OpenGL 4.2

GLSL

Janusz Ganczarski

www.januszg.hg.pl

Wstęp

Język programów cieniowania OpenGL, w skrócie oznaczany GLSL (ang. *OpenGL Shading Language*) lub GLslang został wprowadzony w wersji 2.0 biblioteki OpenGL. Język ten jest używany we wszystkich trzech rodzajach shaderów: wierzchołków, geometrii i fragmentów. Różnice syntaktyczne pomiędzy shaderami sprowadzają się głównie wbudowanych zmiennych oraz zakresu dostępnych funkcji. Profil podstawowy wersji 4.2 biblioteki OpenGL zapewnia obsługę GLSL w wersjach 4.20 oraz 4.10, 4.00, 3.30, 1.50 i 1.40, ale poniżej przedstawimy tylko wersję 4.20. Opis obejmuje pełną specyfikację języka i funkcji poza elementami, które są określone jako przestarzałe. W opisie celowo pominięto schematy gramatyczne języka – zainteresowanego Czytelnika odsyłamy do specyfikacji.

Przypomnijmy, że numer obsługiwanej wersji GLSL zawiera zmienna stanu `GL_SHADING_LANGUAGE_VERSION`, której wartość można odczytać korzystając z funkcji `glGetString`.

Podstawy składni

Składnia GLSL jest oparta na językach C i C++, stąd poniższy opis jest ograniczony do niezbędnego minimum. Czytelnika zainteresowanego bliższymi szczegółami gramatyki języka GLSL zapraszamy do lektury specyfikacji.

Zbiór znaków

Program w języku GLSL jest ciągiem znaków będących podzbiorem znaków UTF-8 (poza komentarzami). 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 GLSL. 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).

Ciąg znaków tworzący program w języku GLSL może być podzielony na wiersze wyznaczane znakami CE lub LF. Podział na wiersze nie ma znaczenia przy kompilacji programu, jest jednak elementem przydatnym np. do diagnostyki błędów. Wiersze numerowane są od 0. Podobnie jak w językach C/C++ znak `\` oznacza, że dany wiersz ma swoją kontynuację w następnej linii. Przykładowo zapis:

```
float f\
oo;
```

jest równoważny zapisowi w pojedynczym wierszu:

```
float foo;
```

Komentarze

GLSL wykorzystuje takie same komentarze jak język C++. Znaki `//` rozpoczynają komentarz obowiązujący do końca linii, a komentarz dowolnej części programu określają pary znaków `/*` i `*/`. Nie są obsługiwane komentarze zagnieżdżone. Wewnątrz komentarza mogą wystąpić dowolne znaki, za wyjątkiem znaku o wartości 0. Użycie w komentarzu `//` znaku `\` spowoduje, że następny wiersz także będzie komentarzem:

```
// komentarz jednowierszowy, z kontynuacją w następnej linii \
a = b; // to jest ciągle pierwszy komentarz
```

Słowa zarezerwowane

GLSL w wersji 4.20 rezerwuje następujące słowa: `attribute`, `const`, `uniform`, `varying`, `coherent`, `volatile`, `restrict`, `readonly`, `writeonly`, `atomic_uint`, `layout`, `centroid`, `flat`, `smooth`, `noperspective`, `patch`, `sample`, `break`, `continue`, `do`, `for`, `while`, `switch`, `case`, `default`, `if`, `else`, `subroutine`, `in`, `out`, `inout`, `float`, `double`, `int`, `void`, `bool`, `true`, `false`, `invariant`, `discard`, `return`, `mat2`, `mat3`, `mat4`, `dmat2`, `dmat3`, `dmat4`, `mat2x2`, `mat2x3`, `mat2x4`, `dmat2x2`, `dmat2x3`, `dmat2x4`, `mat3x2`, `mat3x3`, `mat3x4`, `dmat3x2`, `dmat3x3`, `dmat3x4`, `mat4x2`, `mat4x3`, `mat4x4`, `dmat4x2`, `dmat4x3`, `dmat4x4`, `vec2`, `vec3`, `vec4`, `ivec2`, `ivec3`, `ivec4`, `bvec2`, `bvec3`, `bvec4`, `dvec2`, `dvec3`, `dvec4`, `uint`, `uvec2`, `uvec3`, `uvec4`, `lowp`, `mediump`, `highp`, `precision`, `sampler1D`, `sampler2D` `sampler3D`, `samplerCube`, `sampler1DShadow`, `sampler2DShadow`, `samplerCubeShadow`, `sampler1DArray`, `sampler2DArray`, `sampler1DArrayShadow`, `sampler2DArrayShadow`, `isampler1D`, `isampler2D`, `isampler3D`, `isamplerCube`, `isampler1DArray`, `isampler2DArray`, `usampler1D`, `usampler2D`, `usampler3D`, `usamplerCube`, `usampler1DArray`, `usampler2DArray`, `sampler2DRect`, `sampler2DRectShadow`, `isampler2DRect`, `usampler2DRect`, `samplerBuffer`, `isamplerBuffer`, `usamplerBuffer`, `sampler2DMS`, `isampler2DMS`, `usampler2DMS`, `sampler2DMSArray`, `isampler2DMSArray`, `usampler2DMSArray`, `samplerCubeArray`, `samplerCubeArrayShadow`, `isamplerCubeArray`, `usamplerCubeArray`, `image1D`, `iimage1D`, `uimage1D`, `image2D`, `iimage2D`, `uimage2D`, `image3D`, `iimage3D`, `uimage3D`, `image2DRect`, `iimage2DRect`, `uimage2DRect`, `imageCube`, `iimageCube`, `uimageCube`, `imageBuffer`, `iimageBuffer`, `uimageBuffer`, `image1DArray`, `iimage1DArray`, `uimage1DArray`, `image2DArray`, `iimage2DArray`, `uimage2DArray`, `imageCubeArray`, `iimageCubeArray`, `uimageCubeArray`, `image2DMS`, `iimage2DMS`, `uimage2DMS`, `image2DMSArray`, `iimage2DMSArray`, `uimage2DMSArray` i `struct`.

Specyfikacja GLSL zawiera także wykaz słów zarezerwowanych do użycia w przyszłych wersjach tego języka: `common`, `partition`, `active`, `asm`, `class`, `union`, `enum`, `typedef`, `template`, `this`, `packed`, `resource`, `goto`, `inline`, `noinline`, `public`, `static`, `extern`, `external`, `interface`, `long`, `short`, `half`, `fixed`, `unsigned`, `superp`, `input`, `output`, `hvec2`, `hvec3`, `hvec4`, `fvec2`, `fvec3`, `fvec4`, `sampler3DRect`, `filter`, `sizeof`, `cast`, `namespace`, `using` i `row_major`.

Ponadto zarezerwowane do ewentualnego przyszłego użycia przez GLSL są identyfikatory zawierające dwa znaki podkreślenia „`__`”.

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 i nie mogą być wykorzystywane do deklarowania jako zmienne lub funkcje w shaderze.

Preprocesor

Preprocesor w języku GLSL ma podobne możliwości do preprocesora dostępnego w językach C i C++. Występujące różnice związane przede wszystkim są ze specyficznym przeznaczeniem GLSL.

Dyrektywy

Preprocesor w języku GLSL posiada następujące dyrektywy: `#`, `#define`, `#undef`, `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`, `#error`, `#pragma`, `#extension`, `#version` i `#line`. Ponadto dostępne są operatory `defined` i `##`, które opiszemy nieco dalej.

Każdy znak # (ang. *numer sign*) może być poprzedzony w wierszu tylko przez spacje lub tabulator poziomy. Ponadto znaki te mogą oddzielać # od właściwej nazwy dyrektywy preprocesora. Każda z dyrektyw kończy się znakiem nowego wiersza. Wiersz z pojedynczym znakiem # jest przez preprocesor ignorowany.

Dyrektywy `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` i `#endif` w swoim działaniu są odpowiednikami dyrektyw standardowego preprocesora C++. Wyrażenie następujące po `#if` i `#elif` są ograniczone do wyrażeń operujących na stałych literałach całkowitych oraz identyfikatorach obsługiwanych przez operator `defined`. Błędem jest użycie `#if` lub `#elif` z wyrażeniem zawierającym niezdefiniowaną nazwę makra inne niż argument operatora `defined`. Nie są także obsługiwane stałe znakowe.

Pozostałe dyrektywy preprocesora są ściśle związane ze specyfiką języka GLSL i zostaną szczegółowo omówione w dalszej części tekstu.

Operatory

Semantyka preprocesora GLSL jest zgodna ze zdefiniowaną w preprocesorze C++, nie jest natomiast zgodna z językiem GLSL. Wyrażenia preprocesora są przetwarzanie zgodnie z zachowaniem procesora jednostki centralnej komputera, a nie procesora układu graficznego. W Tabela 1 zestawiono wszystkie operatory preprocesora GLSL w kolejności od najwyższego (1) do najniższego (12) priorytetu.

Priorytet	Rodzaj operatora	Operatory	Kolejność wiązania
1 (najwyższy)	grupowanie w nawiasy	`( )`	-
2	jednoargumentowe	`defined + ~ !`	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	bitowa różnica symetryczna XOR	`^`	od lewej do prawej
10	bitowe OR	`|`	od lewej do prawej
11	logiczne AND	`&&`	od lewej do prawej
12 (najniższy)	logiczne OR	`||`	od lewej do prawej

Tabela 1 Priorytety operatorów preprocesora GLSL

Operator `defined`, podobnie jak w języku C++, może być używany na dwa sposoby:

```
defined identifier

defined ( identifier )
```

Operator konkatenacji (łączenia) symboli `##` działa analogicznie jak w preprocesorze języka C++. Wynikiem działania tego operatora jest poprawny pojedynczy element (token) składający się ze złączonych elementów wejściowych.

Zauważmy, że GLSL nie zawiera operatora `sizeof`.

Makra

Do definiowania makr preprocesora używane są dyrektywy `#define` i `#undef`, których funkcjonalność odpowiada możliwościom języka C++ W tym zakresie. Preprocesor języka GLSL zawiera trzy predefiniowane makra:

- `__LINE__` - zwraca dziesiętną liczbę całkowitą z powiększonym o jeden numerem bieżącego wiersza w obecnym ciągu znaków programu źródłowego,

- `__FILE__` - zwraca dziesiętną liczbę całkowitą z numerem bieżącego ciągu znaków program źródłowego; specyfika makra `__FILE__` związana jest z organizacją budowy tekstu źródłowego programu GLSL, który może składać się z wielu odrębnych ciągów znaków, przy jednoczesnym braku takiego pojęcia jak plik z zapisem źródła programu,
- `__VERSION__` - zwraca dziesiętną liczbę całkowitą z numerem wersji języka GLSL; przykładowo dla wersji 4.20 GLSL makro `__VERSION__` zawiera dziesiętną liczbę 420.

Wszystkie makra zawierające dwa podkreślenia „__” lub prefiks „`GL_`” są zarezerwowane do przyszłego użytku przez preprocesor GLSL. Rozwinięcie makr nie jest realizowane w wierszach zawierających dyrektywy `#extension` i `#version`.

Diagnostyka błędów

Dyrektywa preprocesora `#error` przekazuje komunikat diagnostyczny w dzienniku informacyjnym obiektu shadera. Tekst komunikatu zawarty jest od znaku kończącego dyrektywę `#error` do końca wiersza.

Przy zawansowanej diagnostyce błędów w rozbudowanych programach cieniowania przydatna może być także dyrektywa `#line` preprocesora występujące w jednej z dwóch form:

```
#line line
#line line source-string-number
```

gdzie `line` i `source-string-number` są stałymi wyrażeniami całkowitymi. Użycie tej dyrektywy powoduje takie zachowanie kompilatora GLSL, jakby kompilowany był wiersz tekstu źródłowego programu cieniowania o numerze `line` powiększony o jeden, w zbiorze wierszy o numerze `source_string_number`. Kolejne wiersze programu numerowane są narastająco, aż do ewentualnego wystąpienia kolejnej dyrektywy `#line`.

Kontrola kompilatora

Dyrektywa `#pragma` umożliwia zależną od implementacji kontrolę nad kompilatorem języka GLSL. W przypadku, gdy wyrażenia zawarte w dyrektywie `#pragma` nie są obsługiwane przez daną implementację, preprocesor ignoruje dyrektywę.

GLSL zawiera także kilka predefiniowanych wartości dyrektywy `#pragma`. Pierwsza z nich zarezerwowana jest do użytku w przyszłych wersjach języka GLSL:

```
#pragma STDGL
```

Następne służą do włączania i wyłączania optymalizacji oraz włączania/wyłączania informacji używanych przez debuger przy usuwaniu błędów:

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

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

Domyślnie optymalizacja jest włączona, a generowanie danych dla debugera jest wyłączone.

Wersja GLSL i obsługa profili

Shadery muszą deklarować wersję języka GLSL, w której są napisane. Służy do tego dyrektywa preprocesora:

```
#version number profile
```

gdzie `number` określa wersję języka (w analogicznej konwencji jak opisane wyżej makro `__VERSION__`), a `profile` jest opcjonalnym argumentem zawierającym nazwę profilu. W wersji 4.20 języka GLSL dostępne są dwa profile: `core` (profil podstawowy) i `compatibility` (profil kompatybilny). Domyślnie stosowany jest profil podstawowy.

Dyrektywa `#version` musi wystąpić na samym początku programu, za wyjątkiem komentarzy i białych spacji. Brak tej dyrektywy spowoduje, że shader będzie domyślnie korzystał z wersji 1.10 języka GLSL. Shadery w wersji 4.20 GLSL mogą być konsolidowane razem z shaderami w wersjach 1.40, 1.50, 3.30, 4.00 oraz 4.10 GLSL, co odpowiada wersji 3.1, 3.2, 3.3, 4.0 i 4.1 biblioteki OpenGL (skok w numeracji GLSL związany jest z ujednoliceniem od wersji 3.3 numeracji OpenGL i GLSL). Konsolidacja shadera w wersji 4.20 z shaderami w wersjach 1.30 i starszymi nie jest dopuszczalna. Ponadto, podanie wersji 1.00 oznacza, że shader napisany w OpenGL ES Shading Language, czyli dialekcie GLSL przeznaczonym typowo do obsługi urządzeń przenośnych i wbudowanych (standardowy GLSL w wersji 1.00 nie obsługiwał dyrektywy `#version`).

Implementacja GLSL w wersji 4.20 musi zawierać następujące makro:

```
#define GL_core_profile 1
```

Ponadto implementacje wspierające profil kompatybilny muszą zawierać także poniższe makro:

```
#define GL_compatibility_profile 1
```

Rozszerzenia GLSL

Standardowo kompilator GLSL obsługuje semantykę i syntaktykę programu zgodnie ze specyfikacją określonej wersji języka. Jakiekolwiek rozszerzenie GLSL musi zostać wcześniej włączone. Służy do tego dyrektywa `#extension` preprocesora, która jest prostym, niskopoziomowym mechanizmem do obsługi rozszerzeń języka GLGL. Dyrektywa ta występuje w dwóch postaciach:

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

gdzie `extension_name` jest nazwą rozszerzenia, a argument `behavior` określa sposób działania dyrektywy, który opiszemy poniżej. Kwalifikator `all` oznacza, instrukcja dotyczy wszystkich rozszerzeń dostępnych w danej implementacji GLSL. Oto dopuszczalne wartości argumentu `behavior` dyrektywy `#extension`:

- `require` – program wymaga rozszerzenia `extension_name`; błąd jest generowany w przypadku braku wymganego rozszerzenia lub użycia wersji dyrektywy `#extension all`,
- `enable` – włączenie rozszerzenia o nazwie `extension_name`; w przypadku braku wymaganego rozszerzenia generowanie jest ostrzeżenie, natomiast w przypadku użycia wersji dyrektywy `#extension all` generowany jest błąd,
- `warn` – włączenie rozszerzenia o nazwie `extension_name`, z tym, że kompilator generuje ostrzeżenie przy każdym jego użyciu za wyjątkiem sytuacji, gdy jego obsługa jest niezbędna przez inne włączone (argument `enable`) lub wymagane (argument `require`) rozszerzenie; ostrzeżenie generowane jest także w przypadku, gdy rozszerzenie `extension_name` nie jest dostępne, a w przypadku użycia wersji dyrektywy `#extension all` ostrzeżenie generowane jest przy użyciu każdego z dostępnych rozszerzeń,
- `disable` – wyłączenie rozszerzenia o nazwie `extension_name`; w przypadku użycia wersji dyrektywy `#extension all` następuje wyłączenie wszystkich dostępnych rozszerzeń; w przypadku braku nazwy rozszerzenia `extension_name` generowanie jest ostrzeżenie.

Każde użycie dyrektywy `#extension` modyfikuje poprzednie ustawienie konfiguracji rozszerzeń języka GLSL. Wersja `#extension all` dyrektywy modyfikuje ustawienia wszystkich dostępnych rozszerzeń, ale jest dostępna tylko dla argumentów `warn` i `disable`. Początkowo wszystkie rozszerzenia GLSL są wyłączone co odpowiada poniższej dyrektywie:

```
#extension all : disable
```

Generalnie konfiguracja rozszerzeń dotyczy bieżącej jednostki kompilacji, czyli pojedynczego shadera. Odpowiednie dyrektywy preprocesora powinny wystąpić przed użyciem innych instrukcji GLSL, za wyjątkiem pozostałych dyrektyw preprocesora. Jeżeli będzie to niezbędne, konsolidator może wymagać użycia dyrektywy z odpowiednim rozszerzeniem w więcej niż jednej jednostce kompilacji.

Zmienne, typy i konwersje

Wszystkie zmienne w programie GLSL muszą być zadeklarowane przed pierwszym użyciem. W wielu przypadkach deklaracja zmiennej może być połączona z jej inicjalizacją przy użycia operatora przypisania. W GLSL nie ma typów domyślnych, każda zmienna i funkcja musi mieć zadeklarowany typ oraz opcjonalne kwalifikatory. Trzeba także pamiętać, że GLSL jest językiem o silnej kontroli typów i ściśle określonych możliwościach automatycznej ich konwersji.

Podstawowe typy danych

Podstawowe typy danych w języku GLSL przedstawiono w Tabela 2. Typy będące uchwytami tekstur wydzielono i pogrupowano z podziałem na rodzaj danych tekstury. Typy podstawowe mogą być agregowane przy użyciu tablic i struktur definiowanych przez użytkownika. Zauważmy także, że GLSL nie posiada żadnego odpowiednika typów wskaźnikowych z języków C/C++.

Typ	Opis
`void`	Typ pusty dla funkcji niezwracającej wartości.
`bool`	Typ logiczny, wartość `true` lub `false`.
`int`	Liczba całkowita ze znakiem.
`uint`	Liczba całkowita bez znaku.
`float`	Liczba zmiennoprzecinkowa pojedynczej precyzji.
`double`	Liczba zmiennoprzecinkowa podwójnej precyzji.
`vec2`	Dwuelementowy wektor z liczbami typu `float`.
`vec3`	Trójelementowy wektor z liczbami typu `float`.
`vec4`	Czteroelementowy wektor z liczbami typu `float`.
`dvec2`	Dwuelementowy wektor z liczbami typu `double`.
`dvec3`	Trójelementowy wektor z liczbami typu `double`.
`dvec4`	Czteroelementowy wektor z liczbami typu `double`.
`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`.
`uvec2`	Dwuelementowy wektor z liczbami typu `uint`.
`uvec3`	Trójelementowy wektor z liczbami typu `uint`.
`uvec4`	Czteroelementowy wektor z liczbami typu `uint`.
`mat2`	Macierz 2×2 z liczbami typu `float`.
`mat3`	Macierz 3×3 z liczbami typu `float`.
`mat4`	Macierz 4×4 z liczbami typu `float`.

`mat2x2`	To samo, co `mat2`.
`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).
`mat3x3`	To samo, co `mat3`.
`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).
`mat4x4`	To samo, co `mat4`.
`dmat2`	Macierz 2×2 z liczbami typu `double`.
`dmat3`	Macierz 3×3 z liczbami typu `double`.
`dmat4`	Macierz 4×4 z liczbami typu `double`.
`dmat2x2`	To samo, co `dmat2`.
`dmat2x3`	Macierz 2×3 z liczbami typu `double` (dwie kolumny, trzy wiersze).
`dmat2x4`	Macierz 2×4 z liczbami typu `double` (dwie kolumny, cztery wiersze).
`dmat3x2`	Macierz 3×2 z liczbami typu `double` (trzy kolumny, dwa wiersze).
`dmat3x3`	To samo, co `dmat3`.
`dmat3x4`	Macierz 3×4 z liczbami typu `double` (trzy kolumny, cztery wiersze).
`dmat4x2`	Macierz 4×2 z liczbami typu `double` (cztery kolumny, dwa wiersze).
`dmat4x3`	Macierz 4×3 z liczbami typu `double` (cztery kolumny, trzy wiersze).
`dmat4x4`	To samo, co `dmat4`.
`sampler1D` `image1D`	Uchwyt dostępu do tekstury jednowymiarowej.
`sampler2D` `image2D`	Uchwyt dostępu do tekstury dwuwymiarowej.
`sampler3D` `image3D`	Uchwyt dostępu do tekstury trójwymiarowej.
`samplerCube` `imageCube`	Uchwyt dostępu do tekstury sześciennej.
`sampler2DRect` `image2DRect`	Uchwyt dostępu do tekstury prostokątnej.
`sampler1DArray` `image1DArray`	Uchwyt dostępu do tablicy tekstur jednowymiarowych.
`sampler2DArray` `image2DArray`	Uchwyt dostępu do tablicy tekstur dwuwymiarowych.
`samplerBuffer` `imageBuffer`	Uchwyt dostępu do tekstury buforowej.
`sampler2DMS` `image2DMS`	Uchwyt dostępu do dwuwymiarowej tekstury wielopróbkowania.
`sampler2DMSArray` `image2DMSArray`	Uchwyt dostępu do tablicy dwuwymiarowych tekstur wielopróbkowania.
`samplerCubeArray` `imageCubeArray`	Uchwyt dostępu do tablicy tekstur sześciennych.
`sampler1DShadow`	Uchwyt dostępu do jednowymiarowej tekstury głębi z porównywaniem.
`sampler2DShadow`	Uchwyt dostępu do dwuwymiarowej tekstury głębi z porównywaniem.
`sampler2DRectShadow`	Uchwyt dostępu do tekstury prostokątnej z porównywaniem.
`sampler1DArrayShadow`	Uchwyt dostępu do tablicy jednowymiarowych tekstur głębi z porównywaniem.
`sampler2DArrayShadow`	Uchwyt dostępu do tablicy dwuwymiarowych tekstur głębi z porównywaniem.
`samplerCubeShadow`	Uchwyt dostępu do tekstury sześciennej głębi z porównywaniem.
`samplerCubeArrayShadow`	Uchwyt dostępu do tablicy tekstur sześciennych głębi z porównywaniem.
`isampler1D` `iimaga1D`	Uchwyt dostępu do jednowymiarowej tekstury całkowitej.

isampler2D iimage2D	Uchwyt dostępu do dwuwymiarowej tekstury całkowitej.
isampler3D iimage3D	Uchwyt dostępu do trójwymiarowej tekstury całkowitej.
isamplerCube iimageCube	Uchwyt dostępu do sześciennej tekstury całkowitej.
isampler2DRect iimage2DRect	Uchwyt dostępu do prostokątnej tekstury całkowitej.
isampler1DArray iimage1DArray	Uchwyt dostępu do tablicy jednowymiarowych tekstur całkowitych.
isampler2DArray iimage2DArray	Uchwyt dostępu do tablicy dwuwymiarowych tekstur całkowitych.
isamplerBuffer iimageBuffer	Uchwyt dostępu do tekstury buforowej całkowitej.
isampler2DMS iimage2DMS	Uchwyt dostępu do dwuwymiarowej tekstury wielopróbkowania całkowitej.
isampler2DMSArray iimage2DMSArray	Uchwyt dostępu do tablicy dwuwymiarowych tekstur wielopróbkowania całkowitych.
isamplerCubeArray iimageCubeArray	Uchwyt dostępu do tablicy sześciennych tekstur całkowitych.
atomic_uint	Uchwyt dostępu do całkowitego licznika atomowego bez znaku.
usampler1D uimage1D	Uchwyt dostępu do jednowymiarowej tekstury całkowitej bez znaku.
usampler2D uimage2D	Uchwyt dostępu do dwuwymiarowej tekstury całkowitej bez znaku.
usampler3D uimage3D	Uchwyt dostępu do trójwymiarowej tekstury całkowitej bez znaku.
usamplerCube uimageCube	Uchwyt dostępu do sześciennej tekstury całkowitej bez znaku.
usampler2DRect uimage2DRect	Uchwyt dostępu do prostokątnej tekstury całkowitej bez znaku.
usampler1DArray uimage1DArray	Uchwyt dostępu do tablicy jednowymiarowych tekstur całkowitych bez znaku.
usampler2DArray uimage2DArray	Uchwyt dostępu do tablicy dwuwymiarowych tekstur całkowitych bez znaku.
usamplerBuffer uimageBuffer	Uchwyt dostępu do tekstury buforowej całkowitej bez znaku.
usampler2DMS uimage2DMS	Uchwyt dostępu do dwuwymiarowej tekstury wielopróbkowania całkowitej bez znaku.
usampler2DMSArray uimage2DMSArray	Uchwyt dostępu do tablicy dwuwymiarowych tekstur wielopróbkowania całkowitych bez znaku.
usamplerCubeArray uimageCubeArray	Uchwyt dostępu do tablicy sześciennych tekstur całkowitych bez znaku.

Tabela 2 Podstawowe typy języka GLSL.

Automatyczna konwersja typów

W ściśle określonych sytuacjach wyrażenia w języku GLSL mogą być automatycznie (niejawnie) konwertowane do innego typu. Przedstawia to Tabela 3.

Typ wyrażenia	Typ dopuszczalnej automatycznej konwersji
int	uint
int, uint	float
int, uint, float	double
ivec2	uvec2
ivec3	uvec3
ivec4	uvec4

ivec2, uvec2	vec2
ivec3, uvec3	vec3
ivec4, uvec4	vec4
ivec2, uvec2, vec2	dvec2
ivec3, uvec3, vec3	dvec3
ivec4, uvec4, vec4	dvec4
mat2	dmat2
mat3	dmat3
mat4	dmat4
mat2x3	dmat2x3
mat2x4	dmat2x4
mat3x2	dmat3x2
mat3x4	dmat3x4
mat4x2	dmat4x2
mat4x3	dmat4x3

Tabela 3 Zestawienie dopuszczalnych automatycznych konwersji typów.

Podczas wykonywania niejawnej konwersji typów dla operatorów binarnych może wystąpić wiele typów danych, do których operandy mogą być skonwertowane. Przykładowo podczas dodawania liczby typu `int` do liczby typu `uint`, obie wartości mogą być skonwertowane do liczby typu `uint`, `float` lub `double`. W takich sytuacjach typ zmiennoprzecinkowy wybierany jest wtedy, gdy którykolwiek z operandów jest typu zmiennoprzecinkowego. W przeciwnym wypadku, jeżeli któryś z operandów jest typu całkowitego bez znaku, do konwersji wybierana jest liczba całkowita bez znaku. W pozostałych przypadkach wybierana jest liczba całkowita ze znakiem. Jeżeli argumenty wyrażenia mogą być niejawnie skonwertowane do wielu typów danych, pochodzących z tego samego typu bazowego, do konwersji używany jest typ o najmniejszym rozmiarze składowej.

Nie ma możliwości bezpośredniej konwersji tablic lub struktur. Przykładowo tablica liczb całkowitych `int` nie może być automatycznie skonwertowana do tablicy liczb zmiennoprzecinkowych typu `float`.

Konstruktory

Konstruktory służą do inicjowania wartości zmiennych 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 konstruktory dostępne dla typów skalarnych:

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

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

W przypadku konwersji z typów zmiennoprzecinkowych `float` lub `double` do typu `int` lub `uint` część zmiennoprzecinkowa jest odrzucana. Nie jest zdefiniowana konwersja ujemnej liczby zmiennoprzecinkowej do typu `uint`.

Przy konwersji typów całkowitych i zmiennoprzecinkowych do typu `bool`, wartości 0 i 0,0 konwertowane są na `false`, a wartości niezerowe konwertowane są na `true`. Konwersja w drugą stronę realizowana jest w ten sposób, że wartość `false` jest zamieniana na 0 lub 0,0, a wartość `true` na 1 lub 1,0.

Konstruktor `int( uint )` zachowuje wzorzec bitów w argumencie, co zmieni jego wartość, jeśli argument ma ustawiony bit znaku. Konstruktor `uint( int )` zachowuje wzorzec bitów w argumencie, co zmieni jego wartość, jeśli argument jest ujemny.

Konstruktorów skalarnych można także używać z typami nie skalarnymi. Konwersji podlega wówczas pierwszy element typu nie skalarnego, np. `float( vec3 )` wybierze do konwersji pierwszą składową wektora `vec3`.

Możliwe jest także stosowanie konstruktorów identyczności, takich jak `float( float )`, ale sens ich użycia jest dyskusyjny.

Typ `void`

Typ `void` jest przeznaczony do użycia dla funkcji nie zwracających żadnej wartości. Język GLSL nie ma domyślnego typu zwracanego przez funkcje. Słowo zarezerwowane `void` nie może być używane w innych deklaracjach (za wyjątkiem pustej listy argumentów).

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 zawiera poniższy przykład:

```
bool success;         // deklaracja „success” jako typu bool
bool done = false;    // deklaracja i inicjalizacja „done”
```

Prawa strona operatora przypisania „=” musi być wyrażeniem, którego typ jest `bool`. Wyrażenia używane w warunkach pętli i selekcji (`if`, `for`, `?:`, `while`, `do-while`) muszą być przekształcalne do typu `bool`.

Typy całkowite

Język GLSL w pełni wspiera liczby całkowite ze znakiem i bez znaku. Liczby całkowite bez znaku mają 32-bitową precyzję. Liczby całkowite ze znakiem także używają 32-bitowej reprezentacji, ale w tym zawiera się bit znaku, a sama liczba zapisana jest w kodzie uzupełnień do 2 (tzw. U2). Nadmiar i niedomiar podczas operacji na liczbach całkowitych nie generuje żadnego wyjątku.

Zmienne typów całkowitych mogą być inicjalizowane za pomocą literałów o podstawie dziesiętnej, ósemkowej i szesnastkowej. Liczba ósemkowa rozpoczyna się cyfrą `0`, a liczba szesnastkowa stałą „`0x`” lub „`0X`”. W liczbach szesnastkowych dopuszczalne są zarówno duże jak i małe litery. Liczby bez znaku zawierają końcówkę „`u`” lub „`U`”. Znak minus „–” w literale oznacza unarną negację arytmetyczną i nie jest częścią stałej. Błędne są literały całkowite, które nie mieszczą się w 32-bitowej liczbie.

Poniżej przedstawiamy przykładowe deklaracje i inicjalizacje zmiennych typów całkowitych:

```
int i, j = 42;        // deklaracja i inicjalizacja typu int
```

```
uint k = 3u;           // „u” oznacza, że typ jest uint

int a = 0xffffffff;    // liczba 32-bitowa o wartości równej -1
int b = 0xffffffffU;   // błąd, brak możliwości konwersji
                       // z uint do int
uint c = 0xffffffff;   // liczba 32-bitowa o wartości 0xFFFFFFFF
uint d = 0xffffffffU;  // liczba 32-bitowa o wartości 0xFFFFFFFF

int e = -1;            // literałem całkowitym jest "1", następnie
                       // wykonywana jest negacja i wynik jest
                       // 32-bitową liczbą całkowitą ze znakiem
                       // o wzorcu bitowym 0xFFFFFFFF, czyli liczbą
                       // w kodzie uzupełnień do 2 równą -1
uint f = -1u;          // literałem jest "1u", następnie wykonywana
                       // jest negacja i wynik jest 32-bitową liczbą
                       // całkowitą bez znaku o wzorcu bitowym
                       // 0xFFFFFFFF, zatem f przyjmuje wartość
                       // równą 0xFFFFFFFF

int g = 3000000000;    // przypisywany jest 32-bitowy literał
                       // całkowity wraz z bitem znaku, stąd
                       // g przyjmuje wartość równą -1294967296
int h = 0xA0000000;    // poprawnie, 32-bitowa liczba całkowita
                       // ze znakiem (heksadecymalnie)
int i = 5000000000;    // błąd, liczba wymaga więcej niż 32 bity
int j = 0xFFFFFFFFF;   // błąd, liczba wymaga więcej niż 32 bity
int k = 0x80000000;    // wartość k wynosi -2147483648 == 0x80000000
int l = 2147483648;    // wartość l wynosi -2147483648 (literał
                       // ustawia także bit znaku)
```

Typy `float` i `double`

Typy `float` i `double` w języku GLSL to liczby zmiennoprzecinkowe o odpowiednio pojedynczej i podwójnej precyzji, zgodne ze standardem IEEE 754. GLSL akceptuje liczby w formacie IEEE 754, przy czym wewnętrzna reprezentacja nie musi być zgodna z tym formatem. Przy inicjalizacji można wykorzystać postać wykładniczą liczby oraz opcjonalny przyrostek `f` lub `F` (liczby o pojedynczej precyzji) względnie przedrostek `lf` lub `LF` (liczby o podwójnej precyzji). Znak minus „–” w literale oznacza operator unarny i nie jest częścią stałej. Poniżej przykłady deklaracji i inicjalizacji liczb typu `float`:

```
float a,             // deklaracja
      b = 1.5;       // deklaracja i inicjalizacja
double c,            // deklaracja
       d = 2.0LF;    // deklaracja i inicjalizacja
```

Typy wektorowe

Typy wektorowe w GLSL obejmują wektory dwu-, trzy- i czterowymiarowe z elementami typu `bool`, `int`, `uint`, `float` i `double`. Typowo typy wektorowe przechowują składowe kolorów, współrzędne wierzchołków, tekstur itp. Wektory z elementami typu bool używane są natomiast do porównywania składowych innych wektorów. Oto przykładowe deklaracje typów wektorowych:

```
vec2 texcoord1, texcoord2;
vec3 position;
```

```
vec4 myRGBA;
ivec2 textureLookup;
bvec3 less;
```

Konstruktory wektorów umożliwiają utworzenie wektora ze zbioru danych skalarnych oraz innych wektorów i macierzy, włączając w to wektory o mniejszej liczbie składowych. Pojedynczy element skalarny konstruktora inicjalizuje wszystkie składowe wektora do wartości skalara. W przypadku użycia w konstruktorze różnych argumentów: skalarów i wektorów, będą one wykorzystywane do inicjalizacji składowych wektora w kolejności od lewej do prawej. W razie niezgodności typu danych macierzy z typami argumentów konstruktora dokonywana jest niejawna konwersja zgodnie z opisanymi wcześniej regułami. Oto wybrane konstruktory typów wektorowych i kilka przykładów ich użycia:

```
vec3( float ) // inicjalizacja wszystkich składowych
              // vec3 wartością float
vec4( ivec4 ) // utworzenie vec4 poprzez konwersję każdej składowej
vec4( mat2 )  // vec4 składa się z kolumny 0 i następnie
              // kolumny 1 macierzy mat2
vec2( float, float )   // inicjalizacja vec2 z dwóch liczb float
ivec3( int, int, int ) // inicjalizacja ivec3 z trzech liczb int
bvec4( int, int, float, float ) // użyte 4 konwersje do typu bool
vec2( vec3 )  // pomijamy trzecią składową vec3
vec3( vec4 )  // pomijamy czwartą składową 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( float, vec3 )
vec4( vec2, vec2 )

vec4 color = vec4( 0.0, 1.0, 0.0, 1.0 );
vec4 rgba = vec4( 1.0 );  // ustawia każdą składową na 1.0
vec3 rgb = vec3( color ); // pomijamy czwartą składową
```

Dostęp do elementów wektora możliwy jest na dwa sposoby. Pierwszym z nich jest potraktowanie wektora jak typu tablicowego i użycie operatora „`[]`”, przy czym składowe wektora numerowane są od 0, a do indeksowania można użyć wyłącznie stałego wyrażenia całkowitoliczbowego. Drugą metodą jest użycie selektora „`.`”, czyli potraktowane wektora w sposób analogiczny do struktur z następującymi nazwami pól: { `x`, `y`, `z`, `w` }, { `r`, `g`, `b`, `a` } lub { `s`, `t`, `p`, `q` }. Nazwy te ułatwiają korzystanie ze zmiennych wektorowych reprezentujących różnego rodzaju dane: współrzędne wektorów (punktów), składowe kolorów czy współrzędne tekstur. W przypadku tych ostatnich dokonano jedynie zamiany współrzędnej *r* tekstury na *p* z uwagi na konflikt oznaczenia ze składową *r* koloru.

Dostęp do składowej, która nie występuje w danym typie wektora jest oczywiście błędem. Przy okazji warto zwrócić uwagę, że typy skalarne można traktować jak jednoelementowe wektory:

```
vec2 pos;
float height;
pos.x   // poprawnie
~~pos.z~~   // błąd
height.x   // poprawnie
~~height.y~~   // błąd
```

Analogiczne ograniczenia dotyczą dostępu za pomocą operatora tablicowego „[]”. Ujemny indeks lub przekroczenie wielkości wektora jest błędem. Operatora tablicowego nie można jednak używać do typów skalarnych.

Opisane wyżej pola można dodatkowo łączyć (wyłącznie w zakresie danej grupy) uzyskując inne wektory lub skalary. Jest to odstępstwo od zasad obowiązujących w dostępnie do pól struktur, ale ułatwia operacje na wektorach. Oto kilka przykładów:

```
vec4 v4;
v4.rgba; // otrzymujemy vec4, czyli tak samo jak używając v4
v4.rgb;  // otrzymujemy vec3
v4.b;    // otrzymujemy float
v4.xy;   // otrzymujemy vec2
v4.xgba; // błąd – nazwy składowych pochodzą z różnych zbiorów
```

Nazwy składowych nie muszą występować ani kolejno, ani pojedynczo. Daje to bardzo obszerną ilość możliwych kombinacji np. przestawień i duplikacji, z których część przedstawiają poniższe przykłady:

```
vec4 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 )
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.xx = vec2( 3.0, 4.0 );  // błąd – przypisanie do x wektora
pos.xy = vec3( 1.0, 2.0, 3.0 );  // błąd – niezgodność pomiędzy
                                 // vec2 i vec3
float f = 1.2;
dup = f.xxxx;       // dup = ( 1.2, 1.2, 1.2, 1.2 )
```

Ostatnią jeszcze nieopisaną funkcjonalnością wektorów jest funkcja `length`, która zwraca rozmiar wektora w postaci liczby typu `int` i jest dostępna jako funkcja składowa wektora:

```
vec3 v
const int L = v.length();   // zwracana wartość 3
```

Funkcja `length` nie może być użyta do typów skalarnych.

Typy macierzowe

GLSL ma wbudowane typy macierzowe o rozmiarach 2×2, 2×3, 2×4, 3×2, 3×3, 3×4, 4×2, 4×3 i 4×4, które zawierają elementy typu `float` lub typu `double`. W nazwie typu `matmxn`, `m` oznacza ilość kolumn, a `n` ilość wierszy macierzy. Oto przykładowe deklaracje macierzy:

```
mat2 mat2D;
mat3 optMatrix;
mat4 view, projection;
mat4x4 view;  // alternatywny sposób zadeklarowania mat4
mat3x2 m;     // macierz z 3 kolumnami i 2 wierszami
```

Podobnie jak w przypadku wektorów inicjalizacja macierzy wykonywana jest za pomocą konstruktorów zawierających skalary, wektory lub macierze przy zachowaniu kolumnowego porządku inicjalizacji elementów. Nietypowo funkcjonuje jedynie konstruktor macierzy kwadratowej z pojedynczym argumentem typu `float` lub innym typu skalarnego. Tak konstruowana macierz

wypełniana jest bowiem wartością przekazanego typu tylko na głównej przekątnej, pozostałe elementy przyjmują wartość równą 0.0. Oto przykłady

```
mat2( float )  // macierz wynikowa result[i][j] przyjmuje
mat3( float )  // wartość argumentu float dla każdego i = j
mat4( float )  // oraz wartość 0,0 dla każdego i != j
```

Macierze możemy konstruować z niemal dowolnej kombinacji skalarów i wektorów, przy czym najważniejsza jest zasada kolumnowej kolejności inicjalizacji. Argumenty konstruktora wykorzystywane do inicjalizacji elementów macierzy pobierane są w kolejności od lewej do prawej. Konstruktor musi posiadać taką ilość elementów, która odpowiada ilości elementów konstruowanej macierzy. W razie niezgodności typu danych macierzy z typami argumentów konstruktora dokonywana jest niejawna konwersja zgodnie z opisanymi wcześniej regułami. Oto przykłady konstruktorów macierzy z wektorów i skalarów:

```
mat2( vec2, vec2 )             // jedna kolumna na argument
mat3( vec3, vec3, vec3 )       // jedna kolumna na argument
mat4( vec4, vec4, vec4, vec4 ) // jedna kolumna na argument
mat3x2( vec2, vec2, vec2 )     // jedna kolumna na argument

dmat2( dvec2, dvec2 )                 // jedna kolumna na argument
dmat3( dvec3, dvec3, dvec3 )          // jedna kolumna na argument
dmat4( dvec4, dvec4, dvec4, dvec4 )   // jedna kolumna na argument

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
dmat2x4( dvec3, double,   // pierwsza kolumna
         double, dvec3 )  // druga kolumna
```

Poniżej przedstawiamy trzy równoważne deklaracje macierzy 2×2:

```
mat2x2 a = mat2( vec2( 1.0, 0.0 ), vec2( 0.0, 1.0 ) );
mat2x2 b = mat2( vec2( 1.0, 0.0 ), 0.0, 1.0 );
mat2x2 c = mat2( 1.0, 0.0, 0.0, 1.0 );
```

Nieco inne zasady obowiązują wówczas, gdy argumentem konstruktora macierzy jest inna macierz. W takim przypadku każdy element macierzy wynikowej jest równy elementowi macierzy wejściowej znajdującemu się w analogicznym położeniu (wierszu i kolumnie). Elementy macierzy wynikowej, które nie są w ten sposób zainicjowane (ma to miejsce w przypadku różnic w rozmiarach macierzy wejściowej i wynikowej) uzupełniane są do macierzy jednostkowej. W przypadku konstruowania macierzy na podstawie innej macierzy GLSL dopuszcza tylko jeden argument konstruktora. Oto kilka przykładów:

```
mat3x3( mat4x4 )  // inicjalizacja na podstawie lewej górnej
```

```
                    // części 3x3 macierzy 4x4
mat2x3( mat4x2 )    // inicjalizacja na podstawie lewej górnej
                    // części 2x2 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 1,0, a pozostałe wartość 0,0
```

Dostęp do elementów macierzy umożliwia podwójny „`[][]`” oraz pojedynczy „`[]`” operator tablicowy. W przypadku pojedynczego operatora tablicowego otrzymujemy dostęp do wybranej kolumny macierzy, która jest traktowana jak wektor o rozmiarze odpowiadającym liczbie wierszy macierzy. Kolumny macierzy numerowane są od 0. Podwójny operator tablicowy wybiera pojedynczy element macierzy na podstawie kolejno numeru kolumny i wiersza. Numeracja indeksów kolumn i wierszy macierzy zaczyna się od 0. Przekroczenie zakresu macierzy powoduje zgłoszenie błędu. Oto kilka przykładów dostępu do elementów macierzy:

```
mat4 m;
m[1] = vec4( 2.0 ); // ustawia drugą kolumnę macierzy
                    // na wartości równe 2,0
m[0][0] = 1.0;      // ustawia lewy górny element macierzy na 1,0
m[2][3] = 2.0;   // ustawia czwarty element trzeciej kolumny na 2,0
```

Ostatnią jeszcze nieopisaną funkcjonalnością macierzy jest funkcja `length`, która zwraca ilość kolumn macierzy w postaci liczby typu `int` i jest dostępna jako funkcja składowa macierzy:

```
mat3x4 v
const int L = v.length();   // zwracana wartość 3
```

Uchwyty tekstur

Uchwyty tekstur (np. `sampler2D`) używane są głównie jako parametry funkcji próbkujących tekstury oraz zmienne jednorodne umożliwiające dostęp do wybranych tekstur. Uchwyty tekstur nie mogą być parametrami wyjściowymi (`out`) ani wejściowo-wyjściowymi (`inout`) funkcji. Nie mogą także stanowić operandów w wyrażeniach, za wyjątkiem indeksowania tablic, dostępu do struktury oraz operatora nawiasowego.

Przegląd wszystkich funkcji próbkujących tekstury znajduje się w dalszej części niniejszego tekstu.

Obrazy tekstur

Obrazy tekstur (np. `image2D`) umożliwiają dostęp do zapisu i odczytu oraz operacji atomowych na danych tekseli tekstur na wybranym poziomie mipmapy. Zmienne obrazów tekstury mogą być agregowane w tablicach indeksowanych przez dynamiczne jednorodne wyrażenia całkowite. Przegląd funkcji operujących na obrazach tekstur znajduje się w dalszej części niniejszego odcinka kursu.

Liczniki atomowe

Liczniki atomowe (`atomic_uint`) używane są do obsługi funkcji realizujących operacje atomowe, które jak pozostałe funkcje wbudowane, opisane są w dalszej części niniejszego tekstu.

Struktury

Struktury służą do łączenia innych zdefiniowanych wcześniej typów w jedną całość przy użyciu słowa zarezerwowanego `struct`. Oto definicja przykładowej struktury i deklaracje dwóch zmiennych typu strukturalnego:

```
struct light            // nazwa struktury
{
    float intensity;    // pierwsza składowa struktury
    vec3 position;      // druga składowa struktury
} lightVar;             // opcjonalna deklaracja zmiennej typu light

light lightVar2;        // deklaracja drugiej zmiennej typu light
                        // (bez użycie słowa zarezerwowanego struct)
```

Struktura musi posiadać co najmniej jedną składową. Każda składowa musi być wcześniej zdefiniowanym typem, w szczególności nie są obsługiwane odwołania do niezdefiniowanych struktur. Składowa może posiadać opcjonalny kwalifikator precyzji, przy czym inne kwalifikatory nie są dopuszczalne. Wśród składowych mogą znajdować się oprócz typów podstawowych także tablice o określonym i większym od zera rozmiarze, a także inne struktury. Definicja struktury nie może zawierać wewnętrznych definicji innych struktur oraz struktur anonimowych. Oto poprawne i błędne przykłady definiowania struktur:

```
struct S
{
   float f;
};

struct T
{
    S;              // błąd, anonimowe struktury nie są obsługiwane
    struct { ... }; // błąd, zagnieżdżone struktury
                    // nie są obsługiwane
    S s;            // poprawnie, wewnętrzna nazwana struktura
                    // jest obsługiwana
};
```

Struktury mogą być inicjalizowane przy użyciu konstruktorów, które nazywają się tak samo jak nazwa struktury. Argumenty konstruktora struktury zawierają wartości kolejnych jej pól lub konstruktory pól, oddzielone przecinkami. Wartości przekazywane do konstruktora powinny być tego samego typu jak pola struktur, ale akceptowane są także typy podlegające automatycznej konwersji. Konstruktory struktur mogą być używane przy inicjalizacji struktury oraz w wyrażeniach. Oto przykładowy konstruktor zdefiniowanej wyżej struktury `light`:

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

Dostęp do poszczególnych elementów struktury, podobnie jak w językach C i C++, zapewnia operator „`.`”.

Tablice

Tablice zawierają zmienne tego samego typu i są definiowane z nazwą poprzedzającą nawiasy kwadratowe „`[]`” zawierające opcjonalny rozmiar tablicy. Elementami tablicy mogą być wszystkie typy proste oraz struktury. GLSL umożliwia deklarowanie wyłącznie tablic jednowymiarowych. Jeżeli rozmiar tablicy jest określony w jej deklaracji, musi to być stałe wyrażenie całkowitoliczbowe o

wartości większej od zera. Jeżeli tablica jest indeksowana przy użyciu wyrażenia niespełniającego powyższego warunku, lub gdy tablica jest przekazywana jak argument funkcji, jej rozmiar musi być zadeklarowanych przed pierwszym użyciem. Można w tym celu zadeklarować tablicę bez rozmiaru, a następnie redeklarować ją (w tym samym shaderze) z taką samą nazwą i takim samym typem elementów określając jednocześnie jej wielkość. Nie jest dopuszczalne przekraczanie zakresu wielkości tablicy ani tez indeksowanie elementów przy użyciu ujemnych indeksów. Tablice zadeklarowane jako parametry formalne w deklaracji funkcji muszą mieć określony rozmiar. Przekroczenie zakresu tablicy powoduje niezdefiniowane zachowanie i może zakończyć się nawet przerwaniem działania programu.

Poniżej znajdują się liczne przykłady definicji typów tablicowych oraz typowych błędów przy ich definiowaniu:

```
float frequencies[3];
uniform vec4 lightPosition[4];
light lights[];
const int numLights = 2;
light lights[numLights];

~~float a[5][3];~~  // błąd, nie można deklarować tablicy
~~float[5] a[3];~~  // dwuwymiarowej oraz tablicy tablic
float a[5];
float b[] = a;  // b ma jawnie wielkość określoną na 5
float b[5] = a; // taki sam efekt jak powyżej
```

Tablica może być wartością zwracaną przez funkcję, może także stanowić nazwany lub nienazwany parametr w definicji funkcji:

```
float[5] foo()
{
}
void foo( float[5] );
void foo( float[5] a );
```

Konstruktory tablic mają podobną budowę jak konstruktory wcześniej opisanych typów, z tą różnicą, że umieszczone w nawiasach klamrowych argumenty konstruktora poprzedza operator tablicowy z opcjonalnym rozmiarem tablicy. Konstruktor musi posiadać dokładnie taką samą ilość parametrów jak rozmiar tablicy. Jeżeli rozmiar tablicy nie jest określony wprost, tablica ma rozmiar równy ilości argumentów konstruktora. Argumenty konstruktora są przypisywane kolejno, zaczynając od zerowego indeksu tablicy. Oczywiście każdy argument konstruktora musi być tego samego typu jak elementy tablicy, lub musi być dopuszczalna domyślna konwersja do tego typu. Oto kilka przykładów konstruktorów tablic:

```
float a[5] = float[5]( 3.4, 4.2, 5.0, 5.2, 1.1 );
float a[5] = float[]( 3.4, 4.2, 5.0, 5.2, 1.1 ); // to samo co wyżej
const float c[3] = float[3]( 5.0, 7.2, 1.1 );
const float d[3] = float[]( 5.0, 7.2, 1.1 );

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 );
```

Ostatnią jeszcze nieopisaną funkcjonalnością tablic jest funkcja `length`, która zwraca rozmiar tablicy w postaci liczby typu `int` i jest dostępna jako funkcja składowa tablicy:

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

Funkcja `length` nie może być użyta do tablic o nieokreślonym jawnie rozmiarze.

Operatory i wyrażenia

Operatory dostępne w języku GLSL przedstawiono w Tabela 4 w kolejności od najwyższego (1) do najniższego (17) priorytetu. Zauważmy, że język GLSL nie zawiera żadnych operatorów adresowych i wskaźnikowych. Natomiast zamiast operatorów rzutowania GLSL używa opisanych wcześniej konstruktorów.

Priorytet	Rodzaj	Operator	Kolejność wiązania
1 (najwyższy)	grupowanie w nawiasy	`( )`	-
2	selektory elementów tablic i składowych struktur, wywołanie funkcji, konstruktor, postinkrementacja, postdekrementacja	`[] . () ++ --`	od lewej do prawej
3	preinkrementacja, predekrementacja, jednoargumentowe	`++ -- + - ! ~`	od prawej do lewej
4	multiplikatywne	`* / %`	od lewej do prawej
5	addytywne	`+ -`	od lewej do prawej
6	przesuwanie bitowe	`<< >>`	od lewej do prawej
7	relacje	`< > <= >=`	od lewej do prawej
8	równość	`== !=`	od lewej do prawej
9	bitowe AND	`&`	od lewej do prawej
10	różnica symetryczna XOR	`^`	od lewej do prawej
11	bitowe OR	`|`	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	`= += -= *= /= %= <<= >>= &= ^= |=`	od prawej do lewej
17 (najniższy)	sekwencja	`,`	od lewej do prawej

Tabela 4 Operatory języka GLSL

Operacje na strukturach i tablicach

Operacje na strukturach, poza już wspomnianym operatorem dostępu do pola struktury „`.`”, mogą wykonywać następujące operatory: „==”, „`!=`” (porównanie) oraz „=” (przypisanie). Taki sam zestaw operatorów jest dopuszczalny także dla tablic, które dodatkowo obsługuje omówiony już operator dostępu do elementu „`[]`”. W przypadku tablic operator „`.`”dotyczy jedynie wywołania funkcji `length`. Operatory przypisania i porównania do poprawnego działania wymagają zgodności rozmiaru i typów obu operandów.

Przypisania

Przypisanie wartości do zmiennej jest wykonywane przy użyciu operatora przypisania „=”:

```
lvalue-expression = rvalue-expression
```

Oba operandy (`lvalue-expression` i `rvalue-expression`) operatora przypisania muszą mieć te same typy lub podlegać automatycznej konwersji typów. Każda inna konwersja musi zostać zrealizowana przy pomocy odpowiedniego konstruktora. `lvalue-expression` musi umożliwiać zapis, stąd może nią być np. wbudowany typ, struktura i tablica, pole struktury lub element tablicy.

Inne operatory przypisania (`+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `|=` i `^=`) można opisać ogólnym wyrażeniem:

```
lvalue op= expression
```

które jest odpowiednikiem:

```
lvalue = lvalue op expression
```

gdzie `op` jest odpowiednią operacją wyspecyfikowaną w symbolu operatora, a wyrażenia po obu stronach muszą spełniać opisane wyżej wymogi dla operatora „=".

GLSL dopuszcza odczyt zmiennych przez ich zapisem lub inicjalizacją, jednak ich wartość pozostaje nieokreślona.

Wyrażenia

Wyrażenia w języku GLSL zbudowane są z następujących elementów:

- stałych typu `bool`, `int`, `uint`, `float`, `double` oraz wszystkich typów wektorowych i macierzowych,
- konstruktorów wszystkich typów,
- nazw zmiennych wszystkich typów,
- nazwy tablicy z wyspecyfikowaną metodą `length`,
- indeksowane nazwy tablic,
- wywołania funkcji, które zwracają wartość,
- selektory pól składowych oraz wynik indeksowania tablicy,
- wyrażenie nawiasowe; każde wyrażenie może zostać ujęte w nawiasy, co umożliwia grupowanie operacji; operacje w nawiasach wykonywane są przed operacjami poza nawiasem,
- binarne operatory arytmetyczne `+`, `-`, `*` i `/`, działające na całkowitych lub zmiennoprzecinkowych skalarach, wektorach i macierzach; jeżeli podstawowe typy operandów nie są zgodne stosowane są reguły niejawnej konwersji typów; poprawność operacji i zwracany typ, z uwzględnieniem ewentualnej automatycznej konwersji, określają następujące przypadki (pozostałe są niedozwolone):
  - dwa operandy są skalarami; w takim przypadku wynik także jest skalarem,
  - jeden operand jest skalarem, a inny wektorem lub macierzą; w takim przypadku operacja skalarna jest wykonywana niezależnie na każdej składowej wektora lub macierzy dając w rezultacie tej samej wielkości wektor lub macierz,
  - dwa operandy są wektorami o tym samym rozmiarze; w takim przypadku operacja jest wykonywana na kolejnych elementach wektorów, a wynikiem jest wektor o tej samej wielkości,
  - operatorem jest `+` (dodawanie), `-` (odejmowanie), lub `/` (dzielenie), a operandami są macierze o tych samych wymiarach, w takim przypadku operacja jest wykonywana na kolejnych elementach macierzy, a wynikiem jest macierz o tych samych wymiarach,
  - operatorem jest `*` (mnożenie), gdzie oba operandy są macierzami lub jeden operand jest wektorem a drugi macierzą; prawy operand wektorowy jest traktowany jako wektor kolumnowy, a lewe operand wektorowy jako wektor wierszowy; w tych

wszystkich przypadkach wymagana jest zgodność ilości kolumn lewego operandu z ilością wierszy prawego operandu; operator * oznacza wówczas mnożenie algebraiczne, a wynik jest obiektem o takiej ilości wierszy jak lewy operand i takiej samej ilości kolumn jak prawy operand.

Dzielenie przez zero nie zwraca żadnego wyjątku, ale wynik operacji jest nieokreślony. Algebraiczne operacje na wektorach i macierzach realizują ponadto następujące funkcje wbudowane: `dot` (iloczyn skalarny), `cross` (iloczyn wektorowy), `matrixCompMult` (mnożenie macierzy po elementach) i `outerProduct` (mnożenie wektorów traktowanych jak macierze, które generuje macierz). Te i inne funkcje wbudowane opisane są w dalszej części niniejszego tekstu.

- operator `%` (modulo) działa na liczbach całkowitych ze znakiem i bez znaku oraz na wektorach z liczbami całkowitymi ze znakiem i bez znaku i nie jest zdefiniowany dla innych typów danych; jeżeli typy danych składowych operandów są różne stosowane są zasady automatycznej konwersji danych; operandy nie mogą być wektorami o różnych rozmiarach; jeżeli jeden operand jest skalarem, a drugi wektorem, operacja jest wykonywana przy użyciu skalara na każdej składowej wektora zwracając wektor o tej samej wielkości; jeżeli oba operandy są wektorami o tym samym rozmiarze, wynik jest obliczany dla każdej pary elementów wektorów; wynik jest niezdefiniowany, gdy jeden lub oba operandy są ujemne,
- unarne operatory arytmetyczne − (negacja), −− (postdekrementacja/predekrementacja), ++ (postinkrementacja/preinkrementacja) działają na liczbach całkowitych i zmiennoprzecinkowych, włączając w to wektory i macierze; wszystkie operatory unarne działają na składowych operandów; wynik jest tego samego typu jak operand; dekrementacja i inkrementacja odejmuje lub dodaje wartość 1 lub 1,0, a sama operacja wykonywana jest odpowiednio przed albo po wykonaniu wyrażenia,
- operatory relacyjne > (większy), < (mniejszy), >= (większy lub równy) i <= (mniejszy lub równy) działają wyłącznie na skalarnych wyrażeniach całkowitych i zmiennoprzecinkowych; wynikiem jest skalar typu `bool`; typy operandów muszą być zgodne, przy czym uwzględniana jest domyślna konwersja typów; do porównywania składowych wektorów służą wbudowane funkcje wbudowane: `lessThan`, `lessThanEqual`, `greaterThan` i `greaterThanEqual`,
- operatory równości == (równy) i `!=` (nierówny) działają na wszystkich typach, a rezultatem jest skalar typu `bool`; dla wektorów, macierzy, struktur i tablic operatory działają na wszystkich składowych, polach i elementach; w przypadku niezgodności operandów stosowana jest domyślna konwersja typów; do porównania składowych wektorów służą wbudowane funkcje `equal` i `notEqual`, które zwracają wektor,
- binarne operatory logiczne `&&` (logiczne AND), `||` (logiczne OR) i `^^` (logiczna różnica symetryczna XOR) działają tylko na dwóch wyrażeniach typu `bool`, a wynik jest także wyrażeniem typu `bool`; operator `&&` oblicza prawy operand (wyrażenie) tylko wtedy, gdy lewy operand ma wartość `true`; operator `||` oblicza prawy operand (wyrażenie) tylko wtedy, gdy lewy operand ma wartość `false`; operator `^^` zawsze oblicza oba operandy,
- logiczny operator unarny `!` (logiczna negacja NOT) działa tylko na wyrażeniach typu `bool` i zwraca także wyrażenie typu `bool`; do operacji negacji na wektorach służy wbudowana funkcja `not`,
- operator sekwencji **,** działa na liście wyrażeń oddzielonych od siebie przecinkami, przy czym wszystkie wyrażenia z listy obliczane są w kolejności od lewej do prawej,
- trójelementowy operator selekcji **? :** działa na trzech wyrażeniach (`exp1 ? exp2 : exp3`); operator oblicza pierwsze wyrażenie, które musi zwrócić wartość typu `bool`; jeżeli jego wynik jest równy `true` operator wybiera do wykonania drugie wyrażenie, a w przeciwnym wypadku wykonywane jest trzecie wyrażenie (zawsze wykonywane jest tylko jedno z tych dwóch wyrażeń); drugie i trzecie wyrażenie mogą być dowolnego typu, ale musi to być ten sam typ, oczywiście z uwzględnieniem ewentualnej automatycznej konwersji,

- operator negacji bitowej ~ działa na liczbach całkowitych ze znakiem i bez znaku lub wektorach całkowitych, a wynik jest dopełnieniem bitowym do jedynki każdego bitu operandu włączając w to bit znaku; operacja wykonywana jest na każdej składowej operandu,
- operatory przesunięcia bitowego << i >> działają na liczbach całkowitych ze znakiem i bez znaku lub wektorach całkowitych; jeden operand może być liczbą ze znakiem, a drugi liczbą bez znaku, przy czym wynik jest zawsze tego samego typu jak lewy operand; jeżeli pierwszy operand jest skalarem, to drugi także musi być skalarem; jeżeli pierwszy operand jest wektorem, to drugi operand może być skalarem lub wektorem, a wynik jest obliczany dla każdej składowej; wynik operacji jest nieokreślony, gdy prawy operand jest ujemny względnie większy lub równy liczbie bitów typu bazowego lewego wyrażenia; wartość `E1 << E2` jest równa `E1` (interpretowanemu jako wzorzec bitowy) po przesunięciu o `E2` bitów w lewo; analogicznie wartość `E1 >> E2` jest równa `E1` po przesunięciu o `E2` bitów w prawo; jeżeli `E1` jest liczbą całkowitą ze znakiem, przesunięcie bitowe w prawo zachowuje bit znaku, natomiast jeżeli `E1` jest liczbą całkowitą bez znaku, przesunięcie bitowe w prawo dodaje zerowe bity,
- bitowe operatory logiczne `&` (bitowe AND), `|` (bitowe OR) i `^^` (bitowa różnica symetryczna XOR) działają na liczbach całkowitych ze znakiem i bez znaku lub wektorach całkowitych; operandy nie mogą być wektorami o różnej wielkości; jeżeli jeden operand jest skalarem, a drugi wektorem, to wynik jest obliczany dla każdej składowej wektora i zwracany jest wektor tego samego typu; typ podstawowy operandów (ze znakiem lub bez znaku) musi być zgodny, a wynik także jest tego samego typu podstawowego.

Pełny opis składni wyrażeń zawarty jest w specyfikacji języka GLSL i do lektury tego dokumentu zapraszamy Czytelników zainteresowanych bliżej tą tematyką.

Operacje na wektorach i macierzach

Poza niektórymi wyjątkami dotyczącym macierzy i wektorów operatory działają na wszystkich składowych zmiennej. Przykładowo:

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

jest odpowiednikiem:

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

natomiast:

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

odpowiada:

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

Podobnie funkcjonuje większość operatorów na wektorach liczb całkowitych i zmiennoprzecinkowych oraz typach macierzowych. Wyjątkiem są przypadki, gdy mnożymy macierz

przez wektor, wektor przez macierz oraz macierz przez macierz. Wykonywane są wówczas nie operacja na składowych, ale odpowiednie operacje algebraiczne:

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

jest odpowiednikiem:

```
u.x = dot( v, m[0] ); // m[0] jest lewą kolumną macierzy m
u.y = dot( v, m[1] );  // dot( a, b ) jest iloczynem skalarnym a i b
u.z = dot( v, m[2] );
```

Kolejne wyrażenie

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

jest odpowiednikiem:

```
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;
```

I jeszcze jedno wyrażenie:

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

które jest odpowiednikiem:

```
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;
```

Podobnie wygląda mnożenie innych macierzy i wektorów, przy czym należy pamiętać o spełnianiu przez operandy algebraicznych wymogów tej operacji, czyli zgodności odpowiednich rozmiarów.

Zakres widoczności zmiennych

Zakres widoczności zmiennej zależy od miejsca jej deklaracji. Zmienne zadeklarowane poza funkcjami mają zasięg globalny rozpoczynający się od miejsca deklaracji. Ponowna deklaracja zmiennej w danym zasięgu powoduje błąd kompilacji. Wyjątek dotyczy tablic o domyślnie określonym rozmiarze. Poniżej znajduje się kilka przykładów ilustrujących zasięg zmiennych:

```
int x = 1;
{
    int x = 2, y = x; // y jest inicjalizowane na 2
}
```

```
struct S
{
    int x;
};

{
    S S = S(0);  // "S" jest widoczne tylko jako struktura
                 // oraz jej konstruktor
    S;           // "S" jest teraz widoczne jako zmienna
}

int x = x;  // błąd, jeżeli x nie był wcześniej zdefiniowany
```

Zasięg przestrzeni nazw zmiennych globalnych obejmuje wszystkie jednostki kompilacji (obiekty shadera) danego rodzaju skonsolidowane w ramach jednego obiektu programu. GLSL dopuszcza w takich przypadkach współdzielenie pamięci zmiennych globalnych, których nazwa i typ są zgodne. W przypadku struktur wymagana jest zgodność nazwy struktury i zgodność typów danych poszczególnych składowych. Natomiast w przypadku tablic oprócz zgodności typu wymagana jest zgodność rozmiaru, przy czym rozmiar tablicy określony niejawnie w jednym shaderze, może być określony jawnie w drugim shaderze. Jeżeli żaden shader nie określa jawnie rozmiaru tablicy przyjmowana jest największa niejawnie określona jej wielkość. W przypadku, gdy współdzielona zmienna globalna ma wiele inicjalizacji, wszystkie inicjalizacje muszą być wyrażeniami stałymi o takiej samej wartości. Nieprawidłowości w zakresie współdzielonych zmiennych globalnych sygnalizowana są na etapie konsolidacji obiektu programu.

Kwalifikatory przechowywania

Deklaracje zmiennych mogą posiadać jeden z kwalifikatorów przechowywania wymienionych w Tabela 5. Kwalifikator przechowywania poprzedza w deklaracji typ zmiennej.

Kwalifikator przechowywania	Opis
brak (domyślnie)	Lokalny zapis/odczyt pamięci lub wejściowy parametr funkcji.
`const`	Zmienna, które wartość nie może być zmieniona.
`in`	Połączenie z shaderem z poprzedniego etapu potoku renderingu. Zmienna jest kopiowana i opcjonalnie poddawana interpolacji.
`out`	Połączenie z shaderem z następnego etapu potoku renderingu. Zmienna jest kopiowana i opcjonalnie poddawana interpolacji.
`uniform`	Wartości stałe w obiekcie programu w trakcie przetwarzania prymitywów; połączenie shadera z OpenGL i aplikacją.

Tabela 5 Kwalifikatory przechowywania zmiennych.

Niektóre zmienne shaderów z kwalifikatorami wejściowymi (`in`) i wyjściowymi (`out`) mogą być opcjonalnie modyfikowane przy użyciu pomocniczych kwalifikatorów przechowywania wymienionych w Tabela 6. Dopuszczalne kombinacje podstawowych i pomocniczych kwalifikatorów przechowywania zmiennych przedstawione są w dalszej części tekstu.

Pomocniczy kwalifikator przechowywania	Opis
`centroid`	Interpolacja wartości zmiennej przy użyciu centroidu.
`sample`	Interpolacja wartości zmiennej przy użyciu próbki.
`patch`	Atrybut w ścieżce teselacji.

Tabela 6 Pomocnicze kwalifikatory przechowywania zmiennych.

Zmienne lokalne mogą używać jedynie kwalifikatora `const` lub kwalifikatora domyślnego (pustego). Inicjalizacje w deklaracjach zmiennych globalnych dopuszczalne są jedynie w przypadku zmiennych z domyślnym kwalifikatorem przechowywania (pustym), kwalifikatorem `const` oraz `uniform`. Zauważmy, że zmienne globalne z domyślnym kwalifikatorem przechowywania, niezainicjowane w deklaracji, nie mają określonej wartości przez OpenGL.

Wartość zwracana przez funkcję oraz pola struktur nie mogą zawierać kwalifikatorów przechowywania. Inicjalizację można zastosować jedynie w zmiennych bez kwalifikatora przechowywania oraz w zmiennych z kwalifikatorem `const` lub `uniform`.

Przy porównywaniu zgodności zmiennych wyjściowych w shaderze ze zmiennymi wejściowymi w następnym programowalnym etapie renderingu, pod uwagę brane jest także zgodność pomocniczych kwalifikatorów przechowywania (lub ich brak).

Zastosowanie kwalifikatorów `const`, `in`, `out` i niewymienionego w tym miejscu `inout`, jako kwalifikatorów parametrów funkcji, zostanie opisane w dalszej części niniejszego odcinka kursu.

Domyślny kwalifikator przechowywania

Domyślnie zmienne w GLSL nie mają żadnego kwalifikatora przechowywania. Niekwalifikowane zmienne globalne i lokalne umożliwiają jedynie odczyt i zapis przydzielonego im obszaru pamięci.

Kwalifikator `const`

Kwalifikator `const` określa wartość stałą czasu kompilacji inicjalizowaną podczas deklaracji lub zmienną, która jest przeznaczona tylko do odczytu. Kwalifikator `const` może być użyty z każdym podstawowym typem danych (poza uchwytami tekstur, obrazami tekstur oraz licznikami atomowymi), włączają w to struktury i tablice struktur. Kwalifikator `const` musi być użyty przy deklaracji zmiennej połączonej z jej inicjalizacją. Oto przykłady:

```
const vec3 zAxis = vec3( 0.0, 0.0, 1.0 );
const float ceiling = a + b;    // zmienne a i b nie muszą
                                // koniecznie być stałymi
```

Inicjalizacja wartości stałej zmiennej globalnej może być dokonana wyłącznie przy użyciu wyrażenia stałego. Wyrażeniem takim może być:

- wartość literalna (np. 5 lub `true`),
- zmienna zadeklarowana z kwalifikatorem `const`, która jest zainicjowana wyrażeniem stałym,
- wyrażenie złożone z operatora działającego z operandami, które wszystkie są wyrażeniami stałymi, włączając w to pobranie elementu lub długości stałej tablicy, pobranie pola stałej struktury oraz pobranie składowej stałego wektora; do tej grupy nie jest zaliczany operator sekwencji (`,`) oraz operatory przypisania (=, +=, ...),
- konstruktory, których wszystkie argumenty są wyrażeniami stałymi,
- poprawnie użyta funkcja `length`, niezależnie od tego, czy związany z nią obiekt jest stały,
- wbudowane funkcje, wywołane wyłącznie z wyrażeniami stałymi, z wyjątkiem funkcji próbkujących tekstury i funkcji generujących szum; wbudowane funkcje `dFdx`, `dFdy` i `fwidth` zwracają 0, gdy są wywoływane w inicjalizacji z argumentami będącymi wyrażeniami stałymi; funkcje zdefiniowane przez użytkownika nie mogą być użyte do budowy wyrażenia stałego.

Zmienne wejściowe

Zmienne wejściowe shadera deklarowane są z kwalifikatorem przechowywania `in`. Zmienne takie oznaczają połączenie shadera z poprzednim etapem potoku renderingu OpenGL i muszą być zadeklarowane jako zmienne globalne. Zmienne wejściowe są zmiennymi tylko do odczytu, a ich zapis realizowany jest wyłącznie w poprzednim etapie renderingu. Użycie zmiennych wejściowych o

niezdefiniowanej wartości może, choć nie musi, być zgłoszone przez kompilator jak ostrzeżenie. Wbudowane zmienne wejściowe zostaną opisane w dalszej części tego odcinka kursu.

Zmienne wejściowe shadera wierzchołków zawierają dane atrybutów wierzchołka i deklarowane są z kwalifikatorem `in`. Użycie do takich zmiennych kwalifikatora interpolacji lub pomocniczego kwalifikatora przechowywania powoduje zgłoszenie błędu. Wartości atrybutów wierzchołków przekazywane są za pośrednictwem API OpenGL, które poznaliśmy już wcześniej, lub przy użyciu kwalifikatora formatu `location`. Atrybuty mogą być liczbami typu całkowitego lub zmiennoprzecinkowego, wektorami zmiennoprzecinkowymi i całkowitymi oraz macierzami. Atrybuty mogą być także tablicami zawierającymi powyższe typy danych, ale nie mogą być strukturami.

Poniżej znajdują się deklaracje przykładowych zmiennych wejściowych shadera wierzchołków:

```
in vec4 position;
in vec3 normal;
in vec2 texCoord[4];
```

Maksymalna ilość zmiennych wejściowych shadera wierzchołków zależy od implementacji biblioteki OpenGL. Ograniczenie jest obliczane przy pomocy tzw. indeksów w tablicy atrybutów ogólnych wierzchołka (pisaliśmy o tym w odcinku kursu poświęconemu wierzchołkom). Każdy atrybut skalarny zajmuje jeden indeks, podobnie jak atrybut wektorowy. Macierze zajmują ilość indeksów równą ilości kolumn. Ponieważ typy skalarne zajmują taką samą ilość indeksów jak wektory można zatem zastosować optymalizację polegającą na łączeniu różnych atrybutów wierzchołka w grupy zajmujące jeden lub więcej wektorów.

Zmienne wejściowe shadera kontroli teselacji, shadera ewaluacji teselacji i shadera geometrii są zmiennymi o nazwach takich samych jak zmienne wyjściowe zapisywane przez shader z poprzedniego aktywnego etapu potoku renderingu OpenGL. Różnica polega na tym, że wymienione shadery działają na zbiorze wierzchołków i ich atrybutów, stąd każda zmienna wejściowa musi być deklarowana jako tablica. Oto przykład:

```
in float foo[];  // zmienna wejściowa shadera geometrii dla zmiennej
                 // wyjściowej shadera z poprzedniego etapu potoku
                 // renderingu: out float foo
```

Każdy element takiej tablicy odpowiada jednemu wierzchołkowi przetwarzanego prymitywu. Tablica może mieć opcjonalnie zadeklarowany rozmiar, który oczywiście jest zależny od rodzaju prymitywu. Zmienne wejściowe na tym etapie formalnie mogą posiadać opcjonalne kwalifikatory interpolacji oraz pomocniczy kwalifikator przechowywania `centroid`, jednak nie wywołuje to żadnego efektu. W dalszej części niniejszego odcinka kursu zostanie omówiony kwalifikator formatu wejściowego `layout`, który stosowany jest m.in. ze zmiennymi wejściowymi shadera kontroli teselacji, shadera ewaluacji teselacji oraz shadera geometrii do określenia typu prymitywu wejściowego.

Niektóre zmienne wejściowe i wyjściowe są tablicami, co oznaczałoby, że poprawny interfejs pomiędzy dwoma następującymi po sobie programowalnymi etapami potoku renderingu wymagał by deklaracji w shaderze dwuwymiarowej tablicy. Typowym przypadkiem jest tutaj interfejs pomiędzy shaderem wierzchołków, a shaderem geometrii, gdzie zmienne wyjściowe shadera wierzchołków i zmienne wejściowej shadera geometrii muszą być zgodne co do nazwy, typu i kwalifikatorów, ale shader wierzchołków nie może mieć zadeklarowanej zmiennej wyjściowej w formie tablicy. Ograniczenie to można ominąć umieszczając tablicę w bloku wyjściowym shaderze wierzchołków oraz w bloku wejściowym w shaderze geometrii, deklarowanym jako tablica. Generalnie błędem na etapie konsolidacja obiektu programu zakończy się niezgodność interfejsu pomiędzy zmiennymi wyjściowymi/wejściowymi w kolejnych programowalnych etapach potoku renderingu polegająca na różnicy w nazwie, typie lub kwalifikatorach (oczywiście poza kwalifikatorami `in` i `out`).

Dodatkowo shadery ewaluacji teselacji obsługują zmienne wejściowe określane dla ścieżki wierzchołków, które są deklarowane z kwalifikatorami `in` i `patch`. Powyższe zmienne wejściowe zawierają dane określone dla ścieżki wierzchołków na podstawie informacji zapisanych w zmiennych wyjściowych przez shader kontroli teselacji. Podobnie jak w przypadku innych zmiennych wejściowych, zmienne określane dla ścieżki wierzchołków muszą być zadeklarowane przy użyciu tego samego typu i kwalifikatorów jak zmienne wyjściowe dla ścieżki wierzchołków z poprzedniego programowalnego etapu renderingu, czyli z shadera kontroli teselacji. Dodajmy jeszcze, że kwalifikator `patch` może być użyty do deklaracji zmiennych wejściowych jedynie w shaderze ewaluacji teselacji.

Zmienne wejściowe shadera fragmentów zawierają dane, które typowo są interpolowane z wyjścia poprzedniego programowalnego etapu potoku renderingu. Zmienne te są deklarowane w shaderze z kwalifikatorem przechowywania `in`. Dopuszczalne są także pomocnicze kwalifikatory przechowywania `centroid` i `sample` oraz kwalifikatory interpolacji `flat`, `noperspective` i `smooth`. Błędem jest natomiast użycie do takich zmiennych kwalifikatora `patch`. Zmienne wejściowe shadera fragmentów muszą być zgodne ze zmiennymi wyjściowymi z poprzedniego programowalnego etapu potoku renderingu (zmienne te tworzą tzw. interfejs), przy czym zgodność ta obejmuje oprócz nazwy także typ i pomocnicze kwalifikatory przechowywania.

Zmienne wejściowe shadera fragmentów mogą być wyłącznie: liczbami całkowitymi ze znakiem i bez znaku, wektorami całkowitymi i zmiennoprzecinkowymi, liczbami zmiennoprzecinkowymi, macierzami oraz tablicami i strukturami z powyższymi typami danych. Zmienne wejściowe będące liczbami całkowitymi, wektorami liczb całkowitych bądź typami zmiennoprzecinkowymi podwójnej precyzji muszą być deklarowane z kwalifikatorem interpolacji `flat`.

Poniżej przedstawiamy przykładowe deklaracje zmiennych wejściowych shadera fragmentów:

```
in vec3 normal;
centroid in vec2 TexCoord;
invariant centroid in vec4 Color;
noperspective in float temperature;
flat in vec3 myColor;
noperspective centroid in vec2 myTexCoord;
```

Jeżeli interfejs pomiędzy programowalnymi etapami potoku renderingu jest zbudowany z shaderów znajdujących się w dwóch odrębnych obiektach programu, konsolidator nie ma możliwości dokonania kontroli zgodności pomiędzy zmiennymi wyjściowymi i wejściowymi odpowiednich shaderów. Właściwa współpraca pomiędzy shaderami wymaga w takim przypadku użycia kwalifikatorów formatu wyjściowego/wejściowego lub zastosowania w pełni zgodnych deklaracji bloków zmiennych wyjściowych/wejściowych.

Kwalifikator `uniform`

Kwalifikator `uniform` jest używany do deklarowania zmiennych globalnych, których wartość jest stała podczas przetwarzania prymitywu. Wszystkie zmienne jednorodne są zmiennymi tylko do odczytu i są inicjalizowane zewnętrznie za pomocą API OpenGL. Możliwe jest także inicjalizacja wartości zmiennej jednorodnej w shaderze (za wyjątkiem uchwytów tekstur, obrazów tekstur i liczników atomowych), która obowiązuje od momentu konsolidacji obiektu program. Zmienne jednorodne bez inicjalizacji mają początkową wartość równą 0. Oto przykładowe deklaracje zmiennych jednorodnych:

```
uniform vec4 lightPosition;
uniform vec3 color = vec3( 0.7, 0.7, 0.2 );  // wartość przypisana
                                             // podczas konsolidacji
```

Kwalifikator `uniform` może być użyty z dowolnym podstawowym typem danych w tym także z typami danych zdefiniowanymi przez użytkownika, włączając w to struktury i tablice.

Maksymalna ilość pamięci dostępnej dla zmiennych jednorodnych zależy od implementacji biblioteki OpenGL i jest określona odrębnie dla każdego rodzaju shadera. Zmienne jednorodne zadeklarowane, ale nieużywane nie wliczają się do powyższego limitu. Przy obliczaniu limitu pod uwagę brane są zarówno zmienne jednorodne zdefiniowane przez użytkownika jak i zmienne wbudowane.

Jeżeli konsolidacja obejmuje kilka shaderów, współdzielą one pojedynczą globalną przestrzeń nazw zmiennych jednorodnych. Stąd typy zmiennych jednorodnych o takich samych nazwach i ich inicjalizacje muszą być zgodne we wszystkich shaderach skonsolidowanych w jednym obiekcie programu. Jedyny wyjątek stanowi możliwość wystąpienia inicjalizacji konkretnej zmiennej w danym shaderze przy jednoczesnym braki inicjalizacji tej zmiennej w innym shaderze.

Zmienne wyjściowe

Zmienne wyjściowe shadera deklarowane są przy użycia kwalifikatora przechowywania `out` i muszą być deklarowane jako zmienne globalne. Zmienne te stanowią połączenie shadera z poprzednim etapem potoku renderingu OpenGL, a ich wartość jest ustalana z chwilą zakończenia działania shadera z tego wcześniejszego etapu potoku renderingu.

Shadery nie obsługują zmiennych globalnych z kwalifikatorem przechowywania `inout`, które miałyby jednocześnie służyć jako zmienne wejściowe i wyjściowe. Nie jest także dopuszczalna deklaracja zmiennej zawierająca jednocześnie kwalifikatory `in` i `out`. Zmienne wyjściowe muszą być deklarowane z innymi nazwami niż zmienne wejściowe. Jednakże zmienne można łączyć w wyjściowe i wejściowe bloki interfejsu, które mogą zawierać takie same nazwy zmiennych, z dostępem za pośrednictwem nazwy instancji bloku.

Zmienne wyjściowe shadera wierzchołków, shadera ewaluacji teselacji i shadera geometrii zawierają atrybuty wierzchołka i deklarowane są przy użyciu kwalifikatora przechowywania `out`. W przypadku shadera kontroli teselacji możliwe jest także użycie do zmiennych wyjściowych kwalifikatora przechowywania `patch`. Zmienne wyjściowe mogą być wyłącznie: liczbami zmiennoprzecinkowymi, wektorami zmiennoprzecinkowymi, macierzami, liczbami całkowitymi ze znakiem i bez znaku, wektorami całkowitymi oraz tablicami i strukturami z powyższymi typami danych.

Przykładowe deklaracje pojedynczych zmiennych wyjściowych shadera wierzchołków, shadera ewaluacji teselacji lub shadera geometrii przedstawione są poniżej:

```
out vec3 normal;
centroid out vec2 TexCoord;
invariant centroid out vec4 Color;
noperspective out float temperature;
flat out vec3 myColor;
noperspective centroid out vec2 myTexCoord;
sample out vec4 perSampleColor;
```

Dodatkowo zmienne wyjściowe można połączyć z bloki interfejsu opisane w dalszej części tego odcinka kursu. Bloki wyjściowe umożliwiają m.in. przekazywanie tablic pomiędzy shaderem wierzchołków a shaderem geometrii. Zastosowanie bloków interfejsu umożliwia także zastosowanie takiego samego interfejsu wejściowego dla shadera fragmentów jak dla shadera geometrii dla danych przekazywanych przez shader wierzchołków.

Zmienne wyjściowe shadera kontroli teselacji mogą zawierać dane pojedynczych wierzchołków lub dane ścieżek (zbiorów) wierzchołków. Zmienne wyjściowe z danymi wierzchołków są stablicowane i deklarowane za pomocą kwalifikatora przechowywania `out`, ale bez kwalifikatora `patch`. Natomiast zmienne wyjściowe z danymi ścieżek wierzchołków są deklarowane przy

równoczesnym użyciu kwalifikatorów `out` i `patch`. W obu przypadkach zmienne wyjściowe shadera kontroli teselacji mogą być: liczbami zmiennoprzecinkowymi, wektorami zmiennoprzecinkowymi, macierzami, liczbami całkowitymi ze znakiem i bez znaku, wektorami całkowitymi oraz tablicami i strukturami z powyższymi typami danych. Jednak w przypadku danych wierzchołków zmienne wyjściowe (lub bloki interfejsu) muszą być deklarowane jako tablice. Oto przykład:

```
out float foo[];     // w następnym shaderze będzie
                     // to wejście: in float foo[]
```

Każdy element takiej tablicy odpowiada atrybutowi pojedynczego wierzchołka generowanego prymitywu. Każda tablica opcjonalnie może posiadać rozmiar, który musi być zgodny z deklaracją w kwalifikatorze formatu wyjściowego (opis kwalifikatorów formatu znajduje się w dalszej części niniejszego tekstu) i odpowiadać ilości wierzchołków w wyjściowej ścieżce. W przypadku, gdy zmienna wyjściowa jest tablicą, musi zostać zadeklarowana w bloku wyjściowym, którego nazwa instancji jest zadeklarowana jako tablica (przypomnijmy, że GLSL nie obsługuje tablic dwuwymiarowych).

Każde wywołanie shadera kontroli teselacji ma przyporządkowaną wyjściową ścieżkę oraz może przypisać wartości wyjściowe wierzchołków, ale tylko odpowiadającemu wywołaniu wierzchołkowi, który jest identyfikowany przez wbudowaną zmienną `gl_InvocationID`. Generalnie kolejność wywołań shadera kontroli teselacji dla danej ścieżki wejściowej jest niezdefiniowana, chyba, że zostanie użyta wbudowana funkcja `barrier`. Gdy shader wywoła funkcję `barrier`, jego wykonanie jest przerywane, aż do chwili, gdy inne jego wywołania osiągną to samo miejsce wywołania. Zmienne wyjściowe zapisane przez dowolne wywołanie shadera przed wywołaniem funkcji `barrier`, są dostępne dla pozostałych wywołań shadera już po wyjściu z funkcji `barrier`.

Zasada działania shadera kontroli teselacji, a zwłaszcza wspomniana wyżej nieokreślona kolejność wywoływania shadera pomiędzy poszczególnymi wywołaniami funkcji `barrier`, powoduje, że zmienne wyjściowe określane na wierzchołek lub ścieżkę wierzchołków są w niekiedy nieokreślone. Przy założeniu, że punktami synchronizacji są początek i koniec wywołania shadera oraz wywołanie funkcji `barrier`, stan omawianych zmiennych wyjściowych jest nieokreślony w następujących przypadkach:

- na początku wywołania shadera,
- w punkcie synchronizacji, za wyjątkiem sytuacji, gdy: wartość była zdefiniowana przed poprzednim punktem synchronizacji i nie była następnie zapisywana przez jakiekolwiek wywołanie shadera, wartość była zapisana dokładnie przez jedno wywołanie shadera od poprzedniego punktu synchronizacji, wartość była zapisana przez wiele wywołań shadera od poprzedniego punktu synchronizacji i ostatni zapis wykonany przez te wszystkie wywołania dotyczył tej samej wartości,
- podczas odczytu przez wywołanie shadera w przypadkach, gdy: wartość była niezdefiniowana w poprzednim punkcie synchronizacji i nie była od tego czasu zapisywana przez dane wywołanie shadera, wartość jest zapisana przez inne wywołanie shadera pomiędzy poprzednim i następnym punktem synchronizacji nawet jeśli jej odczyt następuje w kodzie po zapisie.

Zmienne wyjściowe shadera fragmentów zawierają dane fragmentu i są definiowane przy użyciu kwalifikatora przechowywania `out`. Błędem jest użycie w przypadku tych zmiennych pomocniczych kwalifikatorów przechowywania (`centroid`, `sample`, `patch`) lub kwalifikatorów interpolacji (`smooth`, `flat`, `noperspective`). Zmienne wyjściowe shadera fragmentów mogą być wyłącznie: liczbami zmiennoprzecinkowymi typu `float`, wektorami zmiennoprzecinkowymi, liczbami całkowitymi ze znakiem i bez znaku, wektorami całkowitymi lub tablicami z powyższymi typami danych. Jako zmienne wyjściowe shadera fragmentów nie są obsługiwane macierze i

struktury. Przykładowe deklaracje zmiennych wyjściowych shadera fragmentów przedstawione są poniżej:

```
out vec4 FragmentColor;
out uint Luminosity;
```

Bloki interfejsu

Zmienne wejściowe, wyjściowe i jednorodne shaderów mogą być grupowane w tzw. nazwane bloki interfejsu (ang. *named interface blocks*), które są bardziej czytelne i uniwersalne niż deklaracje indywidualnych zmiennych, które tworzą standardowy, nienazwany, interfejs. Bloku może posiadać opcjonalną nazwę instancji, używaną przez shader do odwoływania się do jego składowych. Blok wyjściowy z programowalnego etapu potoku rendering ma swój odpowiednik w bloku wejściowym w następnym programowalnym etapie potoku renderingu. Bloków zmiennych wejściowych i wyjściowych nie można stosować do komunikacji z nieprogramowalną częścią potoku renderingu (odpowiednio blok wejściowy w shaderze wierzchołków i blok wyjściowy w shaderze fragmentów). Funkcjonalność taka może się jednak pojawić w przyszłych wersjach języka GLSL. Dane bloku zmiennych jednorodnych przechowywane są w specjalnym rodzaju obiektu bufora.

Definicję bloku rozpoczyna jedno ze słów zarezerwowanych `in`, `out` lub `uniform` tzw. kwalifikator interfejsu określający rodzaj interfejsu bloku (odpowiednio blok wejściowy, wyjściowy i blok zmiennych jednorodnych). Następnie występuje nazwa bloku i nazwy poszczególnych pól bloku ograniczone, podobnie jak w strukturach, nawiasami sześciennymi „`{}`”. Oto pierwszy przykład bloku zmiennych jednorodnych o nazwie `Transform`, który grupuje cztery zmienne jednorodne:

```
uniform Transform
{
    mat4 ModelViewMatrix;
    mat4 ModelViewProjectionMatrix;
    uniform mat3 NormalMatrix;  // dopuszczalny ponowny kwalifikator
    float Deformation;
};
```

Typy i deklaracje w blokach są takie same jak inne deklaracje zmiennych wyjściowych, wejściowych i jednorodnych poza blokiem, w tym wcześniej zdefiniowane struktury oraz tablice z tymi typami, poza następującymi wyjątkami:

- inicjalizacje nie są dopuszczalne,
- uchwyty tekstur, obrazy tekstur i liczniki atomowe nie są dopuszczalne,
- definicje struktur nie są dopuszczalne wewnątrz bloku.

Kwalifikator interfejsu określa jednocześnie kwalifikator przechowywania wszystkich zmiennych umieszczonych w bloku. W stosunku do zmiennych można także użyć opcjonalnych kwalifikatorów, w tym kwalifikatorów interpolacyjnych, kwalifikatorów przechowywania i pomocniczych kwalifikatorów przechowywania, które muszą być jednak zgodne z rodzajem bloku. Powtórzenie kwalifikatora przechowywania `in`, `out` lub `uniform` w odpowiednio bloku wejściowym, wyjściowym lub zmiennych jednorodnych jest opcjonalne. Oto przykład:

```
in Material
{
    smooth in vec4 Color1;  // poprawnie, wejście wewnątrz bloku
    smooth vec4 Color2;     // poprawnie kwalifikator in pochodzi
                            // z kwalifikatora bloku in Material
    vec2 TexCoord;          // poprawnie, TexCoord jest wejściem
    uniform float Atten;    // błąd, niezgodność interfejsów
};
```

Na potrzeby niniejszego odcinka kursu, zdefiniujemy pojęcie interfejsu, którego zadaniem jest obsługa jednej z dwóch funkcjonalności:

- wszystkie zmienne jednorodne w shaderach wchodzących w skład jednego obiektu programu,
- połączenie pomiędzy sąsiadującymi ze sobą programowalnymi etapami potoku renderingu; obejmuje to wszystkie wyjścia shaderów z pierwszego etapu i wszystkie wejścia shaderów z drugiego etapu.

Nazwa bloku jest używana w celu dopasowania interfejsów. Interfejs wyjściowy jednego z programowalnych etapów potoku renderingu musi być zgodny z interfejsem wejściowym o takiej samej nazwie w następnym etapie potoku renderingu. W przypadku bloku zmiennych jednorodnych aplikacja używa nazwy bloku do jego identyfikacji. Bloki o zgodnych nazwach w ramach interfejsu muszą posiadać taką samą sekwencję typów, sekwencję nazw pól oraz opisywane dalej kwalifikatory formatu. To samo dotyczy bloków deklarowanych jako tablica, w przypadku których wymagana jest zgodność ich rozmiarów. Jakakolwiek niezgodność w powyższym zakresie spowoduje błąd na etapie konsolidacji obiektu programu.

Razem z definicją bloku może wystąpić także nazwa instancji (egzemplarza) bloku. Jeżeli nazwa instancji bloku nie występuje, nazwy zdefiniowane wewnątrz bloku są dostępne na takich samych zasadach jak zmienne globalne zadeklarowane na zewnątrz bloku. Jeżeli nazwa instancji bloku została określona, dostęp do składowych bloku odbywa się za pomocą operatora „.”, podobnie jak dostęp do pól struktur. Oto przykłady:

```
in Light
{
    vec4 LightPos;
    vec3 LightColor;
};

in ColoredTexture
{
    vec4 Color;
    vec2 TexCoord;
} Material;             // nazwa instancji bloku

vec3 Color;             // zmienna Color inna niż Material.Color
vec4 LightPos;          // błąd, zmienna już zdefiniowana
...
... = LightPos;         // dostęp do LightPos
... = Material.Color;   // dostęp do Color w bloku ColoredTexture
```

Poza językiem cieniowania (np. w API OpenGL) składowe bloku są identyfikowane w podobny sposób, za wyjątkiem sytuacji, gdy występuje nazwa instancji bloku. Wówczas API odwołuje się do pól bloku za pośrednictwem nazwy bloku i przy użyciu operatora „.”, nie zaś nazwy jego instancji. Gdy nazwa instancji bloku nie jest zdefiniowana API nie używa nazwy bloku, tylko samą nazwę pola w bloku. Oto przykład:

```
out Vertex
{
    vec4 Position;    // API używa Vertex.Position
    vec2 Texture;
} Coords;             // shader używa Coords.Position

out Vertex2
```

```
{
    vec4 Color;       // API używa Color
};
```

Dla bloków zadeklarowanych jako tablica, indeks tablicy musi być dodany podczas dostępu do elementu bloku, tak jak w poniższym przykładzie:

```
uniform Transform              // API używa Transform[2] do dostępu
{                              // do instancji numer 2
    mat4 ModelViewMatrix;
    mat4 ModelViewProjectionMatrix;
    float Deformation;
} transforms[4];
...
... = transforms[2].ModelViewMatrix;  // dostęp shadera do instancji
                                      // numer 2; API używa
                                      // Transform.ModelViewMatrix
                                      // do pobrania offsetu bloku
                                      // lub innego zapytania
```

Dla bloków zmiennych jednorodnych zadeklarowanych jako tablica, każdy element tablicy odpowiada odrębnemu punktowi wiązania obiektu bufora. Rozmiar tablicy określa ilość niezbędnych punktów wiązania obiektu bufora, stąd tablica bloków zmiennych jednorodnych musi mieć określony rozmiar. Tablica bloków zmiennych jednorodnych może być indeksowana tylko przy użyciu dynamicznego jednorodnego wyrażenia całkowitego.

Gdy używamy API OpenGL do identyfikacji nazwy pojedynczego bloku w tablicy bloków, ciąg znaków z nazwą bloku musi zawierać indeks tablicy (jak w ostatnim przykładzie `Transform[2]`). Jeżeli natomiast API OpenGL odwołuje się do offsetu lub innej właściwości elementu składowego bloku, indeks tablicy nie może wystąpić (w ostatnim przykładzie odpowiednio `Transform.ModelViewMatrix`).

Blok zmiennych wejściowych shadera geometrii musi być zadeklarowany jako tablica i musi spełniać warunki definicji tablicy i zasady konsolidacji określone dla wszystkich zmiennych wejściowych shadera geometrii. Pozostałe tablice bloków wejściowych i wyjściowych muszą mieć określony rozmiar.

Maksymalna ilość bloków zmiennych jednorodnych, która może być użyta na danym etapie renderingu jest zależna od implementacji. Jej przekroczenie spowoduje błąd konsolidacji.

Kwalifikatory formatu

Kwalifikatory formatu (ang. *layout qualifiers*) mogą być stosowane w szeregu formach deklaracji. Przykładowo kwalifikator formatu może być częścią definicji bloku interfejsu lub składowej bloku. Dodatkowo ich stosowanie zależy od rodzaju zmiennej oraz rodzaju program cieniowania. Specyfiką kwalifikatorów formatu jest to, że są one traktowane jako identyfikatory, nie zaś słowa zarezerwowane. Dodatkowo identyfikatory kwalifikatorów formatu, w przeciwieństwie do innych identyfikatorów, zasadniczo nie podlegają sprawdzaniu wielkości znaków.

Deklaracja kwalifikatora formatu składa się ze słowa zarezerwowanego `layout(…)` i umieszczonych w nawiasach identyfikatorów formatu. Identyfikator może występować pojedynczo, definicja może także zawierać kilka identyfikatorów, które są wówczas oddzielone przecinkami. Niektóre kwalifikatory dostępne są we wszystkich rodzajach shaderów, inne obsługują tylko wybrane typy shaderów. Dokładną składnię poszczególnych rodzajów kwalifikatorów formatu, wraz z podziałem na poszczególne rodzaje shaderów, omawiamy poniżej.

Kwalifikatory formatu wejściowego dostępne we wszystkich shaderach

Wspólnym dla wszystkich rodzajów shaderów identyfikatorem kwalifikatora formatu wejściowego jest `location` określający położenie danych zmiennych wejściowych shadera. W przypadku shadera wierzchołków jest to numer ogólnego atrybutu wierzchołka definiowany z poziomu API OpenGL. W pozostałych rodzajach shaderów jest to numer wektora który jest używany przy określaniu zgodności ze zmienną wyjściową z shadera z poprzedniego programowalnego etapu renderingu, nawet jeżeli shader ten znajduje się w innym obiekcie programu.

Oto przykładowa deklaracja zmiennej wejściowej `normal` z przyporządkowanym położeniem o numerze 3:

```
layout( location = 3 ) in vec4 normal;
```

Ilość numerów położenia (lokacji) wykorzystywanych przez daną zmienną wejściową zależy od rodzaju tej zmiennej. Zmienna typu skalarnego lub wektorowego inna niż `dvec3` lub `dvec4` wymaga pojedynczego numeru położenia, natomiast wymienione typy wektorowe `dvec3` i `dvec4` używają dwóch kolejnych numerów położenia. Wyjątek od tej zasady dotyczy shaderów wierzchołków, w których dowolny typ skalarny lub wektorowy, w tym wektory z liczbami zmiennoprzecinkowymi podwójnej precyzji, wykorzystuje pojedynczy numer położenia.

W przypadku zmiennych typu tablicowego ilość wykorzystanych numerów położenia jest iloczynem wielkości tablicy i ilości numerów wykorzystywanych przez jej element. Przykładowo, poniższa zmienna wejściowa używa trzech numerów położenia o wartościach 6, 7 i 8:

```
layout( location = 6 ) in vec4 colors[3];
```

W przypadku zmiennych wejściowych typu macierzowego ilość wykorzystywanych numerów położeni jest określana tak samo jako dla tablicy o wielkości równej ilości wierszy macierzy, zawierającej wektory o liczbie składowych równej ilości kolumn macierzy. Na przykład przedstawiona poniżej zmienna wejściowa wykorzystuje numery położenia 9-12 dla pierwszej macierzy z tablicy `transforms` i numery 13-16 dla drugiej macierzy z tej tablicy:

```
layout( location = 9 ) in mat4 transforms[2];
```

Identyfikator `location` kwalifikatora formatu wejściowego może być także użyty do struktur, nie jest jednak dopuszczalny do poszczególnych pól struktury. Pola struktury mają przyporządkowane kolejne numer położenia, zgodnie z wyżej opisanymi regułami, z numerem początkowym wyspecyfikowanym w kwalifikatorze. Popatrzmy na poniższy przykład:

```
layout( location = 3 ) struct S
{
    vec3 a;        // numer położenia 3
    mat2 b;        // numer położenia 4 dla pierwszej
                   // kolumny i 5 dla drugiej kolumny
    vec4 c[2];     // numery położenia kolejno 6 i 7
} s;
```

Ilość dostępnych numerów położenia zmiennych wejściowych dostępnych dla shadera jest ograniczona i zależy od implementacji. W przypadku shaderów wierzchołków limit ten jest określony przez ilość ogólnych atrybutów wierzchołków. Dla pozostałych rodzajów shaderów limit także jest zależny od implementacji i nie może być mniejszy niż $^1/_4$ maksymalnej ilości składowych wejściowych. Przekroczenie powyższych limitów implementacyjnych zakończy się błędem na etapie konsolidacji programu.

Dodatkowo specyfikacja wprowadza ograniczenia dotyczące sposobu wykorzystania numerów położenia. Poza shaderem wierzchołków nie jest dopuszczalne, aby zmienne wejściowe korzystały z takich samych numerów położenia. W takich sytuacjach generowany jest błąd na etapie konsolidacji. W przypadku shaderów wierzchołków aliasing w tym zakresie jest dopuszczalny zarówno poprzez opisywany mechanizm kwalifikatora formatu wejściowego z identyfikatorem `location` jak i z poziomu API OpenGL. Ponadto implementacja OpenGL może zgłosić błąd na etapie konsolidacji programu, w przypadku, gdy różne dane wejściowe shadera wierzchołków są przyporządkowane do jednego numeru położenia.

Poza shaderem wierzchołków identyfikator `location` kwalifikatora formatu wejściowego zmiennych wejściowych musi być zgodny z analogicznym kwalifikatorem w zmiennej wyjściowej w shaderze z poprzedniego programowalnego etapu renderingu. Dodajmy, że omawianego kwalifikatora formatu wejściowego nie można używać do interfejsu bloków zmiennych wejściowych lub poszczególnych elementów takiego bloku.

Dodajmy jeszcze, że w numery położenia zmiennych wejściowych shadera wierzchołków bez wyspecyfikowanego kwalifikatora formatu wejściowego są określane z poziomu API OpenGL lub ustalane automatycznie na etapie konsolidacji obiektu programu.

Kwalifikatory formatu wejściowego shadera ewaluacji teselacji

W shaderze ewaluacji teselacji kwalifikator formatu wejściowego określa następujące aspekty działania generatora prymitywów w teselatorze: tryb prymitywu, orientację wierzchołków prymitywu oraz sposób podziału krawędzi prymitywów.

Tryb prymitywu używany przez generator prymitywów w teselatorze określa jeden z następujących identyfikatorów: `triangles` (podział trójkątów na mniejsze trójkąty), `quads` (podział czworokątów na trójkąty) oraz `isolines` (podział czworokątów na zbiór linii). Orientację wierzchołków prymitywu określają dwa identyfikatory: `cw` i `ccw`. Pierwszy z nich wskazuje, że orientacja wierzchołków jest zgodna z ruchami wskazówek zegara, drugi oznacza orientację wierzchołków przeciwną do ruchu wskazówek zegara.

Grupa identyfikatorów regulujących sposób podziału krawędzi prymitywów, tj. odstępy pomiędzy wierzchołkami, składa się z trzech identyfikatorów. Pierwszy z nich to `equal_spacing`, który oznacza, że krawędzie są dzielone w zbiór odcinków o równej długości. Drugi identyfikator `fractional_even_spacing` wskazuje, że krawędzie dzielone są na parzystą ilość równych odcinków oraz dwa dodatkowe odcinki o innej długości tzw. "ułamkowe". Trzeci identyfikator z tej grupy `fractional_odd_spacing`, oznacza podział krawędzi na nieparzystą ilość równych odcinków oraz dwa dodatkowe odcinki "ułamkowe" o innej długości.

Dostępny jest jeszcze identyfikator `point_mode` określający, że generator prymitywów w teselatorze produkuje punkty dla każdego unikatowego wierzchołka w teselowanym prymitywie, zamiast generowania odcinków lub trójkątów.

Co najmniej jeden shader ewaluacji teselacji musi mieć zadeklarowany tryb prymitywu w kwalifikatorze formatu wejściowego. Pozostałe grupy identyfikatorów są opcjonalne, a w przypadku braku stosownych deklaracji przyjmowane są domyślne ustawienia: `equal_spacing` i `ccw`. Ponadto tryb generowania punktów domyślnie jest nieaktywny - generator prymitywów produkuje odcinki lub trójkąty. W przypadku wielokrotnych deklaracji kwalifikatorów formatu wejściowego w shaderach kontroli teselacji, wszystkie powyższe deklaracje muszą posiadać te same identyfikatory.

Kwalifikatory formatu wejściowego shadera geometrii

W shaderach geometrii kwalifikator formatu wejściowego zawiera identyfikator określający rodzaj prymitywu wejściowego shadera. Dopuszczalne jest użycie jednego z następujących identyfikatorów: `points` (prymityw `GL_POINTS`), `lines` (prymitywy `GL_LINES`, `GL_LINE_STRIP` lub `GL_LINE_LOOP`), `lines_adjacency` (prymitywy `GL_LINES_ADJACENCY` lub `GL_LINE_STRIP_ADJACENCY`), `triangles` (prymitywy `GL_TRIANGLES`, `GL_TRIANGLE_STRIP` lub `GL_TRIANGLE_FAN`) oraz

`triangles_adjacency` (prymitywy `GL_TRIANGLES_ADJACENCY` lub `GL_TRIANGLE_STRIP_ADJACENCY`).

W deklaracji kwalifikatora formatu wejściowego dopuszczalny jest tylko jeden z powyższych identyfikatorów. Przynajmniej jeden shader geometrii w obiekcie programu musi zawierać deklarację kwalifikatora formatu wejściowego. Jeżeli deklaracja taka pojawia się w kilku shaderach geometrii znajdujących się w jednym obiekcie programu, każda z nich musi opisywać taki sam format wejściowy. Oto przykładowa deklaracja formatu wejściowego shadera geometrii, gdzie wejściem shadera geometrii są trójkąty:

```
layout( triangles ) in;
```

Opcjonalnie kwalifikator formatu wejściowego shadera geometrii może zawierać identyfikator `invocations` określający ilość wywołań shadera, dla każdego przetwarzanego prymitywu wejściowego. Standardowo shader geometrii wywoływany jest jednokrotnie, dla każdego prymitywu. Ilość wywołań nie może przekroczyć limitu zależnego od implementacji. Przykładowo, gdy chcemy wykonać shader geometrii sześć razy dla każdego przetwarzanego trójkąta, deklarujemy następujący format wejściowy:

```
layout( triangles, invocations = 6 ) in;
```

W przypadku, gdy ilość wywołań jest zadeklarowana w jednym z shaderów, wszystkie deklaracje formatu wejściowego w innych shaderach geometrii z danego obiektu programu muszą określać taką samą ilość wywołań.

Deklaracja kwalifikatora formatu wejściowego w sposób jednoznaczny określa rozmiar wszystkich tablic danych wejściowych shadera geometrii, jeżeli nie miały one wcześniej określonego rozmiaru. Rozmiary te zawiera Tabela 7.

Identyfikator	Rozmiar tablic wejściowych
`points`	1
`lines`	2
`lines_adjacency`	4
`triangles`	3
`triangles_adjacency`	6

Tabela 7 Rozmiary tablic danych wejściowych shadera geometrii w zależności od rodzaju prymitywu wejściowego.

Wbudowane zmienne wejściowe shadera geometrii umieszczone są w tablicy `gl_in[]` (opis wszystkich wbudowanych zmiennych shaderów znajduje się w dalszej części tego odcinka kursu), której rozmiar także określa deklaracja kwalifikatora formatu wejściowego. Dopiero po jej wystąpieniu możliwe jest pobranie rozmiaru tej tablicy przy użyciu wyrażenia `gl_in.length()`.

Błędem kompilacji lub konsolidacji zakończy się każda niezgodność rozmiaru tablicy danych wejściowych shadera geometrii z rozmiarem określonym przez deklarację kwalifikatora formatu wejściowego. W przypadku, gdy obiekt programu zawiera wiele shaderów geometrii, zgodność rozmiarów dotyczy wszystkich zadeklarowanych tablic wejściowych. Poniżej znajduje się szereg przykładów takich poprawnych i błędnych deklaracji:

```
// sekwencja poleceń w jednym shaderze...
in vec4 Color1[];        // rozmiar tablicy nieznany
...Color1.length()...    // błąd, length() nieznany
in vec4 Color2[2];       // rozmiar tablicy wejściowej 2
...Color1.length()...    // błąd, Color1 dalej nie ma rozmiaru
in vec4 Color3[3];       // błąd, rozmiary wejścia są niezgodne
layout( lines ) in;      // poprawne dla Color2, rozmiar wejścia
```

```
                          // wynosi 2, i jest zgodny z rozmiarem
                          // wejścia Color2
in vec4 Color4[3];        // błąd, niezgodność z rozmiarem wejścia
...Color1.length()...     // poprawnie, length() jest równe 2,
                          // rozmiar Color1 jest określony przez
                          // deklarację kwalifikatora layout
layout( lines ) in;       // poprawne, inna deklaracja kwalifikatora
                          // formatu wejściowego layout
layout( triangles ) in;   // błąd, deklaracja niezgodna z poprzednią
                          // deklaracją formatu wejściowego layout
```

Kwalifikatory formatu wejściowego shadera fragmentów

Także shadery fragmentów posiadają odrębną grupę identyfikatorów kwalifikatora formatu wejściowego. Pierwsze dwa identyfikatory: `origin_upper_left` i `pixel_center_integer` służą do redeklaracji wbudowanej zmiennej `gl_FragCoord` (opis wszystkich wbudowanych zmiennych shaderów znajduje się w dalszej części tego odcinka kursu). Zmienna ta zawiera położenie fragmentu w układzie współrzędnych okna, w którym standardowo (`pixel_center_integer`) początek układu znajduje się lewym dolnym wierzchołku okna, a współrzędne środków piksela położone są w połowie współrzędnych piksela. Przy takich założeniach współrzędne (x, y) środka piksela znajdującego się w lewym dolnym wierzchołku okna mają wartość $\left(\frac{1}{2}, \frac{1}{2}\right)$. Drugi identyfikator `origin_upper_left` kwalifikatora formatu wejściowego zmienia położenie początku układu współrzędnych okna na jego lewy górny wierzchołek, przy czym wartości y rosną w dół okna. Ponadto identyfikator ten zmienia - przesuwa o pół współrzędnych piksela w obu kierunkach, położenie środka piksela zwracane w zmiennej `gl_FragCoord`. W takim przypadku współrzędne (x, y) położenia środka piksela, o przykładowej domyślnej wartości $\left(\frac{1}{2}, \frac{1}{2}\right)$, z identyfikatorem `origin_upper_left` będą miały wartość $(0,0)$.

Poniżej przedstawiamy kilka przykładowych redeklaracji zmiennej `gl_FragCoord`. Zauważmy, że pojedynczy kwalifikator może zawierać oba identyfikatory, które oddzielane są przecinkami:

```
in vec4 gl_FragCoord;  // redeklaracja nic nie zmieniająca

// wszystkie poniższe redeklaracje modyfikują format wejściowy
layout( origin_upper_left ) in vec4 gl_FragCoord;
layout( pixel_center_integer ) in vec4 gl_FragCoord;
layout( origin_upper_left, pixel_center_integer ) in vec4
gl_FragCoord;
```

Jeżeli zmienna `gl_FragCoord` jest redeklarowana w dowolnym shaderze fragmentów w obiekcie programu, musi być redeklarowana w każdym z shaderów fragmentów, który statycznie używa `gl_FragCoord`. Wszystkie redeklaracje `gl_FragCoord` w shaderach fragmentów w ramach obiektu programu muszą mieć takie same kwalifikatory i muszą wystąpić przed pierwszym użyciem tej zmiennej.

Redeklaracje zmiennej `gl_FragCoord` mają wpływ wyłącznie na współrzędne przekazywane w `gl_FragCoord.x` i `gl_FragCoord.y`. Pozostałe operacje takie jak: rasteryzacja, transformacje czy inne elementu potoku renderingu OpenGL nie są modyfikowane.

Trzeci identyfikator kwalifikatora formatu wejściowego dostępny w shaderze fragmentów `early_fragment_tests` nie jest związany ze zmiennymi wejściowymi shadera, ale określa, na którym etapie przetwarzania fragmentów wykonywany jest shader fragmentów. Standardowo shader fragmentów wykonywany jest przed następującymi operacjami na fragmentach: test

własności piksela, test nożycowy, test szablonu, test głębokości i test zasłaniania. Użycie omawianego kwalifikatora:

```
layout( early_fragment_tests ) in;
```

oznacza, że wymienione operacje na fragmentach są wykonywane przed wywołaniem shadera fragmentów.

Kwalifikatory formatu wyjściowego dostępne we wszystkich shaderach

Wspólnym dla wszystkich rodzajów shaderów identyfikatorem kwalifikatora formatu wyjściowego jest `location` określający położenie danych zmiennych wyjściowych shadera. W przypadku shadera fragmentów kwalifikator `location` określa numer bufora renderingu dla poszczególnych składowych fragmentu. W pozostałych rodzajach shaderów jest to numer wektora, który jest używany przy określaniu zgodności ze zmienną wejściową z shadera z następnego programowalnego etapu renderingu, nawet jeżeli shader ten znajduje się w innym obiekcie programu.

Oto przykładowa deklaracja zmiennej wyjściowej `color` z przyporządkowanym położeniem o numerze 3:

```
layout( location = 3 ) out vec4 color;
```

W przypadku shadera fragmentów dostępny jest także opcjonalny identyfikator `index`, który określa numer indeksu zmiennej wyjściowej dla równania mieszania kolorów. Przykładowo:

```
layout( location = 3, index = 1 ) out vec4 factor;
```

zmiennej wyjściowej `factor` przyporządkowano położenie numer 3 i numer indeksu 1 dla równania mieszania kolorów. Użycie identyfikatora `index` wymaga także jednoczesnego użycia identyfikatora `location`. Dla identyfikatora `index` domyślnie przyjmowana jest wartość równa 0.

Ilość numerów położenia (lokacji) wykorzystywanych przez daną zmienną wyjściową zależy od rodzaju tej zmiennej. Zmienna typu skalarnego lub wektorowego inna niż `dvec3` lub `dvec4` wymaga pojedynczego numeru położenia. Wyjątek stanowią wymienione typy wektorowe `dvec3` i `dvec4`, które używają dwóch kolejnych numerów położenia. W przypadku zmiennych typu tablicowego ilość wykorzystanych numerów położenia jest iloczynem wielkości tablicy i ilości numerów wykorzystywanych przez jej element. Przykładowo, poniższa zmienna wejściowa używa trzech numerów położenia o wartościach 2, 3 i 4:

```
layout( location = 2 ) out vec4 colors[3];
```

W przypadku zmiennych wyjściowych typu macierzowego ilość wykorzystywanych numerów położenia jest określana tak samo jako dla tablicy o wielkości równej ilości wierszy macierzy, zawierającej wektory o liczbie składowych równej ilości kolumn macierzy. Identyfikator `location` kwalifikatora formatu wyjściowego może być także użyty do struktur, nie jest jednak dopuszczalny do poszczególnych pól struktury. Pola struktury mają przyporządkowane kolejne numer położenia, zgodnie z wyżej opisanymi regułami, z numerem początkowym wyspecyfikowanym w identyfikatorze `location`.

Ilość dostępnych numerów położenia zmiennych wejściowych dostępnych dla shadera jest ograniczona i zależy od implementacji. W przypadku shaderów fragmentów limit ten jest określony przez ilość dostępnych obszarów (buforów) renderingu. Dla pozostałych rodzajów shaderów limit także jest zależny od implementacji i nie może być mniejszy niż $^1/_4$ maksymalnej ilości składowych

wyjściowych. Przekroczenie powyższych limitów implementacyjnych zakończy się błędem na etapie konsolidacji programu.

Dodatkowo specyfikacja wprowadza ograniczenia dotyczące sposobu wykorzystania numerów położenia. W shaderze fragmentów nie jest dopuszczalne, aby zmienne wyjściowe korzystały z takich samych numerów położenia. W pozostałych rodzajach shaderów ograniczenie dotyczy korzystania przez zmienne wyjściowe z takiego samego numeru położenia. Naruszenie powyższych zasad spowoduje wygenerowanie błędu na etapie konsolidacji obiektu programu.

Poza shaderem fragmentów identyfikator `location` kwalifikatora formatu wyjściowego zmiennych wyjściowych musi być zgodny z analogicznym kwalifikatorem w zmiennej wejściowej w shaderze z następnego programowalnego etapu renderingu. Dodajmy, że omawianego kwalifikatora formatu wyjściowego nie można używać do interfejsu bloków zmiennych wyjściowych lub poszczególnych elementów takiego bloku.

Dodajmy jeszcze, że w numery położenia zmiennych wyjściowych shadera fragmentów bez wyspecyfikowanego kwalifikatora formatu wyjściowego są określane z poziomu API OpenGL lub ustalane automatycznie na etapie konsolidacji obiektu programu, przy czym wartość indeksu dla równania mieszania kolorów jest wówczas równa 0.

Kwalifikatory formatu wyjściowego shadera kontroli teselacji

Kwalifikator formatu wyjściowego shadera kontroli teselacji zawiera tylko jeden identyfikator `vertices` określający ilość wierzchołków w wyjściowej ścieżce wierzchołków generowanej przez shader kontroli teselacji. Jest to jednocześnie ilość wywołań shadera kontroli teselacji w obsłudze danej ścieżki wierzchołków. Wartość zawarta w omawianym identyfikatorze nie może być mniejsza lub równa 0, ani też przekraczać zależnej od implementacji maksymalnej wielkości ścieżki wierzchołków. Ponadto kwalifikator formatu wyjściowego shadera kontroli teselacji może być stosowany wyłącznie jako określenie interfejsu kwalifikatora out, nie może natomiast być zastosowany do deklaracji bloku wyjściowego, składowej takiego bloku lub deklaracji zmiennej wyjściowej.

Domyślnie wielkość tablicy `gl_out` w shaderze kontroli teselacji jest równa deklaracji w kwalifikatorze formatu wyjściowego tego shadera. Użycie metody `length` dla tej tablicy lub dla innych tablic danych wyjściowych bez wyspecyfikowanej wielości wymaga wcześniejszej deklaracji omawianego kwalifikatora. W przypadku, gdy wielkość tablic danych wyjściowych jest określona jawnie, deklaracja ta musi być zgodna z ilością wierzchołków w wyjściowej ścieżce wierzchołków określonej w kwalifikatorze formatu wyjściowego shadera kontroli teselacji.

Kwalifikator formatu wyjściowego shadera kontroli teselacji musi wystąpić w każdym obiekcie programu zawierającym ten rodzaj shadera, przy czym nie jest wymagana jego deklaracja w każdym shaderze kontroli teselacji występującym w danym obiekcie programu. W przypadku wielokrotnych deklaracji, wszystkie muszą określać taką samą ilość wierzchołków.

Kwalifikatory formatu wyjściowego shadera geometrii

Kwalifikatory formatu wyjściowego w shaderze geometrii określają: rodzaj prymitywy wyjściowego shadera, maksymalną ilość generowanych wierzchołków oraz numer aktywnego strumienia wyjściowego wierzchołków. Identyfikatory kwalifikatora formatu wyjściowego określają interfejs kwalifikatora `out`, nie określają natomiast formatu deklaracji bloku wyjściowego lub formatu deklaracji zmiennej.

Rodzaj prymitywu wyjściowego shadera geometrii określają następujące identyfikatory: `points` (prymityw `GL_POINTS`), `line_strip` (prymityw `GL_LINE_STRIP`) oraz `triangle_strip` (prymityw `GL_TRIANGLE_STRIP`). Co najmniej jeden shader geometrii w obiekcie programu musi zadeklarować rodzaj prymitywu wyjściowego. W przypadku wielu deklaracji, wszystkie shadery geometrii w obiekcie programu muszą deklarować taki sam rodzaj prymitywu wyjściowego.

Maksymalną ilość wierzchołków generowanych przez shader geometrii określa identyfikator `max_vertices` powiązany ze stałą całkowitą. Maksymalna dopuszczalna ilość wierzchołków jaką

może wygenerować shader geometrii zależy od implementacji i jest zawarta w specjalnej zmiennej `gl_MaxGeometryOutputVertices`. Wszystkie shadery geometrii w danym obiekcie programu muszą deklarować taką samą wartość identyfikatora `max_vertices`. Wymagana jest tylko jedna deklaracja tego identyfikatora kwalifikatora formatu wyjściowego w programie, i nie ma wymogu, aby wszystkie shadery geometrii zawierały taką deklarację.

Poniżej znajduje się kilka przykładów użycia deklaracji i redeklaracji kwalifikatorów formatu wyjściowego w shaderach geometrii:

```
layout( triangle_strip, max_vertices = 60 ) out; // kolejność
                                                 // identyfikatorów
                                                 // nie ma znaczenia
layout( max_vertices = 60 ) out;  // redeklaracja poprawna
layout( triangle_strip ) out;     // redeklaracja poprawna
layout( points ) out;             // błąd, redeklaracja zmienia
                                  // rodzaj prymitywu wyjściowego
                                  // z triangle_strip na points
layout( max_vertices = 30 ) out;  // błąd, redeklaracja zmienia
                                  // maksymalną ilość wierzchołków
                                  // generowanych przez shader
                                  // geometrii z 60 na 30
```

Nieco inne zasady obowiązują w przypadku identyfikatora `stream` określającego numer aktywnego strumienia wyjściowego wierzchołków (strumienie numerowane są od 0). Domyślny numer strumienia wyjściowego może zostać zadeklarowany globalnie, np. w poniższy sposób:

```
layout( stream = 1 ) out;
```

Każda nowa deklaracja zmienia domyślny numer strumienia wyjściowego wierzchołków i obowiązuje do ewentualnej kolejnej jego redeklaracji. Początkowy numer strumienia wyjściowego wierzchołków wynosi 0.

Identyfikator stream kwalifikatora formatu wyjściowego shadera geometrii może być także użyty do pojedynczych zmiennych wyjściowych lub bloku takich zmiennych wyjściowych. Deklaracja zmiennych w bloku wyjściowym także może zawierać omawiany identyfikator, ale wymagana jest wówczas jego zgodność z deklaracją przypisaną do bloku. Poniżej przedstawiamy kilka przykładów:

```
layout( stream = 1 ) out;  // domyślny strumień nr 1
out vec4 var1;             // var1 skierowane jest do domyślnego
                           // strumienia nr 1

layout( stream = 2 ) out Block1      // zmienne z Block1 skierowane
{                                    // do strumienia nr 2
    layout( stream = 2 ) vec4 var2;  // ponowna deklaracja numeru
                                     // strumienia wyjściowego
    layout( stream = 3 ) vec2 var3;  // błąd, niezgodność numeru
                                     // strumienia wyjściowego
    vec3 var4;                       // strumień wyjściowy nr 2
};

layout( stream = 0 ) out;    // nowy domyślny strumień nr 0
out vec4 var5;               // var5 skierowana do strumienia
                             // wyjściowego nr 0 (domyślnego)

out Block2                   // zmienne z Block2 skierowane
```

```
{                                   // do domyślnego strumienia nr 0
    vec4 var6;
};
layout( stream = 3 ) out vec4 var7; // zmienna var7 do strumienia
                                    // wyjściowego nr 3
```

Jeżeli blok zmiennych wyjściowych lub zmienna wyjściowa jest zadeklarowana więcej niż, każda z tych deklaracji musi przyporządkowywać zmienne do tego samego strumienia wyjściowego. Użycie numeru nieistniejącego strumienia spowoduje wystąpienie błędu kompilacji. Wbudowane zmienne shadera geometrii są zawsze przyporządkowane do strumienia wyjściowego nr 0.

Kwalifikatory formatu wyjściowego shadera fragmentów

Identyfikatory kwalifikatora formatu wejściowego w shaderze fragmentów dotyczą wyłącznie redeklaracji wbudowanej zmiennej wyjściowej `gl_FragDepth`. Identyfikatory te umożliwiają optymalizację operacji na buforze głębokości, np. poprzez założenie, że wszystkie fragmenty przejdą (lub nie) test głębokości.

Domyślnym identyfikatorem omawianego kwalifikatora jest `depth_any` oznaczający, że początkowa wartość wbudowanej zmiennej `gl_FragDepth` nie jest w żaden sposób modyfikowana, a test głębokości jest zawsze wykonywany po wywołaniu shadera fragmentów. Drugi identyfikator `depth_greater` wskazuje, że końcowa wartość `gl_FragDepth` jest zawsze większa lub równa wartości głębokości interpolowanej dla danego fragmentu, zawartej zmiennej `gl_FragCoord.z`. Kolejny identyfikator `depth_less` oznacza, że wartość `gl_FragDepth` może być jedynie zmniejszana w stosunku do wartości standardowej. Natomiast ostatni identyfikator `depth_unchanged` określa, że pomimo dopuszczalności zmian wartości zmiennej `gl_FragDepth` wewnątrz shadera, pozostała część potoku renderingu OpenGL zakłada niezmienność jej wartości.

Poniżej przedstawiamy kilka przykładowy redeklaracji wbudowanej zmiennej `gl_FragDepth`:

```
// redeklaracja nic nie zmieniająca
out float gl_FragDepth;

// zakładamy, że zmienna może być dowolnie modyfikowana
layout( depth_any ) out float gl_FragDepth;

// zakładamy, że zmienna może być modyfikowana
// jedynie poprzez zwiększanie wartości
layout( depth_greater ) out float gl_FragDepth;

// zakładamy, że zmienna może być modyfikowana
// jedynie poprzez zmniejszanie wartości
layout( depth_less ) out float gl_FragDepth;

// zakładamy, że zmienna nie jest modyfikowana
layout( depth_unchanged ) out float gl_FragDepth;
```

Niezgodność końcowej wartości `gl_FragDepth` z określonym kwalifikatorem formatu wyjściowego oznacza, że wynik testu głębokości dla danego fragmentu jest nieokreślony, jednak nie jest generowany żaden błąd. Jeżeli test głębokości jest pozytywny i shader dokonuje zapisu wartości głębokości, zawartość bufora głębokości będzie zawsze pochodziła ze zmiennej `gl_FragDepth` niezależnie od zgodności kwalifikatora formatu wyjściowego shadera fragmentów.

Jeżeli zmienna `gl_FragDepth` jest redeklarowana w shaderze fragmentów, analogiczna redeklaracja musi wystąpić w każdym shaderze fragmentów z danego obiektu programu, który

dokonuje zapisu tej zmiennej. Redeklaracja musi wystąpić przed pierwszym użyciem zmiennej `gl_FragDepth`.

Kwalifikatory formatu bloku zmiennych jednorodnych

Kwalifikatory formatu bloku zmiennych jednorodnych mogą być użyte tylko do bloków zmiennych jednorodnych lub zmiennych znajdujących się w takich blokach, nie dotyczą natomiast deklaracji zmiennych jednorodnych poza blokiem. Identyfikatory kwalifikatora formatu bloku zmiennych jednorodnych są następujące: `shared`, `packed`, `std140`, `row_major`, `column_major` oraz `binding`.

Ważną cechą semantyczną kwalifikatorów bloku zmiennych jednorodnych jest brak modyfikacji sposobu użycia danych, modyfikowane jest jedynie ułożenie danych w pamięci. Przykładowo semantyka macierzy jest zawsze w układzie kolumnowym, bez względu na to jakie zostaną użyte identyfikatory.

Kwalifikatory formatu bloku zmiennych jednorodnych mogą być deklarowane globalnie, w deklaracji pojedynczego bloku lub w deklaracji pojedynczego elementu bloku. Każda globalna deklaracja modyfikuje poprzednią globalną deklarację, z zachowaniem opisanych dalej reguł nadpisywania identyfikatorów kwalifikatora. Globalna deklaracja wpływa na deklaracje wszystkich następujących po niej bloków zmiennych jednorodnych, chyba, że bloki te lub ich pola zawierają odmienne lokalne deklaracje formatu. Początkowa globalna deklaracja formatu bloku zmiennych jednorodnych wygląda następująco:

```
layout( shared, column_major ) uniform;
```

Pierwsza grupa identyfikatorów: `shared`, `packed` i `std140` określa zasady umieszczenia danych bloków w pamięci GPU. Identyfikator `shared` określa zasady współdzielenia pamięci na dane bloków o takiej samej definicji znajdujących się w różnych obiektach programu lub różnych programowalnych etapach renderingu. Umożliwia to użycie tego samego obiektu buforowego dla rożnych obiektów programu. Identyfikator `packed` także określa zasady wykorzystania pamięci przez kompilator i konsolidator GLSL. W takim przypadku współdzielenie pamięci nie jest gwarantowane, a decyzja o optymalizacji sposobu wykorzystania pamięci jest podejmowana przez kompilator lub konsolidator na podstawie sposobu użycia danych i innych zależnych od implementacji kryteriów. Ostatni z tej grupy identyfikator `std140` określa standardowy format danych bloku zmiennych jednorodnych, wprowadzony w wersji 1.40 GLSL (stąd pochodzi nazwa samego identyfikatora formatu). Podobnie jak w przypadku identyfikatora `shared`, także przy identyfikatorze `std140` pamięć bloków jest współdzielona. Identyfikatory `shared`, `packed` i `std140`, nie mogą być używane do deklaracji pojedynczego pola w bloku. Ich użycie ogranicza się do deklaracji globalnej lub deklaracji pojedynczego bloku, a ich wartości mogą być zmieniane przez kolejne deklaracje.

Druga grupa identyfikatorów `row_major` i `column_major` określa sposób ułożenia danych macierzy w pamięci. Nie ma to wpływu na ułożenie innych danych w bloku. Identyfikator `row_major` oznacza ułożenie danych macierzy kolejno wierszami, natomiast identyfikator `column_major` oznacza ułożenie danych macierzy kolejno kolumnami. Elementy w wierszu/kolumnie macierzy są ułożone w pamięci w sposób ciągły.

Ostatni nieomówiony identyfikator `binding` określa numer punktu wiązania obiektu bufora bloku zmiennych jednorodnych, służącego do pobierania i zapisu zmiennych bloku. Identyfikator ten może być stosowany wyłącznie do deklaracji bloku, nie można go użyć globalnie lub do pojedynczej zmiennej bloku. W przypadku braki identyfikatora binding w deklaracji bloku zmiennych jednorodnych przyjmowana jest domyślnie wartość punktu wiązania równa 0. Wartość punktu wiązania może być zmieniona przed konsolidacją programu przez API OpenGL.

W przypadku, gdy instancja bloku zmiennych jednorodnych jest tablicą, wówczas pierwszemu elementowi tablicy przyporządkowany jest punkt wiązania określony w identyfikatorze binding, a następnym elementom tablicy przyporządkowane są kolejne numery punktów wiązania. Błędem

zakończy się próba przypisaniu punktu wiązania o numerze mniejszym od 0 lub większym od pomniejszonej o jeden maksymalnej ilości punktów wiązania obsługiwanej przez daną implementację OpenGL.

W jednej deklaracji kwalifikatora można użyć więcej niż jednego identyfikatora z danej grupy, ale w efekcie zastosowany zostanie tylko jeden, ostatni z identyfikatorów z danej grupy, wymieniony w deklaracji. Analiza identyfikatorów w deklaracji kwalifikatora wykonywana jest od lewej do prawej. Przykładowo:

```
layout( row_major, column_major )
```

daje w rezultacie kwalifikator `column_major`. Poniżej znajdują się inne przykłady deklaracji kwalifikatorów formatu bloków zmiennych jednorodnych:

```
layout( shared, row_major ) uniform;  // domyślny format to
                                      // shared i row_major
layout( std140 ) uniform Transform    // format bloku to std140
{
    mat4 M1;  // macierz w formacie row_major
    layout( column_major ) mat4 M2;  // macierz w formacie
                                     // column_major
    mat3 N1;  // macierz w formacie row_major
};

uniform T2  // format tego bloku to shared
{
...
};

layout( column_major ) uniform T3  // format bloku to
{                                  // shared i column_major
    mat4 M3;  // macierz w formacie column_major
    layout( row_major ) mat4 m4;   // macierz w formacie row_major
    mat3 N2;  // macierz w formacie column_major
};
```

Kwalifikatory formatu zmiennych jednorodnych

Kwalifikatory formatu zmiennych jednorodnych dotyczą tylko następujących rodzajów zmiennych jednorodnych: uchwytów tekstur, obrazów tekstur oraz liczników atomowych i służą do powiązania zmiennych z określonym buforem lub jednostką tekstury.

Kwalifikatory formatu uchwytów tekstur

Kwalifikator formatu uchwytu tekstury zawiera jeden identyfikator `binding`, który określa numer jednostki tekstury. W przypadku, gdy uchwyt tekstury jest zadeklarowany bez kwalifikatora formatu, domyślnie przyjmuje się, że jest on powiązany z jednostką tekstury o numerze 0. Numer jednostki tekstury przed konsolidacją obiektu programu może zostać zmieniony z poziomu API OpenGL.

W przypadku, gdy kwalifikator formatu uchwytu tekstury został zastosowany do tablicy, numery jednostki tekstury przyporządkowane są kolejno, do poszczególnych elementów tablicy, przy czym pierwszy element tablicy otrzymuje numer jednostki tekstury wyspecyfikowany w kwalifikatorze.

Numer jednostki tekstury nie może być mniejszy od 0 i nie może przekraczać zależnego od implementacji pomniejszonego o jeden maksymalnego numeru jednostki tekstury . Przekroczenie powyższych limitów spowoduje zgłoszenie błędu na etapie kompilacji shadera.

```
// w jednej z jednostek kompilacji...
layout( binding = 3 ) uniform sampler2D s; // s przypisany do
                                           // jednostki tekstury
                                           // numer 3
// w innej jednostce kompilacji...
uniform sampler2D s;    // poprawnie, s jest dalej przypisany
                        // do jednostki tekstury nr 3

// w innej jednostce kompilacji...
layout( binding = 4 ) uniform sampler2D s; // błąd: s przypisany
                                           // do innej jednostki
                                           // tekstury niż nr 3
```

Kwalifikatory formatu obrazów tekstury

Kwalifikator formatu obrazów tekstury (typy podstawowe zawierające w nazwie słowo "`image`") posiada dwa rodzaje identyfikatorów. Pierwszy to `binding`, który ma analogiczne znaczenie jak w przypadku uchwytów tekstur. Drugi rodzaj identyfikatorów służy do określenia formatu danych związanych z deklarowaną zmienną. Dopuszczalna jest jedna z poniższych wartości podzielonych na trzy grupy: `rgba32f`, `rgba16f`, `rg32f`, `rg16f`, `r11f_g11f_b10f`, `r32f`, `r16f`, `rgba16`, `rgb10_a2`, `rgba8`, `rg16`, `rg8`, `r16`, `r8`, `rgba16_snorm`, `rgba8_snorm`, `rg16_snorm`, `rg8_snorm`, `r16_snorm`, `r8_snorm` (grupa formatów zmiennoprzecinkowych, typy z nazwą rozpoczynającą się od "`image`"), `rgba32i`, `rgba16i`, `rgba8i`, `rg32i`, `rg16i`, `rg8i`, `r32i`, `r16i`, `r8i` (grupa formatów całkowitych ze znakiem, typy z nazwą rozpoczynającą się od "`iimage`"), `rgba32ui`, `rgba16ui`, `rgb10_a2ui`, `rgba8ui`, `rg32ui`, `rg16ui`, `rg8ui`, `r32ui`, `r16ui` i `r8ui` (grupa formatów całkowitych bez znaku, typy z nazwą rozpoczynającą się od "`uimage`"). Błędem zakończy się deklaracja identyfikatora niezgodna z rodzajem typu zmiennej obrazu tekstury.

Zmienna obrazu tekstury używana do odczytu danych lub operacji atomowych musi zawierać kwalifikator formatu. Jego brak zakończy się zgłoszeniem błędu. Zmienne z kwalifikatorem pamięci `writeonly` (opis kwalifikatorów pamięci znajduje się w dalszej części tekstu) także muszą zawierać kwalifikator formatu. Zauważmy jednocześnie, że zmienna obrazu tekstury przekazana do odczytu jako parametr funkcji nie może zawierać kwalifikatora pamięci `writeonly`, ale musi zostać zadeklarowana z kwalifikatorem formatu.

Kwalifikatory formatu liczników atomowych

Kwalifikator formatu liczników atomowych posiada dwa rodzaje identyfikatorów. Pierwszy to `binding`, który określa powiązanie zmiennej z określonym punktem wiązania bufora. Drugi identyfikator `offset` wskazuje położenie danej zmiennej w buforze wyrażone w podstawowych jednostkach maszyny BMU (ang. *basic machine units*), czyli najczęściej w bajtach. W pierwszym przykładzie:

```
layout( binding = 2, offset = 4 ) uniform atomic_uint a;
```

licznik atomowy `a` jest powiązana do punktu wiązania numer 2, a jej położenie w buforze określone jest przesunięciem w danych bufora równym 4. Następna deklaracja zmiennej korzystającej z tego

samego punktu wiązania, w przypadku braku identyfikatora `offset`, automatycznie przyjmuje powiększoną o 4 jego wartość w stosunku do poprzednio zadeklarowanej zmiennej. Przykładowo:

```
layout( binding = 2 ) uniform atomic_uint bar;
```

zmienna `bar` ma określone przesunięcie o wartości 8. Możliwa jest jednoczesna deklaracja wielu zmiennych z jednym kwalifikatorem formatu. Każdy z liczników atomowych, licząc od lewej do prawej, otrzyma kolejne wartości przesunięcia.

Początkowe wartości numeru punktu wiązania oraz przesunięcia (położenia) w danych bufora są równe 0. Początkowa wartość przesunięcia dla wybranego punktu wiązania może być określona globalnie (bez deklaracji zmiennej) np. w poniższy sposób:

```
layout( binding = 2, offset = 4 ) uniform atomic_uint;
```

Następująca dalej deklaracja zmiennej typu `atomic_uint`, przypisana do punktu wiązania nr 2 korzysta z globalnie określonej wartości przesunięcia równej 4. Jednocześnie zauważmy, że opisana reguła nie dotyczy samego numeru punktu wiązania:

```
layout( binding = 2 ) uniform atomic_uint bar; // offset równy 4
~~layout( offset = 8 ) uniform atomic_uint bar;~~  // błąd, brak
                                                  // domyślnego numeru
                                                  // punktu wiązania
```

Licznik atomowy mogą oczywiście dzielić ten sam numer punktu wiązania, ale wartości przesunięcia, zarówno określone bezpośrednio lub w sposób domyślny, nie mogą się pokrywać. Poniżej przedstawiamy przykłady poprawnych i błędnych deklaracji liczników atomowych:

```
layout( binding = 3, offset = 4 ) uniform atomic_uint a; // offset=4
layout( binding = 2 ) uniform atomic_uint b;             // offset=0
layout( binding = 3 ) uniform atomic_uint c;             // offset=8
layout( binding = 2 ) uniform atomic_uint d;             // offset=4

~~layout( offset = 4 ) uniform atomic_uint x;~~ // błąd, wymagany numer
                                                // punkt wiązania
layout( binding = 1, offset = 0 ) uniform atomic_uint a; //poprawnie
layout( binding = 2, offset = 0 ) uniform atomic_uint b; //poprawnie
// błąd, wartość offset pokrywa się z licznikami a i c
~~layout( binding = 1, offset = 0 ) uniform atomic_uint c;~~
// błąd, wartość offset pokrywa się z licznikiem a
~~layout (binding=1, offset=2) uniform atomic_uint d;~~
```

Końcowo dodajmy, że błędem kompilacji zakończy sie wskazanie numeru punktu wiązania większego lub równego wartości wbudowanej zmiennej `gl_MaxAtomicCounterBindings` określającej maksymalną ilość punktów wiązania liczników atomowych w danej implementacji OpenGL.

Wariancja i kwalifikator `invariant`

Wariancją (ang. *variance*) nazywamy potencjalne uzyskiwanie rozbieżnych wyników tego samego wyrażenia w różnych obiektach programu. Przykładowo dwa shadery wierzchołków, w różnych obiektach programu, zapisują wartość zmiennej `gl_Position` przy użyciu tego samego wyrażenia z takimi samymi wartościami wejściowymi. Możliwa jest sytuacja, że przy niezależnej kompilacji wartości zapisane do `gl_Position` nie będą dokładnie takie same. Opisany problem

ten może w szczególności wystąpić w obniżonej dokładności geometrii w algorytmach wieloprzebiegowych. Jeżeli zjawisko rozbieżności wyników dla danej zmiennej wyjściowej nie występuje, mówimy, że jest to zmienna inwariantna.

Kwalifikator `invariant`

Jeżeli chcemy zadeklarować zmienną wyjściową jako inwariantną wykorzystujemy do tego celu kwalifikator inwariancji `invariant`. Kwalifikator ten można stosować do wcześniej zadeklarowanej zmiennej lub jako część deklaracji nowej zmiennej:

```
invariant gl_Position;  // deklarowanie wbudowanej zmiennej
                        // gl_Position jako inwariantnej

out vec3 Color;
invariant Color;  // zadeklarowanie istniejącej zmiennej
                  // Color jako inwariantnej

invariant centroid out vec3 SecColor;  // nowa zmienna zadeklarowana
                                       // jako zmienna inwariantna
```

Zmiennymi inwariantnymi mogą być tylko zmienne wyjściowe shadera, włączając w to zmienne, które są jednocześnie wejściem w shaderze z następnego etapu potoku renderingu. Kwalifikator można stosować zarówno do zmiennych wbudowanych jak i zmiennych definiowanych przez użytkownika. W przypadku zmiennych łączących shadery z dwóch programowalnych etapów renderingu, kwalifikator `invariant` musi być użyty w obu shaderach, zarówno w deklaracji zmiennej wejściowej jak i deklaracji zmiennej wejściowej. Kwalifikator `invariant` może być użyty także dla listy wielu wcześniej zadeklarowanych zmiennych, których nazwy rozdzielone są przecinkami. Deklaracja inwariancji dotyczy wyłącznie zmiennych globalnych i musi wystąpić przez pierwszym użyciem zmiennej.

Dla zagwarantowania inwariancji zmiennej wyjściowej w dwóch różnych obiektach programów potrzebne są jeszcze dodatkowe warunki dotyczące zgodności kontroli przepływu i danych wejściowych, które szczegółowo wymieniamy poniżej:

- zmienna wyjściowa jest zadeklarowana z kwalifikatorem `invariant` w obu programach,
- wartości przypisane do zmiennej wyjściowej muszą pochodzić od wyrażeń zbudowanych ze zmiennych wejściowych o takich samych wartościach, z zachowaniem takiej samej kontroli przepływu; dodatkowo wyrażenia te muszą zachowywać: zgodność operatorów i operandów, kolejność obliczeń, typy danych oraz domyślne lub globalne kwalifikatory precyzji,
- funkcje obsługujące tekstury wykorzystywane do obliczeń zmiennej wyjściowej muszą działać na teksturach o takim samym formacie, wartościach tekseli, oraz przy zachowaniu takich samych ustawień filtracji,
- całość kontroli przepływu i danych mających wpływ na wartość zmiennej wyjściowej musi znajdować się w pojedynczej jednostce kompilacji (obiekcie shadera).

Przy porównywaniu zgodności wyjściowych/wejściowych zmiennych inwariantnych shaderów z następujących po sobie programowalnych etapach renderingu w zmiennej wejściowej nie jest wymagane występowanie kwalifikatora `invariant`.

Domyślna wariancja

Początkowo wszystkie zmienne wyjściowe są objęte wariancją. GLSL umożliwia globalne wymuszenie inwariancji wszystkich zmiennych wyjściowych przy użyciu następującej dyrektywy preprocesora:

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

którą należy umieścić przed wszystkim deklaracjami zmiennych i funkcji w shaderze. Dyrektywa nie może być użyta w shaderze fragmentów.

Ogólnie rzecz biorąc, inwariancja jest realizowana kosztem elastyczności w optymalizacji, może więc wpłynąć negatywnie na wydajność. Stąd stosowanie powyższej dyrektywy powinno być ograniczone jako pomoc przy testach i podjęciu decyzji, które zmienne wyjściowe powinny być inwariantne.

Inwariancja w wyrażeniach stałych

Specyfikacja GLSL gwarantuje zachowanie inwariancji dla wyrażeń stałych. Dane wyrażenie stałe musi być oczywiście obliczone w ten sam sposób zarówno ponownie w jednym shaderze jak i w różnych shaderach. Zasada ta dotyczy także shaderów różnego typu.

Kwalifikatory interpolacji

Zmienne wejściowe i wyjściowe, które podlegają interpolacji, mogą być opcjonalnie modyfikowane przy użyciu kwalifikatorów interpolacji wymienionych w Tabela 8.

Kwalifikator interpolacji	Opis
`smooth`	Interpolacja z korekcją perspektywy.
`flat`	Brak interpolacji zmiennej.
`noperspective`	Interpolacja liniowa.

Tabela 8 Kwalifikatory interpolacji.

Kwalifikatory interpolacji mogą być łączone z pomocniczymi kwalifikatorami przechowywania `centroid` i `sample`, nie jest natomiast dopuszczalne łączenie kwalifikatorów interpolacji z pomocniczym kwalifikatorem przechowywania `patch`.

Dopuszczalne jest użycie tylko jednego z wymienionych kwalifikatorów interpolacji. Typ użytych kwalifikatorów przechowywania, interpolacji oraz opisywanego wcześniej kwalifikatora `invariant`, musi być zgodny w deklaracjach zmiennych o takiej samej nazwie we wszystkich razem konsolidowanych shaderach. W przeciwnym wypadku nastąpi błąd konsolidacji. Przy porównywaniu zgodności zmiennych wyjściowych/wejściowych shaderów z następujących po sobie programowalnych etapach renderingu wymagana jest zgodność kwalifikatorów interpolacji (lub ich brak), w przeciwnym wypadku zmienne te nie tworzą interfejsu.

Kwalifikator `flat`

Kwalifikator `flat` wskazuje, że dana zmienna nie będzie interpolowana. Oznacza to, że będzie ona miała taką samą wartość dla każdego fragmentu, pochodzącą z podstawowego wierzchołka prymitywu. Zmienna bez interpolacji może także posiadać kwalifikator `centroid`, lub `sample`, które oznaczają dokładnie to samo, co `flat`.

Kwalifikator `smooth`

Zmienna z kwalifikatorem `smooth` podlega interpolacji z korekcją perspektywy. Popatrzmy jak obliczana jest tak interpolowana wartość na początku na przykładzie odcinka. Przejmijmy, że współrzędne okienkowe środka wygenerowanego fragmentu mają wartość $P_r = (x_d, y_d)$, natomiast współrzędne wierzchołków - końców odcinka, których atrybuty interpolujemy to odpowiednio $P_a = (x_a, y_a)$ i $P_b = (x_b, y_b)$. Wówczas współczynnik interpolacji obliczamy wg równania wektorowego:

$$t = \frac{(P_r - P_a) \cdot (P_b - P_a)}{|P_b - P_a|^2}$$

które w postaci skalarnej wygląda następująco:

$$t = \frac{(x_d - x_a)(x_b - x_a) + (y_d - y_a)(y_b - y_a)}{(x_b - x_a)^2 + (y_b - y_a)^2}$$

Jak łatwo zauważyć współczynnik interpolacji przyjmuje wartości z przedziału $\langle 0; 1 \rangle$. Wartość $t = 0$ otrzymamy, gdy $P_r = P_a$, natomiast wartość $t = 1$ otrzymamy, gdy $P_r = P_b$. Odpowiada to wartości atrybutu z odpowiednio początkowego i końcowego interpolowanych wierzchołków. Współczynnik interpolacji jest następnie używany do obliczenia interpolowanej wartości atrybutu wierzchołka dla danego fragmentu, przy użyciu korekcji perspektywy:

$$f = \frac{(1-t)\frac{f_a}{w_a} + t\frac{f_b}{w_b}}{(1-t)\frac{1}{w_a} + t\frac{1}{w_b}}$$

gdzie f_a i f_b są odpowiednio wartościami atrybutu początkowego i końcowego interpolowanych wierzchołków, a w_a i w_b składowymi w współrzędnych obcinania powyższych wierzchołków.

Odmiennie jest natomiast traktowana wartość głębokości, która jest interpolowana w następujący sposób:

$$z = (1-t)z_a + tz_b$$

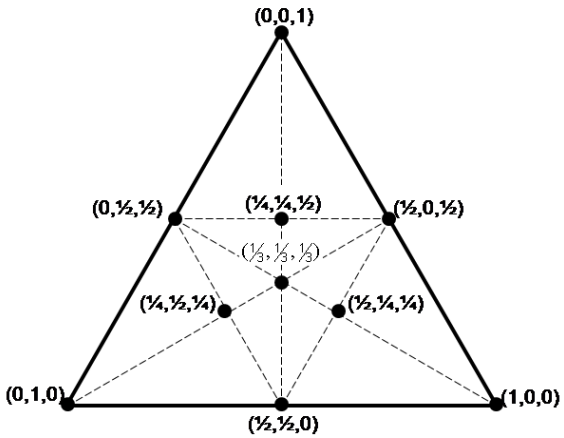
gdzie z_a i z_b to odpowiednio wartości głębokości wierzchołków P_a i P_b.

Nieco bardziej skomplikowana jest interpolacja zmiennych w przypadku, gdy zmienna ta jest użyta do opisu trzech wierzchołków trójkąta. do tego celu używane są tzw. współrzędne barycentryczne trójkąta (ang. *barycentric coordinates*), które nazywane są także lokalnymi współrzędnymi trójkąta.

Współrzędne barycentryczne w trójkątcie to trzy liczby u, v, w, gdzie $u + v + w = 1$ i $u, v, w \geq 0$, przy czym punkt $u = v = w = \frac{1}{3}$ wyznacza jego środek ciężkości (barycentrum) względem wierzchołków. Dowolny punkt P leżący wewnątrz lub na krawędzi trójkąta opisanego przez wierzchołki P_a, P_b i P_c określamy we współrzędnych barycentrycznych równaniem:

$$P = uP_a + vP_b + wP_c$$

Wybrane współrzędne barycentryczne w trójkącie przedstawia Rysunek 1.



Rysunek 1 Współrzędne barycentryczne w trójkącie.

Znając współrzędne kartezjańskie wierzchołków trójkąta $P_1 = (x_1, y_1, z_1)$, $P_2 = (x_2, y_2, z_2)$ i $P_3 = (x_3, y_3, z_3)$ możemy łatwo przeliczać współrzędne kartezjańskie dowolnego punktu $P = (x, y, z)$ trójkąta na jego współrzędne barycentryczne $P = (u, v, w)$ i odwrotnie. Przekształcenie to opisuje poniższe równanie macierzowe:

$$\begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{pmatrix} \begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Współrzędne barycentryczne można także wyrazić jako iloczyny pól następujących trójkątów:

$$u = \frac{\Delta\,(PP_2P_3)}{\Delta\,(P_1P_2P_3)}$$

$$v = \frac{\Delta\,(PP_1P_3)}{\Delta\,(P_1P_2P_3)}$$

$$w = \frac{\Delta\,(PP_1P_2)}{\Delta\,(P_1P_2P_3)}$$

Warto jednocześnie zauważyć, że do jednoznacznego opisu punktu wewnątrz trójkąta wystarczą dwie z trzech współrzędnych barycentrycznych. Trzecią współrzędną możemy otrzymać z założenia o nieujemności współrzędnych: $w = 1 - u - v$.

Popatrzmy teraz jak współrzędne barycentryczne są używane do interpolacji z korekcją perspektywy. Wierzchołkom P_a, P_b i P_c przyporządkowane są odpowiednio atrybuty f_a, f_b i f_c. Interpolowana wartość atrybutu wierzchołka dla danego fragmentu wynosi:

$$f = \frac{u\frac{f_a}{w_a} + v\frac{f_b}{w_b} + w\frac{f_c}{w_c}}{\frac{u}{w_a} + \frac{v}{w_b} + \frac{w}{w_c}}$$

gdzie w_a, w_b i w_c są składowymi w współrzędnych obcinania powyższych wierzchołków.

Odrębnie są traktowane wartości głębokości, które są interpolowane w następujący sposób:

$$z = uz_a + vz_b + wz_c$$

gdzie z_a, z_b i z_c to odpowiednio wartości głębokości wierzchołków P_a, P_b i P_c.

Kwalifikator `noperspective`

W przypadku użycia kwalifikatora `noperspective`, interpolacja dla odcinków wykonywana jest liniowo w przestrzeni okna zgodnie z poniższym równaniem (oznaczenia jak poprzednio):

$$f = (1 - t)f_a + tf_b$$

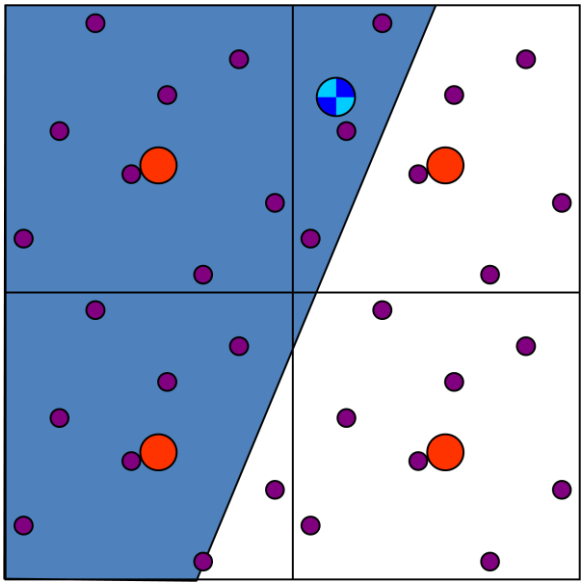
natomiast w przypadku trójkątów interpolacja jest także liniowa w przestrzeni oka, przy czym określa ją równanie (oznaczenia jak poprzednio):

$$f = uf_a + vf_b + wf_c$$

Interpolacja a wielopróbkowanie

W przypadku, gdy wielopróbkowanie jest wyłączone, oraz gdy zmienna wejściowa shadera fragmentów nie zawiera pomocniczego kwalifikatora przechowywania `centroid` lub `sample`, opisana wyżej interpolacja obliczana jest dla dowolnego miejsca wewnątrz piksela, a wartości próbek są jednakowe. Może to być przyczyną różnego rodzaju przekłamań (artefaktów) w renderingu.

W przypadku aktywnego wielopróbkowania pomocnicze kwalifikatory przechowywania `centroid` i `sample` służą do kontroli położenia i częstotliwości próbkowania zmiennych wejściowych shadera fragmentów. Kwalifikator `centroid` wskazuje, że do obliczeń w wielopróbkowaniu wykorzystywany jest punkt, który znajduje się jednocześnie wewnątrz piksela i renderowanego prymitywu (patrz Rysunek 2), a wszystkie próbki przyjmują takie same wartości. W granicznych pikselach obliczenia interpolacji z kwalifikatorem `centroid` mogą być mniej dokładne niż dla zmiennych bez tego kwalifikatora.



Rysunek 2 Rasteryzacja wielokąta w wielopróbkowaniu. Czerwone kropki oznaczają środek piksela. Granatowe kropki symbolizuja próbki. Centroid wielokąta dla jednego z pikseli jest oznaczony kółkiem z szachownicą.

Drugi z wymienionych kwalifikatorów, tj. kwalifikator `sample` określa, że przy włączonym wielopróbkowaniu każda próbka przypadająca na dany piksel może posiadać inną wartość, a końcowa wartość musi być próbkowana z uwzględnieniem położenia poszczególnych próbek.

Kwalifikatory precyzji i dokładność obliczeń

Kwalifikatory precyzji: `highp`, `mediump` i `lowp` zostały dodane do języka GLSL w celu zapewnienia zgodności składniowej z shaderami kompatybilnymi z OpenGL ES 2.0, nie mają natomiast żadnego znaczenia funkcjonalnego. Oczywiście funkcjonalność taką może wprowadzić stosowne rozszerzenie.

Kwalifikatory precyzji

Kwalifikatory precyzji zostały wymienione w Tabela 9.

Kwalifikator precyzji	Opis
`highp`	Brak funkcjonalności. W OpenGL ES 2.0 najwyższy zakres precyzji zmiennych.
`mediump`	Brak funkcjonalności. W OpenGL ES 2.0 zakres precyzji zmiennych mniejszy od `highp` i większy od `lowp`.

`lowp`	Brak funkcjonalności. W OpenGL ES 2.0 najniższy zakres precyzji zmiennych.

Tabela 9 Kwalifikatory precyzji.

W przypadku kwalifikatorów precyzji GLSL zachowuje identyczną składnię jak w OpenGL ES, w której kwalifikator precyzji poprzedza deklarację zmiennej całkowitej lub zmiennoprzecinkowej. Poniżej znajdują się przykładowe deklaracje:

```
lowp float color;
out mediump vec2 P;
lowp ivec2 foo( lowp mat3 );
highp mat4 m;
```

Literały stałe nie mogą zawierać kwalifikatorów precyzji. Dotyczy to także zmiennych typu `bool`. Kwalifikatorów precyzji nie można także stosować do argumentów konstruktorów typów zmiennoprzecinkowych i całkowitych. Kwalifikatory precyzji nie mają także wpływu na podstawowy typ zmiennej.

Przy porównywaniu zgodności zmiennych wyjściowych/wejściowych shaderów z następujących po sobie programowalnych etapach renderingu nie jest wymagana zgodność kwalifikatorów precyzji.

Domyślne kwalifikatory precyzji

GLSL umożliwia definiowanie domyślnego kwalifikatora precyzji dla typów `int` lub `float`, co ma przełożenie na wszystkie typy i składowe zmiennoprzecinkowe i całkowite (skalary, wektory i macierze), definicje zmiennych globalnych, wartości zwracane przez funkcje oraz zmienne lokalne. Domyślny kwalifikator precyzji nie jest stosowany w sytuacjach, gdy deklaracja zmiennej już zawiera kwalifikator precyzji. Dopuszczalne jest wielokrotne definiowane domyślnego kwalifikatora precyzji w ramach jednego zasięgu definicji. W takim przypadku nowa definicja zastępuje poprzednią.

Domyślne kwalifikatory precyzji dla shaderów wierzchołków, shaderów kontroli i ewaluacji teselacji oraz shaderów geometrii zdefiniowane są następująco:

```
precision highp float;
precision highp int;
```

Analogiczne definicje dla shaderów fragmentów są nieco inne:

```
precision mediump int;
precision highp float;
```

Dla wszystkich rodzajów programów cieniowania dostępna jest wbudowane makro `GL_FRAGMENT_PRECISION_HIGH`, którego wartość 1 oznacza dostępność kwalifikatorów precyzji:

```
#define GL_FRAGMENT_PRECISION_HIGH 1
```

Zakresy wartości

Dokładność liczb zmiennoprzecinkowych pojedynczej i podwójnej precyzji jest określona przez standard IEEE 754 odpowiednio dla 32- i 64-bitowych liczb zmiennoprzecinkowych. Obsługiwane jest dzielenie przez 0, które zwraca wartość Inf (nieskończoność). Możliwa, choć niewymagana jest obsługa dodatniego i ujemnego 0, to samo dotyczy obsługi $+Inf$ i $-Inf$. Obsługa wyrażeń nie będących liczbą NaN (ang. *not a number*) także jest opcjonalna. Dotyczy to także funkcji operujących na NaN, które nie muszą zwracać NaN jako wyniku.

Dokładność obliczeń zmiennoprzecinkowych

Dokładność obliczeń można określić poprzez maksymalny względny błąd wyrażany w jednostkach ULP (ang. *units in last place*), czyli odległości pomiędzy dwiema najbliższymi liczbami (granulacja). Dokładność obliczeń zmiennoprzecinkowych na liczbach pojedynczej precyzji dla wybranych wyrażeń i funkcji przedstawia Tabela 10. Dokładność operacji zmiennoprzecinkowych na liczbach podwójnej precyzji jest co najmniej taka sama jak dla liczb pojedynczej precyzji.

Operacja	Precyzja
`a+b`, `a-b`, `a*b`	Prawidłowe zaokrąglanie.
`<`, `<=`, `==`, `>`, `>=`	Poprawny wynik.
`a/b`, `1.0/b`	2,5 ULP dla `b` z przedziału $\langle 2^{-126}; 2^{126}\rangle$
`a*b+c`	Prawidłowe zaokrąglanie pojedynczej operacji lub sekwencji dwóch poprawnie zaokrąglanych operacji.
fmat()	Pochodna po `a*b+c`.
pow(x, y)	Pochodna po $\exp2(x * \log2(y))$.
exp(x), **exp2**(x)	$3 + 2 * \lvert x\rvert$ ULP.
log(x), **log2**(x)	3 ULP poza przedziałem $\langle 0{,}5; 2{,}0\rangle$. Błąd bezwzględny mniejszy od 2^{-21} w przedziale $\langle 0{,}5; 2{,}0\rangle$.
sqrt()	Pochodna po $1.0/\text{inversesqrt}()$.
inversesqrt()	2 ULP.
jawna i niejawna konwersja pomiędzy typami	Prawidłowe zaokrąglanie.

Tabela 10 Dokładność obliczeń zmiennoprzecinkowych na liczbach pojedynczej precyzji w GLSL.

Wbudowane funkcje określone w specyfikacja języka GLSL i opisane w dalszej części niniejszego tekstu, zbudowane z operacji wymienionych w Tabela 10, dziedziczą błędy opisane w tabeli błędy. Dotyczy to przykładowo funkcji geometrycznych, ogólnych oraz wielu funkcji macierzowych. Funkcje niewymienione w Tabela 10 i nie zdefiniowane jako operacje w niej wymienione, mają nieokreśloną precyzję.

Kwalifikator `precise`

Kwalifikator `precise` określa precyzję obliczeń zmiennoprzecinkowych, w sytuacjach, gdy dokładność obliczeń jest bardziej istotna niż ewentualna optymalizacja, mogąca spowodować pogorszenie jakości obliczeń. Przykładowo wyrażenie zmiennoprzecinkowe:

```
result = (a * b) + (c * d);
```

może zostać zoptymalizowane w następujący sposób: zamiast dwóch mnożeń i jednego dodawania, procesor GPU wykona jedno mnożenie i jedną połączoną operację mnożenia z dodawaniem, która może dać inny wynik niż wyrażenie nie poddane optymalizacji. Kwalifikator `precise` gwarantuje, że wyrażenia określające wartość zmiennej zadeklarowanej z jego użyciem będą wykonywane w kolejności zgodnej z zapisem kodu źródłowego shadera, oczywiście z zachowaniem priorytetów operatorów.

Kwalifikator `precise` można zastosować razem z deklaracją zmiennej, lub odrębnie dla wcześniej zadeklarowanej zmiennej (podobnie jak kwalifikator `invariant`):

```
precise out vec4 position;  // deklaracja zmiennej
                            // z kwalifikatorem precise

out vec3 Color;             // deklaracja standardowej zmiennej
precise Color;              // zmienna Color otrzymuje
                            // kwalifikator precise
```

W przypadku zmiennych będących wyjściem jednego shadera i jednocześnie wejściem drugiego shadera, kwalifikator `precise` nie musi być użyty w stosunku do obu tych zmiennych.

Poniżej zamieszczamy kilka przykładów użycia kwalifikatora `precise`:

```
in vec4 a, b, c, d;
precise out vec4 v;

float func( float e, float f, float g, float h )
{
    return (e*f) + (g*h); // brak nakazu kolejności lub
                          // zwartości operatorów
}

float func2( float e, float f, float g, float h )
{
    precise float result = (e*f) + (g*h); // zapewnienie precyzji
                                          // dla dwóch mnożeń
                                          // i zwracanego wyniku
}

float func3( float i, float j, precise out float k )
{
    k = i * i + j;  // precyzja, z uwagi na deklarację k
}

void main()
{
    vec4 r = vec3( a * b ); // precyzja użyta do obliczenia v.xyz
    vec4 s = vec3( c * d ); // precyzja użyta do obliczenia v.xyz
    v.xyz = r + s;                       // precyzja
    v.w = (a.w * b.w) + (c.w * d.w);     // precyzja
    v.x = func( a.x, b.x, c.x, d.x );    // wartości obliczone w func
                                         // nie mają kwalifikatora
                                         // precise
    v.x = func2( a.x, b.x, c.x, d.x );   // precyzja
    func3( a.x * b.x, c.x * d.x, v.x ); // precyzja
}
```

Kwalifikatory pamięci

Kwalifikatory pamięci, wymienione w Tabela 11 dostępne są wyłącznie dla zmiennych jednorodnych będących obrazami tekstur (typ podstawowy zawierających wyraz "`image`").

Kwalifikator pamięci	Opis
`coherent`	Zmienna, której odczyt i zapis są spójne z odczytem i zapisem w innych wywołaniach shadera.
`volatile`	Zmienna, której wartości mogą ulec zmianie w dowolnym momencie wykonywania shadera, przez inny program cieniowania niż bieżąco wywoływany shader.
`restrict`	Zmienna, której użycie sprowadza się tylko do odczytu i zapisy bazowej pamięci w danym shaderze.
`readonly`	Zmienna, która może być użyta do odczytu bazowej pamięci, ale nie może służyć do zapisu bazowej pamięci.

`writeonly`	Zmienna, która może służyć do zapisu bazowej pamięci, ale nie może być użyta do odczytu bazowej pamięci.

Tabela 11 Kwalifikatory pamięci.

Użycie kwalifikatora przechowywania `const` w zmiennej obrazu tekstury oznacza stałość zmiennej, nie zaś niezmienność pamięci, do której zmienna się odwołuje. Stałość pamięci zmiennej zapewnia natomiast kwalifikator `readonly`.

Kwalifikator `coherent`

Kwalifikator `coherent` zapewnia spójność dostępu do pamięci zmiennej obrazu tekstury w różnych wywołaniach shadera. W szczególności, podczas odczytu zmiennej zadeklarowanej z kwalifikatorem `coherent`, zwracana wartość będzie odzwierciedlać wynik uprzedniego zapisu wykonanego przez inne wywołanie shadera. Trzeba jednak pamiętać, że zapis i odczyt takich zmiennych dokonywany jest w dużej mierze w nieokreślonej kolejności. Użycie wbudowanej funkcji `memoryBarrier` zapewnienia kompletność i względną kolejność dostępu do pamięci przy jednym wywołaniu shadera.

Przy dostępie do pamięci przez zmienną zadeklarowaną bez kwalifikatora `coherent`, pamięć używana przez shader może być w danej implementacji umieszona w pamięci podręcznej do przyszłej obsługi tego samego adresu. Zapisy w pamięci podręcznej mogą być tak obsługiwane, że nie będą widoczne dla innych wywołań shadera odwołujących się do tej samej pamięci (zmiennej). Zatem realizacja pamięci podręcznej może spowodować, że zmienne zadeklarowane bez kwalifikatora `coherent` nie są przydatne do obsługi komunikacji pomiędzy wywołaniami shadera, przy czym jednak dostęp do takich zmiennych może zwiększyć wydajność programu.

Kwalifikator `volatile`

Dostęp do zmiennej z kwalifikatorem pamięci `volatile` jest wykonywany przy założeniu, że pamięć może być odczytywana lub zapisywana w dowolnym momencie wywołania shadera, przez inne programy cieniowania niż bieżąco wykonywany shader. Kiedy taka zmienna jest odczytywana, jej wartość musi być ponownie pobrana z bazowej pamięci, nawet jeśli wywołanie shadera wykonało wcześniej odczyt wartości z tej samej pamięci. Natomiast, gdy taka zmienna jest zapisywana, jej wartość musi być zapisana do bazowej pamięci, nawet jeśli kompilator może jednoznacznie stwierdzić, że jej wartość zostanie zastąpiona przez kolejny zapis. Ponieważ odczyt i zapis zmiennej może być również wykonywany przez inne wywołanie tego samego shadera, zmienne zadeklarowane z kwalifikatorem `volatile` są automatycznie jednocześnie traktowane jak zmienne z kwalifikatorem `coherent`.

Kwalifikator `restrict`

Zmienne z kwalifikatorem `restrict` są kompilowane z założeniem, że ich użycie do zapisu i odczytu nie będzie wykonywane w innym shaderze. Aplikacja jest odpowiedzialna za to, aby do pamięci obrazowanej przez zmienną z kwalifikatorem `restrict` nie było żadnej innej referencji, w przeciwnym wypadku dostęp do takiej zmiennej da niezdefiniowany rezultat.

Kwalifikator `readonly`

Kwalifikator `readonly` określa, że dana zmienna może służyć wyłącznie do odczytu bazowej pamięci, bez możliwości jej zapisu. Próba zapisu pamięci przez funkcję `imageStore` lub inną wbudowaną funkcję spowoduje zgłoszenie błędu.

Kwalifikator `writeonly`

Kwalifikator `writeonly` określa, że dana zmienna może służyć wyłącznie do zapisu bazowej pamięci, bez możliwości jej odczytu. Próba odczytu pamięci przez funkcję `imageLoad` lub inną wbudowaną funkcję spowoduje zgłoszenie błędu.

Instrukcje i struktura programu

Podstawowymi blokami w języku GLSL są:

- instrukcje i deklaracje,
- definicje funkcji,
- selekcje (`if`/`else`, `switch`/`case`/`default`),
- iteracje (`for`, `while`, `do`/`while`),
- skoki (`discard`, `return`, `break`, `continue`).

Definiowanie funkcji

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

```
returnType functionName( typ0 arg0, typ1 arg2, ..., typN argN );
```

natomiast definicja funkcji przyjmuje postać:

```
returnType functionName ( typ0 arg0, typ1 arg2, ..., typN argN )
{
    // wykonanie obliczeń
    return returnValue;
}
```

Każda funkcja musi mieć określony zwracany typ. Jeżeli typ `returnValue` jest różny od `returnType` musi być dopuszczalna domyślna konwersja pomiędzy tymi typami. W przeciwnym wypadku zostanie zgłoszony błąd kompilacji. Każdy z argumentów funkcji musi mieć określony typ, który może być poprzedzony opcjonalnym kwalifikatorem parametru (`in`, `out`, `inout`), opcjonalnym kwalifikatorem przechowywania `const` oznaczającym niezmienność argumentu, lub także opcjonalnym kwalifikatorem precyzji. Kwalifikator precyzji może być także użyty w zwracanym typie przez funkcję. Jako argument funkcji oraz zwracany typ dopuszczalne są także tablice, muszą mieć jednak określony rozmiar. Ponadto, jako argument i zwracany typ funkcji język GLSL dopuszcza także struktury.

Wszystkie funkcje, przed pierwszym użyciem muszą być zadeklarowane przez prototyp lub zdefiniowane. Jeżeli funkcja nie zwraca żadnych wartości musi być zadeklarowana jako `void`. Funkcje bezparametrowe można także wywołać z idiomem „`(void)`”. Funkcje definiowane przez użytkownika mogą mieć wiele deklaracji, ale tylko jedną definicję.

Kwalifikatory parametrów funkcji

Parametry funkcji mogą posiadać jeden z kwalifikatorów parametru, które wymienione są w Tabela 12:

Kwalifikator parametru	Opis
brak (domyślnie)	To samo, co kwalifikator `in`.
`const`	Parametr o stałej wartości, który nie może być zmieniany wewnątrz funkcji.
`in`	Parametr wejściowy funkcji. Wszelkie zmiany tego parametru wewnątrz funkcji nie mają wpływu na jego wartość poza funkcją.
`out`	Parametr wyjściowy funkcji. Parametry z tym kwalifikatorem nie wymagają przekazywania do funkcji żadnej konkretnej wartości.
`inout`	Parametr funkcji jest parametrem zarówno wejściowym jak i wyjściowym. Wszelkie zmiany wartości tego parametru będą miały wpływ na jego wartość poza funkcją.

Tabela 12 Kwalifikatory parametru funkcji.

W parametrach funkcji kwalifikator parametru zawsze poprzedza kwalifikator precyzji. W prototypie funkcji (deklaracji) można pominąć nazwy poszczególnych parametrów i użyć tylko samych kwalifikatorów parametrów funkcji.

Zmienne obrazów tekstur zadeklarowane z jednym z kwalifikatorów pamięci `coherent`, `volatile`, `restrict`, `readonly` lub `writeonly` nie mogą być przekazywane jako parametry funkcji, jeżeli odpowiedni kwalifikator nie występuje w deklaracji parametru funkcji. Dopuszczalne jest wystąpienie dodatkowych kwalifikatorów w deklaracji parametru funkcji, ale nie może być ich mniej niż w deklaracji zmiennej. Poniżej przedstawiamy dwa przykłady:

```
vec4 funcA( restrict image2D a )
{
    ...
}
vec4 funcB( image2D a )
{
    ...
}

layout( rgba32f ) uniform image2D img1;
layout( rgba32f ) coherent uniform image2D img2;

funcA( img1 );   // poprawnie, dodanie kwalifikatora
                 // restrict jest dopuszczalne
~~funcB( img2 );~~   // błąd, "nadmiarowy" kwalifikator
                 // coherent nie jest dopuszczalny
```

Dodamy jeszcze, że kwalifikatory formatu nie mogą być deklarowane w parametrach funkcji, nie są one jednak używane przy badaniu zgodność jej argumentów.

Przeładowanie nazw funkcji

Nazwy funkcji można w języku GLSL przeładowywać. Warunkiem poprawności tej operacji jest występowanie różnic w liście ich parametrów. Przeładowane funkcje nie mogą się różnić wyłącznie kwalifikatorami parametrów lub zwracanym typem. Oto kilka przykładów poprawnego i błędnego przeładowania funkcji:

```
vec4 f( in vec4 x, out vec4 y );
vec4 f( in vec4 x, out uvec4 y );      // poprawnie, różne typy
                                       // argumentów funkcji
vec4 f( in ivec4 x, out uvec4 y );     // poprawnie, różne typy
                                       // argumentów funkcji

~~int f( in vec4 x, out ivec4 y );~~       // błąd, różnica tylko
                                       // w zwracanym typie
~~vec4 f( in vec4 x, in vec4 y );~~        // błąd, różnica tylko
                                       // w kwalifikatorze y
~~vec4 f( const in vec4 x, out vec4 y );~~ // błąd, różnica tylko
                                       // w kwalifikatorze x
```

Podczas wywoływania funkcji przeładowanej w pierwszej kolejności wybierana jest funkcja, której argumenty są w pełni zgodne z parametrami przekazanymi w wywołaniu. Jeżeli taka funkcja nie jest dostępna stosowane są wcześniej opisane zasady domyślnej konwersji typów. W przypadku użycia konwersji typów, w pierwszej kolejności wybierana jest funkcja, w której następuje konwersja argumentów z typu `float` do `double`, a w dalszej kolejności funkcja z konwersją argumentów `int` lub `uint` do `float`. Jako ostatnia w kolejce dobierana jest funkcja z konwersją `int` lub `uint` do

`double`. W przypadku braku jednoznaczności w wywołaniu funkcji zgłaszany jest błąd. Oto kilka przykładów użycia przeładowanych funkcji w odniesieniu do prezentowanych wyżej prototypów:

```
f( vec4, vec4 );    // bezpośrednie wywołanie funkcji
                    // vec4 f( in vec4 x, out vec4 y )
f( vec4, uvec4 );   // bezpośrednie wywołanie funkcji
                    // vec4 f( in vec4 x, out ivec4 y )
f( ivec4, vec4 );   // błąd, możliwa konwersja do wszystkich
                    // trzech funkcji f; trzecia funkcja jest
                    // lepsza od dwóch początkowych dla pierwszego
                    // argumentu, natomiast dla drugiego argumentu
                    // pierwsza funkcja jest lepsza od pozostałych
f( vec4, ivec4 );   // poprawnie, konwersja do funkcji
                    // vec4 f( in vec4 x, out vec4 y ), możliwa
                    // jest także konwersja do drugiej funkcji,
                    // ale posiada ona gorsze dopasowane,
                    // trzecia funkcja nie może być wykorzystana
                    // z uwagi na brak możliwości konwersji
                    // vec4 do ivec4
```

Przeładowanie nazw funkcji jest szeroko wykorzystywane przez funkcje wbudowane. Funkcje wbudowane można także redefiniować w programach cieniowania, przy czym shader użyje takiej funkcji tylko wtedy, gdy zostanie ona skonsolidowana w obiekcie programu.

Funkcja `main`

Punktem wejściowym każdego programu cieniowania jest funkcja `main`. Nie jest ona niezbędnym elementem każdego programu cieniowania, ale jeden z shaderów danego typu skonsolidowanych w obiekcie programu musi posiadać funkcję `main`.

Funkcja `main` nie posiada argumentów i nie zwraca żadnej wartości. Inna niż poniższa definicja tej funkcji spowoduje zgłoszenie błędu:

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

Jednorodna i niejednorodna kontrola przepływu

Statyczna i dynamiczna kontrola przepływu dotyczy shadera fragmentów i określa czy wszystkie fragmenty są przetwarzane w ten sam, czy też różny sposób. Oto przykłady:

```
main()
{
    float a = ...;// jednorodna kontrola przepływu
    if (a < b)
    {             // wyrażenie może mieć wartość true dla
    ...           // niektórych fragmentów, ale nie wszystkich
    }             // niejednorodna kontrola przepływu
    else
    {
    ...           // niejednorodna kontrola przepływu
    }

...               // ponownie jednorodna kontrola przepływu
```

```
}
```

Innymi przykładami niejednorodnej kontroli przepływu jest użycie pozostałych instrukcji selekcji oraz instrukcji skoków wraz z wcześniejszym wyjściem z funkcji `main`.

Wywoływanie funkcji

W języku GLSL przy wywołaniu funkcji argumenty wejściowe kopiowane są w momencie wywołania, natomiast argumenty wyjściowe kopiowane są przed zakończeniem działania funkcji. Kolejność obliczania i kopiowania argumentów wejściowych jest od lewej do prawej. Kolejność kopiowania parametrów wyjściowych przy jest niezdefiniowana. Do określenia, który z parametrów jest wejściowy, wyjściowy lub wejściowy i wyjściowy, służą opisywane wcześniej kwalifikatory: `in`, `out` oraz `inout`. Przy braku kwalifikatora przyjmowany jest domyślnie `in`. GLSL dopuszcza możliwość zmiany w ciele funkcji wartości argumentu z kwalifikatorem `in`, gdyż modyfikacji podlega jedynie jego lokalna kopia. Wyjątek stanowią parametry z dodatkowym kwalifikatorem przechowywania `const`, który nie może być użyty przy argumentach z kwalifikatorem `out` oraz `inout`.

Specyfikacja języka GLSL nie dopuszcza możliwości rekurencyjnego wywoływania funkcji, w tym statycznego (czyli wtedy, gdy graf wywołań funkcji zawiera cykle). Powyższe wyłączenie obejmuje potencjaną rekurencję uzyskiwaną przy użyciu wywołań opisanych poniżej podprogramów.

Podprogramy

Podprogramy (ang. *subroutines*) to mechanizm umożliwiający wywoływanie różnych funkcji w trakcie wykonywania programu bez konieczność jego rekompilacji. W stosunku do standardowych technik, np. niejednorodnej kontroli przepływu, wybór wywoływanej funkcji - podprogramu, dokonywany z poziomu API OpenGL za pośrednictwem specjalnego rodzaju zmiennych jednorodnych typu podprogramu.

Przed deklaracją funkcji - podprogramów musi być zadeklarowany typ podprogramu do czego jest używana składnia identyczna jak składnia funkcji, ale z dodatkowym słowem zarezerwowanym `subroutine` rozpoczynającym deklarację. Poniżej schematyczna deklaracja typu podprogramu o nazwie `subroutineTypeName`:

```
subroutine returnType subroutineTypeName( type0 arg0,
                                          type1 arg1,
                                          ...,
                                          typen argn );
```

Podobnie jak w przypadku deklaracji funkcji w deklaracji typu podprogramu można pominąć nazwy argumentów (w powyższym przykładzie `arg0`, `arg1`, …, `argn`). Funkcje są przyporządkowywane do określonego typu podprogramu poprzez użycie w deklaracji słowa zarezerwowanego `subroutine` i umieszczonej w nawiasach liście nazw typów podprogramów zgodnych z daną funkcją. Oto przykład schematu takiej deklaracji:

```
subroutine( subroutineTypeName0, ..., subroutineTypeNameN )
            returnType functionName( type0 arg0,
                                     type1 arg1,
                                     ...,
                                     typen argn )
{
    // wnętrze funkcji
}
```

Lista argumentów funkcji - podprogramu oraz zwracany wynik muszą być zgodne z wyspecyfikowanym typem podprogramu (lub podprogramów). W przeciwnym wypadku zgłoszony

będzie błąd na etapie kompilacji shadera. Funkcje zadeklarowane ze słowem zarezerwowanym `subroutine` muszą posiadać wnętrze (definicję). Ponadto w przypadku funkcji - podprogramów nie jest dopuszczalne przeładowanie ich nazw. Wystąpienie takiej sytuacji spowoduje zgłoszenie błędu na etapie konsolidacji programu.

Jak wspomnieliśmy na początku, sterowanie wywołaniami podprogramów odbywa się za pomocą specjalnych zmiennych jednorodnych podprogramów, które w swojej deklaracji zawierają nazwę typu podprogramu:

```
subroutine uniform subroutineTypeName subroutineVarName;
```

Zmienne jednorodne podprogramów wywoływane są w taki sam sposób jak funkcje. To jaka funkcja - podprogram zostanie wywołany określa wartość zmiennej ustalana z poziomu API OpenGL.

Zmienne jednorodne podprogramów mogą być zadeklarowane w tablicy o jawnie określonym rozmiarze, co umożliwia dodatkowo ich dynamiczne indeksowanie.

Selekcje

Selekcje `if`, `if`/`else` oraz `switch`/`case`/`default` mają taką samą konstrukcję jak w językach C i C++. Wyrażeniem warunkowym występującym w instrukcjach `if` i `if`/`else` może być dowolne wyrażenie typu logicznego `bool`, z wyłączeniem typów wektorowych. Natomiast wyrażenie w instrukcji `switch` musi być typu skalarnego całkowitego. Selekcje mogą być zagnieżdżone.

Iteracje

Iteracje `for`, `while` oraz `do`/`while` mają w języku GLSL analogiczną konstrukcję jak w językach C i C++. Wyrażenie warunkowe w nich występujące musi być wyrażeniem typu logicznego `bool`. Iteracje mogą być zagnieżdżone. Specyfikacja języka GLSL dopuszcza pętle nieskończone, jednak konsekwencje związane z ich działaniem zależne są od implementacji.

Skoki

GLSL definiuje kilka rodzajów instrukcji skoków: `continue`, `break`, `return`, `return wyrażenie` oraz `discard`. Pierwsza instrukcja ma zastosowanie wyłącznie w pętlach, druga w pętlach i instrukcji selekcji `switch`, a ich działanie jest takie same jak w językach C i C++. Specyficzna dla GLSL instrukcja `discard` dostępna jest wyłącznie w programach cieniowania fragmentów, a jej wywołanie powoduje odrzucenie aktualnego fragmentu z przetwarzania w dalszych etapach potoku renderingu. Instrukcja `return` lub `return wyrażenie` powoduje natychmiastowe opuszczenie funkcji. Instrukcja `return` może być także używana w funkcji `main`, ale w przypadku shadera fragmentów nie jest równoważna wykonaniu instrukcji `discard`.

Kolejność kwalifikatorów

Jeżeli deklaracja zmiennej posiada kilka kwalifikatorów mogą one występować w dowolnym porządku, przy czym wszystkie muszą poprzedzać określenie typu zmiennej. Jedynie kwalifikator formatu musi występować na początku deklaracji zmiennej. Ponadto specyfikacja języka GLSL dopuszcza wielokrotne wystąpienie w deklaracji zmiennej kwalifikatorów przechowywania, pomocniczych kwalifikatorów przechowywania, kwalifikatorów pamięci oraz kwalifikatorów interpolacji.

Wbudowane zmienne

Shadery wykorzystują wbudowane zmienne do komunikacji z nieprogramowalną częścią potoku OpenGL. Odmienność zadań poszczególnych rodzajów programów cieniowania powoduje, że posiadają one odrębne zestawy wbudowanych zmiennych.

Przegląd wbudowanych zmiennych

Wbudowane zmienne shadera wierzchołków:

```
in int gl_VertexID;
in int gl_InstanceID;

out gl_PerVertex
{
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
};
```

Wbudowane zmienne shadera kontroli teselacji:

```
in gl_PerVertex
{
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[gl_MaxPatchVertices];

in int gl_PatchVerticesIn;
in int gl_PrimitiveID;
in int gl_InvocationID;

out gl_PerVertex
{
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_out[];

patch out float gl_TessLevelOuter[4];
patch out float gl_TessLevelInner[2];
```

Wbudowane zmienne shadera ewaluacji teselacji:

```
in gl_PerVertex
{
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[gl_MaxPatchVertices];

in int gl_PatchVerticesIn;
in int gl_PrimitiveID;
in vec3 gl_TessCoord;
patch in float gl_TessLevelOuter[4];
patch in float gl_TessLevelInner[2];

out gl_PerVertex
{
    vec4 gl_Position;
    float gl_PointSize;
```

```
    float gl_ClipDistance[];
};
```

Wbudowane zmienne shadera geometrii:

```
in gl_PerVertex
{
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[];

in int gl_PrimitiveIDIn;
in int gl_InvocationID;

out gl_PerVertex
{
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
};

out int gl_PrimitiveID;
out int gl_Layer;
out int gl_ViewportIndex;
```

Wbudowane zmienne shadera fragmentów:

```
in vec4 gl_FragCoord;
in bool gl_FrontFacing;
in float gl_ClipDistance[];
in vec2 gl_PointCoord;
in int gl_PrimitiveID;
in int gl_SampleID;
in vec2 gl_SamplePosition;
in int gl_SampleMaskIn[];

out float gl_FragDepth;
out int gl_SampleMask[];
```

Zmienna `gl_VertexID`

Zmienna `gl_VertexID` jest dostępna tylko dla shadera wierzchołków i zawiera numer przetwarzanego wierzchołka przekazany przez polecenie renderingu (np. `glDrawArrays`). Zmienna ta jest zawsze dostępna, ale jej wartość nie zawsze jest zdefiniowana.

Zmienna `gl_InstanceID`

Zmienna `gl_InstanceID` także dostępna jest tylko w shaderze wierzchołków i zawiera licznik instancji. Licznik ten zawiera numer egzemplarza obiektu, gdy prymitywy rysowane są z odpowiednimi poleceniami renderingu instancyjnego (np. `glDrawArraysInstanced`). W przeciwnym wypadku licznik instancji zawiera wartość 0.

Zmienna gl_Position

Zmienna gl_Position jest przeznaczona do zapisu współrzędnych jednorodnych wierzchołka we współrzędnych obcinania (patrz odcinek kursu poświęcony scenie i przekształceniom oraz dotychczasowe programy przykładowe). Wartość ta jest używana podczas składania prymitywów, obcinania i usuwania prymitywów oraz innych operacji przetwarzania wierzchołków prymitywów. Wartość zmiennej gl_Position jest nieokreślona dopóki nie nastąpi zapis na etapie przetwarzania wierzchołków (shader wierzchołków) lub przetwarzania geometrii (shader geometrii). W przypadku tego ostatniego shadera zapis zmiennej gl_Position może być dokonany dla każdego wierzchołka, a jej wartość dla wierzchołka jest określana (lub pozostaje nieokreślona) w chwili wywołania funkcji EmitVertex.

Zmienna gl_PointSize

Zmienna wyjściowa gl_PointSize określa w pikselach rozmiar rasteryzowanych punktów. Jeżeli zmienna gl_PointSize nie zostanie zapisana jej wartość pozostaje nieokreślona w następnych etapach potoku renderingu. Jako zmienna wejściowa gl_PointSize przyjmuje wartość określoną na poprzednim etapie potoku renderingu.

Zmienna gl_ClipDistance

Ze zmienną gl_ClipDistance mieliśmy już okazję dokładnie się zapoznać w jednym z wcześniejszych programów przykładowych. Reguluje ona mechanizm płaszczyzn obcinania zdefiniowanych przez użytkownika i zawiera odległości wierzchołka od poszczególnych płaszczyzn obcinania. Odległość równa zero oznacza, że wierzchołek zawiera się w płaszczyźnie, wartość dodatnia oznacza, że wierzchołek jest wewnątrz nieobciętej półprzestrzeni wyznaczonej przez płaszczyznę obcinania, a wartość ujemna oznacza, że wierzchołek podlega mechanizmowy obcinania.

Rozmiar tablicy gl_ClipDistance nie jest określony na stałe i musi być predefiniowany w shaderze, przy czym nie może przekraczać wielkości stałej gl_MaxClipDistances, którą opisujemy dalej. Każda aktywna płaszczyzna obcinania definiowana przez użytkownika wymaga włączenia w głównym programie. W przeciwnym wypadku zapis do elementu tablicy gl_ClipDistance odpowiadającemu takiej płaszczyźnie nie wywołuje efektu.

Jako zmienna wejściowa gl_ClipDistance przyjmuje wartość określoną na poprzednim etapie renderingu, za wyjątkiem shadera fragmentów, gdzie jej wartość podlega standardowemu procesowi liniowej interpolacji atrybutów wierzchołków.

Zmienna gl_PrimitiveID

Jako zmienna wyjściowa shadera geometrii, gl_PrimitiveID zawiera identyfikator prymitywu. Zmienna o takiej samej nazwie jest dostępna jako zmienna wejściowa shadera fragmentów i zawiera identyfikator prymitywu pochodzący z podstawowego wierzchołka aktualnie przetwarzanego prymitywu. Jeżeli zmienna gl_PrimitiveID jest używana przez shader fragmentów i jednocześnie aktywny jest shader geometrii, shader geometrii musi dokonać zapisu wartość tej zmiennej, bowiem w przeciwnym wypadku wartość ta będzie nieokreślona w shaderze fragmentów.

Zmienne wejściowe gl_PrimitiveID shadera kontroli teselacji i shadera ewaluacji teselacji zawierają numer prymitywu przetwarzanego przez shader od momentu rozpoczęcia renderingu bieżącego zbioru prymitywów (polecenia renderingu). Analogiczną wartość powyższa zmienna przyjmie w shaderze fragmentów, ale tylko w przypadku, gdy nie jest aktywny shader geometrii.

Zmienna gl_PrimitiveIDIn

Zmienna gl_PrimitiveIDIn dostępna jest do odczytu wyłącznie w shaderze geometrii i zawiera numer prymitywu przetwarzanego przez shader geometrii od momentu rozpoczęcia

renderingu bieżącego zbioru prymitywów. Jest to zatem wartość analogiczna do wartości zmiennej `gl_PrimitiveID` w shaderze kontroli teselacji i shaderze ewaluacji teselacji

Zmienna `gl_InvocationID`

Zmienna wejściowa `gl_InvocationID` jest dostępna w shaderze kontroli teselacji i shaderze geometrii. W shaderze kontroli teselacji zmienna ta zawiera numer wyjściowej ścieżki wierzchołka przypisaną przy wywołania shadera kontroli teselacji. Natomiast w shaderze geometrii zmienna `gl_InvocationID` zawiera numer wywołania przypisany do wywołania shadera geometrii. W obu przypadkach wartość zmiennej zawiera się w przedziale $\langle 0; N-1 \rangle$, gdzie N jest ilością wyjściowych ścieżek wierzchołków lub ilością wywołań shadera geometrii na prymityw.

Zmienna `gl_Layer`

Zmienna wyjściowa `gl_Layer` jest dostępna do zapisu jedynie w shaderze geometrii. Zmienna ta określa warstwę renderingu (lub stronę mapy sześciennej - patrz Tabela 13) w przypadku użycia techniki renderingu wielowarstwowego. W takim przypadku każdy z prymitywów generowanych przez shader geometrii może być emitowany do innej warstwy renderingu – decyduje o tym wartość zapisana do zmiennej `gl_Layer` w trakcie generowania wierzchołków prymitywu. Jeżeli shader geometrii przypisze wartość do zmiennej `gl_Layer`, tryb renderingu wielowarstwowego jest aktywny, ale wymaga to także odpowiednio przygotowanego bufora ramki.

Numer	Wyjściowa strona
0	`GL_TEXTURE_CUBE_MAP_POSITIVE_X`
1	`GL_TEXTURE_CUBE_MAP_NEGATIVE_X`
2	`GL_TEXTURE_CUBE_MAP_POSITIVE_Y`
3	`GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`
4	`GL_TEXTURE_CUBE_MAP_POSITIVE_Z`
5	`GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`

Tabela 13 Numeracja stron tekstury sześciennej.

Specjalne wartości zmiennej `gl_Layer` używane są podczas renderingu do tablicy tekstur sześciennych. Zamiast odwołania wyłącznie do warstwy wybranej tekstury sześciennej, zmienna ta zawiera także stronę tej tekstury sześciennej. Wartość zapisywana do zmiennej `gl_Layer` zawiera wówczas pomnożony przez 6 numer warstwy tablicy tekstur sześciennych powiększony o numer wybranej strony mapy sześciennej zgodny z Tabela 13.

Zmienna `gl_ViewportIndex`

Zmienna wyjściowa `gl_ViewportIndex` jest dostępna tylko w shaderze geometrii i zawiera indeks obszaru renderingu, w którym będą renderowane prymitywy generowane przez shader geometrii. Tak generowane prymitywu podlegają późniejszym transformacjom obszaru renderingu i testom obcinania zgodnie z ustawieniami wybranego obszaru renderingu. Specyfikacja GLSL zaleca, aby wszystkie wierzchołki prymitywu renderowane były w jednym obszarze renderingu. Niezachowanie tego warunku nie jest definiowane i zależy od implementacji.

Jeżeli shader geometrii nie zapisuje wartości do zmiennej `gl_ViewportIndex` domyślnie przyjmuje ona wartość 0. Jeżeli shader geometrii dokonuje zapisu wartości powyższej zmiennej, operacja ta musi być realizowana dla wszystkich ścieżek wykonania shadera.

Zmienna `gl_PatchVerticesIn`

Zmienna wejściowa `gl_PatchVerticesIn` dostępna jest do odczytu w shaderze kontroli teselacji i shaderze ewaluacji teselacji i zawiera ilość wierzchołków w wejściowej ścieżce wierzchołków przetwarzanych przez shader. Ponieważ pojedynczy shader kontroli teselacji i shader

ewaluacji teselacji może operować na ścieżce wierzchołków o różnych rozmiarach, stąd wartość zmiennej `gl_PatchVerticesIn` może różnić się dla poszczególnych ścieżek wierzchołków.

Zmienne `gl_TessLevelOuter` i `gl_TessLevelInner`

Zmienne wyjściowe `gl_TessLevelOuter` i `gl_TessLevelInner` są dostępne wyłącznie w shaderze kontroli teselacji i służą do określenia odpowiednio zewnętrznego i wewnętrznego poziomu teselacji ścieżki (zbioru) wierzchołków. Wartości te są następnie używane przez generator prymitywów do kontroli procesu teselacji. Jako zmienne wejściowe `gl_TessLevelOuter` i `gl_TessLevelInner` są dostępne wyłącznie w shaderze ewaluacji teselacji i zawierają wartości zapisane w shaderze kontroli teselacji. Jeżeli shader kontroli teselacji nie jest dostępny lub nie dokonał zapisu wartości powyższych zmiennych, przyjmują one wartości domyślne określone z poziomu API OpenGL.

Zmienna `gl_TessCoord`

Zmienna wejściowa `gl_TessCoord` jest dostępna wyłącznie w shaderze kontroli teselacji i zawiera dwuelementowe znormalizowane współrzędne kartezjańskie (u, v) lub trójelementowe współrzędne barycentryczne (u, v, w) określające położenie przetwarzanego wierzchołka w odniesieniu do prymitywu poddawanemu procesowi teselacji (ścieżce wierzchołków).

Zmienna `gl_FragCoord`

Zmienna `gl_FragCoord` jest dostępna do odczytu w shaderze fragmentów i zawiera współrzędne (x_f, y_f, z_f, w_f) opisujące położenie fragmentu we współrzędnych okna renderingu i jest obliczana przez stałą funkcjonalność potoku OpenGL podczas procesu generowania fragmentów z prymitywów. Wartości przekazywane w składowych `x` i `y` zależą od opisanych wyżej identyfikatorów `origin_upper_left` i `pixel_center_integer` kwalifikatora formatu wejściowego shadera fragmentów. Składowa `z` zawiera wartość głębokości obliczonej dla fragmentu, która jest używana jako wyjściowa wartość głębokości fragmentu, chyba że shader fragmentów dokona zapisu do zmiennej `gl_FragDepth`.

Wartości współrzędnych (x_f, y_f, z_f, w_f) obliczane są w następujący sposób ze współrzędnych okienkowych fragmentu (x_w, y_w, z_w), wysokości $height$ obszaru renderingu w pikselach oraz składowej w_c ze współrzędnych obcinania (patrz opis przekształceń współrzędnych wierzchołków w poprzednim odcinku kursu OpenGL):

$$x_f = \begin{cases} x_w - \frac{1}{2} & aktywny \text{ pixel\_center\_integer} \\ x_w & w\ przeciwnym\ wypadku \end{cases}$$

$$y'_f = \begin{cases} height - y_w & aktywny \text{ origin\_upper\_left} \\ y_w & w\ przeciwnym\ wypadku \end{cases}$$

$$y_f = \begin{cases} y'_f - \frac{1}{2} & aktywny \text{ pixel\_center\_integer} \\ y'_f & w\ przeciwnym\ wypadku \end{cases}$$

$$z_f = z_w$$

$$w_f = \frac{1}{w_c}$$

W przypadku, gdy aktywne jest wielopróbkowanie, zmienna `gl_FragCoord` może zawierać dowolną lokalizację z wewnątrz piksela lub wartość opisującą jedną z próbek fragmentu. Także użycie kwalifikatora `centroid` nie prowadzi do pobrania wartości zmiennej z wnętrza bieżącego prymitywu.

Zmienna gl_FrontFacing

Zmienna gl_FrontFacing jest zmienną wejściową dostępną do odczytu w shaderze fragmentów. Zawiera ona informację, czy bieżący fragment pochodzi z przedniej strony prymitywu (wartość true) lub tylnej (wartość false) i może być używana np. do emulacji dwustronnego oświetlenia na podstawie kolorów obliczanych przez shader wierzchołków lub geometrii. W przypadku fragmentów generowanych z prymitywów innych niż trójkąty, zmienna gl_FrontFacing przyjmuje zawsze wartość równą true.

Zmienna gl_PointCoord

Zmienna gl_PointCoord jest zmienną wejściową shadera fragmentów i zawiera dwuwymiarowe współrzędne położenia bieżącego fragmentu należącego do prymitywu punktowego, gdy aktywne są sprajty punktowe. Wartości współrzędnych zawierają się w przedziale $\langle 0{,}0; 1{,}0\rangle$. Jeżeli bieżący prymityw nie jest punktem lub sprajty punktowe nie są aktywne wartości zawarte w zmiennej gl_PointCoord są niezdefiniowane.

Zmienna gl_FragDepth

Zmienna gl_FragDepth jest jedną z dwóch predefiniowanych zmiennych wyjściowych shadera fragmentów. Zmienna ta umożliwia zapis alternatywnej wartości głębokości fragmentu w stosunku do wartości głębokości obliczoną przez standardową funkcjonalność potoku renderingu OpenGL, dostępną w zmiennej gl_FragCoord.z. Jeżeli shader zapisuje wartość zmiennej gl_FragDepth, zapis powinien być realizowany na wszystkich ścieżkach wykonania programu. W przeciwnym wypadku wartość głębokości dla tej ścieżki wykonania programu pozostanie nieokreślona.

Jeżeli shader fragmentów wywoła instrukcję discard, następuje odrzucenie bieżącego fragmentu z przetwarzania w dalszych etapach potoku renderingu, a ewentualny zapis zmiennej gl_FragDepth oraz zmiennych wyjściowych zdefiniowanych przez użytkownika jest nieistotny.

Zmienna gl_SampleMaskIn

Tablica gl_SampleMaskIn jest zmienną wejściową dostępną w shaderze fragmentów i zawiera maskę bitową pokrycia próbek przetwarzanego fragmentu prymitywu w procesie wielopróbkowania (aktywna zmienna stanu GL_SAMPLE_MASK). Wielkość tablicy zależy od maksymalnej ilości próbek koloru używanej w wielopróbkowaniu, przy czym kolejne jej elementy zawierają kolejne 32 bity maski pokrycia (odpowiada to wyrażeniu $\left\lceil \frac{s}{32} \right\rceil$, gdzie s jest maksymalną ilością próbek obsługiwanych przez implementację). Wartość bitu maski równa 1 oznacza, że odpowiadająca mu próbka jest używana w procesie obliczania wartości pokrycia fragmentu w wielopróbkowaniu, w przeciwnym wypadku wartość bitu maski pokrycia wynosi 0. W przypadku, gdy rendering realizowany jest w buforze nie obsługującym wielopróbkowania, lub wielopróbkowanie jest wyłączone, wszystkie bity maski pokrycia mają wartość równą 0, poza pierwszym elementem maski, który ma wartość równą 1.

W przypadku, gdy aktywne jest oddzielne przetwarzanie każdej próbki, tablica gl_SampleMaskIn zawiera wartość pokrycia tylko aktualnie przetwarzanej próbki. Numer przetwarzanej próbki zawiera opisywana dalej kolejności wbudowana zmienna gl_SampleID.

Zmienna gl_SampleMask

Tablica gl_SampleMask jest zmienną wyjściową dostępną w shaderze fragmentów i zawiera wyjściową maskę bitową pokrycia próbek przetwarzanego fragmentu prymitywu w procesie wielopróbkowania (aktywna zmienna stanu GL_SAMPLE_MASK). Wartość bitu maski równa 1 oznacza, że odpowiadająca mu próbka jest używana w procesie obliczania wartości pokrycia fragmentu w wielopróbkowaniu, w przeciwnym wypadku wartość bitu maski pokrycia wynosi 0. Wielkość tablicy zależy od maksymalnej ilości próbek koloru używanej w wielopróbkowaniu, przy czym kolejne jej elementy zawierają kolejne 32 bity maski pokrycia i musi być jawnie lub niejawnie

określona w shaderze fragmentów (wielkość tablicy odpowiada wyrażeniu $\left\lceil \frac{s}{32} \right\rceil$, gdzie s jest maksymalną ilością próbek obsługiwanych przez implementację). Jeżeli shader fragmentów dokona zapisu do `gl_SampleMask` jej wartość pozostanie nieokreślona w przypadku, gdy jakiekolwiek wywołanie shadera fragmentów nie doprowadzi do zapisu tej zmiennej.

W przypadku, gdy aktywne jest oddzielne przetwarzanie każdej próbki, shader fragmentów musi dokonać zapisu bitu w tablicy `gl_SampleMask` odpowiadającemu wartość pokrycia tylko aktualnie przetwarzanej próbki. Zmiany bitów dotyczących innych próbek nie wywołują efektu. Numer przetwarzanej próbki zawiera opisywana dalej kolejności wbudowana zmienna `gl_SampleID`.

Jeżeli shader fragmentów wywoła instrukcję `discard`, następuje odrzucenie bieżącego fragmentu z przetwarzania w dalszych etapach potoku renderingu, a ewentualny zapis zmiennej `gl_SampleMask` oraz zmiennych wyjściowych zdefiniowanych przez użytkownika jest nieistotny.

Zmienna `gl_SampleID`

Zmienna `gl_SampleID` dostępna do odczytu w shaderze fragmentów zawiera numer aktualnie przetwarzanej próbki. Wartość zmiennej zawiera się w przedziale od 0 do `gl_NumSamples - 1`, gdzie `gl_NumSamples` to łączna ilość próbek przypadających na jeden piksel obrazu w buforze ramki. W przypadku, gdy wielopróbkowanie nie jest włączone, lub rendering odbywa się do bufora nieobsługującego wielopróbkowania, zmienna `gl_SampleID` zawiera wartość 0. Każde statyczne użycie tej zmiennej w shaderze fragmentów oznacza, że dany shader wywoływany jest oddzielnie na każdą przetwarzaną próbkę.

Zmienna `gl_SamplePosition`

Zmienna `gl_SamplePosition` dostępna do odczytu w shaderze fragmentów zawiera współrzędne położenia bieżącej próbki w buforze wielopróbkowania. Dwuwymiarowe współrzędne położenia próbki (podpiksela) zawierają się w przedziale $\langle 0; 1\rangle$, przy czym środek piksela wyznaczają współrzędne $\left(\frac{1}{2}, \frac{1}{2}\right)$. W przypadku, gdy wielopróbkowanie nie jest włączone, lub rendering odbywa się do bufora nieobsługującego wielopróbkowania, zmienna `gl_SamplePosition` zawiera współrzędne równe $\left(\frac{1}{2}, \frac{1}{2}\right)$. Podobnie jak w przypadku `gl_SampleID`, każde statyczne użycie tej zmiennej w shaderze fragmentów oznacza, że dany shader jest obliczany na każdą próbkę.

Redefiniowanie bloku zmiennych `gl_PerVertex`

Blok zmiennych `gl_PerVertex` może być redefiniowany w shaderach celem wskazania, które elementy standardowego procesu przetwarzania wierzchołków są używane w programie. Jest to niezbędne do ustawienia interfejsu pomiędzy poszczególnym programowalnymi etapami potoku renderingu. Redefinicja może obejmować tylko standardowe elementy bloku `gl_PerVertex`. Oto przykład:

```
out gl_PerVertex
{
    vec4 gl_Position;   // używamy zmiennej gl_Position
    float gl_PointSize; // używamy zmiennej gl_PointSize
    vec4 t;             // błąd, dopuszczalne jest użycie
                        // tylko składowych bloku gl_PerVertex
};                      // pozostałe składowe bloku gl_PerVertex
                        // nie są używane
```

Redefinicja bloku zmiennych `gl_PerVertex` musi wystąpić przed pierwszym użyciem jego składowych w shaderze. Nie jest dopuszczalna więcej niż jedna redefinicja powyższego bloku w shaderze. Ponadto, jeżeli wiele shaderów skonsolidowanych w ramach jednego obiektu programu używa redefinicji bloku `gl_PerVertex`, to wszystkie redefinicje muszą być zgodne i tworzyć spójne

interfejsy pomiędzy poszczególnymi programowalnymi etapami potoku renderingu. Analogiczna zasada dotyczy także shadera współdzielonego przez wiele obiektów programów.

Wbudowane stałe

Opisane poniżej wbudowane stałe dostępne są dla wszystkich rodzajów shaderów. Podane wartości odpowiadają minimalnym wymaganiom określonym przez specyfikację biblioteki OpenGL. W komentarzach oprócz opisu dodano nazwy zmiennych stanu odpowiadających opisywanym stałym.

```
// maksymalna ilość aktywnych atrybutów wierzchołków,
// GL_MAX_VERTEX_ATTRIBS
const int gl_MaxVertexAttribs = 16;

// maksymalna ilość składowych zmiennych jednorodnych
// shaderów wierzchołków, GL_MAX_VERTEX_UNIFORM_COMPONENTS
const int gl_MaxVertexUniformComponents = 1024;

// maksymalna ilość składowych zmiennych
// interpolowanych, GL_MAX_VARYING_COMPONENTS
// (poprzednio gl_MaxVaryingFloats, GL_MAX_VARYING_FLOATS)
const int gl_MaxVaryingComponents = 60;

// maksymalna ilość składowych wyjściowych zmiennych
// interpolowanych zapisywanych przez shader wierzchołków,
// GL_MAX_VERTEX_OUTPUT_COMPONENTS
const int gl_MaxVertexOutputComponents = 64;

// maksymalna ilość składowych wejściowych odczytywanych
// przez shader geometrii, GL_MAX_GEOMETRY_INPUT_COMPONENTS
const int gl_MaxGeometryInputComponents = 64;

// maksymalna ilość składowych wyjściowych zapisywanych
// przez shader geometrii, GL_MAX_GEOMETRY_OUTPUT_COMPONENTS
const int gl_MaxGeometryOutputComponents = 128;

// maksymalna ilość składowych wejściowych odczytywanych
// przez shader fragmentów, GL_MAX_FRAGMENT_INPUT_COMPONENTS
const int gl_MaxFragmentInputComponents = 128;

// maksymalna ilość jednostek teksturujących dostępnych dla shadera
// wierzchołków, GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS
const int gl_MaxVertexTextureImageUnits = 16;

// maksymalna łączna dostępna ilość jednostek teksturujących,
// GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS
const int gl_MaxCombinedTextureImageUnits = 80;

// maksymalna ilość jednostek teksturujących dostępnych dla
// shadera fragmentów, GL_MAX_TEXTURE_IMAGE_UNITS
const int gl_MaxTextureImageUnits = 16;

// maksymalna ilość jednostek obrazu tekstury,
// GL_MAX_IMAGE_UNITS
const int gl_MaxImageUnits = 8;
```

```
// maksymalna ilość jednostek obrazu tekstury
// i zmiennych wyjściowych shadera fragmentów,
// GL_MAX_COMBINED_IMAGE_UNITS_AND_FRAGMENT_OUTPUTS
const int gl_MaxCombinedImageUnitsAndFragmentOutputs = 8;

// maksymalna ilość próbek dostępnych dla poziomu
// tekstury powiązanego jako obraz tekstury,
// GL_MAX_IMAGE_SAMPLES
const int gl_MaxImageSamples = 0;

// maksymalna ilość zmiennych obrazów tekstury
// dostępnych dla shadera wierzchołków,
// GL_MAX_VERTEX_IMAGE_UNIFORMS
const int gl_MaxVertexImageUniforms = 0;

// maksymalna ilość zmiennych obrazów tekstury
// dostępnych dla shadera kontroli teselacji,
// GL_MAX_TESS_CONTROL_IMAGE_UNIFORMS
const int gl_MaxTessControlImageUniforms = 0;

// maksymalna ilość zmiennych obrazów tekstury
// dostępnych dla shadera ewaluacji teselacji,
// GL_MAX_TESS_EVALUATION_IMAGE_UNIFORMS
const int gl_MaxTessEvaluationImageUniforms = 0;

// maksymalna ilość zmiennych obrazów tekstury
// dostępnych dla shadera geometrii,
// GL_MAX_GEOMETRY_IMAGE_UNIFORMS
const int gl_MaxGeometryImageUniforms = 0;

// maksymalna ilość zmiennych obrazów tekstury
// dostępnych dla shadera fragmentów,
// GL_MAX_FRAGMENT_IMAGE_UNIFORMS
const int gl_MaxFragmentImageUniforms = 0;

// maksymalna ilość zmiennych obrazów tekstury
// dostępnych dla wszystkich shaderów,
// GL_MAX_COMBINED_IMAGE_UNIFORMS
const int gl_MaxCombinedImageUniforms = 0;

// maksymalna ilość składowych zmiennych jednorodnych
// shaderów fragmentów, GL_MAX_FRAGMENT_UNIFORM_COMPONENTS
const int gl_MaxFragmentUniformComponents = 1024;

// maksymalna ilość aktywnych buforów (obszarów) renderingu,
// GL_MAX_DRAW_BUFFERS
const int gl_MaxDrawBuffers = 8;

// maksymalna ilość płaszczyzn obcinania definiowanych
// przez użytkownika, GL_MAX_CLIP_DISTANCES
const int gl_MaxClipDistances = 8;

// maksymalna ilość jednostek teksturujących dostępnych dla
// shadera geometrii, GL_MAX_GEOMETRY_TEXTURE_IMAGE_UNITS
const int gl_MaxGeometryTextureImageUnits = 16;
```

```
// maksymalna ilość wierzchołków, które może wygenerować
// shader geometrii, GL_MAX_GEOMETRY_OUTPUT_VERTICES
const int gl_MaxGeometryOutputVertices = 256;

// maksymalna łączna ilość składowych (wszystkich wierzchołków)
// aktywnych zmiennych jednorodnych, które może wygenerować
// shader geometrii, GL_MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS
const int gl_MaxGeometryTotalOutputComponents = 1024;

// maksymalna ilość składowych aktywnych zmiennych
// jednorodnych dostępnych dla shaderów geometrii,
// GL_MAX_GEOMETRY_UNIFORM_COMPONENTS
const int gl_MaxGeometryUniformComponents = 1024;

// maksymalna ilość składowych aktywnych zmiennych
// interpolowanych dostępnych dla shaderów geometrii
const int gl_MaxGeometryVaryingComponents = 64;

// maksymalna ilość składowych zmiennych wejściowych
// w shaderze kontroli teselacji,
// GL_MAX_TESS_CONTROL_INPUT_COMPONENTS
const int gl_MaxTessControlInputComponents = 128;

// maksymalna ilość składowych zmiennych wyjściowych
// w shaderze kontroli teselacji,
// GL_MAX_TESS_CONTROL_OUTPUT_COMPONENTS
const int gl_MaxTessControlOutputComponents = 128;

// maksymalna ilość jednostek teksturujących dostępnych dla
// shadera kontroli teselacji,
// GL_MAX_TESS_CONTROL_TEXTURE_IMAGE_UNITS
const int gl_MaxTessControlTextureImageUnits = 16;

// maksymalna ilość składowych zmiennych jednorodnych
// w shaderze kontroli teselacji,
// GL_MAX_TESS_CONTROL_UNIFORM_COMPONENTS
const int gl_MaxTessControlUniformComponents = 1024;

// maksymalna ilość składowych zmiennych wyjściowych
// w shaderze kontroli teselacji,
// GL_MAX_TESS_CONTROL_TOTAL_OUTPUT_COMPONENTS
const int gl_MaxTessControlTotalOutputComponents = 4096;

// maksymalna ilość składowych zmiennych wejściowych
// w shaderze ewaluacji teselacji,
// GL_MAX_TESS_EVALUATION_INPUT_COMPONENTS
const int gl_MaxTessEvaluationInputComponents = 128;

// maksymalna ilość składowych zmiennych wyjściowych
// w shaderze ewaluacji teselacji,
// GL_MAX_TESS_EVALUATION_OUTPUT_COMPONENTS
const int gl_MaxTessEvaluationOutputComponents = 128;

// maksymalna ilość jednostek teksturujących dostępnych dla
// shadera ewaluacji teselacji,
// GL_MAX_TESS_EVALUATION_TEXTURE_IMAGE_UNITS
```

```
const int gl_MaxTessEvaluationTextureImageUnits = 16;

// maksymalna ilość składowych zmiennych jednorodnych
// w shaderze ewaluacji teselacji,
// GL_MAX_TESS_EVALUATION_UNIFORM_COMPONENTS
const int gl_MaxTessEvaluationUniformComponents = 1024;

// maksymalna ilość składowych ścieżki wierzchołków w shaderze
// kontroli teselacji, GL_MAX_TESS_PATCH_COMPONENTS
const int gl_MaxTessPatchComponents = 120;

// maksymalna wielkość ścieżki wierzchołków (ilość
// wierzchołków w ścieżce), GL_MAX_PATCH_VERTICES
const int gl_MaxPatchVertices = 32;

// maksymalny poziom teselacji obsługiwany przez generator
// prymitywów w teselatorze, GL_MAX_TESS_GEN_LEVEL
const int gl_MaxTessGenLevel = 64;

// maksymalna ilość obszarów renderingu, GL_MAX_VIEWPORTS
const int gl_MaxViewports = 16;

// maksymalna ilość wektorów zmiennych jednorodnych
// shaderów wierzchołków, GL_MAX_VERTEX_UNIFORM_VECTORS;
// wartość zmiennej jest równa 1/4 wartości zmiennej
// gl_MaxVertexUniformComponents
// (GL_MAX_VERTEX_UNIFORM_COMPONENTS)
const int gl_MaxVertexUniformVectors = 256;

// maksymalna ilość wektorów zmiennych jednorodnych
// shaderów fragmentów, GL_MAX_FRAGMENT_UNIFORM_VECTORS;
// wartość zmiennej jest równa 1/4 wartości zmiennej
// gl_MaxFragmentUniformComponents
// (GL_MAX_FRAGMENT_UNIFORM_COMPONENTS)
const int gl_MaxFragmentUniformVectors = 256;

// maksymalna ilość wektorów zmiennych interpolowanych,
// GL_MAX_VARYING_VECTORS; wartość zmiennej jest równa 1/4
// wartości zmiennej gl_MaxVaryingComponents
// (GL_MAX_VARYING_COMPONENTS)
const int gl_MaxVaryingVectors = 15;

// maksymalna ilość liczników atomowych dostępnych
// w shaderze wierzchołków,
// GL_MAX_VERTEX_ATOMIC_COUNTERS
const int gl_MaxVertexAtomicCounters = 0;

// maksymalna ilość liczników atomowych dostępnych
// w shaderze kontroli teselacji,
// GL_MAX_TESS_CONTROL_ATOMIC_COUNTERS
const int gl_MaxTessControlAtomicCounters = 0;

// maksymalna ilość liczników atomowych dostępnych
// w shaderze ewaluacji teselacji,
// GL_MAX_TESS_EVALUATION_ATOMIC_COUNTERS
```

```
const int gl_MaxTessEvaluationAtomicCounters = 0;

// maksymalna ilość liczników atomowych dostępnych
// w shaderze geometrii,
// GL_MAX_GEOMETRY_ATOMIC_COUNTERS
const int gl_MaxGeometryAtomicCounters = 0;

// maksymalna ilość liczników atomowych dostępnych
// w shaderze fragmentów,
// GL_MAX_FRAGMENT_ATOMIC_COUNTERS
const int gl_MaxFragmentAtomicCounters = 8;

// maksymalna ilość liczników atomowych dostępnych
// w obiekcie programu,
// GL_MAX_COMBINED_ATOMIC_COUNTERS
const int gl_MaxCombinedAtomicCounters = 8;

// maksymalna ilość punktów wiązania liczników atomowych
const int gl_MaxAtomicCounterBindings = 1;

// maksymalna ilość obiektów bufora licznika
// atomowego dostępnych w shaderze wierzchołków,
// GL_MAX_VETREX_ATOMIC_COUNTER_BUFFERS
const int gl_MaxVertexAtomicCounterBuffers = 0;

// maksymalna ilość obiektów bufora licznika
// atomowego dostępnych w shaderze kontroli teselacji,
// GL_MAX_TESS_CONTROL_ATOMIC_COUNTER_BUFFERS
const int gl_MaxTessControlAtomicCounterBuffers = 0;

// maksymalna ilość obiektów bufora licznika
// atomowego dostępnych w shaderze ewaluacji teselacji,
// GL_MAX_TESS_EVALUATION_ATOMIC_COUNTER_BUFFERS
const int gl_MaxTessEvaluationAtomicCounterBuffers = 0;

// maksymalna ilość obiektów bufora licznika
// atomowego dostępnych w shaderze geometrii,
// GL_MAX_GEOMETRY_ATOMIC_COUNTER_BUFFERS
const int gl_MaxGeometryAtomicCounterBuffers = 0;

// maksymalna ilość obiektów bufora licznika
// atomowego dostępnych w shaderze fragmentów,
// GL_MAX_FRAGMENT_ATOMIC_COUNTER_BUFFERS
const int gl_MaxFragmentAtomicCounterBuffers = 1;

// maksymalna ilość obiektów bufora licznika
// atomowego dostępnych w obiekcie programu,
// GL_MAX_COMBINED_ATOMIC_COUNTER_BUFFERS
const int gl_MaxCombinedAtomicCounterBuffers = 1;

// maksymalna wielkość w BMU danych obiektu
// bufora licznika atomowego,
// GL_MAX_ATOMIC_COUNTER_BUFFER_SIZE
const int gl_MaxAtomicCounterBufferSize = 16384;

// minimalna wartość przesunięcia współrzędnych tekstury
```

```
// przy próbkowaniu tekstury funkcją textureOffset,
// GL_MIN_PROGRAM_TEXEL_OFFSET
const int gl_MinProgramTexelOffset = -8;

// maksymalna wartość przesunięcia współrzędnych tekstury
// przy próbkowaniu tekstury funkcją textureOffset,
// GL_MAX_PROGRAM_TEXEL_OFFSET
const int gl_MaxProgramTexelOffset = 7;
```

Wbudowane zmienne jednorodne

Wbudowane zmienne jednorodne opisują bieżącą konfigurację potoku OpenGL i są dostępne dla wszystkich rodzajów shaderów. W wersji 4.20 GLSL zmienne te opisują parametry bufora głębokości (zmienna stanu `GL_DEPTH_RANGE`), które są zgrupowane w strukturze `gl_DepthRangeParameters` oraz ilość próbek przypadających na pojedynczy piksel obrazu w buforze ramki:

```
struct gl_DepthRangeParameters
{
    float near;   // minimalna wartość bufora głębokości
    float far;    // maksymalna wartość bufora głębokości
    float diff;   // wynik odejmowania far - near
};
uniform gl_DepthRangeParameters gl_DepthRange;

uniform int gl_NumSamples;
```

Wbudowane funkcje

Język GLSL zawiera szereg funkcji operujących na danych skalarnych i wektorowych. Większość z nich dostępna jest w każdym rodzaju shadera, ale przeznaczenie niektórych funkcji powoduje, że są one dostępne tylko w niektórych typach shaderów. Wbudowane w GLSL funkcje można podzielić na trzy zasadnicze kategorie:

- odzwierciedlające pewną funkcjonalność sprzętu, np. próbkowanie tekstur, które nie mogą być programowo emulowane przez shader,
- realizujące proste operacje (np. `clamp`, `mix`), które są bardzo łatwe do samodzielnej implementacji, ale które mogą być bezpośrednio wspierane sprzętowo,
- reprezentujące operacje sprzętu graficznego, które prawdopodobnie są wspierane przez sprzęt (np. funkcje trygonometryczne).

Wiele funkcji dostępnych w języku GLSL jest nazwanych tak samo jak funkcje z biblioteki języka C. Podstawowa różnica polega jednak na obsłudze przez funkcje GLSL typów wektorowych, nie tylko skalarnych. Mechanizmy języka GLSL umożliwiają zamianę funkcji wbudowanych poprzez prostą ich redeklarację z zachowaniem nazwy i listy argumentów. Jednak należy z tej możliwości korzystać rozważnie, bowiem funkcje wbudowane z założenia są optymalne i zawsze, gdy jest taka możliwość, realizowane sprzętowo.

W znajdującym się dalej opisie funkcji dostępnych w języku GLSL zastosowano pewien schemat zmniejszający ilość prezentowanych prototypów funkcji. Jak typy argumentów i wyjścia funkcji użyto następujących oznaczeń:

- `genType` – oznacza typ `float`, `vec2`, `vec3` lub `vec4`,
- `genIType` – oznacza typ `int`, `ivec2`, `ivec3` lub `ivec4`,
- `genUType` – oznacza typ `uint`, `uvec2`, `uvec3` lub `uvec4`,
- `genBType` – oznacza typ `bool`, `bvec2`, `bvec3` lub `bvec4`,

- `genDType` – oznacza typ `double`, `dvec2`, `dvec3` lub `dvec4`,
- `mat` – oznacza dowolny typ macierzowy z liczbami zmiennoprzecinkowymi pojedynczej precyzji,
- `dmat` – oznacza dowolny typ macierzowy z liczbami zmiennoprzecinkowymi podwójnej precyzji.

Wyjątki od powyższej zasady opisane są odrębnie. W przypadku użycia funkcji z argumentami określonymi jako `genType`, `genIType`, `genUType`, `genBType` lub `genDType` musi być zachowana taka sama ilość składowych w każdym argumencie jak i w zwracanej przez funkcję wartości.

Funkcje trygonometryczne

Grupa funkcji trygonometrycznych została przedstawiona w Tabela 14. Poza funkcjami trygonometrycznymi, znajdują się tu także funkcje kątowe, cyklometryczne (odwrotne do funkcji trygonometrycznych), hiperboliczne i aerahiperboliczne (odwrotne do funkcji hiperbolicznych). Argumenty funkcji trygonometrycznych podawane są w radianach, a opisane operacje wykonywane są na każdej składowej parametru (lub parametrów) funkcji.

Funkcja	Opis
`genType radians( genType degrees )`	Konwersja stopni na radiany, $\frac{\pi}{180} \cdot degrees$.
`genType degrees( genType radians )`	Konwersja radianów na stopnie, $\frac{180}{\pi} \cdot radians$.
`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, zbiór wartości $\langle -\frac{\pi}{2}; \frac{\pi}{2} \rangle$, wartość funkcji nieokreślona, gdy $\lvert x \rvert > 1$.
`genType acos( genType x )`	Funkcja cyklometryczna arcus cosinus, zbiór wartości $\langle 0; \pi \rangle$, wartość funkcji nieokreślona, gdy $\lvert x \rvert > 1$.
`genType atan( genType y, genType x )`	Funkcja cyklometryczna arcus tangens $\frac{y}{x}$, znaki argumentów *x* i *y* określają, w której ćwiartce leży obliczany kąt, zbiór wartości $\langle -\pi; \pi \rangle$, wartość funkcji nieokreślona, gdy *x* i *y* są równe 0.
`genType atan( genType y_over_x )`	Funkcja cyklometryczna arcus tangens, zbiór wartości $\langle -\frac{\pi}{2}; \frac{\pi}{2} \rangle$.
`genType sinh( genType x )`	Funkcja sinus hiperboliczny, $\frac{e^x - e^{-x}}{2}$.
`genType cosh( genType x )`	Funkcja cosinus hiperboliczny, $\frac{e^x + e^{-x}}{2}$.
`genType tanh( genType x )`	Funkcja tangens hiperboliczny, $\frac{\sinh x}{\cosh x}$.
`genType asinh( genType x )`	Funkcja area sinus hiperboliczny (odwrotność `sinh`).
`genType acosh( genType x )`	Funkcja area cosinus hiperboliczny (odwrotność `cosh`), wartość funkcji nieokreślona, gdy $x < 1$.
`genType atanh( genType x )`	Funkcja area tangens hiperboliczny (odwrotność `tanh`), wartość funkcji nieokreślona, gdy $\lvert x \rvert \geq 1$.

Tabela 14 Zestawienie funkcji trygonometrycznych

Funkcje wykładnicze

Funkcje wykładnicze, potęgowe, pierwiastkowe i logarytmiczne przedstawione są w Tabela 15. Opis poniższych funkcji dotyczy operacji na każdej składowej parametru (lub parametrów) funkcji.

Funkcja	Opis
`genType pow( genType x, genType y )`	Funkcja potęgowa x^y, wartość funkcji nieokreślona, gdy $x < 0$ oraz gdy $x = 0$ i $y \leq 0$.

`genType exp( genType x )`	Funkcja wykładnicza e^x.
`genType log( genType x )`	Logarytm naturalny $\ln x$, , wartość funkcji nieokreślona, gdy $x \leq 0$.
`genType exp2( genType x )`	Funkcja wykładnicza 2^x.
`genType log2( genType x )`	Logarytm o podstawie 2, wartość funkcji nieokreślona, gdy $x \leq 0$.
`genType sqrt( genType x )` `genDType sqrt( genDType x )`	Pierwiastek kwadratowy $\sqrt{x}$, wartość funkcji nieokreślona, gdy $x < 0$.
`genType inversesqrt( genType x )` `genDType inversesqrt( genDType x )`	Funkcja $\frac{1}{\sqrt{x}}$, wartość funkcji nieokreślona, gdy $x \leq 0$.

Tabela 15 Zestawienie funkcji wykładniczych

Funkcje ogólne

Funkcje ogólne zestawione w Tabela 16 operują na różnego rodzaju argumentach, przy czym prezentowany opis dotyczy działania na każdej składowej lub składowych argumentu.

Funkcja	Opis

`genType abs( genType x )`
`genIType abs( genIType x )`
`genDType abs( genDType x )`
	Wartość bezwzględna $\begin{cases} x & gdy \quad x \geq 0 \\ -x & gdy \quad x < 0 \end{cases}$.

`genType sign( genType x )`
`genIType sign( genIType x )`
`genDType sign( genDType x )`
	Funkcja $\begin{cases} -1 & gdy \quad x < 0 \\ 0 & gdy \quad x = 0. \\ 1 & gdy \quad x > 0 \end{cases}$

`genType floor( genType x )` `genDType floor( genDType x )`	Największa liczba całkowita, która jest mniejsza lub równa x.
`genType trunc( genType x )` `genDType trunc( genDType x )`	Najbliższa liczba całkowita do x, której wartość bezwzględna jest nie większa niż wartość bezwzględna x.
`genType round( genType x )` `genDType round( genDType x )`	Najbliższa liczba całkowita do x, gdzie wartość ułamkowa 0,5 jest zaokrąglana w sposób określony przez implementację; w szczególności funkcja `round` może zwracać takie same wartości jak `roundEven`.
`genType roundEven( genType x )` `genDType roundEven( genDType x )`	Najbliższa liczba całkowita do x, gdzie wartość ułamkowa 0,5 jest zaokrąglana do najbliższej liczby parzystej (np. 3,5 oraz 4,5 zwracają tą samą wartość 4,0).
`genType ceil( genType x )` `genDType ceil( genDType x )`	Najmniejsza liczba całkowita, która jest większa lub równa x.
`genType fract( genType x )` `genDType fract( genDType x )`	Funkcja $x - \text{floor}\, x$.

`genType mod( genType x, float y )`
`genType mod( genType x, genType y )`
`genDType mod( genDType x, double y )`
`genDType mod( genDType x, genDType y )`
	Funkcja $x - y \cdot \text{floor}\frac{x}{y}$.

`genType modf( genType x,`
`out genType i )`
`genDType modf( genDType x,`
`out genDType i )`
	Ułamkowa część x (zwracana wartość funkcji) oraz całkowita część x (wartość w parametrze `i`), obie zwracane wartości mają taki sam znak jak x.

`genType min( genType x, genType y )`
`genType min( genType x, float y )`
`genDType min( genDType x, genDType y )`
`genDType min( genDType x, double y )`
`genIType min( genIType x, genIType y )`
`genIType min( genIType x, int y )`
`genUType min( genUType x, genUType y )`
`genUType min( genUType x, uint y )`
	Minimum.

`genType max( genType x, genType y )`
`genType max( genType x, float y )`
`genDType max( genDType x, genDType y )`
`genDType max( genDType x, double y )`
`genIType max( genIType x, genIType y )`
`genIType max( genIType x, int y )`
`genUType max( genUType x, genUType y )`
`genUType max( genUType x, uint y )`
	Maksimum.

`genType clamp( genType x,`
`genType minVal,`
`genType maxVal )`
`genType clamp( genType x,`
`float minVal,`
`float maxVal )`
`genDType clamp( genDType x,`
`genDType minVal,`
`genDType maxVal )`
`genDType clamp( genDType x,`
`double minVal,`
`double maxVal )`
`genIType clamp( genIType x,`
`genIType minVal,`
`genIType maxVal )`
`genIType clamp( genIType x,`
`int minVal,`
`int maxVal )`
`genUType clamp( genUType x,`
`genUType minVal,`
`genUType maxVal )`
`genUType clamp( genUType x,`
`uint minVal,`
`uint maxVal )`
	Funkcja $\min(\max(x, minVal), maxVal)$; wartość niezdefiniowana, gdy $minVal > maxVal$.

`genType mix(genType x,`
`genType y,`
`genType a )`
`genType mix( genType x,`
`genType y,`
`float a )`
`genDType mix( genDType x,`
`genDType y,`
`genDType a )`
`genDType mix( genDType x,`
`genDType y,`
`double a )`
	Funkcja liniowego mieszania x i y, $x \cdot (1 - a) + y \cdot a$.

`genType mix( genType x,`
`genType y,`
`genBType a )`
`genDType mix( genDType x,`
`genDType y,`
`genBType a )`
	Funkcja określa na podstawie wartości kolejnych składowych `a` wartości składowych, którego z parametrów wejściowych `x` i `y` są zwracane w jako składowa wyniku; jeżeli wartość składowej `a` wynosi `true`, zwracana jest odpowiednia składowa parametru `y`; w przeciwnym wypadku zwracana jest odpowiednia składowa parametru `x`. Warto zauważyć, że funkcja ta daje odmienne wyniki od `genType mix(genType x, genType y, genType a )`, gdzie `a` jest wektorem liczb typu `bool`.

`genType step( float edge,`
`genType x )`
`genType step( genType edge,`
`genType x )`
`genDType step( genDType edge,`
	Funkcja $\begin{cases} 0{,}0 & gdy \quad x < edge \\ 1{,}0 & gdy \quad x \geq edge \end{cases}$.

`genDType x )`
`genDType step( double edge,`
`genDType x )`	

`genType smoothstep( genType edge0,`
`genType edge1,`
`genType x )`
`genType smoothstep( float edge0,`
`float edge1,`
`genType x )`
`genDType smoothstep( genDType edge0,`
`genDType edge1,`
`genDType x )`
`genDType smoothstep( double edge0,`
`double edge1,`
`genDType x )`
	Funkcja gładkiej interpolacji Hermite'a:
	$\begin{cases} 0{,}0 & gdy & x \le edge0 \\ t \cdot t \cdot (3 - 2 \cdot t) & gdy & edge0 < x < edge1, \\ 1{,}0 & gdy & x \ge edge1 \end{cases}$
	gdzie $t = \text{clamp}\left(\frac{x - edge0}{edge1 - edge0}, 0{,}1\right)$; wartość funkcji niezdefiniowana, gdy $edge0 \ge edge1$.

`genBType isnan( genType x )` `genBType isnan( genDType x )`	Funkcja zwraca `true`, gdy x reprezentuje NaN (symbol nieoznaczony, wartość niebędąca liczbą) w bieżącej implementacji zbioru liczb zmiennoprzecinkowych; w przeciwnym wypadku funkcja zwraca `false`, w tym także dla implementacji bez reprezentacji NaN.
`genBType isinf( genType x )` `genBType isinf( genDType x )`	Funkcja zwraca `true`, gdy x reprezentuje dodatnią lub ujemną nieskończoność w bieżącej implementacji zbioru liczb zmiennoprzecinkowych; w przeciwnym wypadku funkcja zwraca `false`, w tym także dla implementacji bez reprezentacji nieskończoności (Inf).

`genIType floatBitsToInt( genType`
`value )`
`genUType floatBitsToUint( genType`
`value )`
	Funkcja zwraca liczbę całkowitą ze znakiem lub bez znaku zawierającą bitową reprezentację liczby zmiennoprzecinkowej.

`genType intBitsToFloat( genIType`
`value )`
`genType uintBitsToFloat( genUType`
`value )`
	Funkcja zwraca liczbę zmiennoprzecinkową odpowiadającą bitowej reprezentacji zawartej w liczbie całkowitej ze znakiem lub bez znaku. W przypadku przekazania do funkcji wartości NaN wynik jest niezdefiniowany, natomiast w przypadku przekazania do funkcji wartości Inf (nieskończoność) zwracana jest odpowiednia wartość Inf.

`genType fma( genType a, genType b,`
`genType c )`
`genDType fma( genDType a, genDType b,`
`genDType c )`
	Funkcja oblicza wyrażenie $a \cdot b + c$. Funkcja z założenia jest realizowana jako pojedyncza operacja, przy czym specyfikacja dopuszcza potencjalną możliwość różnej precyzji funkcji w stosunku do sekwencji pojedynczych operacji.

`genType frexp( genType x,`
`out genIType exp )`
`genDType frexp( genDType x,`
`out genIType exp )`
	Funkcja zwraca zmiennoprzecinkową mantysę z przedziału $\langle 0{,}5; 1{,}0)$ i wykładnik całkowity potęgi o podstawie 2 (parametr wyjściowy `exp`), zgodnie ze wzorem: $x = significand \cdot 2^{exponent}$. W przypadku przekazania wartości zerowej mantysa i wykładnik są równe 0. Wynik funkcji jest nieokreślony dla wartości NaN i Inf.

`genType ldexp( genType x,`
`in genIType exp )`
`genDType ldexp( genDType x,`
`in genIType exp )`
	Funkcja zwraca liczbę zmiennoprzecinkową z mantysy (parametr `x`) oraz wykładnika całkowitego potęgi o podstawie 2 (parametr `exp`), zgodnie ze wzorem: $significand \cdot 2^{exponent}$. W przypadku przekroczenia

	reprezentowalnego zakresu wartości zmiennoprzecinkowych, wynik funkcji jest nieokreślony.

Tabela 16 Zestawienie funkcji ogólnych

Funkcje pakowania i rozpakowywania zmiennoprzecinkowego

Zestawienie funkcji pakowania i rozpakowywania liczb zmiennoprzecinkowych zawiera Tabela 17. W przeciwieństwie do większości pozostałych funkcji, funkcje z tej grypy nie operują na poszczególnych składowych, ale na całości swoich argumentów.

Funkcja	Opis

`uint packUnorm2x16( vec2 v )`
`uint packSnorm2x16( vec2 v )`
`uint packUnorm4x8( vec4 v )`
`uint packSnorm4x8( vec4 v )`
	Funkcja konwertuje znormalizowana liczbę zmiennoprzecinkową (o wartościach z przedziału $\langle 0; 1\rangle$ lub $\langle -1; 1\rangle$) do 8- lub 16-bitowej liczby całkowitej. Wynik konwersji umieszczany jest w zwracanej 32-bitowej liczbie całkowitej bez znaku. Konwersja pojedynczej składowej wektora `v` wykonywana jest w następujący sposób:
	packUnorm2x16: $\text{round}(\text{clamp}(c, 0,1) \cdot 65535)$
	packSnorm2x16: $\text{round}(\text{clamp}(c, -1,1) \cdot 32767)$
	packUnorm4x8: $\text{round}(\text{clamp}(c, 0,1) \cdot 255)$
	packSnorm4x8: $\text{round}(\text{clamp}(c, -1,1) \cdot 127)$
	Pierwsza składowa wektora `v` jest zapisywana do najmniej znaczących bitów wyniku, natomiast ostatnia składowa wektora `v` jest zapisywana do najbardziej znaczących bitów wyniku.

`vec2 unpackUnorm2x16( uint p )`
`vec2 unpackSnorm2x16( uint p )`
`vec4 unpackUnorm4x8( uint p )`
`vec4 unpackSnorm4x8( uint p )`
	Funkcja traktuje 32-bitową liczbę całkowitą bez znaku `p` jako parę 16-bitowych liczb całkowitych ze znakiem lub bez znaku względnie jako czwórkę 8-bitowych liczb całkowitych ze znakiem lub bez znaku. Następnie każda taka liczba jest konwertowana do znormalizowanej wartości zmiennoprzecinkowej (o wartościach z przedziału $\langle 0; 1\rangle$ lub $\langle -1; 1\rangle$) i umieszczana w wektorze wynikowym. Konwersja pojedynczej liczby f wykonywana jest w następujący sposób:
	unpackUnorm2x16: $\frac{f}{65535}$
	unpackSnorm2x16: $\text{clamp}\left(\frac{f}{32767}, -1,1\right)$
	unpackUnorm4x8: $\frac{f}{255}$
	unpackSnorm4x8: $\text{clamp}\left(\frac{f}{127}, -1,1\right)$
	Pierwsza składowa wynikowego wektora zawiera liczbę skonwertowaną z najmniej znaczących bitów `p`, natomiast ostatnia składowa wektora zawiera liczbę skonwertowaną z najbardziej znaczących bitów `p`.

`double packDouble2x32( uvec2 v )`	Funkcja zwraca liczbę zmiennoprzecinkową podwójnej precyzji odpowiadającą bitowej reprezentacji zawartej w dwóch 32-bitowych liczbach całkowitych bez znaku. Pierwsza składowa wektora `v` zawiera 32 mniej znaczące bity wyniku, natomiast druga zawiera 32 bardziej znaczące bity wyniku. W przypadku przekazania do funkcji wartości NaN lub Inf (nieskończoność) wynik jest niezdefiniowany.
`uvec2 unpackDouble2x32( double v )`	Funkcja zwraca dwie liczby całkowite ze znakiem zawierające bitową reprezentację liczby

	zmiennoprzecinkowej podwójnej precyzji. Pierwsza składowa wektora wynikowego zawiera 32 mniej znaczące bity liczby `v`, natomiast druga zawiera 32 bardziej znaczące bity `v`.
`uint packHalf2x16( vec2 v )`	Funkcja zwraca 32-bitową liczbę całkowitą bez znaku zawierającą dwie liczby zmiennoprzecinkowe połówkowej precyzji (typ `GLhalf`) powstałe z dwóch liczb zmiennoprzecinkowych z wektora `v`. 16 mniej znaczących bitów wyniku zawierają pierwszą składową wektora `v`, natomiast druga składowa zawarta jest w 16 bardziej znaczących bitach wyniku.
`vec2 unpackHalf2x16( uint v )`	Funkcja zwraca dwie liczby zmiennoprzecinkowe powstałe z 32-bitowej liczby całkowitej bez znaku zawierającej dwie liczby zmiennoprzecinkowe połówkowej precyzji (typ `GLhalf`). Pierwsza składowa wektora wynikowego zawiera liczbę powstałą z 16 mniej znaczących bitów liczby `v`, natomiast druga składowa powstaje po konwersji 16 bardziej znaczących bitów `v`.

Tabela 17 Zestawienie funkcji pakowania i rozpakowywania liczb zmiennoprzecinkowych.

Funkcje geometryczne

Funkcje geometryczne, przedstawione w Tabela 18, operują na typach wektorowych traktowanych jako wektory, nie działają natomiast na poszczególnych składowych.

Funkcja	Opis
`float length( genType x )` `double length( genDType x )`	Długość wektora, $\sqrt{x[0]^2 + x[1]^2 + \ldots}$

`float distance( genType p0,`
`genType p1 )`
`double distance( genDType p0,`
`genDType p1 )`
	Odległość między punktami `p0` i `p1`, $\text{length}(p0 - p1)$.

`float dot( genType x,`
`genType y )`
`double dot( genDType x,`
`genDType y )`
	Iloczyn skalarny `x` i `y`, $x[0] \cdot y[0] + x[1] \cdot y[1] + \ldots$

`vec3 cross( vec3 x, vec3 y )` `dvec3 cross( dvec3 x, dvec3 y )`	Iloczyn wektorowy `x` i `y`, $\begin{bmatrix} x[1] \cdot y[2] - y[1] \cdot x[2] \\ x[2] \cdot y[0] - y[2] \cdot x[0] \\ x[0] \cdot y[1] - y[0] \cdot x[1] \end{bmatrix}$.
`genType normalize( genType x )` `genDType normalize( genDType x )`	Normalizacja wektora (wektor o długości równej 1).

`genType faceforward( genType N,`
`genType I,`
`genType Nref )`
`genDType faceforward( genDType N,`
`genDType I,`
`genDType Nref )`
	Funkcja $\begin{cases} N & gdy \quad \text{dot}(Nref, I) < 0 \\ -N & gdy \quad \text{dot}(Nref, I) \geq 0 \end{cases}$.

`genType reflect( genType I,`
`genType N )`
`genDType reflect( genDType I,`
`genDType N )`
	Wektor kierunku odbicia dla wektora `I` padającego na płaszczyznę opisaną jednostkowym wektorem normalnym `N`, $I - 2 \cdot \text{dot}(N, I) \cdot N$.

`genType refract( genType I,`
`genType N,`
`float eta )`
`genDType refract( genDType I,`
`genDType N,`
`float eta )`
	Wektor załamania dla jednostkowego wektora `I` padającego na płaszczyznę opisaną jednostkowym wektorem normalnym `N`, przy współczynniku refrakcji `eta`, $\begin{cases} genType(0.0) & dla \quad k < 0.0 \\ eta \cdot I - (eta \cdot \text{dot}(N, I) + \text{sqrt}(k)) \cdot N & dla \quad k \geq 0.0 \end{cases}$ gdzie $k = 1.0 - eta \cdot eta \cdot (1.0 - \text{dot}(N, I) + \text{sqrt}(k)) \cdot N$.

Tabela 18 Zestawienie funkcji geometrycznych

Funkcje macierzowe

Zestawienie funkcji działających na macierzach przedstawia Tabela 19.

Funkcja	Opis
`mat matrixCompMult( mat x, mat y )`	Mnożenie równoległe elementów macierzy, wynik w pozycji $[i][j]$ jest iloczynem skalarnym $x[i][j]$ i $y[i][j]$. 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 dając w efekcie macierz, w której liczba wierzy odpowiada ilości składowych wektora `c`, a liczba kolumn ilości składowych wektora `r`.

`mat2 transpose( mat2 m )`
`mat3 transpose( mat3 m )`
`mat4 transpose( mat4 m )`
`mat2x3 transpose( mat3x2 m )`
`mat3x2 transpose( mat2x3 m )`
`mat2x4 transpose( mat4x2 m )`
`mat4x2 transpose( mat2x4 m )`
`mat3x4 transpose( mat4x3 m )`
`mat4x3 transpose( mat3x4 m )`
	Transponowanie macierzy. Macierz wejściowa `m` nie jest modyfikowana.

`float determinant( mat2 m )`
`float determinant( mat3 m )`
`float determinant( mat4 m )`
	Wyznacznik macierzy.

`mat2 inverse( mat2 m )`
`mat3 inverse( mat3 m )`
`mat4 inverse( mat4 m )`
	Odwracanie macierzy, macierz wejściowa `m` nie jest modyfikowana. Wartość zwracana przez funkcję jest nieokreślona, gdy macierz `m` jest macierzą osobliwą.

Tabela 19 Zestawienie funkcji macierzowych

Funkcje porównujące wektory

Operatory relacyjne i porównania (<, <=, >, >-, ==, !=) są zdefiniowane do generowania skalarnych wartości logicznych. Do operacji relacji i porównania działających na wektorach i zwracających wektory z liczbami typu `bool` stosuje się funkcje wymienione w Tabela 20. Użyte w opisach funkcji typy `bvec` oznaczają odpowiednio: `bvec2`, `bvec3` lub `bvec4`. Analogiczna zasada dotyczy `vec` - są to typy: `vec2`, `vec3`, `vec4`, `dvec2`, `dvec3` lub `dvec4`; `ivec`, co odpowiada `ivec2`, `ivec3` lub `ivec4` oraz `uvec`, co odpowiada `uvec2`, `uvec3` lub `uvec4`. W każdym przypadku wymiary porównywanych wektorów muszą być zgodne.

Funkcja	Opis

`bvec lessThan( vec x, vec y )`
`bvec lessThan( ivec x, ivec y )`
`bvec lessThan( uvec x, uvec y )`
	Wynik porównania `x < y` składowych wektorów.

`bvec lessThanEqual( vec x, vec y )`
`bvec lessThanEqual( ivec x, ivec y )`
`bvec lessThanEqual( uvec x, uvec y )`
	Wynik porównania `x <= y` składowych wektorów.

`bvec greaterThan( vec x, vec y )`
`bvec greaterThan( ivec x, ivec y )`
`bvec greaterThan( uvec x, uvec y )`
	Wynik porównania `x > y` składowych wektorów.

`bvec greaterThanEqual( vec x, vec y )`
`bvec greaterThanEqual( ivec x, ivec y )`
`bvec greaterThanEqual( uvec x, uvec y )`
	Wynik porównania `x >= y` składowych wektorów.

`bvec equal( vec x, vec y )`	Wynik porównania `x == y` składowych

`bvec equal( ivec x, ivec y )`
`bvec equal( uvec x, uvec y )`
`bvec equal( bvec x, bvec y )`
	wektorów.

`bvec notEqual( vec x, vec y )`
`bvec notEqual( ivec x, ivec y )`
`bvec notEqual( uvec x, uvec y )`
`bvec notEqual( bvec x, bvec y )`
	Wynik porównania `x != y` składowych wektorów.

`bool any( bvec x )`	Funkcja zwraca `true`, gdy dowolna składowa `x` ma wartość `true`.
`bool all( bvec x )`	Funkcja zwraca `true`, gdy wszystkie składowe `x` mają wartość `true`.
`bvec not( bvec x )`	Funkcja zwraca negację logiczną składowych `x`.

Tabela 20 Zestawienie funkcji porównujących wektory.

Funkcje całkowite

Funkcje całkowite przedstawione w Tabela 21 działają na poszczególnych składowych argumentów. Oznaczenie $[a, b]$ wskazuje zbiór bitów liczby, włącznie z bitami a i b, przy czym najmłodszy bit ma numer 0.

Funkcja	Opis

`genUType uaddCarry( genUType x,`
`genUType y,`
`out genUType carry )`
	Dodawanie modulo 2^{32} dwóch 32-bitowych liczb całkowitych bez znaku. Jeżeli wynik jest mniejszy od 2^{32} zmienna `carry` ustawiana jest na wartość 0; w przeciwnym wypadku jej wartość wynosi 1.

`genUType usubBorrow( genUType x,`
`genUType y,`
`out genUType borrow )`
	Odejmowanie dwóch 32-bitowych liczb całkowitych bez znaku, `y` od `x`. Funkcja zwraca różnicę w przypadku, gdy jest ona nieujemna, a w pozostałych przypadkach wartość 2^{32} powiększoną o różnicę. Jeżeli $x \geq y$ zmienna `borrow` ustawiana jest na wartość 0; w przeciwnym wypadku jej wartość wynosi 1.

`void umulExtended( genUType x,`
`genUType y,`
`out genUType msb,`
`out genUType lsb )`
`void imulExtended( genIType x,`
`genIType y,`
`out genIType msb,`
`out genIType lsb )`
	Mnożenie dwóch 32-bitowych liczb całkowitych bez znaku, przy czym 64-bitowy wynik umieszczany jest w dwóch 32-bitowych liczbach: `lsb` zawiera 32 młodsze bity wyniku, `msb` zawiera 32 starsze bity wyniku mnożenia.

`genIType bitfieldExtract( genIType value,`
`int offset,`
`int bits )`
`genUType bitfieldExtract( genUType value,`
`int offset,`
`int bits )`
	Funkcja wydziela bity z przedziału $[offset, offset + bits - 1]$ ze zmiennej `value`. W przypadku zmiennej bez znaku najbardziej znaczący bit wyniku przyjmuje wartość równą 0; natomiast w przypadku zmiennej ze znakiem najbardziej znaczący bit wyniku przyjmuje wartość bitu o numerze równym $offset + bits - 1$.
	Jeżeli wartość `bits` jest równa 0, zwracany wynik także ma wartość równą 0. Wartość funkcji jest nieokreślona, gdy `offset` lub `bits` są ujemne, lub ich suma jest większa niż ilość bitów używana do zapisu zmiennej `value`.

`genIType bitfieldInsert( genIType base,`	Funkcja wstawia `bits` najmniej znaczących

genIType insert, int offset, int bits) genUType bitfieldInsert(genUType base, genUType insert, int offset, int bits)	bitów zmiennej `insert` do zmiennej `base`. Zwracany wynik ma $[offset, offset + bits - 1]$ bitów pobranych z $[0, bits - 1]$ bitów zmiennej `insert`, oraz pozostałe bity pochodzące bezpośrednio ze zmiennej `base`. Jeżeli wartość zmiennej `bits` wynosi 0, funkcja zwraca wartość zmiennej `base`. Wartość funkcji jest nieokreślona, gdy `offset` lub `bits` są ujemne, lub ich suma jest większa niż ilość bitów używana do zapisu operandów.
`genIType bitfieldReverse( genIType value )` `genUType bitfieldReverse( genUType value )`	Funkcja zwraca wartość powstałą z odwrócenia bitów zmiennej `value`.
`genIType bitCount( genIType value )` `genIType bitCount( genUType value )`	Funkcja zwraca ilość bitów zmiennej `value` o wartości równej 1.
`genIType findLSB( genIType value )` `genIType findLSB( genUType value )`	Funkcja zwraca numer najmniej znaczącego bitu zmiennej `value` o wartości równej 1. Jeżeli wartość zmiennej `value` jest równa 0, funkcja zwraca wartość -1.
`genIType findMSB( genIType value )` `genIType findMSB( genUType value )`	Dla liczb dodatnich funkcja zwraca numer najbardziej znaczącego bitu zmiennej `value` o wartości równej 1. Natomiast dla liczby ujemnej funkcja zwraca numer najbardziej znaczącego bitu zmiennej `value` o wartości równej 0. Jeżeli wartość zmiennej `value` jest równa 0 lub -1 (odpowiednio brak w zmiennej `value` bitów o wartości 1 lub 0), funkcja zwraca wartość -1.

Tabela 21 Zestawienie funkcji całkowitych.

Funkcje próbkujące tekstury

Funkcje próbkujące tekstury dostępne są we wszystkich rodzajach shaderów. Jednakże poziom mipmapy jest automatycznie obliczany tylko w shaderach fragmentów. Pozostałe rodzaje shaderów operują na bazowym poziomie mipmapy o numerze 0. Sama tekstura jest inicjowana przy użyciu odpowiednich funkcji API OpenGL. Właściwości tekstur, takie jak rozmiar, format piksela, ilość wymiarów, metoda filtracji, ilość poziomów mipmap, porównanie głębi definiowane są także przez odpowiednie funkcje API OpenGL. Dostęp do tekstur uzyskiwany jest za pośrednictwem uchwytu zawartego w parametrze `sampler` funkcji próbkujących tekstury, których opis zawarto w Tabela 22.

Dane tekstury mogą być zapisane jako liczby zmiennoprzecinkowe, normalizowane liczby całkowite bez znaku oraz liczby całkowite ze znakiem i bez znaku. Rodzaj danych określa wewnętrzny format tekstury. Generalnie wartość zwracana podczas próbkowania tekstury to liczba zmiennoprzecinkowa, liczba całkowita ze znakiem lub liczba całkowita bez znaku, w zależności od rodzaju uchwytu przekazanego do funkcji.

Rodzaj danych tekstury musi zgadzać się typem uchwytu tekstury – nie wszystkie kombinacje są obsługiwane przez OpenGL. Uchwyty zmiennoprzecinkowe obsługują tekstury z danymi w postaci liczb zmiennoprzecinkowych i normalizowanych liczb całkowitych bez znaku. Uchwyty całkowite ze znakiem i uchwyty całkowite bez znaku współpracują wyłącznie z teksturami odpowiednio w formatach całkowitych ze znakiem i całkowitych bez znaku.

Funkcje próbkujące tekstury z liczbami całkowitymi ze znakiem zwracają wektory `ivec4`, a funkcje próbkujące tekstury z liczbami całkowitymi bez znaku zwracają wektory `uvec4`. Próbkowanie tekstury zawierającej dane zmiennoprzecinkowe lub normalizowane liczby całkowite bez znaku zwraca wektory `vec4` z wartościami składowych z przedziału $\langle 0; 1 \rangle$. W zamieszczonych w Tabela 22 prototypach funkcji zwracany typ `gvec4` oznacza odpowiednio `vec4`, `ivec4` lub `uvec4`.

Analogiczna zasada dotyczy argumentu w postaci uchwytu tekstury zawierającego literę „g”, który oznacza uchwyt tekstury zgodny z typem zwracanym przez funkcję, tak jak zostało to opisane we wcześniejszym akapicie.

W funkcji próbkujących tekstury głębi wykonywane jest porównywanie wartości głębokości z tekstury wskazanej w uchwycie z wartością referencyjną głębokości D_{ref}. Próbkowana tekstura musi być teksturą głębi, w przeciwnym wypadku wynik operacji jest nieokreślony. Taki sam nieokreślony rezultat da próbkowanie tekstury głębi przy wyłączonym porównywaniu głębi. Mechanizm próbkowania tekstur głębi opiszemy bliżej w odcinku kursu poświęconemu teksturowaniu.

Parametr `bias` jest opcjonalnym argumentem funkcji dostępnym w shaderach fragmentów, który nie jest akceptowalny w pozostałych rodzajach shaderów. Jeżeli parametr `bias` jest wyspecyfikowany, jego wartość jest dodawana do poziomu mipmapy używanego do operacji dostępu do tekstury. Parametry `bias` i `lod` (LOD, ang. *level of detail* – poziom mipmapy) nie występują w funkcjach próbkujących tekstury prostokątne, tekstury wielopróbkowania oraz tekstury buforowej, ponieważ tekstury te nie posiadają mipmap.

Bezpośredni poziom mipmapy tekstury jest wybierany w następujący sposób: Dla tekstur, które nie mają mipmap, używana jest bezpośrednio tekstura bazowa. Jeżeli tekstura posiada mipmapy i próbkowanie odbywa się w shaderze fragmentów, LOD jest obliczany przez implementację i następnie używany przy próbkowaniu. Jeżeli tekstura posiada mipmapy, ale próbkowanie odbywa się w shaderze wierzchołków, używana jest tekstura bazowa. Niektóre funkcje próbkujące tekstury wymagają jawnego podania pochodnych. Wartości te są niezdefiniowane poza niejednorodną kontrolą przepływu w shaderze fragmentów oraz w pozostałych rodzajach shaderów.

Przy próbkowaniu tekstur sześciennych wektor `P` jest używany do selekcji strony tekstury sześciennej – czyli tekstury dwuwymiarowej. Mechanizm próbkowania tekstur sześciennych opiszemy dokładnie w odcinku kursu poświęconemu teksturowaniu.

Funkcje próbkujące tablice tekstur pobierają dane z następujące warstwy tekstury:

$$\max(0, \min(d - 1, \text{floor}(layer + 0{,}5)))$$

gdzie d jest ilością warstw (wielkością) tablicy tekstur, a $layer$ zawarty jest w parametrach funkcji opisanych poniżej.

Funkcja	Opis

`gvec4 texture( gsampler1D sampler, float P [, float bias] )`
`gvec4 texture( gsampler2D sampler, vec2 P [, float bias] )`
`gvec4 texture( gsampler3D sampler, vec3 P [, float bias] )`
`gvec4 texture( gsamplerCube sampler, vec3 P [, float bias] )`
`float texture( sampler1DShadow sampler, vec3 P [, float bias] )`
`float texture( sampler2DShadow sampler, vec3 P [, float bias] )`
`float texture( samplerCubeShadow sampler, vec4 P [, float bias] )`
`gvec4 texture( gsampler1DArray sampler, vec2 P [, float bias] )`
`gvec4 texture( gsampler2DArray sampler, vec3 P [, float bias] )`
`gvec4 texture( gsamplerCubeArray sampler, vec4 P [, float bias] )`
`float texture( sampler1DArrayShadow sampler, vec3 P [, float bias] )`
`float texture( sampler2DArrayShadow sampler,`
	Próbkowanie tekstury wskazanej poprzez uchwyt `sampler` przy użyciu współrzędnych `P`.
	W przypadku tekstur głębi w funkcjach bez parametru `compare`, ostatnia składowa wektora `P` jest używana jako wartość referencyjna głębokości D_{ref}., a przedostatnia składowa wektora `P` zawiera numer warstwy tablicy tekstur.
	W funkcjach obsługujących tekstury głębi zawierających parametr `compare`, parametr ten zawiera wartość referencyjną głębokości D_{ref}, a numer warstwy tablicy tekstur zawiera ostatnia składowa wektora `P`.
	W przypadku funkcji obsługujących tablice tekstur (bez tablic tekstur głębi) numer warstwy znajduje się w ostatniej składowej wektora `P`.

`vec4 P )`
`gvec4 texture( gsampler2DRect sampler,`
`vec2 P )`
`float texture( sampler2DRectShadow sampler,`
`vec3 P )`
`float texture( gsamplerCubeArrayShadow`
`sampler, vec4 P,`
`float compare )`	

`gvec4 textureProj( gsampler1D sampler, vec2 P`
`[, float bias] )`
`gvec4 textureProj( gsampler1D sampler, vec4 P`
`[, float bias] )`
`gvec4 textureProj( gsampler2D sampler, vec3 P`
`[, float bias] )`
`gvec4 textureProj( gsampler2D sampler, vec4 P`
`[, float bias] )`
`gvec4 textureProj( gsampler3D sampler, vec4 P`
`[, float bias] )`
`float textureProj( sampler1DShadow sampler,`
`vec4 P [, float bias] )`
`float textureProj( sampler2DShadow sampler,`
`vec4 P [, float bias] )`
`gvec4 textureProj( gsampler2DRect sampler,`
`vec3 P )`
`gvec4 textureProj( gsampler2DRect sampler,`
`vec4 P )`
`float textureProj( sampler2DRectShadow`
`sampler, vec4 P )`
	Próbkowanie tekstury z rzutowaniem. Współrzędne tekstury zawarte w wektorze `P` (ale bez ostatniej składowej) są dzielone przez ostatnią składową `P`. Wynikowa trzecia składowa `P` w przypadku tekstur głębi jest używana jako wartość referencyjna głębokości D_{ref}. Po powyższych obliczeniach próbkowanie tekstury przebiega tak jak w odpowiednich wersjach funkcji `texture`.

`gvec4 textureLod( gsampler1D sampler, float P,`
`float lod )`
`gvec4 textureLod( gsampler2D sampler, vec2 P,`
`float lod )`
`gvec4 textureLod( gsampler3D sampler, vec3 P,`
`float lod )`
`gvec4 textureLod( gsamplerCube sampler,`
`vec3 P, float lod )`
`float textureLod( sampler1DShadow sampler,`
`vec3 P, float lod )`
`float textureLod( sampler2DShadow sampler,`
`vec3 P, float lod )`
`gvec4 textureLod( gsampler1DArray sampler,`
`vec2 P, float lod )`
`gvec4 textureLod( gsampler2DArray sampler,`
`vec3 P, float lod )`
`float textureLod( sampler1DArrayShadow`
`sampler, vec3 P, float lod )`
`gvec4 textureLod( gsamplerCubeArray sampler,`
`vec4 P, float lod )`
	Próbkowanie tekstury tak jak w funkcjach `texture`, ale z bezpośrednim wskazaniem poziomu mipmapy λ_{base} w parametrze `lod` oraz ustawieniem pochodnych cząstkowych w poniższy sposób:
	$\frac{\partial u}{\partial x} = 0 \quad \frac{\partial v}{\partial x} = 0 \quad \frac{\partial w}{\partial x} = 0$
	$\frac{\partial u}{\partial y} = 0 \quad \frac{\partial v}{\partial y} = 0 \quad \frac{\partial w}{\partial y} = 0$

`gvec4 textureOffset( gsampler1D sampler,`
`float P, int offset`
`[, float bias] )`
`gvec4 textureOffset( gsampler2D sampler,`
`vec2 P, ivec2 offset`
`[, float bias] )`
`gvec4 textureOffset( gsampler3D sampler,`
`vec3 P, ivec3 offset`
`[, float bias] )`
`gvec4 textureOffset( gsampler2DRect sampler,`
`vec2 P, ivec2 offset )`
`float textureOffset( sampler2DRectShadow`
`sampler, vec3 P,`
	Próbkowanie tekstury tak jak w funkcjach `texture` z dodaniem przesunięcia współrzędnych tekstury o wartość zawartą w parametrze `offset`. Wartość przesunięcia musi być wyrażeniem stałym i jest limitowana ograniczeniami zależnymi od implementacji, które zawarte są w wbudowanych zmiennych, odpowiednio `gl_MinProgramTexelOffset` i `gl_MaxProgramTexelOffset`. Wartość w parametrze `offset` nie jest

`ivec2 offset )`
`float textureOffset( sampler1DShadow sampler,`
`vec3 P, int offset`
`[, float bias] )`
`float textureOffset( sampler2DShadow sampler,`
`vec3 P, ivec2 offset`
`[, float bias] )`
`gvec4 textureOffset( gsampler1DArray sampler,`
`vec2 P, int offset`
`[, float bias] )`
`gvec4 textureOffset( gsampler2DArray sampler,`
`vec3 P, ivec2 offset`
`[, float bias] )`
`float textureOffset( sampler1DArrayShadow`
`sampler, vec3 P,`
`int offset`
`[, float bias] )`
	uwzględniana do numeru warstwy w tablicach tekstur. Zauważmy również, że nie ma w tej grupie funkcji obsługujących tekstury sześcienne.

`gvec4 texelFetch( gsampler1D sampler, int P,`
`int lod )`
`gvec4 texelFetch( gsampler2D sampler, ivec2 P,`
`int lod )`
`gvec4 texelFetch( gsampler3D sampler, ivec3 P,`
`int lod )`
`gvec4 texelFetch( gsampler2DRect sampler,`
`ivec2 P )`
`gvec4 texelFetch( gsampler1DArray sampler,`
`ivec2 P, int lod )`
`gvec4 texelFetch( gsampler2DArray sampler,`
`ivec3 P, int lod )`
`gvec4 texelFetch( gsamplerBuffer sampler,`
`int P )`
`gvec4 texelFetch( gsampler2DMS sampler,`
`ivec2 P, int sample )`
`gvec4 texelFetch( gsampler2DMSArray sampler,`
`ivec3 P, int sample )`
	Pobranie wybranego teksela tekstury na poziomie mipmapy `lod` (jeżeli występuje) o określonych współrzędnych całkowitych w wektorze `P`. Warstwa tekstury w tablicach tekstur jest określona w ostatniej składowej wektora `P`.

`gvec4 texelFetchOffset( gsampler1D sampler,`
`int P, int lod,`
`int offset )`
`gvec4 texelFetchOffset( gsampler2D sampler,`
`ivec2 P, int lod,`
`ivec2 offset )`
`gvec4 texelFetchOffset( gsampler3D sampler,`
`ivec3 P, int lod,`
`ivec3 offset )`
`gvec4 texelFetchOffset( gsampler2DRect`
`sampler, ivec2 P,`
`ivec2 offset )`
`gvec4 texelFetchOffset( gsampler1DArray`
`sampler, ivec2 P,`
`int lod, int offset )`
`gvec4 texelFetchOffset( gsampler2DArray`
`sampler, ivec3 P,`
`int lod,`
`ivec2 offset )`
	Pobranie pojedynczego teksela tekstury jak w funkcji `texelFetch` z przesunięciem współrzędnych tekstury o `offset` jak w funkcji `textureOffset`.

`gvec4 textureProjOffset( gsampler1D sampler,`
`vec2 P, int offset`
`[, float bias] )`
`gvec4 textureProjOffset( gsampler1D sampler,`
`vec4 P, int offset`
`[, float bias] )`
`gvec4 textureProjOffset( gsampler2D sampler,`
	Próbkowanie tekstury z rzutowaniem współrzędnych jak w funkcji `textureProj` z przesunięciem współrzędnych tekstury o `offset` jak w funkcji `textureOffset`.

vec3 P, ivec2 offset [, float bias]) gvec4 textureProjOffset(gsampler2D sampler, vec4 P, ivec2 offset [, float bias]) gvec4 textureProjOffset(gsampler3D sampler, vec4 P, ivec3 offset [, float bias]) gvec4 textureProjOffset(gsampler2DRect sampler, vec3 P, ivec2 offset) gvec4 textureProjOffset(gsampler2DRect sampler, vec4 P, ivec2 offset) float textureProjOffset(sampler2DRectShadow sampler, vec4 P, ivec2 offset) float textureProjOffset(sampler1DShadow sampler, vec4 P, int offset [, float bias]) float textureProjOffset(sampler2DShadow sampler, vec4 P, ivec2 offset [, float bias])	
gvec4 textureLodOffset(gsampler1D sampler, float P, float lod, int offset) gvec4 textureLodOffset(gsampler2D sampler, vec2 P, float lod, ivec2 offset) gvec4 textureLodOffset(gsampler3D sampler, vec3 P, float lod, ivec3 offset) float textureLodOffset(sampler1DShadow sampler, vec3 P, float lod, int offset) float textureLodOffset(sampler2DShadow sampler, vec3 P, float lod, ivec2 offset) gvec4 textureLodOffset(gsampler1DArray sampler, vec2 P, float lod, int offset) gvec4 textureLodOffset(gsampler2DArray sampler, vec3 P, float lod, ivec2 offset) float textureLodOffset(sampler1DArrayShadow sampler, vec3 P, float lod, int offset)	Próbkowanie tekstury z przesunięciem współrzędnych tekstury i określonym poziomem mipmapy jak w funkcjach `textureLod` i `textureOffset`.
gvec4 textureProjLod(gsampler1D sampler, vec2 P, float lod) gvec4 textureProjLod(gsampler1D sampler, vec4 P, float lod) gvec4 textureProjLod(gsampler2D sampler, vec3 P, float lod) gvec4 textureProjLod(gsampler2D sampler,	Próbkowanie tekstury z rzutowaniem współrzędnych i określonym poziomem mipmapy jak w funkcjach `textureProj` i `textureLod`.

`vec4 P, float lod )`
`gvec4 textureProjLod( gsampler3D sampler,`
`vec4 P, float lod )`
`float textureProjLod( sampler1DShadow sampler,`
`vec4 P, float lod )`
`float textureProjLod( sampler2DShadow sampler,`
`vec4 P, float lod )`	

`gvec4 textureProjLodOffset( gsampler1D sampler, vec2 P, float lod, int offset )`
`gvec4 textureProjLodOffset( gsampler1D sampler, vec4 P, float lod, int offset )`
`gvec4 textureProjLodOffset( gsampler2D sampler, vec3 P, float lod, ivec2 offset )`
`gvec4 textureProjLodOffset( gsampler2D sampler, vec4 P, float lod, ivec2 offset )`
`gvec4 textureProjLodOffset( gsampler3D sampler, vec4 P, float lod, ivec3 offset )`
`float textureProjLodOffset( sampler1DShadow sampler, vec4 P, float lod, int offset )`
`float textureProjLodOffset( sampler2DShadow sampler, vec4 P, float lod, ivec2 offset )`
	Próbkowanie tekstury z rzutowaniem oraz przesunięciem współrzędnych tekstury i określonym poziomem mipmapy jak w funkcjach `textureProj`, `textureLod` i `textureOffset`.

`gvec4 textureGrad( gsampler1D sampler, float P, float dPdx, float dPdy )`
`gvec4 textureGrad( gsampler2D sampler, vec2 P, vec2 dPdx, vec2 dPdy )`
`gvec4 textureGrad( gsampler3D sampler, vec3 P, vec3 dPdx, vec3 dPdy )`
`gvec4 textureGrad( gsamplerCube sampler, vec3 P, vec3 dPdx, vec3 dPdy )`
`gvec4 textureGrad( gsampler2DRect sampler, vec2 P, vec2 dPdx, vec2 dPdy )`
`float textureGrad( sampler2DRectShadow sampler, vec3 P, vec2 dPdx, vec2 dPdy )`
`float textureGrad( sampler1DShadow sampler, vec3 P, float dPdx, float dPdy )`
`float textureGrad( sampler2DShadow sampler, vec3 P, vec2 dPdx, vec2 dPdy )`
`float textureGrad( samplerCubeShadow sampler, vec4 P, vec3 dPdx, vec3 dPdy )`
`gvec4 textureGrad( gsampler1DArray sampler,`
	Próbkowanie tekstury tak jak w funkcji `texture` ale z bezpośrednim obliczaniem gradientów. Pochodne cząstkowe `P` są obliczane względem współrzędnych okienkowych x i y:
	$\frac{\partial s}{\partial x} = \begin{cases} \frac{\partial P}{\partial x} & dla\ tekstury\ 1D \\ \frac{\partial P.s}{\partial x} & pozostałe\ tekstury \end{cases}$
	$\frac{\partial s}{\partial y} = \begin{cases} \frac{\partial P}{\partial y} & dla\ tekstury\ 1D \\ \frac{\partial P.s}{\partial y} & pozostałe\ tekstury \end{cases}$
	$\frac{\partial t}{\partial x} = \begin{cases} 0{,}0 & dla\ tekstury\ 1D \\ \frac{\partial P.t}{\partial x} & pozostałe\ tekstury \end{cases}$
	$\frac{\partial t}{\partial y} = \begin{cases} 0{,}0 & dla\ tekstury\ 1D \\ \frac{\partial P.t}{\partial y} & pozostałe\ tekstury \end{cases}$
	$\frac{\partial r}{\partial x} = \begin{cases} 0{,}0 & dla\ tekstury\ 1D\ i\ 2D \\ \frac{\partial P.p}{\partial x} & tekstury\ kubiczne\ i\ in. \end{cases}$

`vec2 P, float dPdx,`
`float dPdy )`
`gvec4 textureGrad( gsampler2DArray sampler,`
`vec3 P, vec2 dPdx,`
`vec2 dPdy )`
`float textureGrad( sampler1DArrayShadow`
`sampler, vec3 P,`
`float dPdx, float dPdy )`
`float textureGrad( sampler2DArrayShadow`
`sampler, vec4 P,`
`vec2 dPdx, vec2 dPdy )`
`gvec4 textureGrad( samplerCubeArray sampler,`
`vec4 P, vec3 dPdx,`
`vec3 dPdy )`
	$\frac{\partial r}{\partial y} = \begin{cases} 0{,}0 & dla\ tekstury\ 1D\ i\ 2D \\ \frac{\partial P.p}{\partial y} & tekstury\ kubiczne\ i\ in. \end{cases}$
	Dla tekstur kubicznych pochodne cząstkowe `P` są obliczane w układzie współrzędnych używanym przed rzutowaniem współrzędnych tekstury na odpowiednią stronę sześcianu.

`gvec4 textureGradOffset( gsampler1D sampler,`
`float P, float dPdx,`
`float dPdy,`
`int offset )`
`gvec4 textureGradOffset( gsampler2D sampler,`
`vec2 P, vec2 dPdx,`
`vec2 dPdy,`
`ivec2 offset )`
`gvec4 textureGradOffset( gsampler3D sampler,`
`vec3 P, vec3 dPdx,`
`vec3 dPdy,`
`ivec3 offset )`
`gvec4 textureGradOffset( gsampler2DRect`
`sampler, vec2 P,`
`vec2 dPdx, vec2 dPdy,`
`ivec2 offset )`
`float textureGradOffset( sampler2DRectShadow`
`sampler, vec3 P,`
`vec2 dPdx, vec2 dPdy,`
`ivec2 offset )`
`float textureGradOffset( sampler1DShadow`
`sampler, vec3 P,`
`float dPdx,`
`float dPdy,`
`int offset )`
`float textureGradOffset( sampler2DShadow`
`sampler, vec3 P,`
`vec2 dPdx, vec2 dPdy,`
`ivec2 offset )`
`gvec4 textureGradOffset( gsampler1DArray`
`sampler, vec2 P,`
`float dPdx,`
`float dPdy,`
`int offset )`
`gvec4 textureGradOffset( gsampler2DArray`
`sampler, vec3 P,`
`vec2 dPdx, vec2 dPdy,`
`ivec2 offset )`
`float textureGradOffset( sampler1DArrayShadow`
`sampler, vec3 P,`
`float dPdx,`
`float dPdy,`
`int offset )`
`float textureGradOffset( sampler2DArrayShadow`
`sampler, vec4 P,`
`vec2 dPdx, vec2 dPdy,`
`ivec2 offset )`
	Próbkowanie tekstury z określonym gradientem i przesunięciem współrzędnych tekstury jak w funkcjach `textureGrad` i `textureOffset`.

`gvec4 textureProjGrad( gsampler1D sampler, vec2 P, float dPdx, float dPdy )`
`gvec4 textureProjGrad( gsampler1D sampler, vec4 P, float dPdx, float dPdy )`
`gvec4 textureProjGrad( gsampler2D sampler, vec3 P, vec2 dPdx, vec2 dPdy )`
`gvec4 textureProjGrad( gsampler2D sampler, vec4 P, vec2 dPdx, vec2 dPdy )`
`gvec4 textureProjGrad( gsampler3D sampler, vec4 P, vec3 dPdx, vec3 dPdy )`
`gvec4 textureProjGrad( gsampler2DRect sampler, vec3 P, vec2 dPdx, vec2 dPdy )`
`gvec4 textureProjGrad( gsampler2DRect sampler, vec4 P, vec2 dPdx, vec2 dPdy )`
`float textureProjGrad( sampler2DRectShadow sampler, vec4 P, vec2 dPdx, vec2 dPdy )`
`float textureProjGrad( sampler1DShadow sampler, vec4 P, float dPdx, float dPdy )`
`float textureProjGrad( sampler2DShadow sampler, vec4 P, vec2 dPdx, vec2 dPdy )`
	Próbkowanie tekstury z rzutowaniem współrzędnych tekstury jak w funkcji `textureProj` oraz określonym gradientem jak w funkcji `textureGrad`. Pochodne cząstkowe `dPdx` i `dPdy` są już po operacji rzutowania.

`gvec4 textureProjGradOffset( gsampler1D sampler, vec2 P, float dPdx, float dPdy, int offset )`
`gvec4 textureProjGradOffset( gsampler1D sampler, vec4 P, float dPdx, float dPdy, int offset )`
`gvec4 textureProjGradOffset( gsampler2D sampler, vec3 P, vec2 dPdx, vec2 dPdy, vec2 offset )`
`gvec4 textureProjGradOffset( gsampler2D sampler, vec4 P, vec2 dPdx, vec2 dPdy, vec2 offset )`
`gvec4 textureProjGradOffset( gsampler2DRect sampler, vec3 P, vec2 dPdx, vec2 dPdy, ivec2 offset )`
`gvec4 textureProjGradOffset( gsampler2DRect sampler, vec4 P, vec2 dPdx, vec2 dPdy, ivec2 offset )`
	Próbkowanie tekstury z rzutowaniem współrzędnych tekstury i określonym gradientem jak w funkcji `textureProjGrad` oraz z przesunięciem współrzędnych tekstury jak w funkcji `textureOffset`.

`float textureProjGradOffset( sampler2DRectShadow sampler, vec4 P, vec2 dPdx, vec2 dPdy, ivec2 offset )`
`gvec4 textureProjGradOffset( gsampler3D sampler, vec4 P, vec3 dPdx, vec3 dPdy, vec3 offset )`
`float textureProjGradOffset( sampler1DShadow sampler, vec4 P, float dPdx, float dPdy, int offset )`
`float textureProjGradOffset( sampler2DShadow sampler, vec4 P, vec2 dPdx, vec2 dPdy, vec2 offset )`	

Tabela 22 Zestawienie funkcji próbkujących tekstury.

Funkcje odpytujące tekstury

W tej grupie występują dwie funkcje (patrz Tabela 23). Pierwsza to funkcja `textureSize` zwracająca rozmiary wybranego poziomu mipmapy (parametr `lod`) tekstury. Do wartości tego parametru dodawana jest zawartość zmiennej stanu `GL_TEXTURE_BASE_LEVEL` określającej bazowy poziom mipmapy, czyli standardowo podstawowy obraz tekstury. Jeżeli tak obliczony poziom mipmapy jest poza zakresem mipmap dostępnych dla danej tekstury wynik działania funkcji `textureSize` jest nieokreślony.

Druga funkcja `textureQueryLod` dostępna jest tylko w shaderze fragmentów i zwraca poziom szczegółowości mipmapy, który byłby użyty podczas standardowego procesu próbkowania tekstury. Przy obliczeniach nie jest stosowane przesunięcie (`bias`) poziomu mipmapy (`LOD`), są natomiast uwzględniane ograniczenia zakresu mipmap zawarte w zmiennych stanu `GL_TEXTURE_MIN_LOD` i `GL_TEXTURE_MAX_LOD`. Ponadto funkcja `textureQueryLod` zwraca tablicę mipmap używanych przy próbkowaniu tekstury (w odniesieniu do bazowego poziomu mipmapy, tj. `GL_TEXTURE_BASE_LEVEL`). W przypadku, gdy próbkowane są dwie mipmapy, zwracana jest zmiennoprzecinkowa liczba zawierająca się pomiędzy dwoma poziomami, gdzie część ułamkowa odpowiada obliczonemu poziomowi szczegółowości. Algorytm używany przez omawianą funkcje przedstawiony jest poniżej:

```
float ComputeAccessedLod( float computedLod )
{
    // obcinanie obliczonego LOD do ograniczenia LOD tekstury
    if( computedLod < GL_TEXTURE_MIN_LOD )
       computedLod = GL_TEXTURE_MIN_LOD;
    if( computedLod > GL_TEXTURE_MAX_LOD )
       computedLod = GL_TEXTURE_MAX_LOD;

    // obcinanie obliczonego LOD do zakresu dostępnych mipmap,
    // maxAccessibleLevel oznacza najmniejszy dostępny poziom
    // mipmapy po odjęciu bazowego poziomu mipmapy
    if( computedLod < 0.0 )
        computedLod = 0.0;
    if( computedLod > (float)maxAccessibleLevel )
        computedLod = (float)maxAccessibleLevel;

    // zwracana wartość zgodnie z filtrem minimalizującym
```

```
    if( GL_TEXTURE_MIN_FILTER == GL_LINEAR ||
        GL_TEXTURE_MIN_FILTER == GL_NEAREST )
    {
        return 0.0;
    }
    else
    if( GL_TEXTURE_MIN_FILTER == GL_NEAREST_MIPMAP_NEAREST ||
        GL_TEXTURE_MIN_FILTER == GL_LINEAR_MIPMAP_NEAREST )
    {
        return ceil( computedLod + 0.5 ) - 1.0;
    }
    else
    {
        return computedLod;
    }
}
```

Funkcja	Opis

`int textureSize( gsampler1D sampler, int lod )`
`ivec2 textureSize( gsampler2D sampler, int lod )`
`ivec3 textureSize( gsampler3D sampler, int lod )`
`ivec2 textureSize( gsamplerCube sampler, int lod )`
`int textureSize( sampler1DShadow sampler, int lod )`
`ivec2 textureSize( sampler2DShadow sampler, int lod )`
`ivec2 textureSize( samplerCubeShadow sampler, int lod )`
`ivec3 textureSize( gsamplerCubeArray sampler, int lod )`
`ivec3 textureSize( samplerCubeArrayShadow sampler, int lod )`
`ivec2 textureSize( gsampler2DRect sampler )`
`ivec2 textureSize( sampler2DRectShadow sampler )`
`ivec2 textureSize( gsampler1DArray sampler, int lod )`
`ivec3 textureSize( gsampler2DArray sampler, int lod )`
`ivec2 textureSize( sampler1DArrayShadow sampler, int lod )`
`ivec3 textureSize( sampler2DArrayShadow sampler, int lod )`
`int textureSize( gsamplerBuffer sampler )`
`ivec2 textureSize( gsampler2DMS sampler )`
`ivec3 textureSize( gsampler2DMSArray sampler )`
	Funkcje zwracają rozmiary wybranego poziomu mipmapy `lod` (jeżeli występuje) tekstury wskazanej poprzez uchwyt `sampler`. Składowe zwracanej wartości są wypełniane kolejnymi wielkościami rozmiarów tekstury: wysokością, szerokością i głębokością. W przypadku tablicy tekstur, ostatnią zwracaną wartością jest ilość warstw (tekstur) w tablicy.

`vec2 textureQueryLod( gsampler1D sampler, float P )`
`vec2 textureQueryLod( gsampler2D sampler, vec2 P )`
`vec2 textureQueryLod( gsampler3D sampler, vec3 P )`
`vec2 textureQueryLod( gsamplerCube sampler, vec3 P )`
`vec2 textureQueryLod( gsampler1DArray sampler, float P )`
	Funkcje zwracają poziom mipmapy, która byłaby użyta podczas próbkowania tekstury (składowa `x` wektora wynikowego) oraz poziom szczegółowości `LOD` w odniesieniu do bazowego poziomu mipmapy (składowa `y` wektora wynikowego). W przypadku niekompletności tekstury zwracana wartość jest

`vec2 textureQueryLod( gsampler2DArray sampler, vec2 P )`
`vec2 textureQueryLod( gsamplerCubeArray sampler, vec3 P )`
`vec2 textureQueryLod( sampler1DShadow sampler, float P )`
`vec2 textureQueryLod( sampler2DShadow sampler, vec2 P )`
`vec2 textureQueryLod( samplerCubeShadow sampler, vec3 P )`
`vec2 textureQueryLod( sampler1DArrayShadow sampler, float P )`
`vec2 textureQueryLod( sampler2DArrayShadow sampler, vec2 P )`
`vec2 textureQueryLod( samplerCubeArrayShadow sampler, vec3 P )`
	niezdefiniowana.

Tabela 23 Zestawienie funkcji odpytujących tekstury.

Funkcje wielopróbkujące tekstury

Funkcje wielopróbkujące tekstury, wymienione w Tabela 24, pobierają jednocześnie wybraną składową z bloku 2×2 sąsiednich tekseli z bazowego poziomu obrazu tekstury i zwracają wynik w czteroelementowym wektorze. W trakcie tej operacji ignorowane są ustawienia filtracji tekstury, a próbkowane składowe sąsiednich tekseli i_0j_1, i_1j_1, i_1j_0 i i_0j_0 układane są w następującej kolejności w wynikowym wektorze: $(i_0j_1, i_1j_1, i_1j_0, i_0j_0)$. Wybór tekseli odpowiada algorytmom standardowej filtracji liniowej tekstury (filtr `GL_LINEAR`). W przypadku, gdy wielopróbkowaniu poddawane są tekstury głębi, każdy z czterech próbkowanych tekseli poddawany jest procesowi porównywania z wartością referencyjną głębokości D_{ref}.

Wynik działania funkcji wielopróbkujących tekstury głębi jest nieokreślony, gdy wskazana tekstura nie jest teksturą głębi, lub porównywania głębokości jest wyłączone. Taka sama sytuacja ma miejsce, gdy wielopróbkujemy pozostałe rodzaje tekstur, a wskazana tekstura jest teksturą głębi.

Funkcja	Opis

`gvec4 textureGather( gsampler2D sampler, vec2 P [, int comp])`
`gvec4 textureGather( gsampler2DArray sampler, vec3 P [, int comp])`
`gvec4 textureGather( gsamplerCube sampler, vec3 P [, int comp] )`
`gvec4 textureGather( gsamplerCubeArray sampler, vec4 P [, int comp])`
`gvec4 textureGather( gsampler2DRect sampler, vec2 P [, int comp] )`
`vec4 textureGather( sampler2DShadow sampler, vec2 P, float refZ )`
`vec4 textureGather( sampler2DArrayShadow sampler, vec3 P, float refZ )`
`vec4 textureGather( samplerCubeShadow sampler, vec3 P, float refZ )`
`vec4 textureGather( samplerCubeArrayShadow sampler, vec4 P, float refZ )`
`vec4 textureGather( sampler2DRectShadow sampler, vec2 P, float refZ )`
	Funkcja próbkuje składową tekstury wskazaną w parametrze `comp` z bloku 2×2 sąsiednich tekseli i_0j_1, i_1j_1, i_1j_0 i i_0j_0 o współrzędnych `P` z bazowego poziomu obrazu tekstury i zwraca je w postaci czteroelementowego wektora w układzie $(i_0j_1, i_1j_1, i_1j_0, i_0j_0)$. Parametr `comp` określa numer próbkowanych składowych tekseli tekstury może przyjmować wartości całkowite od 0 do 3 (odpowiednio składowe: `x`, `y`, `z` i `w`). Składowe tekseli pobierane są już po ewentualnej operacji przestawiania i podstawiania składowych. Jeżeli parametr `comp` nie jest określony, domyślnie przyjmowana jest wartość 0.

`gvec4 textureGatherOffset( gsampler2D sampler, vec2 P, ivec2 offset`	Wielopróbkowanie tekstury tak jak w funkcji `textureGather`

[, int comp]) gvec4 textureGatherOffset(gsampler2DArray sampler, vec3 P, ivec2 offset [, int comp]) gvec4 textureGatherOffset(gsampler2DRect sampler, vec2 P, ivec2 offset [, int comp]) vec4 textureGatherOffset(sampler2DShadow sampler, vec2 P, float refZ, ivec2 offset) vec4 textureGatherOffset(sampler2DArrayShadow sampler, vec3 P, float refZ, ivec2 offset) vec4 textureGatherOffset(sampler2DRectShadow sampler, vec2 P, float refZ, ivec2 offset)	z przesunięciem współrzędnych `P` o `offset` analogicznie jak w funkcji `textureOffset`, przy czym `offset` może być zmienną. Zachowywane są zależne od implementacji minimalne i maksymalne wartości przesunięcia zawarte w zmiennych stanu odpowiednio: `GL_MIN_PROGRAM_TEXTURE_GATHER_OFFSET` i `GL_MAX_PROGRAM_TEXTURE_GATHER_OFFSET`.
gvec4 textureGatherOffsets(gsampler2D sampler, vec2 P, ivec2 offsets[4] [, int comp]) gvec4 textureGatherOffsets(gsampler2DArray sampler, vec3 P, ivec2 offsets[4] [, int comp]) gvec4 textureGatherOffsets(gsampler2DRect sampler, vec3 P, ivec2 offsets[4] [, int comp]) vec4 textureGatherOffsets(sampler2DShadow sampler, vec2 P, float refZ, ivec2 offsets[4]) vec4 textureGatherOffsets(sampler2DArrayShadow sampler, vec3 P, float refZ, ivec2 offsets[4]) vec4 textureGatherOffsets(sampler2DRectShadow sampler, vec2 P, float refZ, ivec2 offsets[4])	Wielopróbkowanie tekstury tak jak w funkcji `textureGatherOffset` przy czym, przesunięcie współrzędnych `P` jest określone dla każdego próbkowanego teksela bazowej tekstury. Dane wektorów przesunięć `offsets` muszą być określone w postaci stałych wyrażeń całkowitych.

Tabela 24 Zestawienie funkcji wielopróbkujących tekstury.

Funkcje licznika atomowego

Funkcje licznika atomowego, wymienione w Tabela 25, działają w sposób atomowy, tj. niepodzielny w stosunku do innych operacji na tym samym liczniku atomowym w tej lub innej instancji shadera. Omawiane funkcje nie dają natomiast gwarancji, że inne metody dostępu do danych licznika także będą wykonywane w sposób niepodzielny. W przypadku konieczności zapewnienia takiego wymogu trzeba użyć innych metod synchronizacji, np. barier obsługiwanych przez funkcję `memoryBarrier`, którą opisujemy pod koniec niniejszego odcinka kursu OpenGL.

Liczniki atomowe działają na 32-bitowych liczbach całkowitych bez znaku. Opisane w Tabela 25 operacje inkrementacji i dekrementacji są zawijane do przedziału wartości $\langle 0; 2^{32} - 1 \rangle$.

Funkcja	Opis
`uint atomicCounterIncrement( atomic_uint c )`	Atomowa (niepodzielna) operacja inkrementacji licznika atomowego `c`. Funkcja zwraca wartość licznika przed inkrementacją.

`uint atomicCounterDecrement( atomic_uint c )`	Atomowa (niepodzielna) operacja dekrementacji licznika atomowego `c`. Funkcja zwraca wartość licznika po dekrementacji.
`uint atomicCounter( atomic_uint c )`	Funkcja zwraca wartość wybranego licznika atomowego.

Tabela 25 Zestawienie funkcji licznika atomowego.

Funkcje obrazu tekstur

Funkcje obrazu tekstur, wymienione w Tabela 26, mogą być używane od odczytu i zapisu indywidualnych tekseli tekstur. Każda zmienna tego typu określa referencję do danych tekstury na określonym poziomie mipmapy. Funkcje obrazu tekstur określają położenie wybranego teksela na podstawie jedno- dwu- lub trójwymiarowych współrzędnych zawartych w argumencie `P` funkcji. W przypadku zmiennych korzystających z tekstur z wielopróbkowaniem, np. typu `image2DMS`, `image2DMSArray`, itp. każdy teksel posiada wiele próbek i omawiane funkcje pozwalają na dostęp do wybranej próbki za pośrednictwem parametru `sample`.

Zapis i odczyt obejmuje typy zmiennoprzecinkowe pojedynczej precyzji, całkowite ze znakiem oraz całkowite bez znaku. Zastosowane w Tabela 26 oznaczenie typu rozpoczynające się na "`gimage`" oznacza odpowiednio "`image`", "`iimage`" lub "`uimage`", na podobnej zasadzie jak używane już wcześniej w opisach funkcji "`gvec`" i "`gsampler`".

Dodajmy jeszcze, że większość funkcji przedstawionych w Tabela 26 to operacje atomowe (niepodzielne) z gwarancją, że w czasie ich wykonania żadna inna operacja atomowa nie zmodyfikuje przetwarzanych danych teksela tekstury. Operacje te są jednak ograniczone tylko do wybranych typów danych. Parametr `image` może określać dostęp do zmiennej całkowitej ze znakiem (typy z nazwą rozpoczynająca się od "`iimage`") i identyfikatorem kwalifikatora formatu `r32i` w połączeniu z parametrem `data` typu `int`, lub określać dostęp do zmiennej całkowitej bez znaku (typy z nazwą rozpoczynająca się od "`uimage`") i identyfikatorem kwalifikatora formatu `r32ui` w połączeniu z parametrem `data` typu `uint`.

Funkcja	Opis

`gvec4 imageLoad( readonly gimage1D image, int P )`
`gvec4 imageLoad( readonly gimage2D image, ivec2 P )`
`gvec4 imageLoad( readonly gimage3D image, ivec3 P )`
`gvec4 imageLoad( readonly gimage2DRect image, ivec2 P )`
`gvec4 imageLoad( readonly gimageCube image, ivec3 P )`
`gvec4 imageLoad( readonly gimageBuffer image, int P )`
`gvec4 imageLoad( readonly gimage1DArray image, ivec2 P )`
`gvec4 imageLoad( readonly gimage2DArray image, ivec3 P )`
`gvec4 imageLoad( readonly gimageCubeArray image, ivec3 P )`
`gvec4 imageLoad( readonly gimage2DMS image, ivec2 P, int sample )`
`gvec4 imageLoad( readonly gimage2DMSArray image, ivec3 P, int sample )`
	Pobranie teksela o współrzędnych `P` z obrazu tekstury `image`. W przypadku obrazu tekstury z wielopróbkowaniem pobierana jest próbka o numerze `sample`. W przypadku, gdy `image`, `P` oraz `sample` określają poprawny teksel, bity reprezentujące wybrany teksel w pamięci w zależności od rodzaju danych obrazu tekstury są konwertowane do wektora typu `vec4`, `ivec4` lub `uvec4`.

`void imageStore( writeonly gimage1D image, int P, gvec4 data )`
`void imageStore( writeonly gimage2D image, ivec2 P, gvec4 data )`
`void imageStore( writeonly gimage3D image,`
	Zapis danych w parametrze `data` do teksela o współrzędnych `P` z obrazu tekstury `image`. W przypadku obrazu tekstury z

`ivec3 P, gvec4 data )`
`void imageStore( writeonly gimage2DRect image,`
`ivec2 P, gvec4 data )`
`void imageStore( writeonly gimageCube image,`
`ivec3 P, gvec4 data )`
`void imageStore( writeonly gimageBuffer image,`
`int P, gvec4 data )`
`void imageStore( writeonly gimage1DArray image,`
`ivec2 P, gvec4 data )`
`void imageStore( writeonly gimage2DArray image,`
`ivec3 P, gvec4 data )`
`void imageStore( writeonly gimageCubeArray image,`
`ivec3 P, gvec4 data )`
`void imageStore( writeonly gimage2DMS image,`
`ivec2 P, int sample,`
`gvec4 data )`
`void imageStore( writeonly gimage2DMSArray image,`
`ivec3 P, int sample,`
`gvec4 data )`
	wielopróbkowaniem zapisywana jest próbka o numerze `sample`. W przypadku, gdy `image`, `P` oraz `sample` określają poprawny teksel, bity reprezentujące dane w `data` są konwertowane do formatu odpowiedniego do rodzaju danych obrazu tekstury.

`uint imageAtomicAdd( gimage1D image, int P,`
`uint data )`
`uint imageAtomicAdd( gimage2D image, ivec2 P,`
`uint data )`
`uint imageAtomicAdd( gimage3D image, ivec3 P,`
`uint data )`
`uint imageAtomicAdd( gimage2DRect image, ivec2 P,`
`uint data )`
`uint imageAtomicAdd( gimageCube image, ivec3 P,`
`uint data )`
`uint imageAtomicAdd( gimageBuffer image, int P,`
`uint data )`
`uint imageAtomicAdd( gimage1DArray image,`
`ivec2 P, uint data )`
`uint imageAtomicAdd( gimage2DArray image,`
`ivec3 P, uint data )`
`uint imageAtomicAdd( gimageCubeArray image,`
`ivec3 P, uint data )`
`uint imageAtomicAdd( gimage2DMS image, ivec2 P,`
`int sample, uint data )`
`uint imageAtomicAdd( gimage2DMSArray image,`
`ivec3 P, int sample,`
`uint data )`
 	
`int imageAtomicAdd( gimage1D image, int P,`
`int data )`
`int imageAtomicAdd( gimage2D image, ivec2 P,`
`int data )`
`int imageAtomicAdd( gimage3D image, ivec3 P,`
`int data )`
`int imageAtomicAdd( gimage2DRect image, ivec2 P,`
`int data )`
`int imageAtomicAdd( gimageCube image, ivec3 P,`
`int data )`
`int imageAtomicAdd( gimageBuffer image, int P,`
`int data )`
`int imageAtomicAdd( gimage1DArray image, ivec2 P,`
`int data )`
`int imageAtomicAdd( gimage2DArray image, ivec3 P,`
`int data )`
`int imageAtomicAdd( gimageCubeArray image,`
`ivec3 P, int data )`
	Atomowe (niepodzielne) obliczenie sumy wartości wskazanej w parametrze `data` oraz teksela o współrzędnych `P` z obrazu tekstury `image`. W przypadku obrazu tekstury z wielopróbkowaniem pobierana jest próbka o numerze `sample`.

`int imageAtomicAdd( gimage2DMS image, ivec2 P,`
`  int sample, int data )`
`int imageAtomicAdd( gimage2DMSArray image,`
`  ivec3 P, int sample,`
`  int data )`	

`uint imageAtomicMin( gimage1D image, int P,`
`  uint data )`
`uint imageAtomicMin( gimage2D image, ivec2 P,`
`  uint data )`
`uint imageAtomicMin( gimage3D image, ivec3 P,`
`  uint data )`
`uint imageAtomicMin( gimage2DRect image, ivec2 P,`
`  uint data )`
`uint imageAtomicMin( gimageCube image, ivec3 P,`
`  uint data )`
`uint imageAtomicMin( gimageBuffer image, int P,`
`  uint data )`
`uint imageAtomicMin( gimage1DArray image,`
`  ivec2 P, uint data )`
`uint imageAtomicMin( gimage2DArray image,`
`  ivec3 P, uint data )`
`uint imageAtomicMin( gimageCubeArray image,`
`  ivec3 P, uint data )`
`uint imageAtomicMin( gimage2DMS image, ivec2 P,`
`  int sample, uint data )`
`uint imageAtomicMin( gimage2DMSArray image,`
`  ivec3 P, int sample,`
`  uint data )`
 	
`int imageAtomicMin( gimage1D image, int P,`
`  int data )`
`int imageAtomicMin( gimage2D image, ivec2 P,`
`  int data )`
`int imageAtomicMin( gimage3D image, ivec3 P,`
`  int data )`
`int imageAtomicMin( gimage2DRect image, ivec2 P,`
`  int data )`
`int imageAtomicMin( gimageCube image, ivec3 P,`
`  int data )`
`int imageAtomicMin( gimageBuffer image, int P,`
`  int data )`
`int imageAtomicMin( gimage1DArray image, ivec2 P,`
`  int data )`
`int imageAtomicMin( gimage2DArray image, ivec3 P,`
`  int data )`
`int imageAtomicMin( gimageCubeArray image,`
`  ivec3 P, int data )`
`int imageAtomicMin( gimage2DMS image, ivec2 P,`
`  int sample, int data )`
`int imageAtomicMin( gimage2DMSArray image,`
`  ivec3 P, int sample,`
`  int data )`
	Atomowe (niepodzielne) obliczenie minimum z wartości wskazanej w parametrze `data` oraz teksela o współrzędnych `P` z obrazu tekstury `image`. W przypadku obrazu tekstury z wielopróbkowaniem pobierana jest próbka o numerze `sample`.

`uint imageAtomicMax( gimage1D image, int P,`
`  uint data )`
`uint imageAtomicMax( gimage2D image, ivec2 P,`
`  uint data )`
`uint imageAtomicMax( gimage3D image, ivec3 P,`
`  uint data )`
`uint imageAtomicMax( gimage2DRect image, ivec2 P,`
`  uint data )`
`uint imageAtomicMax( gimageCube image, ivec3 P,`
	Atomowe (niepodzielne) obliczenie maksimum z wartości wskazanej w parametrze `data` oraz teksela o współrzędnych `P` z obrazu tekstury `image`. W przypadku obrazu tekstury z wielopróbkowaniem pobierana jest próbka o numerze `sample`.

`uint data )`
`uint imageAtomicMax( gimageBuffer image, int P,`
`uint data )`
`uint imageAtomicMax( gimage1DArray image,`
`ivec2 P, uint data )`
`uint imageAtomicMax( gimage2DArray image,`
`ivec3 P, uint data )`
`uint imageAtomicMax( gimageCubeArray image,`
`ivec3 P, uint data )`
`uint imageAtomicMax( gimage2DMS image, ivec2 P,`
`int sample, uint data )`
`uint imageAtomicMax( gimage2DMSArray image,`
`ivec3 P, int sample,`
`uint data )`
 	
`int imageAtomicMax( gimage1D image, int P,`
`int data )`
`int imageAtomicMax( gimage2D image, ivec2 P,`
`int data )`
`int imageAtomicMax( gimage3D image, ivec3 P,`
`int data )`
`int imageAtomicMax( gimage2DRect image, ivec2 P,`
`int data )`
`int imageAtomicMax( gimageCube image, ivec3 P,`
`int data )`
`int imageAtomicMax( gimageBuffer image, int P,`
`int data )`
`int imageAtomicMax( gimage1DArray image, ivec2 P,`
`int data )`
`int imageAtomicMax( gimage2DArray image, ivec3 P,`
`int data )`
`int imageAtomicMax( gimageCubeArray image,`
`ivec3 P, int data )`
`int imageAtomicMax( gimage2DMS image, ivec2 P,`
`int sample, int data )`
`int imageAtomicMax( gimage2DMSArray image,`
`ivec3 P, int sample,`
`int data )`	

`uint imageAtomicAnd( gimage1D image, int P,`
`uint data )`
`uint imageAtomicAnd( gimage2D image, ivec2 P,`
`uint data )`
`uint imageAtomicAnd( gimage3D image, ivec3 P,`
`uint data )`
`uint imageAtomicAnd( gimage2DRect image, ivec2 P,`
`uint data )`
`uint imageAtomicAnd( gimageCube image, ivec3 P,`
`uint data )`
`uint imageAtomicAnd( gimageBuffer image, int P,`
`uint data )`
`uint imageAtomicAnd( gimage1DArray image,`
`ivec2 P, uint data )`
`uint imageAtomicAnd( gimage2DArray image,`
`ivec3 P, uint data )`
`uint imageAtomicAnd( gimageCubeArray image,`
`ivec3 P, uint data )`
`uint imageAtomicAnd( gimage2DMS image, ivec2 P,`
`int sample, uint data )`
`uint imageAtomicAnd( gimage2DMSArray image,`
`ivec3 P, int sample,`
`uint data )`
	Atomowe (niepodzielne) obliczenie bitowej operacji AND z wartości wskazanej w parametrze `data` oraz teksela o współrzędnych `P` z obrazu tekstury `image`. W przypadku obrazu tekstury z wielopróbkowaniem pobierana jest próbka o numerze `sample`.

`int imageAtomicAnd( gimage1D image, int P,`
`  int data )`
`int imageAtomicAnd( gimage2D image, ivec2 P,`
`  int data )`
`int imageAtomicAnd( gimage3D image, ivec3 P,`
`  int data )`
`int imageAtomicAnd( gimage2DRect image, ivec2 P,`
`  int data )`
`int imageAtomicAnd( gimageCube image, ivec3 P,`
`  int data )`
`int imageAtomicAnd( gimageBuffer image, int P,`
`  int data )`
`int imageAtomicAnd( gimage1DArray image, ivec2 P,`
`  int data )`
`int imageAtomicAnd( gimage2DArray image, ivec3 P,`
`  int data )`
`int imageAtomicAnd( gimageCubeArray image,`
`  ivec3 P, int data )`
`int imageAtomicAnd( gimage2DMS image, ivec2 P,`
`  int sample, int data )`
`int imageAtomicAnd( gimage2DMSArray image,`
`  ivec3 P, int sample,`
`  int data )`	

`uint imageAtomicOr( gimage1D image, int P,`
`  uint data )`
`uint imageAtomicOr( gimage2D image, ivec2 P,`
`  uint data )`
`uint imageAtomicOr( gimage3D image, ivec3 P,`
`  uint data )`
`uint imageAtomicOr( gimage2DRect image, ivec2 P,`
`  uint data )`
`uint imageAtomicOr( gimageCube image, ivec3 P,`
`  uint data )`
`uint imageAtomicOr( gimageBuffer image, int P,`
`  uint data )`
`uint imageAtomicOr( gimage1DArray image, ivec2 P,`
`  uint data )`
`uint imageAtomicOr( gimage2DArray image, ivec3 P,`
`  uint data )`
`uint imageAtomicOr( gimageCubeArray image,`
`  ivec3 P, uint data )`
`uint imageAtomicOr( gimage2DMS image, ivec2 P,`
`  int sample, uint data )`
`uint imageAtomicOr( gimage2DMSArray image,`
`  ivec3 P, int sample,`
`  uint data )`
 	
`int imageAtomicOr( gimage1D image, int P,`
`  int data )`
`int imageAtomicOr( gimage2D image, ivec2 P,`
`  int data )`
`int imageAtomicOr( gimage3D image, ivec3 P,`
`  int data )`
`int imageAtomicOr( gimage2DRect image, ivec2 P,`
`  int data )`
`int imageAtomicOr( gimageCube image, ivec3 P,`
`  int data )`
`int imageAtomicOr( gimageBuffer image, int P,`
`  int data )`
`int imageAtomicOr( gimage1DArray image, ivec2 P,`
	Atomowe (niepodzielne) obliczenie bitowej operacji OR z wartości wskazanej w parametrze `data` oraz teksela o współrzędnych `P` z obrazu tekstury `image`. W przypadku obrazu tekstury z wielopróbkowaniem pobierana jest próbka o numerze `sample`.

`int data )`
`int imageAtomicOr( gimage2DArray image, ivec3 P,`
`int data )`
`int imageAtomicOr( gimageCubeArray image,`
`ivec3 P, int data )`
`int imageAtomicOr( gimage2DMS image, ivec2 P,`
`int sample, int data )`
`int imageAtomicOr( gimage2DMSArray image,`
`ivec3 P, int sample,`
`int data )`	

`uint imageAtomicXor( gimage1D image, int P,`
`uint data )`
`uint imageAtomicXor( gimage2D image, ivec2 P,`
`uint data )`
`uint imageAtomicXor( gimage3D image, ivec3 P,`
`uint data )`
`uint imageAtomicXor( gimage2DRect image, ivec2 P,`
`uint data )`
`uint imageAtomicXor( gimageCube image, ivec3 P,`
`uint data )`
`uint imageAtomicXor( gimageBuffer image, int P,`
`uint data )`
`uint imageAtomicXor( gimage1DArray image,`
`ivec2 P, uint data )`
`uint imageAtomicXor( gimage2DArray image,`
`ivec3 P, uint data )`
`uint imageAtomicXor( gimageCubeArray image,`
`ivec3 P, uint data )`
`uint imageAtomicXor( gimage2DMS image, ivec2 P,`
`int sample, uint data )`
`uint imageAtomicXor( gimage2DMSArray image,`
`ivec3 P, int sample,`
`uint data )`
 	
`int imageAtomicXor( gimage1D image, int P,`
`int data )`
`int imageAtomicXor( gimage2D image, ivec2 P,`
`int data )`
`int imageAtomicXor( gimage3D image, ivec3 P,`
`int data )`
`int imageAtomicXor( gimage2DRect image, ivec2 P,`
`int data )`
`int imageAtomicXor( gimageCube image, ivec3 P,`
`int data )`
`int imageAtomicXor( gimageBuffer image, int P,`
`int data )`
`int imageAtomicXor( gimage1DArray image, ivec2 P,`
`int data )`
`int imageAtomicXor( gimage2DArray image, ivec3 P,`
`int data )`
`int imageAtomicXor( gimageCubeArray image,`
`ivec3 P, int data )`
`int imageAtomicXor( gimage2DMS image, ivec2 P,`
`int sample, int data )`
`int imageAtomicXor( gimage2DMSArray image,`
`ivec3 P, int sample,`
`int data )`
	Atomowe (niepodzielne) obliczenie bitowej operacji XOR (różnica symetryczna) z wartości wskazanej w parametrze `data` oraz teksela o współrzędnych `P` z obrazu tekstury `image`. W przypadku obrazu tekstury z wielopróbkowaniem pobierana jest próbka o numerze `sample`.

`uint imageAtomicExchange( gimage1D image, int P,`
`uint data )`
`uint imageAtomicExchange( gimage2D image,`
`ivec2 P, uint data )`
	Atomowe (niepodzielne) zapisanie wartości wskazanej w parametrze `data` do teksela o współrzędnych `P` z obrazu tekstury `image`.

`uint imageAtomicExchange( gimage3D image, ivec3 P, uint data )`
`uint imageAtomicExchange( gimage2DRect image, ivec2 P, uint data )`
`uint imageAtomicExchange( gimageCube image, ivec3 P, uint data )`
`uint imageAtomicExchange( gimageBuffer image, int P, uint data )`
`uint imageAtomicExchange( gimage1DArray image, ivec2 P, uint data )`
`uint imageAtomicExchange( gimage2DArray image, ivec3 P, uint data )`
`uint imageAtomicExchange( gimageCubeArray image, ivec3 P, uint data )`
`uint imageAtomicExchange( gimage2DMS image, ivec2 P, int sample, uint data )`
`uint imageAtomicExchange( gimage2DMSArray image, ivec3 P, int sample, uint data )`
 	
`int imageAtomicExchange( gimage1D image, int P, int data )`
`int imageAtomicExchange( gimage2D image, ivec2 P, int data )`
`int imageAtomicExchange( gimage3D image, ivec3 P, int data )`
`int imageAtomicExchange( gimage2DRect image, ivec2 P, int data )`
`int imageAtomicExchange( gimageCube image, ivec3 P, int data )`
`int imageAtomicExchange( gimageBuffer image, int P, int data )`
`int imageAtomicExchange( gimage1DArray image, ivec2 P, int data )`
`int imageAtomicExchange( gimage2DArray image, ivec3 P, int data )`
`int imageAtomicExchange( gimageCubeArray image, ivec3 P, int data )`
`int imageAtomicExchange( gimage2DMS image, ivec2 P, int sample, int data )`
`int imageAtomicExchange( gimage2DMSArray image, ivec3 P, int sample, int data )`
	Jednocześnie funkcja zwraca pierwotną wartość tego teksela. W przypadku obrazu tekstury z wielopróbkowaniem zapisywana i pobierana jest próbka o numerze `sample`.

`uint imageAtomicCompSwap( gimage1D image, int P, uint data )`
`uint imageAtomicCompSwap( gimage2D image, ivec2 P, uint data )`
`uint imageAtomicCompSwap( gimage3D image, ivec3 P, uint data )`
`uint imageAtomicCompSwap( gimage2DRect image, ivec2 P, uint data )`
`uint imageAtomicCompSwap( gimageCube image, ivec3 P, uint data )`
`uint imageAtomicCompSwap( gimageBuffer image, int P, uint data )`
`uint imageAtomicCompSwap( gimage1DArray image, ivec2 P, uint data )`
`uint imageAtomicCompSwap( gimage2DArray image, ivec3 P, uint data )`
	Funkcja atomowo (niepodzielne) porównuje wartość zawartą w parametrze `data` z wartością teksela o współrzędnych `P` z obrazu tekstury `image`. Jeżeli wartości te są równe wartość `data` jest zapisywana w wybranym tekselu, w przeciwnym wypadku operacja jest przerywana. Funkcja zwraca nową wartość teksela. W przypadku obrazu tekstury z wielopróbkowaniem pobierana i zapisywana jest próbka o numerze `sample`.

uint imageAtomicCompSwap(gimageCubeArray image, ivec3 P, uint data) uint imageAtomicCompSwap(gimage2DMS image, ivec2 P, int sample, uint data) uint imageAtomicCompSwap(gimage2DMSArray image, ivec3 P, int sample, uint data) int imageAtomicCompSwap(gimage1D image, int P, int data) int imageAtomicCompSwap(gimage2D image, ivec2 P, int data) int imageAtomicCompSwap(gimage3D image, ivec3 P, int data) int imageAtomicCompSwap(gimage2DRect image, ivec2 P, int data) int imageAtomicCompSwap(gimageCube image, ivec3 P, int data) int imageAtomicCompSwap(gimageBuffer image, int P, int data) int imageAtomicCompSwap(gimage1DArray image, ivec2 P, int data) int imageAtomicCompSwap(gimage2DArray image, ivec3 P, int data) int imageAtomicCompSwap(gimageCubeArray image, ivec3 P, int data) int imageAtomicCompSwap(gimage2DMS image, ivec2 P, int sample, int data) int imageAtomicCompSwap(gimage2DMSArray image, ivec3 P, int sample, int data)	

Tabela 26. Zestawienie funkcji obrazu tekstur.

Funkcje różniczkowe

Funkcje różniczkowe, wymienione w Tabela 27, dostępne są tylko dla shaderów fragmentów. W wielu funkcjach działających w shaderach fragmentów wewnętrznie używa się pochodnych. Ponieważ numeryczne obliczanie pochodnych jest kosztowne obliczeniowo i/lub numerycznie niestabilne, implementacja biblioteki OpenGL, może aproksymować prawdziwe obliczenia pochodnych przy użyciu szybkich, lecz niekoniecznie dokładnych metod ich obliczania. Pochodne są nieokreślone w niejednorodnej kontroli przepływu w shaderze fragmentów.

Typowo do obliczeń pochodnych w GLSL stosuje się iloraz różnicowy występujący w dwóch wersjach określanych mianem ilorazu różnicowego przedniego i wstecznego (ang. *forward/backward differencing*):

$$dFdx(x) \approx \frac{F(x+dx)-F(x)}{dx}$$
$$dFdx(x) \approx \frac{F(x)-F(x-dx)}{dx}$$

Przyrost wartości argumentu różniczkowanej funkcji w przypadku braku wielopróbkowania jest ograniczony do $dx \leq 1{,}0$. Gdy wielopróbkowanie jest włączone przyrost spełnia ograniczenie $dx < 2{,}0$. Analogicznie obliczane są $dFdy$ - pochodne po zmiennej y.

Implementacja OpenGL może użyć innej metody obliczania pochodnych. Musi ona spełniać szereg poniższych warunków:

- dopuszczalne jest użycie liniowej aproksymacji, przy czym metoda ta dla pochodnych drugiego stopnia $dFdx\big(dFdx(x)\big)$ i wyższych rzędów jest nieokreślona,
- algorytm może zakładać ciągłość funkcji, stąd pochodne przy niejednorodnej kontroli przepływu w shaderze fragmentów są nieokreślone,
- algorytm może dawać różne wyniki dla fragmentów, korzystając ze współrzędnych okienkowych, a nie ekranowych; w przypadku obliczania pochodnych nie muszą być zachowane ogólne warunki powtarzalności operacji OpenGL,
- opcjonalnie funkcja może być obliczana wewnątrz wnętrza prymitywu (przy użyciu interpolacji, nie zaś ekstrapolacji),
- opcjonalnie funkcja obliczająca $dFdx(x)$ zakłada stałość y, analogicznie funkcja obliczająca $dFdy(y)$ zakłada stałość x; jednak pochodne mieszane wyższego rzędu, jak $dFdx\big(dFdy(y)\big)$ i $dFdy\big(dFdx(x)\big)$ są nieokreślone,
- opcjonalnie pochodna stałego argumentu jest równa 0.

Ponadto niektóre implementacje OpenGL mogą udostępniać regulację dokładności obliczania pochodnych funkcji przy pomocy wskazówki renderingu określonej stałą `GL_FRAGMENT_SHADER_DERIVATIVE_HINT` (funkcja `glHint`). Tę i pozostałe wskazówki renderingu opiszemy w następnych odcinkach kursu OpenGL.

Funkcja	Opis
`genType dFdx( genType p )`	Pochodna po x przy użyciu lokalnego różniczkowania dla argumentu wejściowego `p`.
`genType dFdy( genType p )`	Pochodna po y przy użyciu lokalnego różniczkowania dla argumentu wejściowego `p`.
`genType fwidth( genType p )`	Suma wartości bezwzględnych pochodnych po x i y przy użyciu lokalnego różniczkowania dla argumentu wejściowego `p`, np. $\mathrm{abs}(\mathrm{dFdx}(p)) + \mathrm{abs}(\mathrm{dFdy}(p))$.

Tabela 27 Zestawienie funkcji różniczkowych.

Funkcje interpolacyjne

Funkcje interpolacyjne, przedstawione w Tabela 28, są przeznaczone do obliczania interpolowanej wartości zmiennych wejściowych shadera fragmentów dla określonej lokalizacji (x, y), w szczególności różnej od położenia domyślnego, używanego do obliczania wartości zmiennych wejściowych. Dla wszystkich niżej wymienionych funkcji parametr `interpolant` musi być zmienną wejściową lub elementem zmiennej wejściowej zadeklarowanej jako tablica. Jeżeli zmienna przekazana jako parametr `interpolant` została zadeklarowana z kwalifikatorem interpolacji `flat` lub `centroid`, kwalifikator ten nie ma znaczenia dla interpolowanej wartości. Jeżeli natomiast użyto do deklaracji tej zmiennej kwalifikatora `noperspective`, interpolowana wartość będzie obliczona bez użycia korekcji perspektywy (interpolacja liniowa).

Funkcja	Opis

`float interpolateAtCentroid( float interpolant )`
`vec2 interpolateAtCentroid( vec2 interpolant )`
`vec3 interpolateAtCentroid( vec3 interpolant )`
`vec4 interpolateAtCentroid( vec4 interpolant )`
	Funkcja zwraca interpolowaną wartości zmiennej wejściowej `interpolant` na podstawie centroidu - próbki zlokalizowanej w prymitywie i jednocześnie leżącej w obrębie piksela. Zwracana wartość odpowiada wartości zmiennej wejściowej shadera fragmentów zadeklarowanej z kwalifikatorem interpolacji `centroid`.

`float interpolateAtSample( float interpolant,`
`int sample )`
`vec2 interpolateAtSample( vec2 interpolant,`
	Funkcja zwraca interpolowaną wartości zmiennej wejściowej

`int sample )`
`vec3 interpolateAtSample( vec3 interpolant,`
`int sample )`
`vec4 interpolateAtSample( vec4 interpolant,`
`int sample )`
	`interpolant` zgodnie z położeniem próbki o numerze `sample`. Jeżeli bufory wielopróbkowania nie są dostępne, zwracana jest wartość obliczona dla środka piksela. Jeżeli próbka nr `sample` nie istnieje, położenie próbki używane do interpolacji zmiennej wejściowej `interpolant` nie jest zdefiniowane.

`float interpolateAtOffset( float interpolant,`
`vec2 offset )`
`vec2 interpolateAtOffset( vec2 interpolant,`
`vec2 offset )`
`vec3 interpolateAtOffset( vec3 interpolant,`
`vec2 offset )`
`vec4 interpolateAtOffset( vec4 interpolant,`
`vec2 offset )`
	Funkcja zwraca interpolowaną wartości zmiennej wejściowej `interpolant` próbkowanej dla środka piksela z przesunięciem określonym w wektorze `offset`. Wartość parametru `offset` równa $(0,0)$ oznacza środek piksela. Zakres i dopuszczalne stopniowanie wartości składowych wektora `offset` zależą od implementacji.

Tabela 28 Zestawienie funkcji interpolacyjnych.

Funkcje generujące szum

Wymienione w Tabela 29 funkcje generujące szum dostępne są w shaderach wierzchołków, geometrii i fragmentów. Te funkcje stochastyczne zwracają wartości pseudolosowe o następującej charakterystyce:

- zwracane wartości zawierają się w przedziale $\langle -1{,}0; 1{,}0\rangle$ z pokryciem co najmniej w przedziale $\langle -0{,}6; 0{,}6\rangle$,
- średnia zwracanych wartości wynosi 0,0,
- są powtarzalne, czyli dla określonej danej wejściowej zwracają taką samą wartość,
- są statystycznie niezmienne i 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 1,0,
- są klasy C^1, czyli pierwsza pochodna jest funkcją ciągłą.

Funkcja	Opis
`float noise1( genType x )`	Generowanie jednowymiarowego szumu na podstawie wartości bazowej `x`.
`vec2 noise2( genType x )`	Generowanie dwuwymiarowego szumu na podstawie wartości bazowej `x`.
`vec3 noise3( genType x )`	Generowanie trójwymiarowego szumu na podstawie wartości bazowej `x`.
`vec4 noise4( genType x )`	Generowanie czterowymiarowego szumu na podstawie wartości bazowej `x`.

Tabela 29 Zestawienie funkcji generujących szum.

Funkcje shadera geometrii

Funkcje shadera geometrii, przedstawione w Tabela 30, dostępne są wyłącznie w programach cieniowania geometrii i służą do sterowania procesem generowania wierzchołków oraz prymitywów. Wspomniana w opisach funkcji technika wielu strumieni wyjściowych wierzchołków jest dostępna tylko w przypadku, gdy wyjściowy typ prymitywu jest zadeklarowany jako `points`. Ponadto, błędem zakończy się konsolidacja obiektu programu zawierającego shader geometrii, który wywołuje funkcję

EmitStreamVertex lub EndStreamPrimitive, a typem jego prymitywu wejściowego nie jest points.

Funkcja	Opis
void EmitStreamVertex(int stream)	Emisja bieżących wartości zmiennych wyjściowych do bieżącego prymitywu wyjściowego w strumieniu stream, włączając w to zmienne wbudowane gl_PointSize, gl_ClipDistance, gl_Layer, gl_Position, gl_PrimitiveID i gl_ViewportIndex. Wartości wszystkich zmiennych wyjściowych są nieokreślone po wywołaniu EmitStreamVertex. Jeżeli shader geometrii generuje większą ilość wierzchołków niż wynika to z wartości identyfikatora max_vertices kwalifikatora formatu wyjściowego, rezultat wywołania EmitStreamVertex jest nieokreślony. Powyższa funkcja może być używana tylko w przypadku, gdy obsługiwana jest technika wielu strumieni wyjściowych.
void EndStreamPrimitive(int stream)	Emisja bieżącego prymitywu w strumieniu stream zostaje zakończona i jednocześnie rozpoczyna się emisja nowego prymitywu tego samego typu. Funkcja nie emituje wierzchołka. Jeżeli identyfikator kwalifikatora formatu wyjściowego został zadeklarowany jako points, wywoływanie EndStreamPrimitive jest opcjonalne. Gdy shader geometrii kończy działanie bieżący prymityw jest automatycznie kończony. Stąd wywołanie EndStreamPrimitive nie jest konieczne, gdy shader zapisuje pojedynczy prymityw. Powyższa funkcja może być używana tylko w przypadku, gdy obsługiwana jest technika wielu strumieni wyjściowych.
void EmitVertex()	Emisja bieżących wartości zmiennych wyjściowych do bieżącego prymitywu wyjściowego, włączając w to zmienne wbudowane gl_PointSize, gl_ClipDistance, gl_Layer, gl_Position, gl_PrimitiveID i gl_ViewportIndex. Wartości wszystkich zmiennych wyjściowych są nieokreślone po wywołaniu EmitVertex. Jeżeli shader geometrii generuje większą ilość wierzchołków niż wynika to z wartości identyfikatora max_vertices kwalifikatora formatu wyjściowego, rezultat wywołania EmitVertex jest nieokreślony. W przypadku użycia techniki wielu strumieni wyjściowych wywołanie EmitVertex odpowiada EmitStreamVertex(0).
void EndPrimitive()	Emisja bieżącego prymitywu zostaje zakończona i jednocześnie rozpoczyna się emisja nowego prymitywu tego samego typu. Funkcja nie emituje wierzchołka. Jeżeli identyfikator kwalifikatora formatu wyjściowego został zadeklarowany jako

	points, wywoływanie EndPrimitive jest opcjonalne. Gdy shader geometrii kończy działanie bieżący prymityw jest automatycznie kończony. Stąd wywołanie EndPrimitive nie jest konieczne, gdy shader zapisuje pojedynczy prymityw. W przypadku użycia techniki wielu strumieni wyjściowych wywołanie EndPrimitive odpowiada EndStreamPrimitive(0).

Tabela 30 Funkcje shadera geometrii.

Funkcje kontroli wywołania shadera

Funkcje kontroli wywołania shadera (Tabela 31) dostępne są tylko w shaderach kontroli teselacji i służą do kontroli kolejności wykonania wielu wywołań shadera na ścieżkę (zbiór) wierzchołków, które standardowo realizowane są w nieokreślonej kolejności.

Funkcja	Opis
void barrier()	Wszystkie wywołania shadera kontroli teselacji dla danej ścieżki wierzchołków muszą wywołać funkcję barrier, zanim przejdą do realizacji pozostałej części shadera. Funkcja barrier częściowo określa kolejność wykonywania poszczególnych wywołań shadera kontroli teselacji. Po wyjściu z funkcji mamy gwarancję, że wszystkie dane zapisane przez dane wywołanie shadera mogą być bezpiecznie pobrane przez inne wywołania shadera, czyli funkcja barrier umożliwia synchronizację danych. Funkcja barrier może być umieszona wyłącznie w funkcji main shadera kontroli teselacji i nie może stanowić składowej żadnej kontroli przepływu. Ponadto wywołanie funkcji barrier nie jest dopuszczalne po wykonaniu instrukcji return w funkcji main shadera.

Tabela 31 Zestawienie funkcji kontroli wywołania shadera.

Funkcja kontroli pamięci shadera

Shadery każdego rodzaju mogą zapisywać i odczytywać dane tekstury za pomocą zmiennych obrazu tekstury. Choć kolejność operacji odczytu/zapisu dla pojedynczego wywołania shadera jest ściśle określona, to jednak kolejność takich operacji na współdzielonej pamięci przez wiele oddzielnych wywołań shadera, nie jest w żaden sposób określona. Operacje takie mogą być jednak kontrolowane przy pomocy funkcji kontroli pamięci shadera wymienionej w Tabela 32.

Funkcja	Opis
void memoryBarrier()	Funkcja kontroluje kolejność operacji na pamięci wykonywanej przez pojedyncze wywołanie shadera. Funkcja oczekuje na zakończenie wszystkich operacji dostępu do zmiennych obrazu tekstury lub liczników atomowych w wywołaniach shadera zawierających funkcję memoryBarrier. Po wyjściu z funkcji wyniki operacji na pamięci bazowej zmiennych koherentnych (kwalifikator pamięci coherent) będą widoczne dla przyszłych operacji na tej samej pamięci bazowej wykonywanych w innych wywołaniach shadera. W szczególności wartości zapisane w ten sposób w jednym rodzaju shadera mają zagwarantowaną poprawną wartość przy dostępie do bazowej pamięci przez wywołania shadera w następnym programowalnym etapie renderingu (np. wywołania shadera fragmentów dla prymitywu będącego wynikiem działania shadera geometrii).

Tabela 32 Zestawienie funkcji kontroli pamięci shadera.