three.js学习笔记

# 目录

[目录 2](#_Toc12023)

[一、基本知识 4](#_Toc25254)

[1.1什么是WebGL 4](#_Toc3125)

[1.2什么是three.js 4](#_Toc7394)

[1.3基本组成 5](#_Toc28991)

[1.3.1Canvas标签 5](#_Toc5545)

[1.3.2渲染器（Renderer） 5](#_Toc26075)

[1.3.3场景（Scene） 6](#_Toc5922)

[1.3.4照相机（Camera） 6](#_Toc20261)

[二、照相机 7](#_Toc28487)

[2.1正交投影照相机 7](#_Toc29521)

[2.1.1参数介绍 7](#_Toc17171)

[2.1.3基本设置 7](#_Toc25761)

[2.1.4长宽比例 8](#_Toc8233)

[2.1.5照相机位置 9](#_Toc21739)

[2.1.6换个角度看世界 10](#_Toc26816)

[2.2透视投影照相机 11](#_Toc7777)

[2.2.1参数介绍 11](#_Toc21458)

[2.2.2基本设置 11](#_Toc29556)

[2.2.3竖直张角 12](#_Toc19424)

[三、几何形状 13](#_Toc30481)

[3.1基本几何形状 13](#_Toc2257)

[3.1.1立方体 13](#_Toc18312)

[3.1.2平面 14](#_Toc26027)

[3.1.3球体 15](#_Toc6618)

[3.1.4圆形 17](#_Toc14651)

[3.1.5圆柱体 18](#_Toc12351)

[3.1.6正四面体、正八面体、正二十面体 19](#_Toc28908)

[3.1.7圆环面 20](#_Toc3624)

[3.2文字形状 22](#_Toc26456)

[3.2.1下载使用 22](#_Toc7436)

[3.2.2参数介绍 22](#_Toc31832)

[3.2.2实例说明 23](#_Toc20657)

[3.3自定义形状 24](#_Toc25245)

[四、材质 26](#_Toc3812)

[4.1基本材质BasicMaterial 26](#_Toc30494)

[4.2Lambert材质 27](#_Toc32179)

[4.3Phong材质 28](#_Toc22746)

[4.4法向材质 30](#_Toc13474)

[4.5材质的纹理贴图 31](#_Toc27583)

[4.5.1单张图像应用于长方体 31](#_Toc28796)

[4.5.2六张图像应用于长方体 32](#_Toc1973)

[4.5.3棋盘格 33](#_Toc9898)

[五、网格 35](#_Toc23840)

[5.1创建网格 35](#_Toc31792)

[5.2修改属性 36](#_Toc8791)

[5.2.1材质 36](#_Toc1484)

[5.2.2位置、缩放、旋转 37](#_Toc31198)

[六、动画 38](#_Toc8647)

[6.1实现动画效果 38](#_Toc16229)

[6.1.1动画原理 38](#_Toc364)

[6.1.2 setInterval方法 38](#_Toc6625)

[6.1.3 requestAnimationFrame方法 39](#_Toc26453)

[6.1.4如何取舍 40](#_Toc25394)

[6.2使用start.js记录FDS 40](#_Toc2272)

[七、外部模型 41](#_Toc14900)

[7.1支持格式 41](#_Toc13454)

[7.2无材质的模型 41](#_Toc31283)

[7.3有材质的模型 43](#_Toc11290)

[7.3.1代码中设置材质 43](#_Toc27411)

[7.3.2建模软件中设置材质 44](#_Toc22665)

[八、 光与影 45](#_Toc22451)

[8.1环境光 45](#_Toc28551)

[8.2点源光 47](#_Toc25670)

[8.3平行光 48](#_Toc7364)

[8.4聚光灯 48](#_Toc3271)

[8.5阴影 49](#_Toc5942)

[九、 着色器 52](#_Toc13230)

[9.1渲染与着色器 52](#_Toc2409)

[9.1.1渲染 52](#_Toc11141)

[9.1.2着色器 52](#_Toc11581)

[9.1.3顶点着色器 53](#_Toc14878)

[9.1.4片元着色器 53](#_Toc16079)

[9.1.5Three.js与着色器 53](#_Toc19057)

[9.2初窥着色器 53](#_Toc18265)

[9.2.1顶点着色器 54](#_Toc27244)

[9.2.2片元着色器 55](#_Toc2116)

[9.3着色器完整实例 56](#_Toc23608)

[9.3.1着色器程序的位置 56](#_Toc20436)

[9.3.2单独的着色器文件 56](#_Toc32455)

[9.3.3HTML中的着色器代码 56](#_Toc3082)

[9.3.4完整实例 57](#_Toc12193)

[附录1：关键字 59](#_Toc21932)

[附录2：three框架 61](#_Toc6218)

# 一、基本知识

## 1.1什么是WebGL

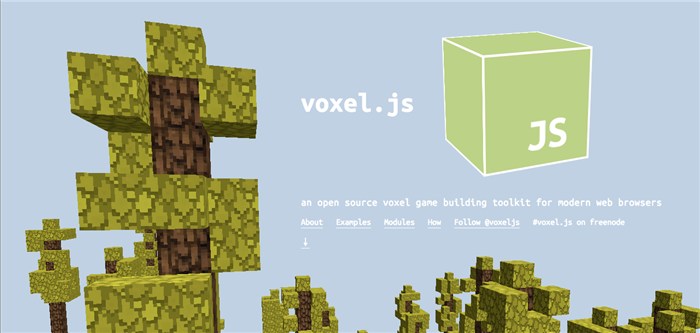
WebGL（全写Web Graphics Library）是一种3D绘图标准，这种绘图技术标准允许把JavaScript和OpenGL ES 2.0结合在一起，通过增加OpenGL ES 2.0的一个JavaScript绑定，WebGL可以为HTML5 Canvas提供硬件3D加速渲染，这样Web开发人员就可以借助系统显卡来在浏览器里更流畅地展示3D场景和模型了，还能创建复杂的导航和数据视觉化。显然，WebGL技术标准免去了开发网页专用渲染插件的麻烦，可被用于创建具有复杂3D结构的网站页面，甚至可以用来设计3D网页游戏等等。

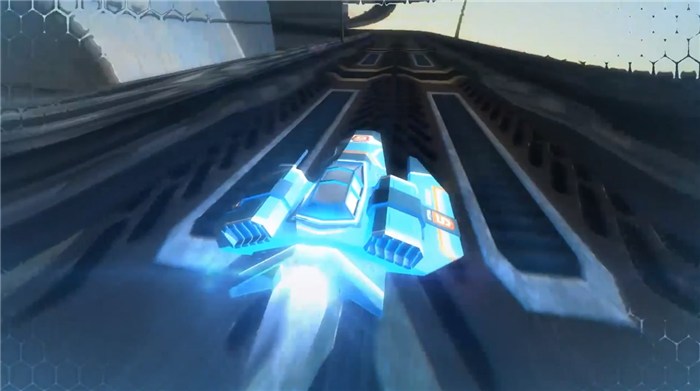
## 1.2什么是three.js

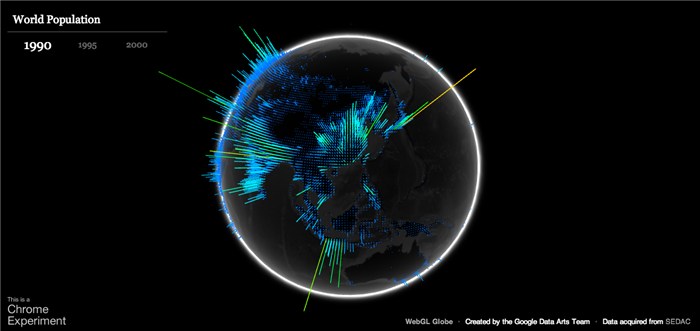
Three.js（https://github.com/mrdoob/three.js）是一个JavaScript 3D库，封装了底层的WebGL图形接口，提供简化、高效的三维图形程序开发。使用Three.js可以使你在不了解图形学知识的前提下，快速方便地开发三维图形应用。

Three.js封装了底层的图形接口，使得程序员能够在无需掌握繁冗的图形学知识的情况下，也能用简单的代码实现三维场景的渲染。我们都知道，更高的封装程度往往意味着灵活性的牺牲，但是Three.js在这方面做得很好。几乎不会有WebGL支持而Three.js实现不了的情况，而且就算真的遇到这种情况，你还是能同时使用WebGL去实现，而不会有冲突。当然，除了WebGL之外，Three.js还提供了基于Canvas、SVG标签的渲染器。

使用Three.js可以实现很多酷炫的效果，比如这个minecraft风格的网页游戏工具库[voxel.js](http://voxeljs.com/" \t "http://www.ituring.com.cn/article/_blank)：







[Mr. doob](https://github.com/mrdoob)是Three.js项目发起人和主要贡献者之一，但由于Three.js是[Github](http://github.com/mrdoob/three.js/" \t "http://www.ituring.com.cn/article/_blank)上一个开源项目，因此有非常多的[贡献者](http://github.com/mrdoob/three.js/" \t "http://www.ituring.com.cn/article/_blank)，甚至有一天，你也可以在贡献者列表中看到自己的名字。

Three.js是基于[MIT协议](https://github.com/mrdoob/three.js/blob/master/LICENSE)进行发布的，因此使用和发布都非常自由。

## 1.3基本组成

一个典型的Three.js程序至少要包括渲染器（Renderer）、场景（Scene）、照相机（Camera），以及你在场景中创建的物体。

### 1.3.1Canvas标签

<**canvas id="****warehouseCanvas" width="100%" height="100%"**></**canvas**>

Canvas相当于画布，通过原生js在画布上作画

### 1.3.2渲染器（Renderer）

渲染器将和Canvas元素进行绑定，如果之前在HTML中手动定义了id为warehouseCanvas的Canvas元素，那么Renderer可以这样写：

**var** renderer = **new *THREE***.WebGLRenderer({  
 **canvas**: **document**.getElementById(**'warehouseCanvas'**)  
});

而如果想要Three.js生成Canvas元素，在HTML中就不需要定义Canvas元素，在JavaScript代码中可以这样写：

**var** renderer = **new *THREE***.WebGLRenderer();  
renderer.setSize(400, 300);  
**document**.getElementsByTagName(**'body'**)[0].appendChild(renderer.**domElement**);

上面代码的第二行表示设置Canvas的宽400像素，高300像素。第三行将渲染器对应的Canvas元素添加到<body>中。

我们可以使用下面的代码将背景色（用于清除画面的颜色）设置为黑色：

renderer.setClearColor(0x000000);

### 1.3.3场景（Scene）

在Three.js中添加的物体都是添加到场景中的，因此它相当于一个大容器。一般说，场景里没有很复杂的操作，在程序最开始的时候进行实例化，然后将物体添加到场景中即可。

**var** scene = **new *THREE***.Scene();

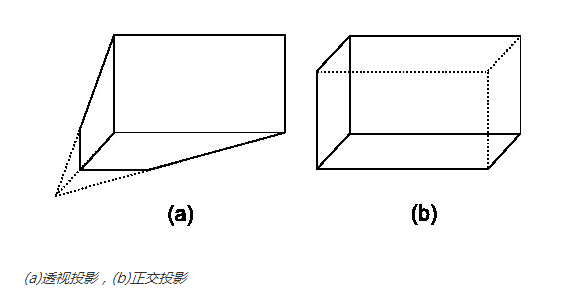
### 1.3.4照相机（Camera）

我们使用Three.js创建的场景是三维的，而通常情况下显示屏是二维的，那么三维的场景如何显示到二维的显示屏上呢？照相机就是这样一个抽象，它定义了三维空间到二维屏幕的投影方式，用“照相机”这样一个类比，可以使我们直观地理解这一投影方式。

而针对投影方式的不同，照相机又分为正交投影照相机与透视投影照相机。

使用透视投影照相机获得的结果是类似人眼在真实世界中看到的有“近大远小”的效果（如下图中的(a)）；而使用正交投影照相机获得的结果就像我们在数学几何学课上老师教我们画的效果，对于在三维空间内平行的线，投影到二维空间中也一定是平行的（如下图中的(b)）。

一般说来，对于制图、建模软件通常使用正交投影，这样不会因为投影而改变物体比例；而对于其他大多数应用，通常使用透视投影，因为这更接近人眼的观察效果。当然，照相机的选择并没有对错之分，你可以更具应用的特性，选择一个效果更佳的照相机。



# 二、照相机

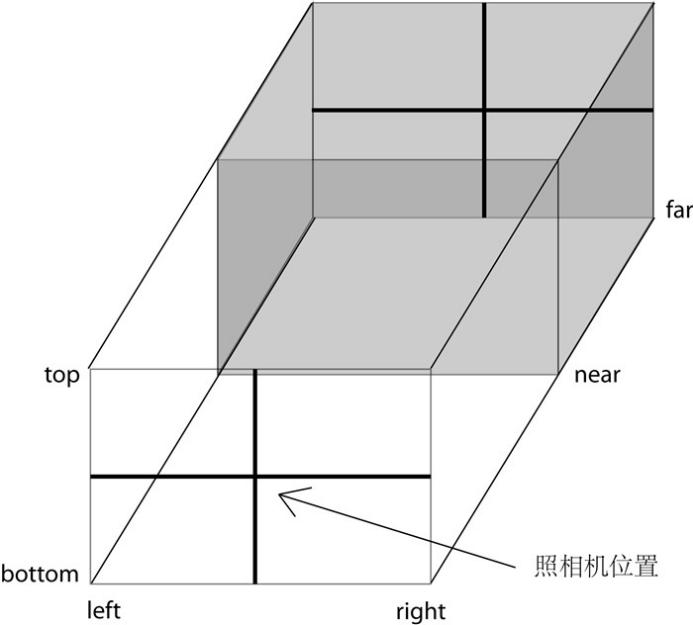
## 2.1正交投影照相机

### 2.1.1参数介绍

正交投影照相机（Orthographic Camera）设置起来较为直观，它的构造函数是：

THREE.OrthographicCamera(left, right, top, bottom, near, far)

这六个参数分别代表正交投影照相机拍摄到的空间的六个面的位置，这六个面围成一个长方体，我们称其为****视景体****（Frustum）。只有在视景体内部（下图中的灰色部分）的物体才可能显示在屏幕上，而视景体外的物体会在显示之前被裁减掉。



为了保持照相机的横竖比例，需要保证(right - left)与(top - bottom)的比例与Canvas宽度与高度的比例一致。

near与far都是指到照相机位置在深度平面的位置，而照相机不应该拍摄到其后方的物体，因此这两个值应该均为正值。为了保证场景中的物体不会因为太近或太远而被照相机忽略，一般near的值设置得较小，far的值设置得较大，具体值视场景中物体的位置等决定。

下面，我们通过一个具体的例子来解释正交投影照相机的设置。

**[例2.1.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter2/2.3.1.html)**

### 2.1.3基本设置

设置照相机：

var camera = new THREE.OrthographicCamera(-2, 2, 1.5, -1.5, 1, 10);

camera.position.set(0, 0, 5);

scene.add(camera);

在原点处创建一个边长为1的正方体，为了和透视效果做对比，这里我们使用wireframe而不是实心的材质，以便看到正方体后方的边：

var cube = new THREE.Mesh(new THREE.CubeGeometry(1, 1, 1),

new THREE.MeshBasicMaterial({

color: 0xff0000,

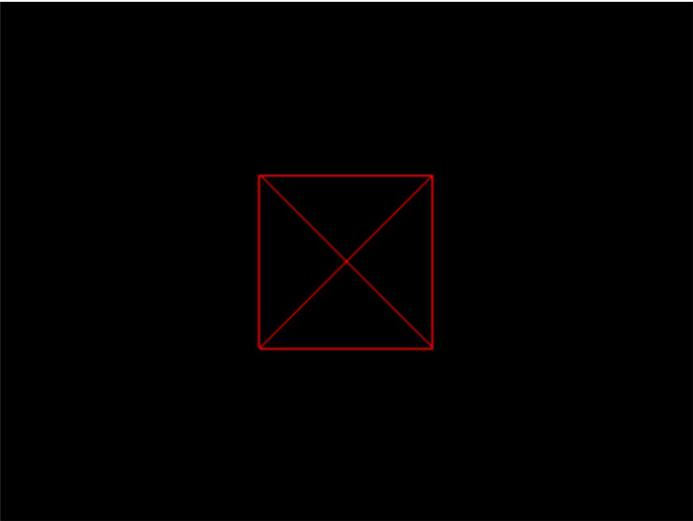
wireframe: true

})

);

scene.add(cube);

得到的效果是：



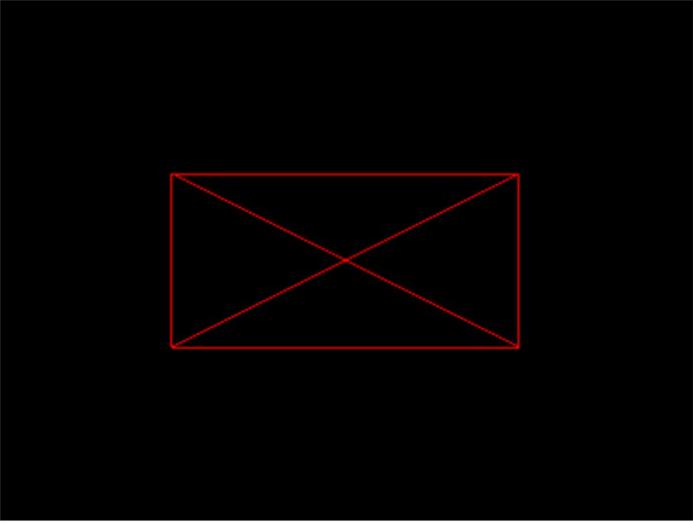
我们看到正交投影的结果是一个正方形，后面的边与前面完全重合了，这也就是正交投影与透视投影的区别所在。

### 2.1.4长宽比例

这里，我们的Canvas宽度是400px，高度是300px，照相机水平方向距离4，垂直方向距离3，因此长宽比例保持不变。为了试验长宽比例变化时的效果，我们将照相机水平方向的距离减小为2：

var camera = new THREE.OrthographicCamera(-1, 1, 1.5, -1.5, 1, 10);

得到的结果是水平方向被拉长了：



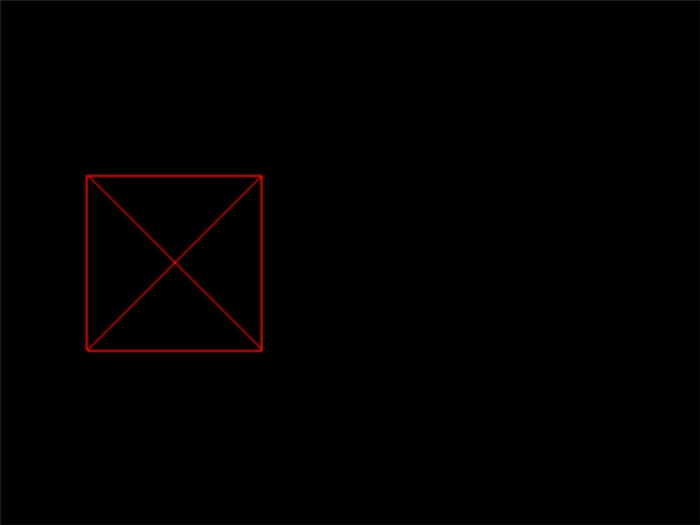
### 2.1.5照相机位置

接下来，我们来看看照相机位置对渲染结果的影响。在之前的例子中，我们将照相机设置在(0, 0, 5)位置，而由于照相机默认是面向z轴负方向放置的，所以能看到在原点处的正方体。现在，如果我们将照相机向右移动1个单位：

var camera = new THREE.OrthographicCamera(-2, 2, 1.5, -1.5, 1, 10);

camera.position.set(1, 0, 5);

得到的效果是物体看上去向左移动了：



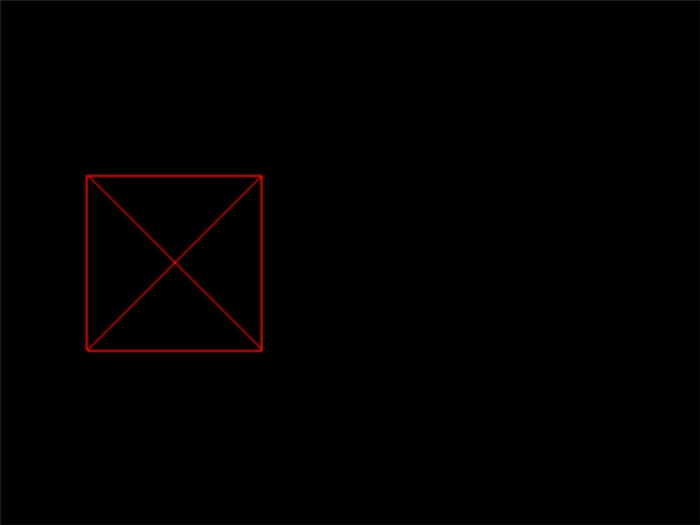
仔细想一下的话，这也不难理解。就好比你人往右站了，看起来物体就相对往左移动了。

那么，正交投影照相机在设置时，是否需要保证left和right是相反数呢？如果不是，那么会产生什么效果呢？下面，我们将原本的参数(-2, 2, 1.5, -1.5, 1, 10)改为(-1, 3, 1.5, -1.5, 1, 10)，即，将视景体设置得更靠右：

var camera = new THREE.OrthographicCamera(-1, 3, 1.5, -1.5, 1, 10);

camera.position.set(0, 0, 5);

得到的结果是：



细心的读者已经发现，这与之前向右移动照相机得到的效果是等价的。

### 2.1.6换个角度看世界

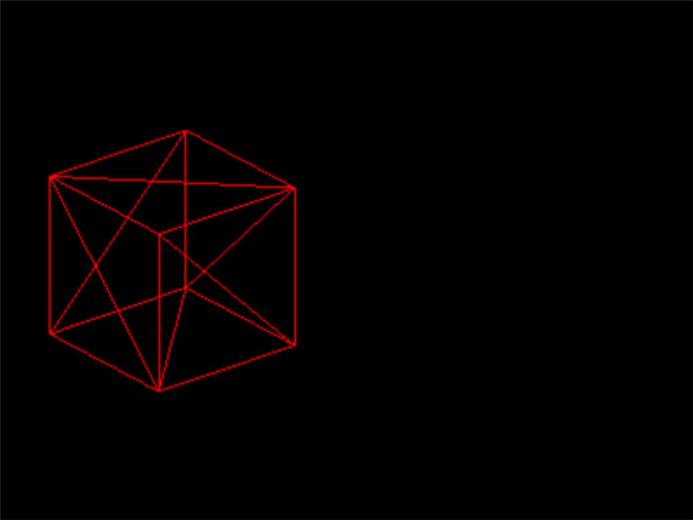
到现在为止，我们使用照相机都是沿z轴负方向观察的，因此看到的都是一个正方形。现在，我们想尝试一下仰望这个正方体。我们已经学会设置照相机的位置，不妨将其设置在(4, -3, 5)处：

camera.position.set(4, -3, 5);

但是现在照相机沿z轴负方向观察的，因此观察不到正方体，只看到一片黑。我们可以通过lookAt函数指定它看着原点方向：

camera.lookAt(new THREE.Vector3(0, 0, 0));

这样我们就能过仰望正方体啦：



不过一定要注意，lookAt函数接受的是一个THREE.Vector3的实例，因此千万别写成camera.lookAt(0, 0, 0)，否则非但不能得到理想的效果，而且不会报错，使你很难找到问题所在。

现在，恭喜你学会设置正交照相机了！虽然它看起来较为简单，但是加入动画、交互等因素后，可以为你的应用程序增色不少的！

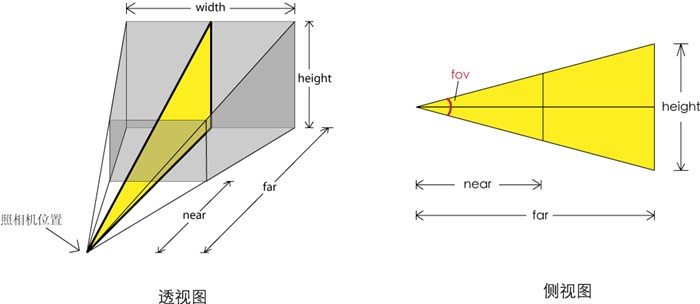
## 2.2透视投影照相机

### 2.2.1参数介绍

透视投影照相机（Perspective Camera）的构造函数是：

THREE.PerspectiveCamera(fov, aspect, near, far)

让我们通过一张透视照相机投影的图来了解这些参数。



透视图中，灰色的部分是视景体，是可能被渲染的物体所在的区域。fov是视景体竖直方向上的张角（是角度制而非弧度制），如侧视图所示。

aspect等于width / height，是照相机水平方向和竖直方向长度的比值，通常设为Canvas的横纵比例。

near和far分别是照相机到视景体最近、最远的距离，均为正值，且far应大于near。

下面，我们从一个最简单的例子学习设置透视投影照相机。

### 2.2.2基本设置

**[例2.2.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter2/2.4.1.html)**

设置透视投影照相机，这里Canvas长400px，宽300px，所以aspect设为400 / 300：

var camera = new THREE.PerspectiveCamera(45, 400 / 300, 1, 10);

camera.position.set(0, 0, 5);

scene.add(camera);

和**[例2.1.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter2/2.3.1.html)**一样，设置一个在原点处的边长为1的正方体：

var cube = new THREE.Mesh(new THREE.CubeGeometry(1, 1, 1),

new THREE.MeshBasicMaterial({

color: 0xff0000,

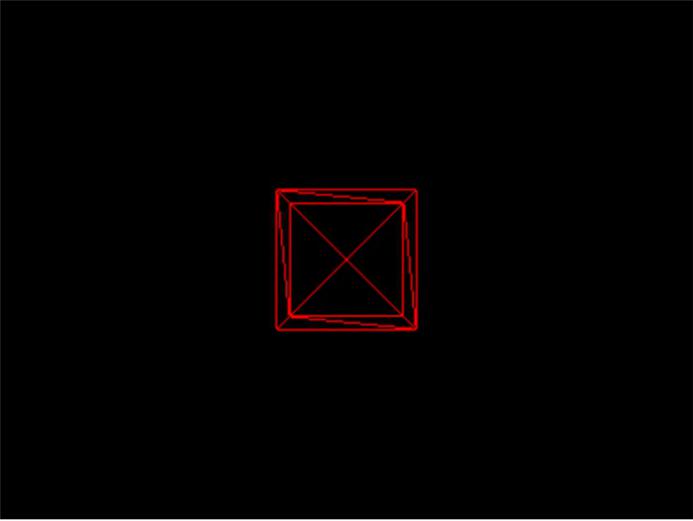
wireframe: true

})

);

scene.add(cube);

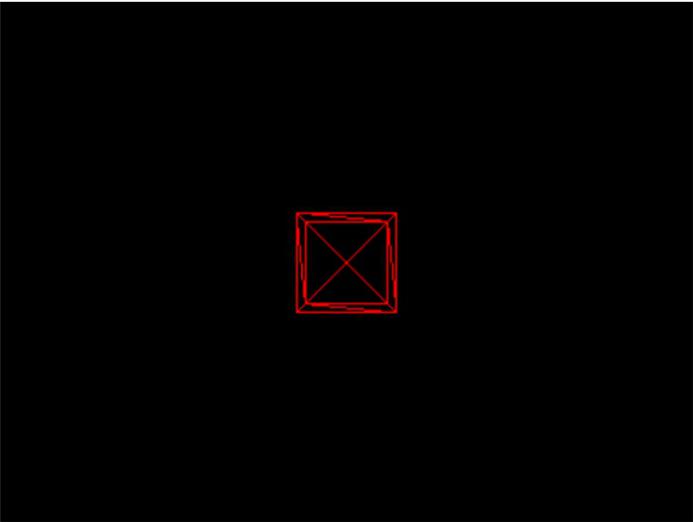
得到的结果是：



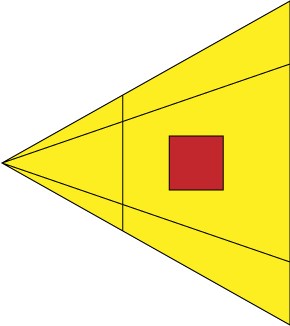
对比**[例2.3.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter2/2.3.1.html)**正方形的效果，透视投影可以看到全部的12条边，而且有近大远小的效果，这也就是与正交投影的区别。

### 2.2.3竖直张角

接下来，我们来看下fov的改变对渲染效果的影响。我们将原来的45改为60，得到这样的效果：



为什么正方体显得更小了呢？我们从下面的侧视图来看，虽然正方体的实际大小并未改变，但是将照相机的竖直张角设置得更大时，视景体变大了，因而正方体相对于整个视景体的大小就变小了，看起来正方形就显得变小了。



注意，改变fov并不会引起画面横竖比例的变化，而改变aspect则会改变横竖比例。

# 三、几何形状

在创建物体时，需要传入两个参数，一个是几何形状（Geometry），另一个是材质（Material）。本章将介绍几何形状的创建，第4章介绍材质，第5章介绍如何使用这两者创建网格。

几何形状（Geometry）最主要的功能是储存了一个物体的顶点信息。WebGL需要程序员指定每个顶点的位置，而在Three.js中，可以通过指定一些特征来创建几何形状，例如使用半径创建一个球体，从而省去程序员一个个指定顶点的工作量。

本节将分别介绍立方体、平面、球体、圆柱体、四面体、八面体等几何形状，以及以三维文字作为几何形状的方法。本节还会介绍通过手动定义顶点位置和面片信息组成几何形状。

## 3.1基本几何形状

### 3.1.1立方体

虽然这一形状的名字叫立方体（CubeGeometry），但它其实是长方体，也就是长宽高可以设置为不同的值。其构造函数是：

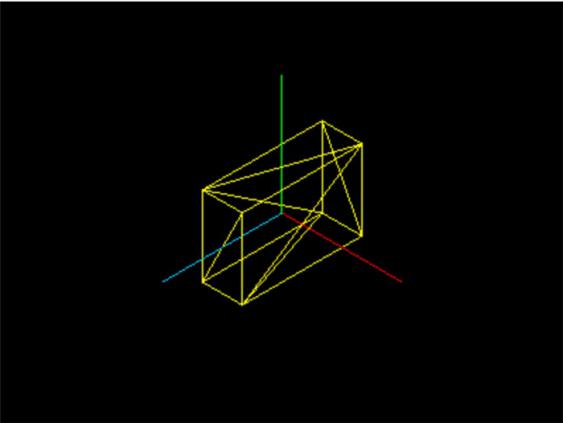
THREE.CubeGeometry(width, height, depth, widthSegments, heightSegments, depthSegments)

这里，width是x方向上的长度；height是y方向上的长度；depth是z方向上的长度；后三个参数分别是在三个方向上的分段数，如widthSegments为3的话，代表x方向上水平分为三份。一般情况下不需要分段的话，可以不设置后三个参数，后三个参数的缺省值为1。其他几何形状中的分段也是类似的，下面不做说明。

#### 3.1.1.1长宽高

创建立方体直观简单，如：new THREE.CubeGeometry(1, 2, 3);可以创建一个x方向长度为1，y方向长度为2，z方向长度为3的立方体。

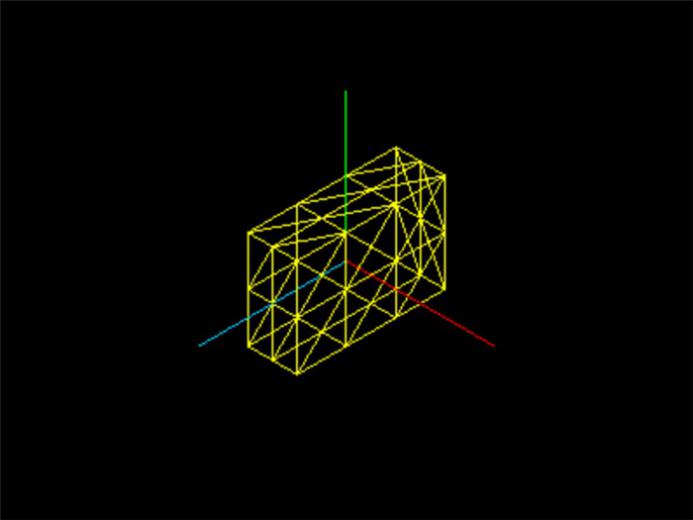
为了更好地表现参数效果，我们在场景中用长度为1的红、绿、蓝线段分别表示x、y、z三个轴。在设置材质，并添加到场景之后（具体方法参见第4章及第5章）的效果是：



物体的默认位置是原点，对于立方体而言，是其几何中心在原点的位置。

#### 3.1.1.2分段

而在设置了分段new THREE.CubeGeometry(1, 2, 3, 2, 2, 3)后，效果如下：



注意这个分段是对六个面进行分段，而不是对立方体的体素分段，因此在立方体的中间是不分段的，只有六个侧面被分段。

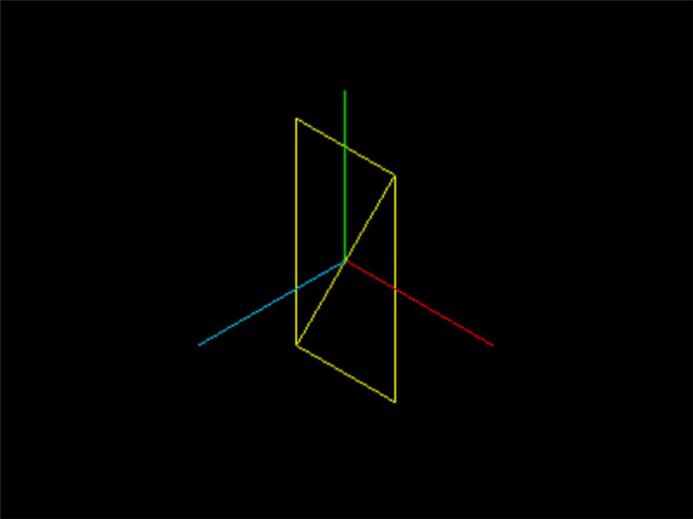
### 3.1.2平面

这里的平面（PlaneGeometry）其实是一个长方形，而不是数学意义上无限大小的平面。其构造函数为：

THREE.PlaneGeometry(width, height, widthSegments, heightSegments)

其中，width是x方向上的长度；height是y方向上的长度；后两个参数同样表示分段。

new THREE.PlaneGeometry(2, 4);创建的平面在x轴和y轴所在平面内，效果如下：



### 3.1.3球体

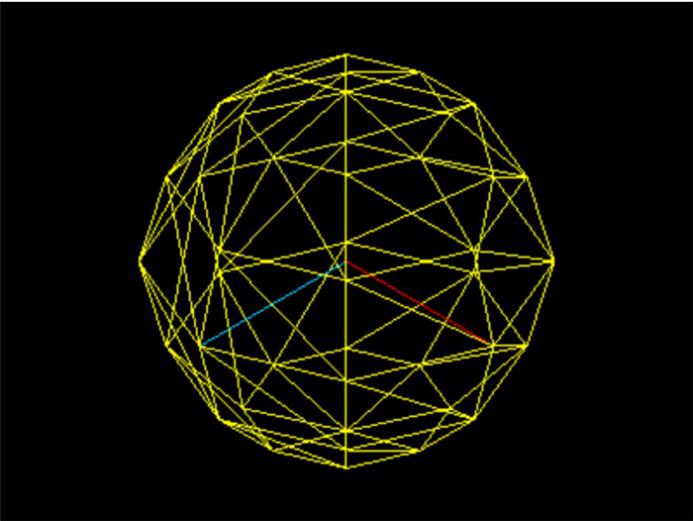
球体（SphereGeometry）的构造函数是：

THREE.SphereGeometry(radius, segmentsWidth, segmentsHeight, phiStart, phiLength, thetaStart, thetaLength)

其中，radius是半径；segmentsWidth表示经度上的切片数；segmentsHeight表示纬度上的切片数；phiStart表示经度开始的弧度；phiLength表示经度跨过的弧度；thetaStart表示纬度开始的弧度；thetaLength表示纬度跨过的弧度。

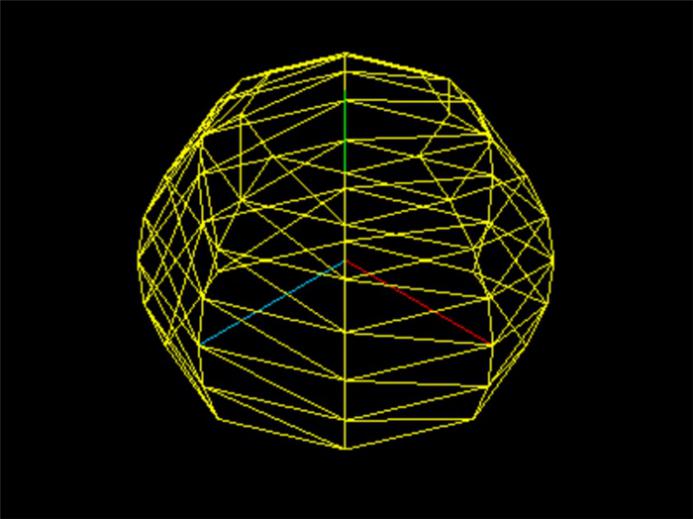
#### 3.1.3.1分段

首先，我们来理解下segmentsWidth和segmentsHeight。使用var sphere = new THREE.SphereGeometry(3, 8, 6)可以创建一个半径为3，经度划分成8份，纬度划分成6份的球体，如下图所示。

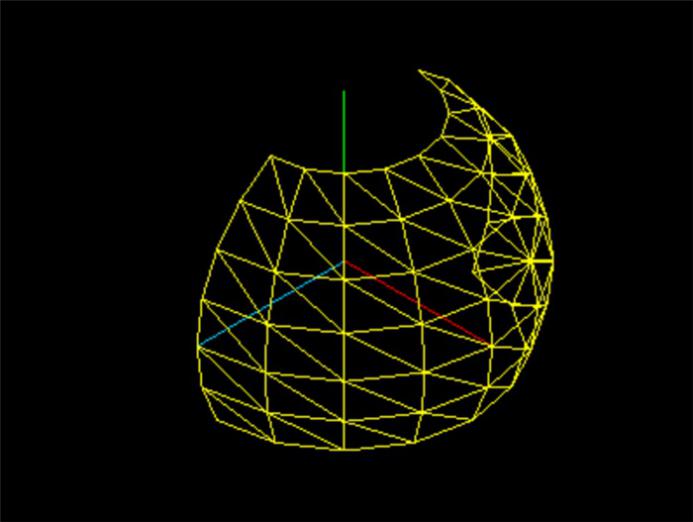


segmentsWidth相当于经度被切成了几瓣，而segmentsHeight相当于纬度被切成了几层。因为在图形底层的实现中，并没有曲线的概念，曲线都是由多个折线近似构成的。对于球体而言，当这两个值较大的时候，形成的多面体就可以近似看做是球体了。

new THREE.SphereGeometry(3, 8, 6)的效果：

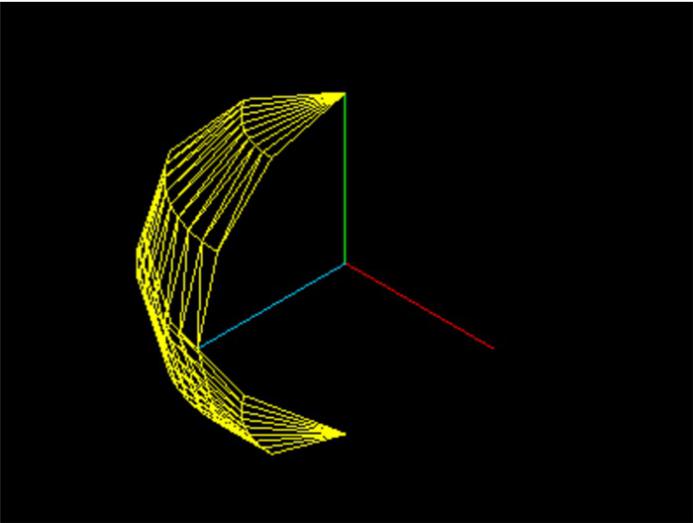


new THREE.SphereGeometry(3, 18, 12)的效果：



#### 3.1.3.2经度弧度

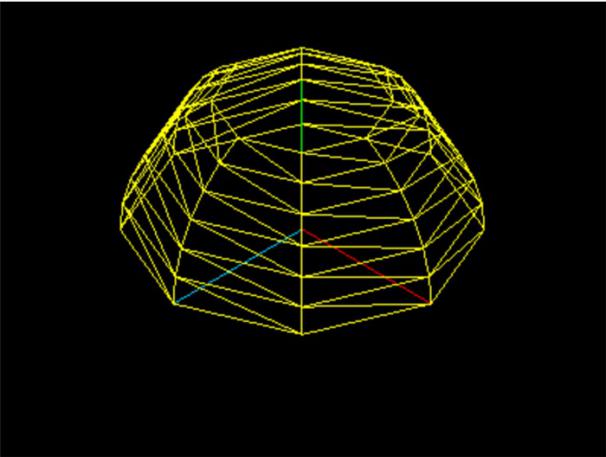
new THREE.SphereGeometry(3, 8, 6, Math.PI / 6, Math.PI / 3)表示起始经度为Math.PI / 6，经度跨度为Math.PI / 3。效果如下：



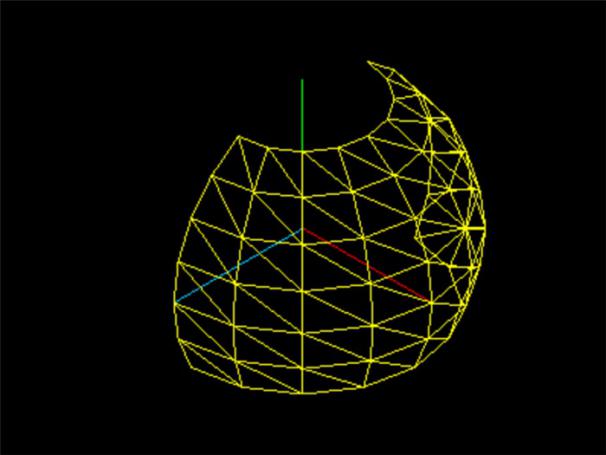
需要注意的是，这里segmentsWidth为8意味着对于经度从Math.PI / 6跨过Math.PI / 3的区域内划分为8块，而不是整个球体的经度划分成8块后再判断在此经度范围内的部分。

#### 3.1.3.3纬度弧度

理解了经度之后，纬度可以同理理解。new THREE.SphereGeometry(3, 8, 6, 0, Math.PI \* 2, Math.PI / 6, Math.PI / 3)意味着纬度从Math.PI / 6跨过Math.PI / 3。效果如下：



我们再来看一个经度纬度都改变了起始位置和跨度的例子：new THREE.SphereGeometry(3, 8, 6, Math.PI / 2, Math.PI, Math.PI / 6, Math.PI / 2)的效果为：

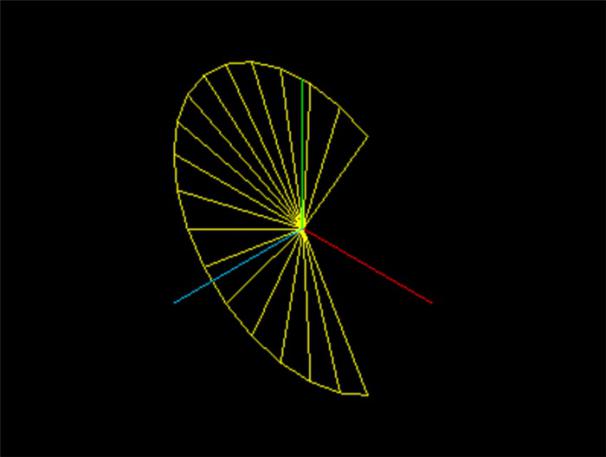


### 3.1.4圆形

圆形（CircleGeometry）可以创建圆形或者扇形，其构造函数是：

THREE.CircleGeometry(radius, segments, thetaStart, thetaLength)

这四个参数都是球体中介绍过的，这里不再赘述，直接来看个例子。new THREE.CircleGeometry(3, 18, Math.PI / 3, Math.PI / 3 \* 4)可以创建一个在x轴和y轴所在平面的三分之二圆的扇形：



### 3.1.5圆柱体

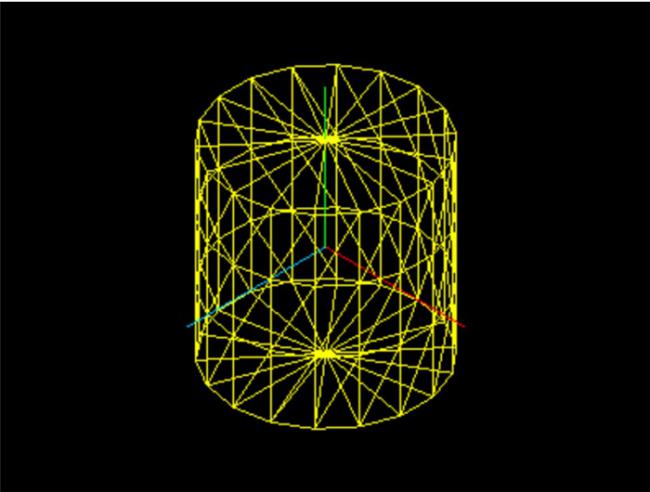
圆柱体（CylinderGeometry）的构造函数是：

THREE.CylinderGeometry(radiusTop, radiusBottom, height, radiusSegments, heightSegments, openEnded)

其中，radiusTop与radiusBottom分别是顶面和底面的半径，由此可知，当这两个参数设置为不同的值时，实际上创建的是一个圆台；height是圆柱体的高度；radiusSegments与heightSegments可类比球体中的分段；openEnded是一个布尔值，表示是否没有顶面和底面，缺省值为false，表示有顶面和底面。

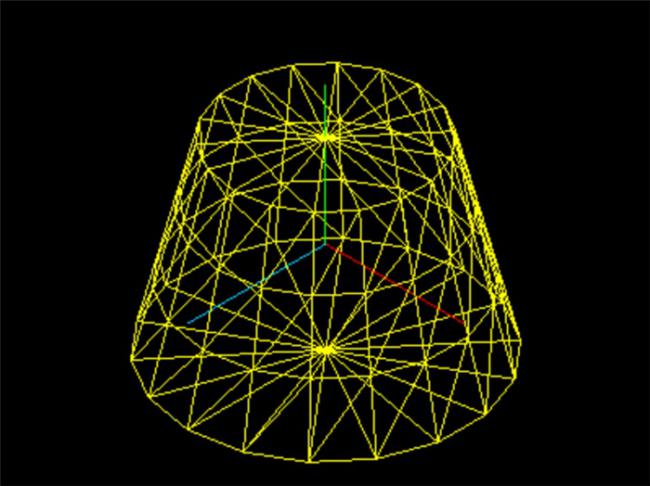
#### 3.1.5.1标准圆柱体

new THREE.CylinderGeometry(2, 2, 4, 18, 3)创建一个顶面与底面半径都为2，高度为4的圆柱体，效果如下：



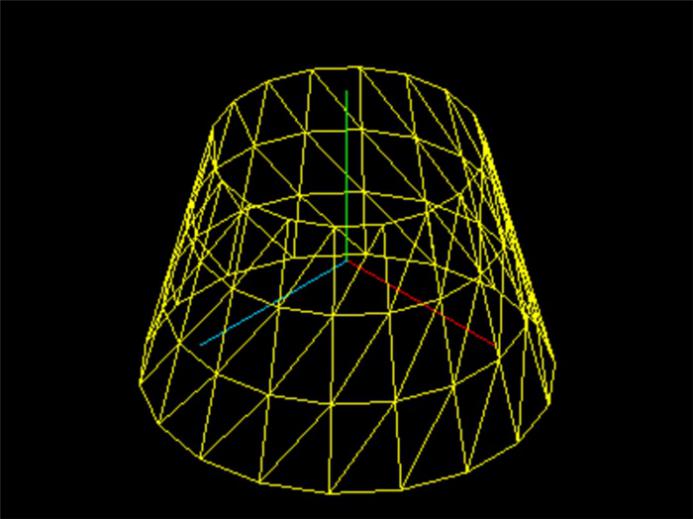
#### 3.1.5.2圆台

将底面半径设为3创建一个圆台：new THREE.CylinderGeometry(2, 3, 4, 18, 3)，效果如下：



#### 3.1.5.3无顶面底面

new THREE.CylinderGeometry(2, 3, 4, 18, 3, true)将创建一个没有顶面与底面的圆台，效果如下：



### 3.1.6正四面体、正八面体、正二十面体

正四面体（TetrahedronGeometry）、正八面体（OctahedronGeometry）、正二十面体（IcosahedronGeometry）的构造函数较为类似，分别为：

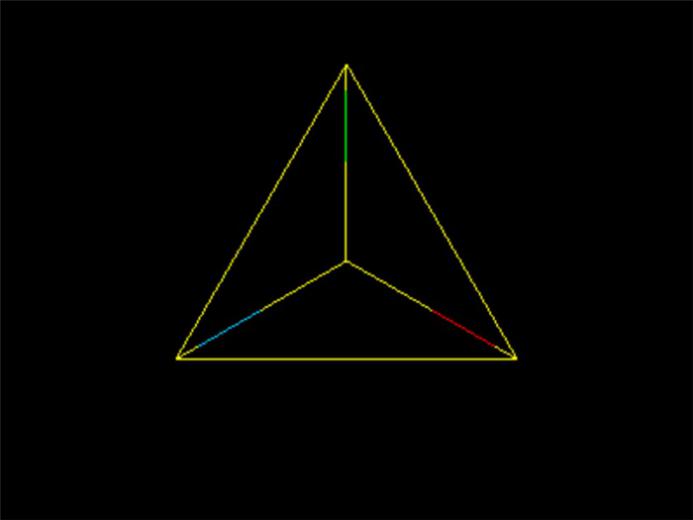
THREE.TetrahedronGeometry(radius, detail)

THREE.OctahedronGeometry(radius, detail)

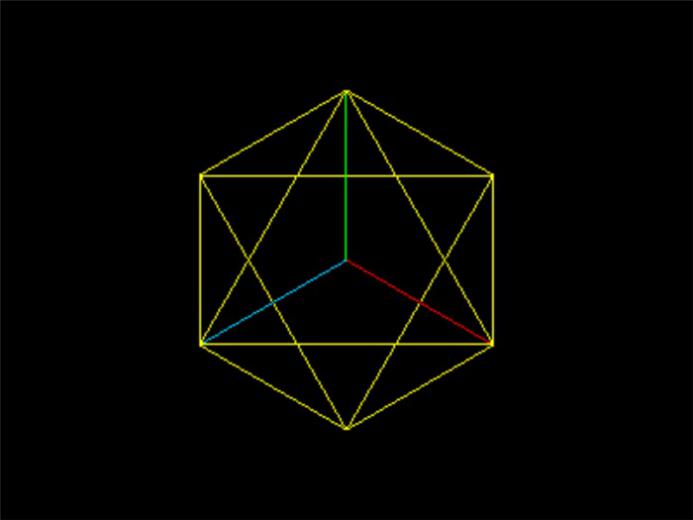
THREE.IcosahedronGeometry(radius, detail)

其中，radius是半径；detail是细节层次（Level of Detail）的层数，对于大面片数模型，可以控制在视角靠近物体时，显示面片数多的精细模型，而在离物体较远时，显示面片数较少的粗略模型。这里我们不对detail多作展开，一般可以对这个值缺省。

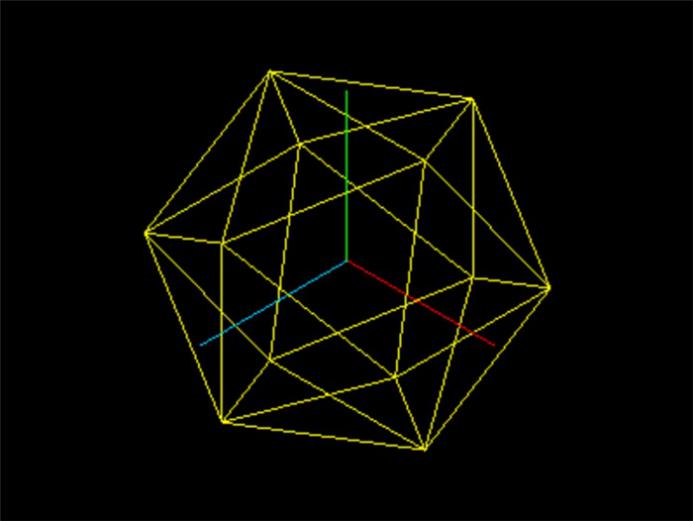
new THREE.TetrahedronGeometry(3)创建一个半径为3的正四面体：



new THREE.OctahedronGeometry(3)创建一个半径为3的正八面体：



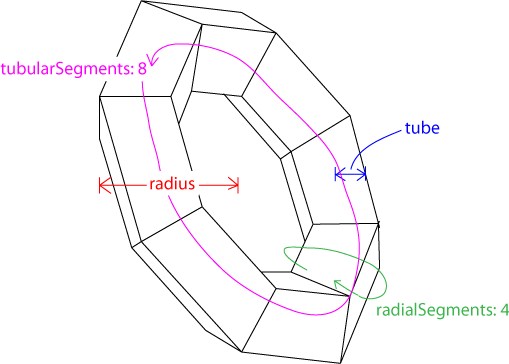
new THREE.IcosahedronGeometry(3)创建一个半径为3的正二十面体：



### 3.1.7圆环面

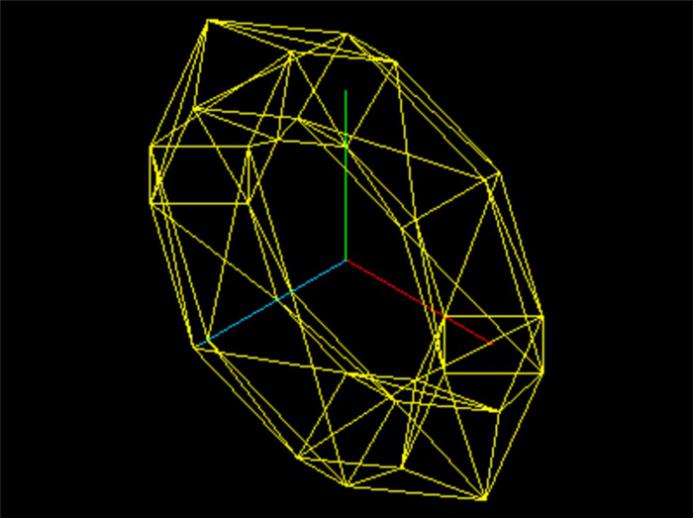
圆环面（TorusGeometry）就是甜甜圈的形状，其构造函数是：

THREE.TorusGeometry(radius, tube, radialSegments, tubularSegments, arc)

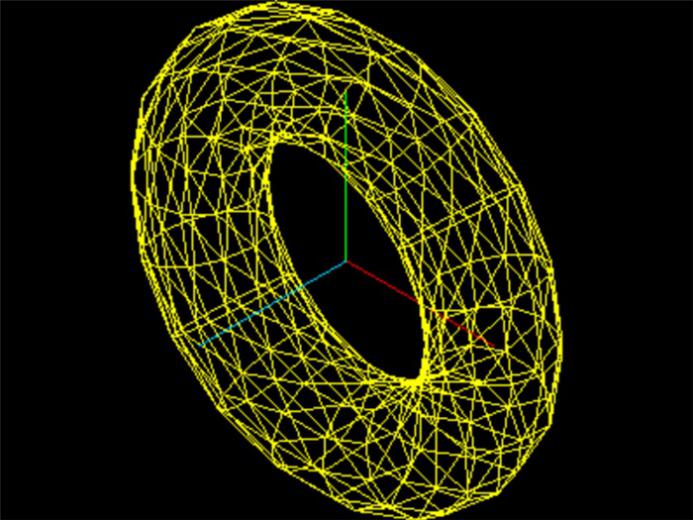


其中，radius是圆环半径；tube是管道半径；radialSegments与tubularSegments分别是两个分段数，详见上图；arc是圆环面的弧度，缺省值为Math.PI \* 2。

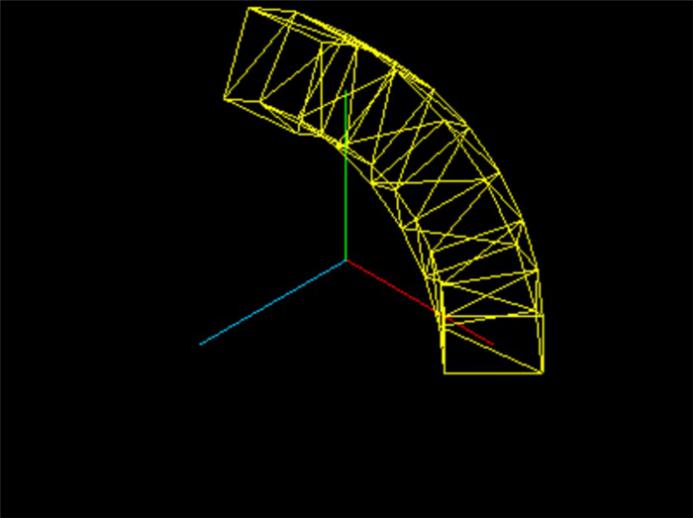
new THREE.TorusGeometry(3, 1, 4, 8)创建一个粗糙的圆环面：



new THREE.TorusGeometry(3, 1, 12, 18)创建一个较为精细的圆环面：



new THREE.TorusGeometry(3, 1, 4, 8, Math.PI / 3 \* 2)创建部分圆环面：



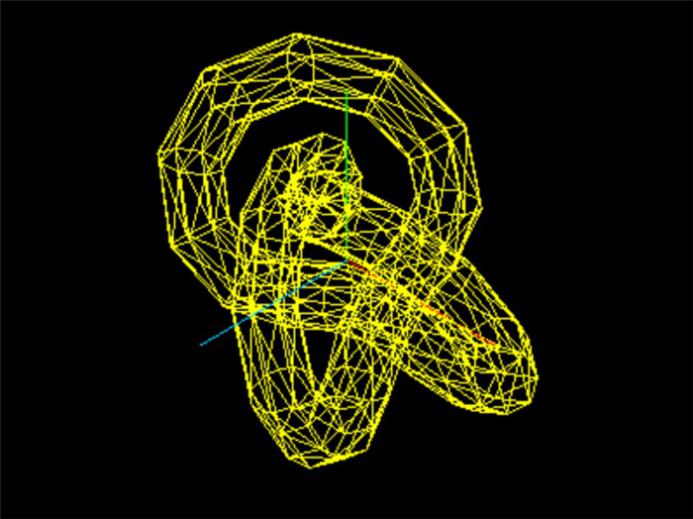
#### 2.5.8圆环结

如果说圆环面是甜甜圈，那么圆环结（TorusKnotGeometry）就是打了结的甜甜圈，其构造参数为：

THREE.TorusKnotGeometry(radius, tube, radialSegments, tubularSegments, p, q, heightScale)

前四个参数在圆环面中已经有所介绍，p和q是控制其样式的参数，一般可以缺省，如果需要详细了解，请学习[圆环结的相关知识](http://en.wikipedia.org/wiki/Torus_knot" \t "http://www.ituring.com.cn/article/_blank)；heightScale是在z轴方向上的缩放。

new THREE.TorusKnotGeometry(2, 0.5, 32, 8)的效果：



## 3.2文字形状

文字形状（TextGeometry）可以用来创建三维的文字形状。

### 3.2.1下载使用

使用文字形状需要下载和引用额外的字体库，具体参见[Three.js GitHub说明](https://github.com/mrdoob/three.js/tree/master/examples/fonts)。

这里，我们以 helvetiker字体为例。我们在[Three.js GitHub master/examples/fonts](https://github.com/mrdoob/three.js/tree/master/examples/fonts)目录下，下载helvetiker\_regular.typeface.json文件放在你的目录下，然后用以下方法加载：

var loader = new THREE.FontLoader();

loader.load('../lib/helvetiker\_regular.typeface.json', function(font) {

var mesh = new THREE.Mesh(new THREE.TextGeometry('Hello', {

font: font,

size: 1,

height: 1

}), material);

scene.add(mesh);

// render

renderer.render(scene, camera);

});

### 3.2.2参数介绍

创建文字形状的流程和之前介绍的基本几何形状是类似的，其构造函数是：

THREE.TextGeometry(text, parameters)

其中，text是文字字符串，parameters是以下参数组成的对象：

size：字号大小，一般为大写字母的高度

height：文字的厚度

curveSegments：弧线分段数，使得文字的曲线更加光滑

font：字体，默认是'helvetiker'，需对应引用的字体文件

weight：值为'normal'或'bold'，表示是否加粗

style：值为'normal'或'italics'，表示是否斜体

bevelEnabled：布尔值，是否使用倒角，意为在边缘处斜切

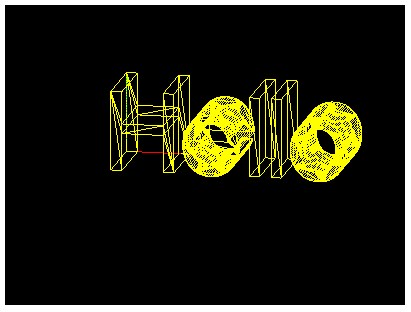
bevelThickness：倒角厚度

bevelSize：倒角宽度

### 3.2.2实例说明

**[例3.2.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter3/3.2.1.html)**

创建一个三维文字：new THREE.TextGeometry('Hello', {size: 1, height: 1})，其效果为：



改变材质和光照之后就能达到这样的效果：



## 3.3自定义形状

对于Three.js没有提供的形状，可以提供自定义形状来创建。

由于自定义形状需要手动指定每个顶点位置，以及顶点连接情况，如果该形状非常复杂，程序员的计算量就会比较大。在这种情况下，建议在*3ds Max*之类的建模软件中创建模型，然后使用Three.js导入到场景中，这样会更高效方便。

自定义形状使用的是Geometry类，它是其他如CubeGeometry、SphereGeometry等几何形状的父类，其构造函数是：

THREE.Geometry()

**[例3.3.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter3/3.3.1.html)**

我们以创建一个梯台为例，首先，初始化一个几何形状，然后设置顶点位置以及顶点连接情况。

// 初始化几何形状

var geometry = new THREE.Geometry();

// 设置顶点位置

// 顶部4顶点

geometry.vertices.push(new THREE.Vector3(-1, 2, -1));

geometry.vertices.push(new THREE.Vector3(1, 2, -1));

geometry.vertices.push(new THREE.Vector3(1, 2, 1));

geometry.vertices.push(new THREE.Vector3(-1, 2, 1));

// 底部4顶点

geometry.vertices.push(new THREE.Vector3(-2, 0, -2));

geometry.vertices.push(new THREE.Vector3(2, 0, -2));

geometry.vertices.push(new THREE.Vector3(2, 0, 2));

geometry.vertices.push(new THREE.Vector3(-2, 0, 2));

// 设置顶点连接情况

// 顶面

geometry.faces.push(new THREE.Face3(0, 1, 3));

geometry.faces.push(new THREE.Face3(1, 2, 3));

// 底面

geometry.faces.push(new THREE.Face3(4, 5, 6));

geometry.faces.push(new THREE.Face3(5, 6, 7));

// 四个侧面

geometry.faces.push(new THREE.Face3(1, 5, 6));

geometry.faces.push(new THREE.Face3(6, 2, 1));

geometry.faces.push(new THREE.Face3(2, 6, 7));

geometry.faces.push(new THREE.Face3(7, 3, 2));

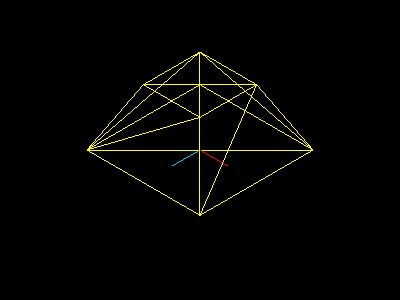
geometry.faces.push(new THREE.Face3(3, 7, 0));

geometry.faces.push(new THREE.Face3(7, 4, 0));

geometry.faces.push(new THREE.Face3(0, 4, 5));

geometry.faces.push(new THREE.Face3(0, 5, 1));

效果是：



需要注意的是，new THREE.Vector3(-1, 2, -1)创建一个矢量，作为顶点位置追加到geometry.vertices数组中。

而由new THREE.Face3(0, 1, 3)创建一个三个顶点组成的面片，追加到geometry.faces数组中。三个参数分别是四个顶点在geometry.vertices中的序号。

在之前版本的Three.js中，可以使用Face4创建四边形，如今已经弃用。由于四个点未必共面，所以使用三角形永远是最安全的方法。

# 四、材质

材质（Material）是独立于物体顶点信息之外的与渲染效果相关的属性。通过设置材质可以改变物体的颜色、纹理贴图、光照模式等。

本章将介绍基本材质、两种基于光照模型的材质，以及使用法向量作为材质。除此之外，本章还将介绍如何使用图像作为材质。

## 4.1基本材质BasicMaterial

使用基本材质（BasicMaterial）的物体，渲染后物体的颜色始终为该材质的颜色，而不会由于光照产生明暗、阴影效果。如果没有指定材质的颜色，则颜色是随机的。其构造函数是：

THREE.MeshBasicMaterial(opt)

其中，opt可以缺省，或者为包含各属性的值。如新建一个不透明度为0.75的黄色材质：

**[例4.1.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter4/4.1.1.html)**

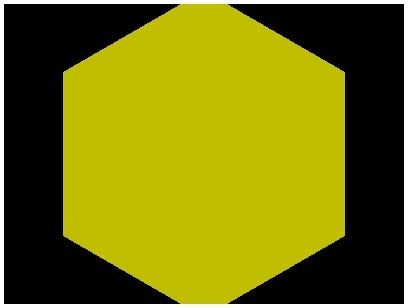
new THREE.MeshBasicMaterial({

color: 0xffff00,

opacity: 0.75}

);

将其应用于一个正方体（方法参见3.1节），效果为：



接下来，我们介绍几个较为常用的属性。

visible：是否可见，默认为true

side：渲染面片正面或是反面，默认为正面THREE.FrontSide，可设置为反面THREE.BackSide，或双面THREE.DoubleSide

wireframe：是否渲染线而非面，默认为false

color：十六进制RGB颜色，如红色表示为0xff0000

map：使用纹理贴图，详见4.5节

对于基本材质，即使改变场景中的光源，使用该材质的物体也始终为颜色处处相同的效果。当然，这不是很具有真实感，因此，接下来我们将介绍更为真实的光照模型：Lambert光照模型以及Phong光照模型。

## 4.2Lambert材质

Lambert材质（MeshLambertMaterial）是符合Lambert光照模型的材质。Lambert光照模型的主要特点是只考虑漫反射而不考虑镜面反射的效果，因而对于金属、镜子等需要镜面反射效果的物体就不适应，对于其他大部分物体的漫反射效果都是适用的。

其光照模型公式为：

Idiffuse = Kd \* Id \* cos(theta)

其中，Idiffuse是漫反射光强，Kd是物体表面的漫反射属性，Id是光强，theta是光的入射角弧度。

当然，对于使用Three.js的Lambert材质，不需要了解以上公式就可以直接使用。创建一个黄色的Lambert材质的方法为：

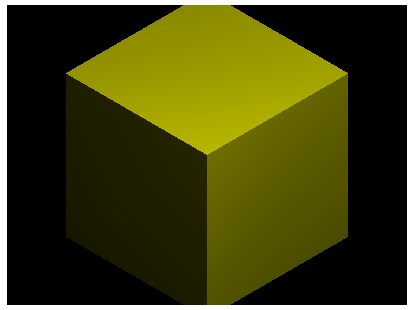
**[例4.2.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter4/4.2.1.html)**

new THREE.MeshLambertMaterial({

color: 0xffff00

})

在使用了光照之后（具体方法参见第8章），得到这样的效果：



color是用来表现材质对散射光的反射能力，也是最常用来设置材质颜色的属性。除此之外，还可以用ambient和emissive控制材质的颜色。

ambient表示对环境光的反射能力，只有当设置了AmbientLight后，该值才是有效的，材质对环境光的反射能力与环境光强相乘后得到材质实际表现的颜色。

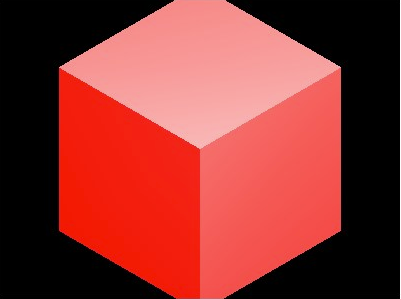
emissive是材质的自发光颜色，可以用来表现光源的颜色。单独使用红色的自发光：

new THREE.MeshLambertMaterial({

emissive: 0xff0000

})

效果为：



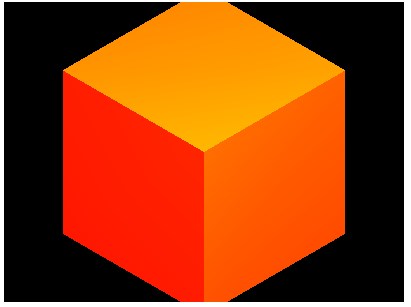
如果同时使用红色的自发光与黄色的散射光：

new THREE.MeshLambertMaterial({

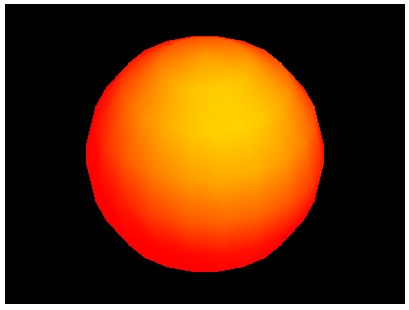
color: 0xffff00,

emissive: 0xff0000})

效果为：



如果将同样的材质用于球体，看起来像不像滚烫的太阳呢？



## 4.3Phong材质

Phong材质（MeshPhongMaterial）是符合Phong光照模型的材质。和Lambert不同的是，Phong模型考虑了镜面反射的效果，因此对于金属、镜面的表现尤为适合。

漫反射部分和Lambert光照模型是相同的，镜面反射部分的模型为：

Ispecular = Ks \* Is \* (cos(alpha)) ^ n

其中，Ispecular是镜面反射的光强，Ks是材质表面镜面反射系数，Is是光源强度，alpha是反射光与视线的夹角，n是高光指数，越大则高光光斑越小。

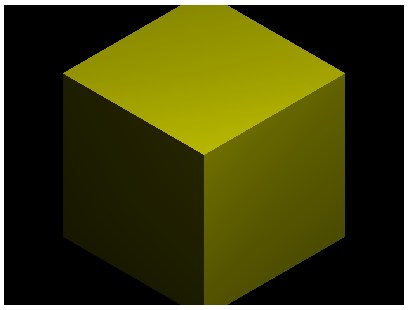
由于漫反射部分与Lambert模型是一致的，因此，如果不指定镜面反射系数，而只设定漫反射，其效果与Lambert是相同的：

**[例4.3.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter4/4.3.1.html)**

new THREE.MeshPhongMaterial({

color: 0xffff00

});

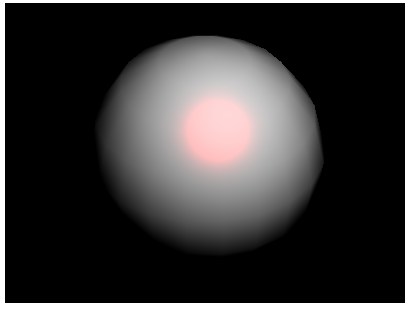


同样地，可以指定emissive和ambient值，这里不再说明。下面就specular值指定镜面反射系数作说明。首先，我们只使用镜面反射，将高光设为红色，应用于一个球体：

var material = new THREE.MeshPhongMaterial({

specular: 0xff0000});var sphere = new THREE.Mesh(new THREE.SphereGeometry(3, 20, 8), material);

效果为：



可以通过shininess属性控制光照模型中的n值，当shininess值越大时，高光的光斑越小，默认值为30。我们将其设置为1000时：

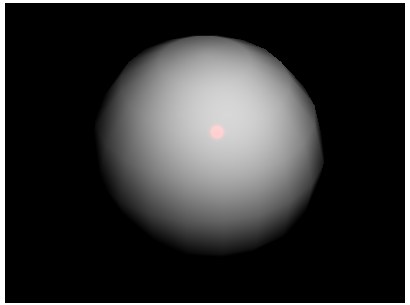
new THREE.MeshPhongMaterial({

specular: 0xff0000,

shininess: 1000

});

效果为：



使用黄色的镜面光，红色的散射光：

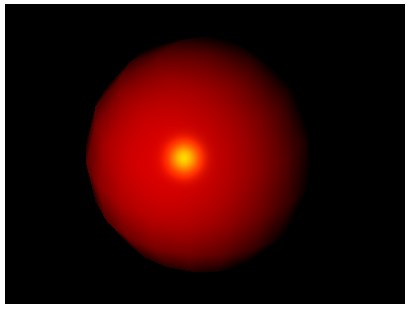
material = new THREE.MeshPhongMaterial({

color: 0xff0000,

specular: 0xffff00,

shininess: 100

});



看起来是不是像个桌球了呢？

## 4.4法向材质

法向材质可以将材质的颜色设置为其法向量的方向，有时候对于调试很有帮助。

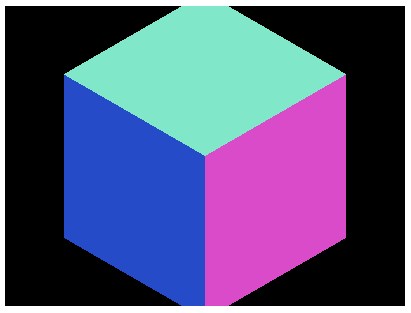
法向材质的设定很简单，甚至不用设置任何参数：

new THREE.MeshNormalMaterial()

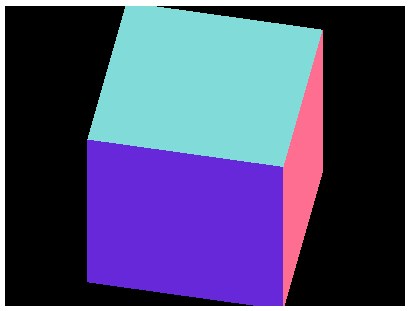
材质的颜色与照相机与该物体的角度相关，下面我们只改变照相机位置，观察两个角度的颜色变化：

**[例4.4.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter4/4.4.1.html)**

camera.position.set(5, 25, 25);的效果：



camera.position.set(25, 25, 25);的效果：



我们观察的是同样的三个面，但是由于观察的角度不同，物体的颜色就不同了。因此，在调试时，要知道物体的法向量，使用法向材质就很有效。

## 4.5材质的纹理贴图

在此之前，我们使用的材质都是单一颜色的，有时候，我们却希望使用图像作为材质。这时候，就需要导入图像作为纹理贴图，并添加到相应的材质中。下面，我们介绍具体的做法。

### 4.5.1单张图像应用于长方体

**[例4.5.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter4/4.5.1.html)**

首先，我们选择一张长宽均为128像素的图像：



将其导入纹理中：

var texture = THREE.ImageUtils.loadTexture('../img/0.png');

然后，将材质的map属性设置为texture：

var material = new THREE.MeshLambertMaterial({

map: texture

});

这样就完成了将图片应用于材质的基本步骤。但是由于现在我们还没使用动画，画面只被渲染了一次，而在导入纹理之前，已经完成了这次渲染，因此看到的只是一片黑。所以，如果没有重绘函数（将在下一章介绍），就需要在完成导入纹理的步骤后，重新绘制画面，这是在回调函数中实现的：

var texture = THREE.ImageUtils.loadTexture('../img/0.png', {}, function() {

renderer.render(scene, camera);

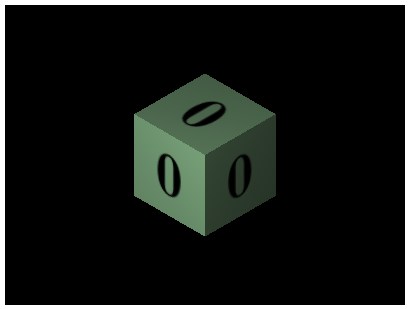
});

var material = new THREE.MeshLambertMaterial({

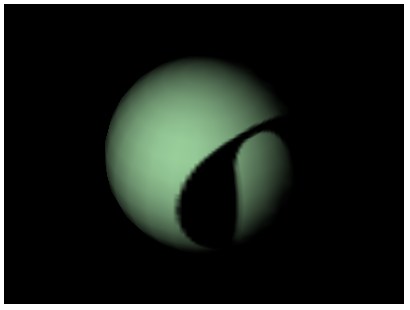
map: texture

});

现在，就能看到这样的效果了：



类似地，如果将其应用于球体，将会把整个球体应用该图像：



### 4.5.2六张图像应用于长方体

**[例4.5.2](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter4/4.5.2.html)**

有时候，我们希望长方体的六面各种的贴图都不同。因此，我们首先准备了六张颜色各异的图像，分别写了数字0到5。然后，分别导入图像到六个纹理，并设置到六个材质中：

var materials = [];for (var i = 0; i < 6; ++i) {

materials.push(new THREE.MeshBasicMaterial({

map: THREE.ImageUtils.loadTexture('../img/' + i + '.png',

{}, function() {

renderer.render(scene, camera);

}),

overdraw: true

}));

}

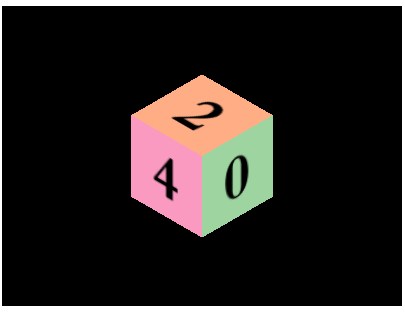
var cube = new THREE.Mesh(new THREE.CubeGeometry(5, 5, 5),

new THREE.MeshFaceMaterial(materials)

);

scene.add(cube);

效果为：



### 4.5.3棋盘格

现在，我们有一个黑白相间的图像：



我们希望用它填满一个屏幕。按照之前的做法依法炮制：

**[例4.5.3](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter4/4.5.3.html)**

var texture = THREE.ImageUtils.loadTexture('../img/chess.png', {},

function() {

renderer.render(scene, camera);

});

效果是：



可是，棋盘格是8横8纵64个小方格组成的，那应该怎么办呢？

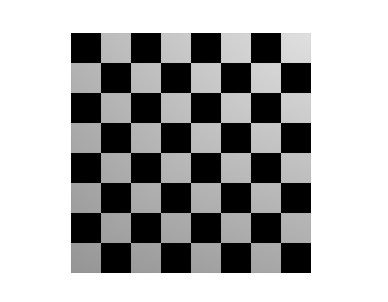
首先，我们需要指定重复方式为两个方向（wrapS和wrapT）都重复：

texture.wrapS = texture.wrapT = THREE.RepeatWrapping;

然后，设置两个方向上都重复4次，由于我们的图像本来是有2行2列，所以重复4次即为8行8列：

texture.repeat.set(4, 4);

最终就得到了棋盘格：



# 五、网格

在学习了几何形状和材质之后，我们就能使用他们来创建物体了。最常用的一种物体就是网格（Mesh），网格是由顶点、边、面等组成的物体；其他物体包括线段（Line）、骨骼（Bone）、粒子系统（ParticleSystem）等。创建物体需要指定几何形状和材质，其中，几何形状决定了物体的顶点位置等信息，材质决定了物体的颜色、纹理等信息。

本章将介绍创建较为常用的物体：网格，然后介绍如何修改物体的属性。

## 5.1创建网格

在上两节中，我们学习了如何创建几何形状与材质，而网格的创建非常简单，只要把几何形状与材质传入其构造函数。最常用的物体是网格（Mesh），它代表包含点、线、面的几何体，其构造函数是：

Mesh(geometry, material)

下面，让我们通过一个具体的例子了解如何创建网格：

**[例5.1.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter5/5.1.1.html)**

var material = new THREE.MeshLambertMaterial({

color: 0xffff00}

);

var geometry = new THREE.CubeGeometry(1, 2, 3);

var mesh = new THREE.Mesh(geometry, material);

scene.add(mesh);

如果material和geometry之后不会复用的话，也可以合在一起写为：

var mesh = new THREE.Mesh(new THREE.CubeGeometry(1, 2, 3),

new THREE.MeshLambertMaterial({

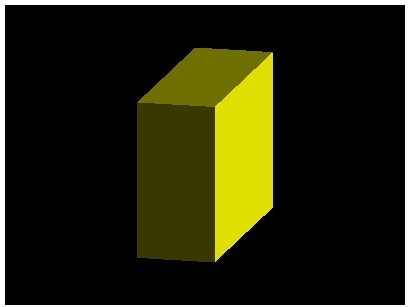
color: 0xffff00

})

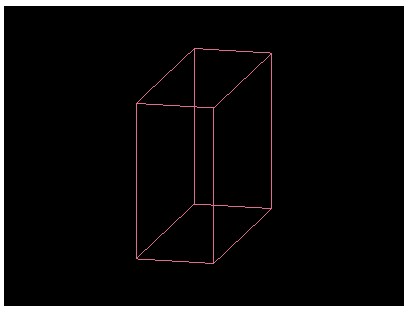
);

scene.add(mesh);

添加光照后，得到的效果为：



如果不指定material，则每次会随机分配一种wireframe为true的材质，每次刷新页面后的颜色是不同的，一种可能的效果是：



## 5.2修改属性

### 5.2.1材质

除了在构造函数中指定材质，在网格被创建后，也能对材质进行修改：

**[例5.2.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter5/5.2.1.html)**

var material = new THREE.MeshLambertMaterial({

color: 0xffff00

});

var geometry = new THREE.CubeGeometry(1, 2, 3);

var mesh = new THREE.Mesh(geometry, material);

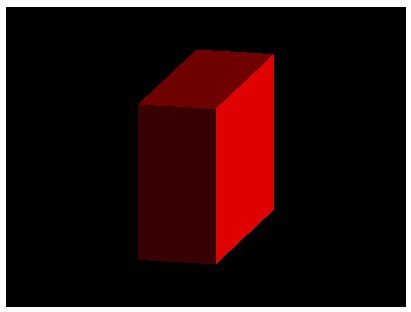
scene.add(mesh);

mesh.material = new THREE.MeshLambertMaterial({

color: 0xff0000

});

最终显示的颜色是红色：



### 5.2.2位置、缩放、旋转

位置、缩放、旋转是物体三个常用属性。由于THREE.Mesh基础自THREE.Object3D，因此包含scale、rotation、position三个属性。它们都是THREE.Vector3实例，因此修改其值的方法是相同的，这里以位置为例。

**[例5.2.2](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter5/5.2.2.html)**

THREE.Vector3有x、y、z三个属性，如果只设置其中一个属性，则可以用以下方法：

mesh.position.z = 1;

如果需要同时设置多个属性，可以使用以下两种方法：

mesh.position.set(1.5, -0.5, 0);

或

mesh.position = new THREE.Vector3(1.5, -0.5, 0);

缩放对应的属性是scale，旋转对应的属性是rotation，具体方法与上例相同，分别表示沿x、y、z三轴缩放或旋转。

# 六、动画

在本章之前，所有画面都是静止的，本章将介绍如果使用Three.js进行动态画面的渲染。此外，将会介绍一个Three.js作者写的另外一个库Stats.js，用来观测每秒帧数（FPS）。

## 6.1实现动画效果

### 6.1.1动画原理

在这里，我们将动态画面简称为动画（animation）。正如动画片的原理一样，动画的本质是利用了人眼的视觉暂留特性，快速地变换画面，从而产生物体在运动的假象。而对于Three.js程序而言，动画的实现也是通过在每秒中多次重绘画面实现的。

为了衡量画面切换速度，引入了每秒帧数FPS（Frames Per Second）的概念，是指每秒画面重绘的次数。FPS越大，则动画效果越平滑，当FPS小于20时，一般就能明显感受到画面的卡滞现象。

那么FPS是不是越大越好呢？其实也未必。当FPS足够大（比如达到60），再增加帧数人眼也不会感受到明显的变化，反而相应地就要消耗更多资源（比如电影的胶片就需要更长了，或是电脑刷新画面需要消耗计算资源等等）。因此，选择一个适中的FPS即可。

NTSC标准的电视FPS是30，PAL标准的电视FPS是25，电影的FPS标准为24。而对于Three.js动画而言，一般FPS在30到60之间都是可取的。

### 6.1.2 setInterval方法

如果要设置特定的FPS（虽然严格来说，即使使用这种方法，[JavaScript也不能保证帧数精确性](http://ejohn.org/blog/how-javascript-timers-work/" \t "http://www.ituring.com.cn/article/_blank)），可以使用JavaScript DOM定义的方法：

setInterval(func, msec)

其中，func是每过msec毫秒执行的函数，如果将func定义为重绘画面的函数，就能实现动画效果。setInterval函数返回一个id，如果需要停止重绘，需要使用clearInterval方法，并传入该id，具体的做法为：

**[例6.1.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter6/6.1.1.html)**

首先，在init函数中定义每20毫秒执行draw函数的setInterval，返回值记录在全局变量id中：

id = setInterval(draw, 20);

在draw函数中，我们首先设定在每帧中的变化（毕竟，如果每帧都是相同的，即使重绘再多次，还是不会有动画的效果），这里我们让场景中的长方体绕y轴转动。然后，执行渲染：

function draw() {

mesh.rotation.y = (mesh.rotation.y + 0.01) % (Math.PI \* 2);

renderer.render(scene, camera);

}

这样，每20毫秒就会调用一次draw函数，改变长方体的旋转值，然后进行重绘。最终得到的效果就是FPS为50的旋转长方体。

我们在HTML中添加一个按钮，按下后停止动画：

<button id="stopBtn" onclick="stop()">Stop</button>

对应的stop函数为：

function stop() {

if (id !== null) {

clearInterval(id);

id = null;

}

}

### 6.1.3 requestAnimationFrame方法

大多数时候，我们并不在意多久重绘一次，这时候就适合用[requestAnimationFrame](https://developer.mozilla.org/en-US/docs/Web/API/window.requestAnimationFrame)方法了。它告诉浏览器在合适的时候调用指定函数，通常可能达到60FPS。

**[例6.1.2](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter6/6.1.2.html)**

requestAnimationFrame同样有对应的cancelAnimationFrame取消动画：

function stop() {

if (id !== null) {

cancelAnimationFrame(id);

id = null;

}

}

和setInterval不同的是，由于requestAnimationFrame只请求一帧画面，因此，除了在init函数中需要调用，在被其调用的函数中需要再次调用requestAnimationFrame：

function draw() {

mesh.rotation.y = (mesh.rotation.y + 0.01) % (Math.PI \* 2);

renderer.render(scene, camera);

id = requestAnimationFrame(draw);

}

因为requestAnimationFrame较为“年轻”，因而一些老的浏览器使用的是试验期的名字：mozRequestAnimationFrame、webkitRequestAnimationFrame、msRequestAnimationFrame，为了支持这些浏览器，我们最好在调用之前，先判断是否定义了requestAnimationFrame以及上述函数：

var requestAnimationFrame = window.requestAnimationFrame

|| window.mozRequestAnimationFrame

|| window.webkitRequestAnimationFrame

|| window.msRequestAnimationFrame;

window.requestAnimationFrame = requestAnimationFrame;

### 6.1.4如何取舍

setInterval方法与requestAnimationFrame方法的区别较为微妙。一方面，最明显的差别表现在setInterval可以手动设定FPS，而requestAnimationFrame则会自动设定FPS；但另一方面，即使是setInterval也不能保证按照给定的FPS执行，在浏览器处理繁忙时，很可能低于设定值。当浏览器达不到设定的调用周期时，requestAnimationFrame采用跳过某些帧的方式来表现动画，虽然会有卡滞的效果但是整体速度不会拖慢，而setInterval会因此使整个程序放慢运行，但是每一帧都会绘制出来；

总而言之，requestAnimationFrame适用于对于时间较为敏感的环境（但是动画逻辑更加复杂），而setInterval则可在保证程序的运算不至于导致延迟的情况下提供更加简洁的逻辑（无需自行处理时间）。

## 6.2使用start.js记录FDS

[stat.js](https://github.com/mrdoob/stats.js)是Three.js的作者[Mr. Doob](http://mrdoob.com/" \t "http://www.ituring.com.cn/article/_blank)的另一个有用的JavaScript库。很多情况下，我们希望知道实时的FPS信息，从而更好地监测动画效果。这时候，stat.js就能提供一个很好的帮助，它占据屏幕中的一小块位置（如左上角），效果为：IMG_256，单击后显示每帧渲染时间：IMG_257。

首先，我们需要下载stat.js文件，可以在<https://github.com/mrdoob/stats.js/blob/master/build/stats.min.js>找到。下载后，将其放在项目文件夹下，然后在HTML中引用：

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

在页面初始化的时候，对其初始化并将其添加至屏幕一角。这里，我们以右上角为例：

**[例6.2.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter6/6.2.1.html)**，**[例6.2.2](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter6/6.2.2.html)**

var stat = null;

function init() {

stat = new Stats();

stat.domElement.style.position = 'absolute';

stat.domElement.style.right = '0px';

stat.domElement.style.top = '0px';

document.body.appendChild(stat.domElement);

// Three.js init ...

}

然后，在上一节介绍的动画重绘函数draw中调用stat.begin();与stat.end();分别表示一帧的开始与结束：

function draw() {

stat.begin();

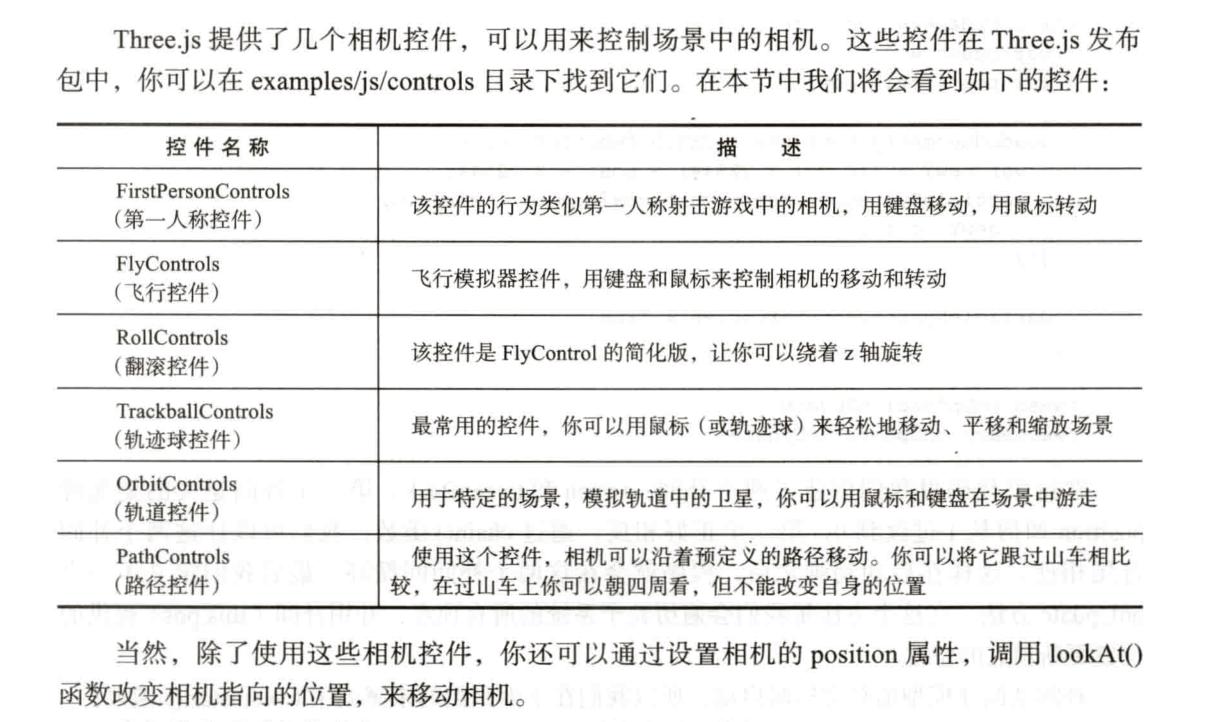
mesh.rotation.y = (mesh.rotation.y + 0.01) % (Math.PI \* 2);

renderer.render(scene, camera);

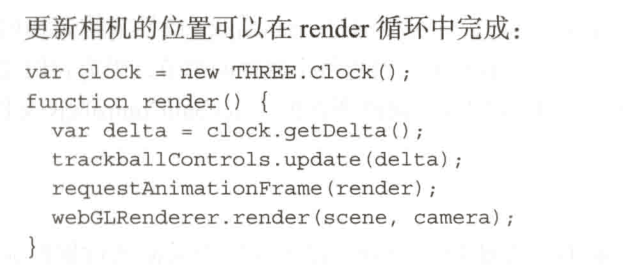
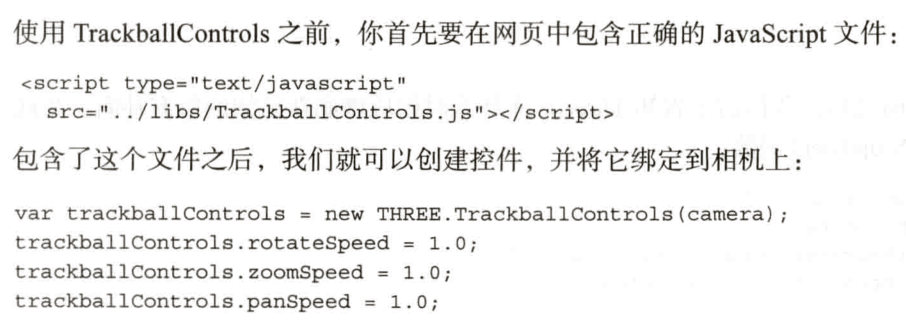
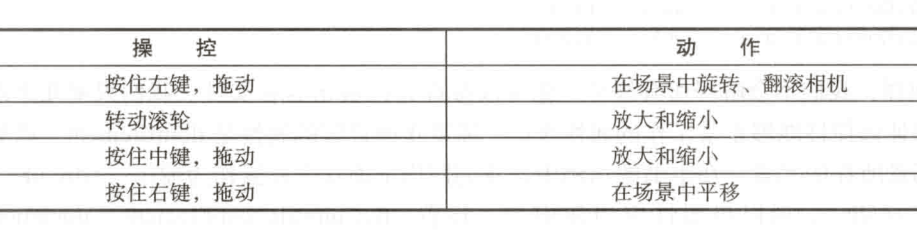
stat.end();

}最终就能得到FPS效果了。

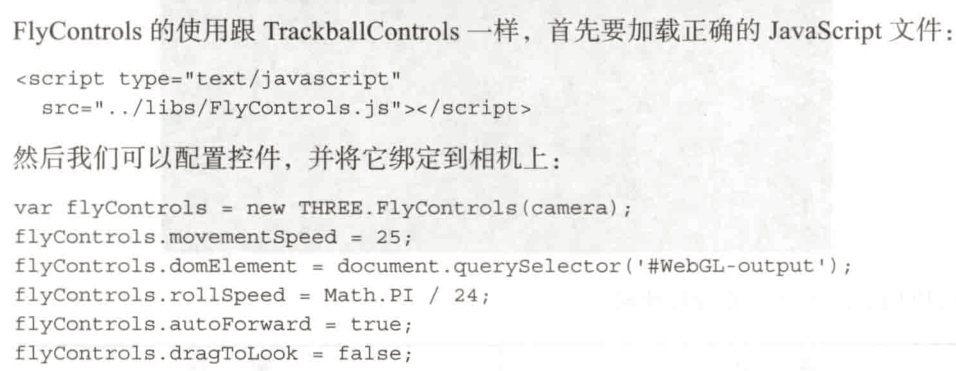
## 6.3相机控件



### 6.3.1轨迹球控件

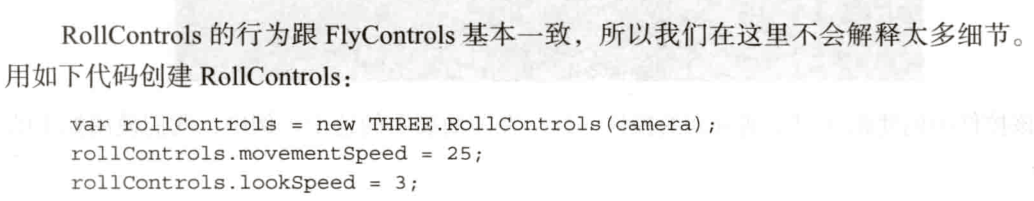


### 6.3.2飞行控件

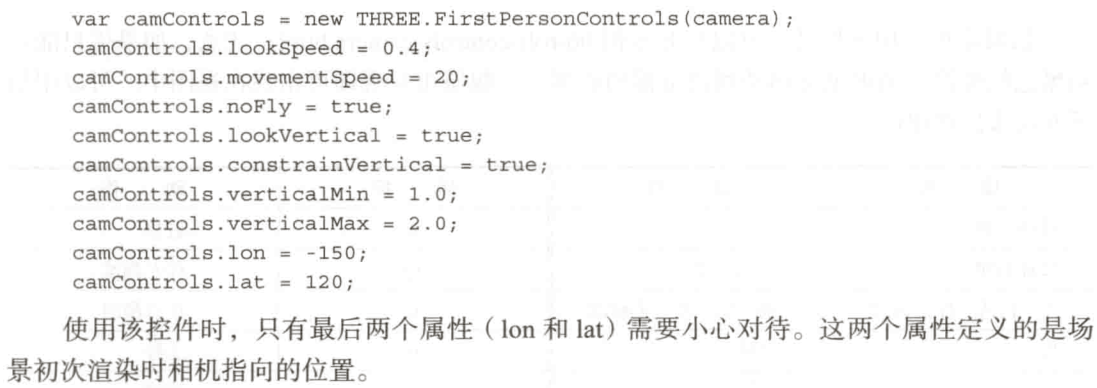
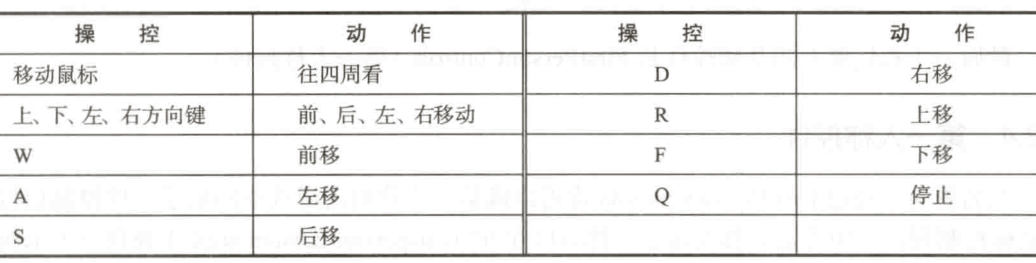


### 6.3.3翻滚控件

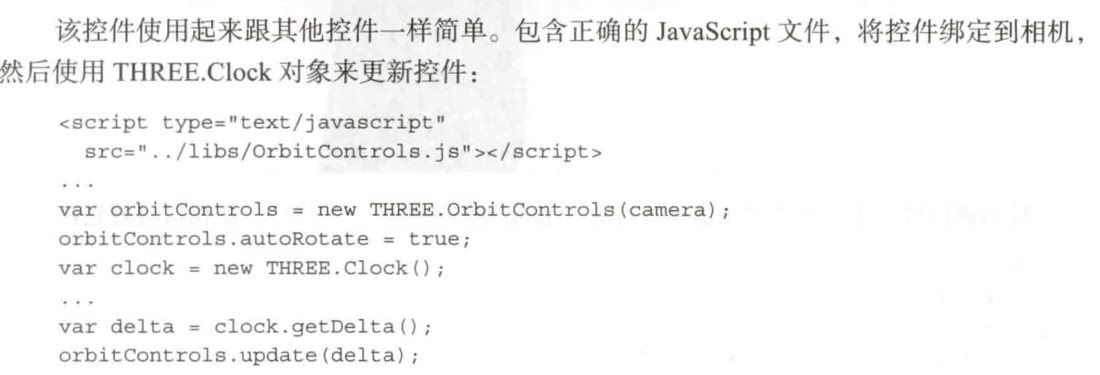
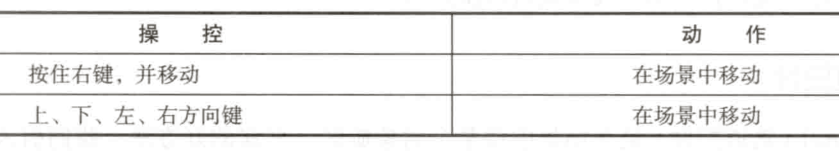
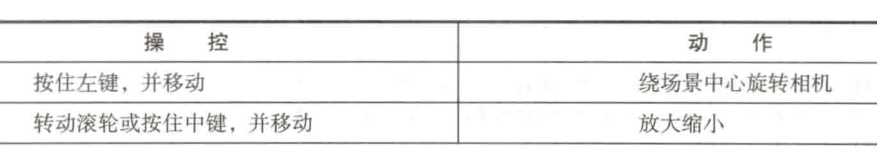




### 6.3.4第一人称控件



### 6.3.5轨道控件



# 七、外部模型

使用Three.js创建常见几何体是十分方便的，但是对于人或者动物这样非常复杂的模型使用几何体组合就非常麻烦了。因此，Three.js允许用户导入由3ds Max等工具制作的三维模型，并添加到场景中。

本章以3ds Max为例，介绍如何导入外部模型。

## 7.1支持格式

Three.js有一系列导入外部文件的辅助函数，是在three.js之外的，使用前需要额外下载，在<https://github.com/mrdoob/three.js/tree/master/examples/js/loaders>可以找到。

\*.obj是最常用的模型格式，导入\*.obj文件需要OBJLoader.js；导入带\*.mtl材质的\*.obj文件需要MTLLoader.js以及OBJMTLLoader.js。另有PLYLoader.js、STLLoader.js等分别对应不同格式的加载器，可以根据模型格式自行选择。

目前，支持的模型格式有：

\*.obj

\*.obj, \*.mtl

\*.dae

\*.ctm

\*.ply

\*.stl

\*.wrl

\*.vtk

## 7.2无材质的模型

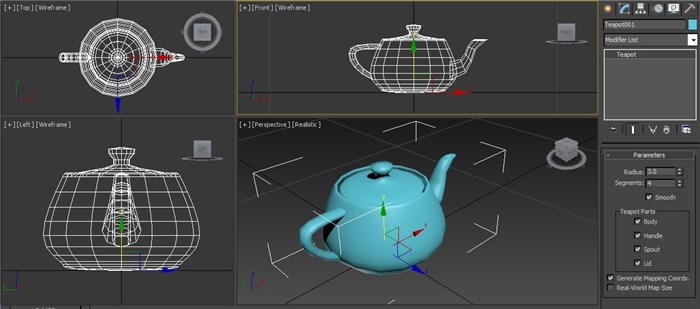
本节中，我们将使用*3ds Max*创建一个茶壶模型，并将导出的没有材质的模型使用Three.js导入场景中。

首先，下载[OBJLoader.js](https://github.com/mrdoob/three.js/raw/master/examples/js/loaders/OBJLoader.js)并在HTML的<head>中使用：

**[例7.2.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter7/7.2.1.html)**

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

然后，我们需要准备一个\*.obj模型，可以使用建模软件导出，也可以在网上下载。这里，我们在*3ds Max*中创建一个茶壶，将其放置在原点处。设置其半径为3，这一单位与我们的three.js场景的单位是一致的。

[[+]查看原图](http://www.ituring.com.cn/download/01Ydl4UN3RGN.big" \t "http://www.ituring.com.cn/article/_blank)

导出成port.obj，在选项中，如果选择了Export materials，会生成一个同名的\*.mtl文件。在本例中，我们不需要导出材质，因此可以不选中这项。看到导出的port.obj文件后，我们就可以进行下一步了。

在init函数中，创建loader变量，用于导入模型：

var loader = new THREE.OBJLoader();

loader导入模型的时候，接受两个参数，第一个表示模型路径，第二个表示完成导入后的回调函数，一般我们需要在这个回调函数中将导入的模型添加到场景中。

loader.load('../lib/port.obj', function(obj) {

mesh = obj; //储存到全局变量中

scene.add(obj);

});

看到的结果是不带材质的茶壶：



我们在重绘函数中让茶壶旋转：

function draw() {

renderer.render(scene, camera);

mesh.rotation.y += 0.01;

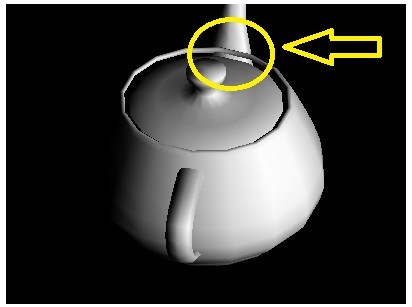
if (mesh.rotation.y > Math.PI \* 2) {

mesh.rotation.y -= Math.PI \* 2;

}

}

可以看到在某些角度时，好像有些面片没有被绘制出来，因而后方的茶嘴似乎穿越到前方了：



这是由于默认的情况下，只有正面的面片被绘制，而如果需要双面绘制，需要这样设置：

var loader = new THREE.OBJLoader();

loader.load('../lib/port.obj', function(obj) {

obj.traverse(function(child) {

if (child instanceof THREE.Mesh) {

child.material.side = THREE.DoubleSide;

}

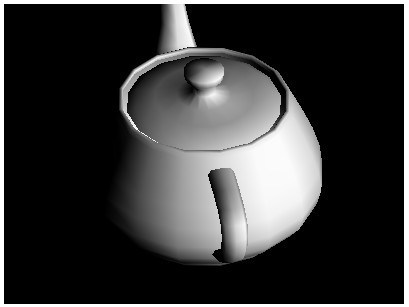
});

mesh = obj;

scene.add(obj);

});

现在，正反面都会被正确绘制了：



## 7.3有材质的模型

模型的材质可以有两种定义方式，一种是在代码中导入模型后设置材质，另一种是在建模软件中导出材质信息。下面，我们将分别介绍这两种方法。

### 7.3.1代码中设置材质

这种方法与**[例7.2.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter7/7.2.1.html)**类似，不同之处在于回调函数中设置模型的材质：

**[例7.3.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter7/7.3.1.html)**

var loader = new THREE.OBJLoader();

loader.load('../lib/port.obj', function(obj) {

obj.traverse(function(child) {

if (child instanceof THREE.Mesh) {

child.material = new THREE.MeshLambertMaterial({

color: 0xffff00,

side: THREE.DoubleSide

});

}

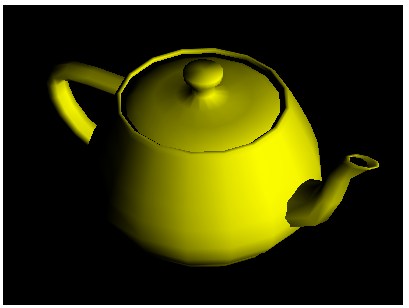
});

mesh = obj;

scene.add(obj);

});

效果为：



### 7.3.2建模软件中设置材质

使用上一节相似的方法导出模型，在选项中选中Export materials，最终导出port.obj模型文件以及port.mtl材质文件。

现在，我们不再使用OBJLoader.js，而是使用MTLLoader.js与OBJMTLLoader.js，并且要按改顺序引用：

**[例7.3.2](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter7/7.3.2.html)**

<script type="text/javascript" src="MTLLoader.js"></script><script type="text/javascript" src="OBJMTLLoader.js"></script>

调用方法略有不同：

var loader = new THREE.OBJMTLLoader();

loader.addEventListener('load', function(event) {

var obj = event.content;

mesh = obj;

scene.add(obj);

});

loader.load('../lib/port.obj', '../lib/port.mtl');

效果是我们在*3ds Max*中设置的材质：



# 光与影

图像渲染的丰富效果很大程度上也要归功于光与影的利用。真实世界中的光影效果非常复杂，但是其本质——光的传播原理却又是非常单一的，这便是自然界繁简相成的又一例证。为了使计算机模拟丰富的光照效果，人们提出了几种不同的光源模型（环境光、平行光、点光源、聚光灯等），在不同场合下组合利用，将能达到很好的光照效果。

在Three.js中，光源与阴影的创建和使用是十分方便的。在学会了如何控制光影的基本方法之后，如果能将其灵活应用，将能使场景的渲染效果更加丰富逼真。在本章中，我们将探讨四种常用的光源（环境光、点光源、平行光、聚光灯）和阴影带来的效果，以及如何去创建使用光影。

## 8.1环境光

环境光是指场景整体的光照效果，是由于场景内若干光源的多次反射形成的亮度一致的效果，通常用来为整个场景指定一个基础亮度。因此，环境光没有明确的光源位置，在各处形成的亮度也是一致的。

在设置环境光时，只需要指定光的颜色：

THREE.AmbientLight(hex)

其中，hex是十六进制的RGB颜色信息，如红色表示为0xff0000。

创建环境光并将其添加到场景中的完整做法是：

**[例8.1.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter8/8.1.1.html)**

var light = new THREE.AmbientLight(0xffffff);

scene.add(light);

但是，如果此时场景中没有物体，只添加了这个环境光，那么渲染的结果仍然是一片黑。所以，我们添加两个长方体看下效果：

var greenCube = new THREE.Mesh(new THREE.CubeGeometry(2, 2, 2),

new THREE.MeshLambertMaterial({color: 0x00ff00})

);

greenCube.position.x = 3;

scene.add(greenCube);

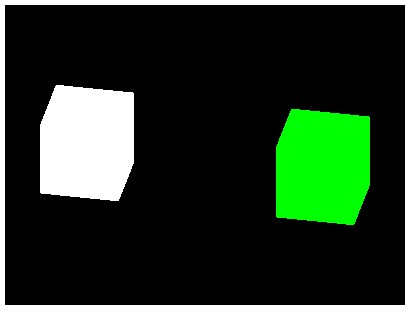
var whiteCube = new THREE.Mesh(new THREE.CubeGeometry(2, 2, 2),

new THREE.MeshLambertMaterial({color: 0xffffff})

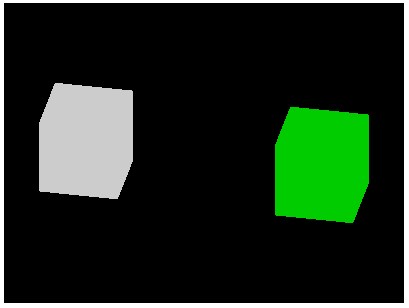
);

whiteCube.position.x = -3;

scene.add(whiteCube);



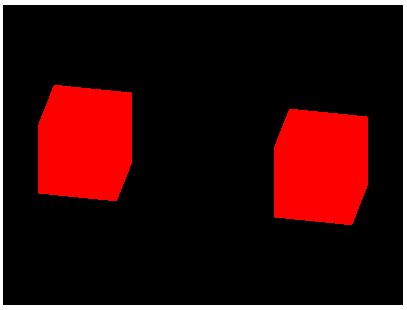
如果想让环境光暗些，可以将其设置为new THREE.AmbientLight(0xcccccc)等，效果为：



那么，如果使用红色的环境光会有什么样的效果呢？

**[例8.1.2](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter8/8.1.2.html)**

我们将环境光设置为红色，场景内同样放置绿色和白色的长方体，效果为：



我们将两个长方体材质的颜色分别设置为绿色和白色，渲染的结果是这两个长方体都被渲染成了环境光的红色，这一结果可能有些出乎你的意料。其实，环境光并不在乎物体材质的color属性，而是ambient属性。ambient属性的默认值是0xffffff。因此，如果将这两个长方体设置为：

**[例8.1.3](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter8/8.1.3.html)**

var greenCube = new THREE.Mesh(new THREE.CubeGeometry(2, 2, 2),

new THREE.MeshLambertMaterial({ambient: 0x00ff00}));

greenCube.position.x = 3;

scene.add(greenCube);

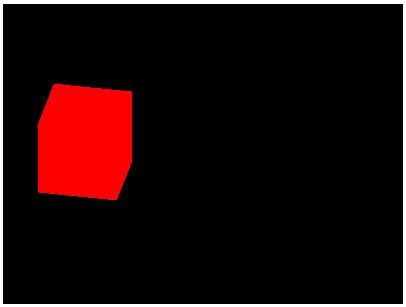
var whiteCube = new THREE.Mesh(new THREE.CubeGeometry(2, 2, 2),

new THREE.MeshLambertMaterial({ambient: 0xffffff}));

whiteCube.position.x = -3;

scene.add(whiteCube);

效果是：



也就意味着ambient为0x00ff00的右边的长方体被渲染成了黑色。这是因为不透明物体的颜色其实是其反射光的颜色，而ambient属性表示的是物体反射环境光的能力。对于0x00ff00的物体，红色通道是0，而环境光是完全的红光，因此该长方体不能反射任何光线，最终的渲染颜色就是黑色；而对于0xffffff的白色长方体，红色通道是0xff，因而能反射所有红光，渲染的颜色就是红色。

前面我们看到，当环境光不是白色或灰色的时候，渲染的效果往往会很奇怪。因此，环境光通常使用白色或者灰色，作为整体光照的基础。

## 8.2点源光

点光源是不计光源大小，可以看作一个点发出的光源。点光源照到不同物体表面的亮度是线性递减的，因此，离点光源距离越远的物体会显得越暗。

点光源的构造函数是：

THREE.PointLight(hex, intensity, distance)

其中，hex是光源十六进制的颜色值；intensity是亮度，缺省值为1，表示100%亮度；distance是光源最远照射到的距离，缺省值为0。

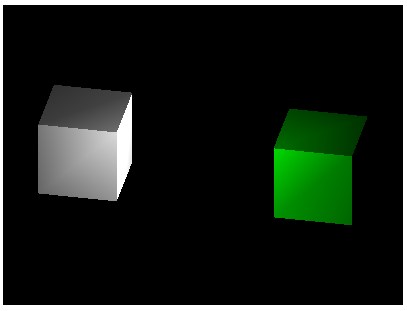
创建点光源并将其添加到场景中的完整做法是：

**[例8.2.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter8/8.2.1.html)**

var light = new THREE.PointLight(0xffffff, 2, 100);

light.position.set(0, 1.5, 2);

scene.add(light);



注意，这里光在每个面上的亮度是不同的，对于每个三角面片，将根据三个顶点的亮度进行插值。

## 8.3平行光

我们都知道，太阳光常常被看作平行光，这是因为相对地球上物体的尺度而言，太阳离我们的距离足够远。对于任意平行的平面，平行光照射的亮度都是相同的，而与平面所在位置无关。

平行光的构造函数是：

THREE.DirectionalLight(hex, intensity)

其中，hex是光源十六进制的颜色值；intensity是亮度，缺省值为1，表示100%亮度。

此外，对于平行光而言，设置光源位置尤为重要。

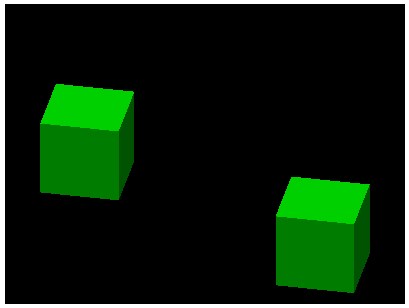
**[例8.3.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter8/8.3.1.html)**

var light = new THREE.DirectionalLight();

light.position.set(2, 5, 3);

scene.add(light);

注意，这里设置光源位置并不意味着所有光从(2, 5, 3)点射出（如果是的话，就成了点光源），而是意味着，平行光将以矢量(-2, -5, -3)的方向照射到所有平面。因此，平面亮度与平面的位置无关，而只与平面的法向量相关。只要平面是平行的，那么得到的光照也一定是相同的。



## 8.4聚光灯

从官网上的定义：

A point light that can cast shadow in one direction.

可以看出，聚光灯是一种特殊的点光源，它能够朝着一个方向投射光线。聚光灯投射出的是类似圆锥形的光线，这与我们现实中看到的聚光灯是一致的。

其构造函数为：

THREE.SpotLight(hex, intensity, distance, angle, exponent)

相比点光源，多了angle和exponent两个参数。angle是聚光灯的张角，缺省值是Math.PI / 3，最大值是Math.PI / 2；exponent是光强在偏离target的衰减指数（target需要在之后定义，缺省值为(0, 0, 0)），缺省值是10。

在调用构造函数之后，除了设置光源本身的位置，一般还需要设置target：

light.position.set(x1, y1, z1);

light.target.position.set(x2, y2, z2);

除了设置light.target.position的方法外，如果想让聚光灯跟着某一物体移动（就像真的聚光灯！），可以target指定为该物体：

var cube = new THREE.Mesh(new THREE.CubeGeometry(1, 1, 1),

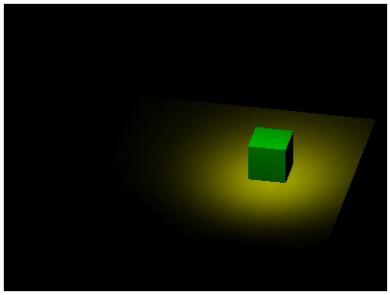
new THREE.MeshLambertMaterial({color: 0x00ff00}));

var light = new THREE.SpotLight(0xffff00, 1, 100, Math.PI / 6, 25);

light.target = cube;

不妨试着让该物体运动起来看下效果吧！

**[例8.4.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter8/8.4.1.html)**



## 8.5阴影

明暗是相对的，阴影的形成也就是因为比周围获得的光照更少。因此，要形成阴影，光源必不可少。

****在Three.js中，能形成阴影的光源只有**THREE.DirectionalLight**与**THREE.SpotLight**；而相对地，能表现阴影效果的材质只有**THREE.LambertMaterial**与**THREE.PhongMaterial**。****因而在设置光源和材质的时候，一定要注意这一点。

下面，我们以聚光灯为例，在[例8.4.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter8/8.4.1.html)的基础上增加阴影效果。

**[例8.5.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter8/8.5.1.html)**

首先，我们需要在初始化时，告诉渲染器渲染阴影：

renderer.shadowMapEnabled = true;

然后，对于光源以及所有要产生阴影的物体调用：

xxx.castShadow = true;

对于接收阴影的物体调用：

xxx.receiveShadow = true;

比如场景中一个平面上有一个正方体，想要让聚光灯照射在正方体上，产生的阴影投射在平面上，那么就需要对聚光灯和正方体调用castShadow = true，对于平面调用receiveShadow = true。

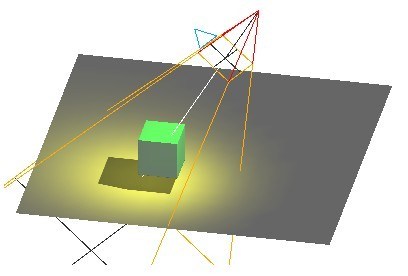
以上就是产生阴影效果的必要步骤了，不过通常还需要设置光源的阴影相关属性，才能正确显示出阴影效果。

对于聚光灯，需要设置shadowCameraNear、shadowCameraFar、shadowCameraFov三个值，类比我们在第二章学到的透视投影照相机，只有介于shadowCameraNear与shadowCameraFar之间的物体将产生阴影，shadowCameraFov表示张角。

对于平行光，需要设置shadowCameraNear、shadowCameraFar、shadowCameraLeft、shadowCameraRight、shadowCameraTop以及shadowCameraBottom六个值，相当于正交投影照相机的六个面。同样，只有在这六个面围成的长方体内的物体才会产生阴影效果。

为了看到阴影照相机的位置，通常可以在调试时开启light.shadowCameraVisible = true。

至此，阴影效果已经能正常显示了：



如果想要修改阴影的深浅，可以通过设置shadowDarkness，该值的范围是0到1，越小越浅。

另外，这里实现阴影效果的方法是[Shadow Mapping](http://en.wikipedia.org/wiki/Shadow_mapping" \t "http://www.ituring.com.cn/article/_blank)，即阴影是作为渲染前计算好的贴图贴上去的，因而会受到贴图像素大小的限制。所以可以通过设置shadowMapWidth与shadowMapHeight值控制贴图的大小，来改变阴影的精确度。

而如果想实现软阴影的效果，可以通过renderer.shadowMapSoft = true;方便地实现。

设置阴影完整的代码是：

renderer = new THREE.WebGLRenderer();

renderer.shadowMapEnabled = true;

renderer.shadowMapSoft = true;

var plane = new THREE.Mesh(new THREE.PlaneGeometry(8, 8, 16, 16),

new THREE.MeshLambertMaterial({color: 0xcccccc})

);

plane.rotation.x = -Math.PI / 2;

plane.position.y = -1;

plane.receiveShadow = true;

scene.add(plane);

cube = new THREE.Mesh(new THREE.CubeGeometry(1, 1, 1),

new THREE.MeshLambertMaterial({color: 0x00ff00})

);

cube.position.x = 2;

cube.castShadow = true;

scene.add(cube);

var light = new THREE.SpotLight(0xffff00, 1, 100, Math.PI / 6, 25);

light.position.set(2, 5, 3);

light.target = cube;

light.castShadow = true;

light.shadowCameraNear = 2;

light.shadowCameraFar = 10;

light.shadowCameraFov = 30;

light.shadowCameraVisible = true;

light.shadowMapWidth = 1024;

light.shadowMapHeight = 1024;

light.shadowDarkness = 0.3;

scene.add(light);

# 着色器

本章将介绍关于渲染的一些高级话题，使用着色器可以更灵活地控制渲染效果，结合纹理，可以进行多次渲染，达到更强大的效果。

## 9.1渲染与着色器

渲染”（Rendering）是即使非计算机专业的都不会觉得陌生的词，虽然在很多人说这个词的时候，并不清楚“渲染”究竟意味着什么。相反，“着色器”（Shader）很可能是大家比较陌生的词，从名字看上去似乎是用来上色的，但它具体能做什么呢？

在解释着色器之前，我们先来聊聊渲染。

### 9.1.1渲染

用通俗的话来说，渲染就是将模型数据在屏幕上显示出来的过程。

这听起来好像很简单呢！但正如你打开一个Word写文档一样，之所以这个过程看起来毫不费力是因为那些繁杂而枯燥的活都让计算机完成了。同样，要渲染出一幅画面GPU也需要做很多工作，如果你有兴趣了解的话，可以查阅渲染流水线（Rendering Pineline）的相关知识。

Three.js最重要的一个好处就是让你在无需知道图形学知识的前提下完成从建模到渲染的一整套工作。因而，在本书中我们不会对图形学知识做展开，否则就是违背了这一本意了。在这里，我们只要理解渲染做的将你的模型数据呈现在屏幕上的过程即可。

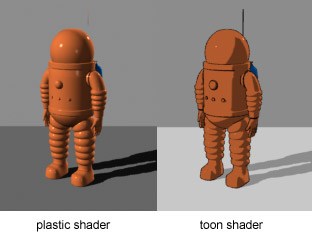
### 9.1.2着色器

在定义了场景中的物体、照相机、光源等等之后，渲染的结果就确定了吗？

在可编程GPU时代到来前，答案是肯定的，但现在我们已经可以通过着色器程序对GPU编程来控制渲染的结果。着色器是屏幕上呈现画面之前的最后一步，用它可以对先前渲染的结果做修改，包括对颜色、位置等等信息的修改，甚至可以对先前渲染的结果做后处理，实现高级的渲染效果。

如果这听上去很抽象，那让我们来看一些具体的例子吧！

比如，我们要渲染一个宇航员，使用同样的模型、同样的光源、同样的照相机，但是不同的着色器，我们就能得到不同的渲染效果：



左图的是塑料效果，右图的是卡通效果，这都是由不同的着色器实现的。

我们知道WebGL是基于OpenGL的，而OpenGL用GLSL（OpenGL Shading Language）这一着色器语言完成着色器工作，因此，WebGL的着色器程序大致与GLSL相同，是一种接近C语言的代码。着色器通常分为几何着色器（Geometry Shader）、顶点着色器（Vertex Shader）、片元着色器（Fragment Shader）等等。由于WebGL基于OpenGL ES 2.0，因此WebGL支持的着色器只有顶点着色器与片元着色器。

### 9.1.3顶点着色器

顶点着色器中的“顶点”指的正是Mesh中的顶点，对于每个顶点调用一次。因此，如果场景中有一个正方体，那么对八个顶点将各自调用一次顶点着色器，可以修改顶点的位置或者颜色等信息，然后传入片元着色器。

### 9.1.4片元着色器

片元是栅格化之后，在形成像素之前的数据。片元着色器是每个片元会调用一次的程序，因此，片元着色器特别适合用来做图像后处理。

### 9.1.5Three.js与着色器

由此，我们看到，着色器可以用来渲染高级的效果。但是对于很多应用而言，并不需要着色器。

WebGL强制需要程序员定义着色器，即使你只是希望采用默认的渲染方法。这似乎有些不近人情，尤其对于对图形学理解不多的开发者而言。

幸运的是，Three.js允许你不定义着色器（就像前面所有章节的例子）采用默认的方法渲染，而仅在你有需要时，才使用自定义的着色器，这大大减少了程序员的工作量，而且对于初学者而言，无疑是减少入门难度的福音。

## 9.2初窥着色器

在这节中，我们将通过具体的着色器代码，初步理解着色器编程。至于具体如何将着色器应用于程序，将在下一节做介绍。

我们回顾一下上节内容，着色器是一段在GPU中执行的接近C语言的代码，顶点着色器对于每个顶点调用一次，片元着色器对于每个片元调用一次。

着色器语言的调试有时候十分困难，很可能报的错让你不明所以。建议使用Chrome和Firefox调试，此外，[Chrome的一个插件](https://chrome.google.com/webstore/detail/webgl-inspector/ogkcjmbhnfmlnielkjhedpcjomeaghda)也可能给你提供一定帮助。另外，从我写着色器的经验来看，最常发生错误的原因就是忘记float类型和int类型不会自动转换的，因此，当你想表达浮点数零的时候，一定要写成0.0而非0。当然，即使我在这里提醒大家了，你仍然会惊讶这一错误发生的频率之高！

### 9.2.1顶点着色器

着色器是类似C语言的代码，即便如此，下面代码仍然可能让你感到困惑：

varying vec2 vUv;

void main(){

// passing texture to fragment shader

vUv = uv;

gl\_Position = projectionMatrix \* modelViewMatrix \* vec4(position, 1.0);

}

我们可以猜测到，和C语言一样，着色器程序也从main函数开始调用。但除此之外……就有点看不懂了吧？

让我们一起来认识一下varing。它是WebGL定义的****限定符****（Qualifier），限定符用于数据类型（Type）之前，表明该变量的性质。

限定符共有四个：

const：这是我们熟悉的常量的意思

attribute：从JavaScript代码传递到顶点着色器中，每个顶点对应不同的值

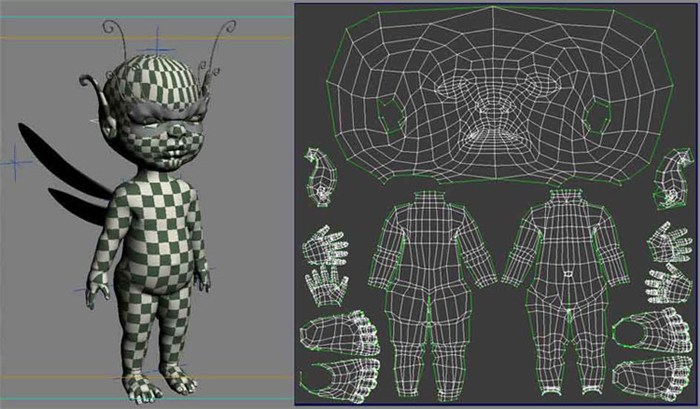
uniform：每个顶点/片元对应相同的值

varying：从顶点着色器传递到片元着色器中

如果不写限定符，那么默认是只有在当前文件中能访问。

所以，varying vec2 vUv;的意思是，声明了一个叫vUv的变量，它的类型为vec2，该变量是为了将顶点着色器中的信息传递到片元着色器中。那么它传递了什么信息呢？我们看到与之相关的只有vUv = uv;，可是uv都没声明过啊！这是哪里来的？

其实，uv是Three.js帮你传进来的一个很有用的属性，它代表了该顶点在**[UV映射](http://en.wikipedia.org/wiki/UV_mapping" \t "http://www.ituring.com.cn/article/_blank)**时的横纵坐标。简单地说，一个物体的模型可能很复杂，对其贴图的一个简单有效的方法就是UV映射，将每个面片贴的图统一映射到一张纹理上，记录每个面片贴图在纹理上对应的位置。得到这样的效果：



而之所以称为u和v，指的就是在纹理映射后的新坐标系。我们也发现，uv变量的类型是vec2，顾名思义就是一个二维的向量，可以使用uv.x和uv.y分别访问到uv两个维度的值。

使用varying vec2 vUv;将uv信息传递到片元着色器是因为片元着色器本身不能访问到uv信息，如果需要得到这一值的话，就需要从顶点着色器中传递过去，我们将其命名为vUv。

那么，gl\_Position = projectionMatrix \* modelViewMatrix \* vec4(position, 1.0);又是在干什么呢？学过图形学的读者一定对投影矩阵、模型矩阵并不陌生，这里做的事情就是计算三维模型在二维显示屏上的坐标。这里，我们看到position也没有预先定义过，不过通过上面的uv应该也能猜测到position也是Three.js为我们提供的一个方便。position是顶点在物体坐标系（而不是世界坐标系）中的位置。这就意味着，一个正方体位于世界坐标系的(2, 0, 0)与位于(0, 0, 0)将不会改变任何顶点的position，这个position是相对于正方体的锚点而言的。

因此，这段顶点着色器的作用就是将uv信息传递到片元着色器中，并按默认的方式计算顶点位置。

### 9.2.2片元着色器

有了前面顶点着色器传过来的vUv信息，我们能做些有意思的事了吧？比如来看看使用颜色表示uv信息如何？

varying vec2 vUv;void main() {

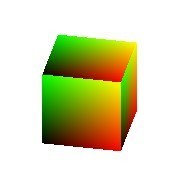
gl\_FragColor = vec4(vUv.x, vUv.y, 1.0, 1.0);

}

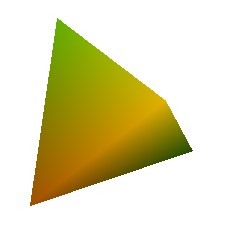
太好了，它看起来很简单！

你能告诉我上面代码是什么意思吗？

来看看你回答得对不对。varying vec2 vUv;同样声明了从顶点着色器传递到片元着色器中的vUv属性，记得要在片元着色器中再写一遍。主程序只有gl\_FragColor = vec4(vUv.x, vUv.y, 1.0, 1.0);，gl\_FragColor用来设置片元的颜色，vec4的四个维度分别表示红、绿、蓝以及alpha通道。因此，这里我们是将vUv的两个维度分别对应到红绿通道，得到的效果是：



现在，你是不是对UV映射有更深的理解了呢？对于正方体而言，每个面都映射到了整个UV纹理，所以呈现了如上结果。而对于正四面体而言，每个面都映射到了UV纹理的一部分，因此呈现的效果是这样的：



## 9.3着色器完整实例

### 9.3.1着色器程序的位置

着色器代码可以写在单独的文件中（顶点着色器的文件名后缀为.vs，片元着色器的文件名后缀为.fs），也可以在HTML文件中定义script标签实现。通常对于较长的着色器代码，建议使用单独的文件；对于较短的着色器代码，在HTML文件中定义也是一个不错的选择。当然，从代码可维护性的角度看，本书更建议使用单独的着色器文件。

### 9.3.2单独的着色器文件

使用单独的着色器文件，需要在javascript代码中导入着色器文件。我们假设顶点着色器定义在shader/my.vs文件中，片元着色器定义在shader/my.fs中。

可以使用Ajax完成导入文件的工作，而如果使用****jQuery****的get函数就可以更方便地实现。

// load shader

$.get('shader/my.vs', function(vShader){

$.get('shader/my.fs', function(fShader){

// TODO

});

});

jQuery的get函数第一个参数为文件路径，第二个参数为导入文件后的回调函数，这里我们在加载完顶点着色器后加载片元着色器。vShader与fShader分别为导入的着色器程序，用来构造着色器材质。

接下来，我们需要在加载完两个着色器后，新建一个THREE.ShaderMaterial，需要传入属性vertexShader与fragmentShader：

$.get('shader/my.vs', function(vShader){

$.get('shader/my.fs', function(fShader){

material = new THREE.ShaderMaterial({

vertexShader: vShader,

fragementShader: fShader

});

});

});

之后可以将material应用于需要该着色器效果的物体上。

### 9.3.3HTML中的着色器代码

在HTML中，可以使用

<script id="vs" type="x-shader/x-vertex">这里的内容相当于.vs文件中的内容

</script>

定义顶点着色器；使用

<script id="fs" type="x-shader/x-fragment">这里的内容相当于.fs文件中的内容

</script>

定义片元着色器。

定义材质时的方法：

material = new THREE.ShaderMaterial({

vertexShader: document.getElementById('vs').textContent,

fragmentShader: document.getElementById('fs').textContent

});

### 9.3.4完整实例

下面，我们通过完整的例子了解着色器的应用。

**[例9.3.1](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter9/9.3.1.html)**，**[例9.3.2](https://github.com/Ovilia/ThreeExample.js/blob/master/Chapter9/9.3.2.html)**

首先，我们创建一个绿色的正方体在场景中旋转，这些都是在前几章中讲解过的：

var scene = null;var camera = null;var renderer = null;var cube = null;

function init() {

//画布

renderer = new THREE.WebGLRenderer({

canvas: document.getElementById('mainCanvas')

});

//场景

scene = new THREE.Scene();

//照相机

camera = new THREE.OrthographicCamera(-5, 5, 3.75, -3.75, 0.1, 100);

camera.position.set(5, 15, 25);

camera.lookAt(new THREE.Vector3(0, 0, 0));

//渲染

scene.add(camera);

//光源

var light = new THREE.DirectionalLight();

light.position.set(3, 2, 5);

scene.add(light);

//实体

cube = new THREE.Mesh(new THREE.CubeGeometry(2, 2, 2),

new THREE.MeshLambertMaterial({color: 0x00ff00})

);

scene.add(cube);

//动画

draw();

}

//动画

function draw() {

cube.rotation.y += 0.01;

if (cube.rotation.y > Math.PI \* 2) {

cube.rotation.y -= Math.PI \* 2;

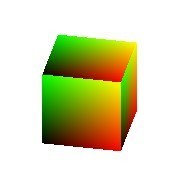
}

renderer.render(scene, camera);

requestAnimationFrame(draw);

}

然后，我们需要定义着色器代码，并导入到应用中。着色器程序参见上节，导入着色器的两种方法在本节也做了介绍。因此，最终得到的结果是：



# 附录1：关键字

Cameras（照相机，控制投影方式）

Camera、OrthographicCamera、PerspectiveCamera

Core（核心对象）

BufferGeometry、Clock（用来记录时间）、EventDispatcher、Face3、Face4、Geometry、Object3D、Projector、Raycaster（计算鼠标拾取物体时很有用的对象）

Lights（光照）

Light、AmbientLight、AreaLight、DirectionalLight、HemisphereLight、PointLight、SpotLight

Loaders（加载器，用来加载特定文件）

Loader、BinaryLoader、GeometryLoader、ImageLoader、JSONLoader、LoadingMonitor、SceneLoader、TextureLoader

Materials（材质，控制物体的颜色、纹理等）

Material、LineBasicMaterial、LineDashedMaterial、MeshBasicMaterial、MeshDepthMaterial、MeshFaceMaterial、MeshLambertMaterial、MeshNormalMaterial、MeshPhongMaterial、ParticleBasicMaterial、ParticleCanvasMaterial、ParticleDOMMaterial、ShaderMaterial、SpriteMaterial

Math（和数学相关的对象）

Box2、Box3、Color、Frustum、Math、Matrix3、Matrix4、Plane、Quaternion、Ray、Sphere、Spline、Triangle、Vector2、Vector3、Vector4

Objects（物体）

Bone、Line、LOD、Mesh（网格，最常用的物体）、MorphAnimMesh、Particle、ParticleSystem、Ribbon、SkinnedMesh、Sprite

Renderers（渲染器，可以渲染到不同对象上）

CanvasRenderer、WebGLRenderer（使用WebGL渲染，这是本书中最常用的方式）、WebGLRenderTarget、WebGLRenderTargetCube、WebGLShaders（着色器）

Renderers / Renderables

RenderableFace3、RenderableFace4、RenderableLine、RenderableObject、RenderableParticle、RenderableVertex

Scenes（场景）

Fog、FogExp2、Scene

Textures（纹理）

CompressedTexture、DataTexture、Texture

Extras

FontUtils、GeometryUtils、ImageUtils、SceneUtils

Extras / Animation

Animation、AnimationHandler、AnimationMorphTarget、KeyFrameAnimation

Extras / Cameras

CombinedCamera、CubeCamera

Extras / Core

Curve、CurvePath、Gyroscope、Path、Shape

Extras / Geometries（几何形状）

CircleGeometry、ConvexGeometry、CubeGeometry、CylinderGeometry、ExtrudeGeometry、IcosahedronGeometry、LatheGeometry、OctahedronGeometry、ParametricGeometry、PlaneGeometry、PolyhedronGeometry、ShapeGeometry、SphereGeometry、TetrahedronGeometry、TextGeometry、TorusGeometry、TorusKnotGeometry、TubeGeometry

Extras / Helpers

ArrowHelper、AxisHelper、CameraHelper、DirectionalLightHelper、HemisphereLightHelper、PointLightHelper、SpotLightHelper

Extras / Objects

ImmediateRenderObject、LensFlare、MorphBlendMesh

Extras / Renderers / Plugins

DepthPassPlugin、LensFlarePlugin、ShadowMapPlugin、SpritePlugin

Extras / Shaders

ShaderFlares、ShaderSprite

# 附录2：three框架

<!DOCTYPE **html**>  
<**html**>  
<**head lang="en"**>  
 <**meta charset="UTF-8"**>  
 <**title**></**title**>  
</**head**>  
<**body onload="***threeStart*();**"**>  
<**div id="main" style="width**: 100%;**height**: 800**px**;**cursor**: **pointer**;**background-color**: **#EEEEEE**;**border**: **none**;**"**>  
 <**label**>当前坐标（请使用z、x、上、下、左、右按键控制当前位置）</**label**>  
 <**input type="text" id="input"**><**br**>  
 <**canvas id="canvas-frame" style="width**: 100%;**height**: 100%;**"**></**canvas**>  
 *<!--<div id="canvas-frame" style="width: 100%;height: 100%;"></div>-->*</**div**>  
</**body**>  
<**script type="text/javascript" src="js/three.js"**></**script**>  
<**script type="text/javascript" src="js/Stats.js"**></**script**>  
<**script type="text/javascript"**>  
  
 *//相机移动速度* **var *speed*** = 10;  
  
 */\*\*  
 \* 框架调用函数  
 \*/* **function** *threeStart*() {  
 *initThree*();  
 *initCamera*();  
 *initScene*();  
 *initLight*();  
 *initObject*();  
 *render*();  
*// animation();* }  
  
 **var *renderer***; *//渲染器* **var *stats***;  
 **var *canvasWidth***; *//画布宽canvas* **var *canvasHeight***; *//画布高  
 /\*\*  
 \* 初始化框架渲染器  
 \*/* **function** *initThree*() {  
 **var** canvas = **document**.getElementById(**'canvas-frame'**);  
 ***canvasWidth*** = canvas.**clientWidth**;  
 ***canvasHeight*** = canvas.**clientHeight**;  
  
 ***renderer*** = **new *THREE***.WebGLRenderer({  
 **canvas**: canvas  
 });  
 ***renderer***.setSize(***canvasWidth***, ***canvasHeight***);  
 ***renderer***.setClearColor(0xEEEEEE); *// black  
  
 //定义div时设置  
// renderer = new THREE.WebGLRenderer({  
// antialias : true  
// });  
// renderer.setSize(canvasWidth, canvasHeight);  
// canvas.appendChild(renderer.domElement);  
// renderer.setClearColor(0xEEEEEE, 1.0);  
 //性能测试框* ***stats*** = **new** *Stats*();  
 ***stats***.**domElement**.style.**position** = **'absolute'**;  
 ***stats***.**domElement**.style.**right** = **'0px'**;  
 ***stats***.**domElement**.style.**top** = **'0px'**;  
 **document**.getElementById(**'main'**).appendChild(***stats***.**domElement**);  
 }  
  
 **var *camera***; *//相机  
 /\*\*  
 \* 初始化相机  
 \*/* **function** *initCamera*() {  
 *//透视投影照相机* ***camera*** = **new *THREE***.PerspectiveCamera(45, ***canvasWidth*** / ***canvasHeight***, 1, 10000);  
*// camera = new THREE.OrthographicCamera(canvasWidth/-2,canvasWidth/2,canvasHeight/2,canvasHeight/-2,1,1000);* ***camera***.position.set(600, 0, 600);  
 ***camera***.up.set(0, 1, 0);  
 ***camera***.**lookAt**(**new *THREE***.Vector3(0, 0, 0));  
 *//正投影照相机  
// camera = new THREE.OrthographicCamera(canvasWidth/-2,canvasWidth/2,canvasHeight/2,canvasHeight/-2,1,1000);  
// camera.position.set(0, 0, 600);  
// camera.up.set(0, 1, 0);  
// camera.lookAt(new THREE.Vector3(0, 0, 0));* }  
  
 **var *scene***; *//场景  
 /\*\*  
 \* 初始化场景  
 \*/* **function** *initScene*() {  
 ***scene*** = **new *THREE***.Scene();  
 }  
  
 **var *light***; *//光源  
 /\*\*  
 \* 初始化光源  
 \*/* **function** *initLight*() {  
*// light = new THREE.DirectionalLight(0x00EEEE);  
// light.position.set(300, 0, 0);  
// scene.add(light);  
 //红光* ***light*** = **new *THREE***.AmbientLight(0xFF0000);  
 ***light***.position.set(100, 100, 200);  
 ***scene***.add(***light***);  
 *//黄光* ***light*** = **new *THREE***.PointLight(0x00FF00);  
 ***light***.position.set(0, 0, 300);  
 ***scene***.add(***light***);  
 }  
  
 **var *cube***; *//景物  
 /\*\*  
 \* 初始化对象（画实物物体）  
 \*/* **function** *initObject*() {  
 */\*\*  
 \* 画一条颜色逐渐变化的线条  
 \*/  
// var geometry = new THREE.Geometry(); //几何体对象  
// var material = new THREE.LineBasicMaterial({vertexColors: THREE.VertexColors}); //线条材质  
// geometry.colors.push(new THREE.Color(0x444444), new THREE.Color(0xFF0000));  
 // 线的材质可以由2点的颜色决定  
// geometry.vertices.push(new THREE.Vector3(-100, 0, 100));  
// geometry.vertices.push(new THREE.Vector3(100, 0, -100));  
// var line = new THREE.Line( geometry, material, THREE.LinePieces );  
// scene.add(line);  
  
 /\*\*  
 \* 画棋盘  
 \*/  
// var geometry = new THREE.Geometry(); //几何体对象  
// var material = new THREE.LineBasicMaterial({vertexColors: THREE.VertexColors}); //材质  
// geometry.colors.push(new THREE.Color(0x444444), new THREE.Color(0xFF0000));  
// geometry.vertices.push(new THREE.Vector3(-500, 0, 0));  
// geometry.vertices.push(new THREE.Vector3(500, 0, 0));  
// for (var i = 0; i < 21; i++) {  
// var line = new THREE.Line(geometry, material);  
// line.position.z = (i \* 50) - 500;//z轴移动  
// scene.add(line);  
//  
// line = new THREE.Line(geometry, material);  
// line.rotation.y = 90 \* Math.PI / 180;//旋转90度  
// line.position.x = (i \* 50) - 500;//x轴移动  
// scene.add(line);  
//  
// }  
  
 /\*\*  
 \* 画圆台  
 \*/  
// var geometry = new THREE.CylinderGeometry(70,100,200);  
// var material = new THREE.MeshLambertMaterial( {color:0xFFFFFF} );  
// var mesh = new THREE.Mesh(geometry, material);  
// mesh.position = new THREE.Vector3(0,0,0);  
// scene.add(mesh);  
  
 //画六个长方体* **var** geometry = **new *THREE***.**CubeGeometry**(200, 100, 50);  
 **var** material = **new *THREE***.MeshLambertMaterial( { **color**:0xFFFFFF} );  
 **var** mesh = **new *THREE***.Mesh(geometry,material);  
 mesh.position.set(0,0,0);  
 ***scene***.add(mesh);  
 **var** mesh2 = **new *THREE***.Mesh(geometry,material);  
 mesh2.position.set(-300,0,0);  
 ***scene***.add(mesh2);  
 **var** mesh3 = **new *THREE***.Mesh(geometry,material);  
 mesh3.position.set(0,-150,0);  
 ***scene***.add(mesh3);  
 **var** mesh4 = **new *THREE***.Mesh(geometry,material);  
 mesh4.position.set(0,150,0);  
 ***scene***.add(mesh4);  
 **var** mesh5 = **new *THREE***.Mesh(geometry,material);  
 mesh5.position.set(300,0,0);  
 ***scene***.add(mesh5);  
 **var** mesh6 = **new *THREE***.Mesh( geometry,material);  
 mesh6.position.set(0,0,-100);  
 ***scene***.add(mesh6);  
 }

*/\*\*  
 \* 画布渲染操作  
 \*/* **function** *render*() {  
 ***renderer***.clear();  
 ***renderer***.render(***scene***, ***camera***);  
 requestAnimationFrame(*render*);  
 *inputValue*();  
 }

*/\*\*  
 \* 动画循环  
 \*/* **function** *animation*() {  
 *//renderer.clear();* ***camera***.position.**z** = ***camera***.position.**z** + 1;  
 ***renderer***.render(***scene***, ***camera***);  
 requestAnimationFrame(*animation*);  
 }

*/\*\*  
 \* 键盘控制时间  
 \*/* **document**.onkeydown = **function** (e) {  
 e = **window**.**event** || e;  
 **switch** (e.**keyCode**) {  
 **case** 37: *//左键* ***camera***.position.**x** = ***camera***.position.**x** - ***speed***;  
 ***renderer***.render(***scene***, ***camera***);  
 **break**;  
 **case** 38: *//向上键* ***camera***.position.**y** = ***camera***.position.**y** - ***speed***;  
 ***renderer***.render(***scene***, ***camera***);  
 **break**;  
 **case** 39: *//右键* ***camera***.position.**x** = ***camera***.position.**x** + ***speed***;  
 ***renderer***.render(***scene***, ***camera***);  
 **break**;  
 **case** 40: *//向下键* ***camera***.position.**y** = ***camera***.position.**y** + ***speed***;  
 ***renderer***.render(***scene***, ***camera***);  
 **break**;  
 **case** 90: *//z键* ***camera***.position.**z** = ***camera***.position.**z** + ***speed***;  
 ***renderer***.render(***scene***, ***camera***);  
 **break**;  
 **case** 88: *//x键* ***camera***.position.**z** = ***camera***.position.**z** - ***speed***;  
 ***renderer***.render(***scene***, ***camera***);  
 **break**;  
 **default**:  
 **break**;  
 }  
 *inputValue*();  
 }

*/\*\*  
 \* 设置当前位置显示在文本框中  
 \*/* **function** *inputValue*() {  
 **document**.getElementById(**"input"**).**value** = ***camera***.position.**x** + **","** + ***camera***.position.**y** + **","** + ***camera***.position.**z**;  
 }  
</**script**>  
</**html**>