



OpenGL®

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

Open  
GL

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.  
Copyright © 2016 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.  
Polish language edition published by HELION S.A. Copyright © 2016.

OpenGL® is a registered trademark of Silicon Graphics Inc. and is used by permission of Khronos.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION  
ul. Kościuszki 1c, 44-100 GLIWICE  
tel. 32 231 22 19, 32 230 98 63  
e-mail: [helion@helion.pl](mailto: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.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

# Spis treści

<b>Spis rysunków .....</b>	<b>11</b>
<b>Spis tabel .....</b>	<b>17</b>
<b>Spis listingów .....</b>	<b>19</b>
<b>O autorze .....</b>	<b>27</b>
<b>Przedmowa .....</b>	<b>29</b>
<b>Wstęp .....</b>	<b>31</b>

## **Część I Podstawy**

<b>Rozdział 1. Wprowadzenie .....</b>	<b>39</b>
OpenGL i potok graficzny .....	39
Początki i ewolucja OpenGL .....	41
Prymitywy, potoki i piksele .....	44
Podsumowanie .....	45
<b>Rozdział 2. Pierwszy program OpenGL .....</b>	<b>47</b>
Tworzenie prostej aplikacji .....	47
Korzystanie z shaderów .....	49
Rysowanie pierwszego trójkąta .....	55
Podsumowanie .....	57
<b>Rozdział 3. Wzdłuż potoku graficznego .....</b>	<b>59</b>
Przekazywanie danych do shadera wierzchołków .....	59
Przekazywanie danych z jednego etapu do drugiego .....	61
Teselacja .....	64

Shadery geometrii .....	66
Składanie prymitywów, przycinanie i rasteryzacja .....	68
Shadery fragmentów .....	71
Operacje dotyczące bufora ramki .....	74
Shadery obliczeniowe .....	75
Korzystanie z rozszerzeń OpenGL .....	76
Podsumowanie .....	80
<b>Rozdział 4. Matematyka w grafice 3D .....</b>	<b>81</b>
Czy to ten okrutny rozdział o matematyce? .....	81
Krótki kurs matematyki dotyczącej grafiki 3D .....	82
Zasady działania przekształceń .....	91
Interpolacja, linie, krzywe i powierzchnie parametryczne .....	106
Podsumowanie .....	112
<b>Rozdział 5. Dane .....</b>	<b>115</b>
Bufory .....	115
Zmienne typu uniform .....	129
Blok magazynowy shadera .....	147
Liczniki niepodzielne .....	152
Tekstury .....	156
Podsumowanie .....	196
<b>Rozdział 6. Shadery i programy .....</b>	<b>197</b>
Opis języka .....	197
Kompilacja, łączenie i sprawdzanie programów .....	207
Podsumowanie .....	222
<b>Część II Zgłębianie tematu</b>	
<b>Rozdział 7. Przetwarzanie wierzchołków i polecenia rysowania .....</b>	<b>225</b>
Przetwarzanie wierzchołków .....	225
Polecenia rysowania .....	231
Przechowywanie przekształconych wierzchołków .....	254
Przycinanie .....	267
Podsumowanie .....	273
<b>Rozdział 8. Przetwarzanie prymitywów .....</b>	<b>275</b>
Teselacja .....	275
Shadery geometrii .....	297
Podsumowanie .....	322

<b>Rozdział 9. Przetwarzanie fragmentów i bufor ramki .....</b>	<b>323</b>
Shadery fragmentów .....	323
Testy dla poszczególnych fragmentów .....	326
Wyjście koloru .....	336
Rendering pozaekranowy .....	342
Antyaliasing .....	360
Zaawansowane formaty bufora ramki .....	373
Sprite'y punktów .....	390
Pobieranie utworzonego obrazu .....	398
Podsumowanie .....	404
<b>Rozdział 10. Shadery obliczeniowe .....</b>	<b>405</b>
Korzystanie z shaderów obliczeniowych .....	405
Przykłady .....	414
Podsumowanie .....	433
<b>Rozdział 11. Zaawansowane zarządzanie danymi .....</b>	<b>435</b>
Eliminacja dowiązań .....	435
Rzadko wypełnione tekstury .....	440
Kompresja tekstur .....	445
Upakowane formaty danych .....	454
Wysokiej jakości filtrowanie tekstur .....	455
Podsumowanie .....	458
<b>Rozdział 12. Sterowanie potokiem graficznym i monitorowanie go .....</b>	<b>459</b>
Kolejki .....	459
Synchronizacja w OpenGL .....	477
Podsumowanie .....	481
<b>Część III W praktyce</b>	
<b>Rozdział 13. Techniki renderowania .....</b>	<b>485</b>
Modele oświetlenia .....	485
Rendering niefotorealistyczny .....	520
Alternatywne metody renderowania .....	524
Grafika dwuwymiarowa .....	551
Podsumowanie .....	561
<b>Rozdział 14. OpenGL o wysokiej wydajności .....</b>	<b>563</b>
Optymalizacja wydajności CPU .....	563
OpenGL o niskim narzucie .....	576
Narzędzia do analizy wydajności .....	595
Podsumowanie .....	615

<b>Rozdział 15. Debugowanie i stabilność .....</b>	<b>617</b>
Debugowanie własnych aplikacji .....	617
Bezpieczeństwo i szybkość .....	623
Podsumowanie .....	626
<b>Dodatek A Narzędzia związane z przykładami .....</b>	<b>627</b>
<b>Dodatek B Format pliku SBM .....</b>	<b>633</b>
<b>Dodatek C Funkcje i wersje OpenGL .....</b>	<b>641</b>
<b>Dodatek D Słowniczek .....</b>	<b>663</b>
<b>Skorowidz .....</b>	<b>669</b>

## 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 przyjrzeliliśmy 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 szczególnie, 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 fetching*). 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 pozostaniemy 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 prześlemy 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 osi 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 rozbić 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 rozbić 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 płatem, 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 płatek, 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 liczbą 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:

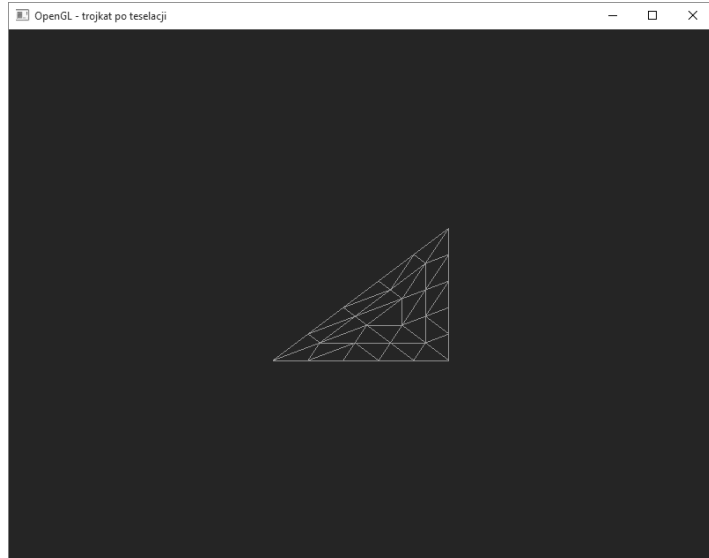
```
void glPolygonMode(GLenum face,
                  GLenum mode);
```

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();
    }
}
```

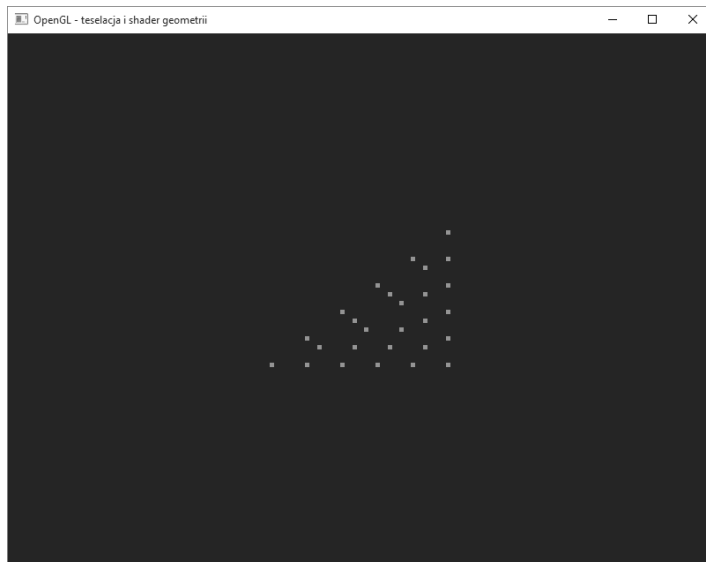
Shader przedstawiony na listingu 3.9 jest bardzo prostym shaderem przekazującym dane dalej, ale przy okazji konwertującym trójkąt na punkty, co umożliwia zobaczenie ich wierzchołków. Pierwszy kwalifikator wskazuje, że shader geometrii oczekuje trójkątów jako danych wejściowych. Drugi kwalifikator informuje, 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. Shader 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 „wewnętrzna” i „zewnętrzna”. 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<sup>1</sup> 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

<sup>1</sup> 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 wierzchołki w przestrzeni okna, a  $x_d$ ,  $y_d$  i  $z_d$  są współrzędnymi wejściowymi wierzchołki 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:

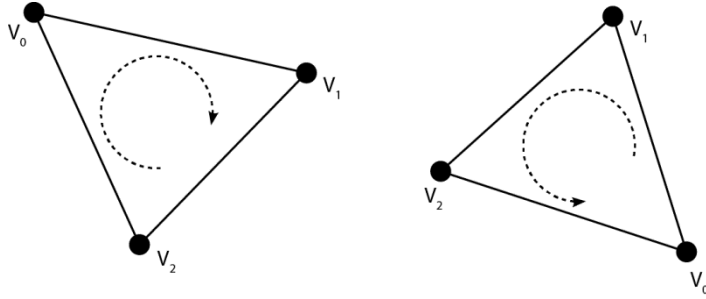
$$a = \frac{1}{2} \sum_{i=0}^{n-1} x_w^i y_w^{i \oplus 1} - x_w^{i \oplus 1} y_w^i$$

We wzorze  $x_w^i$  i  $y_w^i$  to współrzędne  $i$ -tego wierzchołki 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 stałą `GL_CULL_FACE`. Domyślnie OpenGL usunie trójkąty skierowane tyłem. Aby zmienić rodzaj usuwanych trójkątów, wywołaj funkcję `glCullFace()` i przekaz jej wartość `GL_FRONT`, `GL_BACK` lub `GL_FRONT_AND_BACK`.

Ponieważ punkty i linie nie mają żadnego obszaru geometrycznego<sup>2</sup>, 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**<sup>3</sup> (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

<sup>2</sup> 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.

<sup>3</sup> 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 wzorec 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, zastosujemy shader wierzchołków z listingu 3.3 i zmodyfikujemy go tak, aby każdemu wierzchołkowi przypisywał inny kolor (patrz listing 3.11).

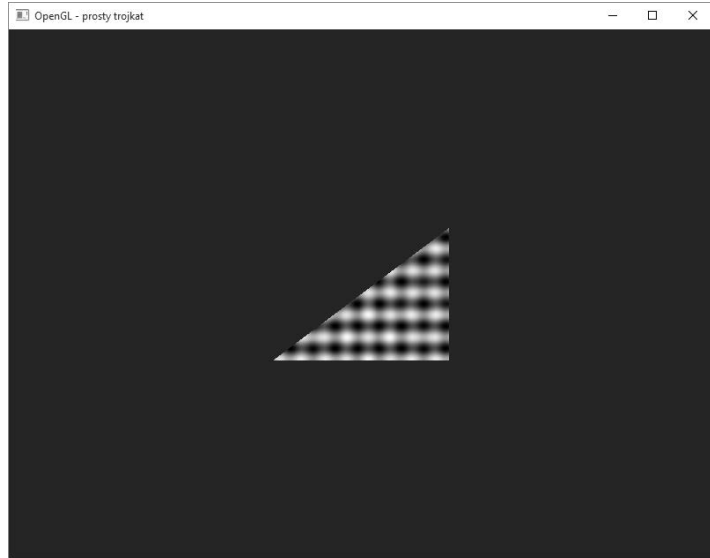
**Listing 3.11.** Shader wierzchołków z danymi wyjściowymi

```
#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

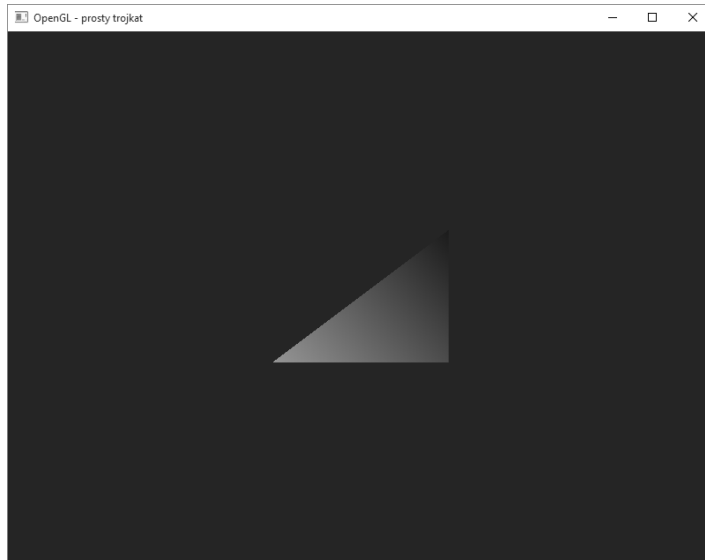
// vs_color to kolor tworzony przez shader wierzchołków.
in vec4 vs_color;

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 nie dotyczących 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<sup>4</sup> 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 przeprowadzanie fragmentów o współrzędnej z większej od wartości z bufora głębi, równej jej lub też różnej od tejej 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

<sup>4</sup> 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 różnorodny wpływ na funkcjonalność OpenGL.

Rozszerzenie to dodatek do bazowej wersji OpenGL. Dostępne rozszerzenia są wymienione na stronie WWW OpenGL w tak zwanym rejestrze rozszerzeń<sup>5</sup>. 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 NVIDIA. Zdarza się, że kilku producentów obsługuje konkretne rozszerzenie, szczególnie jeśli wzrasta jego popularność. Rozszerzenia EXT powstają przy współdziałaniu dwóch twórców sprzętu lub ich

<sup>5</sup> 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);
```

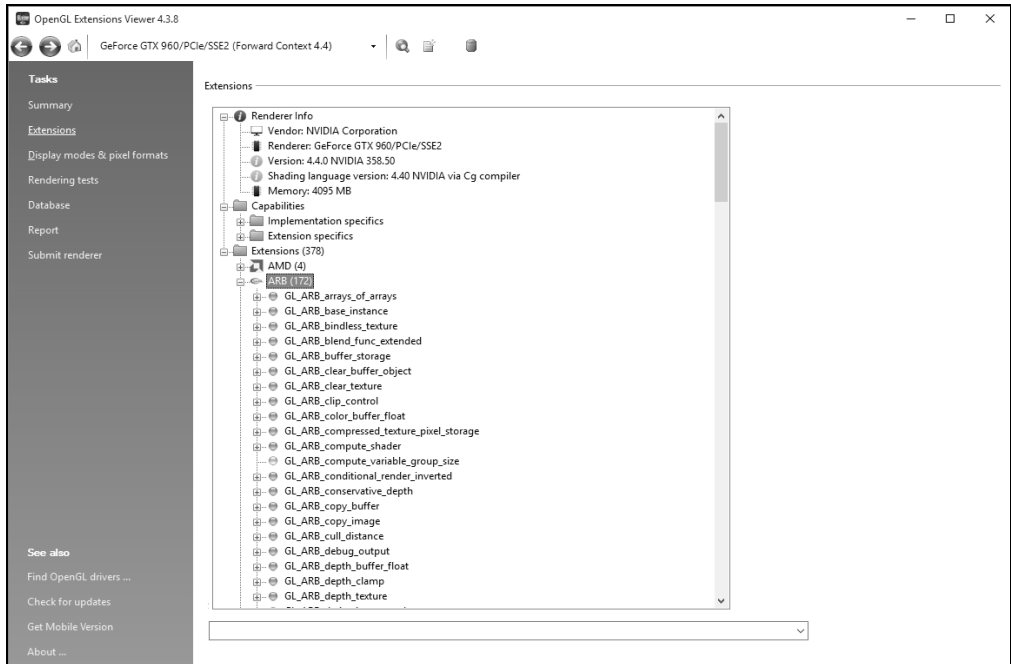
W parametrze `name` przekaż wartość `GL_EXTENSIONS`, a w parametrze `index` — numer rozszerzenia (od 0 do wartości o jeden mniejszej niż liczba rozszerzeń). Funkcja zwraca nazwę rozszerzenia jako tekst. Aby dowiedzieć się, czy konkretne rozszerzenie jest obsługiwane, trzeba pobrać wszystkie nazwy rozszerzeń i sprawdzić, czy jego nazwa pasuje do poszukiwanej. Kod źródłowy dołączony do książki zawiera funkcję pomocniczą wykonującą właśnie to zadanie. Prototyp funkcji `sb7IsExtensionSupported()` 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 niejawni 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<sup>6</sup> 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
(APIENTRY 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 <wglxext.h>
```

Pliki można znaleźć na stronie WWW dotyczącej rejestru rozszerzeń OpenGL. Nagłówek `glxext.h` zawiera zarówno standardowe rozszerzenia OpenGL, jak i wiele rozszerzeń producentów sprzętu. Nagłówek `wglxext.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

<sup>6</sup> 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. Pokrótkę 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

## A

AFR, Alternate Frame Rendering, 610  
aktualizacja  
  atrybutu wierzchołka, 60  
  cząsteczki, 565  
  członków stada, 428  
  macierzy rzutowania, 144  
  stada, 430  
  zawartości bufora, 119  
albedo rozproszenia i odbicia, 493  
algorytm stada, 414, 425, 433  
alokacja pamięci, 116  
analiza wydajności, 595  
animacja  
  koloru, 49  
  zbioru Julii, 542  
antialiasing, 360, 362  
  poprzez filtrację, 361  
  z wielokrotnym próbkowaniem, 363  
API, Application Programming Interface, 39  
aplikacja OpenGL, 48  
ARB, 76  
atrybuty wierzchołka, 59, 128, 636  
automatyczne pobieranie danych, 242  
AZDO, 33, 576

## B

bariera, 412  
  pamięciowa, 151  
  sterująca przepływem, 412  
bezpieczeństwo, 623

biblioteka  
  GLFW, 33  
  SDL, 33  
  vmath, 89  
biblioteki graficzne, 41  
blok  
  interfejsu, 62  
  magazynu shadera, 148  
  uniform, 134, 135, 141  
blokada przegubu, 98  
bloki  
  magazynowe shadera, 147  
  uniform, 132  
boolowskie zapytania o okluzję, 467  
brutalna siła, 550  
budowanie buforów pakietów, 569  
bufor, 115  
  głębi, 75  
  kopiowanie danych, 121  
  licznika niepodzielnego, 153  
  pakietów, 568, 569, 575  
  pakietu, 571  
  poleceń, 595  
  ramki, 74, 347, 354, 386  
  wielopróbkowy, 532  
  wypełnianie danymi, 121

## C

cel, 116  
  tekstury, 158, 160  
cienie, 513

cieniowanie  
 Gourauda, 488  
 komórkowe, 521  
 kreskówkowe, 523  
 opóźnione, 524, 529–531  
 Phong, 490, 492  
 próbek, 368  
 culling, 70  
 obiektów, 583  
 czasomierz, 468  
 czcionki bitmapowe, 559  
 czworobok, 277, 315

**D**

dane, 115  
 indeksowe, 635  
 skompresowane, 448  
 wejściowe shadera wierzchołków, 226  
 wierzchołków, 636  
 debugger, 599  
 debugowanie, 617  
 deklaracja  
 atrybutu wierzchołka, 60  
 bloku magazynu shadera, 148  
 bloku uniform, 132, 133  
 próbki, 436  
 właściwości materiału, 579  
 demo Unreal, 43  
 długość wektora, 87  
 dodanie płaszczyzny, 550  
 domyślny blok uniform, 196  
 dopasowanie interfejsu, 214  
 dostawca, 76  
 dostęp do  
 buforów, 588  
 liczników niepodzielnych, 155  
 mapowanego bufora, 605  
 pamięci, 150, 189  
 tablic tekstur, 178  
 trwale mapowanego bufora, 591  
 dostrajanie aplikacji, 602  
 dowiązanie, 116  
 buforów wierzchołków, 123  
 tekstury, 156  
 działanie shadera wierzchołków, 230  
 dziedzina płata, 294

dzielenie rzutowe, 69  
 dziennik kompilacji, 209

**E**

efekt  
 mgły, 521  
 obwódki świetlnej, 495, 562  
 rozbłysku, 383, 385  
 skybox, 508, 510  
 teselacji, 293  
 efekty atmosferyczne, 517  
 efektywne mapowanie bufora, 603  
 element roboczy, 75  
 elementy opcjonalne, 33  
 eliminacja dowiązań, 435  
 ewolucja OpenGL, 41  
 EXT, 76

**F**

filtr prostokątny, 420  
 filtracja, 168  
 liniowa, 169, 171  
 mipmapowa, 170  
 tekstur, 168, 172, 455  
 format  
 BPTC, 190  
 EAC, 191  
 ETC2, 191  
 KTX, 162  
 RGTC, 190, 446  
 SBM, 633  
 formaty  
 bufora ramki, 373  
 danych obrazu, 182  
 kompresji tekstur, 191  
 zmiennoprzecinkowe, 375  
 fragment, 71  
 fraktal, 539  
 Julii, 594  
 funkcja  
 application::startup(), 52  
 atomicCounterDecrement(), 154  
 EmitPrimitive(), 303  
 EmitVertex(), 303  
 faceforward(), 204

floatBitsToInt(), 207  
 fma(), 206  
 frexp(), 206  
 glAttachShader(), 52, 76  
 glBindBufferBase(), 257  
 glBindImageTexture(), 181  
 glBindSampler(), 166  
 glBindVertexArray(), 53  
 glBufferData(), 148  
 glBufferStorage(), 117  
 glBufferSubData(), 118–121, 137  
 glClearBufferfv(), 48  
 glClearBufferSubData(), 121  
 glClearNamedBufferSubData(), 121  
 glClearTexSubImage(), 157  
 glClientWaitSync(), 479  
 glCompileShader(), 52  
 glCopyBufferSubData(), 122  
 glCreateBuffers(), 116  
 glCreateProgram(), 52  
 glCreateShader(), 52  
 glCreateShaderProgramv(), 214  
 glCreateTextures(), 156  
 glCullFace(), 71  
 glDeleteShader(), 52  
 glDeleteVertexArrays(), 53  
 glDrawArrays(), 55, 57, 144, 266  
 glDrawArraysIndirect(), 248  
 glDrawArraysInstanced(), 238  
 glDrawElementsIndirect(), 248  
 glDrawElementsInstanced(), 238  
 glEnableVertexAttribArray(), 124  
 glGenProgramPipelines(), 212  
 glGenQueries(), 460  
 glGetActiveUniformsiv(), 137  
 glGetError(), 460  
 glGetIntegerv(), 625  
 glGetInternalFormativ(), 192  
 glGetProgramInfoLog(), 211  
 glGetProgramInterfaceiv(), 215  
 glGetProgramiv(), 211  
 glGetShaderInfoLog(), 211  
 glGetTexLevelParameteriv(), 192  
 glGetTextureHandleARB(), 436  
 glGetTextureSamplerHandleARB(), 436  
 glInvalidateTexImage(), 613  
 glInvalidateTexSubImage(), 613  
 glLinkProgram(), 52, 76  
 glMapBuffer(), 120  
 glMapNamedBuffer(), 119, 120  
 glMemoryBarrier(), 151, 190  
 glMultiDrawArraysIndirect(), 248, 580  
 glMultiDrawArraysIndirectCountARB(), 581  
 glMultiDrawElementsIndirectCountARB(), 581  
 glNamedBufferSubData(), 118  
 glObjectLabel(), 622  
 glPointSize(), 54, 57, 68  
 glPolygonMode(), 286  
 glProgramBinary(), 222  
 glReadPixels(), 626  
 glSamplerParameterf(), 166  
 glSamplerParameteri(), 166  
 glShaderSource(), 52, 76  
 glTexBuffer(), 265  
 glTexPageCommitmentARB(), 441  
 glTexStorage2D(), 156  
 glTextureSubImage2D(), 157, 170  
 glTextureView(), 193  
 glUnmapBuffer(), 119  
 glVertexArrayAttribBinding(), 123, 126  
 glVertexAttribFormat(), 454  
 glVertexAttribPointer(), 607  
 glViewport(), 144  
 glWaitSync(), 479, 480, 481  
 imulExtended(), 206  
 intBitsToFloat(), 207  
 ldexp(), 206  
 main, 50  
 matrixCompMult(), 204  
 memoryBarrier(), 152  
 memoryBarrier(), 156  
 mix, 109  
 not(), 204  
 outerProduct(), 204  
 packDouble2x32(), 207  
 packet\_stream::DrawElements, 571  
 packet\_stream::EnableDisable, 572  
 reflect(), 204  
 refract(), 204  
 render(), 49, 54  
 RenderSimplifiedObject(), 462  
 smoothstep, 457  
 texelFetch(), 159  
 texture(), 607  
 textureSize(), 161

## funkcja

- umulExtended(), 206
- vmath::rotate, 98
- vmath::frustum, 106

## funkcje

- matematyczne, 205
- mieszające, 337
- niepodzielne, 150
- OpenGL, 641–654
- porównywania głębi, 333
- przeciążone, 161
- szablonu, 330
- wbudowane, 203
- wykładnicze, 206
- zewnątrzne, 211
- związane z modyfikacją danych, 206

**G**

G-bufor, 526, 528

## generowanie

- danych, 564
- fraktala Julii, 592
- G-bufora, 524
- geometrii, 306
- poziomów mipmap, 172
- skompresowanych danych, 448
- tekstury, 156
- zadań w GPU, 581

geometria, 303, 306

GLSL, 50

## głębina

- ogniskowej, 421
- ostrości, 420, 422, 424

główny profil OpenGL, 42

GPU, Graphics Processing Unit, 40, 610

GPU PerfStudio, 598, 601

gradient, 456

grafika 3D, 81

- dwuwymiarowa, 551

## grupy robocze

- globalne, 407
- lokalne, 407

gwiazdne pole, 391

**H**

harmonogramowane fragmentami czasu, 411

hazard, 589

- pamięciowy, 150

- wynikający z wyścigu, 413

**I**

## iloczyn

- skalarny, 85

- wektorowy, 70, 86

indeksowane polecenia rysowania, 231

## informacje

- o interfejsie, 216

- o rozszerzeniu, 79

- z kompilatora, 208

## inicjalizacja

- bufora, 118

- G-bufora, 525

- tekstur, 156

- tekstury tablicowej, 176

interfejs, 214

interpolacja, 106, 324, 325

- gładka, 457, 458

- Hermite'a, 206

- liniowa, 107, 456

- wektora, 111

interpolowanie, 72

iteracyjny algorytm stada, 426

izolinia, 281

**J**

## jednostka

- obrazu, 181

- renderingu, 44

język GLSL, 159, 197

**K**

kanał alfa, 422

kąty Eulera, 98

Khronos, 33

kierunek rysowania, 70

klasa formatu, 195

klasy danych obrazu, 183

kodowanie RGTC, 447–450



kolejka, 459  
     CPU, 595  
     programowa, 595  
     sprzętowa, 595  
 kolejność rysowania, 285  
 kolor, 336  
     fragmentu, 73  
 komentarz, 637  
 kompilacja, 34  
     programów, 207  
     shaderów, 51, 608  
 kompletność  
     bufora ramki, 354  
     dołączenia, 354  
 kompresja, 192  
     RGTC, 446, 448, 452  
     tekstur, 190, 445  
 komunikacja shadera obliczeniowego, 411  
 komunikat, 620  
 konfiguracja  
     bufora licznika, 153  
     bufora ramki, 345  
     macierzy cienia, 515  
     mapowania bufora, 566  
     potoku, 213  
     shadera geometrii, 304, 308  
     wielopróbkowego bufora ramki, 365  
     wierzchołków, 261  
 konsumowanie G-bufora, 527  
 konteksty debugowania, 617  
 kontenery, 606  
 kopiowanie  
     danych do tekstury, 401  
     danych między buforami ramki, 400  
     obrazu, 360  
 korekcja gamma, 388  
 krzywa, 107  
     Béziara drugiego stopnia, 108  
     Béziara piątego stopnia, 110  
     Béziara trzeciego stopnia, 109  
     B-sklejana, 110  
     gamma, 389  
 kwalifikator  
     danych obrazu, 183  
     formatu, 181  
     layout, , 60 130, 147  
     layout shadera geometrii, 299  
     magazynowy, 324

kwaternion, 99, 101

## L

LDR, Low Dynamic Range, 378  
 liczby całkowite, 386  
 licznik  
     niepodzielny, 152, 154  
     wydajności, 600  
 lista jednokierunkowa, 187, 188  
 listy obiektów, 638  
 lokalne grupy robocze, 75  
 losowość, 536

## Ł

łączenie  
     geometrii, 234  
     programów, 207  
     przekształceń, 99  
     punktów dowiązania, 140

## M

macierz, 88, 199  
     cienia, 515  
     jednostkowa, 94  
     kamery, 144  
     model-widok, 144  
     obrotu, 97  
     ortograficzna, 106  
     patrzenia, 103, 432  
     perspektywy, 105  
     przekształceń, 94  
     rzutowania, 144  
     skalowania, 99  
     TBN, 499  
     transpozycji, 95  
 magazyn danych, 115, 565  
 małe próbkowane, 360  
 mapa  
     cieni, 517  
     normalnych, 498  
     połysku, 511  
     sześcienna, 505, 509  
     środowiska, 502  
     walcowa, 506  
     walcowa równoodległościowa, 504

mapowanie, 115  
 bufora, 119, 566, 603  
 cienia, 513  
 nierówności, 497  
 normalnych, 497–500, 529  
 przemieszczeń, 288  
 tonalne, 376, 378  
 trwałe, 33, 588, 590  
 maskowanie kolorów, 341  
 matematyka, 81  
 materiały, 510  
 mechanizm  
 HUD, 600  
 kompresji RGTC, 446  
 łączący, 210  
 teselacji, 65  
 metoda .length(), 202  
 metody renderowania, 524  
 mgła, 517  
 mierzenie czasu, 468  
 mieszanie, 336  
 mipmapa, 157, 169, 172  
 mnożenie macierzy, 91  
 modele oświetlenia, 485  
 Phong, 486  
 modelowanie przekształceń, 91  
 monitorowanie potoku graficznego, 459

## N

nagłówek pliku, 633  
 .KTX, 162  
 nagłówki części, 634  
 narzędzie  
 dds2ktx, 629  
 GPU PerfStudio, 600  
 ktxtool, 627  
 OpenGL Extensions Viewer, 78  
 sb6mtool, 629  
 Windows Performance Toolkit, 595  
 niepodzielność, 152  
 normalizacja, 83  
 normalna, 310

## O

obiekt  
 bufora ramki, 74  
 bufora uniform, UBO, 132  
 potoku, 212  
 potoku programu, 609  
 programu, 50  
 próbki, 165  
 shadera, 50  
 sprzężenia zwrotnego przekształcenia, 474  
 synchronizacji, 477  
 obracanie punktów, 396  
 obrazy, 180  
 obrót, 90, 91, 97  
 sześcianu, 144, 145  
 obsługa rozszerzenia, 78  
 obszar renderingu, 68  
 obwódka  
 ochronna, 270  
 świetlna, 495  
 odbicie, 87  
 lustrzane, 486  
 materiału, 493  
 odczyt danych  
 tekstury, 159, 165, 403  
 z bufora ramki, 398  
 odległości przycięcia, 273  
 odległość, 553  
 odwrócenie  
 frontem, 70  
 tyłem, 70  
 odwzorowanie środowiska, 501  
 mapa walcowa, 504  
 sferyczne, 502  
 ogniskowa, 420  
 ogon mipmapy, 442  
 ogrodzenia, 477  
 okluzja, 460  
 otoczenia, 533, 537  
 otoczenia w przestrzeni ekranu, 533  
 określanie  
 koloru, 73  
 dowiązań, 141  
 OpenMP, 34, 565  
 operacja culling, 582, 583

operacje  
 logiczne, 340  
 na pikselach, 74  
 na wektorach, 84  
 niepodzielne, 147, 149, 185  
 niepodzielne na obrazach, 185, 186  
 szablonu, 331  
 opróżnianie potoku, 477  
 optymalizacja  
 buforów pakietów, 572  
 shadera, 610  
 wydajności CPU, 563  
 ostrosłup ścięty, 105  
 oświetlenie, 485  
 Blinna-Phonga, 493, 562

## P

pakiet, 569  
 DMA, 597  
 zoptymalizowany, 575  
 pakiety prezentacji, 597  
 pakowanie bloku RGTC, 451  
 parametr  
 access, 181  
 binding, 153  
 bufSize, 216  
 GL\_NUM\_EXTENSIONS, 77  
 index, 320  
 internalformat, 194  
 minlayer, 194  
 normalized, 124  
 numlayers, 194  
 offset, 153  
 pname, 137  
 primitiveMode, 259  
 program, 218, 220  
 shader, 208  
 source, 619  
 stride, 124  
 texture, 167  
 type, 619  
 uniformBlockIndex, 140  
 unit, 167  
 parametry punktów, 393  
 parametryzacja czworoboku, 317  
 patrzenie w przód, 103  
 perspektywa, 105

pętla renderująca, 144, 146, 427  
 pierwszeństwo  
 kolumny, 89  
 wiersza, 89  
 piksel, 44, 74  
 plik  
 sb7.h, 47  
 vmath.h, 89  
 pliki  
 KTX, 162  
 SBM, 633  
 płat, 64, 275  
 pobieranie  
 danych z mechanizmu łączącego, 210  
 indeksów, 136  
 informacji, 136  
 informacji z kompilatora, 208  
 próbki, 366  
 utworzonego obrazu, 398  
 wierzchołków, 59  
 podprocedury  
 shaderów, 217  
 typu uniform, 217  
 pojemność lokalnej grupy roboczej, 408  
 pokrycie próbki, 367  
 pole odległości, 553–558  
 polecenia  
 OpenGL, 568  
 rysowania, 231, 232  
 połysk, 511  
 pomijanie geometrii, 303  
 pomniejszanie, 168  
 potok, 44  
 graficzny, 39, 45, 59, 75, 459  
 z wymiennymi elementami, 213  
 potokowość, 40  
 powierzchnia Béziera, 294  
 powierzchnie wyższego rzędu, 294  
 powiększanie, 168  
 poziomy mipmap, 172  
 pozycja atrybutu wierzchołka, 60  
 półprzestrzeń, 69  
 prefiks, 414  
 profil  
 główny, 44  
 zgodności, 44  
 progowanie, 385

- program
  - graficzny, 406
  - obliczeniowy, 406
  - OpenGL, 47
  - w postaci binarnej, 220
- programy monolityczne, 212
- prototyp funkcji, 77
- próbkowanie
  - centroidalne, 370
  - cienia, 513
  - tekstury rzadkiej, 445
- prymityw, 44
  - zdegenerowany, 56
- przechowywanie przekształconych wierzchołków, 254
- przeciążanie, 49
  - funkcji, 203
- przecięcie
  - promienia i kuli, 544
  - promień-płaszczyzna, 549
- przekazywanie danych, 123
  - do shadera, 59
  - z jednego etapu do drugiego, 61
- przekładanie atrybutów, 126
- przekształcenia
  - geometryczne, 142
  - model-widok, 102
  - obszaru renderingu, 69
  - rzutowania, 104
  - współrzędnych, 94
- przekształcenie, 91, 94
  - obrót, 97
  - skalowanie, 99
  - transpozycja, 95
- przepustowość pamięci, 190, 525
- przestrzenie współrzędnych, 92
- przestrzeń
  - ekranu, 533
  - kolorów sRGB, 388
  - modelu, 92
  - obiektu, 92
  - okna, 92
  - przycięcia, 50, 69, 92
  - świata, 92
  - widoku, 92
  - znormalizowanych współrzędnych urządzenia, 92
- przesunięcie, 90, 91
  - wieloboku, 516
- przetwarzanie
  - prymitywów, 275
  - wierzchołków, 225
- przezroczystość, 49
- przycięcia definiowane przez użytkownika, 271
- przycinanie, 69, 267
  - linii, 269
  - trójkątów, 270
  - względne obiektu, 272
- punkt
  - przycięcia, 549
  - przycięcia w przestrzeni, 544
- punkty
  - dowiązania, 116
  - o dowolnym kształcie, 394
  - okluzji, 534
  - sterujące, 64, 107, 276, 294

## R

- rasteryzacja, 44, 71
- rdzenie cieniowania, 40
- refrakcja, 87
- reguła stada, 429
- renderowanie, 287, 464, 485, 510, 524, 611
  - asteroidy, 254
  - bez dołączenia, 373
  - bez trójkątów, 539
  - czcionek, 555
  - czworoboków, 315, 316
  - do map sześciennych, 353
  - fraktala, 593
  - fraktali Julii, 539
  - gwiazdnego pola, 391
  - kopii, 244
  - kreskówkowe, 522
  - niefotorealistyczne, 520
  - pośrednie, 576
  - pośrednie elastyczne, 577
  - pozaekranowe, 342
  - punktu, 54
  - terenu, 288, 289, 290, 291
  - trójkąta, 56
  - warstw, 348, 359
  - warunkowe, 463, 465, 466
  - wyniku, 189

reset grafiki, 623  
 restart prymitywu, 234  
 rodzaje  
   geometrii, 300  
   prymitywów, 313  
   przekształceń, 94  
   zmiennych próbki, 160  
 rozbłysk, 386  
   odbicia lustrzanego, 486  
   światła, 381  
 rozdzielanie atrybutów, 126  
 rozmiar punktu, 230  
 rozmycie, 384  
 rozszerzenia  
   GLSL, 78  
   OpenGL, 76, 654–662  
 równanie  
   kwadratowe, 108  
   mieszające, 339  
   sześciennie, 109  
 rysowanie, 231  
   asteroid, 253  
   pośrodkie, 246  
   sceny, 357  
   trójkąta, 55  
   wielokrotne geometrii, 236  
 rzucanie cieni, 512  
 rzutowanie, 91, 104  
   ortograficzne, 104, 106  
   perspektywiczne, 104, 105

## S

schemat potoku graficznego, 41  
 scyntylacja, 169  
 selektor NEAREST, 171  
 shader, 40, 49  
   fragmentów, 51, 62, 164, 178, 323  
   geometrii, 66, 297, 303, 305, 313  
   geometrii przekazujący dane dalej, 298  
   obliczeniowy, 75, 405, 425, 433  
   sterowania teselacją, 64, 290  
   wierzchołków, 50, 59, 125, 164, 243, 264  
   wylizania teselacji, 65, 291  
   wysyłający dane, 61  
 skalar, 88  
 skalowanie, 99  
 składanie prymitywów, 68  
 składowe klasy, 195  
 słowo kluczowe  
   in, 61  
   out, 61  
   uniform, 147  
 specjalizacja bufora pakietów, 575  
 spirala, 283  
 splajn, 110  
 spoina, 110  
 sprawdzanie  
   bufora ramki, 354  
   programów, 207  
   zakresu, 625  
 sprężyna, 261, 266  
 sprite'y punktów, 390  
 sprzężenie zwrotne, 260  
   przekształcenia, 254, 259  
 stała GL\_POINTS, 54  
 stan potoku, 475  
 sterowanie  
   aktualizacjami bufora głębi, 333  
   kolejnością rysowania, 285  
   potokiem graficznym, 459  
   przepływem, 467  
 stosowanie  
   barier w aplikacji, 151  
   barier w shaderach, 152  
   rozszerzeń, 77  
   shaderów geometrii, 300  
 strona, 440  
 struktura, 201  
   danych pakietu, 569  
   przyspieszenia, 551  
 strumienie magazynowe, 312  
 suma prefiksowa, 414, 417  
   ekskluzyjna, 415  
   inkluzyjna, 415  
 symulacja fizyczna, 260  
 symulator cząsteczek OpenMP, 567  
 synchronizacja  
   dostępu, 591  
   dostępu do buforów, 588  
   dostępu do liczników, 155  
   dostępu do pamięci, 150, 189  
   ogrodzeniowa, 477  
   shaderów obliczeniowych, 411  
   w OpenGL, 477  
 system sprężyna-masa, 264, 266

sześcian, 143, 147  
 szybkość, 623

## Ś

ściskanie głębi, 333  
 śledzenie promieni, 541, 547, 551  
 światło  
 otoczenia, 486  
 rozproszone, 486

## T

tablica, 201  
 sumowanego obszaru, 419  
 wierzchołków, 53  
 technika śledzenia promienia, 551  
 techniki renderowania, 485  
 teksele, 521  
 tekstura  
 bez dowiązań, 458  
 tekstura samplerBuffer, 265  
 tekstury, 156  
 atlasowe, 440  
 bez dowiązań, 33, 436  
 mapy sześcienniej, 507  
 pola odległości, 552  
 punktów, 390  
 rezydentne, 438  
 rzadkie, 440, 441, 458  
 tablicowe, 175, 176  
 wielopróbkowe, 364  
 teren zrenderowany, 293  
 teselacja, 64, 275, 294  
 czworoboku, 278  
 trójkąta, 280  
 za pomocą izolinii, 281  
 test  
 głębi, 75, 326, 332  
 nożycowy, 74, 326  
 szablonu, 74, 326, 328  
 tokeny typów, 122  
 torus, 523  
 transformacja obszaru renderingu, 69, 319  
 transpozycja, 95  
 obrazu, 384  
 trawa, 239–241  
 trójkąt, 55, 57, 279

tryb  
 filtracji, 165  
 filtracji tekstur, 172  
 podziału teselacji, 284  
 prymitywów shadera geometrii, 300  
 prymitywów teselacji, 276  
 punktowy teselacji, 283  
 zawijania, 165

tworzenie  
 aplikacji, 47  
 bloków uniform, 133  
 bufora, 118  
 buforów, 116  
 egzemplarzy, 236  
 macierzy, 89  
 tekstur, 156  
 tekstury rzadkiej, 440  
 widoku tekstury, 193–195  
 wierzchołków, 56  
 zmiennych, 53

typ  
 float, 137  
 skalarny, 198  
 uniform, 129

typy  
 atrybutów wierzchołka, 228  
 części, 634  
 danych, 198  
 obrazów, 180  
 próbki, 160  
 tekstur, 158, 160

## U

UBO, Uniform Buffer Object, 132  
 uchwyt, 436  
 układ  
 BGRA, 228  
 standardowy, 133, 139  
 współdzielony, 133  
 unia, 570  
 uniform, 129  
 upakowane formaty danych, 454, 455  
 ustawianie  
 danych dla tablicy, 138  
 danych macierzy, 138  
 pojedynczej wartości, 137  
 usuwanie trójkątów, 70

użycie  
 formatów zmiennoprzecinkowych, 375  
 maski, 342

## V

VAO, 52

## W

warstwa, 175  
 abstrakcji, 40  
 wartości  
 parametru primitiveMode, 259  
 uzmiennione, 255  
 wątek, 564, 612  
 wczesne testowanie, 335  
 wczytywanie  
 obiektów, 128  
 pliku .KTX, 163  
 tekstur, 162  
 z pliku, 128, 162  
 wejścia shadera  
 obliczeniowego, 408  
 shadera wierzchołków, 126  
 wektor, 82, 199  
 boczny, 103  
 góry, 103  
 jednorodny, 84  
 jednostkowy, 83  
 ortogonalny, 90  
 ortonormalny, 90  
 znormalizowany, 83  
 wersja binarna programu, 220  
 wersje OpenGL, 42, 641  
 węzeł, 110  
 widoki  
 obrazu HDR, 377  
 tekstur, 193  
 wielokrotne  
 próbkowanie, 75, 363  
 wykorzystanie wierzchołka, 476  
 wielowątkowość, 563  
 wierzchołek, 44  
 bazowy, 234  
 właściwości materiału, 510, 579  
 wskaźnik na funkcję, 79

współczynnik ekspozycji, 378  
 współrzędne  
 barycentryczne, 276  
 jednorodne, 69  
 obiektu, 92  
 okna, 70  
 świata, 93  
 tekstury, 163  
 widoku, 93  
 wydajność CPU, 563  
 kompilacji shadera, 608  
 wygładzanie linii, 362  
 wyjścia shadera obliczeniowego, 408  
 wyjście koloru, 336  
 wykładniki współdzielone, 193  
 wykonywanie shaderów obliczeniowych, 407  
 wykorzystanie wątków, 564  
 wykrywanie krawędzi, 372  
 wyliczanie teselacji, 278, 282, 295  
 wyłączenie interpolacji, 324  
 wynik zapytania, 462  
 wypełnienie listy jednokierunkowej, 187  
 wysoki zakres dynamiczny, 376  
 wyścig, 589  
 wyświetlanie normalnych, 312

## Z

zadania  
 równoległe, 40  
 w locie, 39  
 zapis tekstur w shaderach, 180  
 zapytania  
 czasomierza, 468  
 indeksowane, 473  
 o liczbę prymitywów, 473  
 o okluzję, 460, 463, 466  
 sprzężenia zwrotnego przekształceń, 471  
 stanu potoku, 475  
 zarządzanie  
 danymi, 435  
 zatwierdzeniem tekstury, 443  
 zastosowanie  
 bloku magazynu, 148  
 efektu mgły, 519, 521  
 zatwierdzanie tekstury, 443  
 zawijanie tekstur, 173

zbiór

Julii, 539

Mandelbrota, 540

zero kopiowania, 588

zgodność celów, 194

zmiana typu prymitywu, 309

zmienna, 53

zmiennie uniform, 129

aranżacja, 130

przekształcenia geometryczne, 142

ustawianie, 130

zmiennie wbudowane, 56

zmiennoprzecinkowe bufony ramki, 374

znaczniki

magazynu, 117

mapowania buforów, 120

znak obszaru, 70

znormalizowana przestrzeń urządzenia, 69

zrównoleglenie, 40

zrównoległona suma prefiksowa, 414

zrzut ekranu, 399

zwiększanie

wydajności, 602

liczby kierunków, 535



# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄŻKI**
- 3. ZBIERAJ PROWIZJĘ**

Zmień swoją stronę WWW  
w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

**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 doskonale 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 tekstuowania 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!**



43383	numer katalogowy
	księgarnia internetowa
	<a href="http://helion.pl">http://helion.pl</a>
	zamówienia telefoniczne
	<b>0 801 339900</b>
	<b>0 601 339900</b>

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](mailto:helion@helion.pl)  
<http://helion.pl>

sięgnij po WIĘCEJ



KOD KORZYŚCI

ISBN 978-83-283-2107-6



Informatyka w najlepszym wydaniu

cena: 119,00 zł