OpenGL. Księga eksperta. Wydanie III

Autorzy: Richard S. Wright Jr., Benjamin Lipchak
Tłumaczenie: Wojciech Moch (wstęp, rozdz. 1 – 9), Rafał Jońca (rozdz. 10 –14), Marek Pętlicki (rozdz. 15 – 23, dod. A – C)
ISBN: 83-7361-703-5
Tytuł oryginału: OpenGL Superbible
Format: B5, stron: 1080

Poznaj tajniki programowania realistycznej grafiki 3D

- Stwórz i animuj obiekty trójwymiarowe
- Wykorzystaj mechanizmy renderingu na platformach Windows, Mac OS i Linux
- Zastosuj w swoich projektach algorytmy sprzętowego wspomagania wyświetlania grafiki 3D

OpenGL to platforma programistyczna do tworzenia grafiki trójwymiarowej. Tworzące ją mechanizmy zostały opracowane ponad 20 lat temu w firmie Silicon Graphics i spowodowały prawdziwą rewolucję w świecie grafiki komputerowej. Dzięki bibliotece OpenGL możliwe stało się tworzenie realistycznej grafiki i animacji 3D oraz wykorzystywanie możliwości kart i systemów graficznych do wspomagania jej wyświetlania. To właśnie dzięki rozwojowi OpenGL możemy teraz podziwiać wspaniałe wizualizacje, kinowe efekty specjalne i oszałamiające gry komputerowe.

Książka „OpenGL. Księga eksperta. Wydanie III" to podręcznik dla programistów chcących wykorzystać możliwości biblioteki OpenGL w swoich projektach. Opisuje zarówno podstawy programowania grafiki z wykorzystaniem OpenGL, jak i możliwości zastosowania jej na różnych platformach systemowych i sprzętowych. Przedstawia również sposoby wykorzystania mechanizmów sprzętowego wspomagania wyświetlania grafiki i animacji 3D oraz efektów specjalnych.

- Typowe zastosowania grafiki komputerowej
- Przegląd możliwości bibliotek OpenGL
- Podstawy stosowania OpenGL
- Tworzenie brył trójwymiarowych i korzystanie z kształtów podstawowych
- Przekształcenia geometryczne i przetwarzanie potokowe
- Modele kolorów i oświetlenia
- Przetwarzanie grafiki i mapowanie tekstur
- Modelowanie krzywych i powierzchni
- Wykorzystanie bibliotek OpenGL w różnych systemach operacyjnych
- Buforowanie danych
- Cienie i głębia ostrości
- Operacje na wierzchołkach

Jeśli chcesz stworzyć grafikę 3D zapierającą dech w piersiach, skorzystaj z wiedzy zawartej w tej książce.

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

Spis treści

Rozdział 5.
Kolor, materiały i oświetlenie. Podstawy

Autor: Richard S. Wright Jr

Czego nauczymy się w tym rozdziale?

Jak	Funkcje, z których będziemy korzystać
Ustalać kolor na podstawie składowych RGB	`glColor`
Ustalać model cieniowania	`glShadeModel`
Ustalać model oświetlenia	`glLightModel`
Ustalać parametry oświetlenia	`glLight`
Skonfigurować odblaskowe właściwości materiałów	`glColorMaterial/glMaterial`
Stosować normalne powierzchni	`glNormal`

W niniejszym rozdziale grafika trójwymiarowa w końcu zacznie wyglądać interesująco (chyba że ktoś bardzo lubi modele szkieletowe), a w każdym następnym będzie coraz lepiej. Do tej pory uczyliśmy się biblioteki OpenGL od jej podstaw — jak można budować programy, jak w przestrzeni trójwymiarowej składać obiekty z obiektów podstawowych, jak manipulować tymi obiektami w trzech wymiarach. Cały czas kładliśmy tylko fundamenty, ale nadal nie wiemy, jak będzie wyglądał cały budynek! Parafrazując frazę: „Gdzie jest żarcie?".

Krótko mówiąc — żarcie zaczyna się tutaj. W większości pozostałej części tej książki nauka zostanie odsunięta na plan dalszy, a władzę przejmie magia. Zgodnie ze słowami Arthura C. Clarke'a „każda wystarczająco rozwinięta technologia jest nie do odróżnienia od magii". Oczywiście w kolorach i oświetleniu nie ma żadnej magii, choć czasami można odnieść inne wrażenie. Jeżeli ktokolwiek jest zainteresowany tą „wystarczająco rozwiniętą technologią", czyli matematyką, to odsyłamy do dodatku A.

Alternatywną nazwą tego rozdziału mogłoby być „Dodawane do sceny realizmu". W świecie rzeczywistym na kolor obiektu wpływa dużo więcej czynników niż proste nałożenie koloru znane z biblioteki OpenGL. Obiekty, poza tym, że mają jakiś kolor, mogą być matowe lub błyszczące, a nawet świecić własnym światłem. Widziany kolor obiektu może

być inny w jasnym oświetleniu niż w świetle przytłumionym. Znaczenie ma też kolor padającego na obiekt światła. Oświetlony obiekt może mieć też cieniowane powierzchnie, jeżeli oświetlany będzie pod odpowiednim kątem.

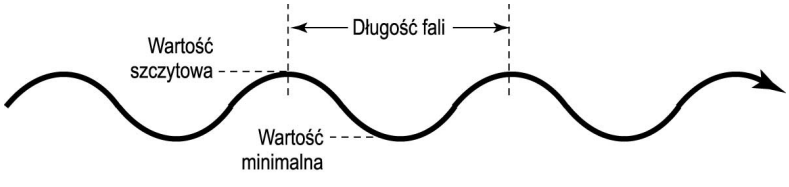
Czym jest kolor?

Pomówmy przez chwilę o samych kolorach. W jaki sposób widzimy kolory? Jak są one tworzone w naturze? Teoria kolorów i metody odbierania kolorów przez ludzkie oko bardzo ułatwią nam zrozumienie programowych metod tworzenia barw. Jeżeli ktoś już doskonale zna tajniki teorii kolorów, spokojnie może pominąć ten podrozdział.

Światło jest falą

Kolor jest po prostu falą świetlną o pewnej długością, widoczną dla ludzkiego oka. Każdy, kto w szkole uważał na lekcjach fizyki, zapewne pamięta, że światło jest jednocześnie zarówno falą, jak i cząsteczką. Światło modelowane jest jako fala przemieszczająca się przez przestrzeń, jak fale na wodzie, ale także jako cząsteczki, padające na ziemie jak krople deszczu. Jeżeli komuś ta koncepcja wydaje się pogmatwana, to chyba już rozumie, dlaczego tak niewiele osób studiuje fizykę kwantową.

Światło, jakie widzimy, tak naprawdę jest mieszanką wielu różnych rodzajów światła. Każdy rodzaj światła identyfikowany jest długością fali. Długość fali światła mierzona jest odległością pomiędzy dwiema kolejnymi wartościami szczytowymi, tak jak pokazano to na rysunku 5.1.

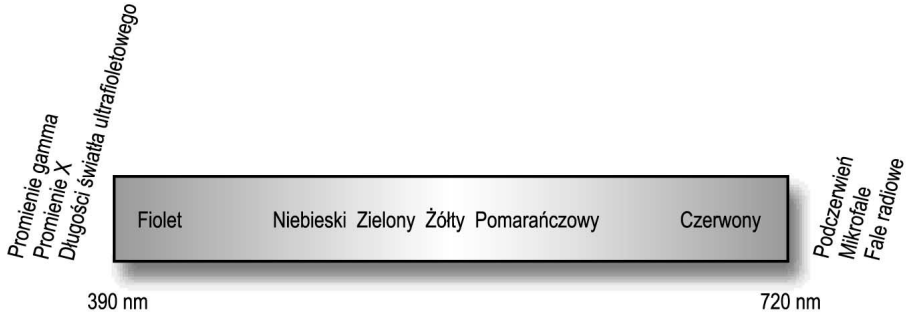


Rysunek 5.1. *Pomiar długości fali świetlnej*

Światło widzialne mieści się w zakresie długości fal od 390 nanometrów (miliardowych części metra) dla światła fioletowego, do 720 nanometrów dla światła czerwonego. Ten zakres nazywany jest często *spektrum światła widzialnego*. Z całą pewnością każdy słyszał już określenia *ultrafiolet* i *podczerwień* — opisują one światło niewidzialne dla ludzkiego oka, leżące tuż poza podanym powyżej spektrum. Całe spektrum widzialne zawiera w sobie wszystkie kolory tęczy (tak jak na rysunku 5.2).

Światło jako cząsteczka

Można sobie pomyśleć: „Dobra, panie mądrala, jeżeli kolor jest tylko długością fali światła, a wszystkie możliwe kolory widoczne są w tęczy, to gdzie w takim razie podział się brąz z czekolady, czerń mojej porannej kawy albo biel kartki papieru?" Odpowiedź na te pytania zaczniemy od stwierdzenia, że czerń nie jest kolorem, podobnie jak i biel. Tak



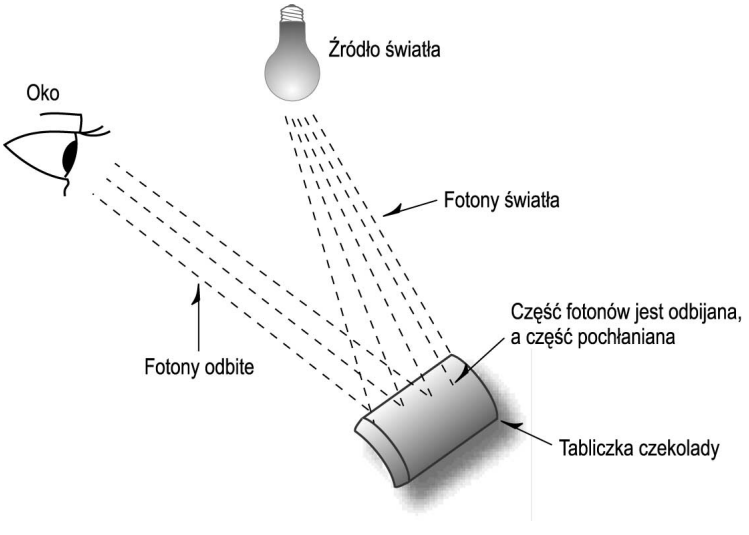
Rysunek 5.2. *Spektrum światłą widzialnego*

naprawdę czerń jest barkiem jakiegokolwiek koloru, a biel to równomierna kombinacja wszystkich kolorów. Oznacza to, że biały obiekt odbija wszystkie kolory w jednakowy sposób, natomiast obiekt czarny tak samo równo pochłania wszystkie długości fali świetlnej.

Jeżeli chodzi o brąz czekolady, jak również inne kolory, z jakimi spotykamy się w życiu, to rzeczywiście jest on kolorem. Kolory rzeczywiste składają się one z różnych mieszanek „czystych" kolorów widzialnego spektrum, w związku z czym pod względem fizycznym są one kolorami złożonymi. Żeby zrozumieć, jak działa takie mieszanie kolorów, trzeba myśleć o świetle jako o cząsteczkach. Każdy oświetlony obiekt bombardowany jest miliardami, miliardami (tutaj muszę przeprosić Carla Sagana) fotonów, czyli maleńkich cząsteczek światła. Przypomnijmy sobie ten fizyczny hokus-pokus, o którym mówiliśmy wcześniej — każdy foton jest również falą posiadającą pewną długość, czyli konkretny kolor ze spektrum.

Każdy obiekt fizyczny składa się z atomów. Sposób odbicia fotonów od obiektu uzależniony jest od rodzaju atomów, z jakich się on składa, liczby każdego rodzaju atomów, a także ich ułożenia (razem z elektronami) wewnątrz obiektu. Część fotonów jest odbijana, a część pochłaniana (pochłaniane fotony są zamieniane w ciepło). Każdy materiał lub mieszanka materiałów lepiej odbija pewne długości fal niż inne. Zasada ta została przedstawiona na rysunku 5.3.

Rysunek 5.3.
Każdy obiekt odbija część fotonów, a część z nich pochłania



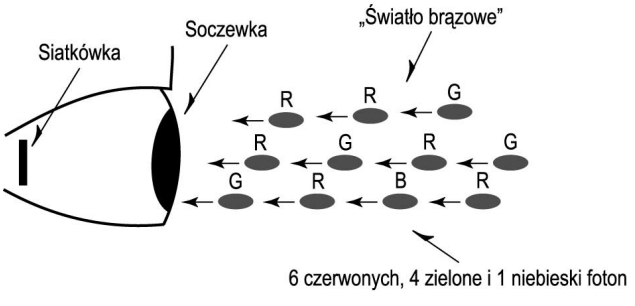
Nasz osobisty wykrywacz fotonów

Światło odbite od naszej tabliczki czekolady wpada do naszego oka, gdzie jest interpretowane jako kolor. Miliardy fotonów wpadających do naszego oka skupiane są w jego czarnej części, gdzie siatkówka działa na zasadzie podobnej do kliszy fotograficznej. Miliony komórek siatkówki pobudzanych jest przez uderzające w nie fotony, co powoduje, że do mózgu zaczynają wędrować sygnały nerwowe, które są następnie interpretowane jako światło i kolor. Im więcej fotonów uderza w siatkówkę, tym bardziej pobudzone są jej komórki. Poziom pobudzenia interpretowany jest w mózgu jako jasność światła. Wynika to z prostej zależności — im jaśniejsze jest światło, tym więcej fotonów wpada do naszego oka.

W naszym oku znajdują się trzy rodzaje komórek. Wszystkie reagują na uderzenia fotonów, ale każdy rodzaj najmocniej reaguje na fale o określonej długości. Jedne komórki pobudzane są światłem czerwonym, drugie światłem zielonym, a trzecie światłem niebieskim. Tak więc światło składające się głównie z czerwonych długości fal najbardziej pobudza w siatkówce komórki reagujące na światło czerwone, a mniej pozostałe. Nasz mózg przetwarza takie sygnały, stwierdzając, że widziane światło jest przede wszystkim czerwone. A teraz prosta matematyka — łączenie różnych długości fal świetlnych o odmiennych intensywnościach da nam w wyniku różne mieszanki kolorów. Jeżeli wszystkie fale będą miały identyczną intensywność, to takie światło będziemy odbierać jako białe. Jeżeli natomiast nie zobaczymy żadnych fal świetlnych, to zinterpretujemy ten stan jako czerń.

Jak widać, każdy odbierany przez nasze oczy „kolor" jest w rzeczywistości mieszaniną światła z całego widzialnego spektrum. „Sprzętowe" elementy naszego oka wykrywają obrazy na podstawie względnego skupienia i natężenia światła czerwonego, zielonego i niebieskiego. Na rysunku 5.4 przedstawiono sposób, w jaki nasze oko odbiera kolor brązowy, czyli mieszankę 60% fotonów czerwonych, 40% fotonów zielonych i 10% niebieskich.

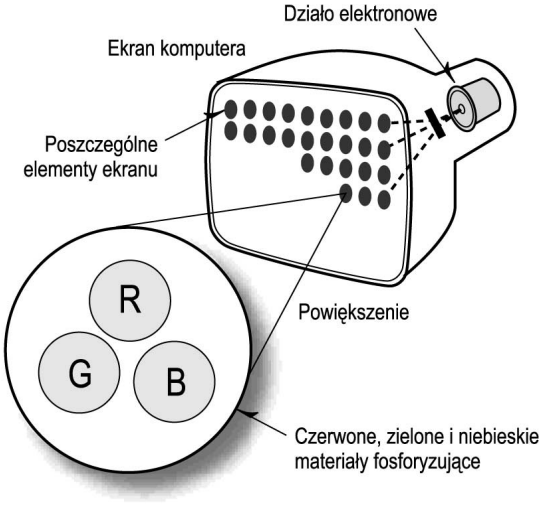
Rysunek 5.4.
Jak nasze oko widzi „kolor" brązowy



Komputer generatorem fotonów

Skoro wiemy już, jak ludzkie oko rozpoznaje kolory, całkiem rozsądnym wydaje się pomysł komputerowego generowania kolorów poprzez definiowanie oddzielnych intensywności składowych czerwonej, zielonej i niebieskiej. Tak się akurat składa, że monitory komputerów zaprojektowane są w sposób pozwalający im emitować trzy rodzaje światła (zgadnijcie jakie), z których każdy może mieć inną intensywność. W tylnej części

monitora znajduje się działo elektronowe wyrzucające elektrony na tylną część ekranu monitora. Na powierzchni tego ekranu znajdują się materiały fosforyzujące, które w czasie bombardowania strumieniem elektronów zaczynają emitować światło czerwone, zielone lub niebieskie. (No dobra, ale jak w takim razie działają wyświetlacze LCD? Znalezienie odpowiedzi na to pytanie pozostawimy Czytelnikowi jako pracę domową). Te trzy fosforyzujące materiały ułożone są bardzo blisko siebie, tak że razem tworzą jedną barwną plamkę na ekranie (rysunek 5.5).



Rysunek 5.5. *Jak monitor komputerowy generuje kolory*

Proszę sobie przypomnieć, jak w rozdziale 2., „Używanie OpenGL", opisywaliśmy polecenie `glColor` pozwalające na definiowanie kolorów poprzez podanie intensywności składowych czerwonej, zielonej i niebieskiej.

Sprzęt komputerów PC

Był taki czas, kiedy najnowocześniejszą kartą graficzną dla komputerów PC była karta Hercules. Karta pozwalała na tworzenie bitmapowych obrazów o rozdzielczości 720×348 punktów. Jej wadą było to, że każdy z punktów mógł mieć tylko dwa stany: zapalony lub zgaszony. W tym czasie tworzenie bitmapowej grafiki na komputerach PC nie było zadaniem łatwym, choć tworzone były wspaniałe monochromatyczne obrazy — nawet trójwymiarowe!

Jeszcze przed powstaniem kart Hercules dostępne były karty *CGA* (ang. *Color Graphics Adapter*). Karta powstała razem z pierwszymi komputerami IBM PC mogła obsługiwać rozdzielczość 320×200 pikseli i wyświetlać na ekranie 4 lub 16 kolorów naraz. Dostępna była też wyższa rozdzielczość (640×200) z dwoma kolorami, ale nie była ona tak efektowna, jak w znacznie tańszej karcie Hercules (monitory kolorowe oznaczały znacznie wyższe koszty). Jak na dzisiejsze standardy karta CGA wygląda mizernie. Jej graficzne możliwości prezentowały się nie najlepiej nawet w porównaniu z możliwościami tanich komputerów Commodore 64 i Atari. Ze względu na brak wystarczających rozdzielczości

obrazu karta nie nadawała się do tworzenia grafiki biznesowej ani nawet modelowania. Najczęściej znajdowała zastosowanie w prostych grach i aplikacjach biznesowych, które mogły najwięcej skorzystać na możliwości wyświetlania kolorowego tekstu. Mówiąc ogólnie, trudno było znaleźć dobry powód, żeby kupić tę nietanią wtedy kartę.

Następna rewolucja w grafice komputerów PC nastąpiła, gdy firma IBM zaprezentowała kartę *EGA* (ang. *Enhanced Graphic Adapter*). Udostępniała ona dodatkowe tryby tekstowe, w których możliwe było wyświetlanie większej liczby niż 25 linii kolorowego tekstu, a w trybie graficznym można było uzyskać rozdzielczość 640×350 pikseli w 16 kolorach! Inne poprawki techniczne wyeliminowały problemy z migotaniem obrazu w karcie CGA i umożliwiły tworzenie lepszej i płynniejszej animacji. Od tego momentu gry platformowe, graficzne aplikacje biznesowe, a nawet proste grafiki trójwymiarowe stały się czymś normalnym w świecie komputerów PC. Był to ogromny krok w stosunku do kart CGA, ale grafika w komputerach PC nadal pozostawała w powijakach.

Ostatnim ważnym standardem, jaki wśród komputerów PC wyznaczyła firma IBM była karta *VGA* (skrót ten oznaczał *Vector Graphics Array*, a nie, jak się powszechnie uważa, Video Graphics Adapter). Karta była o wiele szybsza niż EGA, pozwalała stosować 16 kolorów w wysokich rozdzielczościach (640×480), a w mniejszych (320×200) — nawet 265 kolorów, które były wybierane z palety obejmującej ponad 16 milionów kolorów. W ten sposób otwarty został worek z graficznymi aplikacjami dla komputerów PC — nagle możliwe stało się tworzenie niemal fotorealistycznych obrazów, a na rynku pojawiły się programy do śledzenia promieni (*ray trace*), gry trójwymiarowe, oprogramowanie do obróbki zdjęć i inne.

Firma IBM posiadała jeszcze inną, znacznie bardziej zaawansowaną kartę graficzną dla swoich „stacji roboczych" — 8514. Mogła ona wyświetlać obraz o rozdzielczości 1024×768 w 256 kolorach. Według firmy IBM ta karta miała być używana wyłącznie w aplikacjach CAD i naukowych, jednak klienci zawsze będą chcieli czegoś więcej. Była to krótkowzroczność, która firmę IBM kosztowała pozycję ustanawiającego standardy rynku graficznego komputerów PC. Inni dostawcy zaczęli tworzyć karty Super-VGA, które mogły wyświetlać obraz w rozdzielczości 1024×768 i większej w coraz szerszej palecie kolorów. Na początku było to 800×600, później 1024×768 i więcej, początkowo w 256, a później w 32 000 i 65 000 kolorów. Dzisiejsze karty graficzne wyświetlają obrazy o rozdzielczości 1024×768 i większej, stosując 24-bitową paletę kolorów. Takie karty graficzne dostępne są już nawet w najprostszych konfiguracjach komputerów.

To wszystko daje nam naprawdę ogromne możliwości — na przykład tworzenie fotorealistycznej grafiki trójwymiarowej, żeby wymienić tylko jedno zastosowanie. Gdy Microsoft przeniósł bibliotekę OpenGL na system Windows, umożliwił w ten sposób tworzenie wysokiej jakości aplikacji graficznych dla komputerów PC. Jeżeli połączymy wydajność dzisiejszych procesorów z możliwościami kart graficznych wyposażonych w akceleratory grafiki trójwymiarowej, to uzyskamy możliwości, jakie jeszcze kilka lat temu dostępne były wyłącznie na stacjach roboczych wartych setki tysięcy dolarów. A to wszystko za promocyjną cenę komputera w supermarkecie. Dzisiejsze domowe komputery mogą tworzyć zaawansowane symulacje naukowe, gry i dużo, dużo więcej. Już dzisiaj termin *rzeczywistość wirtualna* wydaje się przestarzały jak rakieta Bucka Rogersa, a grafika trójwymiarowa wydaje się nam czymś absolutnie oczywistym.

Tryby graficzne komputerów PC

Systemy operacyjne Microsoft Windows i Apple Macintosh pod dwoma względami zrewolucjonizowały świat grafiki komputerów PC. Po pierwsze, utworzyły najważniejsze graficzne środowiska systemów operacyjnych wykorzystywanych w biznesie, a wkrótce potem również na rynku konsumenckim. Po drugie, znacząco ułatwiły pracę programistów grafiki. Sprzęt graficzny poddany został „wirtualizacji" na poziomie sterowników. Programiści nie muszą już wysyłać poleceń rysujących bezpośrednio do kart graficznych. Korzystają ze specjalnych interfejsów programistycznych (takich jak OpenGL), a sprawami komunikacji ze sprzętem zajmuje się już system operacyjny.

Rozdzielczość ekranu

W dzisiejszych komputerach stosowane są rozdzielczości ekranów od 640×480 do 1600×1200, a nawet więcej. Najniższe rozdzielczości, takie jak 640×480 uznawane są za odpowiednie dla pewnych zadań graficznych, a osoby z problemami ze wzrokiem stosują te rozdzielczości (często w połączeniu z dużymi monitorami), co ułatwia im pracę z komputerem. Zawsze trzeba mieć na uwadze wielkość okna i ustawienia widoku (rozdział 2.), w którym wyświetlana będzie przestrzeń ograniczająca. Zrównując wielkość rysunku z wielkością okna, bardzo łatwo można dostosowywać się do różnych kombinacji wielkości okna i rozdzielczości ekranu. Dobrze napisane aplikacje wyświetlają mniej więcej taki sam obraz niezależnie od rozdzielczości ekranu. Oczywiście im większa będzie rozdzielczość, tym dokładniejszy obraz powinien zobaczyć użytkownik.

Głębia kolorów

Jeżeli powiększenie rozdzielczości ekranu, czyli liczby dostępnych do rysowania pikseli wpływa na poprawę dokładności i ostrości obrazu, to w podobny sposób zwiększenie liczby dostępnych kolorów powinno poprawiać przejrzystość generowanego obrazu. Obraz wyświetlany przez komputer, który może posługiwać się milionami kolorów, powinien wyglądać o wiele lepiej niż ten sam obraz wyświetlany za pomocą zaledwie 16 kolorów. Pod względem programowym zajmować będziemy się zaledwie trzema głębiami kolorów: 4-bitową, 8-bitową i 24-bitową.

Tryb koloru 4-bitowego

W najgorszym przypadku nasz program będzie musiał działać w zaledwie 16 kolorach — ten tryb nazywany jest 4-bitowym, ponieważ kolor każdego piksela reprezentowany jest za pomocą czterech bitów. Można w nich zapisać wartości od 0 do 15, które będą indeksem w 16-elementowym zbiorze predefiniowanych kolorów (taki ograniczony zbiór kolorów, do których można odwoływać się za pomocą indeksów, nazywany jest *paletą kolorów*). Posiadając zaledwie 16 kolorów, nie mamy zbyt wielkich możliwości tworzenia dokładnego i ostrego obrazu. Najczęściej zupełnie bezpiecznie można całkowicie zignorować tryb 4-bitowy w naszych programach, tym bardziej, że większość najnowszych kart graficznych nie obsługuje już żadnego trybu 4-bitowego koloru.

Tryb koloru 8-bitowego

Tryb koloru 8-bitowego pozwala wyświetlać na ekranie 256 kolorów. Nadal jest to dość poważne ograniczenie, choć poprawa w stosunku do trybu 4-bitowego jest znacząca. Większość akceleratorów graficznych współpracujących z biblioteką OpenGL nie przyspiesza działań na 8-bitowych kolorach, ale stosując rendering programowy można w pewnych warunkach uzyskać satysfakcjonujące wyniki. Największym problemem jest tutaj zbudowanie prawidłowej palety kolorów. Ten temat będziemy krótko omawiać w rozdziale 13., „Wiggle. OpenGL w systemie Windows".

Tryb koloru 24-bitowego

Dzisiaj możliwie najlepszy obraz uzyskiwany jest w trybie koloru 24-bitowego. W tym trybie kolor każdego piksela definiowany jest za pomocą pełnych 24 bitów, w których każde 8 bitów opisuje intensywność składowej czerwonej, zielonej i niebieskiej (8+8+8 = 24). Na ekranie możemy umieścić piksel w dowolnym z ponad 16 milionów kolorów. Dość oczywistą wadą tego trybu jest ilość pamięci, jaka jest potrzebna do zapisania wyglądu ekranów o większych rozdzielczościach (dla rozdzielczości 1024×768 będzie to nieco ponad 2 MB). Poza tym przenoszenie dużych wycinków pamięci również jest wolną operacją, szczególnie jeżeli pracujemy nad szybką animacją. Na całe szczęście dzisiejsze akceleratory graficzne optymalizowane są do działania w tych właśnie trybach kolorów i dostarczane są z wystarczającą ilością wbudowanej pamięci, żeby sprostać rosnącemu zapotrzebowaniu na pamięć.

Tryby koloru 16- i 32-bitowego

Ze względu na oszczędność pamięci i poprawę wydajności w wielu kartach graficznych stosowane są również różne inne tryby kolorów. W celu poprawy wydajności często używany jest tryb 32-bitowy, który czasami nazywany jest trybem *true color* (prawdziwy kolor). Tak naprawdę w tym trybie nie jest wyświetlanych więcej kolorów niż w trybie 24-bitowym, a poprawa wydajności uzyskiwana jest poprzez ustawianie danych każdego piksela do granicy adresów 32-bitowych. Niestety, w związku z tym marnowanych jest osiem bitów (jeden bajt) dla każdego piksela. W dzisiejszych 32-bitowych komputerach PC budowanych z wykorzystaniem procesorów firmy Intel stosowanie adresów pamięci podzielnych przez 32 pozwala na uzyskanie znacznie lepszych wyników dostępu do pamięci. Nowoczesne akceleratory graficzne również stosują tryb 32-bitowy, w którym pierwsze 24 bity wykorzystane są na zapisanie kolorów RGB, a końcowe 8 bitów stosowane jest do przechowywania wartości kanału alfa. Na temat kanału alfa będziemy mówić więcej w następnym rozdziale.

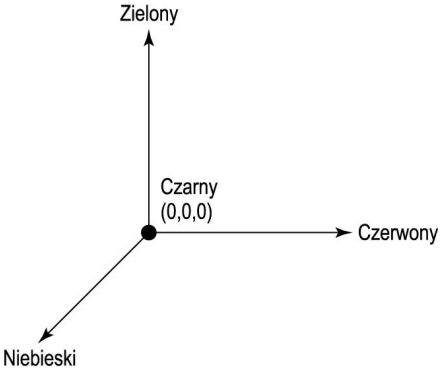
Innym popularnym trybem jest tryb 16-bitowy, który czasami obsługiwany jest w celu poprawienia efektywności wykorzystania pamięci. Pozwala on nadać każdemu pikselowi jeden z 65 536 kolorów. Ten tryb jest niemal tak samo skuteczny jeżeli chodzi o odwzorowywanie obrazów fotograficznych jak tryb 24-bitowy. Praktycznie trudno odróżnić od siebie obraz wyświetlany w obu tych trybach. To właśnie poprawa wykorzystania pamięci i wzrost prędkości wyświetlania obrazów spowodowały, że tryb ten był bardzo popularny w pierwszych akceleratorach grafiki trójwymiarowej przeznaczonych do gier. Jednak dodatkowe kolory w trybie 24-bitowym bardzo podnoszą jakość obrazu, jeżeli do jego generowania wykorzystywane są operacje cieniowania i mieszania kolorów.

Używanie kolorów w OpenGL

Wiemy już, że biblioteka OpenGL definiuje kolory, określając intensywności składowych: czerwonej, zielonej i niebieskiej. Wiemy też, że nowoczesny sprzęt jest w stanie wyświetlić niemal wszystkie możliwe kombinacje tych składowych albo zaledwie kilka. Jak w takim razie możemy określić potrzebny nam kolor, podając wspomniane trzy składowe?

Sześcian kolorów

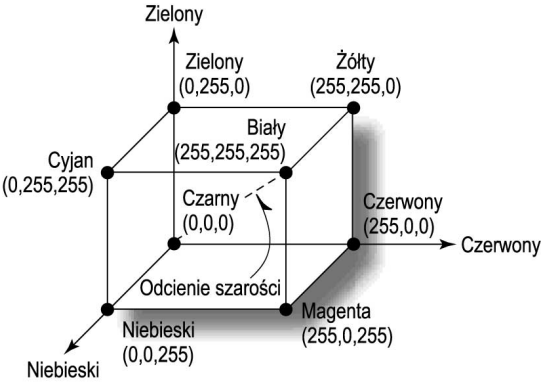
Każdy kolor możemy zdefiniować trzema dodatnimi wartościami, dlatego wszystkie dostępne kolory można przedstawić jako pewną przestrzeń nazywaną *przestrzenią kolorów RGB*. Na rysunku 5.6 przedstawiono, jak wygląda taka przestrzeń kolorów w początku układu współrzędnych z osiami kolorów czerwonego, zielonego i niebieskiego. Współrzędne poszczególnych kolorów określane są na tej samej zasadzie jak współrzędne *x*, *y* i *z*. W początku układu współrzędnych intensywność każdej składowej jest równa zero, co oznacza kolor czarny. W komputerach PC wszystkie informacje o kolorze musza zmieścić się w 24 bitach, co daje 8 bitów na każdą składową. Można więc powiedzieć, że w każdej składowej wartość 255 oznacza pełne nasycenie tej składowej koloru. Oznacza to, że nasz sześcian będzie miał boki o długości 255 jednostek. Dokładnie na przeciwko rogu, w którym panuje czerń (0,0,0), znajduje się róg biały o współrzędnych (255, 255, 255). Wzdłuż każdej z osi, przy wartości 255 znajdują się pełne nasycenia każdego koloru składowego: czerwonego, zielonego i niebieskiego.



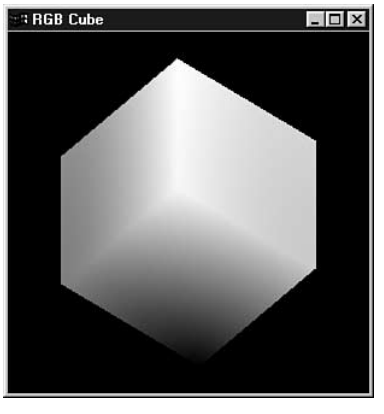
Rysunek 5.6. *Początek układu współrzędnych koloru RGB*

Przedstawiony na rysunku 5.7 „sześcian kolorów”, na swojej powierzchni jak i w swoim wnętrzu zawiera wszystkie możliwe kolory. Na przykład wszystkie możliwe odcienie szarości leżą wewnątrz sześcianu, na linii ukośnej łączącej punkty (0, 0, 0) i (255, 255, 255).

Rysunek 5.8 przedstawia gładko cieniowany sześcian kolorów rysowany w przykładowym programie *CCUBE*. Na powierzchni sześcianu zobaczyć można przejścia kolorów od czerni w jednym rogu, do bieli w przeciwległym. W trzech rogach, odległych od czarnego o 255 jednostek, można zobaczyć kolory czerwony, zielony i niebieski w ich pełnych intensywnościach. Dodatkowo w trzech rogach zobaczyć można kolory żółty, zielono-niebieski (cyan) i fuksyna (magenta), będące kombinacjami kolorów podstawowych. Sześcian kolorów można obracać, aby zobaczyć wszystkie jego strony.



Rysunek 5.7. *Przestrzeń kolorów RGB*



Rysunek 5.8. *Wynik działania programu CCUBE*

Ustalanie koloru rysowania

Przyjrzyjmy się krótko funkcji `glColor`. Jest ona prototypowana następująco:

```
void glColor<x><t>(red, green, blue, alpha);
```

W nazwie funkcji znak `<x>` oznacza liczbę parametrów. Mogą to być trzy parametry oznaczające składowe czerwoną, zieloną i niebieską lub cztery parametry, w których czwarty oznacza składową alfa koloru. Składowa alfa określająca przezroczystość koloru opisana jest dokładniej w dalszej części rozdziału. Na razie będziemy używali trójparametrowej wersji tej funkcji.

Symbol `<t>` w nazwie funkcji określa typ parametrów przyjmowanych przez funkcję. Mogą to być określenia `b` (`byte`), `d` (`double`), `f` (`float`), `i` (`int`), `s` (`short`), `ub` (`unsigned byte`), `ui` (`unsigned int`) lub `us` (`unsigned short`). Kolejna wersja funkcji ma na końcu literę `v`. Oznacza ona, że ta wersja funkcji pobiera tablicę zawierającą wszystkie parametry (litera `v` oznacza *vector*, czyli wektor). W podrozdziale „Opisy funkcji" znajduje się dużo dokładniejszy opis funkcji `glColor`.

Większość programów, z jakimi przyjdzie nam się spotykać, wykorzystuje funkcję `glColor3f`, w której intensywność każdej składowej określana jest liczbą z zakresu od `0.0` do `1.0`. Jednak dla osób z doświadczeniem w programowaniu w systemie Windows znacznie

łatwiejsze może być użycie funkcji `glColor3ub`. Ta wersja pobiera trzy bajty bez znaku, czyli wartości od `0` do `255`. Stosowanie tej funkcji jest bardzo podobne do stosowania makra systemu Windows:

```
glColor3ub(0, 255, 128) = RGB(0, 255, 128)
```

Rzeczywiście, ta metoda może ułatwić dopasowanie kolorów w bibliotece OpenGL do kolorów wykorzystywanych w programie w czasie rysowania niezwiązanego z OpenGL. Mimo to trzeba zaznaczyć, że wewnętrznie biblioteka OpenGL przechowuje dane kolorów jako wartości zmiennoprzecinkowe, więc stosowanie wartości innego typu może spowodować spadek wydajności z powodu koniecznych przekształceń wykonywanych w czasie działania programu. Poza tym w przyszłości może się okazać, że powstaną znacznie pojemniejsze bufory kolorów (tak naprawdę to już zaczynają się pojawiać bufory zapamiętujące wartości zmiennoprzecinkowe), w których przechowywane wartości zmiennoprzecinkowe najdokładniej odwzorują stosowane kolory.

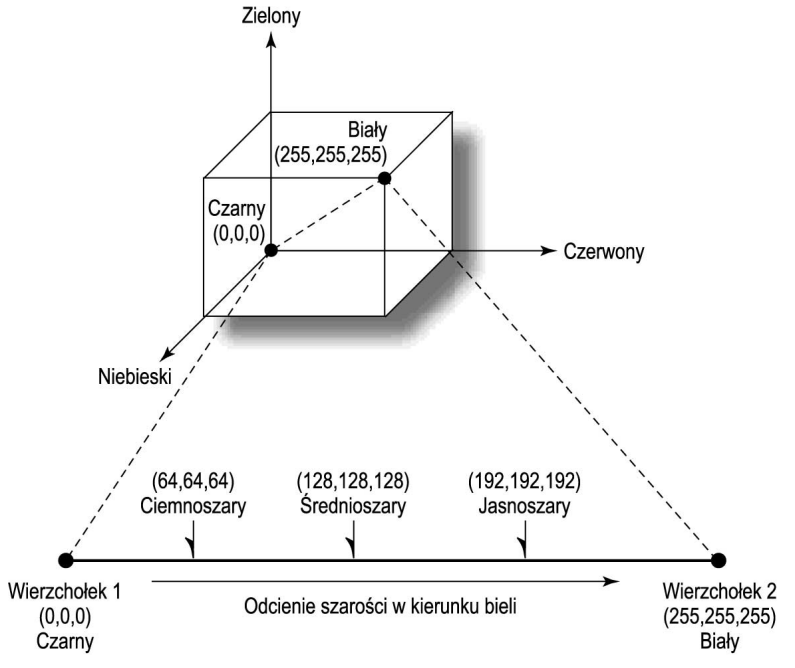
Cieniowanie

Poprzednio podawana definicja funkcji `glColor` mówiła, że ustala ona aktywny kolor rysujący, którym pokrywane są wszystkie obiekty rysowane po wywołaniu tej funkcji. Po omówieniu w poprzednim rozdziale wszystkich obiektów podstawowych biblioteki OpenGL możemy teraz nieco uszczegółowić tę definicję — funkcja `glColor` ustala aktualny kolor rysujący dla wszystkich wierzchołków definiowanych po wywołaniu tej funkcji. W dotychczasowych przykładach rysowaliśmy obiekty tylko w postaci szkieletów, a w ewentualnie wypełnionych obiektach każda powierzchnia miała inny kolor. Jeżeli jednak dla każdego wierzchołka obiektu podstawowego (punktu, linii lub wielokąta) zdefiniujemy inny kolor, to jakiego koloru będzie jego wnętrze?

Odpowiedź na to pytanie zaczniemy od punktów. Każdy punkt definiowany jest tylko jednym wierzchołkiem, dlatego jakikolwiek kolor zostanie przydzielony wierzchołkowi, będzie to aktywny kolor punktu. Proste.

Już jednak linia definiowana jest dwoma wierzchołkami, z których każdy może mieć inny kolor. Kolor samej linii tworzony jest przez model cieniowania. Cieniowanie definiowane jest po prostu jako płynne przejście od jednego koloru do drugiego. W przestrzeni kolorów RGB (musimy wrócić do rysunku 5.7) dowolne dwa punkty mogą być połączone linią prostą.

Płynne cieniowanie sprawia, że wzdłuż linii kolory zmieniają się w taki sam sposób jak kolory linii łączącej dwa punkty wewnątrz sześcianu kolorów. Na rysunku 5.9 przedstawiony został sześcian kolorów z zaznaczonymi punktami kolorów białego i czarnego. Poniżej znajduje się linia definiowana dwoma wierzchołkami — jednym białym, a drugim czarnym. Kolory wybierane w czasie rysowania tej linii pokrywają się dokładnie z kolorami leżącymi na prostej łączącej w sześcianie kolorów punkty biały i czarny. W ten sposób powstaje linia, która na początku jest czarna, a w dalszej części jej kolor zmienia się w coraz jaśniejszy szary, aż na końcu osiąga biel.



Rysunek 5.9. *Rysowanie linii z kolorami zmieniającymi się od czerni do bieli*

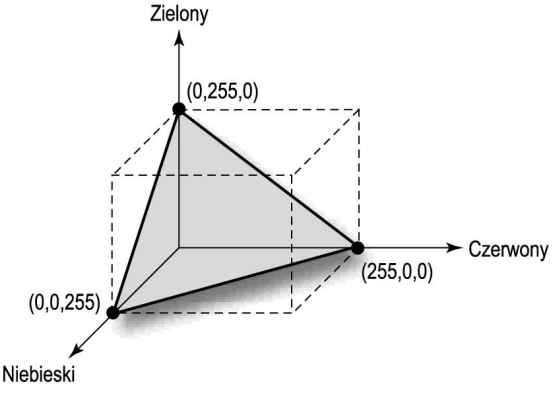
Możemy sami wykonać cieniowanie linii, znajdując matematyczne równanie linii łączącej dwa punkty wewnątrz trójwymiarowej przestrzeni kolorów RGB. Później wystarczy przejść w pętli od jednego końca linii do drugiego, sprawdzając po drodze odpowiednie współrzędne w przestrzeni kolorów wyznaczające kolor każdego piksela linii. W wielu dobrych książkach opisujących grafikę komputerową podawane jest wyjaśnienie algorytmu pozwalającego na uzyskanie doskonałych efektów cieniowania, przeniesienia kolorów linii z sześcianu kolorów na ekran komputera i tym podobne. Na szczęście całą tę pracę wykona za nas biblioteka OpenGL.

W przypadku wielokątów zadanie cieniowania staje się znacznie trudniejsze. Na przykład trójkąt może być również przedstawiony jako płaszczyzna wewnątrz sześcianu kolorów. Na rysunku 5.10 przedstawiono trójkąt, którego wierzchołkom nadano kolory o pełnym nasyceniu składowych czerwonej, zielonej i niebieskiej. Kod wyświetlający taki trójkąt podany został na listingu 5.1. Pochodzi on z programu *TRIANGLE* znajdującego się na płycie CD dołączonej do tej książki.

Listing 5.1. *Rysowanie trójkąta cieniowanego kolorami czerwonym, zielonym i niebieskim*

```
// Włączenie gładkiego cieniowania
glShadeModel(GL_SMOOTH);

// Rysowanie trójkąta
glBegin(GL_TRIANGLES);
  // Wierzchołek czerwony
  glColor3ub((GLubyte)255,(GLubyte)0,(GLubyte)0);
  glVertex3f(0.0f,200.0f,0.0f);
```

Rysunek 5.10.
Trójkąt w przestrzeni kolorów RGB




```
    // Wierzchołek zielony - na dole po prawej stronie
    glColor3ub((GLubyte)0,(GLubyte)255,(GLubyte)0);
    glVertex3f(200.0f,-70.0f,0.0f);

    // Wierzchołek niebieski - na dole po lewej stronie
    glColor3ub((GLubyte)0,(GLubyte)0,(GLubyte)255);
    glVertex3f(-200.0f, -70.0f, 0.0f);
  glEnd();
```

Ustalanie modelu cieniowania

W pierwszej linii listingu 5.1 ustalamy, że modelem cieniowania stosowanym przez bibliotekę OpenGL będzie cieniowanie gładkie, czyli model omawiany wcześniej. Co prawda jest to ustawienie domyślne, ale i tak lepiej wywoływać tę funkcję — upewniamy się w ten sposób, że wszystko będzie działało zgodnie z założeniami.

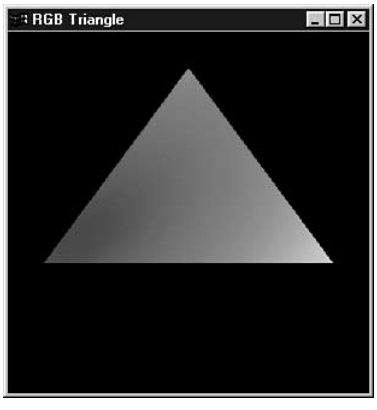
Drugim modelem cieniowania, jaki można włączyć funkcją `glShadeModel`, jest `GL_FLAT`, czyli cieniowanie płaskie. *Cieniowanie płaskie* oznacza, że wewnątrz obiektów podstawowych nie są wykonywane żadne obliczenia związane z cieniowaniem. Mówiąc ogólnie, przy włączonym cieniowaniu płaskim kolor całego obiektu podstawowego definiowany jest kolorem ostatniego wierzchołka definiującego ten obiekt. Jedynym wyjątkiem są obiekty podstawowe `GL_POLYGON`. Tutaj kolor obiektu definiuje kolor pierwszego wierzchołka.

Następnie, kod z listingu 5.1 nadaje górnemu wierzchołkowi kolor czerwony, dolnemu prawemu wierzchołkowi — kolor zielony, a dolnemu lewemu — kolor niebieski. Włączone jest cieniowanie gładkie, dlatego wnętrze trójkąta wypełnione jest płynnym przejściem pomiędzy kolorami wierzchołków.

Wynik działania programu *TRIANGLE* pokazany został na rysunku 5.11. Widać na nim płaszczyznę przedstawioną graficznie na rysunku 5.10.

Wielokąty znacznie bardziej złożone od trójkątów również mogą mieć definiowane różne kolory w poszczególnych wierzchołkach. Jednak w ich przypadku logika cieniowania może być już bardzo skomplikowana. Na całe szczęście dzięki bibliotece OpenGL nigdy nie będziemy musieli się o to martwić. Niezależnie od tego, jak złożony wielokąt zdefiniujemy, OpenGL doskonale wycieniuje jego wnętrze, stosując kolory nadane wierzchołkom.

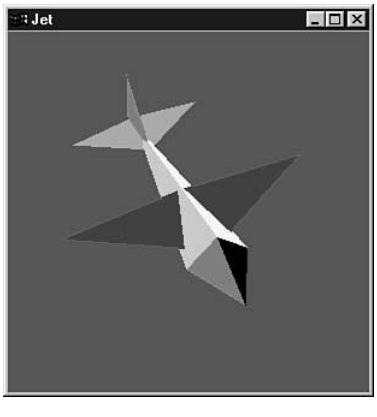
Rysunek 5.11.
Wynik działania programu TRIANGLE



Kolory w świecie rzeczywistym

W rzeczywistości kolory obiektów nie są niezmienne, nie są też cieniowaniem pomiędzy kilkoma wartościami RGB. Rysunek 5.12 przedstawia wynik działania programu *JET* znajdującego się na płycie CD. Jest to prosty samolot odrzutowy, poskładany ręcznie z trójkątów za pomocą tych metod, które opisywaliśmy do tej pory. Tak jak i w innych programach z tego rozdziału, możliwe jest obracanie samolotu poprzez naciskanie klawiszy strzałek.

Rysunek 5.12.
Prosty samolot odrzutowy, zbudowany z różnokolorowych trójkątów



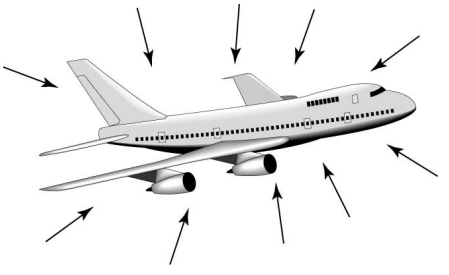
Kolory samolotu zostały tak dobrane, żeby podkreślać jego trójwymiarową strukturę. Nawet pomijając dziwaczną strukturę trójkątów tworzących samolot, można zauważyć, że nie wygląda on tak jak obiekty rzeczywiste. Załóżmy, że zbudowaliśmy model tego samolotu i każdą z powierzchni pomalowaliśmy odpowiednim kolorem. W zależności od użytej farby model sprawiałby wrażenie błyszczącego lub matowego, a kolor każdej płaskiej powierzchni zmieniałby się wraz ze zmianami kąta patrzenia i padania światła.

Biblioteka OpenGL całkiem dobrze radzi sobie z jak najlepszym odwzorowywaniem rzeczywistego wyglądu obiektów pod względem warunków oświetlenia. Każdy obiekt, o ile nie generuje własnego światła, oświetlany jest trzema rodzajami światła: otaczającym (*ambient*), rozproszonym (*diffuse*) i odbitym (*specular*).

Światło otaczające

Światło otaczające nie pochodzi z żadnego konkretnego kierunku. Posiada ono co prawda 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. We wszystkich poprzednich przykładach można zauważyć, że prezentowane obiekty oświetlone są właśnie takim światłem — są pokryte jednakowym kolorem niezależnie od ich położenia i naszego kąta patrzenia. Rysunek 5.13 przedstawia obiekt oświetlony światłem otaczającym.

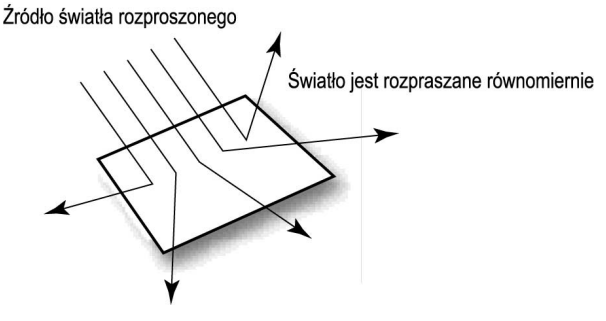
Rysunek 5.13. *Obiekt oświetlony wyłącznie światłem otaczającym*



Światło rozproszone

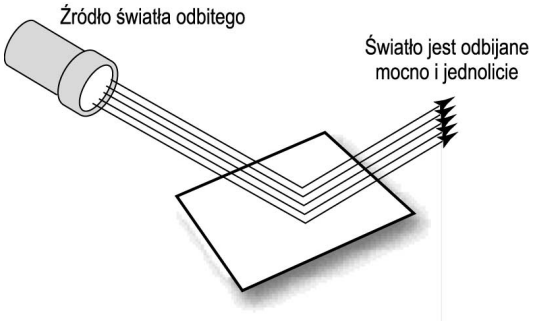
Światło rozproszone pada na obiekt z określonego kierunku, ale jest na nim rozpraszane we wszystkich kierunkach. Mimo to powierzchnie obiektu oświetlone bezpośrednio będą jaśniejsze od powierzchni ustawionych w stosunku do kierunku padania światła pod pewnym kątem. Dobrym przykładem takiego światła będzie światło jarzeniowe albo światło słoneczne wpadające w południe przez okna. Obiekt pokazany na rysunku 5.14 oświetlany jest światłem rozproszonym.

Rysunek 5.14. *Obiekt oświetlony wyłącznie światłem rozproszonym*



Światło odbite

Jeżeli światło pada na obiekt z pewnego kierunku (podobnie jak światło rozproszone), ale odbijane jest w jednym konkretnym kierunku, to mamy do czynienia ze światłem odbitym. Mocno odbite światło najczęściej powoduje wystąpienie na powierzchni oświetlanej jasnej plamy nazywanej *odbłyskiem* (ang. *specular highlight*). Przykładami światła odbitego mogą być latarka lub słońce. Na rysunku 5.15 można zobaczyć obiekt oświetlony wyłącznie światłem odbitym.



Rysunek 5.15. *Obiekt oświetlony światłem odbitym*

A teraz wszystko razem

Nie istnieje źródło światła, które składałoby się wyłącznie z jednego z opisanych wyżej rodzajów światła. Niemal każde światło składa się natomiast z różnych kombinacji tych rodzajów. Na przykład promień czerwonego lasera laboratoryjnego składa się niemal w całości z czerwonego światła odbitego. Jednak cząsteczki kurzu lub dymu mogą rozpraszać ten promień tak, że będzie on widoczny na całej swojej długości. W ten sposób uwidacznia się składowa światła rozproszenia. Dodatkowo, jeżeli promień będzie jasny, a w pokoju nie będą działały żadne inne źródła światła, to na wszystkich obiektach będzie można zobaczyć delikatną czerwoną poświatę. Jest ona spowodowana bardzo małą składową światła otaczającego obecną w promieniu lasera.

Jak widać, każde źródło światła obecne na scenie musi składać się z trzech składowych oświetlenia: otaczającego, rozproszenia i odbitego. Podobnie jak zwykłe kolory każda składowa oświetlenia również definiowana jest składowymi RGBA opisującymi względne intensywności światła czerwonego, zielonego i niebieskiego w każdym rodzaju światła (w opisie koloru światła składowa alfa jest ignorowana). Na przykład, światło wspomnianego czerwonego lasera może być opisywane wartościami składowych podanymi w tabeli 5.1.

Tabela 5.1. *Rozkład składowych światła w promieniu czerwonego lasera*

	Czerwony	Zielony	Niebieski	Alfa
Odbite	0.99	0.0	0.0	1.0
Rozproszone	0.10	0.0	0.0	1.0
Otoczenia	0.05	0.0	0.0	1.0

Proszę zauważyć, że promień czerwonego lasera nie posiada składowej światła zielonego ani niebieskiego. Ważne jest również to, że każda składowa światła (odbitego, rozproszonego i otoczenia) może przyjmować wartości od `0.0` do `1.0`. Można to interpretować w ten sposób, że światło lasera czerwonego będzie miało w pewnych scenach bardzo wysoką wartość składowej światła odbitego, niewielką wartość składowej światła rozproszonego i bardzo małą wartość składowej światła otoczenia. Tam, gdzie będzie padać promień lasera, z pewnością zobaczymy czerwoną plamkę. Poza tym w wyniku panujących w pokoju warunków (dym, kurz itp.), składowa rozproszenia z pewnością pozwoli nam dojrzeć w powietrzu promień lasera. W końcu składowa oświetlenia otoczenia (również

z powodu występujących w powietrzu cząsteczek dymu lub kurzu) niewielką część światła skieruje na elementy wyposażenia pokoju. Bardzo często łączy się składowe oświetlenia rozproszonego i otoczenia ze względu na ich podobną naturę.

Materiały w świecie rzeczywistym

Światło jest tylko jedną częścią równania. W świecie rzeczywistym obiekty mają swoje własne kolory. Wcześniej opisywaliśmy kolor obiektu jako kolor odbitego przez niego światła. Niebieska piłka odbija większość niebieskich fotonów, a pochłania większość pozostałych. Zakładaliśmy wtedy, że światło padające na piłkę zawiera w sobie niebieskie fotony, które mogłyby być odbite i odebrane przez obserwatora. Najczęściej, sceny oświetlane są światłem białym, które jest równomierną mieszanką wszystkich kolorów. Dlatego w świetle białym obiekty ujawniają swoje „naturalne kolory". Niestety, nie zawsze mamy tak doskonałe warunki. Jeżeli nasza niebieską piłkę umieścimy w ciemnym pokoju i oświetlimy ją światłem żółtym, to dla obserwatora piłka będzie czarna, ponieważ całe żółte światło zostanie pochłonięte przez piłkę, a z braku niebieskich fotonów nic nie będzie się od niej odbijało.

Właściwości materiałów

Korzystając z oświetlenia, nie definiujemy wielokątów, nadając im jakiś kolor, ale raczej przypisując im materiał posiadający pewne właściwości odblaskowe. Nie mówimy, że wielokąt jest czerwony, ale że jest zbudowany z materiału odbijającego głównie czerwone światło. Nadal możemy powiedzieć, że obiekt ma czerwoną powierzchnię, ale teraz musimy określić jego właściwości odblaskowe dla światła otoczenia, rozproszenia i odbitego. Materiał może być błyszczący i doskonale odbijać światło odbite, ale jednocześnie pochłaniać większą część światła rozproszenia lub otoczenia. Z drugiej strony materiały matowe mogą pochłaniać wszelkie światło odbite i w żadnych okolicznościach nie mieć błyszczącego wyglądu. Inną właściwością, jaką musimy określić, jest informacja o właściwościach emisji obiektów generujących własne światło, takich jak tylne światła samochodów albo zegarki świecące w ciemnościach.

Dodawanie światła do materiałów

Takie ustawienie parametrów oświetlenia i materiału, żeby uzyskać pożądane efekty, wymaga pewnej praktyki. Nie ma żadnych doskonałych porad, ani ogólnych zasad, do jakich należałoby się stosować. W tym miejscu analiza musi ustąpić miejsca sztuce, a nauka — magii. Biblioteka OpenGL w czasie rysowania każdego obiektu decyduje o kolorze, w jakim będzie rysowany każdy piksel tego obiektu. Sam obiekt ma „kolor" odblasku, a i źródło światła dysponuje własnymi „kolorami". W jaki sposób biblioteka określa kolor nadawany pikselom? Stosowane tu zasady nie są trudne, choć wymaga to użycia choćby najprostszego mnożenia (a nauczyciele mówili, że kiedyś się przyda!).

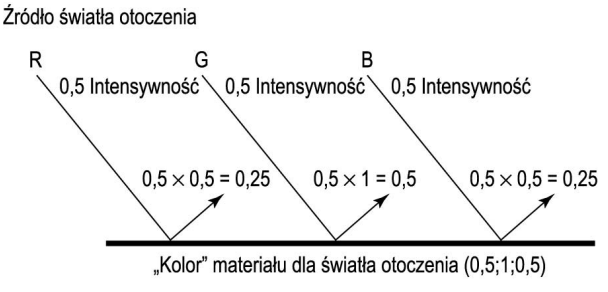
Każdemu wierzchołkowi obiektu podstawowego przepisywana jest wartość koloru RGB wynikająca z iloczynu połączonych efektów oświetlenia obiektu światłem otoczenia, rozproszenia i odbitym przez właściwości odblaskowe materiału dla światła otoczenia, rozproszenia i odbitego. Efekt pełnego oświetlenia obiektu uzyskiwany jest później poprzez zastosowanie płynnego cieniowania wnętrza wielokątów.

Wyliczanie efektów oświetlenia otoczenia

Wyliczając efekty oświetlenia otoczenia, musimy odłożyć na bok pojęcie koloru i zacząć myśleć wyłącznie o intensywnościach składowych czerwonej, zielonej i niebieskiej. Dla źródła światła otoczenia o połowicznej intensywności wszystkich składowych koloru wartości RGB tego źródła wynoszą (0.5, 0.5, 0.5). Jeżeli to źródło oświetla obiekt, którego właściwości odblaskowe dla światła otoczenia wynoszą (0.5, 1.0, 0.5), to wynikowy „kolor" składowych tego światła wynosi:

```
(0.5 * 0.5, 0.5 * 1.0, 0.5 * 0.5) = (0.25, 0.5, 0.25)
```

Jest to wynik mnożenia każdej składowej źródła światła otoczenia przez odpowiadające im właściwości odblaskowe materiału (rysunek 5.16).



Rysunek 5.16. *Wyliczanie koloru oświetlenia otoczenia dla obiektu*

Wynika z tego, że składowe koloru określają tylko procent ilości światła, jakie będzie odbijane od obiektu. W naszym przykładzie składowa czerwona światła miała wartość `0.5`, a właściwości materiału obiektu powodowały, że była ona odbijana tylko w połowie swojej połowicznej intensywności. Prosta matematyka mówi nam, że pół razy pół to ćwierć, czyli `0.25`.

Efekty światła rozproszenia i odbitego

Wyliczenia dla światła otoczenia są jednak wyjątkowo proste. Światło rozproszone również opisywane jest wartościami intensywności RGB, które w taki sam sposób łączą się z właściwościami materiału obiektu. Jednak w tym przypadku światło ma pewien kierunek, dlatego jego intensywność na powierzchni obiektu będzie różna w zależności od kąta padania światła, odległości obiektu od źródła, dowolnego czynnika tłumiącego (na przykład mgły znajdującej się między obiektem a źródłem światła) itp. Dokładnie te same zależności dotyczą również źródeł i intensywności światła odbitego. Efekt netto wartości RGB wyliczany jest w ten sam sposób co dla światła otoczenia, czyli intensywność źródła światła (zmodyfikowana przez kąt padania) mnożona jest przez współczynnik odblasku

materiału. Na końcu wszystkie trzy składowe koloru łączone są ze sobą, tworząc końcowy kolor obiektu. Jeżeli którakolwiek ze składowych będzie miała w wyniku tych wyliczeń wartość większą od `1.0`, to zostanie ograniczona do tej wartości. Nie można przecież mieć intensywności większej od maksymalnej.

Ogólnie, składowe światła otoczenia i rozproszonego mają prawie takie samo działanie w źródłach światła i materiałach, dlatego najbardziej wpływają na efektywny kolor obiektów. Światło odbite i odpowiadające mu właściwości materiału najczęściej są koloru szarego lub białego. Składowa światła odbitego najbardziej zależy od kąta padania, więc miejsca padania światła odbitego najczęściej są białe.

Dodawanie świateł do sceny

Może się wydawać, że powyższe wywody są ogromną ilością nagle podanej teorii. Zwolnimy więc i spokojnie przeanalizujemy przykłady kodu koniecznego, jeżeli chcemy stosować oświetlenie sceny. W ten sposób utrwalimy sobie wszystkie podane wyżej informacje, a na dodatek zademonstrujemy kilka nowych funkcji i wymagań mechanizmu świateł w bibliotece OpenGL. Następnych kilka przykładów opierać się będzie na programie *JET*. Pierwotna wersja nie stosuje żadnych mechanizmów związanych z oświetleniem i rysuje po prostu trójkąty z włączonym mechanizmem usuwania niewidocznych powierzchni (testowanie głębi). Gdy zakończymy wprowadzać do niego poprawki i zaczniemy obracać samolot, metaliczne powierzchnie naszego odrzutowca będą wspaniale lśnić w słońcu.

Włączenie oświetlenia

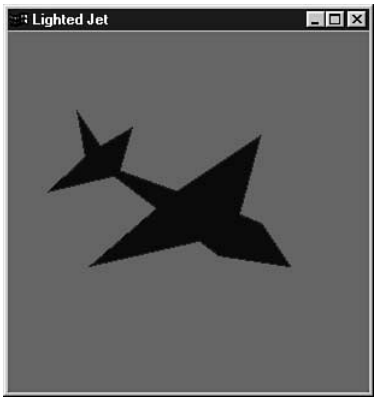
Aby nakazać bibliotece OpenGL przeprowadzanie obliczeń związanych z oświetleniem, wystarczy wywołać funkcję `glEnable` z parametrem `GL_LIGHTING`:

```
glEnable(GL_LIGHTING);
```

Już to proste wywołanie spowoduje, że biblioteka OpenGL w czasie określania kolorów wszystkich wierzchołków sceny będzie stosowała właściwości materiałów i parametry oświetlenia. Niestety, bez sprecyzowania tych właściwości i parametrów nasz obiekt pozostałby ciemny i nieoświetlony tak jak nasz przykładowy samolot z rysunku 5.17. Jeżeli spojrzymy na kod dowolnego z przykładowych programów wywodzących się z programu *JET*, to zobaczymy, że funkcję `SetupRC` wywołujemy zaraz po utworzeniu kontekstu renderowania. W tym właśnie miejscu wykonywane są wszystkie inicjalizacje parametrów oświetlenia.

Konfigurowanie modelu oświetlenia

Pierwszą rzeczą, jaką musimy wykonać zaraz po włączeniu obliczania oświetlenia, to skonfigurowanie modelu oświetlenia. Trzy parametry wpływające na model oświetlenia ustawiane są za pomocą funkcji `glLightModel`.



Rysunek 5.17.
Nieoświetlony samolot nie odbija żadnego światła

Pierwszym parametrem oświetlenia używanym w naszym następnym przykładzie (program *AMBIENT*) jest `GL_LIGHT_MODEL_AMBIENT`. Pozwala on określić globalne światło otoczenia oświetlające wszystkie obiekty w ten sam sposób ze wszystkich stron. Podany poniżej kod definiuje jasne, białe światło:

```
// Jasne białe światło - pełna intensywność wszystkich wartości RGB
GLfloat ambientLight[] = { 1.0f, 1.0f, 1.0f, 1.0f };

// Włączenie oświetlenia
glEnable(GL_LIGHTING);

// Konfigurujemy model oświetlenia tak, żeby stosował
// światło otoczenia zdefiniowane tablicą ambientLight[]
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,ambientLight);
```

Podana tutaj wersja funkcji `glLightModel` — `glLightModelfv` — w pierwszym parametrze pobiera informację o typie oświetlenia, jakie poddajemy modyfikacjom, a w drugim parametrze pobiera tablicę wartości RGBA opisującą światło. Domyślne wartości RGBA dla globalnego światła otoczenia wynoszą `(0.2, 0.2, 0.2, 1.0)`, co oznacza bardzo przyciemnione światło. Pozostałe parametry modelu oświetlenia pozawalają określić, czy oświetlane mają być przednie, tylnie czy może obie strony wielokątów oraz metody obliczania oświetlenia światłem odbitym. Więcej informacji o tych parametrach znajduje się w podrozdziale „Opisy funkcji”.

Ustalanie właściwości materiałów

Mamy już źródło światła otoczenia. Musimy teraz ustalić właściwości materiałów, tak żeby nasze wielokąty mogły odbijać to światło, a my — zobaczyć samolot. Istnieją dwie metody ustalania właściwości materiałów. Pierwsza z nich polega na użyciu funkcji `glMaterial` przed zdefiniowaniem każdego wielokąta lub zbioru wielokątów. Przyjrzymy się poniższemu fragmentowi kodu:

```
GLfloat gray[] = { 0.75f, 0.75f, 0.75f, 1.0f};
...
...
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, gray);
glBegin(GL_TRIANGLES);
```

```
    glVertex3f(-15.0f, 0.0f, 30.0f);
    glVertex3f(0.0f, 15.0f, 30.0f);
    glVertex3f(0.0f, 0.0f, -56.0f);
  glEnd();
```

Pierwszy parametr funkcji `glMaterialfv` określa, czy definiowane będą właściwości materiału przedniej, tylnej czy może obu stron wielokątów (`GL_FRONT`, `GL_BACK`, `GL_FRONT_AND_BACK`). W drugim parametrze podawana jest właściwość, która będzie zmieniana; w tym przypadku właściwości odblaskowe dla światła otoczenia i rozproszenia będą takie same. Ostatnim parametrem jest tablica zawierającą wartości RGBA opisujące odblaskowe właściwości materiału. Wszystkie obiekty podstawowe rysowane po wywołaniu tej funkcji będą miały ustalone przez nią właściwości odblaskowe aż do momentu ponownego wywołania funkcji `glMaterial`.

W większości przypadków składowe światła otoczenia i rozproszenia są takie same, a jeżeli nie chcemy stosować specjalnych efektów oświetlenia (iskrzenia, rozbłysków), nie musimy definiować żadnych właściwości dla światła odbitego. Mimo to przygotowywanie tablicy kolorów i wywoływanie funkcji `glMaterial` przed rysowaniem każdego wielokąta lub grupy wielokątów może być zajęciem bardzo nużącym.

Jesteśmy wiec gotowi na zastosowanie drugiej, lepszej metody ustalania właściwości materiałów — nazywanej *śledzeniem kolorów* (ang. *color tracking*). Dzięki tej metodzie możemy ustalić właściwości materiału już w momencie wywołania funkcji `glColor`. Aby włączyć ten mechanizm, trzeba najpierw wywołać funkcję `glEnable` z parametrem `GL_COLOR_MATERIAL`:

```
glEnable(GL_COLOR_MATERIAL);
```

Następnie funkcją `glColorMaterial` określamy parametry podawane zaraz za wartościami funkcji `glColor`. Na przykład, aby ustawić właściwości materiału dla światła otoczenia i rozproszenia dla przednich stron wielokątów tak, aby śledziły one kolory ustawiane funkcją `glColor`, należy wywołać:

```
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
```

Teraz podany wyżej fragment kodu ustawiający właściwości materiałów moglibyśmy zmodyfikować następująco. Może to wyglądać na tworzenie większej ilości kodu, ale tak naprawdę oszczędza się w ten sposób wiele linii kodu, co przekłada się na szybsze rysowanie wielu różnokolorowych wielokątów:

```
// Włączenie śledzenia kolorów
glEnable(GL_COLOR_MATERIAL);

// Kolory właściwości materiałów dla światła otoczenia i rozproszenia
// będą śledziły wartości podawane w funkcji glColor
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
...
...
glColor3f(0.75f, 0.75f, 0.75f);
glBegin(GL_TRIANGLES);
  glVertex3f(-15.0f, 0.0f, 30.0f);
  glVertex3f(0.0f, 15.0f, 30.0f);
  glVertex3f(0.0f, 0.0f, -56.0f);
glEnd();
```

Na listingu 5.2 podajemy kod, jaki dodany został do funkcji `SetupRC` z przykładowego programu *JET*, w którym ustalamy jasne źródło światła otoczenia, a także właściwości materiałów pozwalające obiektom odbijać światło i w efekcie pojawić się na ekranie. Pozmienialiśmy też kolory samolotu. Teraz różne kolory nadaliśmy poszczególnym sekcjom samolotu, a nie tworzącym go wielokątom. Końcowy wynik (pokazany na rysunku 5.18) nie różni się jednak znacząco od tego, co uzyskaliśmy jeszcze przed włączeniem oświetlenia. Jeżeli jednak zredukujemy światło otoczenia o połowę, to uzyskamy obraz przedstawiony na rysunku 5.19. Aby o połowę obniżyć intensywność oświetlenia, wartości RGBA dla światła otoczenia ustalamy następująco:

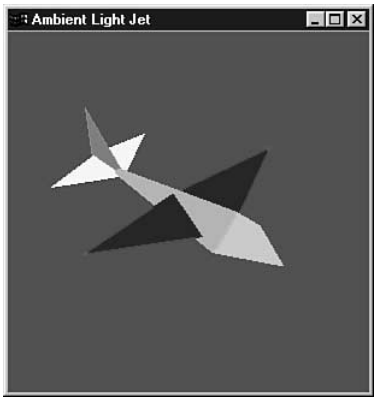
```
GLfloat ambientLight[] = { 0.5f, 0.5f, 0.5f, 1.0f };
```

Listing 5.2. *Konfiguracja warunków oświetlenia otoczenia*

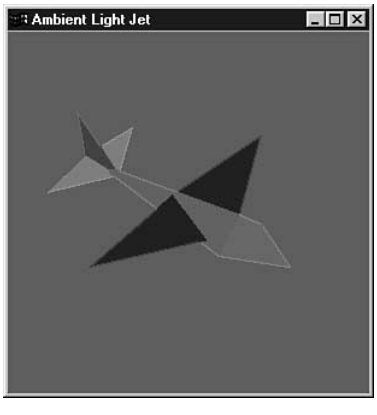
```
// Ta funkcja wykonuje wszystkie konieczne inicjalizacje kontekstu renderowania
// Tutaj ustala i inicjalizuje oświetlenie sceny
void SetupRC()
  {
  // Wartości światła
  // Jasne białe światło
  GLfloat ambientLight[] = { 1.0f, 1.0f, 1.0f, 1.0f };

  glEnable(GL_DEPTH_TEST);      // Usuwanie powierzchni ukrytych
  glEnable(GL_CULL_FACE);       // Nie będziemy prowadzić obliczeń wnętrza samolotu
  glFrontFace(GL_CCW);          // Wielokąty z nawinięciem przeciwnym do ruchu wskazówek
                                ↳zegara

  // Oświetlenie
  glEnable(GL_LIGHTING);        // Włączenie oświetlenia

  // Ustalamy, że model oświetlenia będzie stosował światło otoczenia
  // zdefiniowane w tablicy ambientLight []
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT,ambientLight);

  glEnable(GL_COLOR_MATERIAL);    // Włączenie śledzenia kolorów materiałów

  // Właściwości oświetlenia otoczenia i rozproszenia
  // będą śledzić wartości podawane funkcji glColor
  glColorMaterial(GL_FRONT,GL_AMBIENT_AND_DIFFUSE);

  // Śliczny jasny błękit
  glClearColor(0.0f, 0.0f, 0.5f,1.0f);
  }
```

Teraz wiemy już, jak można obniżyć jasność oświetlenia otoczenia tak, aby uzyskać ciemniejszy obraz. Ta właściwość bardzo przydaje się w symulacjach, w których zmierzch zapada stopniowo albo w momentach, gdy zablokowane zostaje główne źródło światła dla sceny, czyli gdy jeden obiekt znajduje się w cieniu drugiego.

Rysunek 5.18.
Wynik działania przykładowego programu AMBIENT



Rysunek 5.19.
Wynik działania przykładowego programu AMBIENT po przyciemnieniu oświetlenia



Używanie źródeł światła

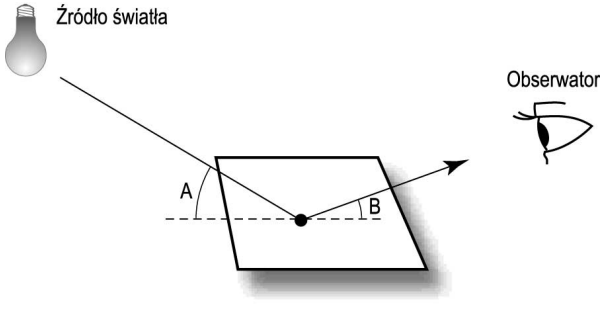
Manipulacje światłem otoczenia z pewnością są użyteczne, ale w większości aplikacji stawiających sobie za zadanie modelowanie świata rzeczywistego konieczne jest zdefiniowanie więcej niż tylko jednego źródła światła. Takie źródła światła, poza różnymi intensywnościami i kolorami, mają również swoje pozycje i kierunki w przestrzeni. Rozmieszczenie takich świateł może mieć znaczący wpływ na wygląd naszej sceny.

Biblioteka OpenGL pozwala na obsługę przynajmniej ośmiu niezależnych źródeł światła umieszczonych w dowolnych punktach na scenie albo poza przestrzenią widoczną. Możemy umieścić źródło światła w nieskończonej odległości od sceny, tak żeby jego promienie padające na scenę były zawsze równoległe albo umieścić je bardzo blisko, tak żeby promieniowało we wszystkich kierunkach. Możliwe jest też zdefiniowanie światła punktowego o dowolnych charakterystykach, które świeciło by wewnątrz zadanego stożka.

Gdzie jest góra?

Definiując źródło światła, musimy poinformować bibliotekę OpenGL gdzie się ono znajduje i w którym kierunku świeci. Bardzo często źródła światła świecą we wszystkich kierunkach, ale możliwe jest też tworzenie świateł kierunkowych. W każdym przypadku na powierzchnię rysowanego obiektu promienie światła (pochodzące z dowolnego źródła innego niż czyste światło otoczenia) padają pod bardzo różnymi kątami. Oczywiście w przypadku świateł kierunkowych nie muszą być oświetlane wszystkie wielokąty, z jakich składa się obiekt. Aby określić stopień zacienienia poszczególnych powierzchni obiektu, biblioteka OpenGL musi być w stanie wyliczyć kąt padania promieni światła.

Na rysunku 5.20 wielokąt (kwadrat) oświetlany jest promieniami światła generowanego przez pewne źródło. Promienie te tworzą z powierzchnią pewien kąt (A), a następnie odbijane są pod kątem (B) w kierunku widza (inaczej byśmy go nie zobaczyli). Do wyliczania wynikowego koloru każdego elementu stosowane są odpowiednie kąty padania promieni światła w połączeniu z parametrami oświetlenia i omawianymi wcześniej właściwościami materiałów. Tak się akurat składa, że konstrukcja biblioteki OpenGL wylicza te wartości dla każdego wierzchołka opisującego wielokąt. Dzięki temu, a także dzięki tworzeniu płynnych przejść pomiędzy kolorami poszczególnych wierzchołków, tworzona jest doskonała iluzja oświetlenia. Czary!



Rysunek 5.20. *Światło odbija się od powierzchni pod różnymi kątami*

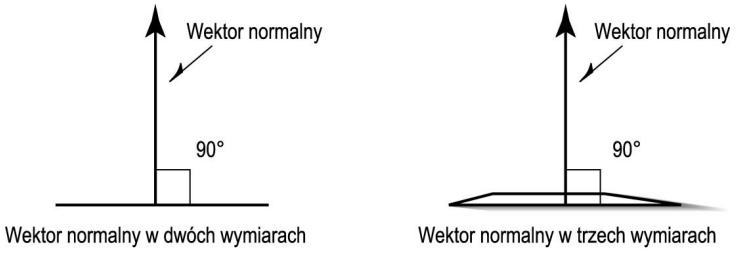
Z punktu widzenia programisty z takimi wyliczeniami oświetlenia wiążą się pewne problemy. Każdy wielokąt tworzony jest jako zbiór wierzchołków będących zaledwie punktami. Na każdy z tych wierzchołków pod pewnym kątem pada promień światła. Ale jak można wyznaczyć kąt pomiędzy punktem a linią (promieniem światła)? Oczywiście z geometrycznego punktu widzenia nie można wyznaczyć kąta pomiędzy punktem a linią umieszczoną w trzech wymiarach, ponieważ w takich warunkach istnieje nieskończona liczba możliwości. Z tego powodu konieczne jest związanie z każdym wierzchołkiem dodatkowej informacji wskazującej kierunek „w górę" wierzchołka, czyli powierzchni wielokąta.

Normalne powierzchni

Linia prowadzona od wierzchołka w kierunku górnym, pod kątem prostym w stosunku do pewnej wyimaginowanej powierzchni (lub naszego wielokąta) nazywana jest *wektorem normalnym*. Termin wektor normalny może kojarzyć się z rozmowami, jakie prowadzą między sobą bohaterowie serialu StarTrek, ale oznacza on po prostu linię prostopadłą

do rzeczywistej lub wyimaginowanej powierzchni. Wektor jest linią wskazującą w pewnym kierunku, a słowo *normalny*, to inne określenie na „prostopadły", czyli ustawiony pod kątem 90°. Bardzo lubią stosować je jajogłowi — jakby *prostopadły* nie było dość paskudnym słowem! Podsumowując — wektor normalny to linia wskazująca kierunek ustawiony pod kątem 90° w stosunku do powierzchni wielokąta. Na rysunku 5.21 przedstawiono przykłady wektorów normalnych w dwóch i trzech wymiarach.

Rysunek 5.21. *Wektor normalny w dwóch i trzech wymiarach*



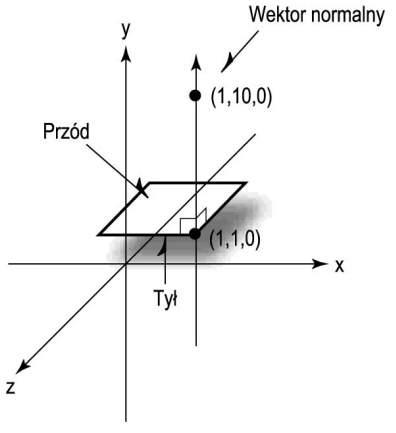
Powstaje tutaj pytanie, dlaczego musimy definiować wektor normalny dla każdego wierzchołka w wielokącie. Dlaczego nie możemy zdefiniować pojedynczego wektora dla wielokąta i stosować go we wszystkich jego wierzchołkach? Oczywiście, że możemy i w kilku pierwszych przykładach będziemy tak robić. Czasami jednak nie będziemy chcieli, żeby wektory normalne były dokładnie prostopadłe do powierzchni wielokąta. Z pewnością każdy zauważył, że nie wszystkie powierzchnie są płaskie! Można tworzyć przybliżenia takich powierzchni, stosując płaskie wielokąty, ale powstają w ten sposób kanciaste powierzchnie o wielu fasetach. Później omawiać będziemy techniki pozwalające na tworzenie iluzji łagodnych krzywizn za pomocą „wykręconych" wektorów normalnych (kolejne czary!). Ale zacznijmy od początku…

Definiowanie normalnej

Aby zobaczyć, jak definiuje się normalną dla wierzchołka, spójrzmy na rysunek 5.22, na którym znajduje się płaszczyzna umieszczona w przestrzeni trójwymiarowej ponad płaszczyzną *xz*. Staraliśmy się jak najbardziej uprościć ten rysunek, aby zademonstrować pojęcie normalnej. Proszę zauważyć linię biegnącą przez wierzchołek `(1, 1, 0)` prostopadle do płaszczyzny. Jeżeli wybierzemy na tej linii dowolny inny punkt, na przykład `(1, 10, 0)`, to linia od pierwszego punktu do drugiego będzie naszym wektorem normalnym. Drugi z podanych punktów określa, że kierunek normalnej wskazuje w górę osi *y*. W ten sposób określana jest też przednia i tylna strona wielokąta, ponieważ wektor normalny wskazuje też przednią stronę powierzchni.

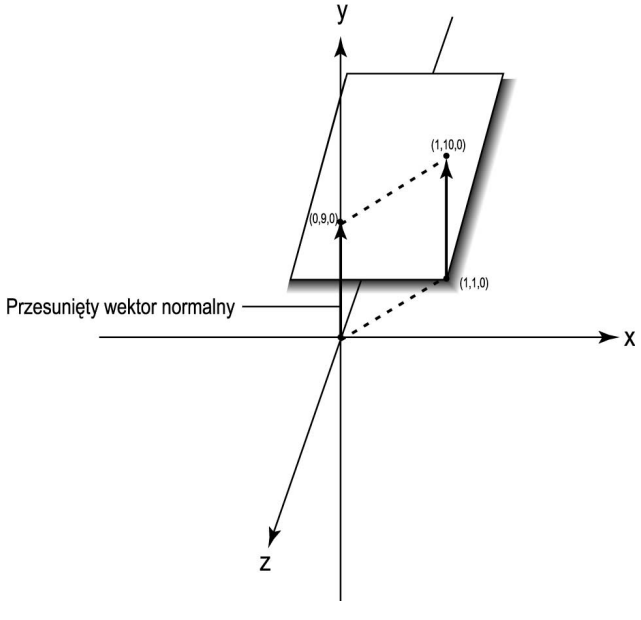
Jak widać, drugi punkt podawany jest jako liczba jednostek w osi *x*, *y* i *z*, opisujących pewien punkt na wektorze normalnym wychodzącym z wierzchołka. Zamiast podawać dwa punkty dla każdego wektora normalnego, możemy odjąć współrzędne wierzchołka od współrzędnych tego drugiego punktu, a otrzymamy trzy wartości współrzędnych określających odległość drugiego punktu od wierzchołka w osi *x*, *y* i *z*. W naszym przykładzie byłoby to:

```
(1, 10, 0) - (1, 1, 0) = (1 - 1, 10 - 1, 0) = (0, 9, 0)
```



Rysunek 5.22. *Wektor normalny oddalający się prostopadle od płaszczyzny*

Na powyższy przykład można spojrzeć jeszcze inaczej. Jeżeli wierzchołek zostanie przesunięty do początku układu współrzędnych, to punkt wyznaczony przez powyższe odejmowanie nadal będzie określał kierunek wektora normalnego w stosunku do powierzchni. Na rysunku 5.23 przedstawiony został wektor przesunięty w ten właśnie sposób.



Rysunek 5.23. *Przesunięty wektor normalny*

Wektor jest wielkością kierunkową informującą bibliotekę OpenGL o kierunku, w jakim ustawione są wierzchołki (lub wielokąty). Poniższy fragment kodu pochodzi z naszego przykładowego programu *JET* i przedstawia określanie wektora normalnego dla jednego z trójkątów.

```
glBegin(GL_TRIANGLES);
  glNormal3f(0.0f, -1.0f, 0.0f);
  glVertex3f(0.0f, 0.0f, 60.0f);
  glVertex3f(-15.0f, 0.0f, 30.0f);
  glVertex3f(15.0f, 0.0f, 30.0f);
glEnd();
```

Nawinięcie wielokątów

Proszę przyjrzeć się kolejności definiowania wierzchołków trójkąta pochodzącego z samolotu. Jeżeli będziemy się mu przyglądać od strony wskazywanej przez wektor normalny, to wierzchołki będą układać się w kierunku przeciwnym do ruchu wskazówek zegara. Nazywane jest to *nawinięciem wielokąta*. Domyślnie przodem wielokąta określa się tę stronę, po której wierzchołki układają się właśnie w kierunku przeciwnym do ruchu wskazówek zegara.

Funkcja `glNormal3f` pobiera w parametrach trzy współrzędne określające wektor normalny wskazujący kierunek prostopadły do powierzchni trójkąta. W tym przykładzie normalne wszystkich trzech wierzchołków mają dokładnie taki sam kierunek wskazujący w dół ujemnych wartości osi *y*. Jest to bardzo prosty przykład, ponieważ cały trójkąt leży w płaszczyźnie *xz*; jest on po prostu wycinkiem brzucha samolotu. Jak będzie można zobaczyć później, często będziemy musieli określać inne wektory normalne dla każdego wierzchołka wielokąta.

Perspektywa definiowania wektora normalnego dla każdego wierzchołka wielokąta może być nieco przerażająca, szczególnie że wiele powierzchni nie leży dokładnie na jednej z głównych płaszczyzn. Bez obaw! Szybko podamy przydatną funkcję, z której możemy korzystać cały czas do wyliczania wektorów normalnych.

Normalne jednostkowe

W czasie gdy biblioteka OpenGL będzie „odprawiać swoje czary", wszystkie wektory normalne muszą zostać zamienione w normalne jednostkowe. Normalna jednostkowa jest zwykłym wektorem normalnym o długości `1`. Normalna przedstawiona na rysunku 5.23 ma długość `9` jednostek. Długość każdej normalnej można uzyskać, podnosząc do kwadratu wartość każdej współrzędnej, sumując wyniki, a następnie wyciągając z tej sumy pierwiastek kwadratowy. Jeżeli teraz podzielimy każdą składową wektora normalnego przez tę długość, to otrzymamy identycznie skierowany wektor, ale o długości równej `1`. W tym przypadku nasz nowy wektor normalny będzie określany współrzędnymi `(0, 1, 0)`. Proces ten nazywany jest *normalizacją*. Tak więc dla potrzeb obliczeń oświetlenia, wszystkie wektory normalne musza zostać znormalizowane. Świetny żargon!

Możemy nakazać bibliotece OpenGL wykonywanie normalizacji wszystkich wektorów normalnych. Trzeba tylko wywołać funkcję `glEnable` z parametrem `GL_NORMALIZE`:

```
glEnable(GL_NORMALIZE);
```

Z tą metodą wiążą się jednak pewne spadki wydajności. Znacznie lepszym wyjściem jest od razu wyliczyć znormalizowany wektor normalny, niż zlecać bibliotece OpenGL wykonywanie tego zadania.

Trzeba zaznaczyć, że wywołanie funkcji `glScale` również będzie wpływać na długość naszego wektora normalnego. Jeżeli będziemy jednocześnie stosować funkcję `glScale` i mechanizmy oświetlenia, to możemy uzyskać różne niepożądane efekty. Po zdefiniowaniu dla jakiegoś obiektu wszystkich normalnych jednostkowych i w użyciu funkcji skalującej ze stałym współczynnikiem skalowania (skalowanie jest takie samo we wszystkich kierunkach), to możemy zamiast parametru `GL_NORMALIZE` zastosować nowy (wprowadzony w wersji 1.2 biblioteki) parametr `GL_RESCALE_NORMALS`. Parametr ten włączany jest prostym wywołaniem:

```
glEnable(GL_RESCALE_NORMALS);
```

W ten sposób informujemy bibliotekę OpenGL, że nasze normalne nie mają jednostkowej długości, ale można je przeskalować o dokładnie taką samą wartość, żeby osiągnęły długość jednostkową. Biblioteka OpenGL sprawdza to, kontrolując macierz model-widok. W efekcie musimy wykonać mniej operacji matematycznych niż w innych przypadkach.

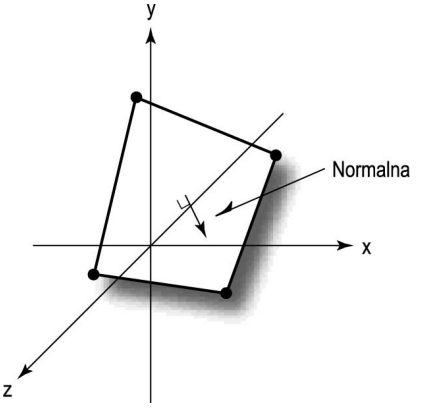
Ze względu na to, że najlepszym rozwiązaniem jest podanie bibliotece OpenGL już znormalizowanych wektorów, w bibliotece *glTools* dostępna jest funkcja pobierająca dowolny wektor i poddająca go normalizacji:

```
void gltNormalizeVector(GLTVector vNormal);
```

Znajdowanie normalnej

Na rysunku 5.24 przedstawiony został kolejny wielokąt, który nie leży w całości na żadnej z płaszczyzn osiowych. Teraz znalezienie wektora normalnego do powierzchni wielokąta nie jest już tak oczywiste, potrzebujemy więc prostego sposobu na wyliczenie wektora normalnego dla dowolnego wielokąta w przestrzeni trójwymiarowej.

Rysunek 5.24. *Nietrywialny problem szukania wektora normalnego*

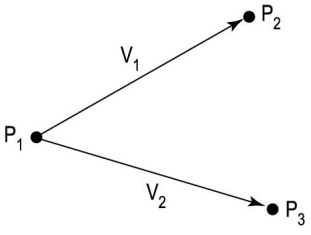


Biorąc trzy punkty leżące na płaszczyźnie dowolnego wielokąta, można w prosty sposób wyliczyć normalną do tej płaszczyzny. Rysunek 5.25 przedstawia trzy punkty — *P1*, *P2* i *P3* — których można użyć do zdefiniowania dwóch wektorów: wektora *V1* od punktu *P1* do punktu *P2* i wektora *V2* od punktu *P1* do punktu *P3*. Dwa wektory w przestrzeni definiują płaszczyznę (a na płaszczyźnie definiowanej przez te dwa wektory leży nasz wielokąt). Jeżeli teraz wykonamy iloczyn wektorowy tych dwóch wektorów (w matematyce zapisuje się to jako `V1 × V2`), to wynikowy wektor będzie prostopadły do naszej płaszczyzny. Rysunek 5.26 przedstawia wektor *V3* będący wynikiem iloczynu wektorowego wektorów *V1* i *V2*.

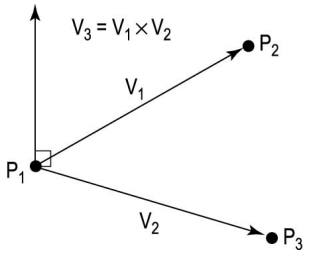
I znowu biblioteka *glTools* zawiera w sobie funkcję wyliczającą wektor normalny na podstawie trzech punktów wielokąta:

```
void gltGetNormalVector(GLTVector vP1, GLTVector vP2, GLTVector vP3,
↳GLTVector vNormal);
```

Rysunek 5.25. *Dwa wektory definiowane przez trzy punkty leżące na płaszczyźnie*



Rysunek 5.26. *Wektor normalny będący iloczynem wektorowym dwóch wektorów*



Aby skorzystać z tej funkcji, trzeba przekazać jej trzy wektory (każdy będący tablicą trzech wartości typu `float`) opisujące punkty naszego trójkąta (muszą być podawane z nawinięciem przeciwnym do ruchu wskazówek zegara) oraz tablicę dodatkowego wektora, do której zostaną wpisane wartości wektora normalnego.

Konfigurowanie źródła

Znamy już teraz zasady, jakich trzeba przestrzegać w czasie tworzenia wielokątów, tak aby mogły na nie oddziaływać źródła światła. Nadszedł czas włączenia świateł! Na listingu 5.3 można zobaczyć kod funkcji `SetupRC` pochodzącej z przykładowego programu *LITJET*. W czasie konfigurowania sceny tworzone jest w nim źródło światła i umieszczane po lewej górnej stronie, nieco za obserwatorem. Składowym światła otoczenia i rozproszonego w źródle `GL_LIGHT0` nadawane są intensywności zdefiniowane w tablicach `ambientLight[]` i `diffuseLight[]`. W efekcie powstaje źródło delikatnego światła białego:

```
GLfloat  ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
GLfloat  diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
...
...
// Konfiguracja i włączenie światła numer 0
glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
```

Listing 5.3. *Konfiguracja oświetlenia i kontekstu renderowania w programie LITJET*

```
// Ta funkcja wykonuje wszystkie konieczne inicjalizacje kontekstu renderowania,
// a także konfiguruje i inicjalizuje oświetlenie sceny
void SetupRC()
  {
  // Współrzędne i wartości oświetlenia
  GLfloat  ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
  GLfloat  diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
```

```
    glEnable(GL_DEPTH_TEST);        // Usuwanie ukrytych powierzchni
    glFrontFace(GL_CCW);            // Wielokąty z nawinięciem przeciwnym do ruchu
                                    ↳wskazówek zegara
    glEnable(GL_CULL_FACE);         // Nie będziemy prowadzić obliczeń wnętrza samolotu

    // Włączenie oświetlenia
    glEnable(GL_LIGHTING);

    // Konfiguracja i włączenie światła numer 0
    glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
    glEnable(GL_LIGHT0);

    // Włączenie śledzenia kolorów
    glEnable(GL_COLOR_MATERIAL);

    // Właściwości oświetlenia otoczenia i rozproszenia
    // będą śledzić wartości podawane funkcji glColor
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

    // Jasnoniebieskie tło
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f );
    }
```

Na koniec światło `GL_LIGHT0` zostaje włączone:

```
glEnable(GL_LIGHT0);
```

Pozycja światła ustalana jest wewnątrz kodu funkcji `ChangeSize`:

```
GLfloat lightPos[] = { -50.f, 50.0f, 100.0f, 1.0f };
...
...
glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
```

W powyższych wierszach w tablicy `lightPos[]` zapisywana jest pozycja źródła światła. Ostatnia wartość w tej tablicy to `1.0`, co oznacza, że źródło światła zlokalizowane jest w podanych współrzędnych. Gdybyśmy podali tam wartość `0.0`, oznaczałoby to, że źródło światła znajduje się w nieskończonej odległości od sceny, a w tablicy podany jest wektor kierunku, z którego będą padały promienie. Później powiemy nieco więcej na ten temat. Światła są podobne do obiektów geometrycznych pod tym względem, że możemy je przemieszczać za pomocą macierzy model-widok. Umiejscawiając światło w momencie wykonywania przekształcenia punktu widzenia, zyskujemy pewność, że będzie ono ustawione prawidłowo, niezależnie od późniejszych przekształceń geometrii.

Ustalanie właściwości materiałów

Proszę zauważyć, że na listingu 5.3 włączany jest mechanizm śledzenia kolorów. Jest on też konfigurowany tak, że śledzeniu podlegają właściwości odblasku światła otoczenia i rozproszenia dla przednich powierzchni wielokątów. Dokładnie tak samo postępowaliśmy w przykładowym programie *AMBIENT*:

```
// Włączenie śledzenia kolorów
glEnable(GL_COLOR_MATERIAL);

// Właściwości oświetlenia otoczenia i rozproszenia
// będą śledzić wartości podawane funkcji glColor
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
```

Definiowanie wielokątów

W związku z obsługą nowego modelu oświetlenia kod renderujący znany z dwóch pierwszych przykładowych programów *JET* musiał zostać znacząco przebudowany. Na listingu 5.4 przedstawiono wycinek kodu pochodzący z funkcji RenderScene z programu *LITJET*.

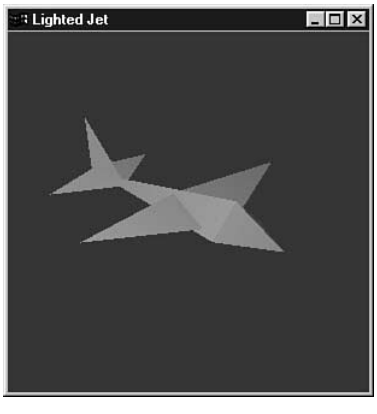
Listing 5.4. *Przykładowy kod ustalający kolor oraz wyliczający wektory normalne wielokątów*

```
GLTVector vNormal;    // Przechowuje wyliczone wektory normalne
...
...
// Ustalenie koloru materiału
glColor3ub(128, 128, 128);
glBegin(GL_TRIANGLES);
  glNormal3f(0.0f, -1.0f, 0.0f);
  glNormal3f(0.0f, -1.0f, 0.0f);
  glVertex3f(0.0f, 0.0f, 60.0f);
  glVertex3f(-15.0f, 0.0f, 30.0f);
  glVertex3f(15.0f,0.0f,30.0f);

  // Wierzchołki tego panelu
  {
  GLTVector vPoints[3] = {{ 15.0f, 0.0f,  30.0f},
                          { 0.0f,  15.0f, 30.0f},
                          { 0.0f,  0.0f,  60.0f}};

  // Wyliczenie wektora normalnego dla tego wielokąta
  gltGetNormalVector(vPoints[0], vPoints[1], vPoints[2], vNormal);
  glNormal3fv(vNormal);
  glVertex3fv(vPoints[0]);
  glVertex3fv(vPoints[1]);
  glVertex3fv(vPoints[2]);
  }

  {
  GLTVector vPoints[3] = {{ 0.0f, 0.0f, 60.0f },
                          { 0.0f, 15.0f, 30.0f },
                          { -15.0f, 0.0f, 30.0f }};

  gltGetNormalVector(vPoints[0], vPoints[1], vPoints[2], vNormal);
  glNormal3fv(vNormal);
  glVertex3fv(vPoints[0]);
  glVertex3fv(vPoints[1]);
  glVertex3fv(vPoints[2]);
  }
```

Wartość wektora normalnego wyliczamy za pomocą funkcji `gltGetNormalVector` pochodzącej z biblioteki *glTools*. Poza tym właściwości materiałów będą ustalane zgodnie z kolorami podawanymi funkcji `glColor`. Można też zauważyć, że trójki wierzchołków definiujących trójkąty nie są zamykane między funkcjami `glBegin` i `glEnd`. Wystarczyło raz określić, że będziemy definiować trójkąty, a do momentu wywołania funkcji `glEnd` każda kolejna trójka wierzchołków będzie wykorzystywana jako definicja nowego trójkąta. Jeżeli będziemy definiować duże ilości wielokątów, to ta technika może bardzo podnieść wydajność naszego programu poprzez wyeliminowanie niepotrzebnych wywołań funkcji i konfigurowania kolejnych wielokątów.

Na rysunku 5.27 przedstawiono wynik działania programu *LITJET*. Samolot nie jest już różnokolorowy, ale pokryty jest jednolitą, jasnoszarą barwą. Kolor został zmieniony w ten sposób, aby uwidocznić wpływ oświetlenia na powierzchnie samolotu. Mimo że wszystkie powierzchnie samolotu mają ten sam „kolor", to jednak możemy wyróżnić wszystkie jego elementy. Obracając samolot klawiszami strzałek, możemy zobaczyć zmiany odcieni poszczególnych powierzchni i sprawdzić wpływ, jaki oświetlenie ma na nie pod różnymi kątami.

Rysunek 5.27. *Wynik działania programu LITJET*



Najbardziej oczywistą metodą poprawienia wydajności tego kodu jest wcześniejsze wyliczenie wszystkich wektorów normalnych i późniejsze wykorzystanie ich w funkcji `RenderScene`. Zanim ktokolwiek zacznie drążyć ten temat, powinien najpierw przeczytać w rozdziale 11., „Wszystko o potokach — szybkie przekazywanie geometrii" informacje o listach wyświetlania i tablicach wierzchołków. Umożliwiają one zapisywanie wyliczonych wartości nie tylko wektorów normalnych, ale również danych wielokątów. Proszę pamiętać, że te przykłady były wyłącznie demonstracją pojęć i z pewnością nie stanowią najefektywniejszego rozwiązania.

Efekty świetlne

Światło otoczenia i rozproszone stosowane w programie *LITJET* są całkowicie wystarczające, aby uzyskać efekt oświetlenia. W zależności od kąta padania promieni światła poszczególne powierzchnie samolotu cieniowane są w różny sposób. W czasie gdy samolot się obraca, kąty te ulegają ciągłym zmianom, a my możemy obserwować zmiany oświetlenia i na tej podstawie określić położenie źródła światła.

Do tej pory ignorowaliśmy jednak składową światła skupienia w źródłach światła, jak również odpowiadającą jej składową właściwości materiału. Co prawda efekty oświetlenia są wyliczane, ale kolory powierzchni samolotu są raczej matowe. Światło otoczenia i rozproszone i odpowiadające im właściwości materiałów są wystarczające w czasie modelowania obiektów z gliny, drewna, tektury, tkaniny i innych matowych materiałów. Jednak samolot składa się głównie z powierzchni metalizowanych, którym przydałoby się nieco połysku.

Odbłyski

Właściwości materiału i światła odbitego nadają naszym materiałom wymaganego połysku. Ma on wybielający wpływ na kolory obiektu, co może doprowadzić do powstania odbłysków, jeżeli kąt padania światła na powierzchnię będzie odpowiedni w stosunku do pozycji obserwatora. Odbłysk powstaje wtedy, kiedy niemal całe światło padające na powierzchnię odbijane jest w stronę obserwatora. Dobrym przykładem takiego odbłysku jest jasna plama na powierzchni, leżącej w słońcu, czerwonej piłki.

Światło odbite

Do źródła światła można bardzo łatwo dodać składową światła odbitego. Poniżej przedstawiono kod konfigurujący oświetlenie pochodzący z programu *LITJET*, w którym dodano instrukcje konfigurujące składową światła odbitego:

```
// Współrzędne i wartości oświetlenia
GLfloat  ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
GLfloat  diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
GLfloat  specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
...
...
// Włączenie oświetlenia
glEnable(GL_LIGHTING);

// Konfiguracja i włączenie światła numer 0
glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
glLightfv(GL_LIGHT0,GL_SPECULAR,specular);
glEnable(GL_LIGHT0);
```

W tablicy `specular[]` zdefiniowana została bardzo jasna składowa białego światła odbitego. Chcemy w ten sposób zasymulować jasne światło słoneczne. Poniższą linią dodajemy do źródła światła `GL_LIGHT0` przygotowaną wcześniej składową światła odbitego:

```
glLightfv(GL_LIGHT0,GL_SPECULAR,specular);
```

Jeżeli byłaby to jedyna zmiana, jaką wprowadzilibyśmy do programu *LITJET*, to wygląd wyświetlanego samolotu nie zmieniłby się ani trochę. Po prostu nie zdefiniowaliśmy jeszcze żadnych właściwości odblaskowych materiału samolotu.

Współczynnik odbicia

Uzupełnienie właściwości materiału o współczynnik odbicia jest niemal tak proste jak uzupełnienie źródła światła o składową światła odbitego. Poniższy fragment kodu pochodzi z programu *LITJET* i ponownie został zmodyfikowany tak, aby dodać do właściwości materiału informację o współczynniku odbicia światła odbitego:

```
// Współrzędne i wartości oświetlenia
GLfloat  specref[] = { 1.0f, 1.0f, 1.0f, 1.0f };
...
...
// Włączenie śledzenia kolorów
glEnable(GL_COLOR_MATERIAL);

// Właściwości oświetlenia otoczenia i rozproszenia
// będą śledzić wartości podawane funkcji glColor
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

// Od tego momentu wszystkie materiały uzyskają połysk
// i będą doskonale odbijały światło odbite
glMaterialfv(GL_FRONT, GL_SPECULAR, specref);
glMateriali(GL_FRONT, GL_SHININESS, 128);
```

Tak jak poprzednio włączamy mechanizm śledzenia kolorów, aby właściwości odbijania światła otoczenia i rozproszonego były zgodne z aktualnym kolorem definiowanym funkcją `glColor`. Oczywiście nie chcemy, żeby funkcja `glColor` definiowała również właściwości odbijania światła odbitego. Tę właściwość zdefiniujemy sami i nie będzie ona podlegała żadnym zmianom.

Dodaliśmy teraz tablicę `specref[]` zawierającą wartości RGBA opisujące współczynnik odbicia dla materiału. Zapisujemy w niej same jedynki, dzięki czemu tworzymy powierzchnie odbijające praktycznie każde światło odbite. Poniższa linia ustala właściwości materiału dla wszystkich definiowanych po niej wielokątów tak, aby zawierały w sobie dane współczynnika odbicia:

```
glMaterialfv(GL_FRONT, GL_SPECULAR, specref);
```

Później już nie wywołujemy funkcji `glMaterial` z parametrem `GL_SPECULAR`, dlatego wszystkie materiały będą miały takie same właściwości dla światła odbitego. Celowo tworzymy ten przykład w taki sposób, ponieważ chcemy, aby cały samolot sprawiał wrażenie, że zbudowany jest z bardzo błyszczących materiałów.

To, co zrobiliśmy w naszej funkcji konfigurującej, jest niezwykle istotne. Spowodowaliśmy, że właściwości odblaskowe materiałów dla światła otoczenia i rozproszonego dla wszystkich tworzonych w przyszłości wielokątów (do czasu aż zmienimy to wywołaniem funkcji `glMaterial` lub `glColorMaterial`), będą zmieniały się razem ze zmianami kolorów tych wielokątów, a jednocześnie właściwości tych samych materiałów dla światła odbitego będą niezmienne.

Wykładnik odbłysku

Jak mówiliśmy już wcześniej, wysoka wartość jasności światła odbitego i odpowiednich właściwości odblaskowych będą rozjaśniały kolory obiektu. W naszym przykładzie zastosowaliśmy wyjątkowo wysoką wartość jasności światła (pełna intensywność) oraz współczynnika odbicia (najlepsze odbicia). W efekcie samolot byłby niemal wyłącznie biały lub szary, za wyjątkiem powierzchni odwróconych od źródła światła (te byłyby całkowicie czarne i nieoświetlone). Możemy złagodzić ten efekt za pomocą poniższej linii kodu umieszczonej zaraz za definicją właściwości odblaskowych materiałów:

```
glMateriali(GL_FRONT, GL_SHININESS, 128);
```

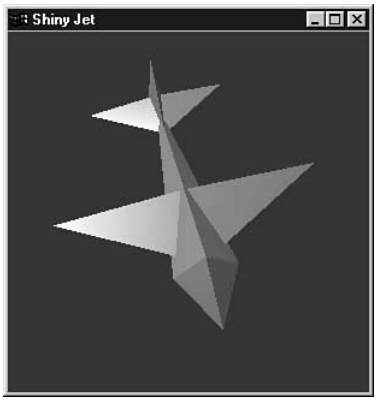
Właściwość `GL_SHININESS` ustala wykładnik odbłysku (ang. *specular exponent*) materiału określający, jak mały i skupiony na być rozmiar odbłysku. Wartość `0` oznacza, że odbłysk w ogóle nie będzie skupiony, co w efekcie prowadzi do równomiernego rozjaśniania kolorów na powierzchni całych wielokątów. Zmieniając tę wartość, zwiększamy skupienie rozbłysku, co powoduje, że na obiekcie pojawia się coraz mniejsza, jasna plama. Im wyższa wartość, tym bardziej błyszcząca będzie powierzchnia obiektu. We wszystkich implementacjach biblioteki OpenGL wartość tego parametru można ustalać w zakresie od `0` do `128`.

Listing 5.5 przedstawia kod funkcji `SetupRC` pochodzący z przykładowego programu *SHINYJET*. Jest to jedyny kod, jaki został zmieniony w stosunku do programu *LITJET* (no, może poza nazwą okna), a na ekranie pojawia się pięknie błyszczący samolot. Na rysunku 5.28 przedstawiony został efekt działania programu, jednak wszystkie efekty można w pełni docenić, dopiero włączając sam program i klawiszem strzałki obracając samolot w świetle słonecznym.

Listing 5.5. *Kod konfigurujący tworzący w programie SHINYJET efekty odbłysków świetlnych*

```
// Ta funkcja wykonuje wszystkie konieczne inicjalizacje kontekstu renderowania,
// a także konfiguruje i inicjalizuje oświetlenie sceny
void SetupRC()
  {
  // Współrzędne i wartości oświetlenia
  GLfloat  ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
  GLfloat  diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
  GLfloat  specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
  GLfloat  specref[] = { 1.0f, 1.0f, 1.0f, 1.0f };

  glEnable(GL_DEPTH_TEST);        // Usuwanie ukrytych powierzchni
  glFrontFace(GL_CCW);            // Wielokąty z nawinięciem przeciwnym do ruchu
                                  ↳wskazówek zegara
  glEnable(GL_CULL_FACE);         // Nie będziemy prowadzić obliczeń wnętrza samolotu

  // Włączenie oświetlenia
  glEnable(GL_LIGHTING);

  // Konfiguracja i włączenie światła numer 0
  glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
  glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
  glLightfv(GL_LIGHT0,GL_SPECULAR, specular);
  glEnable(GL_LIGHT0);
```

Rysunek 5.28.
Wynik działania programu SHINYJET




```
// Włączenie śledzenia kolorów
glEnable(GL_COLOR_MATERIAL);

// Właściwości oświetlenia otoczenia i rozproszenia
// będą śledzić wartości podawane funkcji glColor
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

// Od tego momentu wszystkie materiały uzyskają połysk
// i będą doskonale odbijały światło odbite
glMaterialfv(GL_FRONT, GL_SPECULAR, specref);
glMateriali(GL_FRONT, GL_SHININESS, 128);

// Jasnoniebieskie tło
glClearColor(0.0f, 0.0f, 1.0f, 1.0f );
}
```

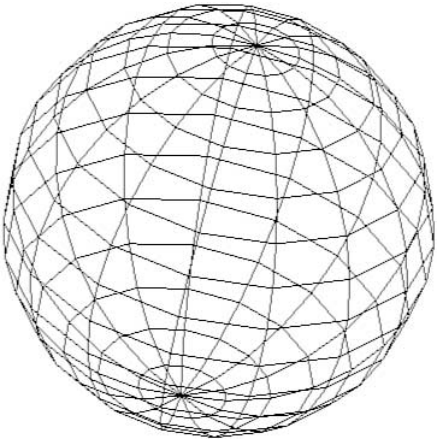
Uśrednianie normalnych

Wcześniej wspominaliśmy, że „wykrzywiając" wektory normalne możemy wytworzyć gładką powierzchnię za pomocą płaskich wielokątów. Za pomocą tej techniki, znanej jako *uśrednianie normalnych* (normal averaging), można uzyskać wiele interesujących iluzji optycznych. Załóżmy, że posiadamy kulę złożoną z czworokątów i trójkątów podobną do przedstawionej na rysunku 5.29.

Jeżeli każdej fasecie tej kuli przypisano by tylko jeden wektor normalny, to wyglądałaby ona jak wielki diament o bardzo wielu fasetach. Jeżeli jednak każdemu wierzchołkowi przypiszemy jego „prawdziwy" wektor normalny, to obliczenia oświetlenia wygenerują wartości kolorów, które zostaną przez bibliotekę OpenGL płynnie interpolowane wewnątrz wielokątów. W ten sposób płaskie wielokąty zostaną wycieniowane tak, jakby były gładką powierzchnią kuli.

Czym jednak jest „prawdziwy" wektor normalny? Aproksymacja wielokątami jest tylko przybliżeniem rzeczywistej powierzchni. Teoretycznie, jeżeli zastosowalibyśmy wystarczającą liczbę wielokątów, to powierzchnia zaczęłaby wyglądać na gładką. Z podobnego pomysłu skorzystaliśmy w rozdziale 3., „Rysowanie w przestrzeni. Geometryczne obiekty

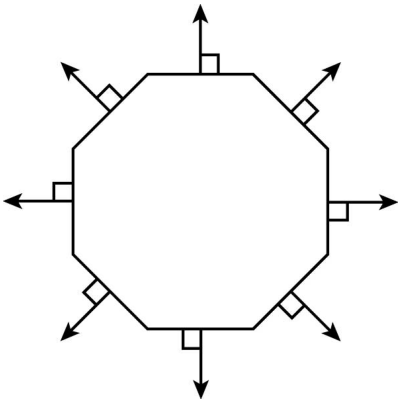
Rysunek 5.29.
Typowa kula zbudowana z czworokątów i trójkątów



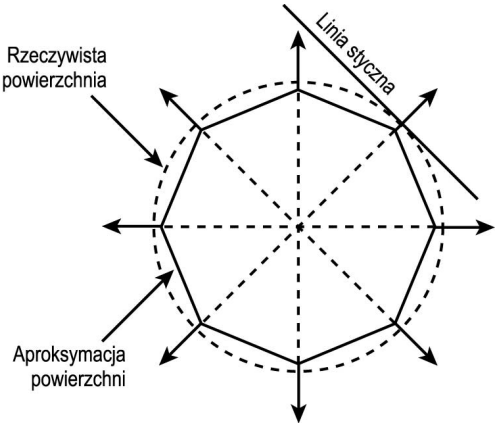
podstawowe i bufory", do rysowania gładkich linii krzywych za pomocą wielu krótkich linii prostych. Jeżeli uznamy, że każdy wierzchołek jest punktem na rzeczywistej powierzchni obiektu, to jego „prawdziwym" wektorem normalnym będzie wektor normalny powierzchni w tym punkcie.

W przypadku naszej kuli wektor normalny mógłby być poprowadzony od środka kuli na zewnątrz poprzez nasz punkt na powierzchni. Na rysunkach 5.30 i 5.31 przedstawimy tę ideę dla prostych dwóch wymiarów. Na rysunku 5.30 każda płaska powierzchnia posiada własny, prostopadły do niej, wektor normalny. Tę metodę wykorzystaliśmy w podanym wcześniej przykładzie programu *LITJET*, jednak już na rysunku 5.31 można zobaczyć, że każdy wektor normalny nie jest prostopadły do jakiejkolwiek linii przybliżającej wygląd kuli, ale do rzeczywistej powierzchni kuli albo *linii stycznej* (ang. *tangent line*) do tej powierzchni.

Rysunek 5.30.
Aproksymowanie za pomocą normalnych prostopadłych do każdej fasety



Linia styczna dotyka krzywej w jednym punkcie, ale jej nie przecina. W trzech wymiarach podobnie zachowuje się płaszczyzna styczna. Na rysunku 5.31 można zobaczyć kontur rzeczywistej powierzchni, jak również to, że normalna jest zawsze prostopadła do linii stycznej do powierzchni.



Rysunek 5.31. *Każda normalna jest prostopadła do rzeczywistej powierzchni kuli*

Jeżeli chodzi o kulę, to wyliczenie normalnych do jej powierzchni nie jest bardzo trudne (normalne mają dokładnie te same wartości co wektory wyprowadzone ze środka kuli). Niestety, dla innych, nietrywialnych powierzchni takie obliczenia z pewnością nie będą proste. W takich przypadkach wylicza się normalne dla wszystkich wielokątów współdzielących jeden wierzchołek, a następnie do wierzchołka przypisuje się wektor normalny będący średnią wszystkich wektorów normalnych wielokątów. W wyniku takiego działania otrzymamy ładną, gładką powierzchnię, mimo że składa się ona z pewnej liczby niewielkich płaskich segmentów.

A teraz wszystko razem

Nadszedł czas na napisanie nieco bardziej rozbudowanego programu. Zademonstrujemy w nim, jak używać normalnych do uzyskania wyglądu gładkich powierzchni, jak przesuwać źródło światła na scenie, tworzyć reflektor kierunkowy. Na koniec wskażemy wadę stosowanego przez bibliotekę OpenGL modelu oświetlenia.

Wszystkie te zadania wykonywał będzie nasz następny program przykładowy — *SPOT*. Na środku naszej przestrzeni widocznej tworzymy kulę za pomocą funkcji `glutSolidSphere`. Kulę oświetlimy za pomocą reflektora, który będzie można przesuwać po scenie. Będziemy też zmieniać „gładkość" powierzchni kuli, a także wskazywać pewne ograniczenia modelu oświetlenia stosowanego w bibliotece OpenGL.

Jak dotąd pozycję źródła światła określaliśmy za pomocą funkcji `glLight`:

```
// Tablica określająca pozycję źródła światła
GLfloat lightPos[] = { 0.0f, 150.0f, 150.0f, 1.0f };
...
...
// Ustalenie pozycji źródła światła
glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
```

W tablicy `lightPos[]` przechowywane są wartości współrzędnych *x*, *y* i *z* określających aktualną pozycję źródła światła na scenie albo kierunek, z którego światło pada na scenę. Ostatnia wartość w tablicy, w tym przypadku `1.0`, oznacza, że źródło światła rzeczywiście

znajduje się na podanej pozycji. Domyślnie, źródła światła świecą we wszystkich kierunkach jednakowo, jednak można zmienić to ustawienie i utworzyć reflektor kierunkowy.

Umieszczając wartość `0.0` na ostatniej pozycji tablicy `lightPos[]`, powodujemy, że źródło światła znajduje się w nieskończonej odległości od sceny, a wszystkie promienie jego światła docierają do sceny z zadanego kierunku. Każde źródło światła kierunkowego (tak nazywają się te źródła) jednakowo oświetla wszystkie obiekty na scenie. Oznacza to, że promienie tego światła są do siebie równoległe. Z drugiej strony promienie światła pozycjonowanego rozchodzą się od źródła we wszystkich kierunkach.

Tworzenie reflektora

Źródła światła typu reflektorów tworzy się dokładnie tak samo jak inne pozycjonowane źródła światła. Kod z listingu 5.6 przedstawia funkcję `SetupRC` pochodzącą z programu *SPOT*. Program ten umieszcza pośrodku okna niebieską kulę. Poza tym tworzy też reflektor, który możemy przesuwać nad kulą w pionie (klawisze ze strzałkami w górę i w dół) oraz w poziomie (klawisze ze strzałkami w lewo i w prawo). Poruszanie reflektora wokół kuli powoduje, że efekty odblaskowe zmieniają się, dostosowując się do nowej pozycji reflektora.

Listing 5.6. *Konfiguracja oświetlenia w programie przykładowym SPOT*

```
// Wartości i współrzędne oświetlenia
GLfloat  lightPos[] = { 0.0f, 0.0f, 75.0f, 1.0f };
GLfloat  specular[] = { 1.0f, 1.0f, 1.0f, 1.0f};
GLfloat  specref[]  = { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat  ambientLight[] = { 0.5f, 0.5f, 0.5f, 1.0f};
GLfloat  spotDir[]  = { 0.0f, 0.0f, -1.0f };

// Ta funkcja wykonuje wszystkie konieczne inicjalizacje kontekstu renderowania,
// a także konfiguruje i inicjalizuje oświetlenie sceny
void SetupRC()
  {
  glEnable(GL_DEPTH_TEST);        // Usuwanie powierzchni ukrytych
  glFrontFace(GL_CCW);            // Wielokąty z nawinięciem przeciwnym do ruchu
                                  ↳wskazówek zegara
  glEnable(GL_CULL_FACE);         // Nie będziemy prowadzić obliczeń wnętrza samolotu

  // Włączenie oświetlenia
  glEnable(GL_LIGHTING);

  // Konfiguracja i włączenie światła numer 0
  // Musimy dostarczyć nieco światła otoczenia, żeby w ogóle
  // zobaczyć jakiekolwiek obiekty
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientLight);

  // Źródło światła będzie emitować wyłącznie światło rozproszone i odbite
  glLightfv(GL_LIGHT0,GL_DIFFUSE,ambientLight);
  glLightfv(GL_LIGHT0,GL_SPECULAR,specular);
  glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
```

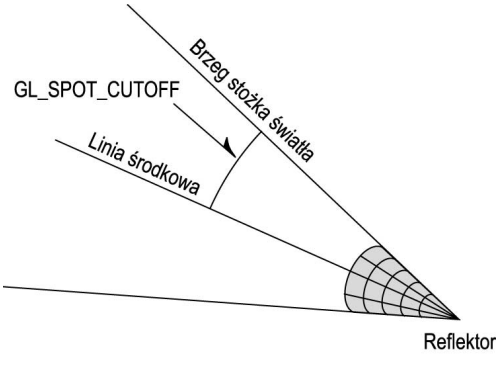
```
// Ze źródła światła tworzymy reflektor
// Kąt odcięcia ma 60 stopni
glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,60.0f);

// Nie możemy zapomnieć o włączeniu tego światła
glEnable(GL_LIGHT0);

// Włączenie śledzenia kolorów
glEnable(GL_COLOR_MATERIAL);

// Właściwości oświetlenia otoczenia i rozproszenia
// będą śledzić wartości podawane funkcji glColor
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

// Od tego momentu wszystkie materiały uzyskają połysk
// i będą doskonale odbijały światło odbite
glMaterialfv(GL_FRONT, GL_SPECULAR,specref);
glMateriali(GL_FRONT, GL_SHININESS,128);

// Czarne tło
glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
}
```

Poniższe linie kodu sprawiają, że pozycjonowane źródło światła zmienia się w reflektor:

```
// Ze źródła światła tworzymy reflektor
// Kąt odcięcia ma 60 stopni
glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,60.0f);
```

Wartość parametru `GL_SPOT_CUTOFF` określa kąt mierzony od linii środkowej stożka światła, do brzegu tego stożka. Dla normalnego światła pozycjonowanego wartość tego kąta wynosi `180°`, co oznacza, że nie jest ono ograniczane jakimkolwiek stożkiem. Jednak już dla reflektorów wartość ta wynosi od `0°` do `90°`. Reflektory generują światło wyłącznie wewnątrz zadanego stożka. Wszystko, co znajduje się poza nim, nie jest oświetlane. Na rysunku 5.32 można zobaczyć, w jaki sposób podawany tu kąt zamieniany jest na szerokość stożka światła.

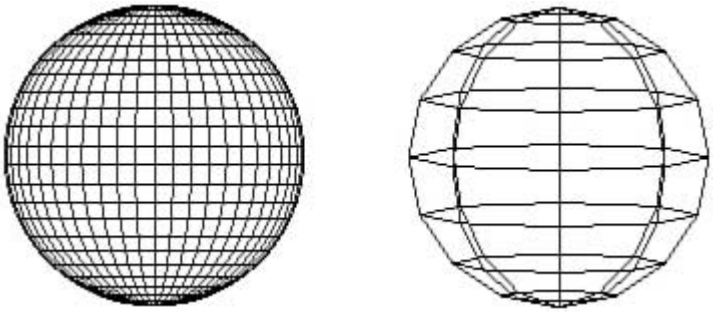
Rysunek 5.32.
Kąty w stożku światła z refletora



Rysowanie reflektorów

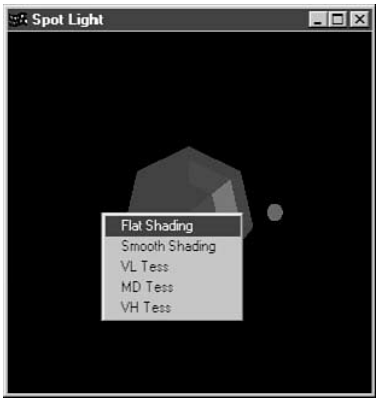
Umieszczając na scenie reflektor, musimy zaznaczyć, skąd pochodzi rzucane przez niego światło. To, że w jakimś miejscu będzie znajdować się źródło światła, nie oznacza od razu, że zobaczymy tam jasną plamę. W przykładowym programie *SPOT* w miejscu reflektora umieścimy też czerwony stożek, a w jego podstawie — żółtą kulę symbolizującą żarówkę.

W programie przygotowaliśmy menu kontekstowe, z pomocą którego będziemy demonstrować kilka rzeczy. Można w nim przełączać się między cieniowaniem płaskim i gładkim, a także wybierać modele kuli o różnych stopniach mozaikowości. *Mozaikowość* (ang. *tessellation*) oznacza zamianę siatki obiektu w siatkę o większej dokładności (składającej się z większej liczby wierzchołków). Na rysunku 5.33 można zobaczyć siatkową reprezentację kuli o wysokiej mozaikowości. Obok niej widać siatkę kuli składającą się z mniejszej liczby wierzchołków.



Rysunek 5.33. *Po lewej stronie znajduje się kula o wysokiej mozaikowości, natomiast po prawej kula składająca się z niewielkiej liczby wierzchołków*

Rysunek 5.34 przedstawia nasz przykładowy program w swojej konfiguracji początkowej, przy czym reflektor oświetlający kulę został przesunięty nieco w prawo (reflektor przesuwany jest klawiszami strzałek). Kula zbudowana jest z niewielkiej liczby płasko cieniowanych wielokątów. W systemie Windows prawym klawiszem myszy wywołujemy menu kontekstowe (w komputerach Mac ten sam efekt uzyskujemy, przytrzymując klawisz *Ctrl* i klikając myszą), z którego możemy wybrać płaski lub gładki model cieniowania, a także trzy różne poziomy mozaikowości modelu kuli. Listing 5.7 przedstawia pełny kod renderujący scenę w tym programie.



Rysunek 5.34. *Przykładowy program SPOT — niska mozaikowość, płaskie cieniowanie*

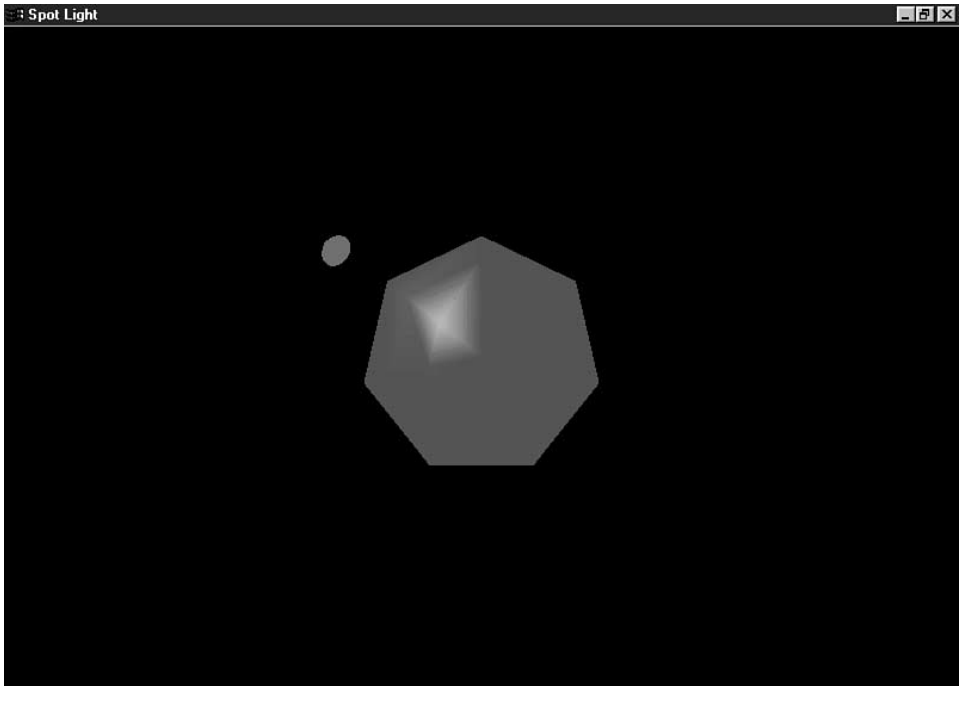
Listing 5.7. *Funkcja renderująca programu SPOT. Można tu zobaczyć, jak przesuwany jest na scenie reflektor*

```
// Wywoływana w celu przerysowania sceny
void RenderScene(void)
  {
  if(iShade == MODE_FLAT)
    glShadeModel(GL_FLAT);
  else          // iShade = MODE_SMOOTH;
    glShadeModel(GL_SMOOTH);

  // Czyszczenie okna aktualnym kolorem czyszczącym
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  // Najpierw trzeba umieścić na scenie źródło światła
  // Zapisać przekształcenie współrzędnych
  glPushMatrix();
    // Obrót układu współrzędnych
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);

    // Zdefiniowanie nowej pozycji i kierunku w obróconym układzie współrzędnych
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
    glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,spotDir);

    // Rysowanie czerwonego stożka obejmującego źródło światła
    glColor3ub(255,0,0);

    // Przesuwamy początek układu współrzędnych, umieszczając stożek w miejscu,
    // w którym znajduje się źródło światła
    glTranslatef(lightPos[0],lightPos[1],lightPos[2]);
    glutSolidCone(4.0f,6.0f,15,15);

    // Rysujemy niewielką żółtą kulę wyglądającą jak żarówka
    // Zapisanie zmiennych stanu oświetlenia
    glPushAttrib(GL_LIGHTING_BIT);

      // Wyłączenie oświetlenia i rysowanie jasnej żółtej kuli
      glDisable(GL_LIGHTING);
        glColor3ub(255,255,0);
      glutSolidSphere(3.0f, 15, 15);

    // Odtworzenie zmiennych stanu oświetlenia
    glPopAttrib();

  // Odtworzenie przekształcenia współrzędnych
  glPopMatrix();

  // Ustawienie koloru materiału i narysowanie kuli na środku ekranu
  glColor3ub(0, 0, 255);

  if(iTess == MODE_VERYLOW)
    glutSolidSphere(30.0f, 7, 7);
  else
    if(iTess == MODE_MEDIUM)
      glutSolidSphere(30.0f, 15, 15);
```

```
    else      // iTess = MODE_MEDIUM;
    glutSolidSphere(30.0f, 50, 50);

  // Wyświetlenie obrazu
  glutSwapBuffers();
  }
```

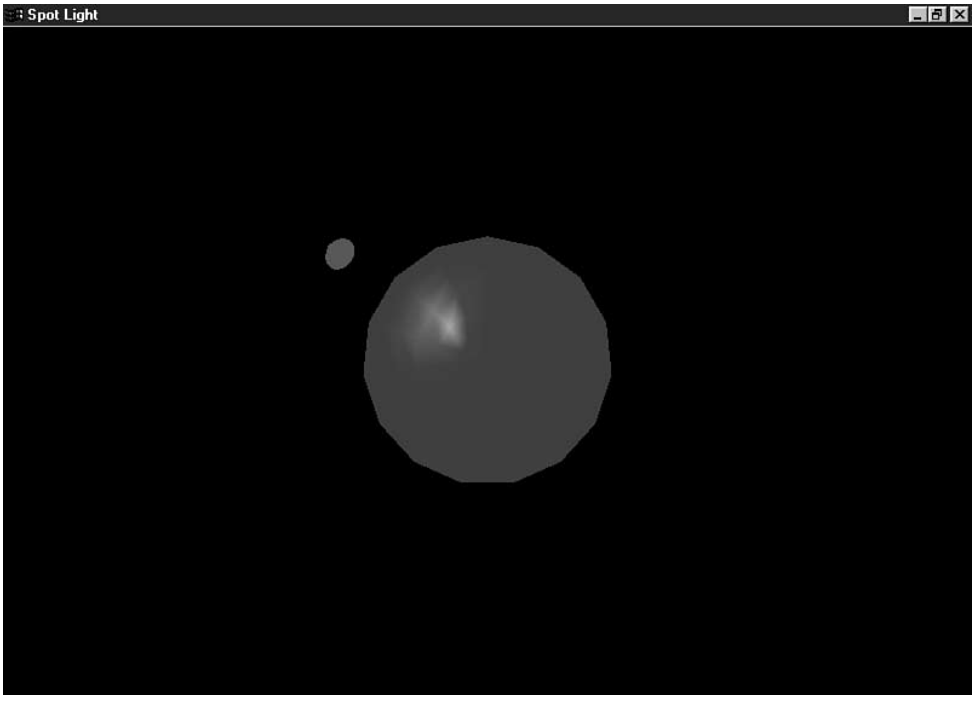
Wartości zmiennych `iTess` i `iMode` ustawiane są w funkcji obsługującej menu. Zmienne te kontrolują liczbę wielokątów, z jakich składa się kula, a także stosowany model cieniowania. Proszę zauważyć, że źródło światła umieszczane jest na scenie jeszcze przed wykonaniem jakiejkolwiek operacji renderowania. Jak wspominaliśmy w rozdziale 2., biblioteka OpenGL jest interfejsem trybu natychmiastowego, dlatego, jeżeli chcemy oświetlić jakieś obiekty na scenie, to źródła światła musimy umieścić na niej jeszcze przed narysowaniem oświetlanych obiektów.

Na rysunku 5.34 widzieliśmy, że kula jest oświetlona w niewyszukany sposób, a powierzchnie wszystkich wielokątów są wyraźnie widoczne. Włączenie trybu gładkiego cieniowania nieco poprawia sytuację, co widać na rysunku 5.35.



Rysunek 5.35. *Kula z gładkim cieniowaniem, ale niewystarczającą mozaikowością*

Jak pokazano na rysunku 5.36, podniesienie stopnia mozaikowości pozwala na znacznie lepsze odwzorowanie artefaktów w czasie przesuwania reflektora wokół kuli. To właśnie te artefakty są jedną z wad modelu oświetlenia stosowanego przez bibliotekę OpenGL. Tak naprawdę, należałoby powiedzieć, że jest to wada modelu oświetlenia opartego na

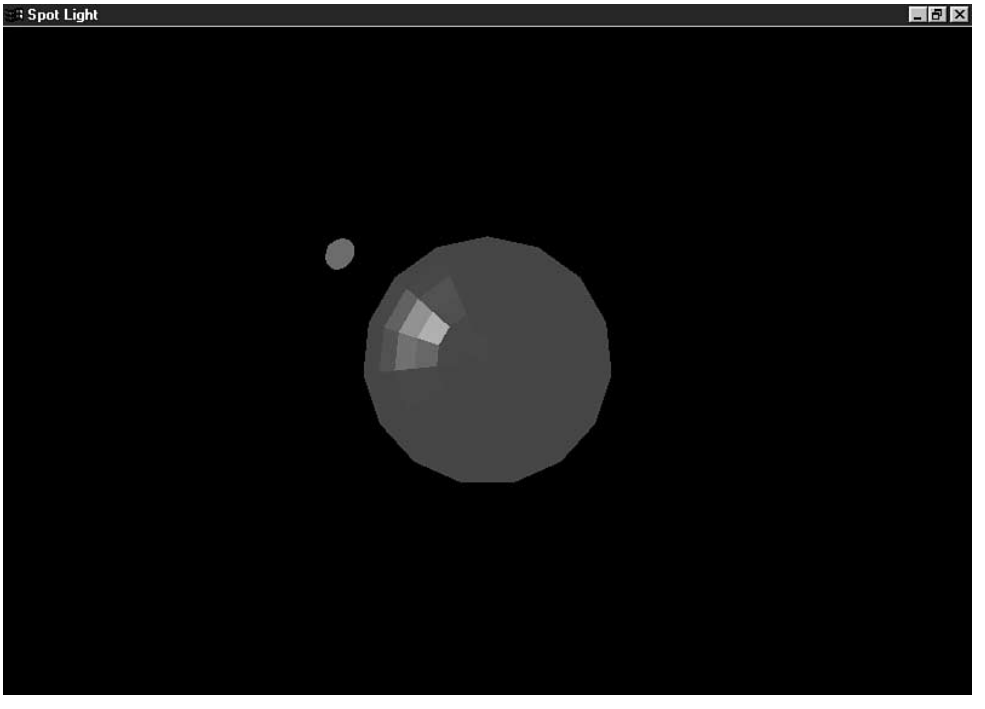


Rysunek 5.36. *Wybranie dokładniejszej siatki wielokątów oznacza lepsze efekty oświetlenia wierzchołków*

wierzchołkach (czyli niekoniecznie samej biblioteki OpenGL!). Wyliczając wartości jedynie w wierzchołkach, a później tworząc interpolacje pomiędzy nimi na powierzchni wielokątów, otrzymujemy tylko zgrubne przybliżenie rzeczywistego oświetlenia. W wielu przypadkach takie rozwiązanie jest wystarczające, ale jak mogliśmy się przekonać w programie *SPOT* — nie we wszystkich. Jeżeli jednak zastosujemy model kuli o bardzo wysokiej mozaikowości i zaczniemy poruszać wokół niej reflektorem, to zauważymy, że wada tego modelu oświetlenia zacznie znikać.

Od czasu, gdy akceleratory grafiki współpracujące z biblioteką OpenGL zaczęły również przyspieszać działania przekształceń geometrycznych i efektów oświetlenia, a procesory bardzo zyskały na mocy obliczeniowej, możemy stosować w naszych scenach obiekty o znacznie większej mozaikowości i uzyskiwać znacznie lepsze efekty świetlne.

Ostatnią rzeczą, o jakiej musimy wspomnieć przy omawianiu programu *SPOT*, będzie można zaobserwować po włączeniu średniego stopnia mozaikowości i płaskiego cieniowania. Jak pokazano na rysunku 5.37, każda faseta kuli jest jednolicie oświetlona. Każdy z wierzchołków ma ten sam kolor modulowany jedynie przez właściwości oświetlenia i wektor normalny wielokąta. W cieniowaniu płaskim każdy wielokąt otrzymuje kolor pobrany z ostatniego wierzchołka definiującego ten wielokąt. Nie następuje tu płynna interpolacja pomiędzy poszczególnymi wierzchołkami.

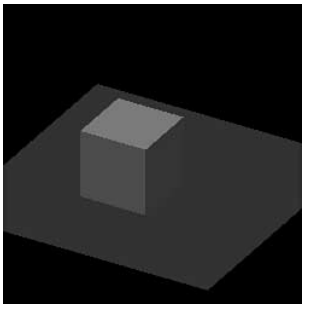


Rysunek 5.37. *Kula złożona z wielu wielokątów*

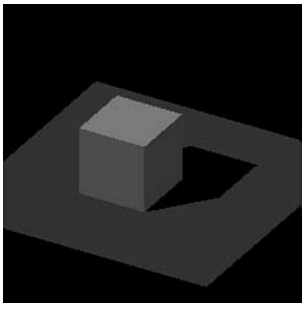
Cienie

W rozdziale dotyczącym kolorów i oświetlenia nie może zabraknąć dyskusji o cieniach. Uzupełnienie sceny o cienie obiektów może bardzo podnieść jej realizm. Na rysunkach 5.38 i 5.39 można zobaczyć dwa obrazy kolorowego sześcianu. Mimo że w obu zastosowano oświetlenie, to jednak ten z cieniem jest znacznie bardziej przekonujący.

Rysunek 5.38.
Oświetlony sześcian bez cieni



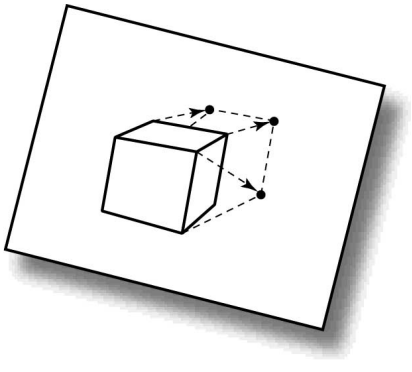
Rysunek 5.39.
Oświetlony sześcian z cieniem



Czym jest cień?

Pod względem pojęciowym rysowanie cieni powinno być prostą operacją. Cień tworzony jest wtedy, gdy jakiś obiekt zatrzymuje światło padające z pewnego źródła światła i nie pozwala mu paść na powierzchnie znajdujące się za tym obiektem. Obszar na zacienionej powierzchni jest bardzo ciemny i ma kształt obiektu rzucającego cień. Możemy więc programowo wygenerować cień, wykonując spłaszczający rzut obiektu rzucającego cień na zacienioną powierzchnię. W takim rzucie obiekt rysowany jest kolorem czarnym lub innym ciemnym, z zachowaniem pewnego stopnia przezroczystości. Istnieje wiele metod i algorytmów generowania cieni, z których część należy do bardzo złożonych. W tej książce skupiać będziemy się przede wszystkim na interfejsie programistycznym OpenGL. Mamy nadzieję, że po opanowaniu tego narzędzia kolejne pozycje sugerowane w dodatku A udostępnią każdemu ogromne możliwości jego wykorzystania w wielu różnych nowych aplikacjach. W rozdziale 18., „Tekstury głębi oraz cienie", opisujemy jedną z nowych bezpośrednich metod generowania cieni oferowaną przez bibliotekę OpenGL. Jednak w tym rozdziale skupimy się na zaprezentowaniu jednej z prostszych metod, doskonale się sprawdzających, jeżeli cień rzucany jest na płaskie powierzchnie (na przykład na ziemię). Spłaszczenie, o którym mówiliśmy wcześniej, zilustrowane zostało na rysunku 5.40.

Rysunek 5.40.
Spłaszczenie obiektu tworzące jego cień



Przy wykorzystaniu zaawansowanych manipulacji macierzowych, o których mówiliśmy już wcześniej, obiekt rzucający cień sprasowywany jest na powierzchni na którą ten cień rzuca. Poniżej postaramy się jak najbardziej uprościć ten proces.

Kod prasujący

Musimy spłaszczyć macierz rzutowania widoku modelu w taki sposób, żeby wszystkie rysowane w niej obiekty były spłaszczone do świata dwuwymiarowego. Niezależnie od ułożenia obiektu w przestrzeni zostanie on sprasowany na powierzchnię, na której położony będzie jego cień. Trzeba przy tym wziąć pod uwagę kierunek i odległość źródła światła. Kierunek źródła światła będzie wpływał na kształt i rozmiar cienia obiektu. Każdy, kto przyglądał się swojemu cieniowi wczesnym rankiem lub późnym wieczorem, wie dokładnie, jak długi i powykrzywiany może się on wydawać.

Pochodząca z biblioteki *glTools* funkcja `gltMakeShadowMatrix` przedstawiona została na listingu 5.8. Pobiera ona trzy punkty leżące na płaszczyźnie, na której ma się pojawić cień (nie mogą być ułożone na jednej linii!), pozycję źródła światła oraz wskaźnik na macierz przekształcenia wyliczonego przez tę funkcję. Nie będziemy za bardzo wgłębiać się w algebrą liniową, ale musimy powiedzieć, że ta funkcja wyznacza współczynniki równania płaszczyzny, na którą padać będzie cień i wykorzystuje je w połączeniu z informacją o pozycji źródła światła do zbudowania macierzy przekształcenia. Jeżeli tę macierz pomnożymy przez aktualną macierz model-widok, to wszystkie kolejne operacje rysowania będą odbywały się na podanej do funkcji płaszczyźnie.

Listing 5.8. *Funkcja tworząca macierz przekształcenia generującego cienie*

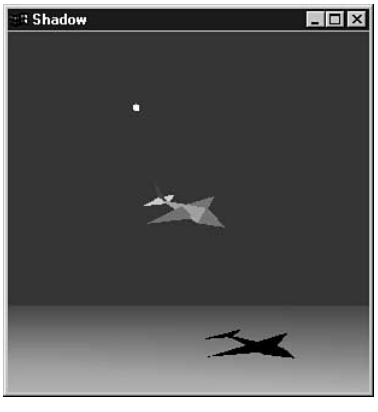
```
// Tworzy macierz rzutowania cienia na podstawie współczynników
// równania płaszczyzny i pozycji źródła światła
// Wartość zwracana zapisywana jest pod wskaźnikiem destMat
void gltMakeShadowMatrix(GLTVector3 vPoints[3], GLTVector4 vLightPos,
↳GLTMatrix destMat)
  {
  GLTVector4 vPlaneEquation;
  GLfloat dot;

  gltGetPlaneEquation(vPoints[0], vPoints[1], vPoints[2], vPlaneEquation);

  // Iloczyn skalarny płaszczyzny i pozycji źródła światła
  dot =   vPlaneEquation[0]*vLightPos[0] +
          vPlaneEquation[1]*vLightPos[1] +
          vPlaneEquation[2]*vLightPos[2] +
          vPlaneEquation[3]*vLightPos[3];

  // Teraz wykonujemy rzutowanie
  // Pierwsza kolumna
  destMat[0] = dot - vLightPos[0] * vPlaneEquation[0];
  destMat[4] = 0.0f - vLightPos[0] * vPlaneEquation[1];
  destMat[8] = 0.0f - vLightPos[0] * vPlaneEquation[2];
  destMat[12] = 0.0f - vLightPos[0] * vPlaneEquation[3];

  // Druga kolumna
  destMat[1] = 0.0f - vLightPos[1] * vPlaneEquation[0];
  destMat[5] = dot - vLightPos[1] * vPlaneEquation[1];
  destMat[9] = 0.0f - vLightPos[1] * vPlaneEquation[2];
  destMat[13] = 0.0f - vLightPos[1] * vPlaneEquation[3];
```

```
// Trzecia kolumna
destMat[2] = 0.0f - vLightPos[2] * vPlaneEquation[0];
destMat[6] = 0.0f - vLightPos[2] * vPlaneEquation[1];
destMat[10] = dot - vLightPos[2] * vPlaneEquation[2];
destMat[14] = 0.0f - vLightPos[2] * vPlaneEquation[3];

// Czwarta kolumna
destMat[3] = 0.0f - vLightPos[3] * vPlaneEquation[0];
destMat[7] = 0.0f - vLightPos[3] * vPlaneEquation[1];
destMat[11] = 0.0f - vLightPos[3] * vPlaneEquation[2];
destMat[15] = dot - vLightPos[3] * vPlaneEquation[3];
}
```

Przykład z cieniem

Aby zademonstrować zastosowanie funkcji z listingu 5.8, zawiesimy nasz samolot wysoko nad ziemią, a źródło światła nad samolotem, lekko przesunięte w lewą stronę. Klawiszami strzałek można obracać samolot w różnych kierunkach, a wtedy cień rzucany przez samolot na ziemię będzie się dostosowywał do nowej pozycji samolotu. Na rysunku 5.41 przedstawiono wynik działania programu *SHADOW*.

Rysunek 5.41. *Wynik działania programu SHADOW*



Listing 5.9 przedstawia metodę tworzenia macierzy rzutowania wykorzystaną w tym przykładzie. Proszę zwrócić uwagę, że macierz tworzona jest jeden raz w funkcji `SetupRC` i zapisywana w zmiennej globalnej.

Listing 5.9. *Tworzenie macierzy rzutowania cienia*

```
GLfloat  lightPos[] = { -75.0f, 150.0f, -50.0f, 0.0f };
...
...
// Macierz przekształcenia rzucanych cieni
GLTMatrix shadowMat;
...
...
// Ta funkcja wykonuje wszystkie konieczne inicjalizacje kontekstu renderowania,
// a także konfiguruje i inicjalizuje oświetlenie sceny
```

```
void SetupRC()
  {
  // Trzy dowolne punkty leżące na podłożu
  // (podane z nawinięciem przeciwnym do ruchu wskazówek zegara)
  GLTVector3 points[3] = {{ -30.0f, -149.0f, -20.0f },
                          { -30.0f, -149.0f, 20.0f },
                          { 40.0f, -149.0f, 20.0f }};

  glEnable(GL_DEPTH_TEST);        // Usuwanie ukrytych powierzchni
  glFrontFace(GL_CCW);            // Wielokąty z nawinięciem przeciwnym do ruchu
                                  ↳wskazówek zegara
  glEnable(GL_CULL_FACE);         // Nie będziemy prowadzić obliczeń wnętrza samolotu

  // Włączenie oświetlenia
  glEnable(GL_LIGHTING);
  ...
// Kod konfigurujący oświetlenie itd.
  ...

  // Tło jasnoniebieskie
  glClearColor(0.0f, 0.0f, 1.0f, 1.0f );

  // Wyliczenie macierzy rzutowania stosowanej do generowania cieni
  gltMakeShadowMatrix(points, lightPos, shadowMat);
  }
```

Na listingu 5.10 przedstawiono kod renderujący stosowany w przykładowym programie generowania cieni. Na początku rysowane jest podłoże, a następnie normalnie rysujemy samolot, tak jak we wszystkich pozostałych przykładach. Na koniec odtwarzamy macierz model-widok, mnożymy ją przez macierz cieni i tworzymy w ten sposób naszą macierz prasującą. Teraz trzeba jeszcze raz narysować samolot (kod został zmodyfikowany tak, żeby funkcja `DrawJet` raz rysowała w normalnych kolorach, a raz tylko w czerni). Po ponownym odtworzeniu macierzy model-widok rysujemy niewielką żółtą kulę ustawioną mniej więcej w miejscu źródła światła. Proszę zauważyć, że przed rysowaniem cienia samolotu wyłączyliśmy mechanizm testowania głębi.

Listing 5.10. *Renderowanie samolotu i jego cienia*

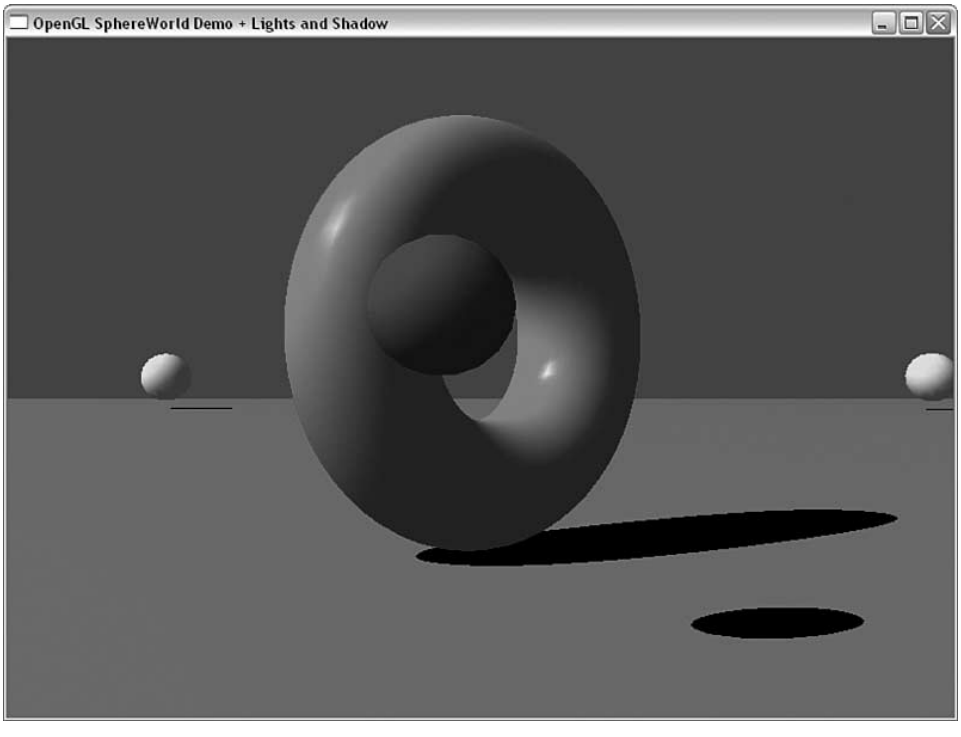
```
// Wywoływana w celu przerysowania sceny
void RenderScene(void)
  {
  // Czyszczenie okna aktualnym kolorem czyszczącym
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  // Rysowanie podłoża. Ręcznie wykonujemy cieniowanie podłoża,
  // stosując coraz ciemniejsze odcienie zielonego,
  // otrzymując w ten sposób iluzję głębi
  glBegin(GL_QUADS);
    glColor3ub(0,32,0);
    glVertex3f(400.0f, -150.0f, -200.0f);
    glVertex3f(-400.0f, -150.0f, -200.0f);
    glColor3ub(0,255,0);
    glVertex3f(-400.0f, -150.0f, 200.0f);
    glVertex3f(400.0f, -150.0f, 200.0f);
  glEnd();
```

```
// Przed wykonaniem obrotów zapisanie macierzy stanu
glPushMatrix();

// Rysowanie samolotu w nowej pozycji i umieszczenie źródła światła
// jeszcze przed jego obróceniem.
glEnable(GL_LIGHTING);
glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
glRotatef(xRot, 1.0f, 0.0f, 0.0f);
glRotatef(yRot, 0.0f, 1.0f, 0.0f);

DrawJet(FALSE);

// Odtworzenie oryginalnego stanu macierzy
glPopMatrix();

// Przygotowania do rysowania cienia i podłoża
// Najpierw wyłączamy oświetlenie i zapisujemy stan rzutowania
glDisable(GL_DEPTH_TEST);
glDisable(GL_LIGHTING);
glPushMatrix();

// Mnożenie przez macierz rzutowania cienia
glMultMatrixf((GLfloat *)shadowMat);

// Teraz obracamy samolot w nowej, spłaszczonej przestrzeni
glRotatef(xRot, 1.0f, 0.0f, 0.0f);
glRotatef(yRot, 0.0f, 1.0f, 0.0f);

// Informujemy funkcję, że rysowany będzie cień
DrawJet(TRUE);

// Odtworzenie normalnej wartości macierzy rzutowania
glPopMatrix();

// Rysowanie źródła światła
glPushMatrix();
glTranslatef(lightPos[0],lightPos[1], lightPos[2]);
glColor3ub(255,255,0);
glutSolidSphere(5.0f,10,10);
glPopMatrix();

// Odtworzenie zmiennych stanu oświetlenia
glEnable(GL_DEPTH_TEST);

// Wyświetlenie wyników
glutSwapBuffers();
}
```

Prostokąt ziemi leży na dokładnie tej samej płaszczyźnie, na której rysowane są cienie, a chcemy być pewni, że cienie jednak zostaną narysowane. Do tej pory nie zastanawialiśmy się, co się stanie, gdy na jednej płaszczyźnie będziemy rysować dwa obiekty. Opisywaliśmy jednak mechanizm testowania głębi stosowany do sprawdzania, jakie elementy mają być rysowane przed innymi. Jeżeli dwa obiekty znajdują się w tej samej odległości od obserwatora, to najczęściej pojawia się ten, który był rysowany jako ostatni. Czasami jednak, w wyniku efektu zwanego *walką w buforze z* (ang. *z-fighting*), na ekranie pojawiają się pomieszane ze sobą części obu obiektów, co powoduje zamazanie obrazu!

Ponowna wizyta w świecie kul

Ostatni przykład tego rozdziału jest zbyt duży, żeby w całości podawać jego kod źródłowy. W poprzednim rozdziale, w programie *SPHEREWORLD*, tworzyliśmy cały trójwymiarowy świat wraz z animacjami i ruchami kamery. W tym rozdziale ponownie odwiedzimy świat kul i dodamy do niego oświetlenie, a torusowi i wszystkim kulom nadamy właściwości ich materiałów. Zastosujemy też opisaną przed chwilą technikę generowania cieni! Do tego przykładu będziemy wracać od czasu do czasu, dodając do niego coraz więcej funkcji biblioteki OpenGL. Na rysunku 5.42 przedstawiony został wynik działania programu *SPHEREWORLD* uzupełnionego o funkcje opisywane w tym rozdziale.



Rysunek 5.42. *W pełni oświetlony świat kul rzucających cienie*

Podsumowanie

W niniejszym rozdziale wprowadziliśmy kilka bardziej zaawansowanych możliwości biblioteki OpenGL. Rozpoczęliśmy od dodawania do sceny kolorów i zastosowania gładkiego cieniowania. Następnie dowiedzieliśmy się, jak można utworzyć jedno lub więcej źródeł światła i zdefiniować ich charakterystyki składowych światła otoczenia, rozproszenia i odbicia. Wyjaśniliśmy też, w jaki sposób poszczególne właściwości materiałów współdziałają ze źródłami światła. Zademonstrowaliśmy kilka efektów specjalnych takich jak generowanie rozbłysków świetlnych na powierzchni obiektów i łagodzenie ostrych krawędzi występujących pomiędzy sąsiadującymi trójkątami.

Omówiliśmy również pozycjonowanie źródeł światła, a także metody tworzenia i manipulowania reflektorami. Opisana przez nas macierz wysokiego poziomu bardzo ułatwia generowanie cieni, jeżeli są one rzucane na płaską powierzchnię.

Opisy funkcji

glColor

Zastosowanie: Ustala aktualny kolor w trybie RGBA.

Plik nagłówkowy: `<gl.h>`

Rodzaje:

```
void glColor3b(GLbyte red, GLbyte green, GLbyte blue);
void glColor3d(GLdouble red, GLdouble green, GLdouble blue);
void glColor3f(GLfloat red, GLfloat green, GLfloat blue);
void glColor3i(GLint red, GLint green, GLint blue);
void glColor3s(GLshort red, GLshort green, GLshort blue);
void glColor3ub(GLubyte red, GLubyte green, GLubyte blue);
void glColor3ui(GLuint red, GLuint green, GLuint blue);
void glColor3us(GLushort red, GLushort green, GLushort blue);
void glColor4b(GLbyte red, GLbyte green, GLbyte blue, GLbyte alpha);
void glColor4d(GLdouble red, GLdouble green, GLdouble blue, GLdouble alpha);
void glColor4f(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);
void glColor4i(GLint red, GLint green, GLint blue, GLint alpha);
void glColor4s(GLshort red, GLshort green, GLshort blue, GLshort alpha);
void glColor4ub(GLubyte red, GLubyte green, GLubyte blue, GLubyte alpha);
void glColor4ui(GLuint red, GLuint green, GLuint blue, GLuint alpha);
void glColor4us(GLushort red, GLushort green, GLushort blue, GLushort alpha);
void glColor3bv(const GLbyte *v);
void glColor3dv(const GLdouble *v);
void glColor3fv(const GLfloat *v);
void glColor3iv(const GLint *v);
void glColor3sv(const GLshort *v);
void glColor3ubv(const GLubyte *v);
void glColor3uiv(const GLuint *v);
void glColor3usv(const GLushort *v);
void glColor4bv(const GLbyte *v);
void glColor4dv(const GLdouble *v);
void glColor4fv(const GLfloat *v);
void glColor4iv(const GLint *v);
void glColor4sv(const GLshort *v);
```

```
void glColor4ubv(const GLubyte *v);
void glColor4uiv(const GLuint *v);
void glColor4usv(const GLushort *v);
```

Opis: Funkcja ustala aktualny kolor, pozwalając podać osobno składowe czerwoną, zieloną i niebieską. Funkcja umożliwia też podanie wartości kanału alfa. Każda składowa reprezentowana jest liczbą z zakresu od zera (`0.0`) do pełnej intensywności koloru (`1.0`). Funkcje posiadające w nazwie przyrostek *v* pobierają wskaźnik na tablicę przechowującą wszystkie składowe koloru. Wszystkie elementy takiej tablicy muszą być tego samego typu. Jeżeli składowa alfa nie zostanie sprecyzowana, to nadawana jest jej domyślna wartość `1.0`. W funkcjach przyjmujących wartości niezmiennoprzecinkowe, wartości z zakresu od zera do maksymalnej wartości dopuszczalnej w danym typie przekształcane są w wartości zmiennoprzecinkowe z zakresu od `0.0` do `1.0`.

Parametry:

red	Składowa czerwona koloru.
green	Składowa zielona koloru.
blue	Składowa niebieska koloru.
alpha	Składowa kanału alfa koloru. Parametr ten stosowany jest wyłącznie w wersjach funkcji pobierających cztery parametry.
**v*	Wskaźnik na tablicę przechowującą wartości składowych czerwonej, zielonej, niebieskiej i alfa.

Wartość zwracana: Brak

Zobacz też: `glColorMaterial`, `glMaterial`

glColorMask

Zastosowanie: Włącza i wyłącza możliwość modyfikowania składowych koloru znajdujących się w buforze koloru.

Plik nagłówkowy: `<gl.h>`

Składnia:

```
void glColorMask(GLboolean bRed, GLboolean bGreen, GLboolean bBlue, GLboolean bAlpha);
```

Opis: Funkcja umożliwia włączanie i wyłączanie pozwolenia na modyfikowanie poszczególnych składowych koloru znajdującego się w buforze koloru (domyślnie można modyfikować wszystkie składowe). Na przykład nadanie parametrowi *bAlpha* wartości `GL_FALSE` spowoduje, że w buforze koloru nie będzie możliwe modyfikowanie kanału alfa.

Parametry:

bRed	`GLboolean` — określa, czy możliwe będą modyfikacje składowej czerwonej koloru.
bGreen	`GLboolean` — określa, czy możliwe będą modyfikacje składowej zielonej koloru.

bBlue GLboolean — określa, czy możliwe będą modyfikacje składowej niebieskiej koloru.

bAlpha GLboolean — określa, czy możliwe będą modyfikacje składowej kanału alfa.

Wartość zwracana: Brak

Zobacz też: glColor

glColorMaterial

Zastosowanie: Pozwala na ustalanie właściwości materiałów wielokątów zgodnie z kolorami nadanymi mu funkcją glColor.

Plik nagłówkowy: <gl.h>

Składnia:

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

Opis: Funkcja pozwala na ustalenie właściwości materiałów bez konieczności wywoływania funkcji glMaterial. Za pomocą tej funkcji możemy sprawić, że właściwości materiałów będą dostosowywały się do kolorów definiowanych funkcją glColor. Domyślnie mechanizm śledzenia kolorów jest wyłączony. Można go włączyć, wywołując funkcję glEnable(GL_COLOR_MATERIAL), a następnie ponownie wyłączyć funkcją glDisable(GL_COLOR_MATERIAL).

Parametry:

face GLenum — określa, czy mechanizm śledzenia kolorów będzie dotyczył przednich (GL_FRONT), tylnych (GL_BACK) lub obu (GL_FRONT_AND_BACK) stron wielokątów.

mode GLenum — definiuje, która z właściwości materiału ma być modyfikowana zgodnie z aktualnym kolorem. Można tu podać wartości GL_EMISSION, GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR albo GL_AMBIENT_AND_DIFFUSE.

Wartość zwracana: Brak

Zobacz też: glColor, glMaterial, glLight, glLightModel

glGetLight

Zastosowanie: Zwraca informacje o aktualnych parametrach źródła światła.

Plik nagłówkowy: <gl.h>

Rodzaje:

```
void glGetLightfv(GLenum light, GLenum pname, GLfloat *params);
void glGetLightiv(GLenum light, GLenum pname, GLint *params);
```

Opis: Tej funkcji można używać do sprawdzania aktualnych wartości ustawień jednego z ośmiu obsługiwanych przez bibliotekę źródeł światła. Wartości zwracane są pod adresem wskazywanym przez parametr *params*. W większości przypadków zwracana jest tablica czterech wartości opisujących składowe RGBA żądanego parametru.

Parametry:

light	GLenum — określa źródło światła, którego dotyczą żądane informacje. Mogą to być wartości z zakresu od 0 do GL_MAX_LIGHTS (specyfikacja biblioteki wymaga obsługiwania minimum ośmiu świateł). Wartości stałe wyliczane są od GL_LIGHT0 do GL_LIGHT7.
pname	GLenum — określa odpytywaną właściwość źródła światła. Możliwe jest podanie jednej z poniższych wartości: GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION, GL_SPOT_DIRECTION, GL_SPOT_EXPONENT, GL_SPOT_CUTOFF, GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION lub GL_QUADRATIC_ATTENUATION.
params	GLfloat* lub GLint* kreśla tablicę wartości całkowitych lub zmiennoprzecinkowych, do której zapisane będą wartości zwracane. Do tablicy zapisywane mogą być dane w postaci tablicy trój- lub czteroelementowej, a także w postaci pojedynczej wartości. W tabeli 5.2 podano rodzaje wartości zwracanych dla każdej właściwości.

Tabela 5.2. *Prawidłowe wartości parametrów oświetlenia dla funkcji glGetLight*

Właściwość	Znaczenie wartości zwracanej
GL_AMBIENT	Cztery składowe RGBA
GL_DIFFUSE	Cztery składowe RGBA
GL_SPECULAR	Cztery składowe RGBA
GL_POSITION	Cztery elementy określające pozycję źródła światła. Pierwsze trzy wartości definiują pozycję źródła światła. Jeżeli czwartemu elementowi nadana zostanie wartość 1.0, oznacza to, że światło rzeczywiście znajduje się na podanej pozycji. Jeżeli jednak czwartemu elementowi nadana zostanie wartość 0.0, to źródło światła jest źródłem kierunkowym, a wszystkie jego promienie padają równolegle z kierunku zdefiniowanego przez pierwsze trzy wartości
GL_SPOT_DIRECTION	Trzy elementy określające kierunek reflektora. Wektor musi być znormalizowany i definiowany we współrzędnych oka
GL_SPOT_EXPONENT	Pojedyncza wartość reprezentująca wykładnik reflektora
GL_SPOT_CUTOFF	Pojedyncza wartość reprezentująca kąt odcięcia reflektora
GL_CONSTANT_ATTENUATION	Pojedyncza wartość reprezentująca stałą tłumienia światła
GL_LINEAR_ATTENUATION	Pojedyncza wartość reprezentująca liniowe tłumienie światła
GL_QUADRATIC_ATTENUATION	Pojedyncza wartość reprezentująca kwadratowe tłumienie światła

Wartość zwracana: Brak

Zobacz też: glLight

glGetMaterial

Zastosowanie: Zwraca aktualne ustawienia właściwości materiału.

Plik nagłówkowy: <gl.h>

Rodzaje:

```
void glGetMaterialfv(GLenum face, GLenum pname, GLfloat *params);
void glGetMaterialiv(GLenum face, GLenum pname, GLint *params);
```

Opis: Funkcja pozwala na sprawdzenie właściwości materiału przedniej lub tylnej strony wielokąta. Wartości zwracane zapisywane są pod adresem pamięci wskazywanym przez parametr *params*. W większości przypadków zwracana jest tablica czterech wartości RGBA opisujących żądaną właściwość.

Parametry:

face GLenum — określa, czy sprawdzane są wartości właściwości przednich (GL_FRONT) lub tylnych (GL_BACK) stron wielokątów.

pname GLenum — definiuje, która z właściwości materiału jest sprawdzana. Można tu podać wartości GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS albo GL_COLOR_INDEXES.

params GLint* lub GLfloat* skazuje na tablicę wartości całkowitych lub zmiennoprzecinkowych, do której będą zapisywane wartości zwracane. W przypadku zapytania o właściwości GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR i GL_EMISSION zwracana jest czteroelementowa tablica zawierająca wartości RGBA opisujące właściwość. Dla zapytania o właściwość GL_SHININESS zwracana jest pojedyncza wartość opisująca wykładnik odblasku materiału. Przy zapytaniu o właściwość GL_COLOR_INDEXES zwracana jest tablica trójelementowa opisująca składowe światła otoczenia, rozproszonego i obitego, zapisane w postaci indeksów kolorów. Parametr GL_COLOR_INDEXES stosowany jest wyłącznie wtedy, gdy stosowane jest oświetlenie kolorami indeksowanymi.

Wartość zwracana: Brak

Zobacz też: glMaterial

glLight

Zastosowanie: Ustala parametry jednego z dostępnych źródeł światła.

Plik nagłówkowy: <gl.h>

Rodzaje:

```
void glLightf(GLenum light, GLenum pname, GLfloat param);
void glLighti(GLenum light, GLenum pname, GLint param);
void glLightfv(GLenum light, GLenum pname, const GLfloat *params);
void glLightiv(GLenum light, GLenum pname, const GLint *params);
```

Opis: Funkcja używana jest do ustalania parametrów jednego z ośmiu obsługiwanych źródeł światła. Dwie pierwsze wersje funkcji pobierają tylko jeden parametr i pozwalają na ustawienie wartości następujących właściwości: GL_SPOT_EXPONENT, GL_SPOT_CUTOFF, GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION i GL_QUADRATIC_ATTENUATION. Dwie pozostałe wersje funkcji stosowane są do ustalania właściwości opisywanych tablicą kilku wartości. Takimi właściwościami są GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR,

GL_POSITION i GL_SPOT_DIRECTION. Tablicowych wersji funkcji glLight można używać też do ustalania wartości właściwości opisywanych pojedynczą wartością; wystarczy w tablicy *params umieścić tylko jeden parametr.

Parametry:

light	GLenum: Określa numer źródła światła poddawanego modyfikacjom. Można podawać tu wartości od 0 do GL_MAX_LIGHTS (przynajmniej 8). Zdefiniowane są stałe wartości wyliczeniowe od GL_LIGHT0 do GL_LIGHT7.
pname	GLenum — określa parametr oświetlenia ustawiany w danym wywołaniu funkcji. W tabeli 5.2 podana została pełna lista parametrów wraz z ich znaczeniami.
param	GLint lub GLfloat — podaje wartość dla parametru oświetlenia definiowanego pojedynczą wartością. Takimi parametrami są GL_SPOT_EXPONENT, GL_SPOT_CUTOFF, GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION i GL_QUADRATIC_ATTENUATION. Parametry te mają znaczenie tylko przy definiowaniu reflektorów.
params	GLint* lub GLfloat* — określa tablicę zawierającą wartości całkowicie opisujące ustawiany parametr. W tabeli 5.2 można znaleźć pełną listę takich parametrów wraz z ich znaczeniami.

Wartość zwracana: Brak

Zobacz też: glGetLight

glLightModel

Zastosowanie: Ustala parametry modelu oświetlenia stosowanego przez bibliotekę OpenGL.

Plik nagłówkowy: <gl.h>

Rodzaje:

```
void glLightModelf(GLenum pname, GLfloat param);
void glLightModeli(GLenum pname, GLint param);
void glLightModelfv(GLenum pname, const GLfloat *params);
void glLightModeliv(GLenum pname, const GLint *params);
```

Opis: Tę funkcję stosuje się do ustalania parametrów modelu oświetlenia stosowanego przez bibliotekę OpenGL. Możliwe jest ustalenie wartości dowolnego z czterech parametrów modelu. Parametr GL_LIGHT_MODEL_AMBIENT stosowany jest do ustalania domyślnego oświetlenia otoczenia. Domyślnie, parametr ten ma przypisaną wartość RGBA (0.2, 0.2, 0.2, 1.0). Do ustawiania tego parametru stosowane mogą być tylko dwie ostatnie wersje tej funkcji, ponieważ pobierają one wskaźnik na tablicę opisującą wszystkie wartości RGBA.

Parametr GL_LIGHT_MODEL_TWO_SIDE pozwala określić, czy oświetlane będą obie strony wielokątów. Domyślnie oświetlane są wyłącznie przednie strony wielokątów, przy czym w czasie wyliczania oświetlenia stosowane są właściwości materiałów zdefiniowane funkcją glMaterial. Określenie

parametru GL_LIGHT_MODEL_LOCAL_VIEWER powoduje zmodyfikowanie obliczeń kątów odbicia światła odbitego w zależności od tego, czy obserwator patrzy w dół wzdłuż ujemnych wartości osi *z* czy też z początku układu współrzędnych oka. W końcu parametr GL_LIGHT_MODEL_COLOR_CONTROL może być stosowany do kontrolowania, czy oświetlenie będzie tworzyło dodatkowy kolor (tekstury będą oświetlane światłem odbitym) czy też wszystkie trzy składowe światła będą łączone w parametrze GL_SINGLE_COLOR.

Parametry:

pname — GLenum — określa modyfikowany parametr oświetlenia. Przyjmowane są wartości GL_LIGHT_MODEL_AMBIENT, GL_LIGHT_MODEL_LOCAL_VIEWER, GL_LIGHT_MODEL_TWO_SIDE, GL_LIGHT_MODEL_COLOR_CONTROL.

param — GLint lub GLfloat — dla parametru GL_LIGHT_MODEL_LOCAL_VIEWER wartość 0.0 oznacza, że kąty padania światła odbitego muszą brać pod uwagę kierunek patrzenia obserwatora równoległy do ujemnych wartości osi *z*. Każda inna wartość oznacza, że obserwator znajduje się w początku układu współrzędnych oka. Dla parametru GL_LIGHT_MODEL_TWO_SIDE wartość 0.0 oznacza, że wyliczane będzie oświetlenie wyłącznie przednich stron wielokątów. Każda inna wartość oznacza, że obliczenia prowadzone będą dla przednich i tylnych stron wielokątów. Parametr ten nie ma żadnego wpływu na punkty, linie i bitmapy. Dla parametru GL_LIGHT_MODEL_COLOR_CONTROL można podać wartości GL_SEPARATE_SPECULAR_COLOR lub GL_SINGLE_COLOR.

params — GLint* lub GLfloat* — dla parametrów GL_LIGHT_MODEL_AMBIENT lub GL_LIGHT_MODEL_LOCAL_VIEWER, podać należy wskaźnik na tablicę wartości całkowitych lub zmiennoprzecinkowych. Z podanej tablicy wykorzystana będzie tylko pierwsza wartość. W przypadku parametru GL_LIGHT_MODEL_AMBIENT z tablicy pobierane są cztery wartości opisujące składowe RGBA światła otoczenia.

Wartość zwracana: Brak

Zobacz też: glLight, glMaterial

glMaterial

Zastosowanie: Ustala właściwości materiału stosowane w modelu oświetlenia.

Plik nagłówkowy: <gl.h>

Rodzaje:

```
void glMaterialf(GLenum face, GLenum pname, GLfloat param);
void glMateriali(GLenum face, GLenum pname, GLint param);
void glMaterialfv(GLenum face, GLenum pname, const GLfloat *params);
void glMaterialiv(GLenum face, GLenum pname, const GLint *params);
```

Opis: Tej funkcji używa się do ustalenia parametrów odblaskowych materiału pokrywającego wielokąty. Właściwości GL_AMBIENT, GL_DIFFUSE i GL_SPECULAR określają sposób, w jaki materiał odbija padające na niego światło. Właściwość GL_EMISSION stosowana jest dla materiałów sprawiających wrażenie emitowania własnego światła. Wartość przypisywana właściwości GL_SHININESS musi

znajdować się w zakresie od 0 do 128, przy czym wyższe wartości tworzą większe efekty rozbłysków na powierzchni materiału. Właściwość GL_COLOR_INDEXES stosowana jest do ustalania właściwości odblaskowych materiałów w trybie kolorów indeksowanych.

Parametry:

face — GLenum — określa, czy właściwości materiałów będą ustalane dla przednich (GL_FRONT), tylnych (GL_BACK) czy też obu (GL_FRONT_AND_BACK) stron wielokątów.

pname — GLenum — w pierwszych dwóch wersjach funkcji określa poddawaną modyfikacjom właściwość przyjmującą pojedynczą wartość. Aktualnie jedyną tego rodzaju właściwością jest GL_SHININESS. Pozostałe dwie wersje funkcji przyjmują też tablice wartości przypisywanych właściwościom. Za ich pomocą można modyfikować właściwości GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS, GL_AMBIENT_AND_DIFFUSE oraz GL_COLOR_INDEXES.

param — GLint lub GLfloat kreśla wartość przypisywaną do właściwości zdefiniowanej parametrem *pname* (aktualnie tylko GL_SHININESS).

params — GLint* lub GLfloat* kreśla tablicę wartości całkowitych lub zmiennoprzecinkowych zawierającą wartości przypisywane ustawianej właściwości.

Wartość zwracana: Brak

Zobacz też: glGetMaterial, glColorMaterial, glLight, glLightModel

glNormal

Zastosowanie: Definiuje normalną dla następnego definiowanego wierzchołka lub zbioru wierzchołków.

Plik nagłówkowy: <gl.h>

Rodzaje:

```
void glNormal3b(GLbyte nx, GLbyte ny, GLbyte nz);
void glNormal3d(GLdouble nx, GLfouble ny, GLdouble nz);
void glNormal3f(GLfloat nx, GLfloat ny, GLfloat nz);
void glNormal3i(GLint nx, GLint ny, GLint nz);
void glNormal3s(GLshort nx, GLshort ny, GLshort nz);
void glNormal3bv(const GLbyte *v);
void glNormal3dv(const GLdouble *v);
void glNormal3fv(const GLfloat *v);
void glNormal3iv(const GLint *v);
void glNormal3sv(const GLshort *v);
```

Opis: Wektor normalny określa kierunek prostopadły do górnej powierzchni wielokąta. Funkcja stosowana jest w obliczeniach związanych z oświetleniem i cieniowaniem. Podanie wektora jednostkowego (o długości 1) bardzo

podnosi prędkość renderowania. Biblioteka OpenGL automatycznie może zamieniać wszystkie wektory normalne w wektory jednostkowe; wystarczy wywołać funkcję `glEnable(GL_NORMALIZE);`.

Parametry:

nx	Określa wartość *x* w wektorze normalnym.
ny	Określa wartość *y* w wektorze normalnym.
nz	Określa wartość *z* w wektorze normalnym.
v	Określa tablicę trójelementową zawierającą wartości *x*, *y* i *z* wektora normalnego.

Wartość zwracana: Brak

Zobacz też: `glTexCoord, glVertex`

glShadeModel

Zastosowanie: Ustala, czy stosowany będzie model cieniowania płaskiego czy też gładkiego.

Plik nagłówkowy: `<gl.h>`

Składnia:

```
void glShadeModel(GLenum mode);
```

Opis: Obiekty podstawowe biblioteki OpenGL zawsze są cieniowane, jednak model cieniowania może być płaski (`GL_FLAT`) lub gładki (`GL_SMOOTH`). W najprostszym scenariuszu przed narysowaniem wielokąta jego kolor musi zostać zdefiniowany wywołaniem funkcji `glColor`. Taki obiekt podstawowy zawsze wypełniany jest kolorem jednolitym, niezależnie od modelu cieniowania. Jeżeli dla każdego wierzchołka zdefiniowany zostanie inny kolor, to rysunek takiego wielokąta będzie wyglądał inaczej w każdym modelu cieniowania. W cieniowaniu gładkim wnętrze wielokąta wypełniane jest interpolacją kolorów wszystkich jego wierzchołków. Oznacza to, że kolor powierzchni wielokąta będzie zmieniał się na odcinku łączącym dwa wierzchołki. W czasie interpolowania, kolory pobierane będą z sześcianu kolorów z odcinka łączącego punkty odpowiadające kolorom dwóch wierzchołków. Jeżeli włączone będzie też oświetlenie, biblioteka OpenGL wykonywać będzie dodatkowe obliczenia mające na celu ustalenie właściwej wartości koloru każdego wierzchołka. W cieniowaniu płaskim kolor nadany ostatniemu wierzchołkowi stosowany jest do wypełnienia całego obiektu podstawowego. Jedynym wyjątkiem od tej zasady są obiekty podstawowe `GL_POLYGON`, które wypełniane są kolorem pierwszego wierzchołka.

Parametry:

mode	`GLenum` — określa stosowany model cieniowania. Można podać tu wartości `GL_FLAT` (cieniowanie płaskie) lub `GL_SMOOTH` (cieniowanie gładkie). Domyślnie stosowane jest cieniowanie gładkie.

Wartość zwracana: Brak

Zobacz też: `glColor`, `glLight`, `glLightModel`