight = 0;
    float3 specular = Ks * lightColor * specularLight;

    color.xyz = emissive + ambient + diffuse + specular;
    color.w = 1;
}
```

Często zakłada się, że normalna wierzchołka w przestrzeni obiektu jest już znormalizowana. W takim przypadku nie byłaby nam potrzebna normalizacja interpolowanej normalnej dla fragmentu.

```
float3 N = normalize(normal);
```

Normalizacja jest jednak potrzebna, ponieważ interpolacja liniowa współrzędnych tekstury może spowodować denormalizację wektora.

Program fragmentów `C5E3f_basicLight` wyraźnie uwidacznia, że język Cg umożliwia opisanie pomysłów programisty w ten sam sposób w programach fragmentów i wierzchołków (oczywiście przy założeniu, że posiadamy wystarczająco nowoczesny procesor graficzny — przedstawiony program wymaga procesorów czwartej generacji). Oczywiście przeprowadzanie obliczeń dla fragmentów jest obarczone większym kosztem. W każdej klatce jest zdecydowanie więcej fragmentów niż wierzchołków, co oznacza, że program będzie wykonywany o wiele częściej. Wynika z tego, że długie programy fragmentów mają znacznie większy wpływ na wydajność niż długie programy wierzchołków. W rozdziale 10. dokładniej omówimy zalety i wady programów wierzchołków i fragmentów.

W pozostałych rozdziałach staramy się unikać złożonych programów fragmentów, aby przykłady mogły być uruchamiane dla większego zbioru procesorów graficznych. Zazwyczaj możemy przenieść obliczenia wykonywane dla wierzchołków do programu fragmentów.

5.4. Tworzenie funkcji modelu oświetlenia

W poprzednim podrozdziale po prostu skopiowaliśmy większość kodu z przykładu dla wierzchołków do przykładu dla fragmentów, ale istnieje lepsze rozwiązanie: należy wszystkie ważniejsze obliczenia umieścić w funkcji modelu oświetlenia.

W złożonym programie Cg obliczanie modelu oświetlenie może być tylko jedną z kilku wykonywanych operacji. W przedstawionych przykładach obliczenie wynikowego oświetlenia wymagało wykonania kilku kroków. Zapewne nikt nie zechce ponownie pisać całego kodu w razie potrzeby obliczenia modelu oświetlenia. Nie trzeba tego robić. W rozdziale 2. wspomnieliśmy o tym, że możemy napisać funkcję wewnętrzną, która zawrze w sobie obliczenia oświetlenia. Z tej samej funkcji możemy korzystać w różnych funkcjach wejścia.

W odróżnieniu od funkcji języka C lub C++, funkcje języka Cg są typowo wstawiane w miejscu wywołania (choć zależy to od profilu — profile zaawansowane, na przykład vp30, obsługują wywołania funkcji oraz wstawianie w miejscu wywołania). Wstawienie funkcji w miejscu wywołania oznacza, że nie pojawi się dodatkowy narzut związany z wywołaniem funkcji. Możemy więc korzystać z funkcji wszędzie tam, gdzie chcemy zwiększyć czytelność kodu, ułatwić testowanie, zwiększyć ilość wielokrotnie używanego kodu i zapewnić łatwiejsze wprowadzanie optymalizacji.

Język Cg, podobnie jak język C, wymaga zadeklarowania funkcji przed jej użyciem.

5.4.1. Deklarowanie funkcji

W języku Cg funkcje deklarujemy w ten sam sposób, co w języku C. Opcjonalnie określamy przekazywane parametry oraz wartość zwracaną przez funkcję. Oto prosta deklaracja funkcji.

```
float getX(float3 v)
{
  return v.x;
}
```

Funkcja jako parametr przyjmuje trójelementowy wektor v a zwraca wartość typu float będącą komponentem *x* wektora v. Słowo kluczowe return służy do określenia zwracanej wartości. Funkcję getX wywołujemy w taki sam sposób, jak inne funkcję języka Cg.

```
// Deklarujemy wektor rysy
float3 myVector = float3(0.5, 1.0, -1.0);

// Pobieramy komponent x z myVector
float x = getX(myVector);
// Teraz x = 0.5
```

Czasem wymaga się, aby funkcja zwróciła kilka wyników zamiast jednego. W takiej sytuacji korzystamy z modyfikatora out (omówionego w podrozdziale 3.3.4), który informuje, że dany parametr jest parametrem wyjściowym. Oto przykład funkcji, która przyjmuje wektor i zwraca komponenty *x*, *y* i *z*.

```
void getComponents(float3 vector,
                   out float x,
                   out float y,
                   out float z)
{
  x = vector.x;
  y = vector.y;
  z = vector.z;
}
```

Zauważmy, że funkcję zadeklarowano jako `void`, ponieważ wszystkie wartości są zwracane przez parametry. Dalsza część kodu obrazuje sposób korzystania z funkcji `getComponents`.

```
// Deklaracja wektora rysy
float3 myVector = float3(0.5, 1.0, -1.0);

// Deklaracja zmiennych rysy
float x, y, z;

// Pobranie komponentów x, y i z z myVector
getComponents(myVector, x, y, z);
// Teraz x = 0.5, y = 1.0, z = -1.0
```

5.4.2. Funkcja oświetlenia

Tworzenie modelu oświetlenia obiektów jest złożonym procesem, zatem można napisać wiele różnych funkcji oświetlenia przyjmujących różne parametry. Na razie jednak zajmiemy się modelem podstawowym i utworzymy dla niego odpowiednią funkcję. Oto pierwszy fragment funkcji.

```
float3 lighting(float3 Ke,
                float3 Ka,
                float3 Kd,
                float3 Ks,
                float shininess,
                float3 lightPosition,
                float3 lightColor,
                float3 globalAmbient,
                float3 P,
                float3 N,
                float3 eyePosition)
{
  // Tutaj wykonujemy obliczenia modelu oświetlenia
}
```

Jedną z wad tego rozwiązania jest to, że funkcja wymaga dużej liczby parametrów. Warto byłoby podzielić parametry na dwie grupy: grupę materiałów i grupę światła a następnie przekazać każdą z grup jako jeden parametr. Język Cg obsługuje struktury, które do tego celu nadają się wprost idealnie.

5.4.3. Struktury

Wspomnieliśmy już w rozdziale 2., że struktury języka Cg deklaruje się w taki sam sposób, jak w języku C lub C++. Stosuje się słowo kluczowe `struct` i podaje listę elementów struktury. Tutaj przedstawiamy przykład struktury, która

zawiera wszystkie parametry dotyczące materiału dla podstawowego modelu oświetlenia.

```
struct Material {
  float3 Ke;
  float3 Ka;
  float3 Kd;
  float3 Ks;
  float shbininess;
};
```

Operator kropki umożliwia dostęp do poszczególnych członków struktury. Dalej znajduje się kod obrazujący sposób deklaracji i dostępu do struktury.

```
Material shiny;
shiny.Ke = float3(0.0, 0.0, 0.0);
shiny.Ka = float3(0.1, 0.2, 0.1);
shiny.Kd = float3(0.2, 0.4, 0.2);
shiny.Ks = float3(0.8, 0.8, 0.8);
shiny.shininess = 90.0;
```

Podobną strukturę możemy zadeklarować dla źródła światła.

```
struct Light {
  float4 position;
  float3 color;
};
```

Teraz ponownie zdefiniujemy funkcję modelu oświetlenia, stosując struktury jako parametry.

```
float3 lighting(Material material,
                Light light,
                float3 globalAmbient,
                float3 P,
                float3 N,
                float3 eyePosition)
{
  // Tutaj wykonujemy obliczenia modelu oświetlenia
}
```

Teraz możemy rozbudować model oświetlenia lub materiału bez modyfikacji parametrów przyjmowanych przez funkcję oświetlenia. Dodatkową zaletą jest to, że w przypadku obliczeń dla kilku źródeł światła, możemy zastosować tablicę struktur `Light`.

5.4.4. Tablice

Język Cg obsługuje tablice w ten sam sposób, co język C. Obecna wersja Cg nie obsługuje wskaźników, zatem zawsze trzeba stosować składnię tablicową (za-

miast składni opartej na wskaźnikach) podczas uzyskiwania dostępu do tablicy. Oto przykład deklaracji i uzyskania dostępu do tablicy w języku Cg.

```
// Deklaracja tablicy czteroelementowej
float3 myArray[4];
int index = 2;

// Przypisanie wektora do drugiego elementu tablicy
myArray[index] = float3(0.1, 0.2, 0.3);
```

Zaawansowane Istnieje ważna różnica w sposobie działania tablic w języku C i Cg. Przypisane tablicy powoduje rzeczywiste skopiowanie całej tablicy a tablice przekazywane jako parametry są przekazywane przez wartość (przed dokonaniem jakichkolwiek zmian kopiowana jest cała tablica) a nie przez referencję.

Tablice możemy przekazywać do funkcji jako parametry. Skorzystamy z tej cechy, aby napisać funkcję obliczającą model oświetlenia obiektu z dwóch różnych źródeł światła (patrz przykład 5.4).

Przykład 5.4. *Program wierzchołków C5E4v_twoLights*

```
void C5E4v_twoLights(float4 position : POSITION,
                     float3 normal   : NORMAL,

                 out float4   oPosition : POSITION,
                 out float4   color     : COLOR,

             uniform float4x4 modelViewProj,
             uniform float3   eyePosition,
             uniform float3   globalAmbient,
             uniform Light    lights[2],
             uniform float    shininess,
             uniform Material material)
{
  oPosition = mul(modelViewProj, position);

 // obliczenie współczynników emisji i otoczenia
  float3 emissive = material.Ke;
  float3 ambient = material.Ka * globalAmbient;

  // przejście przez współczynniki rozproszenia i rozbłysku
  // dla każdego światła
  float3 diffuseLight;
  float3 specularLight;
  float3 diffuseSum  = 0;
  float3 specularSum = 0;
  for (int i = 0; i < 2; i++) {
    C5E5_computeLighting(lights[i], position.xyz, normal, eyePosition, shininess,
diffuseLight, specularLight);
    diffuseSum += diffuseLight;
    specularSum += specularLight;
  }

  // modyfikujemy rozproszenie i rozbłysk względem materiału
  float3 diffuse = material.Kd * diffuseSum;
```

```
  float3 specular = material.Ks * specularSum;

  color.xyz = emissive + ambient + diffuse + specular;
  color.w = 1;
}
```

Przedstawiony kod rozpoczyna się od obliczenia współczynników emisji i otoczenia, ponieważ są one niezależne od źródeł światła. Następnie w pętli `for` przechodzimy przez oba źródła światła i obliczamy dla nich współczynniki rozproszenia i rozbłysku. Same współczynniki są obliczane w funkcji pomocniczej `C5E5_computeLighting`, którą wkrótce się zajmiemy. Najpierw jednak przyjrzyjmy się pętli `for` i innym konstrukcjom sterującym wykonywaniem programu.

5.4.5. Sterowanie wykonywaniem programu

Język Cg obsługuje większość z konstrukcji sterujących wykonywaniem programu z języka C, a dokładniej zapewnia instrukcje:

- `return` i wywołania funkcji,
- `if-else`,
- `for`,
- `while` oraz `do-while`.

Ich działanie jest dokładnie takie samo jak w języku C, ale w niektórych profilach mogą pojawić się pewne ograniczenia, na przykład w niektórych profilach liczba wykonać pętli `for` i `while` musi być znana w trakcie kompilacji programu Cg.

Język Cg rezerwuje słowa kluczowe `goto` i `switch` znane z języka C, ale w obecnej wersji nie obsługuje tych konstrukcji.

5.4.6. Obliczenie modelu oświetlenia rozproszenia i rozbłysku

Pozostało nam jeszcze opisanie funkcji `C5E5_computeLighting`, która oblicza współczynniki rozproszenia i rozbłysku dla podanego źródła światła. Przykład 5.5 zawiera implementację wcześniej opisanego kodu obliczeń oświetlenia.

Przykład 5.5. *Funkcja wewnętrzna C5E5_computeLighting*

```
void C5E5_computeLighting(Light  light,
                          float3 position,
                          float3 normal,
```

```
                      float3 eyePosition,
                      float  shininess,

                  out float3 diffuseResult,
                  out float3 specularResult)
{
  // obliczenie rozproszenia światła
  float3 L = normalize(light.position - position);
  float diffuseLight = max(dot(normal, L), 0);
  diffuseResult = light.color * diffuseLight;

  // obliczenie rozbłysku
  float3 V = normalize(eyePosition - position);
  float3 H = normalize(L + V);
  float specularLight = pow(max(dot(normal, H), 0), shininess);
  if (diffuseLight <= 0) specularLight = 0;
  specularResult = light.color * specularLight;
}
```

5.5. Rozszerzenie modelu podstawowego

Skoro zaimplementowaliśmy podstawowy model oświetlenia, możemy zacząć go rozszerzać. W kolejnych podrozdziałach zajmiemy się następującymi zagadnieniami: zanikiem światła wraz ze wzrostem odległości, efektami reflektorów (światła stożkowe) i światłami kierunkowymi. Każde z rozszerzeń można zaimplementować na poziomie wierzchołków lub fragmentów.

Obliczanie modelu oświetlenia to bardzo złożony temat, więc istnieje wiele różnych technik rozwiązywania problemów. Naszym celem jest przedstawić bazę, na której Czytelnik będzie mógł samodzielnie zbudować skomplikowane efekty.

5.5.1. Zanik światła wraz z odległością

Podstawowy model zakładał taką samą intensywność światła niezależnie od tego, jak daleko oświetlana powierzchnia znajdowała się od źródła światła. Choć jest to dobra aproksymacja niektórych rodzajów oświetlenia (na przykład światła słonecznego), częściej zajmujemy się światłami, których intensywność zmniejsza się wraz ze wzrostem odległości. Własność tę nazywamy *zanikiem wraz z odległością*. W interfejsach OpenGL i Direct3D zanik (*attenuation*) dla konkretnego punktu obliczany jest za pomocą następującego wzoru.

$$attenuation = \frac{1}{k_c + k_L d + k_Q d^2}$$

gdzie:

- d to odległość od źródła światła,
- k_C, k_L i k_Q to stałe sterujące zanikiem.

W wzorze tym k_C, k_L i k_Q to, odpowiednio, stałe, liniowe i kwadratowe współczynniki zaniku. W rzeczywistym świecie intensywność źródła światła maleje według wzoru $1/d^2$, ale stosowanie tego wzoru nie zawsze daje pożądane efekty. Zastosowanie trzech parametrów umożliwia lepsze sterowanie całym efektem zaniku światła.

Współczynniki zaniku modyfikują współczynniki rozproszenia i rozbłysku równania oświetlenia. Równanie ma teraz następującą postać.

$$oświetlenie = emisja + otoczenie + zanik \times (rozproszenie + rozbłysk)$$

Przykład 5.6 przedstawia funkcję Cg obliczającą zanik światła dla podanego położenia powierzchni i struktury `Light` (do struktury dodaliśmy współczynniki `kC`, `kL` i `kQ`).

Przykład 5.6. *Funkcja wewnętrzna C5E6_attenuation*

```
float C5E6_attenuation(float3 position, Light light)
{
  float d = distance(position, light.position);
  return 1 / (light.kC + light.kL * d + light.kQ * d * d);
}
```

Korzystamy z funkcji `distance` znajdującej się w standardowej bibliotece Cg. Oto formalna definicja funkcji `distance`.

`distance(pt1, pt2)`	odległość między punktami `pt1` i `pt2`

Obliczenia zaniku światła musimy dodać do funkcji `C5E5_computeLighting`, ponieważ zanik światła wpływa na współczynniki rozproszenia i rozbłysku źródła światła. Zanik liczymy jako pierwszy. Funkcja wewnętrzna `C5E7_attenuate➥Lighting` z przykładu 5.7 przedstawia wymagane modyfikacje.

Przykład 5.7. *Funkcja wewnętrzna C5E7_attenuateLighting*

```
void C5E5_computeLighting(Light  light,
                          float3 position,
                          float3 normal,
                          float3 eyePosition,
                          float  shininess,

                      out float3 diffuseResult,
                      out float3 specularResult)
{
  // obliczenie zaniku
  float attenuation = computeAttenuation(position, light);
```

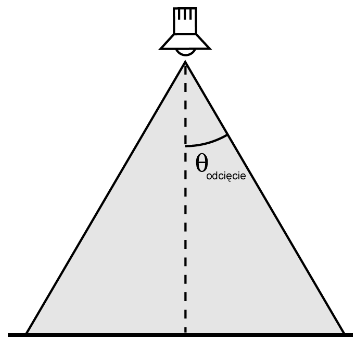
```
    // obliczenie rozproszenia światła
    float3 L = normalize(light.position - position);
    float diffuseLight = max(dot(normal, L), 0);
    diffuseResult = attenuation * light.color * diffuseLight;

    // obliczenie rozbłysku
    float3 V = normalize(eyePosition - position);
    float3 H = normalize(L + V);
    float specularLight = pow(max(dot(normal, H), 0), shininess);
    if (diffuseLight <= 0) specularLight = 0;
    specularResult = attenuation * light.color * specularLight;
}
```

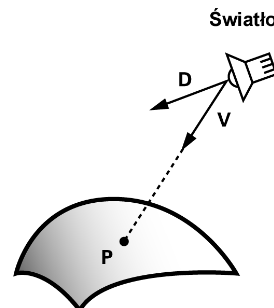
5.5.2. Dodanie efektu reflektora

Innym często stosowanym rozszerzeniem podstawowego modelu jest stosowanie świateł reflektorowych zamiast ogólnych. Kąt odcięcia światła steruje szerokością stożka światła (patrz rysunek 5.18). Oświetlane są tylko te obiekty, które znajdują się wewnątrz stożka.

Aby utworzyć stożek światła, musimy znać położenie i kierunek światła oraz punkt, który zamierzamy oświetlić. Dzięki tym informacjom możemy obliczyć wektory *V* (wektor od światła do wierzchołka) i *D* (kierunek światła), co przedstawia rysunek 5.19.



Rysunek 5.18. *Określanie kąta odcięcia światła*



Rysunek 5.19. *Wektory umożliwiające obliczenie efektu reflektora*

Obliczając iloczyn skalarny dwóch znormalizowanych wektorów, uzyskujemy cosinus kąta między nimi. Używamy go do sprawdzenia, czy punkt *P* znajduje się wewnątrz stożka. Punkt *P* znajduje się w stożku tylko wtedy, gdy `dot(V, D)` jest większy od cosinusa kąta odcięcia światła.

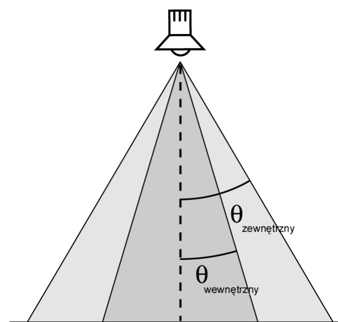
Możemy napisać funkcję sprawdzającą, czy punkt znajduje się wewnątrz stożka (patrz przykład 5.8). Funkcja `C5E8_spotlight` zwraca 1, jeśli *P* znajduje się wewnątrz stożka lub 0 w sytuacji przeciwnej. Do struktury `Light` z przykładu 5.6 dodaliśmy zmienne `direction` (kierunek światła — zakładamy, że ten wektor jest już znormalizowany) i `cosLightAngle` (kosinus kąta odcięcia światła).

Przykład 5.8. *Funkcja wewnętrzna C5E8_spotlight*

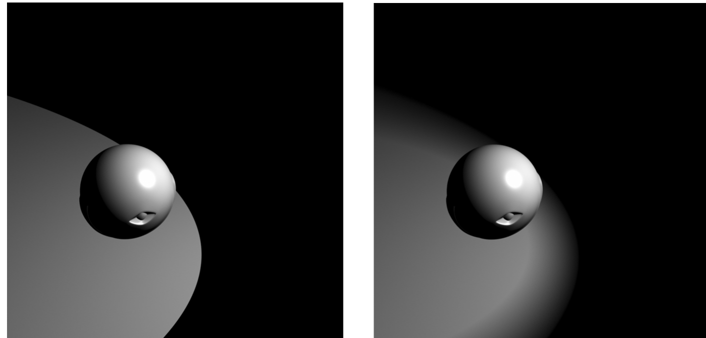
```
float C5E8_spotlight(float3 position,
                     Light light)
{
  float3 V = normalize(position - light.position);
  float cosCone = light.cosLightAngle;
  float cosDirection = dot(V, light.direction);
  if (cosCone <= cosDirection)
    return 1;
  else
    return 0;
}
```

Zmiana intensywności

Wcześniej założyliśmy, że intensywność światła jest taka sama w całym stożku. Bardzo rzadko spotyka się taki równy rozkład światła z reflektora. Aby zwiększyć atrakcyjność efektu, podzielimy stożek na dwie części: stożek wewnętrzny i stożek zewnętrzny. Stożek wewnętrzny zawiera światło o stałej intensywności. Poza nim intensywność zmniejsza się aż do pełnego zaniku na granicy stożka zewnętrznego (patrz rysunek 5.20).Takie podejście pozwoli na utworzenie bardziej interesującego efektu, przedstawionego po prawej stronie na rysunku 5.21.



Rysunek 5.20. *Określanie stożka wewnętrznego i zewnętrznego*



Rysunek 5.21. *Efekt zastosowania stożka wewnętrznego i zewnętrznego*

Bardzo łatwo możemy sprawdzić, czy punkt *P* znajduje się wewnątrz stożka wewnętrznego lub zewnętrznego. Jedyna różnica względem poprzedniego przykładu to zróżnicowanie oświetlenia punktu *P* w zależności od miejsca znajdowania się w stożku.

Jeśli punkt *P* znajduje się w stożku wewnętrznym, otrzymuje pełną intensywność światła. Jeśli znajduje się między stożkami, zmniejszamy intensywność wraz z oddalaniem się od granicy stożka wewnętrznego. W tym przypadku bardzo dobrze sprawdzi się funkcja interpolacji liniowej `lerp`, ale w zaawansowanych profilach można posłużyć się lepszą funkcją.

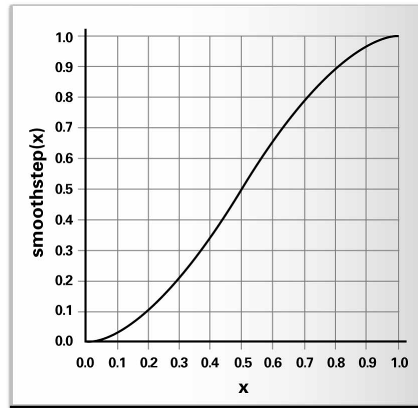
Język Cg udostępnia funkcję `smoothstep`, która tworzy efekt lepszy wizualnie niż proste `lerp`. Niestety funkcja może nie być dostępna w niektórych prostych profilach z powodu ich ograniczonych możliwości. W przykładzie użyjemy funkcji `smoothstep`, choć można ją zastąpić inną funkcją.

Funkcja `smoothstep` dokonuje interpolacji między dwiema wartościami, używając gładkich wielomianów.

`smoothstep(min, max, x)`	zwraca `0`, jeśli `x < min` zwraca `1`, jeśli `x >= max` w przeciwnym razie stosuje gładką interpolację Hermite'a między 1 a 0, stosując wzór: $-2 * ((x - min)/(max - min))^3 + 3 * ((x - min)/(max - min))^2$

Na rysunku 5.22 przedstawiono wykres funkcji `smoothstep`. Używamy tej funkcji, jeśli zależy nam na uzyskaniu ładnie wyglądającego przejścia między wartościami. Inną zaletą funkcji jest to, że zwraca wartości w zakresie [0, 1]. Jeśli poprawnie ustawimy parametry funkcji, zwróci 1.0, gdy *P* znajduje się w stożku wewnętrznym i 0.0, gdy *P* znajduje się w stożku zewnętrznym.

Rysunek 5.22.
Wykres funkcji smoothstep



Jeszcze raz rozszerzamy strukturę `Light`, aby uwzględnić w niej nowe parametry światła. Jeden kąt odcięcia zastępujemy dwoma kątami: jednym dla stożka wewnętrznego i jednym dla stożka zewnętrznego.

Oto końcowa wersja struktury `Light`.

```
struct Light {
  float3 position;
  float3 color;
  float  kC;
  float  kL;
  float  kQ;
  float3 direction;
  float  cosInnerCone;  // Nowy element
  float  cosOuterCone;  // Nowy element
};
```

Funkcja wewnętrzna `C5E9_dualConeSpotlight` przedstawiona w przykładzie 5.9 wykonuje obliczenia niezbędne w oświetleniu reflektorowym.

Przykład 5.9. *Funkcja wewnętrzna C5E9_dualConeSpotlight*

```
float C5E9_dualConeSpotlight(float3 position, Light light)
{
  float3 V = normalize(position - light.position);
  float cosOuterCone = light.cosOuterCone;
  float cosInnerCone = light.cosInnerCone;
  float cosDirection = dot(V, light.direction);
  return smoothstep(cosOuterCone, cosInnerCone, cosDirection);
}
```

Funkcja wewnętrzna `C5E10_spotAttenLighting` łączy w sobie elementy zaniku i światła reflektorowego ze współczynnikami rozproszenia i rozbłysku (patrz przykład 5.10).

Przykład 5.10. *Funkcja wewnętrzna C5E10_spotAttenLighting*

```
void C5E10_spotAttenLighting(Light  light,
                             float3 position,
                             float3 normal,
                             float3 eyePosition,
                             float  shininess,

                         out float3 diffuseResult,
                         out float3 specularResult)
{
  // obliczenie zaniku
  float attenuation = computeAttenuation(position, light);

  // obliczenie efektu reflektora
  float spotEffect = spotlight(position, light);

  // obliczenie rozproszenia światła
  float3 L = normalize(light.position - position);
  float diffuseLight = max(dot(normal, L), 0);
  diffuseResult = attenuation * spotEffect * light.color * diffuseLight;

  // obliczenie rozbłysku
  float3 V = normalize(eyePosition - position);
  float3 H = normalize(L + V);
  float specularLight = pow(max(dot(normal, H), 0), shininess);
  if (diffuseLight <= 0) specularLight = 0;
  specularResult = attenuation * spotEffect * light.color * specularLight;
}
```

5.5.3. Światła kierunkowe

Choć obliczenia takich efektów jak zanik światła lub światła reflektorowe zwiększają wizualną złożoność sceny, wyniki nie zawsze są zauważalne. Rozważmy na przykład promienie słoneczne oświetlające obiekty na ziemi. Wszystkie promienie wydają się pochodzić z jednego kierunku, ponieważ słońce znajduje się bardzo daleko. W takim przypadku dodawanie zaniku lub efektu reflektorowego nie ma sensu, gdyż wszystkie obiekty otrzymują podobną ilość światła. Światło o takich właściwościach nazywane jest *światłem kierunkowym*. Światła kierunkowe nie występują w rzeczywistym świecie, istnieją tylko w wirtualnej rzeczywistości, ale warto identyfikować sytuacje, w których tego rodzaju oświetlenie jest wystarczające. W ten sposób można uniknąć wykonywania niepotrzebnych obliczeń.

5.6. Ćwiczenia

1. **Odpowiedz na pytanie**. Jakie są różnice w wynikowym renderingu oświetlenia wykorzystującego wierzchołki a wykorzystującego fragmenty? Należy skupić się na efektach rozbłysku i reflektorów.

2. **Wypróbuj**. Zmodyfikuj funkcję `C5E5_computeLighting` w taki sposób, by zakładała światła kierunkowe w sposób opisany w podrozdziale 5.5.3. Można zwiększyć wydajność funkcji, usuwając różnicę wektorów a tym samym normalizację.
3. **Wypróbuj**. Przykłady w tym rozdziale opierają się na założeniu, że aplikacja dostarcza informacji o położeniu światła w przestrzeni obiektu. Można je także określić w przestrzeni oka. Przestrzeń oka jest wygodniejsza dla aplikacji, ponieważ nie wymaga przekształcania położenia światła z przestrzeni świata lub oka do przestrzeni obiektu dla każdego przetwarzanego obiektu. Przestrzeń oka upraszcza obliczanie współczynnika rozbłysku, ponieważ oko jest wtedy początkiem układu współrzędnych. Z drugiej strony oświetlanie w przestrzeni oka wymaga przekształcania położenia i normalnej z przestrzeni obiektu do przestrzeni oka dla każdego wierzchołka. Oznacza to konieczność wymnożenia tych wartości przez macierz model-widok i transponowaną odwrotność macierzy model-widok. Jeśli macierz model-widok skaluje wektory, trzeba znormalizować otrzymane wartości, ponieważ po wymnożeniu mogą być zdenormalizowane. Zmodyfikuj funkcję `C5E5_computeLighting`, zakładając określenie położenia `P` i normalnej `N` w przestrzeni oka. Zmień także funkcję `C5E4_twoLights`, aby przekształcała położenie i normalną z przestrzeni obiektu do przestrzeni oka przed przekazaniem wartości do nowej funkcji `C5E5_computeLighting`.
4. **Wypróbuj**. Zmodyfikuj wersję funkcji `C5E5_computeLighting` dla przestrzeni oka z ćwiczenia 3., aby zakładała, że wektor `V` w przestrzeni oka zawsze ma kierunek (0, 0, 1). Rozwiązanie to nazywane jest optymalizacją rozbłysku dla *nieskończonego widoku* i eliminuje potrzebę normalizacji `V` przed obliczaniem `H`. W jaki sposób ta zmiana wpłynęła na rendering? Czy taka optymalizacja jest możliwa dla oświetlenia w przestrzeni obiektu?
5. **Odpowiedz na pytanie**. Które rozwiązanie jest bardziej wydajne: oświetlenie w przestrzeni obiektu lub w przestrzeni oka. Które jest bardziej wygodne dla programisty aplikacji?
6. **Wypróbuj**. Napisz parę programów Cg (wierzchołków i fragmentów), które mieszają obliczenia oświetlenia dla wierzchołków i fragmentów. Współczynniki emisji, otoczenia, rozproszenia obliczamy w programie wierzchołków. Wektory połowy kąta i normalnej obliczamy w programie wierzchołków i przekazujemy do programu fragmentów. W programie fragmentów obliczamy tylko współczynnik rozbłysku dla interpolowanych wektorów normalnej i połowy kąta. Taki podział obliczeń umożliwia wykonanie większości zadań dla wierzchołków, ale najbardziej narażone na błędy rozbłyski są liczone dokładniej (dla fragmentów). Porównaj

jakość i wydajność tego rozwiązania w porównaniu z implementacjami stosującymi tylko obliczenia dla wierzchołków lub tylko dla fragmentów.

7. **Wypróbuj**. Niektóre materiały, jak włosy, płyty winylowe, atłas i malowany metal odbijają światło inaczej niż pozostałe materiały z powodu specyficznej mikrostruktury powierzchni. Zapoznaj się z materiałami dotyczącymi oświetlanie anizotropowego i zaimplementuj je w programie wierzchołków lub fragmentów.