OpenGL®

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

OpenGL®
KSIĘGA EKSPERTA

Wydanie 7

Helion

Tytuł oryginału: OpenGL Superbible: Comprehensive Tutorial and Reference (7th Edition)

Tłumaczenie: Rafał Jońca

ISBN: 978-83-283-2107-6

Authorized translation from the English language edition, entitled: OPENGL SUPERBIBLE: COMPREHENSIVE TUTORIAL AND REFERENCE, Seventh Edition; ISBN 0672337479; by Graham Sellers; and by Richard S. Wright, Jr; and by Nicholas Haemel; published by Pearson Education, Inc, publishing as Addison Wesley.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/opglk7.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/opglk7
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 3.

Wzdłuż potoku graficznego

W tym rozdziale:

- Działanie każdego z etapów potoku graficznego OpenGL.
- Połączenie shaderów z etapami potoku o ustalonej funkcji.
- W jaki sposób napisać program wykorzystujący jednocześnie wszystkie etapy potoku graficznego.

Ten rozdział poświęcimy omówieniu potoku graficznego OpenGL, od jego etapu początkowego aż do samego końca. Opiszemy zarówno etapy dotyczące programowalnych bloków, jak i bloki o stałych funkcjach. W poprzednim rozdziale pokrótce przyjrzeliśmy się etapom shaderów wierzchołków i fragmentów. Opis ten był jednak uproszczony, bo aplikacja dotyczyła tylko rysowania jednego, statycznego trójkąta. Jeśli chcemy narysować coś interesującego w OpenGL, musimy znacznie lepiej poznać OpenGL i jego potok. W tym rozdziale zajmiemy się wszystkimi elementami potoku, ich wzajemnym połączeniem i zaprezentujemy przykład shadera dla każdego etapu potoku.

Przekazywanie danych do shadera wierzchołków

Shader wierzchołków to pierwszy **programowalny** etap potoku OpenGL. Jest to etap szczególny, bo to jedyny wymagany etap w całym potoku. Zanim jednak zostanie uruchomiony shader wierzchołków, ma miejsce etap o stałej funkcji, nazywany **pobieraniem wierzchołków** (ang. *vertex fetchning*). Zapewnia on dane wejściowe dla shadera wierzchołków.

Atrybuty wierzchołka

W GLSL mechanizm pobierania i wysyłania danych z shaderów polega na zadeklarowaniu zmiennych globalnych z kwalifikatorami `in` lub `out`. W rozdziale 2. pojawił się już kwalifikator `out`, gdy na listingu 2.4 ustawialiśmy kolor wyjściowy shadera fragmentów. Na początku potoku OpenGL wykorzystamy słowo kluczowe `in` do uzyskania danych wejściowych dla shadera wierzchołków. W etapach pośrednich mogą być stosowane zarówno elementy `in`, jak i `out`, aby móc przekazywać dane między etapami. Wkrótce do tego dojdziemy. Na razie pozostańmy przy zmiennej z kwalifikatorem `in`. Taka zmienna przyjmie dane z zewnątrz, czyli tak naprawdę otrzyma dane z potoku

OpenGL. W praktyce zostanie wypełniona automatycznie w etapie pobierania wierzchołków (etap o stałej funkcji). Zmienną tę nazywa się **atrybutem wierzchołka** (ang. *vertex attribute*).

Atrybuty wierzchołka to sposób na przekazywanie danych wierzchołków do potoku OpenGL. Aby zadeklarować atrybut wierzchołka, zadeklaruj w shaderze wierzchołków zmienną z kwalifikatorem `in`. Przykład tego typu prezentuje listing 3.1, który jako dane wejściowe deklaruje zmienną `offset`.

Listing 3.1. Deklaracja atrybutu wierzchołka

```
#version 450 core

// 'offset' to wejściowy atrybut wierzchołka.
layout (location = 0) in vec4 offset;

void main(void)
{
   const vec4 vertices[3] = vec4[3](vec4( 0.25, -0.25, 0.5, 1.0),
                                    vec4(t0.25, -0.25, 0.5, 1.0),
                                    vec4( 0.25, 0.25, 0.5, 1.0));

   // Dodaj 'offset' do umieszczonych na sztywno wartości.
   gl_Position = vertices[gl_VertexID] + offset;
}
```

Listing 3.1 dodaje zmienną `offset` jako element wejściowy shadera wierzchołków. Ponieważ jest to wejście do pierwszego shadera w potoku, zostanie on wypełniony automatycznie przez etap pobierania wierzchołków. Możemy wskazać, co ma się znaleźć w zmiennej, wywołując jeden z wielu wariantów funkcji atrybutu wierzchołka — `glVertexAttrib*()`. Prototyp dla `glVertexAttrib4fv()` używanego w tym przykładzie ma postać:

```
void glVertexAttrib4fv(GLuint index,
                       const GLfloat * v);
```

Parametr `index` służy jako referencja do atrybutu, a `v` jest wskaźnikiem nowych danych do umieszczenia w atrybucie. Trudno nie zauważyć, że w przykładzie tuż przed deklaracją atrybutu `offset` pojawił się fragment `layout (location = 0)`. To **kwalifikator układu** (ang. *layout qualifier*), którego użyliśmy do ustawienia **pozycji** atrybutu wierzchołka na wartość `0`. Lokalizacja to wartość, którą przekażemy w `index`, aby odnieść się do atrybutu.

Za każdym razem, gdy wywołujemy jedną z funkcji `glVertexAttrib*()` (a jest ich całkiem sporo), aktualizuje ona wartość atrybutu wierzchołka przekazywanego do shadera wierzchołków. Pozwoli nam to na animowanie trójkąta. Listing 3.2 zawiera uaktualnioną wersję funkcji renderującej, która aktualizuje w każdej klatce wartość zmiennej `offset`.

Listing 3.2. Aktualizacja atrybutu wierzchołka

```
// Funkcja renderująca.
virtual void render(double currentTime)
{
   const GLfloat color[] = { (float)sin(currentTime) * 0.5f + 0.5f,
                             (float)cos(currentTime) * 0.5f + 0.5f,
                             0.0f, 1.0f };
   glClearBufferfv(GL_COLOR, 0, color);
```

```
    // Utworzony wcześniej obiekt programu używany do renderingu.
    glUseProgram(rendering_program);
    GLfloat attrib[] = { (float)sin(currentTime) * 0.5f,
                         (float)cos(currentTime) * 0.6f,
                         0.0f, 0.0f };

    // Aktualizacja wartości atrybutu wejściowego 0.
    glVertexAttrib4fv(0, attrib);

    // Rysowanie trójkąta.
    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

Wskutek uruchomienia programu z listingu 3.2 trójkąt będzie się płynnie obracał wokół okna w płaszczyźnie owalnej.

Przekazywanie danych z jednego etapu do drugiego

Do tej pory przekazywaliśmy dane wejściowe do shadera wierzchołków za pomocą atrybutu wierzchołka i słowa kluczowego `in`. Dodatkowo wykorzystaliśmy bloki o stałej funkcji, doczytując lub zapisując dane we wbudowanych zmiennych typu `gl_VertexID` i `gl_Position`. Wykorzystaliśmy również słowo kluczowe `out` do wysłania danych na zewnątrz w shaderze fragmentów. Istnieje również możliwość wysyłania danych z shadera jednego etapu do shadera następnego etapu — służą do tego te same słowa kluczowe: `in` i `out`. W podobny sposób, jak powstała zmienna ze słowem kluczowym `out` w shaderze fragmentów, co pozwoliło przesłać informację o kolorze, tak w shaderze wierzchołków można użyć zmiennych ze słowem kluczowym `out`. Wszystko, co jest oznaczone jako zmienna wyjściowa w jednym shaderze, trafia do zmiennej wejściowej o tej samej nazwie w następnym shaderze, o ile tylko użyto słowa kluczowego `in`. Jeśli shader wierzchołków zadeklaruje zmienną o nazwie `vs_color` i doda słowo kluczowe `out`, to shader fragmentów może odczytać wartość tej zmiennej, o ile również nosi ona nazwę `vs_color` i posiada słowo kluczowe `in`.

Jeśli zmodyfikujemy nasz prosty shader wierzchołków zgodnie z listingiem 3.3 i wprowadzimy `vs_color` jako zmienną wyjściową oraz dodatkowo zmienimy shader fragmentów zgodnie z listingiem 3.4, aby przyjmował zmienną wejściową `vs_color`, uzyskamy przekazanie wartości z jednego shadera do drugiego. W ten sposób shader fragmentów, zamiast wpisywać stałą wartość, może przekazać kolor uzyskany od shadera wierzchołków.

Listing 3.3. Shader wierzchołków wysyłający dane

```
#version 450 core

// 'offset' i 'color' to wejściowe atrybuty wierzchołka.
layout (location = 0) in vec4 offset;
layout (location = 1) in vec4 color;

// vs_color to wartość wyjściowa do przekazania do następnego shadera.
out vec4 vs_color;

void main(void)
```

```
{
   const vec4 vertices[3] = vec4[3](vec4( 0.25, -0.25, 0.5, 1.0),
                                    vec4(-0.25, -0.25, 0.5, 1.0),
                                    vec4( 0.25, 0.25, 0.5, 1.0));

   // Dodaj 'offset' do umieszczonych na sztywno wartości.
   gl_Position = vertices[gl_VertexID] + offset;

   // Przekazanie do vs_color stałej wartości.
   vs_color = color;
}
```

Listing 3.4. Shader fragmentów z danymi wejściowymi

```
#version 450 core

// Dane z shadera wierzchołków.
in vec4 vs_color;

// Wynik kierowany do bufora ramki.
out vec4 color;

void main(void)
{
   // Proste przypisanie danych koloru z shadera wierzchołków do bufora ramki.
   color = vs_color;
}
```

Listing 3.3 zawiera deklarację drugiej wartości wejściowej, o nazwie `color`, shadera wierzchołków (na pozycji 1.) i zapisuje ją w zmiennej `vs_output`. Shader fragmentów z listingu 3.4 pobiera tę wartość i zapisuje w buforze ramki. W ten sposób wartość koloru przekazujemy bezpośrednio z wywołań `glVertexAttrib*()` przez shader wierzchołków i shader fragmentów aż do wynikowego bufora ramki. Innymi słowy, uzyskaliśmy możliwość rysowania trójkątów o różnych kolorach!

Bloki interfejsu

Deklarowanie zmiennych interfejsu pojedynczo jest najprostszym sposobem komunikacji między poszczególnymi shaderami. Z drugiej strony niemalże wszystkie bardziej złożone aplikacje będą przekazywały dosyć skomplikowane struktury między poszczególnymi etapami — mogą to być tablice, struktury i inne złożone układy zmiennych. Na szczęście istnieje mechanizm łączenia ze sobą kilku zmiennych, noszący nazwę **bloku interfejsu** (ang. *interface block*). Deklaracja bloku interfejsu przypomina deklarację struktury, ale korzysta ze słów kluczowych `in` lub `out` w zależności od tego, czy ma służyć jako wejście, czy wyjście shadera. Przykładową definicję bloku interfejsu przedstawia listing 3.5.

Listing 3.5. Shader wierzchołków z wyjściowym blokiem interfejsu

```
#version 450 core

// 'offset' i 'color' to wejściowe atrybuty wierzchołka.
layout (location = 0) in vec4 offset;
layout (location = 1) in vec4 color;
```

```
// Deklaracja VS_OUT jako deklaracja wyjściowego bloku interfejsu.
out VS_OUT
{
   vec4 color; // Wysłanie 'color' do następnego etapu.
} vs_out;

void main(void)
{
   const vec4 vertices[3] = vec4[3](vec4( 0.25, -0.25, 0.5, 1.0),
                                    vec4(-0.25, -0.25, 0.5, 1.0),
                                    vec4( 0.25, 0.25, 0.5, 1.0));

   // Dodaj 'offset' do umieszczonych na sztywno wartości.
   gl_Position = vertices[gl_VertexID] + offset;

   // Przekazanie do vs_color stałej wartości.
   vs_out.color = color;
}
```

Zauważ, że blok interfejsu z listingu 3.5 zawiera zarówno nazwę bloku (`VS_OUT`), jak i nazwę instancji (`vs_out`). Bloki interfejsu są parowane pomiędzy etapami na podstawie nazwy bloku (w tym wypadku `VS_OUT`), ale wewnątrz shadera stosuje się nazwę instancji (w tym wypadku `vs_out`). Modyfikacja shadera fragmentów, aby używał bloku interfejsu, może wyglądać tak jak na listingu 3.6.

Listing 3.6. Shader fragmentów z wejściowym blokiem interfejsu

```
#version 450 core

// Deklaracja VS_OUT jako wejściowy blok interfejsu.
in VS_OUT
{
   vec4 color; // Odebranie koloru z poprzedniego etapu.
} fs_in;

// Wynik kierowany do bufora ramki.
out vec4 color;

void main(void)
{
   // Proste przypisanie danych koloru z shadera wierzchołków do bufora ramki.
   color = fs_in.color;
}
```

Dopasowywanie bloków interfejsów po nazwie bloku z utrzymaniem niezależności nazw wewnętrznych ma dwa bardzo istotne cele. Po pierwsze, nazwa wewnętrzna stosowana w każdym z etapów może być inna, co pozwala między innymi uniknąć pewnej niezręczności w stosowaniu nazwy `vs_out` w shaderze fragmentów. Po drugie, umożliwia interfejsowi przejście z pojedynczego elementu na tablicę, gdy przekracza granice niektórych etapów. Dotyczy to na przykład etapów związanych z shaderami wierzchołków, teselacji i geometrii, o czym wkrótce się przekonasz. Zauważ, że bloki interfejsu dotyczą tylko przenoszenia danych z jednego etapu do drugiego. Nie można ich użyć do grupowania danych wejściowych shadera wierzchołków i wyjścia z shadera fragmentów.

Teselacja

Teselacja to proces przekształcania prymitywów wysokiego rzędu (nazywanych w OpenGL **płatami** lub **powierzchniami parametrycznymi**) na mniejsze i prostsze prymitywy (trójkąty) w celu właściwego zrenderowania. OpenGL posiada wbudowaną i konfigurowalną funkcję obsługującą teselację — potrafi ona rozbijać czworoboki, trójkąty i linie na znacznie większą liczbę mniejszych trójkątów, linii i punktów, które można bezpośrednio przekazać do sprzętowego mechanizmu rasteryzacji. Faza teselacji znajduje się w potoku OpenGL za etapem shadera wierzchołków i składa się z trzech części: shadera sterowania teselacją, mechanizmu teselacji o stałej funkcji i shadera wyliczenia teselacji.

Shadery sterowania teselacją

Pierwszą fazą trójelementowego etapu teselacji jest **shader sterowania teselacją** (nazywany czasem po prostu **shaderem sterującym**, ang. *tessellation control shader*). Shader ten przyjmuje dane od shadera wierzchołków i ma dwa główne zadania: określić poziom teselacji dla mechanizmu teselacji i wygenerować dane wysyłane do shadera wyliczenia teselacji uruchamianego po głównej fazie dzielenia wierzchołków.

Teselacja w OpenGL działa poprzez rozbicie powierzchni wysokiego poziomu nazywanych **płatami** (ang. *patch*) na punkty, linie i trójkąty. Każdy płat składa się z pewnej liczby **punktów sterujących** (ang. *control points*). Ich liczbę konfiguruje się, wywołując funkcję `glPatchParameteri()` z parametrem `pname` ustawionym na `GL_PATCH_VERTICES` i parametrem `value` ustawionym na liczbę punktów mających tworzyć płat. Prototyp funkcji `glPatchParameteri()` ma postać:

```
void glPatchParameteri(GLenum pname,
                       GLint value);
```

Domyślnie liczba punktów sterujących na płat wynosi 3. Jeśli właśnie taka liczba punktów jest niezbędna (jak to ma miejsce w przykładowej aplikacji), nie trzeba tej funkcji w ogóle wywoływać. Maksymalna liczba punktów sterujących dla pojedynczego płata zależy od implementacji sterowników, ale OpenGL gwarantuje, że nie będzie mniejsza niż 32.

Gdy teselacja zostaje uaktywniona, shader wierzchołków uruchamia się jeden raz na każdy punkt sterujący, a shader sterujący uaktywnia się dla każdej grupy punktów sterujących (każda grupa odpowiada liczbie wierzchołków płata). Innymi słowy, wierzchołki stają się punktami sterującymi, a wynik działania shadera wierzchołków trafia grupami do shadera sterującego jako jego dane wejściowe. Liczbę punktów sterujących na płat można zmienić wewnątrz shadera, więc liczba punktów wejściowych nie musi odpowiadać liczbie punktów wyjściowych. Liczbę punktów sterujących tworzonych przez shader sterujący ustawia się za pomocą odpowiedniego kwalifikatora wyjściowego w kodzie źródłowym shadera. Kwalifikator ma postać:

```
layout (vertices = N) out;
```

W kodzie `N` oznacza liczbę punktów sterujących na płat. Shader sterujący odpowiada za wyliczenie wartości wynikowych punktów sterujących i za ustawienie współczynników teselacji płata wynikowego, który trafi do mechanizmu teselacji o stałej funkcji. Współczynniki teselacji umieszcza się we wbudowanych zmiennych wyjściowych `gl_TessLevelInner` i `gl_TessLevelOuter`. Wszystkie inne dane dla dalszych etapów potoków przekazuje się do zmiennych wyjściowych zdefiniowanych przez użytkownika (dotyczy to zarówno słowa kluczowego `out`, jak i specjalnej, wbudowanej tablicy `gl_out`).

Listing 3.7 przedstawia prosty shader sterowania teselacją. Ustawia liczbę wyjściowych punktów sterujących na 3 (czyli taką samą jak liczba wejściowych punktów sterujących) za pomocą konstrukcji `layout (vertices = 3) out;`. Kopiuje dane wejściowe na wyjście (używa wbudowanych zmiennych `gl_in` i `gl_out`) oraz ustawia oba poziomy teselacji (wewnętrzny i zewnętrzny) na poziom 5. Wyższe wartości spowodują powstanie gęstszej struktury, a niższe — mniej gęstej. Ustawienie wartości 0 jako poziomu teselacji w zasadzie spowoduje pominięcie całego płata.

Listing 3.7. Pierwszy shader sterowania teselacją

```
#version 450 core

layout (vertices = 3) out;

void main(void)
{
   // Tylko, jeśli to wywołanie o identyfikatorze 0…
   if (gl_InvocationID == 0)
   {
      gl_TessLevelInner[0] = 5.0;
      gl_TessLevelOuter[0] = 5.0;
      gl_TessLevelOuter[1] = 5.0;
      gl_TessLevelOuter[2] = 5.0;
   }
   // Zawsze kopiuj wejście na wyjście.
   gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
}
```

Wbudowana zmienna wejściowa `gl_InvocationID` służy jako indeks dla tablic `gl_in` i `gl_out`. Zmienna zawiera bazujący na zerze indeks punktu sterującego płata, nad którym w danym momencie pracuje shader sterowania teselacją.

Mechanizm teselacji

Mechanizm teselacji to część potoku OpenGL o stałej funkcji. Odpowiada za przetworzenie powierzchni wyższego rzędu, reprezentowanych przez płaty, na mniejsze elementy podstawowe, takie jak punkty, linie i trójkąty. Zanim mechanizm otrzyma konkretny płat, shader sterujący uzyskuje punkty sterujące i ustawia współczynniki teselacji odpowiadające za proces konwersji. Utworzone przez mechanizm teselacji wynikowe wierzchołki trafiają do shadera wyliczenia teselacji. Mechanizm teselacji odpowiada za wyliczenie parametrów, które trafią do shadera wyliczenia — shader w razie potrzeby może przekształcić wynikowe prymitywy, zanim będą gotowe do rasteryzacji.

Shadery wyliczenia teselacji

Po wykonaniu teselacji jako stałej funkcji otrzymujemy zbiór wierzchołków utworzonych na podstawie źródłowego płata. Nowe wierzchołki trafiają do **shadera wyliczenia teselacji** (ang. *tessellation evaluation shader*). Shader ten, nazywany często w skrócie shaderem wyliczenia, zostaje uruchomiony dla każdego wierzchołka utworzonego przez mechanizm teselacji. Z tego powodu należy uważać ze złożonymi shaderami wyliczenia, ponieważ mogą być wykonane ogromną liczbę razy. W szczególności należy ograniczyć korzystanie ze złożonych shaderów wyliczenia przy wysokich poziomach teselacji.

Listing 3.8 przedstawia shader wyliczenia, który przyjmuje wierzchołki wejściowe utworzone przez mechanizm teselacji sterowany shaderem sterującym z listingu 3.7. Na początku kwalifikator określa tryb teselacji. W tym wypadku wybraliśmy tryb generowania trójkątów. Pozostałe kwalifikatory, czyli `equal_spacing` i `cw`, wskazują, że wierzchołki mają być równo rozłożone na obszarze płata poddawanego teselacji, a kierunek podawania wierzchołków ma być zgodny z ruchem wskazówek zegara. Wszystkimi dostępnymi opcjami zajmiemy się w rozdziale 8., opisującym szczegółowo proces teselacji.

Listing 3.8. Pierwszy shader wyliczenia teselacji

```
#version 450 core

layout (triangles, equal_spacing, cw) in;

void main(void)
{
   gl_Position = (gl_TessCoord.x * gl_in[0].gl_Position +
                  gl_TessCoord.y * gl_in[1].gl_Position +
                  gl_TessCoord.z * gl_in[2].gl_Position);
}
```

Pozostała część shadera przypisuje zmiennej `gl_Position` wartość, podobnie jak robi to shader wierzchołków. Wyliczenie odbywa się przy użyciu dwóch dodatkowych zmiennych wbudowanych. Pierwszą zmienną jest `gl_TessCoord`, czyli **współrzędna barycentryczna** (ang. *barycentric coordinate*) wierzchołka wygenerowanego przez mechanizm teselacji. Drugą jest składowa `gl_Position` struktury tablicy `gl_in[]`. Odpowiada ona strukturze `gl_out` wykorzystywanej w shaderze sterującym z listingu 3.7. W zasadzie przedstawiony shader realizuje proste przekazanie efektów teselacji. Innymi słowy, obiekt wynikowy ma dokładnie taki sam kształt jak płat wejściowy.

Aby zobaczyć wynik działania mechanizmu teselacji, trzeba poinformować OpenGL, żeby rysował jedynie zarysy wynikowych trójkątów. W tym celu użyjemy funkcji `glPolygonMode()` o następującym prototypie:

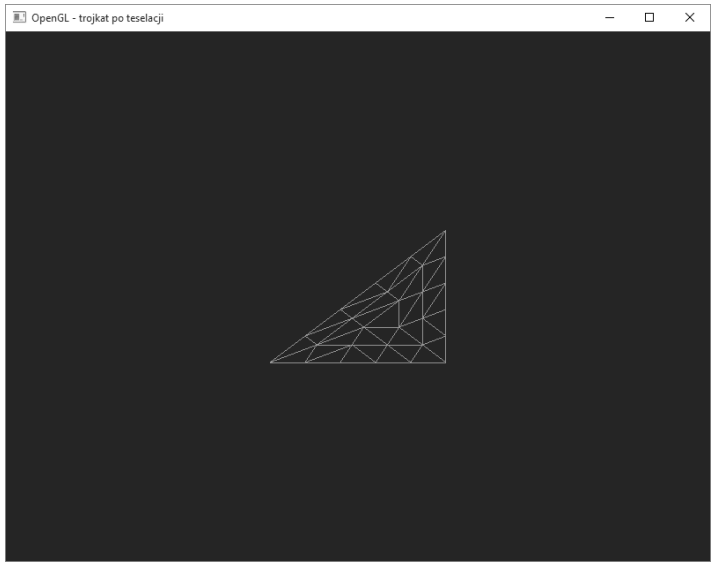
```
void glPolygonMode(GLenum face,
                   GLenum mode);
```

Parametr `face` określa typ wieloboków, które chcemy zmienić. Ponieważ zamierzamy zmienić wszystko, przekazujemy stałą `GL_FRONT_AND_BACK`. Inne typy opiszemy już wkrótce. Parametr `mode` pokazuje sposób renderowania wieloboków. Jako że chcemy rysować w trybie siatki (czyli tylko linie), stosujemy stałą `GL_LINE`. Wynik renderowania jednego trójkąta przy włączonej teselacji i shaderach z listingów 3.7 i 3.8 pokazuje rysunek 3.1.

Shadery geometrii

Shader geometrii (ang. *geometry shader*) to ostatni shader znajdujący się po tak zwanej stronie przedniej — umieszczony jest za etapami wierzchołków i teselacji, ale przed rasteryzacją. Shader geometrii zostaje uruchomiony raz dla prymitywu i ma dostęp do wszystkich danych wejściowych wszystkich wierzchołków tworzących konkretny prymityw. Shader ten jest w pewnym sensie wyjątkowy, jako że potrafi zwiększyć lub zmniejszyć w sposób programowy ilość danych przechodzących przez potok. Co prawda shadery teselacji również wpływają na ilość pracy realizowanej przez potok,

Rysunek 3.1.
Pierwszy trójkąt z teselacją



ale robią to tylko pośrednio w wyniku zmiany poziomu teselacji. Shadery geometrii zawierają dwie funkcje — `EmitVertex()` i `EndPrimitive()` — które jawnie tworzą wierzchołki przekazywane później do etapu rasteryzacji.

Jeszcze jedną ciekawą cechą shaderów geometrii jest to, że potrafią zmienić tryb prymitywu w środku potoku. Na przykład jako wejście mogą przyjmować trójkąty, ale jako wynik emitować linie lub punkty, a nawet tworzyć trójkąty z niezależnych punktów. Przykładowy shader geometrii przedstawiono na listingu 3.9.

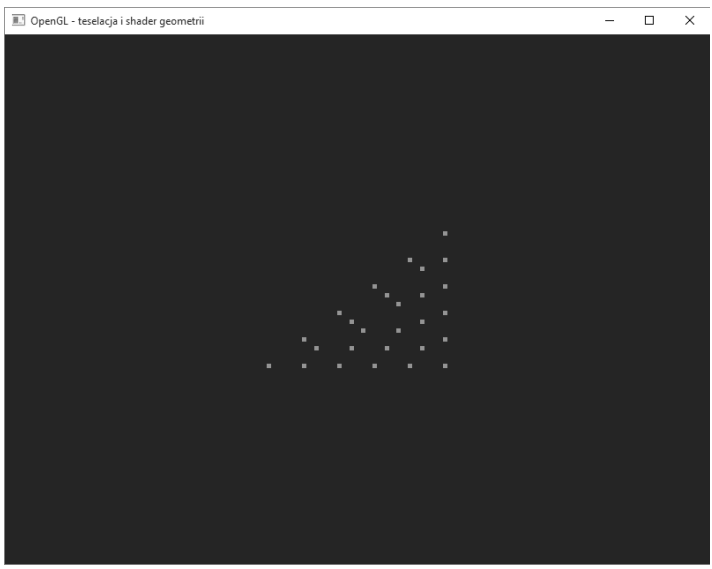
Listing 3.9. Pierwszy shader geometrii

```
#version 450 core

layout (triangles) in;
layout (points, max_vertices = 3) out;

void main(void)
{
   int i;

   for (i = 0; i < gl_in.length(); i++)
   {
      gl_Position = gl_in[i].gl_Position;
      EmitVertex();
   }
}
```

Shader przedstawiony na listingu 3.9 jest bardzo prostym shaderem przekazującym dane dalej, ale przy okazji konwertującym trójkąt na punkty, co umożliwia zobaczenie ich wierzchołków. Pierwszy kwalifikator wskazuje, że shader geometrii oczekuje trójkątów jako danych wejściowych. Drugi kwalifikator informuje, że shader będzie tworzył punkty i będzie ich maksymalnie 3. W funkcji `main()` przekazuje dalej poszczególne składowe tablicy `gl_in`, wykorzystując w tym celu jej funkcję `length()`.

Tak naprawdę wiemy, że tablica będzie miała długość 3, ponieważ przetwarzamy trójkąty, a każdy trójkąt ma 3 wierzchołki. Wynik działania shadera geometrii przypomina wynik działania shadera wierzchołków. Przede wszystkim zapisujemy dane położenia w zmiennej `gl_Position`. Następnie wywołujemy `EmitVertex()`, aby utworzyć nowy wierzchołek w danych wyjściowych. Shadery geometrii automatycznie wywołują funkcję `EndPrimitive()` na końcu shadera, więc w tym konkretnym przykładzie użycie funkcji jest całkowicie zbędne. Shader spowoduje, że przekazany trójkąt zostanie zrenderowany jako 3 osobne punkty.

Po wstawieniu shadera geometrii do przykładu z teselacją pojedynczego trójkąta uzyskamy efekt przedstawiony na rysunku 3.2. Aby efekt był lepiej widoczny, ustawiliśmy rozmiar punktów na `5.0`, wywołując funkcję `glPointSize()`.



Rysunek 3.2. Trójkąt po teselacji i dodaniu shadera geometrii

Składanie prymitywów, przycinanie i rasteryzacja

Po zrealizowaniu wszystkich zadań przedniej części potoku (czyli shadera wierzchołków, teselacji i shadera geometrii) stała część potoku realizuje serię zadań przetwarzających dane reprezentowane przez wierzchołki na serię pikseli, które zostaną pokolorowane i zapisane na ekranie jako element sceny. Pierwszy krok tego procesu polega na składaniu prymitywów, czyli grupowaniu wierzchołków w linie i trójkąty. Składanie odbywa się również dla punktów, choć w tej sytuacji proces jest niezwykle prosty.

Po utworzeniu prymitywów z poszczególnych wierzchołków prymitywy zostają **przycięte** do widocznego obszaru, czyli do okna lub ekranu. Obszar ten może być też mniejszy, więc nazywa się go czasem **obszarem renderingu** (ang. *viewport*). Gdy zostaną określone potencjalnie widoczne obszary prymitywów, następuje przesłanie danych do następnego podsystemu o stałej funkcji, przeprowadzającego rasteryzację. Blok ten określa, które piksele obejmuje swym zasięgiem prymityw (punkt, linię lub trójkąt), i wysyła listę pikseli do następnego etapu, czyli shadera fragmentów.

Przycinanie

Gdy wierzchołki opuszczają przednią część potoku, znajdują się w tak zwanej **przestrzeni przycięcia** (ang. *clip space*). To jeden z wielu systemów współrzędnych wykorzystywanych do reprezentacji położenia. Być może zwróciłeś uwagę na to, że zmienna `gl_Position`, w której w shaderach umieszczaliśmy wartości, jest typu `vec4`, więc położenia wierzchołków wykorzystują pełny, czterokomponentowy wektor. To tak zwane **współrzędne jednorodne** (ang. *homogeneous coordinates*). Do geometrii rzutowej wykorzystywany jest system współrzędnych jednorodnych, ponieważ kryjąca się za nim matematyka jest prostsza niż w wypadku standardowej przestrzeni kartezjańskiej. Współrzędne jednorodne wykorzystują o jeden komponent więcej niż współrzędne kartezjańskie, więc wektor w przestrzeni trójwymiarowej reprezentuje zmienna o czterech komponentach.

Choć wynik działania przedniej części potoku dotyczy czterokomponentowych współrzędnych jednorodnych, przycięcie następuje w przestrzeni kartezjańskiej. W trakcie konwersji OpenGL przeprowadza tak zwane **dzielenie rzutowe** (ang. *perspective division*), czyli dzielenie wszystkich czterech komponentów położenia przez ostatni komponent, nazywany *w*. Po tej operacji następuje odwzorowanie przestrzeni jednorodnej na przestrzeń kartezjańską, a *w* otrzymuje wartość `1.0`. Ponieważ do tej pory ustawialiśmy komponent w zmiennej `gl_Position` właśnie na `1.0`, dzielenie nie będzie miało żadnego efektu. Gdy wkrótce dokładniej zapoznamy się z geometrią rzutową, omówimy efekt ustawienia *w* na wartość inną niż `1.0`.

Po dzieleniu rzutowym wynikowa pozycja znajduje się w **znormalizowanej przestrzeni urządzenia** (ang. *normalized device space*). W OpenGL widoczny obszar znormalizowanej przestrzeni urządzenia to obszar od `–1.0` do `1.0` w wymiarach *x* i *y* oraz od `0.0` do `1.0` w wymiarze *z*. Tylko geometria znajdująca się w tym obszarze może być dla użytkownika widoczna. Wszystko, co znajduje się poza nią, można pominąć. Sześć boków tego sześcianu tworzy płaszczyzny w przestrzeni trójwymiarowej. Ponieważ płaszczyzna dzieli przestrzeń współrzędnych na dwie części, obszar po każdej ze stron nazywa się **półprzestrzenią** (ang. *half-space*).

Przed przekazaniem prymitywów do następnego etapu OpenGL przeprowadza przycięcie na podstawie tego, po której stronie płaszczyzny znajduje się każdy z wierzchołków prymitywu. Płaszczyzna ma część „wewnętrzną” i „zewnętrzną”. Jeśli wszystkie wierzchołki prymitywu znajdują się w części zewnętrznej, cały prymityw jest pomijany. Jeśli wszystkie wierzchołki prymitywu znajdują się w części wewnętrznej, jest on przekazywany dalej bez żadnych zmian. Prymitywy widoczne częściowo (przecinające jedną z płaszczyzn) muszą zostać obsłużone w sposób szczególny. Więcej informacji na ten temat znajduje się w rozdziale 7.

Transformacja obszaru renderingu

Po przycięciu wszystkie wierzchołki geometrii mają współrzędne znajdujące się w zakresie od `–1.0` do `1.0` (koordynaty *x* i *y*). Po dołączeniu współrzędnej *z* w zakresie od `0.0` do `1.0` mówimy o znormalizowanych przestrzeniach urządzenia. Z drugiej strony okno, w którym ma nastąpić rysowanie, ma najczęściej[1] wymiary od (0, 0) w lewym dolnym narożniku do (*w* – 1, *h* – 1) w prawym górnym narożniku, gdzie *w* i *h* to, odpowiednio, szerokość i wysokość okna w pikselach. Aby umieścić geometrię w oknie, OpenGL stosuje **przekształcenie obszaru renderingu** (ang. *viewport transform*), które

[1] Możliwa jest zmiana konwencji dotyczących współrzędnych tak, aby (0, 0) znajdowało się w lewym górnym narożniku ekranu, czyli było stosowane rozwiązanie przyjęte w wielu innych systemach graficznych.

skaluje i przenosi wierzchołki ze współrzędnych znormalizowanych na **współrzędne okna** (ang. *window coordinates*). Skalę i przesunięcie określa się za pomocą granic obszaru renderingu definiowanych przy użyciu funkcji `glViewport()` i `glDepthRange()`. Prototypy obu funkcji są następujące:

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
void glDepthRange(GLdouble nearVal, GLdouble farVal);
```

Transformacja przyjmuje taką oto postać:

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{p_x}{2}x_d + o_x \\ \frac{p_y}{2}y_d + o_y \\ \frac{f-n}{2}z_d + \frac{n+f}{2} \end{pmatrix}$$

We wzorze x_w, y_w i z_w są wynikowymi współrzędnymi wierzchołka w przestrzeni okna, a x_d, y_d i z_d są współrzędnymi wejściowymi wierzchołka w przestrzeni znormalizowanej. Wartości p_x i p_y to szerokość i wysokość okna w pikselach, a n i f to bliska i daleka płaszczyzna we współrzędnych z. Wartości o_x, o_y i o_z oznaczają początek układu współrzędnych.

Usuwanie zbędnych trójkątów

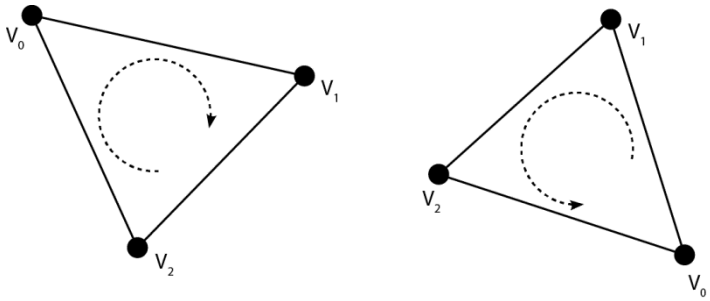
Zanim trójkąty zaczną być przetwarzane, mogą zostać opcjonalnie przekazane do jeszcze jednego etapu, o nazwie **culling**. Określa on, czy płaszczyzna trójkąta jest zwrócona do oglądającego przodem, czy tyłem. Na tej podstawie można określić, czy w ogóle warto takim trójkątem się dalej zajmować. Jeśli trójkąt jest zwrócony przodem do oglądającego, mówimy o **odwróceniu frontem**; w przeciwnym razie mówimy o **odwróceniu tyłem**. Bardzo często pomija się trójkąty odwrócone tyłem, bo jeśli obiekt jest zamknięty, każdy odwrócony tyłem trójkąt będzie zakryty przez trójkąt odwrócony przodem.

Aby ustalić odwrócenie trójkąta, OpenGL określi **znak** obszaru w przestrzeni okna. Jednym ze sposobów określenia obszaru trójkąta jest wyliczenie iloczynu wektorowego dwóch jego krawędzi. Oto wzór:

$$a = \frac{1}{2}\sum_{i=0}^{n-1} x_w^i y_w^{i\oplus 1} - x_w^{i\oplus 1} y_w^i$$

We wzorze x^i_w i y^i_w to współrzędne i-tego wierzchołka trójkąta w przestrzeni okna, a $i \oplus 1$ to $(i + 1)$ modulo 3. Jeśli obszar jest dodatni, trójkąt uważa się za odwrócony frontem. Wartość ujemna oznacza odwrócenie tyłem. Sens tych wyliczeń można zamienić, wywołując funkcję `glFrontFace()` i ustawiając parametr `dir` na wartość `GL_CW` lub `GL_CCW` (gdzie `CW` oznacza zgodnie z ruchem wskazówek zegara, a `CCW` przeciwnie do ruchu wskazówek zegara). Jest to tak zwany **kierunek rysowania** (ang. *winding order*) trójkąta. Ruch wskazówek zegara lub jego odwrotność określa kolejność pojawiania się wierzchołków w przestrzeni okna. Domyślną wartością jest `GL_CCW`, co oznacza, że trójkąty, których wierzchołki są ułożone przeciwnie do ruchu wskazówek zegara, uważa się za ułożone przodem, a te ułożone w odwrotnym kierunku — za skierowane tyłem. Ustawienie wartości `GL_CW` spowoduje po prostu zanegowanie wartości a, efekt będzie więc dokładnie odwrotny. Rysunek 3.3 ilustruje działanie opisanego algorytmu w praktyce.

Gdy już uda się określić kierunek zwrotu trójkąta, OpenGL może pominąć trójkąty zwrócone przodem, tyłem lub nawet oba rodzaje. Domyślnie OpenGL renderuje wszystkie trójkąty niezależnie od sposobu ich zwrotu. Aby włączyć usuwanie zbędnych trójkątów, wywołaj funkcję `glEnable()` z war-



Rysunek 3.3. Ułożenie zgodnie ze wskazówkami zegara (po lewej) i przeciwnie do ruchu wskazówek zegara (po prawej)

tością stałej `GL_CULL_FACE`. Domyślnie OpenGL usunie trójkąty skierowane tyłem. Aby zmienić rodzaj usuwanych trójkątów, wywołaj funkcję `glCullFace()` i przekaż jej wartość `GL_FRONT`, `GL_BACK` lub `GL_FRONT_AND_BACK`.

Ponieważ punkty i linie nie mają żadnego obszaru geometrycznego[2], obliczenia dotyczące strony nie mają dla nich żadnego zastosowania i nie można ich usunąć na tym etapie.

Rasteryzacja

W procesie rasteryzacji określa się, które fragmenty mogą zostać przysłonięte przez prymityw taki jak linia lub trójkąt. Istnieją dziesiątki algorytmów do obsługi tego zadania, ale większość systemów OpenGL bazuje w wypadku trójkątów na metodzie z półprzestrzeniami, bo umożliwia to zrównoleglenie działań. W dużym skrócie — OpenGL określa dla trójkąta otaczający go prostokąt we współrzędnych okna, a następnie testuje każdy fragment, aby stwierdzić, czy znajduje się wewnątrz, czy na zewnątrz trójkąta. W tym celu każdą z trzech krawędzi trójkąta traktuje jako półprzestrzenie dzielące obszar okna na dwie części.

Fragmenty znajdujące się wewnątrz wszystkich trzech krawędzi traktuje się jako miejsce wewnątrz trójkąta, a fragmenty, które choć dla jednej z trzech półpłaszczyzn znajdują się na zewnątrz — jako obszary poza trójkątem. Ponieważ algorytm określający, po której stronie znajduje się punkt, jest relatywnie prosty i nie zależy od niczego poza położeniem linii i sprawdzanym punktem, wiele takich testów można przeprowadzać w tym samym czasie, co daje szansę na ogromne zrównoleglenie tej operacji.

Shadery fragmentów

Shader fragmentów[3] (ang. *fragment shader*) jest ostatnim programowalnym etapem w potoku graficznym OpenGL. Etap ten ma za zadanie określić kolor każdego fragmentu przed wysłaniem go do bufora ramki w celu umieszczenia w oknie systemowym. Po obsłużeniu prymitywu przez proces

[2] Oczywiście po narysowaniu na ekranie punkty i linie zajmują przestrzeń — inaczej nie byłyby widoczne. Ich pole jest jednak tworzone sztucznie i nie może zostać obliczone na podstawie położenia wierzchołków.

[3] Termin **fragment** oznacza element, który ostatecznie najprawdopodobniej zdecyduje o kolorze piksela. Końcowy piksel może nie mieć koloru wyliczonego przez konkretne wywołanie shadera fragmentów ze względu na stosowanie jeszcze wielu innych efektów, takich jak test szablonu, mieszanie kolorów czy wielokrotne próbkowanie. Wszystkie te operacje zostaną omówione w dalszej części książki.

rasteryzacji powstaje lista fragmentów, która musi zostać pokolorowana. Lista ta trafia do shadera fragmentów. To właśnie tu odbywa się największa praca w całym potoku graficznym, bo każdy trójkąt mógł przekształcić się w setki, tysiące, a nawet miliony fragmentów.

Listing 2.4 z rozdziału 2. zawiera przykład pierwszego wykonanego w tej książce shadera fragmentów. Ten bardzo prosty shader deklaruje pojedynczą wartość wejściową, a następnie przypisuje jej konkretną wartość. W aplikacji działającej produkcyjnie shader fragmentów będzie znacznie bardziej złożony i będzie realizował zadania dotyczące wyliczenia oświetlenia, nałożenia materiałów, a nawet wyliczenia głębi fragmentu. Shader fragmentu ma dostęp do kilku wbudowanych zmiennych, na przykład `gl_FragCoord`, która zawiera informację o położeniu fragmentu w oknie. Dzięki tym informacjom można utworzyć unikatowy kolor dla każdego fragmentu.

Listing 3.10 przedstawia shader fragmentów wyliczający kolor na podstawie zmiennej `gl_FragCoord`. Rysunek 3.4 pokazuje wynik działania oryginalnego programu z pojedynczym trójkątem po zmianie shadera na nowy.

Listing 3.10. Tworzenie koloru fragmentu na podstawie położenia

```
#version 450 core

out vec4 color;

void main(void)
{
   color = vec4(sin(gl_FragCoord.x * 0.25) * 0.5 + 0.5,
                cos(gl_FragCoord.y * 0.25) * 0.5 + 0.5,
                sin(gl_FragCoord.x * 0.15) * cos(gl_FragCoord.y * 0.15),
                1.0);
}
```

Kolor każdego piksela na rysunku 3.4 zależy od jego położenia, powstał więc wzorzec układający się zgodnie ze współrzędnymi na ekranie. Shader z listingu 3.10 spowodował powstanie wzorca przypominającego szachownicę.

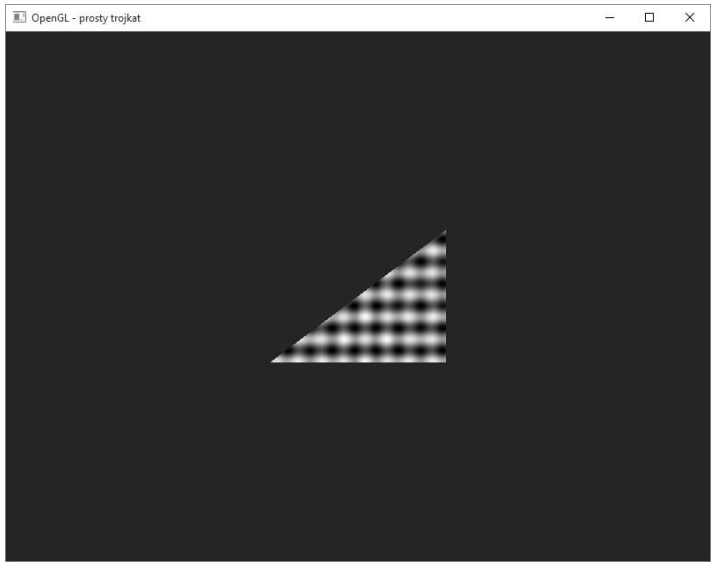
Zmienna `gl_FragCoord` jest jedną z wbudowanych zmiennych dostępnych na poziomie shadera. Podobnie jak w wypadku innych shaderów możemy również zastosować własne dane wejściowe. To, co otrzyma shader fragmentów, zależy od danych wyjściowych ostatniego etapu przed rasteryzacją. Jeśli program korzysta tylko z shadera wierzchołków i shadera fragmentów, możemy przekazać dane do shadera fragmentów bezpośrednio z poziomu shadera wierzchołków.

Dane wejściowe shadera fragmentów nie przypominają danych wejściowych innych etapów, ponieważ OpenGL **interpoluje** wartości względem renderowanego prymitywu. Aby to zademonstrować, zastosujmy shader wierzchołków z listingu 3.3 i zmodyfikujmy go tak, aby każdemu wierzchołkowi przypisywał inny kolor (patrz listing 3.11).

Listing 3.11. Shader wierzchołków z danymi wyjściowymi

```
#version 450 core

// vs_color to dane wyjściowe wysyłane do następnego shadera.
out vec4 vs_color;

void main(void)
```

Rysunek 3.4. Wynik działania listingu 3.10



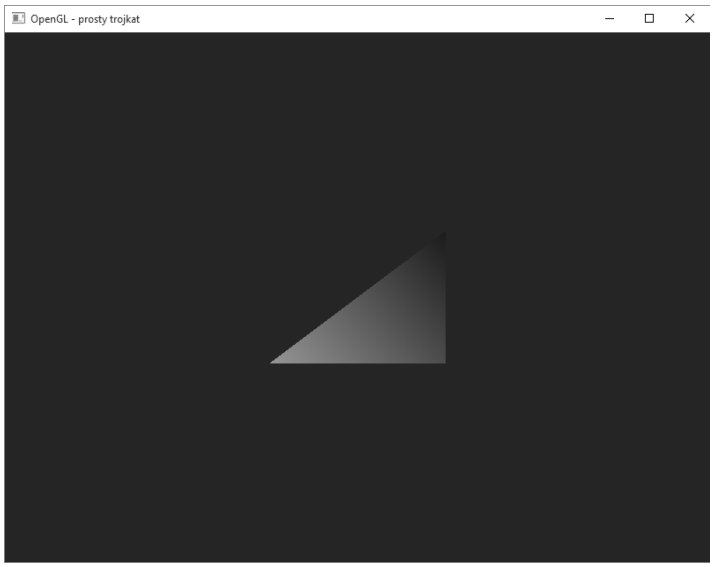

```
{
   const vec4 vertices[3] = vec4[3](vec4( 0.25, -0.25, 0.5, 1.0),
                                    vec4(-0.25, -0.25, 0.5, 1.0),
                                    vec4( 0.25, 0.25, 0.5, 1.0));
   const vec4 colors[] = vec4[3](vec4( 1.0, 0.0, 0.0, 1.0),
                                 vec4( 0.0, 1.0, 0.0, 1.0),
                                 vec4( 0.0, 0.0, 1.0, 1.0));

   // Dodaj 'offset' do umieszczonej na sztywno pozycji.
   gl_Position = vertices[gl_VertexID] + offset;

   // Prześlij w vs_color inny kolor dla każdego wierzchołka.
   vs_color = color[gl_VertexID];
}
```

Jak można zauważyć, w kodzie z listingu 3.11 pojawiła się druga tablica ze stałymi, zawierająca kolory. Dodatkowo kod wykorzystuje wartość `gl_VertexID` do przypisania zmiennej `vs_color` innego koloru. Listing 3.12 zawiera zmodyfikowaną wersję shadera fragmentów, który po prostu wykorzystuje dane wejściowe.

Listing 3.12. Określanie koloru fragmentu na podstawie położenia i danych wejściowych

```
#version 450 core

// vs_color to kolor tworzony przez shader wierzchołków.
in vec4 vs_color;

out vec4 color;

void main(void)
{
   color = vs_color;
}
```

Zastosowanie nowej pary shaderów owocuje wynikiem przedstawionym na rysunku 3.5. Kolor zmienia się płynnie między poszczególnymi wierzchołkami trójkąta.



Rysunek 3.5. Wynik działania listingu 3.12

Operacje dotyczące bufora ramki

Bufor ramki to ostatni etap potoku graficznego OpenGL. Może reprezentować widoczny obszar ekranu lub kilka innych obszarów pamięci wykorzystywanych do przechowywania wartości bazujących na pikselach, ale niedotyczących koloru. Na większości platform oznacza to okno, które widać na komputerze (a czasem nawet cały ekran w trybie pełnoekranowym). Oknem zarządza system operacyjny (a w zasadzie menedżer okien). Bufor ramki oferowany przez menedżer okien to tak zwany domyślny bufor ramki, ale można także użyć własnego bufora ramki renderującego poza widocznym obszarem. Bufor ramki przechowuje takie informacje jak miejsce zapisu danych generowanych przez shader fragmentów czy sposób ich zapisu. Za stan odpowiada tak zwany **obiekt bufora ramki** (ang. *framebuffer object*). Jako część bufora ramki, choć nie jest przechowywana w obiekcie bufora ramki, uważa się również stan operacji na pikselach.

Operacje na pikselach

Po tym, gdy shader fragmentów wygenerował dane wyjściowe z fragmentem, może się stać kilka rzeczy, zanim trafi on ostatecznie do okna, o ile w ogóle do jakiegoś okna trafi. Aplikacja może włączać lub wyłączać różne funkcjonalności. Przede wszystkim możliwe jest użycie **testu nożycowego** (ang. *scissor test*), który sprawdza fragment pod kątem istnienia w zdefiniowanym prostokącie. Jeśli znajduje się wewnątrz, będzie dalej przetwarzany; jeśli znajduje się poza, zostanie odrzucony.

Następnie pojawia się **test szablonu** (ang. *stencil test*). Porównuje on wartość referencyjną zapewnianą przez aplikację z zawartością bufora szablonu, który przechowuje dla każdego piksela poje-

dynczą[4] wartość. Zawartość bufora szablonu nie ma żadnego konkretnego znaczenia semantycznego i może służyć dowolnym celom.

Po teście szablonu pojawia się **test głębi** (ang. *depth test*). To test, w którym porównuje się wartość współrzędnej *z* fragmentu z zawartością **bufora głębi** (ang. *depth buffer*). Bufor głębi to obszar w pamięci, podobnie jak bufor szablonu, który ma wystarczająco miejsca, aby dla każdego piksela przechować jedną wartość. W tym wypadku jest to głębia (odległość od oglądającego) dotycząca każdego piksela.

Standardowo wartości w buforze głębi mieszczą się w zakresie od 0 do 1, gdzie 0 oznacza najbliższy możliwy punkt w buforze, a 1 — najdalszy. Aby określić, czy fragment znajduje się bliżej niż inne fragmenty zrenderowane w tym samym miejscu, OpenGL porównuje komponent *z* fragmentu z wartością już umieszczoną w buforze. Jeśli wartość jest mniejsza, fragment jest widoczny. Sens tego testu można zmienić. Można poprosić OpenGL o przepuszczanie fragmentów o współrzędnej *z* większej od wartości z bufora głębi, równej jej lub też różnej od tejże wartości. To, co się dzieje w teście głębi, wpływa również na to, co OpenGL realizuje w buforze szablonu.

Następnie kolor fragmentu trafia albo do mieszania, albo do etapu operacji logicznej. Wszystko zależy od tego, czy bufor ramki ma przechowywać wartości zmiennoprzecinkowe, znormalizowane czy całkowite. Jeśli zawartość bufora ramki jest zmiennoprzecinkową lub znormalizowaną wartością całkowitą, dochodzi do mieszania. Mieszanie to w OpenGL etap o bardzo dużych możliwościach konfiguracyjnych, poświęcimy mu więc osobny dział.

W dużym skrócie możemy powiedzieć, że OpenGL udostępnia wiele różnych funkcji pobierających komponenty wyjściowe z shadera fragmentów i aktualną zawartość bufora ramki, aby wyliczyć nową wartość ponownie zapisywaną w buforze ramki. Jeżeli bufor ramki zawiera nieznormalizowane wartości całkowite, wówczas można stosować operacje logiczne takie jak AND, OR lub XOR. Wynik takiej operacji ponownie trafia do bufora ramki.

Shadery obliczeniowe

W pierwszej części rozdziału omówiliśmy etapy **potoku graficznego** OpenGL. OpenGL zawiera jednak również etap **shadera obliczeniowego** (ang. *compute shader*), który w zasadzie warto traktować jako potok niezależny od wszystkich innych etapów związanych bezpośrednio z grafiką.

Shadery obliczeniowe to sposób na uzyskanie dostępu do mocy obliczeniowej drzemiącej w nowoczesnych procesorach graficznych. W odróżnieniu od ukierunkowanych graficznie shaderów wierzchołków, teselacji, geometrii i fragmentów shadery obliczeniowe należy traktować jako osobny, jednoetapowy potok. Każdy shader obliczeniowy działa jako jedna jednostka zadaniowa nazywana **elementem roboczym** (ang. *work item*); elementy te zbiera się razem w grupy nazywane **lokalnymi grupami roboczymi** (ang. *local workgroups*). Zbiory grup roboczych mogą być przesłane do OpenGL w celu ich realizacji w potoku obliczeniowym. Shader obliczeniowy nie posiada żadnych ustalonych komponentów wejściowych i wyjściowych poza kilkoma wbudowanymi zmiennymi informującymi

[4] Bufor ramki może przechowywać wiele wartości dotyczących koloru, szablonu lub głębi, jeśli zastosuje się technikę nazywaną **wielokrotnym próbkowaniem** (ang. *multi-sampling*). Tematem tym zajmiemy się w dalszej części książki.

o tym, nad którym elementem pracuje obecnie shader. Wszystkie przetwarzane przez shader dane są jawnie zapisywane w pamięci przez kod shadera — nie są w żaden sposób modyfikowane i przekazywane przez potok. Najprostszy shader obliczeniowy został przedstawiony na listingu 3.13.

Listing 3.13. Shader obliczeniowy, który nic nie robi

```
#version 450 core

layout (local_size_x = 32, local_size_y = 32) in;

void main(void)
{
    // Nic nie rób.
}
```

Pod wszystkimi innymi względami shader obliczeniowy przypomina każdy inny shader OpenGL. Aby go skompilować, należy utworzyć obiekt shadera typu `GL_COMPUTE_SHADER`, przypisać kod źródłowy za pomocą funkcji `glShaderSource()`, skompilować go przy użyciu funkcji `glCompileShader()` i dołączyć do programu, stosując funkcje `glAttachShader()` oraz `glLinkProgram()`. W ten sposób powstanie obiekt programu ze skompilowanym shaderem, który można uruchomić w dowolnym momencie.

Shader z listingu 3.13 informuje OpenGL, że wielkość lokalnej grupy to 32 na 32 elementy, ale nie realizuje później żadnych innych działań. Aby utworzyć shader wykonujący prawdziwą pracę, trzeba lepiej poznać sposób działania OpenGL, więc do tematu wrócimy w dalszej części książki.

Korzystanie z rozszerzeń OpenGL

Wszystkie przykłady przedstawione do tej pory bazowały na podstawowej funkcjonalności OpenGL. Jedną z istotnych zalet OpenGL jest to, że może być rozszerzany przez producentów sprzętu, twórców systemów operacyjnych, a nawet przez wydawców dodatkowych narzędzi. Rozszerzenia mogą mieć bardzo istotny i różnoraki wpływ na funkcjonalność OpenGL.

Rozszerzenie to dodatek do bazowej wersji OpenGL. Dostępne rozszerzenia są wymienione na stronie WWW OpenGL w tak zwanym rejestrze rozszerzeń[5]. Rozszerzenia opisane są jako lista różnic względem konkretnej specyfikacji OpenGL wraz z informacją o numerze wersji. Oznacza to, że tekst rozszerzenia opisuje, jak musi się zmienić specyfikacja głównego OpenGL, jeśli rozszerzenie jest obsługiwane. Z drugiej strony popularne i powszechnie obsługiwane rozszerzenia są z czasem „promowane" do głównej wersji OpenGL. Wynika z tego, że gdy stosujemy najnowsze wersje OpenGL, może się okazać, że nie mamy zbyt wielu interesujących rozszerzeń, bo wszystkie ważne trafiły już do głównej specyfikacji. Pełna lista rozszerzeń wraz z informacją o wersji głównego OpenGL, w której się znalazły, jest dostępna w dodatku C.

Istnieją trzy główne klasyfikacje rozszerzeń: dostawcy, EXT i ARB. Rozszerzenia dostawcy zostały napisane i zaimplementowane na sprzęcie konkretnego dostawcy. Inicjały producenta znajdują się najczęściej w nazwie rozszerzenia — AMD oznacza firmę Advanced Micro Devices, a NV — firmę NVDIA. Zdarza się, że kilku producentów obsługuje konkretne rozszerzenie, szczególnie jeśli wzrasta jego popularność. Rozszerzenia EXT powstają przy współudziale dwóch twórców sprzętu lub ich

[5] Rejestr jest dostępny pod adresem *http://www.opengl.org/registry/*.

większej liczby. Bardzo często na początku istniały jako rozszerzenia jednego producenta, ale gdy inny producent postanowił je wprowadzić (być może z drobnymi poprawkami), powstała wersja EXT. Rozszerzenia ARB stanowią oficjalną część OpenGL, ponieważ zostały zatwierdzone przez ciało standaryzujące OpenGL, czyli Architecture Review Board (ARB). Rozszerzenia tego typu są często obsługiwane przez większość głównych dostawców sprzętu i w wielu wypadkach bazują na rozszerzeniach dostawców lub EXT.

System rozszerzeń może początkowo przerażać, bo istnieją setki rozszerzeń! Z drugiej strony nowa wersja OpenGL powstaje na bazie rozszerzeń, które programiści uznali za przydatne. Mechanizm rozszerzeń pozwala ocenić konkretną propozycję w praktyce. Te, które się sprawdzą, trafiają do głównej specyfikacji; te mniej udane pozostają rozszerzeniami. Ten proces „selekcji naturalnej" zapewnia przenoszenie do głównego OpenGL tylko tych nowych funkcji, które przeszły wcześniej „chrzest bojowy".

Przydatnym narzędziem umożliwiającym łatwe i szybkie sprawdzenie, które rozszerzenia OpenGL obsługuje sprzęt i sterownik zainstalowany w komputerze, jest OpenGL Extensions Viewer firmy Realtech VR. Narzędzie to można pobrać bezpłatnie ze stron firmy (patrz rysunek 3.6).

Wzbogacanie OpenGL rozszerzeniami

Przed użyciem rozszerzenia **musimy** się upewnić, że jest ono obsługiwane przez implementację OpenGL działającą na naszym komputerze. W celu sprawdzenia obsługi rozszerzenia można skorzystać z dwóch dostępnych funkcji. Aby poznać liczbę obsługiwanych rozszerzeń, wywołaj funkcję `glGetIntegerv()` z parametrem `GL_NUM_EXTENSIONS`. Następnie pobierz nazwę każdego obsługiwanego rozszerzenia, wywołując funkcję o następującym prototypie:

```
const GLubyte* glGetStringi(GLenum name,
                            GLuint index);
```

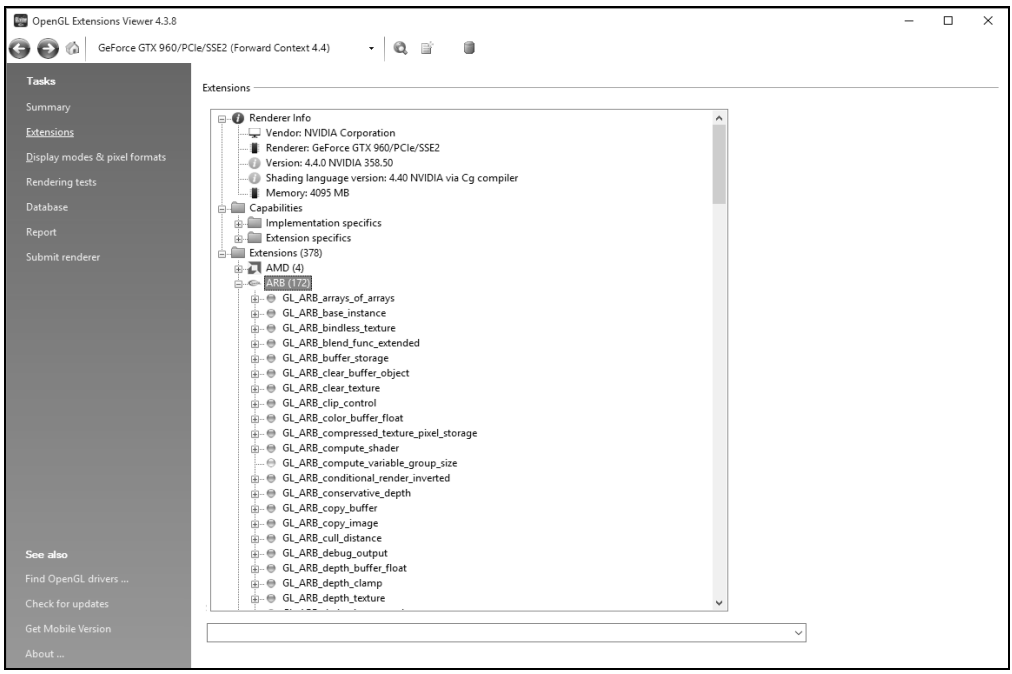
W parametrze `name` przekaż wartość `GL_EXTENSIONS`, a w parametrze `index` — numer rozszerzenia (od 0 do wartości o jeden mniejszej niż liczba rozszerzeń). Funkcja zwraca nazwę rozszerzenia jako tekst. Aby dowiedzieć się, czy konkretne rozszerzenie jest obsługiwane, trzeba pobrać wszystkie nazwy rozszerzeń i sprawdzić, czy jego nazwa pasuje do poszukiwanej. Kod źródłowy dołączony do książki zawiera funkcję pomocniczą wykonującą właśnie to zadanie. Prototyp funkcji `sb7IsExten` ➥`sionSupported()` ma postać:

```
int sb7IsExtensionSupported(const char * extname);
```

Funkcja jest zadeklarowana w nagłówku `<sb7ext.h>`. Przyjmuje nazwę rozszerzenia i zwraca wartość inną od 0, jeśli rozszerzenie jest obsługiwane, lub 0, jeśli nie jest. Aplikacja powinna zawsze sprawdzać wszystkie rozszerzenia, z których chce skorzystać.

Rozszerzenia wzbogacają OpenGL na jeden z czterech sposobów (czasem jest to połączenie kilku wymienionych sposobów):

- Czynią legalnymi rzeczy, które były nielegalne wcześniej, czyli po prostu znoszą wybrane ograniczenia specyfikacji OpenGL.
- Dodają tokeny lub rozszerzają zakres wartości, które można przekazać jako parametry istniejących funkcji.
- Rozszerzają GLSL o nową funkcjonalność, wbudowane funkcje, zmienne lub typy danych.
- Dodają do OpenGL całkowicie nowe funkcje.



Rysunek 3.6. Narzędzie OpenGL Extensions Viewer firmy Realtech VR

W pierwszym przypadku, czyli sytuacji zalegalizowania czegoś, co dawniej uznawane było za błąd, aplikacja nie musi tak naprawdę robić nic poza stosowaniem bez przeszkód nowego zachowania (oczywiście po upewnieniu się, że rozszerzenie jest obsługiwane). Podobnie jest z drugim przypadkiem, ponieważ wystarczy po prostu zacząć używać nowych tokenów we właściwych funkcjach, jeśli tylko zna się ich wartości. Wartości nowych tokenów znajdują się w specyfikacji rozszerzenia, nie ma ich więc w plikach nagłówkowych głównego profilu OpenGL.

Aby włączyć obsługę rozszerzenia w GLSL, trzeba w pierwszych wierszach shadera poinformować kompilator, że chce się użyć konkretnej funkcjonalności. Jeśli w GLSL chcemy wykorzystać hipotetyczne rozszerzenie `GL_ABC_nowa_funkcja`, musimy na początku shadera umieścić wiersz:

```
#extension GL_ABC_nowa_funkcja : enable
```

Informujemy w ten sposób, że chcemy użyć rozszerzenia w shaderze. Jeśli kompilator rozumie rozszerzenie, pozwoli skompilować kod nawet w sytuacji, gdy sprzęt go nie obsługuje. Dopiero próba użycia w kodzie shadera elementów specyficznych dla rozszerzenia spowoduje zgłoszenie ostrzeżenia. Najczęściej rozszerzenia GLSL dodają token preprocesora, aby poinformować o ich obsłudze. Na przykład `GL_ABC_nowa_funkcja` w sposób niejawny doda definicję:

```
#define GL_ABC_nowa_funkcja 1
```

Oznacza to, że w kodzie można użyć poniższej konstrukcji:

```
#if GL_ABC_nowa_funkcja
    // Użyj nowej funkcji.
#else
    // Użyj innego rozwiązania, aby pominąć brakującą funkcjonalność.
#endif
```

W ten sposób można warunkowo kompilować kod wykorzystujący rozszerzenie lub je pomijający — w zależności od tego, co wspiera karta graficzna i wykorzystywana implementacja OpenGL. Jeśli shader bezwzględnie wymaga użycia rozszerzenia i nie będzie bez niego działał, można wstawić bardziej agresywną wersję kodu:

```
#extension GL_ABC_nowa_funkcja : require
```

Jeśli implementacja OpenGL nie obsługuje `GL_ABC_nowa_funkcja`, shadera nie uda się skompilować i zostanie zgłoszony błąd w wierszu z dyrektywą `#extension`. Innymi słowy, rozszerzenia GLSL to funkcjonalności, do których trzeba się jawnie zapisać — aplikacja musi[6] poinformować kompilator o chęci zastosowania rozszerzenia.

Ostatni punkt to rozszerzenia wprowadzające do OpenGL nowe funkcje. Na większości platform nie mamy bezpośredniego dostępu do sterownika OpenGL, więc funkcje rozszerzeń nie pojawiają się „magicznie" na liście funkcji dostępnych do wywołania. Trzeba „poprosić" sterownik OpenGL o **wskaźnik na funkcję**, z której chce się skorzystać. Wskaźniki na funkcje deklaruje się w dwóch częściach. Pierwsza to definicja typu wskaźnika na funkcję, a druga to zmienna przechowująca sam wskaźnik. Rozważmy następujący przykład:

```
typedef void
(APIENTRYP PFNGLDRAWTRANSFORMFEEDBACKPROC) (GLenum mode,
                                            GLuint id);
PFNGLDRAWTRANSFORMFEEDBACKPROC glDrawTransformFeedback = NULL;
```

Zadeklarowaliśmy `PFNGLDRAWTRANSFORMFEEDBACKPROC` jako wskaźnik na funkcję przyjmującą parametry `GLenum` i `GLuint`. Następnie deklarujemy zmienną `glDrawTransformFeedback` o takim właśnie typie. W zasadzie na większości platform deklaracja funkcji `glDrawTransformFeedback()` wygląda tak, jak ją tu przedstawiono. Wydaje się to skomplikowane, ale poniższe pliki nagłówkowe zawierają deklaracje wszystkich prototypów funkcji, typów wskaźników na funkcje i tokenów dostępnych we wszystkich rozszerzeniach OpenGL:

```
#include <glext.h>
#include <glxext.h>
#include <wglext.h>
```

Pliki można znaleźć na stronie WWW dotyczącej rejestru rozszerzeń OpenGL. Nagłówek `glext.h` zawiera zarówno standardowe rozszerzenia OpenGL, jak i wiele rozszerzeń producentów sprzętu. Nagłówek `wglext.h` zawiera rozszerzenia specyficzne dla systemu Windows, z kolei nagłówek `glxext.h` — rozszerzenia specyficzne dla systemu okien X (używanego w systemie Linux i wielu innych implementacjach bazujących na systemie UNIX).

Sposób odpytania się o adres funkcji rozszerzenia zależy od wykorzystywanego systemu. Framework aplikacyjny dołączony do książki ukrywa te różnice, oferując funkcję pomocniczą zadeklarowaną w pliku nagłówkowym `<sb7ext.h>`. Funkcja `sb7GetProcAddress()` ma następujący prototyp:

```
void * sb7GetProcAddress(const char * funcname);
```

Parametrem jest nazwa funkcji rozszerzenia, której adres chcemy pobrać. Jeśli funkcja istnieje, zostanie zwrócony jej adres. Jeśli nie istnieje, zostanie zwrócona wartość `NULL`. To, że OpenGL

[6] W praktyce wiele implementacji włącza domyślnie funkcjonalności niektórych rozszerzeń i nie wymaga podawania ich w kodzie shadera. Z drugiej strony wykorzystanie tego faktu naraża programistę na to, że kod nie będzie działał prawidłowo w innych sterownikach OpenGL. Z tego powodu warto za każdym razem umieścić informację o rozszerzeniu w kodzie shadera.

zwróci poprawny wskaźnik do funkcji, która stanowi część rozszerzenia, nie oznacza jednocześnie, że rozszerzenie jest dostępne. Czasem funkcja stanowi część kilku rozszerzeń, a czasem twórca sterownika udostępnia więcej funkcji, niż jest w stanie obsłużyć sprzęt w konkretnej wersji. Zawsze w celu sprawdzenia dostępności rozszerzenia korzystaj z oficjalnego mechanizmu sprawdzania lub funkcji `sb7IsExtensionSupported()`.

Podsumowanie

W tym rozdziale udaliśmy się w podróż po potoku graficznym OpenGL. Pokrótce opisaliśmy każdy z głównych etapów i przedstawiliśmy przykłady jego użycia, choć żaden z nich nie był szczególnie imponujący. W paru miejscach przemknęło kilka dodatkowych, a przydatnych funkcji OpenGL, choć staraliśmy się przedstawić przykłady, zdradzając szczegóły w stopniu minimalnym. Omówiliśmy rozbudowanie podstawowej funkcjonalności OpenGL za pomocą rozszerzeń. Część przykładów w dalszej części książki do prawidłowego działania wymaga dodatkowych rozszerzeń. W następnych rozdziałach nieco dokładniej omówimy podstawy grafiki komputerowej, aby po pewnym czasie ponownie, choć tym razem znacznie dokładniej, przyjrzeć się każdemu etapowi potoku graficznego. Omówimy wtedy kwestie, które pominęliśmy w tym krótkim wprowadzeniu.

Skorowidz

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

Biblioteka OpenGL jest potężnym systemem graficznym, doskonałym API do generowania grafiki trójwymiarowej w czasie rzeczywistym. System ten nadaje się znakomicie do wizualizacji wszelkiego rodzaju odwzorowań zjawisk fizycznych czy obiektów technicznych, a także do przedstawiania symulacji ze zmieniającymi się parametrami. Często jest wykorzystywany do pisania gier komputerowych. Daje możliwość tworzenia świetnej grafiki na wiele różnych platform z wykorzystaniem tych samych zestawów instrukcji. Co ważne, OpenGL jest całkowicie darmową biblioteką, a dostępność licznych rozszerzeń znakomicie zwiększa zakres jej zastosowań.

Niniejsza książka stanowi doskonałe wprowadzenie w tematykę OpenGL dla każdego programisty, nawet dla osób niezbyt biegłych w zagadnieniach grafiki komputerowej. Zawiera opis całego głównego API, kluczowych rozszerzeń i wszystkich typów shaderów z uwzględnieniem najnowszych elementów biblioteki. Wyjaśniono tu zasady działania OpenGL i opisano zagadnienia potoków graficznych. Stopniowo czytelnik jest zaznajamiany z coraz bardziej złożonymi technikami. W książce znalazły się liczne przykłady kodu działającego na kilku popularnych platformach. Warto podkreślić, że autorzy poza API przedstawili również najlepsze praktyki programistyczne.

W tej książce opisano między innymi:

- podstawy (w tym matematyczne) grafiki 3D czasu rzeczywistego
- najważniejsze techniki renderowania, przekształcania i teksturowania obiektów
- shadery i język GLSL (OpenGL Shading Language)
- kwestie zarządzania danymi i kontroli dostępu do tych danych
- techniki budowania większych aplikacji i wdrażania ich na wielu platformach
- rendering zaawansowany: symulację oświetlenia i efekty artystyczne
- sposoby poprawiania wydajności, redukcji narzutu CPU i analizy zachowania GPU
- nowości w OpenGL, takie jak kompresja tekstur, rysowanie tekstu, rendering czcionek za pomocą pól odległości, wysokiej jakości filtrowanie tekstur i użycie OpenMP

Graham Sellers — architekt oprogramowania w firmie AMD. Reprezentant AMD w OpenGL Architecture Review Board (OpenGL ARB, zrzeszenie firm zajmujących się rozwojem OpenGL), współautor wielu rozszerzeń oraz głównej specyfikacji OpenGL. Właściciel kilku patentów związanych z przetwarzaniem obrazów i grafiką komputerową.

Richard S. Wright Jr. — starszy inżynier oprogramowania w Software Bisque. Twórca oprogramowania na potrzeby astronomii z wykorzystaniem OpenGL.

Nicholas Haemel — dyrektor oprogramowania kamerowego w firmie NVIDIA. Reprezentant firmy NVIDIA w konsorcjum Khronos Group, autor wielu rozszerzeń OpenGL.

OpenGL? Kreatywnych ogranicza tylko wyobraźnia!

Helion

43383 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

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

ISBN 978-83-283-2107-6

9 788328 321076

Informatyka w najlepszym wydaniu

cena: 119,00 zł