Программирование графических приложений

Тема 3
Шейдеры и работа с цветом вершин в WebGL

Цель изучения темы. Изучение способов применения шейдеров для формирования моделей графических объектов, изучение средств работы с цветом в WebGL.

3.1. Вершинный и фрагментный шейдеры

Шейдеры являются одним из базовых элементов любой программы на WebGL. На вход в графический процессор передается лишь набор вершин. Благодаря вершинному и фрагментному шейдерам этот набор сначала превращается в набор примтивов, а затем окрашивается, и мы в итоге видим трехмерные модели.

Если мы посмотрим на любой ранее использовавшийся шейдер (все они пока были однотипны), то увидим, что в них применяется особый язык с Си-подобным синтаксисом.

Это язык шейдеров - OpenGL ES Shading Language (GLSL), который основан на C++. Этот язык используется в двух частях программы WebGL - в вершинном и фрагментном шейдерах. Для написания кода на GLSL используются выражения языка JavaScript.

Вершинный шейдер

Практически в самом начале работы конвейера WebGL буфер вершин передается в вершинный шейдер. Вершинный шейдер сканирует все переданные вершины и выполняет преобразования, которые определил в программе вершинного шейдера разработчик. Именно вершинный шейдер отвечает за матричные преобразования координат, их смещения и т.д. На выходе он генерирует финальные координаты каждой вершины и передает их для дальнейшей обработки. До сих пор было использовано простейшее определение вершинного шейдера:

```
attribute vec3 aVertexPosition;
void main (void) {  gl_Position = vec4(aVertexPosition, 1.0); }
```

Атрибут **aVertexPosition**, имеющий тип vec3, как раз и передает координаты вершины из буфера вершин. И так как каждая вершина представлена тремя координатами x, y, z, то для передачи вершины используется трехмерный вектор vec3.

Как и каждая программа на C/C++, шейдер имеет основную процедуру main, в которой происходит генерация результирующих координат вершин, только уже в виде четырехмерного вектора, а gl_Position - это и есть преобразованные координаты вершины. В данном случае никаких преобразований не производится, и финальная вершина будет иметь те же координаты, что и переданная в шейдер.

Фрагментный шейдер

Фрагментный шейдер наполняет набор примитивов цветом. Он никак не влияет на координаты вершин, только на цветовую составляющую, преобразуя вершины в пиксели или фрагменты.

Ранее также были использованы простейшие определения фрагментных шейдеров, например:

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

Здесь также имеется основная программа main, которая устанавливает цвет для пикселя, представленного вершиной. Так как представление цвета состоит из четырех элементов - RGBA, то для установки цвета используется тип четырехэлементного вектора vec4. В данном случае каждый пиксель примитивов окрашивается в белый цвет. Если бы мы захотели окрасить примитивы в зеленый цвет, следовало задать следующее определение цвета:

```
gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0).
```

3.2. Использование шейдеров в программе

Рассмотрим использование шейдеров в программе и для примера возьмем небольшое веб-приложение, которое рисует набор линий (**ex03_01.html**) без использования дополнительных библиотек:

```
<!DOCTYPE html>
<html>
<head>
<title>Шейдеры</title>
<meta content="charset=utf-8">
</head>
<body>
<canvas id="canvas3D" width="400" height="300">Ваш браузер не поддерживает
        элемент canvas</canvas>
<!-- фрагментный шейдер -->
<script id="shader-fs" type="x-shader/x-fragment">
 void main(void) { gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0); }
</script>
<!-- вершинный шейдер -->
<script id="shader-vs" type="x-shader/x-vertex">
 attribute vec3 aVertexPosition;
 void main(void) {  gl_Position = vec4(aVertexPosition, 1.0);  }
</script>
<script type="text/javascript">
var gl;
var shaderProgram; // программа шейдеров
var vertexBuffer; // буфер вершин
var indexBuffer; //буфер индексов
// установка шейдеров
function initShaders() {
   // получаем скрипты вершинного и фрагментного шейдеров
   var fragmentShader = getShader(gl.FRAGMENT_SHADER, 'shader-fs');
   var vertexShader = getShader(gl.VERTEX_SHADER, 'shader-vs');
   // создаем программу шейдеров
   shaderProgram = gl.createProgram();
   // прикрепляем к этой программе шейдеры
   gl.attachShader(shaderProgram, vertexShader);
   gl.attachShader(shaderProgram, fragmentShader);
   //связываем контекст WebGL с программой шейдеров
   gl.linkProgram(shaderProgram);
   if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
      alert("Не удалось установить шейдеры");
   }
    // начинаем использовать программу шейдеров
   gl.useProgram(shaderProgram);
   // установка атрибута позиции вершин
   shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
         "aVertexPosition");
   gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
}
// Функция создания шейдера - запускается для обоих шейдеров
function getShader(type,id) {
```

```
    // получаем текст программы
    var source = document.getElementById(id).innerHTML;
    // создаем шейдер
    var shader = gl.createShader(type);
    // связываем шейдер с текстом
    gl.shaderSource(shader, source);
    // компилируем шейдер
    gl.compileShader(shader);
    // если все скомпилировалось, возвращаем из функции шейдер
    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        alert("Ошибка компиляции шейдера: " + gl.getShaderInfoLog(shader));
        gl.deleteShader(shader);
        return null;
    }
    return shader;
}
 // установка буферов вершин и индексов
function initBuffers() {

    vertices =[ -0.5, -0.5, 0.0,
                -0.5, 0.5, 0.0,
                 0.0, 0.0, 0.0,
                 0.5, 0.5, 0.0,
                 0.5, -0.5, 0.0];

    indices = [0, 1, 1, 2, 2, 3, 3, 4];
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  vertexBuffer.itemSize = 3;
  indexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
            gl.STATIC_DRAW);
    indexBuffer.numberOfItems = indices.length;
}
function draw() {
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                        vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
    gl.drawElements(gl.LINES, indexBuffer.numberOfItems, gl.UNSIGNED_SHORT,0);
}
window.onload=function(){
    var canvas = document.getElementById("canvas3D");
    try { gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl"); }
    catch(e) {} if (!gl) { alert("Ваш браузер не поддерживает WebGL"); }
    if(gl){
        gl.viewportWidth = canvas.width;
        gl.viewportHeight = canvas.height;
        initShaders();
```

```
        initBuffers();
        draw();
    }
}
</script>
</body>
</html>
```

Результат:

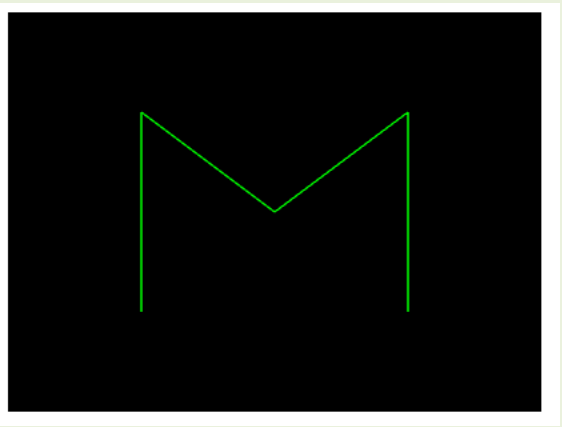

Рис. 3.1

Здесь нас интересуют две функции: initShaders() и getShader(). Поскольку создание обоих шейдеров предусматривает однотипные операции, эти операции были вынесены в отдельную функцию getShader(). В данную функцию передаются два параметра: type (тип шейдера) и id элемента, который содержит код шейдера. Поскольку метод gl.createShader() может создавать шейдеры обоих типов, в этот метод в качестве параметра и передаем тип создаваемого шейдера. В качестве типа используем встроенные константы gl.FRAGMENT_SHADER и gl.VERTEX_SHADER.

В функции initShaders() создаются оба шейдера, инициализируется программа шейдеров shaderProgram, затем устанавливается атрибут. Затем уже в функции отрисовки мы используем ранее созданные шейдеры.

3.3. Синтаксис GLSL

Рассмотрим некоторые базовые моменты синтаксиса GLSL.

Определение типов

В GLSL определены следующие типы:

- void: функция не возвращает никакого значения;
- bool: логическое значение true или false;
- int: целочисленное значение;
- float: числовое знаение с плавающей точкой;
- vec2, vec3, vec4: двух-, трех- и четырехмерные векторы соответственно, которые содержат объекты типа float;
- ivec2, ivec3, ivec4: двух-, трех- и четырехмерные векторы соответственно, которые содержат объекты типа int;
- bvec2, bvec3, bvec4: двух-, трех- и четырехмерные векторы соответственно, которые содержат объекты типа bool;
- mat2, mat3, mat4: матрицы размера 2x2б 3x3 и 4x4 соответственно, которые содержат объекты типа float;

• sampler2D, samplerCube: специальные типы - семплеры для работы с текстурами; с их помощью во фрагментном шейдере мы можем получить цветовые значения текстур и передать их примитиву.

В GLSL можно создавать сложные типы - структуры, которые содержат наборы примитивных типов:

```
struct someStruct { int someInt; vec4 someVec; }
```

В GLSL мы можем не просто объявить переменную, но и добавить к ней определенное поведение. Это делается с помощью модификаторов.

Используются следующие **модификаторы**:

• attribute: атрибут или часть описания вершины, которое передается из программы на WebGL в вершинный шейдер;

• const: константы; эти переменные определяют свое значение только один раз и в процессе программы его уже не меняют;

• uniform: переменные с константными значениями, только эти значения задаются для всего примитива;

• varying: переменная, которая задается в вершинном шейдере и затем передается во фрагментный шейдер, где может быть использована.

В приведенных ранее примерах мы уже использовали модификатор в вершинном шейдере: attribute vec3 aVertexPosition. Здесь aVertexPosition представляет описание вершины и имеет тип трехмерного вектора, поскольку мы определяем вершину в трехмерном пространстве.

Модификаторы для чисел с плавающей точкой

Мы также можем использовать модификаторы для чисел с плавающей точкой - то есть для переменных типа float:

• highp: число с плавающей точкой сохраняет максимальную точность;

• mediump: число со средней степенью точности;

• lowp: диапазон плавающей точки от -2 до 2.

Например, varying highp vec4 vColor - каждое число float в векторе vec4 имеет высокую точность.

Встроенные глобальные переменные GLSL

Кроме задаваемых разработчиком переменных GLSL имеет также набор встроенных глобальных переменных, которые могут использоваться в вершинном или фрагментном шейдерах:

• gl_Position: переменная имеет тип vec4 и указывает на положение вершины: используется в вершинном шейдере в качестве выходного параметра;

• gl_PointSize: имеет тип float и содержит размер точки; используется в вершинном шейдере в качестве выходного параметра;

• gl_FragCoord: имеет тип vec4 и указывает на положение фрагмента в буфере фреймов; используется во фрагментном шейдере в качестве входного параметра;

• gl_FontFacing: имеет тип bool и определяет, принадлежит ли фрагмент лицевому примитиву; используется во фрагментном шейдере в качестве входного параметра;

• gl_PointCoord: имеет тип vec2 и указывает на позицию фрагмента внутри точки; используется во фрагментном шейдере в качестве входного параметра;

• gl_FragColor: имеет тип vec4 и указывает на итоговый цвет фрагмента; используется во фрагментном шейдере в качестве выходного параметра;

• gl_FragData[n]: имеет тип vec4 и указывает на цвет фрагмента для прикрепления цвета n; используется во фрагментном шейдере в качестве выходного параметра.

Например, в вершинном шейдере мы определяем переменную gl_Position для установки координат вершины:

```
attribute vec3 aVertexPosition;
void main(void) { gl_Position = vec4(aVertexPosition, 1.0); }
```

Мы можем, например, добавить сжатие по оси X, разделив значение координаты X вершины на какой-нибудь коэффициент:

```
attribute vec3 aVertexPosition;
const float k=2.0;
 void main(void) {
   float x = aVertexPosition.x / k;
   gl_Position = vec4(x, aVertexPosition.y, aVertexPosition.z, 1.0);
}
```

В итоге фигура сожмется в два раза, так как задан коэффициент 2.0. Переменная *aVertexPosition* представляет вектор {x, y, z}, поэтому мы можем обратиться к каждой составляющей отдельно: *aVertexPosition.x/k* и затем установить все составляющие вектора *gl_Position*.

Втроенные функции

GLSL имеет ряд встроенных функций, которые мы впоследствии будем использовать в программах. Некоторые из них:

- dot(x, y): возвращает скалярное произведение векторов x и y;
- cross(x, y): возвращает векторное произведение векторов x и y;
- matrixCompMult(mat x, mat y): возвращает произведение матриц x и y, которые должны быть одной размерности;
- normalize(x): возвращает нормализованный вектор x, то есть такой вектор, у которого длина равна 1;
- reflect(t, n): отражает вектор t вдоль вектора n;
- sin(angle): возвращает синус угла angle;
- cos(angle): возвращает косинус угла angle;
- pow(x, y): возвращает x в степени y;
- max(x, y): возвращает максимальное из двух значений;
- min(x, y): возвращает минимальное из двух значений.

3.4. Установка цвета вершин

Чтобы установить для каждой вершины цвет, необходимо задать буфер цветов для вершины, а также определить дополнительный атрибут, который будет передавать данные цветов в шейдеры. Код веб-странички будет следующим (**ex03_02.html**):

```
<!DOCTYPE html>
<html>
<head>
<title>Установка цвета вершин</title>
<meta content="charset=utf-8">
</head>
<body>
<canvas id="canvas3D" width="400" height="300">Ваш браузер не поддерживает
        элемент canvas</canvas>
<script id="shader-fs" type="x-shader/x-fragment">
```

```
 varying highp vec4 vColor;
 void main(void) {  gl_FragColor = vColor;  }
</script>
<script id="shader-vs" type="x-shader/x-vertex">
 attribute highp vec3 aVertexPosition;
 attribute vec3 aVertexColor;
 varying highp vec4 vColor;
 void main(void) {
   gl_Position = vec4(aVertexPosition, 1.0);
   vColor = vec4(aVertexColor, 1.0);
}
</script>
<script type="text/javascript">
var gl;
var shaderProgram;
var vertexBuffer; // буфер вершин
var colorBuffer; //буфер цветов
// установка шейдеров
function initShaders() {
 var fragmentShader = getShader(gl.FRAGMENT_SHADER, 'shader-fs');
 var vertexShader = getShader(gl.VERTEX_SHADER, 'shader-vs');
 shaderProgram = gl.createProgram();
 gl.attachShader(shaderProgram, vertexShader);
 gl.attachShader(shaderProgram, fragmentShader);
 gl.linkProgram(shaderProgram);
 if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
alert("Не удалсь установить шейдеры");
   }
 gl.useProgram(shaderProgram);
 shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
          "aVertexPosition");
 gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
 shaderProgram.vertexColorAttribute = gl.getAttribLocation(shaderProgram,
          "aVertexColor");
 gl.enableVertexAttribArray(shaderProgram.vertexColorAttribute);
}
// Функция создания шейдера
function getShader(type,id) {
 var source = document.getElementById(id).innerHTML;
 var shader = gl.createShader(type);
 gl.shaderSource(shader, source);
 gl.compileShader(shader);
 if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
 alert("Ошибка компиляции шейдера: " + gl.getShaderInfoLog(shader));
 gl.deleteShader(shader);
 return null;
 }
 return shader;
}
 // установка буферов вершин и индексов
function initBuffers() {
 var vertices = [
```

```
        0.0,  0.5,  0.0,
       -0.5, -0.5,  0.0,
        0.5, -0.5,  0.0
 ];
 // установка буфера вершин
 vertexBuffer = gl.createBuffer();
 gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
 gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
 vertexBuffer.itemSize = 3;
 vertexBuffer.numberOfItems = 3;
 var colors = [
      1.0, 0.0, 0.0,
      0.0, 0.0, 1.0,
      0.0, 1.0, 0.0
 ];
 colorBuffer = gl.createBuffer();
 gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
 gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
}
 // отрисовка
function draw() {
  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  gl.clear(gl.COLOR_BUFFER_BIT);
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
  gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
               vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
  gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);
}
window.onload=function(){
  var canvas = document.getElementById("canvas3D");
  try { gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl"); }
  catch(e) {} if (!gl) { alert("Ваш браузер не поддерживает WebGL"); }
  if(gl){
 gl.viewportWidth = canvas.width;
 gl.viewportHeight = canvas.height;
 initShaders();
 initBuffers();
 draw();
  }
}
</script>
</body>
</html>
```

После запуска программы у нас получится такой треугольник:

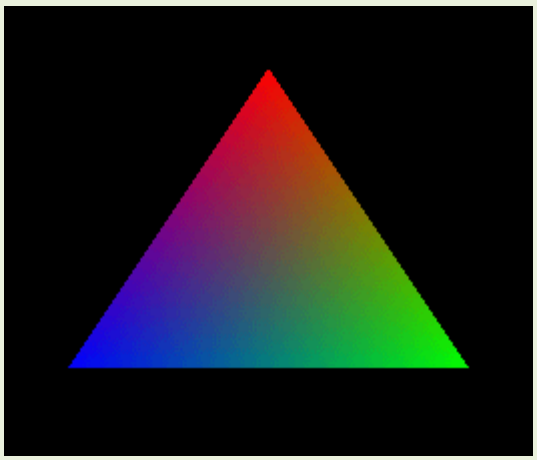

Рис. 3.2

Рассмотрим основную программу WebGL, а потом шейдеры.

Поскольку у нас в шейдеры будут передаваться наборы из двух буферов (цвета и вершин) сначала мы создаем два атрибута (для каждого типа):

```
shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
        "aVertexPosition");
gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
shaderProgram.vertexColorAttribute = gl.getAttribLocation(shaderProgram,
        "aVertexColor");
gl.enableVertexAttribArray(shaderProgram.vertexColorAttribute);
```

Затем в функции *initBuffers()* задаем два буфера. Буфера цветов во многом похож на буфер вершин:

```
var colors = [ 1.0, 0.0, 0.0,  0.0, 0.0, 1.0,  0.0, 1.0, 0.0 ];
colorBuffer = gl.createBuffer();
gl.bindBuffer (gl.ARRAY_BUFFER, colorBuffer);
gl.bufferData (gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
```

Значения 1.0, 0.0, 0.0 передают цвет первой вершины, в данном случае это красный цвет. Следующая тройка 0.0, 0.0, 1.0, окрашивает вторую вершину в синий цвет, а третья тройка - в зеленый.

Если при определении буфера вершин каждая тройка обозначает соответственно координаты x, y, z, то при определении буфера цветов три компоненты отвечают за значения rgb - то есть цветовые компоненты.

Далее похожим образом создаем буфер цветов и связываем с контекстом WebGL. Затем в функции отрисовки *draw()* у нас возникает необходимость установить два атрибута - для передачи вершин и цветов. Поэтому мы производим установку последовательно: сначала для одного, потом для другого. А общий принцип одинаков для всех. Теперь перейдем к шейдерам, где эти атрибуты используются.

В вершинном шейдере мы получаем атрибуты, установленные в основной программе WebGL:

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexColor;
```

Следующая переменная varying highp vec4 vColor; будет передавать цвет из вершинного шейдера во фрагментный, поэтому для нее используется модификатор varying.

Затем в самой программе устанавливаются значения:

```
gl_Position = vec4 (aVertexPosition, 1.0);
vColor = vec4 (aVertexColor, 1.0);
```

Во фрагментном шейдере мы используем для установки окончательного цвета именно цвет, переданный через переменную vColor.

Контрольные вопросы

1. Шейдеры в WebGL.
2. Синтаксис GLSL.
3. Установка цвета вершин в WebGL.