Программирование графических приложений

Тема 8
Матрицы и геометрические преобразования трехмерных объектов

Цель изучения темы. Изучение методов формирования трехмерных графических объектов и использования матричных операций в WebGL.

8.1. Построение трехмерного объекта без использования библиотек

В первой теме уже приводились примеры программ для построения простого трехмерного объекта – куба. Однако там была использована библиотека Three.js, скрывающая в себе технику построения трехмерного изображения.

Далее будут изучаться программы формирования трехмрных изображений на примере того же куба без применения библиотек, что позволит строить затем на этой основе пространственные объекты произвольной конфигурации из вершин и рёбер.

Сначала рассмотрим программу построения каркасного куба. Куб определяется 8 точками в пространстве, которые мы зададим в виде последовательности вершин лицевой и задней граней. Их координаты укажем в массиве vertices:

```
var vertices =[
        // лицевая грань
        -0.5, -0.5, 0.5,   // x0, y0, z0,
        -0.5, 0.5, 0.5,   // x1, y1, z1,
         0.5, 0.5, 0.5,   // x2 y2, z2,
         0.5, -0.5, 0.5,   // x3, y3, z3,
        // задняя грань
        -0.5, -0.5, -0.5,   // x4, y4, z4,
        -0.5, 0.5, -0.5,   // x5, y5, z5,
         0.5, 0.5, -0.5,   // x6, y6, z6,
         0.5, -0.5, -0.5   // x7, y7, z7,
         ];
```

Вершины соединяются между собой линиями, составляющими рёбра куба. Последовательность соединения вершин определяется массивом индексов вершин, пронумерованных от 0 до 7:

```
var indices = [0, 1, 1, 2, 2, 3, 3, 0, 0, 4, 4, 5, 5, 6, 6,7, 7,4, 1, 5, 2, 6, 3, 7];
```

Рёбра рисуются знакомым уже методом gl.drawElements(gl.LINES, , ,).

Полный код программы (**ex08_01.html**):

```
<!DOCTYPE html>
<html>
<head>
<title>WebGL</title>
<meta charset="utf-8" />
</head>
<body>
<canvas id="canvas3D" width="400" height="300">Ваш браузер не поддерживает
        элемент canvas</canvas>
<!-- фрагментный шейдер -->
<script id="shader-fs" type="x-shader/x-fragment">
 void main(void) {     gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);   }
</script>
<!-- вершинный шейдер -->
<script id="shader-vs" type="x-shader/x-vertex">
 attribute vec3 aVertexPosition;
```

```
 void main(void) {     gl_Position = vec4(aVertexPosition, 1.0);   }
</script>
<script type="text/javascript">
var gl;
var shaderProgram;
var vertexBuffer; // буфер вершин
var indexBuffer; //буфер индексов
// установка шейдеров
function initShaders() {
   var fragmentShader = getShader(gl.FRAGMENT_SHADER, 'shader-fs');
   var vertexShader = getShader(gl.VERTEX_SHADER, 'shader-vs');
   shaderProgram = gl.createProgram();
   gl.attachShader(shaderProgram, vertexShader);
   gl.attachShader(shaderProgram, fragmentShader);
   gl.linkProgram(shaderProgram);
   if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
      alert("Не удалось установить шейдеры");
   }
   gl.useProgram(shaderProgram);
   shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
         "aVertexPosition");
   gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
}
// Функция создания шейдера
function getShader(type,id) {
   var source = document.getElementById(id).innerHTML;
   var shader = gl.createShader(type);
   gl.shaderSource(shader, source);
   gl.compileShader(shader);
   if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      alert("Ошибка компиляции шейдера: " + gl.getShaderInfoLog(shader));
      gl.deleteShader(shader);
      return null;
   }
   return shader;
}
 // установка буферов вершин и индексов
function initBuffers() {
   var vertices =[
            // лицевая грань
            -0.5, -0.5, 0.5,
            -0.5, 0.5, 0.5,
             0.5, 0.5, 0.5,
             0.5, -0.5, 0.5,
            // задняя грань
            -0.5, -0.5, -0.5,
            -0.5, 0.5, -0.5,
             0.5, 0.5, -0.5,
             0.5, -0.5, -0.5
             ];

   var indices = [0, 1, 1, 2, 2, 3, 3, 0, 0, 4, 4, 5, 5, 6, 6,7, 7,4, 1, 5, 2, 6, 3, 7];
```

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

Результат:

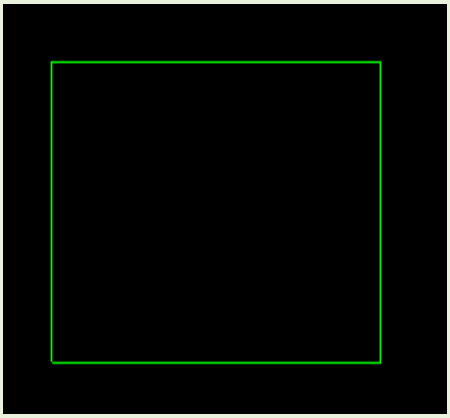

Рис. 8.1

Так как мы смотрим прямо на переднюю грань объекта, то мы будем видеть квадрат. Мы увидим каркас куба, если соответствующим образом изменим в приведенном коде координаты вершин.

Однако мы можем получить трехмерную проекцию куба, повернув уже готовый объект (заданный в своей локальной системе координат) или изменив точку обзора. Для выполнения этих операций нам понадобятся матрицы.

8.2. Проекционные матрицы в WebGL

Мировая матрица переводит локальные координаты объекта в глобальную систему координат с учетом различных преобразований, а ***матрица вида*** переводит глобальное пространство, в котором находятся объекты, в видимое пространство камеры. ***Матрица модели*** (Model-View Matrix) объединяет мировую матрицу и матрицу вида.

Камера в данном случае фактически и есть наш взгляд на трехмерную сцену. И поскольку камера направлена на переднюю грань, мы видели не куб, а просто квадрат. Но стоит нам переместить камеру, и мы увидим куб.

Матрица проекции (Projection Matrix) преобразует трехмерную систему координат объекта в двухмерные для отображения их на экране. Матрица проекции бывает двух типов: ортогональная и перспективная.

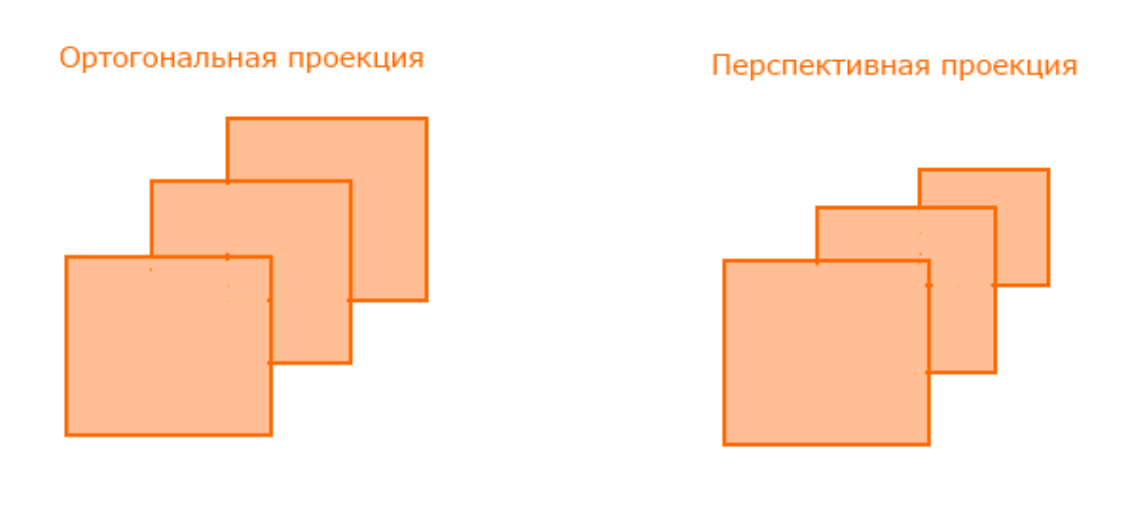

Рис. 8.2

При ортогональной проекции вне зависимости от значения координаты Z все объекты будут отрисовываться как есть с их текущими длиной и шириной. При перспективной проекции будет создаваться видимость глубины или отдаления: близлежащие объекты будут казаться больше, а те, которые находятся дальше, будут меньше, что повысит реалистичность сцены.

8.3. Использование библиотеки glMatrix

Поскольку javascript не имеет встроенных средств для работы с матрицами и векторами, то воспользуемся для этих целей сторонней библиотекой. Одной из наиболее используемых подобных библиотек является ***glMatrix***. Ее можно найти на официальном сайте разработчика: http://glmatrix.net/. Она относительно небольшая - минимизированная версия около 27 кБ, полная - 105 кБ.

Это не единственная библиотека javascript, которую можно использовать в WebGL для работы с матрицами; существуют и другие, например, Sylvester, WebGL-mjs.

Интерфейс библиотеки glMatrix может меняться с изменением версии, поэтому приложения с ее использованием, имеющиеся в интернете, могут не работать с обновленными

библиотеками. В этом случае следует обратиться к документации по библиотеке. Здесь будет использована версия glMatrix 2.0.

Все координаты и индексы будут такими же, как и в примере **ex08_01.html**, только теперь мы применим матрицы.

В программе подключается минимизированная версия библиотеки glMatrix. Сначала определяем ***матрицы модели*** и ***проекции***:

```
var mvMatrix = mat4.create(); // матрица модели
var pMatrix = mat4.create(); // матрица проекции
```

Метод mat4.create() как раз представляет метод библиотеки glMatrix, используемый для создания матрицы 4x4.

В функции установки шейдеров initShaders() создаем две матрицы - по одной для матрицы модели и проекции. Эти матрицы будут передаваться в вершинный шейдер и там применяться к координатам объекта:

```
shaderProgram.MVMatrix = gl.getUniformLocation (shaderProgram, "uMVMatrix");
shaderProgram.ProjMatrix = gl.getUniformLocation (shaderProgram, "uPMatrix");
```

Матрицы определяются как свойства в объекте shaderProgram с помощью метода gl.getUniformLocation(), который одинаков для обеих матриц.

Затем в отличие от предыдущего примера добавляются две новых функции: setupWebGL() и setMatrixUniforms(), которые вызываются в основной функции программы.

Функция setupWebGL() содержит код по настройке трехмерной сцены (установку матриц и т.д.). Вначале устанавливается ***матрица проекции***:

```
mat4.perspective (pMatrix, 1.04, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0);
```

Первый параметр функции - это и есть созданная ранее матрица проекции pMatrix, которая затем будет передаваться в шейдер. Второй параметр - 1.04 - это угол обзора в радианах. Третий параметр gl.viewportWidth/gl.viewportHeight задает отношение ширины к длине (aspect ratio). Четвертый и пятый параметры задают размеры видимой области - ближайшую и самую дальнюю ее точку.

Далее устанавливаем ***матрицу модели***. Сначала для нее устанавливаем ***матрицу идентичности***: mat4.identity(mvMatrix), а затем применяем к ней перемещение и вращение:

```
mat4.translate (mvMatrix,mvMatrix,[0, 0, -2.0]);
mat4.rotate (mvMatrix,mvMatrix, 1.9, [0, 1, 0]);
```

Для ***перемещения*** применяется метод mat4.translate(output, input, vec), где output - итоговая выходная матрица, которая получается после перемещения матрицы input на трехмерный вектор vec. Так как в нашем примере преобразуется одна матрица, параметры output и input совпадают. Третьим параметром идет вектор, на который выполняется перемещение: то есть в данном случае перемещение на 2 единицы по оси Z в сторону от наблюдателя.

Для ***вращения*** применяется метод mat4.rotate(output, input, rad, axis), где output - также итоговая матрица, которая получается поворотом матрицы input на угол rad (в радианах) вокруг оси axis. Поскольку в нашем примере для оси Y указана единица, а для других - нули, то вращение будет идти только относительно оси Y. Например, mat4.rotate(mvMatrix, mvMatrix, Math.PI/2, [1, 0, 0]) определяет матрицу mvMatrix поворота на 90 градусов вокруг оси X.

Мы можем не указывать ось вращения, а воспользоваться специальными методами отдельно для каждой координатной оси: mat4.rotateX(output, input, rad), mat4.rotateY(output, input, rad) и mat4.rotateZ(output, input, rad).

Кроме этих операций мы можем применить ***масштабирование*** с помощью метода mat4.scale(output, input, vec). Вектор vec указывает масштаб, на который изменяются значения матрицы input. Например, mat4.scale(mvMatrix, mvMatrix, [0.5, 1, 1]) определяет сжатие объекта по оси X (то есть уменьшение его ширины) в 2 раза.

Метод setMatrixUniforms() выполняет передачу матриц в вершинный шейдер:

```
gl.uniformMatrix4fv (shaderProgram.ProjMatrix,false, pMatrix);
gl.uniformMatrix4fv (shaderProgram.MVMatrix, false, mvMatrix);
```

То есть здесь мы связываем матрицу shaderProgram.ProjMatrix с матрицей проекции pMatrix, а матрицу shaderProgram.MVMatrix с mvMatrix. После этого значения матриц попадут в вершинный шейдер, где будут применены для определения координат вершин:

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

Обратите внимание на порядок преобразований: матрица проекции умножается на матрицу модели, а не наоборот, и потом все умножается на вектор координат вершин.

Код программы (**ex08_02.html**):

```
<!DOCTYPE html>
<html>
<head>
<title>Преобразования с glMatrix</title>
<meta charset="utf-8" />
</head>
<body>
<canvas id="canvas3D" width="400" height="300">Ваш браузер не поддерживает
        элемент canvas</canvas>
<script  type="text/javascript" src="gl-matrix-min.js"></script>
<!-- фрагментный шейдер -->
<script id="shader-fs" type="x-shader/x-fragment">
 void main(void) {  gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0); }
</script>
<!-- вершинный шейдер -->
<script id="shader-vs" type="x-shader/x-vertex">
 attribute vec3 aVertexPosition;
  uniform mat4 uMVMatrix;
 uniform mat4 uPMatrix;
 void main(void) { gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0); }
</script>
<script type="text/javascript">

var gl;
var shaderProgram;
var vertexBuffer; // буфер вершин
var indexBuffer; //буфер индексов
```

```
var mvMatrix = mat4.create();
var pMatrix = mat4.create();

// установка шейдеров
function initShaders() {
    var fragmentShader = getShader(gl.FRAGMENT_SHADER, 'shader-fs');
    var vertexShader = getShader(gl.VERTEX_SHADER, 'shader-vs');
    shaderProgram = gl.createProgram();
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);
    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
        alert("Не удалось установить шейдеры");
    }
    gl.useProgram(shaderProgram);
    shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
            "aVertexPosition");
    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
    // создания переменных uniform для передачи матриц в шейдер
    shaderProgram.MVMatrix = gl.getUniformLocation(shaderProgram, "uMVMatrix");
    shaderProgram.ProjMatrix = gl.getUniformLocation(shaderProgram, "uPMatrix");
}
function setMatrixUniforms(){
    gl.uniformMatrix4fv(shaderProgram.ProjMatrix,false, pMatrix);
    gl.uniformMatrix4fv(shaderProgram.MVMatrix, false, mvMatrix);
}
// Функция создания шейдера
function getShader(type,id) {
    var source = document.getElementById(id).innerHTML;
    var shader = gl.createShader(type);
    gl.shaderSource(shader, source);
    gl.compileShader(shader);
    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        alert("Ошибка компиляции шейдера: " + gl.getShaderInfoLog(shader));
        gl.deleteShader(shader);
        return null;
    }
    return shader;
}
 // установка буферов вершин и индексов
function initBuffers() {
    var vertices =[
                // лицевая часть
                -0.5, -0.5, 0.5,
                -0.5, 0.5, 0.5,
                 0.5, 0.5, 0.5,
                 0.5, -0.5, 0.5,
                // задняя часть
                -0.5, -0.5, -0.5,
                -0.5, 0.5, -0.5,
                 0.5, 0.5, -0.5,
                 0.5, -0.5, -0.5];
```

```
    var indices = [0, 1, 1, 2, 2, 3, 3, 0, 0, 4, 4, 5, 5, 6, 6,7, 7,4, 1, 5, 2, 6, 3, 7];


// установка буфера вершин
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  vertexBuffer.itemSize = 3;
  // создание буфера индексов
  indexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
            gl.STATIC_DRAW);
    // указываем число индексов это число равно числу индексов
    indexBuffer.numberOfItems = indices.length;
}
function draw() {
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                    vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
    gl.drawElements(gl.LINES, indexBuffer.numberOfItems, gl.UNSIGNED_SHORT,0);
}
function setupWebGL()
{
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    mat4.perspective(pMatrix, 1.04, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0);
    mat4.identity(mvMatrix);
    mat4.translate(mvMatrix,mvMatrix,[0, 0, -2.0]);
    mat4.rotate(mvMatrix,mvMatrix, 1.9, [0, 1, 0]);
}

window.onload=function(){
    var canvas = document.getElementById("canvas3D");
    try {
        gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
    }
    catch(e) {}
     if (!gl) {   alert("Ваш браузер не поддерживает WebGL");    }
    if(gl){
        gl.viewportWidth = canvas.width;
        gl.viewportHeight = canvas.height;
        initShaders();
        initBuffers();
        setupWebGL();
        setMatrixUniforms();
        draw();
    }
}
</script>
</body>
</html>
```

Если запустить веб-страничку, то результат будет таким:

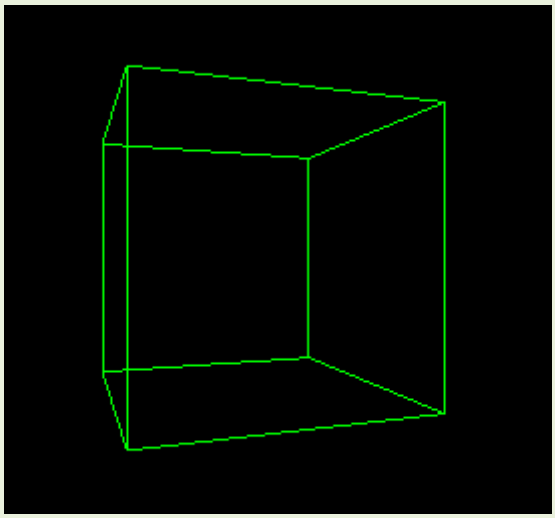

Рис. 8.3

Чтобы увидеть разные грани куба, в примере мы повернули куб на некоторый угол и сместили его по оси Z. Но есть еще и другой способ: можно применять преобразования не к объекту, а к камере, устанавливая ее в определенную точку пространства, поворачивая в заданном направлении, вращая и т.д.

Для ***установки камеры*** в библиотеке glMatrix используется функция mat4.lookAt(matrix, eye, center, up). Она принимает следующие параметры:

- matrix - матрица модели, которая меняется при изменении свойств камеры;
- eye - позиция камеры;
- center - точка, на которую направлена камера;
- up - вектор вертикальной ориентации.

Используем тот же пример, изменив там только функцию setupWebGL():

```
function setupWebGL()
{
   gl.clearColor(0.0, 0.0, 0.0, 1.0);
   gl.clear(gl.COLOR_BUFFER_BIT);
   gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
   mat4.perspective(pMatrix, Math.PI/2, gl.viewportWidth / gl.viewportHeight, 0.1,
         100.0);
   mat4.identity(mvMatrix);
   mat4.lookAt(mvMatrix, [2, 0,-2], [0,0,0], [0,1,0]);
}
```

Здесь нет никакого вращения или перемещения объекта. Камера устанавливается в точку пространства с координатами (2, 0, -2) и направлена на точку пространства (0, 0, 0). Затем устанавливается вектор вертикальной ориентации.

Результат (**ex08_03.html**):

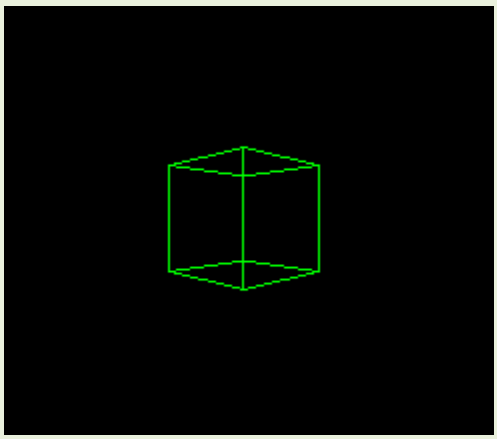

Рис. 8.4

8.4. Использование объекта Object3D для геометрических преобразований

Геометрические преобразования можно выполнять с прменением объекта **Object3D**, от которого наследуются все объекты, попадающие в сцену (камеры, источники света, полигональные модели, системы частиц).

Он отвечает за геометрическое положение объектов в пространстве и их иерархическую связь. С его помощью модно также группировать несколько объектов в одну группу, чтобы они двигались как единое целое.

Тогда при создании сложной фигуры все ее «запчасти» добавляются не на сцену, а «внутрь» объекта **Object3D**, и лишь потом объект выводится на сцену.

Перемещение объекта

Объект хранит в себе информацию о своем положении относительно предка в поле **position** в виде вектора (x, y, z); мы можем напрямую задавать эти величины. Каждую отдельно:

```
var obj = new THREE.Object3D();
obj.position.x=100;
obj.position.y=50;
obj.position.z=-10;
```

Или все вместе:

```
var obj = new THREE.Object3D();
obj.position.set( 100, 50, -10 );
```

Есть также методы для перемещения объекта относительно его локальной системы координат.

- **translateOnAxis (a, b)** перемещает объект в направлении нормализованного вектора **a** на растояние **b** в локальной системе координат.

- **translateX(b), translateY(b), translateZ(b)** - частные случаи метода **translateOnAxis**, где вектор перемещения направлен вдоль соответствующих осей (XYZ).

Тут важно понимать, что obj.position.x+=100 это не тоже самое, что obj.translateX(100). В первом случае мы переместили объект без учета направления его локальных осей. Если перед вызовом **obj.translateX (100);** мы осуществляли поворот объекта, то результат в этих двух случаях будет отличаться.

Код программы (файл **ex08_04.html**):

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Перемещение с Object3D</title>
  <script src="three.min_.js"></script>
</head>
<body>
</body>
  <script src="ex08_04.js"></script>
</html>
```

Файл **ex08_04.js**

```
var scene, camera, renderer, box, new_axis;
var dx=1, dy=0, dz=0;
var direct=1;
init();
animate();
function init(){
 renderer = new THREE.WebGLRenderer();
 renderer.setSize( 800, 600 );
 document.body.appendChild( renderer.domElement );
 camera = new THREE.PerspectiveCamera( 70, 800 / 600, 1, 1000 );
 camera.translateX( 50 );
 camera.translateY( 70 );
 camera.translateZ( 200 );
 scene = new THREE.Scene();
 var box_material=new THREE.MeshLambertMaterial( { color: 0x00FF00 } );
 var box_geometry = new THREE.BoxGeometry( 30, 30, 30, 1, 1, 1 );
 box = new THREE.Mesh( box_geometry, box_material );
 var light1 = new THREE.AmbientLight( 0x666666 );
 var light2 = new THREE.PointLight( 0x666666, 1, 400 );
 light2.position.set( 200, 200, 50 );
 new_axis = new THREE.Object3D();
 new_axis.position.set( 50, 50, 0 );
 new_axis.rotateOnAxis( new THREE.Vector3( 0, 1, 1 ).normalize(), 0.5 );
 var axisHelper_scene = new THREE.AxisHelper( 100 );
 var axisHelper_mesh = new THREE.AxisHelper( 50 );
 new_axis.add( axisHelper_mesh, box );
 scene.add( axisHelper_scene, light1, light2, new_axis );
}

function animate(){
 box.translateOnAxis( new THREE.Vector3( 1, 0, 0 ), dx*direct );
 box.translateY( dy*direct );
 box.translateOnAxis( new THREE.Vector3(0,0,13).normalize(), dz*direct );
 if ( box.position.x>100 ) {direct=-1;   }
 if ( box.position.y>100 ) {direct=-1;   }
 if ( box.position.z>100 ) {direct=-1;   }
 if ( box.position.x<0 ){box.position.x=0;
            direct=1;
            dx=0;
```

```
            dy=1        }
 if ( box.position.y<0 ){box.position.y=0;
            direct=1;
            dy=0;
            dz=1        }
 if ( box.position.z<0 ){box.position.z=0;
            direct=1;
            dz=0;
            dx=1        }
 renderer.render ( scene, camera );
 requestAnimationFrame ( animate );
}
```

Результат:

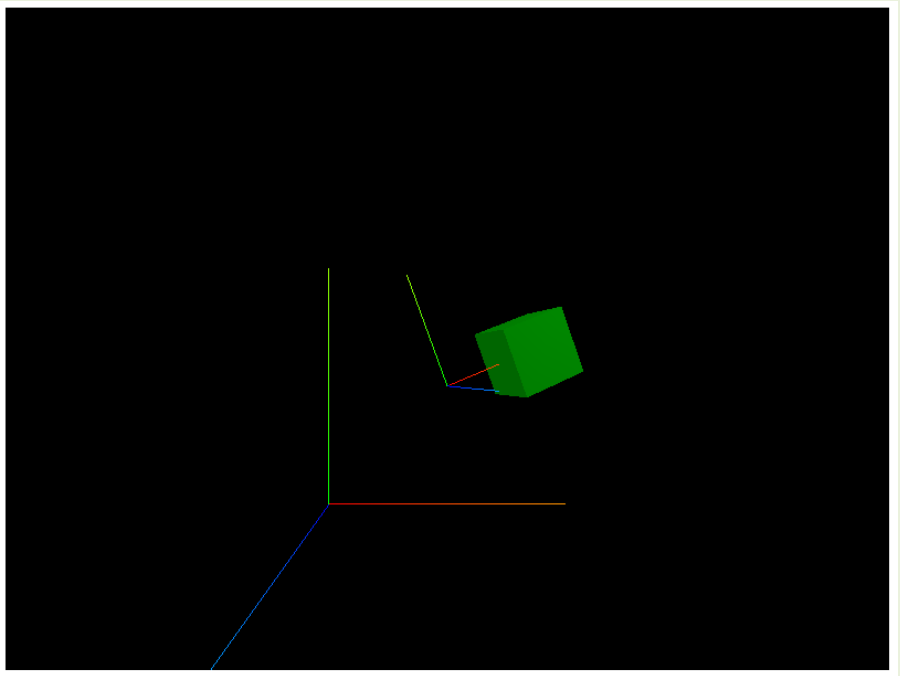

Рис. 8.5

Поворот объекта

Поворот объекта осуществляется вызовом метода **rotateOnAxis(a,b)**, где **a** - нормализованный вектор в пространстве, а **b** - угол поворота в радианах вокруг вектора **a**.

По аналогии с перемещением существуют частные случаи в виде методов **rotateX (b), rotateY (b), rotateZ (b),** осуществляющие поворот вокруг осей координат. Информация о вращении объекта **a** хранится в виде матрицы в поле **a.quaternion** и в виде углов поворота в поле **a.rotation,** где задается последовательностью поворотов вокруг осей XYZ.

Эти поля связанны друг с другом, и при изменении одного происходит автоматическое изменение другого.

Также осуществить поворот можно записав углы Эйлера в радианах

```
var a = new THREE.Object3D();
a.rotation.x = 1.3;
a.rotation.y = 0.8;
a.rotation.z = 2.4;
```

или

```
var a = new THREE.Object3D();
a.rotation.set( 1.3, 0.8, 2.4 );
```

В случае с углами поворота важен порядок, в котором происходит поворот вдоль осей координат. По умолчанию он задан XYZ. Поэтому вызов метода **a.rotation.set (1.3, 0.8, 2.4);** даст разные результаты при разном значении поля **a.rotation.order**

Код программы (файл **ex08_05.html**):

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Поворот с Object3D</title>
  <script src="three.min_.js"></script>
</head>
<body>

</body>
  <script src="ex08_05.js"></script>
</html>
```

Файл ex08_05.js

```
var scene,camera,renderer,box;
var delta = Math.PI/300;
var angle=0;
var dx=delta,dy=0,dz=0;

init();
animate();

function init(){
 renderer = new THREE.WebGLRenderer();
 renderer.setSize(800,600);
 document.body.appendChild( renderer.domElement );
 camera = new THREE.PerspectiveCamera( 70, 800 / 600, 1, 1000 );
 camera.translateX(50);
 camera.translateY(70);
 camera.translateZ(200);
 scene = new THREE.Scene();
 var box_material=new THREE.MeshLambertMaterial( { color: 0x00FF00 } );
 var box_geometry = new THREE.BoxGeometry( 50, 50, 50, 1, 1, 1 );
 box = new THREE.Mesh( box_geometry, box_material);
 var light1 = new THREE.AmbientLight( 0x666666 );
 var light2 = new THREE.PointLight( 0x666666, 1, 400 );
 light2.position.set(200,200,50);
 new_axis = new THREE.Object3D();
 box.position.set(50,50,0);
 box.rotateOnAxis(new THREE.Vector3(0,1,1).normalize(),0.5);
 var axisHelper_scene = new THREE.AxisHelper( 100 );
 var axisHelper_mesh = new THREE.AxisHelper( 50 );
 box.add( axisHelper_mesh,new_axis );
```

```
 scene.add( axisHelper_scene,light1,light2,box );
}

function animate(){
 box.rotateOnAxis (new THREE.Vector3(1,0,0),dx);
 box.rotateY(dy);
 box.rotateOnAxis (new THREE.Vector3(0,0,99).normalize(),dz);
 angle+=delta;
 if(angle>Math.PI/2){      angle=0;
          if(dx>0){              dx=0;     dy=delta;}
          else if(dy>0){         dy=0;     dz=delta;}
          else if(dz>0){         dz=0;     dx=delta;}

 }
 renderer.render( scene, camera );
 requestAnimationFrame( animate );
}
```

Результат:

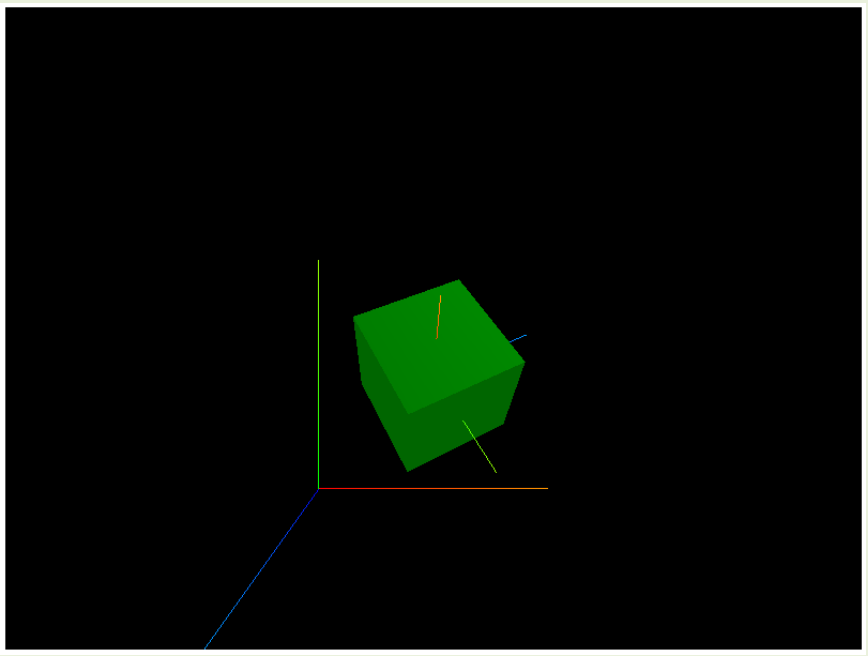

Рис. 8.6

Еще мы можем развернуть объект в направлении определенной точки пространства.

Для этого достаточно вызвать метод **lookAt(b)**, где **b** - вектор с координатами пространства (THREE.Vector3(x,y,z)). Демонстрация метода lookAt – программа слежения за мышью.

Код программы (файл **ex08_06.html**):

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Метод lookAt</title>
  <script src="three.min_.js"></script>
</head>
```

```
<body>
</body>
  <script src="ex08_06.js"></script>
</html>
```

Файл ex08_06.js

```
var scene,camera,renderer,eyeball_left,eyeball_right,body;
init();
animate();
function init(){
 renderer = new THREE.WebGLRenderer();
 renderer.setSize(800,600);
 document.body.appendChild( renderer.domElement );
 camera = new THREE.PerspectiveCamera( 70, 800 / 600, 1, 1000 );
 camera.translateZ(200);
 scene = new THREE.Scene();
 var body_material=new THREE.MeshLambertMaterial( { color: 0xA28D33 } );
 var body_geometry = new THREE.BoxGeometry( 150, 170, 30);
 var eyeball_material=new THREE.MeshLambertMaterial( { color: 0xFFFFFF } );
 var eyeball_geometry = new THREE.SphereGeometry( 20, 32, 32);
 var iris_material=new THREE.MeshLambertMaterial( { color: 0x000000 } );
 var iris_geometry = new THREE.SphereGeometry( 10, 16, 16);
 body = new THREE.Mesh( body_geometry, body_material);
 body.position.set( 0, -20, -15 );
 eyeball_left = new THREE.Mesh( eyeball_geometry, eyeball_material);
 var iris_left = new THREE.Mesh( iris_geometry, iris_material);
 iris_left.translateZ(15);
 eyeball_left.add(iris_left);
 eyeball_left.translateX(-30);
 eyeball_left.lookAt(camera.position);
 eyeball_right = new THREE.Mesh( eyeball_geometry, eyeball_material);
 var iris_right = new THREE.Mesh( iris_geometry, iris_material);
 iris_right.translateZ(15);
 eyeball_right.add(iris_right);
 eyeball_right.translateX(30);
 eyeball_right.lookAt(camera.position);
 var light1 = new THREE.AmbientLight( 0x666666 );
 var light2 = new THREE.PointLight( 0x666666, 1, 400 );
 light2.position.set(50,50,50);
 scene.add( light1, light2, body, eyeball_left, eyeball_right );
}

function animate(){
 renderer.render( scene, camera );
 requestAnimationFrame( animate );
}

renderer.domElement.onmousemove = function(e){
 var direct_point = new THREE.Vector3(e.offsetX-
            renderer.domElement.clientWidth/2+camera.position.x,
            e.offsetY+renderer.domElement.clientHeight/2+camera.position.y, 500);
```

```
 eyeball_left.lookAt(direct_point);
 eyeball_right.lookAt(direct_point);
}

renderer.domElement.onmouseout = function(e){
 eyeball_left.lookAt(camera.position);
 eyeball_right.lookAt(camera.position);
}
```

Результат:

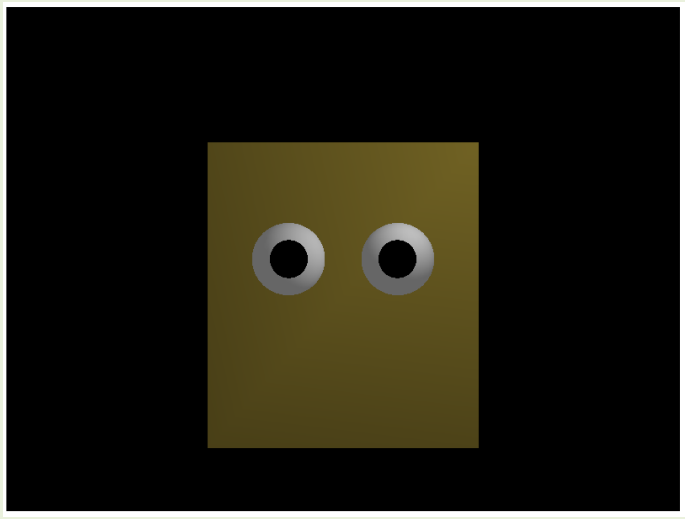

Рис. 8.7

Продвинутая дизайнерская версия программы приведена в разделе примеров к теме (файлы **ex08_06a.html, ex08_06a.js**):

8.5. Окрашивание трехмерных объектов

Окрашивание трехмерных объектов мало чем отличается от окрашивания двухмерных, однако есть некоторые моменты, на которые надо обратить внимание. В качестве примера создадим разноцветный куб.

В данном примере куб создается из треугольников. Каждая сторона куба состоит из двух треугольников, поэтому определяем индексы в буфере индексов для создания 12 треугольников, образующих 6 поверхностей: лицевую, заднюю, верхнюю, нижнюю и две боковых. При этом количество вершин остается тем же, что и в предыдущих примерах. Затем определяем для каждой вершины цвет:

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

```
            ];
  var indices = [ // лицевая грань
            0, 1, 2,
            2, 3, 0,
            //нижняя грань
            0, 4, 7,
            7, 3, 0,
            // левая боковая грань
            0, 1, 5,
            5, 4, 0,
            // правая боковая грань
            2, 3, 7,
            7, 6, 2,
            // верхняя грань
            2, 1, 6,
            6, 5, 1,
            // задняя грань
            4, 5, 6,
            6, 7, 4,
          ];
// установка буфера вершин
 vertexBuffer = gl.createBuffer();
 gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
 gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
 vertexBuffer.itemSize = 3;
 // создание буфера индексов
 indexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
          gl.STATIC_DRAW);
  // указываем число индексов это число равно числу индексов
  indexBuffer.numberOfItems = indices.length;
  // установка цветов для каждой вершины
  var colors = [
     0.0, 0.0, 0.3,
     0.0, 0.0, 1.0,
     0.0, 1.0, 0.0,
     0.0, 0.3, 0.0,
     0.0, 0.0, 0.3,
     0.0, 0.0, 1.0,
     0.0, 1.0, 0.0,
     0.0, 0.3, 0.0,
  ];
  colorBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
}
```

Перед отрисовкой надо включить глубину, тем самым скрыв те поверхности, которые не должны быть отображены.

```
gl.enable(gl.DEPTH_TEST);
```

```
gl.drawElements(gl.TRIANGLES, indexBuffer.numberOfItems,
        gl.UNSIGNED_SHORT,0);
```

Мы устанавливаем атрибут цвета и передаем его в вершинный шейдер, а через него - во фрагментный. Общий принцип будет таким же, как и при создании цветной двухмерной фигуры.

Полный код программы – в файле **ex08_07.html**.

Получим куб, окрашенный в зеленый и синий цвета:

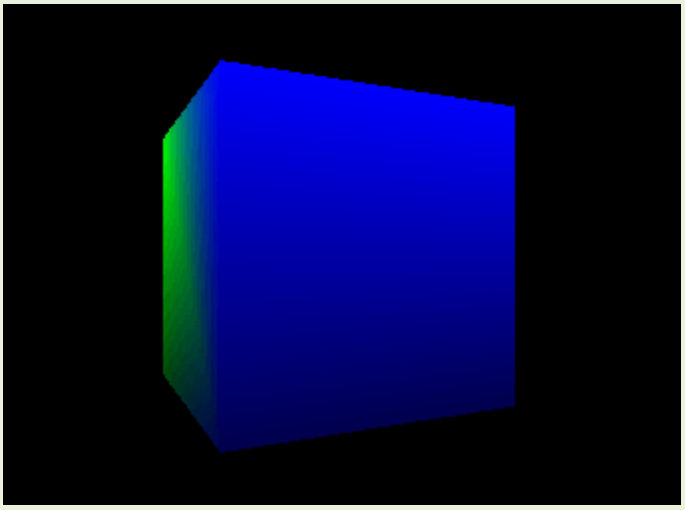

Рис. 8.8

Если не использовать метод *gl.enable(gl.DEPTH_TEST)*, включающий глубину для скрытия поверхностей, которые не должны быть отображены, то результат будет другим:

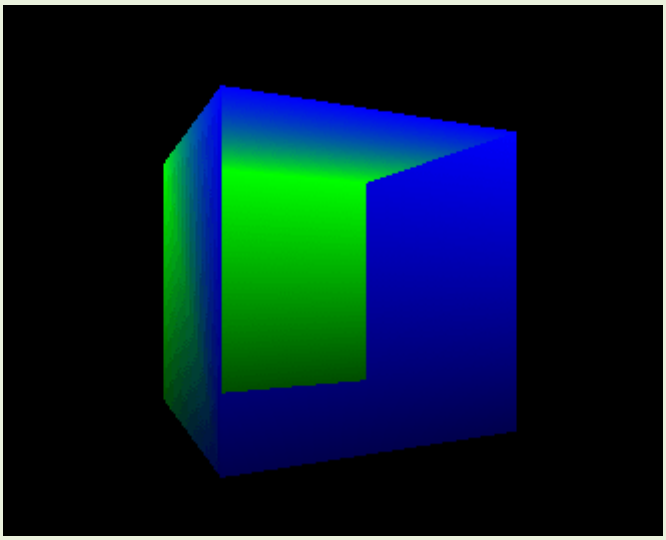

Рис. 8.9

Контрольные вопросы

1. Матрицы в WebGL.
2. Методы библиотеки glMatrix.
3. Трехмерные геометрические преобразования.
4. Окрашивание трехмерных объектов.