Janusz Ganczarski

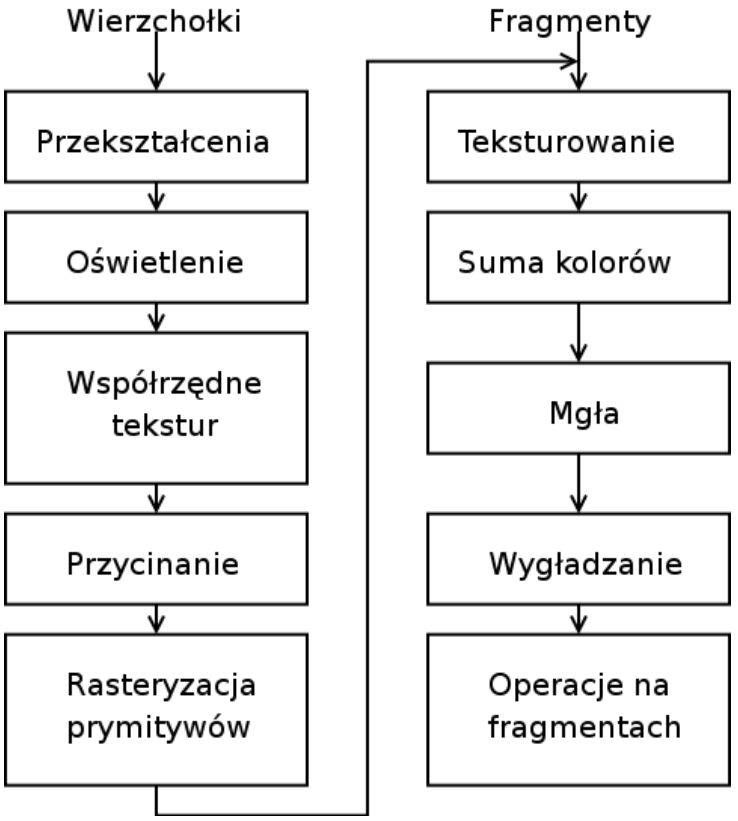
OpenGL
GLSL

Spis treści

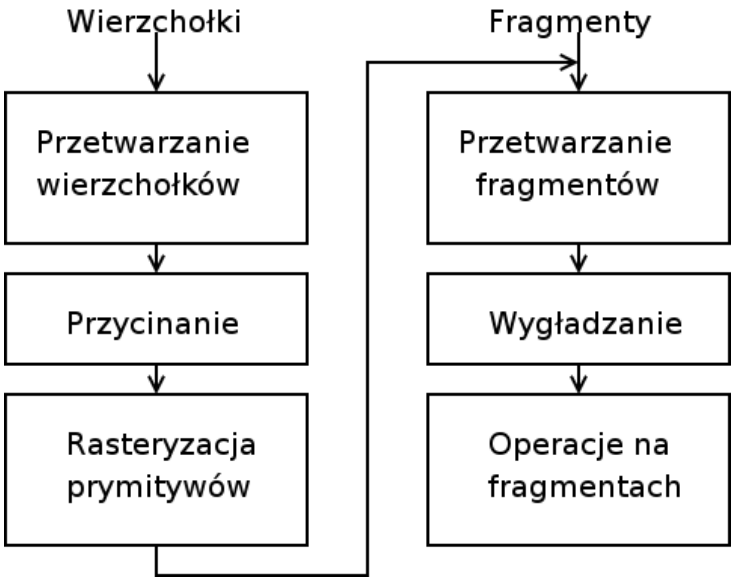
1. GLSL

GLSL (ang. *OpenGL Shading Language* lub *GLslang*), czyli język programów cieniowania, został wprowadzony w wersji 2.0 biblioteki OpenGL. Język ten umożliwia tworzenie zarówno programów cieniowania wierzchołków (ang. *vertex shader*) jak i programów cieniowania fragmentów (ang. *fragment shader*). Przed wprowadzeniem GLSL do podstawowej części biblioteki programy cieniowania opisane były łącznie w czterech rozszerzeniach: `ARB_shader_objects`, `ARB_vertex_shader`, `ARB_fragment_shader` i `ARB_shading_language_100`.

Programy te opisują odpowiednio przekształcenia geometryczne wierzchołków oraz operacje na fragmentach i przejmują odpowiedzialność za niektóre elementy klasycznego potoku graficznego w bibliotece OpenGL - patrz rysunki 1 i 2.



Rysunek 1. Klasyczny potok przetwarzania w OpenGL



Rysunek 2. Programowalny potok przetwarzania w OpenGL

Język GLSL jest wspólny składniowo dla obu rodzajów programów, ale ich odrębny charakter powoduje, że część operacji jest specyficzna tylko dla określonego rodzaju programu.

Mówiąc o języku GLSL trzeba jeszcze wspomnieć o niskopoziomowych programach cieniowania, które są dostępne jako rozszerzenia: `ARB_vertex_program` i `ARB_fragment_program`. To historycznie pierwsza technika programowania potoku graficznego w bibliotece OpenGL (oczywiście poza rozszerzeniami opracowanymi odrębnie przez różnych producentów procesorów graficznych), którą można porównać do języka asemblera w procesorach. Niskopoziomowe programy cieniowania, choć ciągle popularne, najprawdopodobniej nigdy nie wejdą do specyfikacji biblioteki OpenGL.

Wersja 2.1 biblioteki OpenGL udostępnia język GLSL w wersji 1.20. Wersja 1.10 tego języka była dostępna w wersji 2.0 biblioteki OpenGL, a wersję 1.00 opisywały wymienione na wstępnie rozszerzenia. Wersję języka GLSL obsługiwaną przez daną implementację biblioteki OpenGL zawiera zmienna stanu `GL_SHADING_LANGUAGE_VERSION`, której wartość można odczytać korzystając z funkcji `glGetString`.

1.1. Procesor wierzchołków i fragmentów

Za przetwarzanie wierzchołków i zarazem wykonywanie programów cieniowania wierzchołków odpowiedzialny jest tzw. procesor wierzchołków (ang.

vertex processor). Procesor wierzchołków przejmuje wykonanie następujących elementów potoku OpenGL:
— transformacja wierzchołków,
— transformacja i normalizacja wektorów normalnych,
— generowanie i transformacja współrzędnych tekstur,
— obliczanie oświetlenia poszczególnych wierzchołków,
— obliczanie koloru.

Przetwarzaniem fragmentów za pomocą programów cieniowania fragmentów zajmuje się tzw. procesor fragmentów (ang. *fragment processor*). Procesor fragmentów wykonuje następujące operacje na fragmentach:
— obliczenia koloru i współrzędnych tekstury dla piksela,
— odczyt danych tekstury,
— obliczenia mgły,
— sumowanie kolorów.

Kilka pierwszych generacji programowalnych procesorów graficznych, przeznaczonych na rynek konsumencki, zawierało w swojej strukturze odrębne jednostki będące odpowiednikami procesora wierzchołków i procesora fragmentów. Wprowadzony w 2006 roku procesor graficzny N80 firmy NVIDIA oraz w 2007 roku procesor R600 firmy AMD (ATI) zawierają już zunifikowane jednostki przetwarzania zwane procesorami strumieni (ang. *stream procesors*), które w zależności od potrzeb wykonują programy cieniowania wierzchołków, programy cieniowania fragmentów lub nowy rodzaj programów cieniowania - programy cieniowania geometrii (ang. *geometry shader*). Architektura ta nosi angielską nazwę *unified shader*.

1.2. Podstawy składni

Składnia GLSL jest oparta na językach C i C++, stąd osoba znająca te języki ma znacznie ułatwione zadane. Poniższy opis jest ograniczony do niezbędnego minimum, a Czytelnika zainteresowanego bliższymi szczegółami gramatyki języka GLSL zapraszam do lektury specyfikacji.

1.2.1. Zbiór znaków

Program w języku GLSL jest ciągiem znaków będących podzbiorem znaków ASCII. Podzbiór ten zawiera małe litery `a-z`, duże litery `A-Z`, podkreślenie _, cyfry `0-9`, oraz znaki: `.`, `+`, `-`, `/`, `*`, `%`, `<`, `>`, `[`, `]`, `(`, `)`, {, }, ∧, `|`, `&`, ˜, `=`, `!`, `:`, `;`, `,` i `?`. Znak `#` jest zarezerwowany do użycia przez preprocesor.

Ponadto program może zawierać tzw. białe spacje (ang. *white space*), w skład których wchodzą znaki: spacja, tabulator poziomy i pionowy, wysów strony (FF), powrót karetki (CR) i wysów wiersza (LF).

Program w języku GLSL tworzy tablica ciągów znaków, które mogą być podzielone na wiersze. Podział na wiersze nie ma znaczenia przy kompilacji

programu, jest jednak elementem przydatnym np. do diagnostyki błędów. Wiersze numerowane są od 0.

1.2.2. Komentarze

GLSL wykorzystuje takie same komentarze jak język C++, tj. `//` rozpoczyna komentarz obowiązujący do końca linii, a komentarz dowolnej części programu określają pary znaków `/*` i `*/`.

1.2.3. Słowa zarezerwowane

Język GLSL w wersji 1.20 rezerwuje następujące słowa: `attribute`, `const`, `uniform`, `varying`, `centroid`, `break`, `continue`, `do`, `for`, `while`, `if`, `else`, `in`, `out`, `inout`, `float`, `int`, `void`, `bool`, `true`, `false`, `invariant`, `discard`, `return`, `mat2`, `mat3`, `mat4`, `mat2x2`, `mat2x3`, `mat2x4`, `mat3x2`, `mat3x3`, `mat3x4`, `mat4x2`, `mat4x3`, `mat4x4`, `vec2`, `vec3`, `vec4`, `ivec2`, `ivec3`, `ivec4`, `bvec2`, `bvec3`, `bvec4`, `sampler1D`, `sampler2D`, `sampler3D`, `samplerCube`, `sampler1DShadow`, `sampler2DShadow` i `struct`.

Specyfikacja GLSL zawiera także wykaz słów zarezerwowanych do użycia w przyszłych wersjach tego języka: `asm`, `class`, `union`, `enum`, `typedef`, `template`, `this`, `packed`, `goto`, `switch`, `default`, `inline`, `noinline`, `volatile`, `public`, `static`, `extern`, `external`, `interface`, `long`, `short`, `double`, `half`, `fixed`, `unsigned`, `lowp`, `mediump`, `highp`, `precision`, `input`, `output`, `hvec2`, `hvec3`, `hvec4`, `dvec2`, `dvec3`, `dvec4`, `fvec2`, `fvec3`, `fvec4`, `sampler2DRect`, `sampler3DRect`, `sampler2DRectShadow`, `sizeof`, `cast`, `namespace` i `using`.

1.2.4. Identyfikatory

Identyfikatory, czyli nazwy zmiennych, funkcji, struktur i selektorów pól mogą składać się z małych i dużych liter `a-z`, `A-Z`, podkreślenia `_` oraz cyfr `0-9`. Pierwszym znakiem identyfikatora nie może być cyfra, a identyfikatory zaczynające się przedrostkiem `gl_` są zarezerwowane do użycia przez OpenGL.

1.3. Preprocesor

Preprocesor w języku GLSL jest zbliżony do proprocesora wbudowanego w języku C++. Występujące różnice związane przede wszystkim są ze specyficznym przeznaczeniem języka GLSL.

1.3.1. Operatory

Generalnie semantyka preprocesora GLSL jest zbliżona do zdefiniowanej w języku C++. W tabeli 1 zestawiono wszystkie operatory preprocesora GLSL w kolejności od najwyższego (1) do najniższego (12) priorytetu.

priorytet	rodzaj	operator	kolejność wiązania
1	grupowanie w nawiasy	( )	-
2	jednoargumentowe	$defined + - \tilde{} !$	od prawej do lewej
3	multiplikatywne	$\& * / \%$	od lewej do prawej
4	addytywne	+ -	od lewej do prawej
5	przesuwanie bitowe	<< >>	od lewej do prawej
6	relacje	< > <= >=	od lewej do prawej
7	równość	== !=	od lewej do prawej
8	bitowe AND	&	od lewej do prawej
9	różnica symetryczna XOR	$\wedge$	od lewej do prawej
10	bitowe OR	|	od lewej do prawej
11	logiczne AND	&&	od lewej do prawej
12	logiczne OR	||	od lewej do prawej

Tabela 1: Pierwszeństwo operatorów preprocesora GLSL

Operator `defined` może być używany na dwa sposoby:

```
defined identifier
defined (identifier)
```

1.3.2. Instrukcje

Preprocesor w języku GLSL posiada następujące instrukcje: `#`, `#define`, `#undef`, `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`, `#error`, `#pragma`, `#extension`, `#version` i `#line`.

Pojedynczy znak `#` w linii jest ignorowany przez preprocesor. Instrukcje `#define` i `#undef` służą do definiowania i anulowania definicji makr. Ich funkcjonalność odpowiada analogicznym instrukcjom preprocesora języka C++. Także instrukcje `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` i `#endif` mają takie same znaczenie jak w języku C++, jednak preprocesor w języku GLSL nie wspiera stałych znakowych. Ponadto wyrażenia występujące w instrukcjach `#if` i `#elif` są ograniczone do wyrażeń operujących na stałych całkowitych oraz identyfikatorów obsługiwanych przez operator `defined`.

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

Instrukcja `#version` określa wersję języka GLSL, w której napisany jest dany program cieniowania. Jest ona obowiązkowa od wersji 1.20 języka GLSL, a jej brak w programie oznacza, że jest on napisany w wersji 1.10 języka. Instrukcja `#version` musi wystąpić na samym początku programu cieniowania, za wyjątkiem komentarzy i tzw. białych spacji.

Kolejna instrukcja `#pragma` umożliwia kontrolę nad ustawieniami kompilatora zależnymi do implementacji. W przypadku, gdy dane wyrażenie użyte w poleceniu `#pragma` nie jest obsługiwane przez bieżącą implementację, całość polecenia jest ignorowana. Do przyszłych zastosowań zarezerwowane jest wyrażenie:

```
#pragma STDGL
```

Programy napisane w języku GLSL mogą używać do optymalizacji następujących poleceń:

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

przy czym należy pamiętać, że domyślnie optymalizacja jest włączona dla każdego programu cieniowania. Zadaniem drugiej grupa poleceń:

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

jest zebranie podczas kompilacji programu cieniowania informacji przydatnych przy odpluskiwaniu. Domyślnie zbieranie tych informacji jest wyłączone.

Instrukcja `#extension` kontroluje współpracę programów cieniowania z rozszerzeniami języka GLSL obsługiwanymi przez implementację biblioteki OpenGL. Instrukcja ta występuje w dwóch postaciach:

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

Pierwsza postać wykonuje operacje na wybranym rozszerzeniu o nazwie (nie identyfikatorze) `extension_name`. Druga umożliwia globalne operacje na wszystkich dostępnych rozszerzeniach języka GLSL. Kwalifikator `behavior` określa wybrane zachowanie kompilatora GLSL i może przyjąć jedną z wartości:

— `require` - rozszerzenie o nazwie `extension_name` jest wymagane przez program, kwalifikator współpracuje wyłącznie z pierwszą wersją polecenia `#extension`,
— `enable` - włączenie rozszerzenia o nazwie `extension_name`, kwalifikator współpracuje wyłącznie z pierwszą wersją polecenia `#extension`,
— `warn` - kompilator generuje ostrzeżenie przy użyciu rozszerzenia o nazwie `extension_name`, w przypadku wersji instrukcji z `all` ostrzeżenie jest generowane przy użyciu każdego z dostępnych rozszerzeń,
— `disable` - wyłączenie rozszerzenia o nazwie `extension_name` lub wyłącznie wszystkich rozszerzeń (wersja z `all`).

Początkowy stan obsługi rozszerzeń kompilatora GLSL określa instrukcja:

```
#extension all : disable
```

Ostatnia nieopisana instrukcja preprocesora GLSL to `#line`, która jest pewną substytucją makra __LINE__, posiada dwie postacie:

```
#line line
#line line source_string_number
```

gdzie `line` i `source_string_number` są stałymi całkowitymi. Wywołanie instrukcji `#line` powoduje takie zachowanie kompilatora GLSL, jakby kompilowano wiersz tekstu źródłowego programu cieniowania o numerze `line` (powiększonego o jeden) w zbiorze wierszy o numerze `source_string_number`.

1.3.3. Wbudowane makra

GLSL posiada trzy standardowe makra: __LINE__, __FILE__ i __VERSION__. Pierwsze z makr określa numer bieżącej linii zbioru wierzy tekstu źródłowego programu cieniowania. Drugie makro __FILE__ zwraca numer bieżącego zbioru wierszy tekstu źródłowego programu cieniowania. Specyfika powyższych makr związana jest ze sposobem ładowania tekstu źródłowego programów cieniowania do obiektów programów. Ostatnie makro zwraca numer obsługiwanej wersji języka GLSL. W przypadku wersji 1.20 zwraca jest wartość całkowita 120.

Do przyszłego użycia przez preprocesor zarezerwowane są nazwy makr rozpoczynające się dwoma znakami podkreślenia (__) oraz z przedrostkiem `GL_`.

1.4. Podstawowe typy

Wszystkie zmienne w programie GLSL muszą być zadeklarowane przed pierwszym użyciem. W GLSL nie ma typów domyślnych, każda zmienna i funkcja musi mieć zadeklarowany typ oraz opcjonalne kwalifikatory. Trzeba także podkreślić, że GLSL jest językiem o silnej typizacji i niewielkich możliwościach automatycznej konwersji typów.

Podstawowe typy danych w języku GLSL przedstawiono w tabeli 2. Zauważmy, że GLSL nie posiada żadnego odpowiednika typów wskaźnikowych z C/C++. Macierze prostokątne zostały wprowadzone w wersji 1.20 języka GLSL.

nazwa	opis
void	typ pusty dla funkcji nie zwracającej wartości
bool	typ logiczny, wartość true lub false
int	liczba całkowita ze znakiem
float	liczba zmiennoprzecinkowa pojedynczej precyzji
vec2	dwuelementowy wektor z liczbami typu float
vec3	trójelementowy wektor z liczbami typu float
vec4	czteroelementowy wektor z liczbami typu float
bvec2	dwuelementowy wektor z elementami typu bool
bvec3	trójelementowy wektor z elementami typu bool
bvec4	czteroelementowy wektor z elementami typu bool
ivec2	dwuelementowy wektor z liczbami typu int
ivec3	trójelementowy wektor z liczbami typu int
ivec4	czteroelementowy wektor z liczbami typu int
mat2, mat2x2	macierz 2×2 z liczbami typu float
mat3, mat3x3	macierz 3×3 z liczbami typu float
mat4, mat4x4	macierz 4×4 z liczbami typu float
mat2x3	macierz 2×3 z liczbami typu float (dwie kolumny, trzy wiersze)
mat2x4	macierz 2×4 z liczbami typu float (dwie kolumny, cztery wiersze)
mat3x2	macierz 3×2 z liczbami typu float (trzy kolumny, dwa wiersze)
mat3x4	macierz 3×4 z liczbami typu float (trzy kolumny, cztery wiersze)
mat4x2	macierz 4×2 z liczbami typu float (cztery kolumny, dwa wiersze)
mat4x3	macierz 4×3 z liczbami typu float (cztery kolumny, trzy wiersze)
sampler1D	uchwyt dostępu do tekstury 1D
sampler2D	uchwyt dostępu do tekstury 2D
sampler3D	uchwyt dostępu do tekstury 3D
samplerCube	uchwyt dostępu do tekstury sześciennej
sampler1DShadow	uchwyt dostępu do tekstury 1D z porównaniem
sampler2DShadow	uchwyt dostępu do tekstury 2D z porównaniem

Tabela 2: Podstawowe typy GLSL

1.4.1. Niejawne konwersje typów

Jak napisaliśmy na wstępie GLSL ma niewielkie możliwości automatycznej konwersji typów. Sprowadza się to do konwersji z typu `int` do typu `float` oraz konwersji wektorów z elementami całkowitymi (`ivec2`, `ivec3` i `ivec4`) na wektory z liczbami zmiennoprzecinkowymi o analogicznych wymiarach (`vec2`, `vec3` i `vec4`).

1.4.2. Zakres widoczności zmiennych

Widoczność zmiennej zależy od miejsca jej deklaracji. Zmienne zadeklarowane poza funkcjami mają zasięg globalny rozpoczynający się od miejsca deklaracji.

1.4.3. Konstruktory

Konstruktory służą do zainicjowania wartości zmiennej i wykorzystują składnię identyczną jak przy wywołaniu funkcji, przy czym nazwą konstruktora jest podstawowy typ danych lub nazwa struktury zdefiniowanej przez użytkownika. Konstruktory mogą być także wykorzystane do wymuszenia konwersji typów danych, zbudowania większego typu z mniejszych (np. wektory), bądź zredukowania większego typu do mniejszego.

Oto dostępne konstruktory dla typów skalarnych:

```
int (bool)    // konwersja z bool do int
int (float)   // konwersja z float do int
float (bool)  // konwersja z bool do float
float (int)   // konwersja z int do float
bool (float)  // konwersja z float do bool
bool (int)    // konwersja z int do bool
```

W przypadku konwersji z typu `float` do typu `int` część zmiennoprzecinkowa jest odrzucana. Konwersja z typów `int` i `float` do typu `bool` odbywa się w taki sposób, że wartość 0 jest zamieniana na `false`, a każda wartość różna od 0 na `true`. Konwersja w drugą stronę z typu `bool` do typów `int` i `float` działa tak, że wartość `false` jest zamieniana na 0, a wartość `true` na 1.

Konstruktory skalarne można także używać z typami nieskalarnymi. Konwersji podlega wówczas pierwszy element typu nieskalarnego.

1.4.4. Typ void

Typ `void` jest przeznaczony do użycia dla funkcji nie zwracających żadnej wartości. GLSL nie ma domyślnego typu zwracanego przez funkcje.

1.4.5. Typ bool

Zmienne typu logicznego `bool` przyjmują jedną z dwóch wartości: `true` lub `false` i nie muszą być bezpośrednio wspierane przez procesor graficzny. Deklaracje i opcjonalne inicjalizacje zmiennych tego typu są identyczne jak w języku C++. Poniżej przykład:

```
bool success;        // deklaracja
bool done = false;   // deklaracja i inicjalizacja
```

1.4.6. Typ int

Typ `int`, to całkowity ze znakiem o minimalnej precyzji co najmniej 16 bitów (bez znaku). Implementacja może stosować liczby o większej precyzji, nie jest to jednak rozwiązanie przenośne. Podobnie jak w przypadku typu `bool` liczby typu całkowitego nie musza być bezpośrednio wspierane przez procesor graficzny.

Zmienne typu `int` mogą być inicjalizowane za pomocą literałów o podstawie dziesiętnej, ósemkowej i szesnastkowej. Metody zapisu literałów całkowitych są identyczne jak w języku C. Liczba ósemkowa rozpoczyna się cyfrą 0, a liczba szesnastkowa stałą `0x` lub `0X`. Oto przykładowe deklaracje i inicjalizacje zmiennych typu `int`:

```
int i,        // deklaracja
    j = -42;  // deklaracja i inicjalizacja
int o = 07;   // deklaracja i inicjalizacja liczby ósemkowej
int h = 0xA;  // deklaracja i inicjalizacja liczby szesnastkowej
```

1.4.7. Typ float

Typ `float` to liczby zmiennoprzecinkowe o pojedynczej precyzji. GLSL akceptuje liczby w formacie IEEE (analogicznie jak w języku C), przy czym wewnętrzna reprezentacja nie musi być zgodna z tym formatem. Od wersji 1.20 jeżyka GLSL liczba typu `float` może zawierać przyrostek `f` lub `F`. Poniżej przykłady użycia typu `float`:

```
float a,          // deklaracja
      b = 1.5 ;   // deklaracja i inicjalizacja
float c = -5.0f;  // deklaracja i inicjalizacja
```

1.4.8. Typy wektorowe

Typy wektorowe obejmują wektory dwu, trój i czterowymiarowe z elementami typu `bool`, `int` i `float`. Dostęp do elementów wektora możliwy jest na kilka sposobów. Pierwszy jest taki sam jak w przypadku typów tablicowych w językach C i C++ (elementy wektora numerowane są od

0). Alternatywna metoda jest skorzystanie z selektora . (kropka) i następujących nazw pól: $\{x, y, z, w\}$, $\{r, g, b, a\}$ lub $\{s, t, p, q\}$. Nazwy te mogą ułatwić korzystanie ze zmiennych wektorowych reprezentujących różnego rodzaju parametry programu. Pola te można dodatkowo łączyć (wyłącznie w zakresie danej grupy) uzyskując mniejsze wektory:

```
vec4 v4;
v4.rgba;  // analogiczny zapis jak v4
v4.rgb;   // to samo co vec3
v4.st;    // to samo co vec2
```

Możliwych konstruktorów typów wektorowych jest dość dużo, stąd poniżej przedstawiamy jedynie wybrane przykłady:

```
vec3 (float)        // inicjalizacja wszystkich składowych
                    // jedną wartością float
vec4 (ivec4)        // konwersja do float ze składowych int
vec2 (float,float)  // inicjalizacja z dwóch liczb float
ivec3 (int,int,int) // inicjalizacja z trzech liczb float
bvec2 (int,float)   // konwersja do bool z int i float
vec2 (vec3)         // przyjęcie dwóch pierwszych składowych vec3
vec3 (vec4)         // przyjęcie trzech pierwszych składowych vec4
vec3 (vec2,float)   // vec3.x = vec2.x, vec3.y = vec2.y,
                    // vec3.z = float
vec3 (float,vec2)   // vec3.x = float, vec3.y = vec2.x,
                    // vec3.z = vec2.y
vec4 (vec3,float)   // vec4.x = vec3.x, vec4.y = vec3.y,
                    // vec4.z = vec3.z, vec4.w = float
vec4 (float,vec3)   // vec4.x = float, vec4.y = vec3.x,
                    // vec4.z = vec3.y, vec4.w = vec3.z
vec4 (vec2,vec2)    // vec4.x = vec2.x, vec4.y = vec2.y,
                    // vec4.z = vec2.x, vec4.w = vec2.y
```

Oto kilka przykładowych deklaracji zmiennych wektorowych wraz z różnorodnymi konstruktorami:

```
vec4 pos = vec4 (1.0,2.0,3.0,4.0);
pos.xw = vec2 (5.0,6.0);  // pos = (5.0, 2.0, 3.0, 6.0)
pos.wx = vec2 (7.0,8.0);  // pos = (8.0, 2.0, 3.0, 7.0)
pos = vec4 (1.0,2.0,3.0,4.0);
vec4 swiz = pos.wzyx;     // swiz = (4.0, 3.0, 2.0, 1.0)
vec4 dup = pos.xxyy;      // dup = (1.0, 1.0, 2.0, 2.0)
vec4 color = vec4 (1.0,0.0,0.0,1.0);
vec3 rgb = vec3 (color);  // rgb = (1.0,0.0,0.0)
vec4 rgba = vec4 (1.0);   // wszystkie elementy mają wartość 1.0
```

Zwróćmy szczególną uwagę na te przykłady konstruktorów, w których różna jest kolejność składników, oraz te, gdzie nazwy pól zostały wykorzystane po lewej stronie wyrażenia, określając zapisywane składowe wektora.

1.4.9. Typy macierzowe

Wszystkie typy macierzowe w GLSL zawierają elementy typu `float`. Macierze prostokątne zostały wprowadzone w wersji 1.20 języka GLSL (w nazwie mat`m`x`n` `m` oznacza ilość kolumn, a `n` ilość wierszy). Podobnie jak w przypadku wektorów, typy macierzowe także posiadają wiele możliwych konstruktorów. Oto wybrane przykłady:

```
mat2 (float)   // inicjalizacja elementów na głównej
mat3 (float)   // przekątnej wartością float, pozostałe
mat4 (float)   // elementy przyjmują wartość 0
mat2 (vec2,vec2)           // jeden wektor = jedna kolumna
mat3 (vec3,vec3,vec3)      // jeden wektor = jedna kolumna
mat4 (vec4,vec4,vec4,vec4) // jeden wektor = jedna kolumna
mat3x2 (vec2,vec2,vec2)    // jeden wektor = jedna kolumna
mat2 (float,float,       // pierwsza kolumna
      float,float);      // druga kolumna
mat3 (float,float,float, // pierwsza kolumna
      float,float,float, // druga kolumna
      float,float,float) // trzecia kolumna
mat4 (float,float,float,float,  // pierwsza kolumna
      float,float,float,float,  // druga kolumna
      float,float,float,float,  // trzecia kolumna
      float,float,float,float)  // czwarta kolumna
mat2x3 (vec2,float,   // pierwsza kolumna
        vec2,float)   // druga kolumna
mat3x3 (mat4x4)   // inicjalizacja na podstawie lewej górnej
                  // części 3x3 z macierzy 4x4
mat2x3 (mat4x2)   // inicjalizacja na podstawie lewej górnej
                  // części 2x2 z macierzy 4x4, ostatni wiersz
                  // przyjmuje wartości 0,0
mat4x4 (mat3x3)   // inicjalizacja lewej górnej części macierzy
                  // na podstawie macierzy 3x3, prawy dolny
                  // element na głównej przekątnej przyjmuje
                  // wartość 1, a pozostałe wartość 0
```

Odwoływanie się do poszczególnych elementów macierzy możliwe jest przy zastosowaniu operatora indeksowania tablicy. Jeżeli macierz traktujemy jako tablicę jednowymiarową, to jej elementami są wektory odpowiadające poszczególnym kolumnom macierzy. Podobnie jak w językach C i C++ in-

deksy elementów macierzy numerowane są od 0. Poniżej kilka przykładowych operacji na zmiennych macierzowych:

```
mat4 m;
m[1] = vec4 (2.0);  // druga kolumna wypełniona wartościami 2.0
m[0][0] = 1.0;      // pierwszy element pierwszego wiesza
                    // otrzymał wartość 1.0
m[2][3] = 2.0;      // czwarty element trzeciej kolumny
                    // otrzymał wartość 2.0
```

Zauważmy, że dostępność wektorów i macierzy jako typów podstawowych znacznie ułatwia programowanie operacji graficznych, które standardowo korzystają z tego rodzaju danych.

1.4.10. Uchwyty tekstur

Uchwyty tekstur umożliwiają dostęp do danych tekstury jedno, dwu i trójwymiarowych oraz do tekstur sześciennych. Uchwyty stosowane są w funkcjach próbkujących tekstury, które zostaną przedstawione dalej.

1.4.11. Struktury

Struktury definiowane są analogicznie jak w języku C z wykorzystaniem słowa zarezerwowanego `struct`, przy czym deklaracja zmiennej typu strukturalnego nie wymaga użycia słowa `struct`. Wszystkie składowe struktury muszą być wcześniej zdefiniowane. GLSL nie dopuszcza struktur anonimowych ani zagnieżdżonych.

Oto definicja przykładowej struktury zawierającej dwa pola i jednocześnie deklaracja zmiennej tego typu:

```
struct light           // nazwa struktury
{
   float intensity;    // pole typu float
   vec3 position;      // pole typu vec3
} lightVar;            // zmienna typu light (opcjonalnie)

light lightVar2;       // zmienna typu light
```

Argumenty konstruktorów struktur muszą być tego samego typu i występować w takiej samej kolejności jak definicje pól w strukturze. Oczywiście możliwe są opisane wcześniej konwersje typów. Konstruktor struktury ma taką samą nazwę jak sama struktura:

```
light lightVar1 = light (3.0,vec3 (1.0,2.0,3.0));
```

Dostęp do poszczególnych elementów struktury, podobnie jak w językach C i C++ zapewnia operator `.` (kropka).

1.4.12. Tablice

Tablice definiowane są analogicznie jak w języku C zużyciem nawiasów kwadratowych `[ ]`, przy czym GLSL obsługuje wyłącznie tablice jednowymiarowe. Rozmiar tablicy musi być określony stałym wyrażeniem o wartości większej od 0, przy czym podanie rozmiaru tablicy nie jest obowiązkowe. Jeżeli tablica jest indeksowana dowolną zmienną lub jest przekazywana jako parametr funkcji, jej rozmiar musi być z góry określony. Przekroczenie zakresu tablicy jest zachowaniem niezdefiniowanym.

Oto przykładowe deklaracje tablic:

```
float frequencies [3];     // tablica liczb float
vec4 lightPosition [4];    // tablica wektorów vec4
light lights [];           // tablica bez określonego wymiaru
const int numLights = 2;   // stała określająca rozmiar tablicy
light lights [numLights];  // tablica struktur typu light
```

oraz przykładowe konstruktory tablic:

```
// konstruktory dwóch tablice stałe
const float c[3] = float[3](5.0,7.2,1.1);
const float d[3] = float[](5.0,7.2,1.1);

// konstruktory z użyciem zmiennej typu float
float g;
...
float a[5] = float[5](g,1,g,2.3,g);
float b[3];
b = float[3](g,g+1.0,g+2.0);
```

Konstruktor tablicy powinien wskazywać rozmiar zgodny z rozmiarem tablicy. W przypadku niewskazania rozmiaru tablicy w konstruktorze, tablica otrzymuje rozmiar równy ilości elementów konstruktora. Pobranie rozmiaru tablicy umożliwia funkcja `length`:

```
float e[5];
e.length ();   // zwracana wartość 5
```

która, zauważmy, korzysta z operatora `.` (kropka), czyli takiego samego jak przy dostępie do pól struktur.

1.5. Operatory i wyrażenia

Operatory języka GLSL przedstawiono w tabeli 3. Jak widzimy, część operatorów jest zarezerwowana. Operatory języka GLSL zestawiono w ko-

lejności od najwyższego (1) do najniższego (17) priorytetu. Zauważmy, że języka GLSL nie zawiera żadnych operatorów adresowych i wskaźnikowych.

priorytet	rodzaj	operator	kolejność wiązania
1	grupowanie w nawiasy	( )	-
2	selektory elementów tablic i składowych struktur, postinkrementacja, postdekrementacja, wywołanie funkcji	[] . ++ −− ( )	od lewej do prawej
3	preinkrementacja, predekrementacja, jednoargumentowe (tylda zarezerwowana)	++ −− + − ! ~	od prawej do lewej
4	multiplikatywne (moduł zarezerwowany)	∗ / %	od lewej do prawej
5	addytywne	+ −	od lewej do prawej
6	przesuwanie bitowe (zarezerwowane)	<< >>	od lewej do prawej
7	relacje	< > <= >=	od lewej do prawej
8	równość	== ! =	od lewej do prawej
9	bitowe AND (zarezerwowany)	&	od lewej do prawej
10	różnica symetryczna XOR (zarezerwowany)	∧	od lewej do prawej
11	bitowe OR (zarezerwowany)	|	od lewej do prawej
12	logiczne AND	&&	od lewej do prawej
13	logiczna różnica symetryczna XOR	∧∧	od lewej do prawej
14	logiczne OR	||	od lewej do prawej
15	selekcja	? :	od prawej do lewej
16	przypisanie, arytmetyczne przypisanie (moduł, przesuwanie, bitowe i operacje bitowe zarezerwowane)	= + = − = ∗ = / = % = <<= >>= & = ∧ = | =	od prawej do lewej
17	sekwencja	,	od lewej do prawej

Tabela 3: Pierwszeństwo operatorów języka GLSL

Operacje na strukturach, poza już wspomnianym operatorem . (kropka), mogą wykonywać następujące operatory: ==, ! = (porównanie) oraz = przypisanie. Ten sam zestaw operatorów, poza operatorem dostępu do elementu [], jest dopuszczalny także dla tablic. Oczywiście operatory przypisania i porównania wymagają do poprawnego działania zgodności typów obu operandów.

Poza nielicznymi wyjątkami w przypadku macierzy i wektorów operatory działają na wszystkich składowych zmiennej. Przykładowo:

```
vec3 v,u;
float f;
v = u + f;
```

jest równoważne z następującymi wyrażeniami:

```
v.x = u.x + f;
v.y = u.y + f;
v.z = u.z + f;
```

I następny przykład:

```
vec3 v,u,w;
w = v + u;
```

jst równoważny zapisom:

```
w.x = v.x + u.x;
w.y = v.y + u.y;
w.z = v.z + u.z;
```

Wyjątkiem są przypadki, gdy mnożymy macierz przez wektor, wektor przez macierz oraz macierz przez macierz. Wykonywane są wówczas odpowiednie operacje algebraiczne:

```
vec3 v,u;
mat3 m;
u = v * m;
```

jest równoważne:

```
u.x = dot (v,m[0]);  // m[0],m[1],m[2] - kolejne kolumny
u.y = dot (v,m[1]);  // macierzy m, dot (a,b) - iloczyn
u.z = dot (v,m[2]);  // skalarny wektorów a i b
```

I drugi przykład:

```
u = m * v;
```

jest równoważny:

```
u.x = m[0].x * v.x + m[1].x * v.y + m[2].x * v.z;
u.y = m[0].y * v.x + m[1].y * v.y + m[2].y * v.z;
u.z = m[0].z * v.x + m[1].z * v.y + m[2].z * v.z;
```

Trzeci i ostatni przykład:

```
mat3 m, n, r;
r = m * n;
```

odpowiada wyrażeniom:

```
r[0].x = m[0].x * n[0].x + m[1].x * n[0].y + m[2].x * n[0].z;
r[1].x = m[0].x * n[1].x + m[1].x * n[1].y + m[2].x * n[1].z;
r[2].x = m[0].x * n[2].x + m[1].x * n[2].y + m[2].x * n[2].z;
r[0].y = m[0].y * n[0].x + m[1].y * n[0].y + m[2].y * n[0].z;
r[1].y = m[0].y * n[1].x + m[1].y * n[1].y + m[2].y * n[1].z;
r[2].y = m[0].y * n[2].x + m[1].y * n[2].y + m[2].y * n[2].z;
r[0].z = m[0].z * n[0].x + m[1].z * n[0].y + m[2].z * n[0].z;
r[1].z = m[0].z * n[1].x + m[1].z * n[1].y + m[2].z * n[1].z;
r[2].z = m[0].z * n[2].x + m[1].z * n[2].y + m[2].z * n[2].z;
```

Oczywiście warunkiem wykonalności operacji mnożenia macierzy przez wektor, wektora przez macierz oraz macierzy przez macierz jest spełnienie odpowiednich warunków algebraicznych związanych z rozmiarami mnożonych wektorów i macierzy.

1.6. Kwalifikatory typów

Zmienne można deklarować z opcjonalnymi kwalifikatorami. Dostępne kwalifikatory omawiamy poniżej. Domyślnie zmienne nie otrzymują żadnego kwalifikatora. Niekwalifikowane zmienne globalne i lokalne umożliwiają jedynie odczyt i zapis przydzielonego im obszarowi pamięci.

Kwalifikatory można podzielić na określające sposób dostępu do pamięci (`const`, `attribute`, `uniform` i `varying`) oraz określające rodzaj dostępu do parametrów funkcji (`in`, `out` i `inout`). Z niektórymi kwalifikatorami może występować dodatkowy kwalifikator `invariant` wspierający optymalizację obsługi jednakowych zmiennych w niezależnych programach cieniowania. W przyszłości możliwe jest dodanie kwalifikatora `precision` i innych kwalifikatorów związanych z dokładnością zmiennych.

Kolejność użycia kwalifikatorów jest następująca: `invariant` poprzedza kwalifikator dostępu do pamięci; kwalifikator dostępu do pamięci poprzedza kwalifikator dostępu do parametrów funkcji.

1.6.1. const

Kwalifikator `const` określa wartość stałą inicjalizowaną podczas deklaracji i może być użyty z każdym podstawowym typem danych. Zmienne z kwalifikatorem `const` przeznaczone są tylko do odczytu. Ponadto kwalifikator `const` używany jest w deklaracjach tych argumentów funkcji, które nie ulegają zmianie w trakcie jej działania.

Wyrażeniem stałym może być:
— liczba całkowita,
— globalna lub lokalna zmienna typu całkowitego z kwalifikatorem `const`, za wyjątkiem parametrów funkcji deklarowanych jako `const`,
— wyrażenie złożone z operatorów i argumentów, które wszystkie są wyrażeniami stałymi, włączając w to pobranie elementu lub długości tablicy z kwalifikatorem `const`, pobranie pola struktury z kwalifikatorem `const` oraz pobranie elementu wektora z kwalifikatorem `const`,
— konstruktory, których wszystkie argumenty są wyrażeniami stałymi,
— wbudowane funkcje, wywołane wyłącznie z stałymi wyrażeniami, z wyjątkiem funkcji próbkujących tekstury, funkcji stochastycznych oraz funkcji `ftransform`.

1.6.2. attribute

Kwalifikator `attribute` dostępny jest wyłącznie w programach cieniowania wierzchołków i określa zmienne - atrybuty przekazywane przez OpenGL dla każdego wierzchołka. Zmienne z kwalifikatorem `attribute` przeznaczone są tylko do odczytu i muszą być zmiennymi globalnymi. GLSL ogranicza typ zmiennych z kwalifikatorem `attribute` do liczb zmiennoprzecinkowych oraz wektorów i macierzy z liczbami zmiennoprzecinkowymi. Nie są dopuszczalne tablice lub struktury z tym kwalifikatorem.

Standardowe atrybuty wierzchołków, np. podstawowy i drugorzędny kolor, współrzędne, wektor normalny, dostępne są poprzez wbudowane atrybuty. Umożliwia to łatwą integrację pomiędzy klasyczną częścią potoku OpenGL a programem cieniowania wierzchołków. Wbudowane atrybuty programów cieniowania wierzchołków opisujemy dalej.

1.6.3. uniform

Kwalifikator `uniform` określa globalne zmienne jednorodne, których wartość dostępna jest zarówno dla programu cieniowana wierzchołków jak i programu cieniowania fragmentów. Zmienne jednorodne są stałe w obrębie programów cieniowania a ich wartość definiowana jest przez wywołanie odpowiednich funkcji biblioteki OpenGL.

Kwalifikator `uniform` może być użyty z dowolnym typem danych w tym także z typami danych zdefiniowanymi przez użytkownika.

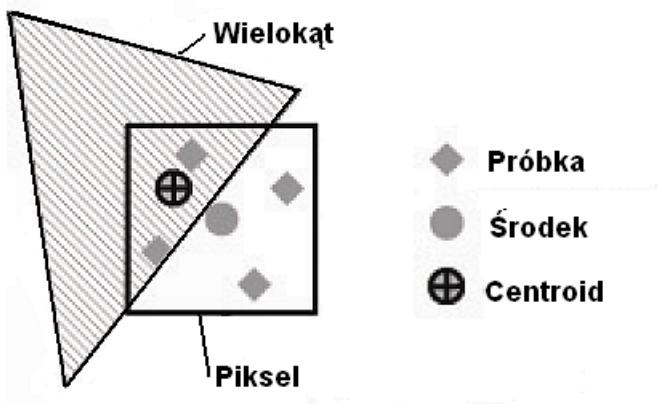
1.6.4. varying

Zmienne z kwalifikatorem `varying` służą do komunikacji pomiędzy programami cieniowania wierzchołków a programami cieniowania fragmentów. Program cieniowania wierzchołków oblicza dla każdego wierzchołka wartość udostępnianej zmiennej, a program cieniowania fragmentów otrzymuje te wartości interpolowane dla każdego fragmentu przy użyciu korekty perspektywy. Zmienna z kwalifikatorem `varying` jest dla programu cieniowania fragmentów zmienną tylko do odczytu.

Kwalifikator `varying` może być używany wyłącznie z liczbami zmiennoprzecinkowymi oraz wektorami i macierzami z liczbami zmiennoprzecinkowymi. Dopuszczalne są także tablice zmiennych tego typu. Nie są natomiast dopuszczalne struktury z tym kwalifikatorem.

W przypadku, gdy żaden program cieniowania wierzchołków nie jest aktywny nieprogramowalna część potoku OpenGL obliczy wartości wbudowanych zmiennych udostępnianych, tak aby mogły być użyte w programach cieniowania fragmentów. Analogicznie, jeżeli nie jest aktywny żaden program cieniowania fragmentów, program cieniowania wierzchołków jest odpowiedzialny za obliczenie wartości zmiennych udostępnianych, które są dalej użyte przez nieprogramowalny potok przekształceń fragmentów.

Wspomniana wyżej interpolacja wartości zmiennych z kwalifikatorem `varying` przy wyłączonym wielopróbkowaniu obliczana jest dla środka piksela. Jednak w przypadku, gdy włączone jest wielopróbkowanie, interpolowana wartość może pochodzić z jednej z próbek, ze środka piksela bądź centroidu (patrz rysunek 3). Może to być przyczyną różnego rodzaju przekłamań (arfetaktów) w renderingu.



Rysunek 3. Interpolacja piksela w wielopróbkowaniu

Problem ten rozwiązuje dodany w wersji 1.20 języka GLSL kwalifikator `centroid`, występujący łącznie z `varying`, który w przypadku włączonego wielopróbkowania wymusza interpolację przy użyciu centroidu. Jeżeli wielopróbkowanie jest wyłączone kwalifikator `centroid` jest ignorowany.

1.6.5. in

Kwalifikator `in` określa parametry wejściowe funkcji. Wszelkie zmiany tego parametru wewnątrz funkcji nie mają wpływu na jego wartość poza funkcją. Jest to domyślny kwalifikator argumentów funkcji.

1.6.6. out

Kwalifikator `out` określa parametr wyjściowy funkcji. Parametry z tym kwalifikatorem nie wymagają przekazywania do funkcji żadnej konkretnej wartości.

1.6.7. inout

Kwalifikator `inout` określa, że parametr funkcji jest parametrem zarówno wejściowym jak i wyjściowym. Wszelkie zmiany wartości tego parametru będą miały wpływ na jego wartość poza funkcją.

1.6.8. invariant

Ostatni opisywany kwalifikator to dodany w wersji 1.20 GLSL kwalifikator `invariant`, którego zadaniem jest ułatwienie optymalizacji obsługi jednakowych inwariantnych (niezmienniczych) zmiennych w niezależnych programach cieniowania.

Kwalifikator `invariant` mogą otrzymać zarówno zmienne wbudowane jak i zmienne zdefiniowane przez użytkownika. Kwalifikator można także użyć w innym miejscu niż deklaracja zmiennej:

```
// dodanie kwalifikatora invariant do wbudowanej
// zmiennej gl_Position
invariant gl_Position;
varying vec3 Color;
// dodanie kwalifikatora invariant do zadeklarowanej
// przez użytkownika zmiennej Color
invariant Color;
```

lub jednocześnie z jej deklaracją:

```
invariant varying vec3 Color;
```

Z uwagi na swoją specyfikę kwalifikator `invariant` mogą otrzymać jedynie wyjściowe z programu cieniowania wierzchołków. Dotyczy do w praktyce zdefiniowanych przez użytkowniak zmiennych z kwalifikatorem `varying`

(zmienne udostępniane) oraz specjalnych zmiennych programów cieniowania wierzchołków `gl_Position` i `gl_PointSize`. Kwalifikator `invariant` musi zostać wyspecyfikowany przed kwalifikatorem `varying`.

W przypadku niezmienniczych zmiennych udostępnianych, które deklarowane są zarówno w programie cieniowania wierzchołków jak i w programie cieniowania fragmentów, obie deklaracje powinny zostać poprzedzone kwalifikatorem `invariant`, lub kwalifikator ten musi zostać dodany przed pierwszym użyciem zmiennej. Zagwarantowanie przyjęcie obsługi zmiennej inwariantnej w obu programach cieniowania wymaga ponadto spełnienia szeregu dodatkowych warunków (wszystkie wymienia specyfikacja języka GLSL). Wymienimy tylko jeden, stanowiący, że cały przepływ danych związany ze zmiennymi inwariantnymi musi zawierać się z jednej jednostce kompilacji (tzw. obiekcie programów cieniowania, które poznamy w następnym odcinku kursu).

Domyślnie wszystkie zmienne wyjściowe są zdefiniowane bez kwalifikatora `invariant`. Można jednak przed deklaracją zmiennych użyć następującej instrukcji preprocesora:

```
#pragma STDGL invariant(all)
```

która wymusza domyślne zastosowanie kwalifikatora `invariant` dla wszystkich zmiennych wyjściowych. Specyfika tej instrukcji wymaga umieszczenia wyłącznie w programie cieniowania wierzchołków.

Generalnie kwalifikator `invariant` gwarantuje swobodę optymalizacji dla kompilatora GLSL, ale wydajność może spaść przy jego stosowani. Stąd specyfikacja GLSL zaleca użycie powyższej instrukcji preprocesora do testów wydajności i na podstawie tego podejmowanie indywidualnych decyzji dotyczących stosowania tego kwalifikatora.

1.7. Instrukcje i struktura programu

Podstawowymi blokami w języku GLSL są:
— instrukcje i deklaracje,
— definicje funkcji,
— selekcje (`if` / `else`),
— pętle (`for`, `while`, `do` / `while`),
— skoki (`discard`, `return`, `break`, `continue`).

1.7.1. Definiowanie funkcji

Program cieniowania w języku GLSL jest zasadniczo sekwencją definicji zmiennych globalnych i funkcji. Deklaracja funkcji (prototyp) wygląda następująco:

```
typZwracany nazwaFunkcji (typ0 arg0, typ1 arg2, ..., typN argN);
```

natomiast definicja funkcji przyjmuje postać:

```
typZwracany nazwaFunkcji (typ0 arg0, typ1 arg2, ..., typN argN)
{
   // wykonanie obliczeń
   return wartośćZwracana;
}
```

Każda funkcja musi mieć określony typ zwracany. Także każdy z argumentów funkcji musi mieć określony typ oraz opcjonalny kwalifikator: `in`, `out`, `inout` i/lub `const`. Jago argumentu funkcji można użyć tablicy, jednak tablica nie może być typem zwracanym przez funkcję. Wszystkie funkcje, przed pierwszym użyciem muszą być zadeklarowane lub zdefiniowane. Jeżeli funkcja nie zwracają żadnych wartości musi być zadeklarowana jako `void`.

Język GLSL dopuszcza przeciążanie funkcji, czyli wykorzystywanie jednej nazwy dla funkcji różniących się listą argumentów. Oczywiście poza zgodnością nazwy funkcji i typów jej argumentów musi występować zgodność typu zwracanego oraz zgodność kwalifikatorów argumentów. Przeciążanie funkcji jest szeroko wykorzystywane przez funkcje wbudowane. Przykładem może być funkcja `dot` obliczająca iloczyn skalarny:

```
float dot (float x, float y);
float dot (vec2 x, vec2 y);
float dot (vec3 x, vec3 y);
float dot (vec4 x, vec4 y);
```

Punktem wejściowym każdego programu cieniowania jest funkcja `main`. Funkcja ta nie posiada argumentów i nie zwraca żadnej wartości:

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

1.7.2. Wywoływanie funkcji

Przy wywołaniu funkcji argumenty wejściowe kopiowane są w momencie wywołania, natomiast argumenty wyjściowe kopiowane są przed zakończeniem działania funkcji. Do określenia, który z parametrów jest wejściowy, wyjściowy lub wejściowy i wyjściowy, służą opisywane wcześniej kwalifikatory: `in`, `out` oraz `inout`. Przy braku kwalifikator przyjmowany jest `in`.

GLSL dopuszcza możliwość zmiany w ciele funkcji wartości argumentu z kwalifikatorem `in`, gdyż modyfikacji podlega jedynie jego lokalna kopia. Wyjątek stanowią argumenty z dodatkowym kwalifikatorem `const`, który, co oczywiste, nie może być użyty przy argumentach z kwalifikatorem `out` oraz `inout`.

Specyfikacja języka GLSL nie dopuszcza możliwości rekurencyjnego wywoływania funkcji.

1.7.3. Instrukcje sterujące

Instrukcje sterujące `if` oraz `if/else` mają taką samą konstrukcję jak w językach C i C++. Wyrażeniem warunkowym w nim występującym może być dowolne wyrażenie typu logicznego `bool`. Wyjątkiem jest wyłączenie typów wektorowych jako wyrażenia w instrukcji `if`. Instrukcje sterujące mogą być zagnieżdżone.

1.7.4. Pętle

Pętle `for`, `while` oraz `do/while` mają w języku GLSL identyczną konstrukcję jak w językach C i C++. Wyrażenie warunkowe w nich występujące musi być wyrażeniem typu logicznego `bool`. Pętle mogą być zagnieżdżone.

GLSL formalnie dopuszcza pętle nieskończone, jednak konsekwencje związane z ich uruchomieniem zależne są od implementacji.

1.7.5. Skoki

GLSL ddefiniuje kilka rodzajów instrukcji skoków: `continue`, `break`, `return` (w tym także ze możliwością zwrócenia wartości) oraz `discard`. Dwie pierwsze instrukcje mają zastosowanie wyłącznie w pętlach, a ich działanie jest takie samo jak w językach C i C++. Instrukcja `discard` dostępna jest wyłącznie w programach cieniowania fragmentów, a jej wywołanie spowoduje odrzucenia aktualnie przetwarzanego fragmentu i jednocześnie brak aktualizacji zawartości bufora ramki.

Instrukcja `return` powoduje natychmiastowe opuszczenie funkcji. Wywołanie `return` w programie cieniowania fragmentów jest równoważne wykonaniu instrukcji `discard`.

Zauważmy, że pomowo zarezerwowania słowa `goto`, język GLSL nie zawiera obsługi tej instrukcji.

1.8. Wbudowane zmienne

Programy cieniowania komunikują się z nieprogramowalną częścią potoku OpenGL za pomocą wbudowanych zmiennych. Z uwagi na odmienne charaktery wykonywanych zadań programy cieniowania wierzchołków i programy cieniowania fragmentów mają częściowo odrębne zestawy wbudowanych zmiennych.

1.8.1. Specjalne zmienne programów cieniowania wierzchołków

Programy cieniowania wierzchołków mają dostępne trzy globalne zmienne specjalne:

```
vec4  gl_Position;
float gl_PointSize;
vec4  gl_ClipVertex;
```

Zmienna `gl_Position` określa pozycję wierzchołka prymitywu po przekształceniach. Jest to zmienna, której wartość musi obliczyć i zapisać każdy program cieniowania wierzchołków. Jeżeli przekształcenie pozycji wierzchołka ma odpowiadać przekształceniom klasycznego potoku OpenGL obliczenie wartości zmiennej `gl_Position` sprowadza się do przemnożenia macierzy rzutowania i macierzy modelowania przez współrzędne wierzchołka:

```
void main ()
{
  gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix *
                gl_Vertex;
}
```

Ponieważ GLSL posiada wbudowaną specjalną zmienną `gl_ModelViewProjectionMatrix` zawierającą wynik mnożenia macierzy rzutowania i macierzy modelowania, wartość zmiennej `gl_Position` można także obliczyć w następujący sposób:

```
void main ()
{
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Jednak najlepszym (i prawdopodobnie najszybszym sposobem) obliczenia „klasycznej" wartości zmiennej `gl_Position` jest użycie funkcji `ftransform`, która wyknuje wszystkie niezbędne operacje:

```
void main ()
{
  gl_Position = ftransform ();
}
```

Przedstawione powyżej trzy programy są najprostszymi możliwymi programami cieniowania wierzchołków, oczywiście przy założeniu, że program ten wykonuje przekształcenia wierzchołków zgodnie z klasycznym potokiem OpenGL. Warto jednak zwrócić uwagę, że jedynie użycie funkcji `ftransform` daje pełną gwarancje wykonania przekształceń wierzchołków w dokładnie taki sam sposób jak w potoku nieprogramowalnym. Z uwagi na stosowaną optymalizację oraz skończoną precyzję liczb zmiennoprzecinkowych dwa pierwsze programy mogą dać nieco odmienne rezultaty.

Zmienna `gl_PointSize` określa (w pikselach) rozmiar rasteryzowanych punktów, a zmienna `gl_ClipVertex` zawiera współrzędne używane z płaszczyznami obcinania zdefiniowanymi przez użytkownika. Jeżeli wartości tych zmiennych nie zostaną określone ich wartość pozostaje niezdefiniowana.

1.8.2. Specjalne zmienne programów cieniowania fragmentów

Programy cieniowania dostępne mają dostępnych pięć globalnych zmiennych specjalnych:

```
vec4  gl_FragCoord;
bool  gl_FrontFacing;
vec4  gl_FragColor;
vec4  gl_FragData [gl_MaxDrawBuffers];
float gl_FragDepth;
```

Dwie pierwsze zmienne są zmiennymi tylko do odczytu. Zmienna `gl_FragCoord` zawiera współrzędne $\left(x, y, z, \frac{1}{w}\right)$ położenia fragmentu w oknie renderingu. Wartość głębi (współrzędna z) program cieniowania fragmentów może wykorzystać do własnych obliczeń tej wartości. Druga zmienna do odczytu `gl_FrontFacing` zawiera informację, czy dany fragment należy do prymitywu zwróconego przodem do obserwatora (wartość `true`) lub tyłem (wartość `false`).

Program cieniowania fragmentów musi w celu komunikacji z pozostałą częścią potoku OpenGL zapisać wartości składowych koloru fragmentu (zmienna `gl_FragColor`) oraz wartość głębi fragmentu (zmienna `gl_FragDepth`). Zapis wartości głębi fragmentu jest opcjonalny, ale jeżeli jest wykonywany, to obowiązkowo dla każdego fragmentu. Jeżeli program cieniowania fragmentów nie zapisuje wartości głębi, jest ona wyznaczana automatycznie przez OpenGL.

Zapis koloru fragmentu nie jest opcjonalny, stąd chyba najprostszym przykładem programu cieniowania fragmentów jest program kolorujący wierzchołki prymitywów na określony kolor:

```
void main()
{
  gl_FragColor = vec4 (1.0,0.0,0.0,1.0);
}
```

Jeżeli program korzysta z techniki wielokrotnych docelowych buforów koloru (rozszerzenie `ARB_draw_buffers`) zapis koloru fragmentu dokonywany jest do odpowiednich elementów tablicy `gl_FragData`. Program cieniowania fragmentów nie może jednocześnie zapisywać wartości zmiennej `gl_FragColor` i elementów tablicy `gl_FragData`. To samo ograniczenie dotyczy również

wszystkich programów cieniowania fragmentów, które są razem skonsolidowane.

Program cieniowania fragmentów może także odrzucić wybrany fragment z dalszego potoku renderingu. Służy do tego instrukcja `discard`. Wartości wszystkich zmiennych wyjściowych pozostają wówczas niezdefiniowane.

1.8.3. Wbudowane atrybuty programów cieniowania wierzchołków

Programy cieniowania wierzchołków mają dostęp do wszystkich atrybutów wierzchołków definiowanych w nieprogramowalnej części potoku OpenGL. Są to kolejno: składowe podstawowego koloru wierzchołka, składowe drugorzędnego koloru wierzchołka, współrzędne wektora normalnego, współrzędne wierzchołka, współrzędne tekstur dla kolejnych jednostek teksturujących oraz współrzędne mgły. Nazwy i typy wymienionych atrybutów przedstawione są poniżej:

```
attribute vec4  gl_Color;
attribute vec4  gl_SecondaryColor;
attribute vec3  gl_Normal;
attribute vec4  gl_Vertex;
attribute vec4  gl_MultiTexCoord0;
attribute vec4  gl_MultiTexCoord1;
attribute vec4  gl_MultiTexCoord2;
attribute vec4  gl_MultiTexCoord3;
attribute vec4  gl_MultiTexCoord4;
attribute vec4  gl_MultiTexCoord5;
attribute vec4  gl_MultiTexCoord6;
attribute vec4  gl_MultiTexCoord7;
attribute float gl_FogCoord;
```

1.8.4. Wbudowane stałe

Przedstawione poniżej wbudowane stałe dostępne są zarówno dla programów cieniowania wierzchołków jak i programów cieniowania fragmentów. Podane wartości stałych odpowiadają minimalnym wymaganiom określonym przez specyfikację biblioteki OpenGL. W komentarzach oprócz opisu stałej dodano także numer wersji OpenGL, w której wprowadzono jej klasyczną (objętą nieprogramowalnym potokiem) wersję. Ponadto komentarze zawierają nazwy zmiennych stanu odpowiadających opisywanym stałym.

```
// maksymalna ilość źródeł światła, OpenGL 1.0
// (GL_MAX_LIGHTS)
const int gl_MaxLights = 8;
```

```
// maksymalna ilość płaszczyzn obcinania, OpenGL 1.0
// (GL_MAX_CLIP_PLANES)
const int gl_MaxClipPlanes = 6;

// maksymalna ilość statycznych (nieprogramowalnych)
// jednostek teksturujących, OpenGL 1.3
// (GL_MAX_TEXTURE_UNITS)
const int gl_MaxTextureUnits = 2;

// maksymalna ilość zbiorów współrzędnych tekstur, OpenGL 2.0
// (GL_MAX_TEXTURE_COORDS)
const int gl_MaxTextureCoords = 2;

// maksymalna ilość aktywnych atrybutów wierzchołków,
// OpenGL 2.0 (GL_MAX_VERTEX_ATTRIBS)
const int gl_MaxVertexAttribs = 16;

// maksymalna ilość zmiennych jednorodnych dla programów
// cieniowania wierzchołków, OpenGL 2.0
// (GL_MAX_VERTEX_UNIFORM_COMPONENTS)
const int gl_MaxVertexUniformComponents = 512;

// maksymalna ilość składowych interpolowanych zmiennych
// przekazywanych z programu cieniowania wierzchołków do
// programu cieniowania fragmentów (zmienne z atrybutem
// varying), OpenGL 2.0 (GL_MAX_VARYING_FLOATS)
const int gl_MaxVaryingFloats = 32;

// maksymalna ilość programowalnych jednostek teksturujących
// dostępnych dla programów cieniowania wierzchołków, OpenGL 2.0
// (GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS)
const int gl_MaxVertexTextureImageUnits = 0;

// maksymalna ilość programowalnych jednostek teksturujących,
// OpenGL 2.0 (GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS)
const int gl_MaxCombinedTextureImageUnits = 2;

// maksymalna ilość programowalnych jednostek teksturujących
// dostępnych dla programów cieniowania fragmentów, OpenGL 2.0
// (GL_MAX_TEXTURE_IMAGE_UNITS)
const int gl_MaxTextureImageUnits = 2;
```

```
// maksymalna ilość zmiennych jednorodnych dla programów
// cieniowania fragmentów, OpenGL 2.0
// (GL_MAX_FRAGMENT_UNIFORM_COMPONENTS)
const int gl_MaxFragmentUniformComponents = 64;

// maksymalna ilość aktywnych docelowych buforów koloru,
// OpenGL 2.0 (GL_MAX_DRAW_BUFFERS)
const int gl_MaxDrawBuffers = 1;
```

1.8.5. Wbudowane zmienne jednorodne

Wbudowane zmienne jednorodne opisują bieżącą konfigurację potoku OpenGL i są dostępne zarówno dla programów cieniowania wierzchołków jak i programów cieniowania fragmentów. Można je podzielić na wiele grup. Opis poszczególnych zmiennych zawarto w poniższych komentarzach.

```
// macierz modelowania
uniform mat4 gl_ModelViewMatrix;

// macierz rzutowania
uniform mat4 gl_ProjectionMatrix;

// iloczyn macierzy modelowania i macierzy rzutowania
uniform mat4 gl_ModelViewProjectionMatrix;

// tablica macierzy tekstur
uniform mat4 gl_TextureMatrix [gl_MaxTextureCoords];

// transponowana odwrócona macierz 3x3 złożona z lewych
// gównych elementów macierzy modelowania
uniform mat3 gl_NormalMatrix;

// odwrotność macierzy modelowania
uniform mat4 gl_ModelViewMatrixInverse;

// odwrotność macierzy rzutowania
uniform mat4 gl_ProjectionMatrixInverse;

// odwrotność iloczynu macierzy modelowania i macierzy rzutowania
uniform mat4 gl_ModelViewProjectionMatrixInverse;

// tablica odwrotności macierzy tekstur
uniform mat4 gl_TextureMatrixInverse [gl_MaxTextureCoords];
```

```
// transponowana macierz modelowania
uniform mat4 gl_ModelViewMatrixTranspose;

// transponowana macierz rzutowania
uniform mat4 gl_ProjectionMatrixTranspose;

// transponowany iloczyn macierzy modelowania
// i macierzy rzutowania
uniform mat4 gl_ModelViewProjectionMatrixTranspose;

// tablica transponowanych macierzy tekstur
uniform mat4 gl_TextureMatrixTranspose [gl_MaxTextureCoords];

// odwrócona i transponowana macierz modelowania
uniform mat4 gl_ModelViewMatrixInverseTranspose;

// odwrócona i transponowana macierz rzutowania
uniform mat4 gl_ProjectionMatrixInverseTranspose;

// odwrócony i transponowany iloczyn macierzy modelowania
// i macierzy rzutowania
uniform mat4 gl_ModelViewProjectionMatrixInverseTranspose;

// tablica odwróconych i transponowanych macierzy tekstur
uniform mat4 gl_TextureMatrixInverseTranspose [gl_MaxTextureCoords];

// znacznik skalowania wektorów normalnych
// (GL_RESCALE_NORMAL)
uniform float gl_NormalScale;

// struktura i zmienna zawierająca współrzędne przedniej i tylnej
// płaszczyzny obcinania (GL_DEPTH_RANGE)
struct gl_DepthRangeParameters
{
  float near;   // przednia płaszczyzna obcinania
  float far;    // tylna płaszczyzna obcinania
  float diff;   // wynik działania far - near
};
uniform gl_DepthRangeParameters gl_DepthRange;
```

```
// współrzędne płaszczyzn obcinania zdefiniowanych
// przez użytkownika (GL_CLIP_PLANEi, GL_MAX_CLIP_PLANES)
uniform vec4 gl_ClipPlane [gl_MaxClipPlanes];

// struktura i zmienna określająca parametry punktów
// (GL_POINT_SIZE, GL_POINT_SIZE_MIN, GL_POINT_SIZE_MAX,
// GL_POINT_FADE_THRESHOLD_SIZE, GL_POINT_DISTANCE_ATTENUATION)
struct gl_PointParameters
{
  float size;                          // rozmiar punktu
  float sizeMin;                       // maksymalny rozmiar
                                       // punktu
  float sizeMax;                       // minimalny rozmiar
                                       // punktu
  float fadeThresholdSize;             // wartość progowa
                                       // używana przy włączonym
                                       // wielopróbkowaniu do
                                       // zmiany wielkości punktu
                                       // oraz zmiany składowej
                                       // alfa koloru punktu
  float distanceConstantAttenuation;   // współczynniki a, b i c
  float distanceLinearAttenuation;     // równania określajacego
  float distanceQuadraticAttenuation; // rozmiar punktu po
                                       // przekształceniach
                                       // geometrycznych
};
uniform gl_PointParameters gl_Point;

// struktura i zmienne opisujące właściwości materiału
// przedniej i tylnej strony wielokąta (GL_EMISSION,
// GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_SHININESS)
struct gl_MaterialParameters
{
  vec4 emission;   // światło emitowane przez obiekt
  vec4 ambient;    // stopień odbicia światła otaczającego
  vec4 diffuse;    // stopień rozproszenia światła rozproszonego
  vec4 specular;   // stopień odbicia światła odbitego
  float shininess; // wykładnik odbłysku swiatła
};
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```

```
// struktura i tablica właściwości źródeł światła
// (GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION,
// GL_SPOT_DIRECTION, GL_SPOT_EXPONENT, GL_SPOT_CUTOFF,
// GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION,
// GL_QUADRATIC_ATTENUATION)
struct gl_LightSourceParameters
{
  vec4 ambient;        // światło otaczające
  vec4 diffuse;        // światło rozproszone
  vec4 specular;       // światło odbite
  vec4 position;       // położenie/kierunek źródła światła
  vec4 halfVector;     // wartość pochodna
  vec3 spotDirection;  // kierunek reflektora
  float spotExponent;  // wykładnik tłumienia kątowego
                       // reflektora
  float spotCutoff;    // kąt odcięcia reflektora
  float spotCosCutoff; // cosinus z wartości spotCutoff
  float constantAttenuation;  // stały współczynnik
                              // tłumienia światła
  float linearAttenuation;    // liniowy współczynnik
                              // tłumienia światła
  float quadraticAttenuation; // kwadratowy współczynnik
                              // tłumienia światła
};
uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];

// struktura i zmienna z parametrem modelu oświetlenia
// (GL_LIGHT MODEL AMBIENT)
struct gl_LightModelParameters
{
  vec4 ambient; // globalne światło otaczające
};
uniform gl_LightModelParameters gl_LightModel;

// struktury i zmienne z wartościami pochodnymi po właściwościach
// materiałów, źródeł światła i parametrów modelu oświetlenia
struct gl_LightModelProducts
{
  vec4 sceneColor; // suma światła emitowanego przez obiekt
                   // oraz iloczynu stopnia odbicia światła
                   // otaczającego i globalnego światła otaczającego
};
```

```
uniform gl_LightModelProducts gl_FrontLightModelProduct;
uniform gl_LightModelProducts gl_BackLightModelProduct;

struct gl_LightProducts
{
  vec4 ambient;   // iloczyn stopnia odbicia światła
                  // otaczającego i światła otaczającego
  vec4 diffuse;   // iloczyn stopnia rozproszenia światła
                  // rozproszonego i światła rozproszonego
  vec4 specular;  // iloczyn stopnia odbicia światła
                  // odbitego i światła odbitego
};
uniform gl_LightProducts gl_FrontLightProduct [gl_MaxLights];
uniform gl_LightProducts gl_BackLightProduct [gl_MaxLights];

// kolor środowiska tekstur (TEXTURE_ENV_COLOR)
uniform vec4 gl_TextureEnvColor [gl_MaxTextureUnits];

// współrzędne równania jednorodnego płaszczyzny używanego
// przy odwzorowaniu liniowym względem kamery (GL_EYE_PLANE)
uniform vec4 gl_EyePlaneS [gl_MaxTextureCoords];
uniform vec4 gl_EyePlaneT [gl_MaxTextureCoords];
uniform vec4 gl_EyePlaneR [gl_MaxTextureCoords];
uniform vec4 gl_EyePlaneQ [gl_MaxTextureCoords];

// współrzędne równania jednorodnego płaszczyzny używanego
// przy odwzorowaniu liniowym względem obiektu (GL_OBJECT_PLANE)
uniform vec4 gl_ObjectPlaneS [gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneT [gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneR [gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneQ [gl_MaxTextureCoords];

// struktura i zmienna opisująca parametry mgły
struct gl_FogParameters
{
  vec4 color;      // składowe koloru mgły
  float density;   // gęstość mgły
  float start;     // początek oddziaływania mgły
  float end;       // koniec oddziaływania mgły
  float scale;     // wartość wyrażenia: 1.0 / (end - start)
};
uniform gl_FogParameters gl_Fog;
```

1.8.6. Wbudowane zmienne udostępniane

W przeciwieństwie do zmiennych udostępnianych definiowanych przez użytkownika, zmienne wbudowane nie mają pomiędzy programami cieniowania wierzchołków i programami cieniowania fragmentów korelacji „jeden do jednego". Stąd każdy z rodzajów programów cieniowania ma swój odrębny zestaw zmiennych udostępnianych, które posiadają specyficzne zależności między sobą.

Programy cieniowania wierzchołków mają możliwość zapisu następujących zmiennych udostępnianych:

```
// podstawowy kolor przedniej strony wielokąta
varying vec4 gl_FrontColor;

// podstawowy kolor tylnej strony wielokąta
varying vec4 gl_BackColor;

// drugorzędny kolor przedniej strony wielokąta
varying vec4 gl_FrontSecondaryColor;

// drugorzędny kolor tylnej strony wielokąta
varying vec4 gl_BackSecondaryColor;

// współrzędne tekstur, rozmiar tablicy zazwyczaj
// będzie równy wartości zmiennej gl_MaxTextureCoords
varying vec4 gl_TexCoord [];

// współrzędne mgły
varying float gl_FogFragCoord;
```

Natomiast programy cieniowania fragmentów mogą odczytywać następujące zmienne udostępniane:

```
// podstawowy kolor fragmentu
varying vec4 gl_Color;

// drugorzędny kolor fragmentu
varying vec4 gl_SecondaryColor;

// współrzędne tekstur, rozmiar tablicy zazwyczaj
// będzie równy wartości zmiennej gl_MaxTextureCoords
varying vec4 gl_TexCoord [];
```

```
// współrzędne mgły
varying float gl_FogFragCoord;

// współrzędne punktu sprajtów (duszków) punktowych
varying vec2 gl_PointCoord;
```

Zauważmy, że nazwy zmiennych `gl_Color` i `gl_SecondaryColor` są identyczne jak nazwy wbudowanych atrybutów programów cieniowania wierzchołków. Na szczęście konflikt nazw nie występuje bowiem powyższe atrybuty widoczne są wyłącznie w programach cieniowania wierzchołków, a zmienne udostępniane dostępne są tylko w programach cieniowania fragmentów.

Wartości zmiennych `gl_Color` oraz `gl_SecondaryColor` są automatycznie obliczane ze zmiennych `gl_FrontColor`, `gl_BackColor`, `gl_FrontSecondaryColor` i `gl_BackSecondaryColor` w oparciu o to, która strona wielokąta jest widoczna.

Zmienna `gl_PointCoord` jest niezdefiniowan jeżeli bieżący prymityw nie jest punktem, lub sprajty punktowe nie są aktywne.

1.9. Wbudowane funkcje

GLSL zawiera szereg funkcji działających na zmiennych skalarnych oraz wektorowych. Większość z nich można wykorzystać zarówno w programach cieniowania wierzchołków jak i w programach cieniowania fragmentów. Wbudowane w GLSL funkcje można podzielić na trzy podstawowe grupy:

— odzwierciedlające pewne funkcje sprzętu, które nie mogą być programowo emulowane przez program cieniowania,
— realizujące podstawowe operacje arytmetyczne (np. minimum), które mogą być wspierane sprzętowo,
— operacje, które mogą być częściowo wspierane przez sprzęt (np. funkcje trygonometryczne).

Wiele funkcji wbudowanych w języku GLSL jest wzorowana na funkcjach bibliotecznych języka C. Różnica polega zazwyczaj na możliwości operowania także na zmiennych wektorowych. W programach cieniowania należy preferować stosowanie funkcji wbudowanych nad odpowiadające im własne funkcje, bowiem z założenia są one optymalne i zawsze, gdy jest taka możliwość, wykonywane sprzętowo.

W programie cieniowania można zastąpić wbudowaną funkcję deklarując i definiując funkcję o takiej samej nazwie i liście parametrów jak funkcja wbudowana. W znajdujących się poniżej opisach typ `genType` oznacza dowolny skalarny lub wektorowy typ zmiennoprzecinkowy: `float`, `vec2`, `vec3` i `vec4`. Analogicznie typ `mat` oznacza dowolny typ macierzowy, `vec` wektor z liczbami zmiennoprzecinkowymi a `bvec` wektor z liczbami typu `bool`.

1.9.1. Funkcje trygonometryczne

W tej grupie funkcji przedstawionych w tabeli 4, poza funkcjami trygonometrycznymi, znajdują się także funkcje kątowe i cylkometryczne (odwrotne do funkcji trygonometrycznych). Parametry funkcji trygonometrycznych podawane są w radianach. W przypadku wystąpienia dzielenia przez zero wynik funkcji jest nieokreślony. Operacje wykonywane są na każdej składowej parametru (lub parametrów) funkcji.

funkcja	opis
genType radians (genType degrees)	konwersja stopni na radiany
genType degrees (genType radians)	konwersja radianów na stopnie
genType sin (genType angle)	funkcja trygonometryczna sinus
genType cos (genType angle)	funkcja trygonometryczna cosinus
genType tan (genType angle)	funkcja trygonometryczna tangens
genType asin (genType x)	funkcja cyklometryczna arcus sinus, dziedzina funkcji $\|x\| \leq 1$, przedział wartości $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$
genType acos (genType x)	funkcja cyklometryczna arcus cosinus, dziedzina funkcji $\|x\| \leq 1$, przedział wartości $[0, \pi]$
genType atan (genType y, genType x)	funkcja cyklometryczna arcus tangens y/x, znaki x i y określają, w której ćwiartce leży obliczany kąt, przedział wartości $[-\pi, \pi]$
genType atan (genType $y_o ver_x$)	funkcja cyklometryczna arcus tangens, przedział wartości $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$

Tabela 4: Zestawienie funkcji trygonometrycznych

1.9.2. Funkcje wykładnicze

Opis funkcji wykładniczych i innych przedstawionych w tabeli 5 dotyczy operacji na każdej składowej parametru (lub parametrów) funkcji.

funkcja	opis
genType pow (genType x, genType y)	funkcja potęgowa x^y, dziedzina funkcji $x \geq 0$, wartość nieokreślona, gdy $x = 0$ i $x \leq 0$
genType exp (genType x)	funkcja wykładnicza e^x
genType log (genType x)	logarytm naturalny, dziedzina funkcji $x > 0$

funkcja	opis
genType exp2 (genType x)	funkcja 2^x
genType log2 (genType x)	logarytm o podstawie 2, dziedzina funkcji $x > 0$
genType sqrt (genType x)	pierwiastek kwadratowy $\sqrt{x}$, dziedzina funkcji $x \geq 0$
genType inversesqrt (genType x)	funkcja $\frac{1}{\sqrt{x}}$, dziedzina funkcji $x > 0$

Tabela 5: Zestawienie funkcji wykładniczych

1.9.3. Funkcje ogólne

Funkcje ogólne zestawione w tabeli 6 operują na różnego rodzaju argumentach, przy czym prezentowany opis dotyczy działania na każdej składowej argumentu.

funkcja	opis
genType abs (genType x)	wartość bezwzględna
genType sign (genType x)	funkcja $\begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$
genType floor (genType x)	największa liczba całkowita, która jest mniejsza lub równa x
genType ceil (genType x)	najmniejsza liczba całkowita, która jest większa lub równa x
genType fract (genType x)	funkcja $x - floor\,(x)$
genType mod (genType x, float y) genType mod (genType x, genType y)	funkcja $x - y \cdot floor\left(x/y\right)$
genType min (genType x, genType y) genType min (genType x, float y)	minimim
genType max (genType x, genType y) genType max (genType x, float y)	maksimum
genType clamp (genType x, genType minVal, genType maxVal) genType clamp (genType x, float minVal, float maxVal)	funkcja $\min\left(max\left(x, minVal\right), maxVal\right)$ wartość niezdefiniowana, gdy $minVal > maxVal$
genType mix (genType x, genType y, genType a)	funkcja $x \cdot (1 - a) + y \cdot a$

funkcja	opis
genType mix (genType x, genType y, float a)	
genType step (float edge, genType x) genType step (genType edge, genType x)	funkcja $\begin{cases} 0 & x < edge \\ 1 & x \geq edge \end{cases}$
genType smoothstep (genType edge0, genType edge1, genType x) genType smoothstep (float edge0, float edge1, genType x)	funkcja $t \cdot t \cdot (3 - 2 \cdot t)$, gdzie t wynosi $clamp\left(\frac{x - edge0}{edge1 - edge0}, 0, 1\right)$ wartość niezdefiniowana dla $edge0 \geq edge1$

Tabela 6: Zestawienie funkcji ogólnych

1.9.4. Funkcje geometryczne

Spośród funkcji geometrycznych przedstawionych w tabeli 7 szczególną uwagę warto zwrócić na `ftransform`, która umożliwia bezpośrednie wyliczenie wartości zmiennej `gl_Position`.

funkcja	opis
float length (genType x)	długość wektora
float distance (genType p0, genType p1)	odległość pomiędzy punktami $length\,(p0 - p1)$
float dot (genType x, genType y)	iloczyn skalarny
vec3 cross (vec3 x, vec3 y)	iloczyn wektorowy
genType normalize (genType x)	normalizacja wektora
vec4 ftransform ()	przekształcenie wierzchołka odpowiadające statycznym przekształceniom w bibliotece OpenGL; funkcja dostępna wyłącznie w programach cieniowania wierzchołków
genType faceforward (genType N, genType I, genType Nref)	jeżeli $dot\,(Nref, I) < 0$ funkcja zwraca N, w przeciwnym wypadku funkcja zwraca $-N$

funkcja	opis
genType reflect (genType I, genType N)	kierunek odbicia wektora I od powierzchni określonej przez jednostkowy wektor normalny N, $I - 2 \cdot dot\,(N, I) \cdot N$
genType refract (genType I, genType N, float eta)	wektor załamania losowego jednostkowego wektora I od powierzchni określonej przez jednostkowy wektor normalny N przy użyciu współczynnika załamania eta, jeżeli wyrażenie $k = 1 - eta^2 \cdot (1 - dot\,(N, I) \cdot dot\,(N, I))$ jest mniejsze od 0, to funkcja zwraca $genType\,(0)$, w przeciwnym wypadku zwracana jest wartość $eta \cdot I - (eta \cdot dot\,(N, I) + sqrt\,(k)) \cdot N$

Tabela 7: Zestawienie funkcji geometrycznych

1.9.5. Funkcje macierzowe

Zestawienie funkcji działających na macierzach przedstawia tabela 8. Funkcje operujące na macierzach prostokątnych zostały wprowadzone w wersji 1.20 języka GLSL.

funkcja	opis
mat matrixCompMult (mat x, mat y)	mnożenie równoległe elementów macierzy $x_{ij} \cdot y_{ij}$; do mnożenia algebraicznego macierzy służy wbudowany operator *
mat2 outerProduct (vec2 c, vec2 r) mat3 outerProduct (vec3 c, vec3 r) mat4 outerProduct (vec4 c, vec4 r) mat2x3 outerProduct (vec3 c, vec2 r) mat3x2 outerProduct (vec2 c, vec3 r) mat2x4 outerProduct (vec4 c, vec2 r) mat4x2 outerProduct (vec2 c, vec4 r) mat3x4 outerProduct (vec4 c, vec3 r) mat4x3 outerProduct (vec3 c, vec4 r)	iloczyn dwóch wektorów, gdzie c jest traktowany jako macierz z jedną kolumną, a wektor r jest macierzą z jednym wierszem
mat2 transpose (mat2 m) mat3 transpose (mat3 m) mat4 transpose (mat4 m)	transponowanie macierzy, macierz wejściowa nie jest modyfikowana

funkcja	opis
mat2x3 transpose (mat3x2 m) mat3x2 transpose (mat2x3 m) mat2x4 transpose (mat4x2 m) mat4x2 transpose (mat2x4 m) mat3x4 transpose (mat4x3 m) mat4x3 transpose (mat3x4 m)	

Tabela 8: Zestawienie funkcji macierzowych

1.9.6. Funkcje porównujące wektory

Przedstawione w tabeli 9 funkcje porównujące wektory działają na elementach składowych wektorów w wyniku zwracając wektor z liczbami typu `bool`. Oczywiście wymiary wektorów wejściowych muszą być równe. Jedynie trzy ostatnie funkcje wykonują operacje logiczne na wektorach z liczbami typu `bool`.

Użyte w opisach funkcji typy `bvec` oznaczają odpowiednio: `bvec2`, `bvec3` lub `bvec4`. Analogiczna zasada dotyczy `vex` - są to typy: `vec2`, `vec3` lub `vec4` oraz `ivec`, co odpowiada `ivec2`, `ivec3` lub `ivec4`. W każdym przypadku wymiary porównywanych wektorów muszą być zgodne.

funkcja	opis
bvec lessThan (vec x, vec y) bvec lessThan (ivec x, ivec y)	wynik porównania $x < y$
bvec lessThanEqual (vec x, vec y) bvec lessThanEqual (ivec x, ivec y)	wynik porównania $x \leq y$
bvec greaterThan (vec x, vec y) bvec greaterThan (ivec x, ivec y)	wynik porównania $x > y$
bvec greaterThanEqual (vec x, vec y) bvec greaterThanEqual (ivec x, ivec y)	wynik porównania $x \geq y$
bvec equal (vec x, vec y) bvec equal (ivec x, ivec y) bvec equal (bvec x, bvec y)	wynik porównania $x = y$
bvec notEqual (vec x, vec y) bvec notEqual (ivec x, ivec y) bvec notEqual (bvec x, bvec y)	wynik porównania $x \neq y$
bool any (bvec x)	wartość `true`, gdy co najmniej jeden element wektora `x` ma wartość `true`

funkcja	opis
bool all (bvec x)	wartość `true`, gdy wszystkie elementy wektora `x` mają wartość `true`
bvec not (bvec x)	negacja logiczna elementów wektora x

Tabela 9: Zestawienie funkcji porównujących wektory

1.9.7. Funkcje próbkujące tekstury

Funkcje próbkujące tekstury dostępne są zarówno dla programów cieniowania fragmentów jak i programów cieniowania wierzchołków. Dostęp do tekstury uzyskiwany jest za pośrednictwem uchwytu - parametry `sampler`. Występujące w części funkcji opcjonalne parametry `bias` dostępne są wyłącznie w programach cieniowania fragmentów. Wartość tego parametru jest dodawana do poziomu szczegółowości mipmap (LOD) przed pobraniem próbki tekstury. Występujący w części funkcji parametr `lod`, to oczywiście wspomniany przed chwilą poziom szczegółowości mipmap.

funkcja	opis
vec4 texture1D (sampler1D sampler, float coord [, float bias]) vec4 texture1DProj (sampler1D sampler, vec2 coord [, float bias]) vec4 texture1DProj (sampler1D sampler, vec4 coord [, float bias]) vec4 texture1DLod (sampler1D sampler, float coord, float lod) vec4 texture1DProjLod (sampler1D sampler, vec2 coord, float lod) vec4 texture1DProjLod (sampler1D sampler, vec4 coord, float lod)	pobranie próbki tekstury jednowymiarowej o współrzędnej `coord`; wersje projekcyjne („Proj”) dzielą współrzędną `s` przez ostatni element `coord`
vec4 texture2D (sampler2D sampler, vec2 coord [, float bias]) vec4 texture2DProj (sampler2D sampler, vec3 coord [, float bias]) vec4 texture2DProj (sampler2D sampler, vec4 coord [, float bias]) vec4 texture2DLod (sampler2D sampler, vec2 coord, float lod)	pobranie próbki tekstury dwuwymiarowej o współrzędnych `coord`; wersje projekcyjne („Proj”) dzielą współrzędne `s` i `t` przez ostatni element `coord`

funkcja	opis
vec4 texture2DProjLod (sampler2D sampler, vec3 coord, float lod) vec4 texture2DProjLod (sampler2D sampler, vec4 coord, float lod)	
vec4 texture3D (sampler3D sampler, vec3 coord [, float bias]) vec4 texture3DProj (sampler3D sampler, vec4 coord [, float bias]) vec4 texture3DLod (sampler3D sampler, vec3 coord, float lod) vec4 texture3DProjLod (sampler3D sampler, vec4 coord, float lod)	pobranie próbki tekstury trójwymiarowej o współrzędnych `coord`; wersje projekcyjne („Proj”) dzielą współrzędne `s`, `t` i `r` przez ostatni element `coord`
vec4 textureCube (samplerCube sampler, vec3 coord [, float bias]) vec4 textureCubeLod (samplerCube sampler, vec3 coord, float lod)	pobranie próbki tekstury sześciennej o współrzędnych `coord`
vec4 shadow1D (sampler1DShadow sampler, vec3 coord [, float bias]) vec4 shadow2D (sampler2DShadow sampler, vec3 coord [, float bias]) vec4 shadow1DProj (sampler1DShadow sampler, vec4 coord [, float bias]) vec4 shadow2DProj (sampler2DShadow sampler, vec4 coord [, float bias]) vec4 shadow1DLod (sampler1DShadow sampler, vec3 coord, float lod) vec4 shadow2DLod (sampler2DShadow sampler, vec3 coord, float lod) vec4 shadow1DProjLod (sampler1DShadow sampler, vec4 coord, float lod) vec4 shadow2DProjLod (sampler2DShadow sampler, vec4 coord, float lod)	porównanie wartości głębi z zawartością tekstury głębi; wersje projekcyjne („Proj”) dzielą współrzędne tekstur przez ostatni element `coord`

Tabela 10: Zestawienie funkcji próbkujących tekstury

1.9.8. Funkcje różniczkowe

Funkcje różniczkowe (tabela numer 11) są dostępne tylko dla programów cieniowania fragmentów. Ponieważ obliczanie wartości pochodnych jest kosztowne i może być niestabilne numerycznie, implementacja może stosować szybkie przybliżone metody ich obliczania. Ponadto implementacja może udostępniać regulację dokładności obliczania funkcji różniczkowych przy pomocy wskazówki renderingu określonej stałą `GL_FRAGMENT_SHADER_DERIVATIVE_HINT`.

funkcja	opis
genType dFdx (genType p)	pochodna w kierunku x z użyciem lokalnego różniczkowania dla parametru p
genType dFdy (genType p)	pochodna w kierunku y z użyciem lokalnego różniczkowania dla parametru p
genType fwidth (genType p)	suma wartości bezwzględnych pochodnych w kierunkach x i y z użyciem lokalnego różniczkowania dla parametru p $abs\left(dFdx\left(p\right)\right) + abs\left(dFdy\left(p\right)\right)$

Tabela 11: Zestawienie funkcji różniczkowych

1.9.9. Funkcje stochastyczne

Wymienione w tabeli 12 funkcje stochastyczne są dostępne dla obu rodzajów programów cieniowania. Funkcje zwracają wartości pseudolosowe o następujących właściwościach:

— zwracane wartości zawierają się w przedziale $[-1, 1]$,
— średnia zwracanych wartości wynosi 0,
— są powtarzalne, czyli dla określonej danej wejściowej zwracają taką samą wartość,
— własności statystyczne nie ulegają zmianom pod wpływem obrotu i przesunięcia,
— po przesunięciu zazwyczaj zwracają inne wartości,
— posiadają wąskie pasmo widocznych częstotliwości ze środkiem położonym pomiędzy 0,5 a 1,
— są klasy C^1, czyli pierwsza pochodna jest funkcją ciągłą.

funkcja	opis
float noise1 (genType x)	jednowymiarowa wartość pseudolosowa na podstawie parametru x
vec2 noise2 (genType x)	dwuwymiarowa wartość pseudolosowa na podstawie parametru x

funkcja	opis
vec3 noise3 (genType x)	trójwymiarowa wartość pseudolosowa na podstawie parametru x
vec4 noise4 (genType x)	czterowymiarowa wartość pseudolosowa na podstawie parametru x

Tabela 12: Zestawienie funkcji stochastycznych

Na koniec opisu języka GLSL jeszcze jedna uwaga. Zawsze należy zapoznać się z dokumentacją udostępnianą przez producentów procesorów kart graficznych. Przykładowo w wydaniu 60 sterowników do kart z procesorami NVIDIA, wszystkie funkcje stochastyczne zwracały wyłącznie wartości 0.

Literatura

[1] Mark Segal, Kurt Akeley: The OpenGL Graphics System. A Specification Version 2.0
[2] Jackie Neider, Tom Davis, Mason Woo: OpenGL Programming Guide „The Red Book"
[3] Richard S. Wright jr, Michael Sweet: OpenGL Księga eksperta, Helion 1999
[4] Richard S. Wright jr, Michael Sweet: OpenGL Księga eksperta Wydanie III, Helion 2005
[5] The official OpenGL web page, http://www.opengl.org
[6] Piotr Andrzejewski, Jakub Kurzak: Wprowadzenie do OpenGL. Programowanie zastosowań graficznych, Kwantum 2000
[7] Kevin Hawkins, Dave Astle: OpenGL. Programowanie gier, Helion 2003
[8] Mark J. Kilgard: The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3. Silicon Graphics, Inc. 1996
[9] Mark J. Kilgard: All About OpenGL Extensions, http://www.opengl.org/resources/features/OGLextensions/
[10] Jon Leech: How to Create OpenGL Extensions, http://oss.sgi.com/projects/ogl-sample/registry/doc/rules.html
[11] Silicon Graphics, Inc: OpenGL® Extension Registry, http://oss.sgi.com/projects/ogl-sample/registry/, http://www.opengl.org/registry/

Spis rysunków

Spis tabel

Skorowidz