Программирование графических приложений

Тема 16
Импорт моделей из графических редакторов

Цель изучения темы. Изучение методов импорта моделей из графических редакторов в WebGL.

До сих пор мы создавали трехмерные объекты, указывая координаты вершин и информацию о цвете вручную, в виде массивов типа Float32Array. Существуют модели, которые существенно проще создавать в 3D-редакторе и затем экспортировать в WebGL, чем разрабатывать их средствами самого WebGL. Тогда возникает необходимость прочитать координаты вершин и информацию о цвете из файлов трехмерных моделей, сконструированных с применением инструментов трехмерного моделирования.

Рассмотрим использование в WebGL моделей, подготовленных в редакторах Blender и 3DStudioMAX.

16.1. Импорт каркасной модели из Blender

Рассмотрим сначала способ загрузки каркасной модели из Blender на примере сравнительно простого объекта – тора.

Помимо своего формата (.blend) Blender поддерживает экспорт в несколько других (obj, dxf, 3ds и др.). Для наших целей подойдет формат Collada — он довольно распространен и, что важно, этот формат является текстовым (если точнее — XML).

Для создания тора мы запускаем Blender, удаляем добавленный по умолчанию куб (Delete или X), затем нажимаем Shift+A для вызова меню добавления и в нем выбираем Mesh -> Torus.

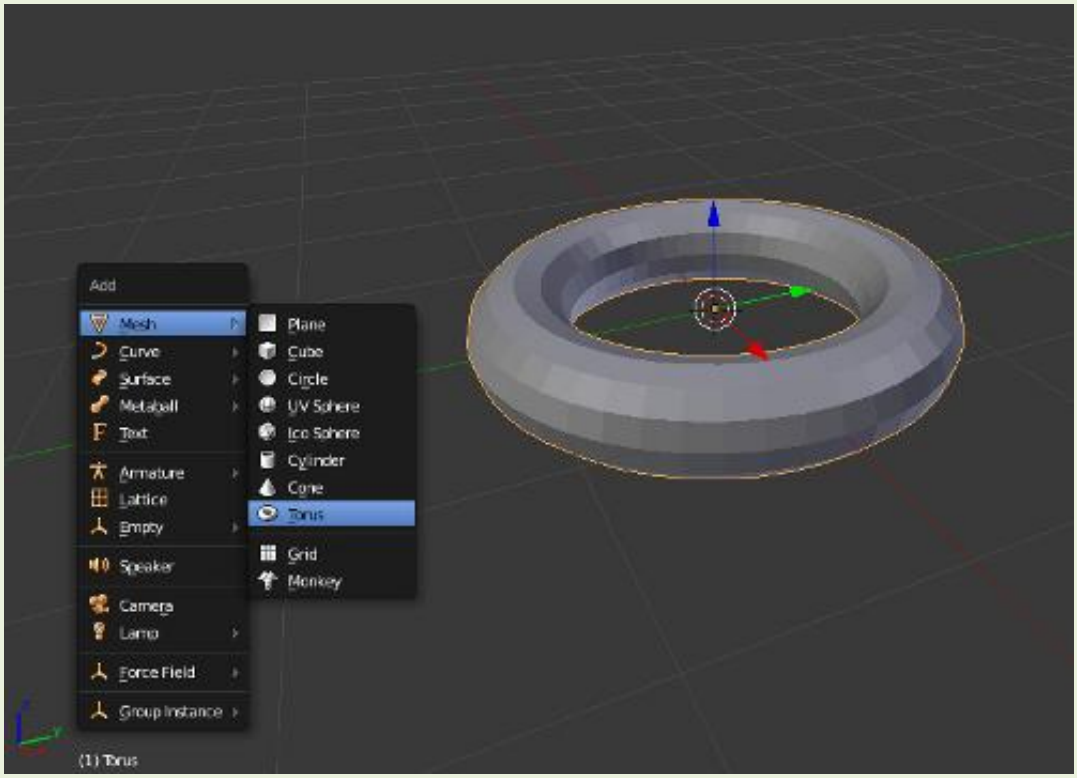

Рис. 16.1

Вот и все действия, которые были нужны для получения тора. Осталось лишь выгрузить его в нужном нам формате. Для этого выбираем File -> Export -> Collada (.dae), а затем помещаем полученный файл в директорию со скриптами проекта. При запуске примера на веб-сервере, файл, возможно, будет заблокирован из-за его расширения, поэтому лучше заменить расширение на .xml (или другой разрешенный формат).

Ключевое место в коде занимает функция, которая читает файл Collada:

```
function parseCollada(colladaXml) {
    var xmlDoc;

    // создаем парсер DOM
    if (window.DOMParser) {
        var parser = new DOMParser();
        xmlDoc = parser.parseFromString(colladaXml, "text/xml");
    } else { // Internet Explorer
        xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
        xmlDoc.async=false;
        xmlDoc.loadXML(colladaXml);
    }

    var jsonResult = {};
    // находим /COLLADA/library_geometries/geometry/mesh/source/float_array
    var mesh = xmlDoc.getElementsByTagName("mesh")[0];
    var verticesSources = mesh.getElementsByTagName("source")[0];
    var verticesNode = verticesSources.getElementsByTagName("float_array")[0];
    // в этой секции хранятся координаты через пробел - получаем их массив
    jsonResult.vertices = verticesNode.textContent.split(" ");

    // находим /COLLADA/library_geometries/geometry/mesh/polylist/p
    var polyNode = mesh.getElementsByTagName("polylist")[0];
    var indecesNode = polyNode.getElementsByTagName("p")[0];
    var allIndices = indecesNode.textContent.split(" ");
    jsonResult.indices = [];
    // нужные нам индексы вершин находятся в 0, 3, 6 и т.д.
    for (var i = 0; i < allIndices.length; i += 2) {
        jsonResult.indices.push(allIndices[i]);
    }
    return jsonResult;
}
```

Формат Collada хранит информацию о камере, освещении и многом другом. Но нам в рамках текущего примера нужны только координаты вершин и индексы. Переменная allIndices содержит индексы вершин, нормалей и UV-координаты. Нам нужны только индексы вершин, поэтому в цикле мы выбираем только каждое третье значение.

Следующая функция предназначена для загрузки модели:

```
function loadJsonModel() {
    var request = new XMLHttpRequest();
    request.open("GET", "torus.xml");
    request.onreadystatechange = function() {
        if (request.readyState == 4) {
            var colladaJson = parseCollada(request.responseText);
            vertices = colladaJson.vertices;
            indices = colladaJson.indices;
            initBuffers();
            drawScene();
        } }
    request.send();
}
```

После скачивания файла с сервера, из него считывается информация, а полученные в результате вершины и индексы записываются в глобальные переменные, которые затем помещаются в буферы и отрисовываются. Остальные функции нужны для отображения модели.

Кроме того в программе добавлено вращение по оси X.

Полный код программы – в файле **ex16_01.html**.

Результат:

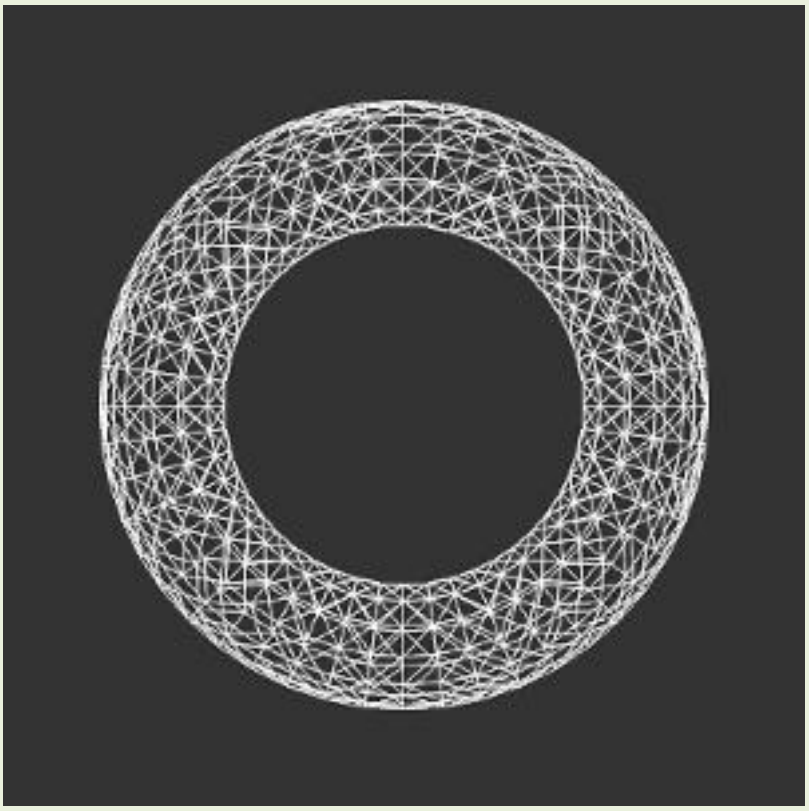

Рис. 16.2

У нас получилось загрузить из Blender каркас тора. Теперь попробуем получить сплошную фигуру, чтобы у нее был цвет и чтобы на ней был виден результат освещения. Целью будет получение сплошной фигуры с заданным цветом и подготовленными нормалями.

16.2. Импорт поверхностной модели из Blender

Так как вершины и индексы у нас уже есть, переход к сплошной фигуре выполняется проще. Во-первых, раз дело касается сплошных поверхностей, нам нужно включить проверку глубины:

```
function webGLStart() {
    var canvas = document.getElementById("canvas");
    initGL(canvas);
    initShaders();
    gl.clearColor(0.2, 0.2, 0.2, 1.0);
    gl.enable(gl.DEPTH_TEST);
    loadJsonModel();
}
```

А во-вторых, нужно заменить отрисовку линиями на отрисовку треугольниками:

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

После этих небольших изменений мы уже получим сплошной тор белого цвета.

Теперь подготовим нормали.

Начинаем с загрузки значений нормалей из XML-файла в формате COLLADA. Для этого в функции parseCollada добавляем следующие строки:

```
var jsonResult = {};
// находим /COLLADA/library_geometries/geometry/mesh/source/float_array
var mesh = xmlDoc.getElementsByTagName("mesh")[0];
var verticesSources = mesh.getElementsByTagName("source")[0];
var verticesNode = verticesSources.getElementsByTagName("float_array")[0];
// в этой секции хранятся координаты через пробел - получаем их массив
jsonResult.vertices = verticesNode.textContent.split(" ");

var normalsSources = mesh.getElementsByTagName("source")[1];
var normalsNode = normalsSources.getElementsByTagName("float_array")[0];
// по аналогии получаем нормали
jsonResult.normals = normalsNode.textContent.split(" ");

// находим /COLLADA/library_geometries/geometry/mesh/polylist/p
var polyNode = mesh.getElementsByTagName("polylist")[0];
var indecesNode = polyNode.getElementsByTagName("p")[0];
var allIndices = indecesNode.textContent.split(" ");
```

Получаем содержимое тега нормалей, в котором расположены значения нормалей через пробел, и записываем всё в массив. После этого мы помещаем нормали в глобальную переменную vertexNormals, на основании которой инициализируем буфер:

```
var vertices;
var indices;
var vertexNormals;
var vertexBuffer;
var indexBuffer;
var vertexNormalBuffer;

function initBuffers() {
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
      gl.STATIC_DRAW);
  vertexBuffer.itemSize = 3;
  vertexBuffer.numItems = vertices.length / 3;
  indexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
      gl.STATIC_DRAW);
  indexBuffer.itemSize = 1;
  indexBuffer.numItems = indices.length;
  vertexNormalBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexNormalBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertexNormals),
      gl.STATIC_DRAW);
```

```
    vertexNormalBuffer.itemSize = 3;
    vertexNormalBuffer.numItems = vertexNormals / 3;
}
```

А буфер используем при отрисовке сцены:

```
function drawScene() {
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    mat4.perspective(pMatrix, 45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0);
    mat4.identity(mvMatrix);
    mat4.translate(mvMatrix, mvMatrix, [0.0, 0, -3.0]);
    mat4.rotateX(mvMatrix, mvMatrix, degToRad(rX));
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute, vertexBuffer.itemSize,
        gl.FLOAT, false, 0, 0);
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexNormalBuffer);
    gl.vertexAttribPointer(shaderProgram.vertexNormalAttribute,
        vertexNormalBuffer.itemSize, gl.FLOAT, false, 0, 0);
    setMatrixUniforms();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
    gl.drawElements(gl.TRIANGLES, indexBuffer.numItems, gl.UNSIGNED_SHORT,
        0);
}
```

Осталось заполнить пробелы в инициализации шейдеров - включить соответствующий атрибут вершины и создать ссылку на матрицу нормалей:

```
function initShaders() {
    var fragmentShader = getShader(gl, "shader-fs");
    var vertexShader = getShader(gl, "shader-vs");
    shaderProgram = gl.createProgram();
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);
    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
        alert("Не инициалируется шейдер");
    }

    gl.useProgram(shaderProgram);
    shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
        "aVertexPosition");
    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
    shaderProgram.vertexNormalAttribute = gl.getAttribLocation(shaderProgram,
        "aVertexNormal");
    gl.enableVertexAttribArray(shaderProgram.vertexNormalAttribute);
    shaderProgram.pMatrixUniform = gl.getUniformLocation(shaderProgram,
        "uPMatrix");
    shaderProgram.mvMatrixUniform = gl.getUniformLocation(shaderProgram,
        "uMVMatrix");
    shaderProgram.nMatrixUniform = gl.getUniformLocation(shaderProgram,
        "uNMatrix");
}
```

Значение атрибута нормали берется из файла COLLADA; рассмотрим теперь происхождение матрицы нормалей. Код функции setMatrixUniforms, которая занимается передачей матриц из JavaScript в программу видеокарты:

```
function setMatrixUniforms() {
    gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false, pMatrix);
    gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false, mvMatrix);
    var normalMatrix = mat3.create();
    mat3.normalFromMat4(normalMatrix, mvMatrix);
    gl.uniformMatrix3fv(shaderProgram.nMatrixUniform, false, normalMatrix);
```

Как видно из этого фрагмента, матрица нормалей формируется из матрицы модель-вид. Для получения матрицы нормалей нужно взять верхнюю левую подматрицу 3x3 матрицы модель-вид, затем транспонировать ее и обратить (начиная с версии 2 библиотеки glMatrix за это преобразование отвечает функция normalFromMat4).

Рассмотрим как в вершинном шейдере правильно преобразовывать нормали и другие вектора, "привязанные" к объекту.

Пусть у нас есть некоторое преобразование, применяемое к вершинам объекта (грани). Не ограничивая общности можно считать, что это линейное преобразования (т.е. в нем нет переноса), так как перенос никак не влияет на вектора, "прикрепленные" к объекту.

Тогда это преобразование задается матрицей 3x3 (в вершинных шейдерах в качестве этой матрицы выступает верхняя левая 3x3 подматрица матрицы modelView) M.

Рассмотрим треугольник ABC. Пусть также задана нормаль n к нему. Тогда выполняются следующие два равенства (через круглые скобки обозначено скалярное произведение):

$$(n, B-A)=0$$
$$(n, C-A)=0$$

Матрица M переводит точки A,B и C в точки M*A, M*B и M*C, также образующие треугольник. Обозначим через n' нормаль к этому треугольнику.

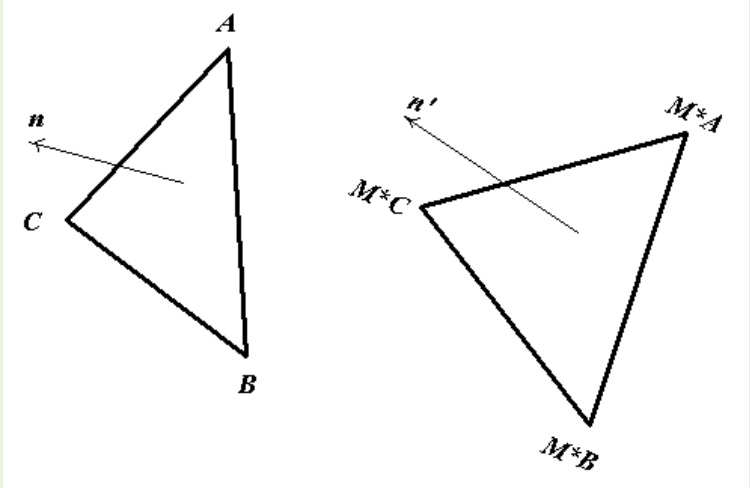

Рис. 16.3

Тогда выполняются следующие равенства:

$$(n', M\cdot(B-A))=0$$
$$(n', M\cdot(C-A))=0$$

Преобразуем их, воспользовавшись свойствами скалярного произведения.

$$0=(n', M\cdot(B-A))=(M^T\cdot n', B-A)$$
$$0=(n', M\cdot(C-A))=(M^T\cdot n', C-A)$$

Из этих равенств следует что вектор MT*n' параллелен n, т.е. отличается от него только длиной. Поскольку длина нас не интересует (направления все равно нормируются), то будем считать, что этот вектор просто совпадает с нормалью.

Тогда мы имеем:

$$M^T \cdot n' = n$$
$$n' = (M^T)^{-1}$$

Таким образом, вектора, "прикрепленные" к объекту (нормаль, касательная и т.п.) преобразуются при помощи верхней левой 3x3 подматрицы матрицы modelView, транспонированной и обращенной.

Порядок, т.е. что выполняется раньше - транспонирование или обращение - не играет роли, т.к. для любой невырожденной матрицы M всегда справедливо

$$(M^T)^{-1} = (M^{-1})^T$$

Для доказательства этого утверждения достаточно протранспонировать выражение

$M*M^{-1}=I$

и воспользоваться следующим свойством операции транспонирования: (A*B)T=(BT)*(AT).

Обратите внимание, что если матрицы modelView задает только повороты и перенос, то верхняя левая 3x3 подматрица будет ортогональной, т.е. $M^{-1} = MT$.

Тогда имеет место следующее тождество:

$$M^T \cdot n' = n$$
$$n' = (M^T)^{-1}$$

Т.е. в этом случае можно для преобразования векторов использовать просто верхнюю левую 3x3 подматрицу матрицы modelView.

На этом работа с нормалями окончена. И хотя кода было довольно много, мы не увидим изменений в отображении тора, так как нормали будут использоваться далее при добавлении освещения на сцену.

Перейдем к добавлению цвета фигуры.

Для простейшего добавления цвета можно указать его прямо в шейдере:

```
void main(void) {    gl_FragColor = vec4(0.0, 0.7, 1.0, 1.0);    }
```

Но дадим пользователю возможность задавать цвет объекта. Для этого воспользуемся, например, библиотекой jscolor (http://jscolor.com/). Скачиваем библиотеку и подключаем скрипт:

```
<script type="text/javascript" src="jscolor.js"></script>
```

Переписываем фрагментный шейдер, чтобы он мог принимать цвет из JavaScript:

```
precision mediump float;
uniform vec3 uColor;
void main(void) {    gl_FragColor = vec4(uColor, 1.0);    }
```

Добавляем uniform-переменную цвета в функцию initShaders и задаем ей начальное значение:

```
shaderProgram.colorUniform = gl.getUniformLocation (shaderProgram, "uColor");
gl.uniform3fv (shaderProgram.colorUniform, [0.0, 0.0, 1.0]);
```

Теперь добавляем на страницу сам элемент выбора цвета с тем же начальным значением цвета, которое мы задавали для шейдера:

```
<body onload="webGLStart();">
```

```
    <input class="color {onImmediateChange:'updateColor(this);'}" value="0000ff" />
    <br />
    <canvas id="canvas" width="500" height="500"></canvas>
</body>
```

И, наконец пишем обработчик на изменение цвета, который просто передает выбранный цвет в uniform-переменную:

```
function updateColor(elem) {  gl.uniform3fv(shaderProgram.colorUniform, elem.rgb); }
```

Полный код программы – в файле **ex16_02.html**.
Результат:

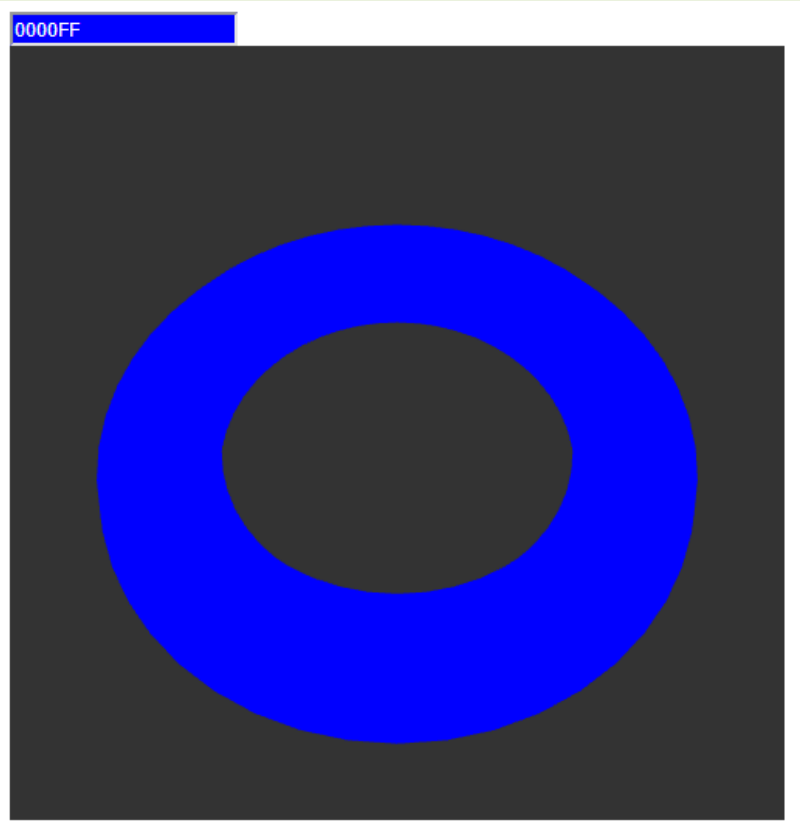

Рис. 16.4

16.3. Импорт модели из Blender в формате JSON

Экспортировать готовую трехмерную модель Blender в формат JSON можно с помощью специального плагина, который можно скачать с сайта threejs.org и подключить к Blender (https://github.com/mrdoob/three.js/tree/master/utils/exporters/blender).

Для загрузки плагина сначала выбираем его подходящую версию для вашей версии Blender (например, папка 2.66). Копируем оттуда папку io_mesh_threejs в директорию, где установлен Blender, в папку addons (например, C:\ProgramFiles\BlenderFoundation\Blender\2.66\scripts\addons).

Теперь нужно активировать плагин. Открываем настройки Blender: File - User Preferences..., выберем вкладку плагинов Addons, Import-Export, найдем three.js format и активируем плагин, установив напротив галочку.

Теперь можно в Blender экспортировать модель в формат JSON. В меню экспорта File - Export теперь доступна вкладка Three.js (.js). Установим настройки:

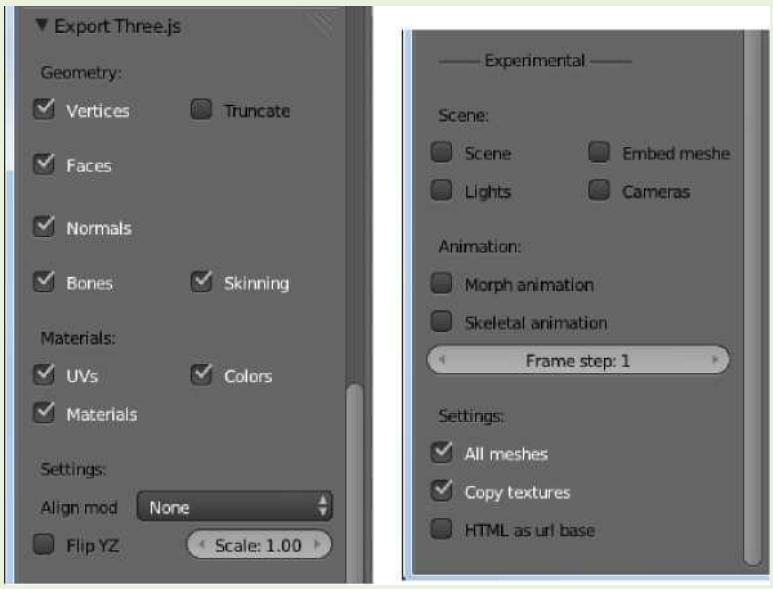

Рис. 16.5

Теперь можно экспортировать модель, нажав на кнопку Export Three.js в правом верхнем углу. Полученный файл (обычно сохраняется в той же папке, что и сама модель) будет иметь расширение js (например, penguin.js). Загрузим модель в папку нашего проекта.

Для использования нашей модели в WebGL-приложении нужно создать вспомогательную функцию

```
function addModelToScene( geometry, materials )
{
var material = new THREE.MeshFaceMaterial( materials );
var model = new THREE.Mesh( geometry, material );
model.scale.set( 3, 3, 3);
scene.add( model );
}
```

Теперь внутри init() загружаем модель с помощью всего лишь двух строчек кода:

```
var jsonLoader = new THREE.JSONLoader();
jsonLoader.load( "spaceship.js", addModelToScene );
```

Полный код программы – в файле **ex16_03.html**.
Результат:

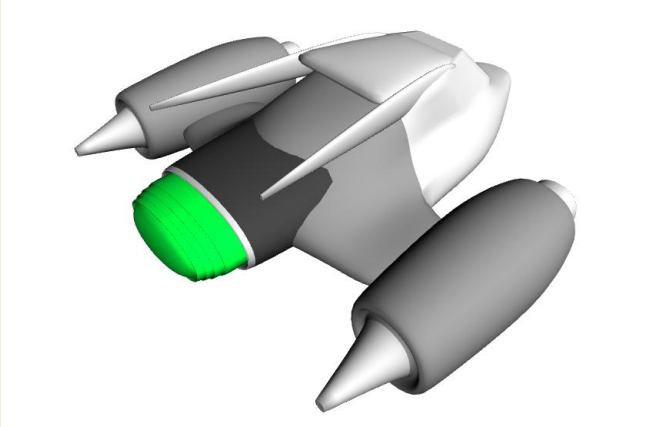

Рис. 16.6

16.2. Импорт модели из 3D Max в формате obj/mtl

Оба рассматриваемых редактора - и Blender и 3ds Max - могут экспортировать файлы трехмерных моделей в хорошо известном формате OBJ, который является текстовым и потому доступным для чтения в любом текстовом редакторе. Формат файлов с определениями геометрических построений OBJ – это открытый формат, который используется многими инструментами, предназначенными для работы с трехмерной графикой. Это означает не только широкое распространение и использование формата, но и наличие большого числа его вариаций.

Поэтому рассмотрим пример загрузки готовой модели из Autodesk 3ds Max с предварительным переводом в форматы .obj/.mtl. Файл формата .obj содержит данные о геометрии модели, файл .mtl - информацию о материале и текстурах.

Ниже приведён пример листинга с содержимым файла obj, который описывает куб:

```
mtllib cube.mtl
oCube
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
v 1.000000 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000
usemtl Material
f1234
f5876
f2673
f3784
f5148
usemtl Material.001
f1562
```

Файл в формате obj состоит из нескольких разделов, среди которых разделы с координатами вершин, определениями граней и материалов. В каждом разделе может определяться множество вершин, нормалей и граней.

Строка, начинающаяся с «mtllib», содержит имя внешнего файла материала. Формат obj поддерживает хранение информации о материале модели во внешнем файле материала (mlt-файл). В данном случае строка указывает, что файл материала имеет имя cube.mtl. Файл MTL может содержать определение нескольких материалов.

Следующая строка определяет имя объекта и обычно в webGL-программе не используется.

Далее в листинге определяются координаты вершин в формате (x,y,z[,w]), где компонент w является необязательным и по умолчанию принимает значение 1.0. В нашем примере определено восемь вершин, потому что моделью является обычный куб.

Затем в файле определяются материалы и грани, использующие эти материалы.

Строка, начинающаяся с «usemtl», определяет название материала в mlt-файле.

Далее определяются грани модели из данного материала как списки вершин, текстур и индексов нормалей: fv1v2v3v4..., где v1, v2, v3, ... - это индексы вершин, нумерация которых начинается с 1 и соответствует нумерации вершин в списке, определенном выше.

В листинге нормали не показаны, но, если грань имеет нормаль, используется следующий формат: fv1//vn1v2//vn2v3//vn3... , где vn1, vn2, vn3, ... - это индексы нормалей, нумерация которых начинается с 1.

Программа на WebGL должна читать перечисленные данные в те же структуры, которые использовались нами прежде, и реализовать следующие шаги:

1. Подготовить массив вершин и прочитать в него координаты вершин из файла модели.
2. Подготовить массив цветов и прочитать в него цвета из файла модели.
3. Подготовить массив нормалей и прочитать в него нормали из файла модели.
4. Подготовить массив индексов и прочитать в него индексы вершин, определяющие треугольники, которые составляют модель.
5. Записать данные, прочитанные при выполнении шагов с 1 по 4, в буферный объект и нарисовать модель вызовом gl.drawElements().

Рассмотрим пример загрузки готовой модели с предварительным переводом в форматы obj/mtl. Файл формата obj содержит данные о геометрии модели, файл mtl - информацию о материале и текстурах. Запустите Autodesk 3ds Max и откройте готовую модель.

Для экспорта модели потребуется нажать на кнопку Главное меню / Export. Появляется меню сохранения файла, где выбираем из списка формат obj**.** Выбираем название для модели (например, model.obj) и нажимаем кнопку Save. Откроется меню с настройками экспорта:

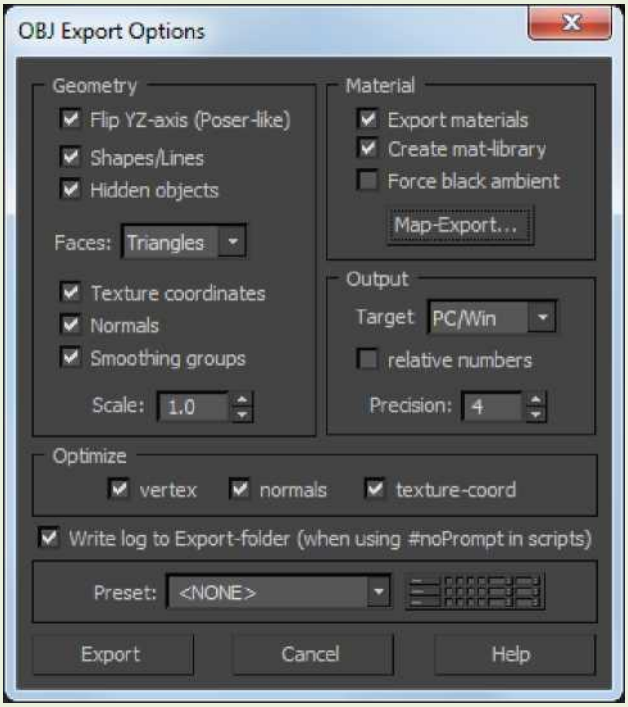

Рис. 16.7

В левой части раздела настроек Geometry нужно установить галочки на:

- Flip YZ-axis (Poser-like);
- Shapes/Lines;
- Hidden objects.

Далее, Faces установим на Triangles («треугольники») и поставим галочки на:

- Texture coordinates;
- Normals;
- Smoothing groups.

В правой части окна Material ставим галочку на:

- Export materials;

- Create mat-library.

Остальные галочки убираем. Далее открываем меню Map- Export ... и убираем все галочки.

Нажимаем на экспорт. В соответствующей папке (например, Мои документы\ 3dsMax\ export) появятся файлы model.obj и model.mtl. Скопируем файлы модели в папку нашего проекта.

Для возможности загрузки моделей нужно подключить к проекту библиотеки загрузчики **MTLLoader.js** и **OBJLoader.js** (можно скачать с сайта threejs.org):

```
<script src="MTLLoader.js"></script>
<script src="OBJLoader.js"></script>
```

Код отображения модели:

```
var loader = new THREE.MTLLoader();  // лоадер mtl
loader.load ( 'cube.mtl',
          function ( materials )
           {
            materials.preload();
            var loader2 = new THREE.OBJLoader();  // лоадер obj
            loader2.setMaterials( materials );
            loader2.load( 'cube.obj',
                        function ( object ) { scene.add( object );  }, onProgress, onError );
          } );
```

Полный код программы – в файле **ex16_04.html**.

Теперь нашу модель можно посмотреть в браузере:

Рис. 16.8

Для использования модели, созданной в Autodesk 3ds Max, кроме рассмотренных форматов могут быть использованы и другие: dae**,** vtk**,** ply, и т.д. Такие модели также можно загрузить на страницу с помощью подходящего лоадера Three.js**.**

Контрольные вопросы

1. Импорт каркасной модели из Blender.
2. Импорт модели из 3D Max.