УДК 004.93

Н.С. СВИРНЕВСКИЙ
Хмельницкий национальный университет

РАЗРАБОТКА WEB-СИСТЕМЫ АФФИННЫХ ПРЕОБРАЗОВАНИЙ ПРОСТРАНСТВА НА ОСНОВЕ WEBGL ТЕХНОЛОГИИ

В статье описывается архитектура программы для решения задач, связанных с аффинными преобразованиями пространства на основе WebGL – технологии, которая обеспечивает 3D-графику в браузере. Три кита лежат в основе WebGL технологии – HTML, JavaScript и OpenGL. Посредством языка шейдеров OpenGL реализуется обработка каждой вершины и нахождения её координат с учетом матриц модели и проекции, причем непосредственно на GPU (Graphic Processor Unit). GPU принимает на вход описание трехмерной сцены в виде массивов вершин и треугольников, а также параметры наблюдателя и по ним строится на экране двумерное изображение сцены для этого наблюдателя. На примере задачи сканирования рупорной антенны продемонстрирована эффективность для архитектуры WebGL приложения использования матрицы композиции аффинных преобразований, определяемой через последовательность перемножения матриц элементарных преобразований.

Ключевые слова: WebGL, 3D-графика, композиция, матрица, аффинные преобразования, антенна.

N.S. SVIRNEVSKYI
Khmelnytskyi National University

DEVELOPMENT OF WEB-SYSTEM OF AFFINE TRANSFORMATIONS OF SPACE BASED ON WEBGL TECHNOLOGY

This article describes the architecture of the program for solving problems related to affine transformations of space based on WebGL - a technology that provides 3D graphics in the browser. Three whales are the basis of WebGL technology - HTML, JavaScript and OpenGL. Using the OpenGL shader language, each vertex is processed and its coordinates are taken into account, taking into account the model and projection matrices, and directly to the GPU (Graphic Processor Unit). GPU takes on the input a description of a three-dimensional scene in the form of arrays of vertices and triangles, as well as the parameters of the observer and a two-dimensional image of the scene for this observer is constructed on the screen. The example of the problem of scanning antenna demonstrates the efficiency for the WebGL application of using the composition matrix of affine transformations, determined through the sequence of multiplication of matrices of elementary transformations.

Keywords: WebGL, 3D-graphics, composition, matrix, affine transformations, antenna.

Введение

WebGL (Web-based Graphics Library) – технология, которая обеспечивает 3D-графику в браузере. Три кита лежат в основе WebGL технологии – HTML, JavaScript и OpenGL ES 2.0.

Анализ исследований и публикаций

В предыдущих публикациях [1–3] автора этой статьи на основе достаточно наглядного и простого практического примера (сканирование рупорной антенны) демонстрируется механизм аффинных преобразований в пространстве. Рассматриваются прямая задача (построение изображений рупорной антенны)[1] и обратная – восстановление параметров рупорной антенны по одному ее изображению [2] и реконструкция трехмерной модели по 2-м изображениям [3]. Эти задачи реализованы на платформе шаблона C++WinAPI приложения. Реализация механизма аффинных преобразований для других платформ, включая приложения для WEB, имеет ряд специфических особенностей, что требует рассматривать подобные задачи по новому, учитывая эти особенности.

Формулирование цели

Разработать архитектуру приложения для WEB с использованием механизма аффинных преобразований пространства на основе WebGL технологии.

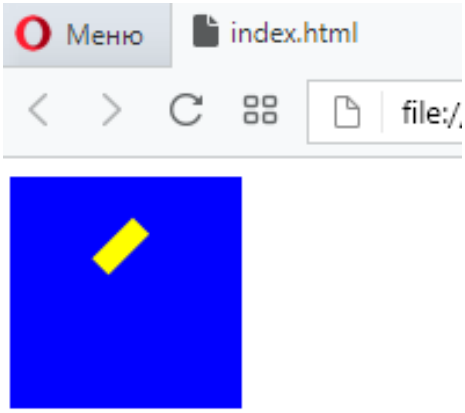


Рис. 1. Рисунок прямоугольника в Web-браузере

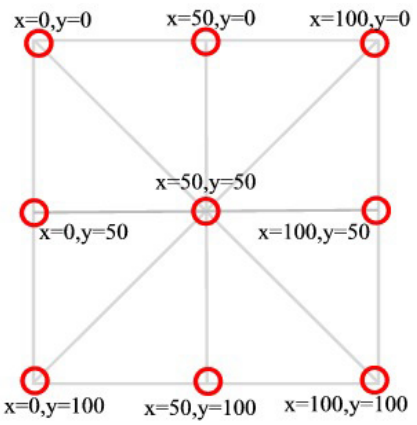


Рис. 2. Схема координатного пространства элемента canvas

Изложение основного материала

Вводная часть. Изложение теории будет идти параллельно с практической реализацией с переходом от простых примеров создания изображений на плоскости к более сложным примерам с 3D-графикой. Ниже приведен простейший код приложения, который обеспечивает 2D-преобразования и рисование прямоугольника в Web-браузере (рис. 1).

Файл index.html

```
<html>
     <head>
         <script src="draw.js" type="text/javascript"> </script>
     </head>
     <body onload="draw_b();">
  <canvas  id="b"  width="100" height="100" >  </canvas>
     </body>
 </html>
```

Файл draw.js

```
function draw_b() {
  var b_canvas = document.getElementById("b");
  var b_context = b_canvas.getContext("2d");
  b_context.fillStyle = "#0000FF";
  b_context.fillRect(0, 0, b_canvas.width, b_canvas.height);
  b_context.fillStyle = "#FFFF00";
  //b_context.rotate(-Math.PI/4);
  //b_context.translate(30,10);
  b_context.setTransform(0.707, -0.707, 0.707, 0.707, 0, 0);
  b_context.fillRect(0, 50, 25, 10);
}
```

В файле index.html на языке HTML определен элемент canvas – растровый холст на Web странице (прямоугольная двумерная сетка). Измерения пространственной области элемента canvas по ширине и высоте задаются в пикселах (рис.2). Верхний левый угол области Canvas имеет координаты x=0, y=0.

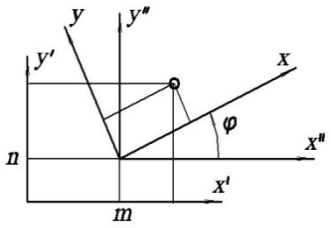
В файле draw.js на языке JavaScript получаем доступ по атрибуту id к элементу canvas и затем к объекту context, где определены методы преобразования пространства и свойства рисования. Метод setTransform (a, b, d, e, c, f) обеспечивает поворот плоскости элемента canvas на угол 45 градусов. Вызов метода fillRect() рисует прямоугольник и заполняет его текущим цветом заливки. Прямоугольник задается левым верхним углом (0, 50), шириной (25) и высотой (10) в направлении слева направо и сверху вниз.

Матричные преобразования

Многие веб-разработчики игнорируют матрицу аффинных преобразований, полагая её слишком сложной для понимания и используя взамен простейшие функции для трансформации типа rotate и translate (в коде они закомментированы). И совершенно зря, матрица преобразований обладает широкими возможностями, вдобавок, в том или ином виде поддерживаются всеми браузерами, а значит, её применение даёт кроссбраузерный код. Сама матрица имеет размер 3x3 и в общем виде записывается так:

$$\begin{pmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{pmatrix}$$

Роль каждого коэффициента матрицы становится понятна из следующего. Рассмотрим схему и уравнения для пересчета координат точки в новой СК [4].



$$\begin{cases} x' = x'' + m = x\cos\varphi + y\sin\varphi + m \\ y' = y'' + n = x\cos\varphi + y\sin\varphi + n \end{cases}$$

Рис. 3. Пересчет координат точки в новой СК

В матричном представлении уравнения запишутся так:

$$(x'\ y') = (x\ y)\begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix} + (m\ n)$$

Обобщим формулы преобразований движения к формулам аффинного преобразования плоскости:

$$(x'\ y') = (x\ y)\begin{pmatrix} a & d \\ b & e \end{pmatrix} + (c\ f)$$

Не так уж часто мы выполняем только одно элементарное преобразование; обычно в приложении

требуется, чтобы мы создали сложное преобразование из нескольких элементарных. Например, нам может понадобиться переместить объект на вектор (3,-4), затем повернуть на 30°, затем масштабировать с помощью множителя (2,-1). Основное преимущество объединенных преобразований состоит в том, что к точке более эффективно применять одно результирующее преобразование, чем ряд преобразований друг за другом. В случае последовательного выполнения любой комбинации операций вращения и масштабирования и отражения результат легко можно записать в виде произведения матриц соответствующих преобразований. Очевидно, что удобнее применять результирующую матрицу (композицию матриц) вместо того, чтобы каждый раз заново вычислять произведение матриц. Однако таким способом нельзя получить результирующую матрицу преобразования, если среди последовательности преобразований присутствует хотя бы один сдвиг (перенос), поскольку операция переноса реализуется отдельно (с помощью сложения). Было бы хорошо иметь математический аппарат, позволяющий включать в композиции преобразований и операцию переноса. Однородные координаты и есть этот математический аппарат. Теперь точки плоскости можно описывать трехэлементным вектором, а матрицы преобразования должны иметь размер 3×3:

$$\begin{aligned} x' &= ax + by + c \\ y' &= dx + ey + f \\ 1' &= 0x + 0y + 1 \end{aligned}; \qquad (x'\ y'\ 1) = (x\ y\ 1)\begin{pmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{pmatrix}$$

Преимущество такого подхода (матричных формул) заключается в том, что совмещение последовательных элементарных преобразований при этом значительно упрощается. Например, два и более поворота можно записать в виде одной матрицы суммарного поворота:

$$R(\alpha)R(\beta) = \begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} \cos\beta & \sin\beta & 0 \\ -\sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos(\alpha+\beta) & \sin(\alpha+\beta) & 0 \\ -\sin(\alpha+\beta) & \cos(\alpha+\beta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Результирующая матрица, полученная произведением двух исходных матриц преобразования, представляет собой совмещение элементарных преобразований. Независимо от количества элементарных преобразований в последовательности, можно всегда произвести совмещение так, чтобы только одна матрица 3×3 представляла всю последовательность преобразований.

Еще больший эффект использования матрицы аффинных преобразований можно получить взамен простейших функций для трансформации при создании 3D-графики на Web-странице. Переход из одной прямолинейной координатной системы к другой описывается в общем случае системой уравнений с соответствующим матричным представлением:

$$\begin{aligned} x^* &= \alpha_1 x + \alpha_2 y + \alpha_3 z + \lambda \\ y^* &= \beta_1 x + \beta_2 y + \beta_3 z + \mu \\ z^* &= \gamma_1 x + \gamma_2 y + \gamma_3 z + \nu \end{aligned}; \qquad (x^*\ y^*\ z^*\ 1) = (x\ y\ z\ 1)\,A$$

$$A = \begin{pmatrix} \alpha_1 & \beta_1 & \gamma_1 & 0 \\ \alpha_2 & \beta_2 & \gamma_2 & 0 \\ \alpha_3 & \beta_3 & \gamma_3 & 0 \\ \lambda & \mu & \nu & 1 \end{pmatrix}$$

Архитектура WebGL приложения

Графические приложения наиболее нуждаются оптимизации, особенно в Web программировании. Изучая каждую функцию OpenGL, стоит задуматься, как вызов этой функции выполняется и сколько времени он может занять. Возможности OpenGL определяются аппаратурой, которая, как известно, имеет тенденцию развиваться. В соответствии с этим в OpenGL добавляются новые функции. К примеру, с появлением GPU (Graphic Processor Unit) весь функционал с последовательной передачей и обработкой каждой вершины примитива практически перестал использоваться. Взамен GPU принимали на вход описание трехмерной сцены в виде массивов вершин и треугольников, а также параметры наблюдателя, и строили по ним на экране двумерное изображение сцены для этого наблюдателя. По мере совершенствования GPU появилась возможность хранить вершины и их атрибуты в различных буферах – непосредственно на GPU.

Последними тенденциями в графической аппаратуре является замена фиксированной функциональности конвейера обработки функций OpenGL программируемостью в тех областях, которые стали слишком сложны. В OpenGL в версии 2.0 добавляется поддержка высокоуровневого шейдерного языка GLSL (OpenGL Shading Language – GLSL). Посредством языка шейдеров OpenGL фиксированная функциональность этапов обработки вершин и фрагментов в конвейере OpenGL заменяется программируемостью, которая позволяет выполнять все, что может фиксированная функциональность и даже намного больше.

Шейдерные программы пишутся на специально разработанных языках, одним из которых является язык GLSL(OpenGL Shading Language), который полностью стал частью OpenGL, начиная с версии 2.0. Синтаксис GLSL основан на языках программирования семейства C. Он включает почти все

распространенные в C/C++ арифметические, логические и битовые операторы, конструкторы для инициализации и прочее.

Программа обычно состоит из двух шейдеров: одного вершинного и одного фрагментного. Может присутствовать и более одного шейдера каждого типа, но должна быть только одна функция main() среди всех фрагментных шейдеров и одна – среди всех вершинных. Обычно самый простой путь заключается в создании одного шейдера каждого типа.

Вершинные шейдеры предназначены для обработки каждой вершины и нахождения её координат с учетом матриц модели и проекций, а также прочих, зависящих от задумки программиста, условий. Для этого вершинный шейдер получает все параметры вершины (координаты, цвет, текстурные координаты и т.д.). Также основная программа может передать шейдеру любые другие, определяемые программистом параметры, включая совершенно произвольные, имеющие смысл только для выполнения общей задачи. Например, шейдеру может быть передан параметр «сила ветра», который совершенно не привязан ни к какому способу представления трехмерных объектов, ни к какому графическому API. Это просто параметр, который будет обрабатываться внутри шейдера.

Пиксельные шейдеры имеют своей целью расчет цвета каждого пикселя и значения глубины (обычно, глубину рассчитывать необязательно) или вынесение решения о том, что пиксель выводит не надо. Пиксельный шейдер может получать на вход текстурные координаты, соответствующие обрабатываемой точке, примешиваемый цвет и, возможно, некоторые другие параметры. На основании этих данных он рассчитывает цвет точки и завершает работу. Пиксельный шейдер вызывается при обработке каждой двумерной точки. Необходимо четко понимать, что шейдеры работают раздельно друг от друга, ими можно подменить не все части рендеринга и это накладывает значительные ограничения на возможные в реализации шейдерами алгоритмы.

Шейдеры взаимодействуют с фиксированной частью конвейера OpenGL, записывая значения во встроенные переменные OpenGL, которые имеют префикс "gl_". Вершинный шейдер ответственен как минимум за одну переменную: gl_Position, и обычно трансформирует вершину в матрицах проекции и моделей. Обязательной работой для фрагментного шейдера является запись цвета фрагмента, в встроенную переменную gl_FragColor. Вершинный шейдер выполняется один раз для каждой вершины, а фрагментный – один раз для каждого фрагмента. Несколько исполнений одного и того же шейдера могут проходить параллельно.

В каждый момент выполнения программы активна лишь одна пара, состоящая из вершинного и пиксельного шейдера. В случае, если программист не установил активной пару шейдеров, работает стандартный шейдер, который обеспечивает всю стандартную функциональность (конвейер) графической библиотеки OpenGL. Обычно пару вершинного и фрагментного шейдеров называют просто шейдером.

Для создания WebGL-приложения на базе WebGL API необходимо:

1. Создать на HTML5 элемент canvas.
2. Получить контекст элемента canvas.
3. Инициализировать рабочую область отображения графики.
4. Инициализировать шейдер с параметрами.
5. Создать буфер данных вершин модели.
6. Создать матрицу преобразования вершин модели на экран.
7. Нарисовать графику.

Ниже приведен простейший код WebGL приложения, который обеспечивает рисование 2-х прямоугольников – белого на фоне черного.

Файл index.html

```
<html>
     <head>
         <script src="draw.js" type="text/javascript"> </script>
     </head>
     <body onload="onLoad();">
         <canvas id="webgl"  width="500" height="500"></canvas>
     </body>
</html>
```

Файл draw.js

```
function onLoad() {
              var canvas = document.getElementById("webgl");
              var gl = initWebGL(canvas);
              initViewport(gl, canvas);
              initShader(gl);
              initMatrices();
              var model = createModel(gl);
              draw(gl, model);
             }

function initWebGL(canvas) {
    gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
        return gl;
    }
```

```
        function initViewport(gl, canvas) {
                   gl.viewport(0, 0, canvas.width, canvas.height);
                   }

        var    shaderProgram,    shaderVertexPositionAttribute,    shaderProjectionMatrixUniform,
shaderModelViewMatrixUniform;
        function initShader(gl) {
          var fragmentShader = createShader(gl, fragmentShaderSource, "fragment");
          var vertexShader = createShader(gl, vertexShaderSource, "vertex");
          shaderProgram = gl.createProgram();
          gl.attachShader(shaderProgram, vertexShader);
          gl.attachShader(shaderProgram, fragmentShader);
          gl.linkProgram(shaderProgram);
          shaderVertexPositionAttribute = gl.getAttribLocation(shaderProgram, "vertexPos");
          gl.enableVertexAttribArray(shaderVertexPositionAttribute);
          shaderProjectionMatrixUniform = gl.getUniformLocation(shaderProgram, "projectionMatrix");
          shaderModelViewMatrixUniform = gl.getUniformLocation(shaderProgram, "modelViewMatrix");
          if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
              alert("Could not initialise shaders");
           }
        }

        function createShader(gl, str, type) {
           var shader;
           if (type == "fragment") {
               shader = gl.createShader(gl.FRAGMENT_SHADER);
               }
               else if (type == "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;
        }

        function initMatrices()
        {
           modelViewMatrix = new Float32Array(
             [1, 0, 0, 0,
              0, 1, 0, 0,
              0, 0, 1, 0,
              0, 0, -5, 1]);
           projectionMatrix = new Float32Array(
        [1, 0, 0, 0,
              0, 1, 0, 0,
              0, 0, 0, 0,
              0, 0, 0, 1]);
        }

        function createModel(gl) {
               var vertexBuffer;
               vertexBuffer = gl.createBuffer();
               gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
               var verts = [
                            .5, .5, 0.0,
                            -.5, .5, 0.0,
                            .5, -.5, 0.0,
                            -.5, -.5, 0.0
                            ];
               gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(verts), gl.STATIC_DRAW);
               var square = {buffer:vertexBuffer, vertSize:3, nVerts:4, primtype:gl.TRIANGLE_STRIP};
               return square;
            }

        function draw(gl, obj) {
           gl.clearColor(0.0, 0.0, 0.0, 1.0);
           gl.clear(gl.COLOR_BUFFER_BIT);
           gl.bindBuffer(gl.ARRAY_BUFFER, obj.buffer);
           gl.useProgram(shaderProgram);
         gl.vertexAttribPointer(shaderVertexPositionAttribute, obj.vertSize, gl.FLOAT, false, 0, 0);
           gl.uniformMatrix4fv(shaderProjectionMatrixUniform,false, projectionMatrix);
           gl.uniformMatrix4fv(shaderModelViewMatrixUniform,false, modelViewMatrix);
           gl.drawArrays(obj.primtype, 0, obj.nVerts);
        }
```

```
var vertexShaderSource =
" attribute vec3 vertexPos;\n" +
" uniform mat4 modelViewMatrix;\n" +
" uniform mat4 projectionMatrix;\n" +
" void main(void) {\n" +
"gl_Position = projectionMatrix * modelViewMatrix * vec4(vertexPos, 1.0);\n" +
" }\n";

var fragmentShaderSource =
" void main(void) {\n" +
" gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);\n" +
"}\n";
```

WebGL приложение управления положением рупорной антенны

Ниже рассмотрена техника использования матрицы аффинных преобразований при создании 3D-графики на Web-странице (рис.4) на примере выполнения геометрических преобразований рупорной антенны через управление изменением любого из 5 параметров ее положения (рис. 5) перемещением мышки (событие "mousemove"). В программе, описанной ниже, при перемещении мышки меняются 2 угла – альфа и бета.

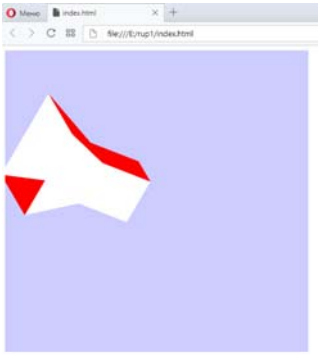


Рис. 4. Пример реализации 3D-графики на Web-странице

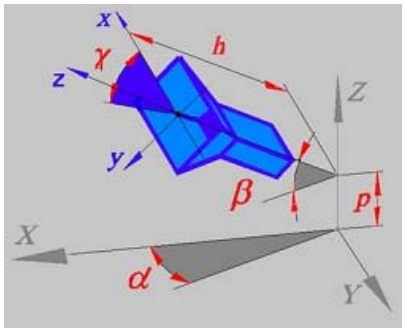


Рис. 5. Параметры положения рупорной антенны

Через OpenGL предоставляется возможность описать модель в глобальной системе координат XYZ. Результирующая матрица преобразований (рис. 6) от начального положения антенны, при котором СК антенны совпадает с положением глобальной СК (рис. 7), до ее конечного положения в пространстве (см. рис. 5) определяется через последовательность перемножения матриц элементарных преобразований (табл. 1).

$$T_p = T_1 T_2 T_3 T_4 T_5 T_0$$

$$T_p = \begin{pmatrix} \cos\chi\cos\alpha + \sin\chi\sin\beta\sin\alpha & \sin\chi\cos\beta & 0 & 0 \\ -\sin\chi\cos\alpha + \cos\alpha\sin\beta\sin\alpha & \cos\chi\cos\beta & 0 & 0 \\ \cos\beta\sin\alpha & -\sin\beta & 0 & 0 \\ h\cos\beta\sin\alpha & -h\sin\beta + p & 0 & 1 \end{pmatrix}$$

Рис. 6. Результирующая матрица преобразований антенны

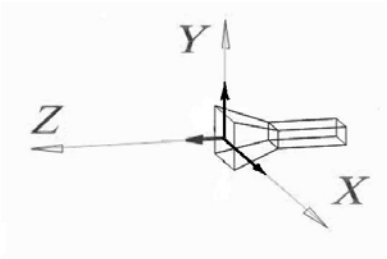


Рис. 7. Начальное положение антенны

Таблица 1

Матрицы элементарных преобразований антенны

1	2
Сдвиг на вдоль оси Z	Поворот вокруг оси Z
$T_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & h & 1 \end{pmatrix}$	$T_2 = \begin{pmatrix} \cos\chi & \sin\chi & 0 & 0 \\ -\sin\chi & \cos\chi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Продовження табл. '

1	2
Поворот вокруг оси X	Поворот вокруг оси Y
$T_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\beta & \sin\beta & 0 \\ 0 & -\sin\beta & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$T_4 = \begin{pmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Сдвиг вдоль оси Y	Проекция в направлении оси Z
$T_5 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & p & 0 & 1 \end{pmatrix}$	$T_0 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Ниже приведен код WebGL приложения, который обеспечивает визуализацию управления положением рупорной антенны (рис.8).

Файл index.html

```
< <html>
     <head>
         <script src="draw.js" type="text/javascript"> </script>
     </head>
     <body onload="onLoad();">
<canvas id="webgl"   width="500" height="500"></canvas>
<script id="fragmentShaderSource" type="x-shader/x-fragment">
precision mediump float;
varying vec4 vColor;
void main(void) {
gl_FragColor = vColor;
}
</script>
<script id="vertexShaderSource" type="x-shader/x-vertex">
attribute vec3 vertexPos;
attribute vec4 aVertexColor;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
varying vec4 vColor;
void main(void) {
gl_Position = projectionMatrix * modelViewMatrix * vec4(vertexPos, 1.0);
vColor = aVertexColor;
}
</script>
     </body>
 </html>
```

Файл draw.js

```
function onLoad() {
              var canvas = document.getElementById("webgl");
              var gl = initWebGL(canvas);
              initViewport(gl, canvas);
              initShader(gl);
              createModel(gl);
              drawScene();
              canvas.addEventListener("mousemove", mouseMoveEvent, false);
              canvas.addEventListener("mousedown", mouseDownEvent, false);
              canvas.addEventListener("mouseup",   mouseUpEvent, false);
             }
function initWebGL(canvas) {
    gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
        return gl;
    }
function initViewport(gl, canvas) {
           gl.viewport(0, 0, canvas.width, canvas.height);
           }
var     shaderProgram,     shaderVertexPositionAttribute,     shaderVertexColorAttribute,
shaderProjectionMatrixUniform, shaderModelViewMatrixUniform;
function initShader(gl) {
  var fragmentShader = createShader(gl, "fragmentShaderSource");
  var vertexShader = createShader(gl, "vertexShaderSource");
  shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);
```

```
          shaderVertexPositionAttribute = gl.getAttribLocation(shaderProgram, "vertexPos");
          gl.enableVertexAttribArray(shaderVertexPositionAttribute);
        shaderVertexColorAttribute = gl.getAttribLocation(shaderProgram, "aVertexColor");
        gl.enableVertexAttribArray(shaderVertexColorAttribute);
          shaderProjectionMatrixUniform = gl.getUniformLocation(shaderProgram, "projectionMatrix");
          shaderModelViewMatrixUniform = gl.getUniformLocation(shaderProgram, "modelViewMatrix");
          if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
              alert("Could not initialise shaders");
            }
        }
        function createShader(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 rightface;
        var colorsrightface;
        var downface;
        var colorsdownface;
        var leftface;
        var colorsleftface;
        var upperface;
        var colorsupperface;
        function createModel(gl) {
               rightface = gl.createBuffer();
               gl.bindBuffer(gl.ARRAY_BUFFER, rightface);
               var vertices = [ 0.40, 0.20, 0.0,
                             0.40, -0.20, 0.0,
                            0.20 ,0.10, -0.40,
                            0.20, -0.10, -0.40,
                            0.20, 0.10, -0.80,
                            0.20, -0.10, -0.80
                            ];
               gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
              rightface = {buffer:rightface, vertSize:3, nVerts:6, primtype:gl.TRIANGLE_STRIP};
colorsrightface = gl.createBuffer();
               gl.bindBuffer(gl.ARRAY_BUFFER, colorsrightface);
               var colors = []
               for (var i=0; i < 6; i++) {
                 colors = colors.concat([1.0, 0.0, 0.0, 1.0]);
               }
               gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
              colorsrightface      =      {buffer:colorsrightface,      vertSize:4,      nVerts:6,
primtype:gl.TRIANGLE_STRIP};
              var vertexBuffer2 = gl.createBuffer();
              gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer2);
               var vertices = [-0.40,-0.20,0.0,
                             0.40,-0.20,0.0,
                            -0.20,-0.10,-0.40,
                            0.20,-0.10,-0.40,
                            -0.20,-0.10,-0.80,
                             0.20,-0.10,-0.80
                            ];
              gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
              downface        =        {buffer:vertexBuffer2,        vertSize:3,        nVerts:6,
primtype:gl.TRIANGLE_STRIP};
               colorsdownface = gl.createBuffer();
               gl.bindBuffer(gl.ARRAY_BUFFER, colorsdownface);
```

```
               var colors = []
               for (var i=0; i < 6; i++) {
                 colors = colors.concat([1.0, 1.0, 1.0, 1.0]);
               }
               gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
               colorsdownface      =      {buffer:colorsdownface,      vertSize:4,      nVerts:6,
primtype:gl.TRIANGLE_STRIP};
              var vertexBuffer3 = gl.createBuffer();
              gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer3);
              var vertices = [ -0.40, 0.20, 0.0,
                           -0.40, -0.20, 0.0,
                          -0.20 ,0.10, -0.40,
                          -0.20, -0.10, -0.40,
                          -0.20, 0.10, -0.80,
                          -0.20, -0.10, -0.80
                          ];
              gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
              leftface        =        {buffer:vertexBuffer3,        vertSize:3,        nVerts:6,
primtype:gl.TRIANGLE_STRIP};
               colorsleftface = gl.createBuffer();
               gl.bindBuffer(gl.ARRAY_BUFFER, colorsleftface);
               var colors = []
               for (var i=0; i < 6; i++) {
                 colors = colors.concat([1.0, 0.0, 0.0, 1.0]);
               }
               gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
               colorsleftface      =      {buffer:colorsleftface,      vertSize:4,      nVerts:6,
primtype:gl.TRIANGLE_STRIP};
             var vertexBuffer4 = gl.createBuffer();
             gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer4);
             var vertices = [-0.40,0.20,0.0,
                          0.40,0.20,0.0,
                         -0.20,0.10,-0.40,
                          0.20,0.10,-0.40,
                         -0.20,0.10,-0.80,
                          0.20,0.10,-0.80
                          ];
              gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
              upperface        =        {buffer:vertexBuffer4,        vertSize:3,        nVerts:6,
primtype:gl.TRIANGLE_STRIP};
              colorsupperface = gl.createBuffer();
               gl.bindBuffer(gl.ARRAY_BUFFER, colorsupperface);
               var colors = []
               for (var i=0; i < 6; i++) {
                 colors = colors.concat([1.0, 1.0, 1.0, 1.0]);
               }
               gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
               colorsupperface     =     {buffer:colorsupperface,     vertSize:4,     nVerts:6,
primtype:gl.TRIANGLE_STRIP};
          }
        function drawScene() {
           gl.clearColor(0.8, 0.8, 1.0, 1.0);
           gl.enable(gl.DEPTH_TEST);
           gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT );
                     initMatrices();
                     draw(gl, rightface, colorsrightface);
                     draw(gl, downface, colorsdownface);
                     draw(gl, leftface, colorsleftface);
                     draw(gl, upperface, colorsupperface);
        }
        var A = 45.0;
        A = A * Math.PI / 180.0;
        var B = -45.0;
        B = B * Math.PI / 180.0;
        var C = 45.0;
        C = C * Math.PI / 180.0;
        var h = 1.0;
        var p = 0.0;
        var lastX = 0 , lastY = 0;
        var mouseState=false;
        function initMatrices()
        {
           modelViewMatrix = new Float32Array(
             [Math.cos(C)*Math.cos(A),  Math.sin(C)*Math.cos(B), -Math.sin(A)*Math.cos(C), 0,
              -Math.sin(C)*Math.cos(A), Math.cos(C)*Math.cos(B), Math.sin(C)*Math.sin(A), 0,
              Math.cos(B)*Math.sin(A),      -Math.sin(B),        Math.cos(B)*Math.cos(A), 0,
              h*Math.cos(B)*Math.sin(A), -h*Math.sin(B)+p,    h*Math.cos(B)*Math.cos(A), 1]);
           projectionMatrix = new Float32Array(
             [1, 0, 0, 0,
              0, 1, 0, 0,
              0, 0, 0, 0,
```

```
      0, 0, 0, 1]);
}
function draw(gl, obj, colobj) {
 gl.useProgram(shaderProgram);
 gl.bindBuffer(gl.ARRAY_BUFFER, obj.buffer);
 gl.vertexAttribPointer(shaderVertexPositionAttribute, obj.vertSize, gl.FLOAT, false, 0, 0);
 gl.bindBuffer(gl.ARRAY_BUFFER, colobj.buffer);
 gl.vertexAttribPointer(shaderVertexColorAttribute, colobj.vertSize, gl.FLOAT, false, 0, 0);
 gl.uniformMatrix4fv(shaderProjectionMatrixUniform,false, projectionMatrix);
 gl.uniformMatrix4fv(shaderModelViewMatrixUniform,false, modelViewMatrix);
 gl.drawArrays(obj.primtype, 0, obj.nVerts);
}
function mouseMoveEvent(e) {
       if (mouseState==true) {
          A-=(lastX-e.pageX)*0.01;
          B-=(lastY-e.pageY)*0.01;
       lastX = e.pageX;
       lastY = e.pageY;
    drawScene();
    }
}
function mouseDownEvent(e) {
       mouseState = true;
       lastX = e.pageX;
       lastY = e.pageY;
}
function mouseUpEvent(e) {
       mouseState = false;
}
```

Выводы

Разработана архитектура программы для решения задач, связанных с аффинными преобразованиями пространства на основе WebGL технологии. На примере задачи сканирования рупорной антенны продемонстрирована эффективность использования матрицы аффинных преобразований, определяемой через последовательность перемножения матриц элементарных преобразований.

Литература

1. Свирневский Н.С. Разработка архитектуры программы для решения задач, связанных с аффинными преобразованиями пространства / Н. С. Свирневский // Вісник Хмельницького національного університету. Технічні науки. – 2017. – № 3. – С. 176–185.

2. Свирневский Н.С. Восстановление параметров пространственного объекта по изображению / Н.С. Свирневский // Вісник Хмельницького національного університету. Технічні науки. – 2011. – № 6. – С. 74–78.

3. Свирневский Н.С. Алгоритм реконструкции трехмерной модели по изображениям / Н. С. Свирневский // Вісник Хмельницького національного університету. Технічні науки. – 2017. – № 2. – С. 212–218.

4. Свирневский Н.С. Основы разработки графических приложений / Н.С. Свирневский, С.С. Ковальчук. – Хмельницкий : ХНУ, 2015. – 270 с.

References

1. Svyrnevskyi N.S. Razrabotka arkhytektury prohrammy dlia reshenyia zadach, sviazannikh s affynnimy preobrazovanyiamy prostranstva [Tekst] / N. S. Svyrnevskyi // Herald of Khmelnytsky National University. – 2017. – № 3. – S. 176-185.

2. Svyrnevskyi N.S. Vosstanovlenye parametrov prostranstvennoho obiekta po yzobrazhenyiu [Tekst] / N.S. Svyrnevskyi // Herald of Khmelnytsky National University. – 2011. – № 6. – S. 74–78.

3. Svyrnevskyi N.S. Alhorytm rekonstruktsyy trekhmernoi modely po yzobrazhenyiam [Tekst] / N. S. Svyrnevskyi // Herald of Khmelnytsky National University. – 2017. – № 2. – S. 212-218.

4. Svyrnevskyi N.S. Osnovy razrabotky hrafycheskykh prylozhenyi / N.S. Svyrnevskyi, S.S. Kovalchuk. – Khmelnytskyi: KhNU, 2015. – 270 s.

Рецензія/Peer review : 07.08.2017 р. Надрукована/Printed :05.09.2017 р.
Рецензент: д.т.н., проф. Сорокатый Р.В.