Программирование графических приложений

Тема 9
Полигональные объекты

Цель изучения темы. Изучение методов работы с полигональными объектами в WebGL.

9.1. Полигональные объекты

Рассмотрим полигональные объекты с точки зрения их геометрии, в каком виде они хранятся в памяти приложения и как интерпретируются при рендеринге.

Форма полигонального объекта в пространстве задаётся с помощью полигональной сетки.

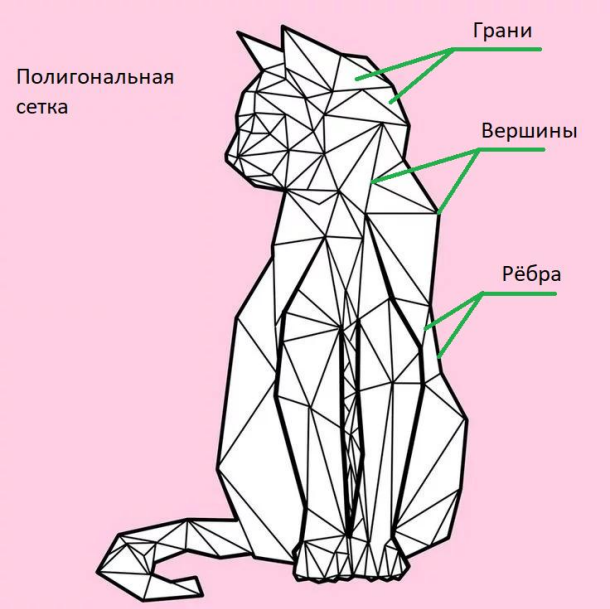

Рис. 9. 1

Вся информация о форме объекта сохраняется в его поле geometry. Она хранится в виде экземпляра объекта THREE.Geometry или его потомка. Рассмотрим структуру этого объекта.

```
var object_geometry = new THREE.Geometry();
object _geometry.vertices = [ new THREE.Vector3(  0,  0, 0),
                              new THREE.Vector3( 80,  0, 0),
                              new THREE.Vector3(  0, 80, 0) ];
object _geometry.faces.push ( new THREE.Face3( 0, 1, 2 ));
object _geometry.computeFaceNormals();
console.log (object _geometry.faces[0].normal);
```

Все вершины записываются в виде массива в поле vertices. Каждая вершина представлена как вектор THREE.Vector3 с координатами XYZ. Доступ к вершинам осуществляется по их индексам в массиве.

В поле faces также в виде массива содержится информация о гранях. Каждая грань представляет из себя объект THREE.face3. В нем вершины, образующие грань, заданы тремя полями, в которых содержатся индексы вершин из массива vertices.

Поле normal содержит вектор нормали к грани. Нормаль нужна для определения в какую сторону направлена лицевая сторона грани

По умолчанию (для оптимизации расчетов) конвеер WebGL не отрисовывает грани, развернутые к нам тыльной стороной. Какая сторона является лицевой, определяется порядком, в котором перечислены индексы вершин.

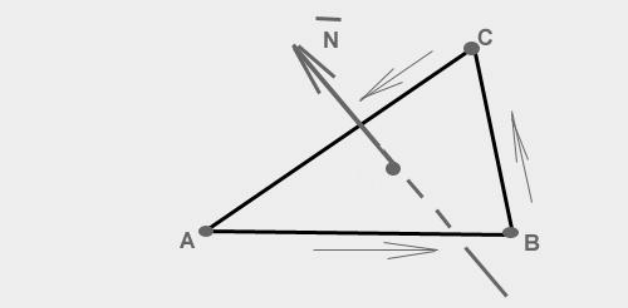

Рис. 9. 2

Мы можем задать какие грани не отрисовывать (и в каком порядке должны следовать вершины для просчета нормали) в настройках рендера или выставив соответствующие флаги в WebGL.

Библиотека three.js предоставляет несколько конструкторов для создания простых пространственных полигональных объектов, являющихся наследниками класса Geometry:

- прямоугольный параллелепипед BoxGeometry();
- сфера (или ее фрагмент) SphereGeometry();
- цилиндр (или его фрагмент) CylinderGeometry ();
- цилиндрическая поверхность с закругленными краями ExtrudeGeometry();
- додекаэдр DodecahedronGeometry();
- икосаэдр IcosahedronGeometry();
- тор (или его фрагмент) TorusGeometry();
- скрученный тор TorusKnotGeometry();
- труба TubeGeometry();
- параметрическая поверхность ParametricGeometry();
- тело вращения LatheGeometry();
- поверхность с заданным списком вершин ConvexGeometry().

Класс Geometry содержит также ряд методов, помогающих конструировать пространственные объекты:

- добавление к объекту другого объекта: merge();
- добавление к объекту полигонального объекта: mergeMesh();
- удаление дубликатов вершин из объекта с перестройкой топологии: mergeVertices();
- создание копии объекта: clone();
- центрирование объекта относительно его локальной системы координат: center();
- построение ограничивающего параллелепипеда объекта (минимальный параллелепипед, в который можно вписать объект; служит для оценки габаритов и оптимизации некоторых алгоритмов, например, просчета столкновений): computeBoundingBox();
- построение ограничивающей сферы объекта (минимальныйая сфера, в которую можно вписать объект): computeBoundingSphere().

Полное описание класса Geometry можно найти в справочных руководствах по Three.js, например: https://documentation.help/three.js-ru/overview.htm

При визуализации пространственных объектов необходимо добавлять на сцену источники света и учитывать материалы поверхностей объектов.

9.2. Источники света

Класс освещения **DirectionalLight** представляет собой источник прямого направленного освещения. Наиболее общий вид конструктора класса имеет вид:

```
DirectionalLight (hex, intensity, distance);
```

Здесь **hex** - значение **RGB**-компонента цвета. Интенсивность света **intensity** по умолчанию обычно равна единице (у всех видов освещения). Значение **distance** равно расстоянию от источника света, на котором интенсивность света станет равной нулю. Пример:

```
var directionalLight = new THREE.DirectionalLight( 0xffffff, 0.5 );
```

Класс **PointLight** представляет точечный источник света. Конструктор класса имеет аналогичный вид:

```
PointLight( hex, intensity, distance );
```

Пример:

```
var light = new THREE.PointLight( 0xff0000, 1, 100 );
light.position.set( 50, 50, 50 );
scene.add( light );
```

Класс **AmbientLight** представляет рассеянное освещение, применяемое ко всем объектам сцены. Оно не имеет направления и затрагивает каждый объект сцены в равной степени, независимо от его расположения. Соответственно, у этого света нет позиции на оси координат. Конструктор класса:

```
AmbientLight( hex );
```

Здесь **hex** - значение RGB-компонента цвета. Пример:

```
var light = new THREE.AmbientLight( 0x404040 );
scene.add( light );
```

Класс **HemisphereLight** представляет полусферическое освещение. Оно менее «жестко» в том смысле, что с ним больше переходных тонов и меньше чисто черных тонов. Конструктор класса:

```
HemisphereLight( skyColorHex, groundColorHex, intensity );
```

Класс **SpotLight** представляет прожектор. Конструктор класса:

```
SpotLight( hex, intensity, distance, angle, exponent );
```

Пример:

```
var spotLight = new THREE.SpotLight( 0xffffff );
spotLight.position.set( 100, 1000, 100 );
scene.add( spotLight ).
```

9.3. Создание материала объекта

При создании материала будущей заготовки можно указать цвет или текстуру, прозрачность. Цвет имеет формат **0xHEX**, где **HEX** - шестнадцатеричное обозначение сочетания красного, синего и зеленого цвета. Например, строка

```
color: 0xDC143C;
```

означает объявление малинового цвета. Прозрачность **opacity** меняется от 0 до 1 (0 - прозрачный, 1 - абсолютно непрозрачный) и применяется вместе со свойством **transparent**. Например:

```
var material = new THREE.MeshBasicMaterial( {
                        color: 0x33CCFF, transparent: true, opacity: 0.6 } );
```

Материалы бывают разных видов. Например, это:

- MeshBasicMaterial - для закрашивания поверхностей фигур однородным цветом;
- MeshLambertMaterial - для градиентной заливки; места, на которые падает свет, изображаются более светлыми;
- MeshNormalMaterial – позволяет подчеркнуть «трехмерность» объекта, раскрашивая его грани в разные цвета;
- MeshPhongMaterial - для блестящих поверхностей; требователен к ресурсам;
- LineBasicMaterial - материал для рисования каркасов;
- LineDashedMaterial - материал для рисования пунктирных каркасов; можно указать параметры dashSize - длину пунктира, и gapSize - длину разрыва (расстояние между пунктирами).

Некоторые материалы можно совмещать, то есть применять одновременно.

Далее в программах параметр *side* регулирует видимость материала сторон двусторонних моделей:

-THREE.FrontSide - виден снаружи (по направлению нормалей) - по умолчанию;
-THREE.BackSide - виден изнутри;
-THREE.DoubleSide - виден с обеих сторон.

9.4. Учёт нормали к поверхности

Рассмотрим простой пример программы создания объекта, состоящего из одной грани, с построением его нормали. Грань будет вращаться вокруг оси Y вместе с вектором нармали к этой грани.

Файл **ex09_01.html**

```
<!DOCTYPE HTML>
<html>
<head>
<title>Нормаль к поверхности</title>
<meta content="charset=utf-8">
<script type="text/javascript" src="three.min_.js"></script>
</head>
<body>
</body>
<script src="ex09_01.js"></script>
</html>
```

Файл **ex09_01.js**

```
var scene,camera,renderer,face;
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.translateY(30);
 camera.translateZ(150);
 scene = new THREE.Scene();
 var face_material=new THREE.MeshLambertMaterial( { color: 0xFFFFFF } );
 var face_geometry = new THREE.Geometry();
 face_geometry.vertices = [ new THREE.Vector3(  0,  0, 0),
                            new THREE.Vector3( 80,  0, 0),
                            new THREE.Vector3(  0, 80, 0) ];
 face_geometry.faces.push( new THREE.Face3( 0, 1, 2 ));
 face_geometry.computeFaceNormals();
 console.log(face_geometry.faces[0].normal);
 face = new THREE.Mesh( face_geometry, face_material );
 var axisHelper_scene = new THREE.AxisHelper( 100 );
 var edges = new THREE.FaceNormalsHelper( face, 20, 0xFF0000, 1 );
 face.add( edges );
 var light1 = new THREE.AmbientLight( 0x666666 );
 var light2 = new THREE.PointLight( 0x666666, 1, 400 );
 light2.position.set(50,50,50);
 scene.add( light1, light2, face, axisHelper_scene );
}

function animate(){
 face.rotateY( Math.PI/180 );
 renderer.render( scene, camera );
 requestAnimationFrame( animate );
}
```

Результат

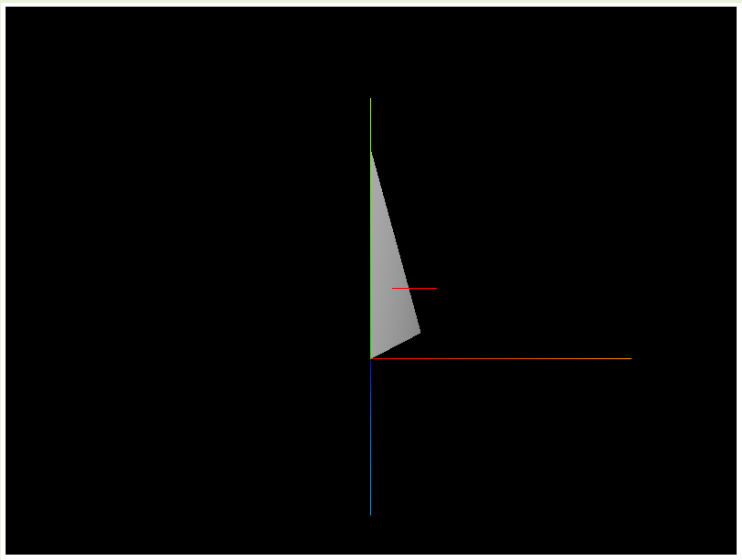

Рис. 9. 3

Метод **computeFaceNormals()** автоматически считает значение нормали в зависимости от порядка вершин.

В нашем примере порядок следования вершин - против часовой стрелки. При этом значение вектора нормали (0,0,1).

Это означает, что нормаль направлена вдоль положительного направления оси Z. Грань будет отрисовываться, пока угол между осью Z и нормалью не превысит 90°.

Немного изменим код:

```
face_geometry.vertices = [ new THREE.Vector3(  0,  0, 0),
                           new THREE.Vector3( 80,  0, 0),
                           new THREE.Vector3(  0, 80, 0) ];
face_geometry.faces.push( new THREE.Face3( 2, 1, 0 ));
face_geometry.computeFaceNormals();
console.log(face_geometry.faces[0].normal);
```

Теперь видимость грани поменяется, так как направление нормали поменяется на противоположное и примет значение (0,0,-1).

В большинстве случаев нет смысла рисовать грани, развернутые к наблюдателю задней стороной. В замкнутых объектах они закрыты от наблюдения лицевыми гранями. Исключения составляют плоские объекты, например бумага или ткань.

Для отображения граней с двух сторон можно при создании материала указать его как двусторонний.

```
scene = new THREE.Scene();
var face_material=new THREE.MeshBasicMaterial( { color: 0xFFFFFF, side:
        THREE.DoubleSide } );
var face_geometry = new THREE.Geometry();
```

Кроме собственной нормали, грань может содержать информацию о нормалях своих вершин (поле geometry.face[].vertexNormals). Они используются для расчета яркости освещения поверхности. Яркость каждой точки зависит от угла падения света и нормали в этой точке. Значение нормали каждого пикселя определяется интерполяцией между тремя вершинами (или из карты нормалей).

Если нормали вершин не заданы, для расчета освещения берётся нормаль самой грани.

Вычислить нормали вершин объекта можно вызовом метода computeVertexNormals(true). При этом если вершина принадлежит нескольким граням, вычислится среднее значение нормали.

Благодаря этому мы получим плавный переход освещения. Если вызвать метод computeVertexNormals() без параметра, в нормали вершин будет скопирована нормаль грани, и мы получим острые ребра на переходах.

Для примера создадим 4 объекта: первый оставим без нормалей; второму посчитаем нормали граней; третьему - нормали вершин, равные нормали грани; четвертому - нормали граней и нормали вершин, вызвав computeVertexNormals(true).

Файл **ex09_02.html**

```
<!DOCTYPE HTML>
<html>
<head>
<title>Варианты нормалей</title>
<meta content="charset=utf-8">
<script type="text/javascript" src="three.min_.js"></script>
</head>
<body>
</body>
```

```
<script src="ex09_02.js"></script>
</html>
```

Файл **ex09_02.js**

```
var scene, camera, renderer, obj1, obj2, obj3, obj4;
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(180);
   scene = new THREE.Scene();
   var obj_material=new THREE.MeshPhongMaterial( { color: 0xff6666, side:
         THREE.DoubleSide } );
   var obj_geometry = new THREE.Geometry();
 obj_geometry.vertices = [
         new THREE.Vector3(-30,30,10), new THREE.Vector3(30,30,10),
         new THREE.Vector3(-30,20,0), new THREE.Vector3(30,20,0),
         new THREE.Vector3(-30,10,-10), new THREE.Vector3(30,10,-10),
         new THREE.Vector3(-30,-10,-10), new THREE.Vector3(30,-10,-10),
         new THREE.Vector3(-30,-20,0), new THREE.Vector3(30,-20,0),
         new THREE.Vector3(-30,-30,10), new THREE.Vector3(30,-30,10)
 ];
 obj_geometry.faces = [
         new THREE.Face3(0,3,1), new THREE.Face3(0,2,3),
         new THREE.Face3(2,5,3), new THREE.Face3(2,4,5),
         new THREE.Face3(4,7,5), new THREE.Face3(4,6,7),
         new THREE.Face3(6,9,7), new THREE.Face3(6,8,9),
         new THREE.Face3(8,11,9), new THREE.Face3(8,10,11)
 ];
 obj1 = new THREE.Mesh( obj_geometry, obj_material );
 obj2 = new THREE.Mesh( obj_geometry.clone(), obj_material.clone() );
 obj3 = new THREE.Mesh( obj_geometry.clone(), obj_material.clone() );
 obj4 = new THREE.Mesh( obj_geometry.clone(), obj_material.clone() );
 obj1.position.set( -40, 40, 0);
 obj2.position.set( 40, 40, 0);
 obj3.position.set( -40, -40, 0);
 obj4.position.set( 40, -40, 0);
 obj2.geometry.computeFaceNormals();
 obj3.geometry.computeVertexNormals(false);
 obj4.geometry.computeVertexNormals(true);
   var light = new THREE.AmbientLight( 0x666666 );
   var light1 = new THREE.DirectionalLight( 0xffffff, 0.5 );
   light1.position.set(100, 100, 100);
   light1.lookAt(new THREE.Vector3(  0, 0, 0 ));
   scene.add( light, light1, obj1, obj2, obj3, obj4 );
}

function animate(){
```

```
 obj1.rotateY( Math.PI/180 );
 obj2.rotateY( Math.PI/180 );
 obj3.rotateY( Math.PI/180 );
 obj4.rotateY( Math.PI/180 );
   renderer.render( scene, camera );
   requestAnimationFrame( animate );
}
```

Результат:

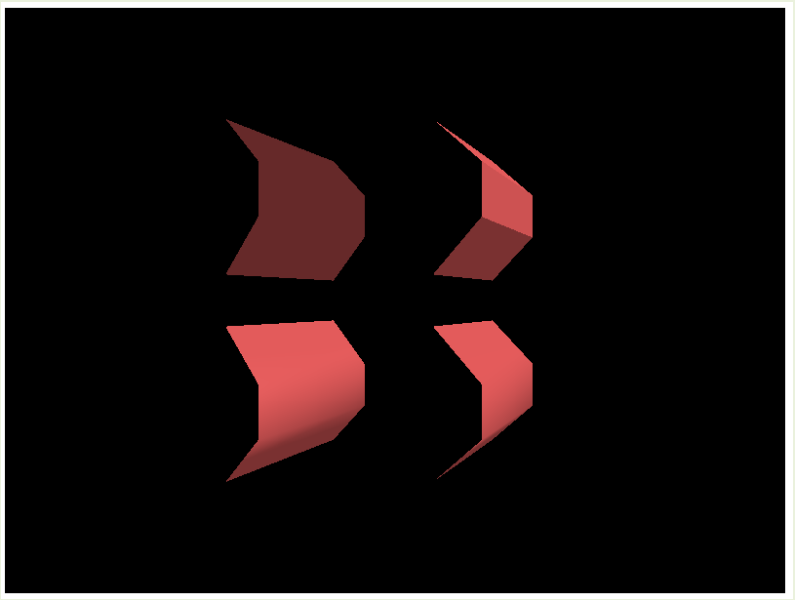

Рис. 9. 4

Из примера видно, что в 1 и 3 случае освещение не просчитывается из-за отсутствия нормалей, во 2 освещение считается по нормали к грани, а в 4 переход между гранями сглажен.

9.5. Рисование осей и координатной сетки

Библиотека Three.js содержит готовые методы для добавления координатных осей и координатной сетки.

Для изображения осей служит метод AxisHelper. Его единственный аргумент - длина осей. Этот метод рисует сразу все три оси.

Для изображения сетки служит метод GridHelper. Он имеет два параметра - size (длина сетки) и step (шаг сетки). Центр сетки приходится по умолчанию на начало координат. Метод setColors позволяет установить цвет линий сетки. Первый аргумент метода отвечает за цвет центральных линий, второй аргумент - за цвет остальных линий сетки.

Тогда оси и три координатные сетки можно изобразить, например, так (**ex09_03.html**):

```
var axes = new THREE.AxisHelper(300);
axes.position.set( 0,0,0 );
scene.add(axes);

var gridXZ = new THREE.GridHelper(100, 20);
gridXZ.setColors( new THREE.Color(0x006600),
new THREE.Color(0x006600) );
gridXZ.position.set( 100,0,100 );
scene.add(gridXZ);
```

```
var gridXY = new THREE.GridHelper(100, 20);
gridXY.position.set( 100,100,0 );
gridXY.rotation.x = Math.PI/2;
gridXY.setColors( new THREE.Color(0x000066), new THREE.Color(0x000066) );
scene.add(gridXY);

var gridYZ = new THREE.GridHelper(100, 20);
gridYZ.position.set( 0,100,100 );
gridYZ.rotation.z = Math.PI/2;
gridYZ.setColors( new THREE.Color(0x660000), new THREE.Color(0x660000) );
scene.add(gridYZ);
```

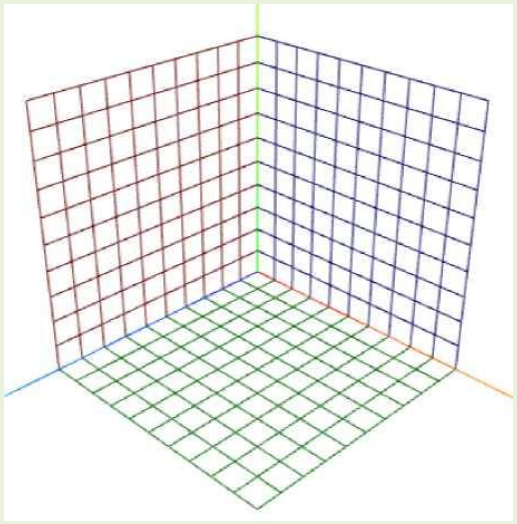

Рис. 9. 5

9.6. Создание пространственных примитивов

Построим сначала несколько простых примитивов с использованием библиотеки three.js. Пользователь сможет управлять камерой мышью, чтобы иметь возможность приближать и поворачивать сцену.

Создание сферы

Создадим, например, шар синего цвета. Последовательно создаем геометрию, материал и сеть:

```
var geometry = new THREE.SphereGeometry(100, 50, 50);
var material = new THREE.MeshLambertMaterial({ color: 0x33CCFF } );
var Sphere1 = new THREE.Mesh( geometry, material );
```

Первый параметр SphereGeometry - радиус сферы; второй и третий параметры - количество сегментов по ширине и высоте сферы. Чем их больше, тем точнее изображается сфера. В качестве материала мы выбрали градиентную заливку. Указываем позицию:

```
Sphere1.position.x = 0;
Sphere1.position.y = 20;
```

Добавляем на сцену:

```
scene.add  (Sphere1);
```

Результат:

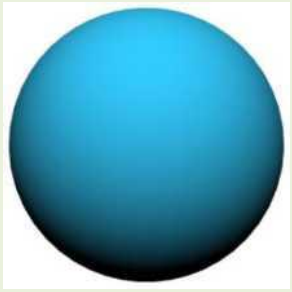

Рис. 9.6

Класс Mesh имеет метод scale, позволяющий растягивать геометрию вдоль указанной оси. Укажем

```
Sphere1.scale.x = 1.5;
```

и получим эллипсоид:

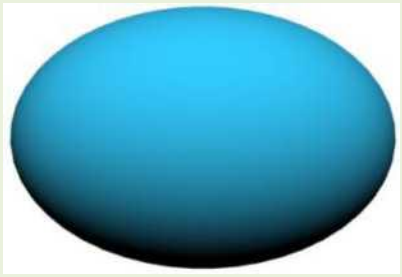

Рис. 9.7

Посмторим, как количество сегментов влияет на изображение. Если в первом примере, например, изменить количество сегментов на значения 12 и 8

```
var geometry = new THREE.SphereGeometry( 100, 12, 8 );
```

то результат будет выглядеть совсем по-другому:

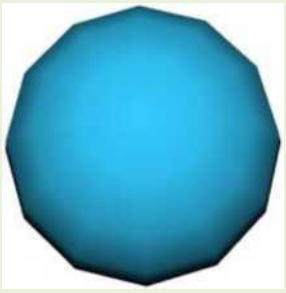

Рис. 9.8

Видим, что чем больше сегментов, тем более реалистичным будет изображение. Но и больше уйдет ресурсов компьютера на прорисовку такой сферы. На практике обычно выбирается разумный компромисс между быстродействием и реалистичностью.

Создание параллелепипеда

Пример создания прямоугольного параллелепипеда с помощью класса BoxGeometry уже рассматривался ранее. Более общее объявление класса имеет вид:

```
BoxGeometry ( width, height, depth, widthSegments, heightSegments, depthSegments );
```

Кроме уже знакомых нам первых трех параметров, отвечающих за ширину, высоту и длину, вторая тройка обозначает количество сегментов, необходимых для детализации изображения соответствующих сторон параллелепипеда. Эти параметры особенно актуальны при наличии текстуры. Если их не указывать, они равны единице.

Вернемся к примеру, когда мы создавали прямоугольный параллелепипед с окраской граней одинаковым цветом. Чтобы окрасить каждую грань в свой цвет, необходимо объявить массив из 6 материалов:

```
var materials = [
new THREE.MeshBasicMaterial( { color: 0xff0000 }), // правая сторона красная
new THREE.MeshBasicMaterial( { color: 0x00ff00 }), // левая сторона зеленая
new THREE.MeshBasicMaterial( { color: 0x0000ff }), // верх синий
new THREE.MeshBasicMaterial( { color: 0xff00ff }),  // низ пурпурный
new THREE.MeshBasicMaterial( { color: 0xffff00 }),  // лицевая сторона желтая
new THREE.MeshBasicMaterial( { color: 0x00ffff })   // задняя сторона цвета циан
];
```

Указываются соответственно правая, потом левая стороны, верх и низ, лицевая и задняя стороны. Для «объединения» массива материалов используется метод **MeshFaceMaterial**:

```
var material = newTHREE.MeshFaceMaterial( materials );
```

Геометрию задаем аналогично:

```
var geometry = newTHREE.BoxGeometry(100, 150, 200);
```

Теперь добавляем параллелепипед, указывая построенный материал:

```
cube = new THREE.Mesh( geometry, material );
cube.rotation.y = - Math.PI / 6;
scene.add( cube );
```

Результат:

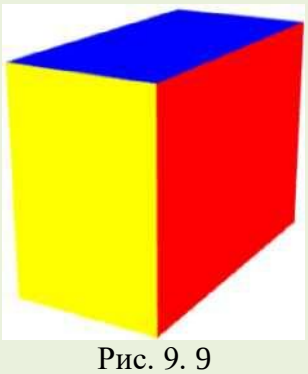

Рис. 9. 9

Создание пирамиды, призмы, цилиндра и конуса

Все эти поверхности создаются с помощью одной и той же команды CylinderGeometry. Создадим геометрию:

```
var radius_top = 0;
var radius_bottom = 128;
```

```
var heigth = 240;
var segments = 3;
var geometry = new THREE.CylinderGeometry (radius_top, radius_bottom, heigth,
        segments );
```

Указываются радиусы верхнего и нижнего основания, высота фигуры, и количество сегментов, равное количеству сторон многоугольников на основаниях. Создадим материал

```
var material= newTHREE.MeshNormalMaterial({color: 0xf2ddc6});
```

Построим треугольную **пирамиду**:

```
var piramida = new THREE.Mesh( geometry, material );
piramida.position.set(-20,0,100);
scene.add( piramida );
```

Результат:

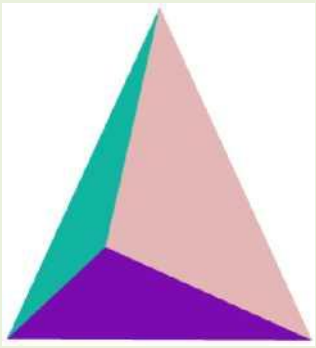

Рис. 9. 10

Выбрав же верхний и нижний радиус одинаковыми, получим **треугольную призму**:

```
var radius_top = 128;
var radius_bottom = 128;
var heigth = 240;
var segments = 3;
var geometry = new THREE.CylinderGeometry(radius_top, radius_bottom, heigth,
        segments );
var material= newTHREE.MeshNormalMaterial({color: 0xf2ddc6});
var prizma = new THREE.Mesh( geometry, material );
prizma.position.set(-20,0,100);
scene.add(prizma);
```

Результат:

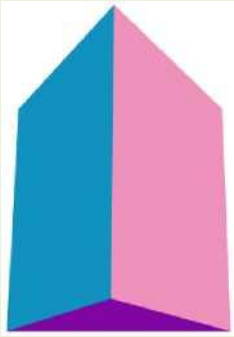

Рис. 9. 11

Если увеличивать количество сторон в основании, будем получать фигуры, всё более похожие на **цилиндр**. Например, при 16-угольнике в основнаии фигуру:

```
var radius_top = 128;
var radius_bottom = 128;
var heigth = 240;
var segments = 16;
var geometry = new THREE.CylinderGeometry (radius_top, radius_bottom, heigth,
        segments );
var material= newTHREE.MeshNormalMaterial ({color: 0xf2ddc6});
var cilindr = new THREE.Mesh (geometry, material);
cilindr.position.set (-20,0,100);
scene.add (cilindr);
```

Результат:

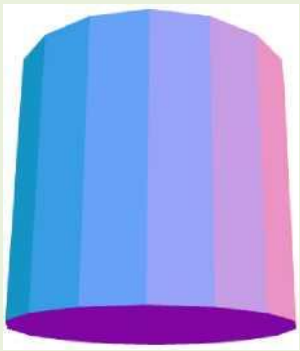

Рис. 9.12

Понятно, что чем больше количество сегментов в основании, тем больше будет похожа фигура на цилиндр. И, наконец, уменьшив верхний радиус до нуля, получим **конус**:

```
var radius_top = 0;
var radius_bottom = 128;
var heigth = 240;
var segments = 16;
var geometry = new THREE.CylinderGeometry (radius_top, radius_bottom, heigth,
        segments );
var material= newTHREE.MeshNormalMaterial ({color: 0xf2ddc6});
var cone = new THREE.Mesh (geometry, material);
cone.position.set (-20,0,100);
scene.add (cone);
```

Результат:

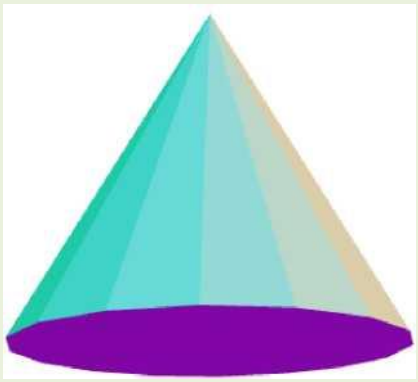

Рис. 9. 13

Наконец, взяв верхний радиус ненулевым, но отличным от нижнего,получим усеченный конус:

```
var radius_top = 64;
var radius_bottom = 128; var heigth = 240;
var segments = 16;
var geometry = new THREE.CylinderGeometry (radius_top, radius_bottom, heigth,
        segments );
var material= newTHREE.MeshNormalMaterial ({color: 0xf2ddc6});
var ucone = new THREE.Mesh (geometry, material);
ucone.position.set (-20,0,100);
scene.add (ucone);
```

Результат:

Рис. 9. 14

В файлах **ex09_04.html** и **ex09_04.js** содежится код программы. Для построения и управления сценой используются библиотеки three.min.js и TrackballControls.js.

9.7. Создание группы пространственных примитивов

Построим ещё несколько примитивов из библиотеки three.js. На этот раз к ним будет применён один и тот же материал и мы будем видеть сегменты поверхностей. Пользователь не сможет управлять камерой или сценой, но все примитивы группы будут вращаться вокруг своих осей с одинаковой скоростью.

Кроме прямоугольного параллелепипеда объекты будут более сложные:

- прямоугольный параллелепипед со сторонами 20, 40, 10, разбитыми на 2, 4, 1 сегментов соответственно

```
var box_geometry = new THREE.BoxGeometry( 20, 40, 10, 2, 4, 1);
```

- фрагмент поверхности сферы радиусом 20 с угловыми размерами 90°, разбитую на 6 сегментов по X и на 8 по Y (если отбросить последние 4 аргумента, получим полную сферу)

```
var sphere_geometry = new THREE.SphereGeometry( 20, 6, 8,0, 90*Math.PI/180,
        45*Math.PI/180, 90*Math.PI/180 );
```

- фрагмент поверхности цилиндра с верхним радиусом 20, нижним 15, высотой 30

```
var cylinder_geometry = new THREE.CylinderGeometry (20,15,30,8,4,false,0,
        270*Math.PI/180);
```

- додекаэдр - фигура, состоящая из правильных пятиугольников

```
var dodecahedron_geometry = new THREE.DodecahedronGeometry( 20, 0 );
```

- икосаэдр - фигура, состоящая из правильных треугольников

```
var icosahedron_geometry = new THREE.IcosahedronGeometry( 20, 0 );
```

- фрагмент поверхности тора; задается 2 радиусами, количеством сегментов и угловым размером

```
var torus_geometry = new THREE.TorusGeometry( 15, 5, 8, 16, 270*Math.PI/180 );
```

- узел из тора, последние 2 параметра - количество скручиваний вдоль кольца и поперек

```
var torusknot_geometry = new THREE.TorusKnotGeometry( 15, 2, 64, 8, 4, 5 );
```

- труба: окружность, выдавленная вдоль заданной траектории (в нашем случае спираль)

```
var CustomSinCurve = THREE.Curve.create (
function ( scale ) { this.scale = scale; },
   function (t) { var tx=Math.cos(8*Math.PI*t), ty=Math.sin(8*Math.PI*t), tz = t*4;
                  return new THREE.Vector3(tx, ty, tz).multiplyScalar(this.scale); } );
var path = new CustomSinCurve( 10 );
var tube_geometry = new THREE.TubeGeometry( path, 80, 2, 8, false );
```

- параметрический объект (в нашем случае квадратичная поверхность)

```
var parametric_geometry = new THREE.ParametricGeometry( function( u, v ){
 var k = 20;
 var tx = (2*u-1)*k, tz = (2*v-1)*k;
 var g = Math.sqrt(tx*tx+tz*tz);
 var ty = g*g/k-k;
 return new THREE.Vector3( tx, ty, tz );
}, 8, 8 );
```

Полный код программы – в файлах **ex09_05.html**, **ex09_05.js**.

Результат:

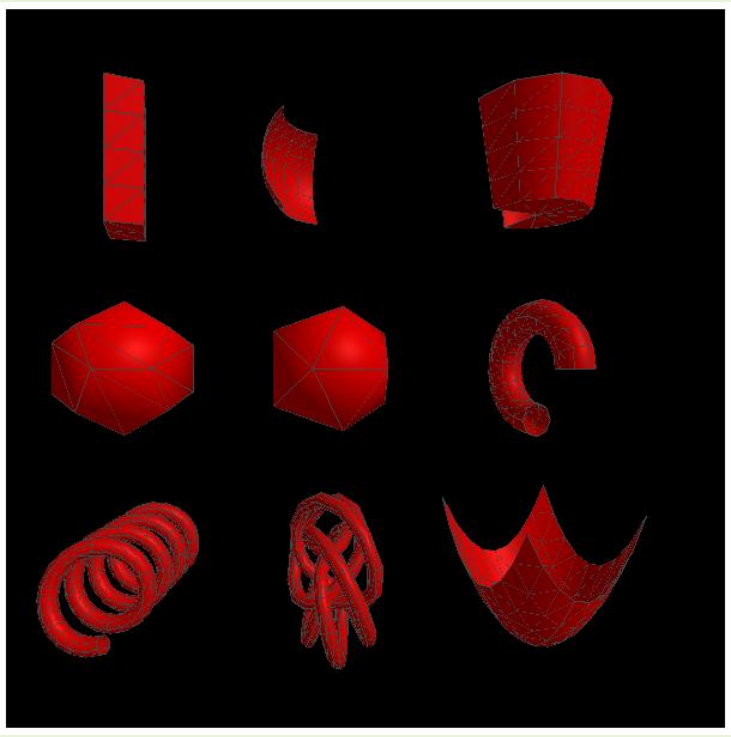

Рис. 9. 15

9.8. Пример конструирования сложных объектов

В примере конструирования сложных объектов будут использованы методы THREE.Geometry.

Метод **merge** добавляет к объекту другой объект:

merge (объект, [матрица преобразований, [смещение индекса материалов]])

Пример:

```
var sphere1 = new THREE.SphereGeometry( 20, 16, 16, 0, Math.PI );
var murshrum1 = new THREE.Geometry();
murshrum1.merge (sphere1, new THREE.Matrix4().set(
        1,                  0,                  0,  0,
        0, Math.cos( -Math.PI/2 ), -Math.sin( -Math.PI/2 ), 40,
        0, Math.sin( -Math.PI/2 ),  Math.cos( -Math.PI/2 ),  0,
        0,                  0,                  0,  1
        ));
```

Второй и третий параметры необязательные.

Метод **mergeMesh** добавляет к объекту полигональный объект:

```
margeMesh( полигональный объект )
```

Пример:

```
murshrum1obj = new THREE.Mesh( murshrum1, new THREE.MeshFaceMaterial(
        murshrum_material ));
var murshrum2 = new THREE.Geometry();
murshrum2.mergeMesh( murshrum1 );
murshrum2obj = new THREE.Mesh( murshrum2, new THREE.MeshFaceMaterial(
        murshrum_material ));
```

Метод **mergeVertices** удаляет дубликаты вершин из объекта и перестраивает топологию:

```
mergeVertices()
```

Пример:

```
console.log( "вершин до: " + murshrum1.vertices.length );
murshrum1.mergeVertices();
console.log( "вершин после: " + murshrum1.vertices.length );
```

В примере ниже объединены 4 объекта-примитива в один составной. После удаления дубликатов на стыках, число вершин сократилось с 455 до 322. Дубликатами считаются вершины, которые находятся друг от друга на растоянии менее чем 0.0001

Метод **clone** создает копию объекта.

```
var box1 = new THREE.BoxGeometry( 10, 10, 10 );
var box2 = box1.clone();
```

Метод **center** центрирует геометрию объекта относительно его локальной системы координат.

```
murshrum2.geometry.center();
```

Объект murshrum2 в примере ниже является копией murshrum1. При склеивании murshrum1 он был специально сдвинут относительно центра, чтобы на примере его клона продемонстрировать центрирование.

Метод **computeBoundingBox** считает ограничивающий прямоугольный параллелепипед объекта.

```
var box1 = new THREE.BoxGeometry( 10, 10, 10 );
box1.computeBoundingBox();
```

Ограничивающий прямоугольный параллелепипед (BoundingBox) - минимальный параллелепипед, в который можно вписать объект. Служит для оценки габаритов и оптимизации некоторых алгоритмов (например, просчет столкновений).

Метод **computeBoundingSphere** считает ограничивающую сферу объекта.

```
var box1 = new THREE.BoxGeometry( 10, 10, 10 );
box1.computeBoundingSphere();
```

Ограничивающая сфера (BoundingSphere) - **м**инимальная сфера, в которую можно вписать объект.

Файл **ex09_06.html**

```
<!DOCTYPE HTML>
<html>
<head>
<title>Сложные объекты</title>
<meta content="charset=utf-8">
```

```
<script type="text/javascript" src="three.min_.js"></script>
</head>
<body>
</body>
<script src="ex09_05.js"></script>
</html>
```

Файл **ex09_06.js**

```
var scene, camera, renderer, murshrum1, bb_murshrum1, murshrum2, bb_murshrum2;
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(150);
   scene = new THREE.Scene();
   var murshrum_material = [
          new THREE.MeshPhongMaterial( { color: 0x723D16 } ),
          new THREE.MeshPhongMaterial( { color: 0xFFFFFF } ),
          new THREE.MeshPhongMaterial( { color: 0x666666 } )
];

var sphere = new THREE.SphereGeometry( 20, 16, 16, 0, Math.PI );
var ring = new THREE.RingGeometry( 20, 10, 32, 1 );
var cylinder = new THREE.CylinderGeometry( 10, 10, 40, 32, 1, true );
var circle = new THREE.CircleGeometry( 10, 32);
var murshrum1 = new THREE.Geometry();
murshrum1.merge( sphere, new THREE.Matrix4().set(
          1,                  0,                  0,  0,
          0, Math.cos( -Math.PI/2 ), -Math.sin( -Math.PI/2 ), 40,
          0, Math.sin( -Math.PI/2 ),  Math.cos( -Math.PI/2 ),  0,
          0,                  0,                  0,  1
));
murshrum1.merge( ring, new THREE.Matrix4().set(
          1,                  0,                  0,  0,
          0, Math.cos( -Math.PI/2 ), -Math.sin( -Math.PI/2 ), 40,
          0, Math.sin( -Math.PI/2 ),  Math.cos( -Math.PI/2 ),  0,
          0,                  0,                  0,  1
), 1);
murshrum1.merge( cylinder, new THREE.Matrix4().set(
          1, 0, 0,  0,
          0, 1, 0, 20,
          0, 0, 1,  0,
          0, 0, 0,  1
), 2);
murshrum1.merge( circle, new THREE.Matrix4().set(
          1,                  0,                  0, 0,
          0, Math.cos( Math.PI/2 ), -Math.sin( Math.PI/2 ), 0,
          0, Math.sin( Math.PI/2 ),  Math.cos( Math.PI/2 ), 0,
```

```
            0,                  0,                  0, 1
), 2);

console.log( "вершин до: " + murshrum1.vertices.length );
murshrum1.mergeVertices();
console.log( "вершин после: " + murshrum1.vertices.length );
murshrum1.computeBoundingBox();
murshrum1 = new THREE.Mesh( murshrum1, new THREE.MeshFaceMaterial(
        murshrum_material ));
var murshrum2 = new THREE.Geometry();
murshrum2.mergeMesh( murshrum1 );
murshrum20 = new THREE.Mesh( murshrum2, new THREE.MeshFaceMaterial(
        murshrum_material ));
murshrum20.geometry.center();
murshrum1.translateX( -40 );
murshrum20.translateX(  40 );

bb_murshrum1 = new THREE.BoundingBoxHelper( murshrum1, 0xff0000 );
bb_murshrum2 = new THREE.BoundingBoxHelper( murshrum20, 0xff0000 );
var light = new THREE.AmbientLight( 0x666666 );
    var light1 = new THREE.DirectionalLight( 0xffffff, 0.5 );
    light1.position.set(100, 100, 100);
light1.lookAt(new THREE.Vector3(  0, 0, 0 ));

  scene.add( light, light1, murshrum1, bb_murshrum1, murshrum20, bb_murshrum2);
}

function animate(){
 murshrum1.rotateX( Math.PI/180 );
 bb_murshrum1.update();
 murshrum20.rotateX( Math.PI/180 );
 bb_murshrum2.update();
  renderer.render( scene, camera );
  requestAnimationFrame( animate );
}
```

Результат:

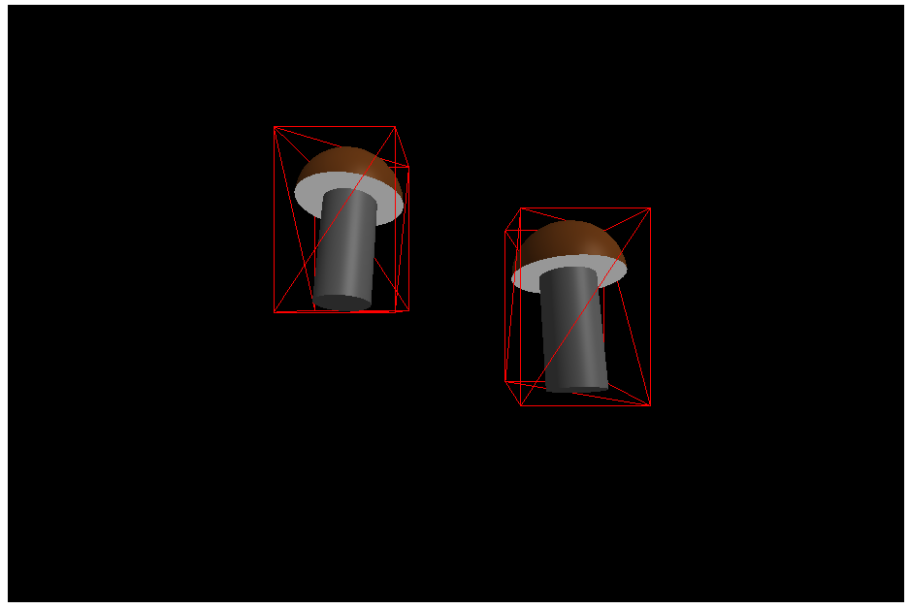

Рис. 9. 16

9.9. Создание структурных объектов с использованием Object3D

Для создания структурных объектов можно использовать класс трехмерных объектов **Object3D**. При создании сложной фигуры все ее «запчасти» добавляются не на сцену, а «внутрь» объекта, и лишь потом объект выводится на сцену.

Технология применяется когда приходится создавать сложную фигуру из нескольких простых объектов. Чтобы они двигались как единое целое, вместо того, чтобы каждому объекту задавать одно и то же, общее для всех, правило движения, можно сгруппировать несколько объектов в одну группу и устанавливать правила движения для этой группы.

Создадим в качестве примера модель снеговика из нескольких простых фигур.

Рис. 9. 17

Для этого сначала объявим глобальный объект:

```
var Snowman = new THREE.Object3D();
```

var Snowman = new THREE.Object3D();

Как видно из рисунка, основу снеговика составляют пять сфер. Начнем с нижней сферы-основания:

```
var material = new THREE.MeshBasicMaterial( {color: 0x33CCFF} );
var geometry = new THREE.SphereGeometry( 60, 36, 36 );
```

```
var sphere1 = new THREE.Mesh( geometry, material);
sphere1.position.set( 0, 60, 0 );
```

Теперь добавляем эту сферу, но не на сцену, а в наш объект **Object3D**:

```
Snowman.add( sphere1 );
```

Аналогично добавим остальные сферы

```
var material = new THREE.MeshBasicMaterial({color: 0x00edff});
var geometry = new THREE.SphereGeometry( 44, 36, 361 );
var sphere2 = new THREE.Mesh( geometry, material);
sphere2.position.set( 0, 140, 0 );
Snowman.add( sphere2 );
var material = new THREE.MeshBasicMaterial({color: 0xafeeee});
var geometry = new THREE.SphereGeometry( 32, 36, 36 );
var sphere3 = new THREE.Mesh( geometry, material);
sphere3.position.set( 0, 206, 0 );
Snowman.add( sphere3 );
varmaterial=newTHREE.MeshBasicMaterial({color:0x1560bd});
var geometry = new THREE.SphereGeometry( 16, 16, 16 );
var sphere4 = new THREE.Mesh( geometry, material);
sphere4.position.set( -50, 156, 0 );
Snowman.add( sphere4 );
sphere5 = sphere4.clone(); sphere5.position.set( 50, 156, 0 );
Snowman.add( sphere5 );
```

Поскольку руки снеговика одинаковые, левую руку мы создали клонированием правой. Далее изобразим нос в виде конуса:

```
var material = new THREE.MeshLambertMaterial({color: 0xf36223});
var geometry = new THREE.CylinderGeometry( 1,7, 40, 8 );
var nose = new THREE.Mesh( geometry, material);
nose.position.set(0, 202, 45);
nose.rotation.x = Math.PI/2;
Snowman.add( nose );
```

Добавляем ведро – усеченный конус:

```
var material = new THREE.MeshLambertMaterial({color: 0x6600ff});
var geometry = new THREE.CylinderGeometry( 24, 34, 60, 18 );
var bucket = new THREE.Mesh( geometry, material);
bucket.position.set(0, 249, -12);
bucket.rotation.x = -Math.PI/11;
Snowman.add( bucket );
```

Два глаза:

```
var material = new THREE.MeshLambertMaterial({color: 0x000000});
var geometry = new THREE.SphereGeometry( 5, 50, 11 );
var eye1 = new THREE.Mesh( geometry, material);
eye1.position.set( -15, 213, 27 );
```

```
Snowman.add( eye1 );
var eye2 = eye1 .clone();
eye2.position.set( 15, 213, 27 );
Snowman.add( eye2 );
```

И, наконец, рот:

```
var material = new THREE.MeshBasicMaterial({color: 0x560319});
var geometry = new THREE.CircleGeometry( 10, 10, 0, Math.PI );
var mouth = new THREE.Mesh( geometry, material);
mouth.rotation.z = Math.PI;
mouth.position.set( 0, 194, 27 );
Snowman.add( mouth );
```

Укажем позицию снеговика и добавим его на сцену. Объект **Object3D** строится от нулевой координаты, т.е. при указании координат позиции объекта в указанном месте будет находиться не центр объекта, (как, например, у куба), а его самая нижняя часть:

```
Snowman.position.set( 0, -100, 0 ); scene.add( Snowman );
```

Снеговик готов. Если теперь в рендере указать

```
Snowman.position.z+= 0.5;
```

то наш снеговик будет двигаться на нас как единое целое.

Последнюю строчку кода движения можно изменить на

```
Snowman.translateZ( 0.5 );
```

Метод translateZ(dist) увеличивает третью координату на число dist (может быть и меньше нуля). Аналогичный смысл имеют методы translateY и translateZ.

Класс Mesh создан на основе класса Object3D, поэтому также имеет эти методы.

Весь код программы - файл **ex09_07.html.**

9.10. Иерархия объектов с использованием Object3D

Для создания иерархии объектов также будет прменён объект **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.

Иерархия задает зависимость одних объектов сцены от других. Будем называть объекты, от которых зависят другие, предками; которые зависят - потомками.

Отношение предок-потомок в иерархии объектов является отношением "один ко многим". У объекта может быть сколько угодно дочерних объектов и только один родительский.

При попытке привязать один объект к нескольким предкам действует правило "кто последний, тот родитель". Таким образом иерархия сцены имеет структуру дерева, в корне которого лежит сам объект сцены.

Информация о положении объекта в иерархии лежит в его полях **parent** - предок и **children** - потомки. Поле **parent** содержит ссылку на объект предок, поле **children** массив ссылок на дочерние объекты. Вызвав метод **a.add(b);** у любого объекта **a и** передав в качестве параметра указатель на объект **b**, мы сделаем **b** дочерним объектом **a**. Таким образом в **b.parent** будет записан объект **a**, а в **a.children** добавлен в конец списка объект **b**.

Разрушается связь объектов вызовом **a.remove(b)**. При вызове этих методов в параметре можно через запятую указать сразу несколько дочерних объектов.

Например:

```
a.add( b, c, d, e, f );
```

Рассмотрим код:

```
var scene, camera, renderer, box, sphere1, sphere2, sphere3, sphere4, sphere5;
var direct=1;
var angle=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 sphere_material=new THREE.MeshLambertMaterial( { color: 0x00FF00 } );
 var sphere_geometry = new THREE.SphereGeometry( 20, 18, 18 );
 sphere1 = new THREE.Mesh( sphere_geometry, sphere_material );
 sphere2 = new THREE.Mesh( sphere_geometry, sphere_material );
 sphere2.translateX( 50 );
 sphere1.add( sphere2 );
 sphere3 = new THREE.Mesh( sphere_geometry, sphere_material );
 sphere3.translateX( 50 );
 sphere2.add( sphere3 );
 sphere4 = new THREE.Mesh( sphere_geometry, sphere_material );
 sphere4.translateX( 50 );
 sphere3.add( sphere4 );
 sphere5 = new THREE.Mesh( sphere_geometry, sphere_material );
 sphere5.translateX( 50 );
 sphere4.add( sphere5 );
 var light1 = new THREE.AmbientLight( 0x666666 );
 var light2 = new THREE.PointLight( 0x666666, 1, 400 );
 light2.position.set( 200, 200, 50 );
 scene.add( sphere1 );
 scene.add( light1 );
 scene.add( light2 );
 console.log( scene.parent );
 console.log( scene.children );
 console.log("========================");
 console.log( sphere1.parent );
 console.log( sphere1.children );
 console.log("========================");
 console.log( sphere2.parent );
 console.log( sphere2.children );
 console.log("========================");
```

```
 console.log( sphere3.parent );
 console.log( sphere3.children );
 console.log("========================");
 console.log( sphere4.parent );
 console.log( sphere4.children );
 console.log("========================");
 console.log( sphere5.parent );
 console.log( sphere5.children );
}

function animate(){
 angle+=0.005*direct;
 sphere1.rotation.z=angle;
 sphere2.rotation.z=angle;
 sphere3.rotation.z=angle;
 sphere4.rotation.z=angle;
 sphere5.rotation.z=angle;
 if ( angle>1.25 || angle<0 ) direct*=-1;
 renderer.render( scene, camera );
 requestAnimationFrame( animate );
}
```

У объекта сцены нет родителей, но есть дочерние объекты (сфера 1, расссеянный свет, точечный свет). У первой сферы родитель - сцена, потомок – сфера 2, далее у каждой сферы по одному родителю и потомку. И у последней сферы нет потомков.

Если посмотреть на результат, то видно, что преобразования, примененные к верхнему уровню иерархии, применяются ко всем элементам лежащим ниже. По аналогии если мы поднимаем плечо, то вместе с ним поднимается предплечье, кисть и пальцы.

Это связано с тем, что у каждого объекта в сцене есть две системы координат, локальная и глобальная (мировая) и в соответствии с ними две матрицы преобразования (также локльная и мировая).

Матрицы преобразования имеют размерность (4x4); при их умножении на вектор координат (x, y, z, 1) получается вектор с кординатами после преобразования (x1, y1, z1, 1).

Мировая система привязана к системе координат сцены. Применяя поворот или перемещение к объекту, мы меняем положение его локальной системы координат относительно локальной системы координат его предка.

В конце при рендеренге все локальные преобразования от объекта до сцены складываются и вычисляется матрица преобразования относительно сцены (мировая). На нее умножаются координаты всех вершин объекта, и получаем координаты вершин относительно сцены.

Полный код приведён в файлах **ex09_08.html, ex09_08.js**.

Результат работы программы:

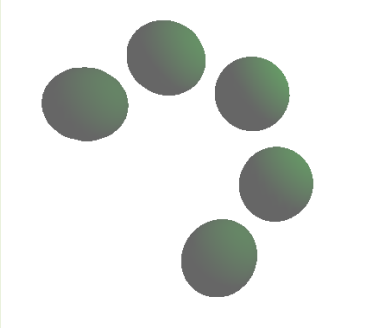

Рис.9.18

Контрольные вопросы

1. Полигональные объекты
2. Источники света
3. Создание материала объекта
4. Учёт нормали к поверхности
5. Конструирование сложных объектов