Лабораторная работа №3

«Работа с цветом и фрагментные шейдеры»

Оглавление

Рисование градиентного треугольника

Создадим массивы, содержащий данные, необходимые для рисования треугольника с вершинами разных цветов. Координаты точек треугольника будут равны: (0, 0,5), (−0,5, −0,5) и (0,5, −0,5), а цвета будут следующие: зелёный (0,0, 1,0, 0,0), красный (1,0, 0,0, 0,0) и синий (0,0, 0,0, 1,0).

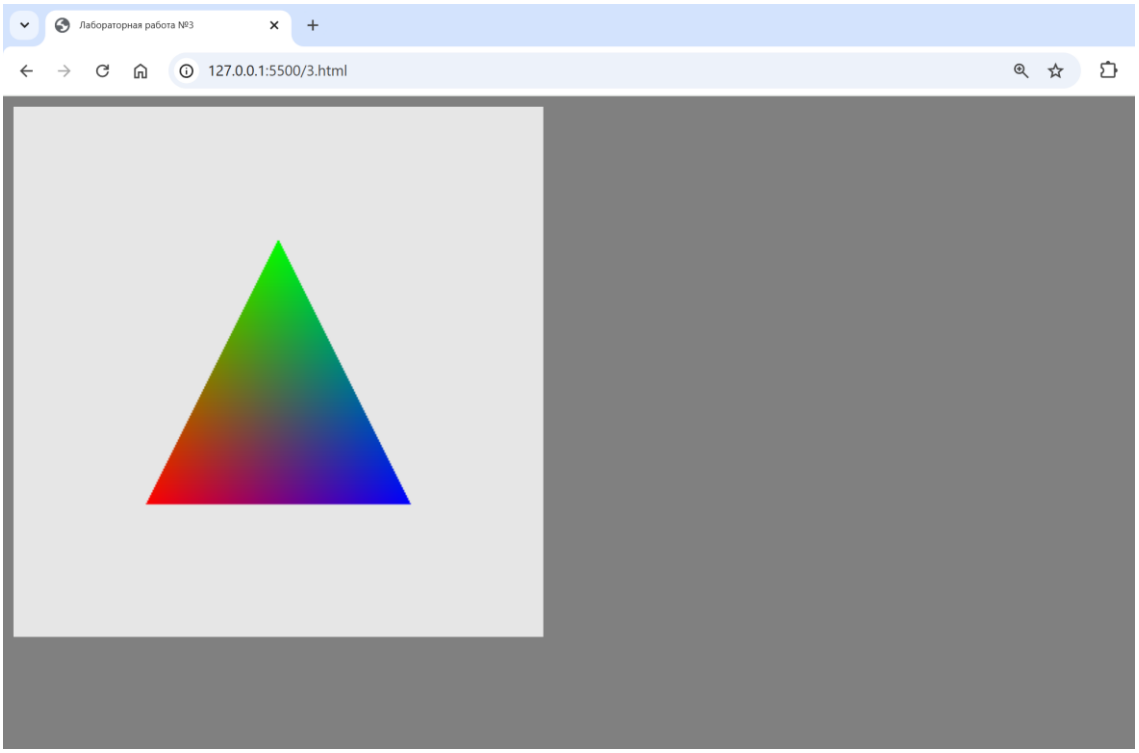

Рис. 1. Градиентный треугольник

Следующий код показывает, как эти данные можно сохранить в массивах 32-битных значений с плавающей запятой:

```
const vertexCoords = new Float32Array([0.0, 0.5,
                                       -0.5, -0.5,
                                       0.5, -0.5]);
```

```
const vertexColors = new Float32Array([0.0, 1.0, 0.0,
                                       1.0, 0.0, 0.0,
                                       0.0, 0.0, 1.0]);
```

Для эффективности объединим координаты и цвета в один большой массив:

```
const vertexData = new Float32Array([
                        0.0, 0.5, 0.0, 1.0, 0.0, // First vertex
                        -0.5, -0.5, 1.0, 0.0, 0.0, // Second vertex
                        0.5, -0.5, 0.0, 0.0, 1.0 // Third vertex
]);
```

Этот массив содержит координаты первой вершины, за которыми следует цвет первой вершины. Далее следуют данные атрибутов второй и третьей вершин. Поскольку данные для каждой вершины объединяются, такой метод хранения данных называется чередованием. В целом, это обеспечивает лучшую производительность, чем использование отдельных массивов.

Передача данных во фрагментный шейдер

Итак, в данном примере мы хотим назначить разные цвета каждой вершине треугольника. Поскольку цвета меняются от вершины к вершине, компоненты цвета должны храниться в вершинном буфере. Но есть проблема. В отличие от вершинных шейдеров, фрагментные шейдеры не могут напрямую обращаться к буферам вершин.

Однако фрагментный шейдер может получить доступ к возвращаемому значению вершинного шейдера. В предыдущей лабораторной работе, вершинный шейдер возвращал только координаты обрабатываемой вершины. Но мы можем расширить это возвращаемое значение, создав структуру. То есть, мы можем определить структуру, содержащую положение вершины и данные о цвете, которые будут переданы фрагментному шейдеру.

Следующий код определяет структуру с именем `DataStruct`, которая содержит два поля. Первое поле содержит координаты вершины, а второе — данные о цвете, которые будут использоваться фрагментным шейдером.

```
// Structure containing vertex coordinates and color
struct DataStruct {
     @builtin(position) pos: vec4f,
     @location(0) colors: vec3f,
}
```

Чтобы передать эту структуру из вершинного во фрагментный шейдер, тип возвращаемого вершинным шейдером значения должен быть установлен как `DataStruct`. Затем вершинный шейдер должен инициализировать `DataStruct`, задать его поля и вернуть структуру. Следующий код показывает, как это можно сделать:

```
// Pass a structure as the return value

@vertex

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

      var outData: DataStruct;

      outData.pos = vec4f(coords, 0.0, 1.0);

      outData.colors = colors;

      return outData;

}
```

Вершинный шейдер создаёт переменную `DataStruct` с именем `outData`, а затем устанавливает её поле `pos` в `vec4f`, содержащее координаты текущей вершины. Поле `colors` устанавливается в `vec3f` и содержит цвет текущей вершины.

Поскольку вершинный шейдер возвращает `DataStruct`, фрагментный шейдер может получить доступ к этой структуре как к аргументу своей функции - точки входа. Он игнорирует поле `pos` и обращается к полю `colors`, которое использует для установки цвета текущего фрагмента. Этот цвет, как и координаты вершины, должен быть представлен в виде вектора, содержащего четыре значения с плавающей точкой (`vec4f`).Следующий код показывает, как это можно сделать:

```
// Receive the structure and access colors

@fragment

fn fragmentMain(fragData: DataStruct) -> @location(0) vec4f {

      return vec4f(fragData.colors, 1.0);

}
```

Код лабораторной работы №3 практически идентичен коду лабораторной работы №2, и оба они создают, по сути, одни и те же объекты. Однако код лабораторной работы №3 отличается в трёх отношениях:

1. Буфер вершин содержит данные о цвете для каждой вершины, а также координаты вершин.
2. Вершинный шейдер возвращает структуру данных, содержащую данные о цвете.
3. Фрагментный шейдер получает структуру данных и использует данные о цвете для назначения цветов.

Буфер вершин

Следующий код записывает данные в буфер вершин.

```
// Create vertex buffer
const vertexBuffer = device.createBuffer({
      label: "Example vertex buffer",
      size: vertexData.byteLength,
      usage: GPUBufferUsage.VERTEX |
             GPUBufferUsage.COPY_DST
});
// Write data to buffer
device.queue.writeBuffer(vertexBuffer, 0, vertexData);
renderPass.setVertexBuffer(0, vertexBuffer);
// Define layout of buffer data
const bufferLayout = {
      arrayStride: 20,
      attributes: [
            { format: "float32x2", offset: 0, shaderLocation: 0 },
            { format: "float32x3", offset: 8, shaderLocation: 1 }
      ],
};
```

Здесь приложение создаёт буфер вершин и записывает данные вершин в буфер, вызывая метод `writeBuffer` объекта `GPUQueue`. Затем оно связывает буфер с объектом `GPURenderPassEncoder`, вызывая метод `setVertexBuffer` объекта `GPURenderPassEncoder`.

Последняя часть кода определяет структуру буфера вершин. В этом случае каждая вершина содержит пять четырёхбайтовых чисел с плавающей точкой, поэтому `arrayStride` равен 5 * 4 = 20. Вершинному шейдеру требуется два значения на вершину, поэтому массив `attributes` содержит два элемента. Первый элемент определяет координаты (`float32x2`), а второй — компоненты цвета (`float32x3`). Значение `offset` первого атрибута равно 0, поскольку он начинается с начала. Значение `offset` второго элемента равно `8`, поскольку он начинается с 8-ого байта от начала данных. Значение `shaderLocation` первого атрибута равно `0`, поэтому вершинный шейдер может получить к нему доступ с помощью `@location(0)`, а значение `shaderLocation` второго атрибута равно `1`, поэтому вершинный шейдер может получить к нему доступ с помощью `@location(1)`.

Задание для самостоятельной работы

- Измените порядок хранения информации в массиве `vertexData` – пусть сначала в нем содержится информация о цвете точки, а затем о ее координатах. Настройте правильно параметры свойства `attributes`. Продемонстрируйте результат работы программы.
- Установите свойство `alphaMode = "premultiplied"`. Нарисуйте два накладывающихся прямоугольника, имеющих разные цвета (все вершины каждого прямоугольника имеют одинаковые цвета и прозрачность). Визуализируйте наложение полупрозрачного прямоугольника на непрозрачный прямоугольник.