第 3 章

WebXR 开发基础

3.1 一行代码让网站支持 3D 和 VR

本章为大家讲解 Babylon.js 的开发基础，让读者从头熟悉如何在 Web 页面中创建一个虚拟的三维世界。创建一个虚拟世界，需要一个场景（Scene）并在场景中添加模型（Model），模型有可能是一个简单的立方体块，也可能是一个复杂的角色，无论是简单的模型还是复杂的模型，大都是由 Mesh 网格组成，如图 3.1 所示。

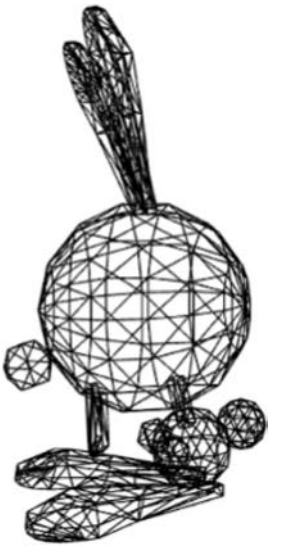


图 3.1 基于 Mesh 网格的三维模型

除此之外，还需一个相机（Camera）去观察虚拟世界，一个灯光（Light）去照亮场景，等等。拥有上述内容后，才能够在 Web 页面观察到一个虚拟的世界，接下来将逐步为大家描述如何实现上述内容。

基于 Web 的 XR 应用（例如 3D、全景、AR、VR 等，或上述四者互相结合的应用）都属于 WebXR 的范畴，相较于基于 C/S 架构的独立应用，WebXR 应用有其不可替代的优势，也存在目前难以消除的缺陷。

WebXR 最大的优势在于，无须额外下载独立应用，通过浏览器（或扫码或单击）即可进行体验，用完即走，不留痕迹。省去用户下载 App 的麻烦之后，对于开发者而言，将面临一个严峻的考验，那就是 XR 应用中有大量的 3D 资源，如何使之轻量化，减少用户等待加载的时间，给用户一个流畅的体验感受？面对这个问题，无论是程序开发者，还是内容生产者，无疑都要花费更多时间在资源和程序的优化上，不过资源的优化工作不在本节讨论范围内，因此不再赘述。

W3C 标准化组织发布的 WebXR Device API（网址请参考本书配套资源）目前可支持一

些主流的 XR 设备，并已被集成至一些常见的 JavaScript 游戏引擎或 3D 图形框架中，例如 A-Frame、Three.js、Babylon.js 等，开发者可以基于这些工具来实现自己的应用。不过目前并非所有的浏览器都已支持 WebXR，浏览器兼容情况如图 3.2 所示，完全支持的浏览器备注了所需的最低版本。

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet
xr	79	79	No	No	No	No	No	79	No	No	No	11.2

完全支持　　不支持

图 3.2　主流浏览器对 WebXR 的兼容情况

可以使用上面提到的 3D 框架来实现部分我们想要的 XR 应用。今天给大家展示的是使用 Babylon.js 来实现网站中展示 3D 模型的案例，案例效果如图 3.3 所示。

图 3.3　一行代码让你的网站支持 3D 和 VR

上述案例中的 3D 展示功能，其核心代码只有下面一行。

```
1. <babylon model = "https://ilab - oss.arvroffer.com/WebXR/course/glb/pepper.glb"></babylon>
```

其中，<babylon></babylon>标签表示将 Babylon.js 中的 Viewer 组件嵌入到页面中，Viewer 是 Babylon.js 内部封装的一个 3D 模型展示组件，model 属性指向了一个.gltf 格式(或.glb)的 3D 模型的 URL，用户可以自定义，完整的页面代码如下所示。

```
1.  <!DOCTYPE html>
2.  <html>
3.  <head>
4.      <meta http - equiv = "Content - Type" content = "text/html; charset = utf - 8" />
5.      <title> 3D Viewer Example </title>
6.      <script src = "babylon.viewer.js"></script>
7.      <link rel = "stylesheet" href = "normalize.min.css" />
8.      <meta name = "viewport" content = "width = device - width, initial - scale = 1"></meta>
9.      <style>
10.         body {
11.             height: 600px;
12.         }
13.
14.         # header {
15.             font - size: 4em;
16.             padding: 5px;
17.         }
18.
19.         .cell {
20.             width:32 % ;
21.             height:60 % ;
22.             margin:8px;
23.             float: left;
24.             padding: 3px;
25.             background - color: # BBBBBB;
26.         }
27.
28.         @media screen and (max - width: 900px) {
29.             body {
30.                 height: unset;
31.             }
32.             .cell {
33.                 width:unset;
34.                 padding: 0;
35.                 font - size:18px;
36.             }
37.
38.             .babylon {
39.                 width:100 % ;
40.                 margin: 0;
```

```
41.              padding: 8px;
42.              box - sizing: border - box;
43.              background: unset;
44.           }
45.        }
46.
47.    </style >
48. </head >
49. < body >
50.      < div id = "header">
51.         WebXR 系列:一行代码让你的网站支持 3D 和 VR.
52.      </div >
53.      < div class = "cell">
54.         < babylon model = "https://ilab - oss.arvroffer.com/WebXR/course/glb/pepper.glb">
</babylon >
55.      </div >
56.      < div class = "cell">
57.         青椒为植物界,双子叶植物纲,合瓣花亚纲,茄科。和红色辣椒统称为辣椒。果实为浆
果。别名很多,大椒、灯笼椒、柿子椒都是它的名字,因能结甜味浆果,又叫甜椒、菜椒。一年生或多
年生草本植物,特点是果实较大,辣味较淡甚至根本不辣,作蔬菜食用而不作为调味料。由于它翠绿
鲜艳,新培育出来的品种还有红、黄、紫等多种颜色,因此不但能自成一菜,还被广泛用于配菜。青椒
由原产中南美洲热带地区的辣椒在北美演化而来,长期栽培驯化和人工选择,使果实发生体积增大、
果肉变厚、辣味消失等性状变化。中国于 100 多年前引入,现全国各地普遍栽培,青椒含有丰富的维
生素 C。
58.      </div >
59.      < div class = "cell">
60.         < babylon model = "https://ilab - oss.arvroffer.com/WebXR/course/glb/pepper2.glb">
61.         </babylon >
62.      </div >
63.      < div class = "cell">
64.         国内甜椒种子较多,表现突出的为进口品种荷椒 13。在山西、山东、内蒙古、吉林、黑龙
江等地均有大面积种植,是出口选型的优良品种。
65. 该品种是国际甜椒种子中优秀品种, 一代杂交种,中早熟,大果形品种,果实方灯笼形。连续坐
果率高,四心室率高。果实整齐度高,味甜,果肉厚 0.8cm 左右,果横径 10～11cm、纵径 12～13cm。果
色较深绿,熟果转红色,果面光亮,单果重 300～400g,最大可达 600g。高产品种,抗各种病害,适应性
强。在国内大部分地区都有种植。果形大,产量高。果形方正,是市场或加工的理想品种。
66.      </div >
67.      < div class = "cell">
68.          < babylon model = " https://ilab - oss. arvroffer. com/WebXR/course/glb/pepper3.
glb">
69.            < vr object - scale - factor = "1">
70.            </vr >
71.            < templates >
72.               < nav - bar >
73.                  < params hide - vr = "false"></params >
74.               </nav - bar >
75.            </templates >
76.         </babylon >
```

```
77.      </div>
78.      <div class = "cell">
79.      温度要求:适应温度范围为 15～35℃ ,适宜的温度范围为 25～28℃ ,发芽温度为 28～30℃ 。
80.      水分条件:喜湿润,怕旱涝,要求土壤湿润而不积水。
81.    光照条件:对光照要求不严,光照强度要求中等,每天光照 10～12h,有利于开花结果。青椒的
生长发育需要充足的营养条件,每生产 1000kg 青椒,需氮 2kg、磷 1kg、钾 1. 45kg,同时还需要适量的
钙肥。对土壤的要求,以潮湿易渗水的沙壤土为好,土壤的酸碱度以中性为宜,微酸性也可。
82.      </div>
83. </body>
84. </html>
```

3.2 场景创建

场景(Scene)表示一个虚拟的场地,一般由环境、房间、道具、角色等共同组成一个虚拟的场景,在有些游戏引擎或 3D 框架中也叫作舞台(Stage)。总之,从 3D 的角度来说,场景就是将一些网格(Mesh)放在一起供用户观看,并且会在其中加入相机(Camera)和灯光(Light)让用户能够看到。当然,除了上述提到的之外,场景可能还会包含一些别的元素,例如 GUI 用户界面,让用户能够与场景产生交互,下面正式开始场景的学习。

3.2.1 快速创建场景

首先,需要打开 Babylon. js 在线开发工具 Playground。可以进入 Babylon. js 官方网站,在首页菜单选项卡中选择 TOOLS→PLAYGROUND 即可进入 Playground,如图 3. 4 所示。

图 3. 4　进入 Babylon. js 的在线开发工具 Playground

Playground 是 Babylon. js 提供的在线编码工具,使得开发者可以直接在网页上实现代码的编写、调试、下载等功能,如图 3. 5 所示。

Playground 的开发界面分为顶部、左半部分和右半部分,顶部的主要的功能如下。

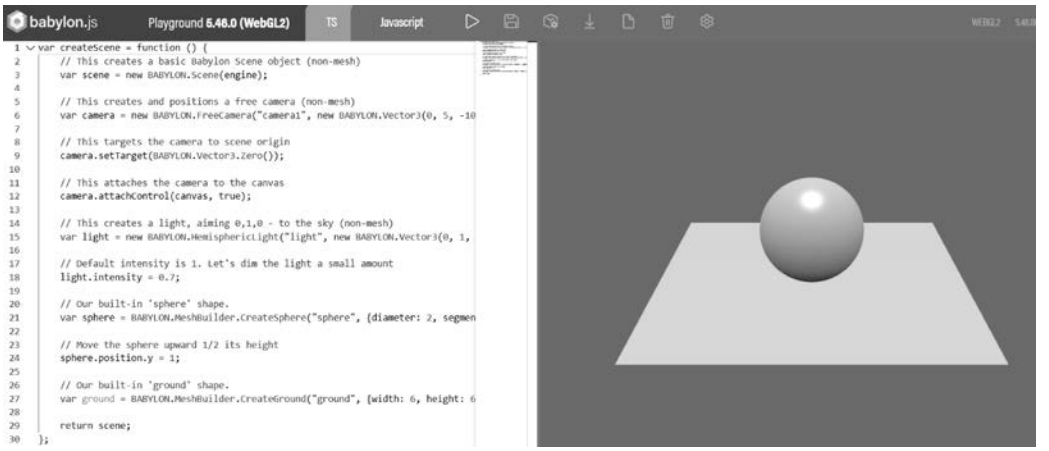


图 3.5　使用在线开发工具编辑代码

1. 语言切换

Babylon.js 支持 TypeScript(简称 TS)和 JavaScript(简称 JS)两种语言开发模式的切换,默认情况下选择的是 JavaScript。如果大家喜欢采用 TS 开发,则可以切换至 TS 选项下进行代码的编写工作。如无特殊说明,本书所有的案例代码实现都是采用 JS 编写。如果要进行编程语言的切换,只需要单击 TS 按钮即可,如图 3.6 所示。

图 3.6　编程语言切换

2. 运行

单击运行按钮,则开始执行当前场景中的脚本,并在画面右半部分显示运行结果,如图 3.7 所示。

图 3.7　运行按钮

3. 保存

由于 Playground 是基于 Web 的,因此单击保存按钮并不会将当前的工程保存至本地,而是会保存至官方服务器,如图 3.8 所示。

图 3.8　保存按钮

开发者单击保存按钮后,会弹出对话框提示用户输入一些与项目相关的信息,如图 3.9 所示。

图 3.9　输入项目工程信息

输入完成后单击 OK 按钮，则会在服务器端生成一个代表当前工程的唯一 ID，用户只需要记录页面的 ID 即可随时进入工程继续编写，如图 3.10 所示。

图 3.10　基于用户 ID 的项目工程识别机制

图 3.10 中的＃R48MTP＃3 代表当前工程的 ID，第一个＃后面为项目 ID，第二个＃后面为版本号，也就是说，用户对网页进行修改之后，再次单击保存按钮或按下 Ctrl＋S 快捷键，则会保存一个新的版本。

4. Inspector（信息监视）面板

如图 3.11 所示，单击最上方工具栏中的 Inspector 按钮，可以打开 Inspector 面板。

图 3.11　工具栏中的 Inspector 按钮

打开该面板后，首先可以看出主窗体分为了 3 部分：最左侧是代码区，中间是场景区，最右侧才是打开的 Inspector 面板区。在 Inspector 面板区域中可以看到场景的节点信息、材质信息、动画信息等，这些信息有助于开发者进行调试。该功能在场景中有较多元素时比

较有用。在图 3.12 中可以看到屏幕右侧的 INSPECTOR 面板中列出了 Nodes(节点)列表、Materials(材质)列表、Textures(纹理)列表等。

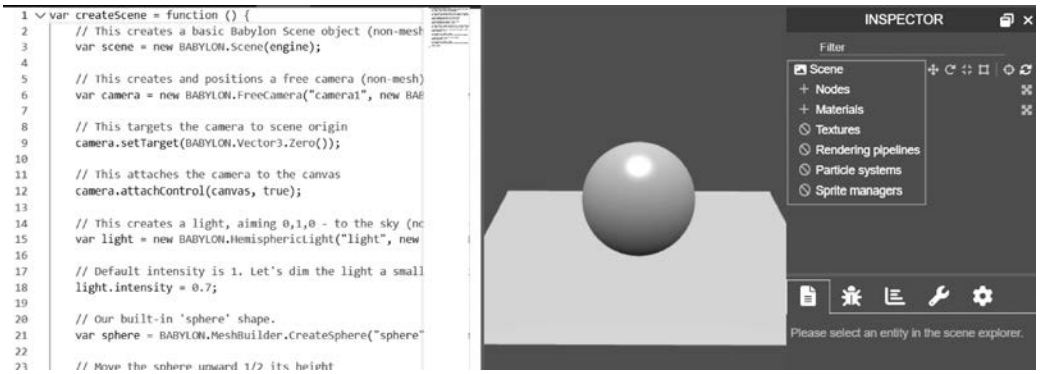


图 3.12 Inspector 组件面板

例如,单击 Scene 下的 Nodes 节点前面的“+”进行节点的展开,可以看到当前场景中的所有物体对象,可以看到,在默认情况下,场景中存在一个相机 camera1、一个平面 ground、一个灯光 light 和一个球体 sphere,如图 3.13 所示。

图 3.13 在 Inspector 面板中查看详细的物体对象

5. 下载

单击下载按钮,如图 3.14 所示,可以将当前的代码打包下载至本地,因为开发的 Web 应用在大部分情况下都需要进行独立的部署或交付,因此需要将页面下载至本地,Babylon 会将整个 HTML 页面进行下载。

图 3.14 项目下载按钮

6. 新建

单击新建按钮即可新建一个项目,如图 3.15 所示。

图 3.15　新建按钮

7. 清除

单击清除代码按钮，如图 3.16 所示，即可清除当前页面上的代码。

图 3.16　清除代码按钮

8. 设置

单击设置按钮，如图 3.17 所示，即可对项目进行设置，包括页面的主题、字体大小、全屏设置等。

图 3.17　设置按钮

熟悉了 Playground 开发环境后，接下来开始编写代码来快速创建场景，在场景中添加一个 Box 立方体。

```
1.  var createScene = function() {
2.      var scene = new BABYLON.Scene(engine);
3.      var box = BABYLON.MeshBuilder.CreateBox("box", {});
4.      scene.createDefaultCameraOrLight(true, true, true);
5.      scene.createDefaultEnvironment();
6.      return scene;
7.  };
```

上述代码的执行结果如图 3.18 所示。

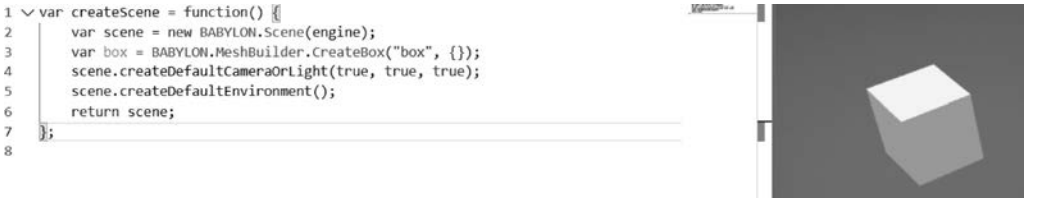


图 3.18　案例执行效果

3.2.2　场景创建 API 说明

通过上述操作，我们发现右侧 Scene 场景创建了一个立方体，同时用户可以使用鼠标拖动让立方体进行旋转，按住鼠标右键可以移动立方体。下面介绍通过上述代码创建一个基础场景的步骤。

1. 创建场景

在 HTML 中调用 createScene()函数,并对场景进行渲染。该函数是 Playground 预留的需要开发者自己进行实现的一个函数。开发者不需要对函数的名字、返回值等进行修改,只需要在函数体中添加对应的代码即可。

2. 初始化空场景

使用下述代码初始化一个空场景:

```
var scene = new BABYLON.Scene(engine);
```

3. 创建默认的摄像头和灯光

使用函数 createDefaultCameraOrLight()创建默认的摄像头和灯光。当创建了 scene 对象后,就可以调用 scene 中的接口函数对场景进行一系列的操作。对于函数 createDefaultCameraOrLight()来说,从函数的名称可知这个函数完成了两件事情,即创建相机和创建灯光,说明该函数并不是一个功能单一的函数,而是一个合并后为了用户方便使用的一个复合函数,可以通过其 API 文档来查看该函数都有哪些参数,这些参数的作用是什么,如图 3.19 所示。

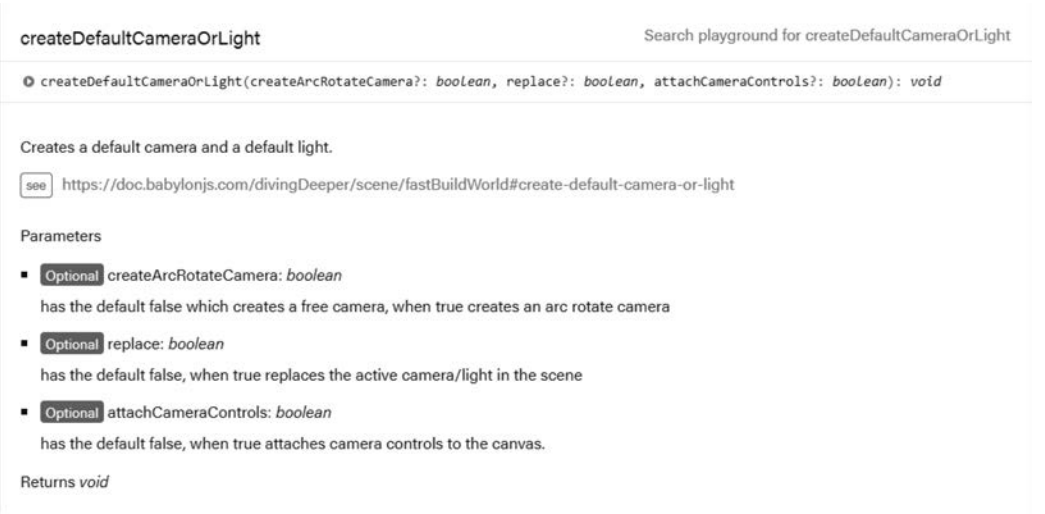


图 3.19 查看函数的 API 文档

由图 3.19 可以看到该函数包含 3 个参数,均为 boolean 类型,依次为:

(1) 是否创建 ArcRotateCamera(该相机会在后面专门介绍);

(2) 是否替换当前场景中的 camera 或 light;

(3) 是否允许控制相机。

从上述代码中可以看到本次传递的参数都是 true。

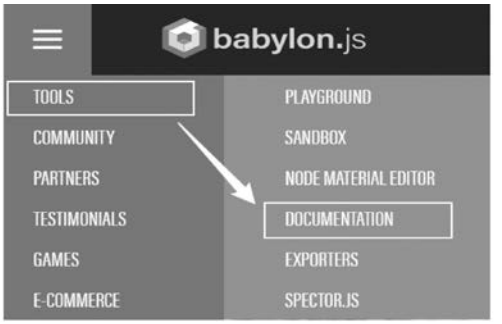
4. 创建物体对象

通过 MeshBuilder.createBox()方法创建一个立方体,并添加至场景中。

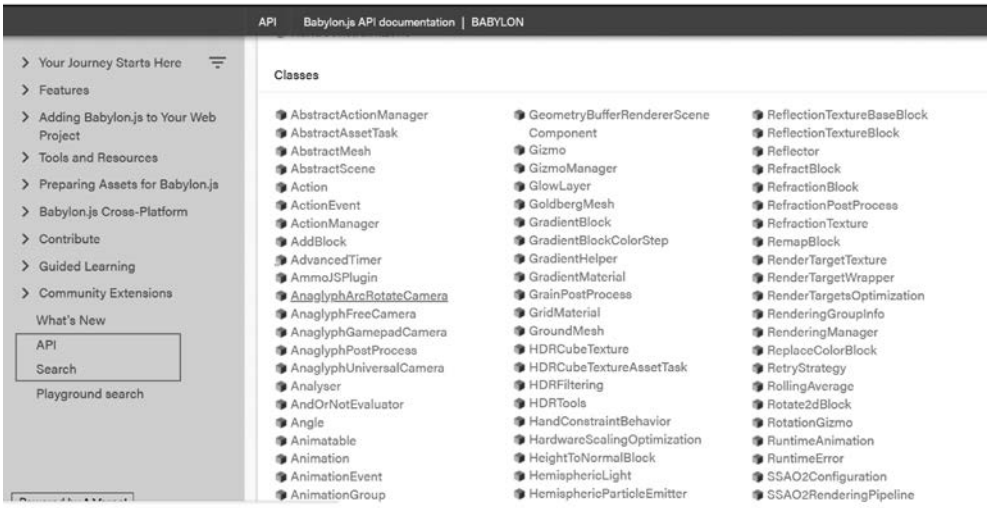
5. 创建默认环境

createDefaultEnvironment()函数用来创建默认环境，Babylon.js 默认的环境包含了一个圆形的地面，用户对于环境的设置可以通过传入配置参数来实现。后续章节将进一步讲解环境的设置。

要查看函数的 API 文档，可以在 Babylon.js 官网首页选择 TOOLS→DOCUMENTATION，然后选择左侧的 API 选项，在右侧的窗口中就可以看到要查询的命名空间、类、变量、接口、函数等，也可以在左侧单击 Search 选项进行指定的搜索，如图 3.20 所示。



(a) 选择DOCUMENTATION选项



(b) API接口查询

图 3.20　访问 API 文档

3.3　场景灯光

3.2 节介绍了场景的快速创建方法，其中调用了场景中提供的一个接口来创建相机和灯光，实际上可以由开发者自己来控制创建灯光的数量，以及创建何种灯光。灯光在场景中

非常重要，会直接影响到场景中创建的 Mesh 的显示效果，包括照明和颜色等。在 Babylon.js 中，场景默认允许创建的灯光数量为 4 个，但是也可以根据开发者的需求增加。图 3.21 展示了一个球体在受到多个灯光照射后的显示效果。

图 3.21　场景灯光照射效果

3.3.1　灯光的类型

在 Babylon.js 中提供了 4 种类型的灯光，每种灯光都有自己的特点和适用的场景，接下来逐一进行介绍。

1. 点光源(Point Light)

点光源是能够模拟在三维空间中由一个点向四周散射的一种光源，类似于生活中的电灯泡，光会从各个方向进行传播。创建点光源的示例代码如下：

```
1.  var light = new BABYLON.PointLight("pointLight", new BABYLON.Vector3(1, 10, 1), scene);
```

2. 方向光(Directional Light)

方向光是指光从指定的一个方向发射而来，并且具有无限的范围，也就是说，方向光会沿着某个方向照亮场景中的所有区域，可以想象方向光类似于太阳光，但是太阳光是有固定方向的，比如地球上的太阳光，自然是从太阳发射出来的那个方向的光。方向光的创建方式如下：

```
1. var light = new BABYLON.DirectionalLight("DirectionalLight", new BABYLON.Vector3(0, -1,
0), scene);
```

3. 聚光灯(Spot Light)

聚光灯是由位置、方向、角度和指数定义的光源类型，表示一个从某位置朝着某方向发射的一束光锥，类似于生活中使用的手电筒，其中有两个参数值得特别说明：一个是角度，聚光灯的角度表示光锥的大小，其单位是以弧度表示；另一个是指数，指数定义了光随着距离传播时衰减的速度。这里与方向光不同，方向光并不会随着距离衰减，而聚光灯可以定义衰减的指数。聚光灯的创建方式如下：

```
1. var light = new BABYLON.SpotLight("spotLight", new BABYLON.Vector3(0, 30, -10), new
BABYLON.Vector3(0, -1, 0), Math.PI / 3, 2, scene);
```

4. 半球光(Hemispheric Light)

半球光是一个模拟环境灯光的简单方法，半球光定义一个方向，通常向上朝向天空，通过设置颜色等属性才能实现完整的效果。

```
1.  var light = new BABYLON.HemisphericLight("HemiLight", new BABYLON.Vector3(0, 1, 0),
scene);
```

上述介绍的4种类型的灯光，其颜色属性都包括自发光、漫反射和镜面反射。重叠的灯光会与我们期望的那样相互作用，例如，红、绿、蓝3种颜色的光重叠会产生白光，每一盏灯都可以打开、关闭，每个灯光的强度也可以设置为0～1的值。

3.3.2 灯光颜色的设置

灯光的3个属性会影响颜色，首先是ground color(地面颜色)，仅适用于半球光；另外两种颜色的属性分别为：

(1) diffuse color——漫反射颜色，漫反射为对象提供基本的颜色；

(2) specular color——高光，高光使对象上产生高光斑点或亮斑。

图3.22 添加多种灯光后的球体

上述两种颜色属性运用于所有4种类型的灯光。

如图3.22展示了一个方向光照亮一个球体，方向光的漫反射颜色为红色，高光颜色为绿色，最终可以看到在右侧三维场景中一个红色的球体，以及绿色的亮斑。

实现该效果的相关代码如下：

```
1.  var createScene = function () {
2.    var scene = new BABYLON.Scene(engine);
3.    var camera = new BABYLON.ArcRotateCamera("Camera", - Math.PI / 2,  Math.PI / 2, 5,
BABYLON.Vector3.Zero(), scene);
4.    camera.attachControl(canvas, true);
5.    var light = new BABYLON.DirectionalLight("DirectionLight",new BABYLON.Vector3(0, - 1,
0),scene);
6.    light.diffuse = new BABYLON.Color3(1,0,0);
7.    light.specular = new BABYLON.Color3(0,1,0);
8.    var sphere = BABYLON.MeshBuilder.CreateSphere("sphere",{},scene);
9.    return scene;
10. };
```

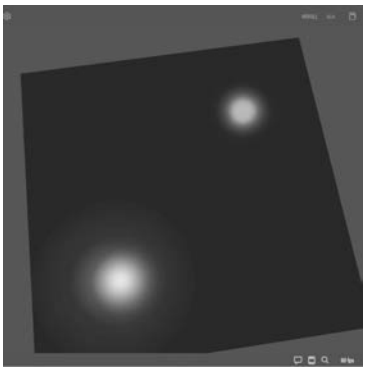


图3.23 两个聚光灯的照射效果

如图3.23所示，展示两个聚光灯的照射效果，其中一个聚光灯的漫反射颜色和高光颜色均为绿色，因此无法明显地看到其亮斑在什么位置；另一个聚光灯打出的漫反射颜色为红色，高光颜色为绿色，可以看到左下角明显的亮斑，而且亮斑的颜色为红色与绿色经过叠加之后的颜色，即黄色，这也验证了前面提到的多个颜色混合的效果。

下面通过另一段代码展示光的颜色之间的混合以及混合的结果。

```
1.  var createScene = function () {
2.      var scene = new BABYLON.Scene(engine);
3.      var camera = new BABYLON.ArcRotateCamera("Camera", -Math.PI / 2,  Math.PI / 4, 5,
BABYLON.Vector3.Zero(), scene);
4.      camera.attachControl(canvas, true);
5.
6.      //红色光
7.      var light = new BABYLON.SpotLight("spotLight", new BABYLON.Vector3(-Math.cos(Math.
PI/6), 1 , -Math.sin(Math.PI/6)), new BABYLON.Vector3(0, -1, 0), Math.PI / 2, 1.5, scene);
8.      light.diffuse = new BABYLON.Color3(1, 0, 0);
9.
10.     //绿色光
11.     var light1 = new BABYLON.SpotLight("spotLight1", new BABYLON.Vector3(0, 1, 1 - Math.
sin(Math.PI / 6)), new BABYLON.Vector3(0, -1, 0), Math.PI / 2, 1.5, scene);
12.     light1.diffuse = new BABYLON.Color3(0, 1, 0);
13.
14.     //蓝色光
15. var light2 = new BABYLON.SpotLight("spotLight2", newBABYLON.Vector3(Math.cos(Math.PI/
6), 1, -Math.sin(Math.PI/6)), new BABYLON.Vector3(0, -1, 0), Math.PI / 2, 1.5, scene);
16.     light2.diffuse = new BABYLON.Color3(0, 0, 1);
17.
18.     var ground = BABYLON.MeshBuilder.CreateGround("ground", {width: 4, height: 4}, scene);
19.
20.     return scene;
21.
22. };
```

上述代码创建了一个地面的 Mesh，同时创建了 3 个聚光灯来照射地面，这 3 个聚光灯的漫反射颜色分别为红、绿、蓝（对光学有基本认识的读者一定知道这 3 个颜色代表什么），这 3 束光通过位置的设置而两两相交，最终得到如图 3.24 所示的效果。

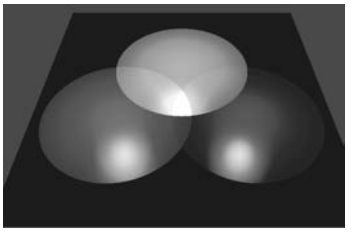


图 3.24　3 束光两两相交的效果

3.3.3　灯光开关和调光器

1. 灯光的开关

每一种灯光都可以通过代码打开或者关闭，具体打开或者关闭的实现方式如下：

```
1.  light.setEnabled(false);
```

给上述代码传入 true 表示打开灯光，传入 false 表示关闭灯光。

2. 灯光的强度

当想要调亮或者调暗灯光的时候，可以设置灯光的 indensity 属性，赋值越大，灯光越强；反之亦然。默认情况下，该属性的取值为 1。

```
1.  light0.intensity = 0.5;
2.  light1.intensity = 2.4;
```

3. 灯光的范围

对于点光源和聚光灯，可以设置 range 属性来限制灯光能够到达的距离。

3.4 场景阴影

在开始介绍本节具体内容之前，先看一下图 3.25。图 3.25 中的场景包含一个类似于丘陵的地面地形、一个运动的圆环和地形上方的两盏灯。最后圆环在运动的过程中，会实时、动态地在地面上投射出阴影效果。

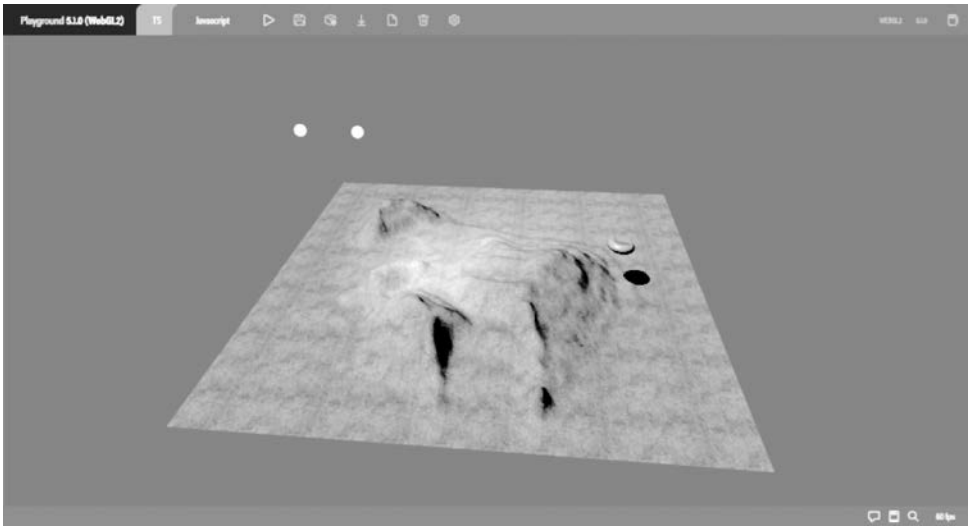


图 3.25 场景阴影效果

图 3.25 中的阴影效果如何实现，就是本节要介绍的内容。

3.4.1 阴影生成

在 Babylon.js 中使用阴影生成器(Shadow Generator)很容易生成阴影。阴影生成器是一个能够从灯光的角度来生成阴影纹理的工具。阴影生成器有两个参数，分别为阴影纹理的大小以及用于阴影纹理计算的光源。下面给出创建阴影的示例代码。

```
1.  var shadowGenerator = new BABYLON.ShadowGenerator(1024, light);
```

然后必须要定义渲染的阴影，比如想要渲染圆环的阴影，那就可以向阴影纹理的渲染列表中添加圆环，代码如下：

```
1.  shadowGenerator.getShadowMap().renderList.push(torus);
```

最后，必须要定义阴影显示的位置，将其接收阴影的属性设置为 true。

```
1.  ground.receiveShadows = true;
```

满足上述3个条件后，就可以在场景中看到阴影了。要想让阴影看起来效果更好，可以激活阴影过滤功能，通过去除阴影的硬边来获得一个更好看的阴影。

Babylon.js提供了3种过滤器，可以选择任何一种进行尝试。

1. 泊松采样过滤

```
1.  shadowGenerator.usePoissonSampling = true;
```

2. 指数阴影纹理

```
1.  shadowGenerator.useExponentialShadowMap = true;
```

3. 模糊指数阴影纹理

```
1.  shadowGenerator.useBlurExponentialShadowMap = true;
```

通过上述方式能够实现软阴影的效果，当然这种阴影的效果会更加消耗渲染资源，软阴影的效果如图3.26所示。

3.4.2 透明物体和阴影

要让透明的物体投射阴影，必须在阴影生成器上打开transparentShadow属性，如图3.27所示为透明物体投射阴影的效果。

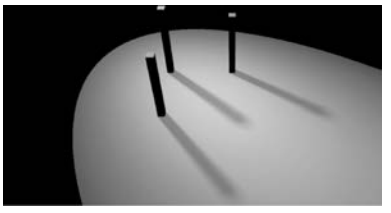


图3.26 软阴影效果

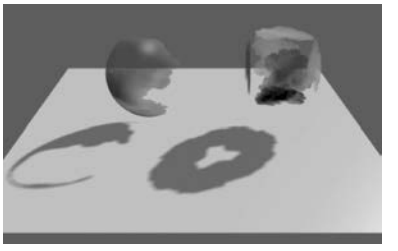


图3.27 透明物体投射阴影效果

3.4.3 灯光与阴影的关系

灯光与阴影可以说是既互相独立又紧密联系的，没有灯光就无法生成阴影，但是在生成阴影的过程中，有一些规则和需要特别注意的地方。

(1) 只有点光源(Point Light)、方向光(Directional Light)、聚光灯(Spot Light)可以投射阴影，其他的光源是无法投射阴影的。

(2) 一个阴影生成器只能供一个灯光使用，如果多个灯光都会投射阴影，则需要分别为每个灯光创建一个阴影生成器。

(3) 点光源使用立方体纹理Cubemaps来渲染，因此在使用点光源的时候需要注意性能问题。

(4) 聚光灯使用透视投影来计算阴影纹理。

(5) 定向灯使用正交投影。自动评估灯光的位置,以获得最佳的阴影纹理,可以通过关闭 light.autoUpdateExtends 来控制此行为。

3.4.4 体积光散射后处理

BABYLON.VolumetricLightScatteringPostProcess 是一个后处理功能,它将根据光源网格计算光散射,具体的使用方法如下:

```
1.    var vls = new BABYLON.VolumetricLightScatteringPostProcess('vls', 1.0, camera,
lightSourceMesh, samplesNum, BABYLON.Texture.BILINEAR_SAMPLINGMODE, engine, false);
```

上述接口中每一个参数的含义如下:

(1) name {string}——后处理名称。

(2) ratio {any}——后处理和/或内部通道的大小(0.5 意味着后处理将具有宽度=canvas.width×0.5 和高度=canvas.height×0.5)。

(3) camera {BABYLON.Camera}——后处理将附加到的相机。

(4) lightSourceMesh{BABYLON.Mesh}——用作光源的网格,以创建光散射效果(例如,具有模拟太阳纹理的广告牌)。

(5) samplesNum{number}——后处理质量,默认值为 100。

(6) samplingMode{number}——后处理过滤模式。

(7) engine{BABYLON.Engine}——巴比伦引擎。

(8) reusable{boolean}——如果在后处理中需要重用,那么就将其值设置为 true。

(9) 场景{BABYLON.Scene}——如果相机参数为空(在渲染管道中添加后处理),则需要场景来配置内部通道。

通过体积光散射后处理,可以实现如图 3.28 所示的光照效果。

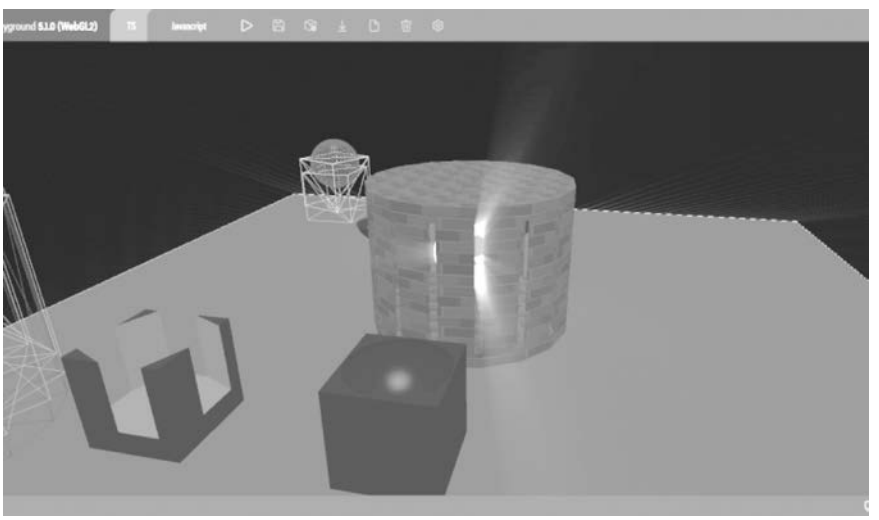


图 3.28 体积光散射后处理效果

3.5 场景交互

3.5.1 如何在场景中进行交互

交互在软件中必不可少，尤其在三维场景中的交互则显得更加重要。目前，在很多 XR 设备中出现了新型的、更加自然化的交互，例如手柄的交互、触摸、眼球跟踪、收拾识别、动作识别等，交互的类型多样，交互的方式也发生了翻天覆地的变化。本节还是围绕三维场景中主要的两种交互——键盘和鼠标的交互来展开。当然，场景交互还支持 GUI、游戏手柄、触摸屏等其他形式。

3.5.2 键盘的交互

下面的代码展示了在场景中如何监听键盘输入的事件。

```
scene.onKeyboardObservable.add((kbInfo) => {
  switch (kbInfo.type) {
    case BABYLON.KeyboardEventTypes.KEYDOWN:
      console.log("KEY DOWN: ", kbInfo.event.key);
      break;
    case BABYLON.KeyboardEventTypes.KEYUP:
      console.log("KEY UP: ", kbInfo.event.code);
      break;
  }
});
```

键盘输入的枚举类型包含 KEYDOWN 和 KEYUP 两种事件，用于监听按键按下和按键抬起，实际上在很多引擎中还会额外提供控制按键按住时间的属性，当然在这里可以自己实现按住事件的逻辑。

3.5.3 鼠标的交互

下面的代码展示了如何在场景中监听鼠标事件。

```
scene.onPointerObservable.add((pointerInfo) => {
  switch (pointerInfo.type) {
    case BABYLON.PointerEventTypes.POINTERDOWN:
      console.log("POINTER DOWN");
      break;
    case BABYLON.PointerEventTypes.POINTERUP:
      console.log("POINTER UP");
      break;
    case BABYLON.PointerEventTypes.POINTERMOVE:
      console.log("POINTER MOVE");
      break;
```

```
12.      case BABYLON.PointerEventTypes.POINTERWHEEL:
13.        console.log("POINTER WHEEL");
14.        break;
15.      case BABYLON.PointerEventTypes.POINTERPICK:
16.        console.log("POINTER PICK");
17.        break;
18.      case BABYLON.PointerEventTypes.POINTERTAP:
19.        console.log("POINTER TAP");
20.        break;
21.      case BABYLON.PointerEventTypes.POINTERDOUBLETAP:
22.        console.log("POINTER DOUBLE-TAP");
23.        break;
24.    }
25. });
```

上述鼠标事件包含鼠标按下、鼠标抬起、鼠标移动、鼠标滚轮、鼠标选取、鼠标单击以及鼠标双击事件，鼠标事件与键盘事件有所不同，因为键盘没有位置信息，每个键盘包含了一个键盘码，而鼠标还包含位置信息。因此，在监听给定事件的时候，还可以得到 pointerInfo 这样一个数据结构，然后在这个数据结构中获取想要的事件信息。

3.6 相机

在 Babylon.js 的众多相机中，最为常用的应该是用于第一人称运动的通用相机 UniversalCamera 和轨道相机 ArcRotateCamera，以及用于虚拟现实体验的 WebXRCamera。

3.6.1 通用相机

通用相机是在 Babylon.js 2.3 引入的，用于由键盘、鼠标、触摸屏、游戏控制器等进行控制，具体取决于用户到底在使用哪种输入设备。通用相机的支持将会取代 FreeCamera、TouchCamera 以及 GamepadCamera，因为通用相机对这几类相机进行了集成，但上述几种相机依然是可以使用的。

目前通用相机是 Babylon.js 的默认相机，如果要在场景中使用类似于 FPS 的控制功能，则可以使用该相机，Babylon.js 官网演示的案例大多使用该相机。如果将 XBox 控制器插入 PC，则可以使用大部分该相机的演示功能。

通用相机的默认操作是：

(1) 键盘——通过左右方向键进行左右移动相机，通过上下方向键前后移动相机。

(2) 鼠标——以相机为原点围绕轴旋转相机。

(3) 触摸——左右滑动可左右移动相机，上下滑动可前后移动相机。

(4) 鼠标滚轮——鼠标上的滚轮或触摸板上的滚动动作。

那么如何去构建一个通用相机呢？下面就是一个创建和使用通用相机的例子，请读者自行执行并观察效果。

```
var createScene = function () {
    //创建基本的 Babylon 场景对象
    var scene = new BABYLON.Scene(engine);

    //创建并定位一个通用相机
     var camera = new BABYLON.UniversalCamera("UniversalCamera", new BABYLON.Vector3(0, 5, -10), scene);

    //启用鼠标滚轮的输入
    camera.inputs.addMouseWheel();

    //通过鼠标滚轮 Y 轴的输入来控制相机在场景中的高度(根据实际情况启用或者禁用)
    //camera.inputs.attached["mousewheel"].wheelYMoveRelative = BABYLON.Coordinate.Y;

    //反转鼠标 Y 轴的朝向
    // camera.inputs.attached["mousewheel"].wheelPrecisionY = -1;

    //定位相机在场景中的初始位置(坐标原点)
    camera.setTarget(BABYLON.Vector3.Zero());

    //将相机附加到画布
    camera.attachControl(true);

//创建一个半球光,坐标为(0,1,0),添加至场景中
    var light = new BABYLON.HemisphericLight("light", new BABYLON.Vector3(0, 1, 0), scene);

    //默认灯光强度为 1,让我们把灯光调暗一点
    light.intensity = 0.7;

    //创建一个球体
    var sphere = BABYLON.MeshBuilder.CreateSphere("sphere",{diameter: 2, segments: 32}, scene);

    //将球体向上移动其高度的 1/2
    sphere.position.y = 1;

    //场景中添加一个 ground 地面(宽度和高度均为 6)
    var ground = BABYLON.MeshBuilder.CreateGround("ground", {width: 6, height: 6}, scene);

    return scene;

};
```

3.6.2 轨道相机

轨道相机始终会朝着给定的目标位置运动,并且可以以目标为中心旋转。开发者可以使用光标和鼠标来控制相机,也可以使用触摸事件来控制。可以将这个相机想象成是一个围绕着地球运行的卫星(这是为何被叫作轨道相机的原因),它相对于“地球”的位置可以通

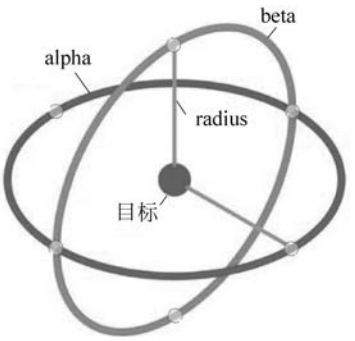


图 3.29　轨道相机工作原理

过 3 个参数来进行设置。

(1) alpha：纵向旋转，以弧度为单位。

(2) beta：维度旋转，以弧度为单位。

(3) radius：半径，指相机到目标的距离。

图 3.29 展示了轨道相机的工作原理。

由于一些技术原因，将 beta 的值设置为 0 或 PI 可能会导致一些问题，因此在这种情况下，beta 会偏移 0.1 弧度。其中 beta 的值为顺时针增加，而 alpha 的值为逆时针方向增加。

当然也可以通过向量 Vector 来设置相机的位置，该值会自动覆盖 alpha、beta 以及 radius 的值，这种方式比计算所需角度容易很多。在用户交互时，无论是使用键盘、鼠标还是滑动，左右方向的操作都会改变 alpha 的值，而上下方向的操作为改变 beta 的值。下面的例子具体展示了如何构造和使用一个轨道相机。

```
1.  var createScene = function () {
2.
3.      //创建一个基本的 Babylon 场景对象
4.      var scene = new BABYLON.Scene(engine);
5.
6.  /********** 轨道相机案例 **************************/
7.
8.      //创建轨道相机添加至场景中
9.      var camera = new BABYLON.ArcRotateCamera("Camera", 0, 0, 10, new BABYLON.Vector3(0,
0, 0), scene);
10.
11.     //设置相机的位置
12.     camera.setPosition(new BABYLON.Vector3(0, 0, -10));
13.
14.     //将相机附加至画布中
15.     camera.attachControl(canvas, true);
16.
17.     /*************************************************************/
18.
19. //创建一个半球光，坐标为(0,1,0)，添加至场景中
20.     var light = new BABYLON.HemisphericLight("light", new BABYLON.Vector3(0, 1, 0), scene);
21.
22.     //材质的设置
23.     var redMat = new BABYLON.StandardMaterial("red", scene);
24.     redMat.diffuseColor = new BABYLON.Color3(1, 0, 0);
25.     redMat.emissiveColor = new BABYLON.Color3(1, 0, 0);
26.     redMat.specularColor = new BABYLON.Color3(1, 0, 0);
27.
28.     var greenMat = new BABYLON.StandardMaterial("green", scene);
29.     greenMat.diffuseColor = new BABYLON.Color3(0, 1, 0);
30.     greenMat.emissiveColor = new BABYLON.Color3(0, 1, 0);
```

```
    greenMat.specularColor = new BABYLON.Color3(0, 1, 0);

    var blueMat = new BABYLON.StandardMaterial("blue", scene);
    blueMat.diffuseColor = new BABYLON.Color3(0, 0, 1);
    blueMat.emissiveColor = new BABYLON.Color3(0, 0, 1);
    blueMat.specularColor = new BABYLON.Color3(0, 0, 1);

    //添加一个平面对象,并附加材质
    var plane1 = BABYLON.MeshBuilder.CreatePlane("plane1", {size: 3, sideOrientation:
BABYLON.Mesh.DOUBLESIDE}, scene);
    plane1.position.x = -3;
    plane1.position.z = 0;
    plane1.material = redMat;

    var plane2 = BABYLON.MeshBuilder.CreatePlane("plane2", {size: 3, sideOrientation:
BABYLON.Mesh.DOUBLESIDE});
    plane2.position.x = 3;
    plane2.position.z = -1.5;
    plane2.material = greenMat;

    var plane3 = BABYLON.MeshBuilder.CreatePlane("plane3", {size: 3, sideOrientation:
BABYLON.Mesh.DOUBLESIDE});
    plane3.position.x = 3;
    plane3.position.z = 1.5;
    plane3.material = blueMat;

     var ground = BABYLON.MeshBuilder.CreateGround("ground1", {width: 10, height: 10,
subdivisions: 2}, scene);

    return scene;

};
```

上述案例的执行结果如图 3.30 所示。

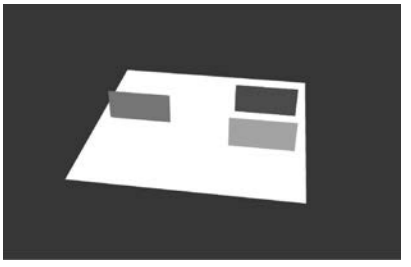


图 3.30　案例实现效果

3.6.3　跟随相机

顾名思义,跟随相机(Follow Camera)需要一个目标网格,相机的位置将会跟随网格的

位置移动至目标的相对位置，当目标物体移动时，相机也会跟随移动。相机主要受以下 3 个参数的控制。

（1）radius：半径，指相机与目标物体（模型）的距离。

（2）heightOffset：相对于目标上方的高度。

（3）rotationOffset：在 *XOY* 平面上，目标旋转的角度。

可通过设置加速度将相机移动到目标的速度设置为最大。下面的代码示例展示了如何使用跟随相机。

```
var createScene = function () {

    //创建一个基本的 Babylon 场景
    var scene = new BABYLON.Scene(engine);

/********** 跟随相机举例 **************************/

    //创建一个跟随相机，并且定位初始位置
    var camera = new BABYLON.FollowCamera("FollowCam", new BABYLON.Vector3(0, 10, -10), scene);

    //相机与目标模型的距离
    camera.radius = 30;

    //相机高度与目标模型的高度差
    camera.heightOffset = 10;

    //相机在 xoy 平面内围绕目标的局部原点旋转值
    camera.rotationOffset = 0;

    //相机从当前位置移动到目标位置的加速度
    camera.cameraAcceleration = 0.005

    //最大加速值
    camera.maxCameraSpeed = 10

    //在此处设置相机的目标网格（启用或者禁用）
    //camera.target = targetMesh;
    //将相机附加到画布中
    camera.attachControl(canvas, true);

    /*****************************************************************/

 //创建一个半球光，坐标为(0,1,0)，添加至场景中
    var light = new BABYLON.HemisphericLight("light", new BABYLON.Vector3(0, 1, 0), scene);

    //材质（准备精灵图像所需的图片）
    var mat = new BABYLON.StandardMaterial("mat1", scene);
    mat.alpha = 1.0;
    mat.diffuseColor = new BABYLON.Color3(0.5, 0.5, 1.0);
    var texture = new BABYLON.Texture("https://i.imgur.com/vxH5bCg.jpg", scene);
```

```
41.     mat.diffuseTexture = texture;
42.
43.     //立方体的每一侧都有不同的面,以显示相机旋转
44.     var hSpriteNb =  3;                  // 精灵图像水平 3 列
45.     var vSpriteNb =  2;                  // 精灵图像垂直 2 行
46.
47.     var faceUV = new Array(6);
48.
49.     for (var i = 0; i < 6; i++) {
50.        faceUV[i] = new BABYLON.Vector4(i/hSpriteNb, 0, (i+1)/hSpriteNb, 1 / vSpriteNb);
51.     }
52.
53.     //通过上述的精灵图像创建立方体并赋予材质
54.     var box = BABYLON.MeshBuilder.CreateBox("box", {size: 2, faceUV: faceUV }, scene);
55.     box.position = new BABYLON.Vector3(20, 0, 10);
56.     box.material = mat;
57.
58.     //创建固体粒子系统,以显示立方体和相机的运动
59.     var boxesSPS = new BABYLON.SolidParticleSystem("boxes", scene, {updatable: false});
60.
61.     //设置立方体粒子位置函数
62.     var set_boxes = function(particle, i, s) {
63.        particle.position = new BABYLON.Vector3(-50 + Math.random()*100, -50 + Math.
random()*100, -50 + Math.random()*100);
64.     }
65.
66.     //添加 400 个立方体
67.     boxesSPS.addShape(box, 400, {positionFunction:set_boxes});
68.     var boxes = boxesSPS.buildMesh();      // 立方体的 mesh
69.
70.   /*****************为相机设置目标************************/
71.     camera.lockedTarget = box;
72.     /*************************************************************/
73.
74.
75.     //立方体移动变量
76.     var alpha = 0;
77.     var orbit_radius = 20
78.
79.
80.     //移动立方体让相机跟随它
81.     scene.registerBeforeRender(function () {
82.      alpha += 0.01;
83.      box.position.x = orbit_radius*Math.cos(alpha);
84.      box.position.y = orbit_radius*Math.sin(alpha);
85.      box.position.z = 10*Math.sin(2*alpha);
86.
87.     //随着相机跟随立方体改变相机的视角
88.      camera.rotationOffset = (18*alpha)%360;
```

```
89.     });
90.
91.     return scene;
92.
93. };
```

上述代码的执行结果如图 3.31 所示。相机将会跟随图中红色方框内的物体进行运动。

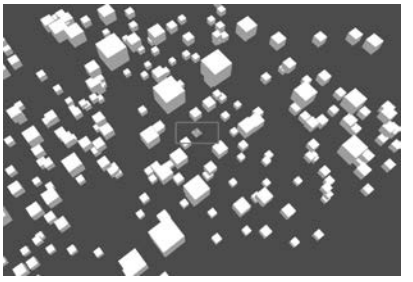


图 3.31　跟随相机实现效果

3.7　动画

无论开发者制作的是一款游戏，还是一款 AR/VR/MR 的应用，动画都是举足轻重的部分。一个 3D 场景，因为有了动画，才会有栩栩如生的效果，从而更加吸引用户在场景中漫游或互动。本节将为大家讲解在 Babylon.js 中动画的使用。

如图 3.32 所示的序列图像很好地展示了一段动画的原理。顺序播放每一个单帧图像，就会在画面中产生马儿跑动的动画效果。每秒切换的图像越多，这段动画就会越流畅，随之而来的感觉就是我们觉得马儿跑得更快。

图 3.32　动画序列原理

3.7.1　设计动画

假设想要实现一个 box(一个立方体)在屏幕上左右移动，第一秒时 box 会从初始位置运动到屏幕右侧，第二秒会返回到屏幕左侧，如此循环往复就形成了一段动画。这里要提出一个概念，那就是 AnimationClip(动画剪辑)。box 往复运动一个循环的这段动画，叫作一个动画剪辑，也即一个 AnimationClip，一个 AnimationClip 不断循环播放就能达到开发者的目的——并非需要 10s 的动画，就做一个 10s 的 AnimationClip。在 Babylon.js 中，创建一个动画的示例代码如下：

```
1.  const frameRate = 10;
2.  const xSlide = new BABYLON.Animation("xSlide", "position.x", frameRate, BABYLON.
Animation.ANIMATIONTYPE_FLOAT,BABYLON.Animation.ANIMATIONLOOPMODE_CYCLE);
```

在上述代码中，frameRate 代表帧速率，帧速率即每秒画面刷新的次数，这里设置的值为 10。接下来还需要设置 3 个关键帧，分别是起始点、box 改变方向时的点和终点。确定了 3 个关键帧，然后让 box 沿着关键帧和帧速率做插值运算，就可以形成一段完整的动画。在 Babylon.js 中设置动画关键帧的代码如下：

```
const keyFrames = [];

    keyFrames.push({
        frame: 0,
        value: 2
    });

    keyFrames.push({
        frame: frameRate,
        value: -2
    });

    keyFrames.push({
        frame: 2 * frameRate,
        value: 2
    });

    xSlide.setKeys(keyFrames);
```

最后将动画添加到 box 上，并且播放动画，就可以实现预想的效果。本案例完整的代码如下：

```
const createScene = () => {
    const scene = new BABYLON.Scene(engine);

    const camera = new BABYLON.ArcRotateCamera("Camera", - Math.PI / 2, Math.PI / 4, 10,
BABYLON.Vector3.Zero());
    camera.attachControl(canvas, true);

    const light1 = new BABYLON.DirectionalLight("DirectionalLight", new BABYLON.Vector3
(0, -1, 1));
    const light2 = new BABYLON.HemisphericLight("HemiLight", new BABYLON.Vector3(0, 1,
0));
    light1.intensity = 0.75;
    light2.intensity = 0.5;

    const box = BABYLON.MeshBuilder.CreateBox("box", {});
    box.position.x = 2;

    const frameRate = 10;

    const xSlide = new BABYLON.Animation("xSlide", "position.x", frameRate, BABYLON.
Animation.ANIMATIONTYPE_FLOAT, BABYLON.Animation.ANIMATIONLOOPMODE_CYCLE);
```

```
18.
19.     const keyFrames = [];
20.
21.     keyFrames.push({
22.         frame: 0,
23.         value: 2
24.     });
25.
26.     keyFrames.push({
27.         frame: frameRate,
28.         value: -2
29.     });
30.
31.     keyFrames.push({
32.         frame: 2 * frameRate,
33.         value: 2
34.     });
35.
36.     xSlide.setKeys(keyFrames);
37.
38.     box.animations.push(xSlide);
39.
40.     scene.beginAnimation(box, 0, 2 * frameRate, true);
41.
42.     return scene;
43. };
```

最终运行的效果如图 3.33 所示。

(a) box从左往右运动　　(b) box从右往左运动

图 3.33　动画播放效果

上述通过代码来实现动画播放的方法当然可行，但是当动画的需求变得复杂之后，通过代码来实现动画效果显然会变得极其复杂。这时候，有一个良好的动画设计工具就显得尤为重要了，所幸 Babylon.js 也为开发者提供了动画曲线编辑器(Animation Curve Editor)，帮助开发者更快速地设计动画，该编辑器界面如图 3.34 所示。可以看出这里的动画曲线编辑器与 Unity 中的动画曲线编辑器能够实现的功能是非常类似的。

3.7.2　序列动画

在大部分动画需求中，一个动画剪辑是无法实现我们想要的结果的，通常会将多个动画

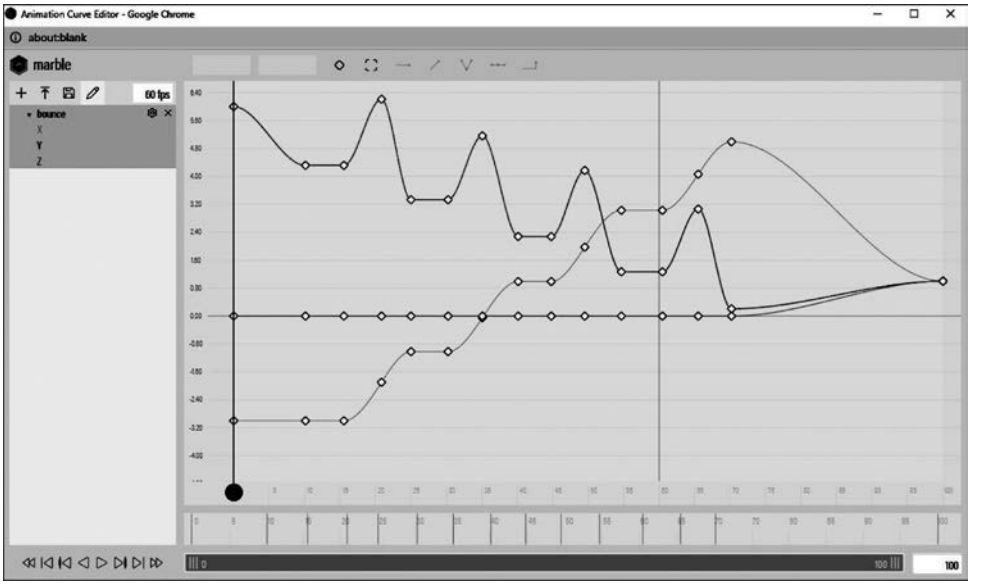


图 3.34　动画曲线编辑器

剪辑进行组合搭配顺序播放，这样为每个动画剪辑来指定播放的时间，就可以构成一整套动画片段。

例如，想要制作一段这样的动画：相机显示一栋带门的建筑，然后相机靠近门并停了下来，门打开，相机进入房间，房间灯亮了，门关闭，相机扫过房间。这一段文字描述的动画涉及多个物体、多个画面，因此属于一组序列动画，其中涉及如下几种动画物体。

(1) Camera(相机)，相机位移而导致画面运动。

(2) Door(门)，开门/关门的动画。

(3) Light(灯光)。

可通过一个时间表来描述这段动画，如图 3.35 所示。

	3		5		7	8	9				13	14	
Timing													
Performers													
Camera	Move to Door				Enter Room				Swing Around				
Door			Opens								Closes		
Lights						Brighten						Off	

图 3.35　动画时间表

描述清楚动画之间的时间关系后，可按照下列步骤来实现想要的效果。

1. 相机向前移动的动画

通过 push()方法为动画添加相机向前移动的关键帧。

```
1.  var movein = new BABYLON.Animation("movein", "position", frameRate, BABYLON.Animation.
ANIMATIONTYPE_VECTOR3,BABYLON.Animation.ANIMATIONLOOPMODE_CONSTANT);
```

```
2.
3.      var movein_keys = [];
4.
5.      movein_keys.push({
6.         frame: 0,
7.         value: new BABYLON.Vector3(0, 5, -30)
8.      });
9.
10.     movein_keys.push({
11.        frame: 3 * frameRate,
12.        value: new BABYLON.Vector3(0, 2, -10)
13.     });
14.
15.     movein_keys.push({
16.        frame: 5 * frameRate,
17.        value: new BABYLON.Vector3(0, 2, -10)
18.     });
19.
20.     movein_keys.push({
21.        frame: 8 * frameRate,
22.        value: new BABYLON.Vector3(-2, 2, 3)
23.     });
24.
25.     movein.setKeys(movein_keys);
```

2. 相机扫过的动画

通过 push()方法为动画添加相机向前扫过的关键帧。

```
1.    var rotate = new BABYLON.Animation("rotate", "rotation.y", frameRate, BABYLON.
Animation.ANIMATIONTYPE_FLOAT,BABYLON.Animation.ANIMATIONLOOPMODE_CONSTANT);
2.
3.      var rotate_keys = [];
4.
5.      rotate_keys.push({
6.         frame: 0,
7.         value: 0
8.      });
9.
10.     rotate_keys.push({
11.        frame: 9 * frameRate,
12.        value: 0
13.     });
14.
15.     rotate_keys.push({
16.        frame: 14 * frameRate,
17.        value: Math.PI
18.     });
19.
20.     rotate.setKeys(rotate_keys);
```

3. 门打开和关闭的动画

通过 push()方法为动画添加门打开和关闭的关键帧。

```
var sweep = new BABYLON.Animation("sweep", "rotation.y", frameRate, BABYLON.Animation.ANIMATIONTYPE_FLOAT,BABYLON.Animation.ANIMATIONLOOPMODE_CONSTANT);

    var sweep_keys = [];

    sweep_keys.push({
       frame: 0,
       value: 0
    });

    sweep_keys.push({
       frame: 3 * frameRate,
       value: 0
    });

    sweep_keys.push({
       frame: 5 * frameRate,
       value: Math.PI/3
    });

    sweep_keys.push({
       frame: 13 * frameRate,
       value: Math.PI/3
    });

    sweep_keys.push({
       frame: 15 * frameRate,
       value: 0
    });

    sweep.setKeys(sweep_keys);
```

4. 灯光变亮和变暗的动画

通过 push()方法为动画添加调整灯光明暗的关键帧。

```
var lightDimmer = new BABYLON.Animation("dimmer", "intensity", frameRate, BABYLON.Animation.ANIMATIONTYPE_FLOAT,BABYLON.Animation.ANIMATIONLOOPMODE_CONSTANT);

    var light_keys = [];

    light_keys.push({
       frame: 0,
       value: 0
    });

    light_keys.push({
```

```
        frame: 7 * frameRate,
        value: 0
    });

    light_keys.push({
        frame: 10 * frameRate,
        value: 1
    });

    light_keys.push({
        frame: 14 * frameRate,
        value: 1
    });

    light_keys.push({
        frame: 15 * frameRate,
        value: 0
    });

lightDimmer.setKeys(light_keys);
```

5. 运行所有动画剪辑

最后调用上述所有的动画剪辑。

```
scene.beginDirectAnimation(camera, [movein, rotate], 0, 25 * frameRate, false);
scene.beginDirectAnimation(hinge, [sweep], 0, 25 * frameRate, false);
scene.beginDirectAnimation(spotLights[0], [lightDimmer], 0, 25 * frameRate, false);
scene.beginDirectAnimation(spotLights[1], [lightDimmer.clone()], 0, 25 * frameRate, false);
```

6. 创建场景中的物体

然后添加上述动画场景中的物体对象，包括场地、门等。

```
var ground = BABYLON.MeshBuilder.CreateGround("ground", {width:50, height:50}, scene);

    var wall1 = BABYLON.MeshBuilder.CreateBox("door", {width:8, height:6, depth:0.1}, scene);
    wall1.position.x = -6;
    wall1.position.y = 3;

    var wall2 = BABYLON.MeshBuilder.CreateBox("door", {width:4, height:6, depth:0.1}, scene);
    wall2.position.x = 2;
    wall2.position.y = 3;

    var wall3 = BABYLON.MeshBuilder.CreateBox("door", {width:2, height:2, depth:0.1}, scene);
    wall3.position.x = -1;
    wall3.position.y = 5;

    var wall4 = BABYLON.MeshBuilder.CreateBox("door", {width:14, height:6, depth:0.1}, scene);
    wall4.position.x = -3;
```

```
    wall4.position.y = 3;
    wall4.position.z = 7;

    var wall5 = BABYLON.MeshBuilder.CreateBox("door", {width:7, height:6, depth:0.1}, scene);
    wall5.rotation.y = Math.PI/2;
    wall5.position.x = -10;
    wall5.position.y = 3;
    wall5.position.z = 3.5;

    var wall6 = BABYLON.MeshBuilder.CreateBox("door", {width:7, height:6, depth:0.1}, scene);
    wall6.rotation.y = Math.PI/2;
    wall6.position.x = 4;
    wall6.position.y = 3;
    wall6.position.z = 3.5;

    var roof = BABYLON.MeshBuilder.CreateBox("door", {width:14, height:7, depth:0.1}, scene);
    roof.rotation.x = Math.PI/2;
    roof.position.x = -3;
    roof.position.y = 6;
    roof.position.z = 3.5;

    }
```

3.8 音频

Babylon.js 的音频是基于 Web Audio 规范的，因此当开发者需要使用声音时，运行 WebXR 的浏览器需要兼容 Web Audio 规范。假设在不支持 Web Audio 规范的浏览器上使用，并不会影响引擎其他功能的使用，只是无法播放音频而已。声音引擎支持环境音、空间音和定向音，可以通过代码或加载.babylon 文件来创建，一般开发过程中使用的音频文件扩展名为.mp3 或.wav。

3.8.1 创建音频文件

创建音频文件的代码如下：

```
var createScene = function () {
    var scene = new BABYLON.Scene(engine);

    var camera = new BABYLON.FreeCamera("FreeCamera", new BABYLON.Vector3(0, 0, 0), scene);

    //载入音频文件,一旦准备好开始自动循环播放
    var music = new BABYLON.Sound("Violons", "sounds/violons11.wav", scene, null, { loop:
true, autoplay: true });

    return scene;
};
```

下面列举 Sound()函数中各个参数的作用。

第一个参数：声音的名称。

第二个参数：要加载的声音的 URL，即音频文件所在的路径。

第三个参数：附加声音的场景。

第四个参数：一旦声音准备好播放，函数就会被回调。

第五个参数：一个 JSON 对象。

也可以监听音乐处于可播放状态时回调函数的状态，表示当音频文件从本地加载并解析完成，或者从 Web 服务器加载或解析完成，接下来会自动播放音乐，代码如下：

```
var createScene = function () {
    var scene = new BABYLON.Scene(engine);

    var camera = new BABYLON.FreeCamera("FreeCamera", new BABYLON.Vector3(0, 0, 0), scene);

    var music = new BABYLON.Sound("Violons", "sounds/violons11.wav", scene,
          function() {
             // Sound has been downloaded & decoded
             music.play();
         }
    );

    return scene;
 };
```

此代码从 Web 服务器加载 music.wav 文件，对其进行解码并使用 play()函数在回调函数中播放一次。如果没有传递参数，那么 play()函数会立即播放声音，当然也可以提供 number 类型的参数，设定在 x 秒后再开始播放声音，具体视需求而定。

3.8.2 通过事件触发音频播放

在一些情况下，需要通过键盘或者鼠标事件来触发音乐的播放或停止，接下来实现音乐与事件的绑定。代码中的 window.addEventListener()函数分别监听鼠标左键和键盘空格键的 Click 事件，当这些动作触发之后，就播放 gunshot.wav 这个游戏中枪械开火的音效，从而实现通过事件的触发来控制音频播放的功能。

```
var createScene = function () {
    var scene = new BABYLON.Scene(engine);

    var camera = new BABYLON.FreeCamera("FreeCamera", new BABYLON.Vector3(0, 0, 0), scene);

    var gunshot = new BABYLON.Sound("gunshot", "sounds/gunshot.wav", scene);

    window.addEventListener("mousedown", function(evt) {
        //单击鼠标左键开火
        if (evt.button === 0) {
```

```
            gunshot.play();
        }
    });

    window.addEventListener("keydown", function (evt) {
        //按下空格键开火
        if (evt.keyCode === 32) {
            gunshot.play();
        }
    });

    return scene;
};
```

3.8.3 音乐属性

可以通过选项对象或 setVolume()函数设置声音的音量,也可以以相同的方式设置播放速率。如果将开发者自己注册到 onended 事件中,还可以在声音播放完毕时收到通知。下面是一个混合所有这些功能的简单示例代码。

```
var volume = 0.1;
var playbackRate = 0.5;
var gunshot = new BABYLON.Sound("Gunshot", "./gunshot-1.wav", scene, null, {
  playbackRate: playbackRate,
  volume: volume
});

gunshot.onended = function() {
  if (volume < 1) {
    volume += 0.1;
    gunshot.setVolume(volume);
  }
  playbackRate += 0.1;
  gunshot.setPlaybackRate(playbackRate);
};
```

3.8.4 通过 ArrayBuffer 来加载音频文件

如果开发者使用自己提供的 ArrayBuffer 调用构造函数,则可以绕过第一阶段的请求(即嵌入式 XHR 请求),下面是一段示例代码。

```
var createScene = function () {
    var scene = new BABYLON.Scene(engine);

    var camera = new BABYLON.FreeCamera("FreeCamera", new BABYLON.Vector3(0, 0, 0), scene);

```

```
    var gunshotFromAB;
    loadArrayBufferFromURL("sounds/gunshot.wav");

    function loadArrayBufferFromURL(urlToSound) {
        var request = new XMLHttpRequest();
        request.open('GET', urlToSound, true);
        request.responseType = "arraybuffer";
        request.onreadystatechange = function() {
            if (request.readyState == 4) {
                if (request.status == 200) {
                    gunshotFromAB = new BABYLON.Sound("FromArrayBuffer", request.response,
scene, soundReadyToBePlayed);
                }
            }
        };
        request.send(null);
    }

    function soundReadyToBePlayed() {
        gunshotFromAB.play();
    }

    return scene;
};
```

3.8.5 通过资源管理器加载音频文件

资源管理器在 WebXR 开发过程中非常有用。通过资源管理器加载音频文件，就可以实现加载进度的展示，而不用让应用处于等待状态。

```
var createScene = function () {
    var scene = new BABYLON.Scene(engine);

    var camera = new BABYLON.FreeCamera("FreeCamera", new BABYLON.Vector3(0, 0, 0), scene);

    var music1, music2, music3;
    //通过 AssetsManager 资源管理器加载音频
     var assetsManager = new BABYLON.AssetsManager(scene);

     var binaryTask = assetsManager.addBinaryFileTask("Violons18 task", "sounds/
violons18.wav");
    binaryTask.onSuccess = function (task) {
        music1 = new BABYLON.Sound("Violons18", task.data, scene, soundReady, { loop: true });
    }

     var binaryTask2 = assetsManager.addBinaryFileTask("Violons11 task", "sounds/
violons11.wav");
    binaryTask2.onSuccess = function (task) {
```

```
        music2 = new BABYLON.Sound("Violons11", task.data, scene, soundReady, { loop: true });
    }

    var binaryTask3 = assetsManager.addBinaryFileTask("Cello task", "sounds/cellolong.wav");
    binaryTask3.onSuccess = function (task) {
        music3 = new BABYLON.Sound("Cello", task.data, scene, soundReady, { loop: true });
    }

    var soundsReady = 0;

    function soundReady() {
        soundsReady++;
        if (soundsReady === 3) {
            music1.play();
            music2.play();
            music3.play();
        }
    }

    assetsManager.load();

    return scene;
};
```

3.9 相机和网格

3.9.1 相机的行为

1. 弹跳行为(Bouncing Behaviour)

在轨道相机 ArcRotateCamera 中,当相机的半径达到最小值或最大值时,会产生一个小的弹跳效果,可以通过下面的属性来配置此行为。

(1) transitionDuration:定义动画的持续时间,以毫秒为单位,默认值为 450ms。

(2) lowerRadiusTransitionRange:定义到达下半径时过渡动画的距离长度,默认值为 2。

(3) upperRadiusTransitionRange:定义到达上半径时过渡动画的距离长度,默认值为−2。

(4) autoTransitionRange:定义一个值,指示是否自动定义了 lowerRadiusTransitionRange 和 upperRadiusTransitionRange。过渡范围将设置为世界空间中边界框对角线的 5%。

要在 ArcRotateCamera 上启用弹跳行为,可执行下面的一行代码:

```
camera.useBouncingBehaviour = true;
```

2. 自动旋转行为(AutoRotation Behaviour)

一般针对一个三维物体进行展示时,当用户没有对场景中的模型进行交互时,可以让相

机围绕目标缓慢旋转,以便于用户观察。该行为可以使用下列属性进行配置。

(1) idleRotationSpeed:相机围绕网格旋转的速度。

(2) idleRotationWaitTime:用户交互后相机开始旋转前等待的时间(以毫秒为单位)。

(3) idleRotationSpinupTime:旋转到完全停止所需的时间(以毫秒为单位)。

(4) zoomStopsAnimation:指示用户缩放是否应停止动画的标志。

要在轨道相机上启用自动旋转行为,可执行下面的一行代码:

```
camera.useAutoRotationBehaviour = true;
```

3. 框架行为(Framing Behaviour)

框架行为 BABYLON.FramingBehaviour 旨在在轨道相机的目标设置为网格时自动定位它。如果想防止相机进入虚拟水平面,那么该框架行为也很有用。可以使用以下属性配置该行为:

(1)BABYLON.FramingBehaviour.IgnoreBoundsSizeMode——相机可以一直向网格移动。

(2) BABYLON.FramingBehaviour.FitFrustumSidesMode——不允许相机比被调整的边界球体所接触的平截头体侧面的点更接近网格。参数有两个,分别为:

- 定义应用于半径的比例,默认为 1。
- 设置要在 Y 轴上应用的比例以定位相机焦点,默认为 0.5(表示边界框的中心)。

(3) defaultElevation——定义水平面上方/下方的角度,以便在触发默认高程空闲行为时返回,以弧度为单位,默认为 0.3。

(4) elevationReturnTime——定义返回到默认 beta 位置(默认为 1500)所需的时间,以毫秒为单位,负值表示相机不应返回默认值。

(5) depthReturnWaitTime——定义相机返回到默认 beta 位置之前的延迟,以毫秒为单位,默认为 1000。

(6) zoomStopsAnimation——定义用户缩放场景时是否应该停止动画。

(7) framingTime——构建网格框架时的过渡时间,以毫秒为单位,默认为 1500。

要在轨道相机上启用框架行为,可执行以下的一行代码:

```
camera.useFramingBehaviour = true;
```

3.9.2 网格的行为

1. PointerDragBehaviour

该行为用于使用鼠标或 VR 控制器围绕平面或轴拖动网格,它可以在 3 种不同的模式下初始化。

(1) dragAxis:将沿着提供的轴进行拖动。

(2) dragPlaneNormal:将沿着法线定义的平面进行拖动。

（3）None：将沿着面向相机的平面进行拖动。

默认情况下，拖动平面/轴将根据对象的方向进行修改。要将指定的轴/平面固定在世界坐标系中，应将 useObjectOrientationForDragging 设置为 false。下面实现一个完整的案例，代码如下：

```
var createScene = function () {
    //创建基本场景
    var scene = new BABYLON.Scene(engine);
    var camera = new BABYLON.FreeCamera("camera1", new BABYLON.Vector3(1, 5, -10), scene);
    camera.setTarget(BABYLON.Vector3.Zero());
    var light = new BABYLON.HemisphericLight("light1", new BABYLON.Vector3(0, 1, 0), scene);
    light.intensity = 0.7;
    var sphere = BABYLON.Mesh.CreateSphere("sphere1", 16, 2, scene);
    sphere.rotation.x = Math.PI/2
    sphere.position.y = 1;
    var ground = BABYLON.Mesh.CreateGround("ground1", 6, 6, 2, scene);

    //定义 pointerDragBehaviour 网格行为
    //var pointerDragBehaviour = new BABYLON.PointerDragBehaviour({});
    //var pointerDragBehaviour = new BABYLON.PointerDragBehaviour({dragPlaneNormal: new
BABYLON.Vector3(0,1,0)});
    var pointerDragBehaviour = new BABYLON.PointerDragBehaviour({dragAxis: new BABYLON.
Vector3(1,0,0)});

    //将指定的轴/平面固定在世界坐标系中
    pointerDragBehaviour.useObjectOrientationForDragging = false;

    //监听拖动事件
    pointerDragBehaviour.onDragStartObservable.add((event) =>{
        console.log("dragStart");
        console.log(event);
    })
    pointerDragBehaviour.onDragObservable.add((event) =>{
        console.log("drag");
        console.log(event);
    })
    pointerDragBehaviour.onDragEndObservable.add((event) =>{
        console.log("dragEnd");
        console.log(event);
    })

    //如果需要手动处理拖动事件(在不移动附加网格的情况下使用拖动行为),则将
moveAttached 设置为 false
    // pointerDragBehaviour.moveAttached = false;

    sphere.addBehaviour(pointerDragBehaviour);

    return scene;

};
```

2. SixDofDragBehaviour

基于指针的原点(例如,相机或 VR 控制器位置),将 Mesh 网格在 3D 空间中进行拖动,默认情况下,通过将网格缓慢移动到指针指向的位置来平滑指针抖动。要删除或修改此行为,可以修改以下字段。

```
sixDofDragBehaviour.dragDeltaRatio = 0.2;
```

默认情况下,(将对象拖离/拖向用户的操作将被放大来确保更容易将物体移动到更远的距离。为了规避或者修改这种情况,可以使用以下内容:

```
sixDofDragBehaviour.zDragFactor = 0.2;
```

需要注意的一点是,为避免在使用具有复杂几何形状的模型时对性能造成较大影响,应将对象包裹在边界框网格中。

3.10 资源管理

在 Babylon.js 引擎中,默认内置的能够加载的资源格式为.babylon 格式,其他的资源格式都需要加载对应的插件来实现,例如,glTF、GLB、OBJ、STL 等格式的模型资源。如果要快速添加所有的加载插件,可以在页面中添加以下脚本:

```
<script src="https://cdn.babylonjs.com/babylon.js"></script>
<script src="https://cdn.babylonjs.com/loaders/babylonjs.loaders.min.js">
</script>
```

在使用 NPM 进行安装时,可以使用下面的命令:

```
npm install --save babylonjs babylonjs-loaders
```

如果开发时采用 TypeScript 语言,则需要在 tsconfig.json 文件中添加如下代码:

```
...
   "types": [
      "babylonjs",
      "babylonjs-loaders",
      ""
   ],
...
```

在完成上述设置工作后,就可以在代码中引用对应的类型了。当使用 Webpack 打包项目时,将使用最小的 minifield 文件。

```
import * as BABYLON from 'babylonjs';
import 'babylonjs-loaders';
```

3.10.1 SceneLoader.Append

所有资源类型都可以用 SceneLoader.Append 接口来进行加载，具体使用方法如下：

```
BABYLON.SceneLoader.Append("./", "duck.gltf", scene, function (scene) {
    //场景中将要执行的动作
});
```

通过字符串加载 Babylon 的资源并添加到场景中。使用的格式是“data：”关键字加表示资源的字符串。

```
BABYLON.SceneLoader.Append("", "data:" + gltfString, scene, function (scene) {
    //场景中将要执行的动作
});
```

还可以通过一个基于 Base64 编码的.glb 二进制文件进行加载。

```
var base64_model_content = "data:;base64,BASE 64 ENCODED DATA...";
BABYLON.SceneLoader.Append("", base64_model_content, scene, function (scene) {
    //场景中将要执行的动作
});
```

3.10.2 SceneLoader.Load

该方法将加载所有 Babylon 资源并且创建一个新的场景。

```
BABYLON.SceneLoader.Load("/assets/", "batman.obj", engine, function (scene) {
    //场景中将要执行的动作
});
```

3.10.3 SceneLoader.ImportMesh

该方法用来向场景中加载网格 Mesh 和骨骼 Skeletons，默认第一个参数为 null，表示加载文件中的所有 Mesh 和 Skeletons。

```
BABYLON.SceneLoader.ImportMesh(["myMesh1", "myMesh2"], "./", "duck.gltf", scene,
function (meshes, particleSystems, skeletons) {
    /场景中使用 mesh 和 skeletons 将要执行的动作
    //对于 glTF 资源，粒子系统通常为 null
});
```

在上述回调函数中，对于 glTF 格式的文件，particleSystems 始终为 null，也就是说，glTF 文件是无法支持 Babylon 引擎中的粒子系统的。

3.10.4 SceneLoader.ImportMeshAsync

该函数为 ImportMesh 的异步版本，可以通过调用返回的 promise 或使用 await 关键字

来获得结果。注意,要在 createScene()函数中使用 await 关键字,必须在其定义中将其标记为 async。

1. 使用 promise

```
1.  const importPromise = BABYLON.SceneLoader.ImportMeshAsync(["myMesh1", "myMesh2"],
"./", "duck.gltf", scene);
2.  importPromise.then((result) => {
3.     //结果包含网格、粒子系统、骨架、动画组和变换节点
4.  })
```

2. 使用 await 关键字

```
1.  const result = await BABYLON.SceneLoader.ImportMeshAsync(["myMesh1", "myMesh2"], "./",
"duck.gltf", scene);
```

3.10.5 SceneLoader.LoadAssetContainer

Container 意为容器。顾名思义,该函数将加载资源,但并不会立即将资源添加至场景中,而是会先放在一个容器中。

```
1.  BABYLON.SceneLoader.LoadAssetContainer("./", "duck.gltf", scene, function (container) {
2.     var meshes = container.meshes;
3.     var materials = container.materials;
4.     ...
5.     // 最后再将所有元素添加至场景中
6.     container.addAllToScene();
7.  });
```

3.10.6 SceneLoader.ImportAnimations

该函数将加载动画文件并合并至场景中。

```
1.  BABYLON.SceneLoader.ImportAnimations("./", "Elf_run.gltf", scene);
```

3.10.7 SceneLoader.AppendAsync

该函数为 SceneLoader.Append 函数的异步版本。

```
1.  BABYLON.SceneLoader.AppendAsync("./", "duck.gltf", scene).then(function (scene) {
2.     //场景中将要执行的动作
3.  });
```

3.10.8 AssetsManager

在项目中,大部分情况下都会加载多个资源,Babylon.js 从 1.14 版本开始引入了资源

管理类。该类可以用于将网格导入到场景中，或加载文本、二进制文件等。下面学习如何使用 AssetsManager 类来加载资源。

1. 初始化并创建任务

在使用 AssetsManager 之前，首先需要通过当前场景创建一个资源管理器。

```
1.  var assetsManager = new BABYLON.AssetsManager(scene);
```

接下来，通过 assetsManager.addMeshTask 函数可以向 assetsManager 添加一个任务，代码如下：

```
1.  var meshTask = assetsManager.addMeshTask("skull task", "", "scenes/", "skull.babylon");
```

每一个任务都可以通过监听成功和失败的回调函数来做进一步的处理。

（1）当加载成功时，加载 Mesh 并初始化坐标。

```
1.  meshTask.onSuccess = function (task) {
2.      task.loadedMeshes[0].position = BABYLON.Vector3.Zero();
3.  }
```

（2）当加载失败时，控制台给出异常日志。

```
1.  meshTask.onError = function (task, message, exception) {
2.      console.log(message, exception);
3.  }
```

2. 任务的类型

AssetsManager 类主要有如下的 8 种任务类型，下面依次说明。

（1）TextFileTask：文本文件任务。

（2）MeshAssetTask：网格资源任务。

（3）TextureAssetTask：纹理资源任务。

（4）CubeTextureAssetTask：立方体纹理资源任务。

（5）ContainerAssetTask：容器资源任务。

（6）BinaryFileAssetTask：二进制文件资源任务。

（7）ImageAssetTask：图像资源任务。

（8）HDRCubeTextureTask：HDR 立方体纹理任务。

3. 运行 AssetsManager

调用下列代码运行所有的任务：

```
assetsManager.load();
```

4. AssetsManager 的回调和观察者

AssetsManager 提供了 4 个回调来帮助开发者更好地监测资源加载信息，它们分别是：

(1) onFinish——所有任务加载完成。

(2) onProgress——加载进度。

(3) onTaskSuccess——任务加载成功。

(4) onTaskError——加载任务报错。

下列代码举例说明 AssetsManager 的具体使用方法,给出了通过 onProgress 方式加载 UI 文本的过程。

```
1.  assetsManager.onProgress = function(remainingCount, totalCount, lastFinishedTask) {
2.   engine.loadingUIText = 'We are loading the scene. ' + remainingCount + ' out of ' +
totalCount + ' items still need to be loaded.';
3.  };
4.
5.  assetsManager.onFinish = function(tasks) {
6.      engine.runRenderLoop(function() {
7.          scene.render();
8.      });
9.  };
```

3.10.9 使用加载进度

默认情况下,在进行资源的加载时,AssetsManager 会显示一个加载界面,如图 3.36 所示。

图 3.36 显示加载界面

要想禁用上述加载界面,可使用下面的代码:

```
assetsManager.useDefaultLoadingScreen = false;
```

如果使用的是 SceneLoader 来加载资源,那么当 SceneLoader 中的 ShowLoadingScreen 设置为 true 时,也会显示上面的加载界面。该属性默认情况下为 true。如果要禁用加载界面,则需使用下列方式:

```
BABYLON.SceneLoader.ShowLoadingScreen = false;
```

也可以通过手动调用以下的函数来打开或者关闭加载 UI，当然，在大部分情况下，都是需要加载界面的。

```
engine.displayLoadingUI();
engine.hideLoadingUI();
```

还可以设置加载 UI 上的文字内容和加载界面的背景颜色。

```
engine.loadingUIText = "text";
engine.loadingUIBackgroundColor = "red";
```

3.11 材质

材质使得场景中的网格物体拥有了颜色和纹理，一个物体的材质如何显示，取决于场景中的灯光，以及材质如何对灯光进行计算并反映到画面中。材质对光有如下 4 种可能的反应方式。

(1) 漫反射(Diffuse)：漫反射表示在灯光照射下物体的基本颜色或质地。

(2) 高光(Specular)：灯光使得材质产生高光。

(3) 自发光(Emissive)：材质的颜色或质地仿佛会自发光。

(4) 环境(Ambient)：由背景灯光或环境光照射的材质颜色或纹理。

漫反射和高光需要开发者创建光源，环境光需要设置场景的环境色或给予场景背景光照。设置场景环境色的代码如下：

```
scene.ambientColor = new BABYLON.Color3(1, 1, 1);
```

3.11.1 材质的创建

通过代码来创建材质(Material)的方式如下：

```
var myMaterial = new BABYLON.StandardMaterial("myMaterial", scene);
```

当材质创建完成后，可以设置上面提到的 4 种颜色中的一个或多个，但是大家要记得，ambientColor 只有在设置了场景的 ambientColor 后才会生效。

```
1.  myMaterial.diffuseColor = new BABYLON.Color3(1, 0, 1);
2.  myMaterial.specularColor = new BABYLON.Color3(0.5, 0.6, 0.87);
3.  myMaterial.emissiveColor = new BABYLON.Color3(1, 1, 1);
4.  myMaterial.ambientColor = new BABYLON.Color3(0.23, 0.98, 0.53);
5.  mesh.material = myMaterial;
```

3.11.2 漫反射

为了了解 diffuseColor 如何对灯光做出反应，下面展示不同颜色的材质对白色、红色、

绿色和蓝色的漫反射聚光灯做出反应的过程。完整的案例代码如下：

```
var createScene = function () {
    var scene = new BABYLON.Scene(engine);
    var camera = new BABYLON.ArcRotateCamera("Camera", -Math.PI / 2, Math.PI / 3, 10,
BABYLON.Vector3.Zero(), scene);
    camera.attachControl(canvas, true);

    var mats = [
        new BABYLON.Color3(1, 1, 0),
        new BABYLON.Color3(1, 0, 1),
        new BABYLON.Color3(0, 1, 1),
        new BABYLON.Color3(1, 1, 1)
    ]

    var redMat = new BABYLON.StandardMaterial("redMat", scene);
    redMat.emissiveColor = new BABYLON.Color3(1, 0, 0);

    var greenMat = new BABYLON.StandardMaterial("greenMat", scene);
    greenMat.emissiveColor = new BABYLON.Color3(0, 1, 0);

    var blueMat = new BABYLON.StandardMaterial("blueMat", scene);
    blueMat.emissiveColor = new BABYLON.Color3(0, 0, 1);

    var whiteMat = new BABYLON.StandardMaterial("whiteMat", scene);
    whiteMat.emissiveColor = new BABYLON.Color3(1, 1, 1);


    //红光
    var lightRed = new BABYLON.SpotLight("spotLight", new BABYLON.Vector3(-0.9, 1 ,
-1.8), new BABYLON.Vector3(0, -1, 0), Math.PI / 2, 1.5, scene);
    lightRed.diffuse = new BABYLON.Color3(1, 0, 0);
    lightRed.specular = new BABYLON.Color3(0, 0, 0);

    //绿光
    var lightGreen = new BABYLON.SpotLight("spotLight1", new BABYLON.Vector3(0, 1,
-0.5), new BABYLON.Vector3(0, -1, 0), Math.PI / 2, 1.5, scene);
    lightGreen.diffuse = new BABYLON.Color3(0, 1, 0);
    lightGreen.specular = new BABYLON.Color3(0, 0, 0);

    //蓝光
    var lightBlue = new BABYLON.SpotLight("spotLight2", new BABYLON.Vector3(0.9, 1,
-1.8), new BABYLON.Vector3(0, -1, 0), Math.PI / 2, 1.5, scene);
    lightBlue.diffuse = new BABYLON.Color3(0, 0, 1);
    lightBlue.specular = new BABYLON.Color3(0, 0, 0);

    //白光
    var lightWhite = new BABYLON.SpotLight("spotLight3", new BABYLON.Vector3(0, 1, 1),
new BABYLON.Vector3(0, -1, 0), Math.PI / 2, 1.5, scene);
```

```
43.     lightWhite.diffuse = new BABYLON.Color3(1, 1, 1);
44.     lightWhite.specular = new BABYLON.Color3(0, 0, 0);
45.
46.     var redSphere = BABYLON.MeshBuilder.CreateSphere("sphere", {diameter: 0.25}, scene);
47.     redSphere.material = redMat;
48.     redSphere.position = lightRed.position;
49.
50.     var greenSphere = BABYLON.MeshBuilder.CreateSphere("sphere", {diameter: 0.25}, scene);
51.     greenSphere.material = greenMat;
52.     greenSphere.position = lightGreen.position;
53.
54.     var blueSphere = BABYLON.MeshBuilder.CreateSphere("sphere", {diameter: 0.25}, scene);
55.     blueSphere.material = blueMat;
56.     blueSphere.position = lightBlue.position;
57.
58.     var whiteSphere = BABYLON.MeshBuilder.CreateSphere("sphere", {diameter: 0.25}, scene);
59.     whiteSphere.material = whiteMat;
60.     whiteSphere.position = lightWhite.position;
61.
62.     var groundMat = new BABYLON.StandardMaterial("groundMat", scene);
63.     groundMat.diffuseColor = mats[0];
64.
65.     var ground = BABYLON.MeshBuilder.CreateGround("ground", {width: 4, height: 6}, scene);
66.     ground.material = groundMat;
67.
68.     /******************** GUI ********************/
69.     var makeYellow = function() {
70.        groundMat.diffuseColor = mats[0];
71.     }
72.
73.     var makePurple = function() {
74.        groundMat.diffuseColor = mats[1];
75.     }
76.
77.     var makeCyan = function() {
78.        groundMat.diffuseColor = mats[2];
79.     }
80.
81.     var makeWhite = function() {
82.        groundMat.diffuseColor = mats[3];
83.     }
84.
85.     var matGroup = new BABYLON.GUI.RadioGroup("Material Color", "radio");
86.     matGroup.addRadio("Yellow", makeYellow, true);
87.     matGroup.addRadio("Purple", makePurple);
88.     matGroup.addRadio("Cyan", makeCyan);
89.     matGroup.addRadio("White", makeWhite);
90.
91.     var advancedTexture = BABYLON.GUI.AdvancedDynamicTexture.CreateFullscreenUI("UI");
```

```
92.
93.     var selectBox = new BABYLON.GUI.SelectionPanel("sp", [matGroup]);
94.     selectBox.width = 0.25;
95.     selectBox.height = "50%";
96.     selectBox.top = "4px";
97.     selectBox.left = "4px";
98.     selectBox.background = "white";
99.     selectBox.horizontalAlignment = BABYLON.GUI.Control.HORIZONTAL_ALIGNMENT_LEFT;
100.        selectBox.verticalAlignment = BABYLON.GUI.Control.VERTICAL_ALIGNMENT_TOP;
101.
102.       advancedTexture.addControl(selectBox);
103.
104.       return scene;
105.
106.  };
```

代码运行后，可以看到画面效果如图 3.37 所示。

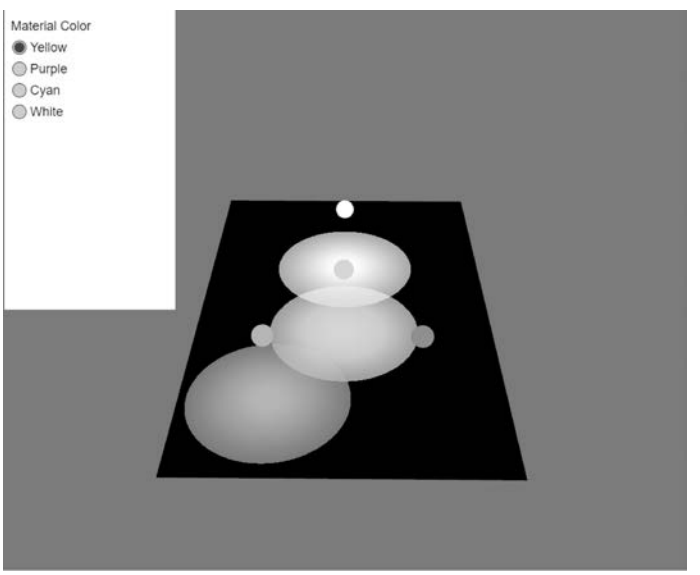


图 3.37　支持多种颜色的漫反射聚光效果

在本案例中，创建了 4 盏聚光灯，它们的漫反射颜色分别为白光、绿光、红光和蓝光。为了便于观察，在这 4 盏灯的位置上创建了 4 个小球，并设置其漫反射颜色。接下来需要观察这 4 盏灯照射到地面（图 3.37 中黑色的 Plane）后，地面呈现出的效果。

尝试将地面的漫反射颜色依次设置为黄色（Yellow）、紫色（Purple）、青色（Cyan）和白色（White），然后分别观察在不同的漫反射颜色下的聚光灯效果。

第一种情况：地面的漫反射颜色为黄色时，对灯光的反应效果如图 3.38 所示。

第二种情况：地面的漫反射颜色为紫色时，对灯光的反应效果如图 3.39 所示。注意观察白光照射后的地面颜色。

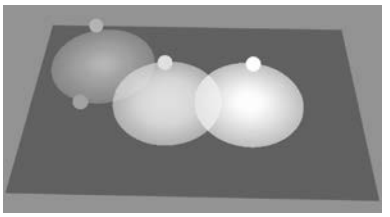


图 3.38　漫反射颜色为黄色时的聚光效果

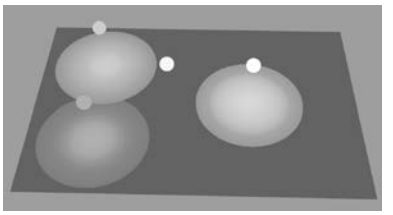


图 3.39　漫反射颜色为紫色时的聚光效果

第三种情况：地面的漫反射颜色为青色时，对灯光的反应效果如图 3.40 所示。同样注意白光的照射区域。

第四种情况：地面的漫反射颜色为白色时，对灯光的反应效果如图 3.41 所示。

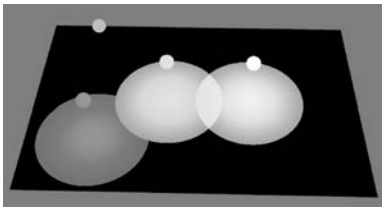


图 3.40　漫反射颜色为青色时的聚光效果

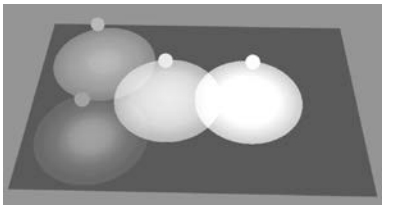


图 3.41　漫反射颜色为白色时的聚光效果

3.11.3　环境光颜色

接下来探索环境光颜色的作用，首先创建如下示例代码。

```
1.  var createScene = function () {
2.      var scene = new BABYLON.Scene(engine);
3.      var camera = new BABYLON.ArcRotateCamera("Camera", -Math.PI / 2,  Math.PI / 4, 5,
BABYLON.Vector3.Zero(), scene);
4.      camera.attachControl(canvas, true);
5.
6.      scene.ambientColor = new BABYLON.Color3(1, 1, 1);
7.
8.      var redMat = new BABYLON.StandardMaterial("redMat", scene);
9.      redMat.ambientColor = new BABYLON.Color3(1, 0, 0);
10.
11.     var greenMat = new BABYLON.StandardMaterial("redMat", scene);
12.     greenMat.ambientColor = new BABYLON.Color3(0, 1, 0);
13.
14.     //无环境光
15.     var sphere0 = BABYLON.MeshBuilder.CreateSphere("sphere0", {}, scene);
16.     sphere0.position.x = -1.5;
17.
18.     //红色环境光
19.     var sphere1 = BABYLON.MeshBuilder.CreateSphere("sphere1", {}, scene);
20.     sphere1.material = redMat;
```

```
21.
22.     //绿色环境光
23.     var sphere2 = BABYLON.MeshBuilder.CreateSphere("sphere2", {}, scene);
24.     sphere2.material = greenMat;
25.     sphere2.position.x = 1.5;
26.
27.     return scene;
28.
29. };
```

上述代码创建了 3 个球体，它们的位置分别为左、中、右。左边的球体没有环境光颜色，中间的球体采用红色的环境光颜色，右边的球体采用绿色的环境光颜色。首先将上面代码中的场景环境光颜色注释掉，运行后只会得到如图 3.42 所示的效果。

图 3.42　无场景环境光下的材质效果

通过运行可以看到，如果不设置场景的环境光，那么即使右边两个球体都设置了环境光颜色，也依然得到的是黑色的球体，这就验证了前面提到的，环境光有效的前提是必须设置场景的 ambientColor。接下来打开设置场景环境光的代码，重新运行场景后会得到如图 3.43 所示的效果。

图 3.43　有场景环境光下的材质效果

由于左侧的小球没有设置 ambientColor，因此无法呈现出物体的颜色，而右侧两个小球都呈现出了环境光颜色。但还可以继续做个测试，因为当下的背景的环境光颜色为白色，因此右侧两个小球都呈现出的是自己的本色。如果改变背景环境光颜色，结果又会如何？将代码修改如下：

```
scene.ambientColor = new BABYLON.Color3(1, 1, 0);
```

上述代码会将场景的环境光颜色设置为黄色，黄色是由红色和绿色混合而成，那么背景的环境光颜色是否会影响小球的环境光颜色呢？运行效果如图 3.44 所示。

图 3.44　场景环境光与物体环境光颜色不冲突情况下的效果

可以观察到右边两个小球的颜色基本没有变化，接下来再将环境光颜色设置为(0,1,0)，即绿色，然后再运行代码，效果如图 3.45 所示。

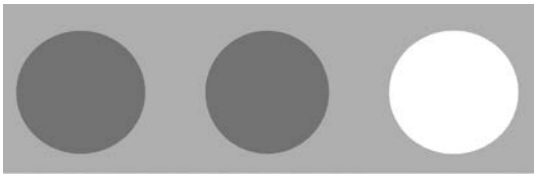


图 3.45　场景的环境光设置为绿色情况下的效果

可以看到背景的绿色并没有与物体的环境颜色进行混合，继续测试将背景的环境光颜色改为(0.5,1,0)，得到如图 3.46 所示的效果。

图 3.46　调整场景环境光改变物体亮度的效果

可以看出红色小球的亮度减半，而绿色小球的亮度不变，因此通过观察上述现象可知，背景的环境光颜色并不是与物体的环境光颜色进行混合，而只是对物体环境光的亮度进行了设置。接下来，向场景中添加如下代码：

```
1.  var light = new BABYLON.HemisphericLight("hemiLight", new BABYLON.Vector3(-1, 1, 0), scene);
2.  light.diffuse = new BABYLON.Color3(1, 0, 0);
```

上述代码向场景中添加了一个半球光，这个光从左上角对场景进行照射，其 diffuseColor 为红色，运行代码后，得到如图 3.47 所示的效果。

可以看到，红色的半球光照亮了 3 个小球，让最左侧的小球也呈现出红色。接下来，为灯光添加高光颜色。

```
light.specular = new BABYLON.Color3(0, 1, 0);
```

将灯光的高光颜色设置为绿色后，再运行场景，可以看到如图 3.48 所示的效果。

此时已经可以看到高光对小球产生的效果了，那么接下来进入这个案例最重要的环节，

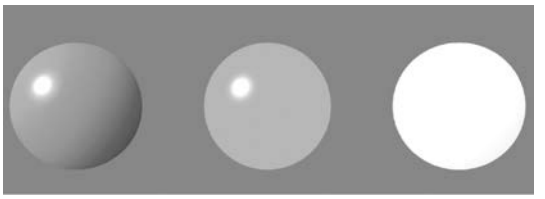


图 3.47　添加场景半球光后的效果

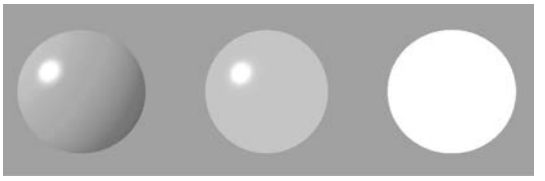


图 3.48　继续为场景半球光添加高光

即环境光的效果，继续添加代码。

```
light.groundColor = new BABYLON.Color3(0, 1, 0);
```

将灯光的背景色设置为绿色后，继续运行场景，可以看到如图 3.49 所示的效果。

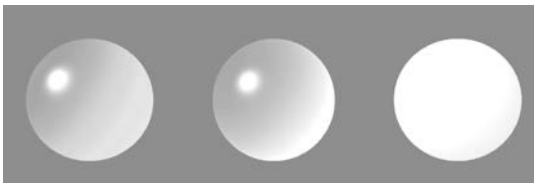


图 3.49　将灯光的背景色更改为绿色

左侧小球的右下角部分呈现绿色，说明灯光的背景色对物体产生了效果，这里可以推测，当不设置材质的环境光颜色时，应该默认为白色，中间的小球本色为红色。受到灯光的背景色影响后，右下角部分呈现黄色，这是红色和绿色混合后呈现的效果，最右侧的小球右下角呈现绿色，这是因为绿色与绿色混合依然为绿色。

3.11.4　透明颜色

材质的透明度设置可以直接设置材质的 alpha 值，0 为全透明，1 为不透明，介于 0 和 1 之间则为不同程度的半透明效果。

```
myMaterial.alpha = 0.5;
```

下面通过一个案例来说明透明材质的使用，代码如下：

```
1.  var createScene = function () {
2.      var scene = new BABYLON.Scene(engine);
```

```
3.      var camera = new BABYLON.ArcRotateCamera("Camera", -Math.PI / 2, 3 * Math.PI / 8, 5,
BABYLON.Vector3.Zero(), scene);
4.      camera.attachControl(canvas, true);
5.
6.
7.      //半球光从左上角照射
8.      var light = new BABYLON.HemisphericLight("hemiLight", new BABYLON.Vector3(-1, 1,
0), scene);
9.      light.diffuse = new BABYLON.Color3(1, 0, 0);
10.     light.specular = new BABYLON.Color3(0, 1, 0);
11.     light.groundColor = new BABYLON.Color3(0, 1, 0);
12.
13.     var redMat = new BABYLON.StandardMaterial("redMat", scene);
14.     redMat.diffuseColor = new BABYLON.Color3(1, 0, 0);
15.
16.     var greenMat = new BABYLON.StandardMaterial("greenMat", scene);
17.     greenMat.diffuseColor = new BABYLON.Color3(0, 1, 0);
18.     greenMat.alpha = 0.5;
19.
20.     //添加一个红色不透明球体
21.     var sphere1 = BABYLON.MeshBuilder.CreateSphere("sphere1", {}, scene);
22.     sphere1.material = redMat;
23.     sphere1.position.z = 1.5;
24.
25.     //添加一个绿色透明球体
26.     var sphere2 = BABYLON.MeshBuilder.CreateSphere("sphere2", {}, scene);
27.     sphere2.material = greenMat;
28.
29.     return scene;
30.
31.  };
```

在场景中创建了两个球体：一个红色球体为不透明材质，一个绿色球体为透明材质，最终实现的效果如图 3.50 所示。可以看到，透明材质在两个物体重叠部分可以实现透明的效果。

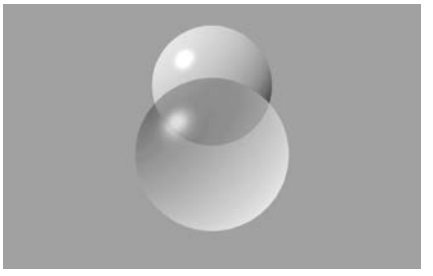


图 3.50 材质的透明度对比

3.11.5 纹理

纹理(Texture)有时候又被称为贴图，两者意思相同。在计算机图形学中，通过一张图片(Image)来作为模型的纹理，与材质的颜色接近。当材质创建完成后，可以设置材质的diffuseTexture、SpecularTexture、emmissiveTexture 或者 embientTexture。这里同样需要注意如果没有设置场景的环境色，那么 embientTexture 的设置也是无效的。具体设置的方法如下(在实际使用过程中，将纹理路径替换为真实的路径即可)：

```
var myMaterial = new BABYLON.StandardMaterial("myMaterial", scene);
myMaterial.diffuseTexture = new BABYLON.Texture("PATH TO IMAGE", scene);
myMaterial.specularTexture = new BABYLON.Texture("PATH TO IMAGE", scene);
myMaterial.emissiveTexture = new BABYLON.Texture("PATH TO IMAGE", scene);
myMaterial.ambientTexture = new BABYLON.Texture("PATH TO IMAGE", scene);
mesh.material = myMaterial;
```

当创建的材质为 StandardMaterial 时，如果没有指定法线，那么 Babylon.js 会自动计算法线。下面通过一个案例来说明材质的不同纹理是如何呈现的，以及针对光照有哪些反应。本案例选择使用一张草地的纹理来赋予材质，纹理样式如图 3.51 所示。

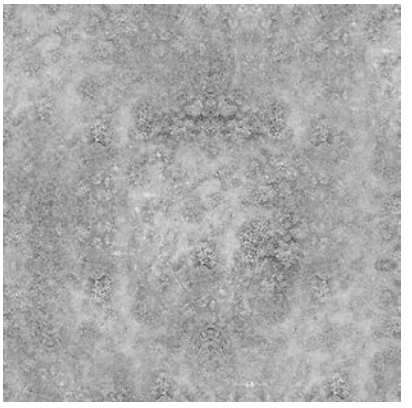


图 3.51　材质纹理样例

本案例的完整代码如下所示。

```
var createScene = function () {
    var scene = new BABYLON.Scene(engine);
    var camera = new BABYLON.ArcRotateCamera("Camera", -Math.PI / 2,  Math.PI / 4, 5,
BABYLON.Vector3.Zero(), scene);
    camera.attachControl(canvas, true);

    //半球光从左上角进行照射
    var light = new BABYLON.HemisphericLight("hemiLight", new BABYLON.Vector3(-1, 1,
0), scene);
    light.diffuse = new BABYLON.Color3(1, 0, 0);
    light.specular = new BABYLON.Color3(0, 1, 0);
    light.groundColor = new BABYLON.Color3(0, 1, 0);

    var grass0 = new BABYLON.StandardMaterial("grass0", scene);
    grass0.diffuseTexture = new BABYLON.Texture("textures/grass.png", scene);

    var grass1 = new BABYLON.StandardMaterial("grass1", scene);
    grass1.emissiveTexture = new BABYLON.Texture("textures/grass.png", scene);

    var grass2 = new BABYLON.StandardMaterial("grass2", scene);
```

```
    grass2.ambientTexture = new BABYLON.Texture("textures/grass.png", scene);
    grass2.diffuseColor = new BABYLON.Color3(1, 0, 0);

    //漫反射纹理
    var sphere0 = BABYLON.MeshBuilder.CreateSphere("sphere0", {}, scene);
    sphere0.position.x = -1.5;
    sphere0.material = grass0;

    //自发光纹理
    var sphere1 = BABYLON.MeshBuilder.CreateSphere("sphere1", {}, scene);
    sphere1.material = grass1;

    //环境光纹理和漫反射颜色
    var sphere2 = BABYLON.MeshBuilder.CreateSphere("sphere2", {}, scene);
    sphere2.material = grass2;
    sphere2.position.x = 1.5;

    return scene;

};
```

上述代码创建了3个球体,3个球体分别设置了3种不同的材质,最左边的球体采用的材质为漫反射纹理,中间的球体采用的材质为自发光纹理,右边的球体采用的材质为环境纹理,并将材质的环境色设置为红色,3种材质采用的纹理都为相同的草地纹理。

场景中只有一盏灯,即一个半球光来模拟环境光照,灯光的diffuseColor为红色,specularColor为绿色,groundColor为绿色。运行场景后,得到如图3.52所示的效果。

图3.52　在物体具有半球光的前提下继续添加材质

3.11.6　透明纹理

前面已经提到过,可以设置材质的透明度,实现代码如下:

```
myMaterial.alpha = 0.5;
```

这种设置透明的方法针对的是整个材质的颜色透明度,而对于纹理来说,图像可能会存在一部分区域为透明区域,是没有颜色信息的,而另一部分为图案,拥有颜色,如图3.53所示的图像中就拥有透明区域。

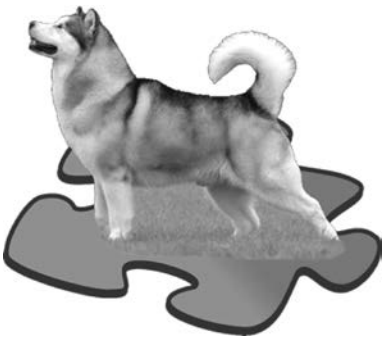


图 3.53 拥有透明区域的纹理

很容易就可以观察到图 3.53 中的图像哪部分为透明区域，但这样的透明纹理具体如何实现，依然通过一个案例来说明，完整的代码如下：

```
1.  var createScene = function() {
2.      var scene = new BABYLON.Scene(engine);
3.      var camera = new BABYLON.ArcRotateCamera("Camera", 3 * Math.PI / 2, Math.PI / 2, 5,
BABYLON.Vector3.Zero(), scene);
4.      camera.attachControl(canvas, false);
5.
6.      var light = new BABYLON.HemisphericLight("light1", new BABYLON.Vector3(0, 1, 0), scene);
7.     light.intensity = 0.7;
8.
9.     var pl = new BABYLON.PointLight("pl", BABYLON.Vector3.Zero(), scene);
10.    pl.diffuse = new BABYLON.Color3(1, 1, 1);
11.    pl.specular = new BABYLON.Color3(1, 1, 1);
12.    pl.intensity = 0.8;
13.
14.    var mat = new BABYLON.StandardMaterial("dog", scene);
15.    mat.diffuseTexture = new BABYLON.Texture("https://upload.wikimedia.org/wikipedia/
commons/8/87/Alaskan_Malamute%2BBlank.png", scene);
16.    mat.diffuseTexture.hasAlpha = true;
17.    mat.backFaceCulling = false;
18.    var box = BABYLON.MeshBuilder.CreateBox("box", {}, scene);
19.    box.material = mat;
20.
21.     return scene;
22.  };
```

上述代码中创建了一个立方体，并将上面的透明纹理赋予立方体，最终运行场景会得到如图 3.54 所示的效果。

3.11.7 显示模型线框

打开网格(mesh)的线框显示方法非常简单，只需将材质的 wireframe 属性开关打开即可，代码如下：

```
materialSphere1.wireframe = true;
```

线框模型的显示效果如图 3.55 所示。

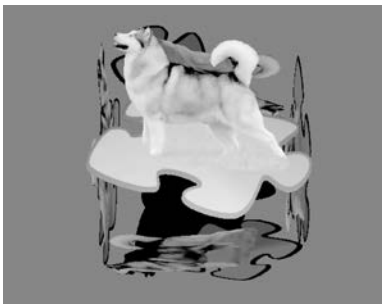


图 3.54　材质的立方体效果

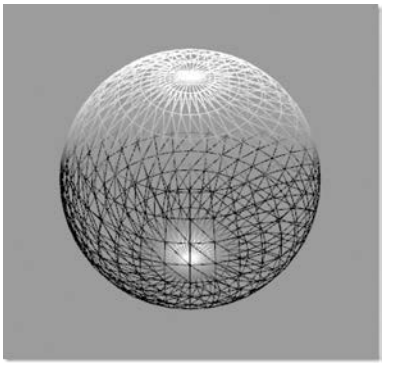


图 3.55　材质的线框模型