Программирование графических приложений

Тема 10
Текстурирование

Цель изучения темы. Изучение методов текстурирования объектов и сцен в WebGL.

10.1. Текстурирование в 2D

В предыдущих темах использовалась окраска трехмерных объектов. Однако при имитации объектов реального мира сложно подобрать цвета и произвести раскраску таким образом, чтобы объект выглядел так же, как и в реальности. Гораздо проще воспользоваться текстурированием, то есть нанести на поверхности объектов изображения, которые помогут имитировать реальность.

Для начала рассмотрим процесс текстурирования без использования библиотек на примере двухмерного объекта - прямоугольника. Полный код примера (**ex10_01.html**):

```
<!DOCTYPE html>
<html>
<head>
<title>Текстурирование 2D</title>
<meta 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">
precision highp float;
uniform sampler2D uSampler;
varying vec2 vTextureCoords;
 void main(void) {  gl_FragColor = texture2D(uSampler, vTextureCoords);  }
</script>
<script id="shader-vs" type="x-shader/x-vertex">
attribute vec3 aVertexPosition;
varying vec2 vTextureCoords;
 void main(void) {
   gl_Position = vec4(aVertexPosition, 1.0);
   vTextureCoords = vec2(aVertexPosition.x+0.5,aVertexPosition.y+0.5);
 }
</script>
<script type="text/javascript">
var gl;
var shaderProgram;
var vertexBuffer;
var indexBuffer;
var texture; // переменная для хранения текстуры
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 удалось установить шейдеры");
   }
   gl.useProgram(shaderProgram);
```

```
    shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
            "aVertexPosition");
    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
}
// 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;
}
function initBuffers() {
    var vertices =[
                -0.5, -0.5, 0.5,
                -0.5, 0.5, 0.5,
                 0.5, 0.5, 0.5,
                 0.5, -0.5, 0.5
                 ];
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  vertexBuffer.itemSize = 3;
    var indices = [0, 1, 2, 2, 3, 0];
    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.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                    vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
    gl.drawElements(gl.TRIANGLES, indexBuffer.numberOfItems,
            gl.UNSIGNED_SHORT,0);
}
function setupWebGL()
{
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT || gl.DEPTH_BUFFER_BIT);
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
}
function setTextures(){
    texture = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, texture);
    var image = new Image();
```

```
   image.onload = function() {
      handleTextureLoaded(image, texture);
      setupWebGL();
      draw();
   }
   image.src = "brickwall.png";
   shaderProgram.samplerUniform = gl.getUniformLocation(shaderProgram, "uSampler");
   gl.uniform1i(shaderProgram.samplerUniform, 0);
}
 function handleTextureLoaded(image, texture) {
   gl.bindTexture(gl.TEXTURE_2D, texture);
   gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
   gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
            image);
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
}
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();
      setTextures();
   }
}
</script>
</body>
</html>
```

В данном случае в качестве текстуры использовано изображение кирпичной стены **brickwall.png**, поэтому получим следующий результат:

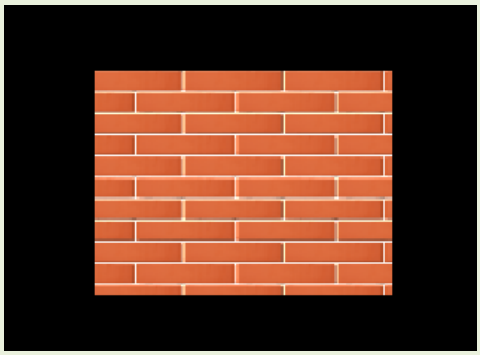

Рис. 10.1

Для хранения загруженной структуры в программе создается глобальная переменная texture.

Элементы большей часть программы встречались в предыдущих темах. Основное отличие от них - функция setTextures(), которая будет вызываться в главной функции вместо метода отрисовки draw(). Поэтому остановимся на функции setTextures().

Для начала мы создаем текстуру: texture = gl.createTexture();

Загрузка текстуры

Для установки изображения в качестве текстуры нам нужен элемент img. Изображение, установленное для данного элемента, и будет устанавливаться в качестве текстуры.

Тут есть два способа. Первый: мы можем заранее определить в структуре веб-страницы элемент img и с помощью его атрибута src установить какое-либо изображение. Второй: мы можем динамически в коде javascript создать этот элемент, как продемонстрировано в данном примере.

Поскольку текстура не сразу загружается, то используем обработку события onload:

```
image.onload = function() {
    handleTextureLoaded(image, texture);
    setupWebGL();
    draw();
}
image.src = "brickwall.png";
```

Также мы устанавливаем некоторую картинку. В нашем случае это изображение 128x128 brickwall.png. Сама установка текстуры происходит в функции

```
handleTextureLoaded(image, texture);.
```

Далее, когда все настройки текстурирования сделаны, происходит отрисовка.

Функция handleTextureLoaded() выполняет важную роль по настройке всех параметров текстурирования.

```
function handleTextureLoaded(image, texture) {
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
            image);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
}
```

Прежде чем перейти к использованию текстуры, ее надо связать с объектом текстуры texture:

```
gl.bindTexture(gl.TEXTURE_2D, texture);
```

Этот метод действует аналогично вызову gl.bindBuffer() при связывании буфера вершин.

Используемый далее метод gl.pixelStorei() указывает следующему за ним методу gl.texImage2D(), как текстура должна позиционироваться. Так, в нашем случае в качестве параметра передается значение gl.UNPACK_FLIP_Y_WEBGL - этот параметр указывает методу gl.texImage2D(), что изображение надо перевернуть относительно горизонтальной оси.

Поворот относительно горизонтальной оси необходим из-за того, что стандартный объект Image, который в данном случае выбран в качестве источника для поставки текстуры, имеет другую систему координат:

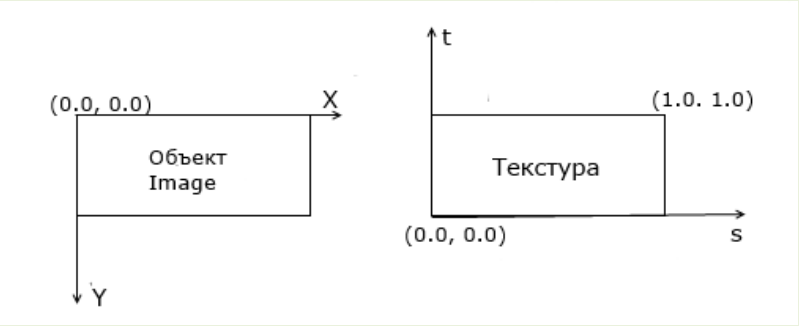

Рис. 10.2

Следовательно, нам надо совместить две координатные системы, чтобы в будущем было проще работать с текстурой.

Затем можно уже загрузить в текстуру изображение из элемента image с помощью метода gl.texImage2D.

Метод gl.texParameteri выполняет настройку параметров текстурирования. Первый вызов этого метода устанавливает значение для параметра gl.TEXTURE_MAG_FILTER - тем самым мы определяем рендеринг текстуры, если она будет увеличена. Второй вызов метода gl.texParameteri, наоборот, определяет поведение рендеринг текстуры, если она будет уменьшена.

Использование текстуры в шейдере

Чтобы использовать текстуру в шейдере и там ее применить к объекту, нам остается в конце функции setTextures установить семплер (инструмент для взятия «пробы» с изображения с целью последующего внесения полученного образца в другой фрагмент этого изображения или в другое изображение, чтобы перенести на него визуальные свойства образца):

```
shaderProgram.samplerUniform = gl.getUniformLocation (shaderProgram, "uSampler");
gl.uniform1i (shaderProgram.samplerUniform, 0);
```

Семплер будет использоваться для забора из текстуры текселей (пикселей на текстуре) и совмещения их с пикселями объекта на экране. В вершинном шейдере мы используем следующий код:

```
attribute vec3 aVertexPosition;
varying vec2 vTextureCoords;
 void main (void) {
   gl_Position = vec4 (aVertexPosition, 1.0);
   vTextureCoords = vec2 (aVertexPosition.x+0.5,aVertexPosition.y+0.5);
 }
```

Вектор vTextureCoords затем будет передан во фрагментный шейдер. Здесь же мы устанавливаем координаты:

```
vTextureCoords = vec2 (aVertexPosition.x+0.5,aVertexPosition.y+0.5);
```

Поскольку вершины реального объекта расположены в пределах от (-0.5, -0.5) до (0.5, 0.5) (так мы определили в нашем буфере вершин), то к координатам вершины, передаваемой через атрибут aVertexPosition, мы прибавляем 0.5. Таким образом, впоследствии мы сможем совместить объект с текстурой, у которой все точки находятся в пределах от (0.0, 0.0) до (1.0, 1.0). Более распространенный способ - использование координат самой текстуры.

Во фрагментном шейдере приводится в действие семплер:

```
precision highp float;
uniform sampler2D uSampler;
varying vec2 vTextureCoords;
 void main(void) {    gl_FragColor = texture2D(uSampler, vTextureCoords);   }
```

Суммируя, можно определить следующие этапы текстурирования:

1. Определение и создание объекта текстуры с помощью метода gl.createTexture().
2. Создание элемента html, который будет выступать источником для текстуры, например, Image, и установка его атрибута src.
3. Определение метода onload, который будет содержать логику, срабатывающую при загрузке изображения в элемент Image.
4. Привязка текстуры с помощью метода gl.bindTexture().
5. Переворачивание текстуры для совмещения ее с координатной системой объекта Image с помощью метода gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
6. Загрузка текстуры в GPU с помощью метода gl.texImage2D().
7. Установка параметров текстуры с помощью метода gl.texParameteri().

10.2. Работа с координатами текстуры

Используя координаты текстуры, мы можем более точно проецировать ее на поверхность объекта. Создадим следующий файл (**ex10_02.html**):

```
<!DOCTYPE html>
<html>
<head>
<title>Текстурирование по координатам 2D</title>
<meta 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">
precision highp float;
uniform sampler2D uSampler;
varying vec2 vTextureCoords;
 void main(void) {    gl_FragColor = texture2D(uSampler, vTextureCoords);  }
</script>
<script id="shader-vs" type="x-shader/x-vertex">
attribute vec3 aVertexPosition;
attribute vec2 aVertexTextureCoords;
varying vec2 vTextureCoords;

 void main(void) {
   gl_Position = vec4(aVertexPosition, 1.0);
   vTextureCoords = aVertexTextureCoords;
 }

</script>
<script type="text/javascript">
var gl;
var shaderProgram;
var vertexBuffer;
```

```
var indexBuffer;
var textureCoordsBuffer; // буфер координат текстуры
var texture; // переменная для хранения текстуры
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 удалось установить шейдеры");
   }
   gl.useProgram(shaderProgram);
   shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
         "aVertexPosition");
   gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
   shaderProgram.vertexTextureAttribute = gl.getAttribLocation(shaderProgram,
         "aVertexTextureCoords");
   gl.enableVertexAttribArray(shaderProgram.vertexTextureAttribute);
}
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;
}
function initBuffers() {
   var vertices =[
            -0.5, -0.5, 0.5, // v1
            -0.5, 0.5, 0.5,  // v2
             0.5, 0.5, 0.5,  // v3
             0.5, -0.5, 0.5  // v4
             ];
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  vertexBuffer.itemSize = 3;
   var indices = [0, 1, 2, 2, 3, 0];
   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;
   // Координаты текстуры
   var textureCoords = [
```

```
            0.0, 0.0,
            0.0, 1.0,
            1.0, 1.0,
            1.0, 0.0
             ];
   // Создание буфера координат текстуры
   textureCoordsBuffer = gl.createBuffer();
   gl.bindBuffer(gl.ARRAY_BUFFER, textureCoordsBuffer);
   gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoords),
         gl.STATIC_DRAW);
   textureCoordsBuffer.itemSize=2; // каждая вершина имеет две координаты
}
function draw() {
   gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
   gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                  vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
   gl.bindBuffer(gl.ARRAY_BUFFER, textureCoordsBuffer);
   gl.vertexAttribPointer(shaderProgram.vertexTextureAttribute,
                  textureCoordsBuffer.itemSize,  gl.FLOAT, false, 0, 0);
   gl.activeTexture(gl.TEXTURE0);
   gl.bindTexture(gl.TEXTURE_2D, texture);
   gl.drawElements(gl.TRIANGLES, indexBuffer.numberOfItems,
         gl.UNSIGNED_SHORT,0);
}
function setupWebGL()
{
   gl.clearColor(0.0, 0.0, 0.0, 1.0);
   gl.clear(gl.COLOR_BUFFER_BIT || gl.DEPTH_BUFFER_BIT);
   gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
}
function setTextures(){
   texture = gl.createTexture();
   gl.bindTexture(gl.TEXTURE_2D, texture);
   var image = new Image();
   image.onload = function() {
      handleTextureLoaded(image, texture);
  }
   image.src = "brickwall.png";
   shaderProgram.samplerUniform = gl.getUniformLocation(shaderProgram, "uSampler");
   gl.uniform1i(shaderProgram.samplerUniform, 0);
}
 function handleTextureLoaded(image, texture) {
   gl.bindTexture(gl.TEXTURE_2D, texture);
   gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
   gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
         image);
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
   gl.bindTexture(gl.TEXTURE_2D, null);
}
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();
        setTextures();
        (function animloop(){
            setupWebGL();
            draw();
            requestAnimFrame(animloop, canvas);
        })();
    }
}
window.requestAnimFrame = (function(){
     return  window.requestAnimationFrame       ||
             window.webkitRequestAnimationFrame ||
             window.mozRequestAnimationFrame    ||
             window.oRequestAnimationFrame      ||
             window.msRequestAnimationFrame     ||
         function(callback, element) {
           return window.setTimeout(callback, 1000/60);
         };
    })();
</script>
</body>
</html>
```

Данная программа будет иметь тот же эффект, что и пример из предыдущего раздела, только в данном случае сопоставление текстуры с объектом будет идти по координатам. В конце функции инициализации буферов есть такой код:

```
// Координаты текстуры
var textureCoords = [
          0.0, 0.0,
          0.0, 1.0,
          1.0, 1.0,
          1.0, 0.0
        ];
// Создание буфера координат текстуры
textureCoordsBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, textureCoordsBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoords),
            gl.STATIC_DRAW);
textureCoordsBuffer.itemSize=2; // каждая вершина имеет две координаты
```

Каждая объявленная в массиве textureCoords точка на текстуре соответствует вершине двухмерного объекта:

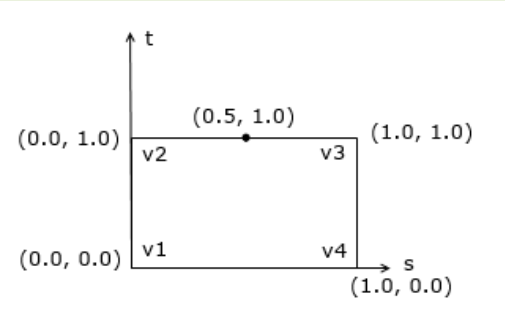

Рис. 10.3

Создание буфера координат текстуры аналогично созданию буфера вершин.

Функция setTextures(), которая выполняет загрузку и настройку текстуры, осталась та же, что и в прошлом примере за тем исключением, что теперь из нее вынесены методы setupWebGL() и draw() в функцию анимации.

После создания буфера координат текстуры надо его содержание передать в шейдер. Чтобы сделать это, сначала надо создать атрибут (это делается в функции настройки шейдеров initShaders):

```
shaderProgram.vertexTextureAttribute = gl.getAttribLocation(shaderProgram,
        "aVertexTextureCoords");
```

Далее в функции отрисовки draw подобно тому, как мы создаем указатель на атрибут для вершинного буфера, то же самое проделываем с буфером координат текстуры:

```
gl.bindBuffer(gl.ARRAY_BUFFER, textureCoordsBuffer);
 gl.vertexAttribPointer(shaderProgram.vertexTextureAttribute,
                textureCoordsBuffer.itemSize, gl.FLOAT, false, 0, 0);
 gl.activeTexture(gl.TEXTURE0);
 gl.bindTexture(gl.TEXTURE_2D, texture);
```

То есть также создаем указатель на атрибут, подключаем его, далее делаем активной текстуру (gl.TEXTURE0) и связываем ее. WebGL поддерживает работу с несколькими текстурами одновременно, а использование gl.TEXTURE0 отсылает нас к первой текстуре.

И в завершение надо получить в шейдере атрибут aVertexTextureCoords и его использовать. Для этого в вершинном шейдере заводим переменную для передачи значений текстуры во фрагментный шейдер, а данные она берет как раз из атрибута aVertexTextureCoords:

```
attribute vec3 aVertexPosition;
attribute vec2 aVertexTextureCoords;
varying vec2 vTextureCoords;
void main(void) {
   gl_Position = vec4(aVertexPosition, 1.0);
   vTextureCoords = aVertexTextureCoords;
 }
```

Во фрагментном шейдере также проходим семплером:

```
gl_FragColor = texture2D(uSampler, vTextureCoords);
```

10.3. Настройка текстурирования

Функция gl.texImage2D

Данный метод загружает текстуру в графический процессор на видеокарте. Он имеет следующий синтаксис:

texImage2D(target, level, internalformat, format, type, elem):

- target: указывает целевой объект для загрузки текстуры;
- level: уровень множественного отображения текстуры;
- internalformat и format: формат и внутренний формат; в WebGL должны иметь одно и то же значение (так, формат gl.RGBA, к примеру, показывает, что для каждого текселя на текстуре должны быть установлены цветовые каналы для красного, зеленого и синего цветов, а также альфа-канал);
- type: тип данных, которых сохраняет все данные текселей текстуры; например, gl.UNSIGNED_BYTE указывает, что для каждого цветового канала в gl.RGBA для сохранения данных выделяется один байт;
- elem: указывает на элемент, который содержит источник текстурирования; это может быть элемент img или Image (также это может быть элемент HTML5 video или canvas).

Все возможные сочетания форматов и типов:

Формат	**Тип**
gl.RGBA	gl.UNSIGNED_BYTE
gl.RGB	gl.UNSIGNED_BYTE
gl.RGBA	gl.UNSIGNED_SHORT_4_4_4_4
gl.RGBA	gl.UNSIGNED_SHORT_5_5_5_1
gl.RGB	gl.UNSIGNED_SHORT_5_6_5
gl.LUMINANCE_ALPHA	gl.UNSIGNED_BYTE
gl.LUMINANCE	gl.UNSIGNED_BYTE
gl.ALPHA	gl.UNSIGNED_BYTE

Формат gl.RGBA понятен: каждый тексель текстуры имеет канал красного, зеленого и синего цветов, а также альфа-канал. Формат gl.RGB - то же самое, только без альфа-канала.

Формат gl.LUMINANCE_ALPHA имеет канал яркости и альфа-канал. И формат gl.LUMINANCE имеет только канал яркости, а формат gl.ALPHA - только альфа-канал.

Например, настройка

```
gl.texImage2D (gl.TEXTURE_2D, 0, gl.LUMINANCE_ALPHA,
        gl.LUMINANCE_ALPHA, gl.UNSIGNED_BYTE, image);
```

даст следующий эффект:

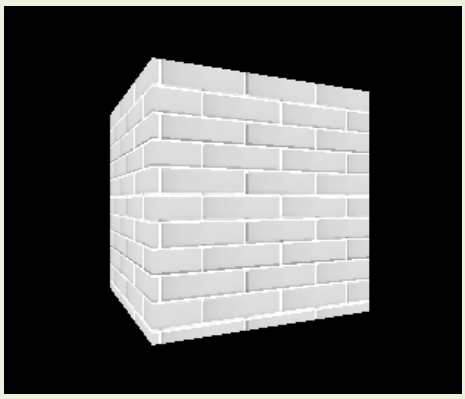

Рис. 10.4

Тип gl.UNSIGNED_BYTE предоставляет по одному байту на каждый канал.

Тип gl.UNSIGNED_SHORT_4_4_4_4 предоставляет для каждого канала в формате RGBA по четыре байта.

Тип gl.UNSIGNED_SHORT_5_5_5_1 предоставляет для каждого каналов красного, зеленого и синего цветов в формате RGBA по пять байт, а для альфа-канала - один байт.

И тип gl.UNSIGNED_SHORT_5_6_5 предоставляет для каналов красного и синего цветов по пять байт и для зеленого цвета - шесть байт в формате RGB.

Определение параметров текстуры

Метод gl.texParameteri() позволяет определить параметры текстуры. Он имеет следующий формальный синтаксис:

texParameteri(target, pname, param)

Сочетания параметров бывают разными и могут влиять на используемые значения:

- target: в зависимости от направления текстурирования может принимать значения gl.TEXTURE_2D, либо gl.TEXTURE_CUBE_MAP;
- pname: указывает на фильтр, который мы хотим установить. Может принимать следующие значения: gl.TEXTURE_MAG_FILTER, gl.TEXTURE_MIN_FILTER, gl.TEXTURE_WRAP_S и gl.TEXTURE_WRAP_T;
- param: предоставляет значение для фильтра pname; то есть в выражении gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST) фильтру текстуры gl.TEXTURE_MAG_FILTER устанавливается значение gl.NEAREST.

Значения, передаваемые параметром param, разнообразны и позволяют создавать различные эффекты. Зачем вообще нужна настройка этих параметров? В реальности текстуры имеют определенные размеры, например, 128x128. Однако поверхность объекта, на которую накладывается текстура, может иметь как большие, так и меньшие размеры. Использование фильтра gl.TEXTURE_MAG_FILTER фактически помогает определить рендеринг текстуры: если она меньше размера объекта, то ее надо увеличить. Фильтр gl.TEXTURE_MIN_FILTER, наоборот, указывает, каким образом надо проводить рендеринг, если размеры поверхности объекта меньше размеров текстуры.

Значение gl.NEAREST позволяет семплеру взять из текстуры цвет того текселя, центр которого находится ближе всего к точке, с которой семплер берет цветовые значения. Это значение может быть установлено как для фильтра gl.TEXTURE_MIN_FILTER, так и для фильтра gl.TEXTURE_MAG_FILTER.

Значение gl.LINEAR: фильтр возвращает средневзвешенное значение соседних четырех пикселей, центры которых находятся ближе всего к точке, с которой семплер берет цветовые значения. Это обеспечивает плавное смешивание цветов. В то же время, поскольку здесь для определения цвета нужны значения четырех пикселей, то и работать данный фильтр будет медленне, чем gl.NEAREST, но при этом более качественно.

Это значение может быть установлено как для фильтра gl.TEXTURE_MIN_FILTER, так и для фильтра gl.TEXTURE_MAG_FILTER.

Сравнение двух фильтров:

Рис. 10.5

Mip-текстурирование

Концепция mip-текстурирования предполагает использование нескольких копий одной текстуры, но с разной детализацией. Это позволяет повышать качество отображения, например, при приближении к объекту.

Mip-текстурирование в WebGL использует ряд фильтров. Подобные фильтры могут использоваться только в качестве значения для фильтра gl.TEXTURE_MIN_FILTER:

- gl.NEAREST_MIPMAP_NEAREST: фильтр использует одну копию текстуры, которая наиболее подходит под размеры текстуры на экране. Выборка семплером значений происходит по алгоритму NEAREST. Самый быстрый способ текстурирования, но при этом менее качественный;
- gl.LINEAR_MIPMAP_NEAREST: фильтр использует одну копию текстуры, которая наиболее подходит под размеры текстуры на экране. Выборка семплером значений происходит по алгоритму LINEAR;
- gl.NEAREST_MIPMAP_LINEAR: фильтр использует две копии текстуры, которые наиболее подходят под размеры текстуры на экране. Выборка семплером значений происходит по алгоритму NEAREST. Выборка цвета пикселя идет параллельно сразу из двух копий, а финальное значение цвета представляет средневзвешенное значение двух выборок;
- gl.LINEAR_MIPMAP_LINEAR: фильтр использует две копии текстуры, которые наиболее подходят под размеры текстуры на экране. Выборка семплером значений происходит по алгоритму LINEAR. Выборка цвета пикселя идет параллельно сразу из двух копий, а финальное значение цвета представляет средневзвешенное значение двух выборок. Наиболее медленный способ, но при этом дающий наилучшее качество.

Само использование этих значений для фильтров еще предполагает, что у нас будет использоваться mip-текстурирование. Перед этим нам надо сгенерировать мипмапы (то есть копии текстуры) с помощью метода gl.generateMipmap(gl.TEXTURE_2D). Этот метод должен вызываться после метода gl.texImage2D(). Изменим в предыдущих примерах текстурирования функцию handleTextureLoaded и ее так, чтобы использовались мипмапы:

```
function handleTextureLoaded(image, texture) {
  gl.bindTexture(gl.TEXTURE_2D, texture);
  gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
```

```
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
            image);
    gl.generateMipmap(gl.TEXTURE_2D);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
            gl.LINEAR_MIPMAP_LINEAR);
}
```

Это даст нам следующий результат:

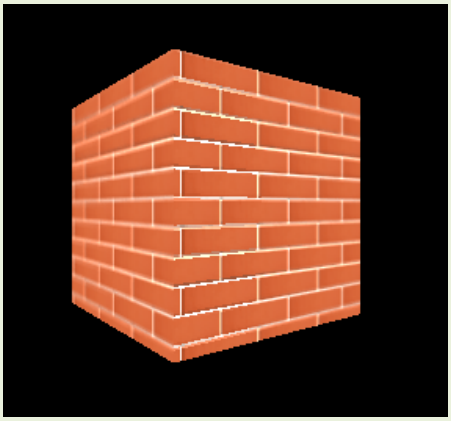

Рис. 10.6

Надо отметить, что mip-текстурирование имеет некоторые ограничения: используемые изображения должны иметь размеры, которые равны степени двойки: 16px, 32px, 64px, 128px и т.д. При этом не обязательно, чтобы высота и ширина были равны.

Texture wrapping

Еще один способ текстурирования называется **texture wrapping**. Этот термин можно перевести как обертывание текстурой. Данный способ определяет поведение семплера при отборе цветов пикселей с текстуры, если заданные координаты текстуры находятся вне диапазона [0.0, 1.0].

В данном случае нам потребуется установить значения для фильтров gl.TEXTURE_WRAP_S и gl.TEXTURE_WRAP_T, которые отвечают за рендеринг текстуры вдоль осей s и t.

Например, у нас определены следующие координаты текстуры в буфере координат текстуры:

```
// Координаты текстуры
var textureCoords = [
        0.0, 0.0,
        0.0, 2.0,
        2.0, 2.0,
        2.0, 0.0,

        0.0, 0.0,
        0.0, 2.0,
        2.0, 2.0,
        2.0, 0.0,

        0.0, 0.0,
        0.0, 2.0,
        2.0, 2.0,
```

```
        2.0, 0.0,

        0.0, 0.0,
        0.0, 2.0,
        2.0, 2.0,
        2.0, 0.0
    ];
```

Тогда функция handleTextureLoaded будет выглядеть следующим образом:

```
function handleTextureLoaded(image, texture) {
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
            image);
    gl.generateMipmap(gl.TEXTURE_2D);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
}
```

В результате получим:

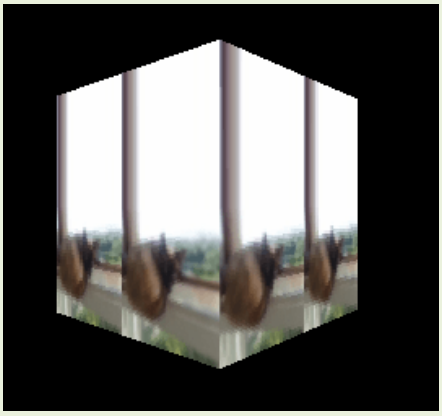

Рис. 10.7

Для параметров мы можем использовать следующие значения:

- gl.CLAMP_TO_EDGE: все координаты текстуры, которые больше 1 и меньше 0, сжимаются до диапазона [0, 1];
- gl.REPEAT: происходит повторение текстуры после выхода вне диапазона [0, 1];
- gl.MIRRORED_REPEAT: повторение текстуры с зеркальным отображением.

Можно комбинировать данные значения как в вышеприведенном примере, где одновременно используются gl.REPEAT и gl.CLAMP_TO_EDGE.

10.4. Текстурирование 3D-объектов

Рассмотрев текстурирование двухмерных объектов, можно перейти и к трехмерным. Фактически все будет так же, как и для двухмерных, только здесь мы добавляем больше вершин для имитации 3D, а также матрицы. Создадим следующий файл (**ex10_03.html**):

```
<!DOCTYPE html>
<html>
<head>
<title>Текстурирование в 3D</title>
```

```
<meta charset="utf-8" />
</head>
<body>
<canvas id="canvas3D" width="400" height="300">Ваш браузер не поддерживает
        элемент canvas</canvas>
<script  type="text/javascript" src="gl-matrix-min.js"></script>
<script id="shader-fs" type="x-shader/x-fragment">
precision highp float;
uniform sampler2D uSampler;
varying vec2 vTextureCoords;
 void main(void) {
   gl_FragColor = texture2D(uSampler, vTextureCoords);
 }
</script>
<script id="shader-vs" type="x-shader/x-vertex">
attribute vec3 aVertexPosition;
attribute vec2 aVertexTextureCoords;
varying vec2 vTextureCoords;
uniform mat4 uMVMatrix;
 uniform mat4 uPMatrix;
 void main(void) {
   gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
   vTextureCoords = aVertexTextureCoords;
 }
</script>
<script type="text/javascript">
var gl;
var shaderProgram;
var vertexBuffer;
var indexBuffer;
var textureCoordsBuffer; // буфер координат текстуры
var texture; // переменная для хранения текстуры
var angle = 2.0;//угол вращения в радианах
var zTranslation = -2.0; // смещение по оси Z
var mvMatrix = mat4.create();
var pMatrix = mat4.create();
// 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 удалось установить шейдеры");
   }
   gl.useProgram(shaderProgram);
   shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
         "aVertexPosition");
   gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
```

```
    shaderProgram.vertexTextureAttribute = gl.getAttribLocation(shaderProgram,
            "aVertexTextureCoords");
  gl.enableVertexAttribArray(shaderProgram.vertexTextureAttribute);

    shaderProgram.MVMatrix = gl.getUniformLocation(shaderProgram, "uMVMatrix");
    shaderProgram.ProjMatrix = gl.getUniformLocation(shaderProgram, "uPMatrix");
}
function setMatrixUniforms(){
    gl.uniformMatrix4fv(shaderProgram.ProjMatrix,false, pMatrix);
    gl.uniformMatrix4fv(shaderProgram.MVMatrix, false, mvMatrix);
}
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;
}
function initBuffers() {
    var vertices =[
                // лицевая часть
                -0.5, -0.5, 0.5,
                -0.5, 0.5, 0.5,
                 0.5, 0.5, 0.5,
                 0.5, -0.5, 0.5,
                // задняя часть
                -0.5, -0.5, -0.5,
                -0.5, 0.5, -0.5,
                 0.5, 0.5, -0.5,
                 0.5, -0.5, -0.5
                 ];
    var indices = [ // лицевая часть
                0, 1, 2,
                2, 3, 0,
                //нижняя часть
                0, 4, 7,
                7, 3, 0,
                // левая боковая часть
                0, 1, 5,
                5, 4, 0,
                // правая боковая часть
                2, 3, 7,
                7, 6, 2,
                // верхняя часть
                2, 1, 6,
                6, 5, 1,
                // задняя часть
```

```
            4, 5, 6,
            6, 7, 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;
 // Координаты текстуры
 var textureCoords = [
            0.0, 0.0,
            0.0, 1.0,
            1.0, 1.0,
            1.0, 0.0,

            0.0, 0.0,
            0.0, 1.0,
            1.0, 1.0,
            1.0, 0.0,
             ];
 // Создание буфера координат текстуры
 textureCoordsBuffer = gl.createBuffer();
 gl.bindBuffer(gl.ARRAY_BUFFER, textureCoordsBuffer);
 gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoords),
          gl.STATIC_DRAW);
  textureCoordsBuffer.itemSize=2;
}
function draw() {
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
  gl.bindBuffer(gl.ARRAY_BUFFER, textureCoordsBuffer);
 gl.vertexAttribPointer(shaderProgram.vertexTextureAttribute,
                textureCoordsBuffer.itemSize, gl.FLOAT, false, 0, 0);
 gl.activeTexture(gl.TEXTURE0);
 gl.bindTexture(gl.TEXTURE_2D, texture);
  gl.enable(gl.DEPTH_TEST);
  gl.drawElements(gl.TRIANGLES, indexBuffer.numberOfItems,
          gl.UNSIGNED_SHORT,0);
}
function setupWebGL()
{
  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.clear(gl.COLOR_BUFFER_BIT || gl.DEPTH_BUFFER_BIT);
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  mat4.perspective(pMatrix, 1.04, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0);
  mat4.identity(mvMatrix);
  mat4.translate(mvMatrix,mvMatrix,[0, 0, zTranslation]);
```

```
    mat4.rotate(mvMatrix,mvMatrix, angle, [0, 1, 0]);
}
function setTextures(){
    texture = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, texture);
    var image = new Image();
    image.onload = function() {
        handleTextureLoaded(image, texture);
   }
    image.src = "brickwall.png";
    shaderProgram.samplerUniform = gl.getUniformLocation(shaderProgram, "uSampler");
    gl.uniform1i(shaderProgram.samplerUniform, 0);
}
 function handleTextureLoaded(image, texture) {
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
            image);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
    gl.bindTexture(gl.TEXTURE_2D, null);
}
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){
        document.addEventListener('keydown', handleKeyDown, false);
        gl.viewportWidth = canvas.width;
        gl.viewportHeight = canvas.height;
        initShaders();
        initBuffers();
        setTextures();
        (function animloop(){
            setupWebGL();
            setMatrixUniforms();
            draw();
            requestAnimFrame(animloop, canvas);
        })();
    }
}
function handleKeyDown(e){
    switch(e.keyCode)
    {
        case 39:
            angle+=0.1;
            break;
```

```
      case 37:
         angle-=0.1;
         break;
      case 40:
         zTranslation+=0.1;
         break;
      case 38:
         zTranslation-=0.1;
         break;
   }
}
window.requestAnimFrame = (function(){
    return  window.requestAnimationFrame       ||
          window.webkitRequestAnimationFrame ||
          window.mozRequestAnimationFrame    ||
          window.oRequestAnimationFrame      ||
          window.msRequestAnimationFrame     ||
       function(callback, element) {
         return window.setTimeout(callback, 1000/60);
       };
   })();
</script>
</body>
</html>
```

Здесь совмещено текстурирование с созданием трехмерного объекта. У нас восемь вершин, по которым создается куб, и восемь координат текстуры, которые соответствуют вершинам. Однако, если мы запустим страничку, то получим не совсем ожидаемые результаты:

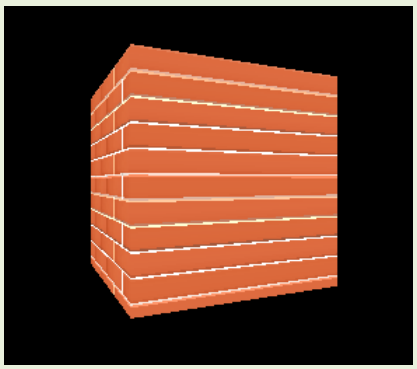

Рис. 10.8

Очевидно, что куб текстурирован неправильно. Так как мы определили всего восемь вершин и координат текстур, то и текстурирование идет только для двух сторон куба, образуемых этими восемью вершинами.

Чтобы оттекстурировать остальные стороны куба, нам надо добавить дополнительные вершины и соответствующие им координаты текстуры. Добавим вершины и координаты еще для двух боковых сторон, изменив функцию initBuffers следующим образом:

```
function initBuffers() {
   var vertices =[
            // лицевая часть
            -0.5, -0.5, 0.5,
```

```
            -0.5, 0.5, 0.5,
             0.5, 0.5, 0.5,
             0.5, -0.5, 0.5,
            // задняя часть
            -0.5, -0.5, -0.5,
            -0.5, 0.5, -0.5,
             0.5, 0.5, -0.5,
             0.5, -0.5, -0.5,
             // левая боковая часть
             -0.5, -0.5, 0.5,
            -0.5, 0.5, 0.5,
            -0.5, 0.5, -0.5,
            -0.5, -0.5, -0.5,
            // правая боковая часть
             0.5, -0.5, 0.5,
            0.5, 0.5, 0.5,
             0.5, 0.5, -0.5,
             0.5, -0.5, -0.5
             ];
  var indices = [ // лицевая часть
            0, 1, 2,
            2, 3, 0,
            // задняя часть
            4, 5, 6,
            6, 7, 4,
            //левая боковая часть
            8, 9, 10,
            10, 11, 8,
            // правая боковая часть
            12, 13, 14,
            14, 15, 12
          ];
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;
// Координаты текстуры
var textureCoords = [
            0.0, 0.0,
            0.0, 1.0,
            1.0, 1.0,
            1.0, 0.0,

            0.0, 0.0,
            0.0, 1.0,
            1.0, 1.0,
            1.0, 0.0,
```

```
            0.0, 0.0,
            0.0, 1.0,
            1.0, 1.0,
            1.0, 0.0,

            0.0, 0.0,
            0.0, 1.0,
            1.0, 1.0,
            1.0, 0.0
        ];
 // Создание буфера координат текстуры
 textureCoordsBuffer = gl.createBuffer();
 gl.bindBuffer(gl.ARRAY_BUFFER, textureCoordsBuffer);
 gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoords),
          gl.STATIC_DRAW);
   textureCoordsBuffer.itemSize=2;
}
```

Так как координаты у нас повторяются, мы можем сократить код, использовав метод push и цикл для добавления координат текстуры:

```
for (var i=0; i<4; i++) { textureCoords.push(0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0); }
```

В итоге после определения боковых сторон результат будет несколько иным (файл **ex10_04.html**):

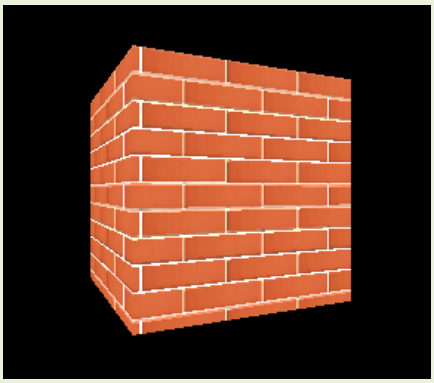

Рис. 10.9

10.5. Множественное текстурирование

При построении трехмерных сцен мы вряд ли будем ограничиваться одним объектом и одной текстурой. Возникает вопрос, как совместить в проекте использование сразу нескольких текстур для разных объектов? Один из способов состоит в последовательной отрисовке объектов и использовании текстур.

Расширим предыдущий пример так, чтобы он использовал несколько объектов и текстур: две кирпичные колонны и каменная стена над ними. Полный код программы приведён в файле **ex10_05.html**. Результат её выполнения:

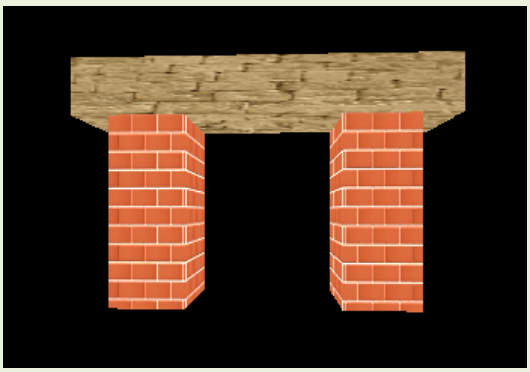

Рис. 10.10

Здесь не использовано ничего нового, просто мы определили для двух объектов свой набор буферов вершин, индексов и координат текстуры. Переменная wallTexture будет хранить в себе текстуру кирпичной стены, а переменная roofTexture - текстуру каменной стены **stone.jpg**.

Для инициализации буферов двух кирпичных колонн, которые по сути будут составлять один объект, используется функция initWallBuffers, а для инициализации буферов каменной стены - функция initRoofBuffers.

Также нам надо два раза инициализировать текстуры и загрузить в них изображения. Это делается в функции setupTextures():

```
function setupTextures() {
   wallTexture = gl.createTexture();
   setTexture("brickwall.png", wallTexture);
   roofTexture = gl.createTexture();
   setTexture("stone.jpg", roofTexture);
}
```

После срабатывания этой функции мы сможем использовать текстуры. Для отрисовки каждого объекта создаем две функции: wallDraw() (отрисовка кирпичной стены) и roofDraw() (отрисовка каменной крыши).

В главной функции последовательно вызываем настройку буферов объектов и их отрисовку:

```
//.....................
      initShaders();
      initRoofBuffers();
      initWallBuffers();
      setupTextures();
      (function animloop(){

         setupWebGL();
         setMatrixUniforms();
         wallDraw();
         roofDraw();
         requestAnimFrame(animloop, canvas);
      })();
//.....................................
```

10.6. Добавление текстур в Three.js

Рассмотрим теперь процесс текстурирования с использованием библиотеки Three.js на

примере наложения текстуры на куб и на плоскость.

Для наложения текстур в Three.js имеется специальный класс ImageUtils с методом loadTexture, главным параметром которого является адрес (путь) к изображению текстуры. На основе текстуры Texture создается материал с параметром map: Texture. Далее - все как обычно, создается фигура с заданной геометрией и нашим материалом.

Для того чтобы текстуры нормально отображались, необходимо, чтобы обращение к веб-страницам происходило через сервер. Для «домашнего» тестирования достаточно установить на компьютере локальный веб-сервер, например, https://greggman.github.io/servez. Тогда наша страница будет доступна в браузере по адресу http://localhost/.

Создадим, например, куб с текстурой **kote.jpg**:

Рис. 10.11

Фрагмент кода создания и текстурирования куба (файл **ex10_06.html**):

```
var geometry = new THREE.BoxGeometry( 128, 128, 128);
var Texture = new THREE.ImageUtils.loadTexture( 'kote.jpg' );
var material = new THREE.MeshBasicMaterial( { map: Texture} );
Cube = new THREE.Mesh( geometry, material );
Cube.rotation.y = Math.PI / 4;
scene.add( Cube );
```

Результат:

Рис. 10.12

Если нужно наложить на разные стороны разные текстуры, то нужно создать массив материалов, аналогично тому, как это мы делали в другое теме, когда раскрашивали параллелепипед разными цветами.

Подготовим и запишем в папку textures картинки с цифрами 1, 2, 3, 4, 5, 6 с соответствующими названиями (1.png, 2.png, и т.д.). Объявим массив материалов и заполним его в цикле:

```
var materials = [];
for (i=1; i<=6; i++)
{
var Texture = new THREE.ImageUtils.loadTexture('textures/' + String(i) + '.png' );
var Material = new THREE.MeshBasicMaterial({ map: Texture, color: 0x00dcff} );
materials.push( Material );
}
```

Далее собираем материал, объявляем геометрию и создаем куб (файл **ex10_07.html**):

```
var material = new THREE.MeshFaceMaterial( materials ) ;
var geometry = new THREE.BoxGeometry( 100, 100, 100);
var Cube = new THREE.Mesh( geometry, material );
Cube.rotation.y = Math.PI / 6;
scene.add( Cube );
```

Результат:

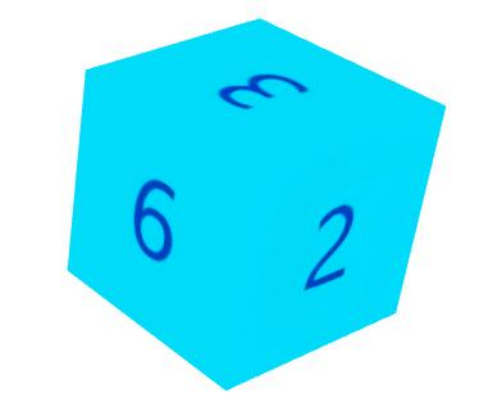

Рис. 10.13

Однородную текстуру можно «размножать» по поверхности. При этом используемые изображения должны иметь размеры, равные степеням двойки: 16px, 32px, 64px, 128px и т.д. Высота и ширина изображения не обязательно должны быть равными. Главное, чтобы их значения были равны 2^n.

Создадим на основе текстуры **checkerboard.jpg** шахматную доску:

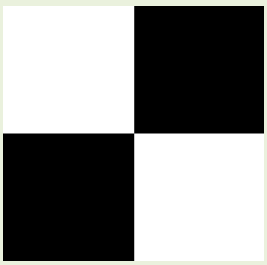

Рис. 10.14

Объявим:

```
var Texture = new THREE.ImageUtils.loadTexture('checkerboard.jpg' );
```

Для размножения текстуры предусмотрено три режима, которые задаются числовыми константами. Это:

```
THREE.RepeatWrapping = 1000;
THREE.ClampToEdgeWrapping = 1001;
THREE.MirroredRepeatWrapping = 1002;
```

В первом случае, который нам сейчас и понадобится, текстура повторяется обычным образом. Во втором случае текстура прижимается к левому нижнему углу, а в третьем - множится с зеркальным отображением. Итак, укажем:

```
Texture.wrapS = THREE.RepeatWrapping;
Texture.wrapT = THREE.RepeatWrapping;
```

Чтобы получилась шахматная доска, наш рисунок нужно повторить четыре раза по горизонтали и четыре раза по вертикали:

```
Texture.repeat.set( 4, 4 );
```

Можно указать также смещение текстуры по горизонтали и вертикали. В скобках указываются не пиксели, а доли исходного изображения, например:

```
Texture.offset.set( 0.5, 0 );
```

Здесь рисунок сдвигается вправо на величину, равную половине его ширины. Осталось создать материал на основе заданной текстуры и геометрию нашей будущей доски, которую мы построим как кусок плоскости с помощью PlaneGeometry:

```
var Material = new THREE.MeshBasicMaterial( { map: Texture, side:
        THREE.DoubleSide } );
var Geometry = new THREE.PlaneGeometry(300, 300, 1, 1);
```

Создаем собственно шахматную доску с указанными геометрией и материалом, ее позицию и разворачиваем:

```
var checkerboard = new THREE.Mesh(Geometry, Material);
checkerboard.position.y = - 1;
checkerboard.rotation.x = Math.PI / 2;
scene.add (checkerboard);
```

Результат (код примера – в файле **ex10_08.html**):

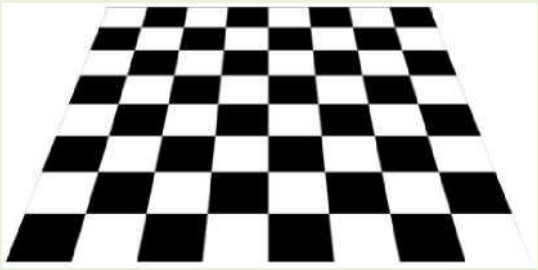

Рис. 10.15

Контрольные вопросы

1. Текстурирование 2D-объектов.
2. Текстурирование 3D-объектов.
3. Множественное текстурирование.