Wprowadzenie do shaderów

Krzysztof Gdawiec

UNIWERSYTET ŚLĄSKI
INSTYTUT INFORMATYKI

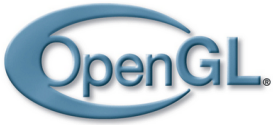
OpenGL (Open Graphics Library) jest wieloplatformową biblioteką pozwalającą na niskopoziomowy dostęp do karty graficznej i maksymalne wykorzystanie jej potencjału.

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.

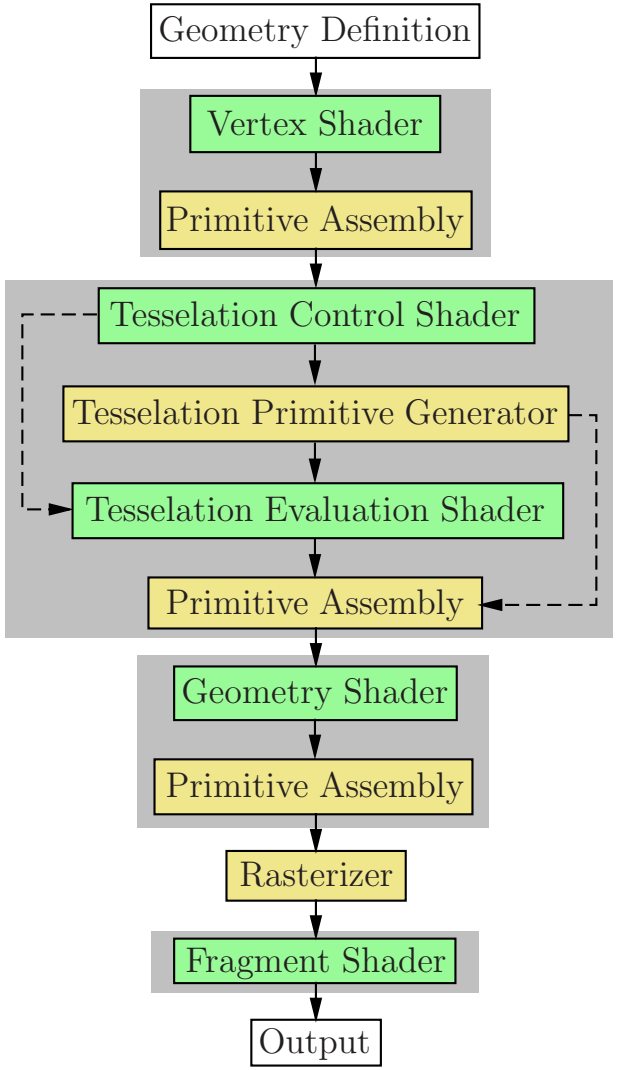
OpenGL	GLSL	Kiedy
1.0	–	1993
1.1	–	1997
1.2	–	1998
1.3	–	2001
1.4	–	2002
1.5	–	2003
2.0	1.10	2004
2.1	1.20	2006

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

Oprócz biblioteki OpenGL przydatne są również inne biblioteki:

- GLFW,
- GLEW (OpenGL Extension Wrangler Library),
- GLM (OpenGL Mathematics),
- DevIL, ResIL, SOIL, FreeImage.

Programowalny potok graficzny



Bufor ramki (ang. frame buffer)

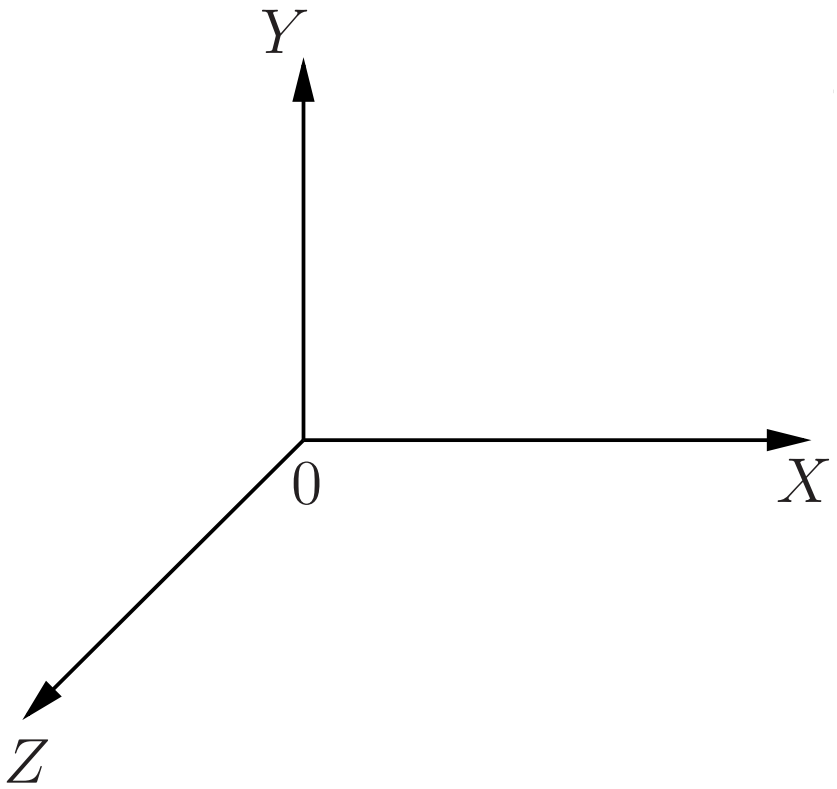
Bufor ramki w OpenGL składa się z:

- 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.

Układ współrzędnych

W OpenGL-u używany jest prawoskrętny układ współrzędnych kartezjańskich.



Typy danych

Typ	L.bitów	Opis
`GLboolean`	1	Typ logiczny
`GLbyte`	8	Liczba całkowita ze znakiem
`GLubyte`	8	Liczba całkowita bez znaku
`GLchar`	8	Znak tekstowy
`GLshort`	16	Liczba całkowita ze znakiem
`GLushort`	16	Liczba całkowita bez znaku
`GLint`	32	Liczba całkowita ze znakiem
`GLuint`	32	Liczba całkowita bez znaku
`GLint64`	64	Liczba całkowita ze znakiem
`GLuint64`	64	Liczba całkowita bez znaku
`GLsizei`	32	Nieujemna liczba całkowita
`GLenum`	32	Typ wyliczeniowy
`GLintptr`	*	Wskaźnik na liczbę całkowitą ze znakiem
`GLsizeiptr`	*	Wskaźnik na nieujemną liczbę całkowitą bez znaku
`GLsync`	*	Wskaźnik na obiekt synchronizujący
`GLbitfield`	32	Pole bitowe
`GLhalf`	16	Liczba zmiennoprzecinkowa połówkowej precyzji
`GLfloat`	32	Liczba zmiennoprzecinkowa pojedynczej precyzji
`GLclampf`	32	Liczba zmiennoprzecinkowa pojedynczej precyzji z $[0, 1]$
`GLdouble`	64	Liczba zmiennoprzecinkowa podwójnej precyzji
`GLclampd`	64	Liczba zmiennoprzecinkowa podwójnej precyzji z $[0, 1]$

Konwencja nazewnictwa funkcji

`rtype Name{1|2|3|4}{b|s|i|i64|f|d|ub|us|ui|ui64}{v}(args)`

- `rtype` – typ zwracany przez funkcję,
- `Name` – nazwa funkcji poprzedzona przedrostkiem `gl`,
- `1, 2, 3, 4` – liczba argumentów funkcji,
- `b|s|i|i64|f|d|ub|us|ui|ui64` – argumenty typu: `GLbyte`, `GLshort`, `GLint`, `GLint64`, `GLfloat`, `GLdouble`, `GLubyte`, `GLushort`, `GLuint`, `GLuint64`,
- `v` – argument funkcji stanowi tablica wartości (w tym wypadku nie występuje określenie liczby argumentów),
- `args` – argumenty funkcji.

Maszyna stanów i wskazówki

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

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

Maszyna stanów i wskazówki

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

Ważną cechą maszyny stanów OpenGL 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

```
void glEnable(GLenum capability);
```

a do wyłączania służy funkcja

```
void glDisable(GLenum capability);
```

Jeśli chcemy sprawdzić czy jakaś zmienna stanu jest włączona, to korzystamy z funkcji

```
GLboolean glIsEnabled(GLenum capability);
```

Jeśli chcemy sprawdzić czy jakaś zmienna stanu jest włączona, to korzystamy z funkcji

```
GLboolean glIsEnabled(GLenum capability);
```

Wiele innych funkcji OpenGL może przyjmować wartości zapamiętywane do czasu ich zmiany. W takim przypadku do odczytania aktualnej wartości zmiennej stanu służą funkcje:

```
void glGetBooleanv(GLenum pname, GLboolean *params);
void glGetDoublev(GLenum pname, GLdouble *params);
void glGetFloatv(GLenum pname, GLfloat *params);
void glGetIntegerv(GLenum pname, GLint *params);
```

W implementacjach OpenGL często możemy natknąć się na dwa sposoby wykonania tego samego zadania – szybszy, ale generujący nieco gorszy graficznie efekt lub wolniejszy, ale generujący atrakcyjniejszy obraz.

W implementacjach OpenGL często możemy natknąć się na dwa sposoby wykonania tego samego zadania – szybszy, ale generujący nieco gorszy graficznie efekt lub wolniejszy, ale generujący atrakcyjniejszy obraz.

W przypadkach występowania dwóch sposobów możemy dać wskazówkę bibliotece OpenGL, którego sposobu chcemy użyć. Służy do tego funkcja:

```
void glHint(GLenum target, GLenum hint);
```

Parametr `target` określa zachowanie, które chcemy zmienić, zaś parametr `hint` informuje bibliotekę OpenGL czy bardziej zależy nam na prędkości czy jakości (`GL_FASTEST`, `GL_NICEST`, `GL_DONT_CARE`). Wszystkie wskazówki domyślnie mają wartość `GL_DONT_CARE`.

Obsługa błędów

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

```
GLenum glGetError();
```

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

Wprowadzenie do GLSL

GLSL (ang. OpenGL Shading Language) czyli język programów cieniowania lub w skrócie shadery wprowadzono w wersji 2.0 OpenGL. Język ten w każdej wersji umożliwia tworzenie dwóch rodzajów programów cieniowania:

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

Programy cieniowanie dostępne w różnych wersjach OpenGL:

- shader geometrii (ang. geometry shader) – od wersji 3.2,
- shader teselacji (ang. tessellation shader) – od wersji 4.0,
- shader obliczeń (ang. compute shader) – od wersji 4.3.

Przed wprowadzeniem GLSL do podstawowej części biblioteki shadery były dostępne w rozszerzeniach: ARB_shader_objects, ARB_vertex_shader, ARB_fragment_shader, ARB_shading_language_100.

Shader wierzchołków wykonuje wszelkiego typu operacje na wierzchołkach, są to m.in.

- transformacja wierzchołków (główne zadanie tego shadera),
- transformacja i normalizacja normali,
- generowanie i transformacja współrzędnych tekstur,
- obliczenie oświetlenia poszczególnych wierzchołków,
- obliczenie koloru.

Przetwarzaniem fragmentów zajmuje się shader fragmentów. Wykonuje on m.in. następujące operacje:

- obliczenie koloru fragmentu (główne zadanie tego shadera),
- obliczenie współrzędnych tekstury,
- odczyt danych tekstury,
- obliczenia mgły,
- sumowanie kolorów.

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

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

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

Składnia GLSL 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 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, w której napisany jest dany shader. Jest ona obowiązkowa od wersji 1.20 GLSL (OpenGL w wersji 2.1). Jej brak oznacza, że jest on napisany w wersji 1.10 języka (OpenGL w wersji 2.0). 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.

```
#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`, `uint`, `float` (typy `bool` i `int` nie muszą być wspierane przez kartę graficzną, ale dotyczy to tylko starych kart)

Wektorowe: `vec2`, `vec3`, `vec4`, `bvec2`, `bvec3`, `bvec4`, `ivec2`, `ivec3`, `ivec4`, `uvec2`, `uvec3`, `uvec4`

Macierzowe: `mat2`, `mat2x2`, `mat3`, `mat3x3`, `mat4`, `mat4x4`, `mat2x3`, `mat2x4`, `mat3x2`, `mat3x4`, `mat4x2`, `mat4x3`

Typy `int`, `uint` oraz ich odpowiedniki wektorowe mogą być poddawane automatycznej konwersji do typu `float` oraz jego odpowiednika wektorowego.

W OpenGL 4.0 wprowadzono obsługę typu `double` i jego typów pochodnych: `dvecX`, `dmatX`, gdzie `X` to wymiar jak w przypadku innych typów wektorowych i macierzowych.

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, ale wyłącznie w zakresie danej grupy (ang. swizzling), np.

```
vec4 v;
v.xyzw;
v.rbg;
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, ale wyłącznie w zakresie danej grupy (ang. swizzling), np.

```
vec4 v;
v.xyzw;
v.rbg;
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`

`in` – określa powiązanie z poprzednim etapem potoku. Zmienna jest kopiowa z poprzedniego etapu. W przypadku funkcji określa parametry wejściowe funkcji. Wszelkie zmiany tego parametru wewnątrz funkcji nie mają wpływu na jego wartość poza funkcją.

Kwalifikatory typów

`const`

`in` – określa powiązanie z poprzednim etapem potoku. Zmienna jest kopiowa z poprzedniego etapu. W przypadku funkcji określa parametry wejściowe funkcji. Wszelkie zmiany tego parametru wewnątrz funkcji nie mają wpływu na jego wartość poza funkcją.

`out` – określa powiązanie z następnym etapem potoku. Zmienna jest przekazywana do kolejnego etapu. W przypadku funkcji określa parametry wyjściowe funkcji. Parametry z tym kwalifikatorem nie są inicjalizowane podczas wywołania funkcji. Przypisanie ich wartości następuje wewnątrz funkcji.

Kwalifikatory typów

`const`

`in` – określa powiązanie z poprzednim etapem potoku. Zmienna jest kopiowa z poprzedniego etapu. W przypadku funkcji określa parametry wejściowe funkcji. Wszelkie zmiany tego parametru wewnątrz funkcji nie mają wpływu na jego wartość poza funkcją.

`out` – określa powiązanie z następnym etapem potoku. Zmienna jest przekazywana do kolejnego etapu. W przypadku funkcji określa parametry wyjściowe funkcji. Parametry z tym kwalifikatorem nie są inicjalizowane podczas wywołania funkcji. Przypisanie ich wartości następuje wewnątrz funkcji.

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

`uniform` – określa globalne zmienne jednorodne dostępne we wszystkich rodzajach shaderów. Zmienne oznaczone tym kwalifikatorem są stałe w obrębie shaderów, a ich wartość definiowana jest przez wywołanie odpowiednich funkcji OpenGL.

`uniform` – określa globalne zmienne jednorodne dostępne we wszystkich rodzajach shaderów. Zmienne oznaczone tym kwalifikatorem są stałe w obrębie shaderów, a ich wartość definiowana jest przez wywołanie odpowiednich funkcji OpenGL.

W przypadku shadera wierzchołków wyróżniamy jeszcze zmienne atrybutowe (w wersji 1.10, 1.20 GLSL oznaczone kwalifikatorem `attribute`). Są to zmienne wejściowe (`in`) reprezentujące atrybuty wierzchołka, np. położenie, kolor, normal.

`uniform` – określa globalne zmienne jednorodne dostępne we wszystkich rodzajach shaderów. Zmienne oznaczone tym kwalifikatorem są stałe w obrębie shaderów, a ich wartość definiowana jest przez wywołanie odpowiednich funkcji OpenGL.

W przypadku shadera wierzchołków wyróżniamy jeszcze zmienne atrybutowe (w wersji 1.10, 1.20 GLSL oznaczone kwalifikatorem `attribute`). Są to zmienne wejściowe (`in`) reprezentujące atrybuty wierzchołka, np. położenie, kolor, normal.

W GLSL 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`

```
void main()
{
   ...
}
```

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 shaderach 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 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 OpenGL

- 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:

```
GLuint glCreateShader(GLenum type);
```

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

- `GL_VERTEX_SHADER`,
- `GL_TESS_CONTROL_SHADER`,
- `GL_TESS_EVALUATION_SHADER`,
- `GL_GEOMETRY_SHADER`,
- `GL_FRAGMENT_SHADER`,
- `GL_COMPUTE_SHADER`.

Funkcja zwraca unikatowy identyfikator obiektu programu cieniowania, który jest wykorzystywany w dalszych operacjach. W przypadku wystąpienia błędu funkcja zwraca wartość 0.

Załadowanie kodu programu cieniowania dla wybranego obiektu realizuje funkcja:

```
void glShaderSource(GLuint shader, GLsizei count, const GLchar **string,
 const GLint *length);
```

Parametr `shader` określa identyfikator obiektu programu cieniowania. Parametr `count` zawiera liczbę łańcuchów znaków składających się na program cieniowania. Tabelę wskaźników do poszczególnych łańcuchów znaków zawiera parametr `string`, a długość kolejnych łańcuchów tablica `length`.

Jeżeli parametr `length` zawiera wartość `NULL`, każdy łańcuch znaków musi być zakończony znakiem `NULL`.

Wywołanie funkcji `glShaderSource` z tym samym identyfikatorem zastępuje poprzedni kod programu cieniowania.

Kompilacja kodu programu cieniowania wykonywana jest przez funkcję:

```
void glCompileShader(GLuint shader);
```

Parametr `shader` to identyfikator obiektu programu cieniowania.

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

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

Obiekt programu możemy utworzyć za pomocą funkcji:

```
GLuint glCreateProgram();
```

Funkcja zwraca unikatowy identyfikator obiektu programu. W przypadku wystąpienia błędu funkcja zwraca wartość 0.

Obiekt programu możemy utworzyć za pomocą funkcji:

```
GLuint glCreateProgram();
```

Funkcja zwraca unikatowy identyfikator obiektu programu. W przypadku wystąpienia błędu funkcja zwraca wartość 0.

Następnie dołączamy program cieniowania do obiektu programu:

```
void glAttachShader(GLuint program, GLuint shader);
```

Parametry `program` i `shader` są to identyfikatory (odpowiednio) obiektu programu i obiektu programu cieniowania. Jeśli wybrany program cieniowania jest już dołączony do obiektu programu generowany jest błąd `GL_INVALID_OPERATION`.

Po dołączeniu programu cieniowania do obiektu programu musimy dokonać konsolidacji:

```
void glLinkProgram(GLuint program);
```

Parametr `program` jest to identyfikator obiektu programu.

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

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

Opcjonalnie przed wykonaniem programów cieniowania zawartych w obiekcie programu możliwe jest sprawdzenie ich poprawności przy użyciu funkcji:

```
void glValidateProgram(GLuint program);
```

Parametr `program` jest to identyfikator obiektu programów.

Wynik sprawdzania poprawności umieszczany jest w zmiennej stanu `GL_VALIDATE_STATUS` obiektu programu.

Użycie programu cieniowania jest dokonywane za pomocą funkcji:

```
void glUseProgram(GLuint program);
```

Parametr `program` jest identyfikatorem obiektu programów. Podanie wartości 0 powoduje wyłączenie programowalnej części potoku OpenGL.

Użycie programu cieniowania jest dokonywane za pomocą funkcji:

```
void glUseProgram(GLuint program);
```

Parametr `program` jest identyfikatorem obiektu programów. Podanie wartości 0 powoduje wyłączenie programowalnej części potoku OpenGL.

Usuwanie obiektu programu cieniowania wykonujemy za pomocą funkcji:

```
void glDeleteShader(GLuint shader);
```

Parametr `shader` jest to identyfikator usuwanego shadera. Jeżeli usuwany obiekt programu cieniowania 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:

```
void glDetachShader(GLuint program, GLuint shader);
```

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

Usuwania obiektu programów dokonujemy za pomocą funkcji:

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

Parametr `program` jest to identyfikator obiektu programów.

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 obiekty programu cieniowania są one odłączane i otrzymują status usuniętego poprzez wewnętrzne wywołanie funkcji `glDeleteShader`.

Zmienne atrybutowe

Pobranie położenia atrybutu w shaderze wierzchołków:

```
GLint glGetAttribLocation(GLuint program, const GLchar* name);
```

Parametr `program` jest to identyfikator obiektu programu, a `name` jest to nazwa zmiennej atrybutowej, której położenie chcemy poznać.

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

Sposób przekazywania wartości do zmiennych atrybutowych zobaczymy przy omawianiu VAO i VBO.

Zmienne jednorodne

Pobieranie położenia aktywnej zmiennej jednorodnej:

```
GLint glGetUniformLocation(GLuint program, const GLchar* name);
```

Parametr `program` jest to identyfikator obiektu programu, a `name` jest to nazwa zmiennej jednorodnej, której położenie chcemy poznać. Funkcja zwraca -1 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 OpenGL i służą do tego funkcje `glUniform`, np.

```
void glUniform2f(GLint location, GLfloat v0, GLfloat v1);
void glUniform2fv(GLint location, GLsizei count, const GLfloat *value);
```

Parametr `location` jest to położenie zmiennej jednorodnej pobrane funkcją `glGetUniformLocation`, `v0`, `v1` wartości jakie chcemy zapisać, `count` liczba zapisywanych elementów, a nie składowych, `value` tablica z wartościami.
Oprócz dwóch powyższych funkcji istnieją też inne wersje funkcji `glUniform`. Należy ich szukać w dokumentacji OpenGL.

W przypadku zmiennych jednorodnych, które są macierzami do zapisu wartości takiej zmiennej korzystamy z funkcji:

```
void glUniformMatrix4fv(GLint location, GLsizei count, GLboolean
transpose, const GLfloat *value);
```

Parametr `location` jest to lokalizacja zmiennej jednorodnej, `cout` jest to liczba macierzy, `transpose` odpowiada za to czy dane przesyłane do zmiennej transponować (`GL_TRUE`) czy nie (`GL_FALSE`), a `value` to wskaźnik na tablicę z danymi macierzy.

OpenGL stosuje zapis kolumnowy macierzy.

Rysowanie za pomocą VBO i VAO

Obiekty buforowe wierzchołków (ang. Vertex Buffer Objects – VBO) pojawiły się w OpenGL 1.5. Obiekty te służą do przechowywania danych o wierzchołkach w pamięci karty graficznej.

Rysowanie za pomocą VBO i VAO

Obiekty buforowe wierzchołków (ang. Vertex Buffer Objects – VBO) pojawiły się w OpenGL 1.5. Obiekty te służą do przechowywania danych o wierzchołkach w pamięci karty graficznej.

Utworzenie unikatowych identyfikatorów obiektów buforowych realizuje funkcja:

```
void glGenBuffers(GLsizei n, GLuint *buffers);
```

`n` liczba generowanych identyfikatorów, `buffers` wskaźnik na tablicę, do której zapisane zostaną wygenerowane identyfikatory.

Sprawdzenie czy dany identyfikator jest identyfikatorem obiektu buforowego:

```
GLboolean glIsBuffer(GLuint buffer);
```

`buffer` identyfikator, który chcemy sprawdzić. Funkcja zwraca `GL_TRUE` jeśli podany identyfikator jest identyfikatorem obiektu buforowego, w przeciwnym razie zwraca `GL_FALSE`.

Usuwanie wybranej grupy obiektów buforowych:

```
void glDeleteBuffers(GLsizei n, const GLuint *buffers);
```

`n` liczba usuwanych identyfikatorów, których identyfikatory znajdują się w tablicy `buffers`.

Utworzony identyfikator należy dowiązać do obiektu buforowego za pomocą funkcji:

```
void glBindBuffer(GLenum target, GLuint buffer);
```

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

- `GL_ARRAY_BUFFER` – obiekt buforowy tablic wierzchołków,
- `GL_ELEMENT_ARRAY_BUFFER` – obiekt buforowy indeksowanych tablic wierzchołków,
- `GL_COPY_READ_BUFFER`, `GL_COPY_WRITE_BUFFER`, `GL_PIXEL_PACK_BUFFER`, `GL_PIXEL_UNPACK_BUFFER`, `GL_TEXTURE_BUFFER`, `GL_TRANSFORM_FEEDBACK_BUFFER`, `GL_UNIFORM_BUFFER`.

Ładowanie danych do obiektu buforowego można zrealizować na dwa sposoby:

- wykorzystanie tradycyjnego modelu aplikacji OpenGL, tj. kopiowaniu danych do obiektu bufora z tablicy zawartej w pamięci operacyjnej komputera,
- wykorzystanie tzw. odwzorowania obiektu buforowego, który pozwala na bezpośrednie operowanie na pamięci obiektu.

Ładowanie danych do obiektu buforowego:

```
void glBufferData(GLenum target, GLsizeiptr size, const GLvoid *data,
GLenum usage);
```

`target` przyjmuje te same wartości co parametr `target` funkcji `glBindBuffer`, parametr `size` to rozmiar danych bufora obiektu buforowego (w bajtach), parametr `data` to wskaźnik na dane ładowane do obiektu buforowego (wartość różna od `NULL` kopiuje dane, podanie `NULL` rezerwuje pamięć bufora obiektu buforowego). Ostatni parametr `usage` zawiera wskazówkę dla biblioteki OpenGL dotyczącą przewidywanej metody dostępu do danych zawartych w obiekcie buforowym.

Wartości parametru `usage`: `GL_STREAM_DRAW`, `GL_STREAM_READ`, `GL_STREAM_COPY`, `GL_STATIC_DRAW`, `GL_STATIC_READ`, `GL_STATIC_COPY`, `GL_DYNAMIC_DRAW`, `GL_DYNAMIC_READ`, `GL_DYNAMIC_COPY`.

Nazwa każdej z wartości `usage` składa się z dwóch części. Pierwsza część mówi o częstości dostępu (modyfikacji i użycia), a druga o naturze dostępu.

Częstość dostępu może być następująca:

- `STREAM` – dane obiektu będą określone raz i używane co najwyżej kilka razy,
- `STATIC` – dane obiektu będą określone raz i używane wiele razy,
- `DYNAMIC` – dane obiektu będą ciągle modyfikowane i używane wiele razy.

Natura dostępu może być następująca:

- `DRAW` – dane będą używane jako źródło dla OpenGL-a do rysowania i specyfikacji obrazu,
- `READ` – dane są używane do ich zwrócenia kiedy aplikacja o nie zapyta,
- `COPY` – dane będą używane jako źródło dla OpenGL-a do rysowania i specyfikacji obrazu.

Natura dostępu może być następująca:

- `DRAW` – dane będą używane jako źródło dla OpenGL-a do rysowania i specyfikacji obrazu,
- `READ` – dane są używane do ich zwrócenia kiedy aplikacja o nie zapyta,
- `COPY` – dane będą używane jako źródło dla OpenGL-a do rysowania i specyfikacji obrazu.

Zmianę całości lub części danych zawartych w obiekcie buforowym:

```
void glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size,
const GLvoid *data);
```

Parametr `target` określa rodzaj obiektu buforowego, `offset` to numer pierwszego zmienianego elementu, `size` to liczba modyfikowanych elementów, `data` wskaźnik do tablicy z nowymi danymi.

W OpenGL 3.0 wprowadzono obiekty tablic wierzchołków (ang. Vertex Array Object – VAO). W pojedynczym VAO możemy umieszczać różne obiekty VBO, a następnie jednym poleceniem wyrysować.

W OpenGL 3.0 wprowadzono obiekty tablic wierzchołków (ang. Vertex Array Object – VAO). W pojedynczym VAO możemy umieszczać różne obiekty VBO, a następnie jednym poleceniem wyrysować.

Utworzenie unikatowych identyfikatorów dla obiektów VAO:

```
void glGenVertexArrays(GLsizei n, GLuint *arrays);
```

`n` liczba generowanych identyfikatorów, `arrays` wskaźnik na tablicę, do której zapisane zostaną wygenerowane identyfikatory.

Sprawdzanie czy dany identyfikator jest identyfikatorem VAO:

```
GLboolean glIsVertexArray(GLuint array);
```

`array` identyfikator, który chcemy sprawdzić. Funkcja zwraca `GL_TRUE` jeśli podany identyfikator jest identyfikatorem VAO, w przeciwnym razie zwraca `GL_FALSE`.

Dowiązanie obiektu VAO:

```
void glBindVertexArray(GLuint array);
```

`array` identyfikator obiektu VAO.

Dowiązanie obiektu VAO:

```
void glBindVertexArray(GLuint array);
```

`array` identyfikator obiektu VAO.

Usunięcie wybranej grupy obiektów VAO:

```
void glDeleteVertexArrays(GLsizei n, const GLuint *arrays);
```

`n` liczba usuwanych identyfikatorów, które znajdują się w tablicy `arrays`.

Mając utworzony i dowiązany obiekt VAO możemy do niego dołączyć obiekty VBO. W tym celu dowiązujemy VBO i ładujemy dane do obiektu tak jak to robiliśmy do tej pory.

Po dodaniu obiektu VBO do VAO musimy powiązać dane zawarte w VBO z odpowiednim atrybutem wierzchołka. W tym celu włączamy tablicę atrybutu wierzchołka:

```
void glEnableVertexAttribArray(GLuint index);
```

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

Powiązanie danych VBO z atrybutem wierzchołka:

```
void glVertexAttribPointer(GLuint index, GLint size, GLenum type,
GLboolean normalized, GLsizei stride, const GLvoid *pointer);
```

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, `normalized` określa czy dane mają być znormalizowane, `stride` określa przesunięcie (w bajtach) pomiędzy poszczególnymi elementami, `pointer` określa położenie w pamięci VBO pierwszego elementu.

Kiedy mamy już utworzony obiekt VAO zapełniony obiektami VBO oraz mamy powiązanie VBO z atrybutami wierzchołka możemy wyrysować obiekt.

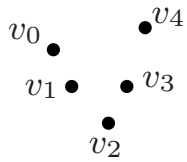
W tym celu najpierw dowiązujemy odpowiedni obiekt VAO za pomocą funkcji `glBindVertexArray`, a następnie za pomocą funkcji `glDrawArrays` rysujemy obiekt.

```
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

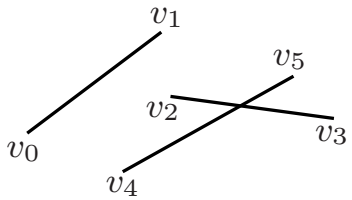
Parametr `mode` określa rodzaj rysowanego prymitywu. Parametr `first` wskazuje pierwszy element, a parametr `count` określa liczbę renderowanych elementów.

W profilu rdzennym parametr `mode` funkcji `glDrawArrays` może przyjąć jedną z wartości: `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_TRIANGLES`, `GL_LINES_ADJACENCY`, `GL_LINE_STRIP_ADJACENCY`, `GL_TRIANGLES_ADJACENCY`, `GL_TRIANGLE_STRIP_ADJACENCY`, `GL_PATCHES`.

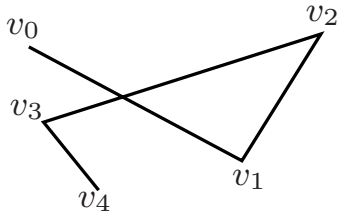
Prymitywy `GL_LINES_ADJACENCY`, `GL_LINE_STRIP_ADJACENCY`, `GL_TRIANGLES_ADJACENCY`, `GL_TRIANGLE_STRIP_ADJACENCY` są związane z shaderem geometrii, a `GL_PATCHES` z shaderem teselacji. Dlatego poznamy je na wykładzie na temat tych shaderów.



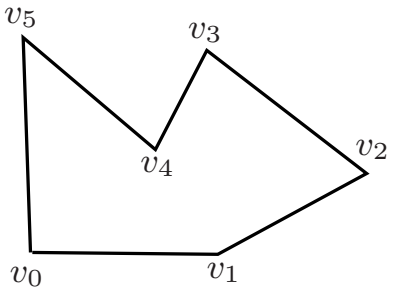
GL_POINTS



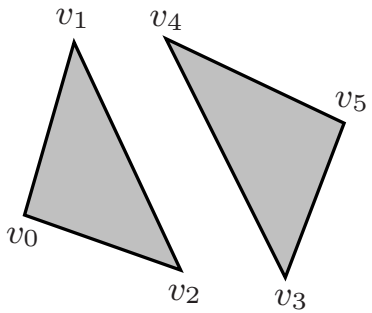
GL_LINES



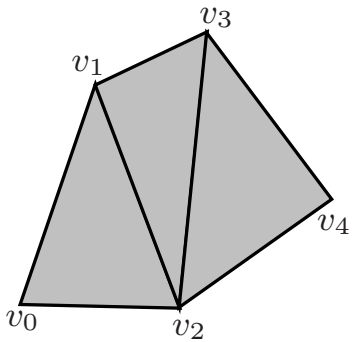
GL_LINE_STRIP



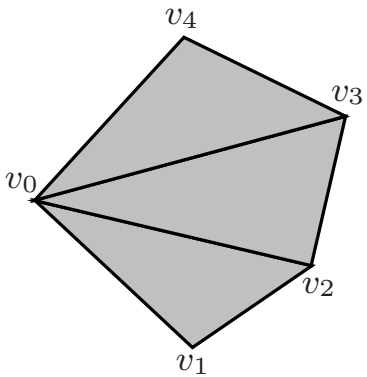
GL_LINE_LOOP



GL_TRIANGLES



GL_TRAINGLE_STRIP



GL_TRIANGLE_FAN