Программирование графических приложений

Тема 6.
Обработка растровых изображений

6.1. Способы растровой обработки текстур
6.2. Точечные преобразования при растровой обработке текстур
6.3. Матричные преобразования при растровой обработке текстур
6.4. Программа демонстрации работы фильтров

Контрольные вопросы

Цель изучения темы. Изучение способов обработки растровых изображений при формировании моделей графических объектов с использованием WebGL.

6.1. Способы растровой обработки текстур

Для обработки растровых текстур применяют **точечные** (пиксельные) и **матричные** преобразования.

Точечное преобразование выполняется последовательно над каждым пикселем изображения и не зависит от свойств соседних пикселей. К таким методам относятся:

- изменение (увеличение или уменьшение) ***яркости*** текущей точки умножением значения яркости на некоторый коэффициент или прибавление к нему некоторого значения (коэффициент может быть постоянным или меняющимся по определенному закону);
- изменение цветового ***баланса*** точки по одному из каналов (R, G, B) или по нескольким каналам с различными коэффициентами;
- ***инверсия*** цветов по различным законам;
- ***замена*** цветов по различным законам: R↔G, R↔B, G↔B.

К **матричным** преобразованиям относятся способы обработки текстур, при которых для расчета значения текущего пикселя учитываются значения соседних с ним пикселей:

- применение к изображению матриц ***свертки***;
- ***медианная*** фильтрация;
- последовательное применение ***нескольких*** фильтров.

Способы применения к изображению матриц ***свертки*** (обычно размером 3x3 или 5x5) используются для достижения эффектов увеличения резкости изображения, тиснения, размытия и других.

Для нормализации матриц свертки к ним применяются ***коэффициенты D и F***. В частности все элементы матрицы фильтра часто делят на сумму этих элементов, чтобы общая яркость исходного изображения не изменилась и не появились цветовые искажения, если они не нужны. Действительно, если преобразуемая область содержит один и тот же цвет, то результат получится как сумма элементов ядра, умноженная на этот цвет. Соответственно, чтобы оставить цвет без изменений, надо разделить результат преобразования на эту сумму.

Также при применении некоторых эффектов (например, тиснения), получающееся значение цветовой компоненты может оказаться больше 255 или меньше 0, то есть выйти за пределы цветового диапазона. Тогда применяется коэффициент F. В частности, если при применении матрицы получается, что в результате преобразования все цвета будут иметь отрицательную величину, нужно задать F=256. Вычитание из F=256 цветовых компонент позволяет получить негативное изображение и т.д. Обычно в алгоритмах при превышении компонентой величины 255, её значение устанавливают равным 255.

Исходная матрица	Применение коэффициента D	Применение коэффициента F	Применение коэффициентов D и F
A_{11} A_{12} A_{13}	A_{11}/D A_{12}/D A_{13}/D	$F+A_{11}$ $F+A_{12}$ $F+A_{13}$	$F+A_{11}/D$ $F+A_{12}/D$ $F+A_{13}/D$
A_{21} A_{22} A_{23}	A_{21}/D A_{22}/D A_{23}/D	$F+A_{21}$ $F+A_{22}$ $F+A_{23}$	$F+A_{21}/D$ $F+A_{22}/D$ $F+A_{23}/D$
A_{31} A_{32} A_{33}	A_{31}/D A_{32}/D A_{33}/D	$F+A_{31}$ $F+A_{32}$ $F+A_{33}$	$F+A_{31}/D$ $F+A_{32}/D$ $F+A_{33}/D$

$$D = 1 \text{ или } D = (A_{11} + A_{12} + A_{13} + A_{21} + A_{22} + A_{23} + A_{31} + A_{32} + A_{33})$$
$$F = 0 \text{ или } F = \pm 128 \text{ или } F = \pm 256$$

Медианный фильтр (применяется к отдельному цвету пикселя или ко всем цветам – яркости - пикселя). При этом пиксель изображения и его соседи в рассматриваемой области выстраиваются в вариационный ряд (по возрастанию или убыванию значений пикселей) и отбирается центральное значение этого вариационного ряда как новое значение пикселя. Медианный фильтр часто используется для подавления точечных и импульсных помех в изображении. Результатом усредненного фильтрования является устранение случайного шума,

содержащегося в изображении. Это происходит потому, что любое случайное резкое изменение в интенсивности пикселя в пределах рассматриваемой области будет сортироваться, т.е. оно будет помещаться либо на вершину, либо на нижнюю часть сортированных значений этой области и не будет учитываться, так как для нового значения элементов всегда отбирается центральное значение.

Последовательное применение ***нескольких*** способов обработки изображения используется для достижения более сложных визуальных эффектов. Например, акварельный эффект (преобразование изображения, при котором оно становится похожим на нарисованное акварелью) достигается медианным усреднением цветов с последующим выделением контуров (границ перехода цветов).

6.2. Точечные преобразования при растровой обработке текстур

Рассмотрим точечные способы обработки растровых изображений средствами WebGL. Изображения будут представлены текстурой, наложенной на квадрат.

Загрузка текстуры:

```
var imageTexture;
function initTexture() {
   imageTexture = gl.createTexture();
   imageTexture.image = new Image();
   imageTexture.image.onload = function() {  loadTextureCallback(imageTexture)   }
   imageTexture.image.src = "univer.png";
}
```

Файл **univer.png**:

Рис. 6.1

Для того чтобы иметь возможность загружать произвольные изображения (не только квадратные и не только размером равным степени двойки), в инициализацию текстур добавляется следующий фрагмент кода.

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
```

Далее через varying-переменную vTextureCoord передаем из вершинного шейдера во фрагментный шейдер текстурные координаты:

```
attribute vec3 aVertexPosition;
attribute vec2 aTextureCoord;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
varying vec2 vTextureCoord;
```

```
void main(void) {
   gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
   vTextureCoord = aTextureCoord;
}
```

Во фрагментном шейдере используем эти координаты для получения цвета:

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

В итоге мы получаем изображение в браузере:

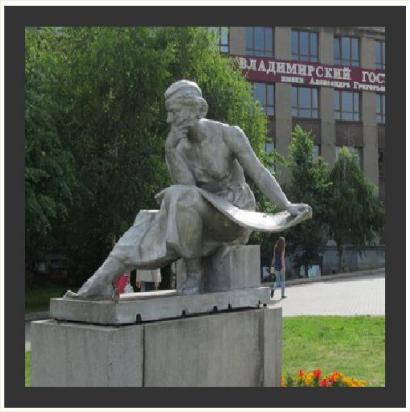

Рис. 6.2

Широкая серая обводка вокруг изображения - это цвет сцены, так как квадрат с текстурой не заполняет собой весь canvas.

Далее мы можем преобразовывать изображение с помощью различных точечных эффектов. Например, мы можем **увеличить яркость**; для этого достаточно немного изменить код фрагментного шейдера:

```
vec4 brightness = vec4 (0.5, 0.5, 0.5, 0);
gl_FragColor = texture2D (uSampler, vTextureCoord)
```

Можно также сделать **инверсию цветов** или, например, поменять местами красный и зеленый цвет:

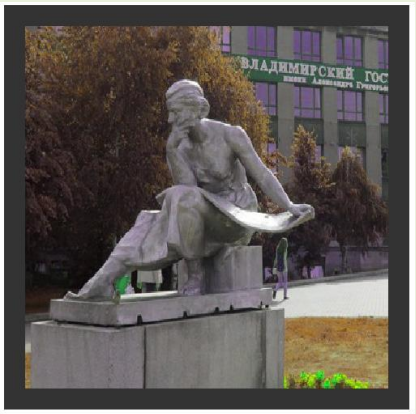

Рис. 6.3

Фрагментный шейдер в этом случае будет выглядеть следующим образом:

```
gl_FragColor = texture2D(uSampler, vTextureCoord).grba;
```

6.3. Матричные преобразования при растровой обработке текстур

Основной же способ обработки растровых изображений связан с применением **растровых фильтров** в виде **матриц свёртки**. Идея фильтров состоит в том, что для генерации итогового пикселя берется исходный пиксель и несколько пикселей по соседству, а затем все они объединяются по определенному правилу.

Рассмотрим один из самых простых фильтров - фильтр **размытия** (Blur). Матрица для него выглядит следующим образом:

$$\frac{1}{5}\begin{bmatrix}0 & 1 & 0\\ 1 & 1 & 1\\ 0 & 1 & 0\end{bmatrix}$$

Следовательно, нужно взять текущий пиксель (единичка в середине матрицы), затем сложить его с правым, левым, нижним и верхним пикселем и результат разделить на 5.

В качестве примера получения значения соседних пикселей во фрагментном шейдере рассмотрим картинку 200x200 пикселей. Текстурные координаты принимают значения в диапазоне от 0 до 1, в которые помещаются все 200 пикселей изображения. Тогда для получения следующего пикселя нам нужно прибавить 1/200 к текущему значению текстурных координат. Для реализации фильтра нам необходимо внести два изменения в текст программы. Во-первых, при инициализации текстуры нужно передать текущий размер изображения во фрагментный шейдер:

```
var textureSizeLocation = gl.getUniformLocation (shaderProgram, "uTextureSize");
gl.uniform2f (textureSizeLocation, texture.image.width, texture.image.height);
```

Во-вторых, необходимо переписать код шейдера, чтобы он учитывал соседние пиксели:

```
precision mediump float;
varying vec2 vTextureCoord;
uniform vec2 uTextureSize;
uniform sampler2D uSampler;
void main(void) {
   vec2 pixelSize = vec2(1.0, 1.0) / uTextureSize;
   gl_FragColor = (
      texture2D(uSampler, vTextureCoord) +
      texture2D(uSampler, vTextureCoord + vec2(pixelSize.x, 0.0)) +
      texture2D(uSampler, vTextureCoord + vec2(-pixelSize.x, 0.0)) +
      texture2D(uSampler, vTextureCoord + vec2(0.0, pixelSize.y)) +
      texture2D(uSampler, vTextureCoord + vec2(0.0, -pixelSize.y))
   ) / 5.0;
}
```

Здесь pixelSize — это смещение на один пиксель в пересчете на текстурные координаты. То есть к значению текущего пикселя мы прибавляем два соседних значения по X и два соседних значения по Y, а затем делим результат на 5 - в соответствие с приведенной выше матрицей. В результате у нас получится размытое изображение (справа показан оригинал, чтобы было хорошо видно разницу):

Рис. 6.4

Однако здесь шейдер ориентирован только на один фильтр. Для еще одного фильтра нам понадобится еще один шейдер. Чтобы обойтись одним шейдером, будем передавать в шейдер матрицу коэффициентов и делитель.

Добавим в программу фильтр **выделения границ** (Edge detection). Матрицу для него возьмем следующую:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Итак, вершинный шейдер:

```
precision mediump float;
varying vec2 vTextureCoord;
uniform vec2 uTextureSize;
uniform float uKernel[9];
uniform float uKernelWeight;
uniform sampler2D uSampler;
void main(void) {
   vec2 pixelSize = vec2(1.0, 1.0) / uTextureSize;
   vec4 colorSum =
      texture2D(uSampler, vTextureCoord + pixelSize * vec2(-1, -1)) * uKernel[0] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2( 0, -1)) * uKernel[1] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2( 1, -1)) * uKernel[2] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2(-1,  0)) * uKernel[3] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2( 0,  0)) * uKernel[4] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2( 1,  0)) * uKernel[5] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2(-1,  1)) * uKernel[6] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2( 0,  1)) * uKernel[7] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2( 1,  1)) * uKernel[8] ;
   gl_FragColor = colorSum / uKernelWeight;
}
```

У нас появилось две uniform-переменных: переменная uKernel, которая представляет собой рассмотренную выше матрицу, и uKernelWeight — делитель. Делитель, в отличие от фильтра размытия, будет равен единице. В colorSum записывается результат от сложения текущего пикселя и всех пикселей-соседей, а затем результат делится на соответствующий коэффициент. Осталось лишь передать все данные из JavaScript:

```
var kernelLocation = gl.getUniformLocation(shaderProgram, "uKernel[0]");
var edgeDetectKernel = [
```

```
   0,  1, 0,
   1, -4, 1,
   0,  1, 0
  ];
gl.uniform1fv(kernelLocation, edgeDetectKernel);
var kernelWeightLocation = gl.getUniformLocation(shaderProgram, "uKernelWeight");
gl.uniform1f(kernelWeightLocation, 1.0);
```

С помощью uniform1fv мы передаем массив чисел с плавающей точкой в uniform-переменную uKernel, а через uniform1f — одно число с плавающей точкой в uKernelWeight.

Если на этом этапе выполнить программу, в браузере мы увидим белый квадрат без изображения, так как в вычислении участвует альфа-канал. Исключим его влияние и установим его значение в единицу, независимо от результатов вычислений:

```
precision mediump float;
varying vec2 vTextureCoord;
uniform vec2 uTextureSize;
uniform float uKernel[9];
uniform float uKernelWeight;
uniform sampler2D uSampler;
void main(void) {
   vec2 pixelSize = vec2(1.0, 1.0) / uTextureSize;
   vec4 colorSum =
      texture2D(uSampler, vTextureCoord + pixelSize * vec2(-1, -1)) * uKernel[0] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2( 0, -1)) * uKernel[1] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2( 1, -1)) * uKernel[2] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2(-1,  0)) * uKernel[3] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2( 0,  0)) * uKernel[4] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2( 1,  0)) * uKernel[5] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2(-1,  1)) * uKernel[6] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2( 0,  1)) * uKernel[7] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2( 1,  1)) * uKernel[8] ;
   gl_FragColor = vec4((colorSum / uKernelWeight).rgb, 1.0);
}
```

В результате получим изображение:

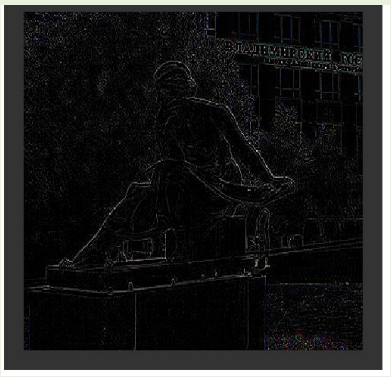

Рис. 6.5

6.4. Программа демонстрации работы фильтров

Теперь, когда матрица фильтра является параметром, мы можем написать программу, демонстрирующую работу нескольких фильтров, которые можно выбирать из выпадающего списка (**ex06_01.html**):

```
<!doctype html>
<html lang="ru">
<head>
<title>Обработка изображений</title>
<meta content="charset=utf-8">
<script type="text/javascript" src="gl-matrix-min.js"></script>
<script type="text/javascript">
  var filters = {
    Original: {
      weight: 1.0,
      kernel: [   0, 0, 0,
                  0, 1, 0,
                  0, 0, 0      ] },
    GaussianBlur: {
      weight: 16.0,
      kernel: [   1, 2, 1,
                  2, 4, 2,
                  1, 2, 1      ] },
    Sharpen: {
      weight: 1.0,
      kernel: [    0, -1,  0,
                  -1,  5, -1,
                   0, -1,  0   ] },
    EdgeDetection: {
      weight: 1.0,
      kernel: [   0,  1, 0,
                  1, -4, 1,
                  0,  1, 0     ] },
    Emboss: {
      weight: 1.0,
      kernel: [  -2, -1, 0,
                 -1,  1, 1,
                  0,  1, 2     ] }
  };
</script>
<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;
  varying vec2 vTextureCoord;
  uniform vec2 uTextureSize;
  uniform float uKernel[9];
  uniform float uKernelWeight;
  uniform sampler2D uSampler;
  void main(void) {
    vec2 pixelSize = vec2(1.0, 1.0) / uTextureSize;
    vec4 colorSum =
      texture2D(uSampler, vTextureCoord + pixelSize * vec2(-1, -1)) * uKernel[0] +
      texture2D(uSampler, vTextureCoord + pixelSize * vec2( 0, -1)) * uKernel[1] +
```

```
            texture2D(uSampler, vTextureCoord + pixelSize * vec2( 1, -1)) * uKernel[2] +
            texture2D(uSampler, vTextureCoord + pixelSize * vec2(-1,  0)) * uKernel[3] +
            texture2D(uSampler, vTextureCoord + pixelSize * vec2( 0,  0)) * uKernel[4] +
            texture2D(uSampler, vTextureCoord + pixelSize * vec2( 1,  0)) * uKernel[5] +
            texture2D(uSampler, vTextureCoord + pixelSize * vec2(-1,  1)) * uKernel[6] +
            texture2D(uSampler, vTextureCoord + pixelSize * vec2( 0,  1)) * uKernel[7] +
            texture2D(uSampler, vTextureCoord + pixelSize * vec2( 1,  1)) * uKernel[8];
        gl_FragColor = vec4((colorSum / uKernelWeight).rgb, 1.0);
    }
</script>
<script id="shader-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;
    attribute vec2 aTextureCoord;
    uniform mat4 uMVMatrix;
    uniform mat4 uPMatrix;
    varying vec2 vTextureCoord;
    void main(void) {
        gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
        vTextureCoord = aTextureCoord;
    }
</script>
<script type="text/javascript">
    var gl;
    function initGL(canvas) {
        try {
            gl = canvas.getContext("experimental-webgl");
            gl.viewportWidth = canvas.width;
            gl.viewportHeight = canvas.height;
        } catch (e) { }    if (!gl) {    alert("Не поддерживается  WebGL"); }
    }
    function getShader(gl, id) {
        var shaderScript = document.getElementById(id);
        if (!shaderScript) {    return null; }
        var str = "";
        var k = shaderScript.firstChild;
        while (k) {
            if (k.nodeType == 3) {  str += k.textContent;  }
            k = k.nextSibling;
        }
        var shader;
        if (shaderScript.type == "x-shader/x-fragment") {
            shader = gl.createShader(gl.FRAGMENT_SHADER);
        } else if (shaderScript.type == "x-shader/x-vertex") {
            shader = gl.createShader(gl.VERTEX_SHADER);
        } else {    return null;   }
        gl.shaderSource(shader, str);
        gl.compileShader(shader);
        if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
            alert(gl.getShaderInfoLog(shader));
            return null;
        }
        return shader;
```

```
    }
    var shaderProgram;
function initShaders() {
  var fragmentShader = getShader(gl, "shader-fs");
  var vertexShader = getShader(gl, "shader-vs");
  shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);
  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
     alert("Could not initialise shaders");
  }
  gl.useProgram(shaderProgram);
  shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
          "aVertexPosition");
  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
  shaderProgram.textureCoordAttribute = gl.getAttribLocation(shaderProgram,
          "aTextureCoord");
  gl.enableVertexAttribArray(shaderProgram.textureCoordAttribute);
  shaderProgram.pMatrixUniform = gl.getUniformLocation(shaderProgram, "uPMatrix");
  shaderProgram.mvMatrixUniform = gl.getUniformLocation(shaderProgram,
          "uMVMatrix");
  shaderProgram.samplerUniform = gl.getUniformLocation(shaderProgram, "uSampler");
}
    var mvMatrix = mat4.create();
    var pMatrix = mat4.create();
    function setMatrixUniforms() {
       gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false, pMatrix);
       gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false, mvMatrix);
    }
    var vertexBuffer;
    var indicesBuffer;
function initBuffers() {
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer (gl.ARRAY_BUFFER, vertexBuffer);
  vertices = [   -1, -1, 0,
                 -1,  1, 0,
                  1,  1, 0,
                  1, -1, 0   ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  vertexBuffer.itemSize = 3;
  vertexBuffer.numItems = 4;
  vertexTextureCoordBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexTextureCoordBuffer);
  var textureCoords = [    0.0, 0.0,
                           0.0, 1.0,
                           1.0, 1.0,
                           1.0, 0.0     ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoords),
          gl.STATIC_DRAW);
  vertexTextureCoordBuffer.itemSize = 2;
  vertexTextureCoordBuffer.numItems = 4;
```

```
  indicesBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indicesBuffer);
  var indices = [      0, 2, 1,
                       0, 3, 2      ];
  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
          gl.STATIC_DRAW);
  indicesBuffer.itemSize = 1;
  indicesBuffer.numItems = 6;
}
   var imageTexture;
   function initTexture() {
      imageTexture = gl.createTexture();
      imageTexture.image = new Image();
      imageTexture.image.onload = function() {
         loadTextureCallback(imageTexture)
      }
      imageTexture.image.src = "univer.png";
   }
function loadTextureCallback(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,
          texture.image);
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
   var textureSizeLocation = gl.getUniformLocation(shaderProgram, "uTextureSize");
   gl.uniform2f(textureSizeLocation, texture.image.width, texture.image.height);
   drawScene();
}
function drawScene() {
   gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
   gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
   mat4.perspective(pMatrix, 45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0);
          mat4.identity(mvMatrix);
   mat4.translate(mvMatrix, mvMatrix, [0.0, 0, -2.0]);
   gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
   gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                          vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
   gl.bindBuffer(gl.ARRAY_BUFFER, vertexTextureCoordBuffer);
   gl.vertexAttribPointer(shaderProgram.textureCoordAttribute,
                          vertexTextureCoordBuffer.itemSize, gl.FLOAT, false, 0, 0);
   var selectedFilter = document.getElementById("filterSelect").value;
   var filterDescription = filters[selectedFilter];
   var kernelLocation = gl.getUniformLocation(shaderProgram, "uKernel[0]");
   gl.uniform1fv(kernelLocation, filterDescription.kernel);
   var kernelWeightLocation = gl.getUniformLocation(shaderProgram, "uKernelWeight");
   gl.uniform1f(kernelWeightLocation, filterDescription.weight);
   gl.activeTexture(gl.TEXTURE0);
   gl.bindTexture(gl.TEXTURE_2D, imageTexture);
   gl.uniform1i(shaderProgram.samplerUniform, 0);
```

```
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indicesBuffer);
    setMatrixUniforms();
    gl.drawElements(gl.TRIANGLES, indicesBuffer.numItems, gl.UNSIGNED_SHORT,
            0);
}
    function webGLStart() {
        var canvas = document.getElementById("canvas");
        initGL(canvas);
        initShaders();
        initBuffers();
        initTexture();
        gl.clearColor(0.2, 0.2, 0.2, 1);
    }
</script>
</head>
<body onload="webGLStart();">
    <br />
    <select id="filterSelect" onchange="drawScene()">
        <option value="Original" selected="selected">Исходное изображение</option>
        <option value="GaussianBlur">Размытие</option>
        <option value="Sharpen">Увеличение резкости</option>
        <option value="EdgeDetect">Выделение границ</option>
        <option value="Emboss">Тиснение</option>
    </select>
    <br />
    <br />
    <canvas id="canvas" width="500" height="500"></canvas>
</body>
</html>
```

Результат:

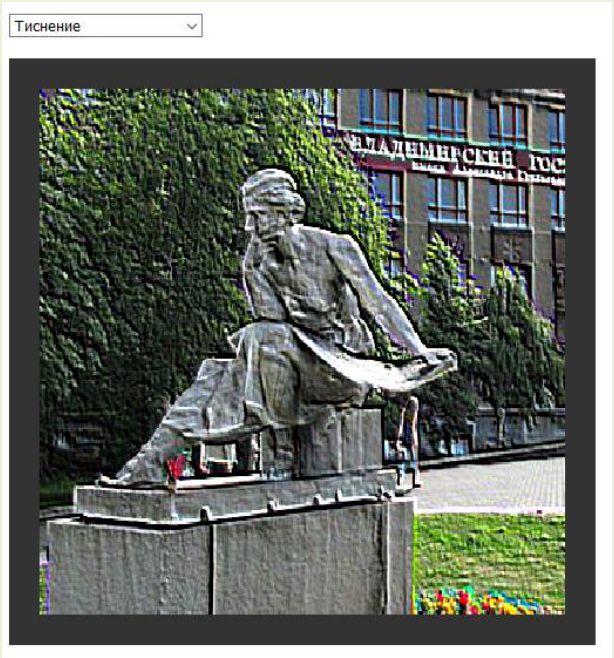

Рис. 6.6

Контрольные вопросы

1. Точечные и матричные преобразования.
2. Растровые фильтры.
3. Использование растровых фильтров в WebGL.