Программирование графических приложений

Тема 7
Двухмерные геометрические преобразования в WebGL

Цель изучения темы. Изучение способов реализации двухмерных геометрических преобразований при формировании графических сцен с использованием WebGL.

7.1. Перенос

При переносе объекта на экранной плоскости координаты всех его точек смещаются на одну и ту же величину. Рассмотрим это преобразование без использования библиотеки матчичных функций gl-matrix. В коде вершинного шейдера JavaScript будем использовать uniform-переменную uTranslation:

```
attribute vec3 aVertexPosition;
uniform vec2 uTranslation;
void main(void) { gl_Position = vec4(aVertexPosition, 1.0) + vec4(uTranslation, 0.0, 0.0); }
```

Мы видим, что текущие координаты рассчитываются как исходные координаты плюс некоторое смещение:

$$X' = X + Tx$$
$$Y' = Y + Ty$$

Здесь X и Y — исходные координаты, Tx и Ty — смещение, которое управляется на стороне JavaScript. Следует заметить, что uTranslation — вектор размерности 2, поэтому мы дополняем его нулями до размерности 4. Код вершинного шейдера довольно прост, так как расчет смещения происходит в JavaScript, а в шейдер передается лишь результат.

Рассмотрим теперь код JavaScript. Перемещать объект сцены мы будем при помощи мыши, поэтому инициализируем события:

```
function initEvents() {
   var canvas = document.getElementById("canvas");
   document.onmousemove=onMouseMove;
   document.onmouseup=onMouseUp;
   canvas.onmousedown=onMouseDown;
}
```

Далее зададим переменные, которыми будем регулировать перемещение объекта:

```
var dragStartX = 0;
var dragStartY = 0;
var dragOffset = [0, 0];
var mousePressed = false;
var translation = [0, 0];
```

Переменная mousePressed фиксирует, что перетягивание объекта на экеранной плоскости началось (над canvas была зажата кнопка мыши), в dragStartX и dragStartY сохраняются координаты зажатия кнопки, а в dragOffset хранится значение того, на сколько пикселей пользователь перетянул объект с момента последнего нажатия кнопки мыши. Также мы храним текущее смещение объекта translation, которое сформировалось после всех наших предыдущих перетягиваний.

Код обработчика нажатия кнопки:

```
function onMouseDown(evt){
   dragStartX = evt.clientX;
   dragStartY = evt.clientY;
   dragOffset = [0, 0];
   mousePressed = true;
}
```

Здесь запоминается, где была зажата кнопка мыши, и устанавливается флаг начала перетягивания. После зажатия кнопки начинается само перетягивание, которое обрабатывается в функции onMouseMove:

```
function onMouseMove(evt) {
   if (mousePressed) {
      var diffX = evt.clientX - dragStartX;
      var diffY = dragStartY - evt.clientY;
      dragOffset = [diffX * 2 / 500, diffY * 2 / 500];
      var finalTranslation = [translation[0] + dragOffset[0], translation[1] + dragOffset[1]]
      gl.uniform2fv(shaderProgram.translationUniform, finalTranslation);
   }
}
```

В этой функции происходит проверка, зажата ли кнопка мыши, и если да, то рассчитывается, насколько сместился курсор по соответствующим координатам в переменных diffX и diffY. Далее преобразуется смещение браузера в смещение внутри сцены (см. далее). Затем рассчитывается итоговое смещение finalTranslation как смещения от всех предыдущих перетягиваний плюс текущее смещение, и передается это итоговое смещение в шейдер.

Рассмотрим как происходит преобразование координат браузера в координаты сцены. Для этого нам нужно знать размеры canvas и размеры сцены. Размеры canvas мы знаем — это квадрат размером 500x500. А сцена, так как мы не перемещали камеру и вообще не делали ничего кроме отрисовки объекта, занимает пространство от -1 до 1 (и по горизонтали, и по вертикали). Получается, что сцена тоже квадратная и имеет размер 2x2. Отсюда получается, что один пиксель браузера равен 2/500 части сцены.

И, наконец, код обработчика отпускания мыши, где мы обновляем смещение и сбрасываем флаг перетягивания:

```
   function onMouseUp(evt){
      if (mousePressed) {
         translation = [translation[0] + dragOffset[0], translation[1] + dragOffset[1]];
         mousePressed = false;
      }
   }
```

Полный код примера (**ex07_01.html**):

```
<!doctype html>
<html lang="ru">
<head>
<title>Перемещение объектов</title>
<meta content="charset=utf-8">

<script id="shader-fs" type="x-shader/x-fragment">
   precision mediump float;
   void main(void) {   gl_FragColor = vec4(0.3, 0.6, 0.1, 1.0);  }
</script>

<script id="shader-vs" type="x-shader/x-vertex">
   attribute vec3 aVertexPosition;
   uniform vec2 uTranslation;
```

```
  void main(void) {  gl_Position = vec4(aVertexPosition, 1.0) + vec4(uTranslation, 0.0,
        0.0);  }
</script>

<script type="text/javascript">
  window.requestAnimFrame = (function() {
 return window.requestAnimationFrame ||
          window.webkitRequestAnimationFrame ||
          window.mozRequestAnimationFrame ||
          window.oRequestAnimationFrame ||
          window.msRequestAnimationFrame ||
          function(callback, element) {
            window.setTimeout(callback, 1000/60);
          };
})();

  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("Шейдеры не инициализируются");
    }
    gl.useProgram(shaderProgram);
    shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
        "aVertexPosition");
    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
    shaderProgram.translationUniform = gl.getUniformLocation(shaderProgram,
        "uTranslation");
  }
  var vertices;
  var vertexBuffer;
  function initBuffers() {
        vertices = [
        -0.25, -0.375, 0,
        -0.25,  0.500, 0,
         0.25,  0.500, 0,
         0.25,  0.500, 0,
         0.25, -0.375, 0,
        -0.25, -0.375, 0,

        -0.25, -0.375, 0,
        -0.25, -1.000, 0,
        -0.50, -1.000, 0,

         0.25, -0.375, 0,
         0.25, -1.000, 0,
         0.50, -1.000, 0,

         0.25,  0.500, 0,
         0.75,  0.500, 0,
         0.75,  0.250, 0,

        -0.25,  0.500, 0,
        -0.50,  0.125, 0,
        -0.75,  0.250, 0,

        -0.125,  0.75, 0,
         0.125,  0.75, 0,
         0.000,  0.50, 0,
];
    vertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
        gl.STATIC_DRAW);
    vertexBuffer.itemSize = 3;
    vertexBuffer.numItems = vertices.length / 3;
  }
```

```
  function drawScene() {
     gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
     gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
     gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
     gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute, vertexBuffer.itemSize,
gl.FLOAT, false, 0, 0);

     gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numItems);
  }
function tick() {
     requestAnimFrame(tick);
     drawScene();
  }

  function webGLStart() {
     var canvas = document.getElementById("canvas");
     initEvents();
     initGL(canvas);
     initShaders();
     gl.clearColor(0.5, 0.06, 0.68, 0.3);
     initBuffers();
     tick();
  }

  function initEvents() {
     var canvas = document.getElementById("canvas");
     document.onmousemove=onMouseMove;
     document.onmouseup=onMouseUp;
     canvas.onmousedown=onMouseDown;
  }
  var dragStartX = 0;
  var dragStartY = 0;
  var dragOffset = [0, 0];
  var mousePressed = false;
  var translation = [0, 0];
  function onMouseDown(evt){
     dragStartX = evt.clientX;
     dragStartY = evt.clientY;
     dragOffset = [0, 0];
     mousePressed = true;
  }

  function onMouseMove(evt) {
     if (mousePressed) {
        var diffX = evt.clientX - dragStartX;
        var diffY = dragStartY - evt.clientY;
        dragOffset = [diffX * 2 / 500, diffY * 2 / 500];
        var finalTranslation = [translation[0] + dragOffset[0], translation[1] +
          dragOffset[1]]
        gl.uniform2fv(shaderProgram.translationUniform, finalTranslation);
     }
  }
```

```
  function onMouseUp(evt){
    if (mousePressed) {
      translation = [translation[0] + dragOffset[0], translation[1] + dragOffset[1]];
      mousePressed = false;
    }
  }
</script>
</head>
<body onload="webGLStart();">
  <canvas id="canvas" width="500" height="500"></canvas>
</body>
</html>
```

После запуска программы начальная сцена выглядит так (затем пользователь может сдвинуть изображение объекта кнопкой мыши):

Рис. 7.1

7.2. Масштабирование

Рассмотрим операцию масштабирования (то есть растягивания и сжатия) объекта сцены также без использования матриц. Функционал переноса объектов в программе сохранится, а масштабирование будет происходить перемещением мыши с зажатой клавишей Shift.

Изменим приведенные выше уравнения переноса объекта: вместо прибавления смещения к координатам используем умножение на коэффициент масштабирования:

$$X' = X * Sx + Tx$$
$$Y' = Y * Sy + Ty$$

Здесь мы сначала умножаем исходную координату на коэффициент масштабирования, а затем добавляем смещение. Шейдер, который отражает эту формулу, имеет вид:

```
attribute vec3 aVertexPosition;
uniform vec2 uTranslation;
uniform vec2 uScale;
void main(void) {
  vec3 scaled = vec3(aVertexPosition.xy * uScale, aVertexPosition.z);
  gl_Position = vec4(scaled, 1.0) + vec4(uTranslation, 0.0, 0.0);
}
```

Пока мы работаем в двумерном пространстве, поэтому масштабирование происходит только по оси X и Y; координату Z мы оставляем без изменений. Причем когда один из коэффициентов принимает отрицательное значение, объект зеркально отражается относительно соответствующей оси.

Если значения uniform-переменных не заданы, они принимают значение 0. Для переноса начальное значение константы переноса как раз равно нулю (изначально переноса нет). В

случае масштабирования необходимо установить в функции initShaders начальные значения коэффициентов [1, 1] (иначе объект сожмется в невидимую точку):

```
shaderProgram.scaleUniform = gl.getUniformLocation(shaderProgram, "uScale");
gl.uniform2fv(shaderProgram.scaleUniform, [1, 1]);
```

Переходим от шейдеров к обработке событий. Для начала добавим несколько глобальных переменных:

```
var scale = [1, 1];
var currentEvent;
var eventType = { drag: 1,  scale: 2  };
```

Вектор scale — текущие коэффициенты масштабирования, в currentEvent отмечается текущие действия (перетягивает пользователь объект или масштабирует), перечень которых описан в eventType.

В функции onMouseDown теперь проверяется, зажата ли клавиша Shift при нажатии кнопки мыши, и устанавливается соответствующее событие:

```
function onMouseDown(evt){
   if (evt.shiftKey) {
      currentEvent = eventType.scale;
   } else { currentEvent = eventType.drag; }
   dragStartX = evt.clientX;
   dragStartY = evt.clientY;
   dragOffset = [0, 0];
   mousePressed = true;
}
```

Функция onMouseMove изменилась более значительно:

```
function onMouseMove(evt) {
   if (mousePressed) {
      var diffX = evt.clientX - dragStartX;
      var diffY = dragStartY - evt.clientY;
      if (currentEvent === eventType.drag) {
         dragOffset = [diffX * 2 / 500, diffY * 2 / 500];
         var finalTranslation = [translation[0] + dragOffset[0], translation[1] +
           dragOffset[1]]
         gl.uniform2fv(shaderProgram.translationUniform, finalTranslation);
      } else if (currentEvent === eventType.scale) {
         dragOffset = [diffX / 100, diffY / 100];
         var finalScale = [scale[0] + dragOffset[0], scale[1] + dragOffset[1]]
         gl.uniform2fv(shaderProgram.scaleUniform, finalScale);
      }
   }
}
```

При рассчете dragOffset коэффициенты делятся на 100, эта величина находится подбором. И, наконец, функция onMouseUp:

```
function onMouseUp(evt){
   if (mousePressed) {
```

```
        if (currentEvent === eventType.drag) {
            translation = [translation[0] + dragOffset[0], translation[1] + dragOffset[1]];
        } else if (currentEvent === eventType.scale) {
            scale = [scale[0] + dragOffset[0], scale[1] + dragOffset[1]]
        }
        mousePressed = false;
    }
}
```

В ней обновляется значение переноса или масштабирования в зависимости от текущего события.

Полный код программы – в файле **ex07_02.html**.

Результат работы программы (оригинал слева, а сжатое и смещенное изображение справа):

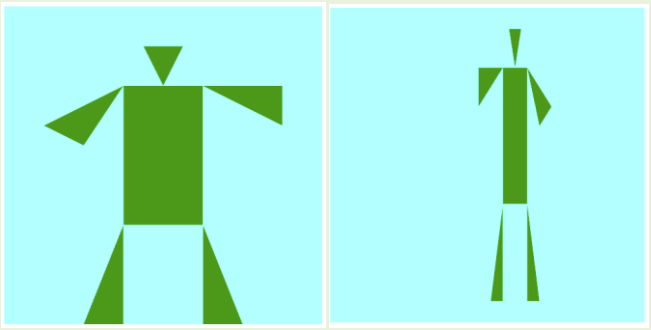

Рис. 7.2

Пользователь может сдвинуть изображение объекта кнопкой мыши, а выполнить его масштабирование – shift+мышь:

7.3. Поворот

Добавим в наш код операцию повотора. В двухмерном приложении объект будет вращаться только в одной плоскости – X0Y.

Рассмотрим уравнения поворота. При выполнении поворота вокруг центра экранной системы координат есть только один параметр - угол поворота. Для того, чтобы повернуть объект на angle градусов, нам понадобится сначала найти синус и косинус этого угла:

S = sin(angle)
C = cos(angle)

Тогда для точки с координатами X и Y итоговая формула будем иметь вид:

X’ = X * C + Y * S
Y’ = Y * C - X * S

Код с вершинного шейдера:

```
attribute vec3 aVertexPosition;
uniform vec2 uTranslation;
uniform vec2 uScale;
uniform float uAngle;
void main(void) {
    float angleRadians = radians(uAngle);
    vec2 uRotation = vec2(sin(angleRadians), cos(angleRadians));
    vec2 rotated = vec2(
        aVertexPosition.x * uRotation.y + aVertexPosition.y * uRotation.x,
```

```
        aVertexPosition.y * uRotation.y - aVertexPosition.x * uRotation.x
    );
    vec3 scaled = vec3(rotated * uScale, aVertexPosition.z);
    gl_Position = vec4(scaled, 1.0) + vec4(uTranslation, 0.0, 0.0);
}
```

Мы видим, что добавилась новая uniform-переменная uAngle типа float — это угол поворота объекта. Далее код шейдера отражает приведенную выше формулу. Предварительно величина угла поворота преобразуется из градусов (которые нам придут из скриптов) в радианы.

В коде javascript при объявлении глобальных переменных добавилась переменная rotation для хранения угла поворота и новый тип события rotate:

```
var translation = [0, 0];
var scale = [1, 1];
var rotation = 0;
var currentEvent;
var eventType = {
    drag: 1,
    scale: 2,
    rotate: 3
};
```

Код функции обработки нажатия мыши:

```
function onMouseDown(evt){
    if (evt.shiftKey) { currentEvent = eventType.scale; }
    else if (evt.ctrlKey) { currentEvent = eventType.rotate; }
      else { currentEvent = eventType.drag; }
    dragStartX = evt.clientX;
    dragStartY = evt.clientY;
    dragOffset = [0, 0];
    mousePressed = true;
}
```

То есть событие поворота начинается с нажатия кнопки мыши при зажатой клавише Ctrl. Затем идет функция обработки самого поворота (когда пользователь перемещает мышь при зажатой кнопке):

```
function onMouseMove(evt) {
    if (mousePressed) {
        var diffX = evt.clientX - dragStartX;
        var diffY = dragStartY - evt.clientY;
        if (currentEvent === eventType.drag) {
            dragOffset = [diffX * 2 / 500, diffY * 2 / 500];
            var finalTranslation = [translation[0] + dragOffset[0], translation[1] +
              dragOffset[1]]
            gl.uniform2fv(shaderProgram.translationUniform, finalTranslation);
        } else if (currentEvent === eventType.scale) {
            dragOffset = [diffX / 100, diffY / 100];
            var finalScale = [scale[0] + dragOffset[0], scale[1] + dragOffset[1]]
            gl.uniform2fv(shaderProgram.scaleUniform, finalScale);
        } else if (currentEvent === eventType.rotate) {
```

```
            dragOffset = diffX;
            gl.uniform1f(shaderProgram.angleUniform, rotation + dragOffset);
        }
    }
}
```

Здесь dragOffset хранит не массив, как в случае с переносом и масштабированием, а одно значение, так как нам нужно значение только одного угла для вращения в плоскости. Затем мы передаем это значение напрямую в шейдер без каких-либо изменений. Это означает, что угол поворота объекта будет равен количеству пройденных мышью пикселей (расстояние считается только при движении мыши по горизонтали).

И, наконец, функция отпускания кнопки мыши:

```
function onMouseUp(evt){
    if (mousePressed) {
        if (currentEvent === eventType.drag) {
            translation = [translation[0] + dragOffset[0], translation[1] + dragOffset[1]];
        } else if (currentEvent === eventType.scale) {
            scale = [scale[0] + dragOffset[0], scale[1] + dragOffset[1]];
        } else if (currentEvent === eventType.rotate) {
            rotation = rotation + dragOffset;
        }
        mousePressed = false;
    }
}
```

Полный код программы – в файле **ex_07_03.html**.

На рисунках повёрнутый и растянутый объект (слева оригинал):

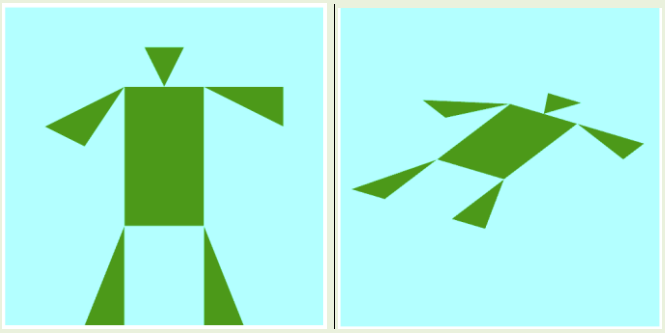

Рис. 7.3

Теперь пользователь может сдвинуть изображение объекта кнопкой мыши, выполнить его масштабирование – shift+мышь, выполнить поворот – ctrl+мышь.

Объект вращается вокруг своего центра, так как центр объекта совпал с нулевыми координатами. Стоит немного поднять объект в область положительных координат, и он начнет кружиться вокруг своего левого нижнего края. Чтобы не менять все координаты объекта, нам достаточно немного изменить шейдер, чтобы получить нужный эффект:

```
attribute vec3 aVertexPosition;
uniform vec2 uTranslation;
uniform vec2 uScale;
uniform float uAngle;
void main(void) {
   vec3 newVertexPosition = vec3(aVertexPosition.x + 0.5,
```

```
                                    aVertexPosition.y + 0.5,
                                    aVertexPosition.z);
    float angleRadians = radians(uAngle);
    vec2 uRotation = vec2(sin(angleRadians), cos(angleRadians));
    vec2 rotated = vec2(
        newVertexPosition.x * uRotation.y + newVertexPosition.y * uRotation.x,
        newVertexPosition.y * uRotation.y - newVertexPosition.x * uRotation.x
    );
    vec3 scaled = vec3(rotated * uScale, newVertexPosition.z);
    gl_Position = vec4(scaled, 1.0) + vec4(uTranslation, 0.0, 0.0);
}
```

Можно увидет как смещение объекта в начале шейдера влияет на его поведение. Именно поэтому важно соблюдать порядок применения геометрических преобразований: сначала поворот, затем масштабирование, затем перенос. Если изменить порядок преобразований, это приведет к другому поведению объекта при манипуляциях мышью.

7.4. Однородные координаты

В однородных координатах точки имеют четыре компоненты: x, y, z и w: первые три из которых - координаты евклидова пространства. Добавление четвертой компоненты обусловлено удобством описания операции переноса: в однородных координатах все геометрические преобразования, включая перенос, выполняются с помощью перемножения матриц. Кроме того, однородные координаты (в отличие от декартовых) позволяют найти точку пересечения двух параллельных линий - линии пересекутся в бесконечности, что важно для перспективной проекции в WebGL. Еще одно преимущество однородных координат - они позволяют отличить точку от направления. Если w = 1, тогда вектор (x, y, z, 1) - это точка в пространстве. Если w = 0, тогда вектор (x, y, z, 0) - это направление.

Возможен перевод декартовых координат в однородные и обратно. При переводе из декартовых координат в однородные достаточно добавить единицу в четвертую компоненту. А при переводе из однородных координат в декартовы необходимо все координаты разделить на w, после чего отбросить четвертую компоненту:

Декартовы(x, y, z) => Однородные(x, y, z, 1)

Однородные(x, y, z, w) => Декартовы(x/w, y/w, z/w)

7.5. Системы координат

Для построения изображения точки на экране её координаты проходят через несколько преобразований систем координат, включая координаты объекта, мировые координаты, координаты наблюдателя и экранные. Рассмотрим их все по порядку.

1. **Локальная** система координат (ЛСК) - система координат объекта. Эта система координат выбирается таким образом, чтобы в ней можно было наиболее просто определить конкретный объект, который необходимо изобразить.

Например, куб проще всего определить в системе координат, центр которой совпадает с одной из его вершин, а оси совпадают с тремя ребрами, выходящими из этой вершины. Шар же проще всего определить в системе координат, совпадающей с его центром. Это и будут локальные системы координат для данных объектов:

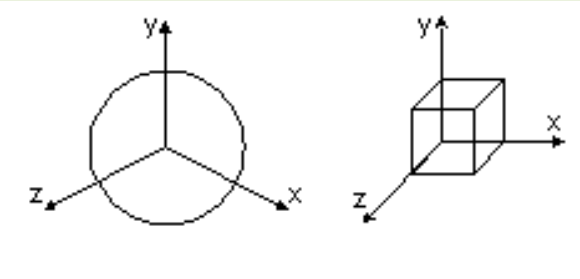

Рис. 7.4

2. **Мировая** система координат (МСК) - фактические координаты, соответствующие реальному положению любого объекта в пространстве.

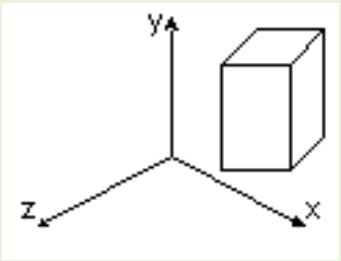

Рис. 7.5

Если в пространстве существует несколько кубов, достаточно описать один раз куб в ЛСК, а затем перенести его несколько раз в МСК в разные места, используя геометрические преобразования (перенос, поворот, масштабирование), то есть располагая их друг относительно друга необходимым образом. Мировые координаты определяют, где расположен тот или иной объект. Наблюдатель (камера) в мировой системе координат может быть расположен в произвольном месте.

3. **Пользовательская** система координат (ПСК) - система координат наблюдателя (камеры). Она опрелеляется положением наблюдателя в мировой системе координат. Начало координат классической ПСК совпадает с началом мировой системы координат; при этом наблюдатель находится на оси Z пользовательской системы координат, а экранная плоскость – в плоскости XOY этой системы координат.

4. Пользовательская система координат **отсечения** (ПСКО) – система координат, в которой все наблюдаемые объекты находятся внутри усеченной пирамиды с вершиной в точке (0,0,0), в которой находится наблюдатель (позиция камеры) и взгляд наблюдателя (ось пирамиды) направлен вдоль отрицательного направления оси Z. Боковые плоскости пирамиды проходят через стороны экрана монитора.

В пространстве может содержаться множество объектов — одни будут позади камеры, другие слева или справа, вне области видимости. Естественно, такие объекты не должны участвовать в рендеринге. Поэтому цель применения ПСКО - определить, какой объем пространства попадёт на экран и каким образом он будет спроецирован. Этот объем ограничивается шестью плоскостями (frustum_planes) - передняя, задняя, левая, правая, верхняя, нижняя (top, bottom, left, right, far, near) - и называется пирамида видимости (frustum):

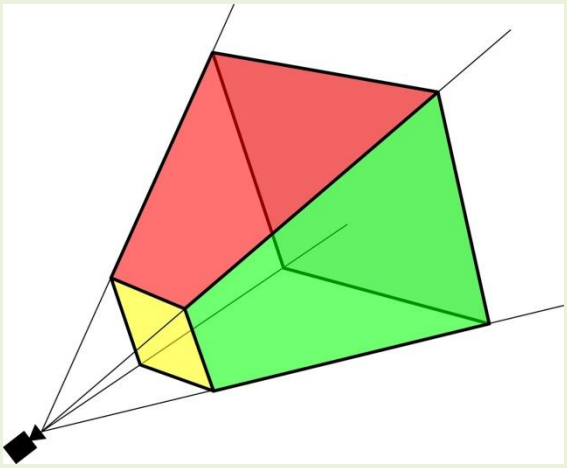

Рис. 7.6

Эти шесть плоскостей заложены в проекционной матрице. Вершины, лежащие за пределами пирамиды видимости, отсекаются и не участвуют в дальнейшей обработке. Кроме того, форма пирамиды видимости задает тип проекции 3D-сцены на 2D-экран.

Если передняя и задняя плоскость будут одинаковых размеров (они отодвинуты вдоль оси Z бесконечно далеко от наблюдателя), то получится ортографическая проекция (в этом случае объекты вблизи и вдалеке будут иметь одинаковые размеры, что создает не очень реалистичную картину). В противном случае будет использована перспективная проекция (именно так мы видим мир в реальной жизни). Перспективное проецирование применяется, чтобы создать у наблюдателя ощущение глубины на проекционном плане, матрица перспективного преобразования должна отобразить пространство пирамиды видимости в нормализованное пространство куба видимости.

В WebGL предполагается, что камера всегда расположена в координатах (0, 0, 0) и направлена в отрицательную сторону оси Z и вместо перемещения камеры перемещаются объекты таким образом, чтобы они находились перед камерой. За этот перенос (и поворот с масштабированием) отвечает матрица вида. Но в WebGL нет отдельной матрицы вида (как и матрицы модели, которая должна располагать объект в мировом пространстве). Вместо этого есть одна матрица модель-вид, совмещающая в себе матрицу модели и матрицу вида.

5. **Экранная** система координат (ЭСК) – это двумерная система координат, которая получается при выводе изображения на экран. Плоскость ЭСК совпадает с передней плоскостью пирамиды видимости ПСКО.

То есть на данном этапе пирамида видимости проецируется на переднюю плоскость для получения 2D-координат - именно это изображение в дальнейшем станет тем, что мы увидим на экране. Нам уже известно, что для получения декартовых координат необходимо разделить x, y и z на w (это называется перспективное деление). После деления мы получаем три компоненты: координаты x и y определяют положение точки на плоскости, а координата z отвечает за глубину точки, что помогает WebGL определить, какие объекты находятся ближе к наблюдателю и попадут на экран, а какие будут перекрыты и не будут отображены.

Экранная система координат сохраняется при **визуализизации изображения** (при этом нормализованная ЭСК приводится к реальной). На этом этапе всех преобразований экранные координаты выводятся на canvas (который в частности не обязательно должен быть квадратным). В отличие от предыдущих преобразований здесь не используется матрица. За визуализацию в WebGL отвечает функция viewport.

Таким образом последовательность получения изображения трехмерной сцены на экране:

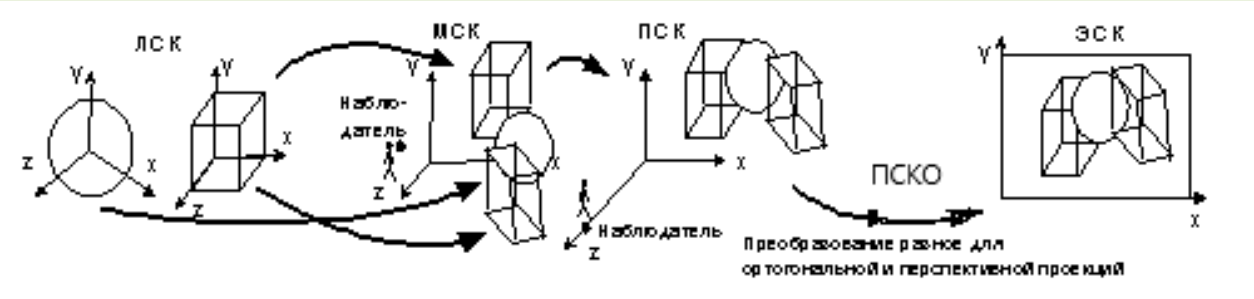

Рис. 7.7

1. Установка объектов в локальных системах координат.
2. Преобразование объектов из локальных систем координат в мировую систему координат.
3. Преобразование сцены из мировой системы координат в пользовательскую.
4. Преобразование сцены в пользовательскую систему координат с удалением лишних объектов и определением вида проецирования.
5. Преобразование из пользовательской системы координат **отсечения** в экранную.
6. Визуализация сцены на экране.

Преобразования координат *ЛСК→МСК→ПСК→ПСКО→ЭСК→Визуализация* можно изобразить в виде схемы:

Рис. 7.8

В WebGL часть преобразований объединяется в одну операцию, и цепочка преобразований выглядит так: *ЛСК →ПСК → ЭСК → Визуализация.*

Тогда мы получим такую схему:

Рис. 7.9

То есть при реализации в WebGL пять преобразований координта заменяются на три.

7.6. Матричные операции

Двухмерные геометрические преобразования - поворот, перенос и масштабирование – в графических программах выполняются с применением операций матричной алгебры. Для общности в нашем коде будет использоваться размерность матриц 4x4 несмотря на то, что будут рассматриваться двухмарные задачи; координата Z всегда будет равна нулю.

При умножении матрицы 4x4 на вектор координат точки объекта получим:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ j & k & l & m \\ n & p & q & r \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} a*x+b*y+c*z+d*1 \\ e*x+f*y+g*z+h*1 \\ j*x+k*y+l*z+m*1 \\ n*x+p*y+q*z+r*1 \end{bmatrix}$$

Если матрица преобразования будет единичной, после преобразования вектор координат не изменится:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1*x+0*y+0*z+0*1 \\ 0*x+1*y+0*z+0*1 \\ 0*x+0*y+1*z+0*1 \\ 0*x+0*y+0*z+1*1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Применение матрицы переноса, которая будет перемещать объект:

$$\begin{bmatrix} 1 & 0 & 0 & tX \\ 0 & 1 & 0 & tY \\ 0 & 0 & 1 & tZ \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1*x+0*y+0*z+tX*1 \\ 0*x+1*y+0*z+tY*1 \\ 0*x+0*y+1*z+tZ*1 \\ 0*x+0*y+0*z+1*1 \end{bmatrix} = \begin{bmatrix} x+tX \\ y+tY \\ z+tZ \\ 1 \end{bmatrix}$$

То есть мы получим те же уравнения переноса:

X’ = X + tX
Y’ = Y + tY

Применение матрицы масштабирования:

$$\begin{bmatrix} sX & 0 & 0 & 0 \\ 0 & sY & 0 & 0 \\ 0 & 0 & sZ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} sX*x+0*y+0*z+0*1 \\ 0*x+sY*y+0*z+0*1 \\ 0*x+0*y+sZ*z+0*1 \\ 0*x+0*y+0*z+1*1 \end{bmatrix} = \begin{bmatrix} sX*x \\ sY*Y \\ sZ*Z \\ 1 \end{bmatrix}$$

Применение матрицы поворота вокруг оси Z:

$$\begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta*x-sin\theta*y+0*z+0*1 \\ sin\theta*x+cos\theta*y+0*z+0*1 \\ 0*x+0*y+1*z+0*1 \\ 0*x+0*y+0*z+1*1 \end{bmatrix} = \begin{bmatrix} cos\theta*x-sin\theta*y \\ ysin\theta*x+cos\theta*y \\ z \\ 1 \end{bmatrix}$$

С учетом того, что все геометрические преобразования, включая комбинированные, выполняются с помощью одной и той же матричной операции, код вершинного шейдера будет для всех преобразований выглядеть одинаково:

```
attribute vec3 aVertexPosition;
uniform mat4 uMVMatrix;
void main(void) {  gl_Position = uMVMatrix * vec4(aVertexPosition, 1.0);  }
```

То есть нам не нужно менять вершинный шейдер при применении нового преобразованя или изменении порядка применения преобразований. Но часть вычислений перешла в код javascript. Рассмотрим все изменения по порядку.

В initShaders нам больше не нужны uTranslation, uScale и uAngle, вместо них мы будем использовать только uMVMatrix:

```
shaderProgram.mvMatrix = gl.getUniformLocation(shaderProgram, "uMVMatrix");
gl.uniformMatrix4fv(shaderProgram.mvMatrix, false, createIdentityMatrix());
```

Каждое преобразование будем храниться в отдельной матрице, каждой из которых в качестве начального значения задается единичная матрица:

```
var mTranslation = createIdentityMatrix();
var mScale = createIdentityMatrix();;
var mRotation = createIdentityMatrix();;
```

Функцию для создания единичной матрицы - createIdentityMatrix.

Также изменилась функция обработки событий мыши. Теперь мы записываем каждое преобразование в соответствующую матрицу, а в конце формируем итоговую матрицу перемножением всех матриц преобразований:

```
function onMouseMove(evt) {
  if (mousePressed) {
    var diffX = evt.clientX - dragStartX;
    var diffY = dragStartY - evt.clientY;
    if (currentEvent === eventType.drag) {
       dragOffset = [diffX * 2 / 500, diffY * 2 / 500];
       var finalTranslation = [translation[0] + dragOffset[0], translation[1] +
           dragOffset[1]];
       mTranslation = createTranslationMatrix(finalTranslation[0], finalTranslation[1]);
      } else if (currentEvent === eventType.scale) {
         dragOffset = [diffX / 100, diffY / 100];
         var finalScale = [scale[0] + dragOffset[0], scale[1] + dragOffset[1]]
         mScale = createScaleMatrix(finalScale[0], finalScale[1]);
      } else if (currentEvent === eventType.rotate) {
         dragOffset = diffX;
         mRotation = createRotationMatrix(rotation + dragOffset);
      }
      var mvMatrix = multiplyMatrices(mScale, mRotation);
      mvMatrix = multiplyMatrices(mvMatrix, mTranslation);
      gl.uniformMatrix4fv(shaderProgram.mvMatrix, false, mvMatrix);
   }
}
```

За умножение матриц отвечает функция multiplyMatrices:

```
function multiplyMatrices(m1, m2) {
   return [
      m1[0] * m2[0] + m1[1] * m2[4] + m1[2] * m2[8] + m1[3] * m2[12],
      m1[0] * m2[1] + m1[1] * m2[5] + m1[2] * m2[9] + m1[3] * m2[13],
      m1[0] * m2[2] + m1[1] * m2[6] + m1[2] * m2[10] + m1[3] * m2[14],
      m1[0] * m2[3] + m1[1] * m2[7] + m1[2] * m2[11] + m1[3] * m2[15],

      m1[4] * m2[0] + m1[5] * m2[4] + m1[6] * m2[8] + m1[7] * m2[12],
      m1[4] * m2[1] + m1[5] * m2[5] + m1[6] * m2[9] + m1[7] * m2[13],
      m1[4] * m2[2] + m1[5] * m2[6] + m1[6] * m2[10] + m1[7] * m2[14],
```

```
        m1[4] * m2[3] + m1[5] * m2[7] + m1[6] * m2[11] + m1[7] * m2[15],

        m1[8] * m2[0] + m1[9] * m2[4] + m1[10] * m2[8] + m1[11] * m2[12],
        m1[8] * m2[1] + m1[9] * m2[5] + m1[10] * m2[9] + m1[11] * m2[13],
        m1[8] * m2[2] + m1[9] * m2[6] + m1[10] * m2[10] + m1[11] * m2[14],
        m1[8] * m2[3] + m1[9] * m2[7] + m1[10] * m2[11] + m1[11] * m2[15],

        m1[12] * m2[0] + m1[13] * m2[4] + m1[14] * m2[8] + m1[15] * m2[12],
        m1[12] * m2[1] + m1[13] * m2[5] + m1[14] * m2[9] + m1[15] * m2[13],
        m1[12] * m2[2] + m1[13] * m2[6] + m1[14] * m2[10] + m1[15] * m2[14],
        m1[12] * m2[3] + m1[13] * m2[7] + m1[14] * m2[11] + m1[15] * m2[15]
    ];
}
```

Функция умножает одну матрицу 4x4 на другую, в результате чего также получается матрица 4x4. Код, который отвечает за создание матриц преобразований:

```
// единичная матрица
function createIdentityMatrix() {
    return [
        1, 0, 0, 0,
        0, 1, 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1
    ];
}
// матрица переноса
function createTranslationMatrix(tx, ty) {
    return [
        1, 0, 0, 0,
        0, 1, 0, 0,
        0, 0, 1, 1,
        tx, ty, 0, 1
    ];
}
// матрица масштабирования
function createScaleMatrix(sx, sy) {
    return [
        sx, 0,  0, 0,
        0,  sy, 0, 0,
        0,  0,  1, 0,
        0,  0,  0, 1
    ];
}
// матрица поворота по оси Z
function createRotationMatrix(angle) {
    var angleRad = angle * Math.PI / 180;
    var c = Math.cos(angleRad);
    var s = Math.sin(angleRad);
    return [
        c, -s, 0, 0,
        s,  c, 0, 0,
```

```
      0,  0, 1, 0,
      0,  0, 0, 1
   ];
}
```

В библиотеке gl-matrix есть весь необходимый функционал матричных преобразований и она оптимизирована в плане быстродействия.

Полный код программы, выполняющей геометрические преобразования с помощью матриц – в файле **ex_07_04.html**.

Наша программа выполняет те же двухмерные геометрические преобразования того же плоского объекта, но с применением матричных операций в однородном пространстве:

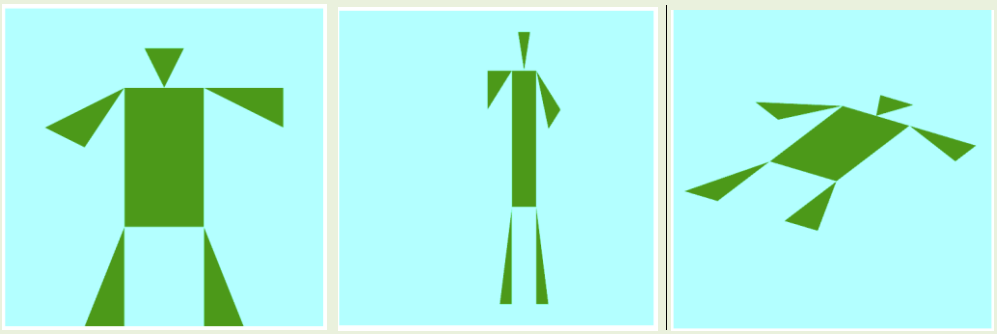

Рис. 7.10

Вновь пользователь может сдвинуть изображение объекта кнопкой мыши, выполнить его масштабирование – shift+мышь, выполнить поворот – ctrl+мышь.

Контрольные вопросы

1. Геометрические проеобразования.
2. Однородное пространство.
3. Системы координат в WebGL.