Программирование графических приложений

Тема 14
Анимация и пользовательский ввод

Цель изучения темы. Изучение методов анимации трехмерных объектов и способов управления объектами пользователем в WebGL.

14.1. Анимация объектов

В предыдущих темах уже встречалась анимация объектов в WebGL, реализованная с использование библиотеки Three.js. Рассмотрим способы анимации объектов без использования библиотек. В качестве примера анимации создадим вращающийся куб. За основу будет взят уже рассмотренный код построения статического куба **ex08_01.html** (тема 8).

Полный код программы будет следующим (**ex14_01.html**):

```
<!DOCTYPE html>
<html>
<head>
<title>Анимация в WebGL</title>
<meta charset="utf-8" />
<head>
</head>
<body>
<canvas id="canvas3D" width="400" height="300">Ваш браузер не поддерживает
        элемент canvas</canvas>
<script  type="text/javascript" src="gl-matrix-min.js"></script>

<script id="shader-fs" type="x-shader/x-fragment">
 void main(void) {    gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);   }
</script>
<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;
var indexBuffer;
var angle = 0.0; // угол поворота
var mvMatrix = mat4.create();
var pMatrix = mat4.create();
function initShaders() {
   var fragmentShader = getShader(gl.FRAGMENT_SHADER, 'shader-fs');
   var vertexShader = getShader(gl.VERTEX_SHADER, 'shader-vs');
   shaderProgram = gl.createProgram();
   gl.attachShader(shaderProgram, vertexShader);
   gl.attachShader(shaderProgram, fragmentShader);
   gl.linkProgram(shaderProgram);
   if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
      alert("Не удалось установить шейдеры");
   }
   gl.useProgram(shaderProgram);
```

```
    shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
            "aVertexPosition");
    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
    shaderProgram.MVMatrix = gl.getUniformLocation(shaderProgram, "uMVMatrix");
    shaderProgram.ProjMatrix = gl.getUniformLocation(shaderProgram, "uPMatrix");
}
function setMatrixUniforms(){
    gl.uniformMatrix4fv(shaderProgram.ProjMatrix,false, pMatrix);
    gl.uniformMatrix4fv(shaderProgram.MVMatrix, false, mvMatrix);
}
function getShader(type,id) {
    var source = document.getElementById(id).innerHTML;
    var shader = gl.createShader(type);
    gl.shaderSource(shader, source);
    gl.compileShader(shader);
    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        alert("Ошибка компиляции шейдера: " + gl.getShaderInfoLog(shader));
        gl.deleteShader(shader);
        return null;
    }
    return shader;
}
function initBuffers() {
    var vertices =[
                -0.5, -0.5, 0.5,
                -0.5, 0.5, 0.5,
                 0.5, 0.5, 0.5,
                 0.5, -0.5, 0.5,
                -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];
    vertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
    vertexBuffer.itemSize = 3;
    indexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
            gl.STATIC_DRAW);
    indexBuffer.numberOfItems = indices.length;
}
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);
    angle += 0.01;
    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, angle, [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();
        // функция анимации
        (function animloop(){
            setupWebGL();
            setMatrixUniforms();
            draw();
          requestAnimFrame(animloop, canvas);
        })();
    }
}
// настройка анимации
window.requestAnimFrame = (function(){
     return  window.requestAnimationFrame       ||
             window.webkitRequestAnimationFrame ||
             window.mozRequestAnimationFrame    ||
             window.oRequestAnimationFrame      ||
             window.msRequestAnimationFrame     ||
            function(callback, element) {
              return window.setTimeout(callback, 1000/60);
            };
})();
</script>
</body>
</html>
```

В отличие от примера со статическим кубом, здесь есть несколько изменений.

Во-первых, добавлена переменную angle, которая будет определять угол поворота. Через заданный интервал времени эта переменная будет изменяться, и соответственно будет меняться угол поворота куба.

Во-вторых, в самом конце кода добавлена функцию requestAnimationFrame для создания анимации.

Результат:

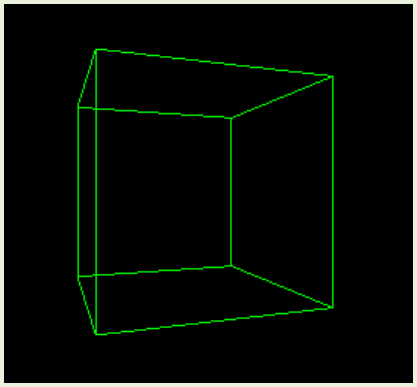

Рис. 14.1

Использование requestAnimationFrame

Для создания анимации раньше разработчики могли использовать специальные функции javascript:

```
setTimeout(callback, timeoutInMilliseconds)
setInterval(callback, timeoutInMilliseconds)
```

Обе функции в качестве параметра callback принимают функцию, которая срабатывает через определенное время timeoutInMilliseconds. Применительно к нашей задаче можно написать:

```
window.onload=function(){
    //.....................
    initShaders();
    initBuffers();
    setInterval(drawloop, 16.7);
}
function drawloop() {
    setupWebGL();
    setMatrixUniforms();
    draw();
}
```

Здесь функция setInterval вызывает функцию drawloop каждые 16,7 миллисекунд, то есть 60 раз в секунду. Результат выполнения кода будет аналогичен нашему примеру. Но разработчики браузеров в последних версиях стали внедрять новое решение - метод requestAnimationFrame, который является рекомендуемым способом для создания скриптовой анимации. Здесь, в отличие от использования методов setInterval() и setTimeout(), браузер сам оптимизирует анимацию. Если в случае с setInterval() мы устанавливаем интервал анимации, то при использовании requestAnimationFrame() браузер сам определяет оптимальный интервал - ведь в одно и то же время в браузере могут выполняться сразу несколько анимаций, которые влияют друг на друга. Поэтому для создания более плавного эффекта браузер может изменять темп анимации, эффективно определяя нужный интервал.

В данном случае мы задействуем requestAnimationFrame, и в качестве параметра данный метод получает функцию, выполняемую перед перерисовкой.

```
window.requestAnimFrame = (function(){
    return  window.requestAnimationFrame       ||
```

```
            window.webkitRequestAnimationFrame ||
            window.mozRequestAnimationFrame    ||
            window.oRequestAnimationFrame      ||
            window.msRequestAnimationFrame     ||
           function(callback, element) {    return window.setTimeout(callback, 1000/60);
            };
})();
```

Это кроссбраузерное определение метода. Далее в функции настройки контекста setupWebGL() увеличиваем угол на некоторое число и применяем его при повороте матрицы:

```
angle += 0.01;
//.....................
mat4.rotate(mvMatrix,mvMatrix, angle, [0, 1, 0]);
```

В главной функции мы используем следующую конструкцию, в которую выносим функции по настройке матриц и отрисовке:

```
(function animloop(){
   setupWebGL();
   setMatrixUniforms();
   draw();
   requestAnimFrame(animloop, canvas);
})();
```

В программе функция *requestAnimFrame(animloop,canvas)* как раз обращается к *window.requestAnimationFrame*, передавая в параметрах функцию обратного вызова и элемент, который содержит всю анимацию, то есть в данном случае *canvas*.

Подобным образом мы можем задать анимацию перемещения или масштабирования. Также можно анимировать перемещение камеры, а не куба.

14.2. Обработка пользовательского ввода

Обработка пользовательского ввода предполагает обработку событий клавиатуры и мыши и не сильно отличается от общей модели, которая применяется в javascript, например, при создании игр с использованием элемента canvas.

Создадим программу, которая будет по нажатию клавиш со стрелками перемещать куб вперед, назад, а также вращать его (**ex14_02.html**):

```
<script type="text/javascript">
var gl;
var shaderProgram;
var vertexBuffer;
var indexBuffer;
var angle = 2.0;//угол вращения в радианах
var zTranslation = -2.0; // смещение по оси Z
var mvMatrix = mat4.create();
var pMatrix = mat4.create();
function initShaders() {
   var fragmentShader = getShader(gl.FRAGMENT_SHADER, 'shader-fs');
   var vertexShader = getShader(gl.VERTEX_SHADER, 'shader-vs');
   shaderProgram = gl.createProgram();
   gl.attachShader(shaderProgram, vertexShader);
```

```
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);
    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
        alert("Не удалось установить шейдеры");
    }
    gl.useProgram(shaderProgram);
    shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
            "aVertexPosition");
    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
    shaderProgram.MVMatrix = gl.getUniformLocation(shaderProgram, "uMVMatrix");
    shaderProgram.ProjMatrix = gl.getUniformLocation(shaderProgram, "uPMatrix");
}
function setMatrixUniforms(){
    gl.uniformMatrix4fv(shaderProgram.ProjMatrix,false, pMatrix);
    gl.uniformMatrix4fv(shaderProgram.MVMatrix, false, mvMatrix);
}
function getShader(type,id) {
    var source = document.getElementById(id).innerHTML;
    var shader = gl.createShader(type);

    gl.shaderSource(shader, source);
    gl.compileShader(shader);
    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        alert("Ошибка компиляции шейдера: " + gl.getShaderInfoLog(shader));
        gl.deleteShader(shader);
        return null;
    }
    return shader;
}
function initBuffers() {
    var vertices =[
                -0.5, -0.5, 0.5,
                -0.5, 0.5, 0.5,
                 0.5, 0.5, 0.5,
                 0.5, -0.5, 0.5,
                -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];
    vertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
    vertexBuffer.itemSize = 3;
    indexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
            gl.STATIC_DRAW);
    indexBuffer.numberOfItems = indices.length;
}
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, zTranslation]);
    mat4.rotate(mvMatrix,mvMatrix, angle, [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){
        document.addEventListener('keydown', handleKeyDown, false);
        gl.viewportWidth = canvas.width;
        gl.viewportHeight = canvas.height;
        initShaders();
        initBuffers();
        // функция анимации
        (function animloop(){
            setupWebGL();
            setMatrixUniforms();
            draw();
          requestAnimFrame(animloop, canvas);
        })();
    }
}
function handleKeyDown(e){
    switch(e.keyCode)
    {
        case 39:  // стрелка вправо
            angle+=0.1;
            break;
        case 37:  // стрелка влево
            angle-=0.1;
            break;
        case 40:  // стрелка вниз
            zTranslation+=0.1;
            break;
        case 38:  // стрелка вверх
            zTranslation-=0.1;
            break;
```

```
    }
}
// настройка анимации
window.requestAnimFrame = (function(){
    return  window.requestAnimationFrame       ||
            window.webkitRequestAnimationFrame ||
            window.mozRequestAnimationFrame    ||
            window.oRequestAnimationFrame      ||
            window.msRequestAnimationFrame     ||
        function(callback, element) {
          return window.setTimeout(callback, 1000/60);
        };
})();
</script>
```

Вначале для фиксации перемещения задаются две переменные:

```
var angle = 2.0; //  угол поворота в радианах
var zTranslation = -2.0; // смещение по оси Z
```

Дальше, если у нас задан контекст WebGL, мы регистрируем обработчик события нажатия клавиши: document.addEventListener('keydown', handleKeyDown, false). Соответственно добавляем в конце функцию обработчика:

```
function handleKeyDown(e){
   switch(e.keyCode)
   {
      // изменяем угол поворота
      case 39:  // стрелка вправо
         angle+=0.1;
         break;
      case 37:  // стрелка влево
         angle-=0.1;
         break;
      // изменяем смещение по оси Z
      case 40:  // стрелка вниз
         zTranslation+=0.1;
         break;
      case 38:  // стрелка вверх
         zTranslation-=0.1;
         break;
   }
}
```

С помощью свойства e.keyCode мы можем узнать код нажатой клавиши и в зависимости от этого выполнить определенные действия. Так как у нас задействована функция анимации requestAnimFrame, то в цикле функции для установки матриц получают новые значения и, таким образом, изменяют положение объекта.

Результат:

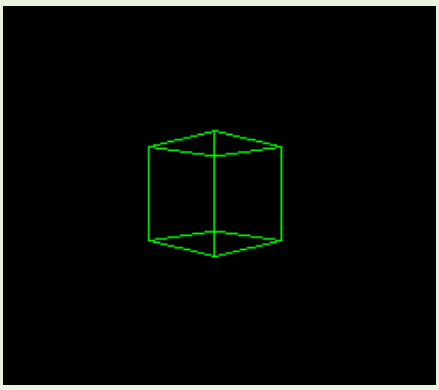

Рис. 14.2

14.3. Управление клавиатурой при использовании Three.js

Обработка нажатий клавиш в программах управления графитческими объектами, сформированными с использованием библиотеки Three.js, по сути ничем не отличается от приведённой в предыдущем разделе.

Создадим простой пример управления объектом клавиатурой. За основу вновь возьмем уже рассмотренную программу построения пространственных примитивов. В код программы добавим класс Key:

```
var Key =
{
_pressed: {},
A:          65,
W:          87,
D:          68,
S:          83,
SPACE:      32,
VK_LEFT:    37,
VK_RIGHT:   39,
VK_UP:      38,
VK_SPACE: 32,
VK_ENTER: 13,
isDown: function(keyCode) { return this._pressed[keyCode]; },
                onKeydown: function(event) { this._pressed[event.keyCode] = true; },
                onKeyup: function(event) { delete this._pressed[event.keyCode]; }
};
```

Поскольку JavaScript работает с клавиатурой через коды клавиш, здесь указываются названия (для наглядности) клавиш и их коды. При необходимости можно добавить свои клавиши или удалить ненужные.

Теперь добавляем созданные обработчики событий на веб-страницу:

```
window.addEventListener( 'keyup', function(event) { Key.onKeyup(event); }, false);
window.addEventListener('keydown', function(event) { Key.onKeydown(event); }, false);
```

Теперь для того, чтобы определить нужные действия при нажатии на клавишу, достаточно использовать конструкцию типа:

```
if (Key.isDown(Key.VK_LEFT))  // если нажата стрелка «влево»
{ действия}
```

Подобную проверку нужно осуществлять внутри функции render(). Создадим пример кубика, который под управлением стрелок движется вправо и влево вдоль оси X системы координат, а при нажатии на enter перемещается вверх по оси Y (**ex14_03.html, ex14_03.js**).

Сначала внутри функции init() создаем куб обычным образом:

```
var geometry = new THREE.BoxGeometry( 50, 50, 50);
var material = new THREE.MeshNormalMaterial({color: 0x00ff00});
Cube = new THREE.Mesh( geometry, material );
scene.add( Cube );
```

Далее создадим отдельную функцию dynamo():

```
function dynamo()
{
 if (Key.isDown(Key.VK_LEFT)) // движение влево
  { Cube.position.x -= 10; }
 if (Key.isDown(Key.VK_RIGHT)) // движение вправо
  { Cube.position.x += 10; }
 if (Key.isDown(Key.VK_ENTER)) // подскок
  { Cube.position.y += 10; }
}
```

Эта функция будет вызываться внутри render():

```
function render()
 {
  dynamo(); //управление с помощью клавиатуры
  controls.update(); //управление камерой с помощью мышки
  renderer.render(scene, camera);
}
```

В результате получим страничку с кубиком, управляемым клавишами ←, → и Enter.

Рис. 14.3

Управление камерой мышью также сохранится. Если повернуть камеру мышью, система координат сцены изменит своё положение относительно экрана, и кубик по клавишам управления будет перемещаться не обязательно параллельно границам экрана.

14.4. Обработка событий мышки

Рассмотрим обработку события наведения мышки на трехмерный объект. В нашей программе при наведении указателем мыши на один из объектов сцены будет инициироваться

некоторое действие (например, объект меняет свой цвет).

Как работает алгоритм: нужно направить в точке указателя луч вглубь сцены и посмотреть, какие объекты сцены он пересекает. Если луч по пути пересек какой-то объект, значит, указатель мыши «попадает» по нему. При этом луч может пересечь по пути несколько объектов. Все они будут запомнены в том порядке, в котором они встретились лучу. Сначала первый встреченный объект, потом второй, и т.д.

По идее, нужно перебрать абсолютно все объекты сцены на предмет проверки пересечения с этим лучом. Это может занять довольно длительное время. На самом деле обычно требуется, чтобы на клик реагировали далеко не все, а только определенные объекты сцены. Поэтому на практике те элементы (объекты), клики на которые нужно фиксировать, помещаются в специальный массив. Объявим, например, глобальный массив objects:

```
var objects = [ ];
```

В нашем примере фиксируются события наведения указателя мыши на прямоугольные параллелепипеды, поэтому все они будут записываться в этот массив. Как обычно, объявляем геометрию, материал и т.д.:

```
var geometry = newTHREE.BoxGeometry( 100, 100, 100 );
for ( vari = 0; i< 10; i ++ )
{
 // создаем параллелепипед
var object = new THREE.Mesh( geometry,
new THREE.MeshBasicMaterial({ color: Math.random() * Oxffffff, opacity: 0.5 } ) );
// здесь можно задать расположение и масштаб объекта
scene.add( object ); // выводим параллелепипед на сцену
objects.push( object ); // запоминаем параллелепипед в массиве objects
}
```

Теперь наши параллелепипеды записаны в массив **objects**, и при наведении мыши каждый параллелепипед будет проверяться, попал ли указатель на него или нет.

Для того, чтобы определить объекты, на которые пришелся указатель мышки, нужно в точке нахождения указателя направить луч от наблюдателя вглубь сцены. Начало такого луча - положение камеры:

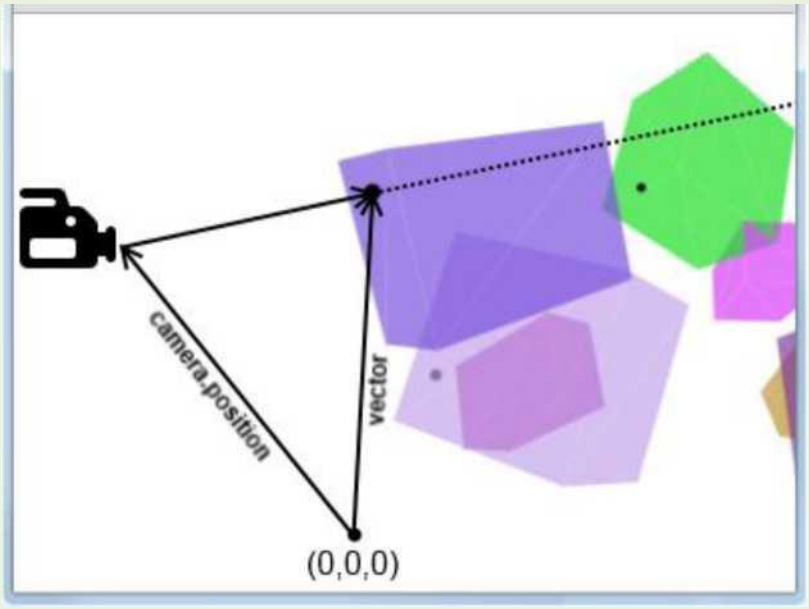

Рис. 14.4

Но чтобы провести прямую, нужно две точки. Вторая точка связана с координатами места указателя мыши на экране, которые определяются как числа event.clientX, event.clientY. Эти координаты двухмерные, а нам нужно их перевести в трехмерные координаты сцены. Для осуществления такого перевода предусмотрен класс Projector() с методом unprojectVector:

```
projector = new THREE.Projector();
var vector = new THREE.Vector3 ( ( event.clientX / window.innerWidth ) * 2 - 1,
                                 - ( event.clientY / window.innerHeight ) * 2 + 1, 0.5 );
projector.unprojectVector( vector, camera );
```

Теперь, зная две точки, через которые проходит луч, можно его построить. Для построения луча используется класс Raycaster, для которого требуется указать начальную точку луча и его направление (направляющий вектор):

```
var raycaster = new THREE.Raycaster (camera.position,
                                     vector.sub (camera.position ).normalize() );
```

Направляющий вектор получается вычитанием координат указателя мышки и камеры (*a.sub(b)* дает вектор *a-b*) с последующим приведением к нормализованному виду (получение вектора единичной длины той же направленности, путем деления вектора на его длину).

Теперь нужно найти объекты, которые пересек наш луч. Это делает класс Raycaster, запоминая найденные объекты в специальном массиве. Обратиться к этому массиву можно с помощью метода intersectObjects:

```
var intersects = raycaster.intersectObjects( objects );
```

Тогда условие, попал ли клик на какие-либо объекты, запишется в виде:

```
if ( intersects.length> 0 ){ ... }
```

В массиве intersects теперь будет храниться следующая информация об объектах:

object: объект, на который наведён указатель;
distance: расстояние до точки расположения указателя (на объекте!);
point: трехмерные координаты точки расположения указателя;
face: грань объекта, на котором находится указатель;
faceIndex: номер этой грани.

Объекты находятся в массиве по порядку встречи с лучом (нумерация идет с нуля). Поэтому объект, на который навели мышкой, будет первым. Тогда определить координаты точки, на которую попал указатель, можно, например, так:

```
intersects[0].point;
```

В нашем примере куб красится в новый произвольный цвет:

```
if ( intersects.length> 0 ) // если массив не пуст
{
intersects[0].object.material.color.setHex( Math.random()*0xffffff );
... // другие действия
}
```

Если вдруг нужно покрасить все параллелепипеды, встретившиеся на пути луча, то можно

использовать конструкцию:

```
for ( var i in intersects )
{
intersects[ i ].object.material.color.setHex( Math.random()*0xffffff );
... // другие действия
}
```

При изменении размера окна браузера камера сдвинется, и raycaster будет «промахиваться». Для решения этой проблемы можно использовать метод onWindowResize, обновляющий камеру после события изменения размера окна браузера. Создается функция обновления камеры:

```
functiononWindowResize()
{
camera.aspect = window.innerWidth / window.innerHeight;
camera.updateProjectionMatrix();
renderer.setSize( window.innerWidth, window.innerHeight );
}
```

Теперь внутри функции init() добавим эту реакцию на событие изменения размера окна:

```
window.addEventListener( 'resize', onWindowResize, false );
```

Если вдруг и окно сцены не равно размеру всего окна браузера, то, во избежание трудоемких вычислений положения камеры, проще вставить окно приложения в отдельный фрейм.

Полный код программы - в файлах **ex14_04.html**, **ex14_04.js**.

Результат:

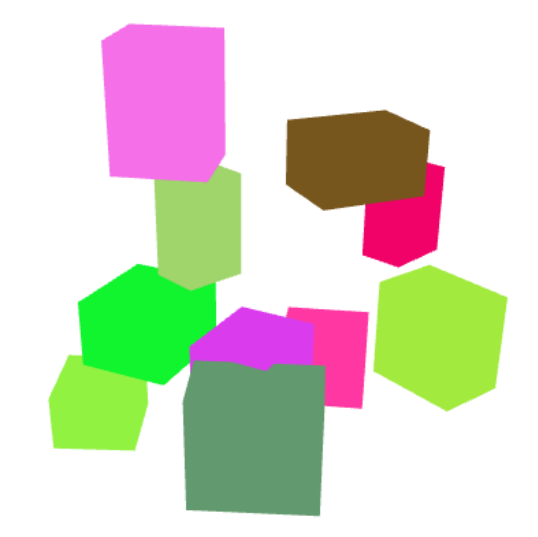

Рис. 14.5

Здесь мы рассмотрели методологию создания обработчика события мышки при указании её на объект. В следующем пункте рассмотрим пример применения.

14.6. Перемещение объектов по клику мыши

При реализации перемещения объектов мышкой бывает удобно не просто перемещать

объект за курсором мыши, а перемещать вдоль какой-либо плоскости. Рассмотрим технику перемещения объектов мышкой на примере трехмерных шашек.

В шашках объекты достаточно перемещать вдоль плоскости шахматной доски.

Создадим шахматную доску с шашками. Для их перемещения понадобится два выбора луча. Алгоритм будет следующий:

- первый луч будет отслеживать клик на шашке; как только клик произведен, переменная SELECTED начинает ссылаться на объект шашки;
- отслеживаем перемещение мышки; если SELECTED не null, то проводим второй луч в доску и перемещаем шашку в место попадания;
- если мышку отпускаем (и SELECTED не null), шашка остается на новом месте, SELECTED обращаем в null.

При этом шашка не может лежать на доске где попало, поэтому в последнем пункте нужно добавить итоговое смещение к центру ближайшей черной клетки. Если же положение шашки слишком неопределенно, лучше вернуть ее на место. Потому после клика на шашке будем запоминать ее предыдущее положение (x_previous, z_previous).

Создадим ***шахматную доску*** на основе текстуры размером 128x128 пикселей (2^7x2^7 пикселей) – файл **checkerboard.jpg**:

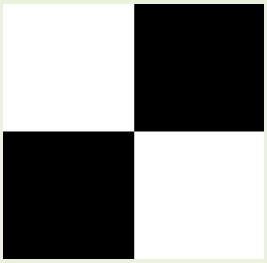

Рис. 14.6

Шахматную доску мы получим, размножив эту текстуру по плоской поверхности. Объявим:

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

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

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

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

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

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

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

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

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

Здесь рисунок сдвигается вправо на величину, равную половине его ширины.

Осталось создать материал на основе заданной текстуры и геометрию нашей будущей доски, которую мы построим как кусок плоскости с помощью PlaneGeometry:

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

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

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

Полный код программы в файлах **ex14_05.html, ex14_05.js.**

Результат:

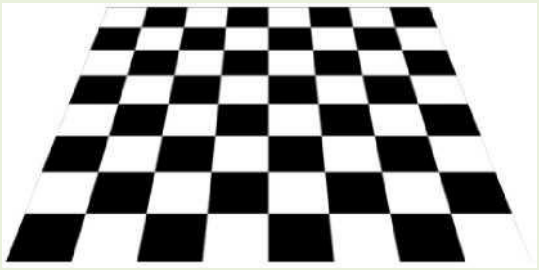

Рис. 14.7

Добавим ***шашки*** - цилиндры**:**

```
projector = new THREE.Projector();
objects.push( checkerboard );
var geometry = new THREE.CylinderGeometry( 22, 22, 16, 36 );
var material = new THREE.MeshPhongMaterial( { color: 0x9b2d30, specular: 0x00b2fc,
        emissive: 0x000000, shininess: 40, shading: THREE.FlatShading,
blending: THREE.NormalBlending, depthTest: true } );
var x0 = -175;
z0 = 175;
for ( var i = 0; i < key_count; i ++ )
{
 var object = new THREE.Mesh( geometry, material );
 if (i==4) { x0 = -175 - 7 * 50; z0 = 175 - 50; }
 if (i==8) { x0 = -175 - 16 * 50; z0 = 175 - 2*50; }
 object.position.set( x0 + i * 2*50, 0, z0);
 scene.add( object );
 objects.push( object );
}
```

Получившийся код – в файлах **ex14_06.html, ex14_06.js.**
Результат:

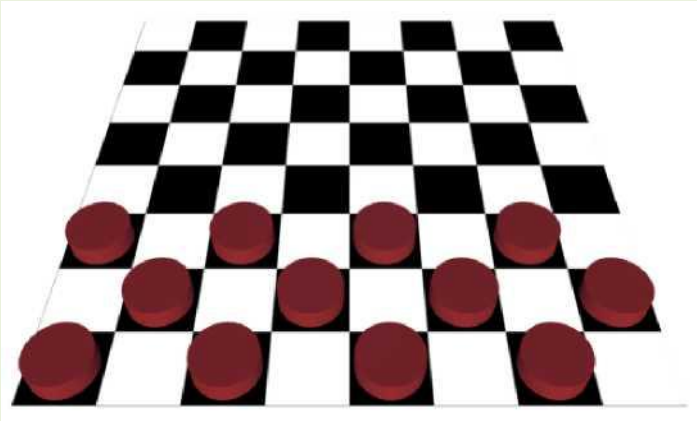

Рис. 14.8

Саму доску мы тоже добавили в массив objects, чтобы, если мышка будет над доской, случайно не переместить камеру.

Возможность перемещения камеры останется при движении мышки в стороне от доски. Таким образом, пользователь может расположить доску наиболее удобным для себя образом.

Итак, событие щелчка мышки:

```
function onDocumentMouseDown( event )
{
 var vector = new THREE.Vector3 ( ( event.clientX / window.innerWidth ) * 2 - 1,
                                  ( event.clientY / window.innerHeight ) * 2 + 1, 0.5 );
 projector.unprojectVector( vector, camera );
 var raycaster = new THREE.Raycaster(
                          camera.position, vector.sub( camera.position ).normalize() );
 var intersects = raycaster.intersectObjects( objects );
  if ( intersects.length > 0 )
  {
   controls.enabled = false;
   var k = objects.indexOf(intersects[ 0 ].object);
   if (k==0) { return; } // мышка кликнула над доской
   SELECTED = intersects[ 0 ].object;
   x_previous = SELECTED.position.x;
   z_previous = SELECTED.position.z;
  }
}
```

Перемещаем мышку:

```
function onDocumentMouseMove( event )
{
 var vector = new THREE.Vector3( ( event.clientX / window.innerWidth ) * 2 - 1,
                                 ( event.clientY / window.innerHeight ) * 2 + 1, 0.5 );
 projector.unprojectVector( vector, camera );
 var raycaster = new THREE.Raycaster(
                          camera.position, vector.sub( camera.position ).normalize() );
 if ( SELECTED )
```

```
  {
   var intersects = raycaster.intersectObject( checkerboard );
    SELECTED.position.x = intersects[ 0 ].point.sub(vector).x;
    SELECTED.position.z = intersects[ 0 ].point.sub(vector).z;
    SELECTED.position.y = 0;
    container.style.cursor = 'move';
  }
}
```

И, наконец, осталось расписать событие, когда отпускаем мышку. При этом найдем расстояние *dr* от шашки до центра ближайшего черного квадрата. Если оно окажется слишком велико (*flag* останется равным *false*), вернем шашку на место:

```
function onDocumentMouseUp( event )
{
 controls.enabled = true;
 flag = false;
 if ( SELECTED )
  {
    // находим центр ближайшей черной клетки
   for (i=0; i<8; i++)
    {
     for (j=0; j<8; j++)
     {
      if ((i+j)%2==0)
      {
       var dx = Math.abs( SELECTED.position.x - ( -175 + 50 * i ) );
       var dz = Math.abs( SELECTED.position.z - ( 175 - 50 * j ) );
       var dr = Math.sqrt( dx*dx + dz*dz);
       if ( dr < 22 )
       {
         SELECTED.position.x =-175 + 50 * i;
         SELECTED.position.z =175 - 50 * j;
         flag = true;
         break;
       }
      }
    }
  }
 if (!flag) // черная клеточка слишком далека
  {
   SELECTED.position.x = x_previous;
   SELECTED.position.z = z_previous;
  }
 SELECTED = null;
 }
 container.style.cursor = 'auto';
}
```

Теперь шашки можно двигать (файлы **ex14_07.html**, **ex14_07.js**).

В нашей программе шашки могут перемещаться произвольно и непрерывно по всей доске, но встают всегда в центр черного квадрата. Можно сразу реализовать эффект, когда шашка

движется «дискретно» только вдоль центров квадратов. Для этого достаточно перенести цикл нахождения центра ближайшего черного квадрата из функции onDocumentMouseUp в onDocumentMouseMove.

14.7. Управление гранями объекта

При клике на объект можно не только определить его номер, но и номер грани (по которой кликнули), с помощью конструкции

```
intersects[0].facelndex
```

Воспользуемся этим и создадим разноцветный кубик, при нажатии на грань которого выводится сообщение о соответствующем цвете. Объявим projector и глобальный массив:

```
var projector = new THREE.Projector();
var objects = [ ];
```

куда потом поместим единственный объект - наш куб.

Построение прямоугольного параллелепипеда уже рассмотривалось в темах 1 и 9. Наш код будет основан на уже рассмотренном примере построения куба с разноцветными гранями:

Объявление класса имеет вид:

```
var materials = [
  new THREE.MeshBasicMaterial( { color: 0xff0000 }),
  new THREE.MeshBasicMaterial( { color: 0x00ff00 }),
  new THREE.MeshBasicMaterial( { color: 0x0000ff }),
  new THREE.MeshBasicMaterial( { color: 0xff00ff }),
  new THREE.MeshBasicMaterial( { color: 0xffff00 }),
  new THREE.MeshBasicMaterial( { color: 0x00ffff })
];
var material = new THREE.MeshFaceMaterial( materials );
var geometry = new THREE.BoxGeometry( 200, 200, 200, 1, 1 );
Cube = new THREE.Mesh( geometry, new THREE.MeshFaceMaterial(materials) );
scene.add( Cube );
objects.push( Cube);
```

Результат (полный код - в файлах **ex14_08.html, ex14_08.js**):

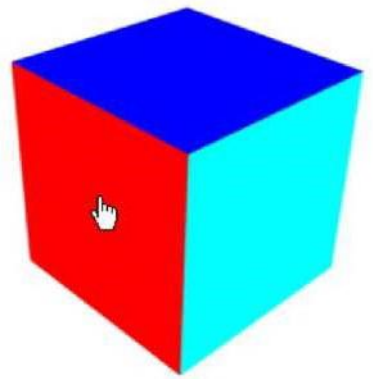

Рис. 14.9

При нашем способе объявления кубика на каждой его стороне находится две «полуграни»:

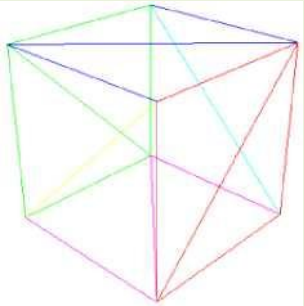

Рис. 14.10

Поэтому при клике на стороне кубика, на первую его сторону приходятся полуграни с номерами 0 и 1, на вторую - 2 и 3, и т.д. На главной странице создадим раздел для сообщения о цвете куба:

```
<div id = "info" style-'position: absolute; "> Кликните на сторону кубика </div>
```

Учтем это и создадим функцию клика на кубе:

```
function onDocumentMouseDown( event )
 {
 var vector = new THREE.Vector3( ( event.clientX / window.innerWidth ) * 2 - 1,
                                 - ( event.clientY / window.innerHeight ) * 2 + 1, 0.5 );
 projector.unprojectVector( vector, camera );
 var raycaster = new THREE.Raycaster( camera.position,
                                      vector.sub( camera.position ).normalize() );
 var intersects = raycaster.intersectObjects( objects );
 if ( intersects.length > 0 )
 {
  var index = Math.floor( intersects[0].faceIndex / 2 );
  switch (index)
  {
   case 0:sssr = 'Вы нажали на красную грань!'; break;
   case 1:sssr = 'Вы нажали на зеленую грань!'; break;
   case 2:sssr = 'Вы нажали на синюю грань!'; break;
   case 3:sssr = 'Вы нажали на пурпурную грань!'; break;
   case 4:sssr = 'Вы нажали на желтую грань!'; break;
   case 5:sssr = 'Вы нажали на светло-голубую грань!'; break;
  }
  info.innerHTML = sssr;
 }
}
```

Результат (полный код - в файлах **ex14_09.html, ex14_09.js**):

Рис. 14.11

Эффект изменения формы курсора мышки при наведении на кубик реализуется аналогично примеру **ex14_07.html**.

Контрольные вопросы

1. Анимация объектов.
2. Использование requestAnimationFrame.
3. Обработка пользовательского ввода.