servasoftware

TWaver® MONO (HTML5 3D)

开发手册

Version 2.1.7

Jan 2016
Serva Software
info@servasoftware.com
http://www.servasoftware.com
PO Box 8143, Wichita Falls, Texas, USA 76307

目录

阅读之前

如果您准备使用TWaver HTML5 3D进行应用开发，那么本文档适合您阅读。请注意“HTML5”和“3D”两个关键字。如果关注的是Flex、Java、.NET等技术，或关注的是2D技术（例如拓扑图），则不在本文档讨论范围内，请阅读其他相关产品分支文档。

为了简化起见，除非明确说明，本文中对TWaver HTML5 3D简称为“twaver”，或“mono”。“mono”也是TWaver HTML5 3D的内部产品代号。请注意，“MONO DESIGN”是一个基于mono的3D编辑器，用于做机房、物体等的建模，和mono本身并不是一个产品，请注意区别。关于MONO DESIGN的使用，请阅读本站相关的使用手册。

“HTML5”本文中也常被简称为“H5”，也是业界常用的简称方式。

“JavaScript”本文中简称“js”。

一句话解释什么是mono

mono是基于h5技术的3d引擎，用它可以在网页上开发3d应用，而无需插件。h5的3d标准是webGL，一个简化版的openGL，允许使用js语言在浏览器中直接编写3D代码。由于webGL定义的接口非常底层，所以在浏览器中直接写webGL能识别的代码是无法想像的。mono则把webGL的底层接口进行二次封装，形成了一个“3D引擎”。开发者直接使用mono.js这个库，就可以在浏览器里面更容易的撰写3d代码。

所以，总结mono是：一个mono.js文件，一个3d引擎，一个3d开发sdk，一个webGL开发框架。

什么应用适合使用mono

mono产品的核心特点是：3D、Web、无插件、跨平台，适合创建轻量的跨平台的网页三维应用。如果您的应用符合下面的一项或多项需求，可考虑使用mono引擎：

- 在网页上显示3d场景
- 不想使用插件
- 页面能跨浏览器，跨PC、平板、手机显示

并非所有3d需求都适合mono。mono更适合简单、轻量的3d呈现，而不适合很“重”的3d呈现。如果您的应用有以下情形，则不建议使用mono，可转而考虑使用本地桌面3D技术：

- 硬件配置较差，例如无独立显卡，CPU、内存等配置远低于主流性能
- 浏览器有限制，例如只能使用低版本的IE浏览器、不允许安装Chrome等非IE浏览器

- 3d场景超大、超复杂（例如场景顶点数达千万级）、逼真度要求极高，需要加载大量素材资源，各种特效，刷新率及交互要求极高的应用。例如大型3D游戏、建筑业BIM设计、复杂的工业3D图纸设计等

目前，mono的典型应用行业场景有：

- 机房3D可视化呈现
- 电力变电站可视化呈现
- 智能园区/楼宇3D可视化
- 各类工业自动化监控系统可视化
- 3D库房管理
- 复杂大数据3D呈现

以上仅罗列目前mono客户的普遍需求。实际上由于mono是一个通用的3D引擎，它可以做各类行业的3D场景呈现。mono本身并没有太多限制，有限制的可能是我们的思维和创新意识。有新想法再加上mono，就一定能在不同行业创造出令人耳目一新的、全新用户体验的可视化系统。

软硬件的要求

硬件方面，所有的3D技术对硬件要求都较高，mono作为一个3D引擎也不例外。3D程序对硬件的要求体现在：显卡、CPU、内存几个方面。虽然很难用硬性指标来描述和约束硬件的具体要求，不过您还是可以参考以下几条主要的原则：

- 显卡：最好是具有独立显卡的电脑，显存容量越大越好
- CPU：越快越好，主流的i5、i7处理器都可以很好的支撑3D应用
- 内存：内存和显存都是越多越好。1G的显存和2G的内存是需要的，2G的显存和4G的内存则可以更流畅的运行3D程序

软件方面的要求主要是指操作系统和浏览器。Mac OS和Windows 7及以上版本都可以很好的支持webGL。如果您的机器依旧使用Windows XP操作系统，估计硬件配置也不会很好，不推荐在这样的机器上运行3D程序。

浏览器方面，各种主流的新版浏览器基本都可以很好的支持webGL。包括Windows IE11及以上、Windows Edge、Chrome、Safari、FireFox、Opera等。

- Microsoft IE v11及以上
- Microsoft Edge
- Safari （推荐）
- Chrome （推荐）
- Firefox
- Opera

总结为一句话：**IE必须11以上，Chrome最佳，其他没问题。**

如果还不能确认您的浏览器是否支持webGL，可以直接用浏览器访问页面https://get.webgl.org/。如果能在页面上看到一个旋转的立方体，说明支持webGL，否则就是有问题。

需要提醒您一点：网页3D程序虽然是B/S架构下的产物，但它最终还是运行在您客户端本地机器上。所以上述软硬件要求也是指您桌面本机的配置要求，和服务器没有关系。再强大的服务器主机，对您浏览器的3D程序也是没有任何帮助的。

另外，有一些企业，开发者会在一个虚拟机中创建开发环境。此时，如果虚拟机的配置并没有设置显卡等资源，或配置较低达不到上述要求，也会出现各种问题。所以不建议在虚拟机中运行和开发mono程序。

对开发者的要求

学习mono对开发者的要求并不是很大。但是由于mono是关于“3D”的技术，对大部分开发者来说可能有一定的“恐惧感”。实践证明，克服这一恐惧感并不是难事。

mono是基于h5的，所以开发者必须熟悉js语言。事实上js语言是一个非常简单的脚本语言，几个小时就可以掌握它的基本语法。js语言对比C++、Java等高级面向对象语言有很多不同，建议能够熟悉和理解js中的对象、function和prototype等语言机理。

熟悉html标签和h5的canvas技术对使用好mono会有更多的帮助，但并非必须。

mono包含了什么

虽然mono的产品下载包可能很大，但本质上，mono的核心只有一个mono.js文件，大小只有几百k，无任何外部及第三方js依赖。mono的产品包中包含了文档、demo、MONO DESIGN编辑器等内容，所以体积更大。mono.js是一个经过混淆的js脚本文件，它暴漏了引擎必要的API

开发接口，而混淆了不必要暴露的内部代码实现。mono的源代码并不对开发者开放，是一个闭源的3d引擎商业产品。

mono的产品包中还包含了大量的demo程序。运行并研读这些程序代码，是非常有效的快速学习mono的手段。

mono产品包中还包含了可以运行的MONO DESIGN编辑器程序。它可以用来做房间和各种简单物体的建模、保存、导入导出等。具体使用方法请阅读本站相关文档。

目的

本节的任务和目的是用mono写一个最简单的网页3D程序。这一节花不了几分钟的时间，可以让开发者快速体验如何用mono做一个简单的网页3D程序。

准备工具

首先需要一个js的代码编辑工具。复杂的有微软的Visual Studio、Eclipse、NetBeans等IDE，也有相对简单流行的WebStorm等编辑工具。或者使用最简单的工具：写字板，例如UltraEditor、EditPlus、NotePlus等编辑工具，也足够了。复杂的IDE具有强大的功能和代码组织、调试能力，适合在大型项目中使用。如果只是js初学者，或简单尝试和体验mono的使用，使用写字板程序能更清晰的看清楚程序结构和逻辑，简单方便。

其次需要准备好mono.js文件，它包含在mono产品包的libs目录中。

好了，就这些。

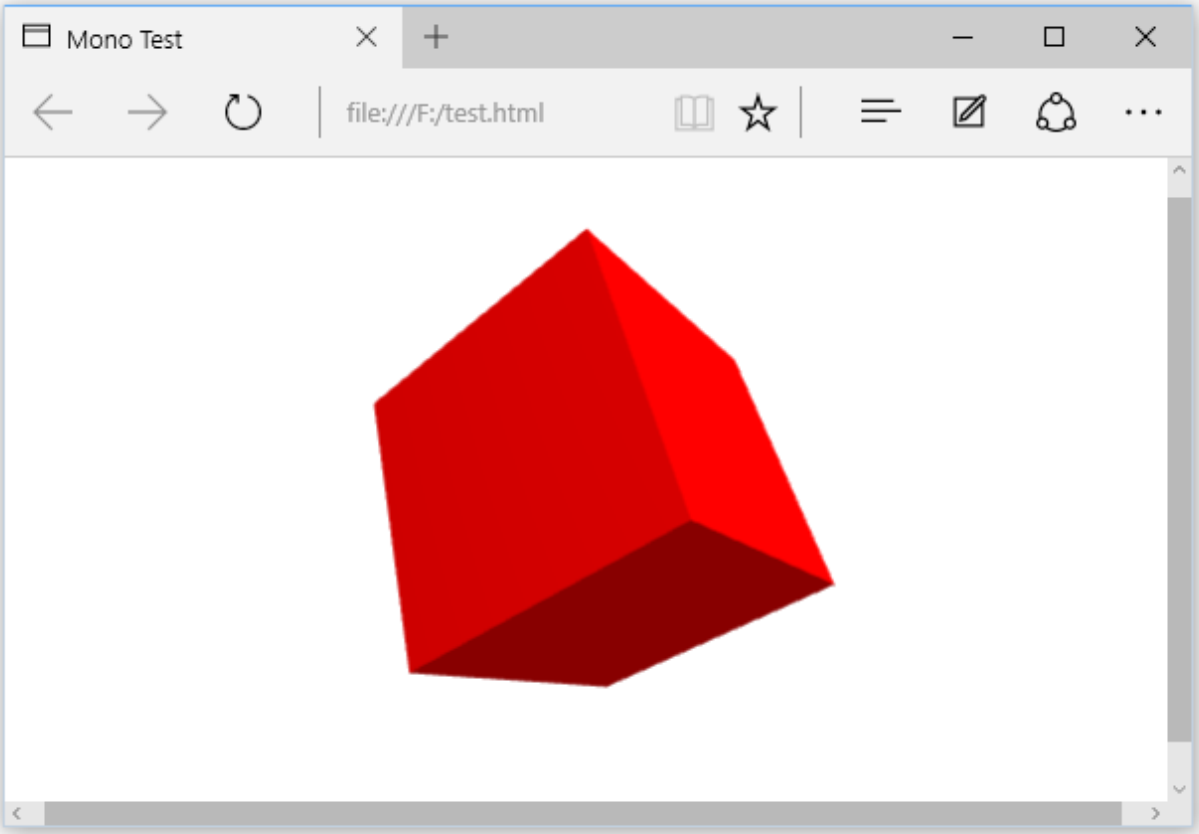
开始写程序

打开写字板，创建一个test.html文件。将下方代码粘贴进文件保存。注意，确保mono.js和test.html两个文件一起放在某个目录下。

```
<!DOCTYPE html>
<html>
<head>
    <title>Mono Test</title>
    <script type="text/javascript" src = "mono.js"></script>
    <script type="text/javascript">

        function init(){
            var box = new mono.DataBox();
            var network= new mono.Network3D(box, null, monoCanvas);
            mono.Utils.autoAdjustNetworkBounds(network,document.documentElement,'clier

            var pointLight = new mono.PointLight(0xFFFFFF,1.5);
            pointLight.setPosition(1000,1000,1000);
            box.add(pointLight);
            box.add(new mono.AmbientLight(0x888888));

            var cube = new mono.Cube(200, 200, 200);
            cube.s({
                'm.type': 'phong',
                'm.color': 'red',
                'm.ambient': 'red',
            });
            cube.setRotation(-Math.PI/10, -Math.PI/5, Math.PI/10);
            box.add(cube);
        }

    </script>
```

```
</head>
<body onload = 'init()'>
    <div>
        <canvas id="monoCanvas"/>
    </div>
</body>
</html>
```

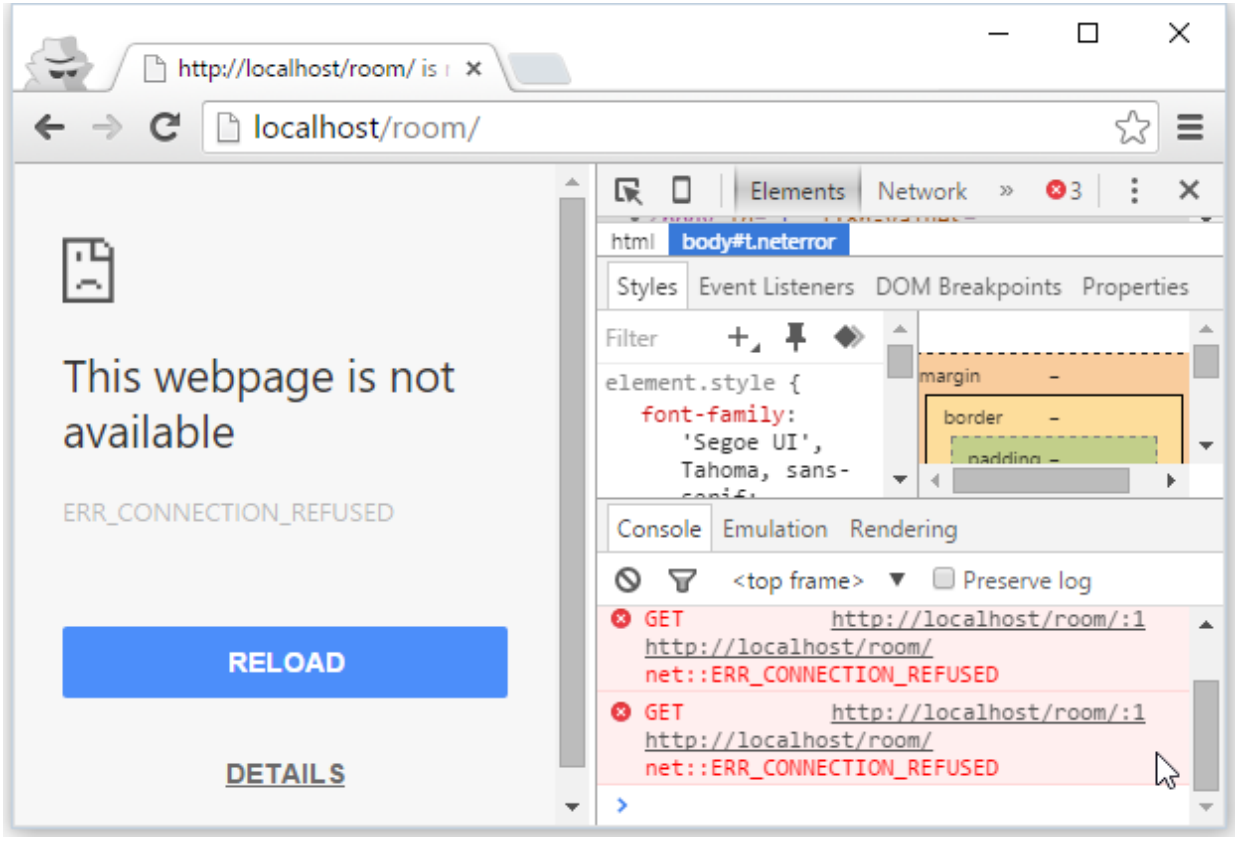
直接双击test.html文件，即可用浏览器打开这个页面。下图是在Windows10上使用默认Edge浏览器打开页面的效果：



如果您能看到截图上的图形，恭喜，您已经完成了第一个mono程序。

> **Note:**
> 请确认默认浏览器程序版本符合mono的要求，尤其是IE，要求IE11或Edge，低版本IE不能支持webGL。如果不能正常看到上述程序，请检查浏览器程序及版本，是否符合mono要求。

如果发现程序异常，在浏览器中可以直接按F12查看异常信息，进行错误定位和调试：



程序说明

简单解释一下这个程序的内容。test.html本质是一个html网页文件，里面只包含了一个普通的html必须的元素，包括title、body，body中放了一个div，div中放了一个canvas标签。另外，head中放了两个script标签，第一个引入了mono.js文件，第二个则直接写了一个init全局函数，其中的代码来创建3D场景。init函数在body的onload中进行调用，也就是页面加载结束后，浏览器会立刻执行init函数。

init函数包含了mono最简单的使用方法。下面的注释逐行解释了每一句代码的作用：

```
<!DOCTYPE html>
<html>
<head>
    <title>Mono Test</title>
    <script type="text/javascript" src = "mono.js"></script>
    <script type="text/javascript">

        function init(){
            //创建数据容器，容纳所有3d对象
            var box = new mono.DataBox();
            //创建3d画布，显示3d场景。monoCanvas是html页面中canvas标签对象的id值
            var network= new mono.Network3D(box, null, monoCanvas);
            //当页面调整大小时，自动调整network画布宽度，铺满整个页面
            mono.Utils.autoAdjustNetworkBounds(network,document.documentElement,'clier

            //在坐标1000、1000、1000处放置一个颜色为0xFFFFFF色、强度为1.5的点光源
            var pointLight = new mono.PointLight(0xFFFFFF,1.5);
            pointLight.setPosition(1000,1000,1000);
            box.add(pointLight);
            //在场景中增加颜色为0x888888的环境光
            box.add(new mono.AmbientLight(0x888888));

            //创建一个大小为200、200、200的立方体
            var cube = new mono.Cube(200, 200, 200);
```

```
            //设置立方体材质、颜色等样式
            cube.s({
                'm.type': 'phong',
                'm.color': 'red',
                'm.ambient': 'red',
            });
            //设置立方体在x、y、z轴向的旋转角度
            cube.setRotation(-Math.PI/10, -Math.PI/5, Math.PI/10);

            //将立方体置入databox数据容器进行显示
            box.add(cube);
        }

    </script>
</head>
<body onload = 'init()'>
    <div>
        <canvas id="monoCanvas"/>
    </div>
</body>
</html>
```

程序主要分为几个步骤：

- 创建容器和画布。通过代码new一个mono.DataBox和mono.Network3D即可。databox是放置所有3D对象的容器，是使用mono必须创建的对象。即使不显式的new实例，在new mono.Network3D时其内部也会创建一个databox实例。您必须掌握的概念是：databox是装载所有3D对象的容器，而network则是显示3D场景的画布。两个对象一个负责数据管理，一个负责图形绘制，搭配使用，相互配合，缺一不可。程序中还使用了autoAdjustNetworkBounds这个工具函数保证network始终铺满页面
- 设置灯光。3D场景中，必须设置灯光才能看到场景，否则，3D场景会“漆黑一片”，这和现实世界的逻辑相同。灯光分多种，简单而言，常用的就2类：点光源和环境光。点光源模拟了一个类似“灯泡”一样的光源，环境光则模拟了我们现实世界的环境光。一般应同时搭配使用，增强3D场景的光照感和逼真度。代码中使用了一个点光源和一个环境光，并设置了光的颜色、强度等属性
- 添加3D物体。mono中3D物体都进行了面向对象化的封装，例如我们直接new mono.Cube就可以创建一个立方体。设置大小、颜色、位置、旋转、材质等信息，使用“box.add(cube)”的方式添加到数据容器中，就完成了3D物体的创建和显示

请认真阅读和理解上面代码中每一行代码的作用和含义。当然，html中的每一个标签的含义也应该做到完全理解。如果您理解了以上全部内容，就基本上理解了mono开发3D程序的基本原理和过程，可以做mono开发了。其他再复杂的3D场景，也是基于相同的基本原理和过程的，只不过数据更多、属性样式更复杂而已。

总结

可见，十几行js代码就能创建一个最简单的mono程序了，任何一个程序员在很短时间就可以掌握mono的开发。再次总结编写一个最简单的mono程序步骤：1、创建box容器和network画布；2、创建灯光；3、创建3D物体。

为什么复习数学

向量、矩阵是数学中的基础概念。虽然大家在大学时候肯定都学习过《线性代数》高中就学习过向量，并可能考试得过高分，但很可能现在已经都还给了老师了。一般的程序开发可能用不到向量和矩阵，但在3D技术中，这些概念却是非常重要的基础知识。

这里我们不打算把整个《线性代数》重新学一遍，使用mono也不需要那么深入的数学知识。这里只是在使用mono过程中可能用到的基础概念，简单、快速的进行重新回顾，让大家重新掌握这些基础数学知识。

如果对这些知识还比较熟悉，或者还没有耐心复习这些枯燥的数学概念，可以先跳过本章节，待以后再阅读。

向量

向量的定义和基本概念

数量的定义：数学中，把只有大小但没有方向的量叫做数量（或纯量），物理中常称为标量。

向量的定义：既有大小又有方向的量叫做向量（亦称矢量）。在线性代数中的向量是指n个实数组成的有序数组，称为n维向量。α=（a1，a2，…，an）称为n维向量.其中ai称为向量α的第i个分量。（”a1″的”1″为a的下标，”ai”的”i”为a的下标,其他类推）。

几何表示：向量可以用有向线段来表示。有向线段的长度表示向量的大小，箭头所指的方向表示向量的方向。（若规定线段AB的端点A为起点，B为终点，则线段就具有了从起点A到终点B的方向和长度。这种具有方向和长度的线段叫做有向线段。）

坐标表示：在平面直角坐标系中，分别取与x轴、y轴方向相同的两个单位向量i，j作为基底。a为平面直角坐标系内的任意向量，以坐标原点O为起点作向量OP=a。由平面向量基本定理知，有且只有一对实数（x，y），使得a=向量OP=xi+yj，因此把实数对（x，y）叫做向量a的坐标，记作a=（x，y）。这就是向量a的坐标表示。其中（x，y）就是点P的坐标。向量OP称为点P的位置向量。

向量的模和向量的数量：向量的大小，也就是向量的长度(或称模)。向量a的模记作|a|。

注：
1、向量的模是非负实数，是可以比较大小的。
2、因为方向不能比较大小，所以向量也就不能比较大小。对于向量来说“大于”和“小于”的概念是没有意义的。例如，“向量AB>向量CD”是没有意义的。

单位向量：长度为单位1的向量，叫做单位向量．与向量a同向且长度为单位1的向量，叫做a方向上的单位向量，记作a0，a0=a/|a|。

零向量：长度为0的向量叫做零向量，记作0。零向量的始点和终点重合，所以零向量没有确定的方向，或说零向量的方向是任意的。

相等向量：长度相等且方向相同的向量叫做相等向量，向量a与b相等，记作a=b。所有的零向量都相等。
当用有向线段表示向量时，起点可以任意选取。任意两个相等的非零向量，都可用同一条有向线段来表示，并且与有向线段的起点无关．同向且等长的有向线段都表示同一向量。

自由向量：始点不固定的向量，它可以任意的平行移动，而且移动后的向量仍然代表原来的向量。在自由向量的意义下，相等的向量都看作是同一个向量。

相反向量：与a长度相等、方向相反的向量叫做a的相反向量，记作-a。有-（-a）=a；零向量的相反向量仍是零向量。

平行向量：方向相同或相反的非零向量叫做平行（或共线）向量．向量a、b平行（共线），记作a∥b。

向量的运算

以下介绍向量的基本运算。介绍之前，假设：

设a=（x，y），b=(x’，y’)

向量的加法

向量的加法满足平行四边形法则和三角形法则：
AB+BC=AC
a+b=(x+x’，y+y’)
a+0=0+a=a

向量加法的运算率：

交换律：a+b=b+a
结合律：(a+b)+c=a+(b+c)

mono中的相应函数，mono.Vec2/mono.Vec3/mono.Vec4均适用：

```
1 //向量加
2 mono.Vec2#add
```

向量的减法

如果a、b是互为相反的向量，那么a=-b，b=-a，a+b=0

AB-AC=CB

a-b=(x-x',y-y')

mono中的相应函数，mono.Vec2/mono.Vec3/mono.Vec4均适用：

```
//向量减
mono.Vec2#sub
```

向量的数乘

实数λ和向量a的乘积是一个向量，记作λa，且|λa|=|λ|·|a|。

- 当λ > 0时，λa与a同方向；
- 当λ < 0时，λa与a反方向；
- 当λ=0时，λa=0，方向任意。
- 当a=0时，对于任意实数λ，都有λa=0。

> 注：
> 按定义知，如果λa=0，那么λ=0或a=0。

实数λ叫做向量a的系数，乘数向量λa的几何意义就是将表示向量a的有向线段伸长或压缩。

- 当|λ| > 1时，表示向量a的有向线段在原方向（λ > 0）或反方向（λ < 0）上伸长为原来的|λ|倍；
- 当|λ| < 1时，表示向量a的有向线段在原方向（λ > 0）或反方向（λ < 0）上缩短为原来的|λ|倍。

数与向量的乘法满足下面的运算律：

- 结合律：(λa)·b=λ(a·b)=(a·λb)。
- 向量对于数的分配律（第一分配律）：(λ+μ)a=λa+μa.
- 数对于向量的分配律（第二分配律）：λ(a+b)=λa+λb.
- 数乘向量的消去律：①如果实数λ≠0且λa=λb，那么a=b。②如果a≠0且λa=μa，那么λ=μ。

mono中的相应函数，mono.Vec2/mono.Vec3/mono.Vec4均适用：

```
//向量数乘
mono.Vec2#multiply
```

向量的点乘

数量积（dot product; scalar product，也称为点积）是接受在实数R上的两个向量并返回一个实数值标量的二元运算。

代数定义：

两个向量a = [a1, a2,…, an]和b = [b1, b2,…, bn]的点积定义为：

a·b=a1b1+a2b2+……+anbn。

几何定义：

两个向量的数量积（内积、点积）是一个数量，记作a·b。
a·b=|a|·|b|·cos〈a，b〉

由此可以得出：如果ab夹角为90度，则cos=0，点积也为零，向量AB*向量AC=向量AB的模*向量AC的模*cosΘ，因为垂直，所以Θ=π/2，cosπ/2=0。因此相互垂直的向量点积为零。

向量的数量积的坐标表示：a·b=x·x'+y·y'。

向量的数量积的运算率：

- a·b=b·a（交换率）
- （a+b)·c=a·c+b·c（分配率）

向量的数量积与实数运算的主要不同点：

- 向量的数量积不满足结合律，即：(a·b)·c≠a·(b·c)；例如：(a·b)^2≠a^2·b^2。
- 向量的数量积不满足消去律，即：由a·b=a·c(a≠0)，推不出b=c。
- |a·b|≠|a|·|b|
- 由|a|=|b|，推不出a=b或a=-b。

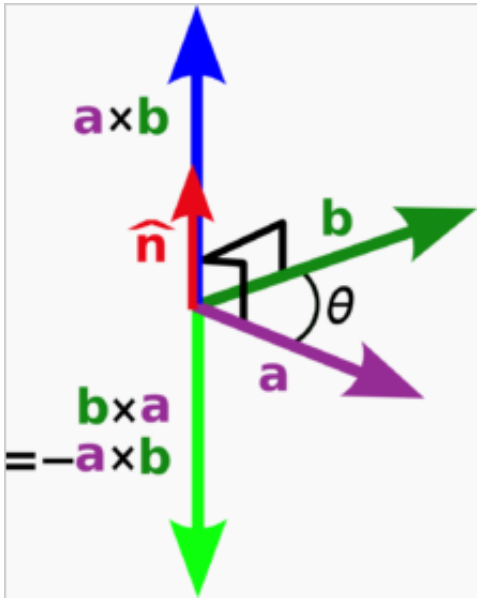
mono中的相应函数，mono.Vec2/mono.Vec3/mono.Vec4均适用：

```
1 //向量点乘
2 mono.Vec2#dot
```

向量的叉乘

向量积，或称外积、叉积、叉乘，与点积不同，它的运算结果是一个向量而不是一个标量。并且两个向量的叉积与这两个向量的和垂直。定义为：

定义：两个向量a和b的向量积（外积、叉积）是一个向量，记作a×b。若a、b不共线，则a×b的模是：|a×b|=|a|·|b|·sin〈a，b〉；a×b的方向是：垂直于a和b，且a、b和a×b按这个次序构成右手系。若a、b共线，则a×b=0。



向量的向量积性质：

- |a×b|是以a和b为边的平行四边形面积
- a×a=0
- a∥b〈=〉a×b=0

向量的向量积运算律：

- a×b=-b×a
- （λa）×b=λ（a×b）=a×（λb）
- （a+b）×c=a×c+b×c

> 注：
> 向量没有除法，“向量AB/向量CD”是没有意义的。

mono中的相应函数，mono.Vec2/mono.Vec3/mono.Vec4均适用：

```
//向量叉乘
mono.Vec2#cross
```

在mono中使用向量

在mono中，二维向量可以用mono.Vec2这个类，三维向量可以使用mono.Vec3。平面或空间的一个点、一个轴，都可以用一个二维或三维向量来表达。由于mono是3D引擎，因此使用mono.Vec3最为普遍。

例如，空间（x=100，y=200，z=300）这个点可以用如下向量表达：

```
var point = new mono.Vec3(100, 200, 300);
```

它是一个空间的点，也是一个从坐标原点到这个点的一个向量。

同样，如果一个轴线，例如y轴，可以用new mono.Vec3(0, 1, 0)来表示。

> 注意：
> 1、在mono中，一个空间的点、一根轴（也就是一个方向），都是一个向量。请务必理解这一概念。
> 2、在mono中，一个mono.Vec3所表达的是一个三维自由向量。和固定向量不同，自由向量只确定方向与大小，而不在意位置。因此Vec3所表达的向量起始点都是坐标原点。

在mono中，向量的运算都定义在了mono.Vec2/Vec3/Vec4中。罗列如下：

```
//加
mono.Vec2#add
//减
mono.Vec2#sub
//数乘
mono.Vec2#multiply
```

```
//点乘
mono.Vec2#dot
//叉乘
mono.Vec2#cross
```

同样，对于mono.Vec3、mono.Vec4以上函数也适用。

技巧：绕轴旋转的物体

任务：让一个物体绕指定的位置的一个轴做某个角度的旋转

思路：这一任务完全可以通过向量来解决。我们可以使用向量Vec3的内置函数rotateFromAxisAndCenter函数来实现。这个函数的作用是：让当前向量绕指定位置的指定轴进行指定角度进行旋转，并返回新的位置向量。解释如下：

```
//函数定义
mono.Vec3#rotateFromAxisAndCenter : axis, angle, center
//axis: 旋转的轴。例如new mono.Vec3(0, 1, 0)表示垂直轴。注意这个轴向量是一个自由向量，没有
//angle: 要旋转的角度。
//center: 旋转的轴的具体角度。例如new mono.Vec3(200, 300, 500)表示轴axis在x=200、y300
```

下面代码让立方体绕垂直y轴在x=50、z=100的位置逆时针不停旋转：

```
var node = new mono.Cube(100, 100, 100);
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
});

var rotationFunc = function(){
  var position = cube.getPosition();

  var axis = new mono.Vec3(0, 1, 0);
  var angle = Math.PI/180;
  var center = new mono.Vec3(50, 0, 100);
  var newPosition = position.rotateFromAxisAndCenter(axis, angle, center);

  cube.setPosition(newPosition);
}

setInterval(rotationFunc, 10);
```

此外，在mono的任意3D对象上都提供了Element#rotateFromAxis(axisDir, axisPosition, angle)方法，也可以实现完全相同的效果。仅仅是函数名称、参数略有不同而已。

```
/*
* 针对Element的操作。定轴旋转，返回旋转后的向量。
* axisDir: 轴的方向，比如 vec3(0,1,0) 表示垂直向上的y轴方向
* axisPosition: 表示轴的位置，比如vec3(1，0，0)，则axis + center 表示的就是在x为1的位置
* angle: 要旋转的角度
*/
Element#rotateFromAxis(axisDir, axisPosition, angle)
```

矩阵

> 矩阵是3D中的重要基础概念。不过由于mono对于大部分相关运算都做了封装，所以开发者一般不需要直接使用矩阵API。但理解矩阵的概念和基本运算，会对更好的使用mono有积极的作用。

在数学中，矩阵（Matrix）是一个按照长方阵列排列的复数或实数集合 ，最早来自于方程组的系数及常数所构成的方阵。这一概念由19世纪英国数学家凯利首先提出。 矩阵是高等代数学中的常见工具，也常见于统计分析等应用数学学科中。在物理学中，矩阵于电路学、力学、光学和量子物理中都有应用；计算机科学中，三维动画制作也需要用到矩阵。

定义

由 m × n 个数aij排成的m行n列的数表称为m行n列的矩阵，简称m × n矩阵。记作：

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ a_{31} & a_{32} & \cdots & a_{3n} \\ \cdots & \cdots & & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

这m×n 个数称为矩阵A的元素，简称为元，数aij位于矩阵A的第i行第j列，称为矩阵A的(i,j)元，以数 aij为(i,j)元的矩阵可记为(aij)或(aij)m × n，m×n矩阵A也记作Amn
元素是实数的矩阵称为实矩阵，元素是复数的矩阵称为复矩阵。而行数与列数都等于n的矩阵称为n阶矩阵或n阶方阵[5]。

运算

矩阵的基本运算包括矩阵的加法，减法，数乘，转置，共轭和共轭转置。

mono中的矩阵

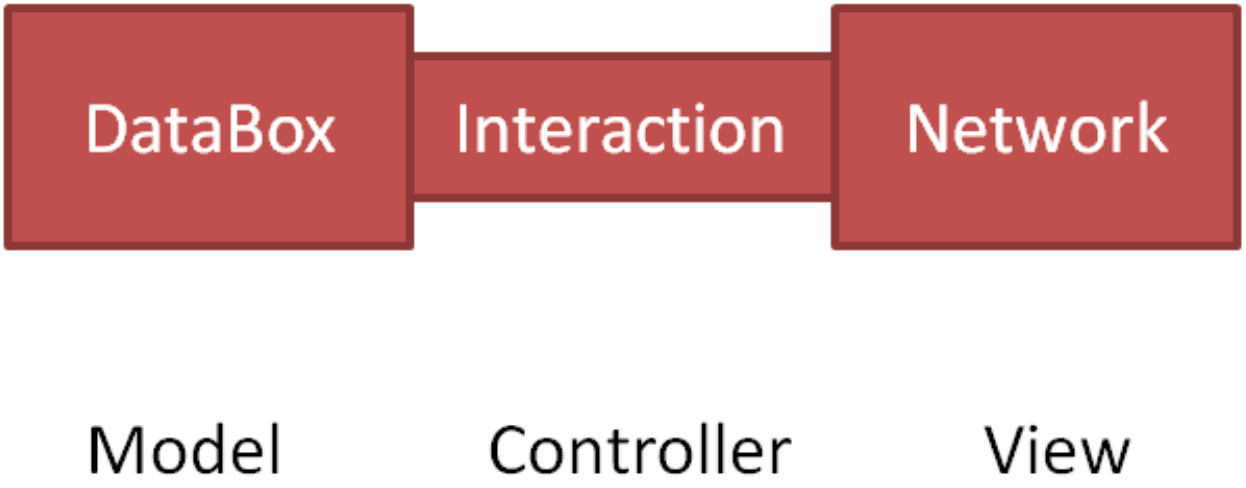
在mono中，用mono.Mat3/mono.Mat4表示2阶/3阶矩阵。它封装了矩阵的基本运算。

MVC设计架构

您需要理解mono以及twaver产品的每一个分支都是基于“MVC”（或其变种）的设计架构。这对您日常使用mono进行开发非常重要。

这是一个“架构泛滥”的年代，各种类似MVC的热词满天飞，您可能很不明觉厉，或不以为然。其实不必大谈各种架构的理论和好处，我们只是用很朴素的想法来理解mono的MVC设计就行了：

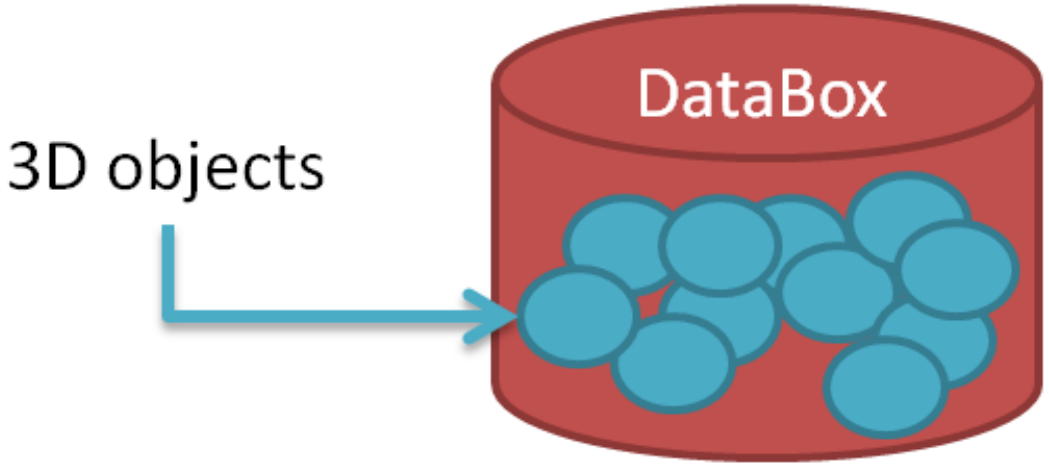
mono采用MVC的思路把数据、绘制、交互三者进行了分离设计。



具体说：

- M——Model，数据：它主要是指mono中的databox容器，及立方体/球体/圆柱体等等一系列基础数据对象。容器负责装载和管理这些3D数据对象。这些，都是纯粹的数据定义和管理，就和一个Array数组与一堆Object对象的意思一样，只是更复杂一些而已。
- V——View，视图：它主要是指mono中的network画布对象，它代表了数据的一个“3D呈现的视图”，作用是“如何用一个角度去表达/绘制这些databox中的对象”。视图的主要任务是“绘制”，用自己的方式（例如3D）进行绘制，把绘制的细节封装起来，让使用者更简单。
- C——Controller，控制器：控制器是指负责在视图和数据之间交互的控制部分。例如：数据变化后，如何让视图立刻实现通知和更新、怎么更新？用户拖拽了一个物体位置后，如何将数据的变化同步给databox？等等。控制器是负责数据的变化同步，以及用户和视图之间的各种交互。典型的部分是mono中Interaction相关的类和接口。

DataBox是一个对象管理容器。所有要显示的3D对象都放置在这个容器中统一管理。容器会负责3D对象的增、删、查等工作，同时监听数据的变化并通过事件监听机制进行广播通知。Network则是负责具体显示的画布，它必须连接到一个具体DataBox容器，并将容器中的数据绘制到canvas画布上。DataBox和Network作为数据管理者和画布渲染者独立工作、各司其职。



MVC的组织方式让程序更加的灵活，让类的组织更清晰、容易理解、容易扩展。反映到mono上，我们可以直接带来的灵活性和好处有：

- 可以一个databox连接多个network视图，实现一套对象多出呈现的要求。每个network可以显示不同的样式和风格，而内存中却只有一套数据和一个databox。
- 可以只关注业务数据的变化，而不关心绘制细节。例如，一个物体的颜色变了，我们直接“cube.setStyle(‘m.color’, ‘red’)”即可实现立刻更新，而不用关心“如何通知、如何重绘”等细节。
- 数据的各种变化，都可以实现灵活监听。例如一个对象的任何属性变化、场景中数据的创建和删除等，都可以通过监听容器完成。
- 可以灵活定制交互，而无需修改network或databox。例如，设置一个“可移动Interaction”就可以允许用户直接通过鼠标移动物体，设置“可选择Interaction”就可以允许选中物体，等等。

MVC带来的灵活性还有很多。无论我们是否喜欢，MVC都是无处不在，是整个软件行业的基础设计思路之一。mono中的MVC并不复杂，对于初学者，我们只要时刻掌握这样的概念即可：

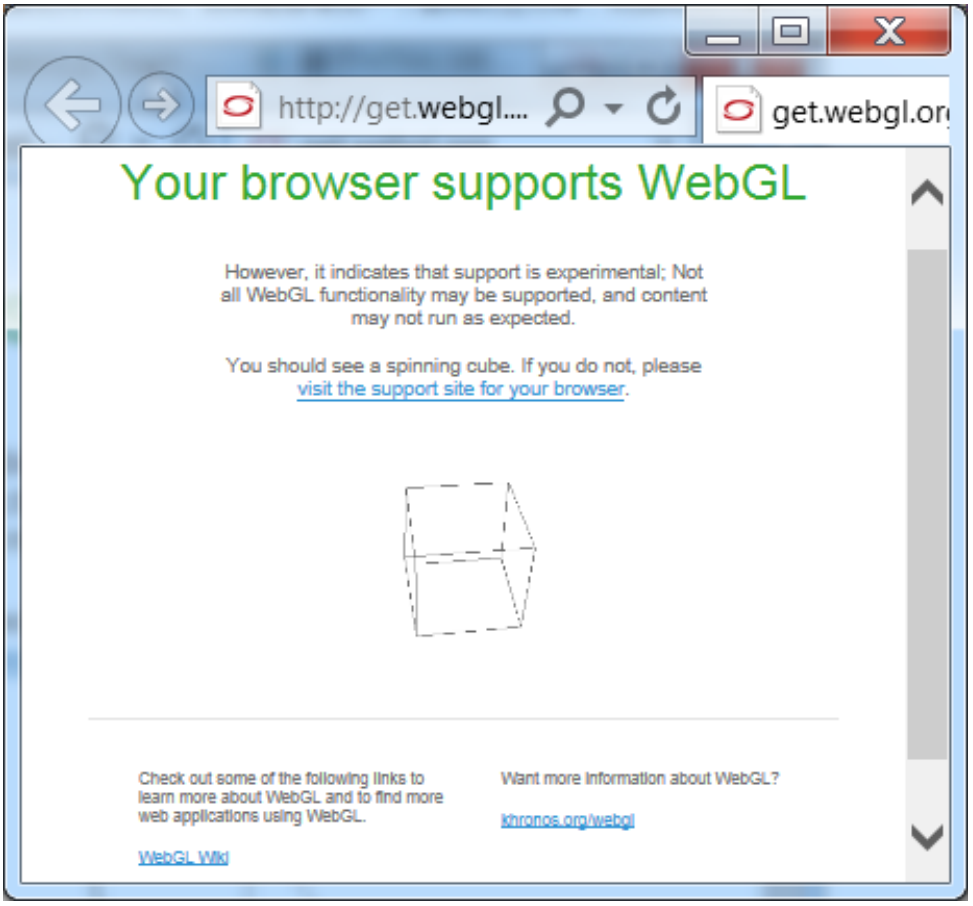
databox是容器，network是视图，interaction是交互，分别代表M、V、C就行了，不必理会各种天花乱坠的“玄妙”定义。

支持浏览器

开发者需要在支持WebGL的浏览器上调试运行mono程序。包括：

- Chrome v30及以上
- IE v11及以上
- Safari
- FireFox
- Opera

可以点击网址get.webgl.org来检查浏览器是否支持WebGL。如果能够看到一个旋转的立方体，则说明浏览器支持WebGL（如下图）。



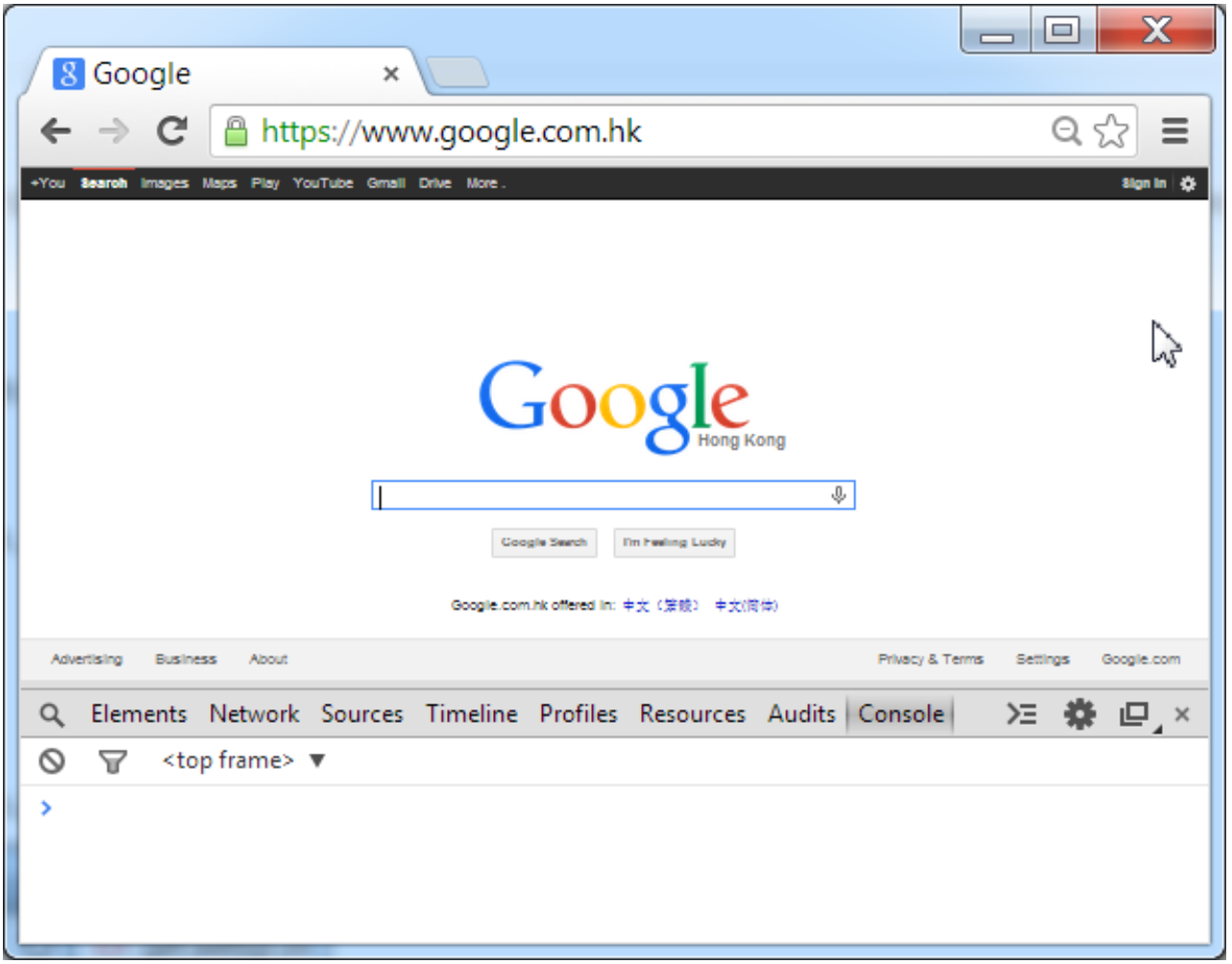
开发工具及调试

JavaScript开发可以选择使用Eclipse、NetBeans、Visual Studio等大型IDE工具，也可以选择直接使用文本编辑工具如EditPlus、UltraEdit等。前者可以提供一些自动提示、集成调试等功能，缺点是程序比较庞大笨重。后者则轻量快速直接，但无自动提示等功能。开发者可以根据自身情况合理选择代码编辑工具。

对于开发测试浏览器，推荐使用Google的Chrome。Chrome速度快、对WebGL支持好、调试方便，是WebGL开发者首选浏览器。

在部署测试页面程序时，可以选择用IDE中的内置的调试方法，也可以直接放入如Tomcat等Web服务器中并用浏览器直接访问。

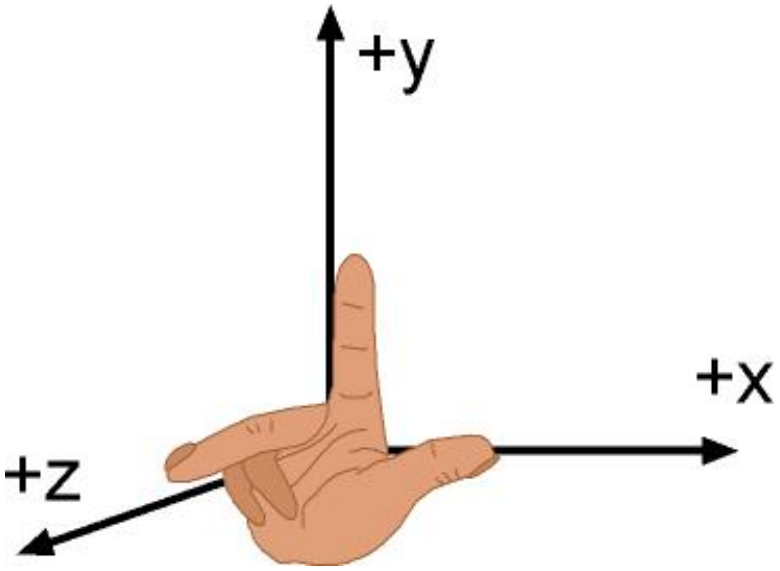
Chrome浏览器提供了内置的调试工具，非常方便。可以直接按F12打开开发工具窗口，其中提供了控制台、查看页面元素、断点跟踪、性能测试等功能（见下图）。



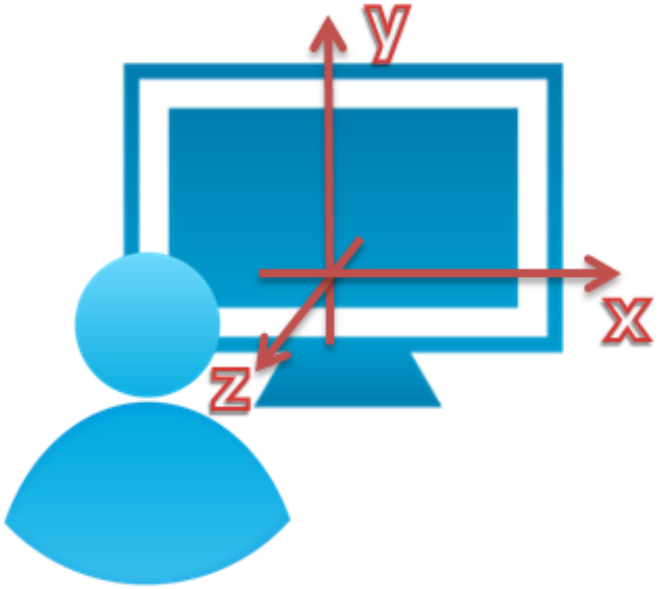
更多关于Chrome开发工具的功能介绍，请参考Google官方文档：https://developers.google.com/chrome-developer-tools/

理解坐标系

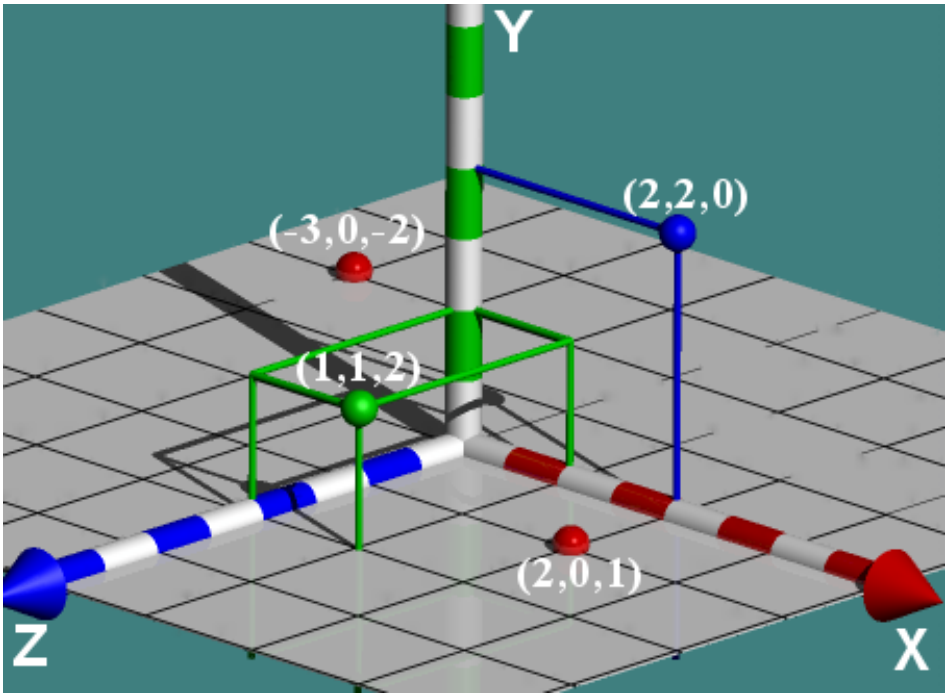
在mono中，3D场景的坐标是一个空间直角坐标系。直角坐标系是指在原点O，做三条互相垂直的数轴，它们都以O为原点，分别叫做x轴(横轴）、y轴(纵轴)、z轴(竖轴)，统称坐标轴。在mono中，坐标系方向遵循右手法则，即以右手拇指、食指、中指相互交叉，拇指指向x轴正方向，食指指向y轴正方向，则中指指向的即是z轴正方向。



简单说：**从左向右的水平轴是x轴，垂直向上的轴是y轴，从屏幕指向人面部的水平轴是z轴。**再次示意如下图：

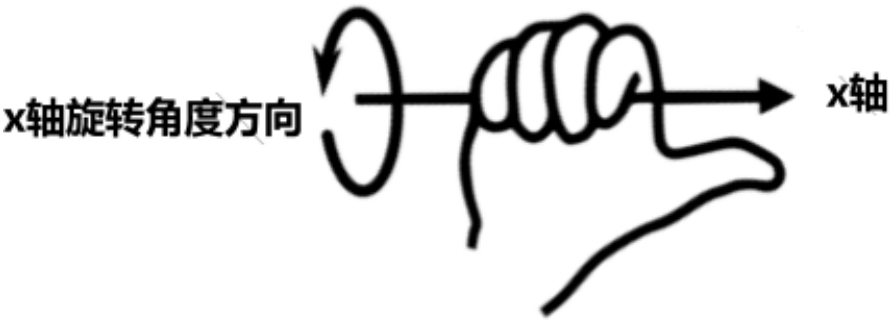


在mono中，空间的一个点由(x, y, z)三个数值组成。下图展示了不同数值其具体表示的点的位置。熟练掌握坐标点的位置，对mono开发具有重要的作用。请认真仔细观察以下图中空间几个点的位置和对应坐标，是否和您的理解一致。如果能够正确理解，则说明已经正确掌握了mono中3D的坐标概念。



除了位置以外，3D物体还有旋转角度。一个3D物体可以在x、y、z三个轴向分别进行旋转。

旋转角度的方向遵守“右手法则”：右手拇指指向要旋转的轴，其余四指的方向，就是角度旋转的方向。如下图：



如果旋转角度为正值，则沿图中箭头方向旋转；如为负值，则反方向旋转。

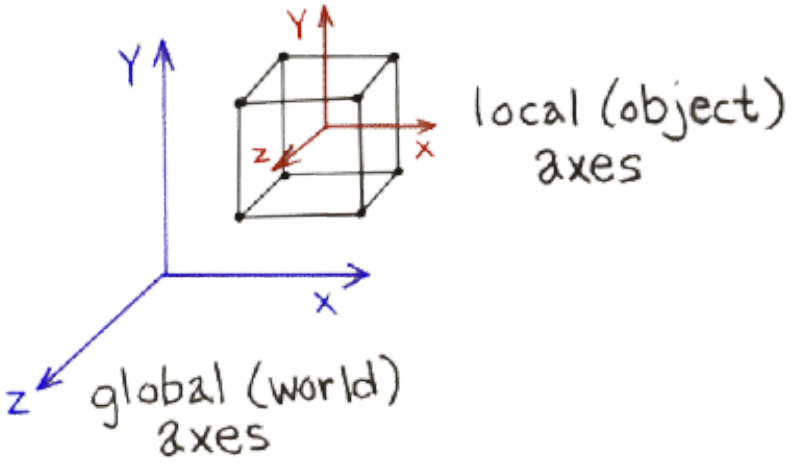
世界坐标系与本地坐标系

3D中的坐标系分为世界坐标系和本地坐标系。世界坐标系（World Coordinate System）是系统的绝对坐标系，是指所有3D物体所在的空间中全局的、绝对的坐标。而本地坐标系（Local

Coordinate System ）则是指相对以某一3D物体自身为原点的局部坐标系，物体的旋转或平移等操作都是围绕局部坐标系进行的。

例如一个飞速行走中的汽车，汽车的位置信息可以通过世界坐标系来定义其在地面空间的位置，而汽车雨刮的摆动位置和角度则是相对汽车的本地坐标系进行指定。这样有利于使用和理解的方便。

世界坐标系和本地坐标系可以进行转换。



数值单位

在mono中，所有的角度数值都统一用弧度表示。即：Math.PI表示180度，一周360度则为2*Math.PI。例如：

```
1 node.setRotation(Math.PI, Math.PI/2, Math.PI);
```

对于长度则没有具体单位。例如空间（100，200，300）的点表示其x值为100、y值为200、z值为300，但单位是厘米、米还是其他长度单位，mono是没有定义的，也没有实际意义，它只是逻辑上的数值而已。我们可以根据应用自行设定长度所代表的具体单位。

```
1 node.setPosition(100, 200, 300);
```

DataBox的作用

和twaver的2D产品概念类似，DataBox和Network是mono中最重要的基础对象。DataBox作为容器负责3D对象的管理，Network作为3D画布负责3D的渲染和呈现。

简单理解DataBox：它是一个不可见的内存容器对象，负责管理所有的3D对象，像一个复杂一点的数组，像一个简单一点的内存数据库。

使用DataBox

DataBox是管理所有数据对象的容器，在MVC框架中处于M（模型）层，它是视图的数据提供者，它的主要功能是对数据进行装载、卸载，并对数据元素的变化进行监听。

使用DataBox非常简单直接new一个mono.DataBox实例即可。下面的代码创建了一个DataBox和一个立方体对象，并添加到容器中：

```
1 var box = new mono.DataBox();
2
3 var cube = new mono.Cube(10, 10, 10);
4 box.add(cube);
```

需要理解的是：上面的代码并没有Network画布对象出现。DataBox也并不依赖Network而存在，反过来Network则必须依赖DataBox才能存在。这也容易理解：数据是客观存在的，不管是否显示出来。只有当数据需要被3D画布（Network）画到屏幕上的时候，才需要用到Network。

继续创建Network：

```
1 var network = new mono.Network3D(box, camera, myCanvas);
```

上面代码会创建一个network画布并和之前的box连接起来，渲染到HTML页面id为'myCanvas'的canvas标签上。

数据操作

DataBox最主要的任务是数据对象的增加、删除、查找、变化监听。其中，监听包括了数据出入的监听、每一个数据的属性变化的监听。

主要API罗列：

- add(data, index) 往数据容器中添加一个数据对象。一般index不用设置，可忽略

- addByDescendant(data) 把一个数据对象连其子孙对象一起添加到容器中
- remove(data) 把一个数据对象从容器中删除
- removeById(id) 删除指定id的数据对象
- contains(data) 判断容器是否包含某个数据对象
- containsById(id) 判断容器是否包含指定id的数据对象
- getDataById(id) 获得容器中指定id的数据对象
- getDatas() 返回容器所有数据对象
- getDataAt(index) 返回容器指定index位置的数据对象。一般index不常用
- size() 返回容器中数据对象的总数量
- lightsSize() 返回数据容器中光源对象的总数量
- getLights() 返回容器中所有的光源对象
- clear() 清空数据容器。所有数据对象从容器中清除
- removeSelection 把当前处于“选中状态”的数据对象从容器中清除
- clearSelection 清除当前所有处于“选中状态”数据对象的选中状态，让它们处于非选中状态。注意数据对象不会从容器中删除（留意和removeSelection的区别）
- clearEditing 取消所有当前正在处于“编辑状态”数据对象的编辑状态，让它们处于正常的非编辑状态

以下一段程序展示了对DataBox中数据的增删改查：

```
var box = new mono.DataBox();
for(var i = 0; i < 3; i++) {
    var element = new mono.Element("element_" + i);
    element.setName("element_" + i);
    box.add(element);
}
box.forEach(function(element) {
    console.log(element.getName()); //element_0,element_1,element_2
});
var element0 = box.getDataAt(0);
element0.setName("newElement");
console.log(element0.getName());//newElement
box.remove(element0);
console.log(box.size());//3
box.clear();
console.log(box.size());//0
```

事件监听

DataBox可以对容器中发生的事件进行监听。主要有两类事件：1、数据的变化；2、数据属性的变化。

数据的变化

数据的变化包括数据的添加、删除、清空。用box的addDataBoxChangeListener函数可以添加监听器对这些事件进行监听。下面代码展示了如何使用：

```
box.addDataBoxChangeListener(function(e){
```

```
  if(e.kind === 'add') {
    var newElement = e.data;
    console.log('new element added: '+ newElement.getId());
  }
});
```

监听器会收到事件对象e，事件上的kind是事件类型：

- e.kind === 'add': 有对象被添加。新对象是e.data
- e.kind === 'remove': 有对象被删除。删除的对象是e.data
- e.kind === 'clear': 容器所有对象被清空。

通过removeDataBoxChangeListener函数删除监听器对象。

```
var listener = function(e){
//...
}
box.addDataBoxChangeListener(listener);
//...
box.removeDataBoxChangeListener(listener);
```

数据属性的变化

DataBox不但能监听数据的进出的变化，还能监听box中每一个数据对象的任意一个属性的变化。数据属性的变化包括name、tooltip等基本属性的变化、style属性的变化、client属性的变化。数据的变化包括数据的添加、删除、清空。用box的addDataPropertyChangeListener函数可以添加监听器对任意数据对象的任意属性变化进行监听。

属性变化监听器会传入一个事件对象e，它的主要属性有：

- e.source：发生属性变化的数据对象，例如一个立方体对象；
- e.property：变化的属性名称。例如立方体的名称发生变化，则property的值是'name'；
- e.oldValue：变化前的数值；
- e.newValue：变化后的数值；

有了这些信息，就可以方便的监听到某个数据发生的变化，并根据需要进行相应的动作。

当某些属性发生变化后，打印调试日志、提交后台进行保存等等，都是非常常见的典型用法。

下面代码利用这一方法，禁止用户修改一个物体的垂直位置，让它始终保持在地面上：

```
box.addDataPropertyChangeListener(function(e){
  var element = e.source, property = e.property, oldValue = e.oldValue, newValue
  if(property == 'position' && oldValue.y != newValue.y){
    element.setPositionY(oldValue.y);
  }
});
```

每当y发生变化，就立刻再次把它设置回去。这样就阻止了y发生变化的可能。

导入导出json

DataBox支持把内部的对象导出为json字符串，也支持把json字符串定义的数据导入到DataBox中。使用mono.JsonSerializer这个类即可完成这一工作。

创建JsonSerializer：

```
//为指定box对象创建一个序列化器
var jsonSerializer = new mono.JsonSerializer(box);
```

序列化box中的数据：

```
//把box中的所有对象序列化到json字符串中
var json = jsonSerializer.serialize();
console.log(json);
```

打印结果类似下方格式：

{“v”:”2.0.6",”platform”:”html5",”dataBox”:{“class”:”TGL.DataBox”},”datas”:[{“class”:”TGL.PointLight”,”ref”:0,”p”:{“position”:{“x”:1000,”y”:1000,”z”:1000},”color”:{“r”:1,”g”:1,”b”:1},”ambient”:{},”diffuse”:{},”specular”:{},”intensity”:1.5}},{“class”:”TGL.PointLight”,”ref”:1,”p”:{“position”:{“x”:-1000,”y”:-1000,”z”:-1000},”color”:{“r”:1,”g”:1,”b”:1},”ambient”:{},”diffuse”:{},”specular”:{},”intensity”:2}},{“class”:”TGL.AmbientLight”,”ref”:2,”p”:{“color”:{“r”:0.5333333333333333,”g”:0.5333333333333333,”b”:0.5333333333333333},”ambient”:{},”diffuse”:{},”specular”:{}}},{“class”:”TGL.Cube”,”ref”:3,”p”:{“width”:100,”height”:100,”depth”:100},”s”:{“m.type”:”phong”,”m.texture.image”:”box.jpg”}}]}

反序列化json字符串。可以指定返回的对象放在某个对象的下方（孩子）：

```
//把json字符串反序列化，并作为rootParent对象的孩子
jsonSerializer.deserialize(json,rootParent)
```

导出的json格式看上去很凌乱、不宜阅读。不过不必理会，json字符串的主要用途是直接被传输到后台数据库或文件进行保存，也可以作为javascript变量在代码中保存，实现数据的持久化。

使用Network

Network3D是一个用于交互的视图组件，可以展示3D场景，并实现用户和3D场景之间的交互，比如旋转镜头，选中3D对象，通过鼠标或键盘移动3D对象等。

在网页中添加Network

在网页中插入Network3D，首先自定义一个div，并创建一张画布canvas:

```
<div id = "mainDiv" >
    <canvas id="myCanvas" width="800" height="800"/>
</div>
```

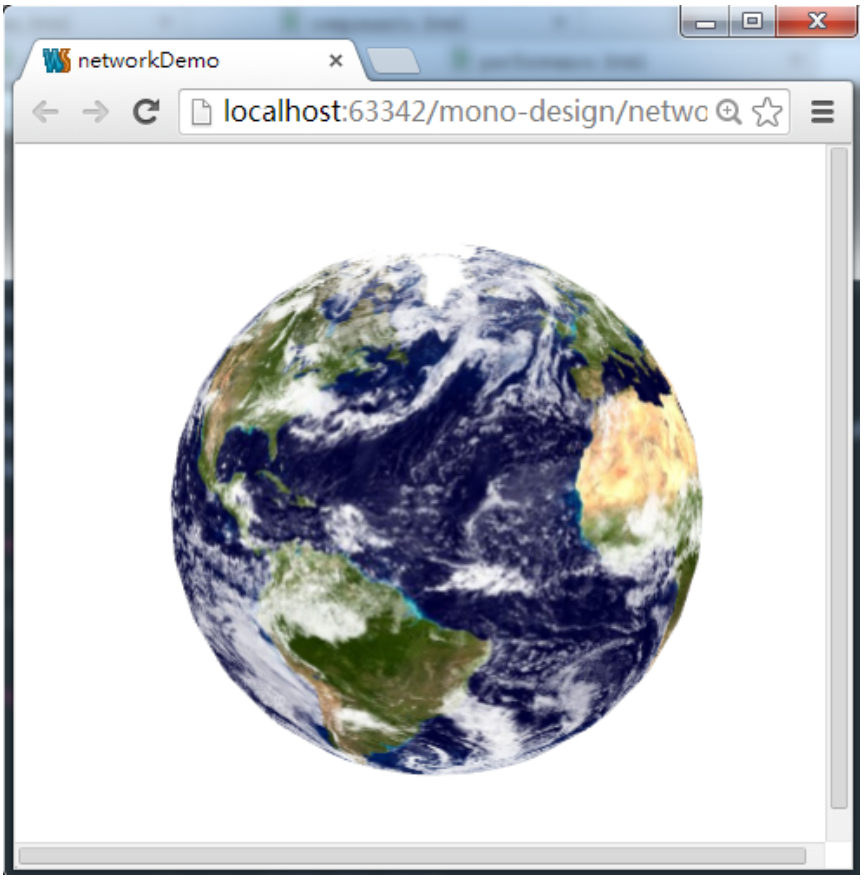
Network3D需要由一个与之关联的DataBox驱动，显示DataBox中的网元。当初始化Network3D时，DataBox默认会绑定在Network3D上。通过network3D构造方法创建network3D，并与DataBox、Camera和canvas绑定：

```
//构造方法
Network3D=function(dataBox,camera,canvas,parameters){};
```

下面的例子在html中创建了一个network3D，并展现出一个3D物体。

```
<!DOCTYPE html>
<html>
<head>
<title>Room Inspection</title>
<script type="text/javascript" src = "libs/t.js"></script>
<script type="text/javascript" src = "libs/twaver.js"></script>
<script type="text/javascript">
var network, interaction;
function load(){
    var box = new mono.DataBox();
    var camera = new mono.PerspectiveCamera(30, 1.5, 0.1, 10000);
    var target = new mono.Vec3(150,50,150);
    camera.setPosition(1000,500,1000);
    camera.lookAt(target);
    network = new mono.Network3D(box, camera, myCanvas);
    mono.Utils.autoAdjustNetworkBounds(network,document.documentElement,'clientWidth','clientHeight
    box.add(new mono.AmbientLight(0xffffff));
    createBall(box);
}
function createBall(box){
    var ball=new mono.Sphere(300,100);
    ball.setStyle('m.texture.image','images/earth2.png');
    ball.setStyle('m.type','phong');
    ball.setStyle('m.texture.repeat',new mono.Vec2(1,1));
    ball.setStyle('m.specularStrength',100);
    box.add(ball);
}
    </script>
</head>
<body onload = 'load()'>
    <div id = "mainDiv" >
        <canvas id="myCanvas" width="800" height="800"/>
    </div>
</body>
</html>
```

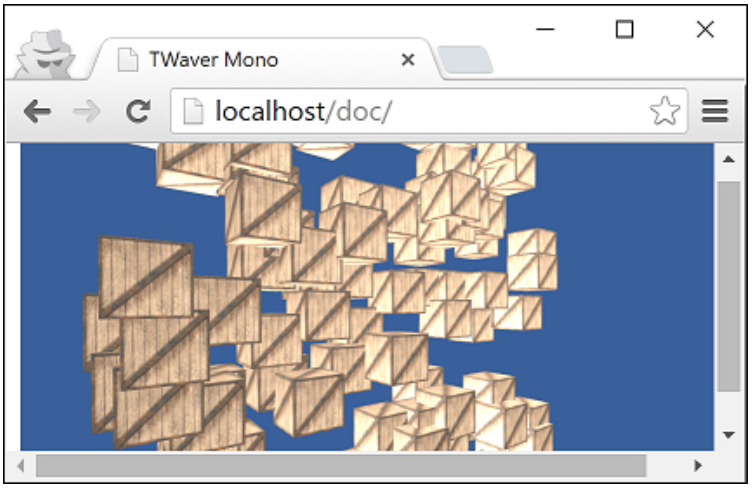
执行代码后运行效果如图。



设置network背景颜色

默认network使用白色作为背景。修改network空白区域的背景颜色，可以使用函数network.setClearColor()完成。

```
var box = new mono.DataBox();
var network= new mono.Network3D(box, null, monoCanvas);
network.setClearColor('#39609B');
```



自适应缩放

自适应缩放会自动调整Network3D的Bounds值，使3D物体在network中的呈现更加友善。可通过如下方法实现：

```
autoAdjustNetworkBounds = function(network, o, w, h, left, top) {}
```

鼠标键盘交互事件

mono中为我们提供了非常友善的用户鼠标键盘交互事件，下面列出常用的事件。

鼠标交互事件

- 按住左键移动：旋转物体
- 按住右键移动：平移物体
- 滚轴：缩放物体

键盘交互事件

- pageup:沿着z轴正方向移动
- pagedown:沿着z轴负方向移动
- left:沿着x轴正方向移动
- up:沿着x轴负方向移动
- right:沿着y轴正方向移动
- down:沿着y轴负方向移动
- ctrl+A:全选
- ctrl+C:复制
- ctrl+V:粘贴

自动渲染

我们可以通过setRenderSelectFunction()方法过滤哪些选中的网元需要绘制选择边框，哪些不需要绘制边框。

```
setRenderSelectFunction = function(f) {}
```

我们还可以在分别监听到渲染前后，使用setBeforeRenderFunction()设置渲染Network3D前的执行方法，使用setRenderCallback()来设置渲染Network3D之后的回调方法。方法原型如下：

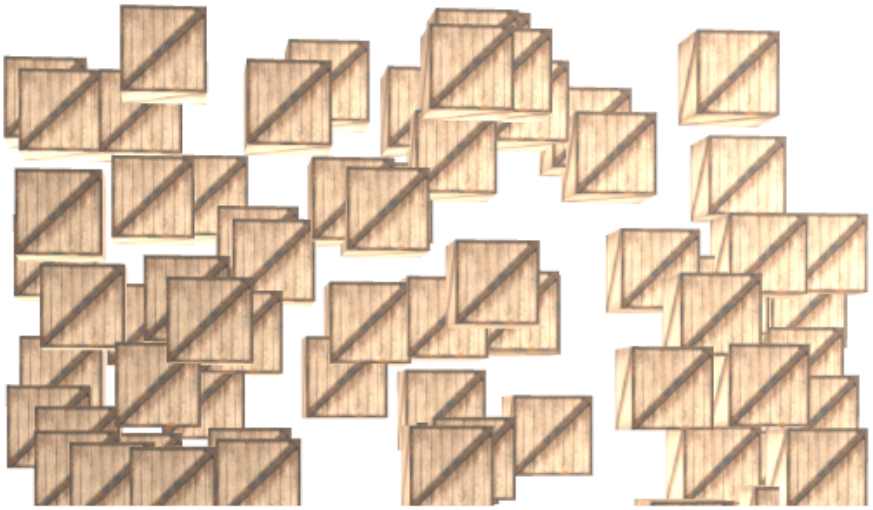
```
mono.Network3D.prototype.setRenderCallback = function(f) {}
//调用setRenderCallback方法
network.setRenderCallback(function(){
    //do something
});

mono.Network3D.prototype.setBeforeRenderFunction = function(f) {}
//调用setBeforeRenderFunction方法
network.setBeforeRenderFunction(function(){
//do something
});
```

使用雾霾效果

雾霾是当代中国热词，mono自然应该支持这一特效功能。通过network的setUseFog(boolean)方法可以开启雾霾效果，setFogDensity(int)方法设置雾霾的等级。

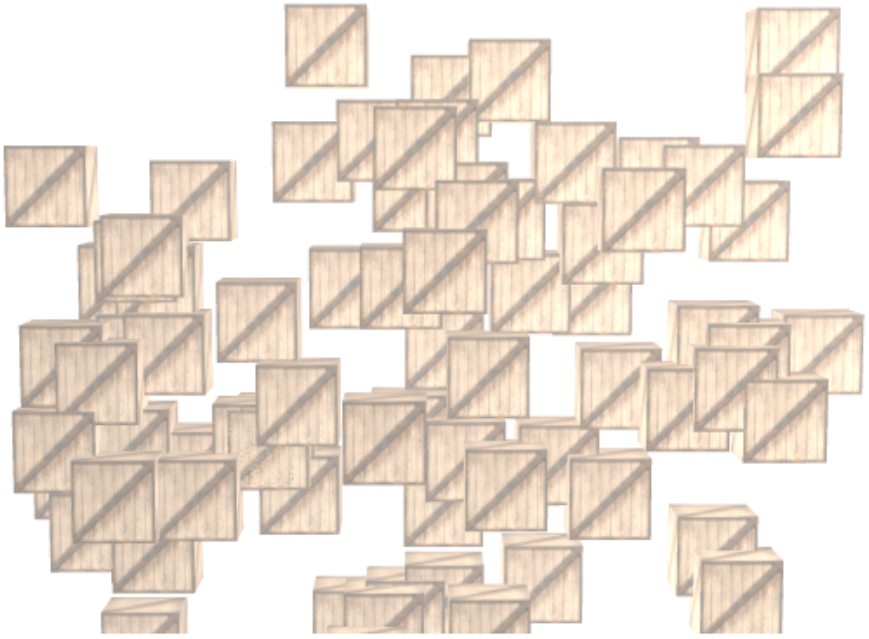
下面代码随机创建了许多立方体：

```
for(var i=0;i<100;i++){
  var cube=new mono.Cube(100, 100, 100);
  cube.s({
    'm.type': 'phong',
    'm.texture.image': 'box.jpg',
  });
  cube.setPosition(Math.random()*1000-500, Math.random()*1000-500, Math.random()*200-100);
  box.add(cube);
}
```



设置雾霾效果和指数：

```
1 network.setUseFog(true);
2 network.setFogDensity(1);
```



设置雾霾指数为2：

```
network.setUseFog(true);
network.setFogDensity(2);
```

继续增大雾霾指数：

```
network.setUseFog(true);
network.setFogDensity(3);
```

就好像北京的天气，几乎什么都看不见了。

理解Interaction

在mono的MVC结构中，DataBox提供了Model的能力，Network提供了View的能力，而Interaction则提供了Controller的能力。Interaction是在画布和数据之间提供的交互能力。例如通过键盘快捷键漫游、通过鼠标进行移动镜头等，都是交互的一部分。

每一个Network必须要设置Interaction才能有交互能力。一个没有任何Interaction的Network不会响应任何用户的交互，只能看，不能操作。Mono中已经封装了许多不同作用的Interaction，主要要包括：

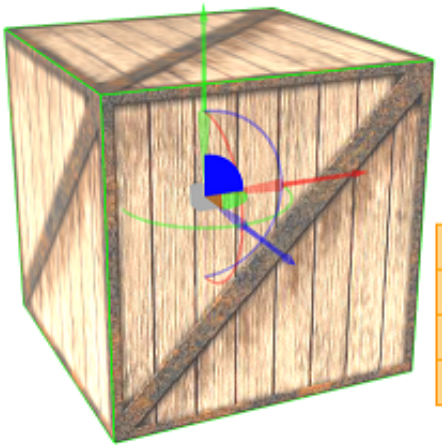
- DefaultInteraction：默认交互，提供了默认的交互能力，例如键盘/鼠标镜头漫游等
- SelectionInteraction：选中交互，提供了点击选中物体的能力，也支持ctrl+鼠标进行批量框选
- EditInteraction：编辑交互，提供基于鼠标进行位置拖拽、角度拖拽、拉伸拖拽等综合编辑能力

Network可以同时设置多个Interaction，这些Interaction会同时起作用。例如，Network默认就内置了DefaultInteraction和SelectionInteraction这两个交互器，所以默认情况下Network可以相应鼠标/键盘漫游、点击选中等交互动作。

由于Network默认就内置了DefaultInteraction和SelectionInteraction这两个最常用的交互器，所以一般情况下，我们不需要特别为Network进行交互设置。如果要重新设置交互，可以把新的Interaction数组设置给network。

下面的例子为network增加了编辑交互器：

```
1 var network = new mono.Network3D();
2
3 var defaultInteraction = new mono.DefaultInteraction(network);
4 var selectionInteraction = mono.SelectionInteraction(network);
5 var editInteraction = new mono.EditInteraction(network);
6
7 network.setInteractions([defaultInteraction, selectionInteraction, editInteracti
```



	x	y	z
p	-4.99	4.04	0.00
r	0°	0°	0°
s	1.00	1.00	1.00

如图，物体可以被选中、被拖拽，编辑交互已经生效。

可以通过network.getInteractions()获取到已经置入的交互器：

```
var interactions = network.getInteractions();
```

由于DefaultInteraction是最重要、最常见的交互器，为了方便获取，network提供了getDefaultInteraction()方法来特别获得DefaultInteraction对象：

```
var defaultInteraction = network.getDefaultInteraction();
```

可见，通过灵活组合和设置Interaction，就可以让画布实现各种不同的交互效果。下面通过Interaction来实现几个技巧。

技巧：禁止交互

有时候，我们希望将3D场景进行视频一样的自动漫游和回放，而不希望用户进行交互和干扰。此时我们可以暂时将交互器删掉（设置一个空数组即可）：

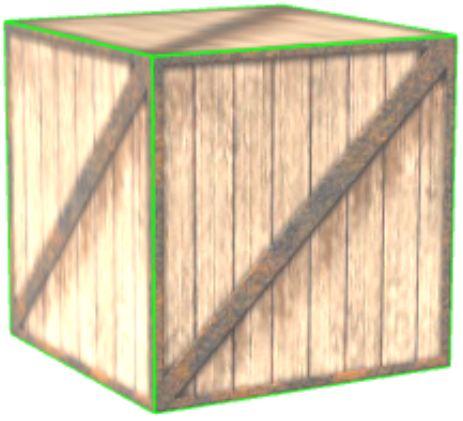
```
var oldInteractions = network.getInteractions();
network.setInteractions([]);
```

等播放完毕需要恢复交互能力，将原来的交互器设置回去即可：

```
network.setInteractions(oldInteractions);
```

技巧：禁止物体选中

由于默认network内置了SelectionInteraction，点击物体后会被选中，显示为一个绿色高亮边框，如下图：



方法一：设置交互

我们可以重新设置交互数组，不再设置SelectionInteraction，即可去掉选中交互：

```
network.setInteractions([new mono.DefaultInteraction(network)]);
```

方法二：重载network.isSelectable函数

在不从新设置Interactions的情况下，也可以通过重写network的isSelection(element)函数，来更加精确的定义哪些物体可以选中、哪些物体不可以选中。isSelection函数会传入每一个3D对象element，返回true则表示可以选中，false表示不能选中。

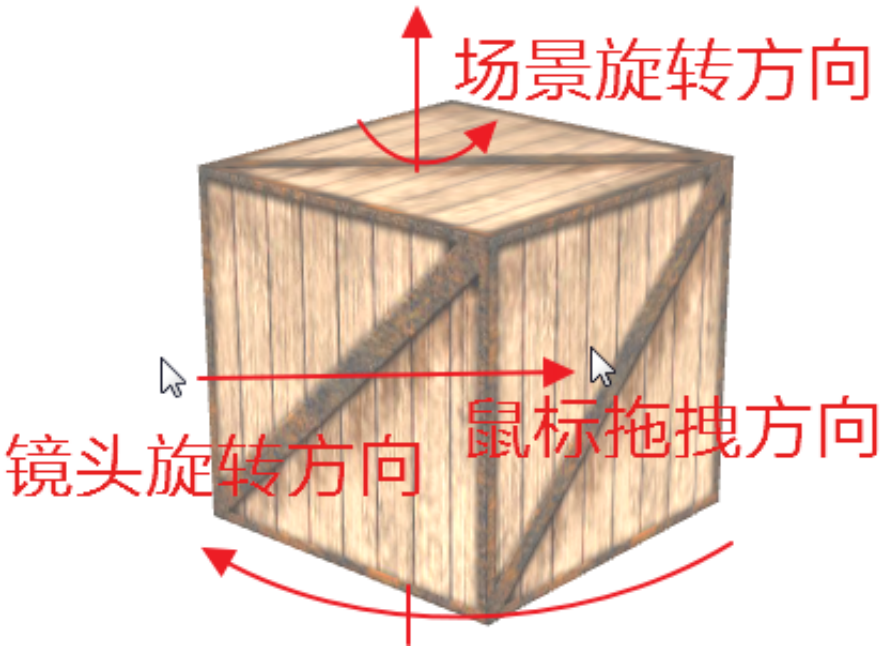
```
network.isSelectable = function(element){
  return false;
};
<pre>

上面的代码禁止一切物体选中。下面的代码则只允许场景中client的type为'rack'的机柜物体可以选中：
<pre>
network.isSelectable = function(element){
  return element.getClient('type') === 'rack';
};
```

使用DefaultInteraction

DefaultInteraction是mono提供的最重要的一个交互器，它包含了常用的交互功能，包括可以在3D场景中旋转镜头，通过鼠标滚轮缩放镜头，键盘操作镜头等。以下介绍主要控制参数。

rotateSpeed

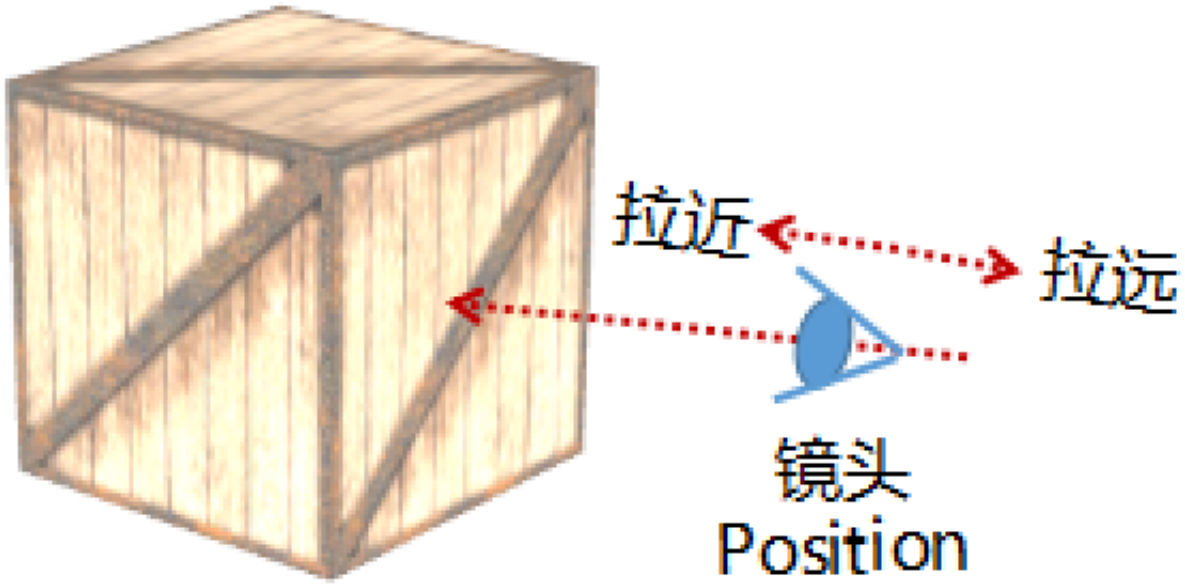
控制鼠标左键拖拽旋转镜头速度。默认在第三人称视角模式下，鼠标左键拖拽会让镜头绕焦点所在轴进行旋转。rotateSpeed就是控制镜头的旋转速度。



- 作用：控制鼠标拖拽镜头旋转速度
- 默认值：1
- 建议值：0.5~10。越小拖拽反应越迟钝，越大拖拽反应越灵敏
- 用法：interaction.rotateSpeed = 2 或 setRotateSpeed()/getRotateSpeed()

zoomSpeed

控制鼠标滚轮缩放场景的速度。鼠标滚轮的滚动，会导致镜头的拉近和拉远，表现为场景的放大和缩小。zoomSpeed就是控制镜头的拉近/拉远的速度。

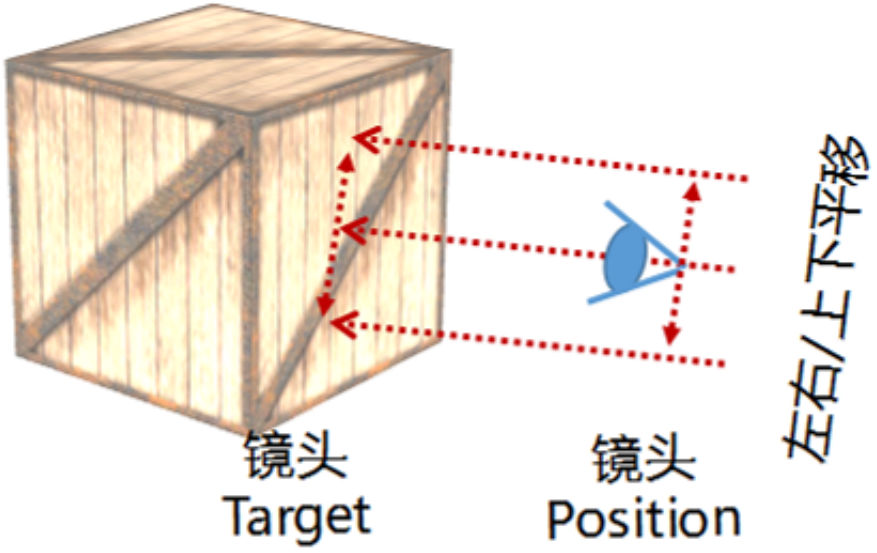


- 作用：控制鼠标滚轮镜头拉动速度
- 默认值：10
- 建议值：1~20。越小镜头移动越迟钝，越大镜头移动越灵敏

- 用法：interaction.zoomSpeed= 2 或 setZoomSpeed()/getZoomSpeed()

panSpeed

控制鼠标右键平移镜头的速度。鼠标右键拖拽会在垂直面内左右、上下平移镜头的位置。平移时，镜头的位置（position）和焦点（target）会同时移动，以保证镜头lookat的方向不变。panSpeed就是控制镜头平移的速度。

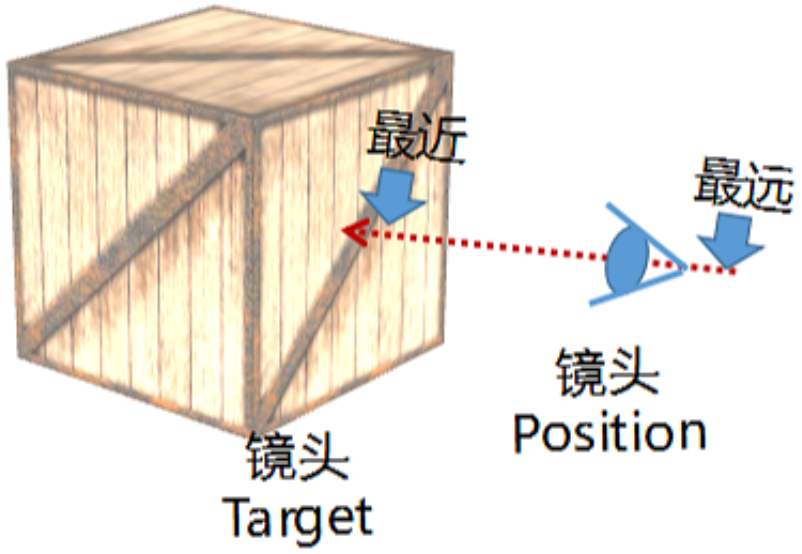


- 作用：控制鼠标右键拖拽平移镜头的速度
- 默认值：3
- 建议值：1~10。越小镜头移动越迟钝，越大镜头移动越灵敏
- 用法：interaction.panSpeed= 2 或 setPanSpeed()/getPanSpeed()

minDistance/maxDistance

通过鼠标滚轮拉近/拉远镜头位置的极值。这是非常有用的两个参数，尤其是maxDistance。当滚轮不断拉远镜头位置，最终会导致场景和镜头距离过远而彻底消失，有时候这让用户感觉不舒服。所以，最好的使用方法是适当的设置镜头位置极值，避免过远或过近产生的视觉不适。

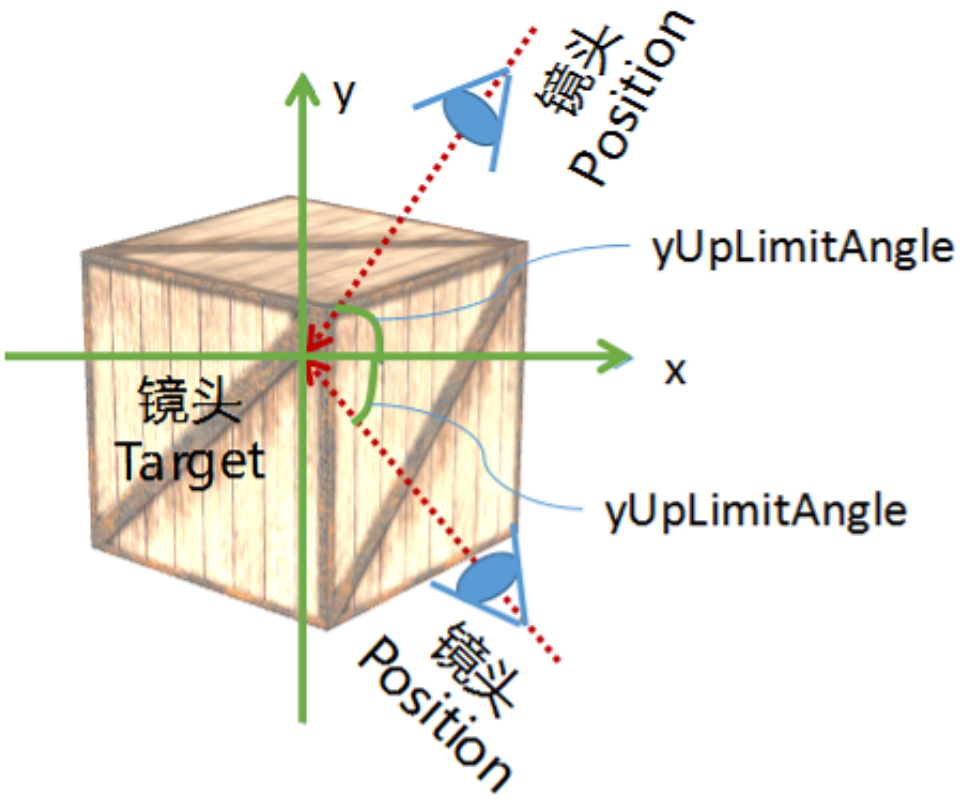
这两个极值没有什么绝对的标准，应当根据场景情况而定。例如，一个尺寸为10,000见方的场景，可以将maxDistance设置为20,000~50,000左右，是合适的。最小值也是一样，假如target焦点盯在一个设备面板上进行lookat，则最小值设置为50可以保证镜头不至于离目标过近，产生不适感或进入近切面盲区。



- 作用：控制鼠标移动镜头位置的最近/最远点
- 默认值：minDistance为0，maxDistance为Infinity
- 建议值：根据场景大小而定
- 用法：interaction.minDistance = 20 或 setMinDistance()/getMinDistance()

yLowerLimitAngle/yUpLimitAngle

通过鼠标操作镜头在y轴位置的角度极值，也就是镜头可以俯视/仰视的角度极值，默认是正负90度（-Math.PI/2 ~ Math.PI/2）。0角度则是水平线位置。具体看下图：



这两个参数中，yLowerLimitAngle更加常用，主要用于控制镜头是否可以移动到水平面的下

方。例如，将yLowerLimitAngle设置为0，则镜头只能在水平面上方移动，而不能进入水平面下方。主要用于控制镜头不可以看到物体的“底部”。例如一个建筑物、一个房间等场景，从下方向上方仰视是没有什么意义的，用yLowerLimitAngle参数可以进行控制，让用户的感觉更舒适。

- 作用：控制鼠标移动镜头俯视/仰视角度
- 默认值：yLowerLimitAngle为-Math.PI/2，yUpLimitAngle为Math.PI/2
- 建议值：保持默认值，或选用yLowerLimitAngle=0、 yUpLimitAngle=Math.PI/2的设置（此时只能俯视不能仰视）
- 用法：interaction.yLowerLimitAngle= -Math.PI/4 或 用getY***/setY***方法()

使用EditInteraction

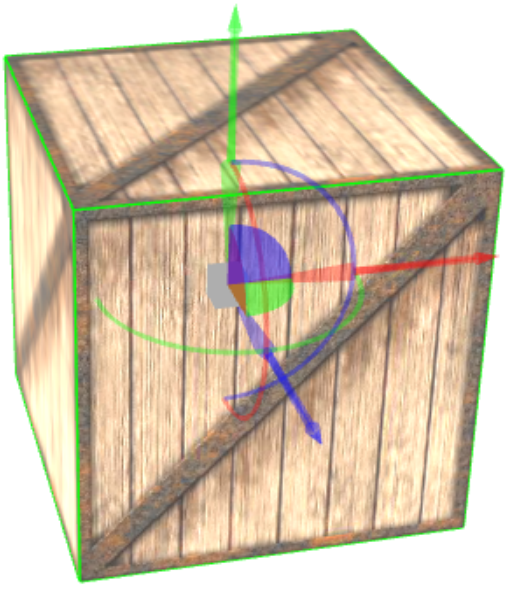
EditInteraction是一个负责编辑的交互器。用户对3D对象的编辑主要是只通过鼠标对位置、角度、拉伸缩放进行编辑。由于x、y、z三个轴都有位置、角度、拉伸缩放的数值，因此如何在界面上设计这9个交互，是一个很麻烦的事。好在mono已经提供了比较清晰完善的解决方案。

操作EditInteraction

下面的代码是如何给network添加EditInteraction：

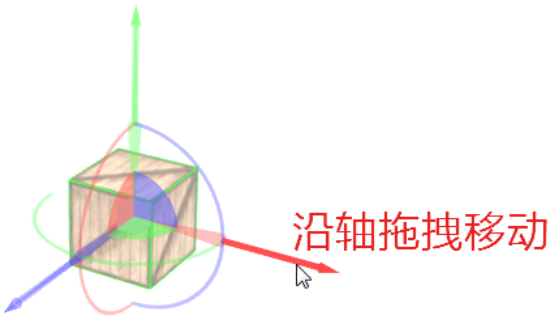
```
var defaultInteraction = new mono.DefaultInteraction(network);
var selectionInteraction = new mono.SelectionInteraction(network);
var editInteraction = new mono.EditInteraction(network);
network.setInteractions([defaultInteraction, selectionInteraction, editInteracti
```

下图是EditInteraction交互状态下的交互显示方式：

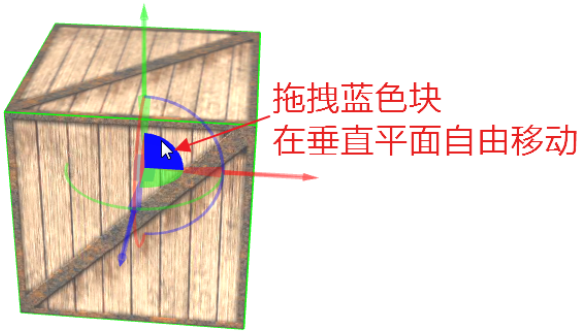


Position：鼠标沿着三个轴箭头拖拽，可以将物体沿着轴进行移动。x、y、z三个轴向的箭头分

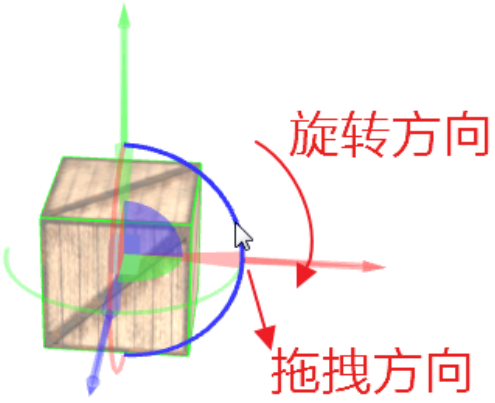
别用不同的颜色表示，如下图：



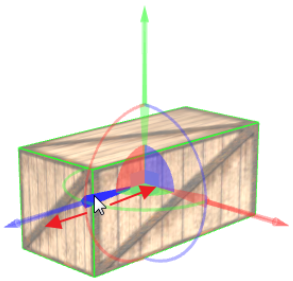
此外，拖拽三个直角扇形区域，也可以让物体沿着这个扇形区域所在的面进行移动。例如：蓝色扇形区域代表的是红色轴和绿色轴所组成的面，拖拽蓝色扇形即可让物体在这个面上自由拖拽移动，如下图：



Rotation：鼠标沿着三个弧线拖拽，可以将物体沿着弧线对应的角度方向进行旋转。x、y、z三个轴向的旋转弧线分别用不通的颜色表示，如下图：

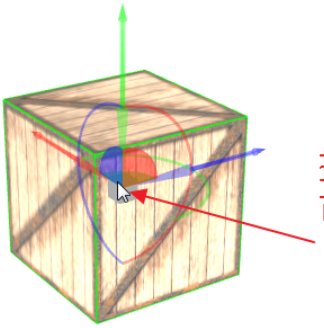


Scale：鼠标拖住轴线跟部的锥形体并沿着弧线拖拽，可以将物体沿着弧线对应的方向进行拉伸缩放。x、y、z三个轴向的锥形体分别用不通的颜色表示，如下图：



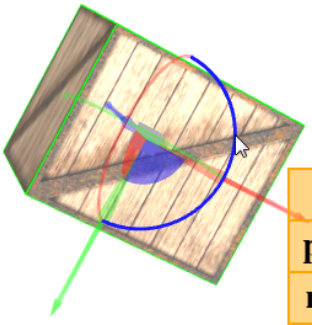
拖拽锥形体
沿轴进行拉伸缩放

除了单独在某个轴向进行拉伸之外，还可以拖拽最中心的灰色立方块进行三个轴向的等比例拉伸：



拖拽中心灰色立体块
可以整体scale

此外，为了在拖拽过程中实时了解到位置、角度、拉伸缩放的数值变化，mono还在鼠标旁边显示了一个包含9个数值的、实时变化的表格，作为参考：



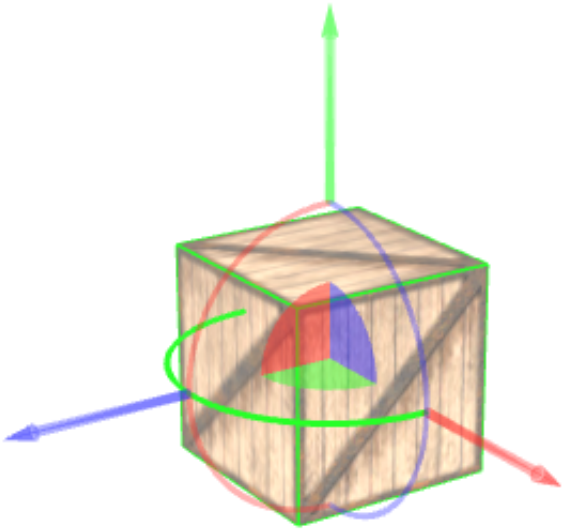
拖拽过程中
实时显示9个数值

	x	y	z
p	0.00	0.00	0.00
r	167°	0°	33°
s	1.20	1.00	1.00

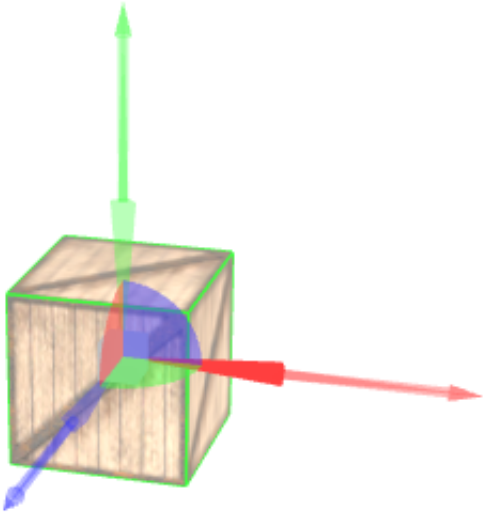
参数控制EditInteraction

通过setShowHelpers(false)可以控制是否显示箭头、弧线、椎体这些交互用的视觉元素。如果关闭，则不能通鼠标进行交互操作。一般不会这样使用。

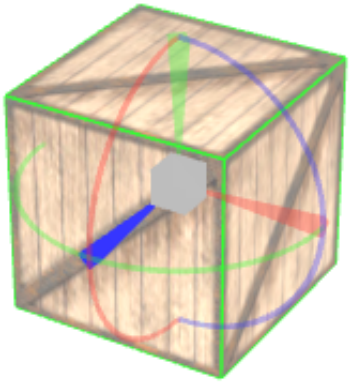
通过setScaleable(boolean)可以控制是否显示拉伸缩放的三个“椎体色块”。如果不想让用户修改scale，可以采用这种方法关闭scale。显示效果如下：



通过setRotateable(boolean)可以控制是否显示拖拽旋转角度的弧形线。如果不想让用户修改物体的角度，可以采用这种方法关闭rotation。显示效果如下：



通过setTranslateable(boolean)可以控制是否可以拖拽箭头或扇形角来移动物体的位置。如果不想让用户修改物体的位置，可以采用这种方法关闭position。显示效果如下：



通过EditInteraction.setScaleRate(scaleRate)函数可以控制拉伸缩放的比例速度。越大拉伸的

就越快，越小拉伸的就越慢。

> 以上几个函数可以混合使用，控制更精细的编辑效果。

EditInteraction中的默认拖拽行为

上面介绍的是用户用鼠标操作编辑helper产生的编辑行为。其实，EditInteraction模式下，用鼠标直接拖拽物体本身进行移动，也可以产生编辑效果。只是这种操作会有很大的歧义：拖拽物体移动，是修改水平面的位移，还是垂直面的位移？是要旋转还是拉伸？为了定义此时的编辑模式，mono定义了EditInteraction.setDefaultMode(defaultMode)函数来设置鼠标拖拽要编辑的模式，其中defaultMode可以是：

- 移动位置，可以选一个轴和两个轴的组合，包括：TranslateX、TranslateY、TranslateZ、TranslateXY、TranslateXZ、TranslateYZ
- 旋转：可以绕三个轴任意一个旋转，包括：RotateX、RotateY、RotateZ
- 拉伸缩放：可以沿三个轴任意一个轴拉伸，包括：ScaleX、ScaleY、ScaleZ

下面设置拖拽物体沿着水平面（X和Z轴所在的平面）进行移动：

```
1 EditInteraction editInteraction = new mono.EditInteraction(network);
2 editInteraction.setDefaultMode('TranslateXZ');
```

EditInteraction中的快捷键

EditInteraction内置了一些用于编辑用的快捷键，包括：

- del：删除选中的对象。会有确认消息框弹出；

技巧：如何控制对象一起或部分移动？

在编辑状态下，很多时候一个大的物体是由许多小的物体一起组成的。这些小的物体可能已经通过ComboNode进行运算变成了一个大的物体，也可能以parent父子关系的形式独立存在。对于前者，它们已经变成了一个物体，选中、移动都是一个对象，很容易理解。对于后者，如果要让物体的一部分移动，或整体一起移动，就需要考虑如何实现了。

原则：在mono中，**当移动parent对象时，所有的children及其子孙，都会跟着移动；而移动孩子，父亲不会跟着移动。**这一点请一定牢记。

基于这一原则：如果选中多个孩子进行移动，就可以实现物体“部分移动”的效果；如果选中父亲进行移动，就可以实现“整体移动”的效果。

这样说起来很容易理解，但实际操作中，最终用户怎么能够知道哪个对象是“父对象”，哪个对象是“自对象”呢？如果不能准确的选中最根的根对象，移动某个孩子对象很容易把整个物体拖“散架”。为了解决这个问题，可以用这样的思路：当用户要进行“整体移动”时，无论用户选择了物体的哪个部分，都自动判断其跟对象，并选中跟对象。用一个while循环一直找到父对象为

null就可以了。

有时候，根父对象不一定是个头最大、最直观的那个对象。此时我们也可以自定义谁是整个物体的“移动代言人”，给它单独设置一个client标记。当物体任何部分选中后，都自动找到它，设置为选中。

技巧：物体如何整体删除？

当物体由多个对象组成并用parent关系进行组织时，如果没有选中全选中所有对象，删除物体会发生残留，因为没有被选中的对象会残留下来。此时可以自动找到跟节点，并用DataBox的removeByDescendant方法删除这个节点的所有子孙，这样就不会发生残留了。

理解HTML事件机制

鼠标事件主要指双击、单机、拖拽等。我们经常需要对用户的这些鼠标操作进行响应，此时需要使用鼠标事件处理机制。鼠标、键盘事件机制本身是浏览器定义的，和mono并无直接关系。这里主要介绍如何通过处理鼠标事件，来完成3D应用中常见的一些需求。

鼠标键盘事件是通过html定义的addEventListener和removeEventListener来完成的。它的定义如下：

监听HTML事件方法

```
element.addEventListener(event, function, useCapture);
```

event定义了事件的名称，function是事件的响应函数，useCapture是捕获还是冒泡时间执行响应。具体相关用法请参阅相关w3c标准和教程。

下面的代码给html页面上id为'myBtn'的按钮添加了一个单击事件，点击后将id为'demo'的元素内容改为"Hellow World"：

```
document.getElementById("myBtn").addEventListener("click", function(){
    document.getElementById("demo").innerHTML = "Hello World";
});
```

在mono中，network是一个js对象，它本身并不是一个显示在页面中的DOM元素。要为network的canvas元素添加事件，可以通过network.getRootView()函数获得network在html页面中对应的顶层DOM元素，然后再添加事件。

下面的代码在network上添加了一个单击事件，并在控制台打印消息：

```
network.getRootView().addEventListener('click', function(e){
  console.log('network clicked');
});
```

以下列出了常用的鼠标和键盘事件。更多关于DOM事件的介绍，请参阅网络。

鼠标事件

属性	描述
onclick	当用户点击某个对象时调用的事件句柄。
oncontextmenu	在用户点击鼠标右键打开上下文菜单时触发
ondblclick	当用户双击某个对象时调用的事件句柄。
onmousedown	鼠标按钮被按下。
onmouseenter	当鼠标指针移动到元素上时触发。
onmouseleave	当鼠标指针移出元素时触发
onmousemove	鼠标被移动。
onmouseover	鼠标移到某元素之上。
onmouseout	鼠标从某元素移开。
onmouseup	鼠标按键被松开。

键盘事件

属性	描述
onkeydown	某个键盘按键被按下。
onkeypress	某个键盘按键被按下并松开。
onkeyup	某个键盘按键被松开。

拖动事件

事件	描述
ondrag	该事件在元素正在拖动时触发
ondragend	该事件在用户完成元素的拖动时触发
ondragenter	该事件在拖动的元素进入放置目标时触发
ondragleave	该事件在拖动元素离开放置目标时触发

ondragover	该事件在拖动元素在放置目标上时触发
ondragstart	该事件在用户开始拖动元素时触发
ondrop	该事件在拖动元素放置在目标区域时触发

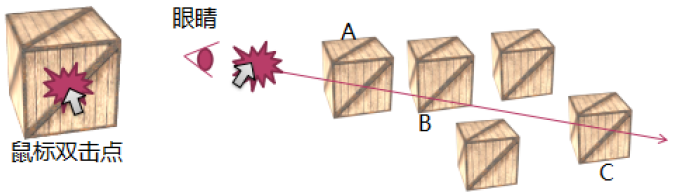
常用技巧

双击到了哪个物体？

在3D场景中，经常会需要判断用户双击了哪一个物体。例如：双击一个门将其打开、双击摄像头播放一段视频，等等。要做到这些，首先要能判断用户双击了哪个3D物体。

当我们在network上随意位置进行双击，实际上是从这一点发出了一条无限远的"射线"，凡是在这条射线上的物体，都是可以被双击到的物体。因此双击一个点可能会点击到很多物体。通常我们会选取离眼睛（镜头）最近的作为目标，但也有例外：例如我们可能会忽略离眼睛很近的透明玻璃窗，而选取玻璃后面的机柜。这些业务逻辑，都需要我们用代码来进行判断。

好在network已经提供了network.getElementsByMouseEvent(event)方法，可以直接返回所有在双击射线上穿过的物体的数组，按从近到远的顺序。这样我们就可以直接处理了。



下面的代码把双击到的全部物体拿到并打印出：

```
network.getRootView().addEventListener('dblclick', function(e){
  var objects= network.getElementsByMouseEvent(e);
  console.log(objects);
});
```

需要注意的是：这里的数组中的object并不直接是3D对象，而是一个封装了包含3D对象在内更多双击点信息的数据结构，主要信息包括：

```
Object {
  distance: 175.17295246548093 //双击点到镜头的距离
  element: //双击点的3D对象
  face: M  //双击点所在的物体面
  faceIndex: 4 //双击点物体面的索引值
  side: -1   //双击点物体表面是外面还是内面
}
```

可以根据这些信息，做更复杂的双击点判断。例如根据距离、反正面等进行判断。下面代码继续改进，判断双击了哪个机柜（离眼睛最近的一个）：

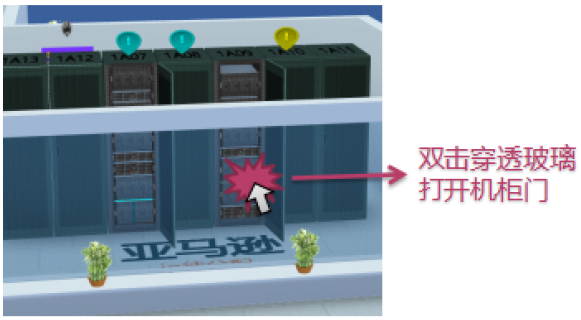
```
//模拟创建一个机柜的立方体
var cube=new mono.Cube(100, 100, 100);
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'rack.jpg',
});

//为立方体添加type='rack'的client属性，作为机柜物体的标记
cube.setClient('type', 'rack');
box.add(cube);

//添加network的双击事件
network.getRootView().addEventListener('dblclick', function(e){
  var objects = network.getElementsByMouseEvent(e);
  var clickedElement;
  if(objects){
    for(var i=0;i<objects.length;i++){
      var object=objects[i];
      //注意：object并不是3D对象，而是一个包含了很多事件信息的对象。3D对象存储在object.element变量中
      var element=object.element;
      if(element && element.getClient('type') === 'rack'){
        clickedElement=element;
        break;
      }
    }
  }
  if(clickedElement){
    console.log('clicked rack: ' + clickedElement);
  }
});
```

下面的函数可以用于判断"第一个"被双击到的物体。其中，忽略了Billboard类型的物体。一般Billboard作为信息显示的辅助对象，不参与物理实体的碰撞检测。

```
findFirstObjectByMouse: function(network, e){
  var objects = network.getElementsByMouseEvent(e);
```

```
  if (objects.length) {
    for(var i=0;i<objects.length;i++){
      var first = objects[i];
      var object3d = first.element;
      if(! (object3d instanceof mono.Billboard)){
        return first;
      }
    }
  }
  return null;
}
```

下面是一个实际应用的3D场景。在隔着玻璃双击机柜门时，依旧可以实现打开机柜大门，而不受眼前的玻璃窗影响，实现了“隔山打牛”的效果。这也是利用了上面的逻辑判断实现的。



做一个自己的Tooltip

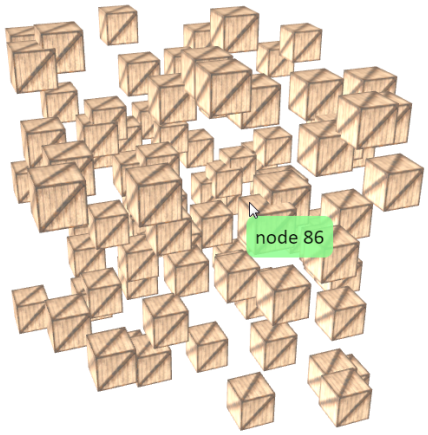
在mono中，并没有内置的tooltip机制。但是根据上面介绍的事件监听机制，要自己完全动手做一个也并不难。大概思路是：

- 先在页面上创建一个不可见的div对象，作为tooltip文字显示标签；
- 为network添加一个mousemove事件，监听鼠标在画布上的移动；
- 当鼠标指向位置没有物体，继续隐藏div；
- 当鼠标指向一个3D物体，将3D物体的信息（例如name等）显示在div上，并将div位置移动到鼠标所在位置，设置为可见

代码很简单，可以看一下：

```
//随机产生100个立方体并设置不同的name以便进行测试
for(var i=0; i<100; i++){
  var cube=new mono.Cube(10, 10, 10);
  cube.setName('node '+i);
  cube.s({
    'm.type': 'phong',
    'm.texture.image': 'box.jpg',
  });
  cube.setPosition(Math.random()*100-50, Math.random()*100-50, Math.random()*100-50);
  box.add(cube);
}

//创建一个隐藏的div对象作为tooltip
var tipDiv=document.createElement('div');
tipDiv.style.display = 'block';
tipDiv.style['font-family']='Calibri';
tipDiv.style['font-size']='12px';
tipDiv.style.position = 'absolute';
tipDiv.style['padding']='5px';
tipDiv.style.background='rgba(144,254,144,0.85)';
tipDiv.style['border-radius']='5px';
tipDiv.style.visibility='hidden';
document.body.appendChild(tipDiv);

//函数用于找到鼠标指向的第一个物体
var findFirstElement = function(e){
  var objects = network.getElementsByMouseEvent(e);
  if (objects.length) {
    return objects[0].element;
  }
}

network.lastPointedElement;

//添加mouseover监听器
network.getRootView().addEventListener('mousemove',function(e){
  tipDiv.style.top=(e.y+15)+'px';
  tipDiv.style.left=(e.x+15)+'px';
  var pointedElement=findFirstElement(e);
  if(pointedElement===network.lastPointedElement) {
    return;
  }
  network.lastPointedElement=pointedElement;
  //如果有物体，显示div并显示其信息；否则，隐藏div
  if(pointedElement){
    tipDiv.innerHTML=pointedElement.getName();
    tipDiv.style.visibility='visible';
  }else{
    tipDiv.style.visibility='hidden';
  }
});
```

显示效果如下图：



当然这个tooltip并不完美。首先显示样式简陋，这个读者可以自行完善。其次，tip的显示非常“刚性”，没有像操作系统一般的tooltip那样有一个延迟的过程。要完善这一点，可以通过启动一个定时器，延迟一点时间（例如1~2秒）后再显示div。这里有一点复杂的是，当timer尚未触发，而新的事件又发生时，应该首先清除原来的定时器。具体代码不再列出，有兴趣的读者可以自行尝试。

常见对象速览

mono中提供的3D对象很多，常用的例如立方体、球体、圆柱体等。这些对象都有自身的特定形状、功能、属性和典型用法。这里列出了最常用到的对象，供您快速了解mono的模型能力。

图示	名称	说明
	mono.Cube 立方体	**极常用** 由6个面组成的立方体，是3D中最常见的物体。它可以单独表示空间一个物体，例如方形的服务器、盒子、书等，也可以和其他物体一起组合成更复杂的物体。立方体是最被常用的对象，也是效率极高的一个模型对象。主要参数是长、宽、高。 **典型用途** 服务器、门体/窗体、书、盒子、机柜等
	mono.Sphere 球体	**常用** 球体是一个曲面，由横向、纵向两个方向多个切片拼凑模拟而成。切片数量决定了球体的光滑度。20*20的切片会产生400个面，因此球的顶点数较多，过多的对象数量和切片数量都可能会导致性能下降，应谨慎使用。主要参数是半径。 **典型用途** 各类球形、椭球形、部分球形物体
	mono.Cylinder 圆柱体	**极常用** 圆柱体由侧面、顶面、底面组成。侧面曲面由n个切片模拟而成，切片越多越光滑，过多切片会影响性能。圆柱上下表面可封闭或露空。由于上下表面半径可相同也可不同，因此圆柱体可变形为圆锥、半锥等。设置切片也可以形成多面棱锥体，例如切片设置为4可成四面锥体，类似金字塔，等等。圆柱体主要参数有高、上下半径。 **典型用途** 柱子等
	mono.Torus 圆环	**不常用** 由一个圆管组成的环形物，光滑度由切面径向和走向径向的切片数量控制。它顶点较多，效率较差，应谨慎使用。场景中圆环物体不多见
	mono.PathNode 路径体	**常用** 路径体是一种复杂的形状，由一个任意走向路径和任意形状的横切面组成。最常用法是各类“管子”。圆形横切面的路径体会组成圆管，方形横切面路径体会组成方管，等等。可以控制两端是否封闭，圆管横切面是完整圆还是半圆，等等。顶点较多，效率较差，场景中数量不宜过多，并仔细控制好切片数量。 **典型用途** 管线、管道
	mono.TextNode 文字	**不常用** 它可以创建一串3D的文字字符，一般用于英文字符。文字的形状一般在计算机的字体文件中保存，在3D中并没有直接的支持，所以需要把字库的字符形状信息转换、提供给mono。在MONO Design编辑器中，程序提供了几种英文字体形状信息，存储在“***_regular.typeface.js”的js文件中，您可以自行查找。例如，“helvetiker_regular.typeface.js”文件中存储了helvetiker字体的正常体信息，“helvetiker_bold.typeface.js”文件中存储了helvetiker字体的粗体信息，等等。mono可以根据这个字库信息获得每个字符的形状，并生成一个复杂的3D文字模型。如果您需要更多的字体信息，可以到网站http://typeface.neocracy.org/在线提交、生成和下载。一般一个英文字体对应的js文件大约在数百kb左右。由于汉字字体形状复杂、字符数量巨大，一般**不建议使用中文字符。**文字的顶点数量很多，过多使用会影响效率。 **典型用途** 文字
	mono.ShapeNode 形状块	**常用** 有厚度的一个任意多边形。例如一个多边形的地板、后厚度的某省份形状立体块，等等。 **典型用途** 地板、3D地图块、人为划出的安全区域块等
	mono.PathCube 路径块	**常用** 切面为矩形的一个路径体。它比PathNode要简单、性能高，但没有PathNode灵活。主要参数有高度、厚度、路径。最多用法就是建筑物墙体。 **典型用途**

		墙体
	mono.Particle 粒子系统	**不常用** 粒子系统对象很特别，它由很多内部的点组成，每个点有一个贴图。通过程序动画控制点的数量和位置，可以实现一种复杂运动的粒子效果。主要用途是模拟烟雾、火焰等效果，一般和动画技术相结合。 **典型用途** 烟雾、火焰特效
	mono.CSG 运算体	**常用** 运算体由2个或以上3D对象运算组合而成。运算可以是加、减、交集、反交集等。因此运算体可以由基本物体组合成千变万化的复杂形状体。运算体在运算时会消耗大量CPU进行计算。一旦计算完成，其顶点等结果信息便保存在运算体中，每一帧3D渲染和显示时将不会再次进行计算。因此运算体的渲染效率主要取决于运算结果的顶点数和复杂度，和运算前的物体没有直接关系。需要注意的是：**一旦运算完成，运算体和运算前的物体将不再有关系，它们都是独立存在的3D对象。** **典型用途** 各类复杂物体。例如桌面挖掉一个洞等
	mono.Billboard 公告牌	**常用** 公告牌是一种特殊的3D物体，它只有一个图片组成，这张图片会永远朝前面向镜头（也就是我们用户的眼睛），而无论场景和镜头如何变化。也可以通过参数控制图片垂直方向是否朝向用户眼睛。在游戏软件场景中，经常会有这样的场景：一个移动的物体（人、机器等）会在上方显示一个图片，图片上显示了文字、状态信息等，这个图片会永远朝向用户的眼睛，而无论物体如何移动。Billboard的这一特性使得它非常合适制作3D场景中的各种信息展示、文字说明、提示警告等。 **典型用途** 设备告警、温度、文字信息

三个基础属性

每一个3D对象都有位置、角度、缩放比例三个最常用、最基础的属性。这三个属性是mono使用者必须熟知掌握的，它是创建3D应用的基础。这三个属性都是由3个数字组成，风别代表x、y、z三个轴的数值，可以表示为（x、y、z）。例如设置一个3D物体的位置是x=100、y=200、z=300，则：node.setPosition(100, 200, 300)即可；如果设置一个物体的旋转角度x轴30度、y轴20度、z轴10度，则：node.setRotation(Math.PI/180 * 30, Math.PI/180 * 20, Math.PI/180 * 10)即可（请注意角度数值要从“度”转换为“弧度”）。

在mono中，三个数字值可以用一个mono.Vec3来表示，它是一个三维向量数据结构定义。因此，设置坐标还可以写成：

```
var position = new mono.Vec3(100, 200, 300);
node.setPosition(position);

var scale = node.getScale();
var xScale = scale.x;
```

此外，mono还提供了2维向量mono.Vec2、4维向量mono.Vec4等，用法类似。关于向量和使用方法，请见相关章节介绍。

位置——Position

每个物体都有空间位置，node.setPosition(x, y, z)可设置一个物体的x、y、z坐标值。位置是3D物体最常用的属性，因此为了简化，也可以缩写为setX/setY/setZ的形式。以下代码都是正确的用法：

```
//一次性设置3个数值
node.setPosition(100, 200, 300);

//单独设置一个轴向数值
node.setX(100);
node.setY(200);
node.setZ(300);

//用三维向量来设置
var position = new mono.Vec3(100, 200, 300);
node.setPosition(position);

//把两个物体放在同一个位置
node1.setPosition(node2.getPosition());

//把两个物体水平对齐
var position = node1.getPosition();
position.x += 100;
node2.setPosition(position);
```

位置貌似是一个非常简单的属性，但它也有一些需要注意的地方，否则我们可能会经常犯一些使用的错误。

使用向量加减法处理坐标变换

把物体的位置坐标看作一个三维向量，则可以使用向量的加减法运算来计算物体之间的相对位置。例如：假设物体A的坐标向量a、物体B的坐标向量b，则：

- a+b=A再移动b后新的位置
- a-b=B相对于A的坐标位置

向量的加减法函数：

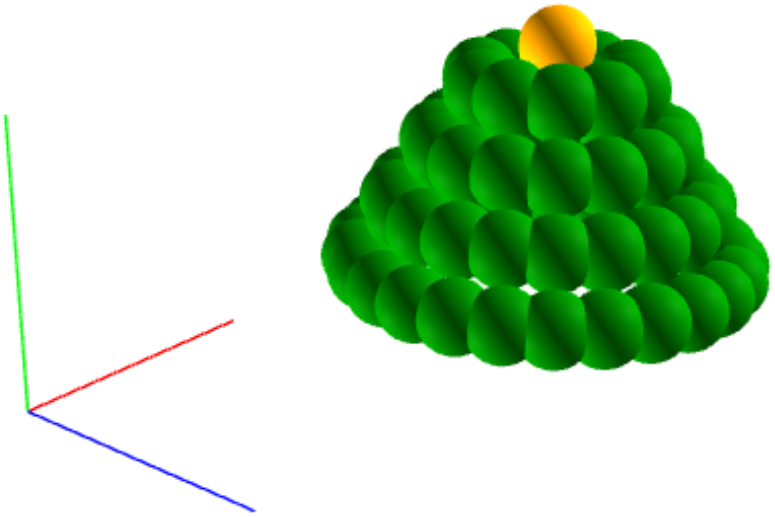
- add：向量相加
- substract：向量相减

物体A从自身处移动距离(10, 10, 10)，可以使用如下代码：

```
node.setPosition(node.getPosition().add(new mono.Vec3(10, 10, 10)));
```

下面代码在一个球旁边创建了一系列环绕的球，球的位置使用了向量的加法，简单易懂：

```
//定义一个函数，在指定坐标处创建一个指定颜色的球体
var ballCreateFunc = function(x, y, z, color){
  var ball = new mono.Sphere(10);
  ball.s({
    'm.type': 'phong',
    'm.color': color,
    'm.ambient': color,
  });
  ball.setPosition(x, y, z);
  return ball;
}

//创建中心球
var centerBall = ballCreateFunc(100, 100, 100, 'orange');
box.add(centerBall);

//创建一个函数，可以在指定球周围创建指定数量的球，组成一个圆环。同时指定圆环的尺寸和纵向偏移
var starCreateFunc = function(box, centerBall, count, radius, y){
  for(var i=0; i<count; i++){
    var angle = Math.PI * 2 / count * i;
    var x = radius * Math.cos(angle);
    var z = radius * Math.sin(angle);
    var starBall = ballCreateFunc(0, 0, 0, 'green');
    //请注意向量的add用法
    var starBallPosition = centerBall.getPosition().add(new mono.Vec3(x, y, z));
    starBall.setPosition(starBallPosition);
    box.add(starBall);
  }
}

//创建多个球组成的圆环
starCreateFunc(box, centerBall, 10, 20, -10);
starCreateFunc(box, centerBall, 15, 30, -26);
starCreateFunc(box, centerBall, 20, 40, -42);
starCreateFunc(box, centerBall, 25, 50, -58);
```

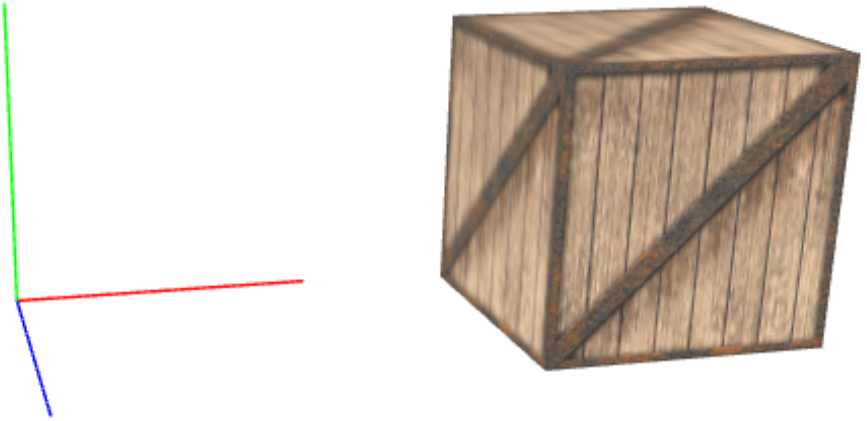


当然，这个例子如果使用setParent将中心球设置为父节点，也是可以做到类似的简化效果的。使用parent的注意事项请见下方说明。

显示原点坐标

为了调试和观察方便，可以通过函数network.setShowAxis(true)来让画布显示坐标原点的位置，通过函数network.setShowAxisText(true)来控制是否显示坐标轴文字：

```
1 network.setShowAxis(true);
2 network.setShowAxisText(false);
```



更多关于画布的使用方法，请参阅相关章节。

位置的单位是什么

一个常被问到的问题是：当设置node.setPosition(100, 100, 100)时，100代表的数值单位是什么，是100米、100厘米还是100英寸呢？其实mono并不具体约定位置的长度单位，仅仅是逻辑数值而已。在使用mono建模时，可以自行约定一个单位转换，例如“1=1cm”或其他任何约定。只要场景中的物

体都约定这一转换，这些物体就可以等比例协调的显示在一起，否则应先进行转换（或用scale进行缩放）。

当创建大场景物体（例如大型建筑物）时，为了操作方便，可以将物体进行缩小，例如1：100，然后再建模，以免模型的尺寸在建模软件中数字太大，使用麻烦。当把建筑物加载到mono场景中时，再进行比例还原。

有parent时的注意事项

mono中任何对象都可以设置父对象，自己也就变成了父对象的一个子对象。如果父对象和自对象分别设置坐标，那么自对象的坐标到底是什么含义呢？是世界坐标，还是本地坐标呢？这是一个mono使用者经常容易糊涂的一个问题。

> 注意：
> 对象的位置坐标是指相对父对象所在位置为原点的父对象本地坐标，而不是世界坐标。如果父对象的position是（100、200、300），子对象的position是（50，50，50），则自对象在世界坐标系中的绝对位置则是（150、250、350）。

举例：如果有一个桌子对象坐标（100，100，100），上面中心放一个杯子子对象，则杯子的的坐标设置为（0，0，0）即可。当然严格说杯子还有一个高度，为了让杯子保持桌面上方，应该设置杯子坐标为（0，杯子高度/2，0）更完美。

再继续思考：如果父对象移动，子对象会发生什么变化？mono的机制是：子对象永远会跟随父对象进行一起移动。由于子对象与父对象永远跟随，它相对父对象的位置并未发生任何变化，也就是其自身的position的x、y、z都没有发生任何变化。但它的世界坐标系中的绝对位置会跟着父对象的移动而变化。

为什么机柜的一半在地板下方

如果new一个水平面作为地板，再new一个立方体作为机柜，则机柜会有一半在地板的下方。这个很正常：因为地板和机柜的位置都是（0，0，0）。机柜作为一个立方体它的原点在其中心位置，也就是高度的中心点，自然有一半在地板的下方了。

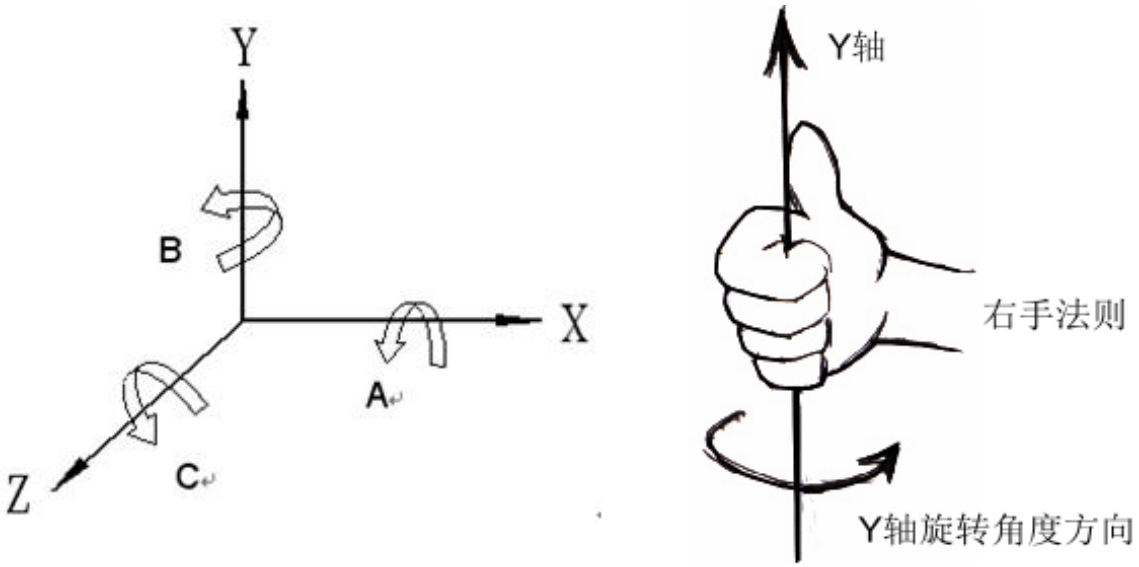
如果想要设置机柜在地板的上方，则应该设置机柜的位置y为高度height的一半：

```
var rack = new mono.Cube(100, 200, 100);
rack.setPositionY(rack.getHeight()/2);  //ok
rack.setY(100); //ok
```

- position：物体位置信息，通过get/set方法操作。默认位置为（0, 0, 0）。不同物体自身的原点位置略有差异，一般而言都是在物体的自身的中心点。例如立方体的原点在自己内部中心点。所以，当新建一个立方体并设置坐标为(0, 0, 0)时，立方体将有一半在水平坐标下方。用法举例：node.setPosition(100, 200, 300)
- rotation：物体在x、y、z轴向的旋转角度，通过get/set方法操作。默认值为（0, 0, 0）。例如，node.setRotation(0, Math.PI, 0)表示node在y轴方向旋转180度
- scale：物体在x、y、z轴方向的缩放比例，通过get/set方法操作。默认值为（1, 1, 1）。例如，node.setScale(1, 2, 1)表示node在y轴方向上放大到2倍。此时node的高度会变大（但是通过getHeight()获取的高度值height并未改变，仅仅是高度的比例增加了一倍）

旋转——Rotation

每一个3D物体都可以绕x、y、z轴进行旋转。旋转角度的方向遵循右手法则：右手拇指指向坐标轴的方向，四指的方向即旋转角度的方向。另外需要注意：角度的数值是弧度，即Math.PI=180度。如使用度，则需要转换。



下方代码演示了如何使用角度：

```
//设置旋转角度x轴30度、y轴60度、z轴90度
node.setRotation(Math.PI/180 * 30, Math.PI/180 * 60, Math.PI/180 * 90);

//设置物体绕垂直轴旋转30度
node.setRotationY(node.getRotation().y + Math.PI/180 * 30);

//获得当前某轴旋转角度
var yRotation = node.getRotation().y;
var zRotation = node.getRotationZ();
```

技巧：双击开门

3D场景中，双击打开一扇门是一个常见需求。它的本质是让门绕一个垂直的轴（一般是门的左边缘或右边缘）进行旋转。为了简化，我们假设门和墙都是一个简单的立方体，将墙挖掉一个门洞，放入门。最后为画布添加双击监听，如果双击门对象，将门旋转90度。

首先，下面代码创建一个挖了门洞的墙体和一个立方体的门。墙体挖洞，是用墙的立方体“减去”门洞立方体，得到的结果，需要用到ComboNode对象，具体请参阅其使用方法。

```
  var wall = new mono.Cube(300, 200, 20);
  wall.s({
    'm.type': 'phong',
    'm.texture.image': 'wall.png',
    'm.repeat': new mono.Vec2(30, 20),
    'left.m.repeat': new mono.Vec2(1, 20),
    'right.m.repeat': new mono.Vec2(1, 20),
    'top.m.repeat': new mono.Vec2(30, 1),
  });
  wall.setPositionY(wall.getHeight()/2);
```

```
  var cutoff = new mono.Cube(100, 180, 20);
  cutoff.s({
    'm.type': 'phong',
    'm.texture.image': 'wall.png',
    'm.repeat': new mono.Vec2(1, 18),
  });
  cutoff.setPositionY(cutoff.getHeight()/2);
  wall=new mono.ComboNode([wall, cutoff], ['-']);
  box.add(wall);

  var door = new mono.Cube(100, 180, 5);
  door.s({
    'm.type': 'phong',
    'm.texture.image': 'door.jpg',
  });
  door.setPositionY(cutoff.getHeight()/2);
  box.add(door);

  //双击门
  network.getRootView().addEventListener('dblclick', function(e){
    var filterFunc = function(element){
      return element===door;
    }
    var firstClickObject=network.getFirstElementByMouseEvent(e, false, filterFunc);
    if(firstClickObject){
      var element=firstClickObject.element;
      if(element===door){
        animateDoorOpen(door);
      }
    }
  });
}
```

执行效果如下：

上面代码中，双击门事件是通过network的getFirstElementByMouseEvent函数完成的。给他一个过滤函数，用于决定哪个对象响应用户的双击事件。这里直接返回element===door，也就是说，只有双击了门对象，才执行开门动作。

上面代码中，animateDoorOpen是具体执行门打开动画的函数。它的内容如下：

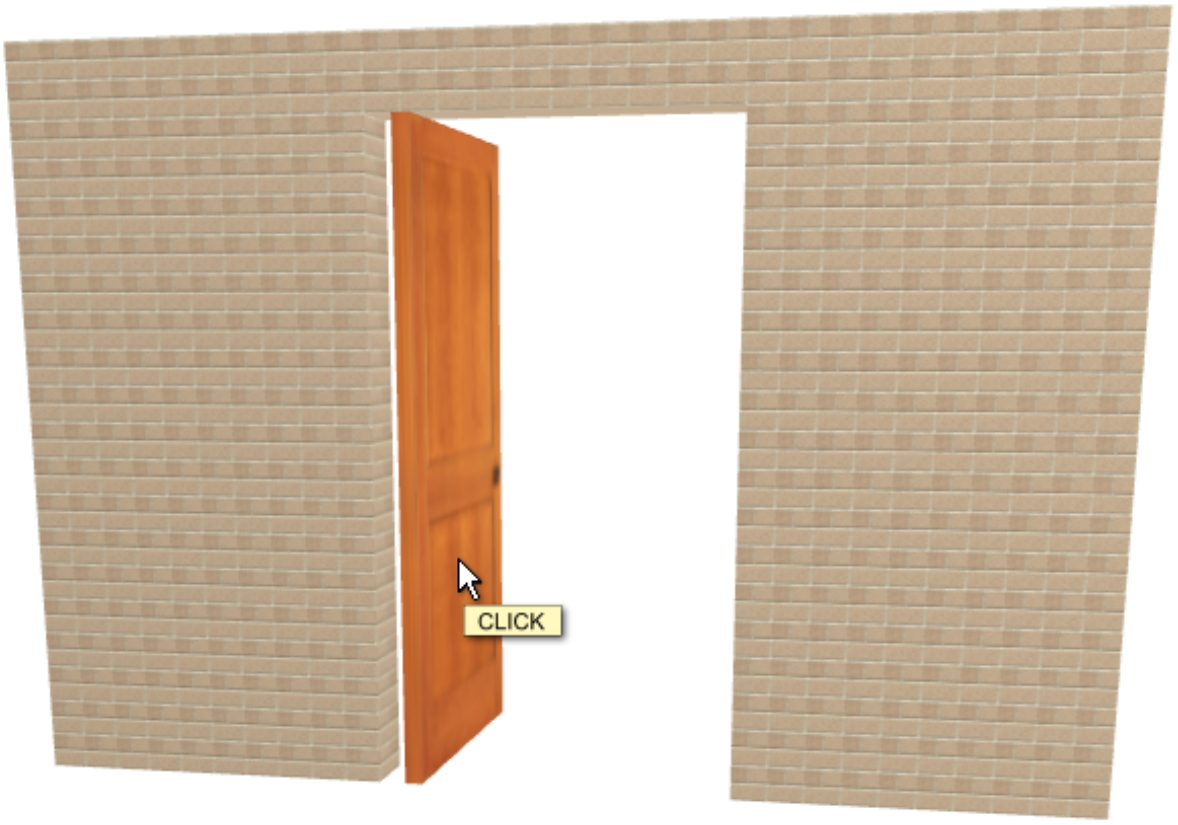
```
function animateDoorOpen(door){
  if(door.getClient('animating')) {
    return;
  }
  door.setClient('animating', true);
  var axis = new mono.Vec3(0, 1, 0);
  var axisPosition = new mono.Vec3(-door.getWidth()/2+10, 0, 0);
  var opened=door.getClient('opened');
  var animation=new twaver.Animate({
    from: 0,
    to: Math.PI/2,
    dur: 1000,
    easing: 'bounceOut',
    onUpdate: function (value) {
      var angle = value;
      if(opened){
        angle=Math.PI/2-value;
      }
      var angleStep = angle - door.getRotationY();
      door.rotateFromAxis(axis, axisPosition, angleStep);
    },
    onDone: function(){
      door.setClient('opened', !opened);
      door.setClient('animating', false);
    },
  });
  animation.play();
}
```

其中，通过'animating'这个client属性来标记"当前正在执行动画"，防止用户快速多次双击造成动画混乱。'opened'这个client属性，用于记录门是打开还是关闭状态，以便执行开门还是关门的动作。

client属性是每个3D对象携带的"哈希表"一样的容器，可以携带用户任意设置的key-value标记，方便记录一些我们自定义的状态。client属性是twaver的一个常规用法，请参阅相关章节了解其使用方法。

> 整个动画使用了twaver.Animate这个类，它是twaver的animation框架，包含在twaver 2d的twaver.js库中，因此需要引入twaver.js。

双击门，动画打开；再次双击，门动画关闭。执行效果如下图：



缩放比例——Scale

一个3D物体可以在x、y、z轴设置一个缩放比例scale数值，来改变其大小。例如一个10x10x10的立方体，在y方向设置scale为2后，最终看上去的大小就变成10x20x10。当然，此时它的尺寸还是10x10x10。3D场景中，物体最终看到的大小是由尺寸和缩放比例同时决定的。

使用scale最大的用途是设置物体的缩放变化，而不是用来设置物体的尺寸。虽然说10x10x10的立方体设置（10、10、10）的scale后最终看到的大小也是100x100x100，但在模型上它和一个size是100x100x100的立方体还是不同的。一般也不建议使用scale来设置物体的大小。

scale不像rotation、position那么常用。常用于球（Sphere）、圆柱（Cylinder）等物体的变形处理，例如扁形的球或圆柱等。也可以用于物体交互过程中的动态缩放，实现例如“双击地板从地板长出来一个机柜”这样的需求。

技巧：地板上冒出的机柜

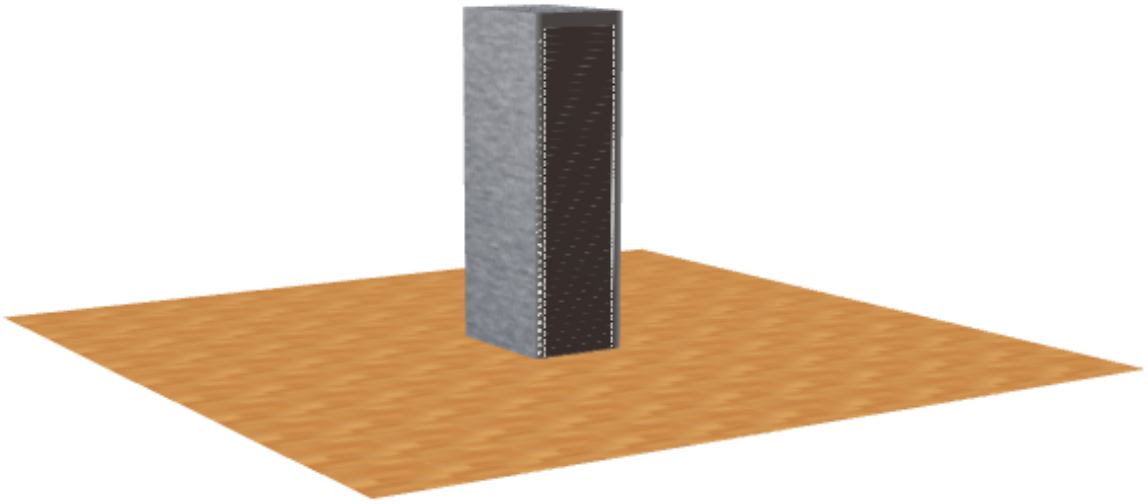
下面就用scale来实现“从地板冒出机柜”的效果。首先空间放一个水平面，然后让一个机柜从地板上动画冒出来。每个机柜的y轴scale从0到1动画变化，就可以实现这个效果。

代码如下：

```
1 var floor = new mono.Plane(500, 500);
2 floor.setRotationX(Math.PI/2);
3 floor.s({
4   'm.texture.image': 'floor.png',
```

```
  'm.repeat': new mono.Vec2(10, 10),
  'm.side': mono.DoubleSide,
});
box.add(floor);

var rack= new mono.Cube(60, 200, 80);
rack.s({
  'm.type': 'phong',
  'm.texture.image': 'metal.png',
  'front.m.texture.image': 'rack.png',
});
rack.setPositionY(rack.getHeight()/2);
box.add(rack);
```

运行效果如下：



需要注意以下两点：

1、使用mono.Plane制作地板平面时，要注意默认plane的方向是垂直的。要变成水平，需要在x轴旋转90度；

2、rack立方体默认中心位置为立方体的中心点，所以立方体会有一半在地板下方。要设置其y的位置为其高度的一半，才能让它立在地板上方；

接下来，我们使用动画，来动态修改rack的y向scale从0到1，就可以产生“地板上冒出来”的效果：

```
function animateRack(rack){
  if(rack.getClient('animating')) {
    return;
  }
  rack.setClient('animating', true);
  var animation=new twaver.Animate({
    delay: 1000,
    from: 0,
    to: 1,
    dur: 4000,
    easing: 'elasticOut',
    onUpdate: function (value) {
      var scale = value;
      rack.setScaleY(scale);
      rack.setPositionY(rack.getHeight() * scale /2);
    },
    onDone: function(){
      rack.setClient('animating', false);
    },
```

```
  });
  animation.play();
}
```

在动画中，要注意每一帧都要重新设置rack的y值，因为它的scale发生变化，会导致rack底部不再和地板平齐，要每次动态调整。

定义好动画函数后，直接调用，即可触发动画：

```
animateRack(rack);
```

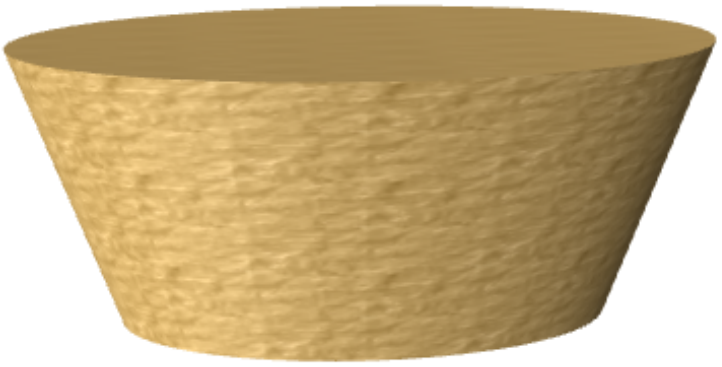
可以再进一步关联到双击等事件触发动画，就更具实际意义，这里不再赘述。

技巧：一个逼真的花盆

我们做一个盆体上粗下细、扁形的花盆。用一个圆柱体，设置顶半径大、底半径小，并设置x或z向的scale为0.6，则一个扁形的半锥花盆就做出来了：

```
var body = new mono.Cylinder(100, 70, 80, 50);
body.s({
  'm.type': 'phong',
  'm.texture.image': 'mud.jpg',
  'm.repeat': new mono.Vec2(8, 3),
});
body.setScaleX(0.6);
box.add(body);
```

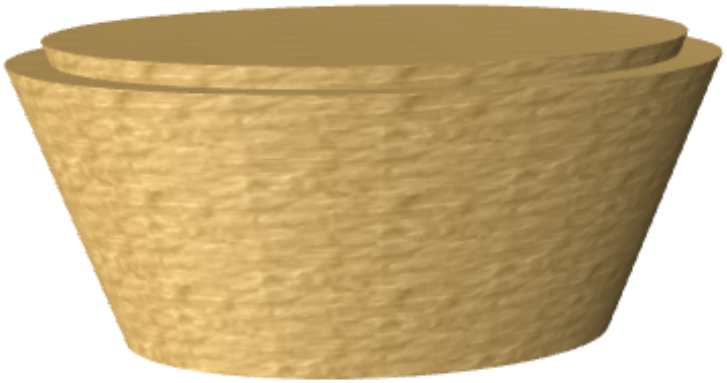
效果如下：



接下来挖空盆体。挖空的部分，其实也是一个相同形状的锥形体，只是尺寸略小。我们可以直接把body进行clone再缩小一点scale（例如0.9），再把y位置提高一点，看它的形状是否合适：

```
var cufoffScale = 0.9;
var cutoff = body.clone();
cutoff.setScale(body.getScale().x * cufoffScale, body.getScale().y * cufoffScale, boc
cutoff.setPositionY(10);
box.add(cutoff);
```

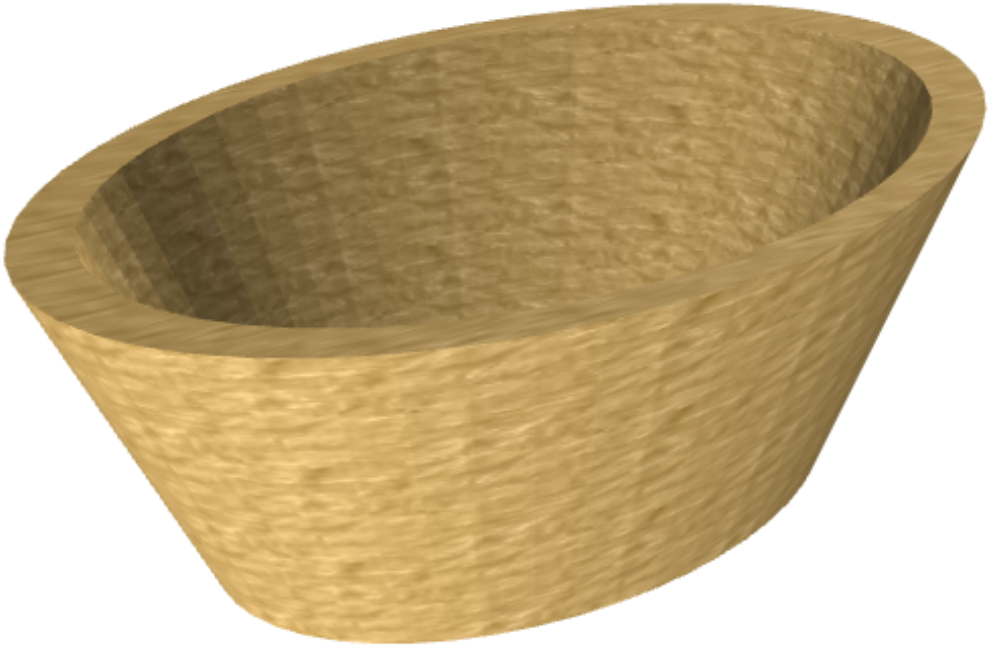
上面代码把盆体复制了一个0.9倍大小的相同物体，并把高度拉升10，效果如下：



然后，用ComboNode运算体，从body挖掉cutoff，得到新的盆体。要注意：添加下面代码后，应该注释掉body和cutoff添加到box中的代码。因为新的盆体已经产生，body和cutoff这两个素材体就没必要存在于box中了。

```
var vase = new mono.ComboNode([body, cutoff], ['-']);
box.add(vase);
```

效果如下：

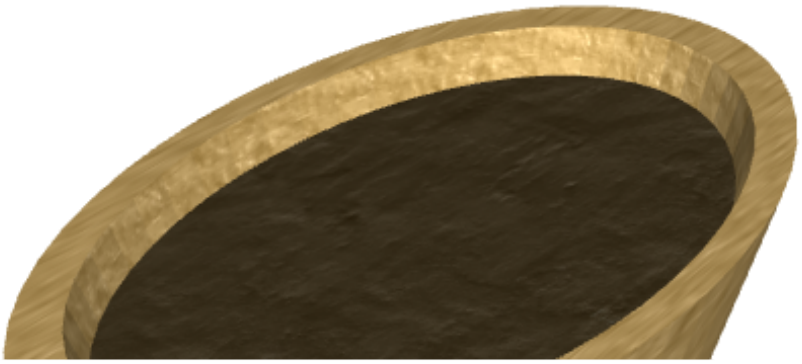


有了花盆，再放点土进去就更真实了。土放在花盆中，本质也是一个形状和花盆类似的椎体，只是高度更小一点（因为土的高度要低于花盆的上缘）。因此继续用这种clone+scale的方式。同时把土的颜色改成暗黑色，模拟土壤的样子：

^ Back to Top

```
var mudScale = 0.85;
var mud = body.clone();
mud.setScale(body.getScale().x * mudScale, body.getScale().y * mudScale, body.getSca
mud.s({
  'm.color': '#555555',
  'm.ambient': '#555555',
  'm.repeat': new mono.Vec2(1, 1),
});
mud.setPositionY(-5);
box.add(mud);
```

这里clone了一个0.7的小椎体，并去掉了纹理repeat，设置颜色为暗灰色。垂直方向再向下移动5，让土壤更低一点、和花盆内壁更紧凑一点。最后添加到box中，效果如下：



有了花盆，最后就剩下栽植物了。找一个植物的图片，注意使用png格式并处理成背景透明，设置为图片尺寸宽高为2的幂值备用。

创建一个mono.Plane，设置植物贴图到plane上，让它的位置立于土壤的上方。代码如下：

```
var plant = new mono.Plane(150, 200);
plant.s({
  'm.texture.image': 'plant.png',
  'm.transparent': true,
  'm.side': mono.DoubleSide,
});
plant.setPositionY(130);
plant.setRotationY(Math.PI*2/count*i);
box.add(plant);
```

这种方法，是将一个植物图片放在土壤上方。在旋转一定角度后，就有点露馅了：

因此，可以多放一些同样的图片，并旋转一定的角度均匀排列。改造代码如下：

```
var count = 6;
for(var i=0;i<count;i++){
  var plant = new mono.Plane(150, 200);
  plant.s({
    'm.texture.image': 'plant.png',
    'm.transparent': true,
    'm.side': mono.DoubleSide,
  });
  plant.setPositionY(130);
  plant.setRotationY(Math.PI/count*i);
  box.add(plant);
}
```

上面代码一次放置了6片贴图并在180度内（不必360度，因为一片贴图会分布在左右两侧）均匀排列。这样旋转起来，就不会露馅了：

用这种方法，稍加修改贴图、形状，就可以做出更多种类的盆景、植物模型。

使用Style注意事项

基本使用原则

Style是mono对象内置的属性样式。在使用style之前，一些概念请熟知：

- Style是内置的样式表，因此不要将自定义的属性放置在style中。不要使用node.setStyle('my.property', 'my value')这样的代码来存储自己定义的属性。使用Client来保存自己定义的属性，例如：node.setClient('my.property', 'my value')。这样会避免冲突和混乱；
- Style中用到的各种贴图、影射图，为了达到最佳效果，强烈建议都使用2的幂值尺寸提供贴图。例如256×256、512×512、128×1024等；
- 要清除一个属性，直接将属性值设置为null或undefined即可；

Style和Client的区别

从机制上，style和client都是mono对象携带的一个"属性表"，里面以key-value的形式存储对象的属性。不通的是，style可以认为是mono自用的属性表，里面放置了大量mono预定义的属性，不建议开发者把自定义属性放在这里面（虽然api上可以进行set）；client则是专供开发者存储自定义属性的地方。

设置Style的方法

每一个3D对象都有很多style属性来控制其外观样式。style是通过一个"key-value"的结构进行设置和维护的，统一通过setStyle/getStyle进行设置或获取。

函数setStyle也可以简写为s，setStyle也可以连写，以简化代码。

```
//方法一：一次设置一个属性
node.setStyle('m.color', 'red');
node.setStyle('m.ambient', 'red');
node.setStyle('m.color', 'red').setStyle('m.ambient', 'red')//也可以连写

//方法二：一次设置多个属性
node.s({
  'm.color': 'red',
  'm.ambient': 'red',
});
```

还有一个重要的用法是：style可以单独设置在对象的某一个具体面上，而不影响其他面的属性。此时需要把面的id放在style key的最前面。

例如：要设置一个立方体的顶面为红色，其他面为绿色，而且已知立方体顶面的id是'top'，则代码如下：

```
var node = new mono.Cube(10, 10, 10);
node.s({
  'm.color': 'green',
  'top.m.color': 'red',
});
```

具体3D对象有哪些面、这些面对应的id是什么，需要查阅相关对象介绍。简单的id都容易猜到，例如立方体的六个面id分别是'top', 'bottom', 'front', 'back', 'left', 'right'，圆柱体的顶面、底面、侧面分别是'top', 'bottom', 'side'等等。

下面列出各类常用样式，供开发者速查。

基本样式

m.texture.image

材质贴图。可以是图片url，可以是base64编码的url，也可以是一个canvas对象。

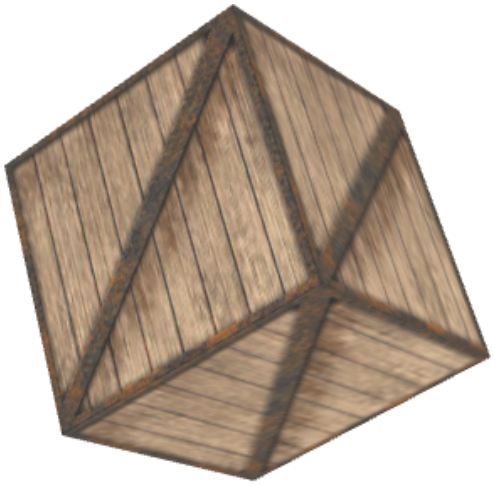
使用图片贴图需要注意以下事项：

- 图片支持的格式：可以是jpg、gif、png等web常用格式
- 图片越大越好吗？图片越大越高清，在3D中显示越清晰。但过大的图片尺寸会带来内存和性能方面的消耗，因此一般不建议超过1024。由于GPU会进行动态运算，因此小尺寸的图片在3D中也不会出现2D中常见的“锯齿”，仅是清晰度较低、较模糊而已
- 图片尺寸有要求吗？图片尺寸（包括宽和高）强烈建议设置为2的幂值，也就是例如64、128、256、512、1024….，这样可以有效的避免贴图闪烁问题，还可以提高性能。这也是底层opengl及显卡的一贯要求，具体请搜索相关理论文章
- 使用png是否可以透明？如果使用有透明区域或透明度的png图片，默认贴上去也不会有透明效果的，透明区域会默认显示白色。如果需要透明，要配合启用m.transparent属性，设置为true即可

```
//使用图片
node.setStyle('m.texture.image','../images/default_texture.png');
//使用canvas对象
var canvas = document.createElement('canvas');
canvas['width'] = 128;
canvas['height'] = 128;
var context = canvas.getContext('2d');
context.fillStyle = 'red';
context.fillRect(10, 10, 50, 50);
node.setStyle('m.texture.image', canvas);
```

假设准备一张图片'box.jpg'，使用图片材质举例：

```
cube.s({
  'm.texture.image': 'box.jpg',
});
```



m.type

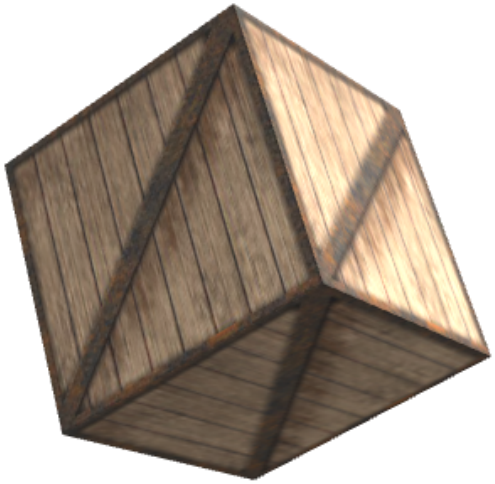
材质的类型。可以设置basic或phong

可以设置basic和phong两个类型：

- ‘basic’：基本类型，不支持光照，效率最高
- ‘phong’：冯氏类型，支持光照

下面代码为立方体设置贴图，并设置为phong类型：

```
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
});
```



注意

- 请注意设置phong类型后，贴图外观的变化
- 使用phong类型贴图，必须要配合使用灯光对象，同时注意设置灯光的强度、颜色及摆放位置。具体见灯光对象的使用介绍
- phong类型不如basic类型效率高，但一般情况下对效率影响不大

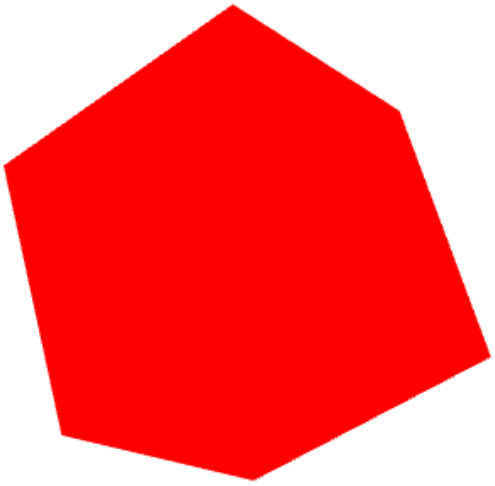
m.color

材质的固有颜色，默认白色。

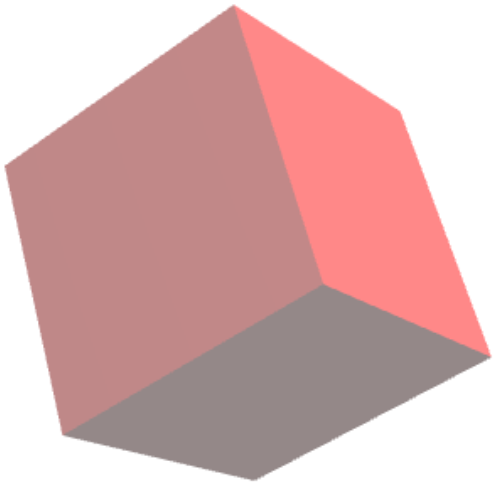
使用图片贴图时，一般不需要设置固有颜色（也可以认为固有颜色是白色），除非需要对图片进行“染色”。3D物体可以设置图片贴图，也可以设置一个固定的纯颜色（例如红色），还可以混合使用图片贴图+颜色。混合使用时，颜色会对图片进行“染色”。

basic类型，仅使用颜色，不使用图片贴图，物体表现为纯色，没有层次感：

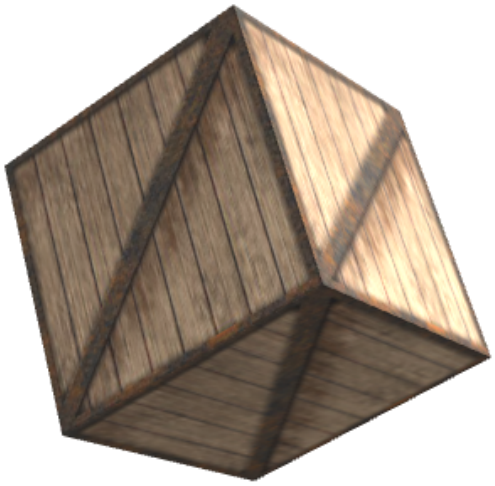
```
1 node.setStyle('m.color', 'red');
```



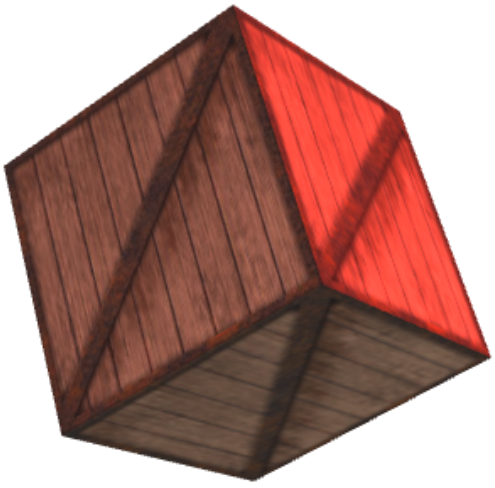
启用phong类型，增加层次感：



使用贴图，不使用颜色：



混合使用图片和颜色：



m.ambient

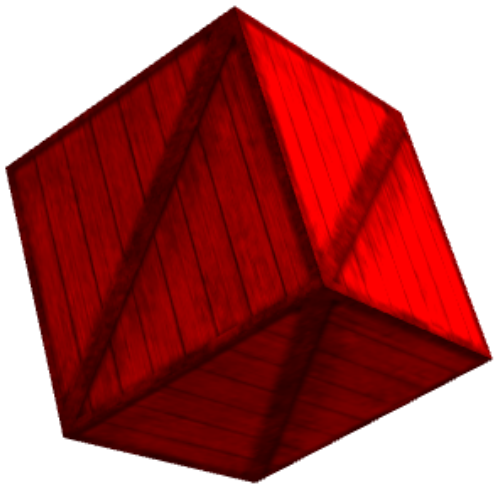
材质的反光颜色，默认白色。

m.color是物体的固有颜色，而m.ambient则是对光照反射出来的颜色。一般m.ambient配合m.color使用，两者设置相同数值，让物体的颜色或染色更“彻底”。该属性属于对光照的控制，只有phong类型才有意义。

在3D应用中，动态设置物体染色，可以用于“设备告警”、“故障提示”等用途。也可以做例如物体被“选中”等效果。

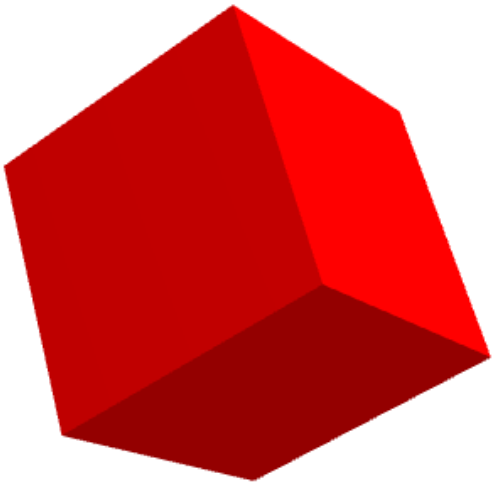
basic类型，仅使用颜色，不使用图片贴图，物体表现为纯色，没有层次感：

```
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
  'm.color': 'red',
  'm.ambient': 'red',
});
```



如不用图片，仅用颜色，效果如下：

```
cube.s({
  'm.type': 'phong',
  'm.color': 'red',
  'm.ambient': 'red',
});
```



m.texture.repeat

材质的重复次数，是一个mono.Vec2数值。

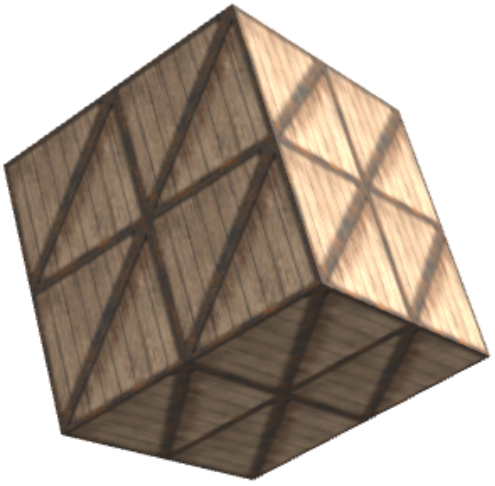
纹理重复是指纹理沿横向和纵向的重复次数，例如墙、底面等，用一个小的贴图进行不断的重复进行贴图。重复次数是一个（x、y）二维数值，类型是mono.Vec2。默认值是（1、1），重复一次，也就相当于没有重复。设置new mono.Vec2(5, 2)则在横向重复5次，纵向重复2次。

注意
使用贴图重复，贴图的尺寸必须设置为2的幂值，否则贴图在重复一次后，会用最后一排（列）像素贴剩余的区域，导致显示异常。相关原理请查阅有关贴图重复的3D理论。

特殊用法
重复次数也可以设置为小数。例如设置为0.5则是重复半次，也就是使用了贴图的一半。

设置2×2次纹理重复：

```
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
  'm.texture.repeat': new mono.Vec2(2, 2),
});
```



设置（0.5、0.5）次重复效果：

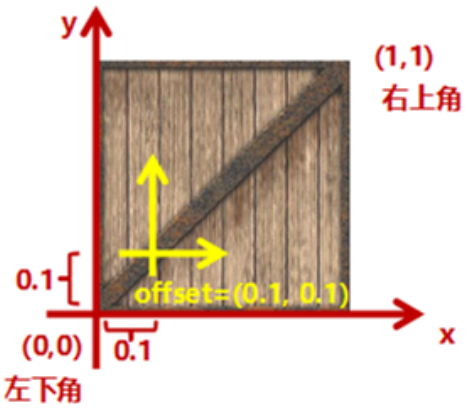
```
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
  'm.texture.repeat': new mono.Vec2(0.5, 0.5),
});
```

m.texture.offset

纹理重复的起始偏移，是一个mono.Vec2数值，分别定义横向和纵向在图片上坐标的偏移。mono在绘制图片时，会从这个偏移的点开始绘制。偏移一般是一个小数，(0.5, 0)表示横向纹理从半个图片开始重复。所以偏移设置成整数没有什么意义。

> 3D中常称贴图在物理表面的对应坐标为"UV"坐标。这里是指u,v两个坐标轴(它和空间模型的X, Y, Z轴是类似的)。它定义了图片上每个点的位置的信息。图片的横向用U、纵向用V，左下角为坐标原点。UV坐标与3D模型是相互联系的, 用来决定物体表面的纹理贴图的位置。

下图解释了一个贴图在一个物体表面贴图时，从图片左下角(0.1, 0.1)偏移的位置开始做贴图：



做（0.1, 0.1）偏移，代码如下：

```
var cube=new mono.Cube(100, 100, 100);
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
  'm.texture.repeat': new mono.Vec2(1, 1),
  'm.texture.offset': new mono.Vec2(0.1, 0.1),
});
box.add(cube);
```

映射关系如下图：

m.specularStrength

材质的反光强度，默认值1。

反光强度只有在phong类型材质下才有意义，并和灯光的位置、灯光强度、镜头的角度有关。数值越大，反光强度和聚焦度也会越大、更“镜面”感。

> 使用m.specularStrength可以控制简单的反光效果；想要设置复杂的有纹理有环境贴图的反射效果，请参阅下方的m.envmap.image和m.specularmap.image参数。

使用高光表面的情形：

- 镜面、玻璃表面
- 光滑度较高的表面，如金属等

注意过大的数值或过强强度的灯光，会导致物体表面过亮，最终一片白色。

注意
对于高光并不均匀（例如凸凹不平的“毛玻璃”表面）的物体表面，可以使用高光映射图片（specularmap）机制，进行更加精确控制。详见m.specularmap介绍。

默认值1：

设置为50：

```
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
  'm.specularStrength': 50,
});
```

设置为100：

```
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
  'm.specularStrength': 100,
});
```

m.wireframe

启用三维线模式，默认false。

三维线模式下，物体将不再渲染表面，仅将顶点连线用线条显示。三维线模式可以更方便的观察物体的顶点、场景的的空间透视情况，也可以用作“物体虚化”等特效。三维线模式下，三维线的颜色就是材质本身的颜色（m.color）。同样，三维线本身也可以设置phong类型，也可以设置高光、反光等属性，只是一般实用意义不大。

设置立方体wireframe模式：

```
cube.s({
  'm.wireframe': true,
});
```

```
var torus = new mono.Torus(100, 30, 40, 20);
torus.setRotationX(Math.PI/2);
torus.s({
  'm.color': 'red',
  'm.wireframe': true,
});
```

m.visible

设置材质是否可见。当设置false后，物体材质将不可见（物体由材质组成，因此物体也将不可见）。默认true。用此属性可以动态隐藏/显示一个3D物体，或3D物体的一个（一些）面。例如：双击立方体，将立方体的顶面隐藏，看到立方体内部情况。隐藏3D物体某个面，需要配合使用物体面的id。例如隐藏立方体的顶面，则要使用'top'这个立方体顶面id，将其作为前缀即可：

```
cube.setStyle('top.m.visible', false);
```

具体请查阅各3D对象详细说明。

设置立方体右侧面不可见：

```
cube.s({
  'right.m.visible': false,
});
```

注意：
截图中消失的面的区域变成了白色，看上去有点奇怪。实际上这个面的消失露出了后面的白色环境背景。因此一般单面透明应配合m.side的双面材质来使用，见下方说明。

m.transparent

设置材质是否启用透明。m.visible是控制整个材质是否可见，而m.transparent则可以控制材质的部分透明。请注意：启用透明不意味着一定透明，材质的透明区域和像素点取决于贴图是否有透明像素点，所以这要求材质要使用支持透明的图片，例如png。

默认不启用透明。此时，即使使用有透明区域的png图片，物体表面也不会有透明效果，而是用白色进行填充。

我们先制作一个中间被挖空的png材质图片：

通过下面代码设置到立方体的front面：

```
var cube=new mono.Cube(100, 100, 100);
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
  'm.side': mono.DoubleSide,
  'front.m.texture.image': 'box-transparent.png',
});
box.add(cube);
```

上面代码并未设置m.transparent为true，因此看上去会如下图：

透明区域被填充为白色。设置m.transparent为true，效果如下：

```
var cube=new mono.Cube(100, 100, 100);
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
  'm.side': mono.DoubleSide,
  'front.m.texture.image': 'box-transparent.png',
  'front.m.transparent': true,
});
box.add(cube);
```

则产生局部透明效果：

透明表面叠加的严重问题

在使用m.transparent参数来设置物体表面透明时，如果多个透明表面相互叠加，有时候会产生遮挡关系错乱的现象。这是因为透明时，深度检测很难区分先后绘制关系，导致后面的面先绘制，反过来又挡住了前面不透明的区域。

要解决遮挡错乱的问题，应该：

- 尽量避免多个透明面相互遮挡；
- 不启用m.transparent,而使用m.alphaTest。具体见后续m.alphaTest参数介绍；
- 通过建模创建镂空区域，而不使用透明图片。例如用外部建模工具建模，或用mono.ComboNode进行挖除运算等；

m.opacity

不透明度。它是一个0-1的值，是针对m.transparent设计的。如果启用了m.transparent，则贴图的不透明度用m.opacity控制，默认值是1，完全不透明。

下面代码设置了立方体front面透明贴图、半透明材质、染色：

```
var cube=new mono.Cube(100, 100, 100);
cube.s({
  'm.type': 'phong',
  'm.side': mono.DoubleSide,
  'm.texture.image': 'box.jpg',
  'front.m.texture.image': 'box-transparent.png',
  'front.m.transparent': true,
  'front.m.color': '#0B4C5F',
  'front.m.ambient': '#0B4C5F',
  'front.m.opacity': 0.6,
});
box.add(cube);
```

opacity还可以配合envmap等技术混合使用，产生更逼真的物理效果。下面的代码创建了一个半透明的、高光反射的球体：

```
var cube=new mono.Cube(100, 100, 100);
cube.s({
  'm.type': 'phong',
  'm.side': mono.DoubleSide,
  'm.texture.image': 'box.jpg',
});
box.add(cube);

var ball = new mono.Sphere(35,100);
ball.s({
  'm.type': 'phong',
  'm.opacity': 0.8,
  'm.transparent': true,
  'm.envmap.image': ['posx.jpg','negx.jpg','posy.jpg','negy.jpg','posz.jpg','negz.jpg'],
});
ball.setScaleY(0.4);
ball.setPositionY(50);
box.add(ball);
```

球体有透明度，同时也有反光：

m.alphaTest

设置alpha测试的透明阀值。alpha测试是和m.transparent对应的另外一种设置透明区域的方法，两者不应该同时启用。alpha测试的原理是在绘制一个表面时，先对图片的透明像素进行alpha检测，如果大于阀值则认为像素透明、不绘制，否则认为不透明，绘制贴图内容。这样就产生了透明效果。

alphaTest是一个0-1之间的小数。alphaTest不能和m.transparent同时使用。

优点：alpha测试简单方便，而且可以有效的避免m.transparent产生的“多个透明表面叠加产生遮挡错乱”的问题。
缺点：alpha测试的透明效果不够细致，透明区域的边缘会产生锯齿或白色残留。

下面代码是使用m.transparent设置了立方体6个面为透明，产生了遮挡错乱：

```
var cube=new mono.Cube(100, 100, 100);
cube.s({
  'm.type': 'phong',
  'm.side': mono.DoubleSide,
  'm.texture.image': 'box-transparent.png',
  'm.transparent': true,
});
box.add(cube);
```

下面的代码去掉了m.transparent设置，而使用了alphaTest方法，再看效果的变化：

```
var cube=new mono.Cube(100, 100, 100);
cube.s({
  'm.type': 'phong',
  'm.side': mono.DoubleSide,
  'm.texture.image': 'box-transparent.png',
  'm.alphaTest': 0.9,
});
box.add(cube);
```

虽然alpha测试有效的解决了遮挡关系的问题，但也产生了比较粗糙的透明边缘：

经验

实际使用中，有的物体即使遮挡关系发生一定的错误，也不会很大的影响视觉，例如多片植物贴图按角度排列模拟一个复杂植物，此时可以使用m.transparent。如果有非常清晰锐利的透明贴图，且其透明形状具有物理意义，则可以启用m.alphaTest关闭m.transparent。

当然，最彻底的解决方法就是建模，用模型做出物体的实际物理形状，而不用透明贴图去模拟。上面的例子中，可以用Cube挖去6个Sphere球体得到一个ComboNode，同样可以达到相同的效果。

m.side

设置材质是双面还是单面。可以设置正面、反面、双面。默认值是正面。

- mono.FrontSide：材质设置正面，反面不存在（透明），默认值；
- mono.BackSide：材质设置反面，正面不存在（透明）；
- mono.DoubleSide：材质设置双面，正反面都用相同材质；

注意：请注意未设置材质的一面，是“未设置材质”，看不见、摸不到，相当于透明、不存在，而不是白色。

设置双面材质：

```
cube.s({
  'right.m.visible': false,
  'm.side': mono.DoubleSide,
});
```

设置更多面不可见：

```
cube.s({
  'm.side': mono.DoubleSide,
  'right.m.visible': false,
  'front.m.visible': false,
  'top.m.visible': false,
});
```

高级属性

m.specularmap.image

它依赖于参数m.envmap.image，请见下一节。

m.envmap.image

m.envmap.image是环境贴图，m.specularmap.image是环境贴图的反光强度控制。这两个参数，后者是专门为前者服务的，不使用前者，后者则没有任何作用和意义。

特别注意：
m.envmap.image+m.specularmap.image与m.specularStrength的联系和区别：
这两种方案都可以对光照反射进行控制，但是它们是“互斥”的两种方案，实际使用时只能选其一，不能混合使用。

- 环境贴图方案：环境贴图方案是用环境天空盒+反射控制图片，对物体表面的反光进行非常精确的控制，是一种复杂的仿真效果，需要较多贴图，可以做出有纹理、有凸凹感的反射，需要对反光原理有所理解；
- specularStrength反光强度方案：这种方案只使用一个数字参数来控制整个面对光源的反光强度，反光时，光源会在物体表面形成渐变的光晕，强度越大光晕越明亮，强度越小反光光晕越暗淡。这种方案简单方便，但是质感平滑单一，没有凸凹感；

环境贴图是六张图片url组成的字符串数组，代表一个环境立方体的6面贴图。环境贴图会被反光的表面反射出来，和镜面或光滑表面可以反射背景类似。环境贴图是一个非常有效的提高物体表面质感、光滑度的方法。它可以逼真的模拟镜面反射效果，对于比较光滑的表面，设置环境贴图可以增加逼真度。m.specularmap.image则是一张定义明暗区域的贴图，它可以控制一个面的每个区域反射环境贴图的强度。越亮的区域（specularmap贴图白色的部分）反光约多、越明亮（如镜面），越暗的地方则反光越少，越接近物体本身的贴图材质。

一般而言，物体都不会是镜面一样完全反射环境贴图，而是既有自身的材质，又有叠加的环境反射，例如尤其表面的桌面等。因此，环境贴图不一定非要高清，而比较暗淡模糊的环境反射也可以很好的增加材质的质感。越镜面的表面，可以设置更白的specularmap以及真实高清的envmap，而越粗糙的表面则可以设置更暗淡的specularmap，同时可以用有纹理的specularmap来控制反光的纹理和颗粒感。

对于一些室内场景（例如机房里面的机柜反射的房间内场景），模糊暗淡一点的环境贴图就足可以增加质感，而不必使用尺寸很大的高清贴图，降低GPU的负担。在机房中，使用一套高清的蓝天白云环境贴图，还不如使用一张模糊的、顶部有天花板灯光的环境贴图更接近真实场景。

实际使用中，为了简单方便，也可以把数组中6个图片都直接使用一张相同的贴图，也可以获得较好效果。总之对于一般的应用而言，环境贴图不必是真实场景，用简单模糊的贴图增加一定的质感即可。

环境反射还可以配合半透明、染色、normalmap等属性一起使用，增加逼真度。

假设准备了一张如下贴图作为环境贴图：

我们使用下方代码来把它作为环境贴图用在立方体对象中：

```
var pic = 'envmap.jpg';
var envmap = [pic, pic, pic, pic, pic, pic,];
var cube= new mono.Cube(100, 100, 100);
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
  'm.envmap.image': envmap,
});
box.add(cube);
```

上面的例子是采用模糊环境贴图的做法。对于比较光滑的镜面，物体自身发出的光线很少，主要以环境反射为主，此时更清晰的环境贴图会更真实。

下面例子准备了一个真实的6个天空盒贴图，并将其设置为立方体的环境贴图。为了模拟镜面效果，立方体本身不再设置贴图：

```
var envmap=['posx.jpg','negx.jpg','posy.jpg','negy.jpg','posz.jpg','negz.jpg'];
var cube= new mono.Cube(100, 100, 100);
cube.s({
  'm.type': 'phong',
  'm.envmap.image': envmap,
});
box.add(cube);
```

也可以保留立方体本身的贴图，贴图和反射图混合、融合，形成“半反射”的效果：

```
var envmap=['posx.jpg','negx.jpg','posy.jpg','negy.jpg','posz.jpg','negz.jpg'];

var cube= new mono.Cube(100, 100, 100);
cube.s({
```

```
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
  'm.envmap.image': envmap,
});
box.add(cube);
```

将Cube改为Sphere对象：

```
var ball= new mono.Sphere(100, 100, 100);
ball.s({
  'm.type': 'phong',
  'm.envmap.image': envmap,
});
box.add(ball);
```

进一步，可以把某一个面设置为反射环境贴图。环境贴图的反射强度可以由'm.specularmap.image'进行控制，如果使用纯白色，则意味着镜面一样的完全反射。下面代码创建了一个单面镜面反射的立方体：

```
var envmap=['posx.jpg','negx.jpg','posy.jpg','negy.jpg','posz.jpg','negz.jpg'];

var cube= new mono.Cube(100, 100, 100);
cube.s({
  'm.type': 'phong',
  'm.side': mono.DoubleSide,
  'm.texture.image': 'box.jpg',
});
cube.s({
  'front.m.texture.image': 'white.png',
  'front.m.envmap.image': envmap,
  'front.m.specularmap.image': 'white.png',
});
box.add(cube);
```

下图是综合使用了染色、半透明、环境贴图的效果，您可以亲自试一试：

下面代码使用了一个有皮革纹理的specularmap控制了反光贴图，在木材贴图的表面叠加产生了皮革纹理质感，同时还能反射出淡淡的环境贴图（因为皮革贴图较暗所以反射的环境也较淡）：

```
var envmap=['posx.jpg','negx.jpg','posy.jpg','negy.jpg','posz.jpg','negz.jpg'];

var cube= new mono.Cube(100, 100, 100);
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
  'm.specularmap.image': 'leather.jpg',
  'm.envmap.image': envmap,
});
box.add(cube);
```

继续变化。使用下面一张有明暗纹理和文字的specularmap+纯色材质，混合效果如下：

```
var cube= new mono.Cube(100, 100, 100);
cube.s({
  'm.type': 'phong',
  'm.color': 'green',
  'm.specularmap.image': 'specularmap.jpg',
  'm.envmap.image': envmap,
});
box.add(cube);
```

继续增加一定的半透明，则可以呈现“又透光、又反光”的效果：

```
var cube=new mono.Cube(300, 300, 1);
cube.s({
  'm.texture.image': 'box.jpg',
});
box.add(cube);

var envmap=['posx.jpg','negx.jpg','posy.jpg','negy.jpg','posz.jpg','negz.jpg'];

var ball= new mono.Sphere(100);
ball.s({
  'm.type': 'phong',
  'm.side': mono.DoubleSide,
  'm.texture.image': 'earth.png',
  'm.specularmap.image': 'specularmap.jpg',
  'm.envmap.image': envmap,
  'm.transparent': true,
  'm.opacity': 0.8,

});
box.add(ball);
```

可见，灵活使用specularmap，即使用纯色，不使用贴图，也可以做出有质感的材质表面。

m.normalmap.image

法线贴图，是一张纹理图，用来控制物体表面的凸凹高度，也就是物体表面法线方向偏移的控制。纹理图中，每个像素的RGB分别存储该像素对应法线的XYZ分量，只要把法线的分量由（-1，1） 映射成（0，255）就可了。观察一张法线图，以蓝色为主，是因为朝向图面外的法线（0，0，1）都被编码成（0，0，127）了（读入OpenGL后即 (0,0,0.5)），而图上越红的地方表明法线越向右，越绿的地方表明法线越向上，就可以理解了。总体来说，就是一张紫蓝色的图。发现贴图一般由美工制作，在photoshop中可以使用插件制作。

用normalmap制作凸凹平面，实际上表面依旧是一个平面，只是通过干扰每一个像素点的光线反射来“欺骗”了人的眼睛而已。这种方法简单方便，效果细腻，避免通过建模来生成真实的表面顶点，增加了物体的质感，提高了建模的效率，在3D技术中已经被广泛采用。

当然使用normalmap也会降低一定的渲染效率，实际使用中可权衡效率和效果来适当采用。

更多关于normalmap的原理，请参阅各大搜索引擎。

准备一张normalmap图片如下图（注意设置成2的幂尺寸）：

将其设置为立方体的normap贴图：

```
var cube=new mono.Cube(100, 100, 100);
cube.s({
  'm.color': 'green',
  'm.ambient': 'green',
  'm.type': 'phong',
  'm.normalmap.image': 'normalmap.jpg',
});
box.add(cube);
```

通过放大细节可以看出，normalmap可以有效的在表面生成不规则的、难以通过建模实现的凸凹感：

再试一个贴图：

normalmap还可以和正常的图片贴图一起使用，产生融合的效果：

```
cube.s({
  'm.type': 'phong',
  'm.normalmap.image': 'normal.png',
  'm.texture.image': 'box.jpg',
});
```

m.lightmap.image

光照贴图。光照贴图是用一张单独的灰度图贴图，对光照进行控制，就是将物体表面的目标点和光照贴图的控制点明暗进行融合，再进行渲染。光照贴图是一种“预处理”技术，预先把想要的光照效果提前做出来用贴图保存，在渲染时直接使用，打到快速产生光照效果的作用。

使用光照贴图，可以代替通过光源进行实时光照计算和渲染。而且光照贴图可以提前进行非常精细的绘制，效果更加逼真。光照贴图的缺点是，它是一个静态的图片，渲染出来的光照效果也是固定不变的，即使场景中的灯光发生了变化，光照效果也不会跟着改变。

所以，光照贴图特别适合做静态的展示场景，而不适合物体频繁移动的场景。一般而言，使用光照贴图的表面，’m.type’就不必再使用’phong’类型，而直接使用’basic’即可。因为phong类型表面会实时计算光照效果，而使用贴图后光照效果已经在lightmap中固化，就不需要实时计算了，因此使用不响应光照的basic类型即可，这也可以适当提高效率。当然使用lightmap本身也会消耗一些性能，可权衡使用。

下面代码把一个basic类型材质的立方体前面的一个面设置了下方的lightmap贴图：

```
var cube=new mono.Cube(100, 100, 100);
cube.s({
  'front.m.lightmap.image': 'lightmap.jpg',
  'm.texture.image': 'box.jpg',
});
box.add(cube);
```

效果如下：

下面代码创建了一个沿着path路径走向的墙体。直接使用basic类型和单色渲染：

```
var path = new mono.Path();
path.moveTo(-50, -100, 0);
path.lineTo(50, -100, 0);
path.lineTo(50, 100, 0);
path.lineTo(-50, 100, 0);

var wall = new mono.PathCube(path, 10, 100);
wall.s({
  'm.color': 'green',
  'm.ambient': 'green',
});
box.add(wall);
```

虽然添加了灯光，可basic类型材质不会产生任何光照效果。

如果使用phong类型：

```
wall.s({
  'm.color': 'green',
  'm.ambient': 'green',
  'm.type': 'phong',
});
```

phong类型材质可以动态根据光源位置计算出自身光照效果，但仍不够细腻。

假如我们想让墙从上到下产生渐暗的光线效果，先准备这样一张简单的渐变灰度图：

然后再用lightmap参数进行设置：

```
wall.s({
  'm.color': 'green',
  'm.ambient': 'green',
```

```
  'm.type': 'phong',
  'm.lightmap.image': 'lightmap.jpg',
});
```

墙体mono.PathCube对象通体各面设置同一个lightmap并不合适，例如顶面的光照效果更适合高亮。因此这里对墙体的inside和outside两个面单独设置光照贴图，其他面保持纯色。调整一下墙体颜色，并取消了phong类型，直接用默认的basic类型：

```
var wall = new mono.PathCube(path, 10, 100);
wall.s({
  'm.color': '#CEF6F5',
  'm.ambient': '#CEF6F5',
  'outside.m.lightmap.image': 'lightmap.jpg',
  'inside.m.lightmap.image': 'lightmap.jpg',
});
box.add(wall);
```

竟然也可以获得非常好的光影效果。如果再设置纹理贴图，则可以更真实：

```
var wall = new mono.PathCube(path, 10, 100);
wall.s({
  'm.color': '#CEF6F5',
  'm.ambient': '#CEF6F5',
  'outside.m.lightmap.image': 'lightmap.jpg',
  'inside.m.lightmap.image': 'lightmap.jpg',
  'm.texture.image': 'wall_texture.png',//设置砖墙贴图
});
box.add(wall);
```

专用属性

m.vertical

专用于控制mono.Billboard是否始终保持垂直于水平面。默认false，始终保持和镜头lookat方向垂直。

下面代码演示了m.vertical分别为true和false的情形：

```
var cube=new mono.Cube(30, 30, 30);
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
});
box.add(cube);

var billboard = new mono.Billboard();
billboard.s({
  'm.texture.image': 'billboard.png',
  'm.vertical': false,
});
billboard.setScale(20,30,1);
billboard.setPositionY(30);

box.add(billboard);
```

```
billboard.s({
  'm.texture.image': 'billboard.png',
  'm.vertical': true,
});
```

当true时，billboard始终保持正面平行面对用户眼睛，可以方便的让用户看清billboard的数值和内容，但随着镜头的视角旋转，物理上并不会与底面保持垂直，感觉上会有点怪。当false时，物理上感觉更合理，但是随着旋转角度的不同，billboard面对用户的角度也会发生变化，一定程度上会影响用户对billboard内容的观察。

参数设置无优劣之分，可实际需求进行变化。一般而言，如果billboard是显示的温度、湿度等纯数值信息，没有太多物理意义，可保持false，让用户更清晰的看清billboard的内容。如billboard是植物、花束、树木、装饰标牌等，具有一定的物理意义，则可以设置为true，让它看上去更加真实。

m.texture.flipX/flipY

对贴图进行横向/纵向反转，默认false。下面的代码演示了反转前和翻转后的效果：

无反转：

var cube=new mono.Cube(30, 30, 30);

cube.s({

‘m.type’: ‘phong’,

‘m.texture.image’: ‘box.jpg’,

});

box.add(cube);

front面flipX反转：

```
var cube=new mono.Cube(30, 30, 30);
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
  'front.m.texture.flipX': true,
});
box.add(cube);
```

m.polygonOffset

m.polygonOffset是为了解决z-fighting（或称为深度冲突）而设置的参数。在“镜头”章节我们介绍了为什么当物体两个表面距离过近会产生闪烁现象。实际上在3D中，如果发现有画面闪烁的问题，大多数情况下是z-fighting引起的。他的本质是引擎在所设置的场景精度中无法区分过近两个面的深度（z，Depth），也就是距离镜头的距离。

深度偏移包含3个具体参数，它们总是要一起使用：

- m.polygonOffset: boolean值，是否启用深度偏移几只。默认false
- m.polygonOffsetFactor：深度偏移值，设置成和m.polygonOffsetUnits相同即可。整数表示深度增加（离镜头更远），负数表示深度减少（离镜头更近）；
- m.polygonOffsetUnits：深度偏移值，设置成和m.polygonOffsetFactor相同即可。整数表示深度增加（离镜头更远），负数表示深度减少（离镜头更近）；

深度偏移数值一般不宜过大，具体数值和场景大小和镜头参数有关。过小可能不能解决闪烁现象，过大可能产生遮挡关系错

误。实际使用中可以通过反复调试确定。

下面用代码来复现这一问题。假如我们要在箱子的表面贴一张纸，这张纸用一个立方体来模拟。纸所在的平面和箱子所在的平面将会重叠。代码如下：

```
var cube=new mono.Cube(100, 100, 100);
cube.s({
  'm.type': 'phong',
  'm.texture.image': 'box.jpg',
});
box.add(cube);

var plate=new mono.Cube(40, 20, 50);
plate.s({
  'm.type': 'phong',
  'm.texture.image': 'label.png',
  'm.transparent': true,
});
plate.setPosition(-20, 25, 25);
box.add(plate);
```

贴纸和箱子表面产生了z-fighting，在移动镜头时不断产生随机遮挡，导致闪烁。

通过m.polygonOffset可以一定程度解决这一问题。首先在闪烁的两个面中，选其中的一个，启用m.polygonOffset为true，同时设置m.polygonOffsetFactor和m.polygonOffsetUnits两个参数为一个数值（例如-1）作为强制深度偏移。如果设置为-1，则mono会认为这个面比另外一个面的深度小1，也就是更接近镜头。

改进代码如下：

```
var plate=new mono.Cube(40, 20, 50);
plate.s({
  'm.type': 'phong',
  'm.texture.image': 'label.png',
  'm.transparent': true,
  'front.m.polygonOffset': true,  //启用深度偏移
  'front.m.polygonOffsetFactor': -1,  //偏移量1
  'front.m.polygonOffsetUnits': -1,   //偏移量1
});
```

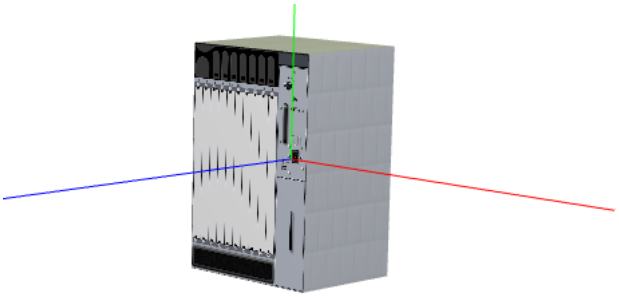
运行效果：

这样，就强制为两个在一起的平面其中一个的深度设置了偏移，让镜头“认为”这个平面的深度在计算结果的基础上再增加offset，让物体不再重叠，避免闪烁。

使用深度偏移会导致渲染性能降低，不宜过多使用。

mono.Cube——立方体

立方体对象（mono.Cube）是一个由6个面组成的立方体，是使用中最常见的3D物体。通过贴图设置，可以用来表示电信中例如设备、机架、仪表等常见物体。立方体的自身坐标原点在其中心位置，如下图：



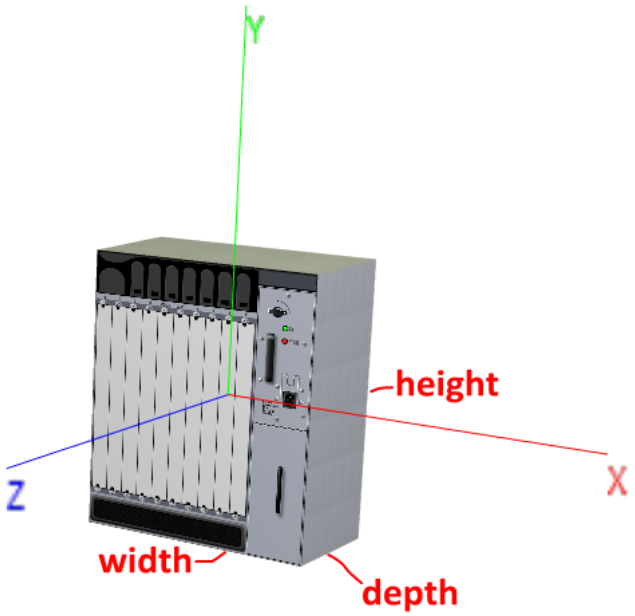
立方体的每一个面都可以单独设置材质图片、是否纹理、纹理重复次数等等。6个面的名字分别用前、后、左、右、上、下进行区分，具体key值如下。

- left：左侧面。例如：设置左侧面贴图，node.setStyle('left.m.texture.image', 'picture.png');
- right：右侧面。例如：设置右侧面贴图，node.setStyle('right.m.texture.image', 'picture.png');
- front：前面（正面）。例如：设置正面贴图，node.setStyle('front.m.texture.image', 'picture.png');
- back：后面（背面）。例如：设置背面贴图，node.setStyle('back.m.texture.image', 'picture.png');
- top：顶面。例如：设置顶面贴图，node.setStyle('top.m.texture.image', 'picture.png');
- bottom：底面。例如：设置底面贴图，node.setStyle('bottom.m.texture.image', 'picture.png');

如果设置style时不指定哪个面前缀，则表示应用于所有面。例如，下面代码将texture-picture.png设置为所有面的贴图，而将左侧单独设置为left-picture.png贴图：

```
node.setStyle('m.texture.image', 'texture-picture.png')
node.setStyle('left.m.texture.image', 'left-picture.png')
```

立方体的尺寸分别为width、height、depth，分别对应x轴的宽度、y轴的高度、z轴的深度。如下图：

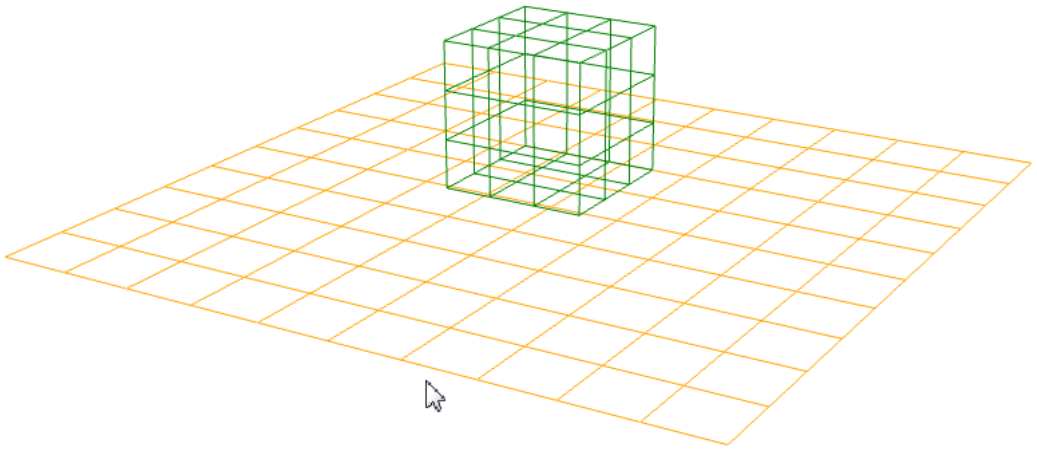


立方体的旋转可以通过setRotation(x, y, z)完成，分别指定在三个轴的旋转角度即可。例如，node.setRotation(Math.PI/2, 0 0)表示将node在x轴旋转90度。也就是：将右手握住x轴，将node沿着手指旋转的方向旋转90度。

```
//构造方法
var node = new mono.Cube(width, height, depth,segmentsW, segmentsH, segmentsD);
//其中参数分别是立方体的宽、高、深、横向切片数量、纵向切片数量和深度切片数量，默认情况下都为1。
```

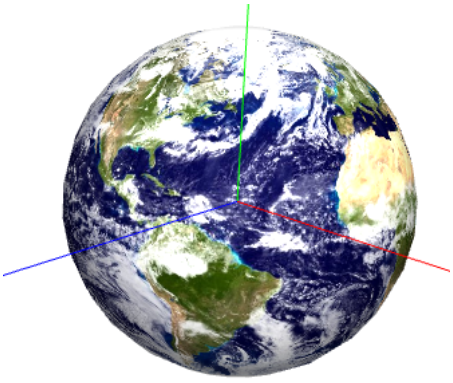
```
//创建一个segments为3的cube
 var cube = new mono.Cube(100,100,100,3,3,3);
            cube.setStyle('m.wireframe',true).setStyle('m.color','green');
            cube.setPosition(0,70,0);
            cube.setSelectable(false);
            box.add(cube);
```

效果如下：



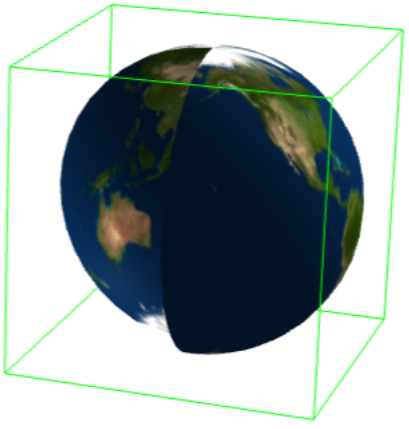
mono.Sphere——球体

球体对象（mono.Sphere）表面是一个曲面，由半径控制其尺寸，其曲面是由一系列纵向+横向均匀切分的小平面拼接而成，圆滑度取决于切分横向和纵向的分片数量，数量越大曲面越圆滑但性能越低。球体的坐标位于其内部中心点。



球体可以在横向和纵向方向上设置开始角度、延伸角度，来创建不完整的球体形状。

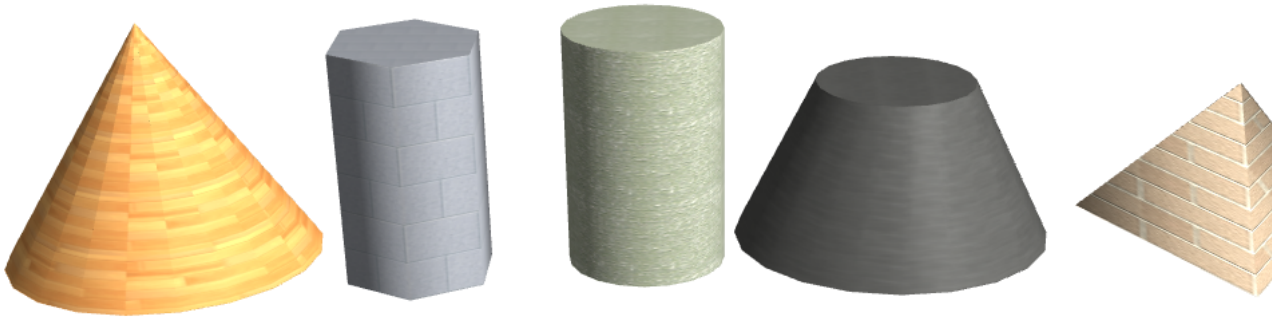
```
//Sphere构造函数
var node = new mono.Sphere(radius, segmentsW, segmentsH, longitudeStart, longitudeLength, latitudeStart, latitudeLength);
//radius - 球体半径
//segmentsW - 横向切片数量，默认值22
//segmentsH - 纵向切片数量，默认值15
//longitudeStart - 经度（纵向）起始角度
//longitudeLength - 经度（纵向）延伸角度
//latitudeStart - 纬度（横向）起始角度
//latitudeLength - 纬度（横向）延伸角度

//下面代码创建了一个不完整的球体
var node=new mono.Sphere(50, 20, 20, 0, Math.PI*1.5, 0, Math.PI);
```



mono.Cylinder——圆柱体

圆柱体对象（mono.Cylinder）表面是由一个纵向直线、横向圆形的曲面组成，由长宽高控制其尺寸，还可设置顶面和底面是否封闭。需要留意的是，圆柱的顶面

和底面的尺寸可以不同。侧面圆滑度由分片数量进行控制，分片数量越大侧面越圆滑但性能越低。圆柱体可以有一些特别用法：圆锥（顶面半径为0）、金字塔形状（顶面半径0、侧面分片为4）、半金字塔形状（顶面半径为底面半径的一半、侧面分片为4）等等。可以通过灵活设置上下面的半径、侧面分片数量，来创建很多变种的圆柱体。以下形状都是各种圆柱的变体：



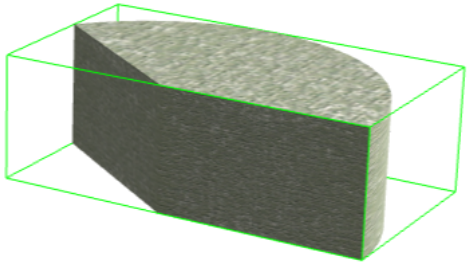
圆柱体的材质分为顶面、底面和侧面，具体key值如下。

- top：顶面。例如：设置顶面贴图，node.setStyle('top.m.texture.image', 'picture.png');
- bottom：底面。例如：设置底面贴图，node.setStyle('bottom.m.texture.image', 'picture.png');
- side：侧面。例如：设置侧面贴图，node.setStyle('side.m.texture.image', 'picture.png');

如果设置style时不指定哪个面前缀，则表示应用于所有面。例如，下面代码将texture-picture.png设置为所有面的贴图，而将侧面单独设置为side-picture.png贴图：

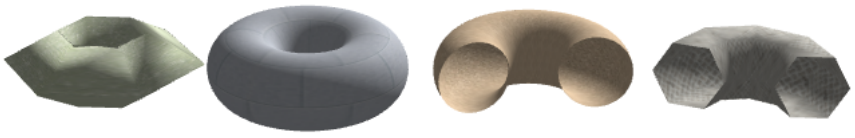
```
node.setStyle('m.texture.image', 'texture-picture.png')
node.setStyle('side.m.texture.image', 'side-picture.png')
```

```
//构造函数
var node = new mono.Sphere(radiusTop, radiusBottom, height, segmentsR, segmentsH, openTop, openBottom,arcLength,arcStart);
//radiusTop - 顶面圆柱
//radiusBottom - 底面圆柱
//height - 圆柱高度
//segmentsR - 侧面（圆弧）分片数量，默认20
//segmentsH - 高度分片数量。一般高度上不需要分片，默认1
//openTop - 顶面是否镂空。true为镂空，false为封闭
//openBottom - 底面是否镂空。true为镂空，false为封闭
//arcLength - 圆柱的圆弧所占长度。例如Math.PI*2则为完整圆柱，Math.PI/2则为1/4圆柱体
//arcStart - 圆柱的圆弧开始角度

//举例：下面代码创建了一个不完整的圆柱体
var node=new mono.Cylinder(50, 50, 30, 20, 1, false, false,Math.PI*0.8,0);
```



mono.Torus——圆环

圆环（mono.Torus）对象是一个类似“手镯”形状的圆环物体。它的切面和形状都是圆形，其圆滑度都可通过切片数量进行控制。此外，圆环还可以通过角度控制其是否封闭。360度的圆环是一个封闭的圆环，而180度的圆环则是一个被切去一半的半个圆环残体。如果将切面切片和环形切片都设置成3，则会形成一个类似三角管体组成的三角形形状体。典型用法：弧形门把手、弯管连接头体、各种圆环形状体等。以下都是各种圆环的变形体。



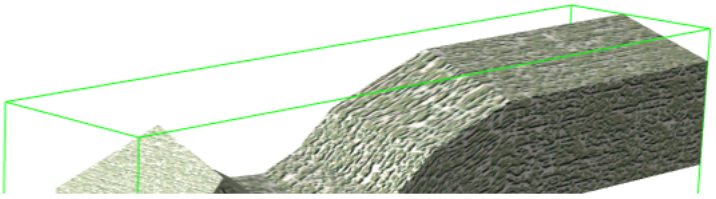
```
//构造函数
var node = new mono.Torus(radius, tube, segmentsR, segmentsT, arc);
//radius - 圆环的半径尺寸
//tube - 横切面圆弧的半径尺寸
//segmentsR - 圆环的分片数量，默认值8
//segmentsT - 横切面圆弧的分片数量，默认值6
//arc - 圆环体所占角度。例如：Math.PI*2则为完整一圈，Math.PI则为半圈
```

mono.PathNode——路径体

路径体（mono.PathNode）这是一种复杂的形状体，由两个任意形状进行控制：切面形状，以及前进走向。最终形状是该切面形状沿着前进走向进行移动而形成的物体。例如，一个圆形切面沿着一个多边形移动，就会形成一个复杂的管线物体。这种形状还可以控制两个端头是否封闭、封闭的形状和尺寸，横切方向是否闭合、闭合角度、闭合样式等。通过控制这些参数，可以创建例如管线、弯管、香肠体、切开的管线等。

下面图中的路径体，路径是一个由直线、曲线等组成的路线，横切面是一个5个分片的圆形。

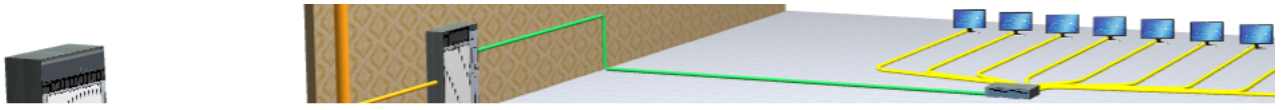
```
var path = new mono.Path();
path.moveTo(0,100,0);
path.lineTo(200,100,0);
path.curveTo(300,0,0, 400,100,0);

var node=new mono.PathNode(path, 10, 50, 5);
```



在创建较长路径的管线时，手工指定管线拐角处的曲线弧度会比较繁琐。mono提供了一个内置的方法：PathNode.adjustPath可以自动将给定的路径进行曲线处理。

```
var path = new mono.Path();
//...
path = mono.PathNode.prototype.adjustPath(path, radius, times);
//其中radius是自动弯角的弧度，times是弯角的分片数量。例如：
path = mono.PathNode.prototype.adjustPath(path,0.5,20);
```

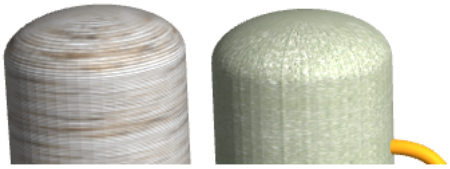
下图展示了使用adjustPath函数的效果：



路径体可以设置两个端头（开始端头、结束端头）是否封闭、平面封闭还是弧形封闭、封闭弧形的尺寸等等。

- 设置端头样式：setStartCap/setEndCap(capStyle); capStyle可以取：'none' 无封闭（开放）、'plain' ：平面封闭、'round'：半球形封闭
- 设置端头封闭面尺寸：setStartCapSize/setEndCapSize(size)；size为1，则封闭面尺寸为横截面半径的1倍，2则为2倍，以此类推

下图中的所有对象都是由设置了端头样式的PathNode对象组成：



下面代码使用了弧形封闭且设置了封闭面的尺寸大小：

```
var path = new mono.Path();
path.moveTo(0,100,0);
path.lineTo(200,100,0);
path.curveTo(300,0,0, 400,100,0);
path.lineTo(450,100,0);

var node=new mono.PathNode(path, 100, 50, 100,'round', 'round');
node.setStartCapSize(5);
node.setEndCapSize(1);
node.setStyle('m.texture.image','../images/wall01_inner_3d.png').setStyle('m.type','phong').setStyle('m.side',mono.DoubleSide).setSt
```

路径体可以沿着径向，设置圆弧的完整度。通过构造函数指定arc、arcStart、cutSurface可以设置完整度的长度、角度、样式。

- setArc：横切面圆弧完整度弧度。例如2*Math.PI是完整圆形。
- setArcStart：设置横切面圆弧开始角度。
- setCutSurface：设置不完整弧度切割面的样式。'none'为镂空、'plain'为直连平面、'center'为直中心线拐角切面。

下图演示了不完整弧度、切面样式的例子。用一个路径构造了两个管子，小尺寸管保持完整，大尺寸管切掉1/4：

```
var path = new mono.Path();
path.moveTo(0,100,0);
path.lineTo(200,100,0);
path.curveTo(300,0,0, 400,100,0);
path.lineTo(450,100,0);

var node=new mono.PathNode(path, 100, 50, 100,'round', 'round', null, Math.PI*1.5, Math.PI/2, 'center');
node.setStartCapSize(2);
node.setEndCapSize(1);
node.setStyle('m.texture.image','../images/wall01_inner_3d.png').setStyle('m.type','phong').setStyle('m.side',mono.DoubleSide).setSt
box.add(node);

node=new mono.PathNode(path, 100, 30, 100,'round', 'round');
node.setStartCapSize(2);
node.setEndCapSize(1);
node.setStyle('m.texture.image','../images/water-texture-1.png').setStyle('m.type','phong').setStyle('m.side',mono.DoubleSide).setSt
box.add(node);
```

下图中则是展示了cutSurface分别使用'plain'和'center'时的情形：

路径体还有一种特别的用法，就是不但可以指定路径走向，还可以指定横切面的任意形状，这个形状也可以通过一个任意路径的走向来定义。这样，就可以创建各种横截面形状的“管子”。下图展示了这一用法：

```
var path = new mono.Path();
path.moveTo(0,100,0);
path.lineTo(200,100,0);
path.curveTo(300,0,0, 400,100,0);
path.lineTo(450,100,0);

var shape = new mono.Path();
shape.moveTo(0,0,0);
shape.curveTo(50,50,0, 100,50,0);
shape.lineTo(100,0,0);
shape.curveTo(50,0,0, 0,-50,0);
shape.lineTo(0, 0,0,0);

var node=new mono.PathNode(path, 100, 50, 100,'plain', 'plain', shape);
node.setStyle('m.texture.image','../images/metal02.png').setStyle('m.type','phong').setStyle('m.side',mono.DoubleSide).setStyle('m.r
box.add(node);
```

其中，横截面形状、路径走向如下图所示：

//构造函数

```
var node = new mono.PathNode(path, segments, radius, segmentsR, startCap, endCap, shape, arc, arcStart, cutSurface);
```

//path – 路径走向

//segments – 路径方向分片数量，默认64。**注意：分片数量沿路径方向等距分割，无论路径形状如何。因此，此数值过小可能会影响路径方向形状的圆滑度。过大则会降低效率。开发中应仔细调整合适数值**

//radius – 横截面圆形半径

//segmentsR – 横截面圆形分片数量，默认值8

//startCap – 起始端头样式，'none'镂空、'plain'平面封闭、'round'弧形面封闭

//endCap – 结束端头样式，'none'镂空、'plain'平面封闭、'round'弧形面封闭

//shape – 横截面形状。如不设置则横截面为圆形。默认值空

//arc – 横截面所占弧度。默认为Math.PI*2，完整圆形

//arcStart – 横截面圆弧开始角度，默认0度开始

//cutSurface – 不完整弧度切割面的样式。'none'为镂空、'plain'为直连平面、'center'为直中心线拐角切面

mono.TextNode——文字

文字（mono.TextNode）物体是一串3D化的文字字符。文字在3D中并没有直接的支持方式，所以和其他复杂形状一样，需要给出其具体的形状然后进行切片、分片来模拟。由于3D中没有直接的字体数据，需要额外提供字体的具体形状。在mono中没有内置任何字体信息，需要开发者额外去创建并引入到程序中。在MONO Design编辑器中，提供了几种英文字体形状信息，存储在类似"***_regular.typeface.js"的js文件中。例如，"helvetiker_regular.typeface.js"文件中存储了helvetiker字体的正常体信息，"helvetiker_bold.typeface.js"文件中存储了helvetiker字体的粗体信息，等等。如需要更多的字体，可以到网站http://typeface.neocracy.org/在线提交、生成和下载。一般一个英文字体对应的js文件大约在数百kb左右。由于中文字体形状复杂、字符数量巨大，一般不建议使用中文字符。典型用法：各种文字标识、设备标签。

MONO Design编辑器中提供的字体资源有：

- gentilis_bold.typeface.js
- gentilis_regular.typeface.js
- helvetiker_bold.typeface.js
- helvetiker_regular.typeface.js
- kaushan_script_regular.typeface.js
- optimer_bold.typeface.js
- optimer_regular.typeface.js

其中regular为正常体、bold为粗体。开发者可以将这些js引入程序中直接使用。下图显示了一个具有纹理、染色效果的文字字符串：

```
//构造函数
var node = new mono.TextNode(text, size,height,font,weight);
//text - 文字字符串。请注意要使用字体中包含的字符。例如使用一个英文字体就不能输入中文字符
//size - 文字的尺寸。数值越大，文字越高大
//height - 文字的厚度（深度）。请注意该参数并不影响文字大小，而是影响文字的厚度
//font - 文字字体名称。例如：引入的字体文件是helvetiker_bold.typeface.js，则font设置为'helvetiker'
//weight - 粗体或正常体。数值'normal'为正常体,'bold'为正常体
```

下面代码创建了一个字符串，并设置纹理、显示：

```
var node=new mono.TextNode('TWaver Mono Design', 10,1,'optimer','bold');
node.setStyle('m.texture.image','../images/metal02.png').setStyle('m.type','phong').setStyle('m.repeat',new mono.Vec2(3,3));
box.add(node);
```

使用setFont改变textNode的字体。

```
node.setFont('gentilis');
```

mono.ShapeNode——形状块

形状块（mono.ShapeNode）是一个有厚度的、任意形状的3D物体。典型应用：地板、地图块等。下图是用形状块显示的美国Ohio州的3D轮廓图：

```
//构造函数
var node = new mono.ShapeNode(shapes, curveSegments, amount, horizontal, repeat);
//shapes - 块图形状
//curveSegments - 轮廓边缘分片数量
//amount - 块图的厚度（深度）
//vertical - 形状是水平方向还是垂直方向。true为垂直，false为水平。默认true，垂直
//repeat - 重复纹理块的尺寸大小。越大，纹理图片重复次数越少
```

上面Ohio州轮廓图，可以用下面代码创建：

```
var path=new mono.Path();

path.moveTo(734.98, 266.32);
path.lineTo(718.95, 262.25);
path.lineTo(714.20, 254.88);
path.lineTo(707.70, 255.09);
path.lineTo(708.08, 194.67);
path.lineTo(731.26, 194.02);
path.lineTo(743.43, 198.46);
path.lineTo(738.57, 200.39);
path.lineTo(747.81, 201.93);
path.lineTo(762.18, 199.56);
path.lineTo(783.77, 187.79);
path.lineTo(783.74, 219.64);
path.lineTo(781.16, 220.93);
path.lineTo(782.28, 227.36);
path.lineTo(777.38, 243.14);
path.lineTo(763.05, 251.49);
path.lineTo(761.76, 259.03);
path.lineTo(759.44, 260.32);
path.lineTo(756.78, 257.11);
path.lineTo(751.71, 270.12);
path.lineTo(747.34, 270.98);
path.lineTo(741.76, 263.29);
path.lineTo(734.98, 266.32);

var node=new mono.ShapeNode(path, 100, 10);
node.setPositionX(-700);
node.setPositionY(-150);
node.setStyle('m.texture.image','../images/grass.png').setStyle('m.type','phong');
```

mono.PathCube——路径方块

路径方块物体（mono.PathCube）是一个矩形沿着某个路径移动形成的3D物体。它的纵切面是一个矩形，并沿着一个路径方向进行建模。它最常见的用途就是各种**房间的墙体。**下图是一个典型的路径方块：

```
//构造函数
var node = new mono.PathCube(path, width, height, curveSegements, repeat);
//path - 走向路径
//width - 横截面立方体宽度
//height - 横截面立方体高度
//curveSegements - 路径方向分片数量
//repeat - 纹理贴图多少尺寸重复一次。例如10则贴图会每隔10重复一次。越大则纹理单位贴图越大。
```

例如，要创建一个上图形状的墙体，并增加一个墙体加强的地基，可以如此写代码：

```
var path = new mono.Path();
path.moveTo(0, 0,0);
path.lineTo(1000, 0, 0);
path.lineTo(1000, 500, 0);
path.lineTo(500, 500, 0);
path.lineTo(500, 1000, 0);
path.lineTo(0, 1000, 0);
path.lineTo(0, 700, 0);
path.curveTo(-400,500,0,0,300,0);
path.lineTo(0,250,0);

var wall = new mono.PathCube(path,20,300,32,40);
wall.setStyle('m.texture.image', '../images/wall02_3d.png');
wall.setStyle('inside.m.texture.image', '../images/wall02_3d.png');
wall.setStyle('top.m.texture.image', '../images/wall02_3d.png');
wall.setStyle('bottom.m.texture.image', '../images/wall02_3d.png');
wall.setStyle('aside.m.texture.image', '../images/metal02.png');
wall.setStyle('zside.m.texture.image', '../images/metal02.png');
box.add(wall);

wall = new mono.PathCube(path,50,100,32,40);
wall.setStyle('m.texture.image', '../images/wall04_3d.png');
wall.setStyle('inside.m.texture.image', '../images/wall01_inner_3d.png');
wall.setStyle('top.m.texture.image', '../images/metal08.png');
wall.setStyle('bottom.m.texture.image', '../images/metal08.png');
wall.setStyle('aside.m.texture.image', '../images/metal02.png');
wall.setStyle('zside.m.texture.image', '../images/metal02.png');
box.add(wall);
```

运行效果如下：

还可以使用png局部透明图片，来模拟现实世界中的例如篱笆等特殊墙体。将墙体换成有透明区域的图片后，运行上述例子，效果如下：

mono.Particle——粒子系统

粒子系统（mono.Particle）是一个非常特别的3D物体。和其他复杂形状的3D物体不同，它没有很多具体的面，而只是由一系列内部的顶点组成，每个顶点显示给定的贴图。当它处于静止状态时，它显示一堆空间离散分布的"颗粒物"；当动画来控制顶点的数量和位置，可以实现一种复杂运动的粒子效果。它非常适合模拟例如烟雾、火焰、喷水等效果。

```
//构造函数
var smoke = new mono.Particle(vertices,colors, material);
//vertices - 顶点坐标数组
//colors - 颜色数组，每个顶点的颜色
//material - 顶点贴图。所有顶点使用相同贴图

//一般而言，为了让粒子系统更逼真、高效，还需要仔细设置以下参数：
smoke.sortParticles = false; //忽略粒子系统中顶点重新排序。重要！
smoke.setStyle('m.depthTest',false); //忽略深度检测。重要！
smoke.setStyle('m.transparent',true); //启用材质透明。材质图片中的透明区域（例如png图片）会有透明、半透明效果。重要！
smoke.setStyle('m.opacity',0.1); //设置材质的不透明度。强行让材质的不透明度为0.1（也就是90%的透明度）。注意：此属性和材质图片本身的透明度无关。它和m.t

smoke.setStyle('m.color',0xffffFF); //材质染色，可以通过该颜色对烟雾图片进行染色
smoke.setStyle('m.size',40); //材质的贴图尺寸
smoke.setStyle('m.texture.image','../images/smoke2.png');//设置材质图片
```

下面代码创建了一个静态的、200个顶点的粒子系统：

```
var sphereRadius = 80;
var particleCount = 200;
var smoke = new mono.Particle();
smoke.setClient("sphereRadius", sphereRadius);

for (var p = 0; p &lt; particleCount; p++) {
  var radius = sphereRadius;
  var angle = Math.random() * (Math.PI * 2);
  var pX = Math.sin(angle) * radius, pY = Math.random() * radius, pZ = Math.random() * radius, particle = new mono.Vec3(pX, pY, pZ);
  particle.velocity = new mono.Vec3(Math.random()*2, Math.random()*2, 0);
  smoke.vertices.push(particle);
}

smoke.sortParticles = false;
smoke.setStyle('m.color',0xffffFF).setStyle('m.size',40).setStyle('m.transparent',true).setStyle('m.opacity',0.1).setStyle('m.textur
smoke.setStyle('m.depthTest',false).setStyle('m.depthWrite',false);
smoke.setStyle('m.color', 'red');
```

下面代码进一步添加了动画效果，通过循环调用network的render进行不断刷新，每次刷新再重新计算粒子系统内部顶点的位置，即可达到烟雾飘动的动画效果。可以用于显示机房监控烟雾、温度等信息。

```
var smoke = new mono.Particle();
var network;

function load(){
  var box = new mono.DataBox();
  var camera = new mono.PerspectiveCamera(30, 1.5, 0.1, 10000);
  camera.setPosition(50,200,500);

  network= new mono.Network3D(box, camera, myCanvas);
  var interaction = new mono.DefaultInteraction(network);
  interaction.zoomSpeed = 30;
  network.setInteractions([new mono.SelectionInteraction(network), interaction]);
  mono.Utils.autoAdjustNetworkBounds(network,document.documentElement,'clientWidth','clientHeight');

  var pointLight = new mono.PointLight(0xFFFFFF,1);
  pointLight.setPosition(10000,10000,10000);
  box.add(pointLight);
  box.add(new mono.AmbientLight(0x888888));

  var sphereRadius = 80;
  var particleCount = 200;

  smoke.setClient("sphereRadius", sphereRadius);

  for (var p = 0; p &lt; particleCount; p++) {     var radius = sphereRadius;     var angle = Math.random() * (Math.PI * 2);     var
      particle.y = 0;
      var radius = sphereRadius;
      var angle = Math.random() * (Math.PI * 2);
      particle.x = Math.cos(angle) * radius;
    }
    //continue;
    var t = Date.now() / 1000 % 3;
    particle.x += Math.cos(t * particle.velocity.x) * 5;
    particle.y += Math.sin(t * particle.velocity.y) * 2;
  }
  network.render();
  setTimeout(changeSmoke,20);
}
```

效果如下图：

mono.CSG——运算体

运算体（mono.CSG）是由多个3D物体进行组合运算得出的物体。例如立方体A合并球体B、球体A减掉圆柱体B。目前mono支持的运算有union（合并，A加上B）、substract（减除，A减掉B的部分）、intersect（交叉、A和B的公共部分）。下图显示了两个物体合并和减除的情况：

构造一个运算体物体，需要给定一个具体的3D物体，然后运算体物体再进行相互运算。对于运算的结果，**必须要调用toMesh()函数进行处理**，才能加入DataBox进行显示。

```
//创建一个立方体
var cube = new mono.Cube(100,15,100);
cube.setStyle('m.texture.image', '../images/default_texture.png');

//创建一个圆柱体
var cylinder = new mono.Cylinder(30,30,50);
cylinder.setStyle('m.texture.image', '../images/floor.png');

var csg1=new mono.CSG(cube); //立方体对应的运算体对象
var csg2=new mono.CSG(cylinder);  //圆柱体对应的运算体对象
var csg=csg1.substract(csg2).toMesh();  //立方体减去圆柱体，生成残留对象，并进行mesh处理，返回运算结果3D对象

box.add(csg);
```

运行上述代码，显示效果：

运算体物体在mesh前，可以持续多次进行运算，最后再进行mesh加入DataBox，以形成更复杂的运算体。例如：

```
var cube = new mono.Cube(100,15,100);
cube.setStyle('m.texture.image', '../images/default_texture.png');

var cylinder = new mono.Cylinder(30,30,50);
cylinder.setStyle('m.texture.image', '../images/floor.png');

var csg1=new mono.CSG(cube);
var csg2=new mono.CSG(cylinder);
var csg=csg1.substract(csg2);

for(var i=0;i&lt;4;i++){
  cylinder = new mono.Cylinder(20,20,40);
  cylinder.setPosition(35-70* (i%2),0, 50-70* (i/2));
  cylinder.setStyle('m.texture.image', '../images/wall04_3d.png');
  cylinder.setStyle('m.texture.repeat', new mono.Vec2(4,3));
  csg=csg.substract(new mono.CSG(cylinder));
}

csg=csg.toMesh();
box.add(csg);
```

运行结果如下：

mono.Billboard——公告牌

公告牌对象（mono.Billboard）是一种特殊的3D物体，它只有一个图片组成，而且这张图片会永远朝前面向镜头（也就是我们用户的眼睛），无论场景如何变化。在游戏软件场景中，经常会有这样的场景：一个移动的物体（人、机器等）会在上方显示一个图片，图片上显示了文字、状态信息等，这个图片会永远朝向用户的眼睛，而无论物体如何移动。公告牌的这一特性使得它非常合适制作3D场景中的各种信息展示、文字说明、提示警告等。其典型用法是用于显示设备告警、设备状态信息，也可以显示花草树木等装饰物。下图是一些公告牌的应用场景：

一般公告牌会设置为某设备的自对象，也就是通过billboard.setParent(node)方法为其设置父节点。这样公告牌会跟随父对象一起移动。

```
//创建一个设备节点
var node = new mono.Cube(20,20,20);
node.setStyle('m.texture.image','../images/bbb.png');
box.add(node);

//构造一个公告牌对象
var billboard=new mono.Billboard();

billboard.setStyle('m.texture.image','../images/billboard5.png'); //设置公告牌图片
billboard.setScale(20,40,1);  //设置图片的宽高放大比例。第三个z轴参数没有意义，传入1即可
billboard.setPosition(0,0,0);  //设置公告牌位置

billboard.setStyle('m.alignment',mono.BillboardAlignment.bottomCenter); //设置公告牌在附属节点上的对齐位置。附属节点是公告牌的父节点，会随其移动
billboard.setStyle('m.transparent',true); //设置图片透明模式。此时图片的透明区域会显示为镂空

billboard.setParent(node); //设置公告牌的父节点（跟随节点）
```

显示效果如下：

另外我们也可以设置Billboard始终垂直效果：

```
billboard.setStyle('m.vertical',true);//设置billboard为垂直效果
```

显示效果如下,左图为未设置垂直效果,右图为始终垂直效果：

mono.LatheNode——车削对象

车削对象（mono.LatheNode）是一个路径体绕某轴线旋转一圈（或一定角度）形成的封闭（或开放）的3D物体。下图中，红色的路径体沿着黄色的轴旋转一周，就形成了一个车削对象。

```
//创建车削对象
var node = new mono.LatheNode(path, segmentsH, segmentsR,arc, startClosed, endClosed);
//path - 车削旋转的形状
//segmentsH - 轴向的分片次数。如路径体在轴的方向比较复杂，则需要多分配分片数量，才能得到更平滑的效果。默认64
//segmentsR - 径向的分片次数。车削对象的径向是一个完整的圆（或圆的一部分），要得到光滑的边缘，需要设置较多径向分片。默认20
//arc - 径向旋转角度。如径向旋转角度不足一周（2*Math.PI），则会形成一个残缺的车削对象
//startClosed - 起始面是否封闭。true为封闭，false为镂空
//endClosed - 截止面是否封闭。true为封闭，false为镂空
```

下面代码创建了一个简单的路径体和车削对象：

```
var path = new mono.Path();
//以下z点均会被忽略
path.moveTo(-50, 0, 0);
path.lineTo(-10, 10, 0);
path.lineTo(-10, 40, 0);
path.lineTo(-70, 60, 0);
path.lineTo(-80, 80, 0);

var node = new mono.LatheNode(path, 50, 20, Math.PI*2, true, false);
node.setStyle('m.texture.image','../images/bbb.png').setStyle('m.type','phong').setStyle('m.side',mono.DoubleSide).setStyle('m.repea
box.add(node);
```

效果如下：

如适当设置角度，并进行嵌套，可以很容易做出容器、液体的效果。下图是两个LatheNode组成的物体，内部对象采用水状贴图，模拟了液体效果：

可设置旋转角度，形成残缺旋转体：

```
var node = new mono.LatheNode(path, 50, 20, Math.PI*1.5, true, true);
```

mono.Plane——平面对象

平面（mono.Plane）是一个非常简单的平面对象。它可设置宽、高、贴图，表示一个空间的水平平面或垂直平面。平面对象只能创建一个矩形平面，不能创建任意形状的平面，也没有厚度的概念，使用有一定局限性。如需要创建有厚度、边缘复杂的的平面，可使用mono.ShapeNode。

```
//创建平面对象
var node = new mono.Plane(width, height);
//width - 平面宽度
//height - 平面高度

//创建一个简单的平面
var node = new mono.Plane(200,200,1,1,true);
node.setStyle('m.texture.image','../images/floor.png').setStyle('m.type','phong').setStyle('m.side',mono.DoubleSide).setStyle('m.repe
box.add(node);
```

效果如下：

```
//创建wireframe样式的平面对象
 var floor = new mono.Plane(500,500,10,10);
            floor.setStyle('m.wireframe',true).setStyle('m.color','orange').setStyle('m.side', mono.DoubleSide);
            floor.setRotation(Math.PI/2,Math.PI,Math.PI/2);
            floor.setSelectable(false);
            box.add(floor);
```

效果如下：

```
//也可以使用另外一种样式设置方法
 var plane = new mono.Plane(200,200,20,20);
            plane.s({
                'm.color' : '#FF6666',
                'm.wireframe' : true
            });
            plane.setSelectable(false);
            plane.setRotation(Math.PI/2,Math.PI,Math.PI/2);
            box.add(plane);
```

效果如下：

mono.ComboNode —— 组合体

组合体和运算体类似，它和由多个对象根据一定的运算符，运算出来的一个新的物体，它比mono.CSG更加方便，不需要人为去相加、相减，而是通过给定的运

算符，mono内部就处理了这几个物体的运算，比如：一个圆形和立方体运算后可以得到下面效果：

代码如下：

```
var cube1 = createCube(120,50,50);
cube1.setPosition(0,0,0);

var cube2 = createCylinder(50);
cube2.setPosition(0,0,0);

var combo = new mono.ComboNode([cube2,cube1],['-']);
box.add(combo);
```

创建ComboNode需要传入两个参数：combos，operators

combos：需要组合的原型对象

operators：组合对象之间的运算符，支持'+'，'-'，'^'（相加、相减、相交）

有了这样的组合体对象，我们就可以组合出各种形状的物体，下面是组合成一个灭火器的样例：

mono.Light——光源对象

mono.Light是所有灯的基类，在3D场景中，光照是非常重要的元素，它模拟了现实世界，在3D场景中的物体可以反射出光照效果。类的定义如下：

```
Light = function(color) {};
```

Light类中包含的方法如下：

- setCastShadow(castShadow):设置是否需要显示灯光的阴影
- setColor(color):设置光照的颜色值
- getColor():获取光照的颜色值
- setAmbient(ambient):设置环境光照的颜色值
- getAmbient(): 获取环境光照的颜色值
- setDiffuse(diffuse):设置散射光的颜色值
- getDiffuse():获取散射光的颜色值
- setSpecular(specular):设置镜面光的颜色值
- getSpecular(): 获取镜面光的颜色值

通过继承Light，主要实现了四种光源，分别为：(环境光源)AmbientLight、(点光源)PointLight、(聚光源)SpotLight、(方向光源)DirectionalLight。类的定义如下：

```
/**
 * 环境光，环境光是那些在环境中进行了充分的散射，无法辨认其方向的光。
```

```
 * @param {Number} hex 一个包含RGB的十六进制数组
 */
AmbientLight = function(hex) {};

/**
 * 点光源。在3D场景中创建一个指定颜色的点光源元素
 * @param {Number} hex 一个包含RGB的十六进制数组
 * @param {Number} intensity 光照强度值，如果为空，默认设置为1
 * @param {Number} distance 光照衰减的距离值
 */
PointLight = function ( hex, intensity, distance ) {};

/**
 * 聚光源。在3D场景中创建一个指定颜色的聚光灯元素
 * @param {Number} hex 一个包含RGB的十六进制数组
 * @param {Number} intensity 光照强度值，如果为空，默认设置为1
 * @param {Number} distance 光照衰减的距离值
 * @param {Number} angle 最大可扩散的光照弧度
 * @param {} exponent
 */
mono.SpotLight = function ( hex, intensity, distance, angle, exponent ) {};

/**
 * 方向光源。在3D场景中创建一个指定颜色的方向光照
 * @param {Number} hex 一个包含RGB的十六进制数组
 * @param {Number} intensity 光照强度值，如果为空，默认设置为1
 */
mono.DirectionalLight = function ( hex, intensity ) {};
```

光源中常用的方法如下：

- PointLight.setIntensity(intensity):设置光照的强度
- PointLight.getIntensity():获取光照的强度值
- PointLight.setDistance(distance):设置光照的衰减距离
- PointLight.getDistance():获取光照的衰减距离
- PointLight.setDistance(distance):设置光照衰减的距离值
- PointLight.getDistance():获取光照衰减的距离值
- SpotLight.setAngle(angle):设置最大可扩散的光照弧度
- SpotLight.getAngle():获取最大可扩散的光照弧度
- AmbientLight.clone():将一个点光源对象克隆出新的对象

下面代码说明了如何使用光源：

```
var light=new mono.PointLight(0x00ff00,0.5);
    light.setPosition(100,300,600);
    box.add(light);
    box.add(new mono.AmbientLight(0xff00ff));
```

下面例子分别展示了未添加光源、添加AmbientLight 、添加PointLight以及添加SpotLight的效果。

```
LightDemo<script src="mono.js" type="mce-text/javascript"></script><script src="twaver.js" type="mce-text/javascript"></script><scrip
var network, interaction;        function load(){            var box = new TGL.DataBox();            var camera = new TGL.Perspect
// ]]></script>
```

为添加任何灯光效果如下：

为场景添加AmbientLight效果如下：

为场景添加PointLight

```
 //添加点光源
            var light=new TGL.PointLight(0x33ff00,0.5);
            light.setPosition(100,300,600);
            box.add(light);
```

效果如下：

为场景添加SpotLight

```
<script src="../libs/t.js"></script><script>// <![CDATA[
var network, interaction;         function load(){            var box = new mono.DataBox();            var camera = new mono.Perspe
// ]]></script>
```

效果如下：

mono.Terrain——地形对象

地形对象（mono.Terrain）有两个面组成，两个面分别为layer0、layer1，例如在海岛的场景中，layer0呈现水面，layer1呈现地形（layer0和layer1不可互换，即layer1表示地形），如下图 T-1

地形对象的每个面都可以单独设置材质素材，纹理重复次数，是否透明，透明度等等

例如：两个面分别设置纹理重复次数

```
terrain.setStyle('layer1.m.texture.repeat', new mono.Vec2(30, 30));
terrain.setStyle('layer0.m.texture.repeat', new mono.Vec2(5, 5));
```

如果设置style时不指定哪个面前缀， 且值不为数组，则表示应用于所有面

例如：设置所有面都不透明

```
terrain.setStyle('m.transparent', false);
```

如果设置style时不指定哪个面前缀，且值为数组，则数组元素分别对应一个面

例如：设置layer0和layer1贴图分别为water.jpg和grass.jpg

```
terrain.setStyle('m.texture.image', ['./images/water.jpg', './images/grass.jpg']);
```

控制面是否显示，例如取消水面的显示

```
terrain.setStyle('layer0.m.visible',false);
```

```
//mono.Terrain构造函数
var terrain = new mono.Terrain(width ,depth, segmentsW , segmentsD , heightUnit, heightMap , baseLayerHeigh);
//width：宽度，X轴方向
//depth：深度，Z轴方向
//segmentsW：横向的切片数量
//segmentsW：横向的切片数量
```

```
//heightUnit：layer1上每个点的高度，取决于heightMap图片中对应像素点的颜色（同比例对应）rgb的平均值，所以计算所得值，最大为255（r,g,b取值范围0-255），he
//heightMap：layer1上每个点的高度，解析于heightMap图片中同比例像素点颜色rgb的平均值，最大值为255。见图-2，图T-1中高度依据图T-2,Layer1上点的高度 = hei
//baseLayerHeigh：layer0的高度
```

图T-2

在现实中有不同的地质环境，如草地，森林，高山，黄土高原，湿地，沙滩等等，因此在地形对象中会有使用多张贴图的需求，多张贴图的设置方法，同时地形对象的贴图类型必须为terrain

```
terrain.setStyle('layer1.m.type', 'terrain');
terrain.setStyle('layer1.m.texture.image', './images/beach.jpg');
terrain.setStyle('layer1.m.texture1.image', './images/grass.jpg');
terrain.setStyle('layer1.m.texture2.image', './images/rock.jpg');
```

目前最多支持3张贴图，这3张贴图的显示规则一：

设置layer1.m.blendRange属性

```
terrain.setStyle('layer1.m.blendRange', new mono.Vec2(0.28,0.5));
```

根据Y值权重所在范围取对应贴图，Y值权重 = Y值/Y值最大值，权重在小于0.28，贴图为beach.jpg；权重大于0.5，贴图为rock.jpg，

显示效果见图-3

显示规则二：

设置layer1.m.textureb.image属性(见图-5)

```
terrain.setStyle('layer1.m.textureb.image', './images/terrain_splats.png');
```

列举个数据算式说明

a 代表layer1上像素点的颜色值

b 代表与a同比例在layer1.m.texture.image上的点的颜色值

c 代表与a同比例在layer1.m.texture.image1上的点的颜色值

d 代表与a同比例在layer1.m.texture.image2上的点的颜色值

e 代表与a同比例在layer1.m.textureb.image上的点的颜色值

e.r、e.g、e.b分别代表e的rgb值

a = (b*e.r + c*e.g + d*e.b)/(e.r + e.g + e.b)

效果见图-4

图-5

mono.Line——线条对象

线条对象（mono.Line）是即可以充当连接各物体的桥梁，也可以充分展现线条的艺术美。

```
//创建Rectangle矩形形状
var line = mono.Line.createRectangle(40,40,500);
line.setType(TGL.LinePieces);
line.setPositionY(-30);
line.setPositionX(20);
line.setStyle('m.color','red');
box.add(line);
```

效果展示如下：

```
//创建Helix螺旋状线条
var line = mono.Line.createHelix(-10,25,20,10,300);
line.setPositionY(30);
line.setPositionX(20);
line.setStyle('m.color','red');
box.add(line);
```

```
//创建Ellipse椭圆线条
 var line = mono.Line.createEllipse(12,8,100,0,Math.PI/2,true);
 line.setType(TGL.LinePieces);
 line.setPositionY(30);
 line.setPositionX(20);
 line.setStyle('m.color','red');
 box.add(line);
```

使用镜头

twaver的2d产品中并没有镜头的概念，但镜头在3D中是一个重要的概念，需要我们有所了解。镜头代表了我们的眼睛，它决定了3D场景中我们看到的场景。在mono的3D场景中，我们的眼睛放在哪里、看向哪里，决定了3D画布绘制出来的结果。

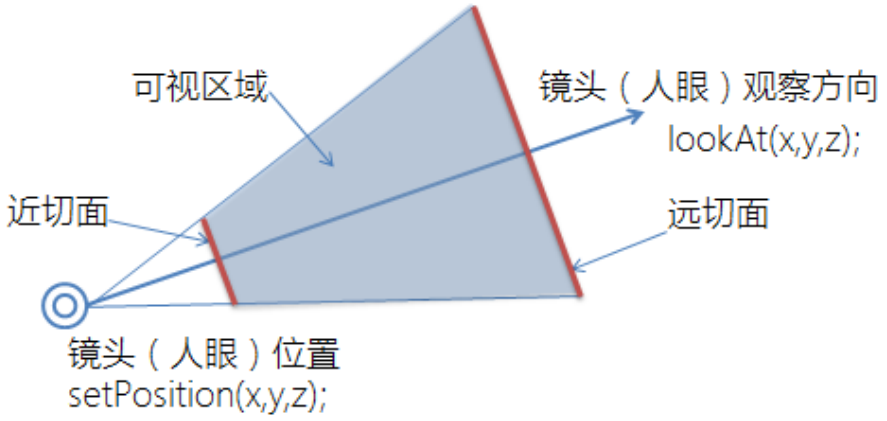
mono中，每一个network画布都内部创建了一个镜头对象，它是mono.PerspectiveCamera的一个实例，一般我们不需要设置它，而是通过函数network.getCamera()得到它进行使用。

```
1 //获得network镜头对象
2 var camera = network.getCamera();
```

拿到camera对象后，就可以对它进行各种参数设定和使用了。

镜头的位置和焦点

使用镜头，最基础最常用的两个参数是镜头的所在位置（Position）和镜头看向空间哪一个点（Target）。镜头如同我们人的眼睛：我们人所处的位置，就是镜头的Position；我们人眼睛看向的那个点，就是镜头的Target，也就是朝向。严格来说，Target是一个从镜头发出到这个Target点的一条射线，也就是一个向量，设置Target为向量上任意一个点，都可以看到相同的效果，对渲染出来的结果没有不同。不过，Target的不同，对第三人称视角下的交互会产生影响。



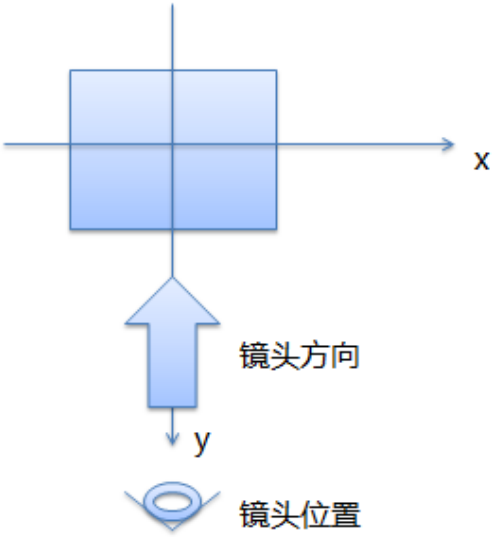
每一个network都有一个内置的camera对象，通过network.getCamera()函数可以拿到camera。

下面的代码创建了一个立方体，由于未设置立方体的位置因此它的中心点是原点（0，0，0）。camera也未设置Target因此镜头的方向也是原点。我们把镜头设置到正方体的正前方（0，0，500）位置，从正面去看这个立方体：

镜头设置Position的方法是setPosition，设置Target的方法是lookat或lookAt或look均可。

```
1 var rack= new mono.Cube(100, 100, 100);
2 rack.s({
3   'm.type': 'phong',
4   'm.texture.image': 'box.jpg',
```

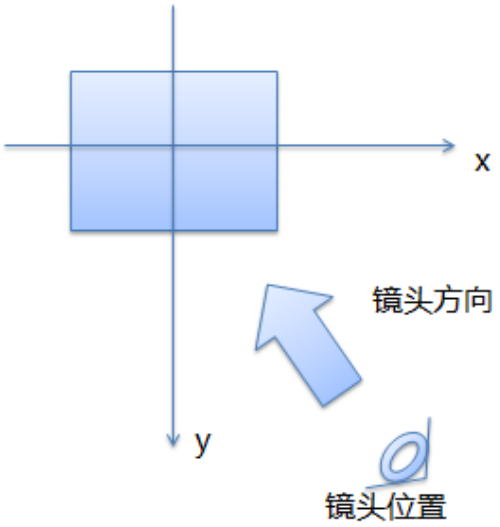
```
});
box.add(rack);

var camera=network.getCamera();
camera.setPosition(0, 0, 500);
```



运行结果

看到的是一个完全正面的立方体。下面的代码把镜头位置增大x和y，也就是拉到偏右、偏上方的位置，再看原点的立方体，效果如下：

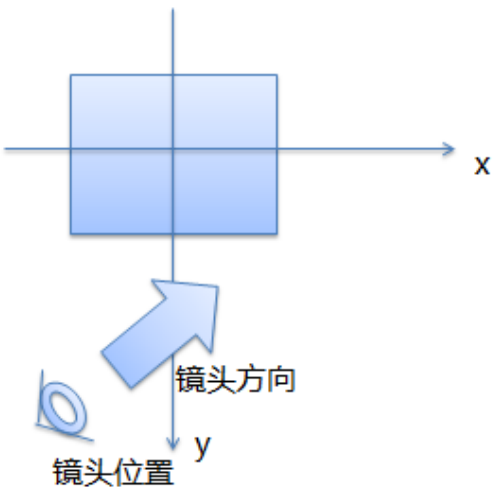
```
camera.setPosition(200, 100, 500);
```



运行结果

如果把镜头位置放在左下方，朝向物体的右上方，则看到的景象会变成：

```
camera.setPosition(-50, -50, 200);
camera.lookat(50, 50, 50);
```



运行结果

可见，镜头的位置、镜头的朝向，决定了我们能够看到的场景。

> **注意：**
> mono中镜头camera.lookAt(x, y, z)、camera.lookat(x, y, z)、camera.look(x, y, z)三个函数完全通用，没有区别，方便开发者记忆和使用。

镜头快捷键

mono默认提供了对镜头控制的快捷键。

- A键：Position和Target同时向左偏，场景向右移动；
- D键：Position和Target同时向右偏，场景向左移动；
- W键：Position向Target位置靠近，场景变近、变大；
- S键：Position远离Target位置，场景变远、变小；
- R键：Position和Target同时向上偏，场景向下移动；
- F键：Position和Target同时向下偏，场景向上移动；

镜头自动巡航

自动巡航是一种动画，主要是自动控制镜头的position和target，沿着预定义好的任意轨迹进行动画移动和观察，产生对场景“自动巡视”的效果。自动巡航可以用来做场景的自动巡视、设备巡检、故障定位等应用。

巡航的本质是通过程序自动控制camera的position和target。例如，target不变，position围绕target旋转，就会产生“围绕某个物体转圈观察”的效果。开发者可以自行通过代码来控制自己想要的镜头动画，也可以通过mono的mono.Inspection类直接使用已经封装好了的巡航函数。Inspection类可以接受一个轨迹点组成的数组，数组中每一个点代表需要巡视的一个3D空间的点，在运行时，镜头会沿着这些点定义的轨迹进行动画移动。

```
1 new inspection = new mono.Inspection(network, defaultInteraction, points, callback, callbac
```

- network – network对象
- defaultInteraction – network上设置的DefaultInteraction。它会控制用户交互时的镜头目标点，因此这里需要指定它
- points – 巡航点数组。数组中每一个点都是一个mono.Vec3(x,y,z)实例
- callback – 巡航结束后的后续动作，是一个回调函数

- callbackDelay – 巡航结束后，延迟多少时间（毫秒），再执行后续动作（回调函数）

下面代码演示了如何使用Inspection创建轨迹，并进行播放。代码中，巡航结束后，延迟2秒自动重复巡航。

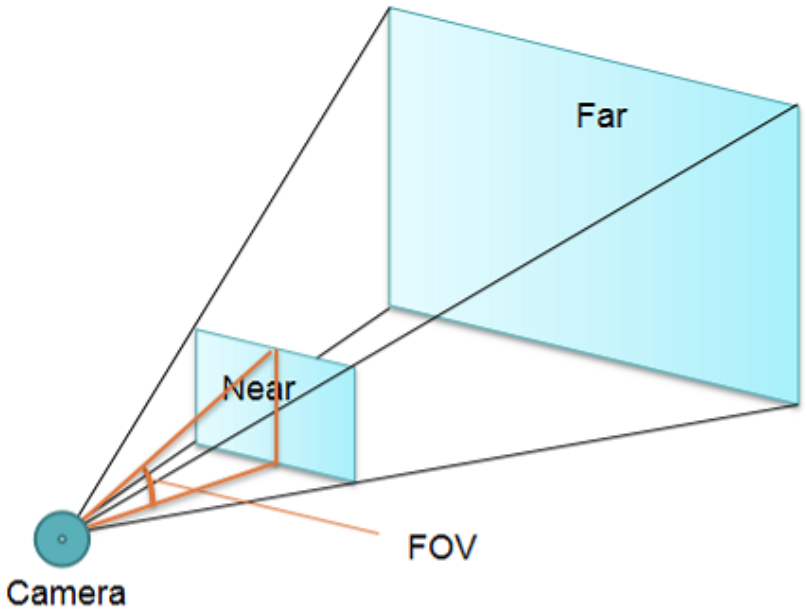
代码如下：

```
var playInspection = function() {
var points=new Array();
points.push(new mono.Vec3(1000,500,1000));
points.push(new mono.Vec3(500,250,500));
points.push(new mono.Vec3(500,100,-500));
points.push(new mono.Vec3(-500,100,-500));
points.push(new mono.Vec3(-500,100,600));
points.push(new mono.Vec3(300,100,300));
points.push(new mono.Vec3(250,100,250));

var inspection = new Inspection(network, interaction, points, playInspection, 2*1000);
inspection.play();
}
```

mono.Inspection最主要的函数就是play()，用来启动动画。

深入理解镜头

除了position、target两个参数外，镜头还有近切面、远切面、视场角等概念，在开发中也常用到，需要进一步理解。先看一张图：



这个图解释了镜头的基本概念。当人的眼睛放置在3D场景中时，我们能看到最远的地方是距离为far的面，最近的地方是距离为near的面。太远或太近的物体，都一律看不见，这和现实世界道理类似。于是产生了几个非常重要的定义：

- near：近切面距离。例如：如果100，则表示距离镜头100以内的物体一律切掉，看不见。现实世界中，当一个物体离我们眼睛过近，也是无法看清的；
- far：远切面距离。例如：如果设置10000，则表示距离镜头超过10000的物体，一律切掉，看不见。显示世界中，当物体离我们肉眼太远，就会无法看清；
- fov：Field of View，视场角。这个角度决定了镜头能够看到的场景的广度，数值越大，则看到场景的角度越大（类似广角镜头）。

上面的图可以看出：3D中的物体只有出现在near和far组成的四棱锥空间内，才可能被镜头看到，否则看不见。因此，near、far、fov组成了镜头最重要的三个参数。

mono中，通过setNear、setFar、setFov方法即可设置近切面、远切面、视场角：

```
camera.setNear(20);
camera.setFar(50000);
camera.setFov(60);
```

mono中，Near、Far、Fov的默认值：

- Near：默认值1；
- Far：默认值50000；
- Fov：默认值50度角；

那么，应用中，应该如何设置这三个数值呢？这是一个需要讨论的问题。

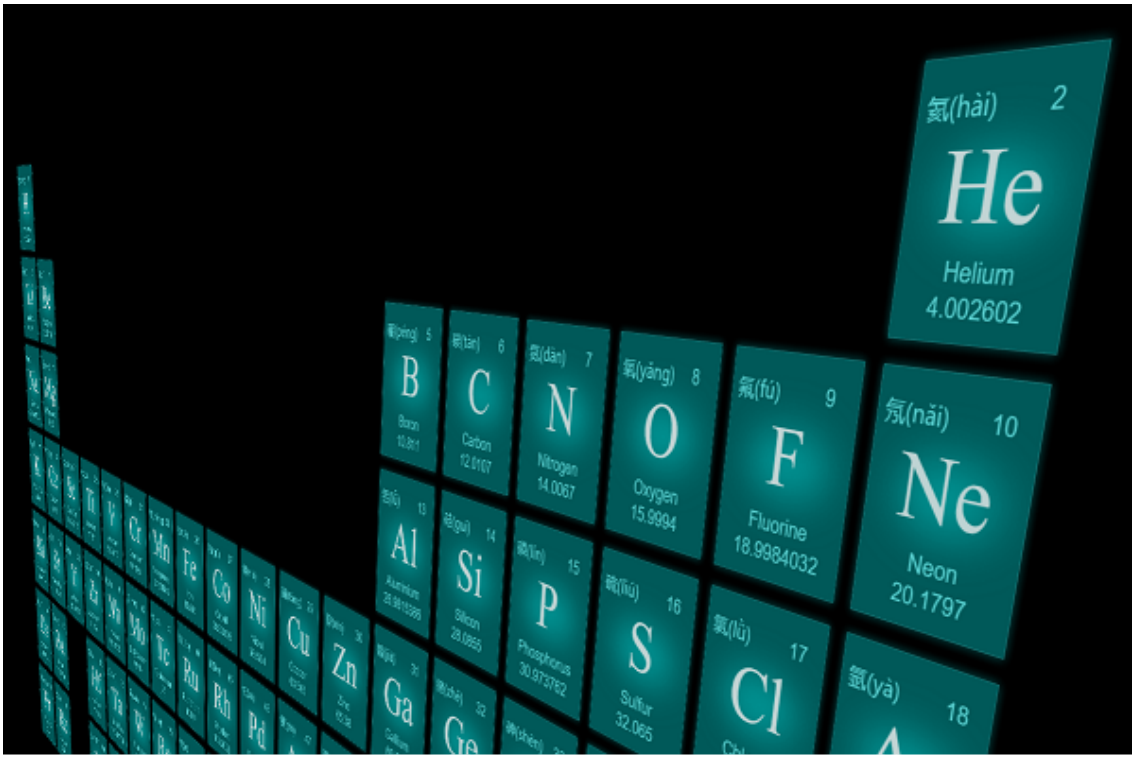
如何设置Fov

首先看Fov。对于大多数场景，使用默认数值会产生舒服的标准镜头感觉，场景没有拉伸，视觉感觉合理舒服，也不会产生“头晕”的感觉。如果没有特别必要，可以直接是用默认值即可。对于某些场景，也可以故意提高数值，产生夸张的“广角”效果，也能产生更“宏大”的感觉。

下面的程序分别是用了30度和60度的情况，可以观察其视觉上的变化：

30度视场角：

60度视场角：

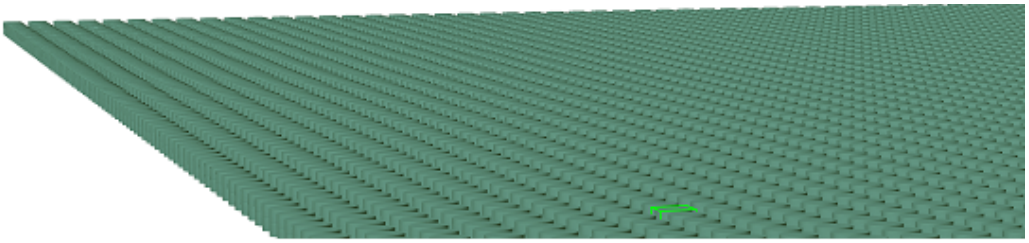


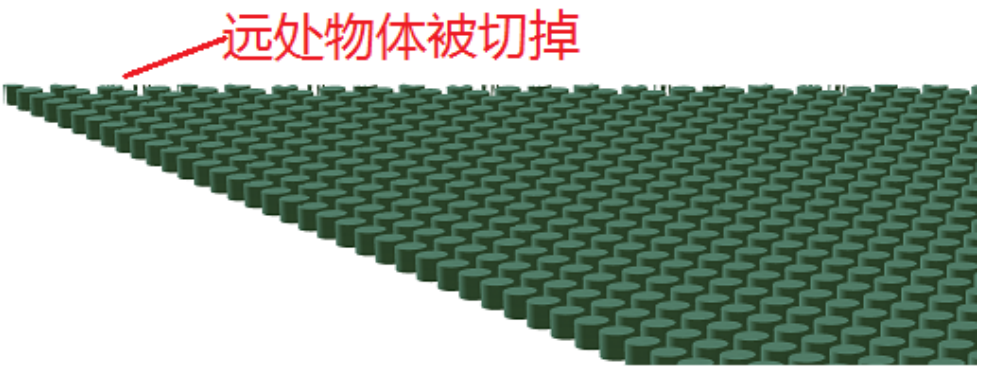
当角度越大，产生的立体感越强，但较近物体也会产生更大的变形。

如何设置远近切面

再看near和far。似乎near越小越好、far越大越好，因为能看到更多的场景。这一想法自然正确，但是在实际应用中，过大的near-far数值范围，会导致显卡要将near-far空间范围内纵向做更多的“切片”，并把物体的每个面啮合到这些切片中，以便判断这些面的前后遮挡关系。near-far数值越大，需要的切片就越多；或者说显卡提供的切片数量一定，则切片的粒度就会越粗。越粗的切片，就导致越粗的遮挡关系。当场景中两个非常接近的面不能被精细的分到不通的切片中时，就会产生“无法区分遮挡关系”的状况，表现为两个面随机相互遮挡，产生闪烁。

当far过小，场景中如果有物体处于far之外，则会消失看不见。一些形状的物体表现为被切掉了一部分。下面的场景far设置了5000和50000时，部分远处的物体被视线切掉了：





因此：设置near和far是一个取舍的过程：过大的near-far范围差，会减少物体被远近视线切掉的机会，同时会导致场景内切片变粗，产生场景内物体闪烁的可能；太小的near-far差，可能导致物体容易超出视线，被视角切掉。但会降低物体闪烁的机会。

如果是一个大场景，里面的物体尺寸都很大，例如尺寸很大的楼宇等，可以将far增大（例如到50000），near也可以适当增大（例如10-100）。

如果是一个小场景，里面的物体都很小，例如一桌子的书本杯子手表等，可以将far调小到比桌子直径大几倍就可以了（例如5000），near则调小（例如1），避免小物体的面相邻很近产生闪烁。

如果一个场景有非常大的建筑物体，又有戒指一样的小微物体，还要不闪烁、能看到全景，这本身就会产生矛盾，应当尽量避免。也可以根据镜头的位置进行动态near、far设置。

使用第一人称视角

3D的镜头交互分为第一人称视角和第三人称视角。所谓的第一人称视角，其实是针对交互而言的，也就是network的Interaction，详见Network章节。第一人称视角的本质，是交互时移动的是camera的position，还是target。

例如，通过鼠标拽动场景，如果：

- position不变：则拖拽导致的是target观察角度的变化，体现为场景的旋转，仿佛我们站在原地，眼睛向左或右侧旋转。这是第一人称视角；
- target不变：则拖拽导致的是position的变化，也就是我们眼睛盯住的点不变，而是我们站立的位置围绕着target点发生旋转，仿佛我们围绕目标物体在旋转一样。这是非第一人称视角；

mono中，交互默认是第三人称视角。如果要启用第一人称视角，可以从network获得Interaction并进行设置：

```
var interaction = network.getDefaultInteraction();
interaction.fpsMode = true;
```

这样便启动了第一人称视角模式。更多关于Interaction的用法介绍，请参照network章节。

使用平行镜头

mono中默认的、最常用的镜头是mono.PerspectiveCamera，它是基于透视投影（Perspective Projection）进行定义。还有一种镜头是采用平行投影的镜头，在mono中通过mono.OrthoCamera进行定义。

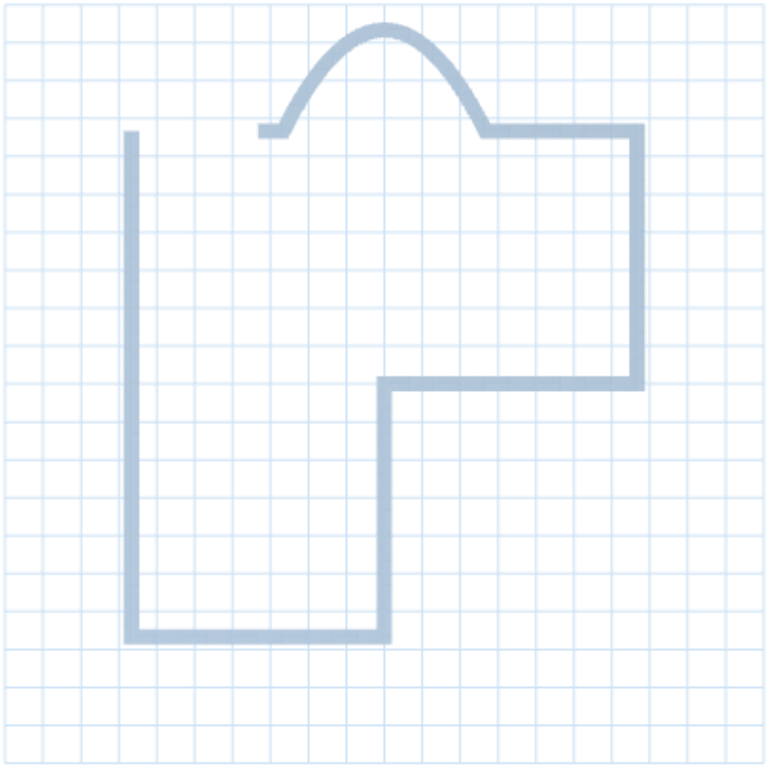
平行投影镜头用法和透视投影镜头类似，new实例后set到network对象上即可生效。它在3D中不常用，但可用来做某个角度的2D视角。例如，采用平行投影镜头从垂直上方向下方观察，可以得到精确的类似2D的平面图。

下面的程序就是采用了mono.OrthoCamera镜头，从垂直上方向下方观察一个不规则墙体的形状：

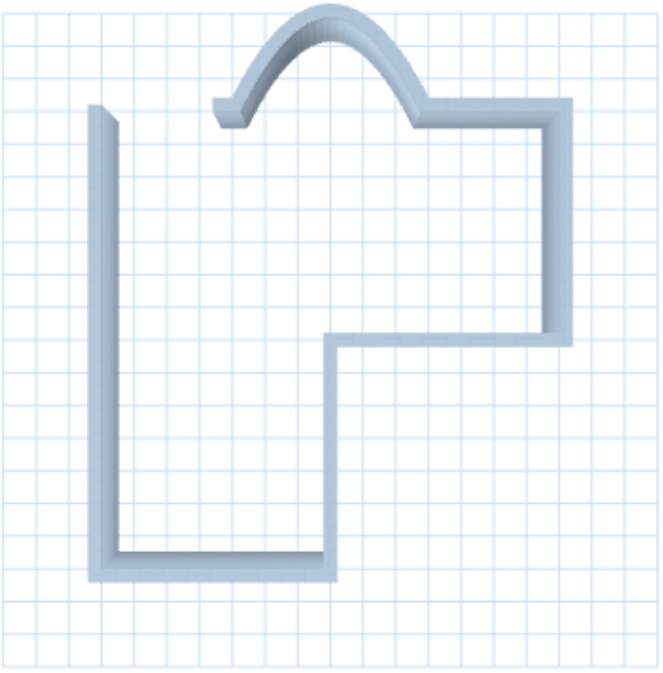
```
var box = new mono.DataBox();
var network= new mono.Network3D(box, null, monoCanvas);
network.getDefaultInteraction().maxDistance = 1000;
network.getCamera().setPosition(50, 400, 50);
network.getCamera().lookat(50, 0, 50);
network.isSelectable = function(element){
  return false;
};
mono.Utils.autoAdjustNetworkBounds(network,document.documentElement,'clientWidth','clientH
var pointLight = new mono.PointLight(0xFFFFFF,0.5);
pointLight.setPosition(1000,1000,1000);
box.add(pointLight);
var pointLight = new mono.PointLight(0xFFFFFF,0.5);
pointLight.setPosition(-1000,-1000,-1000);
box.add(pointLight);
box.add(new mono.AmbientLight(0xaaaaaa));

var path = new mono.Path();
path.moveTo(0, 0,0);
path.lineTo(100, 0, 0);
path.lineTo(100, 50, 0);
path.lineTo(50, 50, 0);
path.lineTo(50, 100, 0);
path.lineTo(0, 100, 0);
path.lineTo(0, 70, 0);
path.curveTo(-40,50,0,0,30,0);
path.lineTo(0,25,0);

var wall = new mono.PathCube(path,3,30,32,10);
wall.s({
  'm.texture.image': 'wall2.png',
  'm.type': 'phong',
  'm.color': '#CEE3F6',
  'm.ambient': '#CEE3F6',
  'outside.m.lightmap.image': 'outside_lightmap.jpg',
  'inside.m.lightmap.image': 'inside_lightmap.jpg',
});
wall.setPosition(0, 0, 100);
box.add(wall);

var floor = new mono.Plane(150, 150, 20, 20);
floor.s({
  'm.wireframe': true,
  'm.color': '#CEE3F6',
  'm.side': mono.DoubleSide,
});
floor.setRotation(Math.PI/2,Math.PI,Math.PI/2);
floor.setPosition(50, 0, 50);
box.add(floor);
```

```
50
51 var camera = new mono.OrthoCamera();
52 camera.setPosition(network.getCamera().getPosition());
53 camera.lookat(50, 0, 50);
54 network.setCamera(camera);
```

运行效果如下：



如果去掉最后几行设置OrthoCamera镜头的代码，而使用默认镜头，则显示效果如下：



可见，平行投影在进行房间平面布局图等场景能发挥独特的作用。

> **注意：**
>
> 上面的例子中使用了m.lightmap.image属性为墙壁设置了光照影射图，增加逼真度。这也是mono中经常采用的一种简单、有效的增强立体感、质感的一种方法。详细用法请参阅样式表一章。

技巧：如何避免闪烁

如上所述，在mono中，物体产生闪烁，是因为GPU已经不能提供足够精细的切面来正确的判断两个面的遮挡关系，因此每一帧产生随机的相互遮挡，导致闪烁产生。这是现在的3D技术都会产生的问题，需要在应用中进行调整和取舍。

更多关于这一现象的原理，请查阅搜索引擎“z-fighting”关键字。

知道闪烁的原理，就可以有针对性的进行调整。可以从以下几个方面考虑：

- 减小near-far范围。这是最直接有效的方式。一个1-50000范围的场景比一个30-10000范围的场景无疑需要更多的切片才能精细的辨别相邻很近两个面的遮挡关系。因此，如果没有大物体，可以适当调小far值（5000-10000调整），或增大near值（10-100范围调整）。near和far值无法给一个固定参考，因为这和场景的大小和物体的大小有关系；
- 避免两个面离得太近。在一个1-50000的场景中，距离0.1的两个平行的面，需要大量的切片才能确认其遮挡关系，因此很容易产生闪烁，尤其在镜头拉远以后。可以将距离设置为1就会好非常多；
- 如果两个面确实在同一个位置且有遮挡关系（例如贴在墙面上的一幅画），可以考虑用polygon offset来明确设置面的遮挡关系（具体见后面章节）。需要留意的是，polygon offset的大量使用会对性能产生影响；
- 避免是用透明。透明是一个很好的效果，但是大量的透明面相互叠加和遮挡，会造成遮挡关系难以计算的问题，产生闪烁。如果是多个透明面导致的闪烁，可以适当减少透明面的是用；

技巧：如何避免远处物体被切掉

如上所述，远端物体被切掉是因为它的位置已经超出了far的范围，因此不再显示。如果要避免这种现象，应该调大far的数值。

由于鼠标滚轮可以通过滚动缩小整个场景，也就是把镜头拉向身后非常远的距离，因此再大的far值也不可能彻底避免这一现象。也可以通过设置交互的最远距离来限定镜头的位置：

```
network.getDefaultInteraction().maxDistance = 1000;
```

上面的代码给network的DefaultInteraction设置了一个最大距离值，当鼠标滚轮缩小（拉伸镜头）到10000的位置时，就不能再继续缩小（拉伸镜头）。设置interaction的maxDistance是一个良好的习惯。它可以避免让用户不经意把整个场景缩小到消失不见，产生不好的用户体验。

使用内置简单动画

整个twaver产品线中，twaver html5 2d中提供了基础的、灵活的animation动画框架，它功能最强大、完善、通用。此外，mono在3d中也提供了更简单的动画mono.Animation，也可以完成非常简单的动画效果。

如果您需要的只是非常简单的3D动画，例如物体的移动、旋转等，可以直接使用mono中的Animation来完成；如果您需要的是比较复杂的、自定义的动画，应使用twaver html5中的Animation框架。它需要用到twaver 2d，具体用法参阅TWaver HTML5开发指南。这里只介绍mono.Animation的用法。

mono.Animation的定义如下：

```
1 mono.Animation =function(element, startValue, endValue, duration, callback)
```

其中：

- element – 发生动画的3D对象
- startValue – 动画要驱动的数值开始值。例如，沿x轴从0度旋转至90度，则开始值是0度
- endValue – 动画要驱动的数值结束值。例如，沿x轴从0度旋转至90度，则结束值是90度
- duration – 动画帧插入的时间间隔，单位是毫秒（1/1000秒）。如果设置了该值，则表示network刷新一帧后，会强行等待指定时间，再进行下一帧的绘制。数值越大，动画越慢、不流畅，但CPU负载越低。反之，动画会更加流畅，但负载会更大。设置为0则会全速渲染
- callback – 动画结束时的回调函数。例如，可利用该机制在动画结束后继续重复播放

主要方法

- play: function () – 开始播放动画
- function getAllAnimationTypes() – 获得所有支持的动画类型
- ‘LeftMove’ – 向左移动
- ‘RightMove’ – 向右移动
- ‘FrontMove’ – 向前移动
- ‘BackMove’ – 向后移动
- ‘LeftRotationClockwise90’ – 以左侧为中心顺时针旋转90度
- ‘LeftRotationAnticlockwise90’ – 以左侧为中心逆时针旋转90度
- ‘LeftRotationClockwise120’ – 以左侧为中心顺时针旋转120度
- ‘LeftRotationAnticlockwise120’ – 以左侧为中心逆时针旋转120度
- function playAnimation (element, animation) – 为指定对象播放指定类型的动画
- function playAlarmAnimation(element, alarmBillboard) – 为指定对象播放告警动画。alarmBillboard是一个表示告警样式的公告牌对象，可按喜好进行定义。

基本概念

灯光是3D中的重要元素，它可以有效的提高场景的仿真度。mono中已经封装了许多种类的灯光对象，可以拿来直接使用。但是如何在应用中合理的的使用好灯光效果，还不是一件容易的事。

在使用灯光之前，应理解以下概念：

- 灯光也是一个3D对象（mono.Light），和mono.Cube这样的立方体对象一样，有自己的位置，有自己的style参数，需要添加到DataBox中。不同的是，它本身是“看不见、摸不到”的，是用来“照亮别人”的；
- 多个、多种灯光对象，可以同时存在。这些灯光会在物体上产生叠加效果；
- 只有m.type为phong的表面，才响应和产生灯光效果。对于m.type为basic的表面，灯光是不起任何作用的；
- 灯光对象不一定能实现所谓的“高大上”的光影效果。使用basic材质+各种map贴图，一样能产生更复杂精细的光照效果；
- 灯光及各种光影相关的map技术，都会或多或少的影响3D渲染性能，应按需权衡使用；

灯光的局限

mono中的灯光也有一定的局限。

首先，mono的产品定位是企业应用，因此它更注重的是实用性：简单易用、高性能、跨平台是首要要求，而复杂的灯光效果会增加复杂度、降低渲染性能。因此mono中的灯光设计并不非常强大，一些复杂的灯光特效、阴影、光线追踪等技术都未提供。这些技术可能在仿真、家居、游戏等行业应用需求会比较多。

其次，mono是基于webGL的引擎，受webGL标准、浏览器、开发语言等方面的限制，其功能和效率与直接使用OpenGL/DirectX等底层3D接口无法比拟的。因此mono中的灯光设计也是遵循“简单、够用”的原则。

如果使用mono做3D企业应用，我们也应该有同样的设计思路：追求简单、清晰、够用的光影效果，而不过度追求复杂效果和逼真度。否则，mono这种webGL技术的3D引擎不一定能够满足要求。

灯光的分类和使用策略

之前对象详解章节介绍了灯光对象的种类。mono.Light基类下定义了4种光源：

- 环境光源：mono.AmbientLight
- 点光源：mono.PointLight
- 聚光源：mono.SpotLight

- 方向光源：mono.DirectionalLight

光源对象主要的参数有：颜色（setColor/getColor）、环境光颜色（setAmbient/getAmbient）、散射光值（setDiffuse/getDiffuse）、镜面光颜色（setSpecular/getSpecular）。这些参数大多都不太常用。

以上灯光对象，一般，只有mono.AmbientLight环境光和mono.PointLight电光源比较常用。

以下例子中，均使用下面的代码创建的模拟场景：一个地板和一个3D物体矩阵。代码如下：

```
for(var i=0; i<10; i++){
  for(var j=0; j<10; j++){
    var cube=new mono.Cube(10, 10, 10);
    cube.setName('node '+i);
    cube.s({
      'm.type': 'phong',
      'm.texture.image': 'box.jpg',
    });
    cube.setPosition((i-5)*20+cube.getWidth(), cube.getHeight()/2, (j-5)*20+cube
    box.add(cube);
  }
}

var floor=new mono.Cube(220, 1, 220);
floor.s({
  'm.type': 'phong',
  'm.texture.image': 'default_texture.png',
  'm.repeat': new mono.Vec2(10, 10),
});
floor.setPositionY(-floor.getHeight()/2);
box.add(floor);
```

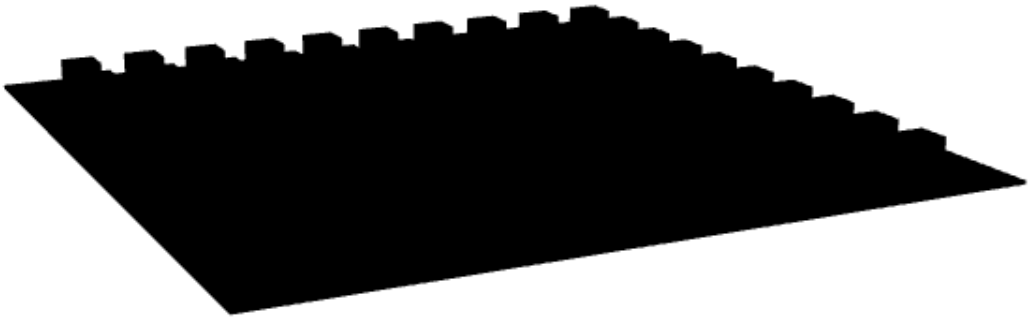
使用环境光

环境光是一个必须的光源，除非整个场景中m.type全都是basic类型。对于有phong类型表面的场景：

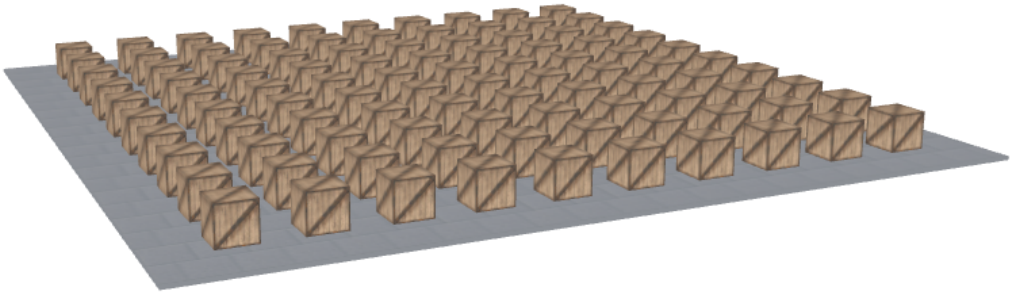
- 没有环境光和任何其他类型光源：所有phong表面将呈现完全的黑色，什么也看不清；
- 只有环境光，没有其他类型光源：所有phong表面材质将亮度呈现一致，不同表面不会产生明暗差异；
- 只有点光源，没有环境光：phong表面根据所在位置产生强烈明暗差异；
- 环境光+点光源：最常用方法，会较好的融合环境光和点光源产生的明暗效果；

环境光对象不需要位置，只需要颜色值。颜色值决定了环境的整体色调。没有特别必要，一般设置成灰度颜色，例如白色、灰色等。

使用上面的场景，物体表面均使用phong材质，且不添加任何光源，则场景会乌黑一片：



如果将物体的所有类型改为basic，则物体完全表现为素材本身贴图，其色彩、明暗完全由贴图控制：



看上去效果似乎还不错，但实际上非常依赖于贴图的质量。如果贴图本身具有一定的明暗甚至阴影效果，则能够产生较好的立体感；否则会显得很“平”，尤其在不使用贴图而使用颜色的情况下。下面代码修改为使用颜色而不使用贴图：

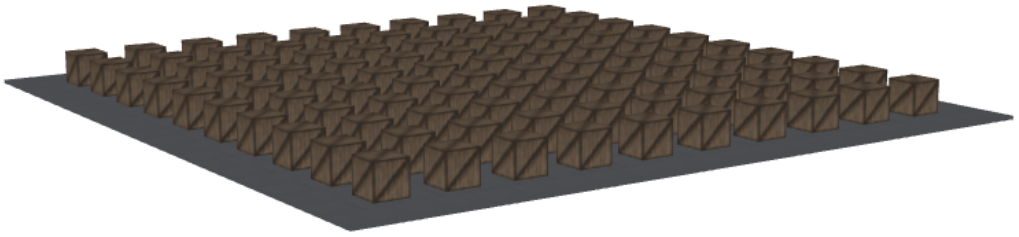
```
for(var i=0; i<10; i++){
  for(var j=0; j<10; j++){
    var cube=new mono.Cube(10, 10, 10);
    cube.setName('node '+i);
    cube.s({
      //'m.type': 'phong',
      //'m.texture.image': 'box.jpg',
      'm.color': '#04B4AE',
      'm.ambient':'#04B4AE',
    });
    cube.setPosition((i-5)*20+cube.getWidth(), cube.getHeight()/2, (j-5)*20+cube
    box.add(cube);
  }
}
```

可见，不使用贴图的物体表面，在任何位置都呈现出完全相同的颜色，缺少因位置产生的明暗变化，没有立体感。

继续改进，采用phong材质+0x888888颜色的环境光：

```
box.add(new mono.AmbientLight(0x888888));
```

在使用贴图的情况下：



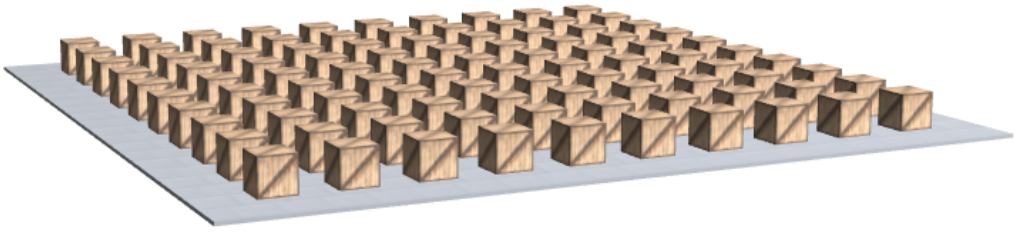
在使用颜色的情况下：

场景整体偏暗，可以通过修改环境光的颜色0x888888来调整。但物体依旧呈现为环境光下的单色，缺少立体感。

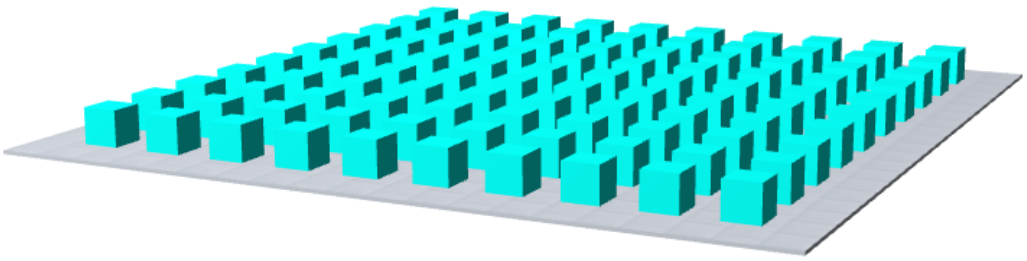
继续改进。在场景的上方（1000、1000、100）位置添加一个强度为1.5的白色(0xFFFFFF)点光源：

```
var pointLight = new mono.PointLight(0xFFFFFF,1.5);
pointLight.setPosition(1000,1000,1000);
box.add(pointLight);
```

贴图情况下：

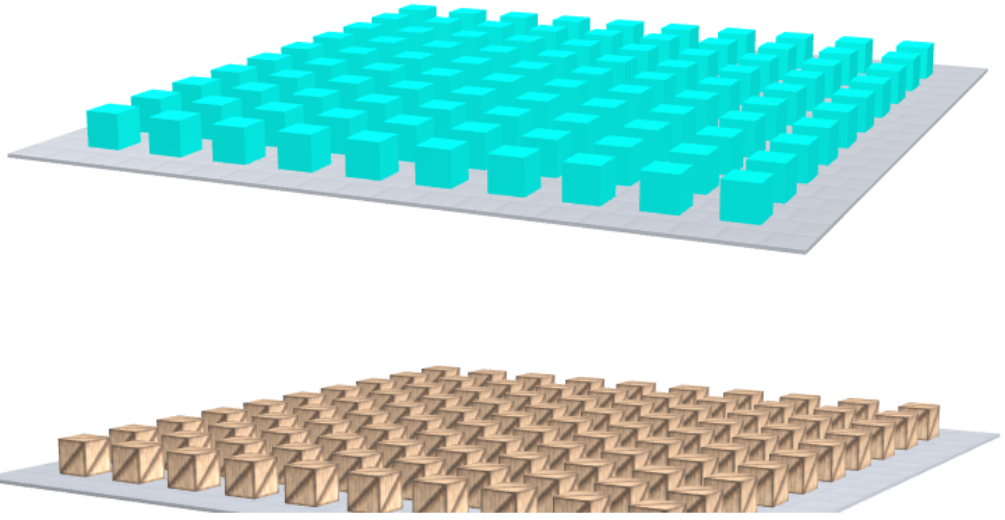


单色情况下：



已经能产生较好的明暗变化。单个点光源产生的明暗变化可能会比较生硬。适当降低一些灯光强度，在不通的角度和位置多增加一些光源，会产生更柔和的效果。这和摄影、摄像技术类似。下面综合使用了3个点光源和一个环境光：

```
var pointLight = new mono.PointLight(0xFFFFFF,1);
pointLight.setPosition(1000,1000,1000);
box.add(pointLight);

var pointLight = new mono.PointLight(0xFFFFFF,1.2);
pointLight.setPosition(-1000,-1000,-1000);
box.add(pointLight);

var pointLight = new mono.PointLight(0xFFFFFF,0.5);
pointLight.setPosition(1000,1000,0);
box.add(pointLight);

box.add(new mono.AmbientLight(0x888888));
```



此时，无论是贴图还是单色渲染，都能产生比较自然的光影变化。

使用点光源

如上面例子所述，在实际使用中，环境光+多点光源的方案是效果最好的。在实际使用中，应该主要考虑点光源的数量、位置、强度，以及环境光的组合，调整出最佳的效果。

使用点光源的注意事项：

- 点光源数量不宜过少或过多。一般使用3-4个点光源，并合理安排光源位置；
- 灯光位置要分散开，并置于场景较远的位置；
- 三角光源布局法：A灯位于(1000,1000,1000)，强度1；B灯位于(-1000,-1000,-1000)，强度1；C灯位于(1000,1000,0)，强度0.5；

使用光照相关样式

配合光源对象，使用一些光照相关的style样式，可以更好的增加逼真度。

- m.specularStrength：反光强度。将光滑、镜面的物体表面设置更高的数值，可以提高逼真度；
- m.envmap.image：环境映射图。将光滑的反光表面设置环境映射图，可以有效的提高逼真度；
- m.lightmap.image：光照贴图。如果使用光源对象无法表现表面的细腻光线效果，可以使用预先定制好的光照贴图。虽然不是实时计算渲染的，但是速度快、逼真度高，甚至可以完全代替光源机制。具体使用见下节介绍；

下图是综合使用了光源、各种map贴图及相关style产生的效果：

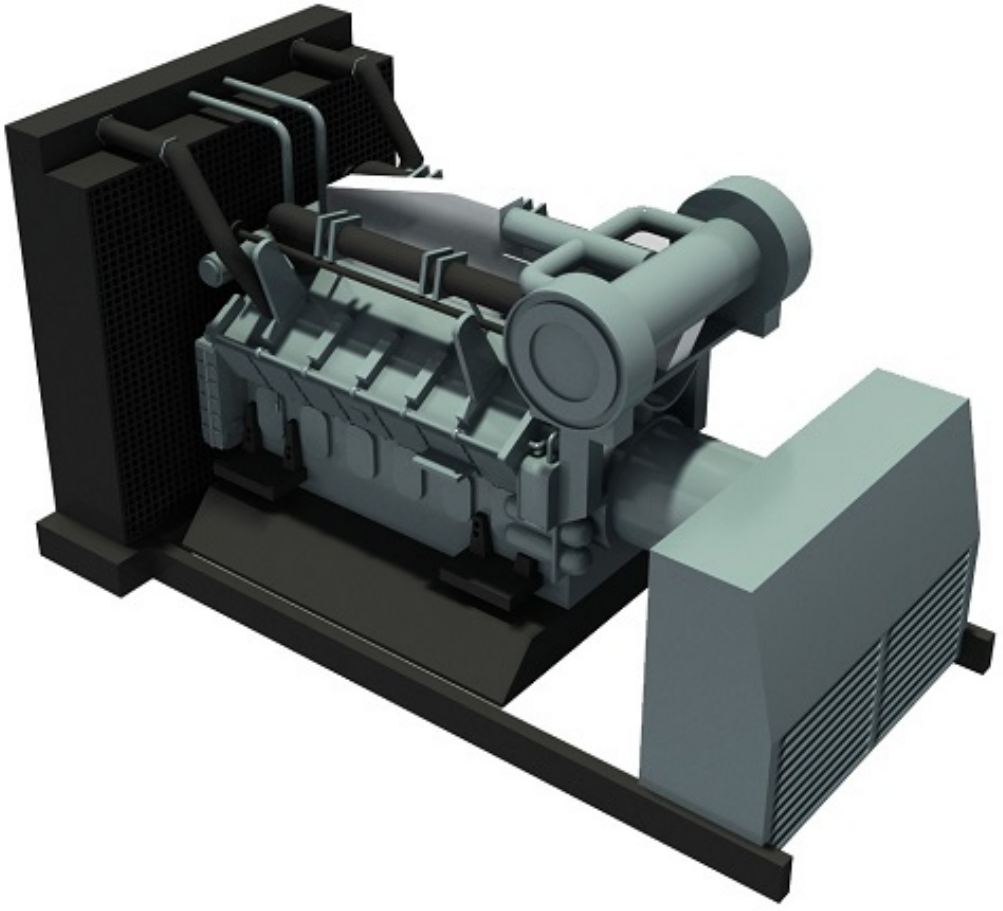
不使用灯光

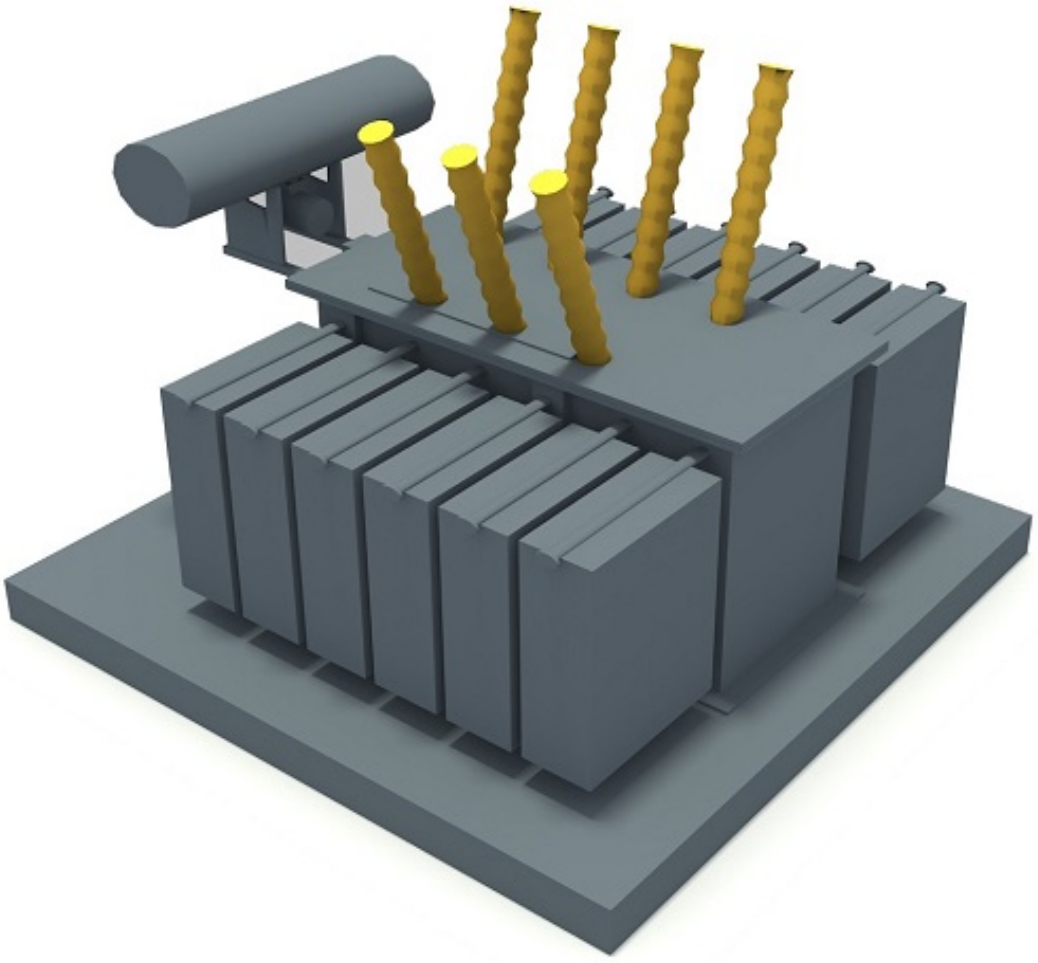
对于复杂细腻的光照效果，靠光源是难以实现的。此时，可以完全使用basic材质+lightmap机制来实现。lightmap可以提前在3D Max等软件中进行提前渲染，然后在mono中进行使用。

对于这种不使用光源对象的方式，反而可以做出更加细腻的效果。它适合渲染相对静止不动的物体和场景，相对位置如果经常变化的则不是特别适合这种方式。

下图是一个完全使用basic+map机制产生的静态物体，具体见mono产品包中的/mono-design/demo/watch/index.html例子。

下面是更多的使用这种方式制作的静态物体模型：





使用告警

Mono中的告警机制和TWaver 2D中完全一样。Mono定义了告警级别，并在DataBox中内置了告警管理容器AlarmBox。AlarmBox可以通过函数box.getAlarmBox()获得。当告警发生时，可以创建告警对象并添加到AlarmBox中即可。

告警级别

告警级别都定义在mono.AlarmSeverity类中，它代表了不同严重级别的告警，并具有不同的颜色定义。

告警级别	严重值	名称	标志代码	颜色
mono.AlarmSeverity.CRITICAL	500	Critical	C	#FF0000
mono.AlarmSeverity.MAJOR	400	Major	M	#FFA000
mono.AlarmSeverity.MINOR	300	Minor	m	#FFFF00
mono.AlarmSeverity.WARNING	200	Warning	W	#00FFFF
mono.AlarmSeverity.INDETERMINATE	100	Indeterminate	N	#C800FF
mono.AlarmSeverity.CLEARED	0	Cleared	R	#00FF00

创建告警对象

下面代码构造了一个新的告警对象：

```
1 var alarm = new mono.Alarm(id, elementId, alarmSeverity, isAcked, isCleared);
```

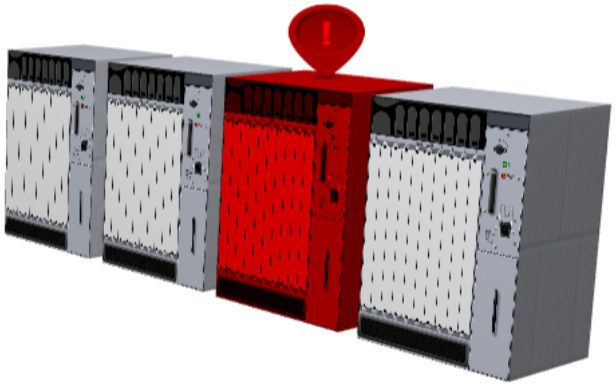
其中:

- id – 告警的id
- elementId – 告警对应的3D对象的id
- alarmSeverity – 告警级别
- isAcked – 告警是否已经确认
- isCleared – 告警是否已经清除

下面代码演示了如何创建一个告警，并添加到DataBox中。

```
1 var alarm = new mono.Alarm(id, id, mono.AlarmSeverity.CRITICAL);
2 box3d.getAlarmBox().add(alarm);
```

下图显示了发生告警的设备与未发生告警设备的对比：



本页列出了TWaver HTML5 3D经常用到的功能和使用方法，方便您日常速查。

浏览器如何允许加载本地资源

如果3D场景在浏览器上无法显示，或物体表面呈黑色时，可在Chrome中按F12按钮，查看Console控制台日志。如能看到“Unable to get image data from canvas because the canvas has been tainted by cross-origin data.”的错误提示，则意味着浏览器的安全策略被设置为禁止浏览本地资源。浏览器为了阻止欺骗，会追踪 image data。当你把一个“跟canvas的域不同的”图片放到canvas上，这个canvas就成为 “tainted”(被污染的，脏的)，浏览器就不让你操作该 canvas 的任何像素。

这个问题有两种解决方案

方案一：有服务器环境，将项目部署在web服务器上，最简单的tomcat。

方案二：设置浏览器

On Windows：

Chrome:

1、得到Chrome的安装路径，例如：

```
C:\Users\-your-user-name\AppData\Local\Google\Chrome\Application
```

2、在命令行窗口，输入安装路径，加上参数

```
-allow-file-access-from-files
```

例如：

```
Chrome installation path\chrome.exe --allow-file-access-from-files
```

回车执行，启动Chrome

3、测试的一个临时方法:：复制一个Chrome的快捷方式，右键->属性->目标的文本框中加上参数

```
--allow-file-access-from-files
```

例如：

```
"Chrome installation path\chrome.exe" --allow-file-access-from-files
```

IE11：默认可以加载本地图片

Firefox：

1、Firefox的用户请在浏览器的地址栏输入“about:config”，回车

2、在过滤器（filter）中搜索“security.fileuri.strict_origin_policy”

3、将security.fileuri.strict_origin_policy设置为false

4、关闭目前开启的所有Firefox窗口，然后重新启动Firefox。

On Mac：

Chrome：从命令行窗口中启动，启动命令为

```
open /Applications/Google\ Chrome.app --args --allow-file-access-from-files
```

Safari：

1、Safari->偏好设置->高级->勾选“在菜单栏中显示‘开发’菜单”

2、开发->勾选“启用WebGL”

3、开发->勾选“停用本地文件限制”

Firefox：

1、Firefox的用户请在浏览器的地址栏输入“about:config”，回车

2、在过滤器（filter）中搜索“security.fileuri.strict_origin_policy”

3、将security.fileuri.strict_origin_policy设置为false

4、关闭目前开启的所有Firefox窗口，然后重新启动Firefox。

如何为浏览器设置WebGL

要运行WebGL，必须有一个支持它的浏览器。先来看看在桌面平台上有哪些设备和平台已经支持WebGL了

- Google Chrome 9及以上版本
- Mozilla Firefox 4以及上版本
- Safari 5.1及以上版本（仅限于Mac OS X操作系统，不包括Windows操作系统；所有情况下必须强制开启WebGL支持，请参考下文）
- Opera Next 即Opera 12 alpha及以上版本
- IE并不支持WebGL(IE11支持WebGL)，但是可以下载并安装IEWebGL这个插件，或Google Chrome Framework来运行一些WebGL应用

那么这些浏览器该如何启用WebGL呢，请看下面解决方法：

Chrome浏览器

需要为Chrome加入一些启动参数，以下具体操作步骤以Windows操作系统为例

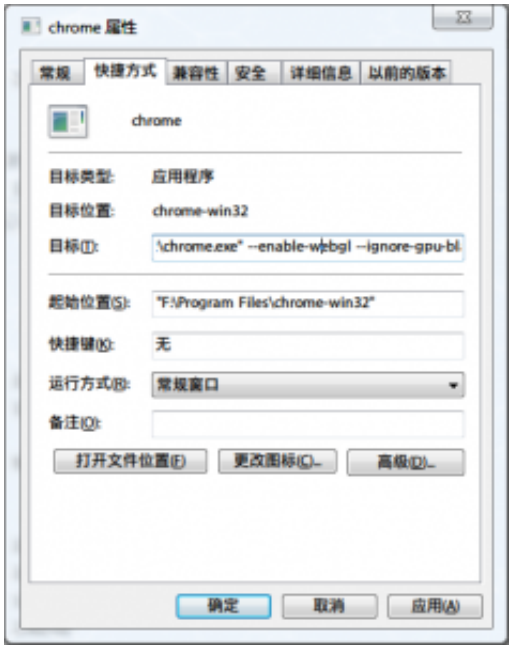
1、找到Chrome浏览器的快捷方式，如果没有就创建一个快捷方式(右键点击chrome.exe，选择“创建快捷方式”或者“发送到”→“桌面快捷方式”)，右键点击快捷方式，选择属性。

2、在目标框内，双引号的后边，加入以下内容

```
--enable-webgl --ignore-gpu-blacklist --allow-file-access
```

点击确定

3、设置完成的后的快捷方式属性窗口看起来应当是这样的，注意“目标”文本框：



4、关闭目前开启的所有Chrome窗口，然后用此快捷方式启动Chrome浏览器。

其中：

```
1 --enable-webgl
```

的意思是开启WebGL支持

```
1 --ignore-gpu-blacklist
```

的意思是忽略GPU黑名单，也就是说有一些显卡GPU因为过于陈旧等原因，不建议运行WebGL，这个参数可以让浏览器忽略这个黑名单，强制运行WebGL

```
1 --allow-file-access-from-files
```

的意思是允许从本地载入资源

Firefox浏览器

1、Firefox的用户请在浏览器的地址栏输入“about:config”，回车

2、在过滤器（filter）中搜索“webgl”

3、将webgl.force-enabled设置为true

4、将webgl.disabled设置为false

5、在过滤器（filter）中搜索“security.fileuri.strict_origin_policy”

6、将security.fileuri.strict_origin_policy设置为false

7、关闭目前开启的所有Firefox窗口，然后重新启动Firefox。

其中前两个设置是强制开启WebGL支持，最后一个security.fileuri.strict_origin_policy的设置是允许从本地载入资源

如果强制开启了WebGL，还是不能运行，更多信息参考Mozilla Blocklisting/Blocked Graphics

Drivers

我在Window XP上测试Firefox的WebGL时，在强制开启了WebGL后，仍然得到Error: WebGL: Error during ANGLE OpenGL ES initialization

悲催的发现Firefox对厂家的不支持和硬件不支持(分别在虚拟机上和显卡为Intel G31/G33 chipset)

On Windows

All vendors other than AMD/ATI, NVIDIA, Intel are blocked (bug 623338). This was required primarily by various crashes on virtual machines with unusual vendor names (bug 621411). We're open to whitelisting more vendors if needed.

… …

If force-enabling a feature doesn't work, that probably means that your hardware doesn't support it. For example, layers acceleration currently requires support for 4Kx4K textures, which rules out some graphics cards, like the Intel G31/G33.

Safari浏览器

1、进入“偏好设置 (Preferences)”菜单并点击高级 (Advanced)

2、选中复选框“在菜单栏中显示‘开发’菜单 (Show Develop menu in the menu bar)”

3、从“开发 (Develop)”菜单中启用 webGL，勾选“启用WebGL”

4、允许从本地载入资源，勾选“停用本地文件限制”

这里要强调：如果电脑的显卡非常老旧，或者是板载的集成显卡，那么需要在浏览器中强制开启WebGL支持；另外因为其他的一些原因（比如**操作系统是Windows XP**），在正常安装以上浏览器之后还是不能运行WebGL，最好也强制开启WebGL支持

如何判断自己目前的浏览器是否支持WebGL？

测试当前浏览器是否支持WebGL

- 如果显示Yay，说明浏览器支持WebGL
- 如果显示Nay，说明浏览器目前还不能运行WebGL

- 术语、功能列表及性能指标

术语描述

术语	描述
Entity	3D基本对象。它是MONO中最常见的一类3D对象，主要包括立方体，球体，圆柱体，环形，路径体，文字，形状体，路径方体，组合体等。
Link	连线，是一种网元，它表现为3D场景中连接两个物体的连线。它在电信拓扑中用于展示链路或基于数据的连线。比如带有物理或逻辑意义的电缆，电路，路由线路，环路等。
Particle	粒子系统，它是一种特殊的网元，没有具体的面，而是由一系列内部的顶点组成的，每个顶点显示给定的贴图。它在3D场景中可以展出烟雾，火焰等效果。
Billboard	公告牌，它是一种特殊的3D物体，它只有一个图片组成，而且这张图片会永远朝前面向镜头（也就是我们用户的眼睛），而无论场景如何变化。
Light	灯光，在3D场景中灯光效果是非常重要的元素，它模拟的现实世界中光照效果，使得3D场景看起来更加逼真，灯光主要有方向灯，环境灯，聚光灯，点光源几种。
Camera	镜头，在3D场景中模拟相机的效果，是用于将3D场景头投射到2D屏幕的一种3D对象。
Alarm	告警，是电信拓扑系统中的一种预定义的对象，是用于展示OSS系统中服务端实际发生的告警。
DataBox	数据容器，是一个内存容器，负责装载、管理、监控各种网络数据，它是所有MONO可视化组件的数据来源，是整个MONO组件的数据中心和引擎。
Network3D	拓扑组件，它是MONO的核心组件，用于显示网络拓扑图、设备机架图，并且提供完善的用户交互机制和丰富的界面展示功能。

产品名称与版本

MONO DESIGN v1.9

主要功能

模块	特性	说明
DataBox 数据容器	支持大数据量	可支持10万数量级以上的网元（普通PC、API方式）
	高效率	可在10秒左右加载10万个节点（普通PC、API方式）
	基本3D对象	支持立方体，球体，圆柱体，圆环，路径体，文字，形状体，路径方体，管线，粒子系统，组合体，平面，公告牌，连线
	3D对象的wireframe	支持3D对象的wireframe的显示方式
	自定义属性	支持通过API或者XML来定义和显示自定义属性
	数据监控	可对数据的进/出/修改/清空等进行监听和控制
	数据管理	可对数据进行增删改查
	批量操作	支持对数据插入修改的批量操作
	数据容器属性的监控	可对数据容器的属性进行监控
	遍历网元	支持通过类型、深广度等多种网元遍历功能
	延迟加载	可定义数据的延迟加载，提高内存利用率与效率
	网元快速查找	可通过QuickFinder快速查找网元
	分组管理	支持对3D对象进行分组，拆除分组的操作
	支持JSON数据	可加载JSON文本定义的数据，和导出JSON格式的文本数据
	支持序列化	可将整个容数据序列化用于前后台通讯
	支持远程数据	可通过HTTP等协议远程加载网络数据
	选中管理	可对数据的选中进行管理，如添加/删除/获取/清空所有选中的网元
	选中监听	可以监听网元是否选中的状态

	选择过滤器	可添加网元是否可选中的过滤器
	告警对象化	可通过AlarmModel加载Alarm告警对象
	告警容器	可通过AlarmBox管理告警对象，对告警对象进行增删改查
	告警对象快速查找	可通过QuickFinder快速查找告警对象
	告警图元映射	可通过AlarmElementMapping自定义告警与图元映射
	告警级别	可通过AlarmSeverity自定义告警级别、颜色、比较器
	告警状态	可通过AlarmState对告警状态进行管理，如获取最高级别的新发告警，新增/移除新发告警，确认告警
	告警传播	可通过AlarmStatePropagator进行告警传播规则自定义
	告警冒泡显示效果	新发告警上来后，支持告警的冒泡显示效果
Network 拓扑组件	事件交互	支持多种交互方式，默认交互，选择交互，编辑交互
	平移，缩放，旋转	支持设置平移、缩放、旋转的速度
	框选	支持选中多个对象和框选对象
	帮助组件	编辑模式下支持显示帮助组件
	鼠标、键盘支持	右键平移、鼠标双击/单击事件定义、键盘操作等
	背景图	支持设置背景颜色，背景透明度
	坐标轴	支持设置network3D上是否显示坐标轴和坐标轴文字
	导出图片	支持将Network3D场景导出成位图
	灯光	3D场景中支持环境光，方向光，聚光灯，点光的设置
	镜头	支持透视投射镜头和正投射两种方式。支持设置透视投射镜头的角度，横纵比，最远、最近距离
编辑器	对象建模	用于对服务器，路由器，硬盘，指示灯，桌椅板凳等各种对象进行建模
	拖拽对象	支持拖拽基本体和对象模板
	模板存储	支持保存对象模型，支持本地和云服务器两种存储方式
	模板分类	可对模板的分类，模板库进行管理
	属性管理	支持对3D模型设置相关属性参数，包括大小，贴图，光照强度，自定义属性等设置
	树形显示	支持所有3D对象的树形层次显示
	创建连线	支持任意物体之间创建连线
	对齐方式	支持多种对齐方式，顶部，底部，中间对齐，地板上，地板下对齐
	obj格式文件	支持导入obj格式文件
	dxf格式文件	支持导入autocad格式文件
	房间建模	可对房间的大场景进行建模，支持任意路径的地板，墙单独建模
	房间2D，3D显示	支持对房间的2D和3D场景的切换显示
	房间的标尺显示	支持显示房间的标尺，房间各个点的坐标
	房间的属性管理	可对房间的属性进行管理，如墙，地板的贴图，位置，高度，厚度，各个点的坐标
	插入3D对象	支持billboard，管道，对象模板插入到房间中
	多楼层	支持多楼层的管理
	编辑器基本设置	可以设置是否显示坐标轴，地板，背景色，镜头，灯光的参数
其他功能	动画支持	支持平移，旋转，缩放，告警冒泡的动画
	巡航动画	支持自定义巡航路径，自动巡航动画
	流动效果	支持流动效果的显示，比如管道的水流
	空气流动	支持空气流动效果

性能规格

测试环境

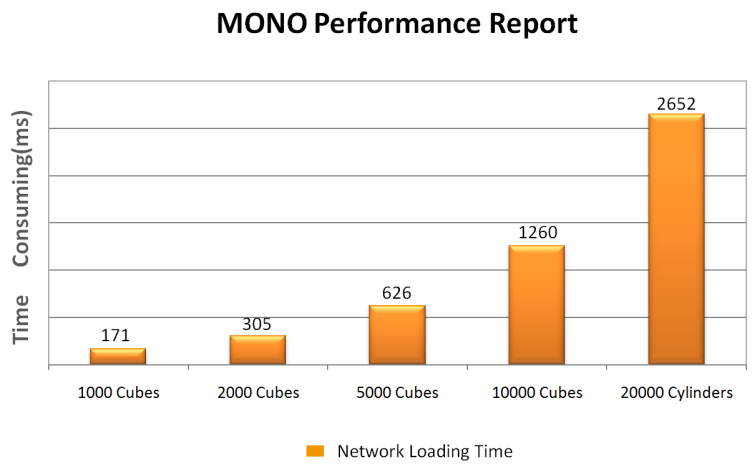
MONO Design 1.9

Google Chrome Version 37.0.2062.124

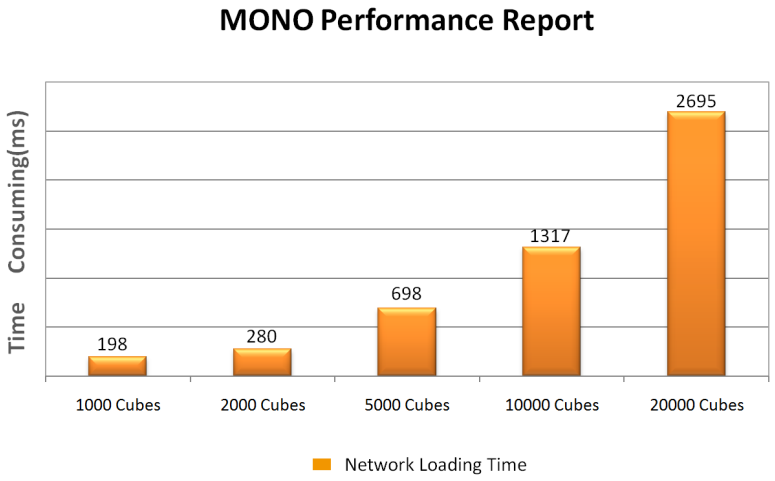
Mac OSX 10.9.3

2.8 GHz Intel Core i7

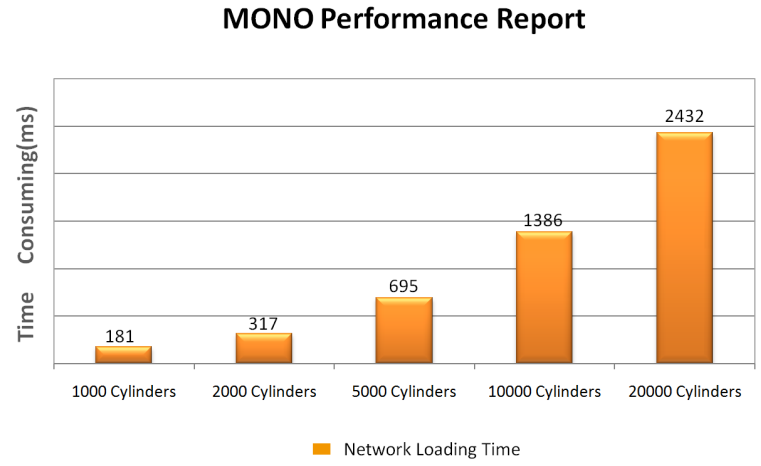
测试一：创建无贴图的立方体



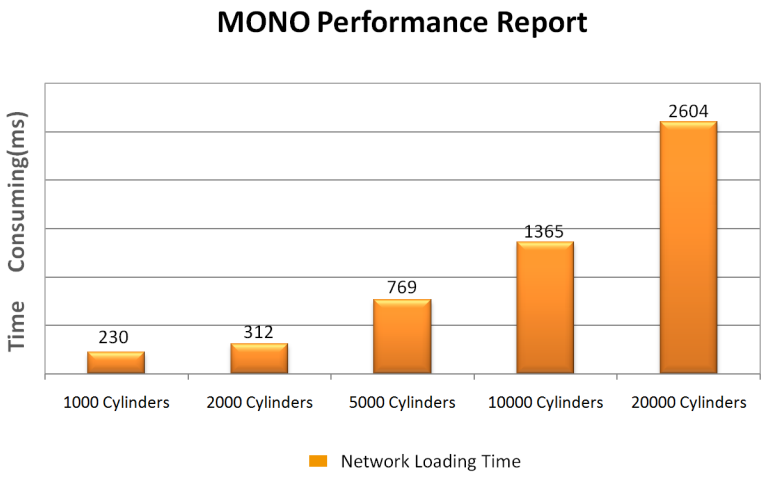
测试二: 创建有贴图的立方体



测试三：创建无贴图的圆柱体(30个面)



测试四：创建有贴图的圆柱体(30个面)



测试五：创建10000个立方体，所占资源内存：

纯模型数据(json)	模型数据带样式(json)	运行占用内存	图片资源占用
1.0 M	2.0 M	50. M	与场景大小有关，没有固定的大小，比如一个场景有50种不同的贴图资源，每种500K算，图片占用大概25M

测试源代码:performance.zip

了解更多文档，请至 TWaver 在线文档中心：http://doc.servasoft.com/

关于 TWaver 和赛瓦软件

赛瓦软件（Serva Software）成立于 2004 年，是美国赛瓦集团（Serva Group）全资子公司、高新技术企业、“双软”认证企业，主要面向中国及亚洲地区提供产品和技术服务。赛瓦软件总部位于美国 Texas 州 Wichita Falls。

赛瓦软件 TWaver 产品是专业的行业 UI 可视化组件产品，提供丰富的 2D/3D 展示能力，支持各种技术平台。TWaver 广泛应用于电信、电力、金融等行业，可以 2D 或 3D 的方式呈现复杂的网络拓扑数据、地图、设备结构图等。爱立信、思科、惠普、华为、中兴、亿阳、浪潮、南瑞等几百家电信软件企业都是 TWaver 的长期用户。

联系我们

赛瓦软件（上海）有限公司

地址：上海市徐汇区中山西路 2025 号 1921 室
邮编：200235
电话：86-21-64398788
传真：86-21-64395374
官方网址：http://servasoft.com

技术热线

TWaver 技术支持邮箱：tw-service@servasoftware.com
TWaver 技术支持电话：86-21-64398788（转 TWaver 技术支持）
TWaver 技术紧急救援：135-6491-6007（Paul）