

GLSL. Введение

Компьютерная графика

Хранение и компиляция GLSL шейдеров

- GLSL шейдеры принято хранить в виде исходных кодов (хотя в OpenGL 4.1 и появилась возможность загружать шейдеры в виде бинарных данных)
- Такой подход был использован для лучшей переносимости шейдеров на различные аппаратные и программные платформы
- Исходные коды компилируются драйвером
- Они могут быть скомпилированы лишь после создания действующего контекста OpenGL
- Драйвер сам генерирует внутри себя оптимальный двоичный код, который понимает данное оборудование. Это гарантирует, что один и тот же шейдер будет правильно и эффективно работать на различных платформах.

Шаги загрузки и компиляции

- Сначала выделяются идентификаторы в виде GLuint, под шейдеры — **glCreateShader** и шейдерную программу **glCreateProgram**
- На идентификатор шейдера загружается исходный код, который передается драйверу **glShaderSource**
- После шейдер компилируется **glCompileShader**
- Несколько шейдеров разных типов, прикрепляются к программе **glAttachShader**
- Последний шаг — линкование прикрепленных шейдеров в одну шейдерную программу **glLinkProgram**

Обобщённая структура шейдера

```
#version version_number
```

```
attribute type attribute_name;
```

```
in type in_variable_name;
```

```
out type out_variable_name;
```

```
uniform type uniform_name;
```

```
void main() {
```

```
    // делаем, вычисляем, экспериментируем
```

```
    ...
```

```
    // Присваиваем результат работы шейдера выходной переменной
```

```
    out_variable_name = weird_stuff_we_processed;
```

```
}
```

Атрибут (attribute)

```
attribute vec2 coord;
```

```
void main() {
```

```
    gl_Position = vec4(coord, 0.0, 1.0);
```

```
}
```

- Данные передаваемые программой вершинному шейдеру (другим шейдерам данные не доступны)
- Данные приходят шейдеру на каждую вершину
- Данные доступны только для чтения

Количество атрибутов

OpenGL гарантирует возможности передачи по крайней мере 16 4-х компонентных атрибутов, иначе говоря, в шейдер можно передать как минимум 64 значения для каждой вершины

Узнать максимальное количество входных переменных-вершин, передаваемых в шейдер, можно узнать обратившись к атрибуту *GL_MAX_VERTEX_ATTRIBS*

```
GLint nrAttributes;  
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &nrAttributes);  
std::cout << "Maximum" << nrAttributes;
```

Юниформ (uniform)

```
uniform vec4 color;  
void main() {  
    gl_FragColor = color;  
}
```

- Данные посылаемые в шейдер приложением
- В отличии от атрибута они глобальны
- И для фрагментных шейдеров, и для вершинных
- Если объявить юниформ переменную с одинаковым именем в вершинном и фрагментом шейдере, они будут общими для них
- Не зависят от того какая сейчас вершина обрабатывается, они остаются неизменными, пока их не изменит приложение

in и out переменные

- переменные, передаваемые из вершинного шейдера в фрагментный
- интерполируются вдоль грани с учетом в том числе и перспективы

```
//Переменная передаётся в следующий шейдер, уходит  
out vec3 viewVec;
```

```
//Переменная приходит в шейдер  
in vec3 viewVec;
```


Встроенные переменные

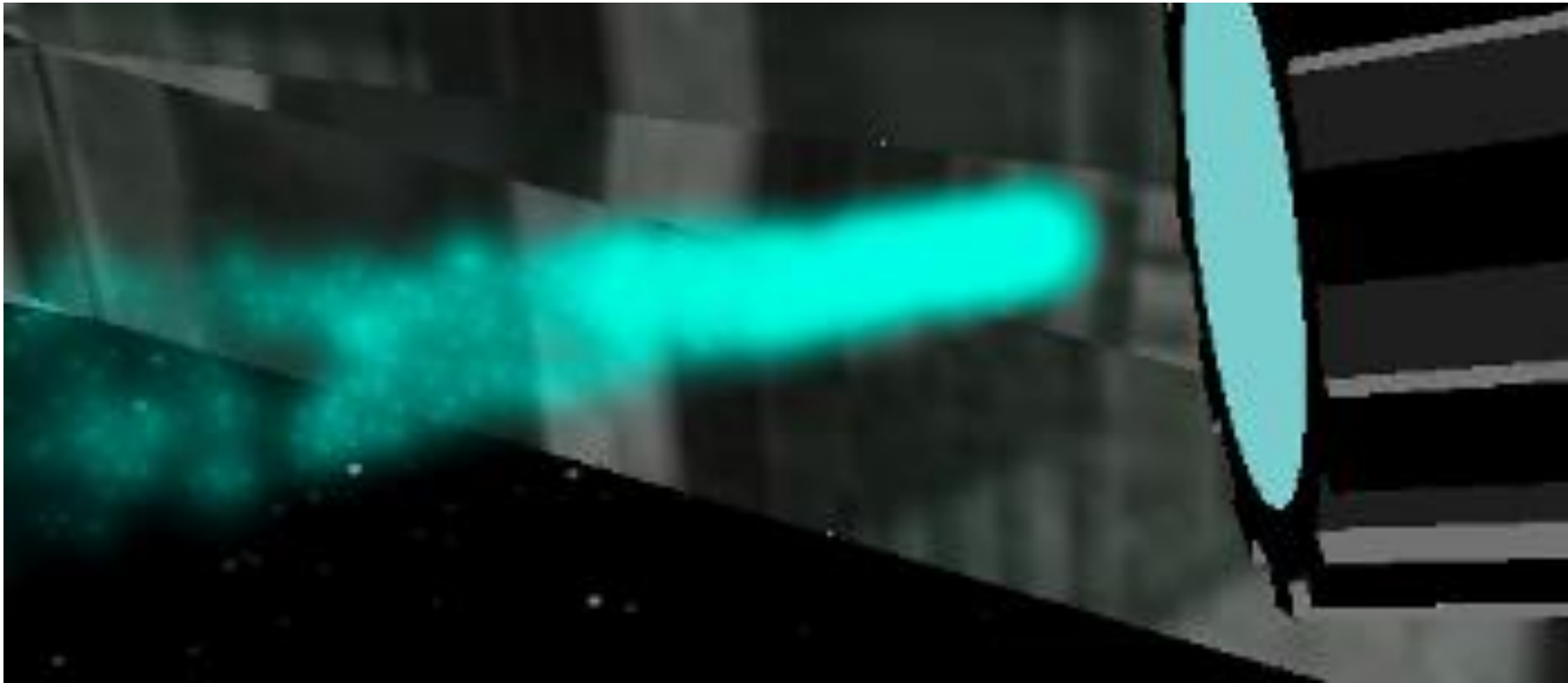
```
attribute vec2 coord;
void main() {
    gl_Position = vec4(coord, 0.0, 1.0);
}
```

```
uniform vec4 color;
void main() {
    gl_FragColor = color;
}
```

- `gl_Position` — это встроенная переменная для записи обработанной вершинным шейдером позиции вершины
- Далее данные о вершине идут дальше по конвейеру
- `gl_FragColor` — в неё записывается обработанный фрагментным шейдером цвет фрагмента

Размер точки

`gl_PointSize` : float, размер точки



Типы данных GLSL

Scalar — bool, int, uint, float, double

Array

Vector

Matrix

Sampler — Встроенный тип данных (сэмплер)

Struct — Пользовательские типы данных

Квалификаторы для чисел с плавающей точкой

`highp` — число с плавающей точкой сохраняет максимальную точность — положение вершины

`mediump` — число со средней точностью — текстурные координаты

`lowp` — число с низкой точностью — значения цвета

//Например

```
lowp vec4 color=vec4(1.0,1.0,0.0,1.0); // lowp достаточно для хранения цвета
```

Задание и проверка поддержки точности

Задать точность для всех переменных определённого типа можно с помощью ключевого слова **precision**

```
precision highp float;    //теперь все float считаются highp
precision mediump int;    //все int считаются mediump
```

```
#ifdef GL_FRAGMENT_PRECISION_HIGH
precision highp float;
#else
precision mediump float;
#endif
```

Array

```
float a[5];
```

```
...
```

```
float b[] = a;
```

```
float b[5] = a;
```

```
float b[] = float[](1,2,3,4,5);
```

Vector

`vecn` (например, `vec4`) — это `vector`, содержащий в себе `n` значений типа `float`

`bvecn` (например, `bvec4`) — это `vector`, содержащий в себе `n` значений типа `boolean`

`ivec n` (например, `ivec4`) — это `vector`, содержащий в себе `n` значений типа `integer`

`uvecn` (например, `uvecn`) — это `vector`, содержащий в себе `n` значений типа `unsigned integer`

`dvecn` (например, `dvecn`) — это `vector`, содержащий в себе `n` значений типа `double`

`vec2, vec3, vec4`

`ivec2, ivec3, ivec4`

`bvec2, bvec3, bvec4`

`vec2 v = vec2(1.0, 2.0);`

Доступ к компонентам векторов

как к обычным массивам

через . с помощью {x, y, z, w}

через . с помощью {r, g, b, a}

через . с помощью {s, t, r, q}

```
vec4 Vec4;
```

```
Vec4[0] = Vec4.x = Vec4.r = Vec4.s
```

```
Vec4[1] = Vec4.y = Vec4.g = Vec4.t
```

```
Vec4[2] = Vec4.z = Vec4.b = Vec4.r
```

```
Vec4[3] = Vec4.w = Vec4.a = Vec4.q
```

```
vec3 v2 = Vec4.rgb;
```

```
//vec4 v3 = Vec4.xуба; ошибка!
```

```
vec2 myVec = vec2(0.0,1.0);
```

```
float myF0 = myVec[0]; // myF0=0.0
```

```
float myF1 = myVec[1]; // myF1=1.0
```

```
vec2 myVeci = myVec.xy // myVec2 = {0.0,1.0}
```

```
myVeci = myVec.xx // myVec2 = {0.0,0.0}
```

```
myVeci = myVec.yy // myVec2 = {1.0,1.0}
```


swizzling

Из вектора, при обращении к данным через точку, можно получить не только одно значение, но и целый вектор, используя следующий синтаксис, который называется **swizzling**

```
vec2 someVec;  
vec4 differentVec = someVec.xyxx;  
vec3 anotherVec = differentVec.zyw;  
vec4 otherVec = someVec.xxxx +  
anotherVec.yxzy;
```

```
vec2 vect = vec2(0.5f, 0.7f);  
vec4 result = vec4(vect, 0.0f, 0.0f);  
vec4 otherResult = vec4(result.xyz, 1.0f);
```

Matrix

mat2, mat3, mat4

```
mat2 fMatrix = { 0.0f, 0.1, 2.1f, 2.2f };
```

```
vec4 c[3][2] = { vec4[2](vec4(0.0), vec4(1.0)),  
                vec4[2](vec4(0.0), vec4(1.0)),  
                vec4[2](vec4(0.0), vec4(1.0))  
                };
```

```
vec4 b[2] = vec4[2](vec4(0.0), vec4(0.1));
```

```
vec4[3][2] a = vec4[3][2](b, b, b);
```

Доступ к компонентам матриц

```
mat4 m;  
m[2] = vec4(1.0, 2.0, 3.0, 4.0);  
m[0][0] = 1.0;
```

Метод length

```
vec4 a[3][2];  
a.length() // this is 3  
a[x].length() // this is 2
```

Sampler

```
uniform sampler2D gSampler;
```

```
//Получение цвета пикселя в текстуре по заданным координатам  
texture2D(sampler2D sampler, vec2 coord)
```

```
vec3 tempColor= texture2D(gSampler,gl_TexCoord[0].xy);
```

struct

```
struct LightStruct {  
    vec4 position;  
    vec4 color;  
    float force;  
} LightVar;
```

```
LightVar=LightStruct(vec4(0.5,1.0,0.0,0.0), //position  
                    vec4(0.0,1.0,1.0,0.0),//color  
                    0.5);                //force
```

```
vec4 pos=LightVa.position;  
float f=LightVar.force;
```

const

```
const float zero = 0;      // Ошибка!  
const float zero = 0.0; // Правильно
```

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);  
const mat4 identity = mat4(1.0);
```

Функции, операторы и блоки в GLSL

Операции и Выражения

Функции

Блоки {}

Операторы

Операции и выражения

Операции

+, -, *, /, =, +=, ++,
<<, >>, ==, !=, >=,
!, &&, ^^, ||

// на некоторых GPU
x = x/2.0; // медленнее
y = y*0.5; // быстрее

Примеры встроенных функций GLS

`abs(x); round(x)`

`mod(x); fmod(x)`

`sqrt(x); pow(x, y)`

`max(x, y); min(x, y)`

`sin(angle); cos(angle); tan(angle)`

`log(x)`

`dot(x, y)` — скалярное произведение векторов x и y

`cross(x, y)` — векторное произведение векторов x и y

`matrixCompMult(mat x, mat y)` — произведение матриц x и y , которые должны быть одной размерности

`normalize(x)` — нормализованный вектор x

`reflect(t, n)` — отражает вектор t вдоль вектора n

`vec4 texture2D (sampler2D sampler, vec2 coord)` — выборка текстур

Функции

```
vec4 diffuse(vec3 normal,vec3 light,vec4 baseColor) {  
    return baseColor * dot(normal, light);  
}
```

Важно: Функции в GLSL не могут быть рекурсивными

```
// Создание матрицы, для двухмерного вращения
mat2 rot(in float a) {
    return mat2(cos(a),sin(a),-sin(a),cos(a));
}
```

```
// Создание матрицы, для двухмерного вращения (альтернативный вариант)
void rot2(in float a, out mat2 result) {
    result = mat2(cos(a),sin(a),-sin(a),cos(a));
}
```

```
// Используем функцию
vec2 pos1 = rot(17.0) * pos;
```

```
// альтернативный вариант
mat2 m;
rot2(17.0, m);
pos1 = m * pos;
```

Операторы управления

if

switch

discard // для фрагментного шейдера

// приводит к прекращению дальнейшей обработки текущего фрагмента

//без его попадания в буфер цвета

break

continue

do

for

while

```
if (color.a < 0.25) {  
    color *= color.a;  
}  
else {  
    color = vec4(0.0);  
}
```

```
for(int i = 0; i < 3; i++) {  
    sum += i;  
}
```

//Важно. Циклы стараться не применять!
//Особенно во фрагментных шейдерах

```
float value = gl_TexCoord[0].x;  
if (value > 0.5) {  
    discard();  
    //прекращение выполнения программы и запрещение обработки пикселя  
}
```

Ограничения оператора discard

discard доступен только в фрагментном шейдере

Вызов discard вычеркивает пиксель как будто у него не только альфа равна 0, но и не учитывает его в буфере глубины и остальных буферах.

После вызова все вычисления по заданному пикселю прекращаются

discard не зависит от режима блендинга

И главное: discard является тяжелой операцией. Производители рекомендуют не использовать discard, потери производительности будут большими

Дело в том что при вызове discard чипу нужно остановить ВСЕ процессоры и перестроить буфер глубины, перестроить фрагмент и т.д. и только после этого можно продолжать работать

Макросы

Присутствуют макросы:

`#if`, `#ifdef`, `#elif`, `#else`, `#endif`, `#define`, `#pragma` и прочие

Отсутствует:

`#include`

`#define PI 3.14`

Контроль версии

```
__VERSION__
```

4 версия это 400

версия 3.3 это 330

```
#if __VERSION__ > 400
```

```
// Тут расположен код для старших версий, начиная с 4 версии и выше
```

```
#else
```

```
// Тут код для остальных версий
```

```
#endif
```



Вариант 1

Vertex shader

```
in vec3 position;  
  
void main() {  
    // Напрямую передаем vec3 в vec4  
    gl_Position = vec4(position, 1.0);  
}
```

Fragment shader

```
void main() {  
    gl_FragColor = vec4(0.5f, 0.0f, 0.0f, 1.0f);  
}
```

Вариант 2

Vertex shader

```
in vec3 position;
```

```
void main() {  
    // Напрямую передаем vec3 в vec4  
    gl_Position = vec4(position, 1.0);  
}
```

Fragment shader

```
out vec4 color;
```

```
void main() {  
    color = vec4(0.5f, 0.0f, 0.0f, 1.0f);  
}
```

Вариант 3

Vertex shader

```
in vec3 position;
```

```
void main() {  
    // Напрямую передаем vec3 в vec4  
    gl_Position = vec4(position, 1.0);  
}
```

Fragment shader

```
uniform vec4 color1;
```

```
void main() {  
    gl_FragColor = color1;  
}
```

Вариант 4

Vertex shader

```
in vec3 position;

void main() {
    // Напрямую передаем vec3 в vec4
    gl_Position = vec4(position, 1.0);
}
```

Fragment shader

```
uniform vec4 color1;
out vec4 color;

void main() {
    color = color1;
}
```

Вариант 5

Vertex shader

```
#version 330 core
```

```
in vec3 position;
```

```
out vec4 vertexColor; // Передаем цвет во фрагментный шейдер
```

```
void main() {
```

```
    // Напрямую передаем vec3 в vec4
```

```
    gl_Position = vec4(position, 1.0);
```

```
    // Устанавливаем значение выходной переменной в темно-красный цвет.
```

```
    vertexColor = vec4(0.5f, 0.0f, 0.0f, 1.0f);
```

```
}
```


Fragment shader

```
#version 330 core
```

```
in vec4 vertexColor;
```

```
// Входная переменная из вершинного шейдера (то же название и тот же тип)
```

```
out vec4 color;
```

```
void main() {
```

```
    color = vertexColor;
```

```
    // или
```

```
    // gl_FragColor = vertexColor;
```

```
}
```

Вариант 6

Vertex shader

```
#version 330 core
```

```
in vec3 position;  
uniform vec4 color1;
```

```
out vec4 vertexColor; // Передаем цвет во фрагментный шейдер
```

```
void main() {  
    // Напрямую передаем vec3 в vec4  
    gl_Position = vec4(position, 1.0);  
    // Устанавливаем значение выходной переменной в темно-красный цвет.  
    vertexColor = color1;  
}
```

Fragment shader

```
#version 330 core
```

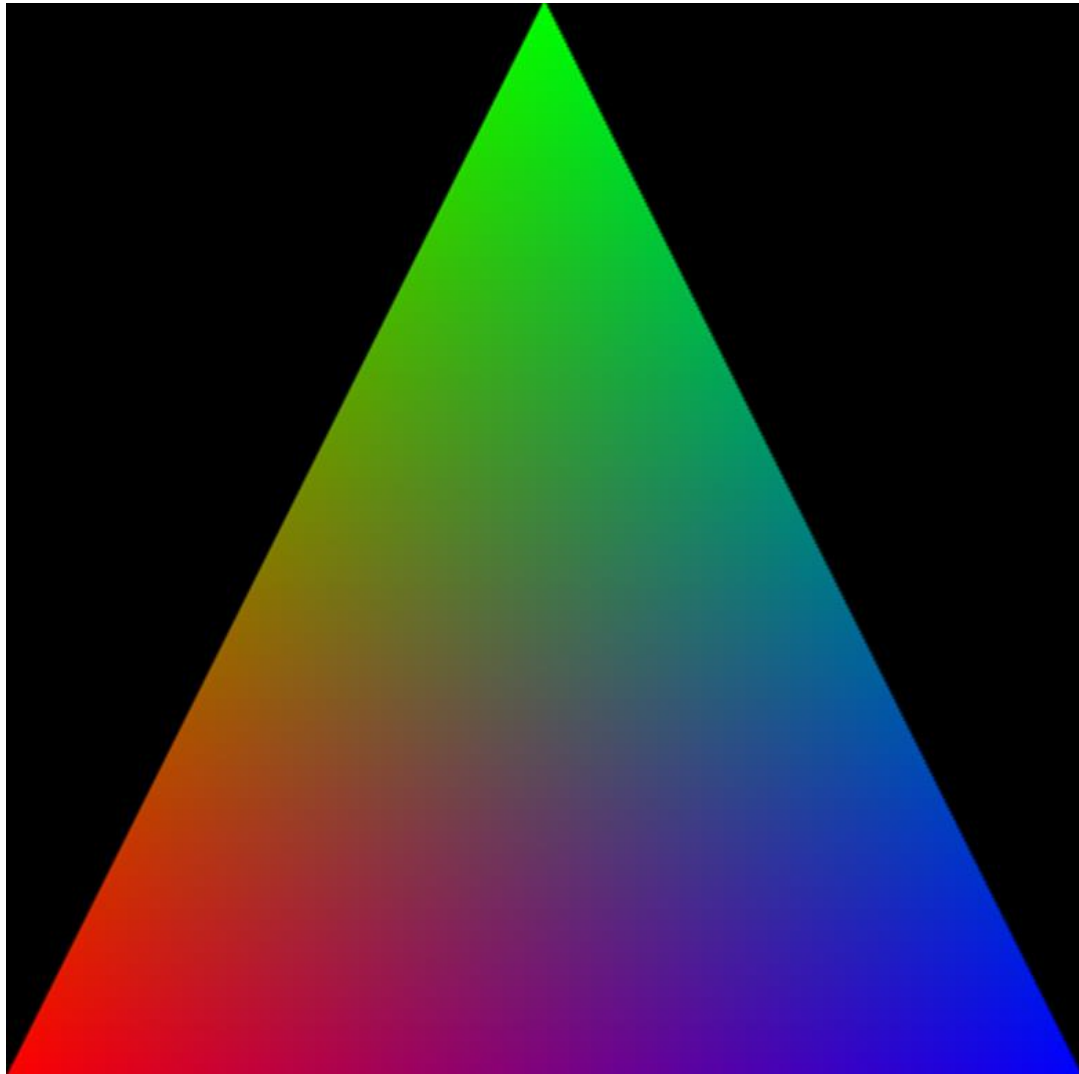
```
in vec4 vertexColor;
```

```
// Входная переменная из вершинного шейдера (то же название и тот же тип)
```

```
out vec4 color;
```

```
void main() {  
    color = vertexColor;  
}
```

Градиентная заливка



Vertex shader

```
#version 330 core
```

```
in vec3 position;
```

```
in vec4 color;
```

```
out vec4 vertexColor; // Передаем цвет во фрагментный шейдер
```

```
void main() {
```

```
    // Напрямую передаем vec3 в vec4
```

```
    gl_Position = vec4(position, 1.0);
```

```
    // Устанавливаем значение выходной переменной равным атрибуту цвета
```

```
    vertexColor = color;
```

```
}
```

Fragment shader

```
#version 330 core
```

```
in vec4 vertexColor;
```

```
// Входная переменная из вершинного шейдера (то же название и тот же тип)
```

```
out vec4 color;
```

```
void main() {
```

```
    color = vertexColor;
```

```
}
```

Шейдер с discard

```
// Раскрашиваем примитив в красный цвет, но в середине будет квадратная дыра
void main(void)
{
    gl_FragColor = vec4(1.0, 0., 0., 1.0);
    if(gl_FragCoord.x>0.25 && gl_FragCoord.x<0.75)
        if(gl_FragCoord.y>0.25 && gl_FragCoord.y<0.75)
            discard; // Игнорируем результат работы шейдера
}
```


Рекомендации по оптимизации шейдеров

Старайтесь сократить размер шейдера, особенно фрагментного

Часто многие расчеты можно вынести из фрагментного в вершинный

Старайтесь не использовать ветвления в фрагментном шейдере

Старайтесь использовать встроенные функции

Старайтесь использовать встроенные функции

```
float f1,f2,f3,f4;
```

```
...
```

```
float c = f1+f2+f3+f4; // медленно
```

```
float v = dot(vec4(f1,f2,f3,f4),vec4(1.0)); // быстро
```