Three.js 入门指南

前言

作者简介：张雯莉，上海交通大学软件学院数字艺术方向的在读研究生，擅长图形图像处理、网页前端设计。个人网站：http://zhangwenli.com，电子邮箱：OviliaZhang@gmail.com，Github：http://github.com/Ovilia。

欢迎读者给本书提出宝贵意见，也欢迎交流网页前端设计的其他话题。

献给所有将创造令人心动的应用的程序员

在 Intel 公司实习阶段，我制作了一个基于 Web Audio 的库 jWebAudio，为了演示这个库的三维音效效果，我决定自学 Three.js。由于之前有计算机图形学等课程的基础，而且 Three.js 中的很多概念是十分容易理解的，最终我在三天内快速地完成了三维打砖块游戏 Arcalands。用 Three.js 创建三维图形应用的高效性让我有些吃惊。后来，我又在多个项目中使用了 Three.js，慢慢加深了对它的了解。

回顾学习 Three.js 的过程，我发现虽然目前网上使用 Three.js 的实例很多，但真正系统介绍该库的教程很少，官方文档又不齐全。对于入门者而言，仅仅看着别人的例子和 API 有时候还不足以“入门”。而我自己是通过很多个项目的实践，才慢慢对 Three.js 有了比较全面的了解。因此，我希望通过这本书给想学习 Three.js 的读者提供一个比较系统的入门介绍。

本书特色

本书是目前市场上唯一一本介绍 Three.js 技术的书，旨在通过系统化的介绍，让初学者能够着手使用 Three.js，在网页上创建炫酷的三维图形应用，并学会持续学习进阶知识方法。

本书每个章节都通过具体的例子阐释相关知识点，所有代码都可以在 Github 上找到。

本书读者

我们常说，这是一个信息爆炸的时代。这就意味着，虽然信息的获取变得越来越廉价，但学习新技术的成本却在不断增加。就作者个人而言，每次在学习一个新技术之前都要斟酌再三，因为可学习的内容越来越多，学习的机会成本也就变大了。所以，在阅读本书前，请你回答以下问题，如果您有一个回答“是”，那么本书就是为您打造的：

- 我学过 JavaScript，想要快速开发一款三维网页游戏，但我没有什么网页游戏开发经验。
- 我想要使用 WebGL，但是我没学过 OpenGL，对图形渲染也没什么概念。
- 我听说过 Three.js，正好想要学学，苦于没有一个完整的教程。
- 我对 Three.js 比较熟悉，想要更全面地了解它，并学习一些高阶的知识。
- 我是来打酱油的，说不定会看到什么感兴趣的内容。

如果您有一个回答“是”，那么本书现在并不太适合您，或许您可以稍后再来看看：

- 我完全不懂 JavaScript
- 我想要学习 OpenGL、WebGL 这些比较底层的图形接口
- 我赶着加班……哎，需求又改了！

本书结构

本书针对 Three.js 的几个重要话题分章节介绍。

- 第 1 章介绍 Three.js 和 WebGL 的背景资料，并通过简单的例子帮助读者实现第一个 Three.js 应用。
- 第 2 章介绍照相机的设定。
- 第 3、4、5 章分别介绍几何形状、材质和网格，即如何在场景中添加物体。
- 第 6 章介绍如何实现动画效果。
- 第 7 章介绍如何导入外部模型。
- 第 8 章介绍添加光源和阴影效果。
- 第 9 章介绍高阶话题——着色器。

对于了解如何使用 Three.js 创建简单应用的读者可以跳过第 1 章，否则建议首先阅读第 1 章。对于初学者，建议按本书顺序阅读；对于比较有经验的读者，可以选择感兴趣的话题直接阅读。

寻求帮助

1. 代码

在每一章，本书都会用具体的例子来说明，代码可以在 https://github.com/Ovilia/ThreeExample.js 找到。书中在介绍到相关代码时，也会给出链接。

2. 文档

当你知道应该查什么关键字的时候，查阅文档是最高效的。

Three.js 的官方文档可以在 http://threejs.org/docs 找到，但是由于 Three.js 版本更新很快（在本书的写作过程中，就经历了版本从 58 到 61 的变化，目前本书代码使用的版本是 59），使用的时候一定要注意代码的版本和文档的版本是否一致。有些文档是过时的，和代码是不对应的，而且这份文档也不完整，这时最好参考源代码进一步了解。但即便如此，文档对于我们了解 Three.js 还是能有不少帮助。

3. Google

你碰到的问题很可能别人也碰到过，因此，在提问之前记得 Google！

4. StackOverflow

如果你搜不到类似问题，那么在 StackOverflow 上提问吧！

5. 阅读源码

Three.js 的源码可以在 https://github.com/mrdoob/three.js/tree/master/build 找到。当怀疑文档和代码不一致时，一个很有效的办法是查阅源码。当然，读源码也不是让你逐行阅读，搜索关键字即可。

当这一切都不奏效时……

前几天，听学长说起在微软面试时的一道题：当你遇到一个没人知道的问题时怎么办？

面试官给出了一个不错的解答：问你周围的人，碰到这样的情况他怎么办。

有时候，你得不到直接的答案，但可以间接地询问如何获取答案。我想，作为程序员，我们或多或少都会遇到似乎没人能解决的问题，你可能没有意识到，但当你“不去管它，睡一觉醒来突然来了灵感”时，其实这同样是种解决方法。我们会像八仙过海一样解决各种各样

看似不可能的问题，那么，下次你觉得山重水复疑无路时，记得问问你的朋友是如何寻求帮助的。

致谢

本书的顺利完成离不开很多人的帮助和关心，首先要感谢的就是我的导师肖双九博士。除了教会我对本书有直接影响的计算机图形学课程，肖老师还教会我很多受益终身的学术知识和人生哲学，而且在平时生活中也对我非常关心照顾。

Three.js 技术是我在 Intel 实习阶段学习的，因此不得不提 Intel 大学合作经理颜历女士一路以来对我的关心。她有些像我的精神导师，在我困惑迷茫的时候，给了我很多中肯的建议。即使在我离开 Intel 后，我们也保持着联系，她也一直给予我关心和肯定，让我对自己选择的未来更有信心。

我还要特别感谢 5 位为本书作审核的同学，其中三位同学有丰富的 Three.js 开发经验，另外两名从初学者的角度为本书提出了建议。他们分别是：徐雪桥、单震宇、叶家彬；史鑫、王佳骏。其中，徐雪桥现就读于美国 Carnegie Mellon University，具有丰富的网页开发经验，擅长使用 JavaScript 制作动画以及数据可视化，曾参与包括 Three.js 在内的多项开源项目。其他几位是我在交大的同学，都是各自擅长领域内的牛人，非常感谢他们能够抽出很多时间参与本书的审核工作。

在此，也要感谢图灵社区提供了这样一个平台，使得本书得以面世。非常感谢图灵编辑董苗苗老师对我的悉心指导，为本书的顺利发布提供了非常重要的帮助。

此外，我最想感谢作为读者的你。这是我第一次写书，因此你对本书的支持将对我是一种莫大的鼓励！

最后，祝大家享受阅读本书的过程，创造出令人心动的应用！

第 1 章 概述

本章将介绍 WebGL 与 Three.js 的背景知识，如何下载、使用 Three.js。阅读完本章后，你将学会使用 Three.js 实现一个最简单的功能。

1.1 WebGL 与 Three.js

本节介绍 WebGL 与 Three.js 的相关概念，并通过两者实现同样功能的代码表现 Three.js 的简洁性。

1.1.1 什么是 WebGL

WebGL 是基于 OpenGL ES 2.0 的 Web 标准，可以通过 HTML5 Canvas 元素作为 DOM 接口访问。

听起来挺像回事儿的，但是这是什么意思呢？

如果你了解 OpenGL，那么我解释起来就比较轻松了。WebGL 可以看做是将 OpenGL ES（OpenGL for Embedded Systems，OpenGL 嵌入式版本，针对手机、游戏机等设备相对较轻量级的版本）移植到了网页平台，像 Chrome、Firefox 这些现代浏览器都实现了 WebGL 标准，使用 JavaScript 就可以用你熟悉的、类似 OpenGL 的代码编写了。

如果你不了解 OpenGL，那也没关系，因为正如 Three.js 不需要你了解 OpenGL 或 WebGL 一样，本书也不需要你预先知道这些知识。你可以把 WebGL 简单地认为是一种网络标准，定义了一些较底层的图形接口，至于究竟多底层，稍后我们和 Three.js 代码对比来看。本书不会过多涉及 WebGL 的相关知识，如果读者想学习的话，市场上有不少相关书籍可供参考。

现在，我们知道了 WebGL 是一个底层的标准，在这些标准被定义之后，Chrome、Firefox 之类的浏览器实现了这些标准。然后，程序员就能通过 JavaScript 代码，在网页上实现三维图形的渲染了。如果这对你来说还是太抽象，别着急，稍后我们会用具体的例子来说明。

1.1.2 什么是 Three.js

Three.js 是一个 3D JavaScript 库。

如此简介的描述背后，是作者对其强大功能的自信。

那么，Three.js 究竟能用来干什么呢？

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。

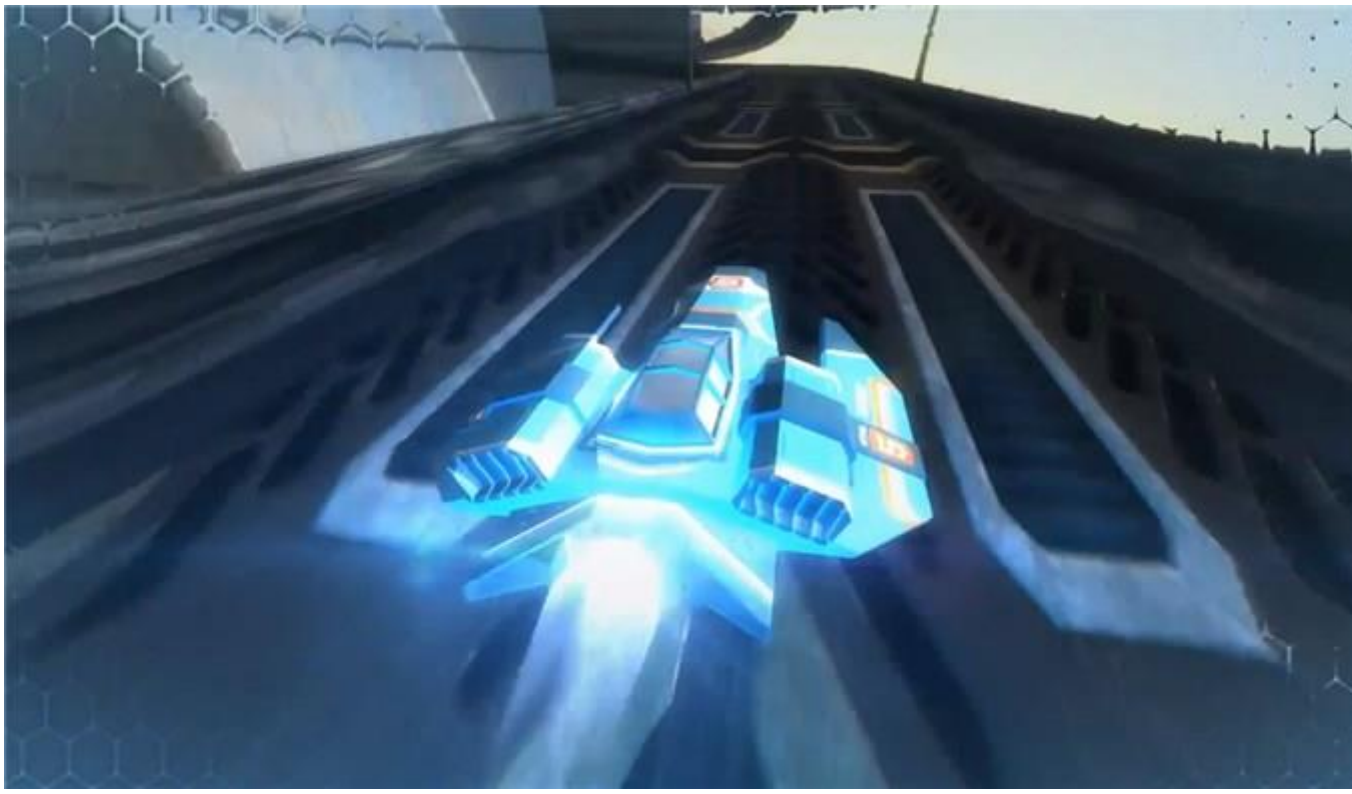
当然，除了 WebGL 之外，Three.js 还提供了基于 Canvas、SVG 标签的渲染器，但由于通常 WebGL 能够实现更灵活的渲染效果，所以本书主要针对基于 WebGL 渲染器进行说明。

应用实例

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：

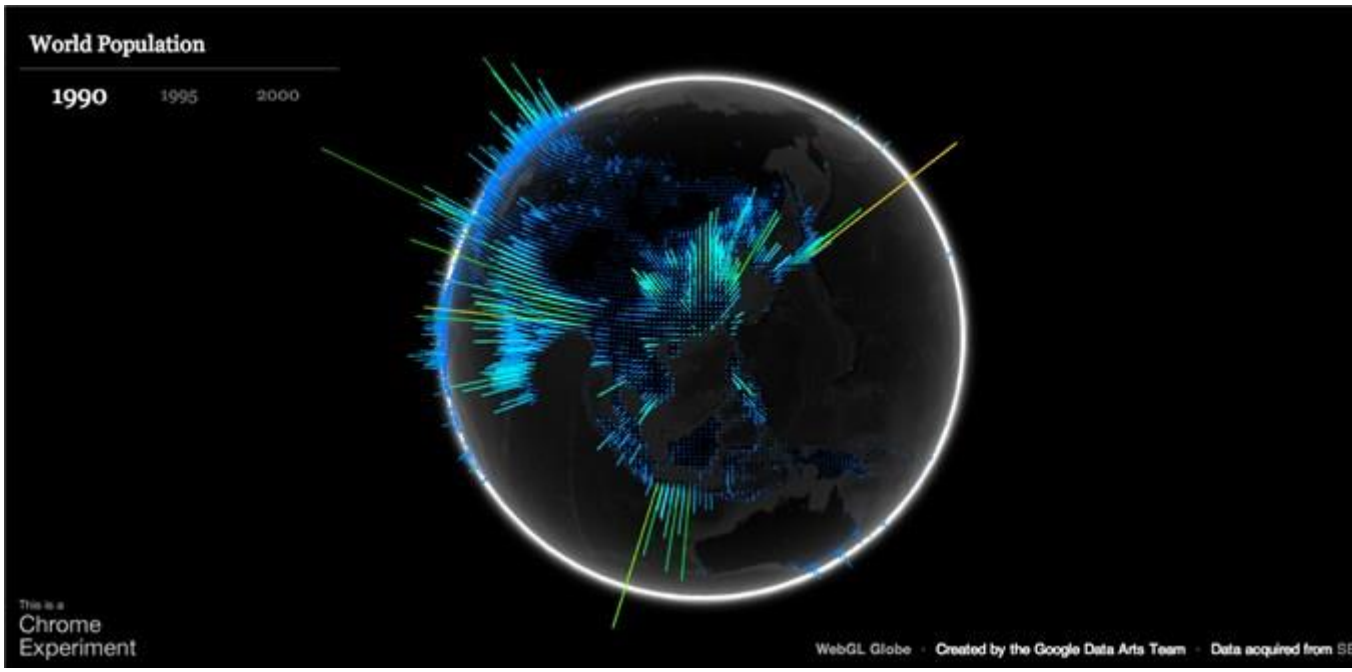
[+]查看原图

精美绝伦的游戏效果：



[+]查看原图

或是绚丽的数据可视化效果：



[+]查看原图

更多应用可以在 Three.js 官网查看。

Three.js 作者

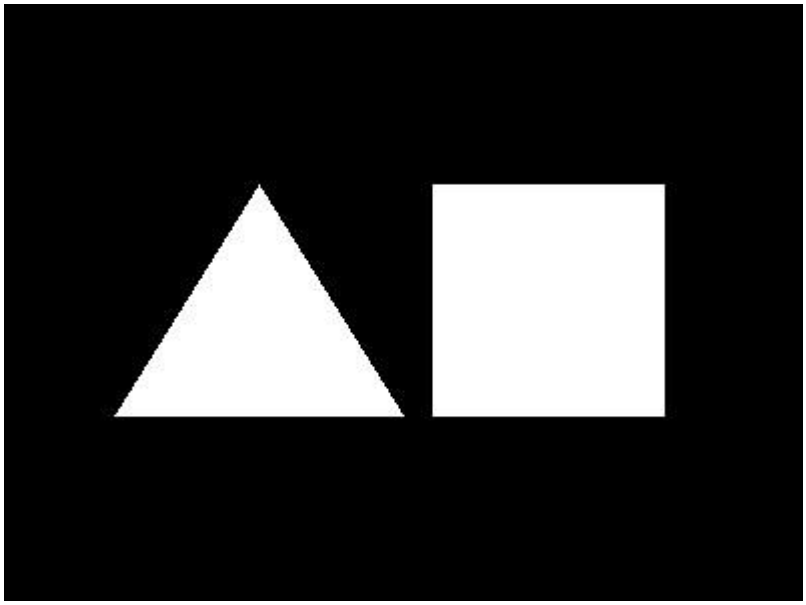
Mr. doob 是 Three.js 项目发起人和主要贡献者之一，但由于 Three.js 是 Github 上一个开源项目，因此有非常多的贡献者，甚至有一天，你也可以在贡献者列表中看到自己的名字。

使用协议

Three.js 是基于 MIT 协议进行发布的，因此使用和发布都非常自由。

1.1.3 WebGL vs. Three.js

为了比较说明 Three.js 能大大简化 WebGL 的开发，我们使用最简单的例子进行比较：渲染黑色背景下的白色正方形和三角形。效果如图：



Three.js 需要 30 行左右的代码：

例 1.1.1

```
var renderer = new THREE.WebGLRenderer({

    canvas: document.getElementById('mainCanvas')

});

renderer.setClearColor(0x000000); // black



var scene = new THREE.Scene();



var camera = new THREE.PerspectiveCamera(45, 4 / 3, 1, 1000);

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

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

scene.add(camera);



var material = new THREE.MeshBasicMaterial({

       color: 0xffffff // white

});

// plane

var planeGeo = new THREE.PlaneGeometry(1.5, 1.5);

var plane = new THREE.Mesh(planeGeo, material);

plane.position.x = 1;

scene.add(plane);



// triangle

var triGeo = new THREE.Geometry();

triGeo.vertices = [new THREE.Vector3(0, -0.8, 0),
```

```
        new THREE.Vector3(-2, -0.8, 0), new THREE.Vector3(-1, 0.8, 0)];

triGeo.faces.push(new THREE.Face3(0, 2, 1));

var triangle = new THREE.Mesh(triGeo, material);

scene.add(triangle);



renderer.render(scene, camera);
```

如果接触过图形学知识，这里的代码应该很容易理解，如果不懂也没关系，接下来几章会进行详细说明。所以在此就不花费篇章解释这几行代码了。

以下摘录实现相同功能的 WebGL 代码，来自博客 http://learningwebgl.com/blog/?p=28。

```
var gl;

function initGL(canvas) {

    try {

        gl = canvas.getContext("experimental-webgl");

        gl.viewportWidth = canvas.width;

        gl.viewportHeight = canvas.height;

    } catch (e) {

    }

    if (!gl) {

        alert("Could not initialise WebGL, sorry :-(");

    }

}



function getShader(gl, id) {
```

```
var shaderScript = document.getElementById(id);

if (!shaderScript) {

    return null;

}



var str = "";

var k = shaderScript.firstChild;

while (k) {

    if (k.nodeType == 3) {

        str += k.textContent;

    }

    k = k.nextSibling;

}



var shader;

if (shaderScript.type == "x-shader/x-fragment") {

    shader = gl.createShader(gl.FRAGMENT_SHADER);

} else if (shaderScript.type == "x-shader/x-vertex") {

    shader = gl.createShader(gl.VERTEX_SHADER);

} else {

    return null;

}



gl.shaderSource(shader, str);
```

```
    gl.compileShader(shader);

    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        alert(gl.getShaderInfoLog(shader));
        return null;
    }

    return shader;
}

var shaderProgram;

function initShaders() {
    var fragmentShader = getShader(gl, "shader-fs");
    var vertexShader = getShader(gl, "shader-vs");

    shaderProgram = gl.createProgram();
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);

    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
        alert("Could not initialise shaders");
    }
```

```
    gl.useProgram(shaderProgram);

    shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram, "aVe
rtexPosition");
    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

    shaderProgram.pMatrixUniform = gl.getUniformLocation(shaderProgram, "uPMatrix");
    shaderProgram.mvMatrixUniform = gl.getUniformLocation(shaderProgram, "uMVMatrix
");
}

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

function setMatrixUniforms() {
    gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false, pMatrix);
    gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false, mvMatrix);
}

var triangleVertexPositionBuffer;
var squareVertexPositionBuffer;

function initBuffers() {
    triangleVertexPositionBuffer = gl.createBuffer();
```

```
    gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer);
    var vertices = [
         0.0,  1.0,  0.0,
        -1.0, -1.0,  0.0,
         1.0, -1.0,  0.0
    ];
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
    triangleVertexPositionBuffer.itemSize = 3;
    triangleVertexPositionBuffer.numItems = 3;

    squareVertexPositionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexPositionBuffer);
    vertices = [
         1.0,  1.0,  0.0,
        -1.0,  1.0,  0.0,
         1.0, -1.0,  0.0,
        -1.0, -1.0,  0.0
    ];
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
    squareVertexPositionBuffer.itemSize = 3;
    squareVertexPositionBuffer.numItems = 4;
}

function drawScene() {
```

```
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    mat4.perspective(45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0, pMatrix);

    mat4.identity(mvMatrix);

    mat4.translate(mvMatrix, [-1.5, 0.0, -7.0]);
    gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer);
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute, triangleVertexPosi
tionBuffer.itemSize, gl.FLOAT, false, 0, 0);
    setMatrixUniforms();
    gl.drawArrays(gl.TRIANGLES, 0, triangleVertexPositionBuffer.numItems);

    mat4.translate(mvMatrix, [3.0, 0.0, 0.0]);
    gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexPositionBuffer);
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute, squareVertexPositi
onBuffer.itemSize, gl.FLOAT, false, 0, 0);
    setMatrixUniforms();
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, squareVertexPositionBuffer.numItems);
}

function webGLStart() {
    var canvas = document.getElementById("lesson01-canvas");
    initGL(canvas);
```

```
    initShaders();

    initBuffers();



    gl.clearColor(0.0, 0.0, 0.0, 1.0);

    gl.enable(gl.DEPTH_TEST);



    drawScene();

}
```

从上面的代码我们不难发现，使用原生 WebGL 接口实现同样功能需要 5 倍多的代码量，而且很多代码对于没有图形学基础的程序员是很难看懂的。由这个例子我们可以看出，使用 Three.js 开发要比 WebGL 更快更高效。尤其对图形学知识不熟悉的程序员而言，使用 Three.js 能够降低学习成本，提高三维图形程序开发的效率。

1.2 开始使用 Three.js

本节介绍如何下载使用 Three.js 创建你的第一个程序。

1.2.1 准备工作

开发环境

Three.js 是一个 JavaScript 库，所以，你可以使用平时开发 JavaScript 应用的环境开发 Three.js 应用。如果你没什么偏好的话，我会推荐 Komodo IDE。

调试建议使用 Chrome 或者 Firefox 浏览器。如果你使用的是 Firefox，那么 Firebug 会是你必不可少的插件；如果你使用的是 Chrome，那么直接使用控制台调试即可。这些和 JavaScript 的调试是相同的，因此本书不作进一步展开。

下载

首先，我们需要在 Github 下载 Three.js 的代码。

在 https://github.com/mrdoob/three.js/tree/master/build 可以看到 `three.js` 和 `three.min.js` 两个文件，前者是没有经过代码压缩的，因此适用于调试阶段；后者是经过代码压缩的，调试起来会不太方便，但文件较小，适用于最终的发布版。保存一个文件到本地，这里我们可以选择 three.js。

引用

在使用 Three.js 之前，我们需要在 HTML 文件中引用该文件：

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

然后就能通过全局变量 `THREE` 访问到所有属性和方法了。

1.2.2 Hello, world!

接下来，我们终于要真正使用 Three.js 了！

首先，在 HTML 的 `<head>` 部分，需要声明外部文件 `three.js`。

```
<head>

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

</head>
```

WebGL 的渲染是需要 HTML5 Canvas 元素的，你可以手动在 HTML 的 `<body>` 部分中定义 Canvas 元素，或者让 Three.js 帮你生成。这两种选择一般没有多大差别，我们在此手动在 HTML 中定义：

```
<body onload="init()">

    <canvas id="mainCanvas" width="400px" height="300px" ></canvas>

</body>
```

在 JavaScript 代码中定义一个 `init` 函数，在 HTML 加载完后执行：

```
function init() {

    // ...

}
```

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。这些话题将在后面几章中进一步展开，这里我们将介绍如何快速地使用这些东西。

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 `mainCanvas` 的 Canvas 元素，那么 Renderer 可以这样写：

```
var renderer = new THREE.WebGLRenderer({

    canvas: document.getElementById('mainCanvas')

});
```

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);
```

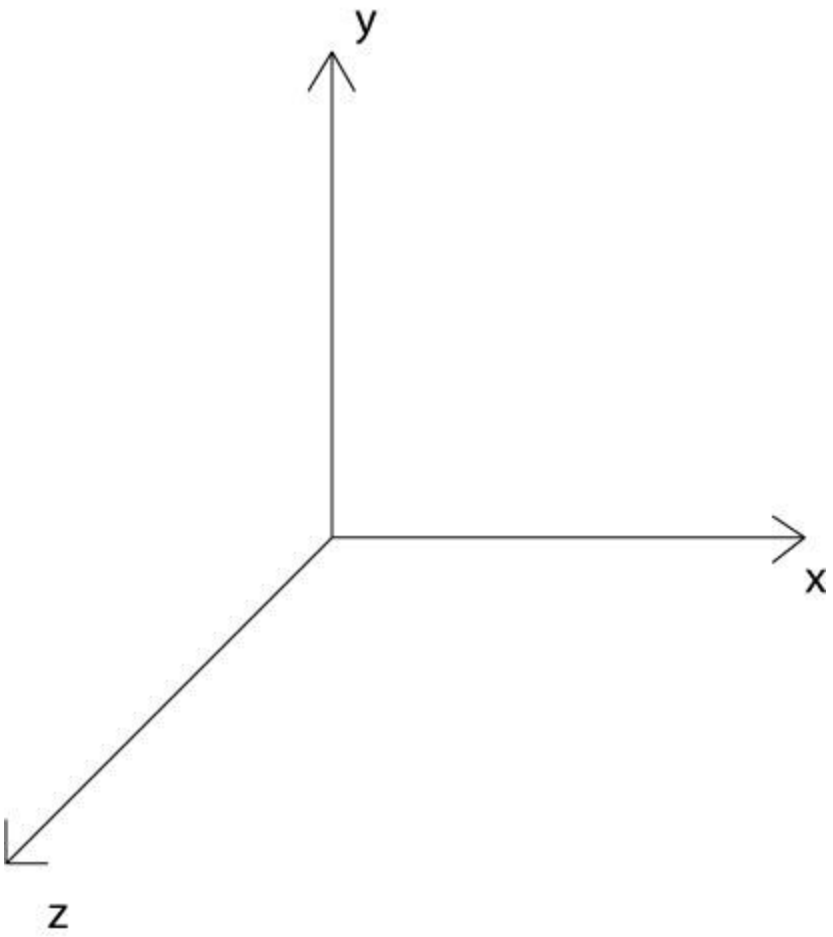
场景（Scene）

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，场景来没有很复杂的操作，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();
```

照相机（Camera）

在介绍照相机设置前，我们先来简单了解下坐标系。WebGL 和 Three.js 使用的坐标系是右手坐标系，看起来就是这样的：



这里，我们定义了一直透视投影的照相机，具体原理将在下一章中展开。

```
var camera = new THREE.PerspectiveCamera(45, 4 / 3, 1, 1000);

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

scene.add(camera);
```

值得注意的是，照相机也需要被添加到场景中。

长方体

我们要创建一个 x、y、z 方向长度分别为 `1`、`2`、`3` 的长方体，并将其设置为红色。

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

        new THREE.MeshBasicMaterial({
```

```
            color: 0xff0000

        })

);

scene.add(cube);
```

这段代码也是比较容易理解的，虽然你现在可能还不知道 MeshBasicMaterial 是什么，但是大致可以猜测出这是一种材质，可以用来设置物体的颜色。还是要提醒下，一定要记得把创建好的长方体添加到场景中。

那么这里长度为 `1` 的单位是什么呢？这里的长度是在物体坐标系中的，其单位与屏幕分辨率等无关，简单地说，它就是一个虚拟空间的坐标系，`1` 代表多少并没有实际的意义，而重要的是相对长度。

渲染

在定义了场景中的物体，设置好的照相机之后，渲染器就知道如何渲染出二维的结果了。这时候，我们只需要调用渲染器的渲染函数，就能使其渲染一次了。

```
renderer.render(scene, camera);
```

完整代码

最终，这是我们得到的完整的 `init` 函数：

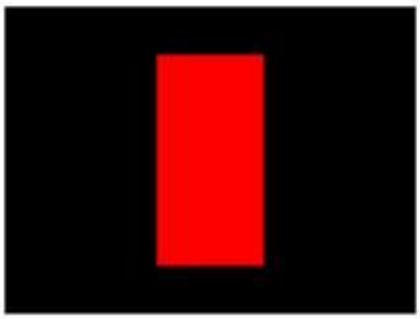
例 1.2.1 `<= 点此查看完整源代码，下略`

```
function init() {

    // renderer

    var renderer = new THREE.WebGLRenderer({
```

```
        canvas: document.getElementById('mainCanvas')
    });
    renderer.setClearColor(0x000000); // black


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


    // camera
    var camera = new THREE.PerspectiveCamera(45, 4 / 3, 1, 1000);
    camera.position.set(0, 0, 5);
    scene.add(camera);


    // a cube in the scene
    var cube = new THREE.Mesh(new THREE.CubeGeometry(1, 2, 3),
            new THREE.MeshBasicMaterial({
                color: 0xff0000
            })
    );
    scene.add(cube);


    // render
    renderer.render(scene, camera);
}
```

渲染的效果

是：



[+]查看原图

出问题了？

第一件事就是打开浏览器的控制台，查看报错信息。如果错误信息是 `ReferenceError: THREE is not defined`，那么就是页面没有成功加载 Three.js 库，很有可能的原因是文件的路径写错了，一定要小心检查。

如果存在其他问题，请参见前言中“寻求帮助”部分。

1.3 Three.js 功能概览

上一节，我们了解了如何建立一个简单的 Three.js 应用，可能有读者会对各种概念表示困惑，那么下面就让我们看下 Three.js 官网文档中的一些重要的对象，在你需要寻求帮助时，就能够知道关键词是什么。

```
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，在最后一章作介绍）

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
```

我们看到，Three.js 功能是十分丰富的，一时间想全部掌握有些困难。本书将从 Three.js 程序常用的功能着手，带领大家入门 Three.js。

在接下来的章节中，我们将会先详细介绍照相机、几何形状、材质、物体等入门级知识；然后介绍使用动画、模型导入、加入光照等功能；最后，对于学有余力的读者，我们将介绍着色器，用于更高级的图形渲染。

第 2 章 照相机

本章将介绍照相机的概念，以及如何使用 Three.js 设置相应的参数。对于熟悉图形学照相机概念的读者，可以直接阅读 2.3 和 2.4 节。

2.1 什么是照相机

什么是照相机？这个问题似乎太简单了，用来拍照的机器。咔嚓！

可是，在图形学中照相机的概念并非如此。

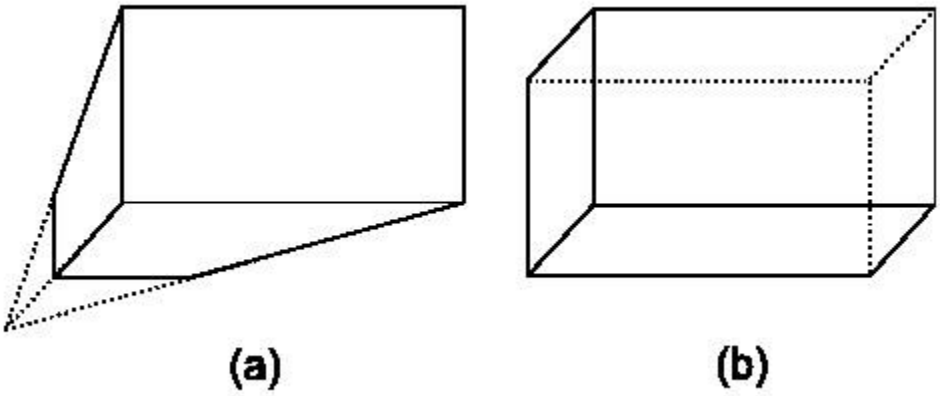
我们使用 Three.js 创建的场景是三维的，而通常情况下显示屏是二维的，那么三维的场景如何显示到二维的显示屏上呢？照相机就是这样一个抽象，它定义了三维空间到二维屏幕的投影方式，用“照相机”这样一个类比，可以使我们直观地理解这一投影方式。

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。我们需要为自己的程序选择合适的照相机。这两者分别是什么，以及两者有何差异，我们将在下节中作介绍。

2.2 正交投影 vs 透视投影

举个简单的例子来说明正交投影与透视投影照相机的区别。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)）。



(a)透视投影，(b)正交投影

那么，你的程序需要正交投影还是透视投影的照相机呢？

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。

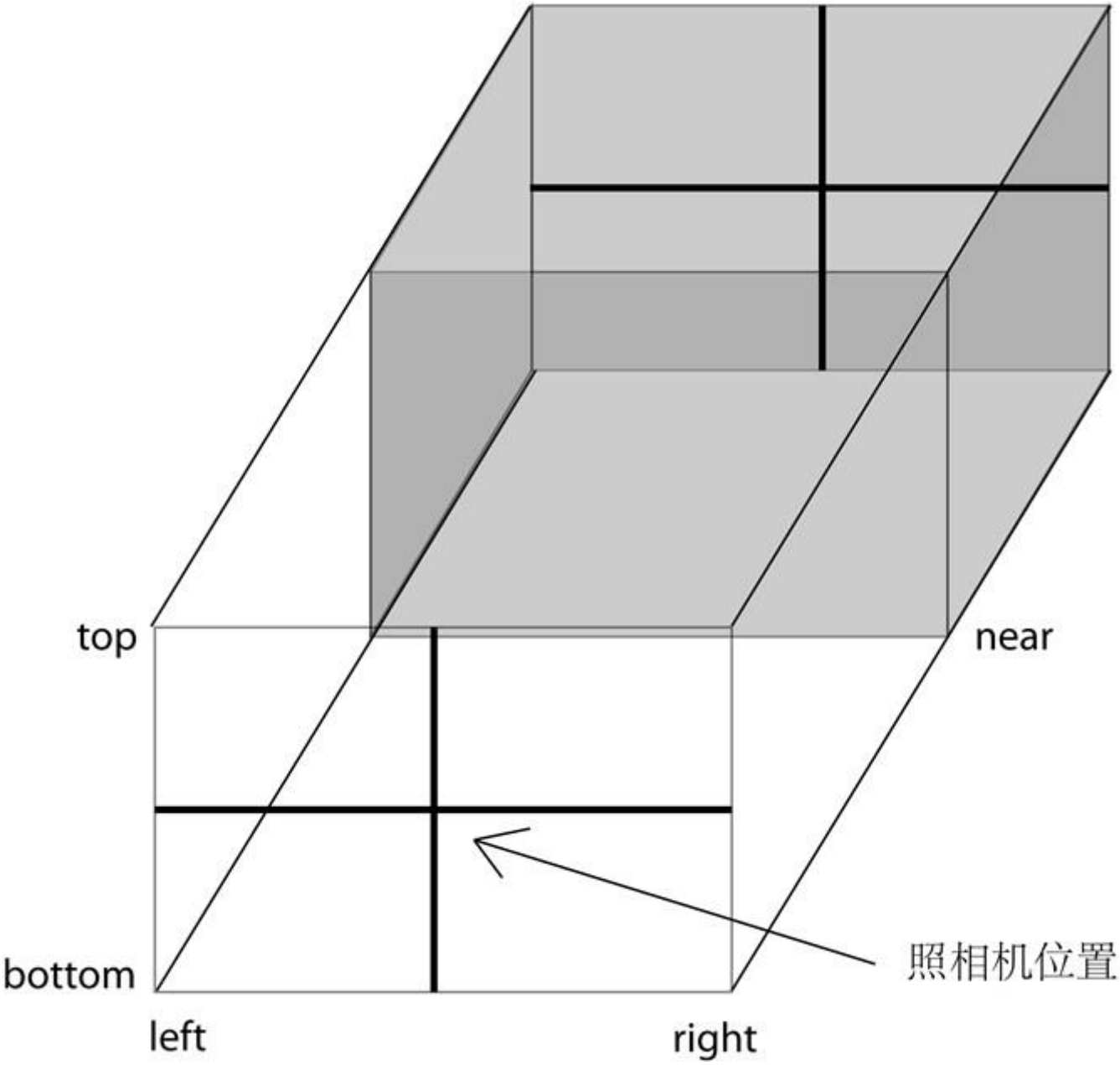
2.3 正交投影照相机

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，这两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。



[+]查看原图

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。

实例说明

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 2.3.1

基本设置

设置照相机：

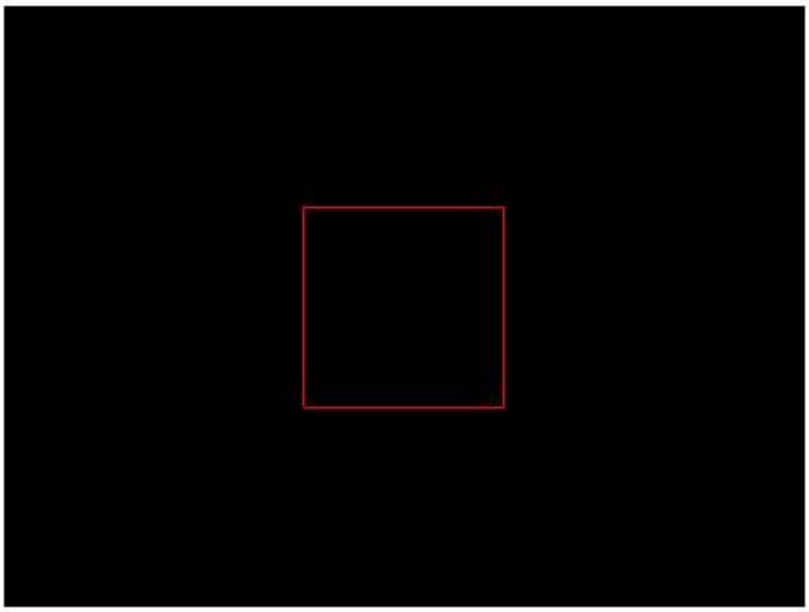
```
var camera = new THREE.OrthographicCamera(-2, 2, 1.5, -1.5, 1, 10);

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：



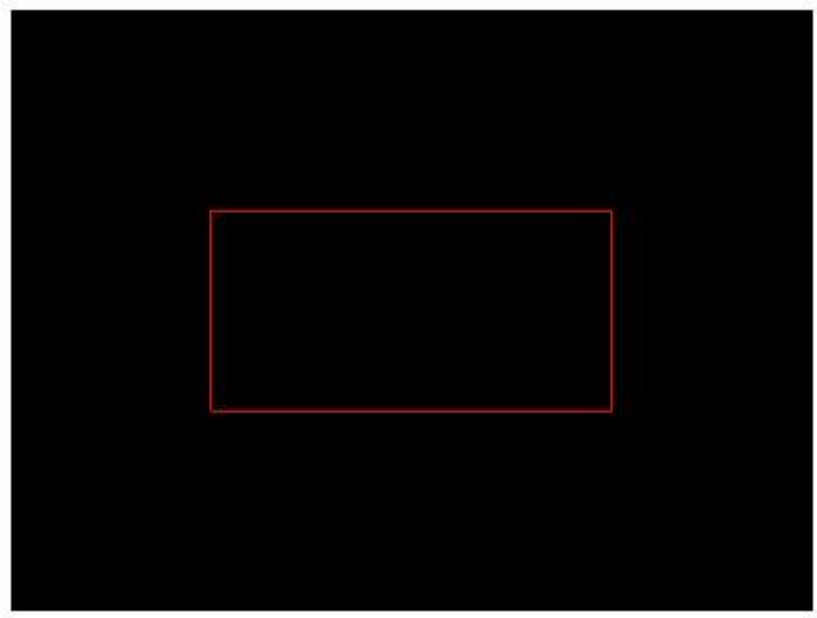
我们看到正交投影的结果是一个正方形，后面的边与前面完全重合了，这也就是正交投影与透视投影的区别所在。

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：

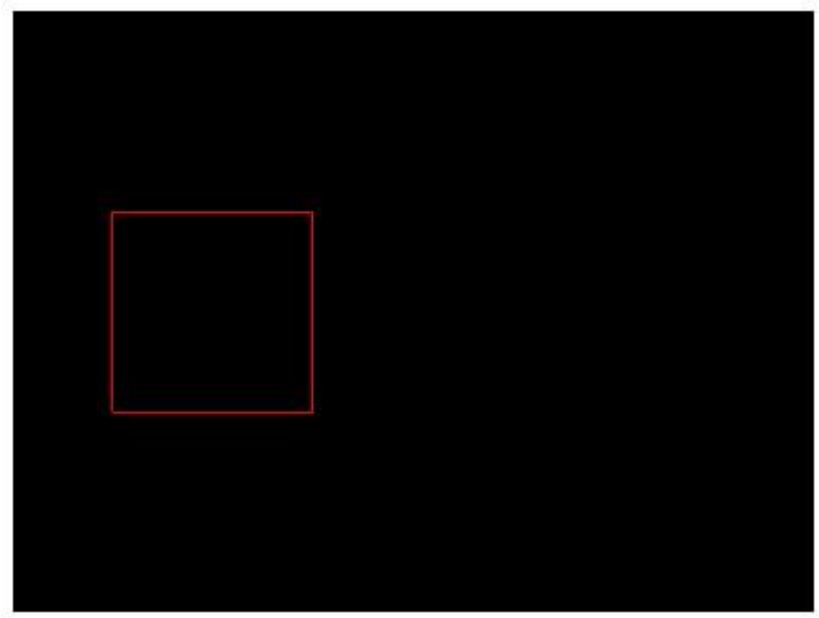


照相机位置

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：

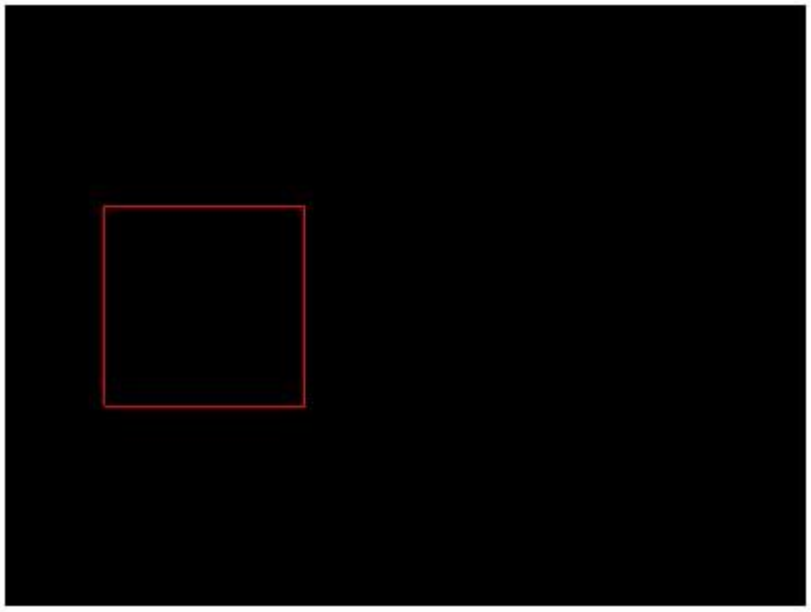


仔细想一下的话，这也不难理解。就好比你人往右站了，看起来物体就相对往左移动了。

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。

换个角度看世界

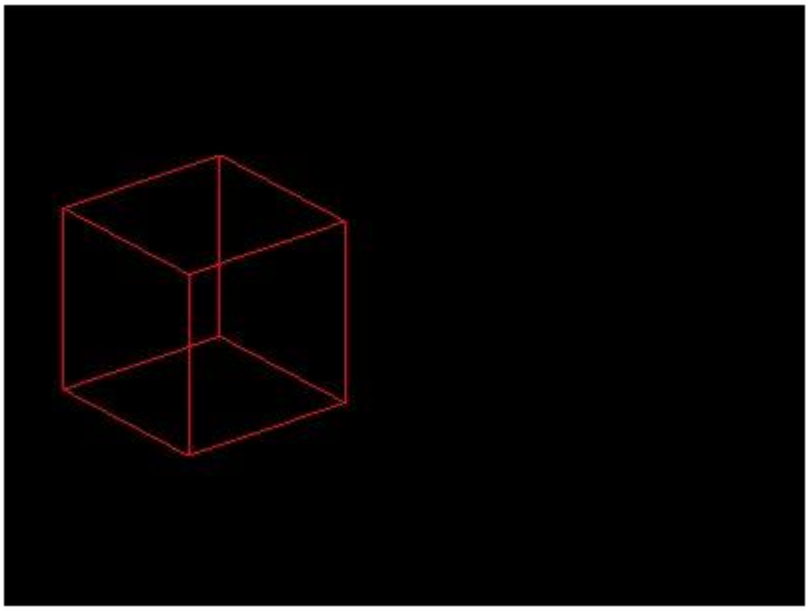
到现在为止，我们使用照相机都是沿 z 轴负方向观察的，因此看到的都是一个正方形。现在，我们想尝试一下仰望这个正方体。我们已经学会设置照相机的位置，不妨将其设置在 `(4, -3, 5)` 处：

```
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！

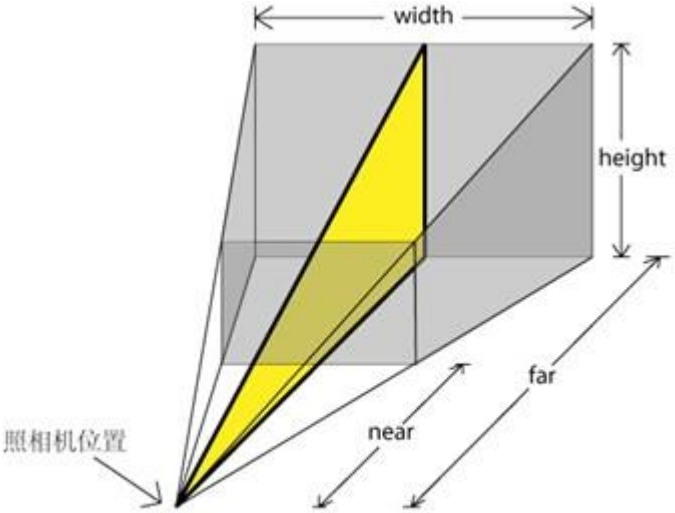
2.4 透视投影照相机

参数介绍

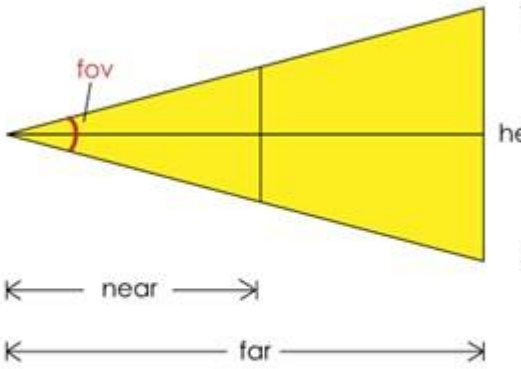
透视投影照相机（Perspective Camera）的构造函数是：

```
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



侧视图

[+]查看原图

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`。

实例说明

下面，我们从一个最简单的例子学习设置透视投影照相机。

基本设置

例 2.4.1

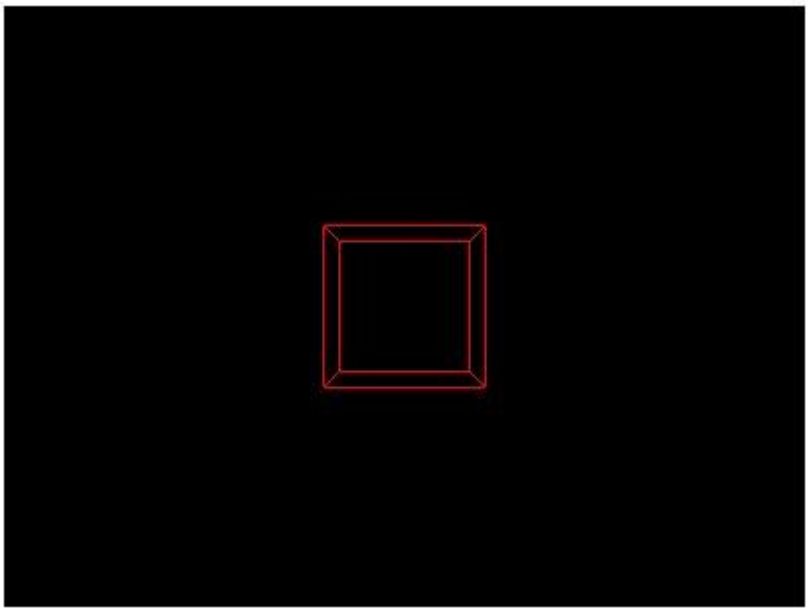
设置透视投影照相机，这里 Canvas 长 `400px`，宽 `300px`，所以 aspect 设为 `400 / 300`：

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

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

scene.add(camera);
```

和**例 2.3.1** 一样，设置一个在原点处的边长为 1 的正方体：

```
// a cube in the scene

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

       new THREE.MeshBasicMaterial({

           color: 0xff0000,

           wireframe: true

       })

);

scene.add(cube);
```

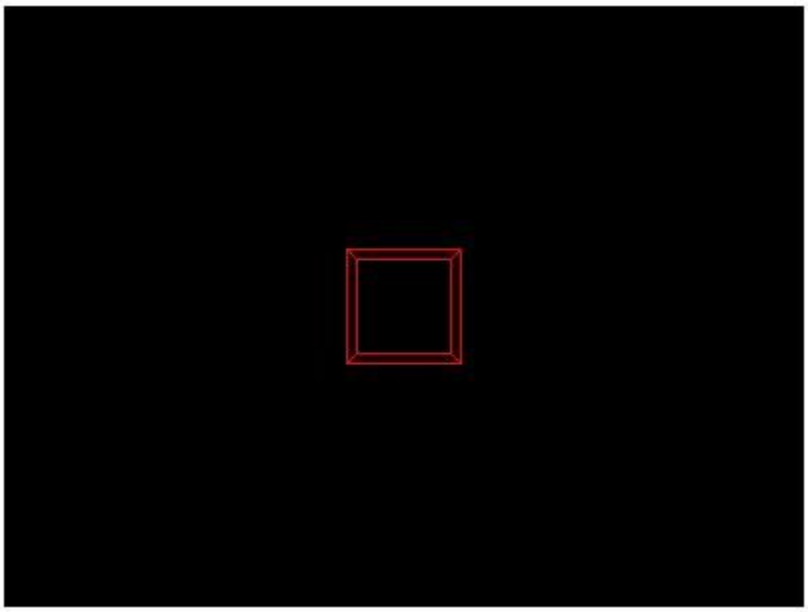
得到的结果是：



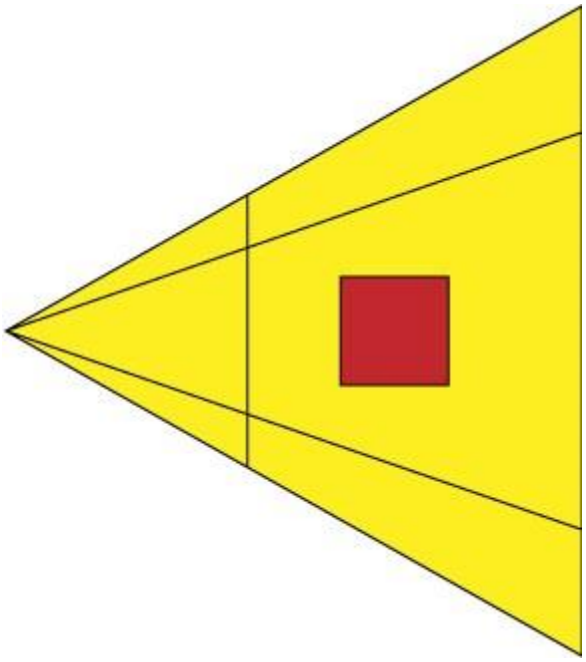
对比**例 2.3.1** 正方形的效果，透视投影可以看到全部的 12 条边，而且有近大远小的效果，这也就是与正交投影的区别。

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。这一效果类似 2.3 节，此处不再重复说明。

3.1 基本几何形状

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：

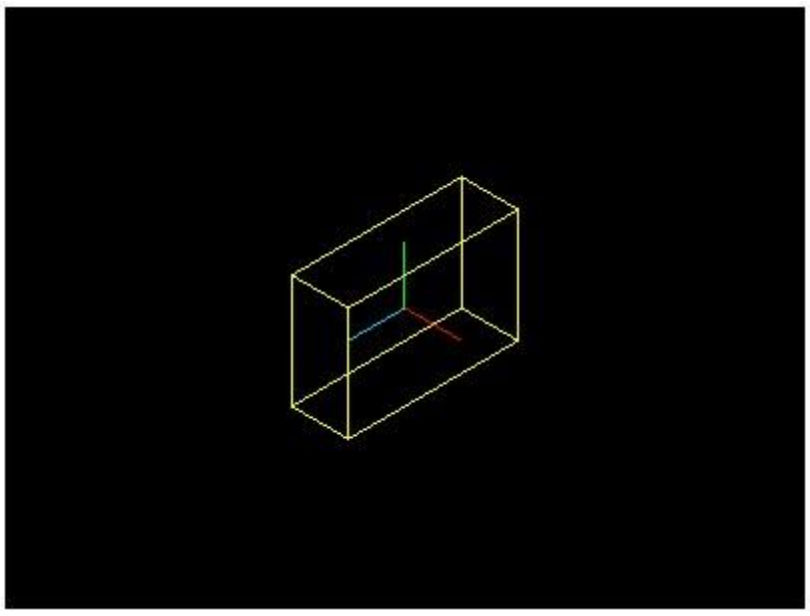
```
THREE.CubeGeometry(width, height, depth, widthSegments, heightSegments, depthSegments)
```

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。

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。

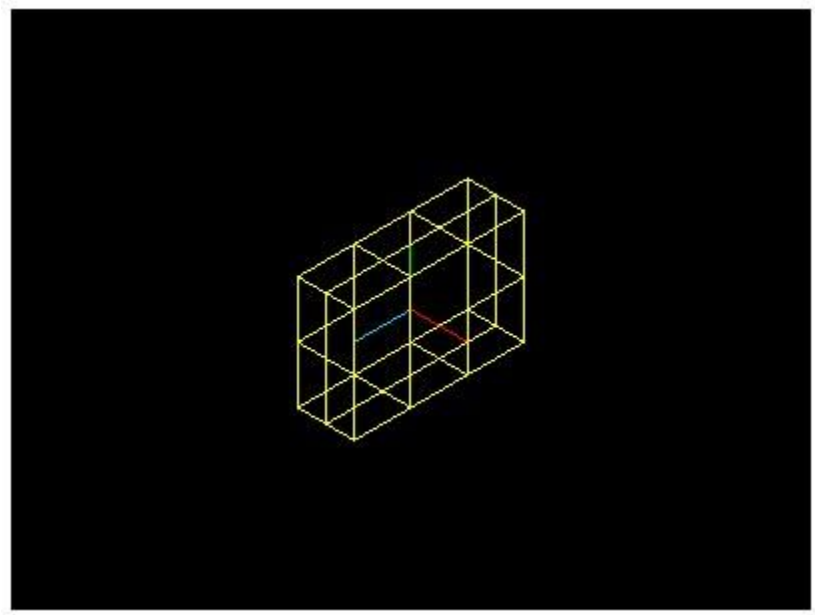
为了更好地表现参数效果，我们在场景中用长度为`1`的红、绿、蓝线段分别表示 x、y、z 三个轴。在设置材质，并添加到场景之后（具体方法参见第 4 章及第 5 章）的效果是：



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。

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。

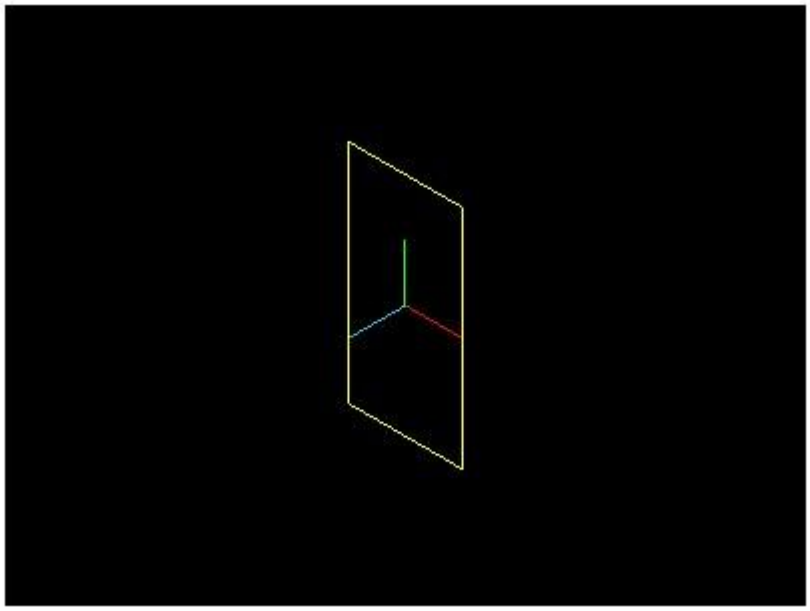
平面

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：



如果需要创建的平面在 x 轴和 z 轴所在的平面内，可以通过物体的旋转来实现，具体的做法将在 5.2 节作介绍。

球体

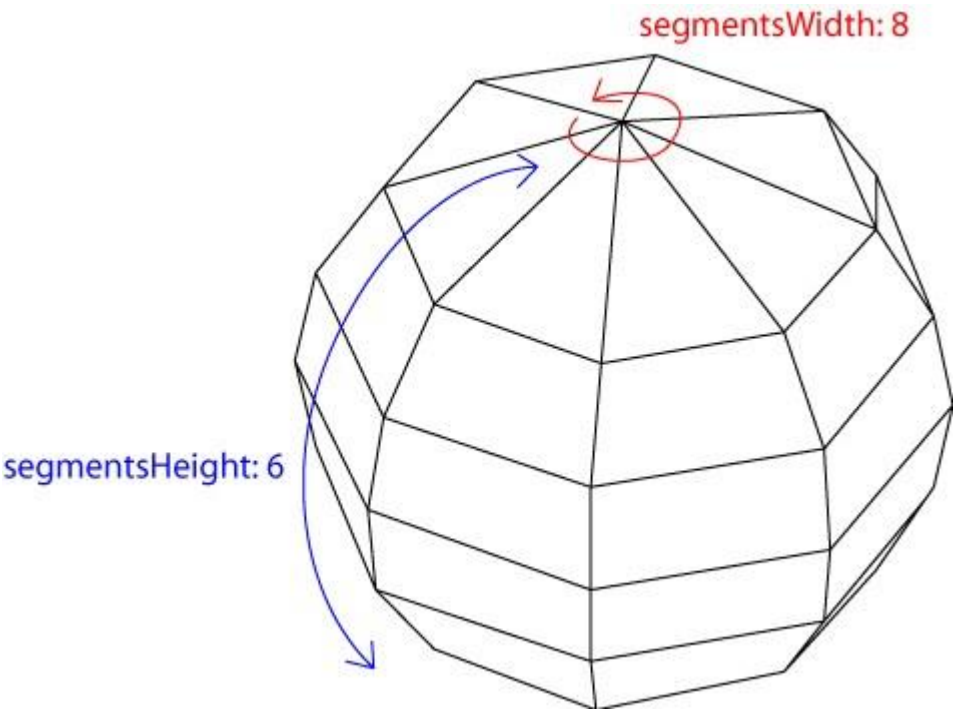
球体（SphereGeometry）的构造函数是：

```
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。

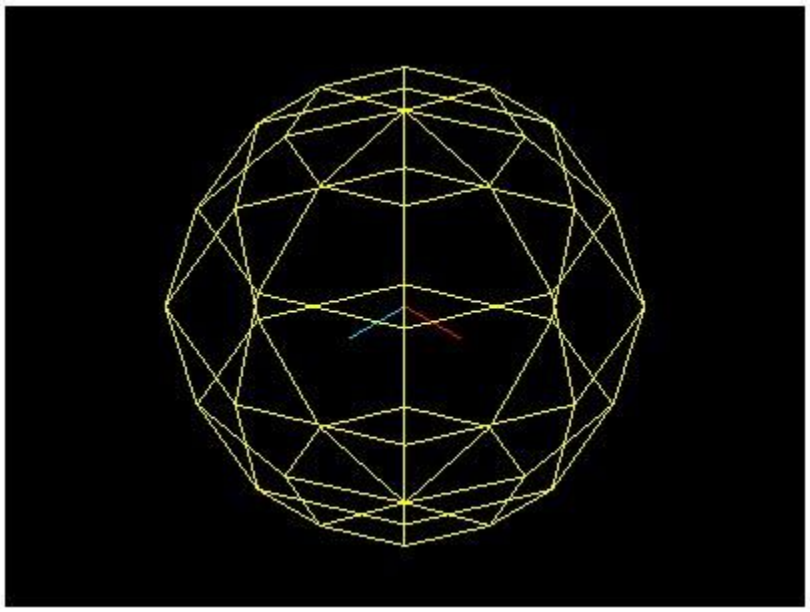
分段

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。

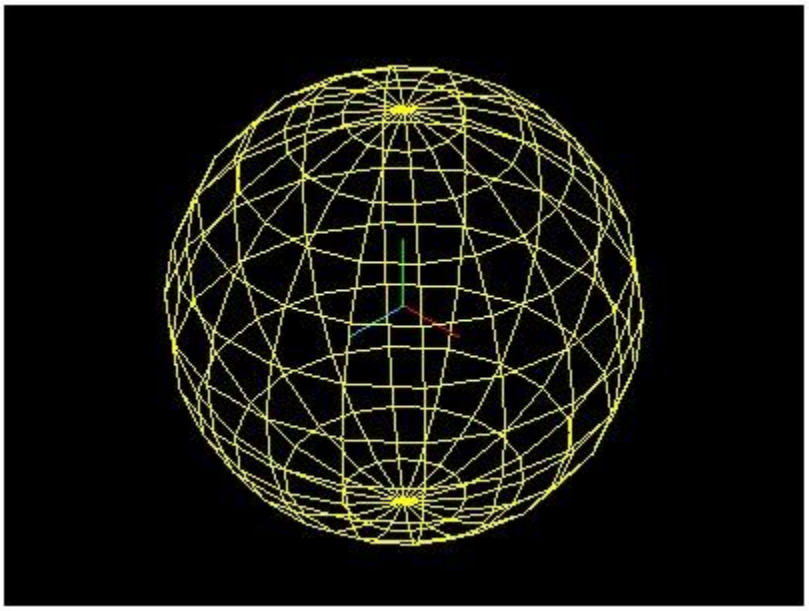


`segmentsWidth` 相当于经度被切成了几瓣，而 `segmentsHeight` 相当于纬度被切成了几层。因为在图形底层的实现中，并没有曲线的概念，曲线都是由多个折线近似构成的。对于球体而言，当这两个值较大的时候，形成的多面体就可以近似看做是球体了。

`new THREE.SphereGeometry(3, 8, 6)` 的效果：

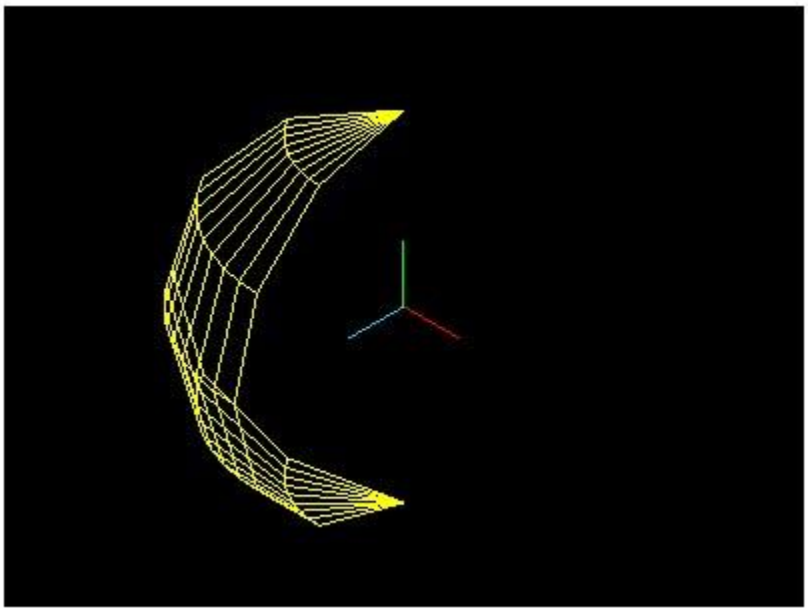


`new THREE.SphereGeometry(3, 18, 12)`的效果：



经度弧度

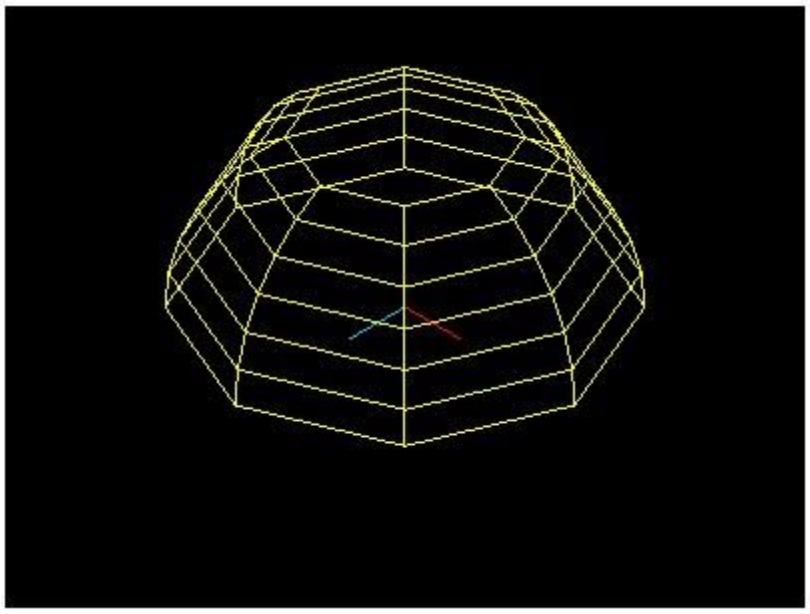
`new THREE.SphereGeometry(3, 8, 6, Math.PI / 6, Math.PI / 3)`表示起始经度为`Math.PI / 6`，经度跨度为`Math.PI / 3`。效果如下：



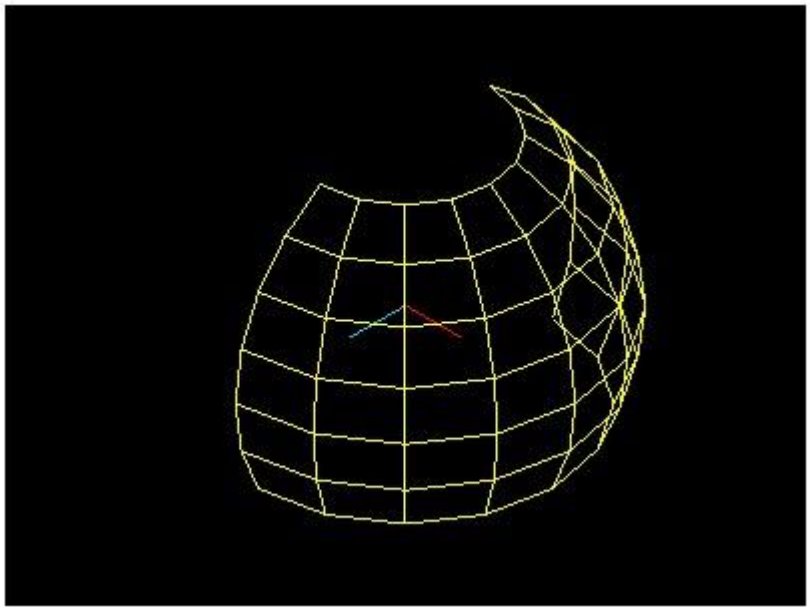
需要注意的是，这里 `segmentsWidth` 为 `8` 意味着对于经度从 `Math.PI / 6` 跨过 `Math.PI / 3` 的区域内划分为 `8` 块，而不是整个球体的经度划分成 `8` 块后再判断在此经度范围内的部分。

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：

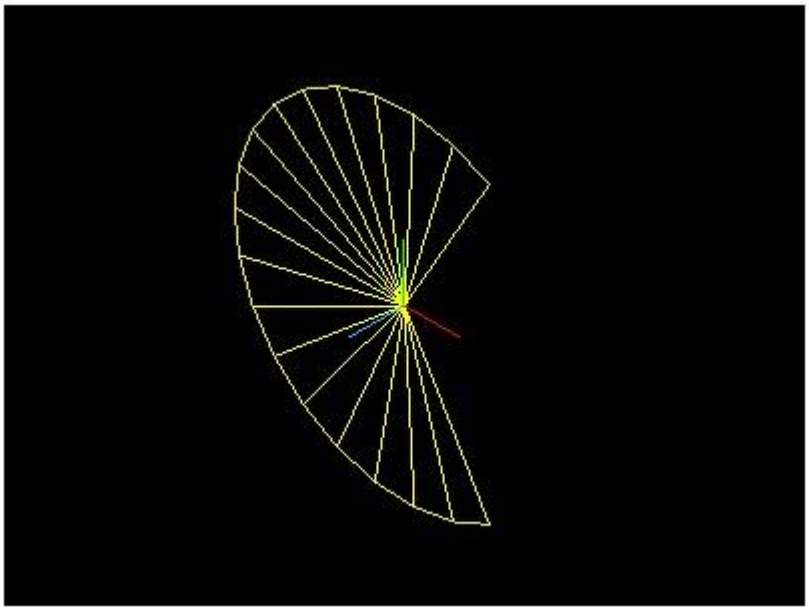


圆形

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：



圆柱体

圆柱体（CylinderGeometry）的构造函数是：

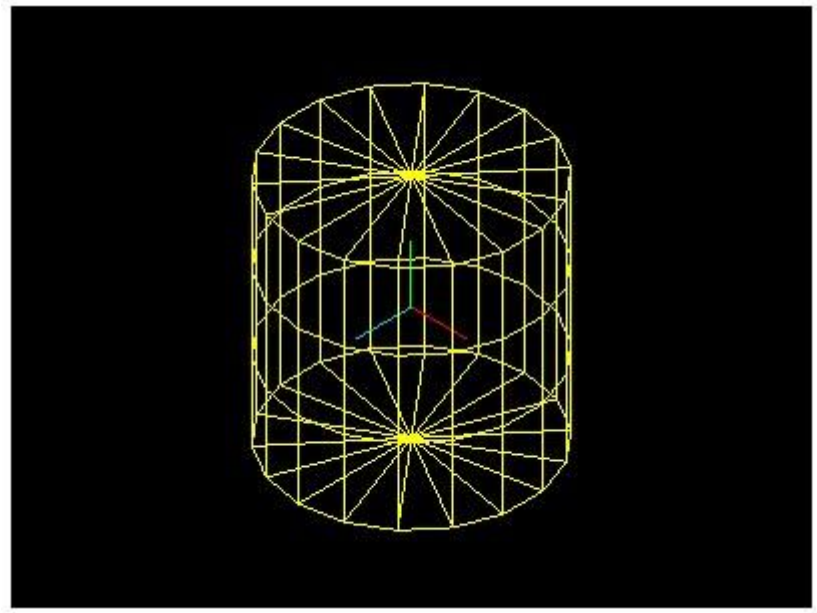
```
THREE.CylinderGeometry(radiusTop, radiusBottom, height, radiusSegments, heightSegments, openEnded)
```

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。

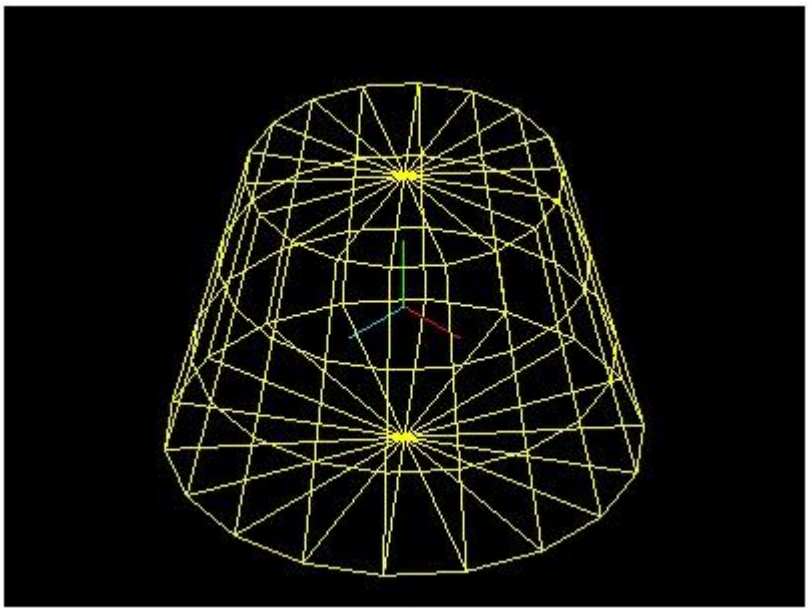
标准圆柱体

`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：



圆台

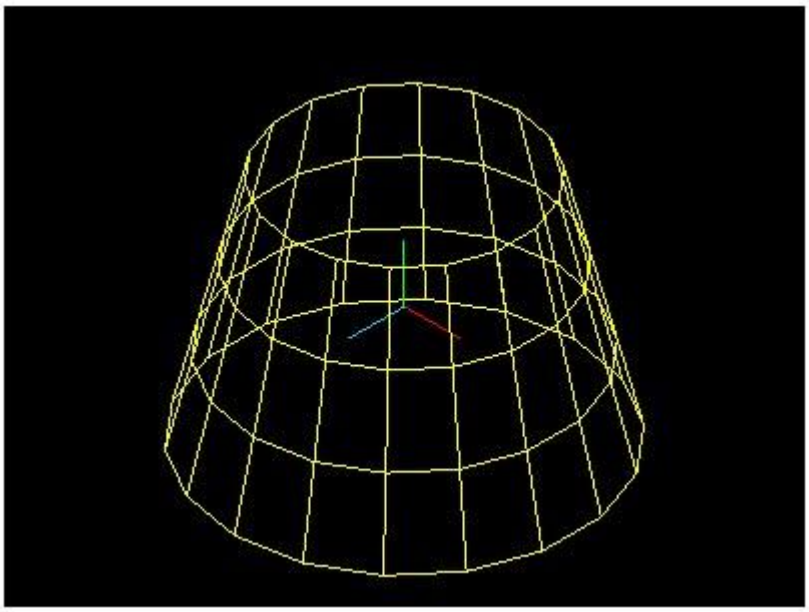
将底面半径设为 `3`创建一个圆台：`new THREE.CylinderGeometry(2, 3, 4, 18, 3)`，效果如下：



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：



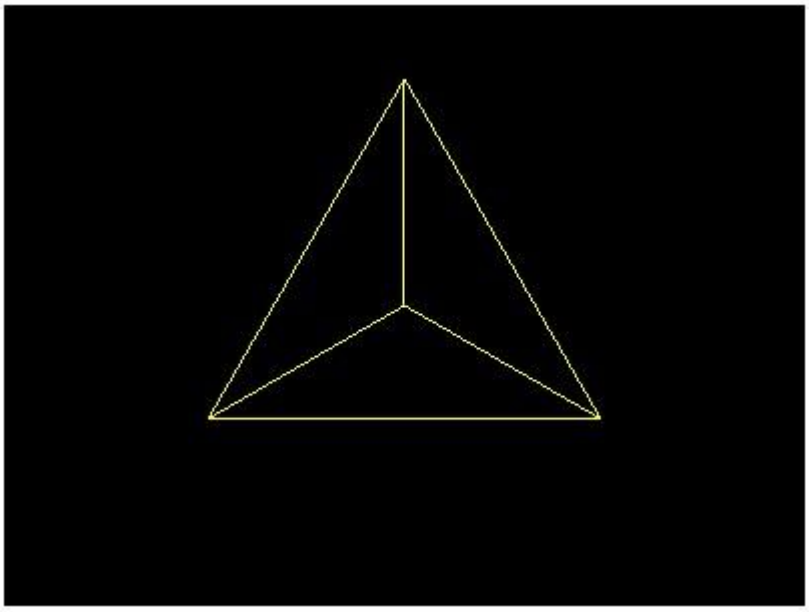
正四面体、正八面体、正二十面体

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：

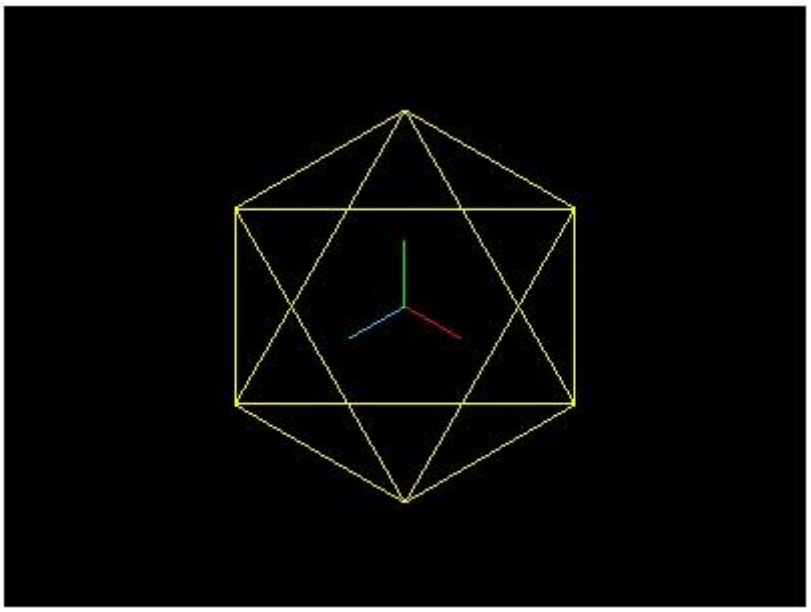
```
THREE.TetrahedronGeometry(radius, detail)

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。

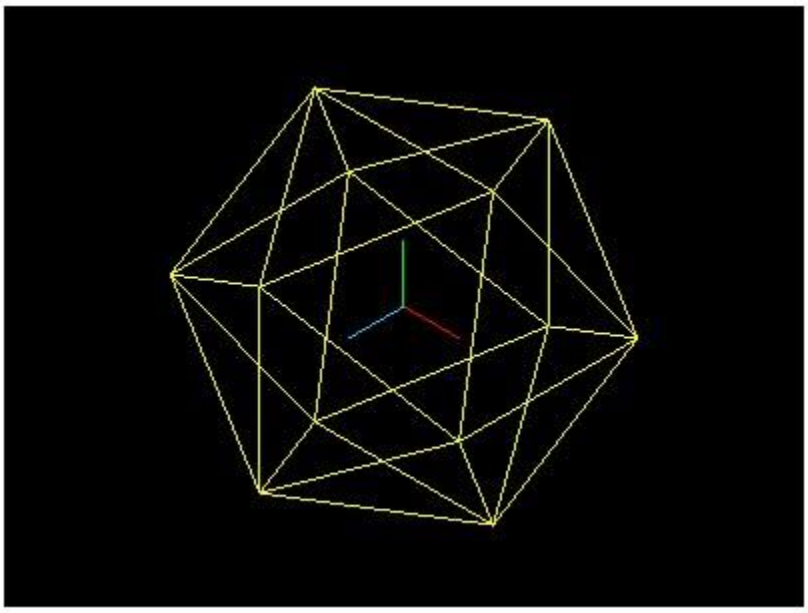
`new THREE.TetrahedronGeometry(3)` 创建一个半径为 `3` 的正四面体：



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



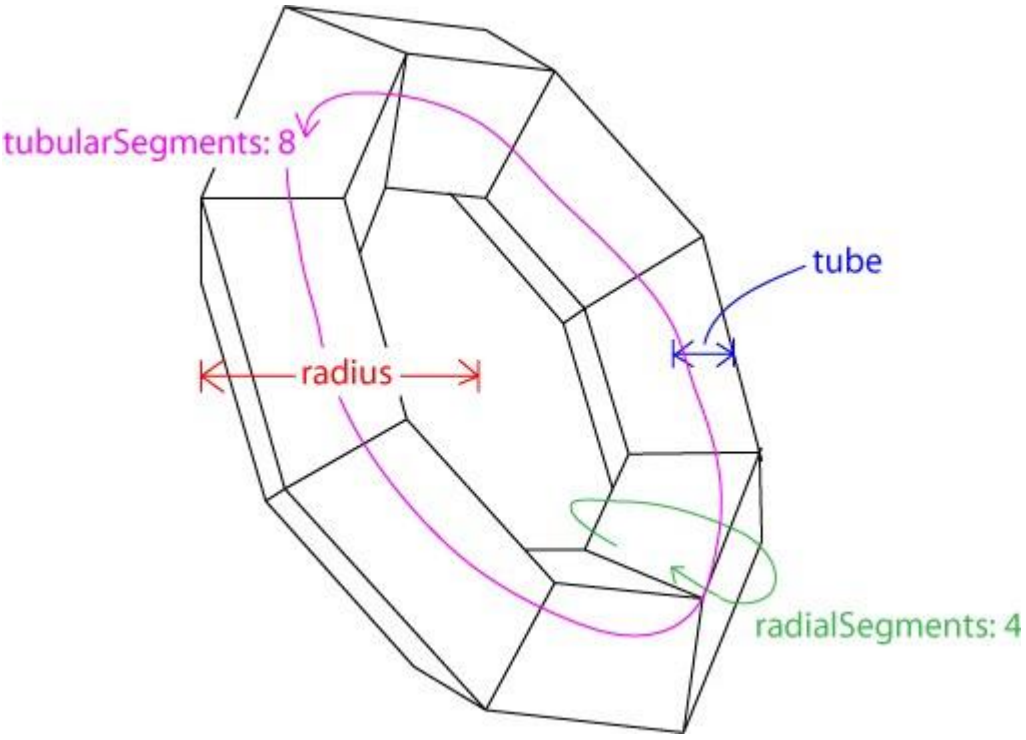
`new THREE.IcosahedronGeometry(3)`创建一个半径为 `3` 的正二十面体：



圆环面

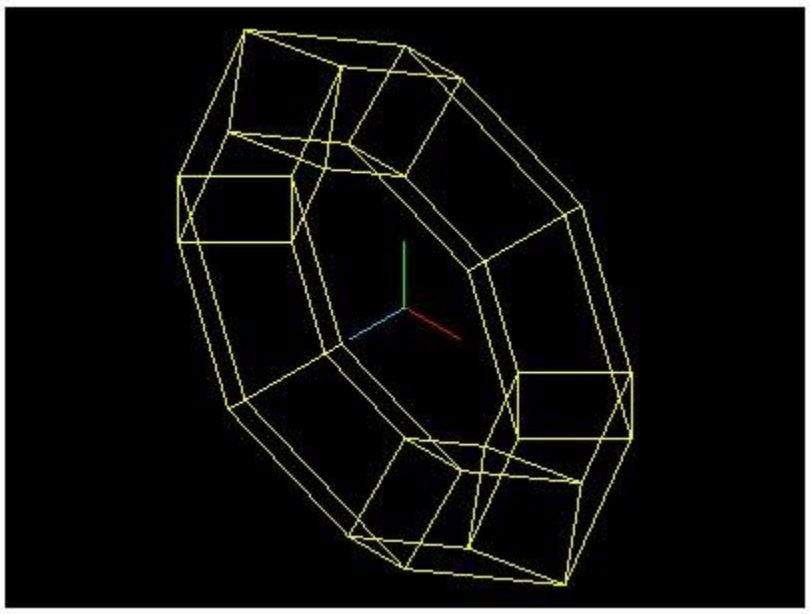
圆环面（TorusGeometry）就是甜甜圈的形状，其构造函数是：

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

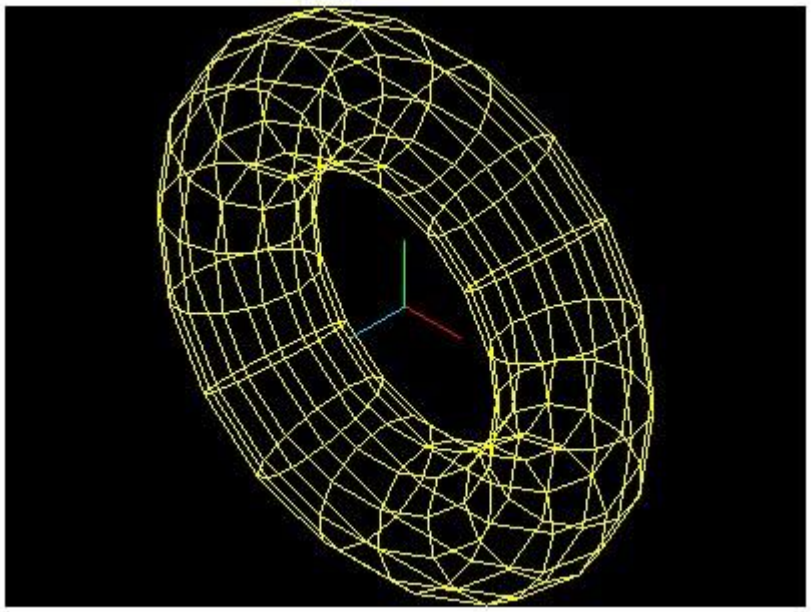


其中，`radius`是圆环半径；`tube`是管道半径；`radialSegments`与`tubularSegments`分别是两个分段数，详见上图；`arc`是圆环面的弧度，缺省值为`Math.PI * 2`。

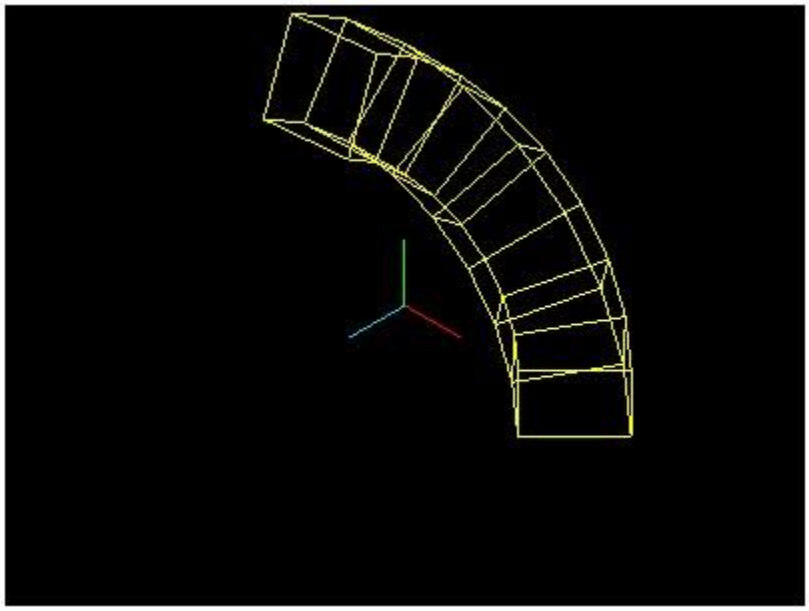
`new THREE.TorusGeometry(3, 1, 4, 8)`创建一个粗糙的圆环面：



`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：



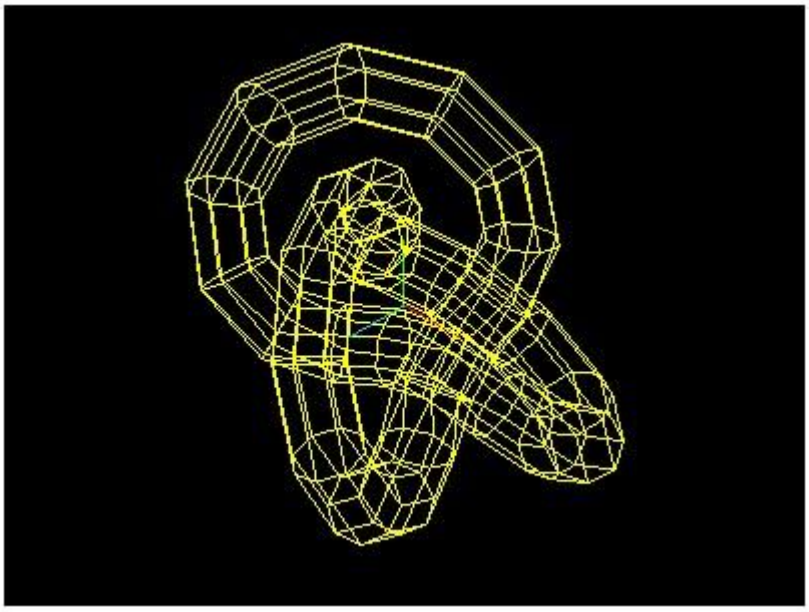
圆环结

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；`heightScale` 是在 z 轴方向上的缩放。

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



本小节完整代码请参见**例 3.1.1**

3.2 文字形状

推荐 0 收藏

文字形状（TextGeometry）可以用来创建三维的文字形状。

下载使用

使用文字形状需要下载和引用额外的字体库，可以在 http://typeface.neocracy.org/下载。这里，我们以 `helvetiker` 字体为例。首先在 http://typeface.neocracy.org/fonts.html 下载对应的压缩包，解压后将 `helvetiker_regular.typeface.js` 文件放在你的目录下，然后在 HTML 文件中引用该文件：

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

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：

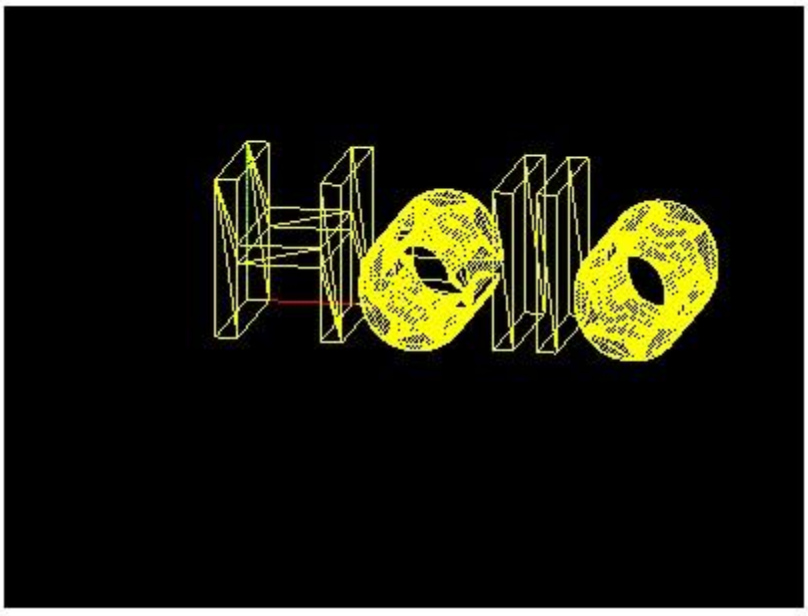
- `size`：字号大小，一般为大写字母的高度
- `height`：文字的厚度
- `curveSegments`：弧线分段数，使得文字的曲线更加光滑
- `font`：字体，默认是`'helvetiker'`，需对应引用的字体文件
- `weight`：值为`'normal'`或`'bold'`，表示是否加粗
- `style`：值为`'normal'`或`'italics'`，表示是否斜体
- `bevelEnabled`：布尔值，是否使用倒角，意为在边缘处斜切

- `bevelThickness`：倒角厚度
- `bevelSize`：倒角宽度

实例说明

例 3.2.1

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()
```

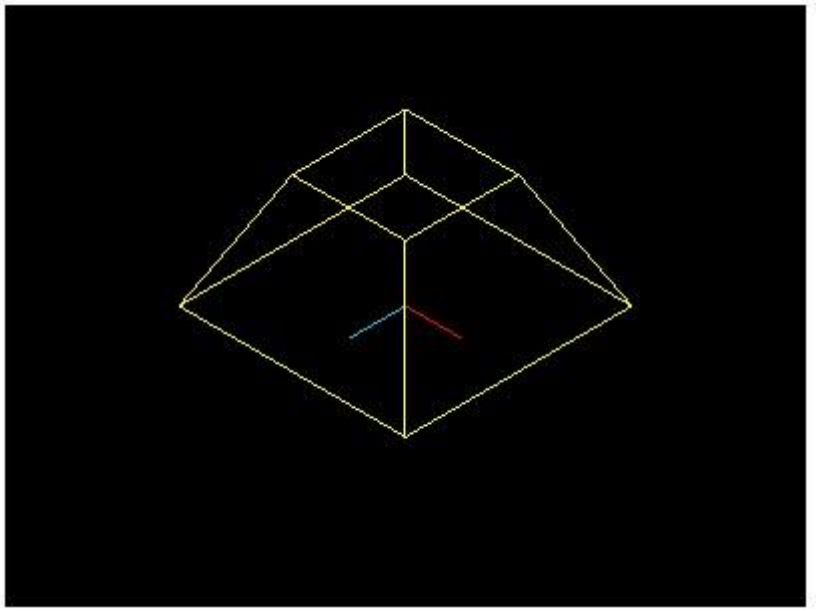
例 3.3.1

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.Face4(0, 1, 2, 3));
// 底面
geometry.faces.push(new THREE.Face4(4, 5, 6, 7));
// 四个侧面
```

```
geometry.faces.push(new THREE.Face4(0, 1, 5, 4));

geometry.faces.push(new THREE.Face4(1, 2, 6, 5));

geometry.faces.push(new THREE.Face4(2, 3, 7, 6));

geometry.faces.push(new THREE.Face4(3, 0, 4, 7));
```

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.Face4(0, 1, 2, 3)`创建一个四个顶点组成的面片，追加到`geometry.faces`数组中。四个参数分别是四个顶点在`geometry.vertices`中的序号。如果需要设置由三个顶点组成的面片，可以类似地使用`THREE.Face3`。

第 4 章 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

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：

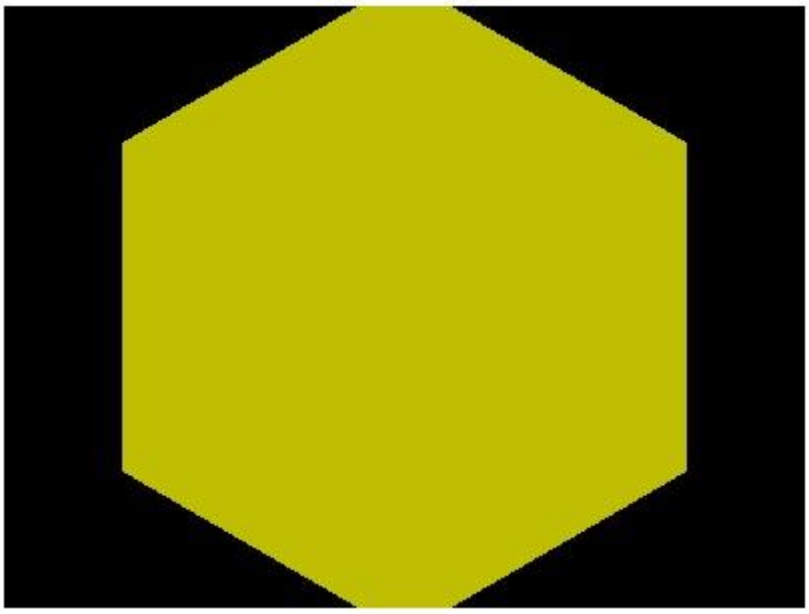
```
THREE.MeshLambertMaterial(opt)
```

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

```
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。

4.2 Lambert 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)
```

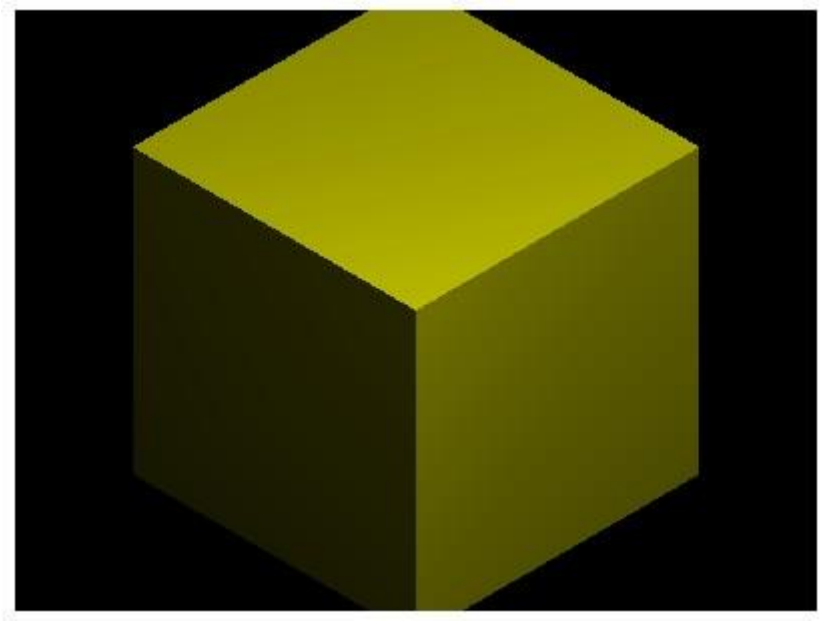
其中，`Idiffuse` 是漫反射光强，`Kd` 是物体表面的漫反射属性，`Id` 是光强，`theta` 是光的入射角弧度。

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

```
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：



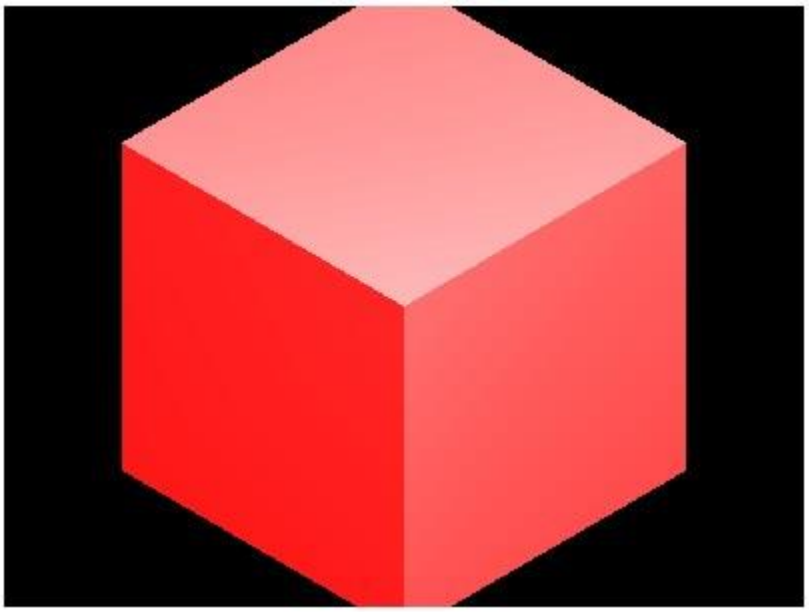
`color` 是用来表现材质对散射光的反射能力，也是最常用来设置材质颜色的属性。除此之外，还可以用 `ambient` 和 `emissive` 控制材质的颜色。

`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

})
```

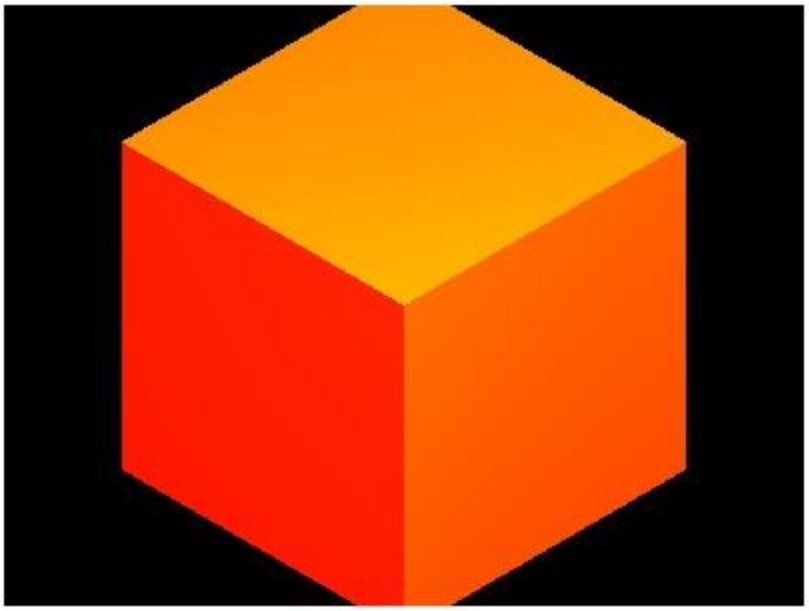
效果为：



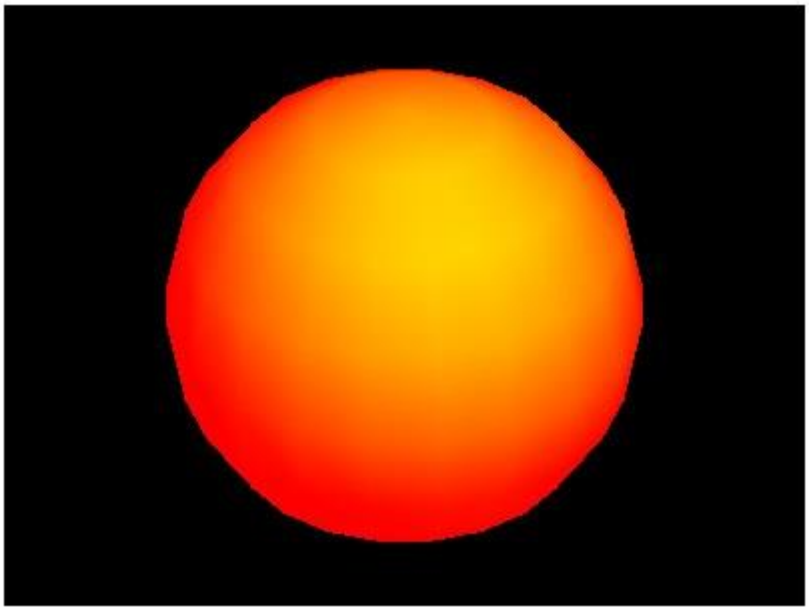
如果同时使用红色的自发光与黄色的散射光：

```
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？



4.3 Phong 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：

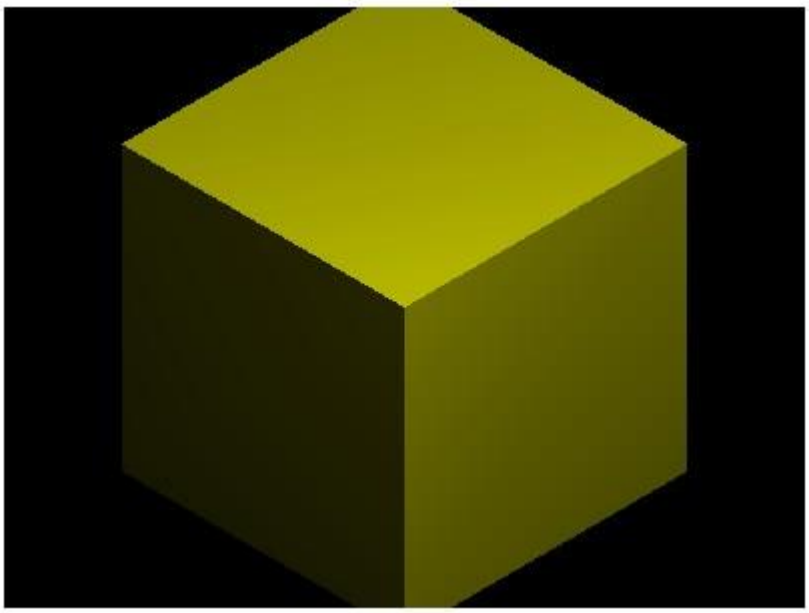
```
Ispecular = Ks * Is * (cos(alpha)) ^ n
```

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

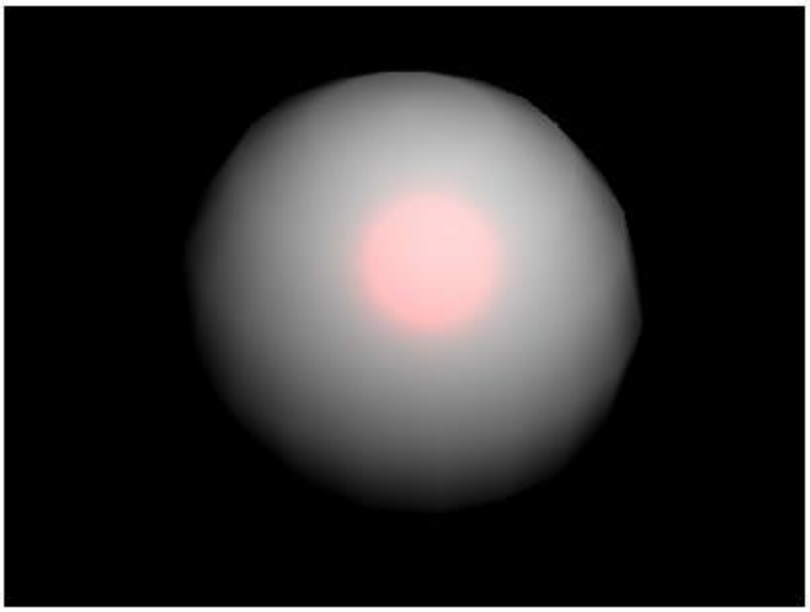
```
new THREE.MeshPhongMaterial({

    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);
```

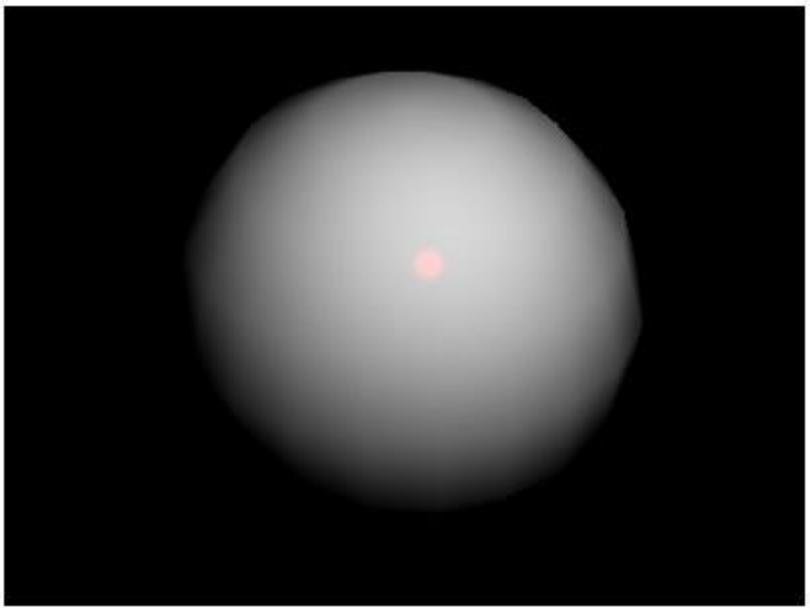
效果为：



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：

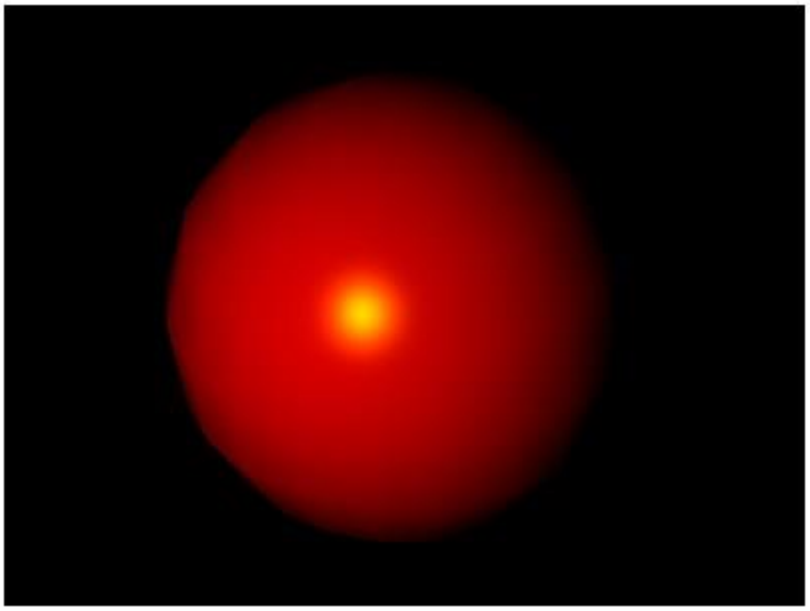
```
new THREE.MeshPhongMaterial({

    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

推荐 0 收藏

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。

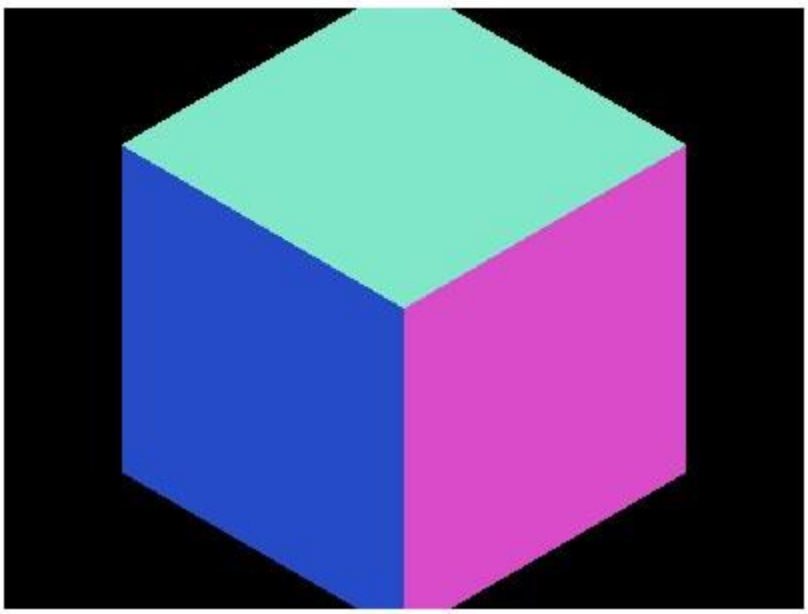
法向材质的设定很简单，甚至不用设置任何参数：

```
new THREE.MeshNormalMaterial()
```

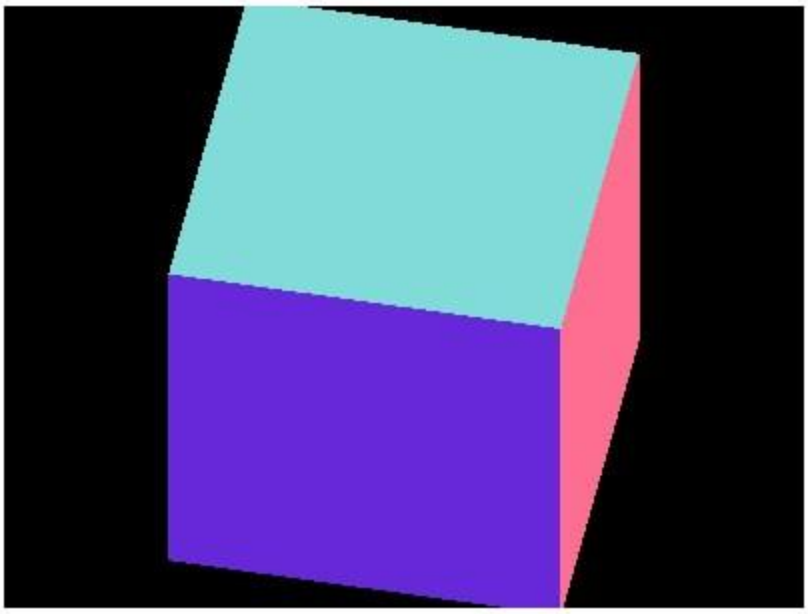
材质的颜色与照相机与该物体的角度相关，下面我们只改变照相机位置，观察两个角度的颜色变化：

例 4.4.1

`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。

单张图像应用于长方体

例 4.5.1

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`：

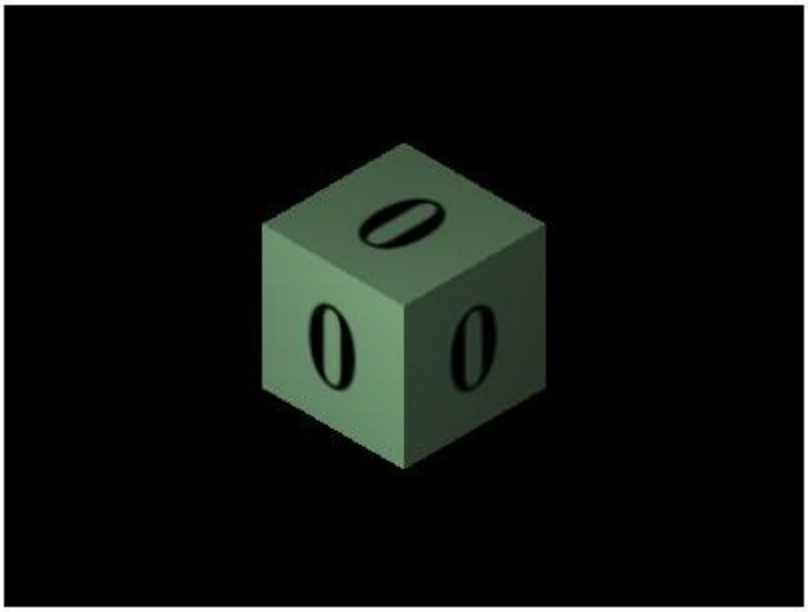
```
var material = new THREE.MeshLambertMaterial({

    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：

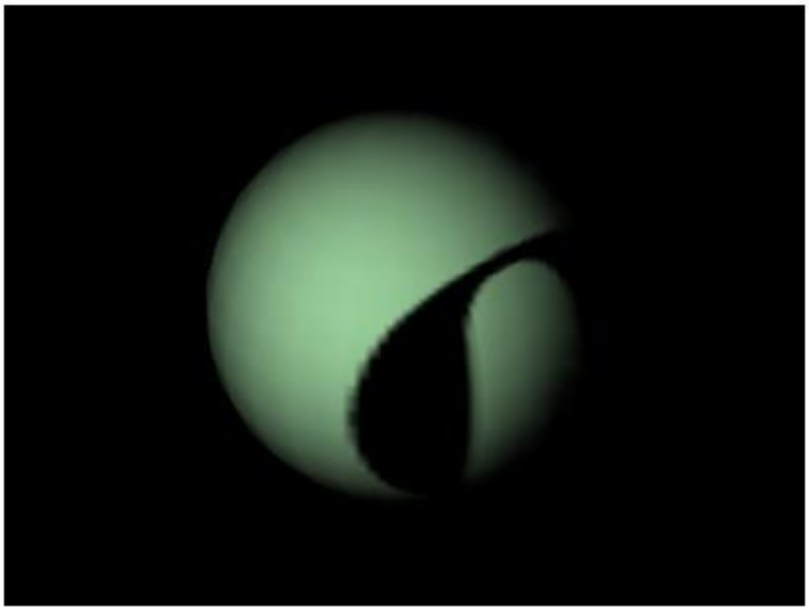
```
var texture = THREE.ImageUtils.loadTexture('../img/0.png', {}, function() {

    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：



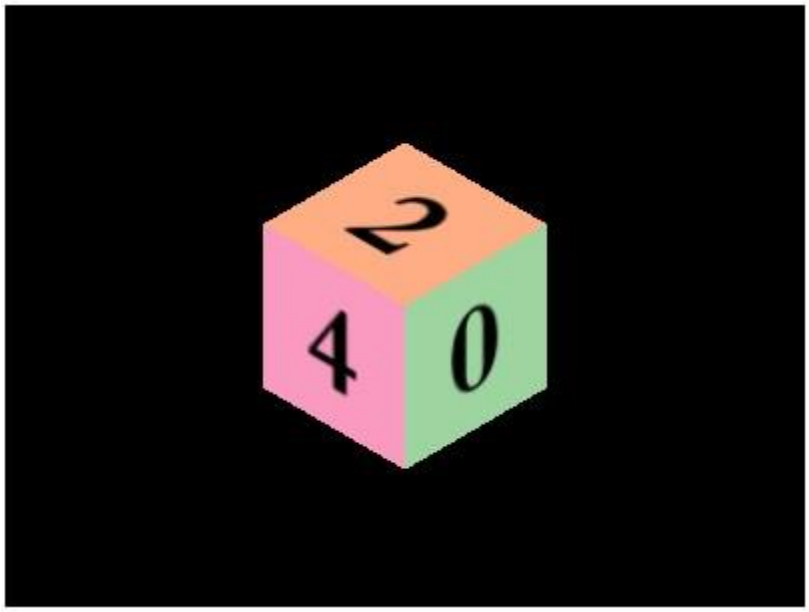
六张图像应用于长方体

例 4.5.2

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：



棋盘格

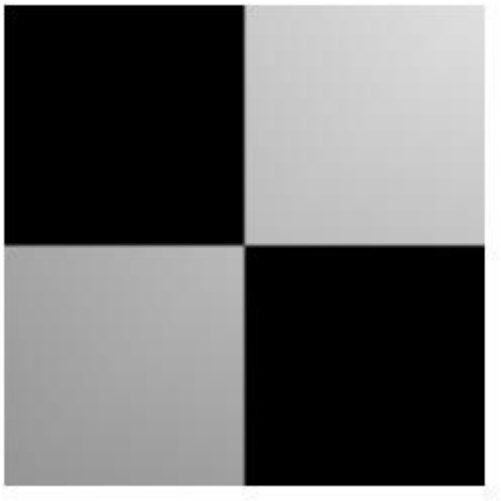
现在，我们有一个黑白相间的图像：

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

```
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？

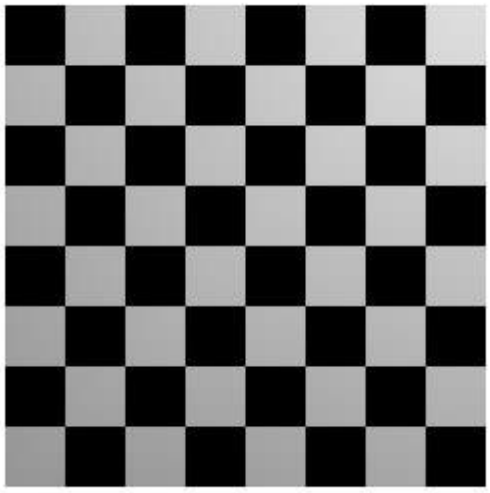
首先，我们需要指定重复方式为两个方向（`wrapS` 和 `wrapT`）都重复：

```
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：



第 5 章 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：

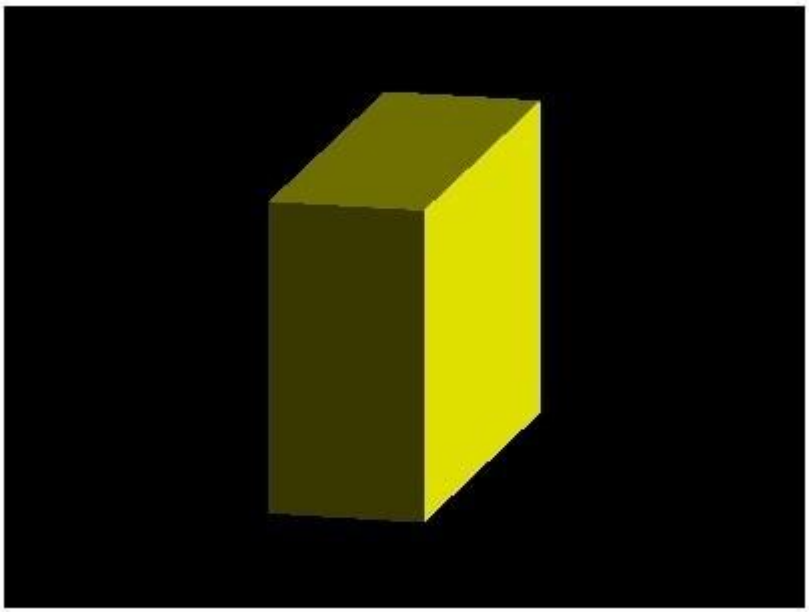
例 5.1.1

```
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);
```

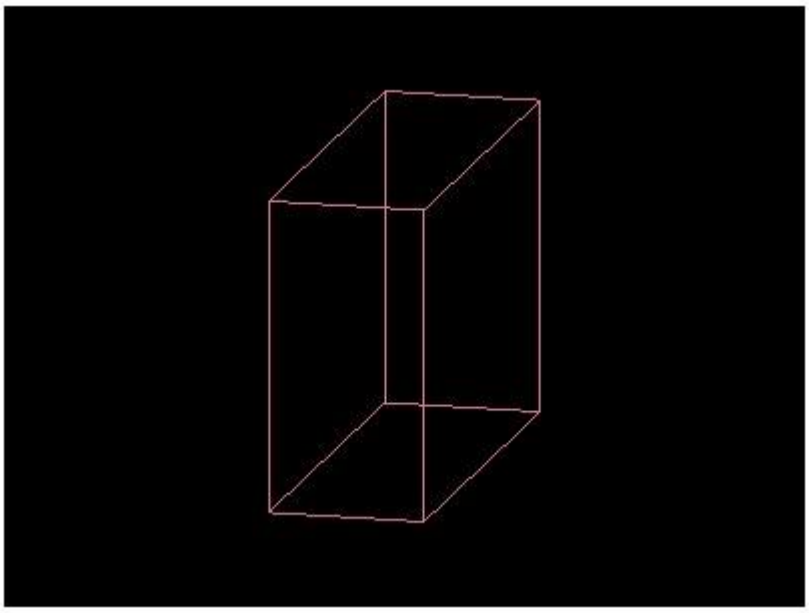
如果 `material` 和 `geometry` 之后不会复用的话，也可以合在一起写为：

```
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

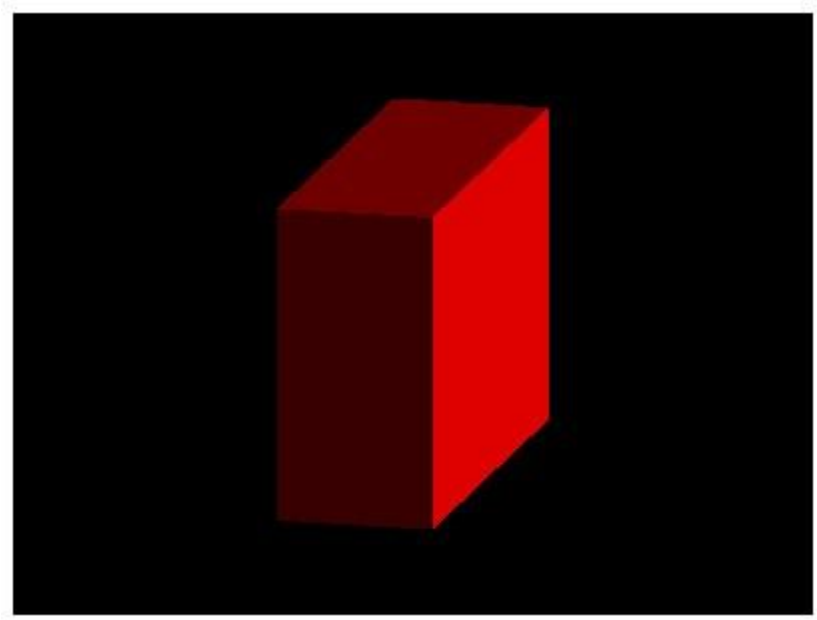
材质

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

```
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：



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

`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。

第 6 章 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，用来观测每秒帧数（FPS）。

6.1 实现动画效果

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。

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），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

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;

    }

}
```

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 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

`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;
```

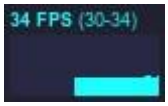
如何取舍

`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 stat.js 记录 FPS

stat.js 是 Three.js 的作者 Mr. Doob 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： 

，单击后显示每帧渲染时间： 

。

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，例 6.2.2

```
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 完整的例子

本节我们将使用一个弹球的例子来完整地学习使用动画效果。

首先，我们把通用的框架部分写好，按照 4.1 节的方法实现动画重绘函数，并按 6.2 节的方法加入 stat.js 库：

```
var requestAnimationFrame = window.requestAnimationFrame

       || window.mozRequestAnimationFrame

       || window.webkitRequestAnimationFrame

       || window.msRequestAnimationFrame;

window.requestAnimationFrame = requestAnimationFrame;
```

```
var scene = null;
var camera = null;
var renderer = null;

var id = null;

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);

    renderer = new THREE.WebGLRenderer({
        canvas: document.getElementById('mainCanvas')
    });
    scene = new THREE.Scene();

    id = requestAnimationFrame(draw);
}
```

```
function draw() {

    stat.begin();



    renderer.render(scene, camera);



    id = requestAnimationFrame(draw);



    stat.end();

}



function stop() {

    if (id !== null) {

        cancelAnimationFrame(id);

        id = null;

    }

}
```

然后，为了实现弹球弹动的效果，我们创建一个球体作为弹球模型，创建一个平面作为弹球反弹的平面。为了在 `draw` 函数中改变弹球的位置，我们可以声明一个全局变量 `ballMesh`，以及弹球半径 `ballRadius`。

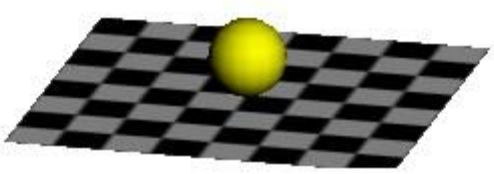
```
var ballMesh = null;

var ballRadius = 0.5;
```

在 `init` 函数中添加球体和平面，使弹球位于平面上，平面采用棋盘格图像作材质：

```
// ball
```

```
ballMesh = new THREE.Mesh(new THREE.SphereGeometry(ballRadius, 16, 8),
    new THREE.MeshLambertMaterial({
        color: 0xffff00
}));
ballMesh.position.y = ballRadius;
scene.add(ballMesh);


// plane
var texture = THREE.ImageUtils.loadTexture('../img/chess.png', {}, function() {
    renderer.render(scene, camera);
});
texture.wrapS = texture.wrapT = THREE.RepeatWrapping;
texture.repeat.set(4, 4);
var plane = new THREE.Mesh(new THREE.PlaneGeometry(5, 5),
        new THREE.MeshLambertMaterial({map: texture}));
plane.rotation.x = -Math.PI / 2;
scene.add(plane);
```

现在，每帧绘制的都是相同的效果：



为了记录弹球的状态，我们至少需要位置、速度、加速度三个矢量，为了简单起见，这里弹球只做竖直方向上的自由落体运动，因此位置、速度、加速度只要各用一个变量表示。其中，位置就是 `ballMesh.position.y`，不需要额外的变量，因此我们在全局声明速度 `v` 和加速度 `a`：

```
var v = 0;

var a = -0.1;
```

这里，`a = -0.1` 代表每帧小球向 y 方向负方向移动 `0.1` 个单位。

一开始，弹球从高度为 `maxHeight` 处自由下落，掉落到平面上时会反弹，并且速度有损耗。当速度很小的时候，弹球会在平面上作振幅微小的抖动，所以，当速度足够小时，我们需要让弹球停止跳动。因此，定义一个全局变量表示是否在运动，初始值为 `false`：

```
var isMoving = false;
```

在 HTML 中定义一个按钮，点击按钮时，弹球从最高处下落：

```
function drop() {
```

```
    isMoving = true;

    ballMesh.position.y = maxHeight;

    v = 0;

}
```

下面就是最关键的函数了，在 `draw` 函数中，需要判断当前的 `isMoving` 值，并且更新小球的速度和位置：

```
function draw() {

    stat.begin();



    if (isMoving) {

        ballMesh.position.y += v;

        v += a;



        if (ballMesh.position.y <= ballRadius) {

            // hit plane

            v = -v * 0.9;

        }



        if (Math.abs(v) < 0.001) {

            // stop moving

            isMoving = false;

            ballMesh.position.y = ballRadius;

        }
```

```
    }


    renderer.render(scene, camera);


    id = requestAnimationFrame(draw);


    stat.end();
}
```

这样就实现小球的弹动效果了。最终的代码为：

例 6.3.1

```
var requestAnimationFrame = window.requestAnimationFrame
        || window.mozRequestAnimationFrame
        || window.webkitRequestAnimationFrame
        || window.msRequestAnimationFrame;
window.requestAnimationFrame = requestAnimationFrame;



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


var id = null;


var stat = null;
```

```
var ballMesh = null;
var ballRadius = 0.5;
var isMoving = false;
var maxHeight = 5;

var v = 0;
var a = -0.01;

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);

    renderer = new THREE.WebGLRenderer({
        canvas: document.getElementById('mainCanvas')
    });
    scene = new THREE.Scene();

    camera = new THREE.OrthographicCamera(-5, 5, 3.75, -3.75, 0.1, 100);
    camera.position.set(5, 10, 20);
    camera.lookAt(new THREE.Vector3(0, 3, 0));
```

```
scene.add(camera);



// ball

ballMesh = new THREE.Mesh(new THREE.SphereGeometry(ballRadius, 16, 8),

    new THREE.MeshLambertMaterial({

        color: 0xffff00

}));

ballMesh.position.y = ballRadius;

scene.add(ballMesh);



// plane

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

    renderer.render(scene, camera);

});

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

texture.repeat.set(4, 4);

var plane = new THREE.Mesh(new THREE.PlaneGeometry(5, 5),

        new THREE.MeshLambertMaterial({map: texture}));

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

scene.add(plane);



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

light.position.set(10, 10, 15);

scene.add(light);
```

```
    id = requestAnimationFrame(draw);
}

function draw() {
    stat.begin();

    if (isMoving) {
        ballMesh.position.y += v;
        v += a;

        if (ballMesh.position.y <= ballRadius) {
            // hit plane
            v = -v * 0.9;
        }

        if (Math.abs(v) < 0.001) {
            // stop moving
            isMoving = false;
            ballMesh.position.y = ballRadius;
        }
    }

    renderer.render(scene, camera);
```

```
    id = requestAnimationFrame(draw);


    stat.end();
}


function stop() {
    if (id !== null) {
        cancelAnimationFrame(id);
        id = null;
    }
}


function drop() {
    isMoving = true;
    ballMesh.position.y = maxHeight;
    v = 0;
}
```

第 7 章 外部模型

第 3 章中我们了解到，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

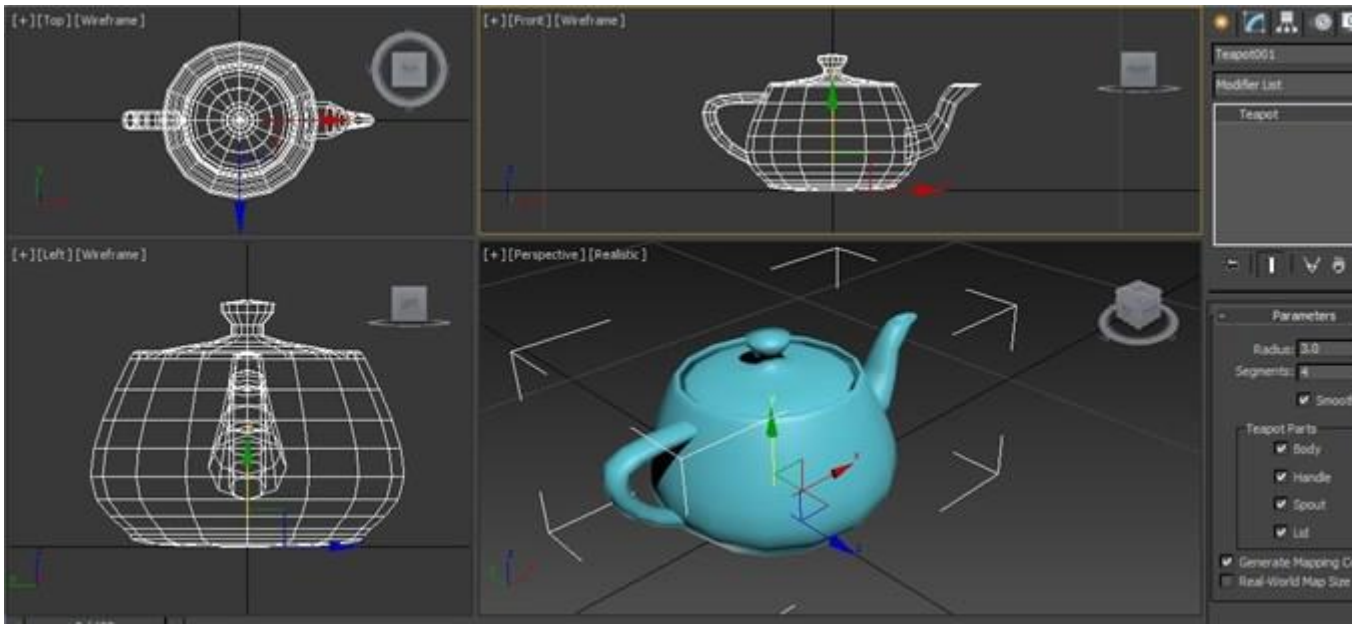
本节中，我们将使用 *3ds Max* 创建一个茶壶模型，并将导出的没有材质的模型使用 Three.js 导入场景中。

首先，下载 OBJLoader.js 并在 HTML 的 `<head>` 中使用：

例 7.2.1

```
<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

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。

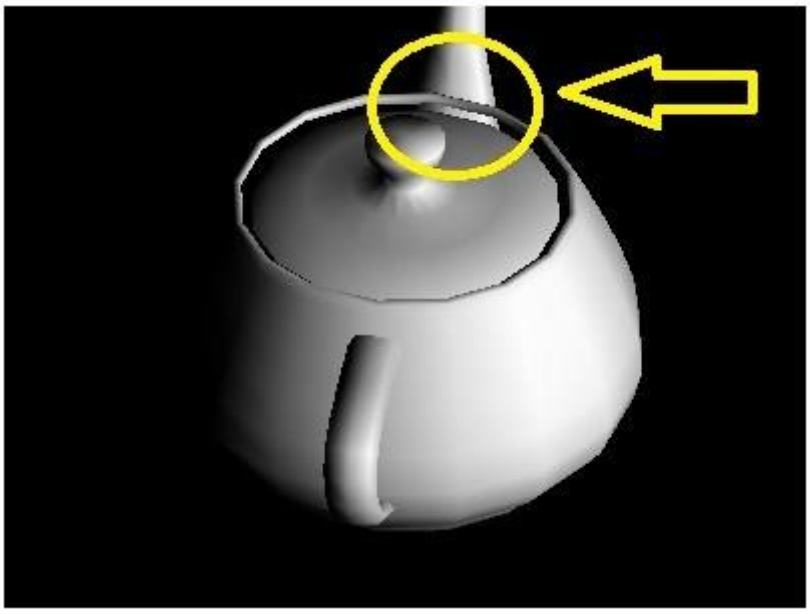
```
loader.load('../lib/port.obj', function(obj) {

    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;

    }

}
```

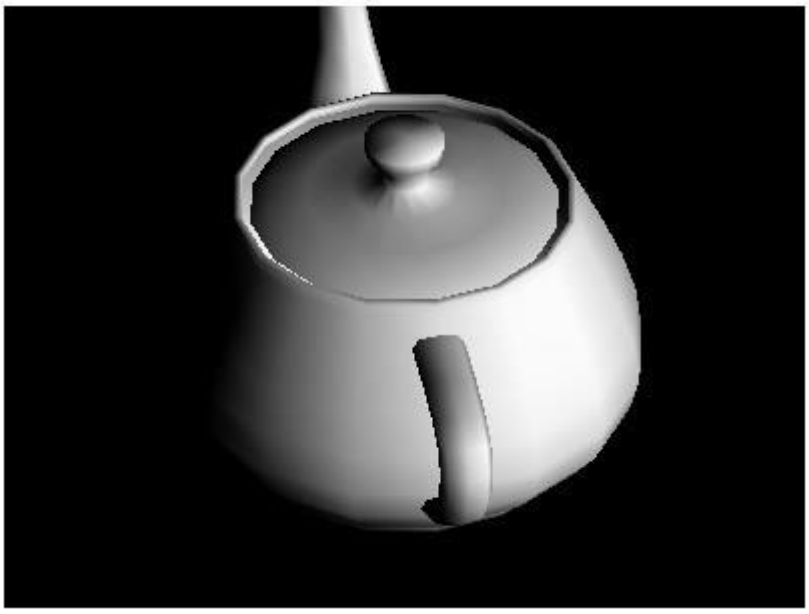
可以看到在某些角度时，好像有些面片没有被绘制出来，因而后方的茶嘴似乎穿越到前方了：



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

推荐 0 收藏

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。

代码中设置材质

这种方法与**例 7.2.1** 类似，不同之处在于回调函数中设置模型的材质：

例 7.3.1

```
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：

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

```
<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：

第 8 章 光与影

推荐 0 收藏

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

推荐 0 收藏

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

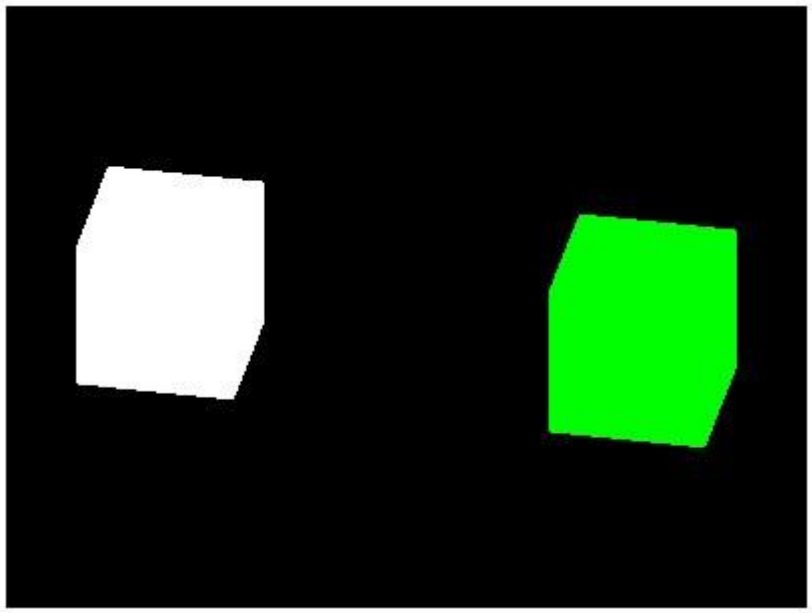
```
var light = new THREE.AmbientLight(0xffffff);

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。

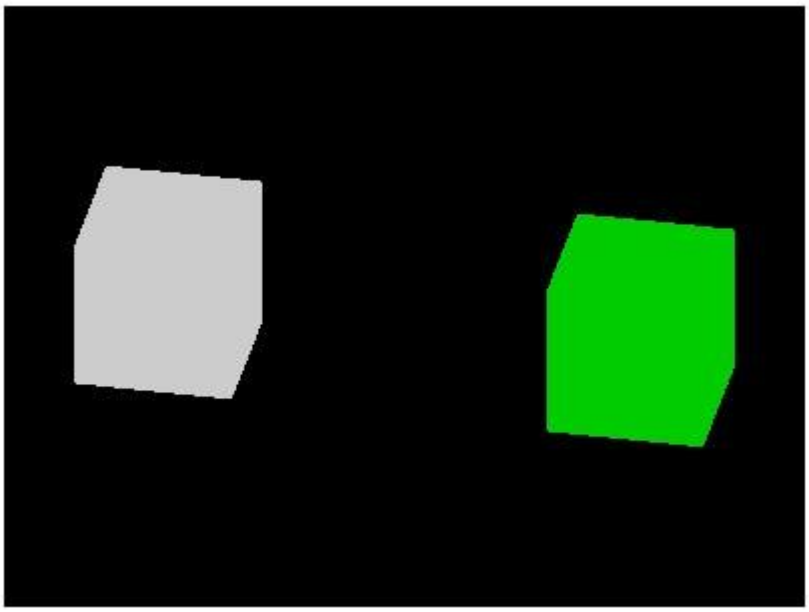
所以，我们添加两个长方体看下效果：

```
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);
```



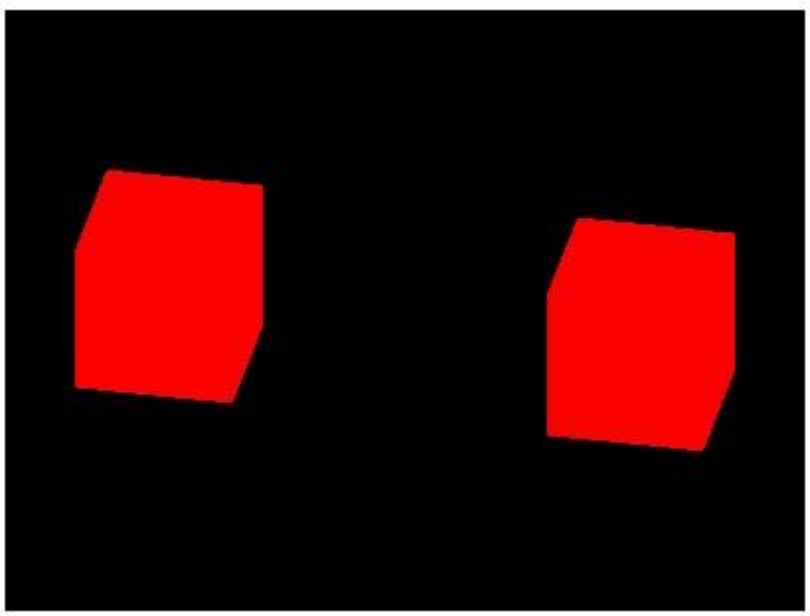
如果想让环境光暗些，可以将其设置为 `new THREE.AmbientLight(0xcccccc)` 等，效果为：



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

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：



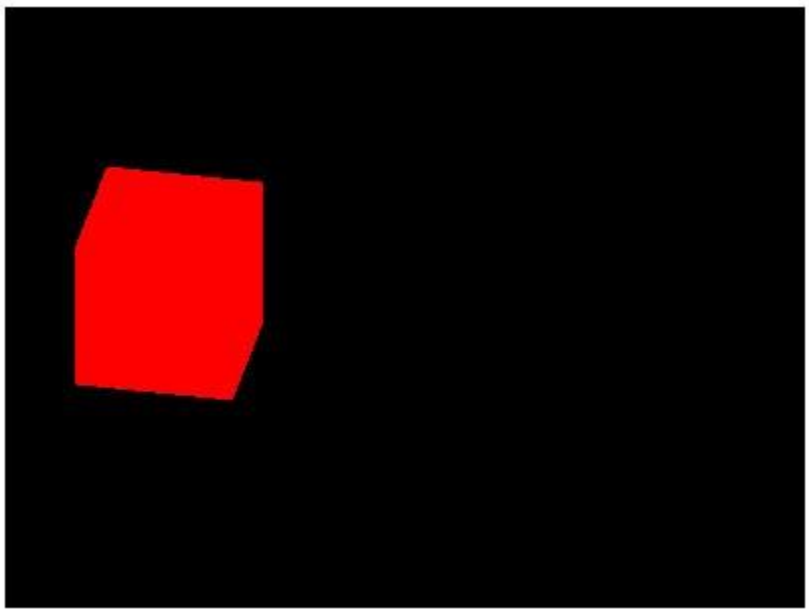
我们将两个长方体材质的颜色分别设置为绿色和白色，渲染的结果是这两个长方体都被渲染成了环境光的红色，这一结果可能有些出乎你的意料。其实，环境光并不在乎物体材质的 `color` 属性，而是 `ambient` 属性。 `ambient` 属性的默认值是 `0xffffff` 。因此，如果将这两个长方体设置为：

例 8.1.3

```
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 点光源

推荐 0 收藏

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：

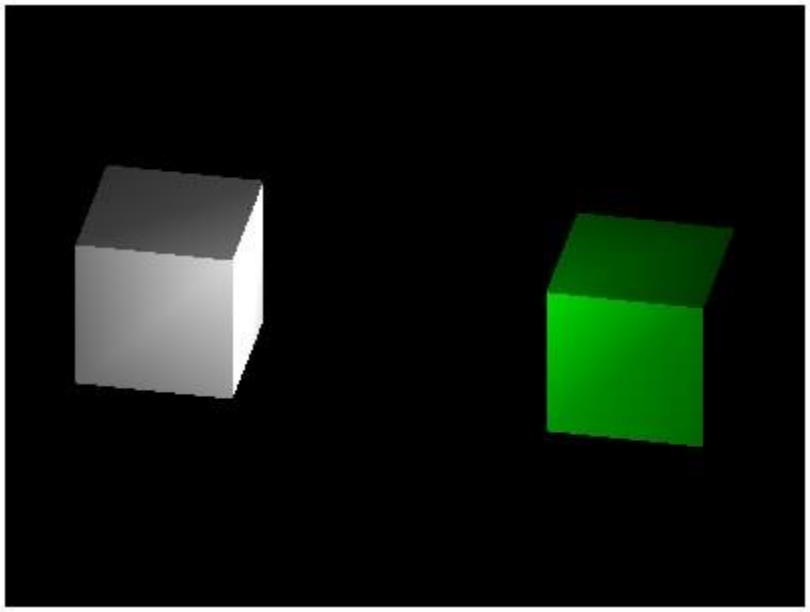
```
THREE.PointLight(hex, intensity, distance)
```

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

```
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)
```

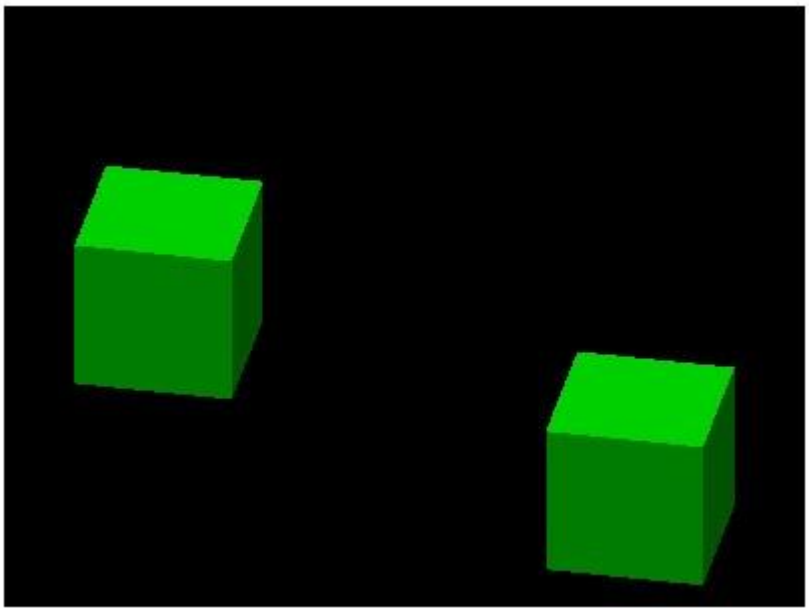
其中，`hex` 是光源十六进制的颜色值；`intensity` 是亮度，缺省值为 1，表示 100%亮度。

此外，对于平行光而言，设置光源位置尤为重要。

例 8.3.1

```
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` ：

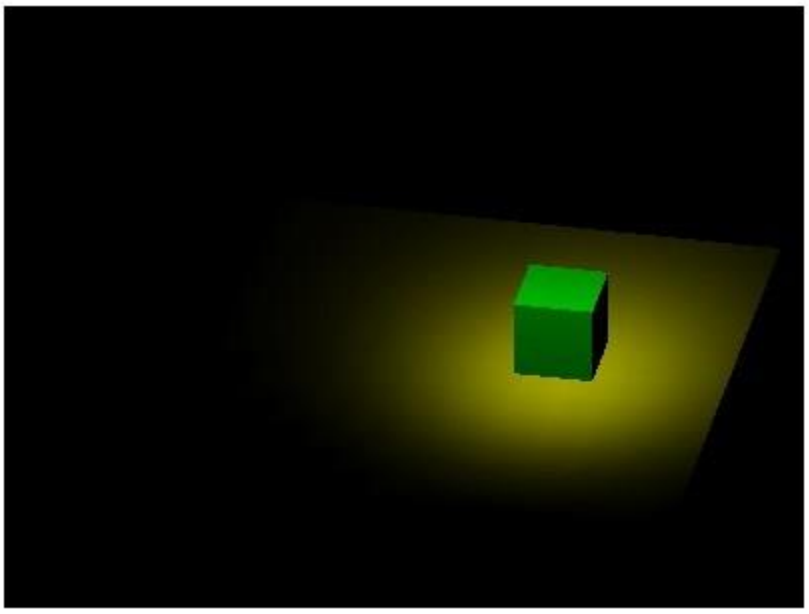
```
light.position.set(x1, y1, z1);

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



8.5 阴影

推荐 0 收藏

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 的基础上增加阴影效果。

例 8.5.1

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`。

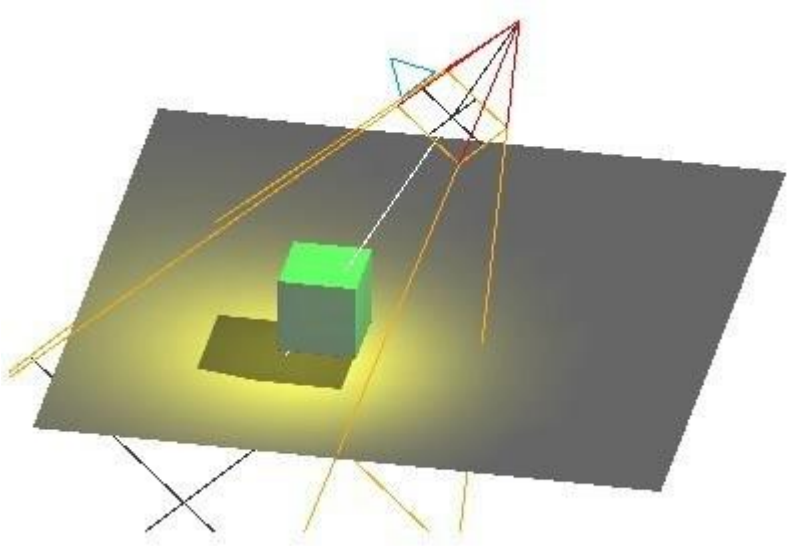
以上就是产生阴影效果的必要步骤了，不过通常还需要设置光源的阴影相关属性，才能正确显示出阴影效果。

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，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);
```

第 9 章 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。

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。

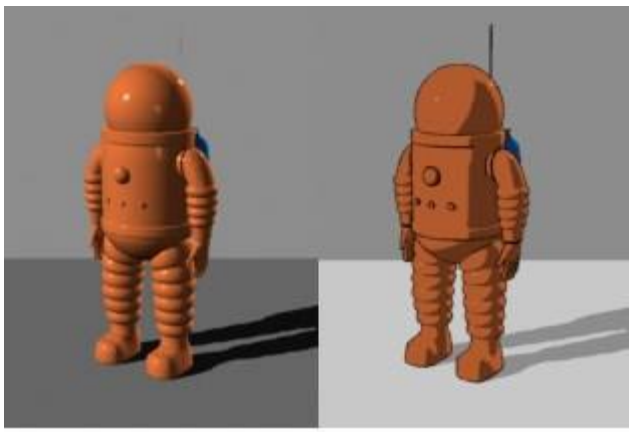
着色器

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：



plastic shader　　toon shader

图片来源

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。

顶点着色器

定点着色器中的“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。

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。

Three.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！

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。

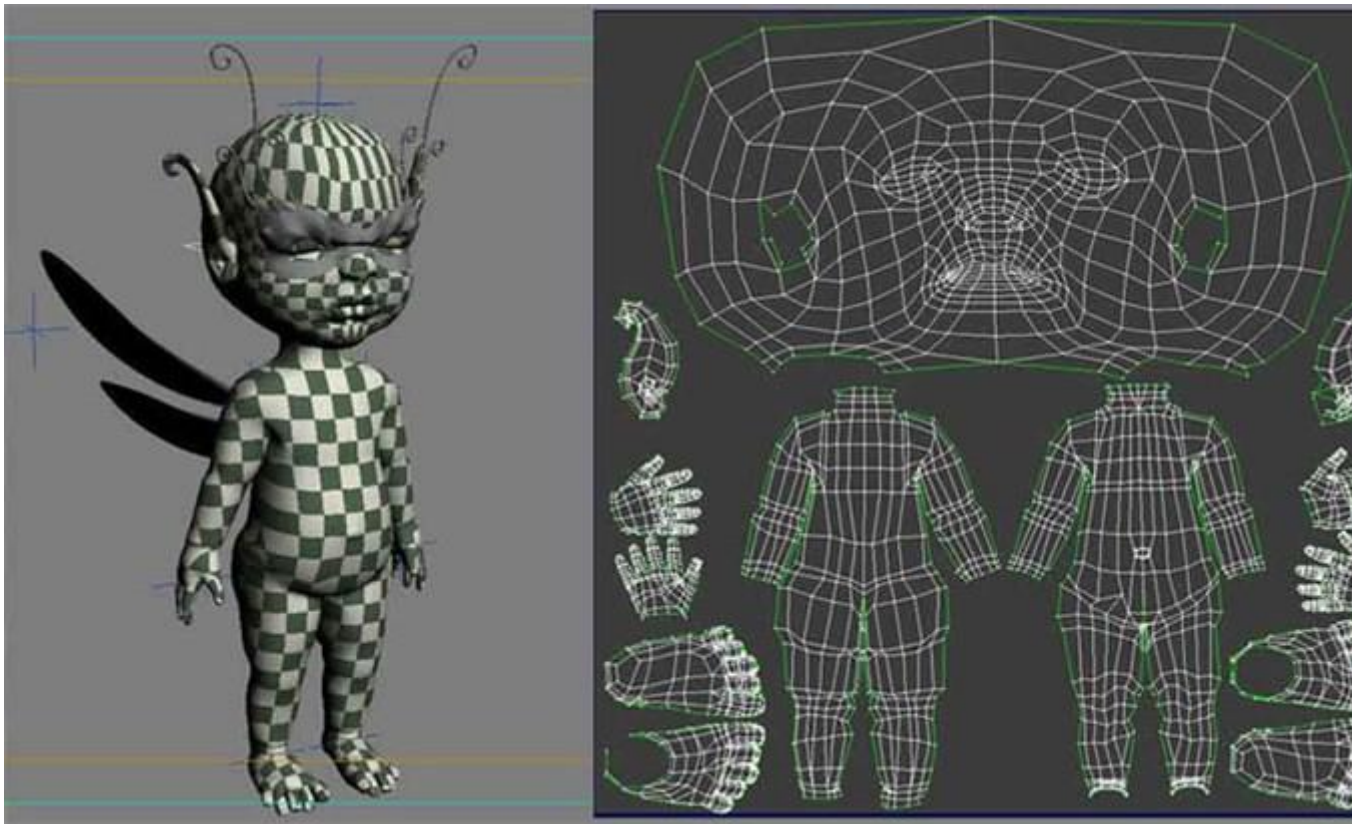
限定符共有四个：

- `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**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：



[+]查看原图

图片来源

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。

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？

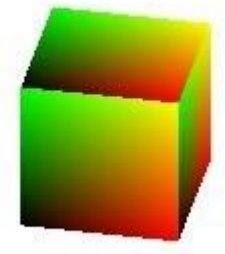
```
varying vec2 vUv;
```

```
void main() {

    gl_FragColor = vec4(vUv.x, vUv.y, 1.0, 1.0);

}
```

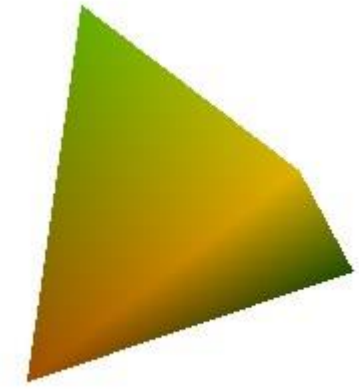
太好了，它看起来很简单！

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

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。

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。`vData` 与 `fData` 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。

HTML 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

});
```

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，例 9.3.2

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() {

    renderer = new THREE.WebGLRenderer({

        canvas: document.getElementById('mainCanvas')

    });

    scene = new THREE.Scene();



    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));

    scene.add(camera);



    var light = new THREE.DirectionalLight();

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

    scene.add(light);



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

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

    scene.add(cube);
```

```
    draw();
}

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：