OpenGL®

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

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

Jak sprawnie tworzyć podstawowe obiekty, oświetlać je i cieniować?

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

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

OpenGL®
KSIĘGA EKSPERTA

Wydanie 5

Helion

OpenGL. Księga eksperta. Wydanie V

Autorzy: Richard S. Wright, Nicholas Haemel, Graham Sellers, Benjamin Lipchak
Tłumaczenie: Łukasz Piwko
ISBN: 978-83-246-2976-3
Tytuł oryginału: OpenGL SuperBible: Comprehensive Tutorial and Reference (5th Edition)
Format: 172×245, stron: 688

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

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

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

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

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

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

Spis treści

Rozdział 6.
Myślenie niekonwencjonalne — programy do cieniowania

Autor: Richard S. Wright, Jr

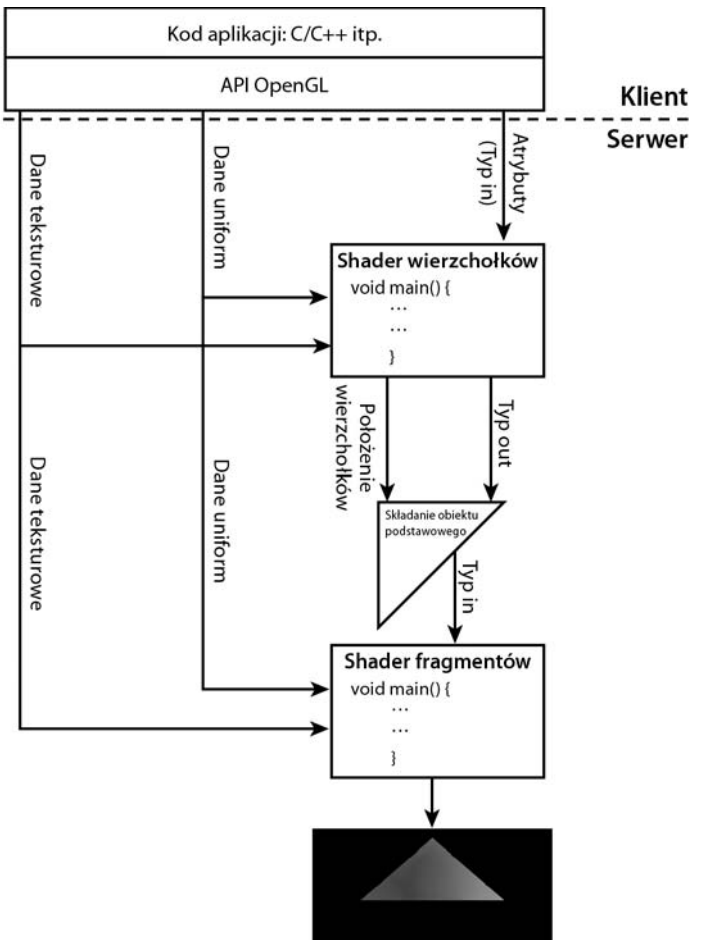
Po raz pierwszy ze shaderami spotkaliśmy się w rozdziale 3., w którym opisane zostały podstawowe techniki renderowania. Osoby, które ten rozdział pominęły, aby od razu przejść do pisania shaderów, powinny w razie potrzeby do niego wrócić i dokładnie zapoznać się z atrybutami i danymi typu `uniform` oraz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

GLSL 101

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

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

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



Rysunek 6.1. Schemat architektury shadera

Zmienne i typy danych

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

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

Typy wektorowe

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

Tabela 6.1. Wektorowe typy danych języka GLSL

Typ	Opis
`vec2, vec3, vec4`	Dwu-, trzy- i czteroskładnikowy wektor wartości zmiennoprzecinkowych
`ivec2, ivec3, ivec4`	Dwu-, trzy- i czteroskładnikowy wektor wartości całkowitoliczbowych
`uvec2, uvec3, uvec4`	Dwu-, trzy- i czteroskładnikowy wektor wartości całkowitoliczbowych bez znaku
`bvec2, bvec3, bvec4`	Dwu-, trzy- i czteroskładnikowy wektor wartości logicznych

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

```
vec4 vVertexPos;
```

Wektor można także zainicjalizować przy użyciu konstruktora:

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

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

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

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

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

Podczas pracy z kolorami korzystamy z identyfikatorów `rgba`.

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

Identyfikatorów `stpq` używany w pracy ze współrzędnymi tekstur.

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

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

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

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

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

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

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

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

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

zostałaby wykonana znacznie szybciej, gdyby użyto rodzimej notacji wektorowej:

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

Typy macierzowe

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

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

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

Aby pobrać ostatnią kolumnę macierzy:

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

Jeszcze precyzyjniejsze zapytanie:

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

Tabela 6.2. Typy macierzowe języka GLSL

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

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

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

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

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

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

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

Kwalifikatory zmiennych

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

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

Tabela 6.3. Kwalifikatory zmiennych

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

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

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

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

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

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

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

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

Prawdziwy shader

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

Listing 6.1. *Shader wierzchołków ShadedIdentity*

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

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

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

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

Listing 6.2. *Shader fragmentów ShadedIdentity*

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

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

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

Wersje języka GLSL

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

```
#version 330
```

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

```
#version 400
```

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

Deklaracje atrybutów

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

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

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

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

Deklarowanie danych wyjściowych

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

```
out vec4 vVaryingColor;
```

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

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

Przetwarzanie wierzchołków

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

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

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

Przetwarzanie fragmentów

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

```
out vec4 vFragColor;
```

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

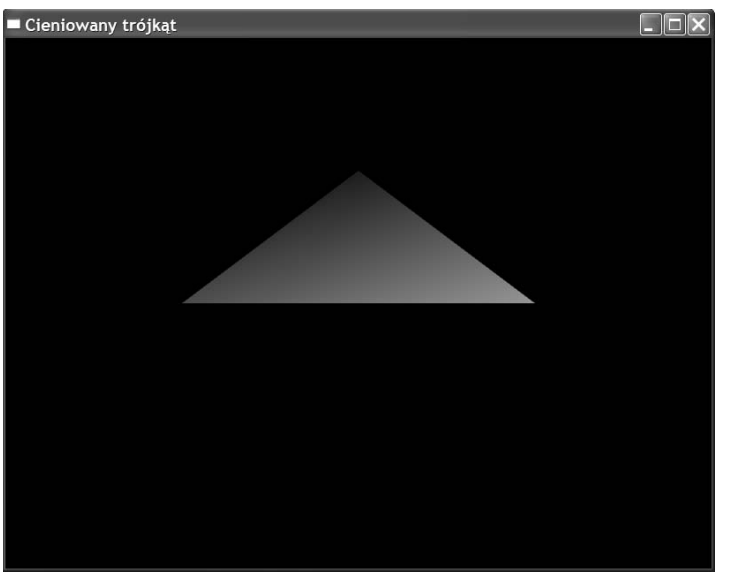
Do shadera fragmentów zostaje przesłana płynnie interpolowana wartość koloru pochodząca z shadera wierzchołków. Deklaruje się ją jako zmienną z kwalifikatorem `in`:

```
in vec4 vVaryingColor;
```

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

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

Efekt działania tego shadera przedstawia rysunek 6.2.



Rysunek 6.2. Wynik działania programu ShadedTriangle

Kompilowanie, wiązanie i konsolidowanie

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

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

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

Listing 6.3. *Funkcja gltLoadShaderPairWithAttributes*

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

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

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

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

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

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

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

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

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

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

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

    // Próba konsolidacji
    glLinkProgram(hReturn);

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

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

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

Określanie atrybutów

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

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

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

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

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

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

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

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

Pobieranie kodu źródłowego

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

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

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

```
GLchar *fsStringPtr[1];

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

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

Kompilowanie shaderów

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

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

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

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

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

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

Dołączanie i wiązanie

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

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

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

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

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

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

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

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

Konsolidacja shaderów

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

```
glLinkProgram(hReturn);

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

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

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

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

Praktyczne wykorzystanie shadera

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

```
glUseProgram(myShaderProgram);
```

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

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

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

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

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

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

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

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

Wynik tych wszystkich naszych działań przedstawia rysunek 6.2.

Wierzchołek prowokujący

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

```
out vec4 vVaryingColor;
```

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

```
flat out vec4 vFlatColor;
```

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

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

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

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

Dane uniform shadera

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

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

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

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

Znajdowanie danych uniform

Po skompilowaniu i skonsolidowaniu shadera musimy w nim „znaleźć" lokalizację naszej zmiennej uniform. Służy do tego funkcja glGetUniformLocation.

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

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

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

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

Zmienne uniform skalarne i wektorowe

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

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

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

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

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

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

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

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

Tablice uniform

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

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

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

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

```
uniform vec4 vColor;
```

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

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

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

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

Gdybyśmy jednak mieli w shaderze taką tablicę wartości kolorów:

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

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

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

Najprostsze ustawienie jednej zmiennoprzecinkowej wartości `uniform` może wyglądać tak:

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

Macierze uniform

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

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

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

Płaski shader

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

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

Listing 6.4. *Shader wierzchołków FlatShader*

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

// Macierz przekształcenia
uniform mat4   mvpMatrix;

// Dane wejściowe wierzchołków
in vec4 vVertex;

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

Listing 6.5. *Shader fragmentów FlatShader*

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

// Tworzenie jednolitej geometrii
uniform vec4 vColorValue;

// Wyjście koloru fragmentu
out vec4 vFragColor;


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

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

```
uniform mat4    mvpMatrix;
```

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

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

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

```
uniform vec4 vColorValue;
```

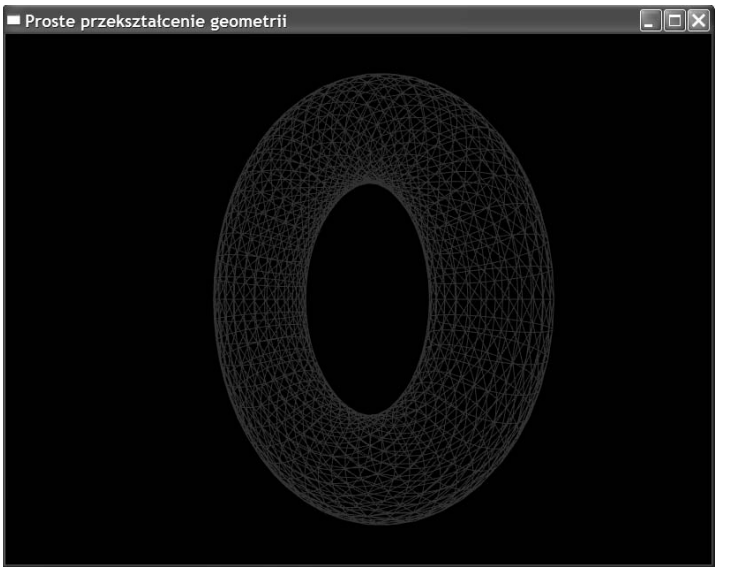
Po stronie klienckiej program wczytuje oba pliki shaderów i w funkcji `SetupRC` tworzy wskaźniki do obu zmiennych `uniform`.

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

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

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

Listing 6.6. *Praktyczne zastosowanie utworzonego płaskiego shadera*

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

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

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

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

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



Rysunek 6.3. Okno programu FlatShader

```
modelViewMatrix.PopMatrix();


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

Funkcje standardowe

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

Funkcje trygonometryczne

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

Tabela 6.4. Funkcje trygonometryczne

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

Funkcje wykładnicze

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

Tabela 6.5. Funkcje wykładnicze

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

Funkcje geometryczne

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

Tabela 6.6. Funkcje geometryczne

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

Funkcje macierzowe

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

Funkcje porównywania wektorów

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

Inne często używane funkcje

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

Tabela 6.7. Funkcje macierzowe

Funkcja	Opis
`mat matrixCompMult(mat x, mat y)`	Mnoży dwie macierze składnik po składniku. To nie jest to samo, co mnożenie macierzy w algebrze liniowej.

`mat2 outerProduct(vec2 c, vec2 r)`
`mat3 outerProduct(vec3 c, vec3 r)`
`mat4 outerProduct(vec4 c, vec4 r)`
`mat2x3 outerProduct(vec3 c, vec2 r)`
`mat3x2 outerProduct(vec2 c, vec3 r)`
`mat2x4 outerProduct(vec4 c, vec2 r)`
`mat4x2 outerProduct(vec2 c, vec4 r)`
`mat3x4 outerProduct(vec4 c, vec3 r)`
`mat4x3 outerProduct(vec3 c, vec4 r)`
	Zwraca macierz będącą iloczynem zewnętrznym dwóch podanych wektorów.

`mat2 transpose(mat2 m)`
`mat3 transpose(mat3 m)`
`mat4 transpose(mat4 m)`
`mat2x3 transpose(mat3x2 m)`
`mat3x2 transpose(mat2x3 m)`
`mat2x4 transpose(mat4x2 m)`
`mat4x2 transpose(mat2x4 m)`
`mat3x4 transpose(mat4x3 m)`
`mat4x3 transpose(mat3x4 m)`
	Transponuje podaną macierz

`float determinant(mat2 m)`
`float determinant(mat3 m)`
`float determinant(mat4 m)`
	Zwraca wyznacznik podanej macierzy

`mat2 inverse(mat2 m)`
`mat3 inverse(mat3 m)`
`mat4 inverse(mat4 m)`
	Zwraca odwróconą wersję podanej macierzy

Tabela 6.8. Funkcje porównywania wektorów

Funkcja	Opis

`bvec lessThan(vec x, vec y)`
`bvec lessThan(ivec x, ivec y)`
`bvec lessThan(uvec x, uvec y)`
	Zwraca wynik porównywania `x < y` każdej pary składników

Tabela 6.8. Funkcje porównywania wektorów — *ciąg dalszy*

Funkcja	Opis

`bvec lessThanEqual(vec x, vec y)`
`bvec lessThanEqual(ivec x, ivec y)`
`bvec lessThanEqual(uvec x, uvec y)`
	Zwraca wynik porównywania `x <= y` każdej pary składników

`bvec greaterThan(vec x, vec y)`
`bvec greaterThan(ivec x, ivec y)`
`bvec greaterThan(uvec x, uvec y)`
	Zwraca wynik porównywania `x > y` każdej pary składników

`bvec greaterThanEqual(vec x, vec y)`
`bvec greaterThanEqual(ivec x, ivec y)`
`bvec greaterThanEqual(uvec x, uvec y)`
	Zwraca wynik porównywania `x >= y` każdej pary składników

`bvec equal(vec x, vec y)`
`bvec equal(ivec x, ivec y)`
`bvec equal(uvec x, uvec y)`
`bvec equal(bvec x, bvec y)`
	Zwraca wynik porównywania `x == y` każdej pary składników

`bvec notEqual(vec x, vec y)`
`bvec notEqual(ivec x, ivec y)`
`bvec notEqual(uvec x, uvec y)`
`bvec notEqual(bvec x, bvec y)`
	Zwraca wynik porównywania `x != y` każdej pary składników

`bool any(bvec x)`	Zwraca wartość `true`, jeśli którykolwiek składnik `x` ma wartość `true`
`bool all(bvec x)`	Zwraca wartość `true`, jeśli wszystkie składniki `x` mają wartość `true`
`bvec not(bvec x)`	Zwraca dopełnienie `x` dla każdego komponentu

Tabela 6.9. Inne często używane funkcje

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

Tabela 6.9. Inne często używane funkcje — *ciąg dalszy*

Funkcja	Opis
`anyFloat round(anyFloat x)`	Zwraca wartość całkowitą najbliższą wartości `x`. Ułamek 0,5 może zostać zaokrąglony w obie strony, w zależności od implementacji
`anyFloat roundEven(anyFloat x)`	Zwraca wartość całkowitą najbliższą wartości `x`. Ułamek 0,5 jest zaokrąglany do najbliższej parzystej liczby całkowitej
`anyFloat ceil(anyFloat x)`	Zwraca wartość najbliższej liczby całkowitej większej od `x`
`anyFloat fract(anyFloat x)`	Zwraca część ułamkową wartości `x`
`anyFloat mod(anyFloat x, float y)` `anyFloat mod(anyFloat x, anyFloat y)`	Zwraca wartość bezwzględną wyniku działania `x mod y`
`anyFloat modf(anyFloat x, out anyFloat i`	Zwraca część ułamkową wartości `x` zapisaną w `i`

`anyFloat min(anyFloat x, anyFloat y)`
`anyFloat min(anyFloat x, float y)`
`anyInt min(anyInt x, anyInt y)`
`anyInt min(anyInt x, int y)`
`anyUInt min(anyUInt x, anyUInt y)`
`anyUint min(anyUInt x, uint y)`
	Zwraca mniejszą spośród wartości `x` i `y`

`anyFloat max(anyFloat x, anyFloat y)`
`anyFloat max(anyFloat x, float y)`
`anyInt max(anyInt x, anyInt y)`
`anyInt max(anyInt x, int y)`
`anyUInt max(anyUInt x, anyUInt y)`
`anyUint max(anyUInt x, uint y)`
	Zwraca większą spośród wartości `x` i `y`

`anyFloat clamp(anyFloat x,`
`anyFloat minVal,`
`anyFloat maxVal)`
`anyFloat clamp(anyFloat x,`
`float minVal,`
`float maxVal);`
`anyInt clamp(anyInt x,`
`anyInt minVal,`
`anyInt maxVal)`
`anyInt clamp(anyInt x,`
`int minVal,`
`int maxVal)`
	Zwraca wartość `x` przyciętą do przedziału `minVal-maxVal`

Tabela 6.9. Inne często używane funkcje — *ciąg dalszy*

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

Symulowanie światła

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

Światło rozproszone

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

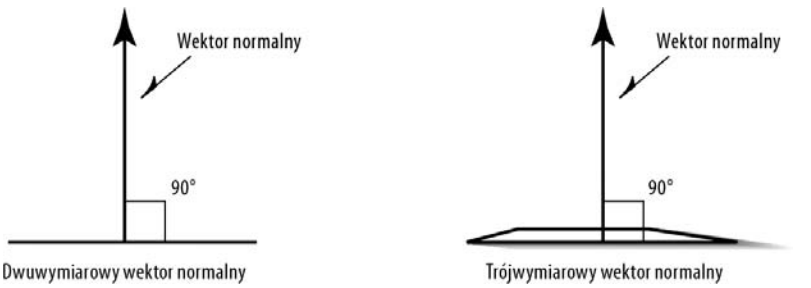
Do określenia natężenia światła w danym wierzchołku potrzebne są dwa wektory. Jeden z nich określa kierunek w stronę źródła światła. W niektórych technikach oświetleniowych dostarczany jest tylko wektor skierowany w stronę źródła światła. Takie światło nazywamy **kierunkowym** (ang. *directional*), ponieważ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

Normalne do powierzchni

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

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

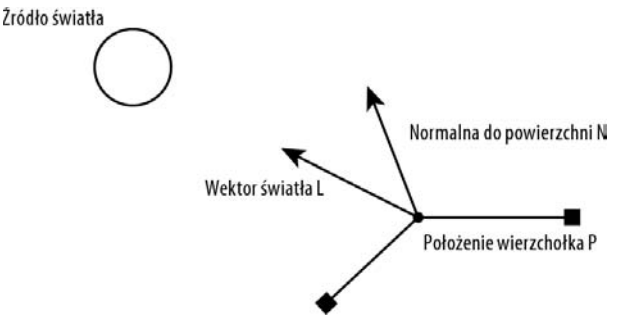
Rysunek 6.4.
Dwu- i trójwymiarowy wektor normalny



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

Oświetlanie wierzchołków

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



Rysunek 6.5. Podstawowe wektory światła rozproszonego

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

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

Shader światła rozproszonego

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

Listing 6.7. *Shader wierzchołków światła rozproszonego*

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

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

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

// Kolor
smooth out vec4 vVaryingColor;
```

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

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

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

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

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

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

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

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

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

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

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

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

```
smooth out vec4 vVaryingColor;
```

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

```
vFragColor = vVaryingColor;
```

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

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

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

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

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

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

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

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

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

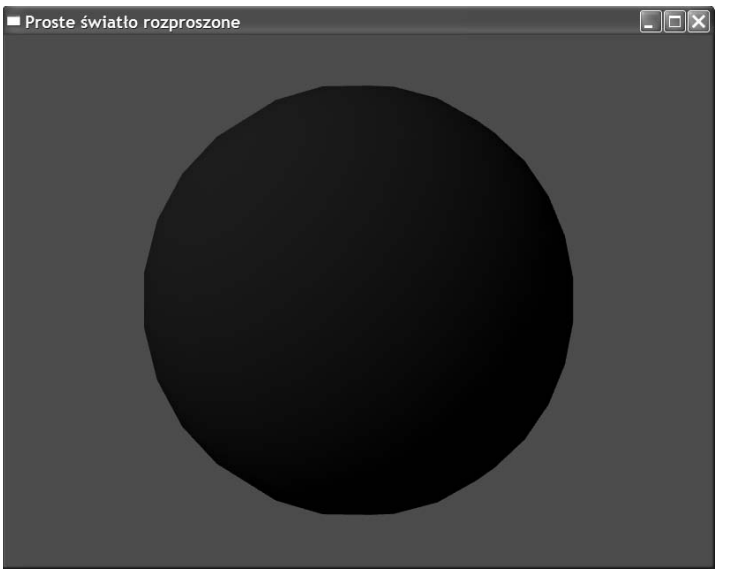
```
// Ta funkcja wykonuje wszystkie działania związane z inicjalizowaniem w kontekście renderowania.
void SetupRC(void)
    {
    // Tło
    glClearColor(0.3f, 0.3f, 0.3f, 1.0f );

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);

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

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

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

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

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

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

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

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

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

    modelViewMatrix.PopMatrix();


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

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

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

Rysunek 6.6.
Program DiffuseLight



Model oświetlenia ADS

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

Światło otaczające

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

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

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

Światło rozproszone

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

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

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

Światło odbicia zwierciadlanego

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

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

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

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

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

Shader ADS

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

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

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

Listing 6.9. *Shader wierzchołków programu ADSGouraud*

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

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

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

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

// Kolor do shadera fragmentów
smooth out vec4 vVaryingColor;

void main(void)
    {
```

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

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

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

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

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

// Dodanie światła otoczenia
vVaryingColor += ambientColor;


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


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

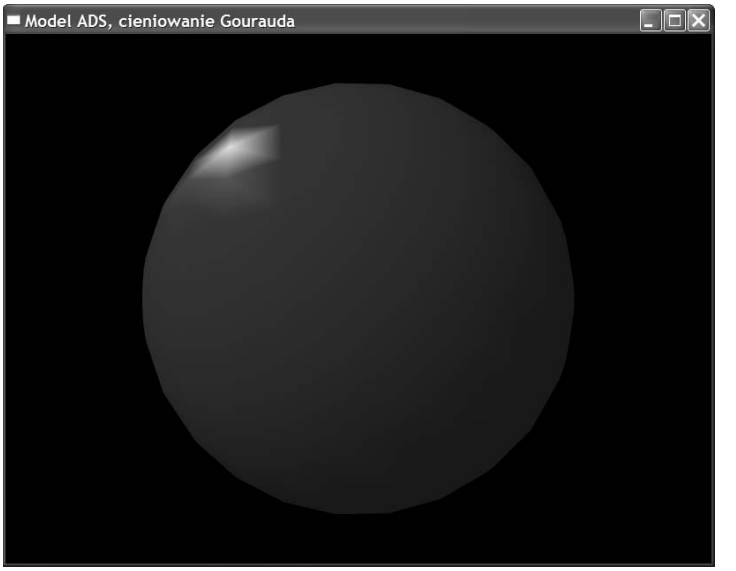
Nie pokazujemy całego kodu shadera, ponieważ przypisuje on tylko przychodzącą wartość zmiennej `vVaryingColor` do koloru fragmentów:

```
vFragColor = vVaryingColor;
```

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

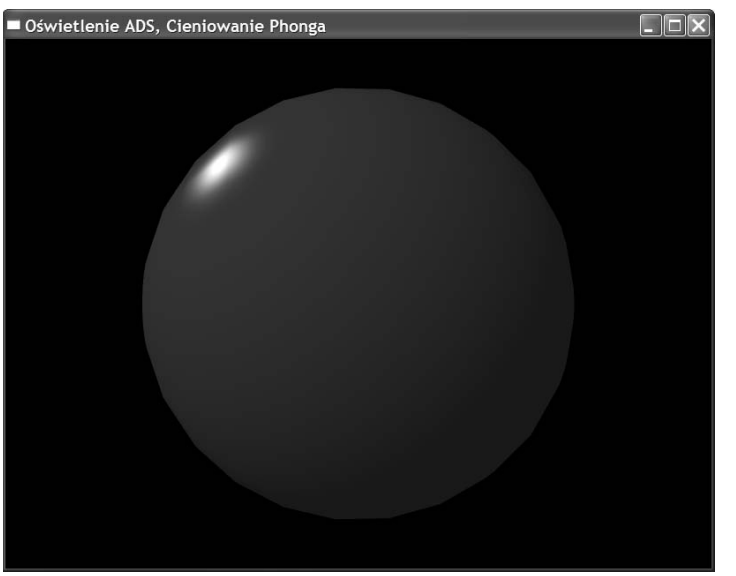
Cieniowanie Phonga

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



Rysunek 6.7. Oświetlanie wierzchołkowe metodą cieniowania Gourauda

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



Rysunek 6.8. Oświetlenie pikselowe (cieniowanie Phonga)

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

Listing 6.10. *Shader wierzchołków ADSPhong*

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

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


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

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

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

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

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


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

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

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

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

Listing 6.11. *Shader fragmentów programu ADSPhong*

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

out vec4 vFragColor;

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

smooth in vec3 vVaryingNormal;
smooth in vec3 vVaryingLightDir;


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

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

    // Dodanie składowej światła otaczającego
    vFragColor += ambientColor;


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

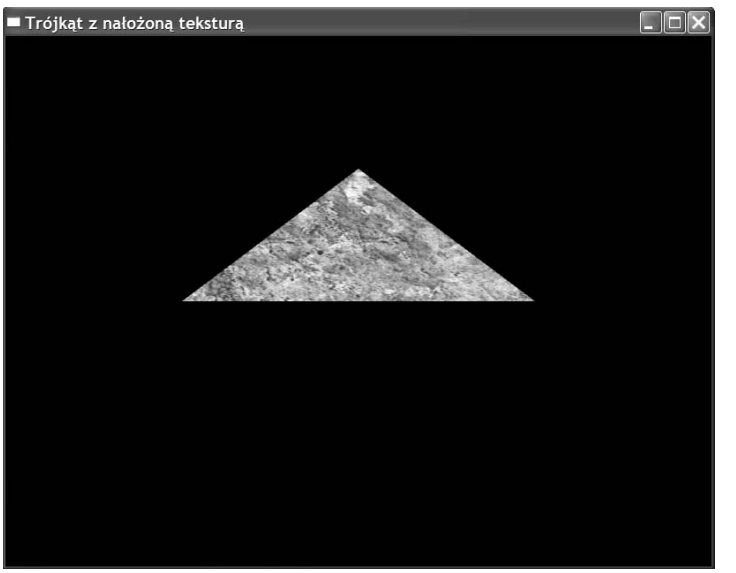
Przy obecnym rozwoju techniki wybór takich zaawansowanych metod wysokiej jakości renderowania jest często uzasadniony. Poprawa jakości obrazu jest znaczna, a utrata wydajności często zaniedbywana. Jednak w niektórych sytuacjach, np. przy programowaniu mało wydajnego sprzętu (takiego jak układ wbudowany)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

Korzystanie z tekstur

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

Nic, tylko teksele

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



Rysunek 6.9. Program rysujący trójkąt z nałożoną teksturą

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

Listing 6.12. *Shader wierzchołków programu TexturedTriangle*

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

in vec4 vVertex;
in vec2 vTexCoords;

smooth out vec2 vVaryingTexCoords;

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

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

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

Listing 6.13. *Shader fragmentów programu TexturedTriangle*

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


uniform sampler2D colorMap;

out vec4 vFragColor;
smooth in vec2 vVaryingTexCoords;


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

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

```
uniform sampler2D colorMap;
```

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

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

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

```
glUseProgram(myTexturedIdentityShader);
glBindTexture(GL_TEXTURE_2D, textureID);
GLint iTextureUniform = glGetUniformLocation(myTexturedIdentityShader,
                                                                "colorMap");
glUniform1i(iTextureUniform, 0);

triangleBatch.Draw();
```

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

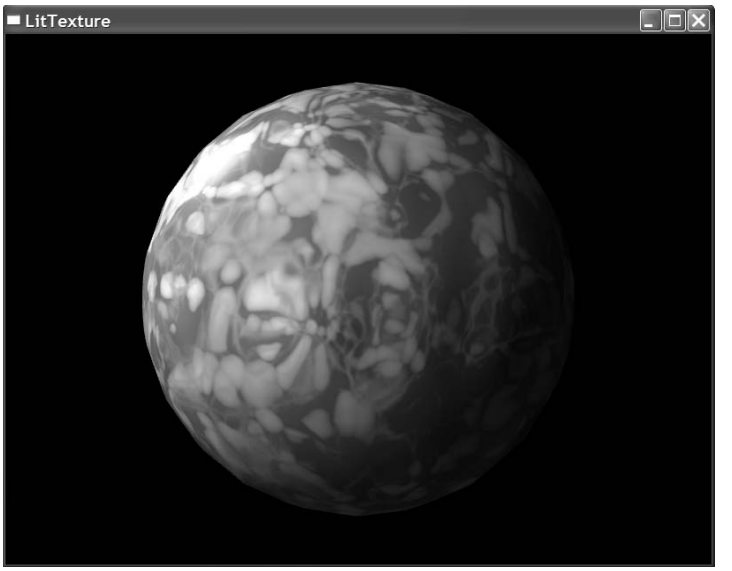
```
vFragColor = texture(colorMap, vVaryingTexCoords.st);
```

Oświetlanie tekseli

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

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

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



Rysunek 6.10. Połączenie światła z teksturą w programie LitTexture

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

Listing 6.14. *Shader fragmentów programu ADSTexture*

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

out vec4 vFragColor;

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

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

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

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

```
// Dodanie składowej światła otaczającego
vFragColor += ambientColor;

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

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

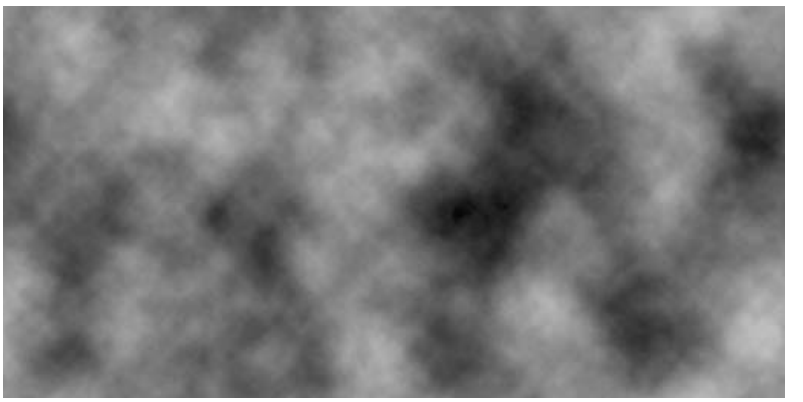
Anulowanie przetwarzania fragmentów

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

```
if(vColorValue.a < 0.1f)
discard;
```

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

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



Rysunek 6.11. Tekstura z chmurą użyta do zademonstrowania efektu erozji

Listing 6.15. *Shader fragmentów programu Dissolve*

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

out vec4 vFragColor;

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

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

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

    if(vCloudSample.r < dissolveFactor)
        discard;


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

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

    // Dodanie składowej światła otaczającego
    vFragColor += ambientColor;
```

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

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

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

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

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

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

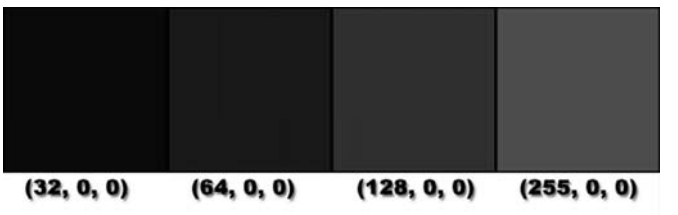
Należy również zauważyć, że działania te wykonujemy we wczesnej fazie działania shadera. Nie ma przecież sensu wykonywać czasochłonnych obliczeń pikselowych, jeśli fragment i tak nie zostanie narysowany. Rysunek 6.12 przedstawia jedną klatkę z naszej animacji.

Rysunek 6.12. Okno programu Dissolve

Teksturowanie w stylu kreskówkowym — teksele w roli światła

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

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



Rysunek 6.13. Jednowymiarowa tabela wyszukiwania kolorów

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

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

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

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

Listing 6.16. *Shader wierzchołków programu ToonShader*

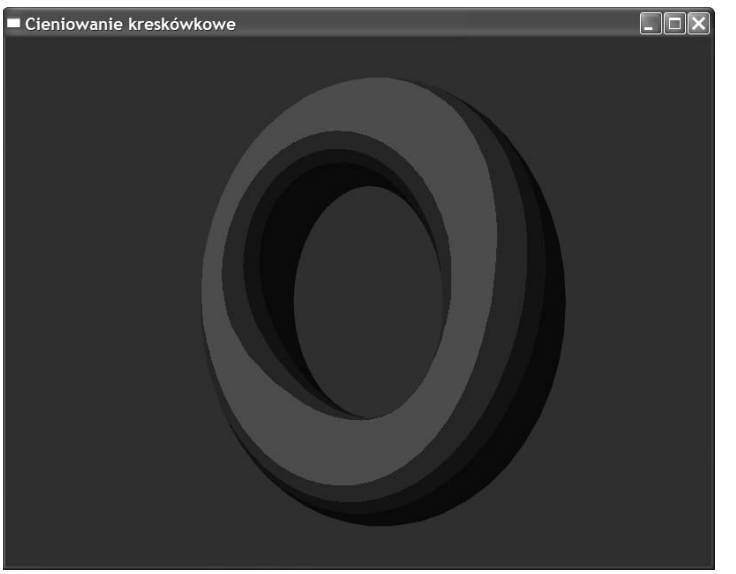
```
// Shader cieniowania kreskówkowego
// Shader wierzchołków
// Richard S. Wright Jr
// OpenGL. Księga eksperta
#version 330

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

smooth out float textureCoordinate;

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


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

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

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

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

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

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

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

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

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



Rysunek 6.14. Torus cieniowany techniką cieniowania kreskówkowego

Podsumowanie

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

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

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

Skorowidz

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Y

Z

Ź

OpenGL

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

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

W tym wyczerpującym podręczniku znajdziesz:

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

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

Nr katalogowy: 5790

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

helion.pl
księgarnia internetowa

Cena: 119,00 zł

ISBN 978-83-246-2976-3

9 788324 629763