OpenGL ES Shading Language 1.0 - podręczna ściąga

1 W dwóch zdaniach

OpenGL ES Shading Language to język programowania, taki jak C++ lub Java, ale przeznaczony konkretnie do generowania efektów wizualnych w grach komputerowych. Do pisania programów w SL (Shading Language) wystarczy komputer z kartą graficzną oraz aktualna przeglądarka. Działające przykłady można znaleźć na stronie: **www.shadertoy.com**.

SL stosuje się do kilku różnych zadań. Najważniejszym z nich jest obliczanie koloru pikseli (nazywanych fragmentami). Właśnie na takich programach się skupimy.

2 Budowa programu

Program jest ciągiem **instrukcji**. Instrukcje kończy się średnikiem, chociaż jest kilka wyjątków. Instrukcje wykonywane są jedna po drugiej:

```
*instrukcja 1*;
*instrukcja 2*;
```

Instrukcje, które nie kończą się średnikiem, kończą się parą nawiasów klamrowych z dalszym zestawem instrukcji (tzw. blokiem kodu):

```
*instrukcja sterująca* {
  *instrukcja*;
}
```

Takie instrukcje pozwalają sterować programem - ominąć jakieś instrukcje albo powtarzać je wielokrotnie. Wrócimy do nich za chwilę.

3 Komentarze

W programie można umieszczać też tekstowe komentarze. Należy je jednak oddzielić od instrukcji.

Jednolinijkowy komentarza zaczyna się od `//`, a kończy wraz z daną linijką:

```
instrukcja; // ten komentarz zajmie jedną linijkę
```

Wielolinijkowy komentarz zaczyna się od `/*`, a kończy na `*/`:

```
instrukcja;
/*
 ten komentarz może zająć dowolną ilość linijek
 */
instrukcja;
```

4 Zmienne

Zmienne pozwalają przechowywać w pamięci liczby oraz wartości prawda/fałsz.

Zmienne posiadają nazwy, które muszą być pojedynczym słowem (bez spacji oraz wiodących cyfr).

Zmienną można dodać do programu pisząc jej **typ** oraz nazwę - nazywamy to deklaracją zmiennej. Deklaracja zmiennej jest instrukcją.

```
float x;
```

Nazwa zmiennej jest najprostszym rodzajem **wyrażenia**. Każde wyrażenie posiada typ oraz wartość.

Do zmiennej można wpisać wartość innego wyrażenia. Należy napisać nazwę zmiennej, znak = oraz inne wyrażenie, którego wartość zostanie przypisana do zmiennej. Aby przepisać wartość zmiennej x do zmiennej y, można napisać:

```
y = x;
```

Możliwe jest jednoczesne zdefiniowanie zmiennej oraz przypisanie do niej wartości:

```
float y = x;
```

Jeśli typy zmiennych nie będą się zgadzać, program wyświetli komunikat o błędzie.

5 Typy danych

Każde wyrażenie posiada **typ**. Dostępnych jest tylko kilka typów:

5.1 Typy proste

`bool`	wartości prawda / fałsz
`int`	liczba całkowita
`float`	(przybliżona) liczba rzeczywista

5.2 Typy wektorowe

`vec2, vec3, vec4`
dwie, trzy lub cztery liczby rzeczywiste (jako wektor)

`bvec2, bvec3, bvec4`
dwie, trzy lub cztery wartości prawda / fałsz

`ivec2, ivec3, ivec4`
dwie, trzy lub cztery liczby całkowite

5.3 Macierze

`mat2, mat3, mat4`
macierz liczb rzeczywistych 2x2, 3x3 lub 4x4

5.4 Tekstury

`sampler2D, samplerCube`
tekstura (obraz / film / muzyka)

6 Typy złożone

Poza typami wbudowanymi, istnieją dwa typy złożone - struktury oraz tablice. W języku SL wykorzystywane są jednak stosunkowo rzadko, dlatego nie będziemy o nich mówić. Zainteresowani mogą poczytać o C++ - w SL tablice i struktury działają analogicznie.

7 Wartości dosłowne

Trzy typy proste można w programie umieścić **dosłownie**. Dla typu **bool** dosłowne wartości to **true** i **false**:

```
bool prawda = true;
bool falsz = false;
```

Dla typu int dosłowne wartości to liczby całkowite z przedziału od -1023 do 1023:

```
int x;
x = -1023;
x = 1023;
```

Dla typu float dosłowne wartości to liczby zawierające znak `.` (nie wolno o nim zapomnieć). Przechowują ok. 3 cyfry znaczące ale mogą być przesunięte o kilka miejsc w lewo lub w prawo. Zwykle nie przechowują wartości dokładnie.

```
float x;
x = 3.14;
x = 0.00501;
x = 12300.0;
```

8 Operatory

Wyrażenia można łączyć ze sobą za pomocą operatorów. Tak połączone wyrażenia tworzą kolejne wyrażenie. Oto wszystkie operatory w kolejności pierwszeństwa:

Operator	Opis
`(x)`	Pozwala wymusić pierwszeństwo dla wyrażenia x
`funkcja(argument1, …)`	Uruchamia określoną funkcję przekazując do niej zestaw argumentów*
`struktura.pole`	Wybiera z danej struktury pole o określonej nazwie*
`x++` `x--`	Zwiększa lub zmniejsza wartość x o 1. Wartością jest poprzednia wartość zmiennej
`++x` `--x`	Zwiększa lub zmniejsza wartość x o 1. Wartością jest nowa wartość zmiennej
`+x` `-x`	Zachowuje lub zmienia znak zmiennej.
`!x`	Zmienia wartość **true** na **false** i odwrotnie
`x * y` `x / y`	Mnoży lub dzieli x przez y
`x + y` `x - y`	Dodaje lub odejmuje x i y

`x < y`
`x <= y`
`x > y`
`x >= y`
	Porównuje wartości x i y. Wartością jest **true** lub **false**

`x == y` `x != y`	Sprawdza czy x i y są równe lub nierówne. Wartością jest **true** lub **false**
`x && y`	Logiczna koniunkcja (x i y)
`x ^^ y`	Logiczna różnica (x różne od y)
`x || y`	Logiczna alternatywa (x lub y)
`x ? y : z`	Jeśli x to **true**, wartością jest y. W przeciwnym razie z

`x = y`
`x += y`
`x -= y`
`x *= y`
`x /= y`
	Przypisuje, dodaje, odejmuje, mnoży lub dzieli x przez y. Zmienia wartość x. x musi być nazwą zmiennej

`x, y`	Ignoruje x, wartością jest y

Jedyne operatory, które modyfikują wartości zmiennych to `++`, `--`, `+=`, `-=`, `*=`, `/=` oraz `=`. Pozostałe wykorzystują wartości innych wyrażeń żeby coś obliczyć.

(*) wyjaśnienie dalej

9 Funkcje

Funkcje to zdefiniowane wcześniej ciągi instrukcji, które pozwalają w prosty sposób wykonywać różne zadania. Funkcje mogą przyjmować zestaw argumentów oraz zwracać obliczoną wartość.

Można definiować własne funkcje. Przykładem jest funkcja **main**, która jest obowiązkową częścią każdego programu. Przed funkcją main można zdefiniować więcej funkcji, z których będzie można korzystać dalej. Definicja wygląda zawsze następująco: zwracany typ, nazwa funkcji, zestaw argumentów umieszczony w nawiasach okrągłych oraz ciąg instrukcji w nawiasach klamrowych. Jeśli funkcja zwraca jakiś typ danych, na końcu funkcji musi znajdować się instrukcja `return <wyrażenie>;`:

```
float kwadrat(float x) {
  float k = x * x;
  return k;
}
```

Niektóre funkcje nie zwracają żadnej wartości - można wtedy użyć typu pustego - **void**. Robi się tak w przypadku obowiązkowej funkcji **main**.

Ta sama funkcja może być zdefiniowana kilka razy dla różnych zestawów parametrów - właściwy wariant zostanie wybrany dopiero przy jej użyciu - w zależności od typów wykorzystanych argumentów. Tam gdzie funkcja może przyjmować wiele różnych rodzajów argumentów, w dalszych tabelach użyto typu T.

SL posiada dość bogaty zestaw domyślnie zdefiniowanych funkcji:

9.1 Funkcje do konstruowania wektorów

`vec2 vec2(float x, float y)`
: buduje wektor 2-elementowy z dwóch liczb rzeczywistych

`vec3 vec3(float x, float y, float z)`
: buduje wektor 3-elemenetowy z trzech liczb rzeczywistych

`vec4 vec4(float x, float y, float z, float w)`
: buduje wektor 4-elementowy z czterech liczb rzeczywistych

W przypadku powyższych funkcji (oraz funkcji do budowania wektorów liczb całkowitych oraz wektorów prawda/fałsz), na liście argumentów można użyć mniejszych wektorów, żeby oszczędzić sobie pisania:

```
vec2 maly_wektor = vec2(0.0, 1.0);
vec3 wiekszy_wektor = vec3(-1.0, maly_wektor);
vec4 najwiekszy_wektor = vec4(maly_wektor,
                              maly_wektor);
```

Analogicznie działają funkcje do konstrukcji macierzy.

9.2 Funkcje trygonometryczne

`T radians(T degrees)`
: zamienia kąt w stopniach na radiany

`T degrees(T radians)`
: zamienia kąt w radianach na stopnie

`T sin(T angle)` sinus
`T cos(T angle)` cosinus
`T tan(T angle)` tangens
`T asin(T x)` arcus sinus
`T acos(T x)` arcus cosinus
`T atan(T y, T x)` arcus tangens
`T atan(T y_przez_x)` arcus tangens ilorazu

9.3 Funkcje wykładnicze

`T pow(T x, T y)` x^y
`T exp(T x)` e^x
`T log(T x)` $\ln(x)$
`T exp2(T x)` 2^x
`T log2(T x)` $\log_2(x)$
`T sqrt(T x)` $\sqrt{x}$
`T inversesqrt(T x)` $1/\sqrt{x}$

9.4 Różne funkcje

`T abs(T x)`
wartość bez znaku
`T sign(T x)`
-1.0, 0.0 lub 1.0, zależnie od znaku x
`T floor(T x)`
największa całkowita liczba <= x
`T ceil(T x)`
najmniejsza całkowita liczba >= x
`T fract(T x)`
(dodatnia) część ułamkowa x
`T mod(T x, T y)`
reszta z dzielenia x przez y
`T mod(T x, float y)`
reszta z dzielenia x przez y
`T min(T x, T y)`
mniejsza z wartości x i y
`T min(T x, float y)`
mniejsza z wartości x i y
`T max(T x, T y)`
większa z wartości x i y
`T max(T x, float y)`
większa z wartości x i y
`T clamp(T x, T min, T max)`
x przycięte do wartości min i max
`T clamp(T x, float min, float max)`
x przycięte do wartości min i max
`T mix(T x, T y, T a)`
liniowo pomieszane x i y. Dla a == 0.0, wartością jest x. Dla a == 1.0, wartością jest y
`T mix(T x, T y, float a)`
liniowo pomieszane x i y. Dla a == 0.0, wartością jest x. Dla a == 1.0, wartością jest y
`T step(T step, T x)`
0.0 jeśli x < step. W przeciwnym razie 1.0
`T step(float edge, T x)`
0.0 jeśli x < step. W przeciwnym razie 1.0
`T smoothstep(T start, T stop, T x)`
dla x < start, wartość to 0.0, dla x > stop, wartość to 1.0. Pomiędzy - gładkie przejście
`T smoothstep(float start, float stop, T x)`
dla x < start, wartość to 0.0, dla x > stop, wartość to 1.0. Pomiędzy - gładkie przejście

9.5 Funkcje geometryczne

`float length(T x)`
długość wektora
`float distance(T p0, T p1)`
odległość pomiędzy punktami
`float dot(T x, T y)`
iloczyn skalarny wektorów
`vec3 cross(vec3 x, vec3 y)`
iloczyn wektorowy
`T normalize(T x)`
wektor o identycznym zwrocie i długości 1.0 (znormalizowany)
`T faceforward(T N, T I, T Nref)`
zwraca N jeśli dot(Nref, I) < 0, w przeciwnym razie -N
`T reflect(T I, T N)`
odbija wektor I w kierunku N
`T refract(T I, T N, float eta)`
załamuje wektor I w kierunku N

9.6 Funkcje macierzowe

`mat matrixCompMult(mat x, mat y)`
mnoży x przez y element po elemencie

9.7 Funkcje porównujące wektory

`bvec lessThan(T x, T y)`	x < y
`bvec lessThanEqual(T x, T y)`	x <= y
`bvec greaterThan(T x, T y)`	x > y
`bvec greaterThanEqual(T x, T y)`	x >= y
`bvec equal(T x, T y)`	x == y
`bvec equal(bvec x, bvec y)`	x == y
`bvec notEqual(T x, T y)`	x != y
`bvec notEqual(bvec x, bvec y)`	x != y

`bool any(bvec x)`
true jeśli którykolwiek element x to true
`bool all(bvec x)`
true jeśli wszystkie elementy x to true
`bvec not(bvec x)`
logiczna negacja x

9.8 Odczytywanie wartości z tekstur

Dla zwięzłości, pominięto kilka alternatywnych (rzadko stosowanych) form poniższych funkcji.

`vec4 texture2D(sampler2D tex, vec2 coord)`
pobiera kolor z tekstury; coord powinien zawierać się w przedziale (0, 1)
`vec4 textureCube(samplerCube tex, vec3 coord)`
pobiera kolor z tekstury sześciennej

10 Wektory

Wektory grupują kilka składowych. Każda składowa ma nazwę, która pozwala wybrać jej wartość. Najczęściej spotykane nazwy pól to x, y, z, w. Do wartości pola można dobrać się za pomocą operatora `.`:

```
vec2 wektor = vec2(0.0, 1.0);
float jedynka = wektor.y;
```

Można stosować także alternatywne nazwy pól: rgba (nawiązujący do palety kolorów RGB), oraz stpq (wynikający z nazw zwyczajowo stosowanych w grafice 3d). Wszystkie te nazwy oznaczają jednak te same pola:

```
wektor.x == wektor.r; // true
wektor.r == wektor.s; // true
```

Wektory pozwalają na powielanie i mieszanie składowych za pomocą specjalnych pól mieszających:

```
vec2 zero_jeden == vec2(0.0, 1.0);
vec2 jeden_zero = zero_jeden.yx;
vec4 cztery_zera = zero_jeden.xxxx;
```

11 Instrukcje sterujące

Kilka instrukcji pozwala na ominięcie lub powtórzenie jakichś innych instrukcji.

11.1 If-else

Instrukcja **if** wykona blok kodu tylko jeśli warunek podany w nawiasach będzie mieć wartość **true**:

```
float x = 5.9;

if(x > 5.0) {
  x = 5.0;
}
```

Za instrukcją **if** można umieścić instrukcję **else** - wykona się ona jeśli blok **if** nie zostanie wykonany:

```
float x = 5.9;

if(x > 5.0) {
  x = 1.0;
} else {
  x = 0.0;
}
```

11.2 Pętla while

Dostępnych jest kilka różnych form pętli - `for`, `while`, oraz `do-while`. Pętla **while** jest najbardziej ogólna więc powiemy tylko o niej.

Pętla **while** będzie wykonywać blok kodu tak długo jak warunek podany w nawiasach będzie mieć wartość **true**:

```
int i = 0;

while(i < 20) {
  x *= 1.61;
  ++i;
}
```

While działa podobnie do **if**, ale powtarza blok kodu, aż warunek przestanie być spełniany.

12 Wbudowane zmienne

Programy mają do dyspozycji dwie zmienne globalne:

`vec4 gl_FragCoord`
: pozycja danego fragmentu (piksela)

`vec4 gl_FragColor`
: kolor danego fragmentu (piksela)

Jedyny obowiązek funkcji main to wpisanie odpowiedniego koloru do zmiennej **gl_FragColor**.

Kolejne elementy tego wektora odpowiadają trzem składowym koloru (czerwony, zielony, niebieski) oraz wartości krycia danego piksela. Przyjmują wartości od 0.0 do 1.0.

13 Zmienne jednorodne

Program wyświetlający grafikę może wysłać do karty graficznej dodatkowe zmienne, które będą dostępne przy obliczaniu wszystkich pikseli danej klatki animacji. Takie zmienne oznacza się słowem kluczowym **uniform**.

Do zmiennych jednorodnych nie można przypisywać żadnych wartości.

Zmienne jednorodne dostępne na stronie **shadertoy.com** to:

`vec3 iResolution`
: rozdzielczość okna (szerokość, wysokość, 1.0) w pikselach

`float iGlobalTime`
: czas odtwarzania w sekundach

`sampler2D/samplerCube iChannel0..3`
: tekstura wybrana w danym kanale

`float iChannelTime[4]`
: czas odtwarzania tekstury

`vec3 iChannelResolution[4]`
: rozdzielczość tekstury

`vec4 iMouse`
: kiedy trzymany jest jakiś przycisk, w polach **xy** - pozycja myszki; w polach **zw** - pozycja kliknięcia

`vec4 iDate`
: data (rok, miesiąc, dzień, czas w sekundach)

14 Przykładowy program

```
void main() {
  float d = sin(gl_FragCoord.x / 10.0);
  d = d * d;
  gl_FragColor = vec4(d, d, d, 1.0);
}
```