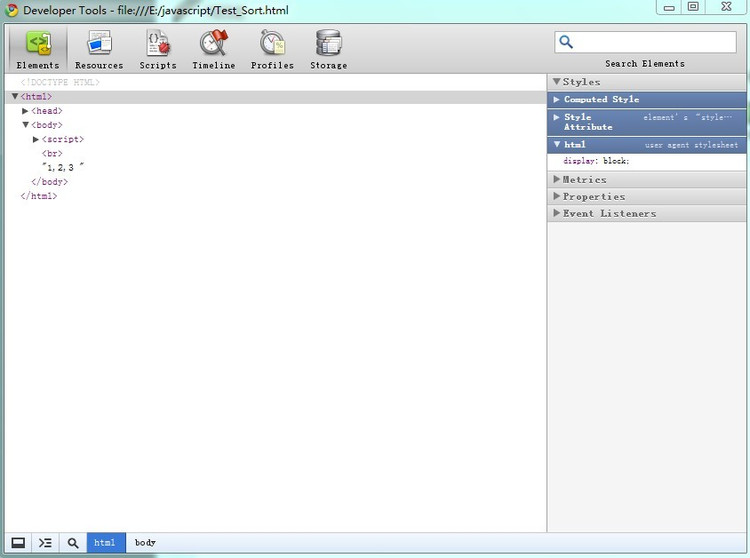
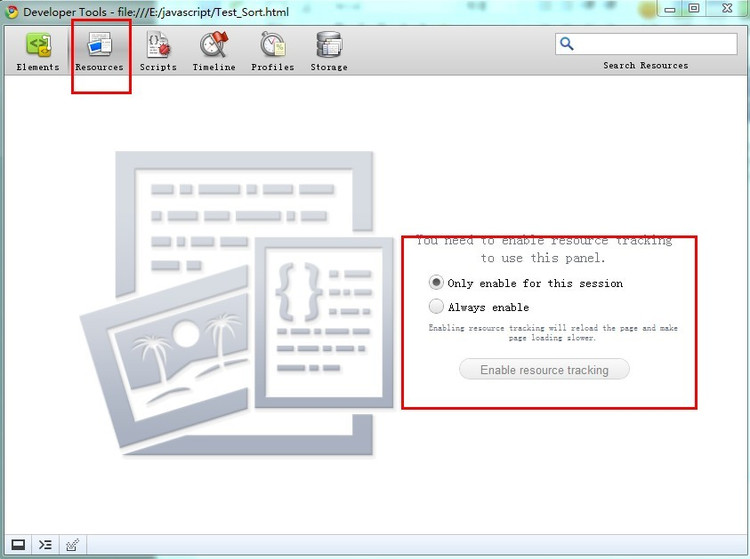
（5.29始）3JS学习

## 如何在chrome中调试Js

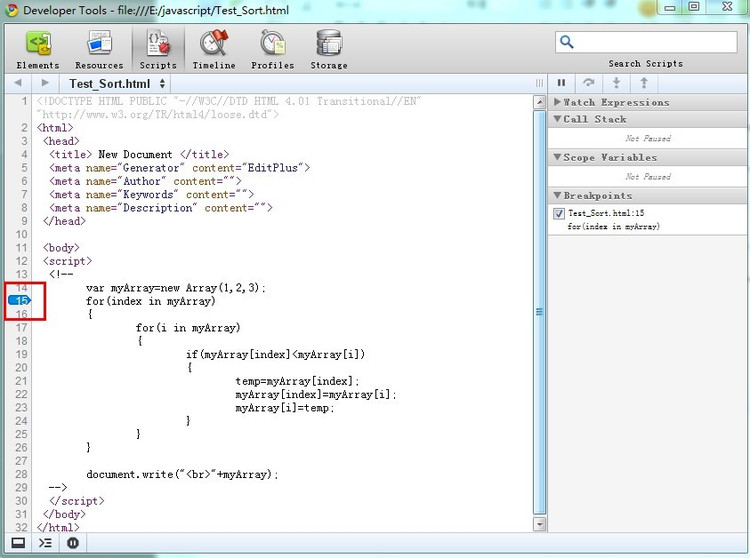
1. 将写好的Javascript代码用chrome打开。
2. 在页面，单击右键，并且选择“审查元素”。
3. 弹出窗体，如图所示：



然后，点击“Resources”标签，如图所示：

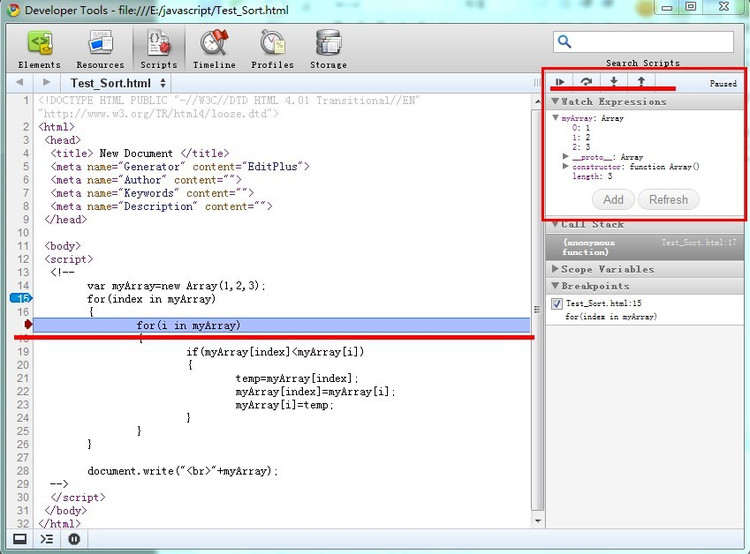


然后点击“enable resource tracking”,导入源文件，并且选择“Script”标签，如下：



添加断点的方法就是在所在的代码行的数字上双击即可！！

debug控制方式：

1. 首先是刷新调试的网页，即重新载入代码。
2. 此时弹出Jscript标签页面，并且中止在断点处。如下：
3. 

## Three.js原目录内容

Build目录：包含两个文件，three.js 和three.min.js 。这是three.js最终被引用的文件。一个已经压缩，一个没有压缩的js文件。

Docs目录：这里是three.js的帮助文档，里面是各个函数的api，可惜并没有详细的解释。试图用这些文档来学会three.js是不可能的。

Editor目录：一个类似3D-max的简单编辑程序，它能创建一些三维物体。

Examples目录：一些很有趣的例子demo，可惜没有文档介绍。对图像学理解不深入的同学，学习成本非常高。

Src目录：源代码目录，里面是所有源代码。

Test目录：一些测试代码，基本没用。

Utils目录：存放一些脚本，python文件的工具目录。例如将3D-Max格式的模型转换为three.js特有的json模型。

.gitignore文件：git工具的过滤规则文件，没有用。

CONTRIBUTING.md文件：一个怎么报bug，怎么获得帮助的说明文档。

LICENSE文件：版权信息。

README.md文件：介绍three.js的一个文件，里面还包含了各个版本的更新内容列表。

## 三大组建

在Three.js中，要渲染物体到网页中，我们需要3个组建：场景（scene）、相机（camera）和渲染器（renderer）。有了这三样东西，才能将物体渲染到网页中去。

记住关建语句：有了这三样东西，我们才能够使用相机将场景渲染到网页上去。

创建这三要素的代码如下：

[View Raw Code](http://www.hewebgl.com/article/getarticle/50)[?](http://www.oriontransfer.co.nz/software/jquery-syntax)

|  |
| --- |
| var scene = new THREE.Scene(); *// 场景* |
| var camera = new THREE.PerspectiveCamera(75, window.innerWidth/window.innerHeight, 0.1, 1000);*// 透视相机* |
| var renderer = new THREE.WebGLRenderer(); *// 渲染器* |
| renderer.setSize(window.innerWidth, window.innerHeight); *// 设置渲染器的大小为窗口的内宽度，也就是内容区的宽度* |
| document.body.appendChild(renderer.domElement); |

### 场景

在Threejs中场景就只有一种，用THREE.Scene来表示，要构件一个场景也很简单，只要new一个对象就可以了，代码如下：

var scene = new THREE.Scene();

场景是所有物体的容器，如果要显示一个苹果，就需要将苹果对象加入场景中

### 相机

另一个组建是相机，相机决定了场景中那个角度的景色会显示出来。相机就像人的眼睛一样，人站在不同位置，抬头或者低头都能够看到不同的景色。

场景只有一种，但是相机却又很多种。和现实中一样，不同的相机确定了呈相的各个方面。比如有的相机适合人像，有的相机适合风景，专业的摄影师根据实际用途不一样，选择不同的相机。对程序员来说，只要设置不同的相机参数，就能够让相机产生不一样的效果。

在Threejs中有多种相机，这里介绍两种，它们是：

透视相机（THREE.PerspectiveCamera）、这里我们使用一个透视相机，透视相机的参数很多，这里先不详细讲解。后面关于相机的那一章，我们会花大力气来讲。定义一个相机的代码如下所示：（已经迫不及待想看看相机的参数了，点这里）

|  |
| --- |
| var camera = new THREE.PerspectiveCamera(75, window.innerWidth/window.innerHeight, 0.1, 1000); |

### 渲染器

最后一步就是设置渲染器，渲染器决定了渲染的结果应该画在页面的什么元素上面，并且以怎样的方式来绘制。这里我们定义了一个WebRenderer渲染器，代码如下所示：

var renderer = new THREE.WebGLRenderer();

renderer.setSize(window.innerWidth, window.innerHeight);

document.body.appendChild(renderer.domElement);

注意，渲染器renderer的domElement元素，表示渲染器中的画布，所有的渲染都是画在domElement上的，所以这里的appendChild表示将这个domElement挂接在body下面，这样渲染的结果就能够在页面中显示了。

渲染应该使用渲染器，结合相机和场景来得到结果画面。实现这个功能的函数是

renderer.render(scene, camera);

渲染函数的原型如下：

render( scene, camera, renderTarget, forceClear )

各个参数的意义是：

scene：前面定义的场景

camera：前面定义的相机

renderTarget：渲染的目标，默认是渲染到前面定义的render变量中

forceClear：每次绘制之前都将画布的内容给清除，即使自动清除标志autoClear为false，也会清除。

渲染有两种方式：实时渲染和离线渲染 。

先看看离线渲染，想想《西游降魔篇》中最后的佛主，他肯定不是真的，是电脑渲染出来的，其画面质量是很高的，它是事先渲染好一帧一帧的图片，然后再把图片拼接成电影的。这就是离线渲染。如果不事先处理好一帧一帧的图片，那么电影播放得会很卡。CPU和GPU根本没有能力在播放的时候渲染出这种高质量的图片。

实时渲染：就是需要不停的对画面进行渲染，即使画面中什么也没有改变，也需要重新渲染。下面就是一个渲染循环：

function render() {

cube.rotation.x += 0.1;

cube.rotation.y += 0.1;

renderer.render(scene, camera);

requestAnimationFrame(render);

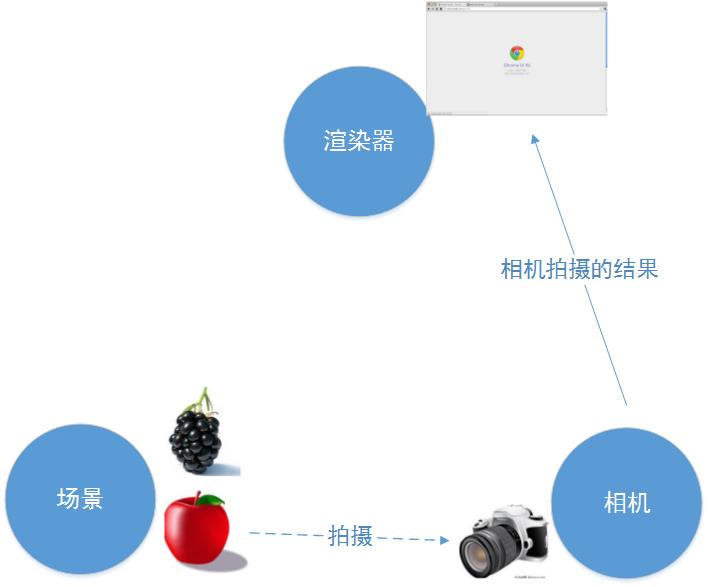
}

其中一个重要的函数是requestAnimationFrame，这个函数就是让浏览器去执行一次参数中的函数，这样通过上面render中调用requestAnimationFrame()函数，requestAnimationFrame()函数又让rander()再执行一次，就形成了我们通常所说的游戏循环了。

### 三者关系

Three.js中的场景是一个物体的容器，开发者可以将需要的角色放入场景中，例如苹果，葡萄。同时，角色自身也管理着其在场景中的位置。

相机的作用就是面对场景，在场景中取一个合适的景，把它拍下来。

渲染器的作用就是将相机拍摄下来的图片，放到浏览器中去显示。他们三者的关系如下图所示：

## 框架

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Three框架</title>

<script src="js/Three.js" data-ke-src="js/Three.js"></script>

<style type="text/css">

div#canvas-frame {

border: none;

cursor: pointer;

width: 100%;

height: 600px;

background-color: #EEEEEE;

}

</style>

<script>

var renderer;

function initThree() {

width = document.getElementById('canvas-frame').clientWidth;

height = document.getElementById('canvas-frame').clientHeight;

renderer = new THREE.WebGLRenderer({

antialias : true

});

renderer.setSize(width, height);

document.getElementById('canvas-frame').appendChild(renderer.domElement);

renderer.setClearColor(0xFFFFFF, 1.0);

}

var camera;

function initCamera() {

camera = new THREE.PerspectiveCamera(45, width / height, 1, 10000);

camera.position.x = 0;

camera.position.y = 1000;

camera.position.z = 0;

camera.up.x = 0;

camera.up.y = 0;

camera.up.z = 1;

camera.lookAt({

x : 0,

y : 0,

z : 0

});

}

var scene;

function initScene() {

scene = new THREE.Scene();

}

var light;

function initLight() {

light = new THREE.DirectionalLight(0xFF0000, 1.0, 0);

light.position.set(100, 100, 200);

scene.add(light);

}

var cube;

function initObject() {

var geometry = new THREE.Geometry();

var material = new THREE.LineBasicMaterial( { vertexColors: THREE.VertexColors} );

var color1 = new THREE.Color( 0x444444 ), color2 = new THREE.Color( 0xFF0000 );

// 线的材质可以由2点的颜色决定

var p1 = new THREE.Vector3( -100, 0, 100 );

var p2 = new THREE.Vector3( 100, 0, -100 );

geometry.vertices.push(p1);

geometry.vertices.push(p2);

geometry.colors.push( color1, color2 );

var line = new THREE.Line( geometry, material, THREE.LinePieces );

scene.add(line);

}

function render()

{

renderer.clear();

renderer.render(scene, camera);

requestAnimationFrame(render);

}

function threeStart() {

initThree();

initCamera();

initScene();

initLight();

initObject();

render();

}

</script>

</head>

<body onload="threeStart();">

<div id="canvas-frame"></div>

</body>

</html>

（6.7）3JS学习（二）||基础第二章点线面

## 在Threejs中定义一个点

在三维空间中的某一个点可以用一个坐标点来表示。一个坐标点由x,y,z三个分量构成。

空间几何中，点可以用一个向量来表示，在Three.js中也是用一个向量来表示的，代码如下所示：

|  |
| --- |
| THREE.Vector3 = function ( x, y, z )  { |
|  |
| this.x = x || 0; |
| this.y = y || 0; |
| this.z = z || 0; |
|  |
| }; |

前面我们已经知道了THREE是Three.js引擎的一个全局变量。

那么THREE.Vector3呢，就是表示Vector3是定义在THREE下面的一个类。(以后要用Vector3，就必须要加THREE前缀。)

当然Three.js的设计者，也可以不加THREE这个前缀，但是他们预见到，Three.js引擎中会有很多类型，最好给这些类型加一个前缀，以免与开发者的代码产生冲突。

THREE.Vector3被赋值为一个函数。这个函数有3个参数，分别代表x坐标，y坐标和z坐标的分量。函数体内的代码将他们分别赋值给成员变量x，y，z。看看上面的代码，中间使用了一个“||”（或）运算符，就是当x=null或者undefine时，this.x的值应该取0。

### 点的操作

在3D世界中点可以用THREE.Vector3D来表示。对应源码为/src/math/Vector3.js（注意：源码所在的位置，可能不同版本不一样，请自己搜索Vector3关键词来确定）。在您继续学习之前，你可以打开该文件浏览一下，推荐使用WebStorm，如果没有，你也可以用NotePad++。

现在来看看怎么定义个点，假设有一个点x=4，y=8，z=9。你可以这样定义它：

var point1 = new THREE.Vecotr3(4,8,9);

另外你也可以使用set方法，代码如下：

var point1 = new THREE.Vector3();

point1.set(4,8,9);

我们这里使用了set方法，为了以后深入学习的方便，这里将Vector3的常用方法列出如下，为了不影响文章的连贯性，我们专门列出了一个网页来介绍它。

### 使用THREE.LineBasicMaterial类型来定义一种线条的材质，

LineBasicMaterial( parameters )

Parameters是一个定义材质外观的对象，它包含多个属性来定义材质，这些属性是：

Color：线条的颜色，用16进制来表示，默认的颜色是白色。

Linewidth：线条的宽度，默认时候1个单位宽度。

Linecap：线条两端的外观，默认是圆角端点，当线条较粗的时候才看得出效果，如果线条很细，那么你几乎看不出效果了。

Linejoin：两个线条的连接点处的外观，默认是“round”，表示圆角。

VertexColors：定义线条材质是否使用顶点颜色，这是一个boolean值。意思是，线条各部分的颜色会根据顶点的颜色来进行插值。

Fog：定义材质的颜色是否受全局雾效的影响。

## 线条的深入理解

在Threejs中，一条线由点，材质和颜色组成。

点由THREE.Vector3表示，Threejs中没有提供单独画点的函数，它必须被放到一个THREE.Geometry形状中，这个结构中包含一个数组vertices，这个vertices就是存放无数的点（THREE.Vector3）的数组。这个表示可以如下图所示：three.js向量

为了绘制一条直线，首先我们需要定义两个点，如下代码所示：

var p1 = new THREE.Vector3( -100, 0, 100 );

var p2 = new THREE.Vector3( 100, 0, -100 );

请大家思考一下，这两个点在坐标系的什么位置，然后我们声明一个THREE.Geometry，并把点加进入，代码如下所示：

var geometry = new THREE.Geometry();

geometry.vertices.push(p1);

geometry.vertices.push(p2);

geometry.vertices的能够使用push方法，是因为geometry.vertices是一个数组。这样geometry 中就有了2个点了。

然后我们需要给线加一种材质，可以使用专为线准备的材质，THREE.LineBasicMaterial。

最终我们通过THREE.Line绘制了一条线，如下代码所示:

var line = new THREE.Line( geometry, material, THREE.LinePieces );

最终，line就是我们要的线条了。

### 实例：画一条彩色线

**1、首先，我们声明了一个几何体geometry**

var geometry = new THREE.Geometry();

几何体里面有一个vertices变量，可以用来存放点。

**2、定义一种线条的材质，使用THREE.LineBasicMaterial类型来定义，它接受一个集合作为参数**

这里使用了顶点颜色vertexColors: THREE.VertexColors，就是线条的颜色会根据顶点来计算。

var material = new THREE.LineBasicMaterial( { vertexColors: THREE.VertexColors } );

**3、接下来，定义两种颜色，分别表示线条两个端点的颜色**

var color1 = new THREE.Color( 0x444444 ),

color2 = new THREE.Color( 0xFF0000 );

**4、定义2个顶点的位置，并放到geometry中**

var p1 = new THREE.Vector3( -100, 0, 100 );

var p2 = new THREE.Vector3( 100, 0, -100 );

geometry.vertices.push(p1);

geometry.vertices.push(p2);

**5、为4中定义的2个顶点，设置不同的颜色**

geometry.colors.push( color1, color2 );

geometry中colors表示顶点的颜色，必须材质中vertexColors等于THREE.VertexColors 时，颜色才有效，如果vertexColors等于THREE.NoColors时，颜色就没有效果了。那么就会去取材质中color的值。

**6、定义一条线** //将线加入场景

定义线条，使用THREE.Line类，代码如下所示：

var line = new THREE.Line( geometry, material, THREE.LinePieces );

第一个参数是几何体geometry，里面包含了2个顶点和顶点的颜色。第二个参数是线条的材质，或者是线条的属性，表示线条以哪种方式取色。第三个参数是一组点的连接方式.

然后，将这条线加入到场景中，代码如下：

scene.add(line);

这样，场景中就会出现刚才的那条线段了。

## 坐标平面

这个平面的效果如下所示：

它横竖分别绘制了20条线段，在摄像机的照射下，就形成了这般模样.

**1.定义2个点**

在x轴上定义两个点p1(-500,0,0)，p2(500,0,0)。

geometry.vertices.push( new THREE.Vector3( - 500, 0, 0 ) );

geometry.vertices.push( new THREE.Vector3( 500, 0, 0 ) );

|  |
| --- |
| var cube; |
| function initObject() { |
| var geometry = new THREE.Geometry(); |
| *// B begin* |
| geometry.vertices.push( new THREE.Vector3( - 500, 0, 0 ) ); |
| geometry.vertices.push( new THREE.Vector3( 500, 0, 0 ) ); |
| *// B end* |
|  |
| **2.算法**  这两个点决定了x轴上的一条线段，将这条线段复制20次，分别平行移动到z轴的不同位置，就能够形成一组平行的线段。  同理，将p1p2这条线先围绕y轴旋转90度，然后再复制20份，平行于z轴移动到不同的位置，也能形成一组平行线。  经过上面的步骤，就能够得到坐标网格了。  for ( var i = 0; i <= 20; i ++ ) { |
|  |
| var line = new THREE.Line( geometry, new THREE.LineBasicMaterial( { color: 0x000000, opacity: 0.2 } ) ); |
| line.position.z = ( i \* 50 ) - 500; |
| scene.add( line ); |
|  |
| var line = new THREE.Line( geometry, new THREE.LineBasicMaterial( { color: 0x000000, opacity: 0.2 } ) ); |
| line.position.x = ( i \* 50 ) - 500; |
| line.rotation.y = 90 \* Math.PI / 180; |
| scene.add( line ); |
|  |
| } |
| } |

## 让场景动起来

《古兰经》上有这样一个故事：一天穆罕穆德告诉人们说大山会向我们走来。于是人们就远望大山，看它怎么走过来，可是等了好长时间大山还是纹丝不动的在那里，人们就问穆罕穆德，大山也没向我们走来啊。默罕默德告诉人们：既然大山没向我们走来，那我们就向大山走去吧。于是人们来到了大山的山顶，人们征服了那座大山。

这个故事揭示了场景动起来的方法，第一种方法是让物体在坐标系里面移动，摄像机不动。第二种方法是让摄像机在坐标系里面移动，物体不动。这样场景就能够动起来了。

摄像机可以理解我们自己的眼睛。

### 渲染循环

物体运动还有一个关键点，就是要渲染物体运动的每一个过程，让它显示给观众。渲染的时候，我们调用的是渲染器的render() 函数。代码如下：

renderer.render( scene, camera );

如果我们改变了物体的位置或者颜色之类的属性，就必须重新调用render()函数，才能够将新的场景绘制到浏览器中去。不然浏览器是不会自动刷新场景的。

如果不断的改变物体的颜色，那么就需要不断的绘制新的场景，所以我们最好的方式，是让画面执行一个循环，不断的调用render来重绘，这个循环就是渲染循环，在游戏中，也叫游戏循环。

为了实现循环，我们需要javascript的一个特殊函数，这个函数是requestAnimationFrame。

调用requestAnimationFrame函数，传递一个callback参数，则在下一个动画帧时，会调用callback这个函数。

function animate()

{

render();

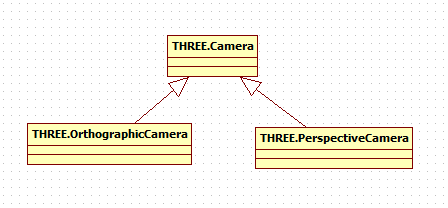
requestAnimationFrame( animate );

}

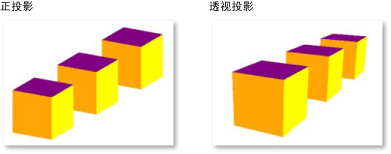
这样就会不断的执行animate这个函数。也就是不断的执行render()函数。在render()函数中不断的改变物体或者摄像机的位置，并渲染它们，就能够实现动画了。

### 认识相机

在Threejs中相机的表示是THREE.Camera，它是相机的抽象基类，其子类有两种相机，分别是正投影相机THREE.OrthographicCamera和透视投影相机THREE.PerspectiveCamera。类图如下所示：



正投影相机有时候也叫正交投影摄像机，下图显示了正交摄像机投影和透视投影之间的差别。

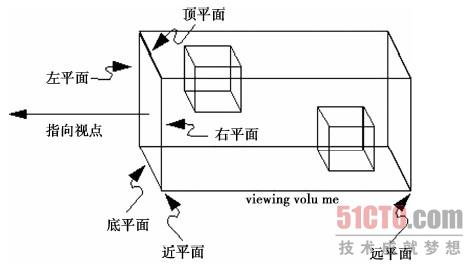


#### 正投影相机

下面我们来介绍正投影相机，正投影的构造函数如下所示：

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

结合下面一个图，我们来看看，各个参数的意思。



1、 left参数

left：左平面距离相机中心点的垂直距离。从图中可以看出，左平面是屏幕里面的那个平面。

2、 right参数

right：右平面距离相机中心点的垂直距离。从图中可以看出，右平面是屏幕稍微外面一点的那个平面。

3、 top参数

top：顶平面距离相机中心点的垂直距离。上图中的顶平面，是长方体头朝天的平面。

4、 bottom参数

bottom：底平面距离相机中心点的垂直距离。底平面是头朝地的平面。

5、near参数

near：近平面距离相机中心点的垂直距离。近平面是左边竖着的那个平面。

6、far参数

far：远平面距离相机中心点的垂直距离。远平面是右边竖着的那个平面。

有了这些参数和相机中心点，我们这里将相机的中心点又定义为相机的位置。通过这些参数，我们就能够在三维空间中唯一的确定上图的一个长方体。这个长方体也叫做视景体。

投影变换的目的就是定义一个视景体，使得视景体外多余的部分裁剪掉，最终图像只是视景体内的有关部分。

好了，看一个简单的例子：

var camera = new THREE.OrthographicCamera( width / - 2, width / 2, height / 2, height / - 2, 1, 1000 );

scene.add( camera );

这个例子将浏览器窗口的宽度和高度作为了视景体的高度和宽度，相机正好在窗口的中心点上。这也是我们一般的设置方法，基本上为了方便，我们不会设置其他的值。

#### 透视投影相机

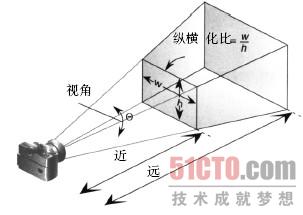
透视投影是更符合我们视觉的投影，

正因为，透视相机这么有魅力，所以在各种应用中运用非常广泛。

透视投影相机的构造函数如下所示：

PerspectiveCamera( fov, aspect, near, far )

我们来欣赏一幅图来看看这个函数的各个参数的意思：



1、视角fov：这个最难理解,我的理解是,眼睛睁开的角度,即,视角的大小,如果设置为0,相当你闭上眼睛了,所以什么也看不到,如果为180,那么可以认为你的视界很广阔,但是在180度的时候，往往物体很小，因为他在你的整个可视区域中的比例变小了。

2、近平面near：这个呢，表示你近处的裁面的距离。补充一下，也可以认为是眼睛距离近处的距离，假设为10米远，请不要设置为负值，Three.js就傻了,不知道怎么算了,

3、远平面far：这个表示你远处的裁面,

4、纵横比aspect：实际窗口的纵横比，即宽度除以高度。这个值越大，说明你宽度越大，那么你可能看的是宽银幕电影了。

好了，看看下面一个简单的例子：

var camera = new THREE.PerspectiveCamera( 45, width / height, 1, 1000 );

scene.add( camera );

视角越大，中间的物体越小，这是因为，视角越大，看到的场景越大，那么中间的物体相对于整个场景来说，就越小了。

## 五彩的光源

### 基类

在Threejs中，光源用Light表示，它是所有光源的基类。它的构造函数是：

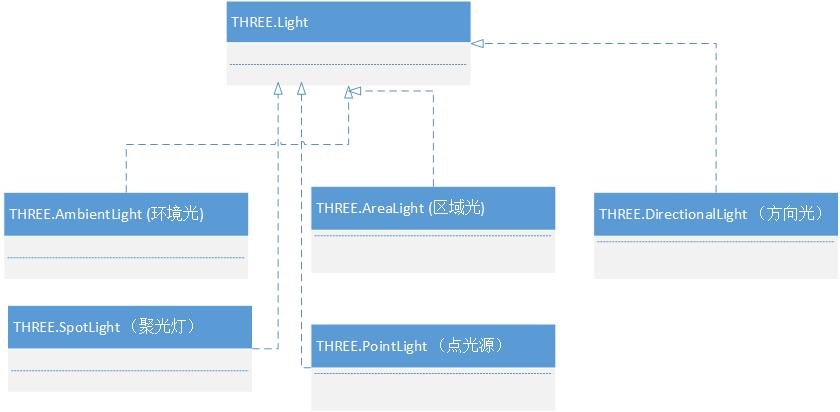
THREE.Light ( hex )

它有一个参数hex，接受一个16进制的颜色值。例如要定义一种红色的光源，我们可以这样来定义：

Var redLight = new THREE.Light(0xFF0000);

### 派生出来的其他种类光源

THREE.Light只是其他所有光源的基类，要让光源除了具有颜色的特性之外，我们需要其他光源。看看，下面的类图，是目前光源的继承结构。



#### 环境光

环境光是经过多次反射而来的光称为环境光，无法确定其最初的方向。环境光是一种无处不在的光。环境光源放出的光线被认为来自任何方向。因此，当你仅为场景指定环境光时，所有的物体无论法向量如何，都将表现为同样的明暗程度。 （这是因为，反射光可以从各个方向进入您的眼睛）

环境光用THREE.AmbientLight来表示，它的构造函数如下所示：

THREE.AmbientLight( hex )

它仍然接受一个16进制的颜色值，作为光源的颜色。环境光将照射场景中的所有物体，让物体显示出某种颜色。环境光的使用例子如下所示：

var light = new THREE.AmbientLight( 0xff0000 );

scene.add( light );

只需要将光源加入场景，场景就能够通过光源渲染出好的效果来了。

#### 点光源(泛光灯？)

点光源：由这种光源放出的光线来自同一点，且方向辐射自四面八方。例如蜡烛放出的光，萤火虫放出的光。

点光源用PointLight来表示，它的构造函数如下所示：

PointLight( color, intensity, distance )

这个类的参数稍微复杂一些，我们花点时间来解释一下：

Color：光的颜色

Intensity：光的强度，默认是1.0,就是说是100%强度的灯光，

distance：光的距离，从光源所在的位置，经过distance这段距离之后，光的强度将从Intensity衰减为0。 默认情况下，这个值为0.0，表示光源强度不衰减

#### 聚光灯

聚光灯：这种光源的光线从一个锥体中射出，在被照射的物体上产生聚光的效果。使用这种光源需要指定光的射出方向以及锥体的顶角α。聚光灯示例如图所示：

聚光灯的构造函数是：

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

函数的参数如下所示：

Hex：聚光灯发出的颜色，如0xFFFFFF

Intensity：光源的强度，默认是1.0，如果为0.5，则强度是一半，意思是颜色会淡一些。和上面点光源一样。

Distance：光线的强度，从最大值衰减到0，需要的距离。 默认为0，表示光不衰减，如果非0，则表示从光源的位置到Distance的距离，光都在线性衰减。到离光源距离Distance时，光源强度为0.

Angle：聚光灯着色的角度，用弧度作为单位，这个角度是和光源的方向形成的角度。

exponent：光源模型中，衰减的一个参数，越大衰减约快。

#### 方向光（平行光）

平行光又称为方向光（Directional Light），是一组没有衰减的平行的光线，类似太阳光的效果。

方向光的模型如图：



方向光的构造函数如下所示：

THREE.DirectionalLight = function ( hex, intensity )

其参数如下：

Hex：关系的颜色，用16进制表示

Intensity：光线的强度，默认为1。因为RGB的三个值均在0~255之间，不能反映出光照的强度变化，光照越强，物体表面就更明亮。它的取值范围是0到1。如果为0，表示光线基本没什么作用，那么物体就会显示为黑色。

平行光有一个方向，方向由位置和原点（0,0,0）来决定，方向光只与方向有关，与离物体的远近无关。

### 材质与光源的关系

最常见的材质之一就是Lambert材质，这是在灰暗的或不光滑的表面产生均匀散射而形成的材质类型。比如一张纸就是Lambert表面。 首先它粗糙不均匀，不会产生镜面效果。我们在阅读书籍的时候，没有发现书上一处亮，一处不亮吧，它非常均匀，这就是兰伯特材质。

有的朋友觉得纸不粗糙啊，你怎么说它粗糙吗？人的肉眼是不好分辨出来，它粗不粗糙的。

Lambert材质表面会在所有方向上均匀地散射灯光，这就会使颜色看上去比较均匀。想想一张纸，无论什么颜色，是不是纸的各个部分颜色都比较均匀呢。

Lambert材质会受环境光的影响，呈现环境光的颜色，与材质本身颜色关系不大。

PS:环境光就是在场景中无处不在的光，它对物体的影响是均匀的，也就是无论你从物体的那个角度观察，物体的颜色都是一样的，这就是伟大的环境光。

你可以把环境光放在任何一个位置，它的光线是不会衰减的，是永恒的某个强度的一种光源。