Grafika interaktywna

Krzysztof Gdawiec

UNIWERSYTET ŚLĄSKI
INSTYTUT INFORMATYKI

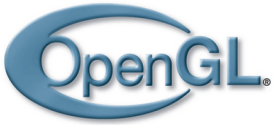
OpenGL (Open Graphics Library) jest wieloplatformową biblioteką pozwalającą na niskopoziomowy dostęp do karty graficznej i maksymalne wykorzystanie jej potencjału.

Powstała w roku 1993 na bazie języka Iris GL opracowanego przez firmę Silicon Graphics Inc.

Za rozwój biblioteki odpowiedzialna była organizacja ARB (Architectural Review Board), w której skład początkowo wchodziły m.in. 3DLabs, Apple, ATI, NVIDIA, Intel, IBM, HP, Sun Microsystems. W 2006 roku ARB zostało wcielone do grupy Khronos, która zajmuje się do dziś rozwojem biblioteki OpenGL.

Odmiany biblioteki OpenGL:

- OpenGL – używana na komputerach



- OpenGL ES (Embedded Systems) – używana w systemach wbudowanych, np. w konsolach, telefonach itp.

- WebGL – używana w przeglądarkach internetowych, bazuje na OpenGL ES

- OpenGL SC (Safety Critical) – używana tam gdzie potrzebna jest duża niezawodność, np. awionika, medycyna, przemysł itp.

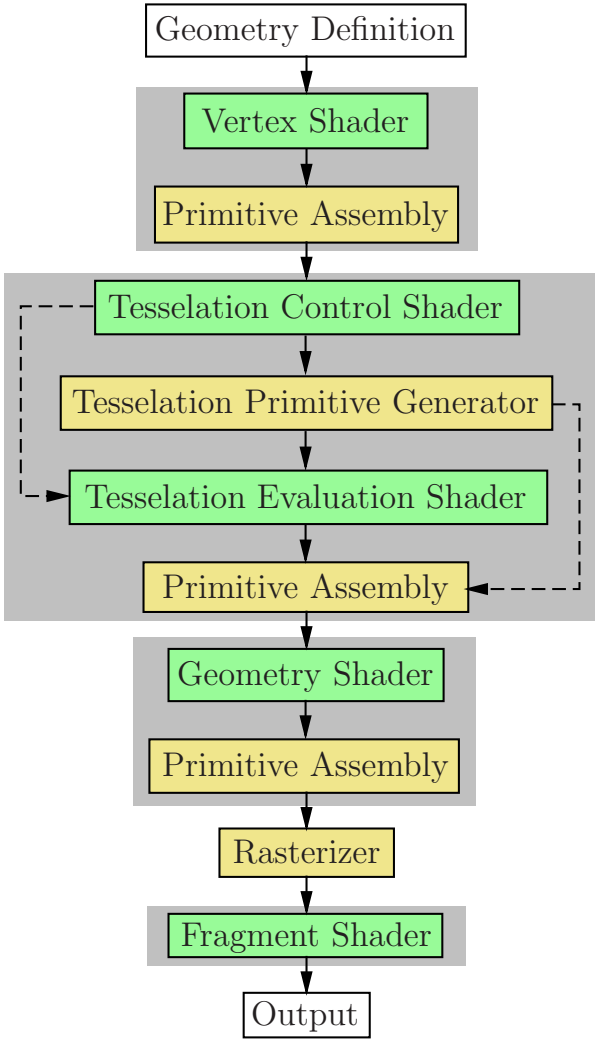
OpenGL	GLSL	Kiedy
1.0	–	1993
1.1	–	1997
1.2	–	1998
1.3	–	2001
1.4	–	2002
1.5	–	2003
2.0	1.10	2004
2.1	1.20	2006

OpenGL	GLSL	Kiedy
3.0	1.30	08.2008
3.1	1.40	03.2009
3.2	1.50	08.2009
3.3	3.30	03.2010
4.0	4.00	03.2010
4.1	4.10	07.2010
4.2	4.20	08.2011
4.3	4.30	08.2012
4.4	4.40	07.2013
4.5	4.50	07.2014
4.6	4.60	07.2017

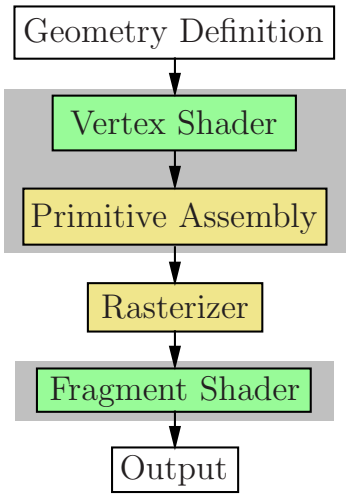
- WebGL 1.0: 02.2011
  oparty na OpenGL ES 2.0 i GLSL ES 1.0, które są oparte na OpenGL 2.0 i GLSL 1.20
- WebGL 2.0: 01.2017
  oparty na OpenGL ES 3.0, który oparty jest na OpenGL 3.3 z elementami OpenGL 4.2

Programowalny potok graficzny

OpenGL



WebGL



Dostęp do WebGL w HTML5 I

Na stronie umieszczamy element `canvas`, w którym będziemy używali WebGL.

```
1  <!DOCTYPE html>
2  <html lang="pl">
3  <head>
4      <meta charset="utf-8" />
5
6      <title>WebGL</title>
7  </head>
8
9  <body>
10     <canvas id="gl-canvas" width="512" height="512">
11         Oops ... Twoja przegladarka nie wspiera elementu
               canvas z HTML5.
12     </canvas>
13 </body>
14 </html>
```

W JavaScript pobieramy kontekst WebGL za pomocą funkcji `getContext` elementu `canvas`.

```
1  var canvas = document.getElementById( "gl-canvas" );
2  var gl = canvas.getContext( "webgl" );
```

Argumentem funkcji `getContex` jest nazwa kontekstu, który chcemy pobrać. Dla WebGL jest to `webgl`, ale w starszych przeglądarkach nazwa mogła być różna: `experimental-webgl`, `webkit-3d` (przeglądarki oparte na Webkit), `moz-webgl` (Firefox). Zatem najbezpieczniej jest pobierać kontekst dla każdej z nazw aż do momentu powodzenia pobierania kontekstu.

```
1  var names = ["webgl", "experimental-webgl", "webkit-3d", "
       moz-webgl"];
2  var gl = null;
3  for( var i = 0; i < names.length; ++i )
4  {
5    try
6    {
7      gl = canvas.getContext( names[i] );
8    }
9    catch( e ){}
10
11   if( gl )
12     break;
13 }
```

Bufor ramki (ang. frame buffer)

Bufor ramki w WebGL składa się z:

- bufora koloru (ang. color buffer),
- bufora głębi (ang. depth buffer, z-buffer),
- bufora szablonowego (ang. stencil buffer).

Bufor koloru składa się z kilku buforów: przedniego lewego, przedniego prawego, tylnego lewego, tylnego prawego. Zazwyczaj wyświetlana na monitorze jest zawartość przednich buforów, a zawartość tylnych buforów jest niewidoczna. Dla monitora monoskopowego wyświetlany jest jedynie lewy przedni bufor.

Konwencja nazewnictwa funkcji

```
rtype Name{1|2|3|4}{i|f}{v}(args)
```

- `rtype` – typ zwracany przez funkcję,
- `Name` – nazwa funkcji,
- `1, 2, 3, 4` – liczba argumentów funkcji,
- `i|f` – argumenty typu: `int`, `float`,
- `v` – argument funkcji stanowi tablica wartości (w tym wypadku nie występuje określenie liczby argumentów),
- `args` – argumenty funkcji.

Maszyna stanów

Maszyna stanów WebGL to zbiór wszystkich zmiennych wewnętrznych (zmiennych stanu) i ustawień biblioteki.

Ważną cechą maszyny stanów WebGL jest zachowywanie zmiennych stanu do czasu, aż zostaną one zmienione przez jakąś funkcję.

Maszyna stanów

Maszyna stanów WebGL to zbiór wszystkich zmiennych wewnętrznych (zmiennych stanu) i ustawień biblioteki.

Ważną cechą maszyny stanów WebGL jest zachowywanie zmiennych stanu do czasu, aż zostaną one zmienione przez jakąś funkcję.

Wiele zmiennych stanu jest po prostu włączana i wyłączana. Do włączania stanu `capability` służy funkcja

```
gl.enable( capability );
```

a do wyłączania służy funkcja

```
gl.disable( capability );
```

Zmienna `gl` jest to kontekst WebGL.

Jeśli chcemy sprawdzić czy jakaś zmienna stanu jest włączona, to korzystamy z funkcji (zwracana jest wartość boolowska)

```
gl.isEnabled( capability );
```

Jeśli chcemy sprawdzić czy jakaś zmienna stanu jest włączona, to korzystamy z funkcji (zwracana jest wartość boolowska)

```
gl.isEnabled( capability );
```

Wiele innych funkcji WebGL może przyjmować wartości zapamiętywane do czasu ich zmiany. W takim przypadku do odczytania aktualnej wartości zmiennej stanu służą funkcje z rodziny `get`. Dla różnych własności istnieją odrębne wersje funkcji `get`, np. `getShaderParameter` służy do pobrania własności shadera.

Obsługa błędów

W WebGL występują zmienne stanu oznaczające wystąpienie błędu. Informacje o kodzie bieżącego błędu zwraca funkcja:

```
gl.getError();
```

Znaczenie poszczególnych kodów jest następujące:

- `gl.NO_ERROR` – brak błędu,
- `gl.INVALID_ENUM` – argument typu wyliczeniowego poza dopuszczalnym zakresem,
- `gl.INVALID_VALUE` – argument liczbowy poza dopuszczalnym zakresem,
- `gl.INVALID_OPERATION` – operacja niewykonalna w obecnym stanie,

- `gl.INVALID_FRAMEBUFFER_OPERATION` – obiekt bufora ramki jest niekompletny,
- `gl.OUT_OF_MEMORY` – brakuje pamięci do wykonania operacji.

- `gl.INVALID_FRAMEBUFFER_OPERATION` – obiekt bufora ramki jest niekompletny,
- `gl.OUT_OF_MEMORY` – brakuje pamięci do wykonania operacji.

Wystąpienie błędu nie powoduje przerwania wykonywania programu – nie jest wykonywana jedynie funkcja odpowiedzialna za jego powstanie. Wyjątek stanowi błąd braku pamięci.

- `gl.INVALID_FRAMEBUFFER_OPERATION` – obiekt bufora ramki jest niekompletny,
- `gl.OUT_OF_MEMORY` – brakuje pamięci do wykonania operacji.

Wystąpienie błędu nie powoduje przerwania wykonywania programu – nie jest wykonywana jedynie funkcja odpowiedzialna za jego powstanie. Wyjątek stanowi błąd braku pamięci.

Funkcja `gl.getError` zwraca tylko jedną z wartości podanych powyżej. Jeżeli ustawionych jest więcej znaczników niż jeden, to funkcja `gl.getError` i tak będzie zwracać tylko jedną wartość, która po wywołaniu funkcji jest kasowana. Po ponownym wywołaniu funkcja `gl.getError` zwróci albo nowy kod błędu albo znacznik `gl.NO_ERROR`.

Wprowadzenie do GLSL ES

GLSL ES (ang. OpenGL ES Shading Language) jest to język programów cieniowania lub w skrócie shaderów.

Język ten umożliwia tworzenie dwóch rodzajów programów cieniowania:

- shader wierzchołków (ang. vertex shader),
- shader fragmentów (ang. fragment shader).

Za przetwarzanie wierzchołków i wykonanie programów cieniowania wierzchołków odpowiedzialny jest procesor wierzchołków (ang. vertex processor). Wykonuje on m.in. następujące elementy potoku WebGL:

- transformacja wierzchołków,
- transformacja i normalizacja normali,
- generowanie i transformacja współrzędnych tekstur,
- obliczanie oświetlenia poszczególnych wierzchołków,
- obliczanie koloru.

Przetwarzaniem fragmentów za pomocą shaderów fragmentów zajmuje się procesor fragmentów (ang. fragment processor). Wykonuje on m.in. następujące operacje:

- obliczenia koloru i współrzędnych tekstury piksela,
- odczyt danych tekstury,
- obliczenia mgły,
- sumowanie kolorów.

Przetwarzaniem fragmentów za pomocą shaderów fragmentów zajmuje się procesor fragmentów (ang. fragment processor). Wykonuje on m.in. następujące operacje:

- obliczenia koloru i współrzędnych tekstury piksela,
- odczyt danych tekstury,
- obliczenia mgły,
- sumowanie kolorów.

W 2006 r. NVIDIA, a w 2007 r. AMD (ATI) wprowadziły zunifikowane jednostki przetwarzania zwane procesorami strumieniowymi, które w zależności od potrzeb wykonują programy cieniowania wierzchołków lub fragmentów.

Składnia GLSL ES oparta jest na językach C i C++.

Składnia GLSL ES oparta jest na językach C i C++.

Instrukcje preprocesora: `#`, `#define`, `#undef`, `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`, `#error`, `#pragma`, `#extension`, `#version`, `#line`.

Składnia GLSL ES oparta jest na językach C i C++.

Instrukcje preprocesora: `#`, `#define`, `#undef`, `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`, `#error`, `#pragma`, `#extension`, `#version`, `#line`.

Instrukcja `#error` służy do generowania komunikatów diagnostycznych, które umieszczane są w logu programu cieniowania.

Składnia GLSL ES oparta jest na językach C i C++.

Instrukcje preprocesora: `#`, `#define`, `#undef`, `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`, `#error`, `#pragma`, `#extension`, `#version`, `#line`.

Instrukcja `#error` służy do generowania komunikatów diagnostycznych, które umieszczane są w logu programu cieniowania.

Instrukcja `#version` określa wersję GLSL ES, w której napisany jest dany shader. Jej brak oznacza, że jest on napisany w wersji 100 języka. Instrukcja ta musi wystąpić na samym początku programu cieniowania.

Instrukcja `#pragma` umożliwia kontrolę nad ustawieniami kompilatora zależnymi od implementacji. Mamy kontrolę nad optymalizacją:

```
#pragma optimize(on)
#pragma optimize(off)
```

Domyślnie optymalizacje są włączone dla każdego shadera. Mamy również kontrolę nad zbieraniem, podczas kompilacji programu cieniowania, informacji przydatnych przy debugowaniu:

```
#pragma debug(on)
#pragma debug(off)
```

Domyślnie zbieranie tych informacji jest wyłączone.

Instrukcja `#extension` kontroluje współpracę programów cieniowania z rozszerzeniami języka GLSL ES.

```
#extension extension_name : behavior
#extension all : behavior
```

`extension_name` jest to nazwa rozszerzenia, którego chcemy zmienić zachowanie określone za pomocą parametru `behavior`. Wartości `behavior` są następujące:

- `require` – rozszerzenie o nazwie `extension_name` jest wymagane przez program,
- `enable` – włączenie rozszerzenie o nazwie `extension_name`,
- `warn` – kompilator generuje ostrzeżenie o użyciu rozszerzenia,
- `disable` – wyłączenie rozszerzenia `extension_name` lub wszystkich rozszerzeń `all`.

Początkowo wszystkie rozszerzenia są wyłączone.

Podstawowe typy danych

`void`, `bool`, `int`, `float`

Wektorowe: `vec2`, `vec3`, `vec4`, `bvec2`, `bvec3`, `bvec4`, `ivec2`, `ivec3`, `ivec4`

Macierzowe: `mat2`, `mat3`, `mat4`

Typ `int` oraz jego odpowiedniki wektorowe mogą być poddawane automatycznej konwersji do typu `float` oraz jego odpowiednika wektorowego.

Dostęp do składowych wektora można uzyskać poprzez odwołanie się jak do elementu w tablicy w C. Drugim sposobem jest użycie kropki i nazwy pola: `{x, y, z, w}`, `{r, g, b, a}`, `{s, t, p, q}`. Nazwy można łączyć (wyłącznie w zakresie danej grupy), np.

```
1  vec4 v;
2  v.xyzw;
3  v.rbg;
4  v.st;
```

Do inicjowania współrzędnych wektorów służy szereg różnych konstruktorów.

Dostęp do składowych wektora można uzyskać poprzez odwołanie się jak do elementu w tablicy w C. Drugim sposobem jest użycie kropki i nazwy pola: `{x, y, z, w}`, `{r, g, b, a}`, `{s, t, p, q}`. Nazwy można łączyć (wyłącznie w zakresie danej grupy), np.

```
1  vec4 v;
2  v.xyzw;
3  v.rbg;
4  v.st;
```

Do inicjowania współrzędnych wektorów służy szereg różnych konstruktorów.

W przypadku macierzy dostęp do składowych można uzyskać jedynie tak jak w C. Podobnie jak dla wektorów istnieje szereg konstruktorów inicjujących zawartość macierzy.

Operatory arytmetyczne, bitowe, porównania, przypisania i logiczne są takie same jak w C.

W przypadku wektorów i macierzy operatory działają na wszystkich składowych, wyjątkiem jest operator mnożenia wektora przez macierz, macierzy przez wektor lub macierzy przez macierz wówczas operator ten zachowuje się tak jak to jest zdefiniowane w algebrze liniowej.

Kwalifikatory typów

`const`

Kwalifikatory typów

`const`

`attribute` – kwalifikator używany dla zmiennych, które są przekazywane do shadera wierzchołków z WebGL, tzw. atrybuty wierzchołka, np. współrzędne wierzchołka, normal.

Kwalifikatory typów

`const`

`attribute` – kwalifikator używany dla zmiennych, które są przekazywane do shadera wierzchołków z WebGL, tzw. atrybuty wierzchołka, np. współrzędne wierzchołka, normal.

`uniform` – kwalifikator używany dla zmiennych, których wartość nie zmienia się dla całego modelu; są one tylko do odczytu; są przekazywane z WebGL.

Kwalifikatory typów

`const`

`attribute` – kwalifikator używany dla zmiennych, które są przekazywane do shadera wierzchołków z WebGL, tzw. atrybuty wierzchołka, np. współrzędne wierzchołka, normal.

`uniform` – kwalifikator używany dla zmiennych, których wartość nie zmienia się dla całego modelu; są one tylko do odczytu; są przekazywane z WebGL.

`varying` – kwalifikator używany dla zmiennych, które używane są do przekazywania danych z shadera wierzchołków do shadera fragmentów; zmienne w obu shaderach muszą mieć tą samą nazwę; zmienna w shaderze wierzchołków jest do zapisu, a w shaderze fragmentów do odczytu.

`in` – określa parametr wejściowy funkcji, tzn. wszelkie zmiany tego parametru wewnątrz funkcji nie mają wpływu na jego wartość poza funkcją; parametr musi mieć wartość.

`in` – określa parametr wejściowy funkcji, tzn. wszelkie zmiany tego parametru wewnątrz funkcji nie mają wpływu na jego wartość poza funkcją; parametr musi mieć wartość.

`out` – określa parametry wyjściowy funkcji, tzn. zmiany tego parametru wewnątrz funkcji są widoczne poza funkcją; parametry z tym kwalifikatorem nie wymagają przekazywania do funkcji żadnej konkretnej wartości.

`in` – określa parametr wejściowy funkcji, tzn. wszelkie zmiany tego parametru wewnątrz funkcji nie mają wpływu na jego wartość poza funkcją; parametr musi mieć wartość.

`out` – określa parametry wyjściowy funkcji, tzn. zmiany tego parametru wewnątrz funkcji są widoczne poza funkcją; parametry z tym kwalifikatorem nie wymagają przekazywania do funkcji żadnej konkretnej wartości.

`inout` – kwalifikator ten określa, że parametr funkcji jest parametrem zarówno wejściowym jak i wyjściowym; wszelkie zmiany wartości tego parametru będą miały wpływ na jego wartość poza funkcją oraz parametr musi mieć wartość.

Kwalifikatory precyzji

Dowolna zmienna `int`, `float` musi mieć deklarację precyzji. Do tego celu służą kwalifikatory precyzji: `highp`, `mediump`, `lowp`.
Przykład:

```
1  lowp float color;
2  varying mediump vec2 Coord;
3  lowp ivec2 foo(lowp mat3);
4  highp mat4 m;
```

Kwalifikatory precyzji

Dowolna zmienna `int`, `float` musi mieć deklarację precyzji. Do tego celu służą kwalifikatory precyzji: `highp`, `mediump`, `lowp`.
Przykład:

```
1  lowp float color;
2  varying mediump vec2 Coord;
3  lowp ivec2 foo(lowp mat3);
4  highp mat4 m;
```

Do ustawienia domyślnego kwalifikatora precyzji używamy polecenia:

```
precision qualifier type;
```

gdzie `qualifier` – kwalifikator precyzji, `type` – typ (`int`, `float`).

Shader wierzchołków ma ustawiony domyślny kwalifikator precyzji dla obu typów danych.

W przypadku shadera fragmentów ustawiony jest jedynie domyślny kwalifikator precyzji dla typu `int`. Dla typu `float` musimy zrobić to sami, np.

```
precision mediump float;
```

Shader wierzchołków ma ustawiony domyślny kwalifikator precyzji dla obu typów danych.

W przypadku shadera fragmentów ustawiony jest jedynie domyślny kwalifikator precyzji dla typu `int`. Dla typu `float` musimy zrobić to sami, np.

```
precision mediump float;
```

Sterowanie przepływem programu

W GLSL ES instrukcje warunkowe (`if`, `if else`) jak i pętle (`for`, `while`, `do while`) mają taką samą konstrukcję jak w C.

Definiowanie własnych funkcji

- Zasady tworzenia funkcji są takie same jak w C.
- Jako argumentu funkcji można użyć tablicy, jednak tablica nie może być typem zwracanym przez funkcję.
- Wszystkie funkcje przed pierwszym użyciem muszą być zadeklarowane lub zdefiniowane.
- Funkcje można przeciążać.
- Funkcje nie mogą być rekurencyjne.

- każdy shader musi mieć funkcję `main`

```
1  void main()
2  {
3    ...
4  }
```

  W przypadku shadera wierzchołków w funkcji `main` musi znaleźć się przypisanie wartości do zmiennej `gl_Position` typu `vec4`. Jest to pozycja wierzchołka w układzie przycinania.
  W przypadku shadera fragmentów musimy przypisać wartość zmiennej `gl_FragColor` typu `vec4`. Jest to kolor jaki ma przyjąć fragment.

- w shaderze fragmentów oprócz instrukcji `return` mamy instrukcję `discard`, a jej wywołanie spowoduje odrzucenie aktualnie przetwarzanego fragmentu i jednocześnie brak aktualizacji zawartości bufora ramki.

Wbudowane funkcje

GLSL ES zawiera szereg różnych funkcji działających na zmiennych skalarnych i wektorowych. Pełną listę można znaleźć w dokumentacji.

W shaderach należy preferować funkcje wbudowane nad odpowiadające im własne implementacje ponieważ z założenia są one optymalne i zawsze, gdy jest taka możliwość są wykonywane sprzętowo.

Używanie shaderów w WebGL

- Pojedyncze programy cieniowania umieszczane są i kompilowane w tzw. obiektach programu cieniowania (ang. shader object).
- Jeden lub więcej takich obiektów jest umieszczanych w tzw. obiekcie programu (ang. program object), który po procesie konsolidacji wykonuje kod shadera.
- Jeden obiekt programu może przechowywać równocześnie różne typy shaderów.

Do utworzenia obiektu programu cieniowania służy funkcja:

```
gl.createShader( type );
```

Parametr `type` określa rodzaj programu cieniowania i przyjmuje jedną z wartości:

- `gl.VERTEX_SHADER`,
- `gl.FRAGMENT_SHADER`.

Funkcja zwraca unikatowy obiekt obiektu programu cieniowania (shadera), który jest wykorzystywany w dalszych operacjach. W przypadku wystąpienia błędu funkcja zwraca wartość `null`.

Załadowanie kodu programu cieniowania dla wybranego obiektu realizuje funkcja:

```
gl.shaderSource( shader, source );
```

Parametr `shader` określa obiekt shadera. Parametr `source` zawiera łańcuchy znaków (`DOMString`), z których składa się źródło shadera.

Wywołanie funkcji `gl.shaderSource` z tym samym obiektem shadera zastępuje poprzedni kod programu cieniowania.

Kompilacja kodu programu cieniowania wykonywana jest przez funkcję:

```
gl.compileShader( shader );
```

Parametr `shader` to obiekt shadera.

Wynik kompilacji shadera jest zawarty w zmiennych stanu obiektu programu cieniowania, które można odczytać za pomocą funkcji:

```
gl.getShaderParameter( shader, gl.COMPILE_STATUS )
```

Funkcja zwraca wartość boolowską (`true` – kompilacja powiodła się, `false` – kompilacja nie powiodła się).

Przy kompilacji shadera generowany jest log zawierający m.in. komunikaty diagnostyczne. Pobranie treści logu umożliwia funkcja `gl.getShaderInfoLog`.

Obiekt programu możemy utworzyć za pomocą funkcji:

```
gl.createProgram();
```

Funkcja zwraca obiekt programu. W przypadku wystąpienia błędu funkcja zwraca wartość `null`.

Obiekt programu możemy utworzyć za pomocą funkcji:

```
gl.createProgram();
```

Funkcja zwraca obiekt programu. W przypadku wystąpienia błędu funkcja zwraca wartość `null`.

Następnie dołączamy program cieniowania do obiektu programu:

```
gl.attachShader( program, shader );
```

Parametry `program` i `shader` są to obiekty (odpowiednio) programu i shadera. Jeśli wybrany shader jest już dołączony do obiektu programu generowany jest błąd `gl.INVALID_OPERATION`.

Po dołączeniu programu cieniowania do obiektu programu musimy dokonać konsolidacji:

```
gl.linkProgram( program );
```

Parametr `program` jest to obiekt programu.

Wynik procesu konsolidacji programów jest przechowywany w zmiennej stanu obiektu programów. Wartość zmiennej stanu można odczytać korzystając z funkcji:

```
gl.getProgramParameter( program, gl.LINK_STATUS )
```

Funkcja zwraca wartość boolowską (`true` – konsolidacja powiodła się, `false` – konsolidacja nie powiodła się).

Przy operacjach na obiekcie programów generowany jest log zawierający m.in. komunikaty diagnostyczne. Pobranie treści logu umożliwia funkcja `gl.getProgramInfoLog`.

Użycie programu cieniowania jest dokonywane za pomocą funkcji:

```
gl.useProgram( program );
```

Parametr `program` jest obiektem programu. Podanie wartości `null` powoduje wyłączenie programowalnej części potoku WebGL, a zachowanie funkcji rysujących nie jest zdefiniowane.

Użycie programu cieniowania jest dokonywane za pomocą funkcji:

```
gl.useProgram( program );
```

Parametr `program` jest obiektem programu. Podanie wartości `null` powoduje wyłączenie programowalnej części potoku WebGL, a zachowanie funkcji rysujących nie jest zdefiniowane.

Usuwanie obiektu programu cieniowania wykonujemy za pomocą funkcji:

```
gl.deleteShader( shader );
```

Parametr `shader` jest to obiekt usuwanego shadera. Jeżeli usuwany obiekt shadera nie należy do żadnego obiektu programu operacja usuwania jest wykonywana natychmiastowo. W przeciwnym wypadku obiekt otrzymuje status usuniętego.

Odłączenie programu cieniowania od obiektu programów wykonuje funkcja:

```
gl.detachShader( program, shader );
```

Parametry `program` i `shader` są to obiekty (odpowiednio) programu i shadera. Jeżeli odłączany shader ma status usuniętego i nie jest dołączony do innego obiektu programu jest on usuwany bezpośrednio po odłączeniu.

Usuwania obiektu programów dokonujemy za pomocą funkcji:

```
gl.deleteProgram( program );
```

Parametr `program` jest to obiekt programu.

Jeżeli usuwany obiekt programu jest częścią aktualnego potoku renderowania obiekt otrzymuje jedynie status usuniętego. Fizyczne usunięcie obiektu programu następuje po zaprzestaniu jego wykonywania w każdym kontekście renderingu.

Jeżeli usuwany obiekt programu ma dołączone shadery są one odłączane i otrzymują status usuniętego poprzez wewnętrzne wywołanie funkcji `gl.deleteShader`.

Zmienne atrybutowe

Pobranie położenia atrybutu w shaderze wierzchołków:

```
gl.getAttribLocation( program, name );
```

Parametr `program` jest to obiekt programu, a `name` jest to nazwa zmiennej atrybutowej (`DOMString`), której położenie chcemy poznać. Funkcja zwraca lokalizację atrybutu (`int`).

Pobranie położenia zmiennej atrybutowej jest możliwe dopiero po pomyślnym zakończeniu konsolidacji programów cieniowania.

Sposób przekazywania wartości do zmiennych atrybutowych zobaczymy przy omawianiu VBO.

Zmienne jednorodne

Pobieranie położenia aktywnej zmiennej jednorodnej:

```
gl.getUniformLocation( program, name );
```

Parametr `program` jest to identyfikator obiektu programu, a `name` jest to nazwa zmiennej jednorodnej (`DOMString`), której położenie chcemy poznać. Funkcja zwraca lokalizację zmiennej jednorodnej (`WebGLUniformLocation`).
Funkcja zwraca `null` jeśli obiekt programów nie zawiera aktywnej zmiennej jednorodnej o podanej nazwie lub nazwa rozpoczyna się zarezerwowanym przedrostkiem `gl_`.
Pobranie położenia aktywnej zmiennej jednorodnej jest możliwe dopiero po pomyślnym zakończeniu konsolidacji programów cieniowania.

Zapis wartości zmiennych jednorodnych jest możliwy jedynie z poziomu nieprogramowalnej części potoku WebGL i służą do tego funkcje `gl.uniform`, np.

```
gl.uniform2f( location, v0, v1 );
gl.uniform2fv( location, values );
```

Parametr `location` jest to położenie zmiennej jednorodnej pobrane funkcją `gl.getUniformLocation`, zaś `v0, v1` wartości, a `values` tablica (`Float32Array`) wartości jakie chcemy zapisać.
Oprócz dwóch powyższych funkcji istnieją też inne wersje funkcji `gl.uniform`. Należy ich szukać w dokumentacji WebGL.

W przypadku zmiennych jednorodnych, które są macierzami do zapisu wartości takiej zmiennej korzystamy z funkcji `gl.uniformMatrix`, np.

```
gl.uniformMatrix4fv( location, transpose, values );
```

Parametr `location` jest to lokalizacja zmiennej jednorodnej, `transpose` odpowiada za to czy dane przesyłane do zmiennej transponować (`true`) czy nie (`false`), a `values` to tablica (`Float32Array`) z danymi macierzy.

WebGL stosuje zapis kolumnowy macierzy!

Rysowanie obiektów za pomocą VBO

Obiekty buforowe wierzchołków (ang. Vertex Buffer Objects – VBO) pojawiły się po raz pierwszy w OpenGL 1.5 i są wykorzystywane w każdej wersji WebGL. Obiekty te służą do przechowywania danych o wierzchołkach w pamięci karty graficznej.

Rysowanie obiektów za pomocą VBO

Obiekty buforowe wierzchołków (ang. Vertex Buffer Objects – VBO) pojawiły się po raz pierwszy w OpenGL 1.5 i są wykorzystywane w każdej wersji WebGL. Obiekty te służą do przechowywania danych o wierzchołkach w pamięci karty graficznej.

Utworzenie unikatowego obiektu VBO realizuje funkcja:

```
gl.createBuffer();
```

Funkcja zwraca obiekt VBO (`WebGLBuffer`).

Sprawdzenie czy dany obiekt jest obiektem buforowym:

```
gl.isBuffer( buffer );
```

`buffer` to obiekt, który chcemy sprawdzić. Funkcja zwraca `true` jeśli podany identyfikator jest obiektem buforowym, w przeciwnym razie zwraca `false`.

Usuwanie określonego obiektu VBO:

```
gl.deleteBuffer( buffer );
```

`buffer` jest to obiekt, który chcemy usunąć.

Utworzony obiekt VBO należy dowiązać za pomocą funkcji:

```
gl.bindBuffer( target, buffer );
```

`buffer` jest to obiekt VBO, a parametr `target` określa rodzaj obiektu buforowego:

- `gl.ARRAY_BUFFER` – obiekt buforowy tablic wierzchołków,
- `gl.ELEMENT_ARRAY_BUFFER` – obiekt buforowy indeksowanych tablic wierzchołków.

Jeśli jako `buffer` podamy `null` to aktualny obiekt buforowy zostaje odwiązany.

Ładowanie danych do obiektu buforowego:

```
gl.bufferData( target, data, usage );
```

`target` przyjmuje te same wartości co parametr `target` funkcji `gl.bindBuffer`, parametr `data` to tablica (`Float32Array`) z danymi. Ostatni parametr `usage` zawiera wskazówkę dla WebGL dotyczącą przewidywanej metody dostępu do danych zawartych w obiekcie buforowym.

Wartości parametru `usage`:

- `gl.STREAM_DRAW` – dane obiektu będą określone raz i używane do rysowania przez WebGL co najwyżej kilka razy,
- `gl.STATIC_DRAW` – dane obiektu będą określone raz i używane do rysowania przez WebGL wiele razy,
- `gl.DYNAMIC_DRAW` – dane obiektu będą ciągle modyfikowane i używane do rysowania przez WebGL wiele razy.

Wartości parametru `usage`:

- `gl.STREAM_DRAW` – dane obiektu będą określone raz i używane do rysowania przez WebGL co najwyżej kilka razy,
- `gl.STATIC_DRAW` – dane obiektu będą określone raz i używane do rysowania przez WebGL wiele razy,
- `gl.DYNAMIC_DRAW` – dane obiektu będą ciągle modyfikowane i używane do rysowania przez WebGL wiele razy.

Zmiana całości lub części danych zawartych w obiekcie buforowym:

```
gl.bufferSubData( target, offset, data );
```

Parametr `target` określa rodzaj obiektu buforowego, `offset` to numer pierwszego zmienianego elementu, `data` to tablica (`Float32Array`) z nowymi danymi.

Po utworzeniu obiektów VBO i załadowaniu do nich danych musimy powiązać dane zawarte w VBO z odpowiednim atrybutem wierzchołka. W tym celu włączamy tablicę atrybutu wierzchołka:

```
gl.enableVertexAttribArray( index );
```

Parametr `index` jest to lokalizacja zmiennej atrybutowej, którą otrzymujemy z funkcji `gl.getAttribLocation`.

Do wyłączenia tablicy atrybutu wierzchołka służy funkcja:

```
gl.disableVertexAttribArray( index );
```

Powiązanie danych VBO z atrybutem wierzchołka:

```
gl.vertexAttribPointer( index, size, type, normalized, stride,
offset );
```

Parametr `index` jest to lokalizacja zmiennej atrybutowej, `size` określa liczbę składowych przypadających na jeden wierzchołek (wartość 1, 2, 3 lub 4), `type` określa typ danych (`gl.FLOAT`, `gl.FIXED`), `normalized` określa czy dane mają być znormalizowane (`true`, `false`), `stride` określa przesunięcie (w bajtach) pomiędzy poszczególnymi elementami, `offset` określa położenie w pamięci VBO pierwszego elementu, podajemy w bajtach (`0` – dane są ściśle zapakowane).

W przypadku użycia tego samego programu cieniowania dla różnych obiektów przed wyrysowaniem każdego z obiektów musimy powiązać jego dane z programem cieniowania (funkcje `gl.bindBuffer`, `gl.vertexAttribPointer`).

W przypadku użycia tego samego programu cieniowania dla różnych obiektów przed wyrysowaniem każdego z obiektów musimy powiązać jego dane z programem cieniowania (funkcje `gl.bindBuffer`, `gl.vertexAttribPointer`).

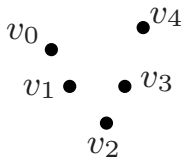
Rysowania obiektu dokonujemy za pomocą funkcji:

```
gl.drawArrays( mode, first, count );
```

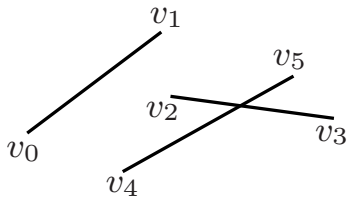
Parametr `mode` określa rodzaj rysowanego prymitywu. Parametr `first` wskazuje pierwszy element, a parametr `count` określa liczbę renderowanych elementów.

Parametr `mode` funkcji `gl.drawArrays` może przyjąć jedną z wartości:

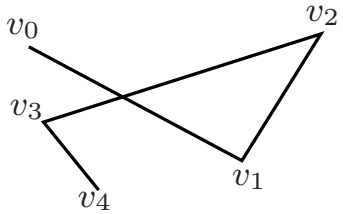
- `gl.POINTS`,
- `gl.LINES`,
- `gl.LINE_STRIP`,
- `gl.LINE_LOOP`,
- `gl.TRIANGLES`,
- `gl.TRIANGLE_STRIP`,
- `gl.TRIANGLE_FAN`.



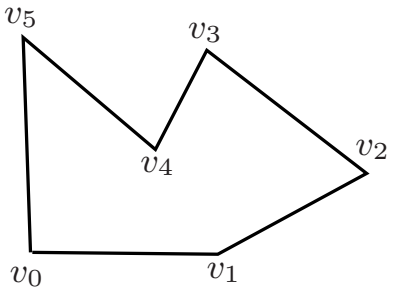
gl.POINTS



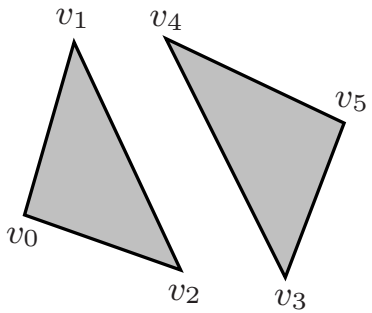
gl.LINES



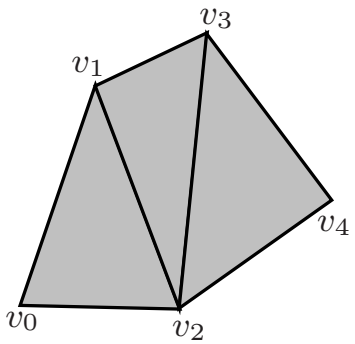
gl.LINE_STRIP



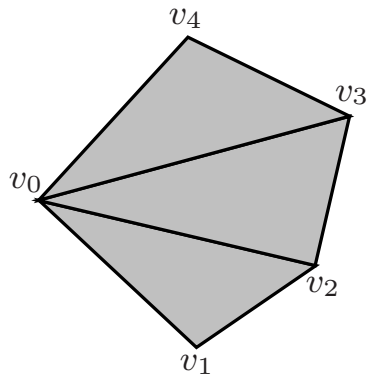
gl.LINE_LOOP



gl.TRIANGLES



gl.TRAINGLE_STRIP



gl.TRIANGLE_FAN