Оглавление

Шейдеры

WebGPU поддерживает три типа шейдеров, каждый из которых выполняет свою основную функцию:

- вершинный шейдер — задаёт положение каждой вершины в сцене;
- фрагментный шейдер — задаёт цвет и глубину каждого отрисовываемого фрагмента;
- вычислительный шейдер — выполняет математические операции.

Несмотря на различные роли, каждый шейдер WebGPU написан на языке шейдеров WebGPU (WGSL). WGSL основан частично на JavaScript, частично на языке программирования C, но, в отличие от JavaScript и C, функции WGSL выполняются на графических процессорах, а не на центральных процессорах.

Основы WGSL

В этом разделе рассматриваются пять основных аспектов разработки на WGSL:

1. Определения функций и точки входа
2. Типы данных
3. Встроенные операторы и функции
4. Переменные и хранилище
5. Константы и переопределенные переменные

Определения функций и точки входа

На высоком уровне шейдер WebGPU содержит одно или несколько объявлений структур данных, за которыми следует одно или несколько определений функций. При определении функции следует учитывать два важных правила:

1. Имени функции должно предшествовать ключевое слово `fn`, а за ней в скобках должно следовать ноль или более параметров.
2. Функция WGSL не обязательно возвращает значение. Если она возвращает значение, за списком её параметров должна следовать жирная стрелка (`->`) и тип данных возвращаемого значения.

Каждый параметр должен иметь идентификатор, за которым следует двоеточие и тип данных. Это показано в следующем коде, где определяется функция с именем `foo`, возвращающая значение типа `bar`. Функция принимает аргумент с именем `arg` типа `argtype`:

```
// Example shader function

fn foo(arg: argtype) -> bar {

      return x;

}
```

Шейдер может иметь несколько определений функций, но только одна функция может служить точкой входа шейдера. О точках входа важно знать два важных момента:

1. Точке входа должен предшествовать атрибут, специфичный для типа шейдера (`@vertex` для вершинных шейдеров, `@fragment` для фрагментных шейдеров, `@compute` для вычислительных шейдеров).

2. Объект конвейера должен знать точку входа для каждого типа шейдера, который будет выполняться. Это можно задать вручную, присвоив имя функции свойству `entryPoint`. Если оно не задано, WebGPU будет искать атрибут шейдера. Если атрибут есть только у одной функции, эта функция будет принята в качестве точки входа.

Например, следующий код определяет вершинный шейдер. Поскольку это точка входа, ей должен предшествовать `@vertex`.

```
// Example entry point

@vertex

fn main(arg1: argtype1, arg2: argtype2, …) -> … {

      return …

}
```

Вы можете задать любое имя функции, но параметры и возвращаемое значение будут зависеть от типа кодируемого вами шейдера.

Типы данных

Одним из самых сложных аспектов кодирования на WGSL является отслеживание всех типов данных. В этом обсуждении они делятся на пять групп:

- скалярные типы — каждый экземпляр содержит одно значение;
- векторные типы — каждый экземпляр содержит несколько значений;
- матричные типы — каждый экземпляр содержит несколько строк и столбцов;

- массивы — контейнеры нескольких элементов одного типа;
- структурные типы — именованные группы связанных структур данных.

Скалярные типы

В отличие от JavaScript, в WGSL нет типов данных для строк или неопределённых переменных. Существует только пять основных скалярных типов, каждый из которых перечислен в таблице:

Тип данных	Описание
`bool`	Boolean value (true/false)
`u32`	Unsigned 32-bit integer
`i32`	Signed 32-bit integer
`f16`	16-bit floating-point value
`f32`	32-bit floating-point value

Многие графические процессоры поддерживают 16-битные значения с плавающей запятой, что может обеспечить прирост производительности для приложений, которым необходимо хранить и обрабатывать значительное количество значений. Они поддерживаются не всеми графическими процессорами, и вы можете проверить поддержку половинной точности, проверив свойство features графического адаптера и увидев, есть ли в нём элемент с именем shader-f16. Следующий код показывает, как это можно сделать:

```
// Check if the client supports 16-bit floating-point values
if (adapter.features.has("shader-f16")) {
      …
} else {
      …
}
```

Векторные типы

Одним из преимуществ использования графических процессоров для вычислений является возможность обработки нескольких значений за один такт. Однако эта группа значений должна храниться в специальных контейнерах, называемых векторами. Когда шейдер выполняет векторную операцию, он обрабатывает все значения вектора одновременно. По этой причине такие операции обычно называются операциями SIMD (Single-Instruction, Multiple Data).

В WGSL каждый векторный тип имеет формат `vecN<type>`, где `N` — количество значений в векторе, а `type` — тип каждого скалярного значения в векторе. Например, если переменная имеет тип `vec3<f32>`, это вектор, содержащий три 32-битных значения с плавающей запятой. При использовании переменной в операции все три значения будут обработаны одновременно.

Для упрощения кодирования WGSL предоставляет псевдонимы для распространённых типов векторов.

Псевдоним	Тип данных	Описание
vec2u	vec2<u32>	Содержит два 32-bit unsigned integers
vec3u	vec3<u32>	Содержит три 32-bit unsigned integers
vec4u	vec4<u32>	Содержит четыре 32-bit unsigned integers
vec2i	vec2<i32>	Содержит два 32-bit signed integers
vec3i	vec3<i32>	Содержит три 32-bit signed integers
vec4i	vec4<i32>	Содержит четыре 32-bit signed integers
vec2f	vec2<f32>	Содержит два 32-bit floating-point values
vec3f	vec3<f32>	Содержит три 32-bit floating-point values
vec4f	vec4<f32>	Содержит четыре 32-bit floating-point values
vec2h	vec2<f16>	Содержит два 16-bit floating-point values
vec3h	vec3<f16>	Содержит три 16-bit floating-point values
vec4h	vec4<f16>	Содержит четыре 16-bit floating-point values

Чтобы создать вектор, необходимо присвоить значение каждому его элементу. WGSL поддерживает объявления переменных, начинающиеся с `let`, `const` и `var`, а общий синтаксис объявления вектора выглядит следующим образом:

```
let var_name = vec_type(elements);
```

Например, следующий код создаёт вектор, содержащий четыре 32-битных числа с плавающей точкой:

```
let float_vec = vec4f(1.0, 2.0, 3.0, 4.0);
```

Аналогично, этот код создаёт константный вектор, содержащий три 32-битных целых числа со знаком:

```
const int_vec: vec3i = vec3i(1, 2, 3);
```

Шейдер может получить доступ к элементам вектора тремя основными способами:

- По индексу: `float_vec[0]`, `int_vec[1]`
- Именованые поля `r`, `g`, `b`, `a`: `float_vec.r`, `int_vec.b`
- Именованые поля `x`, `y`, `z`, `w`: `float_vec.y`, `int_vec.w`

Помимо доступа к отдельным элементам, шейдер может создавать подвекторы исходного вектора, используя несколько индексов. Это показано в следующем коде:

```
let my_vec = vec4f(0.0, 1.0, 2.0, 3.0);

let sub_vec = my_vec.yz; // sub_vec содержит (1.0, 2.0)
```

При доступе к элементам с использованием точечной нотации порядок важен. `my_vec.yz` — это двухэлементный вектор, содержащий второй и третий элементы `my_vec`. Напротив, `my_vec.zy` — это вектор, содержащий третий и второй элементы `my_vec`.

Типы матриц

Типы матриц похожи на типы векторов, но вектор хранит свои значения в строке, а матрица — в строках и столбцах. Другое отличие заключается в том, что матрицы могут хранить только значения с плавающей точкой, а не целые числа или булевы значения.

Каждый тип матрицы имеет формат `matMxN<type>`, где `M` — количество строк, `N` — количество столбцов, а `type` — тип каждого элемента матрицы. Значения `M` и `N` могут быть `2`, `3` или `4`.

Например, если переменная имеет тип `mat3x4<f32>`, это матрица с 12 значениями с плавающей точкой, расположенными в 3 строках и 4 столбцах.

Как и в случае с типами векторов, обсуждавшимися ранее, WGSL упрощает программирование, предоставляя псевдонимы для типов матриц.

Матрицы WGSL структурированы по столбцам. То есть, если `mat` — матрица, то `mat[0]` — вектор, содержащий элементы первого столбца матрицы. `mat[0][1]` — второй элемент первого столбца матрицы.

Чтобы прояснить это, рассмотрим следующий код, который инициализирует матрицу 3x2 с именем `mat_3_2`:

```
// Инициализирует матрицу 3x2 шестью значениями

let mat_3_2 = mat3x2f(0.0, 1.0, 2.0, 3.0, 4.0, 5.0);
```

Результирующая матрица будет состоять из трёх строк и двух столбцов. Поскольку элементы хранятся в порядке по столбцам, первый столбец будет содержать значения 0.0, 1.0 и 2.0, а второй — 3.0, 4.0 и 5.0.

Псевдоним	Тип данных	Описание
`mat2x2f`	`mat2x2<f32>`	Содержит четыре floats в двух строках и двух столбцах
`mat2x3f`	`mat2x3<f32>`	Содержит шесть floats в двух строках и трех столбцах
`mat2x4f`	`mat2x4<f32>`	Содержит восемь floats в двух строках и четырех столбцах
`mat3x2f`	`mat3x2<f32>`	Содержит шесть floats в трех строках и двух столбцах
`mat3x3f`	`mat3x3<f32>`	Содержит девять floats в трех строках и трех столбцах
`mat3x4f`	`mat3x4<f32>`	Содержит двенадцать floats в трех строках и четырех столбцах
`mat4x2f`	`mat4x2<f32>`	Содержит восемь floats в четырех строках и двух столбцах
`mat4x3f`	`mat4x3<f32>`	Содержит двенадцать floats в четырех строках и трех столбцах
`mat4x4f`	`mat4x4<f32>`	Содержит шестнадцать floats в четырех строках и четырех столбцах
`mat2x2h`	`mat2x2<f16>`	Содержит четыре half-floats в двух строках и двух столбцах
`mat2x3h`	`mat2x3<f16>`	Содержит шесть half-floats в двух строках и трех столбцах
`mat2x4h`	`mat2x4<f16>`	Содержит восемь half-floats в двух строках и четырех столбцах
`mat3x2h`	`mat3x2<f16>`	Содержит шесть half-floats в трех строках и двух столбцах
`mat3x3h`	`mat3x3<f16>`	Содержит девять half-floats в трех строках и трех столбцах
`mat3x4h`	`mat3x4<f16>`	Содержит двенадцать half-floats в трех строках и четырех столбцах
`mat4x2h`	`mat4x2<f16>`	Содержит восемь half-floats в четырех строках и двух столбцах
`mat4x3h`	`mat4x3<f16>`	Содержит двенадцать half-floats в четырех строках и трех столбцах
`mat4x4h`	`mat4x4<f16>`	Содержит шестнадцать half-floats в четырех строках и четырех столбцах

Шейдеры могут создавать матрицы из векторов. Например, если `vec1` и `vec2` содержат по четыре элемента, следующий код создаёт из векторов матрицу размером 4x2:

```
// Создаёт матрицу размером 4x2 из двух четырёхэлементных векторов

const mat = mat4x2f(vec1, vec2);
```

В результате `mat[0]` будет содержать элементы `vec1`, а `mat[1]` — элементы `vec2`.

Встроенные операторы и функции

WGSL предоставляет все операторы и функции, необходимые для обработки чисел, включая арифметические и тригонометрические операции (`sin`, `cos` и `tan`). Также рассматриваются функции для возведения в степень (`pow`, `exp`, `log`), сравнения (`min`, `max`, `clamp`) и аппроксимации (`round`, `floor`, `ceil`, `trunc`).

Эти функции могут работать непосредственно с векторами и матрицами. Они особенно важны при кодировании шейдеров, поэтому важно понимать их назначение.

Векторные операции и функции

Можно выполнять арифметические действия с векторами, используя те же операторы, что и для скаляров, при этом операции выполняются покомпонентно. То есть элементы выходного вектора вычисляются путём простой операции над соответствующими элементами входных векторов.

Например, предположим, что `v1` содержит (`1.0`, `1.0`, `1.0`), а `v2` содержит (`2.0`, `2.0`, `2.0`). Следующие операции выполняются покомпонентно:

```
let v3 = v1 + v2; // v3 = (3.0, 3.0, 3.0)

let v4 = v1 / v2; // v4 = (0.5, 0.5, 0.5)
```

Операции между векторами и скалярами также выполняются покомпонентно. Это показано в следующем примере кода:

```
let v3 = v1 + 5.0; // v3 = (6.0, 6.0, 6.0)

let v4 = v1 / 4.0; // v4 = (0.25, 0.25, 0.25)
```

Помимо базовых операторов, WGSL предоставляет функции для работы с векторами:

Функция	Описание
`length(vec)`	Возвращает длину вектора
`normalize(vec)`	Возвращает вектор с тем же направлением, что и `vec`, но длиной 1
`distance(vec1, vec2)`	Вычисляет расстояние между двумя точками
`dot(vec1, vec2)`	Возвращает скалярное произведение `vec1` и `vec2`
`cross(vec1, vec2)`	Возвращает векторное произведение `vec1` и `vec2`
`fma(vec1, vec2, res)`	Выполняет умножение и сложение

Функция `fma` умножает соответствующие элементы двух векторов и складывает результаты с элементами третьего вектора. Если входные данные — `a`, `b` и `c`, `fma` возвращает `a * b + c`. Основные преимущества этой функции — точность (снижение ошибки округления) и скорость (обычно выполняется за одну операцию). Функцию `fma` можно вызывать как со скалярными аргументами, так и с векторами.

Операции и функции с матрицами

Как и векторы, матрицы можно складывать и вычитать покомпонентно, используя операторы `+` и `-`. Но матрицы можно складывать и вычитать только при одинаковом количестве строк и столбцов.

WGSL не поддерживает деление матриц, но поддерживает умножение матриц. С матрицами оператор `*` можно использовать тремя способами:

- `matrix * scalar` – умножает каждый элемент матрицы на скаляр
- `matrix * vector` – возвращает вектор

- `matrix * matrix` – возвращает матрицу

Помимо арифметических операторов, WGSL предоставляет две полезные функции для работы с матрицами. Функция `transpose` принимает матрицу и возвращает матрицу с поменянными местами строки и столбцы. Функция `determinant` принимает матрицу и возвращает значение, равное определителю матрицы.

Типы массивов

Как и вектор, массив содержит несколько элементов одного типа. Но между массивами и векторами есть как минимум три важных различия:

1. Массив может хранить более четырёх элементов.
2. WGSL не предоставляет операций, обрабатывающих каждый элемент массива одновременно — шейдеру необходимо итерировать массив, чтобы выполнить операцию с его элементами.
3. В то время как вектор может содержать только скалярные значения, массив может содержать скаляры, векторы, матрицы и структуры. Но все элементы должны иметь один и тот же тип.

WGSL поддерживает два типа массивов, различие между которыми заключается в количестве элементов, или размере. Большинство массивов имеют фиксированный размер, и их размер должен быть указан при объявлении. Размер массива, размер которого определяется временем выполнения, можно изменить во время выполнения. Массивы фиксированного размера можно определить где угодно, но массивы размера времени выполнения можно использовать только в определённых ситуациях.

При объявлении массива фиксированного размера необходимо указать тип его элементов и количество элементов в массиве. Следующий код создаёт массив с именем `arr`, содержащий три элемента:

```
// Массив, содержащий три 32-битных беззнаковых целых числа

var arr = array<u32, 3>;
```

По возможности, шейдеры должны использовать векторы вместо массивов. Но если объекту необходимо хранить больше значений, чем может вместить вектор, массивы — единственный вариант. По этой причине, когда приложение передаёт большие буферы графическому процессору, к базовым данным часто обращаются как к массиву значений.

Типы структур

Когда шейдеру требуется доступ к связанным элементам данных, их обычно хранят в структуре. Это не даёт преимуществ в производительности, но упрощает код, организуя поля внутри именованного контейнера.

Как и в языке программирования C, структура объявляется ключевым словом `struct`, за которым следуют объявления элементов в фигурных скобках. Например, следующий код объявляет структуру с именем `Complex` и двумя элементами с плавающей точкой: `real` и `imag`:

```
// Example structure

struct Complex {

     real: f32,

     imag: f32

};
```

После этого объявления шейдер может создавать структуры, вызывая имя со значениями элементов в скобках. Следующий код показывает, как это работает:

```
const z = Complex(2.0, 4.0);
```

Структуры особенно полезны, когда шейдер получает связанные элементы данных из приложения.

Переменные и хранилище

WGSL поддерживает спецификаторы `let`, `const` и `var`, но существуют существенные различия между их использованием в WGSL и JavaScript. Использование `let` или `const` в шейдере означает объявление значения. Это объявление присваивает значение переменной, но данные нигде не сохраняются. Кроме того, `let` можно использовать только внутри функции, тогда как `const` можно использовать как внутри, так и вне функций.

Когда объявление начинается с `var`, оно создаёт переменную, данные которой будут храниться в памяти. Это сложнее, чем использование `let` и `const`, поскольку объявление `var` должно определять тип памяти, в которой будут храниться данные переменной.

Чтобы задать тип памяти переменной, после `var` можно указать имя адресного пространства в угловых скобках. Например, следующий код объявляет 32-битную переменную с плавающей точкой, которая будет храниться в приватном адресном пространстве:

```
var<private> foo: f32;
```

Другой способ задать тип переменной — указать значение. В следующем коде подразумевается, что переменная будет хранить значение с плавающей точкой:

```
var<private> foo = 1234.0;
```

В этих объявлениях ключевое слово `private` указывает, что переменная должна храниться в приватном адресном пространстве. Это доступно для глобальных переменных, объявленных вне функции шейдера. Если переменная предназначена для использования внутри функции, её адресное пространство должно быть установлено как `function`. Если переменная объявлена внутри функции без адресного пространства, подразумевается, что она будет храниться в пространстве функции. Это показано в следующем коде:

```
// Stored in the function address space

var foo = 1234.0;
```

Адресные пространства для переменных:

Адресное пространство	Режим доступа по умолчанию	Описание
`function`	read–write	Переменные, объявленные внутри функций
`private`	read–write	Глобальные переменные
`storage`	read	Данные в буфере хранения
`uniform`	read	Данные в `uniform` буфере
`handle`	read	Переменные сэмплеров и текстур
`workgroup`	read–write	Данные, используемые вычислительными шейдерами

Буферы хранят данные общего назначения. Текстуры хранят данные текстур. Для доступа к ресурсу шейдеру необходимо объявить переменную в адресном пространстве хранилища, пространства `uniform` переменных или пространства `handle` переменных. Переменные, предоставляющие доступ к ресурсам, называются переменными ресурсов.

Данные атрибутов (например, координаты) изменяются от вершины к вершине. `uniform` данные остаются постоянными (однородными) для всех вершин. По этой причине `uniform` данные хранятся в `uniform` буферах, доступ к которым осуществляется через `uniform` адресное пространство.

`Handle`-пространство используется только приложениями, обращающимися к текстурам. `workgroup` пространство рабочих групп используется только приложениями, создающими вычислительные шейдеры.

Константы и переопределенные переменные

Передача данных из приложения (JavaScript) в шейдер (WGSL) обычно представляет собой сложный процесс. Вам необходимо создать специальные объекты ресурсов (буферы и текстуры) и связать эти ресурсы со специально отформатированными данными и другими объектами.

Если вы просто хотите задать скалярные значения в шейдере, вы можете воспользоваться константами (которые не имеют никакого отношения к ключевому слову `const`). Ьетод `createRenderPipeline` принимает объект, включающий объект `vertex` и объект `fragment`. Объекты `vertex` и `fragment` имеют необязательное свойство с именем `constants`, которое может быть задано как последовательность пар «имя-значение». Следующий код показывает, как это работает:

```
// Create a render pipeline that defines constants
const renderPipeline = device.createRenderPipeline({
      layout: …,
      vertex: {
            module: …,
            entryPoint: …
            constants: {
                  0: false,
                  22: 5.4,
                  num_pixels: 256,
            }
      },
});
```

В этом коде `constants` присваивается объекту, содержащему три пары «имя-значение», разделённые двоеточиями. Каждое имя может быть целым числом или строкой. Каждое значение может быть целым числом, числом с плавающей точкой или булевым значением.

Внутри шейдера к этим парам «имя-значение» можно получить доступ через специальные переменные, называемые `override`-переменными. `override`-переменная может быть объявлена одним из двух способов. Если имя константы — целое число, соответствующее объявление начинается с атрибута `@id`, содержащего имя в скобках. Общая форма объявления выглядит следующим образом:

```
@id(name) override var_name: type;
```

Например, если объект констант отображает `0` в `false`, соответствующее объявление в шейдере можно записать следующим образом:

```
@id(0) override example: bool;
```

Несмотря на то, что в шейдере этой переменной не присваивается значение, при его выполнении ей будет присвоено значение `false`.

Если имя константы — строка, атрибут `@id` не нужен. Вместо этого имя переменной должно быть равно имени константы. Например, если объект констант отображает `num_pixels` в `256`, соответствующее объявление в шейдере можно записать следующим образом:

```
override num_pixels: u32;
```

Значение переменной не задаётся в шейдере, но благодаря объекту констант `num_pixels` будет установлено равным `256` при выполнении шейдера.

`override`-переменные удобны, когда приложению необходимо передавать простые скалярные значения в шейдер во время компиляции за пределами буферов и других ресурсов.

Вершинные шейдеры

Вершинный шейдер (vertex shader) – это программа, описывающая характеристики вершины (координаты, цвет и другие), а вершина – это точка в двух- или трехмерном пространстве, например, угол или вершина двух- или трехмерной фигуры.

Вершинный шейдер выполняется один раз для каждой вершины при рендеринге, и его основная цель — задать координаты вершины в пространстве. Как минимум, он должен вернуть `vec4f`, определяющий координаты вершины, но он также может вернуть структуру, содержащую координаты в дополнение к другим данным.

Рассмотрим теперь как пишутся вершинные шейдеры. Следующий код представляет вершинный шейдер:

```
@vertex

fn vertexMain(@location(0) coords: vec2f, @location(1) colors: vec3f)
-> @builtin(position) vec4f {

     return vec4f(coords, 0.0, 1.0);

}
```

Эта функция является точкой входа для вершинного шейдера, поскольку её имени предшествует `@vertex`. Она принимает два параметра (`vec2f` и `vec3f`) и возвращает вектор.

В этом коде необходимо пояснить два момента: атрибут `@location`, предшествующий обоим параметрам, и атрибут `@builtin`, предшествующий возвращаемому значению.

Атрибут `@location`

Атрибут `@location` позволяет шейдерам получать доступ к данным в ресурсах (буферах и текстурах). В приведённом выше коде этот атрибут предшествует аргументам шейдера: координатам и цветам.

Вершинный шейдер может получить доступ к данным в вершинном буфере через параметры своей функции - точки входа. Каждый параметр должен иметь атрибут `@location(n)`, где `n` соответствует полю `shaderLocation` в объекте разметки.

Рассмотрим код определяющий объект разметки для буфера вершин, содержащий координаты и цвета:

```
// Create a layout object for the vertex buffer
const bufferLayout = {
      arrayStride: 20,
      attributes: [
            { format: "float32x2", offset: 0, shaderLocation: 0 },
            { format: "float32x3", offset: 8, shaderLocation: 1 }
      ],
};
```

Первый элемент массива `attributes` имеет `shaderLocation`, равный 0. В результате шейдер может получить к нему доступ через параметр, которому предшествует `@location(0)`. Поскольку формат элемента — `float32x2`, тип параметра должен быть `vec2f`.

Второй элемент массива атрибутов имеет `shaderLocation`, равный 1. Шейдер может получить к нему доступ через параметр, которому предшествует `@location(1)`. Поскольку формат элемента — `float32x3`, тип параметра должен быть `vec3f`.

Атрибут `@builtin`

Вершинный шейдер выполняется один раз для каждой обрабатываемой вершины. Во время каждого выполнения вершинный шейдер может обращаться к трём специальным переменным, называемым встроенными переменными:

Встроенная переменная	Чтение/запись	Тип данных	Описание
`position`	запись	`vec4f`	Координаты вершины
`vertex_index`	чтение	`u32`	Индекс вершины
`instance_index`	чтение	`u32`	Индекс экземпляра

Переменная `position` особенно важна, поскольку вершинному шейдеру необходимо как минимум задать позицию каждой вершины для рендеринга. По этой причине шейдеры WebGPU возвращают либо переменную `position`, либо структуру, содержащую переменную `position` в качестве первого поля.

Чтобы сообщить WebGPU, что к переменной следует обращаться как к встроенной, необходимо объявить её с помощью `@builtin(varname)`, где `varname` — имя встроенной переменной.

Переменная `vertex_index` становится важной, когда шейдеру необходимо обрабатывать вершины по-разному в зависимости от их порядка. Аналогично, переменная `instance_index` полезна для приложений, которые рендерят несколько копий одной фигуры.

Фрагментные шейдеры

Фрагментный шейдер (fragment shader) – это программа, реализующая обработку фрагментов изображений, например, определение освещенности, где под фрагментом подразумевается простейший элемент изображения, своего рода пиксель. Фрагмент, по сути, похож на пиксель, но у пикселя есть цвет и координаты, а у фрагмента — цвет, координаты и глубина.

Фрагментные и вершинные шейдеры имеют много общего: оба пишутся на языке WGSL и могут обращаться к таким атрибутам, как `@location` и `@builtin`. Но в то время, как основная роль

вершинного шейдера заключается в определении координат вершин, основная роль фрагментного шейдера заключается в назначении цвета и глубины фрагментам.

Фрагментный шейдер выполняется один раз для каждого обрабатываемого фрагмента. Во время выполнения он может обращаться к ряду встроенных переменных:

Встроенная переменная	Чтение/запись	Тип данных	Описание
`position`	чтение	`vec4f`	Координаты фрагмента
`frag_depth`	запись	`f32`	Глубина фрагмента
`front_facing`	чтение	`bool`	`true`, если фрагмент находится на передней поверхности
`sample_index`	чтение	`u32`	Индекс сэмпла
`sample_mask`	чтение/запись	`u32`	Указывает, какие сэмплы охватываются текущим примитивом

Первая переменная, `position`, может вызывать путаницу, поскольку имеет то же имя и тип, что и переменная `position`, используемая в вершинных шейдерах. Но во фрагментном шейдере положение фрагмента является входной переменной, которую нельзя изменить. Более того, её содержимое заметно отличается:

- Первые два числа с плавающей точкой в `position` определяют положение фрагмента в двумерном пространстве. Чтобы отличить это пространство от пространства вершин, координаты обозначаются (u, v).
- Третье число в `position` определяет входную глубину фрагмента.
- Четвёртое число с плавающей точкой в `position` определяет перспективный делитель, который используется в перспективных преобразованиях.

Следующая переменная в таблице, `frag_depth`, — это выходное значение, содержащее глубину фрагмента, которая находится в диапазоне от 0,0 до 1,0. Если фрагментный шейдер не присваивает этой переменной значение, оно будет установлено равным третьему числу в переменной `position`.

Переменная `front_facing` принимает значение `true`, если текущий фрагмент лежит на фронтальном примитиве, и `false` в противном случае.

Последние две переменные в таблице относятся к сэмплированию — процессу, посредством которого шейдер извлекает данные из текстуры.

Фрагментный шейдер отвечает как минимум за установку цвета фрагмента, поэтому может показаться странным, что ни одна из переменных в таблице не относится к цвету. Причина проста: каждый фрагментный шейдер должен возвращать `vec4f`, содержащий компоненты цвета фрагмента (красный, зелёный, синий и альфа-канал). Благодаря этому требованию нет необходимости назначать имя возвращаемому значению.