第5章　虚拟现实程序开发

Unity 3D作为目前主流的虚拟现实开发工具，是由Unity Technologies开发的跨平台专业三维游戏引擎，不仅在游戏领域大放光彩，而且已渗透到各个行业，譬如城市规划、教育、航天工业、房地产、文物古迹、广告、军事模拟、医疗培训、商业零售等。此外，其强大的三维引擎功能也将成为未来科技社会的重要工具。

5.1　Unity基础知识

5.1.1　Unity的历史

2004年，三位来自丹麦哥本哈根的热爱游戏的年轻人Joachim Ante、Nicholas Francis和David Helgason为帮助所有喜爱游戏的年轻人实现创作的梦想，决定一起开发一个易于使用、与众不同且价格低廉的游戏引擎。

2005年6月，Unity 1.0.1发布。

2008年6月，Unity支持Wii。

2008年10月，Unity支持iPhone。

2009年3月，Unity 2.5加入了对Windows的支持。

2009年10月，Unity 2.6独立版开始免费。

2010年4月，Unity支持iPad。

2010年11月，Unity推出Assets Store。

2013年11月，Unity跟Xbox ONE合作，Xbox ONE将可以使用Unity开发游戏。

2014年5月，Unity 4.5发布，加入了在iOS装置上支持OpenGL ES 3.0。

2014年11月26日，Unity 4.6发布，正式导入新的UI系统“UGUI”。

2015年3月4日，Unity 5正式发布，Unity还发布了Unity Cloud Build。

5.1.2　下载与安装

Mac OS和Windows操作系统都可以使用对应的Unity版本进行开发。Unity的官方网站提供了Unity安装包的下载。

目前Unity分为免费的个人版和付费的专业版。两种版本的引擎内容完全一样，但是专业版提供了更多额外服务。对于初学者或一般的独立开发者而言，个人版就可以满足所有需求，而相对于高级开发者，最好使用专业版。同时，Unity也通过不断地更新和嫁接更多的服务平台及技术来升级其版本。

在浏览器地址栏中输入 Unity 官方网址 https://Unity 3D. com/cn，进入 Unity 下载地址。直接单击个人版，下载的将是最新版本的 Unity。若想下载历史版本，需找到下载页面的最下面的 Unity 旧版本，即可下载自己想要的版本。下载页面如图 5.1 所示。

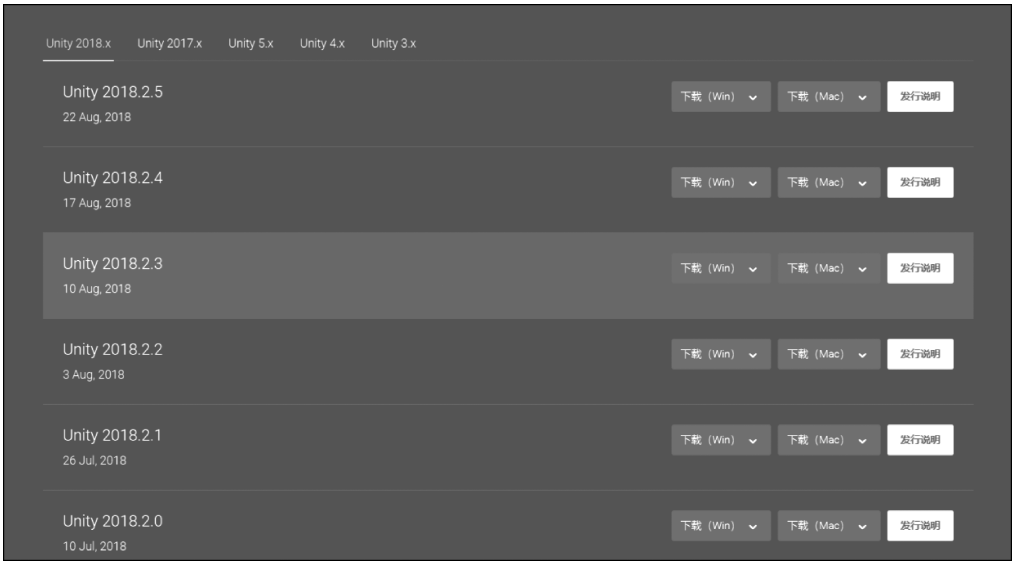


图 5.1　Unity 下载页面

也可以选择自己想要的版本(这里用 5.6.5 版本做介绍)，然后下载对应的 Windows 系列或者 Mac 系列，如图 5.2 所示。

然后会弹出一个下载任务(如图 5.3 所示)。

图 5.2　Unity 选择安装程序

图 5.3　Unity 下载保存路径

下载完毕后，双击下载好的 UnityDownloadAssistant-5. 6. 5f1. exe，弹出如图 5.4 所示界面，单击 Next 按钮，勾选 I accept the term of the license agreement，单击 Next 按钮；选择 64bit，选择保存路径，第一个选项 Specify location of files downloaded during installation 是指定安装过程中下载文件的位置，可以选择默认或者选择 Download to...下载到指定文件地址；第二个选项 Unity install folder 为指定安装路径。另外，下载完毕后，需到官网注册账号。

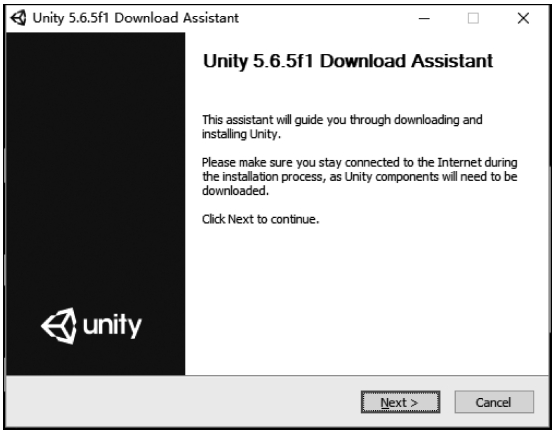


图 5.4　Unity 下载助手

5.1.3　Unity 编辑器

Unity 已经历多个版本的迭代更新。相对于传统的游戏引擎而言，Unity 添加了新的实时渲染构架(SRP)，使得游戏画面达到影视级别，可让游戏开发者的生活变得更加轻松，并帮助开发者们快速制作出更强大、更有趣的游戏。

一般来说，Unity 的工程由若干个场景组成。通过不断加载场景实现游戏场景的替换。Unity 具有高度自由的可视化编辑页面，可以让开发者轻松管理场景，高效快捷地进行开发。下面通过创建工程介绍编辑器界面。

步骤 1：创建工程

(1) 启动 Unity 后可以看到如图 5.5 所示的界面，单击 NEW 按钮，出现新工程页面(见图 5.6)。

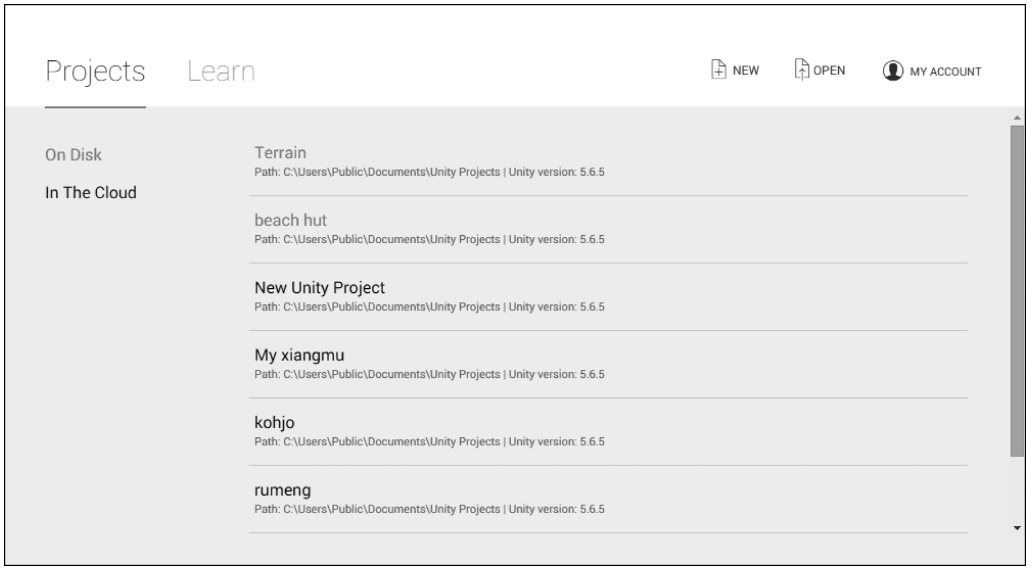


图 5.5　Unity 工程页面

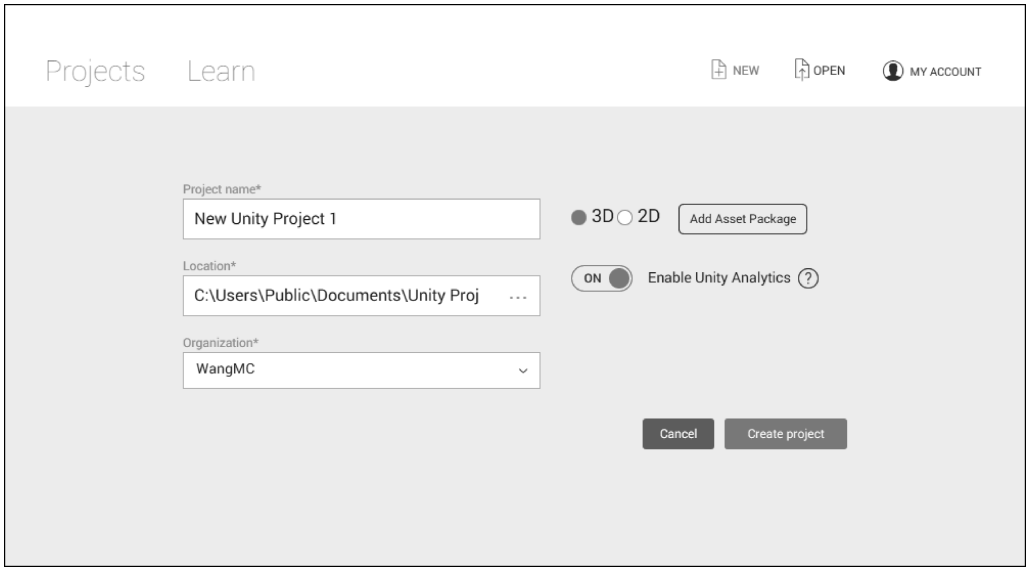


图 5.6　Unity 新工程页面

(2) 在新工程界面中,在左边第一栏中输入工程名称,第二栏中输入保存路径,在右边 3D 和 2D 单选按钮中选择将要创建的项目类型。

(3) 单击 Add Asset Package 按钮,弹出如图 5.7 所示的界面,勾选想要添加的包,单击 Done 按钮完成添加,并且会在 Add Asset Package 右侧显示添加数量。

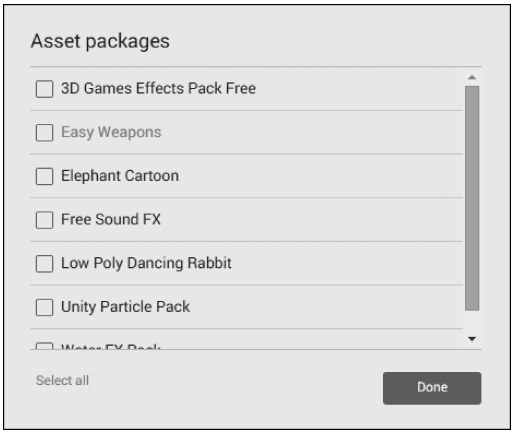


图 5.7　Unity 元素包选择

(4) 设置好后单击 Create project 按钮完成创建,然后进入 Unity 主界面,如图 5.8 所示。

主界面的左上角有一排主菜单按钮,如图 5.9 所示

这些主菜单按钮不是固定的,会因导入插件而发生相应的位置改变。它们的功能描述如下。

File: 创建、打开、保存场景和工程,发布及调试游戏。

Edit: 撤销、重做、剪裁、复制、粘贴、运行、暂停、工程设置等。

Assets: 创建导入资源等。

GameObject: 创建各类游戏对象。

图 5.8　Unity 主界面

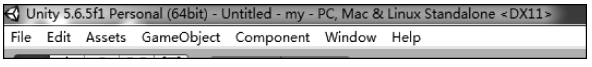


图 5.9　Unity 主菜单按钮

Component：为游戏对象添加各类组件。

Window：各类窗口。

Help：帮助文档。

Unity 的界面布局也是相当人性化的，开发者们可以按照自己喜欢的风格去设置。单击 Layout 按钮，可以选择自己喜欢的屏幕风格方式，然后单击 Save Layout 按钮保存布局，如图 5.10 所示。

Project 视图罗列了工程的所有资源。常用的资源有游戏脚本、预制体、材质、动画、纹理贴图等。这些资源要在 Hierarchy 视图中使用。在 Project 视图中右击，弹出的工程菜单如图 5.11 所示。

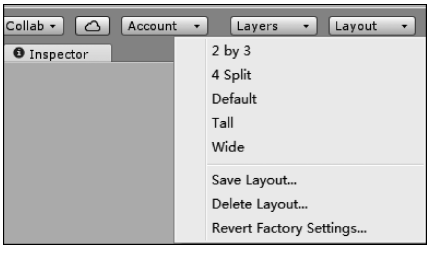


图 5.10　Unity 布局选择

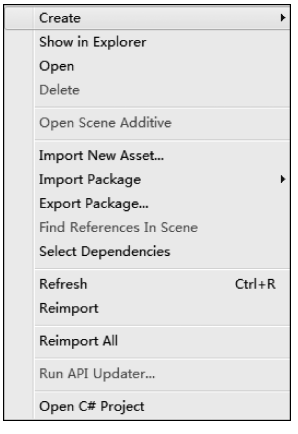


图 5.11　Unity 工程菜单

步骤 2：创建资源

Create 菜单项下的子菜单主要用于创建各种资源，如图 5.12 所示。可创建文件夹、C#、Javascript、Shader、Testing、Prefab、Audio Mixer、Material 等。

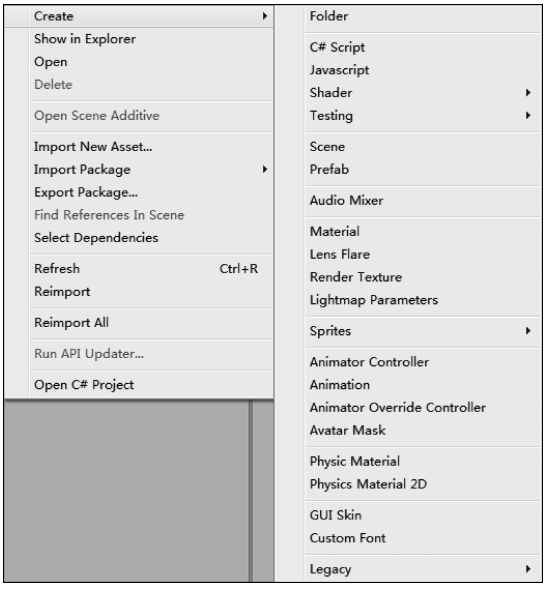


图 5.12　Unity 创建工程

步骤 3：创建材质

在 Project 视图中右击，选择 Create→Material 命令，创建一个新材质并命名为 mat，在 Inspector 视图中就会显示它的属性，如图 5.13 所示。

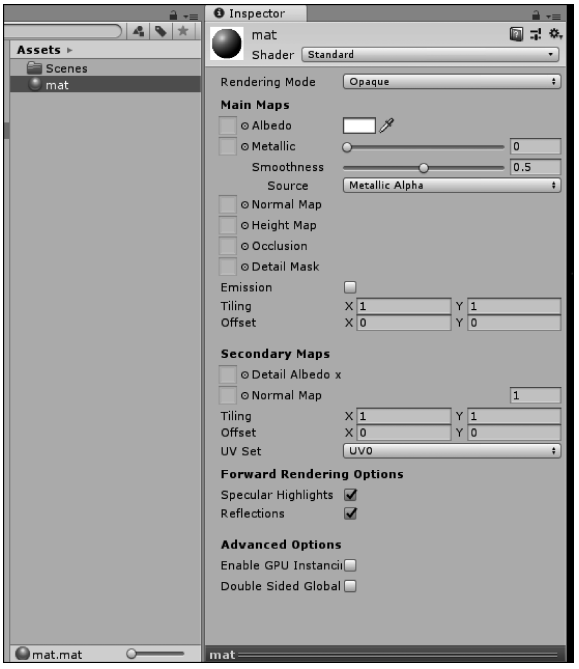


图 5.13　Unity 创建材质

单击图片中的 Albedo 左侧的圆按钮(见图 5.14),为其选择适当的贴图。贴图可以来源于网上或者 Unity 自带的资源。

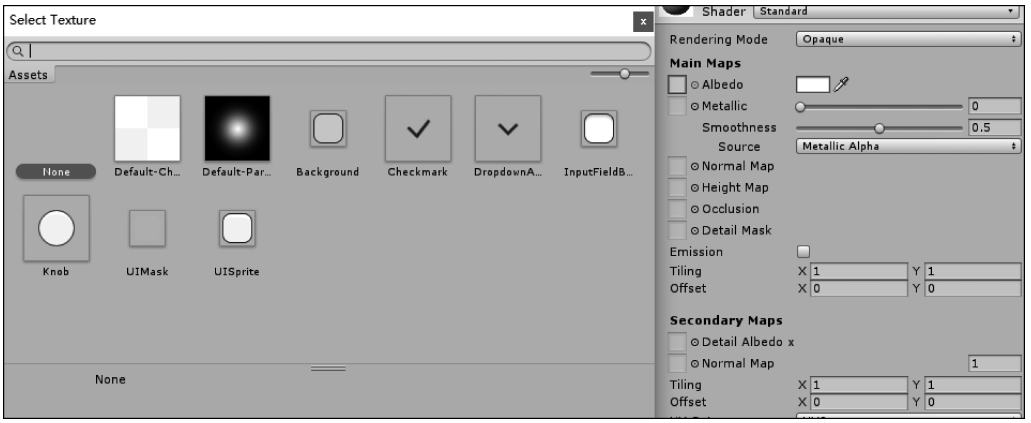


图 5.14　Unity 选择贴图

这里就以 Unity 自带的资源导入讲解:首先单击菜单栏中的 Assets 按钮,选择 Import→Environment→Import 命令,如图 5.15 所示。

导入成功后,如图 5.16 所示。

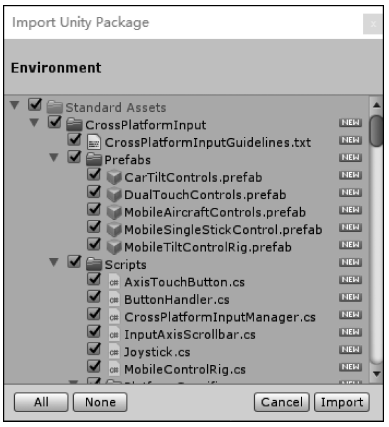


图 5.15　Unity 导入环境资源

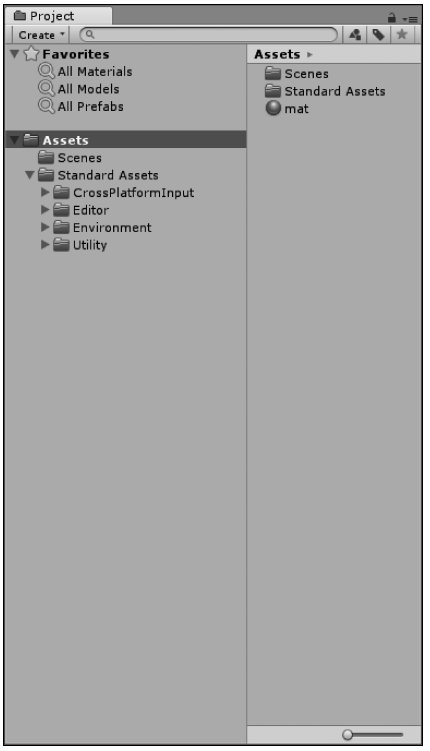


图 5.16　Unity 导入资源成功

Unity 会自动生成一个 Standard Assets 文件夹,环境资源就在里面,文件夹的名字为 Environment。此时,再单击材质球的 Albedo 左边圆按钮,就可以看到刚才的资源了,如

图 5.17 所示。

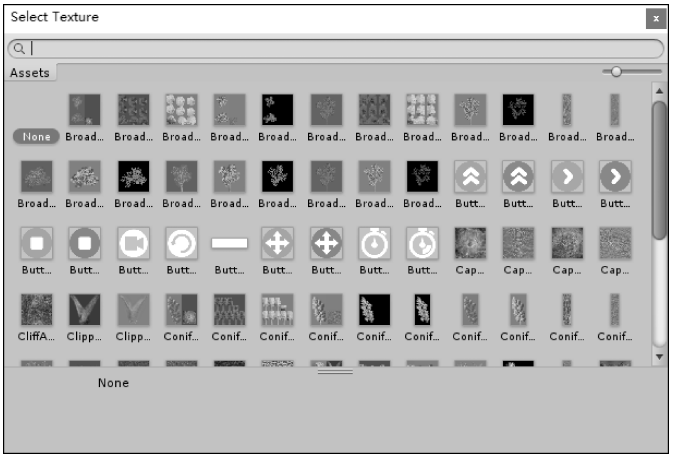


图 5.17　Unity 选择纹理(1)

选择 SimpleFoam 的图片作为纹理，如图 5.18 所示。

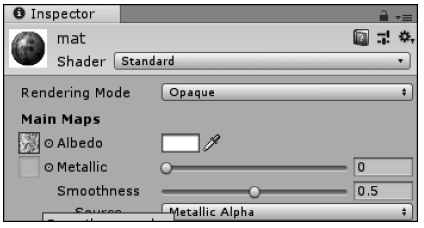


图 5.18　Unity 选择纹理(2)

步骤 4：创建立方体

(1) 单击导航菜单栏 GameObject→3D Object→Cube 命令，如图 5.19 所示。

(2) 下面要做的是把材质球 mat 给 Cube，这里有两种方式。第一种方式：用鼠标左键按住材质球 mat，直接拖放到 Cube 上，如图 5.20 所示。

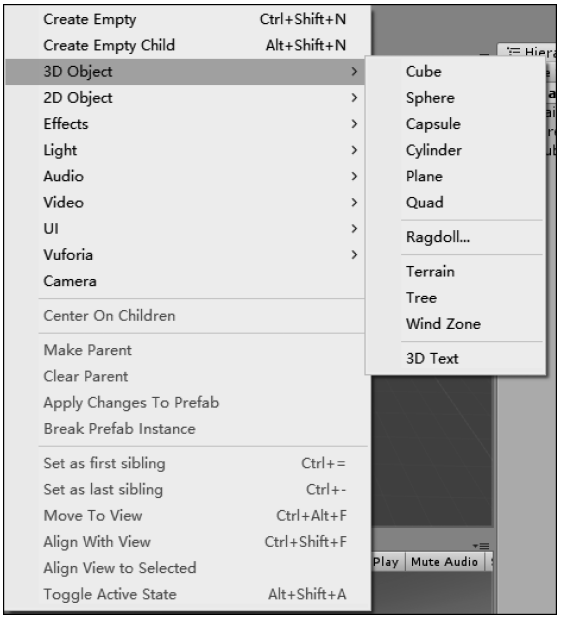


图 5.19　Unity 选择纹理(3)

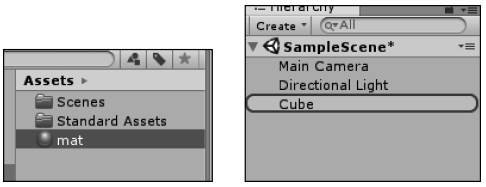


图 5.20　Unity 选择纹理(4)

第二种方式：单击 Cube 选项，然后打开 Inspector 属性中的 Material，如图 5.21 所示。

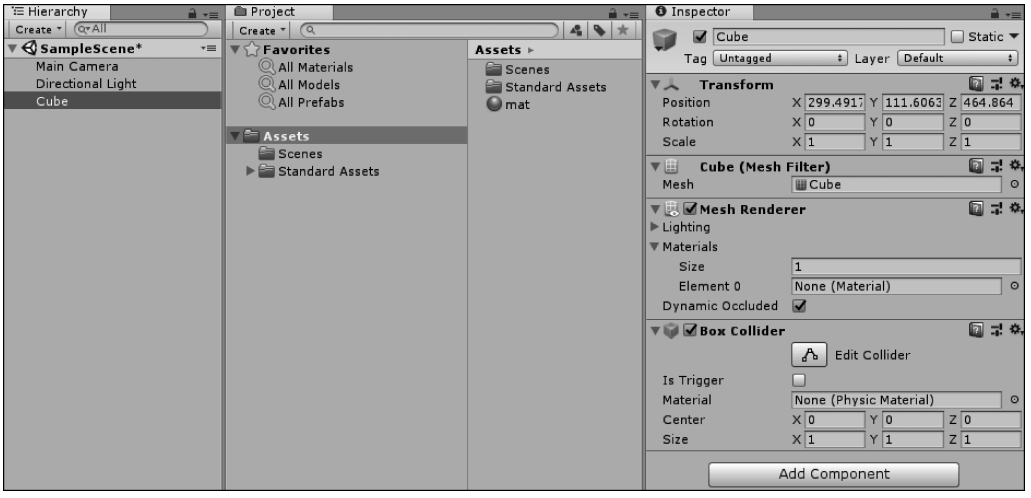


图 5.21　Unity 选择纹理(5)

(3) 最后把材质球 mat 拖放到 None(Physic Material)里面，或者单击 None(Physic Material)右边圆按钮，单击选中 mat，如图 5.22 所示。

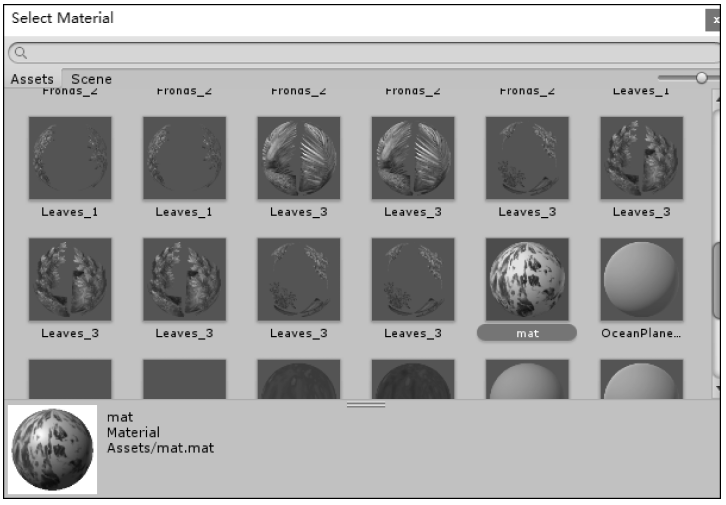


图 5.22　Unity 选择纹理(6)

(4) 最后就可以看到 Cube 中放入材质球 mat 的效果，如图 5.23 所示。

图 5.24 是在 Scene 视图和 Hierarchy 视图中显示使用了选中资源的游戏对象。

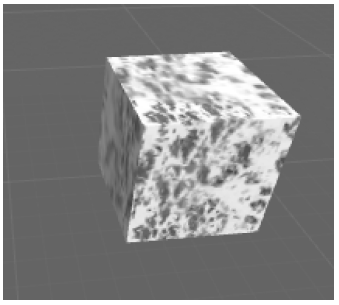


图 5.23　Unity 材质效果

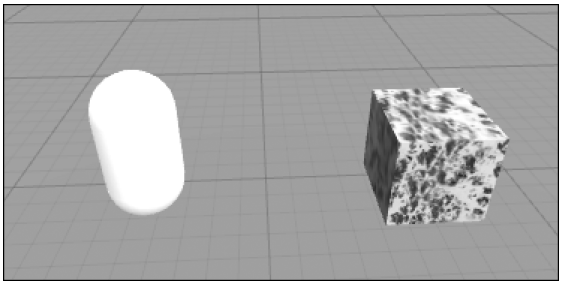


图 5.24　Scene 视图中 Find References In Scene 效果

在 Hierarchy 视图中只显示相关游戏对象，如图 5.25 所示。

步骤 5：创建胶囊体

下面创建一个胶囊体作为对比。选择该资源依赖的所有资源，例如之前创建的 mat 材质依赖于 SimpleFoam 贴图。选中 mat，右击，再单击 Select Dependencies 选项，如图 5.26 所示，资源本身和所依赖的所有资源会以蓝色高亮显示。

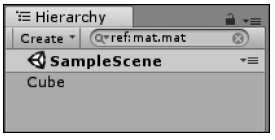


图 5.25　Hierarchy 视图

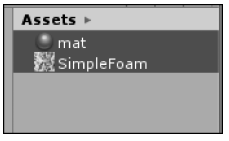


图 5.26　创建一个胶囊体

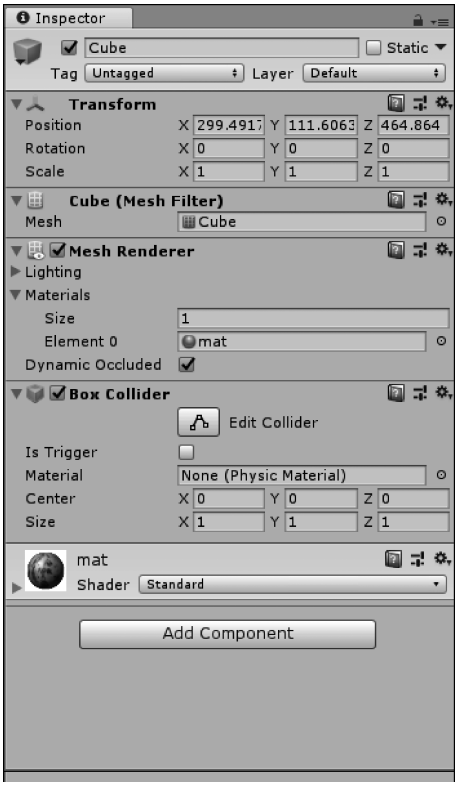


图 5.27　Inspector 中的 Cube 属性

在 Inspector 视图中，可以显示当前选中游戏对象的所有组件及组件的属性。组件脚本中的公开变量在此视图中会以属性的方式呈现，在视图属性中可以直接修改属性的值。如果属性是 GameObject 或者 Transform 等类型，可以直接将游戏对象进行拖动来完成指定操作。

例如，单击导航菜单栏的 GameObject→3D Object→Cube 命令创建一个立方体。单击 Cube 选项，在 Inspector 视图中可以看到 Cube 的属性，如图 5.27 所示。

在 Scene 视图中可以显示对游戏对象进行可视化操作的界面，用户可以通过工具及其快捷键对游戏物体进行快速操作。

工具栏：在主界面的菜单栏下方有一排工具栏，如图 5.28 所示。工具栏中的图标从左到右依次为手工具、移动工具、旋转工具、缩放工具以及 Gizmo 工具。下面分别介绍。

图 5.28　Unity 工具栏

(1) 手工具。用于控制观察摄像机,其快捷键为 Alt+Q。

如果按着 Alt 键再按住鼠标左键拖动,则是改变摄像机的位置。

如果按着 Alt 键再按住鼠标右键拖动,则是改变摄像机的观察距离。

如果按住 Control 键再拖动,则是改变摄像机的观察方向。

(2) 移动工具。单击移动工具图标或者按快捷键 Alt+W,即可选中游戏物体。当开始使用该工具后,会在选中的游戏物体中心位置显示 3 个箭头,如图 5.29 所示。

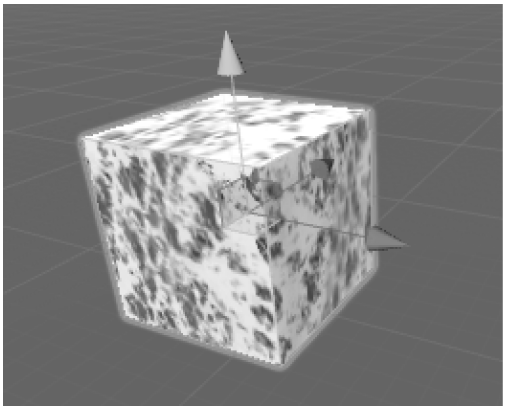


图 5.29　移动物体

这三个箭头分别表示 X 轴正方向、Y 轴正方向、Z 轴正方向。利用鼠标选中 X、Y、Z 轴箭头来移动物体位置,若单击中央则三个轴一起移动。

(3) 旋转工具。单击旋转工具图标或使用快捷键 Alt+E,可以旋转选中的物体。使用该工具的时候,物体周围会有三个线圈,如图 5.30 所示。

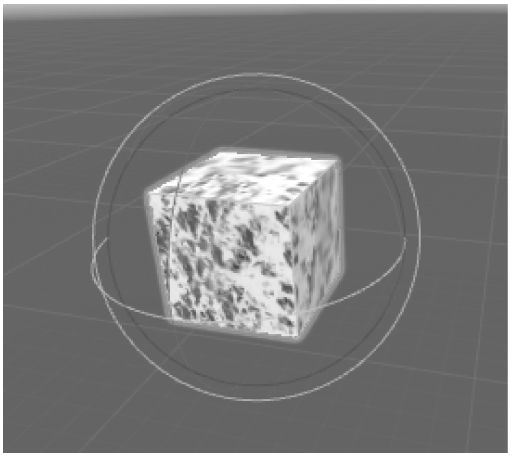


图 5.30　旋转物体

这三个线圈分别代表 X 轴旋转、Y 轴旋转、Z 轴旋转。用鼠标选中线圈(不要松开),再移动鼠标进行旋转物体操作。

(4) 缩放工具。单击缩放图标或者按快捷键 R,可以实现物体的缩小或者放大。选中该工具后,单击要缩放的物体,如图 5.31 所示。

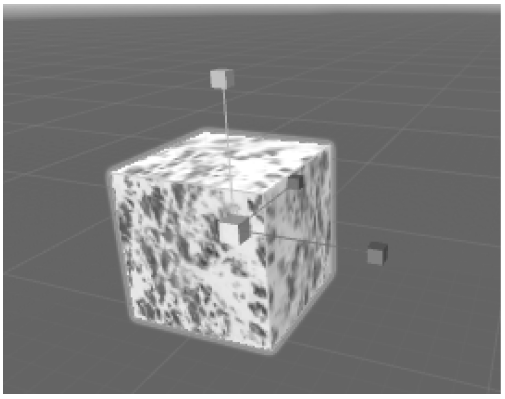


图 5.31　缩放物体

选中图 5.31 中的三条轴可分别进行 X 轴方向、Y 轴方向、Z 轴方向的缩放。选中中央则代表整体缩放。

(5) Gizmo 工具。这是一些快速操作的小工具，例如，Scene Gizmo 是在 Scene 视图右上角显示的小工具，它主要由 6 个圆锥体和 1 个立方体构成，如图 5.32 所示。

单击不同的轴可实现相应方向的切换。单击 Scene 视图左上角的 Shaded 按钮，进入 Scene 视图的渲染模式菜单，如图 5.33 所示。

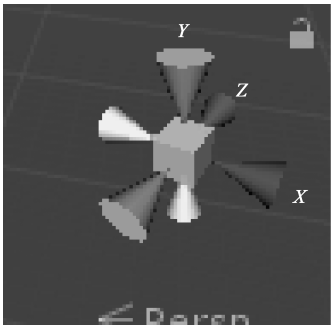


图 5.32　Scene Gizmo

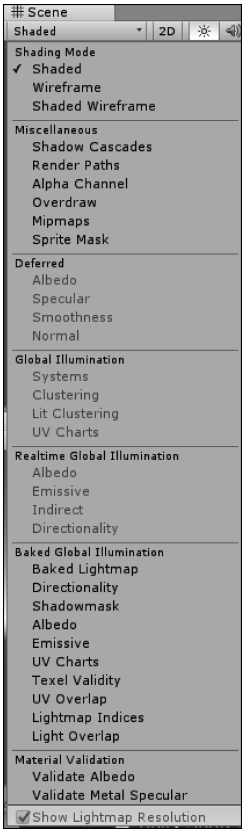


图 5.33　Scene Gizmo 菜单

默认的渲染模式是 Textured 模式，所有游戏对象的贴图都正常显示，如图 5.34 所示。

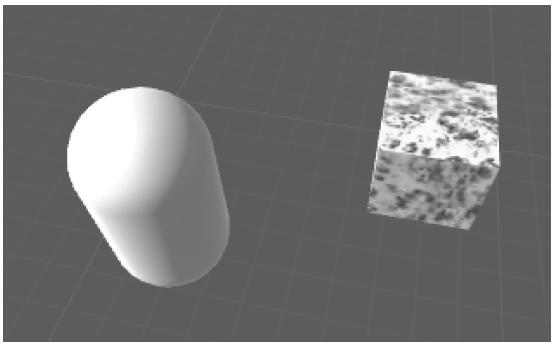


图 5.34　Textured 模式

此外还有 Wireframe 渲染模式，所有游戏对象的贴图都不显示，仅将游戏对象的网格模型以线框的形式呈现，如图 5.35 所示。

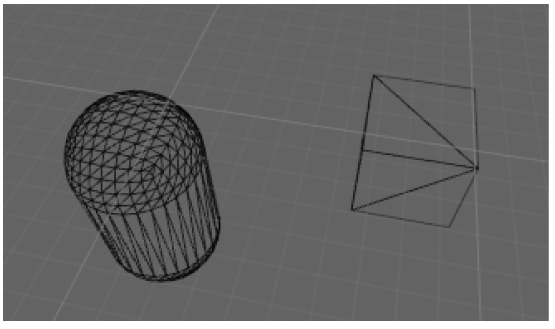


图 5.35　Wireframe 模式

在 Game 视图中显示游戏运行时的图像。运行游戏后，即可在 Game 视图看到游戏效果。Game 视图的显示取决于相机所观察到的景象。通常游戏工程中会有多个摄像机协同工作，此时显示的内容是多个相机的叠加。

单击 Game 视图标签下的按钮，会显示分辨率菜单，可以选择其中一项指定 Game 视图下的游戏画面的分辨率，如图 5.36 所示。

图 5.36　分辨率菜单

Game 视图工具栏如图 5.37 所示。其中右边两个按键的作用如下。

图 5.37　Game 视图右侧菜单

Maximize On Play：是否在运行时最大化显示。

Mute Audio：是否静音。

步骤 6：创建预制体

(1) 场景中创建 Cube，然后在 Project 视图中右击，再选择 Create→Prefab 命令，即可成功创建一个预制体，并改名为 Cube，如图 5.38 所示。

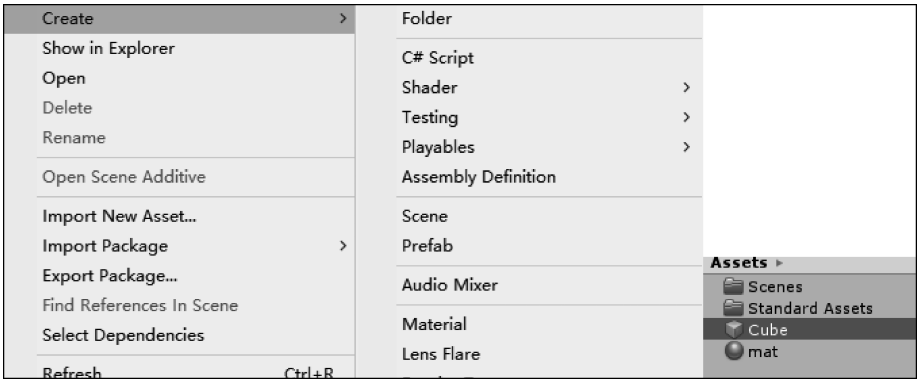


图 5.38　创建预制体(1)

(2) 把 Hierarchy 视图中的 Cube 拖动到 Project 视图中的 Cube 上，完成预制体的制作并和 Cube 预制体相关联。此时颜色会由灰色变为蓝色。单击 Hierarchy 视图中的 Cube，在 Inspector 中单击 Select 按钮，这时会高亮显示对应的预制体，如图 5.39 所示。

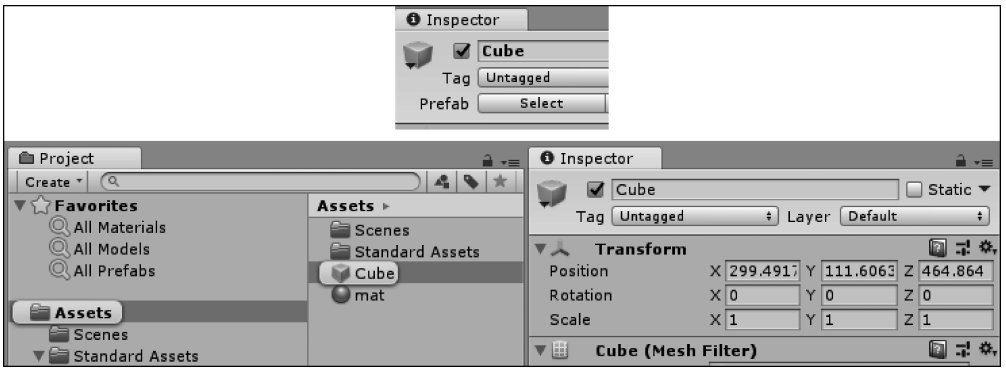


图 5.39　创建预制体(2)

(3) 预制体的实例化。实例化的过程就是将预制体复制一份放入场景里。具体地，在 Project 视图中选中 Cube，并拖动到 Inspector 视图中，可直接实例化一个对象。该操作并不是简单的复制，而是具有相关性的。

5.2 场景创建

5.2.1 游戏物体与组件

游戏物体是一个具有一定功能(组件)的模型,由以下两部分组成。

(1) 物体(基本框架):只是一个实体,但不能动,如汽车。

(2) 组件(功能):实现各种功能的代码,如汽车的"驾驶功能"组件可以使汽车运动起来,汽车的"刚体"组件使汽车具有碰撞功能。

每个物体必须包含一个 Transform 组件,如图 5.40 所示。

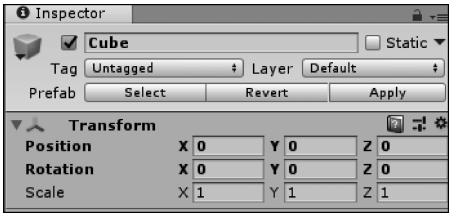


图 5.40 Transform 组件

其他的组件,根据需要进行选择。脚本在经过编译并实施到游戏物体之后,也变成了一种组件,脚本组件是一种用户可以自己创建的组件。在场景中的物体,会按规则把其身上的所有组件应用一遍,以改变自身属性。

5.2.2 场景视图操作

在场景视图中可以进行设置图像质量、场景浏览、设置灯光、设置摄像机等操作。下面分别进行介绍。

1. 质量设置

选择菜单 Edit→Project Settings→Quality(Levels:Fastest、Fast、Simple、Good、Beautiful、Fantastic)命令,如图 5.41 所示。

单击 Quality 命令,打开 QualitySettings 视图,如图 5.42 所示。

2. 场景浏览

选中 Scene 视图,但不要选择任何物体,可以用方向键进行前、后、左、右移动。

慢速移动:单击上下箭头进行前后移动;单击左右箭头进行左右移动。

快速移动:一直按住 Shift 键,然后单击上下箭头进行前后移动;单击左右箭头进行左右移动。

选中小手图标,然后按住鼠标左键可拖动整个场景。

滚动鼠标中轮,则整个场景前后移动。

按住鼠标右键,则可旋转整个场景。

此外还可以采用飞行模式进行场景浏览:按住鼠标右键,按 W、A、S、D 键可前、后、左、右浏览当前场景,按 Q、E 键可上下移动场景。

按住 Shift+Ctrl 组合键,即可移动物体。

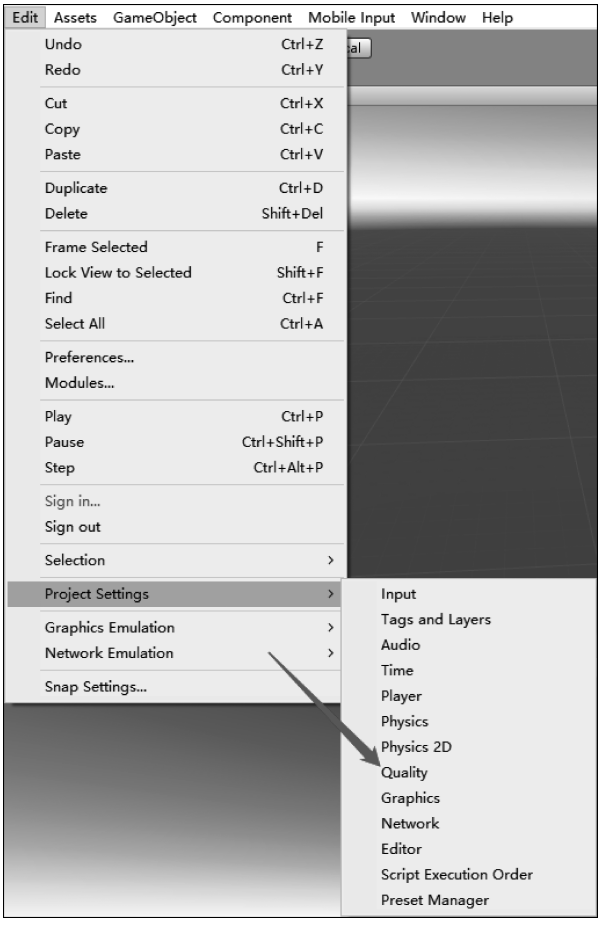


图 5.41　质量设置

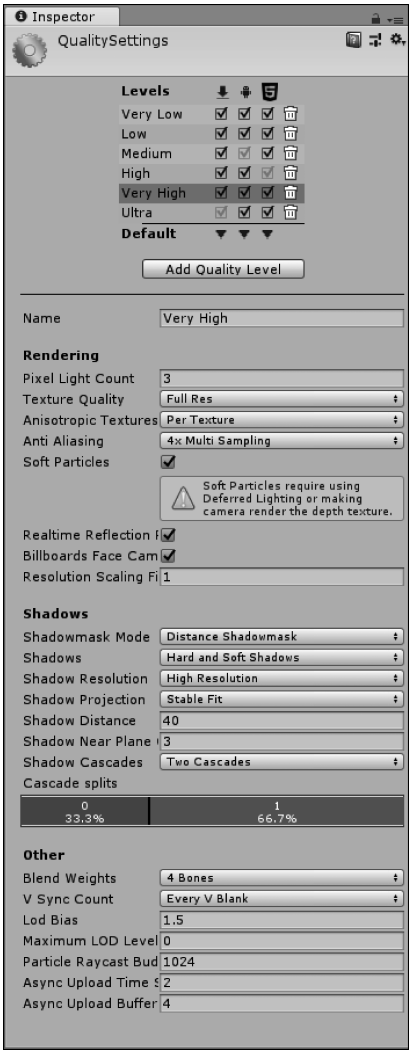


图 5.42　质量设置菜单

3. 设置灯光

场景的颜色和基调由灯光定义。

灯光类型如图 5.43 所示，其中包括：

(1) 点灯光(Point)：模拟蜡烛和灯泡的效果。

(2) 聚光灯(Spot)：模拟手电筒或汽车头灯的效果。

(3) 方向灯(Directional)：可平行的发射光线，可以模拟太阳的效果。

(4) 区域灯(Area <baked only>)：主要在创建灯光贴图时使用。

(5) 在灯光的属性中，可以设置灯光的阴影(Shadow Type：No Shadows、Hard Shadows、Soft Shadows)。

(6) 绘制光晕项(Draw Halo)：打开灯光的光晕效果。

(7) 渲染模式(Render Mode)：通过灯光设置影响灯光的显示效果及操作效率。Auto

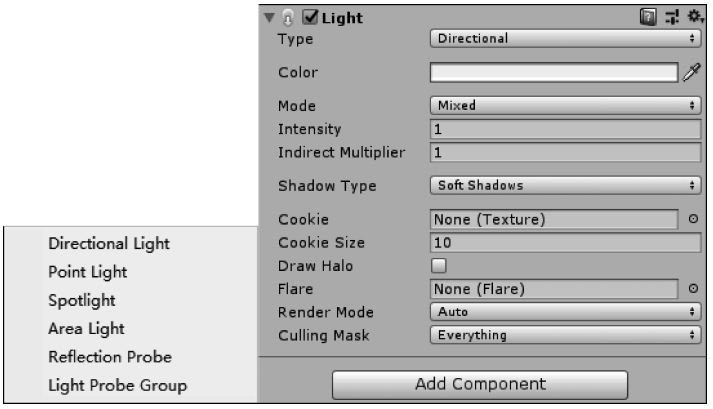


图 5.43　灯光类型

项是在游戏运行时，根据用户设置的质量来决定其渲染效果。可选项有 Auto、Important、Not Important。

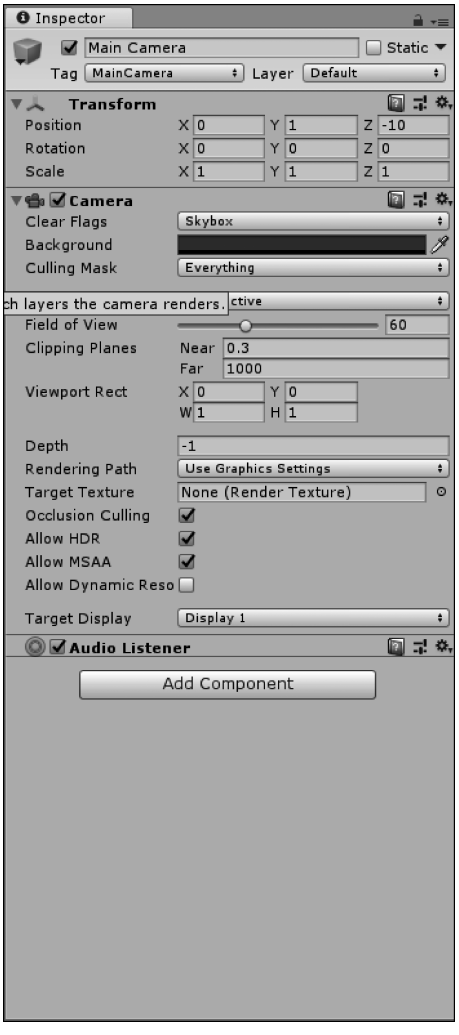


图 5.44　摄像机

（8）Culling Mask：主要通过层的方式，设置场景中的哪些物体可以被此灯光照亮，以及哪些不可以。

4. 设置摄像机

每一个场景中至少存在一个摄像机。摄像机相当于人的眼睛，它把所看到的影像输出到屏幕上。若想创建各种摄像机效果，如创建小地图的效果、分屏效果等，可在一个场景中使用多个摄像机。

在赛车类游戏中，可以使用大的摄像机视野来增加速度感。此外，还可以使用正交摄像机视图创建图形用户界面。

摄像机的设置界面如图 5.44 所示，下面分别介绍。

（1）Clear Flags(清除标签项)：主要设置摄像机在渲染过程中如何设置其背景，以及在渲染过程中如何产生颜色、深度信息等。其中，主要选项如下。

① Skybox（天空盒）和实体颜色项（Solid Color）：主要设置摄像机在渲染过程中对于屏幕中不存在物体的部分，也就是场景空白地方的处理。例如，若场景中的某些部分不存在物体时，可以使用天空盒项使其成为天空盒的内容，或者也可以直接设置这些不存在物体的部位为实体颜色，也就是此处设置的背景颜色。

② Depth only(深度项)：常用于一个场景中存在多个摄像机的情况，此时可以设置摄像机的渲染顺序，进而将一个摄像机的内容叠加在另外一个摄像机上。

③ Don't Clear(不进行清除项)：既不会清除摄像机的颜色信息，也不会清除摄像机的深度信息。这样可将每一帧的渲染都叠加在上一帧上，从而形成一种涂抹拖尾的效果(仅在一些特殊的材质效果中使用此项)。

④ Culling Mask(剔除遮罩)：主要通过层的方式，设置场景中的哪些物体可以被渲染，以及哪些不可以。

⑤ Projection(投影项)：设置当前的摄像机是透视摄像机还是正交摄像机。

⑥ Clipping Planes(剪切平面项)：主要设置摄像机远近剪切平面的位置。只有在两个剪切平面中的物体才会被摄像机渲染，即被看到。

(2) Viewport Rect(视图矩形)：主要用于场景中存在多个摄像机时，设置其中一个摄像机视口的长方形尺寸，使其更好地叠加在另一个画面上。例如，显示小地图效果时，可调节此项中的 W 值，使其视图变小，以显示在另外一个摄像机的视角中。

(3) Depth(深度)：用于场景中有多个摄像机的情况，深度值较大的摄像机渲染的画面一定会叠加在深度值小的摄像机渲染的画面上。

(4) Rendering Path(渲染路径)：用于设置整个场景的渲染质量。

(5) Target Texture(目标纹理)：把当前摄像机渲染的内容保存在一个纹理中，然后此纹理可以实施在其他表面上，创建如水面的反射效果。

(6) Allow HDR：用于打开 HDR 效果。

5.2.3 游戏地形

本节介绍游戏地形的创建。

步骤 1：创建一个新工程

单击菜单栏 Assets→Import Package，选择 Characters(人物)和 Environment(环境)命令，如图 5.45 所示。

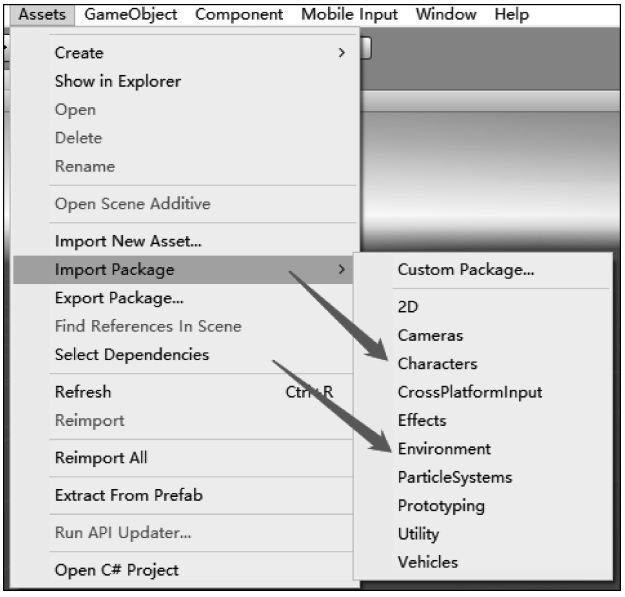


图 5.45 创建新工程

步骤 2：导入资源文件

单击后弹出如图 5.46 所示框，单击 Import 按钮。

步骤 3：设置地形参数

成功导入资源文件后，如图 5.47 所示。

此时，在 Standard Assets 里面会有 Characters 和 Environment 选项，接下来，在 Hierarchy 面板上右击选择 3D Object→Terrain。这样面板 Hierarchy 中就会出现新建的地形了。

单击地形 Terrain，在 Inspector 视图中会显示其地形工具，如图 5.48 所示。

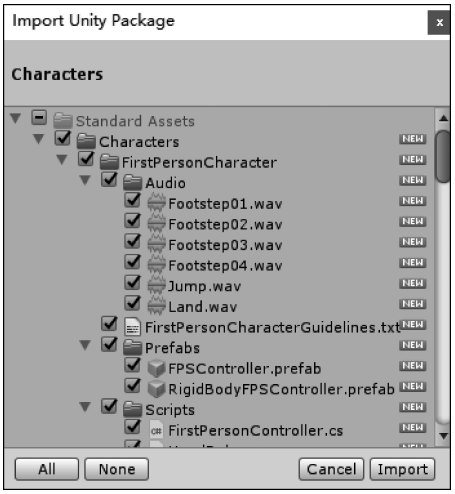


图 5.46　导入游戏地形

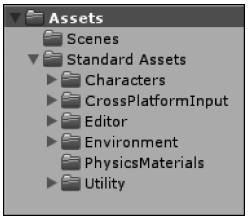


图 5.47　导入成功

图 5.48　地形工具

在该工具栏最右边的齿轮图标表示地形设置，单击会出现如图 5.49 所示视图。

在这个窗口中可以进行地形的参数设置，包括：

(1) Terrain Width：全局地形总宽度，单位为 Unity 统一单位(m)。

(2) Terrain Height：全局地形允许的最大高度，单位为 m。

(3) Terrain Length：全局地形总长度，单位为 m。

(4) Heightmap Resolution：全局地形生成的高度图的分辨率。

(5) Detail Resolution：全局地形所生成的细节贴图的分辨率，数字越小性能越好。但是也要考虑质量。

(6) Control Texture Resolution：全局把地形贴图绘制到地形上时所使用的贴图分辨率。

(7) Base Texture Resolution：全局用于远处地形贴图的分辨率。

步骤 4：定制地形

如果有美术人员制作好的图，则可以直接导入，如图 5.50 所示。

单击 Import Raw Heightmap 按钮，选中需要的资源后，会弹出属性设置框，如图 5.51 所示。

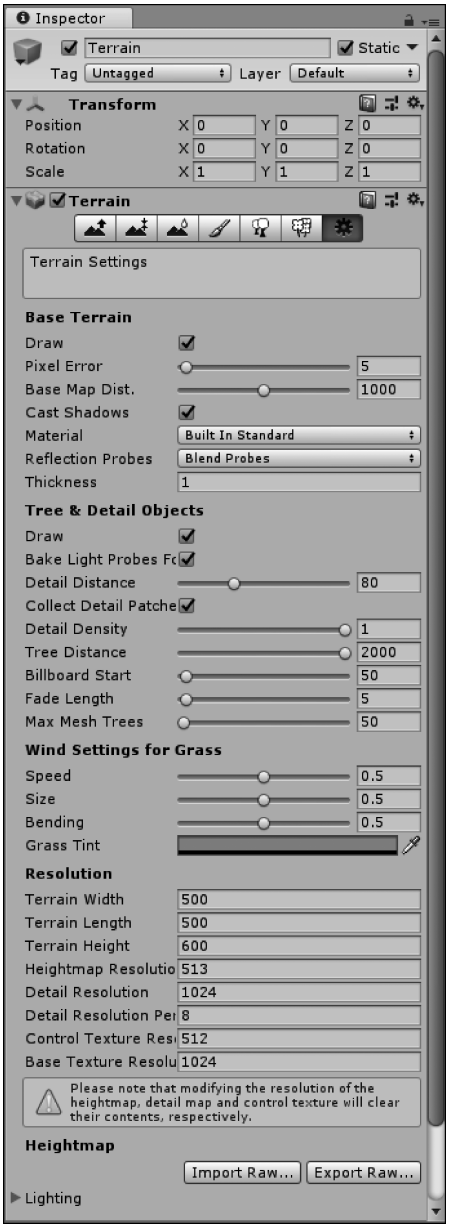


图 5.49　设置地形参数

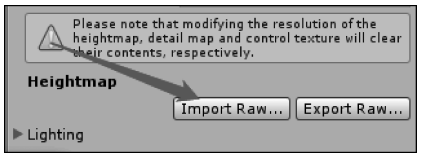


图 5.50　导入地形

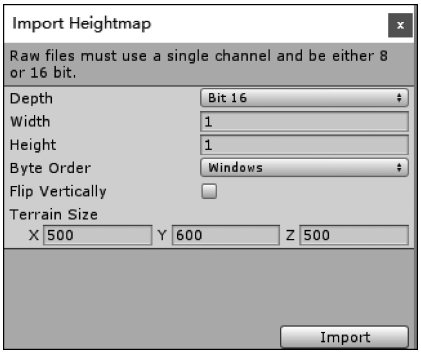


图 5.51　属性设置

步骤 5：绘制地形

(1) 在 Hierarchy 面板中选中地形。在 Inspector 视图中查看信息，以下 7 个横排按钮就是绘制地形工具，如图 5.52 所示。

Paint Texture 功能从左往右依次是提高和降低高度(此功能配合 Shift 键可以使地形瞬间平整)，绘制目标高度、平滑高度，绘制地形，绘制树木，绘制花草，设置。

Brushes 区包含各种样式的笔刷，可以用来控制贴图和地形风格。

Details 区包含笔刷设置，可以通过 Edit Details 添加笔刷材质。Brush Size 用来控制笔

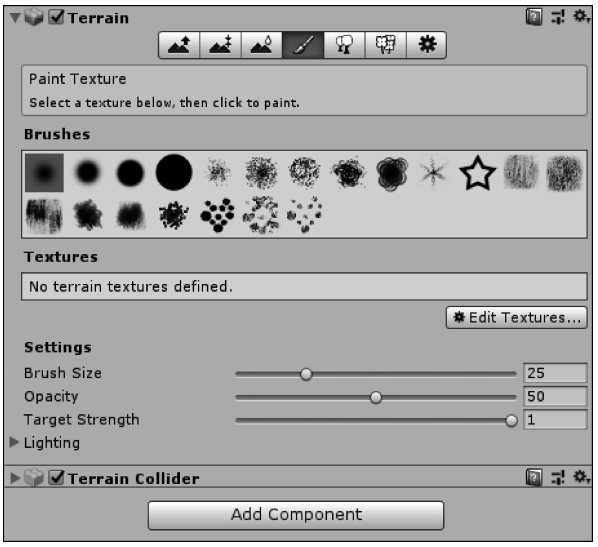


图 5.52　绘制地形工具

刷大小；Opacity 用来控制贴图使用的纹理的透明度或者说浓度；Target Strength 用来调整目标强度，强度越小，那么贴图纹理所产生的影响越小。

(2) 使用系统自带的材质为地形贴图。

创建 Terrain 后，在 Project 面板右击选择 Import Package→Terrain Assets(包含树木、绿草资源)，在 Hierarchy 面板中选中 Terrain。

在 Inspector 面板中的 Terrain 下选择笔刷。单击 Edit Texture 按钮，再单击 Add Texture 按钮，在弹出的对话框中选择左边的 Albedo(RGB)Smoothness(A)，如图 5.53 所示。

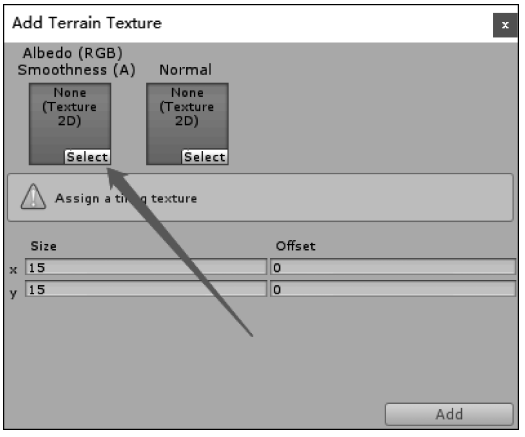


图 5.53　加入地形纹理

(3) 弹出材质列表，选择其中之一，根据地形贴上材质。第一次是完全覆盖，以后的导入材质不再覆盖首次的材质，可通过画笔控制进行材质覆盖。如图 5.54 所示，单击图 5.53 中的 Select 按钮后，选择一张图片。如果是第一张图片，将完全覆盖地形。

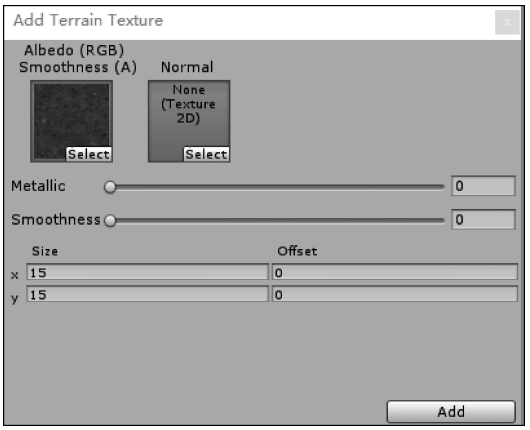


图 5.54　选择图片

(4) 选择图片。单击 Add 按钮后的效果如图 5.55 所示。

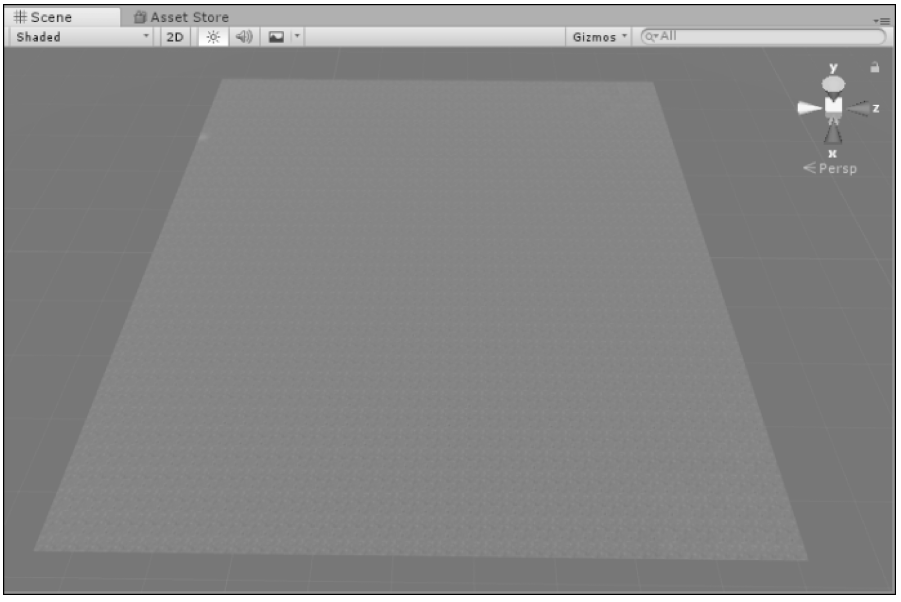


图 5.55　地形效果

例如要在地形图上绘制山脉,单击地形工具的第二个按钮,如图 5.56 所示,在 Brushes 中选择自己喜欢的风格,这里的 Height 设置为 100,然后单击 Flatten 按钮。整个地形的绘制只需要单击一次 Flatten 按钮。

这个时候用鼠标左键就可以在地形上绘制山脉了。要注意的是,这个时候 Height 默认值是 100(相当于平面),如果绘制山脉,可以手动输入数值大于 100 即可;如果要绘制深坑或者沟渠,输入的数值要低于 100,最低不能低于 0。也可以用 Height 右边的滑块来调节高度。效果图如图 5.57 所示。

可以看到,鼠标在地形上操作的时候,当高度或者深度达到 Height 值时,就会变成平面。改变 Height 的高度或者深度就可以继续操作了(注:若不想改变数值,不要单击 Flatten 按钮)。

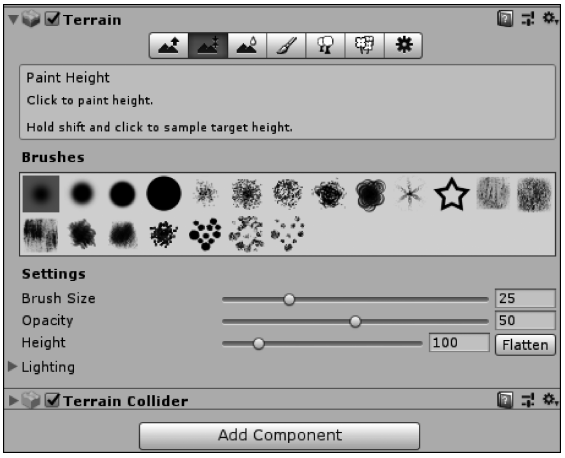


图 5.56　地形工具

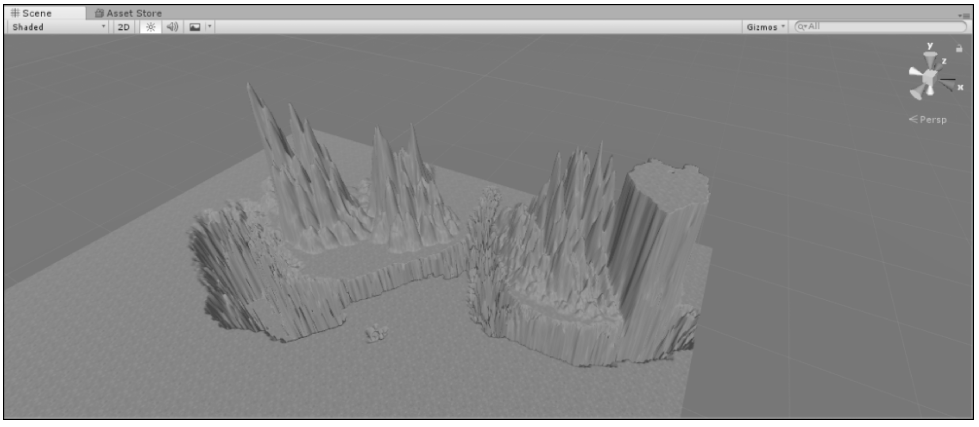


图 5.57　地形效果

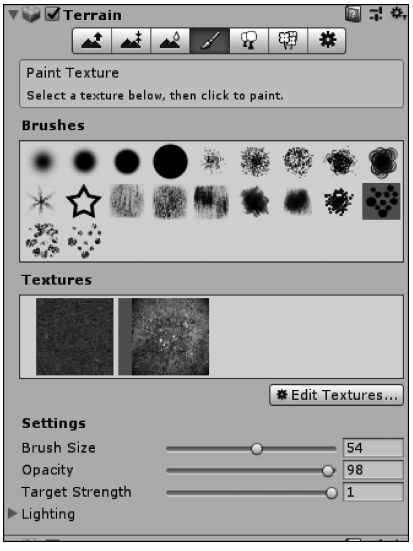


图 5.58　编辑纹理

绘制完山脉和沟渠后，如果想改变山体颜色，可以继续单击 Edit Textures 按钮。单击 Add Texture→Select→挑选图片。选完图片后再单击该图片，如图 5.58 所示。

由此可见，选中图片后，图片会呈现高亮状态。此时可给山脉添加颜色。因为不是第一张添加的图片，所以不会是全局图片，可以通过 Brush Size 调节上色范围和通过 Opacity 改变上色力度。山脉涂色效果如图 5.59 所示。

如图 5.59 所示，左边是 Opacity 值设为 5 后涂的，右边是值设为 90 后涂的，两者的颜色浓度是不一样的。如果想继续涂上不同的颜色，可以选择继续添加图片。另外，还可以通过改变 Target Strength 值改变纹理强度。

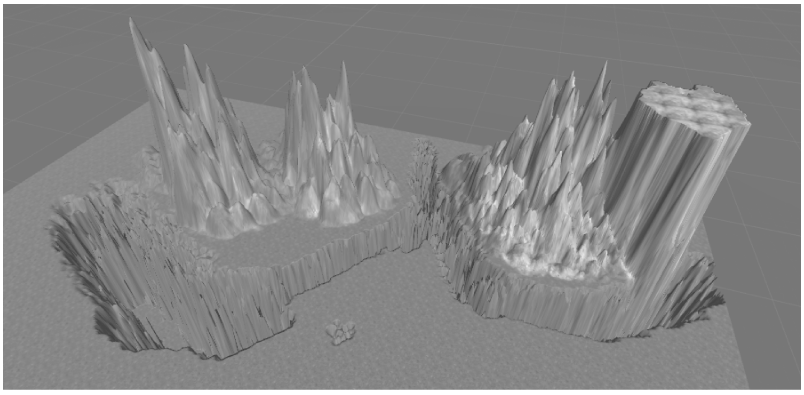


图 5.59　地形效果

(5) 删除图片的操作,如图 5.60 所示,单击 Remove Texture 按钮。

(6) 添加树木：在工具栏单击树木按钮。单击 Edit Tress→Add Tree 命令,弹出的对话框选择如图 5.61 所示,单击 Tree Prefab 右边的圆形按钮,可在里面选择自己喜欢的树木风格。

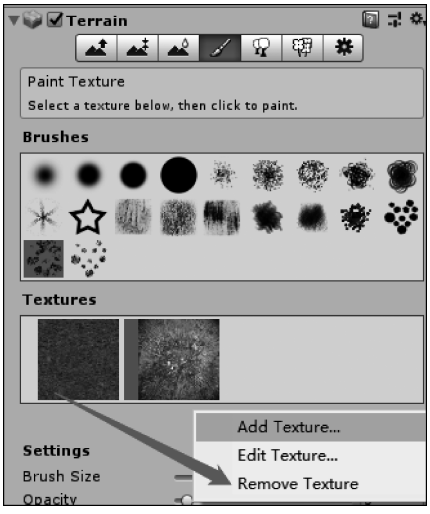


图 5.60　删除纹理

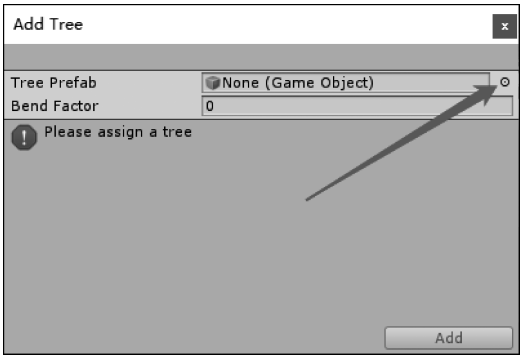


图 5.61　添加树木

然后在地形上单击,通过修改 Tree Density(树的密度)和 Brush Size(范围)改变种树的范围和大小,如图 5.62 所示。

关于 Tree 的 Settings 参数详解如下。

Bush Size：笔刷的半径,以地形单位 m 计算。

Tree Density：树木密度,值越大树木越多。

Color Variation：每棵树的颜色所能够使用的随机变量值。

Tree Height：树的基准高度。

Tree Height-Variation：树高的随机变量。

Tree Width：树的基准宽度。

Tree Width-Variation：树宽的随机变量。

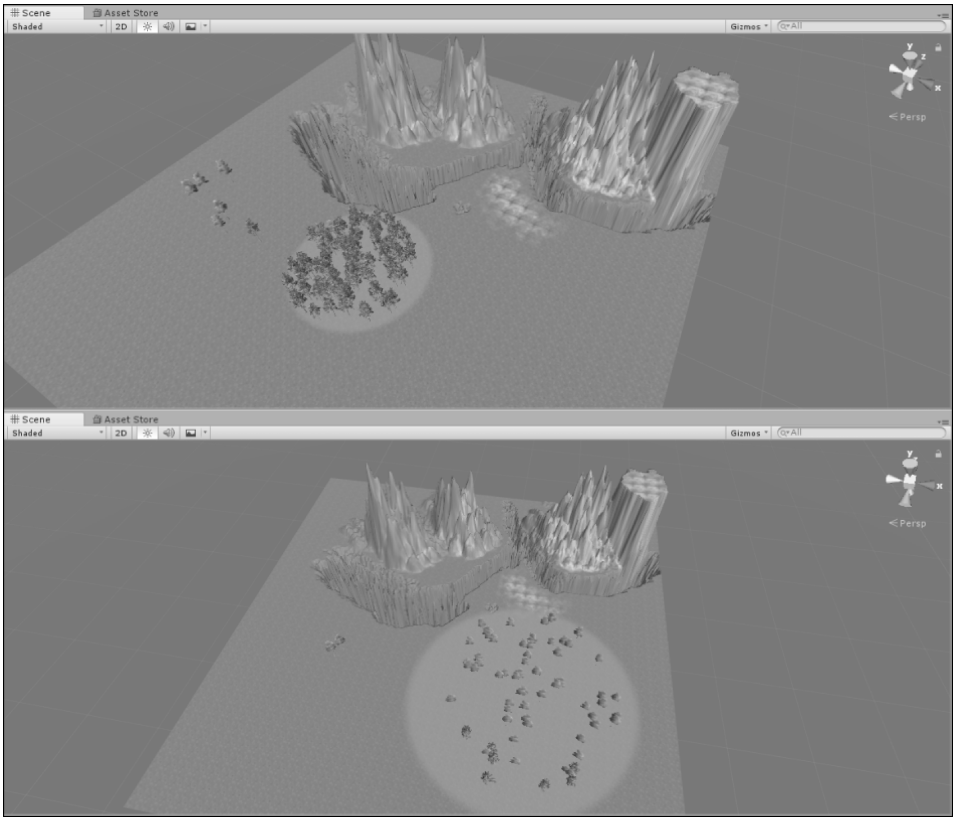


图 5.62　添加了树木的效果

(7) 添加草,如图 5.63 和图 5.64 所示。

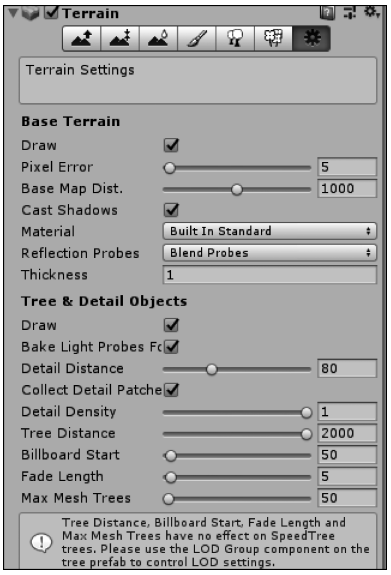


图 5.63　添加草

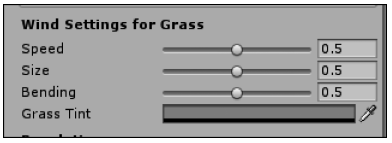


图 5.64　草地设置

以下 3 项都是地形基本渲染设置。

Pixel Error：像素误差，较高的值可能渲染较快，但是贴图可能不是非常精确。

Base Map Dist：若贴图到摄像机的距离超过此值，会使地形贴图以低分辨率显示。

Cast Shadows：让地形产生阴影，例如山峰产生的阴影。

以下 6 个参数为树木或者细节对象渲染参数设置。Draw 选项表示是否渲染除地形以外的对象。当需要在各种物体的地形上调整时，非常有用。

Detail Distance：当距摄像机超过这一距离时，细节停止显示。

Detail Density：详细密度。更细小的渲染粒度。

Tree Distance：当距摄像机的距离超过该值时，树木停止显示。

Billboard Start：当距摄像机的距离超过该值时，树木以广告牌形式开始显示。

Fade Length：树木从网格过渡到广告牌的距离。

Max Mesh Trees：使用网格形式进行渲染的最大树木数量。

以下 4 项为风力设置参数。

Speed：风吹过草地的速度。

Size：同一时间受到风影响草的数量。

Bending：草跟随风进行弯曲的强度。

Grass Tint：对于地形上使用的所有草和细节网格的总体渲染颜色。

单击 Edit Details→Add Grass Texture 命令后会弹出一个对话框，如图 5.65 所示。

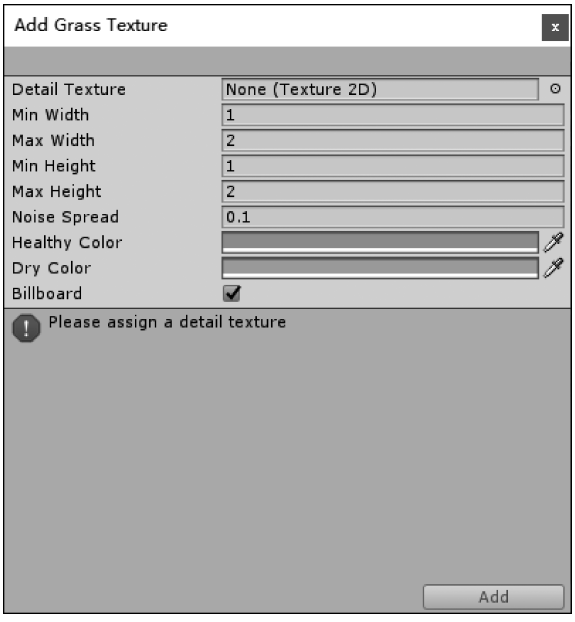


图 5.65　细节编辑

单击 Detail Texture 右边的圆形按钮，选择草的图片。草的图片如图 5.66 所示。

然后改变尺寸，单击 Add 按钮，效果如图 5.67 所示。

(8) 添加 Wind(风)：在 Hierarchy 面板上右击，选择 3D Object→Wind Zone 选项，如图 5.68 所示。

就可以看到树木和草随风摆动，形成很真实的三维场景效果。

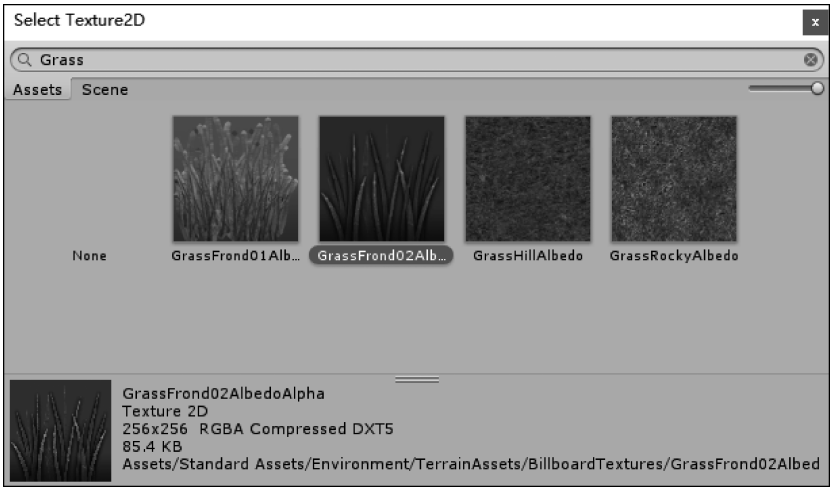


图 5.66　选择草纹理

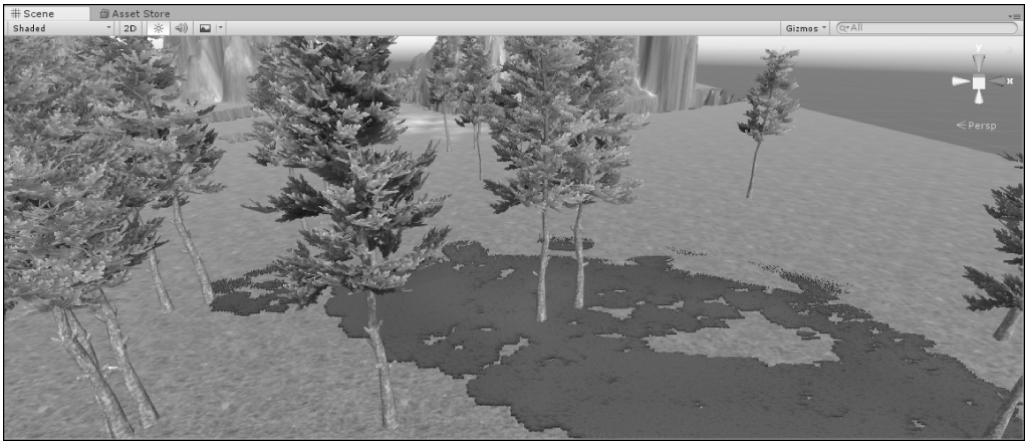


图 5.67　添加了草地的效果

(9) 游戏体验。

如果想以第一人称视角体验，如图 5.69 所示，依次选择 Standard Assets→Characters→FirstPersonCharacter→Prefabs，找到第一人称控制器，将 FPSController 直接拖入场景中即可使用。也可以通过 Project 工程下的搜索功能直接查找。

这里的 FPSController 是第一人称控制器，上面有写好的脚本，可以直接用 W、A、S、D 按键控制前、后、左、右，用 Space 键跳，按着快捷键 Shift＋W 加速跑，用鼠标控制朝向。如果想要改变速度，则选中 FPSController，查看 Inspector 中的脚本，对速度进行修改，如图 5.70 所示。

最后直接单击 Play 按钮运行，进行游戏体验。

这样，一个简单的游戏场景就完成了。

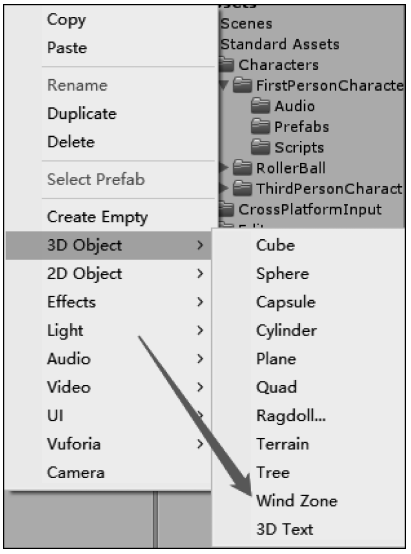


图 5.68　编辑风力

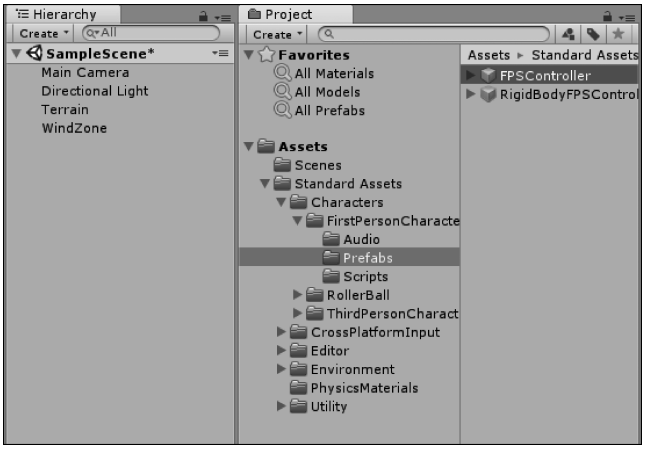


图 5.69　第一人称视角体验

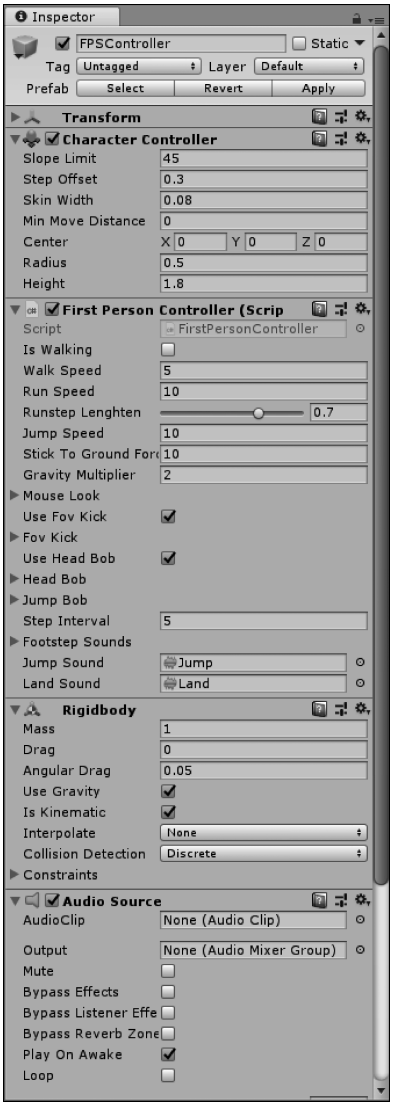


图 5.70　FPSController 设置

5.3　物理引擎

刚体能让游戏对象被物理引擎所控制，它能通过受到推力和扭力实现真实的物理表现效果。所有游戏对象必须包含刚体组件实现重力，并通过脚本施加力或者与其他对象进行交互，这一切都通过 NVIDIA 公司的 PhysX 物理引擎实现。

5.3.1　属性

刚体让你的游戏对象处于物理引擎的控制之下，从而实现真实碰撞及其他各种效果。

通过给刚体施加外力移动它，与以前的通过设置移动其位置具有非常大的不同。

这两者之间最大的差异在于力(Forces)的使用，刚体能接受推力和扭力，变换不可以。变换同样可以实现位置变化与旋转，但这与通过物理引擎实现是不一样的。给刚体施加力移动它的同时也会影响对象的变换数值，这也是为什么只能使用这两者之一的原因，如果同时操作了刚体的变换，那么在执行碰撞和其他操作的时候会出问题。

必须显式地将刚体组件添加到游戏对象上，通过选择菜单项 Component→Physics→Rigidbody 即可添加，之后对象就处于物理引擎控制之下，其会受到重力的影响而下落，也能够通过脚本来受力，不过可能还需要添加一个 Collider 或者 Joint 让它的表现更符合你的期望。

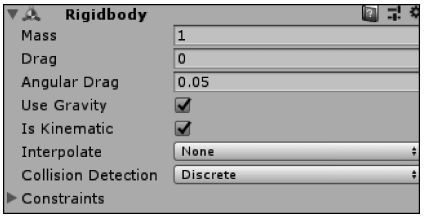


图 5.71　属性设置

Unity 中的物理引擎的属性设置如图 5.71 所示。下面对各项属性分别进行介绍。

(1) Mass：质量，单位为 kg，建议不要让对象之间的质量差达 100 倍以上。

(2) Drag：空气阻力，0 表示没有阻力，infinity 表示立即停止移动。

(3) Angular Drag：扭力的阻力，数值意义同上。

(4) Use Gravity：是否受重力影响。

(5) Is Kinematic：是否为 Kinematic 刚体，如果启用该参数，则对象不会被物理所控制，只能通过直接设置位置、旋转和缩放操作，一般用来实现移动平台，或者带有 Hinge Joint 的动画刚体。

(6) Interpolate：如果刚体运动时有抖动，尝试修改此参数，None 表示没有插值，Interpolate 表示根据上一帧的位置来做平滑插值，Extrapolate 表示根据预测的下一帧的位置来做平滑插值。

(7) Freeze Rotation：如果选中该选项，那么刚体将不会因为外力或者扭力而发生旋转，只能通过脚本的旋转函数来进行操作。

(8) Collision Detection：碰撞检测算法，用于防止刚体因快速移动而穿过其他对象。

(9) Constraints：刚体运动的约束，包括位置约束和旋转约束，勾选表示在该坐标上不允许进行此类操作。

5.3.2　详细描述

本节对其他在 Unity 中被使用的物理引擎属性进行详细描述。

(1) Parenting：当一个对象处于物理引擎控制之下时，它的运动将会与其父对象的移动半独立开。如果移动任意父对象，将会拉动刚体子对象。另外，刚体在重力及碰撞影响下还会下落。

(2) Scripting：控制刚体的方法主要是通过脚本来施加推力和扭力，通过在刚体对象上调用 AddForce()和 AddTorque()方法。再次注意，当使用物理引擎来控制刚体的时候，不要直接操作对象的变换数值。

(3) Animation：主要用于创建纸娃娃效果，完成在动画与物理控制之间的切换。可以将刚体设置为 Is Kinematic，当设置为 Kinematic 模式时，它将不再受到外力影响。这时只能通过变换方式来操作对象，但是 Kinematic 刚体还会影响其他刚体，但它自己不会再受物理引擎控制。例如，连在 Kinematic 刚体上的 Joints 还会继续影响连接的另一个非 Kinematic 刚体，同时也能够给其他刚体产生碰撞力。

(4) Colliders：碰撞体是另一类必须手动添加的组件，用来让对象能够发生碰撞。当两个刚体接触到一起的时候，除非两个刚体都设置了碰撞属性，否则物理引擎是不会计算它们的碰撞的。没有碰撞的刚体在进行物理模拟的时候将会简单地穿过其他刚体。

(5) Composed Colliders：由多个基本的碰撞体对象组合而成，形成一个独立的碰撞体对象。当你有一个复杂的模型，而又不能使用 Mesh Collider 的时候就可以使用组合碰撞体。

(6) Continuous Collision Detection：CCD 用来防止快速移动的物体穿过其他对象。

当使用默认的离散式碰撞检测时，如果前一帧对象在墙这一面，下一帧对象已到了墙的另一面，那么碰撞检测算法将检测不到碰撞的发生。若将该对象的碰撞检测属性设置为 Continuous，这时碰撞检测算法将会防止对象穿过所有的静态碰撞体；设置为 Continuous Dynamic 还将会防止穿过其他设置为 Continuous 或者 Continuous Dynamic 的刚体。CCD 只支持 Box、Sphere 和 Capsule 的碰撞体。

(7) Use The Right Size：当使用物理引擎的时候，游戏对象的大小比刚体的质量更重要。如果发现刚体的行为不是你所期望的，比如移动太慢、漂浮，或者不能正确地进行碰撞，可尝试修改模型的缩放值。Unity 的默认单位是 unit(1unit＝1m)，物理引擎的计算也是按照这个单位来的。例如，一个摩天大楼的倒塌与一个由积木搭成的玩具房子的倒塌是完全不一样的，因此不同大小的对象在建模时都应该按照统一的比例。

对于一个人类角色模型来说，假设他有 2m 高，可以创建一个默认高度为 1m 的 Box 来作为参照物，所以一个角色应该是 Box 的两倍高。当然，如果不能直接修改模型本身，也可以通过修改导入模型的缩放来调整比例。在 Project 面板中选中模型，调整其 Importer 属性，注意不是变换里的缩放。如果你的游戏需要实例化具有不同缩放值的对象，也可以调整变换里的缩放值，但是物理引擎在创建这个对象的时候会额外多做一点儿工作，这可能会影响其性能，但并不会太严重。同样要注意的是，如果这个对象具有父对象，non-uniform scales 也会引起一些问题。基于以上原因，尽量在制作模型的时候就按照 Unity 的比例来建模。

(8) Hints：两个刚体的相对质量决定它们在碰撞的时候将会如何反应。给刚体设置更大的质量并不会让它下降得更快，如果要实现这个目的，可使用 Drag 参数。低的阻力值使得对象看起来更重，反之更轻。典型的 Drag 值为 0.001(固体金属)～10(羽毛)。如果想同时使用变换和物理来控制对象，那么给它一个刚体组件并将其设置为 Kinematic。如果想通过变换来移动对象，同时又想收到对象的碰撞消息，那么必须给它一个刚体组件。

(9) Mass(质量)：在物理学中，质量越大，惯性越大。这里的单位可以自己统一规定，但是官方给出的建议是场景中的物体质量最好不要相差 100 倍以上，主要防止两个质量相

差太大的物体碰撞后会产生过大的速度，从而影响游戏性能。

(10) Drag(阻力)：这里指的是空气阻力，当游戏物体受到某个作用力的时候，这个值越大越难移动。如果设置成无限的话，物体会立即停止移动。

(11) Angular Drag(角阻力)：同样指的是空气阻力，只不过是用来阻碍物体旋转的。如果设置成无限的话，物体会立即停止旋转。

(12) Use Gravity(使用重力)：勾选此项，游戏对象就会受到重力影响。

(13) Is Kinematic(是否动态)：勾选此项，表示游戏对象不受物理引擎的影响，但这不等同于没有刚体组件。这通常用于需要用动画控制的刚体，这样就不会因为惯性而受影响。

(14) Interplate(差值类型)：如果刚体移动时运动不是很平滑，可以选择以下一种平滑方式。

① None(无差值)：不使用差值平滑。

② Interpolate(差值)：根据上一帧来平滑移动。

③ Extrapolate(推算)：根据推算下一帧物体的位置来平滑移动。

(15) Collision Detection(碰撞检测方式)。

① Discrete(离散)：默认的碰撞检测方式。当物体 A 运动很快的时候，有可能前一帧还在 B 物体的前面，后一帧就在 B 物体后面了，这种情况下不会触发碰撞事件，所以如果需要检测这种情况，那就必须使用后两种检测方式。

② Continuous(连续)：这种方式可以与有静态网格碰撞器的游戏对象进行碰撞检测。

③ Continuous Dynamic(动态连续)：这种方式可以与所有设置了 2 或 3 方式的游戏对象进行碰撞检测。

(16) Freeze Position/Rotation(冻结位置/旋转)：可以对物体在 X、Y、Z 三个轴上的位置/旋转进行锁定，其不会受到相应的力而轻易改变，但可以通过脚本来修改。

(17) 最后顺便再提一下恒力组件(Constant Force)，由于比较容易理解在此就不做详细介绍了。恒力组件一共有 4 个参数，分别是 Force/Relative Force(世界/相对作用力)、Torque/Relative Torque(世界/相对扭力)。这些参数代表了附加在刚体上的 X、Y、Z 轴方向恒力的大小，另外还要注意必须是刚体才可以添加恒力。有兴趣可以自己尝试一下给物体一个 Y 轴方向的力，物体就会像火箭一样飞向天空。

5.3.3 碰撞器

碰撞体的类型包括以下 6 种。

(1) 盒子碰撞器：Box Collider。

(2) 球体碰撞器：Sphere Collider。

(3) 胶囊碰撞器：Capsule Collider。

(4) 网络碰撞器：Mesh Collider。

(5) 车轮碰撞器：Wheel Collider。

(6) 地形碰撞器：Terrain Collider。

在碰撞器之间可以添加物理材质，用于设定物理碰撞后的效果。添加完成后，它将开始

相互反弹，反弹的力度是由物理材质决定的，如图 5.72 所示。

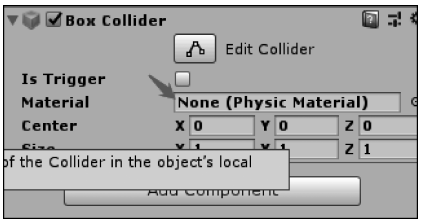


图 5.72　碰撞检测设置

碰撞检测：两个游戏对象必须有 Collider。对于双方都要检测的物体，至少其中一个必须是刚体。如果刚体是运动的，那么在双方都没有设置碰撞体的 Is Trigger 属性的时候，双方都可以通过 OnCollisionEnter 函数检测碰撞；如果至少一个碰撞体的 Is Trigger 被设置，那么双方可以通过 OnTriggerEnter 检测碰撞。

Is Trigger(触发器)：碰撞器的某一属性，用于判断是否使用触发器。

触发器事件：使用触发器时需在物体上绑定 Rigibody(刚体)组件。若无刚体，那么碰撞触发事件为 OnCollisionEnter()，若 Is Trigger 勾选之后碰撞触发事件为 OnTriggerEnter()。

了解了相关的属性和组件后，下面用几个简单的例子讲解。

1. 跳跳球

(1) 首先在 Hierarchy 面板中右击，选择 3D Object 里面的地面 Plane 和 Sphere，然后把 Sphere 调到适当的位置，如图 5.73 所示。

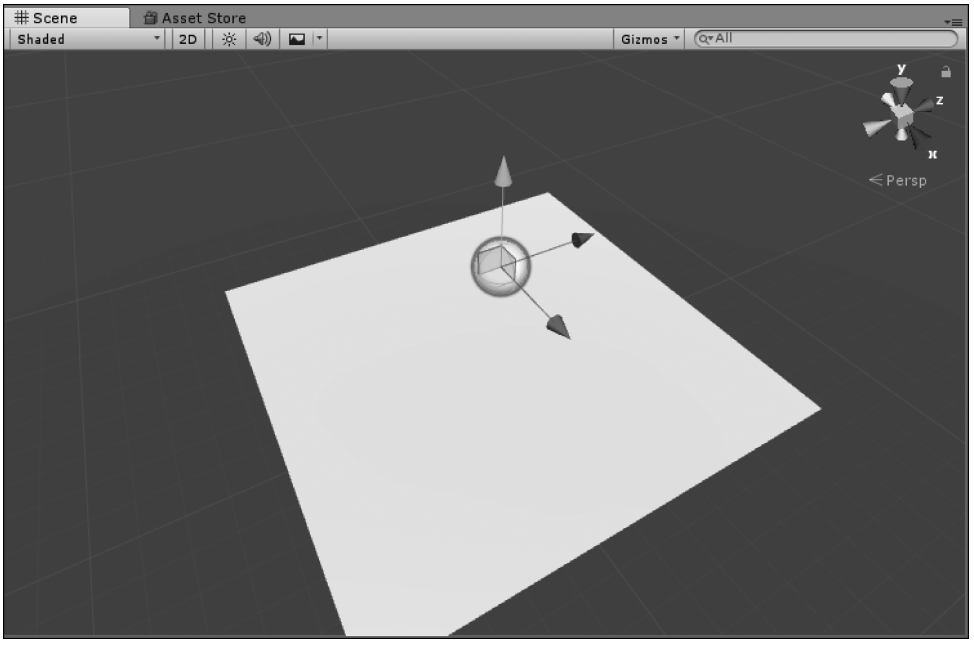


图 5.73　跳跳球的碰撞检测

(2) 然后在 Project 工程中右击，新建物理材质球 Physic Material，如图 5.74 所示。

① Dynamic Friction：动态摩擦力的值通常为 0～1。值为 0 的效果像冰，而设为 1 时，物体运动将很快停止，除非有很大的外力或重力来推动它。

② Static Friction：静态摩擦力的值同样为 0～1，用于表示物体在表面静止的摩擦力。当值为 0 时，效果像冰；当值为 1 时，物体移动十分困难。

③ Bounciness：表面的弹力(反弹系数)。0 代表不反弹，1 代表反弹将没有任何能量损失。

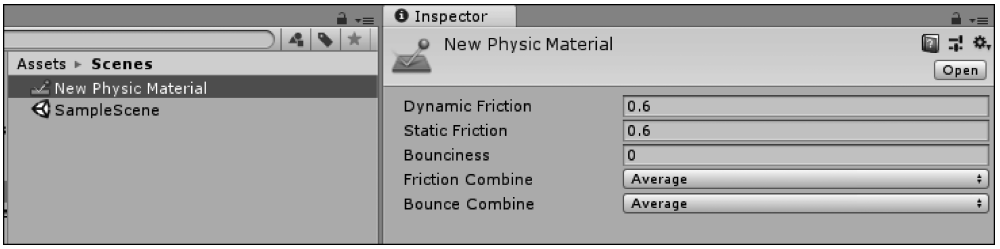


图 5.74　物理材质选择

④ Friction Combine：摩擦力结合模式。定义两个碰撞物体的摩擦力是如何结合起来并相互作用的。

反弹球的物理材质设置如图 5.75 所示。

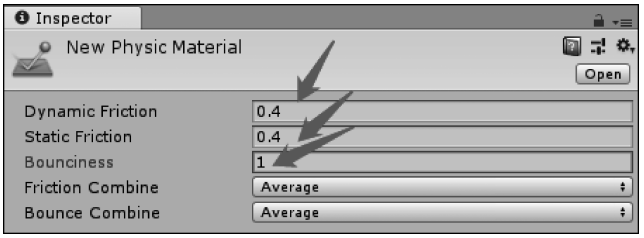


图 5.75　反弹球的物理材质设置

(3) 把物理材质球拖动给 Sphere，并且给 Sphere 添加一个刚体。添加刚体有以下两种方式。

第一种：选中要添加刚体的物体，在属性 Inspector 中找到 Add Component，单击后在搜索框里填入刚体名称，如图 5.76 所示。

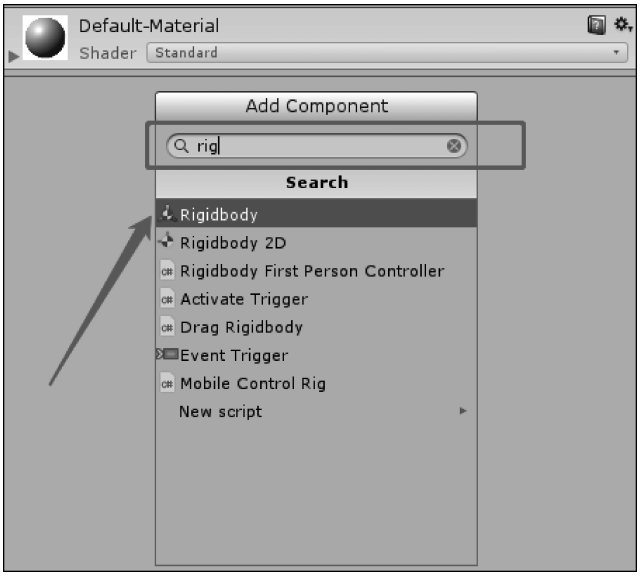


图 5.76　添加刚体方法(1)

第二种：选中要添加刚体的物体后，单击菜单栏 Component→Physics→Rigidbody 命令，如图 5.77 所示。

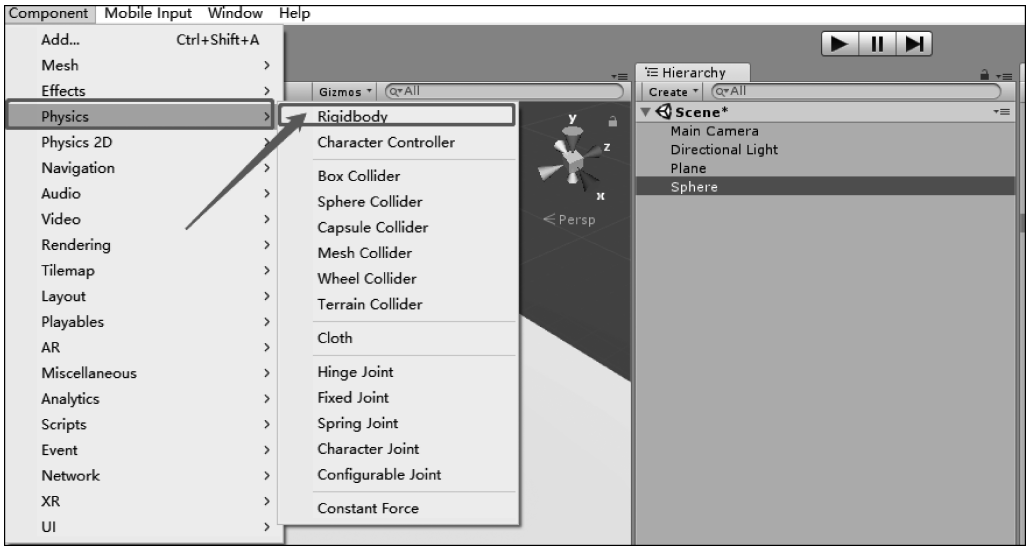


图 5.77　添加刚体方法(2)

(4) 添加完刚体后，物体就拥有了重力系统，这时单击 Play 按钮，物体就可以在地面 Plane 上跳动了。小球会不停地跳动，并且每次累加一定的量向上跳动。

2. 正方体和球体的碰撞

(1) 创建几个 Cube 和一个 Sphere，都添加上刚体 Rigidbody。其中一个作为斜坡使用，如图 5.78 所示。

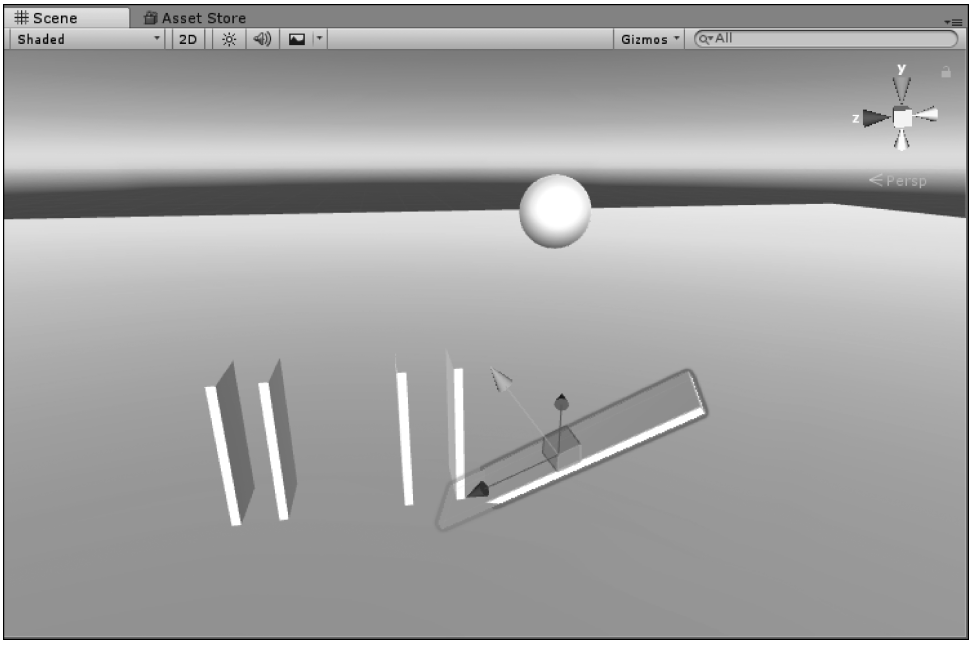


图 5.78　添加刚体方法(3)

(2) 单击 Play 按钮,运行后便会模拟现实生活中的碰撞。因为这些三维物体和地面 Plane 都带有 Box Collider、Capsule Collider 或 Mesh Collider 等属性,所以能产生碰撞效果。

5.4 粒子系统

粒子系统表示三维计算机图形学中模拟一些特定的模糊现象的技术,而这些现象用其他传统的渲染技术难以实现真实感的物理运动规律。经常使用粒子系统模拟的现象有火、爆炸、烟、水流、火花、落叶、云、雾、雪、尘、流星尾迹或者像发光轨迹这样的抽象视觉效果等。

在 Unity 的编辑器中集成了粒子系统模块。

5.4.1 主面板 Particle System

打开粒子系统主面板,如图 5.79 所示。其中各个选项的含义如下。

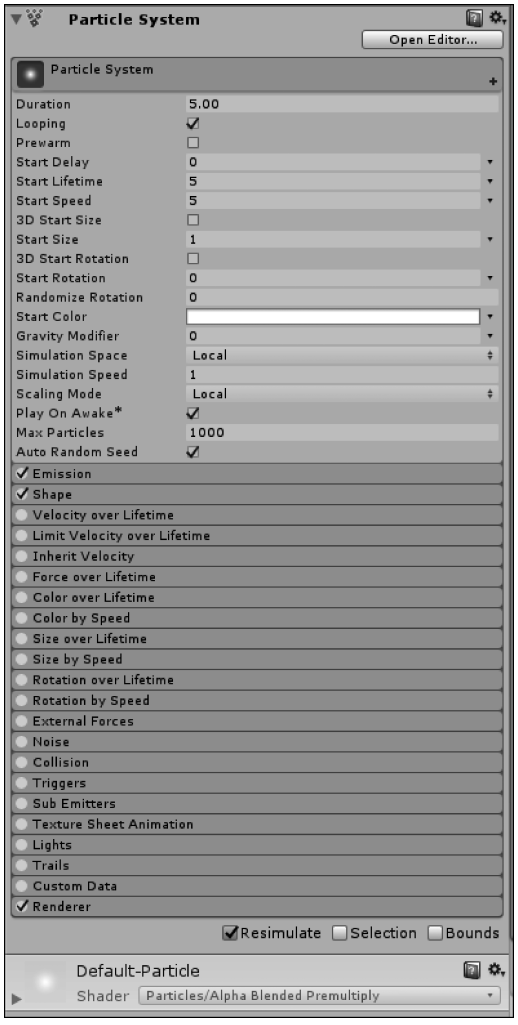


图 5.79 粒子系统主面板

Duration:粒子发射周期。如图 5.79 所示,在发射 5s 以后进入下一个粒子发射周期。如果没有勾选 Looping,5s 之后粒子会停止发射。

Looping:粒子按照周期循环发射。

Prewarm:预热系统。例如,有一个空间大小的粒子系统,若想在最开始的时候让粒子充满空间,但是粒子发射速度有限时,应该勾选 Prewarm。

Start Delay:粒子延时发射。勾选后,延长一段时间才开始发射。

Start Lifetime:粒子从发生到消失的时间长短。

Start Speed:粒子初始发生时的速度。

3D Start Size:用于粒子在某一个方向上的扩大。

Start Size:粒子初始的大小。

3D Start Rotation:用于粒子在某一个方向上的旋转。

Start Rotation:粒子初始旋转。

Randomize Rotation:随机旋转粒子方向。在三维圆形粒子的情况下,无用处。

Start Color:粒子初始颜色,可以调整加上渐变色。

Gravity Modifier:重力修正。

Simulation Space:设为 Local,此时粒

子会跟随父级物体移动；设为 World，此时粒子不会跟随父级移动；设为 Custom，粒子会跟着指定的物体移动。

Simulation Speed：根据 Update 模拟的速度。

5.4.2 Emission 模块

在上述主面板中，包含一个 Emission 模块，它的各个选项的含义如下。

Rate Over Time：单位时间内生成粒子的数量。

Rate Over Distance：随着移动距离产生的粒子数量。只有当粒子系统移动时，才发射粒子。

Time：从第几秒开始。

Min：最小粒子数量。

Max：最大粒子数量。粒子的数量会在 Min 与 Max 之间随机设置。

Cycles：在一个周期中循环的次数。

Interval：两次 Cycles 的相隔时间。

粒子发射模块的示意图，如图 5.80 所示。

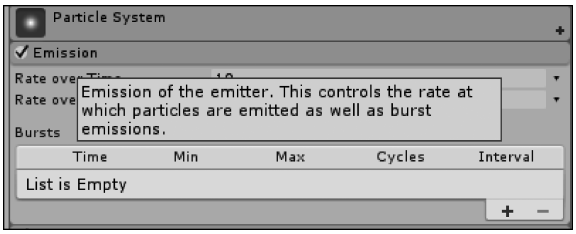


图 5.80　发射模块

如果使用 Trails 模块，必须在 Renderer 中给 Trail Material 赋值。

Ratio：分配给某个粒子拖尾的概率。

Lifetime：存在拖尾的时间间隔。

Minimum Vertex Distance：定义粒子在其 Trail 接收到新顶点之前必须行进的距离。接受新顶点以为其重新定位 Trail。

Texture Mode：以下选项设置纹理模式。

World Space：如果选用，即使应用 Local Simulation Space，Trail 顶点也不会随着粒子系统的物体移动。同时，Trail 会进入世界坐标系，且忽略任何粒子系统的移动。

Die with Particles：Trail 跟随粒子系统销毁。

Size affects Width：如果勾选的话，Trail 的宽度会乘以粒子系统的尺寸。

Size affects Lifetime：Trail 的 Lifetime 乘以粒子系统的尺寸。

Inherit Particle Color：Trail 的颜色会根据粒子的颜色调整。

Color over Trail：用于控制 Trail 在曲线上的颜色。

Width over Trail：用于控制 Trail 在曲线上的宽度。

Trails 模块如图 5.81 所示。

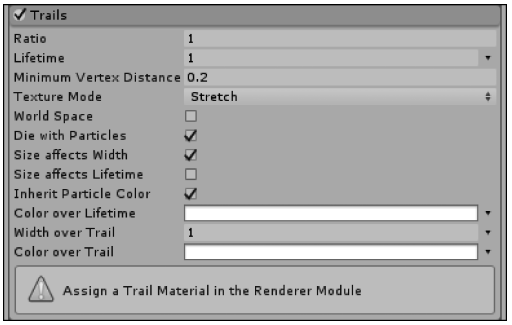


图 5.81　Trails 模块

5.4.3　粒子系统参数设置

粒子系统的参数主要是定义粒子发射器的形状，控制发射方向位置等。粒子发射器的形状(Shape)包括球体(Sphere)、半球体(Hemisphere)、圆锥(Cone)、立方体(Box)、网格(Mesh)。可沿着表面法线或随机方向施加初始力。设置界面如图 5.82 所示。

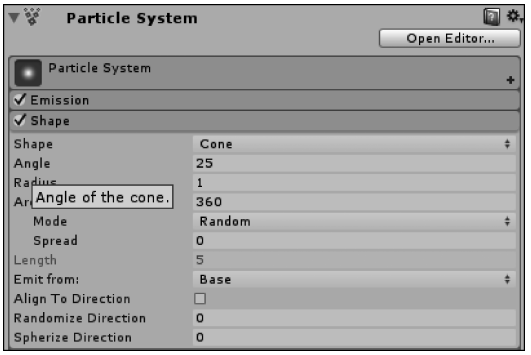


图 5.82　粒子系统参数设置

下面分别对不同形状的粒子发射器的参数设置进行介绍。

1. 球体(Sphere)

Radius：球体的半径。

Emit from Shell：从球体外壳发射。如果禁用此项，粒子将从球体内部发射。

Align To Direction：是否沿着球体表面法线方向发射。

Randomize Direction：粒子是在随机方向还是沿着球体表面法线方向发射。

2. 半球体(Hemisphere)

Radius：半球体的半径。

Emit from Shell：从半球体外壳发射。如果禁用此项，粒子将从半球体内部发射。

Randomize Direction：随机方向，粒子是在随机方向还是沿着半球体表面法线方向发射。

3. 圆锥(Cone)

Angle：锥形斜面和垂直方向的夹角。如果角度为 0 就是圆柱，粒子将在一个方向发射。

Radius：发射点的半径(锥形底面的半径)。如果值接近零，则将从一点发射。

Length：圆锥的高——地面和顶面的距离。受 Emit From 参数影响，仅从内部

(Volume)或内部外壳(Volume Shell)发出时可用。

Emit from：确定从哪里发射出。可能的值有底部(Base)、底部外壳(Base Shell)、内部(Volume)和内部外壳(Volume Shell)。

Base：粒子从圆锥体底面的任意位置沿着体积方向发射出去。

Base Shell：从地面的周长边沿着侧表面的方向发射。

Randomize Direction：随机方向。

4. 立方体(Box)

Box X：立方体 X 轴，立方体 X 轴的缩放。

Box Y：立方体 Y 轴，立方体 Y 轴的缩放。

Box Z：立方体 Z 轴，立方体 Z 轴的缩放。

Randomize Direction：随机方向，粒子是在随机方向还是沿着立方体 Z 轴方向发射。

5. 网格(Mesh)

Type：发射类型，粒子可从顶点(Vertex)、边(Edge)或面(Triangle)发射。

Mesh：网格选择，即发射形状。

Velocity over Lifetime：生命周期内的速度。直接动画化粒子的速率，演示简单的视觉行为(例如飘荡的烟雾)，分为 X、Y、Z 轴来调节，正数改变正方向的速度，负数改变负方向的速度。

Space：Local/World，速度值为本地坐标系还是世界坐标系中的值。

Limit velocity over lifetime：生命周期内的速度限制，基本上用于模拟阻力。如果超过设定的限定速度，就会抑制或固定速率。

Separate Axis：分离轴。用于设置每个轴控制。

Speed：限制的速度。

Dampen：阻尼，范围为 0～1，控制应减慢的超过速率的幅度。例如，阻尼为 1 的时候，表示生命周期结束的时候，将超过的速率减慢为 0，也就是速度降到限定的速度；当值为 0.5 的时候，则将超过的速率减慢至原来的一半。

5.4.4 粒子动画

对粒子动画的设置，如图 5.83 所示。

Grid：用网格实现。

Sprite：通过相同尺寸的 Sprite 实现粒子动画。

Tiles：网格的行列数。

Animation：以下选项设置动画效果。

Whole Sheet：动画作用于整个表格。

Single Row：动画只用于单独一行。有一个随机的选项可以选择或者是选择单独的一行来作动画。

Frame over Time：根据时间播放帧，横坐标是时间(s)，纵坐标是帧数。

Start Frame：开始帧。

Cycles：在 1s 之内循环播放的次数。

Flip U：翻转 U。

Flip V：翻转 V。

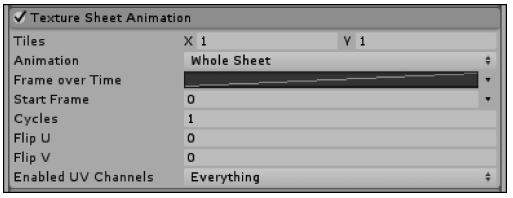


图 5.83　粒子动画

5.4.5　碰撞检测

粒子系统的碰撞行为包括两种：跟平面碰撞，跟世界里的物体碰撞。

现在以平面碰撞为例进行介绍。

在图 5.84 中单击右面的＋按钮可添加一个 Plane 碰撞体，在 Hierarchy 视图中粒子系统的子物体中出现。对平面碰撞的参数设置如下。

Visualization：选择碰撞体出现的形式。

Grid：Scene 视图中可以看到网格，Game 视图中什么也没有。

Solid：Scene 和 Game 视图中都会看到一个平面。

Scale Plane：碰撞体的大小。

Visualize Bounds：是否显示粒子的碰撞体。

Dampen：阻尼(取值 0～1，值为 1 时，粒子被吸附在碰撞体面上)。

Bounce：弹力系数。

Lifetime Loss：碰撞后粒子损失多少生命时间。

Min Kill Speed：粒子碰撞损失多少速度。

Radius Scale：碰撞偏移(值越大，粒子与碰撞体发生碰撞的点越远离碰撞体)。

Send Collision Message：是否发送碰撞事件。

5.4.6　新建粒子发射器

用于设置粒子生命过程中是否产生新的发射器。

在图 5.84 中单击右面的＋按钮便可以新建粒子发射器，在 Hierarchy 视图中粒子系统的子物体中出现，可以像一个新的粒子系统一样去编辑。单击圆圈可选择已创建好的粒子系统。

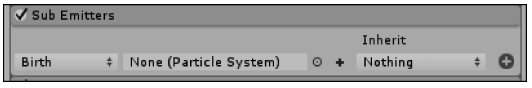


图 5.84　粒子发射器设置

在新建的粒子发射器设置界面可以做如下设置。

Color：设置颜色。

Speed Range：速度的取值范围(在这个区间里的速度分别对应上面的颜色)。

Size over Lifetime：粒子生命周期中的大小模块(基本操作与颜色一样)。

Size by Speed：粒子大小随速度的变化模块(基本操作与颜色一样)。

Rotation over Lifetime：粒子生命周期中的旋转模块(基本操作与颜色一样)。

Rotation by Speed：粒子的旋转随速度的变化模块(基本操作与颜色一样)。

Inherit Velocity：继承速度(基本不用)。

External Forces：外部作用力模块(可控制风域的倍增系数)。

Color by Speed：用于设置粒子在整个生命周期中颜色的变化，基本操作与 Start Color 一样，如图 5.85 所示。

图 5.85　设置颜色

Force over Lifetime：设置粒子在 X、Y、Z 轴的力，如图 5.86 所示。其中，Space 表示坐标系。力是有加速度的，所以粒子的速度不同于 Velocity over Lifetime 模块速度固定，而是变化的，所以可用于模拟风。

图 5.86　设置粒子在 X、Y、Z 轴的力

Limit Velocity over Lifetime 设置粒子在 X、Y、Z 轴的速度，如图 5.87 所示。

图 5.87　设置粒子在 X、Y、Z 轴的速度

(1) Separate Axes：是否限制轴的速度。

(2) Speed：粒子的发射速度。

(3) Dampen：阻尼(取值为 0～1)。

5.4.7　粒子系统实例

下面通过一个实例练习新建一个粒子系统。

首先在 Hierarchy 中右击，单击 Particle System 选项，然后在 Inspector 面板中修改相关参数，Start Size 和 Start Color 分别修改为 0.5 和绿色，如图 5.88 所示。

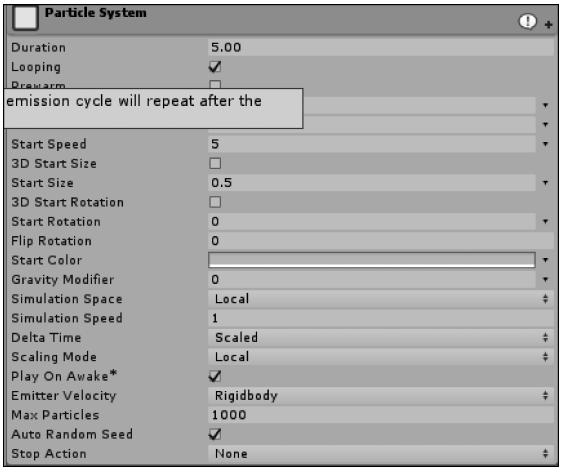


图 5.88　设置粒子渲染模式(1)

在 Shape 中选择 Hemisphere(当然也可以选择自己喜欢的形状,修改出更美的效果),如图 5.89 所示。

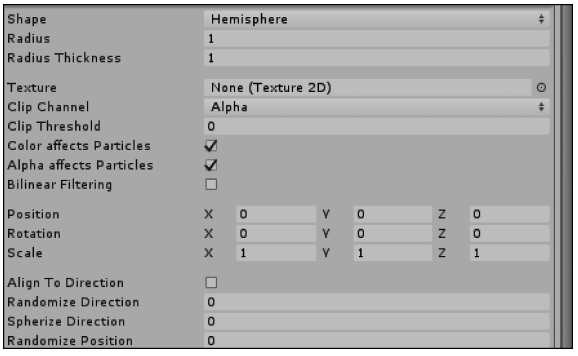


图 5.89　设置粒子渲染模式(2)

Color over Trail 选择绿色,如图 5.90 所示。

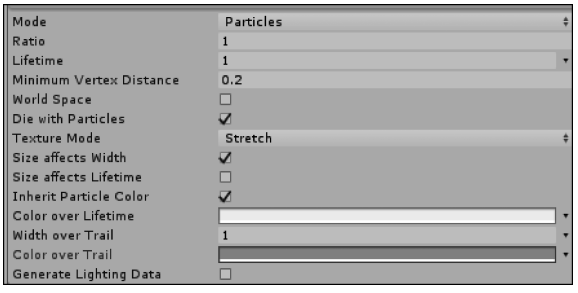


图 5.90　设置粒子渲染模式(3)

然后在 Project 工程中创建一个材质球 Material,并改变其 Shader 类型,如图 5.91 所示。

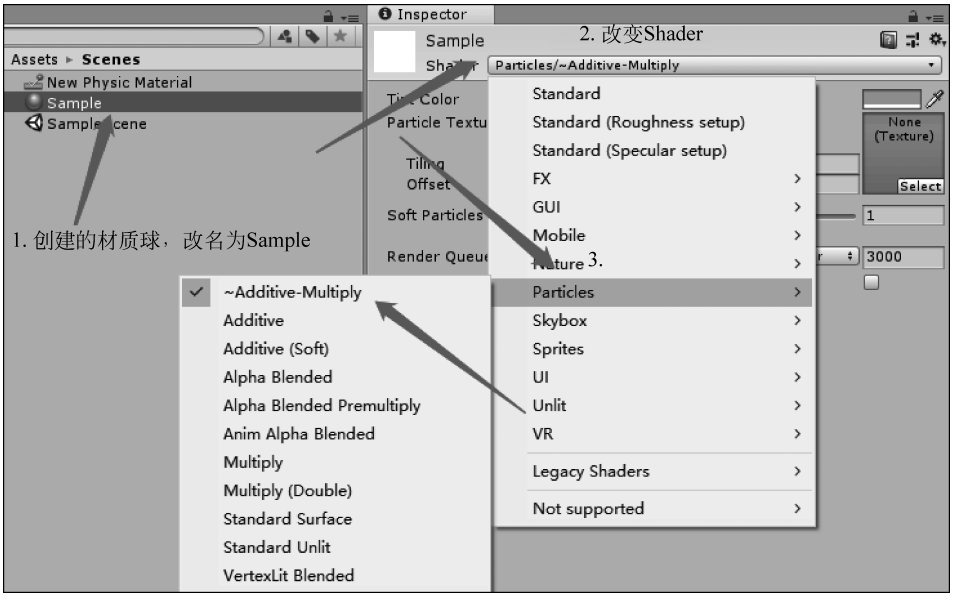


图 5.91　设置粒子渲染模式(4)

修改完后设置为相应的粒子系统。这样就可以修改粒子系统 Trail 的颜色了。效果如图 5.92 所示。

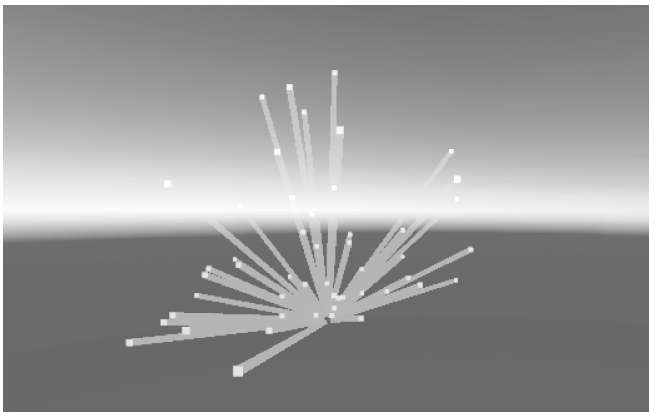


图 5.92　设置粒子渲染效果

5.5　Unity 脚本

脚本是一款游戏的灵魂。Unity 3D 脚本用于界定用户在游戏中的行为，是游戏制作中不可或缺的一部分，它能实现各个文本的数据交互并监控游戏运行状态。

5.5.1　按顺序创建脚本

在了解 Unity 脚本之前，先看看官方给的创建脚本的顺序表格，如图 5.93 所示。

按照这个脚本的创建顺序，首先创建三个 Cube，如图 5.94 所示。

再创建三个脚本，选择 Create→C# Script 命令，如图 5.95 所示。

效果如图 5.96 所示。

可以看出，不管运行多少次，执行顺序是不会改变的。接着再做一个测试，把 Exec 的 Update()方法注释，运行后效果如图 5.97 所示。

Exec 即便删除了 Update()方法，它也不会直接执行 LateUpdate()方法，而是等待 Exec1 和 Exec2 中的 Update()方法都执行完毕以后，再去执行所有的 LateUpdate()方法。

通过这两个例子，就可以很清楚地断定 Unity 后台是如何执行脚本的。每个脚本的 Awake()、Start()、Update()、LateUpdate()、FixedUpdate()等，所有的方法在后台都会被汇总到一起。

```
后台的 Awake()
{
//这里暂时按照图 5.93 中的脚本执行顺序,后面会谈到其实可以自定义该顺序
脚本 2 中的 Awake();
脚本 1 中的 Awake();
脚本 0 中的 Awake();
}
```

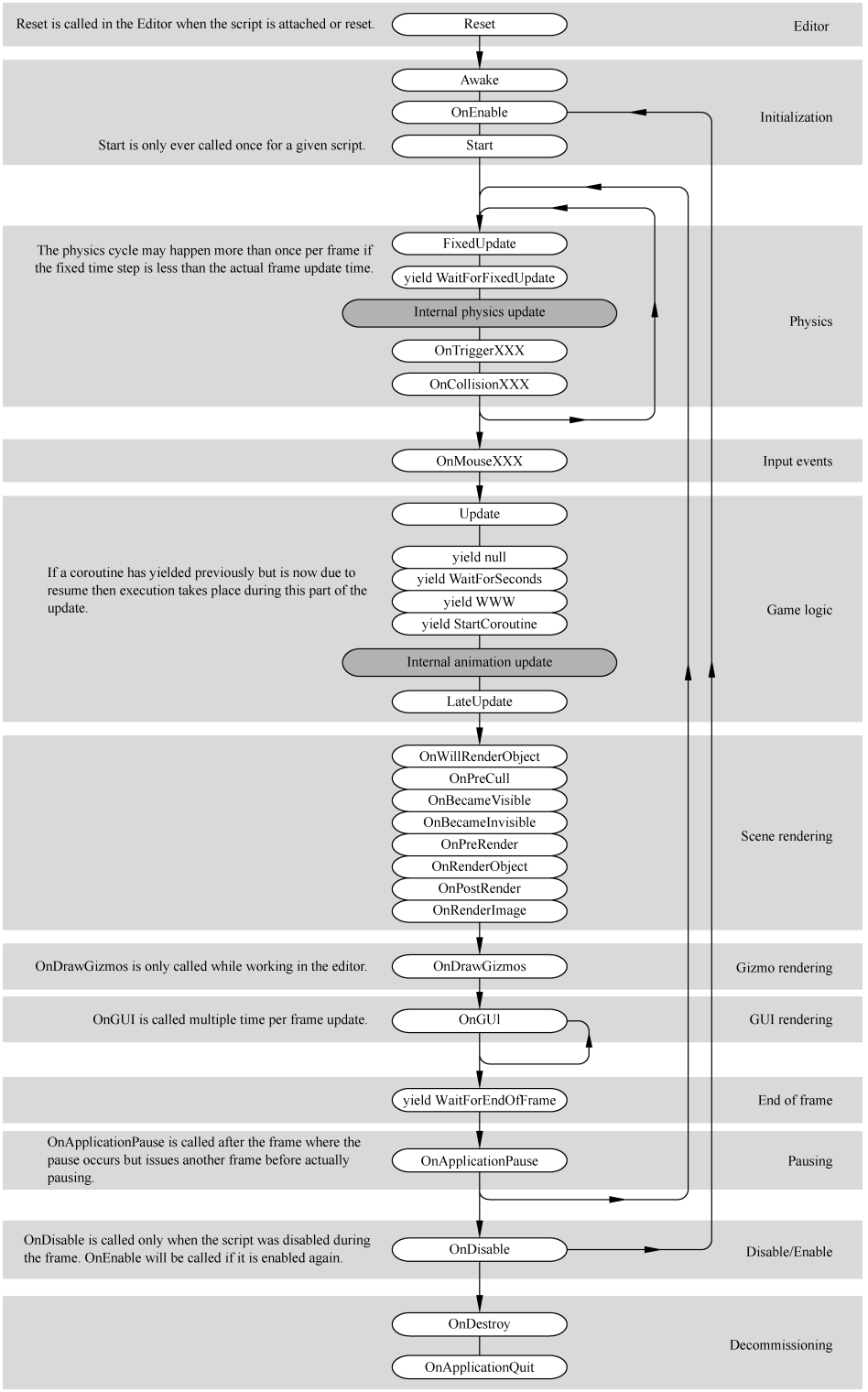

Reset is called in the Editor when the script is attached or reset.
Reset
Editor
Awake
OnEnable
Start is only ever called once for a given script.
Start
Initialization
The physics cycle may happen more than once per frame if the fixed time step is less than the actual frame update time.
FixedUpdate
yield WaitForFixedUpdate
Internal physics update
OnTriggerXXX
OnCollisionXXX
Physics
OnMouseXXX
Input events
Update
If a coroutine has yielded previously but is now due to resume then execution takes place during this part of the update.
yield null
yield WaitForSeconds
yield WWW
yield StartCoroutine
Internal animation update
LateUpdate
Game logic
OnWillRenderObject
OnPreCull
OnBecameVisible
OnBecameInvisible
OnPreRender
OnRenderObject
OnPostRender
OnRenderImage
Scene rendering
OnDrawGizmos is only called while working in the editor.
OnDrawGizmos
Gizmo rendering
OnGUI is called multiple time per frame update.
OnGUI
GUI rendering
yield WaitForEndOfFrame
End of frame
OnApplicationPause is called after the frame where the pause occurs but issues another frame before actually pausing.
OnApplicationPause
Pausing
OnDisable is called only when the script was disabled during the frame. OnEnable will be called if it is enabled again.
OnDisable
Disable/Enable
OnDestroy
OnApplicationQuit
Decommissioning

图 5.93　脚本创建顺序

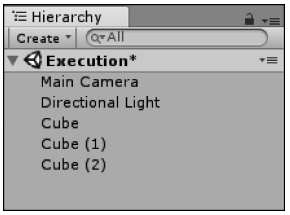


图 5.94　创建三个 Cube

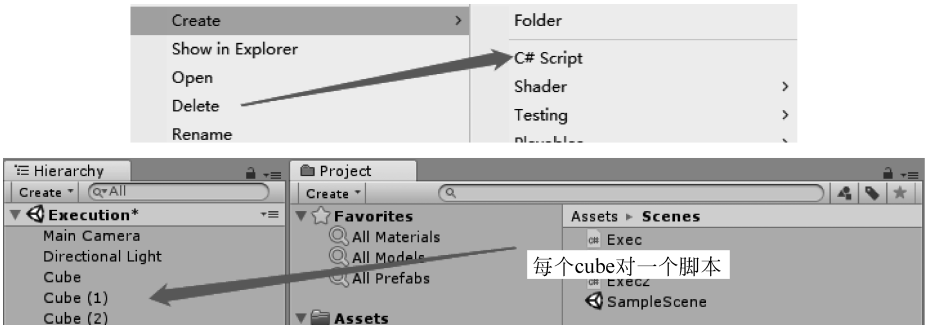


图 5.95　创建三个脚本

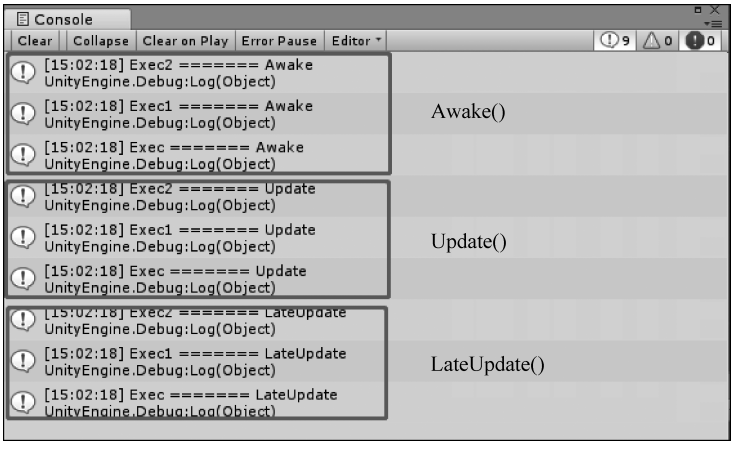


图 5.96　创建三个脚本后的效果

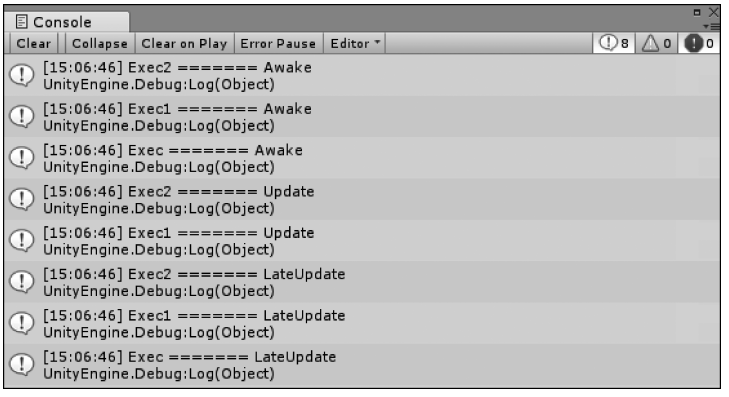


图 5.97　把 Exec 的 Update()方法注释掉的结果

后台的方法 Awake、Update、LateUpdate 等，都是按照顺序，等所有游戏对象上脚本中的 Awake 执行完毕之后，再去执行 Start、Update、LateUpdate 等方法的。

```
后台的 Update( )
{
//这里暂时按照图 5.93 中的脚本执行顺序，后面会谈到其实可以自定义该顺序
脚本 2 中的 Update();
脚本 1 中的 Update();
脚本 0 中的 Update();
}
```

5.5.2 执行顺序

现在考虑实现这样一种情况：在脚本 0 的 Awake()方法中创建一个立方体对象，然后在脚本 2 的 Awake()方法中去获取这个立方体对象。代码如下。

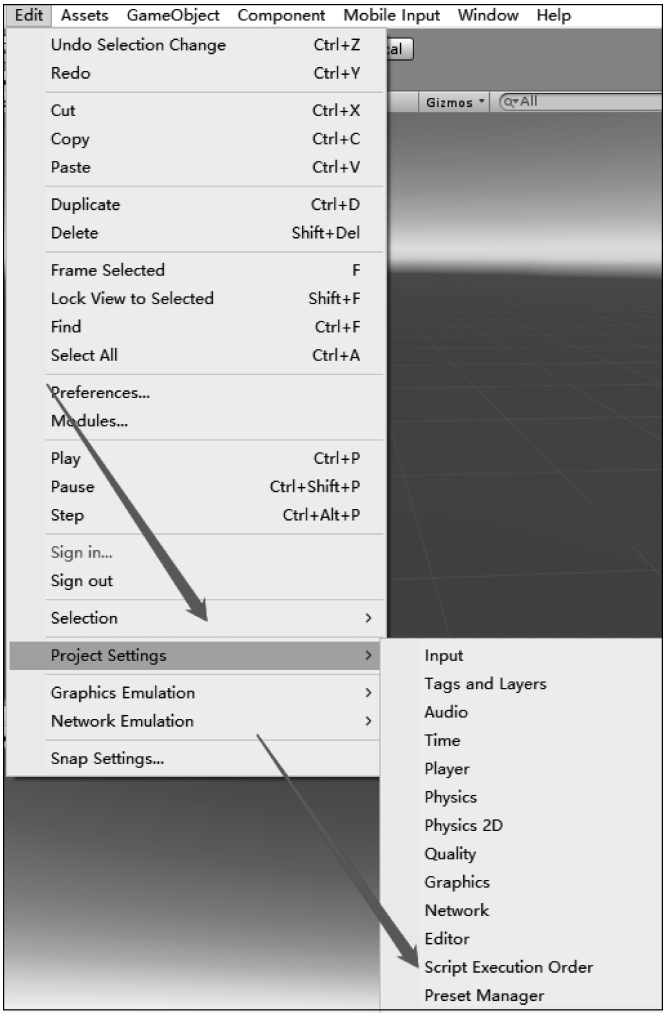
```
using UnityEngine;
using System.Collections;
public class Exec : MonoBehaviour
{
    void Awake()
    {
        GameObject.CreatePrimitive(PrimitiveType.Cube);
    }
}
//Exec2.cs
using UnityEngine;
using System.Collections;
public class Exec2 : MonoBehaviour
{
    void Awake()
    {
        GameObject go = GameObject.Find("Cube");
        Debug.Log(go.name);
    }
}
```

如果脚本的执行顺序是先执行 Exec，然后再执行 Exec2，那么 Exec2 中的 Awake 就可以正确地获取到该立方体对象；但如果脚本的执行顺序是先执行 Exec2，然后是 Exec，那么 Exec2 肯定会报空指针错误。

在实际项目中的脚本会非常多，它们的先后执行顺序往往并不明确。有人认为是按照栈结构来执行的，即后绑定到游戏对象上的脚本先执行。但一般地，建议在 Awake()方法中创建游戏对象或 Resources.Load(Prefab)对象，然后在 Start()方法中获取游戏对象或者组件。事件函数的执行顺序是固定的，这样就可以确保万无一失了。

另外，Unity 也提供了一个方法设置脚本的执行顺序，在 Edit→Project Settings→Script Execution Order 菜单项中，如图 5.98 所示。

单击右下角的＋按钮将弹出下拉框，包括游戏中的所有脚本。脚本添加完毕后，可以用



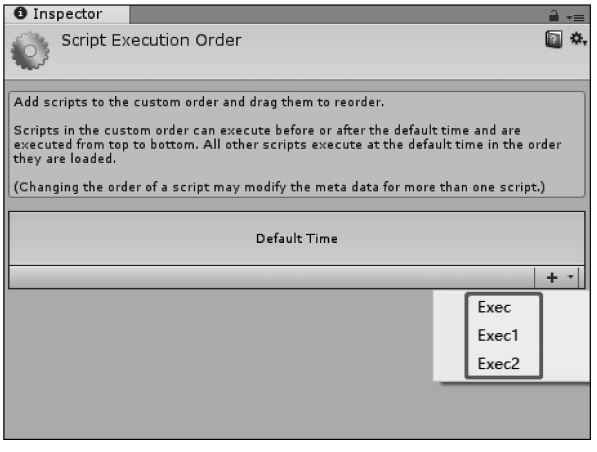


图 5.98　Inspector 面板

鼠标拖动脚本来为脚本排序，脚本名后面的数字越小，脚本越靠上，越先执行。其中，Default Time 表示没有设置脚本执行顺序的那些脚本的执行顺序，如图 5.99 所示。

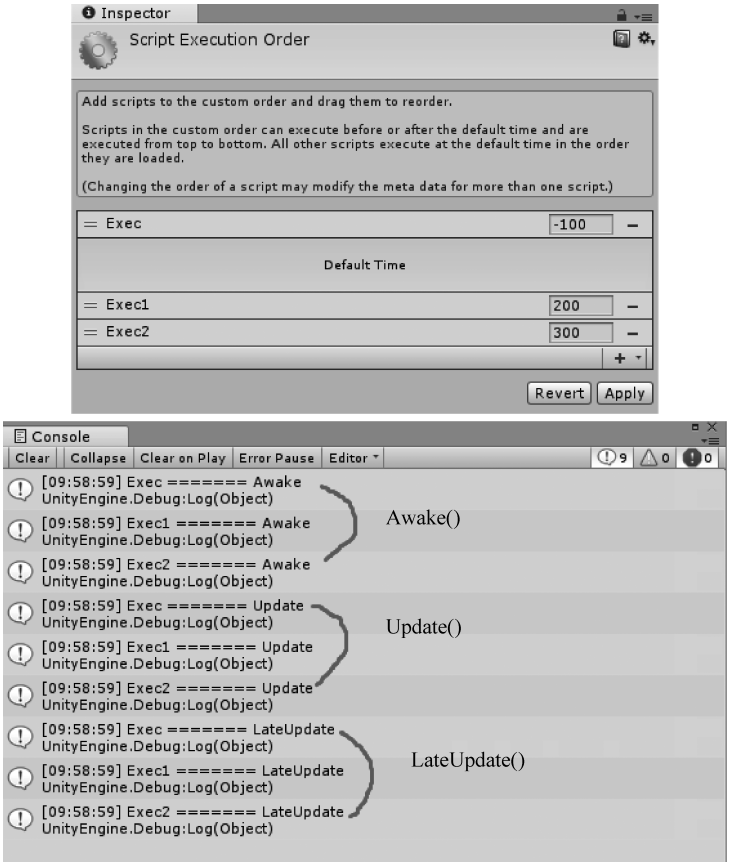


图 5.99　为脚本排序

5.5.3　脚本的编译顺序

由于脚本的编译顺序会涉及特殊文件夹，比如上面提到的 Plugins、Editor 还有 Standard Assets 等标准的资源文件夹，所以脚本的放置位置就非常重要了。下面用一个例子来说明不同文件夹中的脚本的编译顺序，如图 5.100 所示。

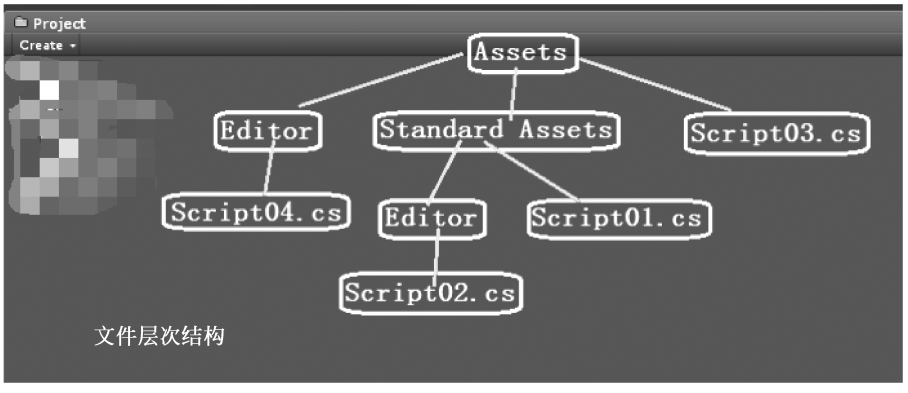


图 5.100　脚本的编译顺序

在项目中建立了如图 5.100 所示的文件夹层次结构，并在编译项目之后，会在项目文件夹中生成一些包含 Editor、firstpass 这些字样的项目文件。产生的项目文件结构如图 5.101 所示。

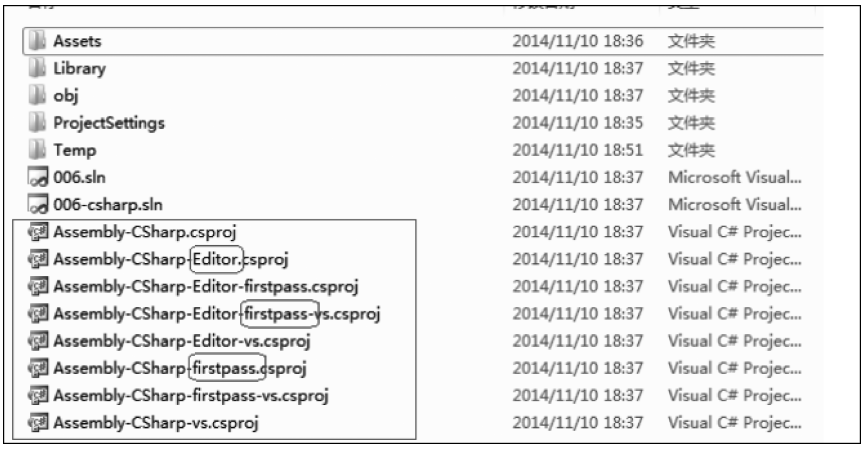


图 5.101　产生的项目文件

下面来详细探讨一下这些字样的具体意思，以及它们与脚本的编译顺序的联系。

(1) 首先从脚本语言类型来看，Unity3D 支持 3 种脚本语言，都会被编译成 CLI 的 DLL。

如果项目中包含 C# 脚本，那么 Unity3D 会产生以 Assembly-CSharp 为前缀的工程，名字中包含 vs 的是给 Virtual Studio 使用的，不包含 vs 的是给 MonoDevelop 使用的。

项目中的脚本语言的工程前缀和工程后缀如下。

① C# Assembly-CSharp csproj。

② UnityScript Assembly-UnityScript unityproj。

③ Boo Assembly-Boo booproj。

如果项目中这 3 种脚本都存在，那么 Unity 将会生成 3 种前缀类型的工程。

(2) 对于每一种脚本语言，根据脚本放置的位置(部分根据脚本的作用，比如编辑器扩展脚本，就必须放在 Editor 文件夹下)，Unity 会生成 4 种后缀的工程。其中，firstpass 表示先编译，Editor 表示放在 Editor 文件夹下的脚本。

在上面的示例中，得到了两套项目工程文件，分别被 Virtual Studio 和 MonoDevelop 使用(后缀包不包含 vs)。为简单起见，我们只分析 vs 项目，得到的文件列表如下。

Assembly-CSharp-firstpass-vs. csproj

Assembly-CSharp-Editor-firstpass-vs. csproj

Assembly-CSharp-vs. csproj

Assembly-CSharp-Editor-vs. csproj

它们的编译顺序如下。

(1) 所有在 Standard Assets、Pro Standard Assets 或者 Plugins 文件夹中的脚本会产生一个 Assembly-CSharp-firstpass-vs. csproj 文件，并且先编译。

(2) 所有在 Standard Assets/Editor、Pro Standard Assets/Editor 或者 Plugins/Editor

文件夹中的脚本产生 Assembly-CSharp-Editor-firstpass-vs. csproj 工程文件，接着编译。

(3) 所有在 Assets/Editor 外面的，并且不在(1)和(2)中的脚本文件(一般这些脚本就是我们自己写的非编辑器扩展脚本)会产生 Assembly-CSharp-vs. csproj 工程文件，被编译。

(4) 所有在 Assets/Editor 中的脚本产生一个 Assembly-CSharp-Editor-vs. csproj 工程文件，被编译。

5.6 用户界面

本节主要介绍 Unity 中的图形用户界面(GUI)编程。Unity 有一个非常强大的 GUI 脚本 API，它允许用户使用脚本快速创建简单的菜单和 GUI。

5.6.1 简述

Unity 提供了使用脚本创建 GUI 界面的能力。Unity 并没有提供一套原生的可视化 GUI 开发工具，但是可以在 Unity Asset 商店找到一些使用某种形式的图形化脚本编写 GUI 的工具。Autodesk Scaleform 也提供了一个可以单独购买并整合进 Unity 的插件，如果读者对 Scaleform 插件的 Unity 版本感兴趣，可以用 Scaleform Unity Plugin。

Unity 提供了两个主要的类来创建 GUI。其中，GUI 类用于 GUI 控件的固定布局，GUILayout 类用于创建手动放置的 GUI 控件。这两个类之间的区别将在后面详细讲述。

Unity 也提供了 GUISkin 资源。它可以被应用于给定的 GUI 控件，且提供一种通用的外观和感觉。一个 GUISkin 只是 GUIStyle 对象的集合。每个 GUIStyle 对象定义了单个 GUI 控件的样式，比如按钮、标签或者文本域。

GUIText 组件可被用于渲染单个的文本元素，GUITexture 组件可以被用于渲染二维材质到屏幕。GUIText 和 GUITexture 都适用于为游戏绘制 GUI 元素(就像 HUD)，但这些组件不适用于在游戏中绘制菜单。对于游戏中的菜单(如等级选择和选项设置页面)应该使用 GUI 和 GUILayout 类。

这些不同的类、资源和组件在每一个本文中都会阐述。

5.6.2 创建菜单

首先讲述一下如何使用 GUI 和 GUILayout 在 Unity 中创建菜单，并展示如何使用 GUISkin 和 GUIStyle 自定义 GUI 控件的外观。

1. OnGUI

GUI 的渲染是通过创建脚本并定义 OnGUI()函数执行的。所有的 GUI 渲染都应该在该函数中执行或者在一个被 OnGUI()调用的函数中执行。

例如在下面的代码中，给出了一个包含 OnGUI()函数的类。

```
Demo.cs
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Demo : MonoBehaviour {
    //初始化
```

```
    void Start () {
    }
    //更新
    void Update () {
    }
    void OnGUI()
    {
        float buttonWidth = 100;
        float buttonHeight = 50;
        float buttonX = (Screen.width - buttonWidth) / 2.0f;
        float buttonY = (Screen.height - buttonHeight) / 2.0f;
        //在屏幕中间绘制一个 button 组件
        if (GUI.Button(new Rect(buttonX, buttonY, buttonWidth, buttonHeight), "Press Me!"))
        {
            //在调试控制台打印一些文字
            Debug.Log("Thanks!");
        }
    }
}
```

程序代码和运行效果分别如图 5.102 和图 5.103 所示。

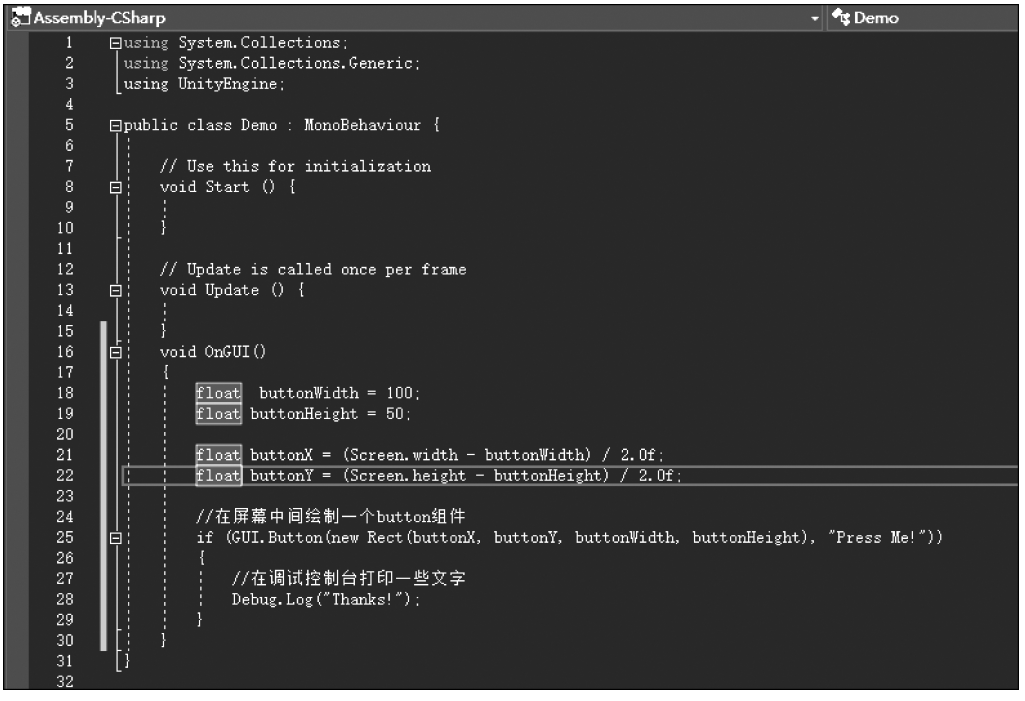


图 5.102　创建界面的代码

2. GUIStyle

大多数通用控件，比如按钮和标签，允许在控件上呈现指定的文本或者材质。如果想在一个控件上指定文本与材质，那么必须使用 GUIContent 结构。这个类与 GUIStyle 具有紧密合作关系。GUIContent 定义渲染什么，GUIStyle 定义怎样渲染。

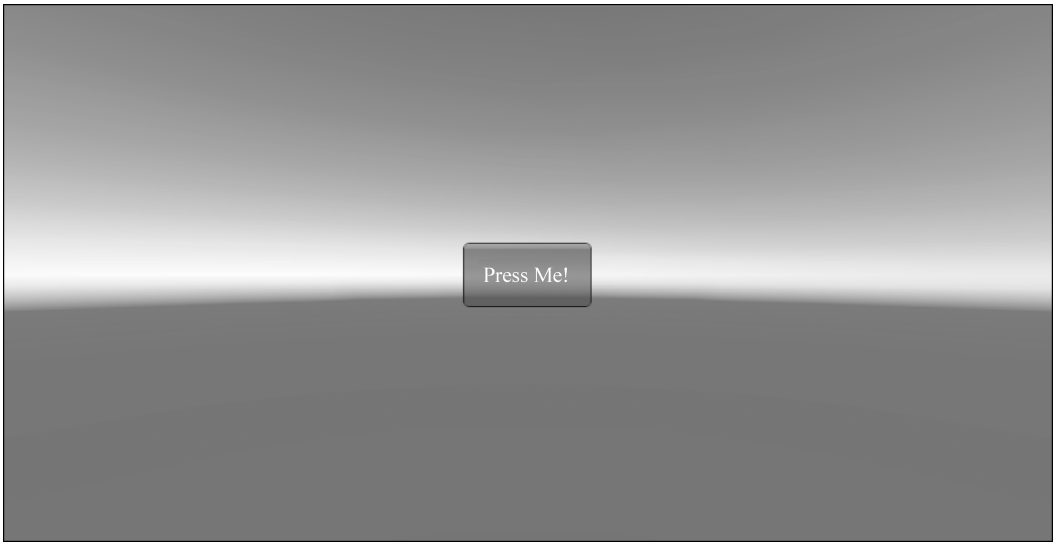


图 5.103　运行效果

例如,要为一个 Button 项添加图片文字信息,可以采用如下代码。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class DemoTwo : MonoBehaviour {
    public GUISkin Mygui;
    string text = "hello text";
    // 初始化
    void Start(){
    }
    // 更新
    void Update() {
    }
    void OnGUI()
    {
        GUI.skin = Mygui;
        float buttonWidth = 180;
        float buttonHeight = 50;
        float buttonX = (Screen.width - buttonWidth) / 2.0f;
        float buttonY = (Screen.height - buttonHeight) / 2.0f;
        GUI.Button(new Rect(buttonX, buttonY, buttonWidth, buttonHeight), text,"Photo");
    }
}
```

3. GUISkin

在 Project 工程面板中右击,单击 Create 命令,创建 GUISkin,如图 5.104 所示。

运行效果如图 5.105 所示,由于采用了 GUISkin,Button 显示出了被改变的形状和文字信息。

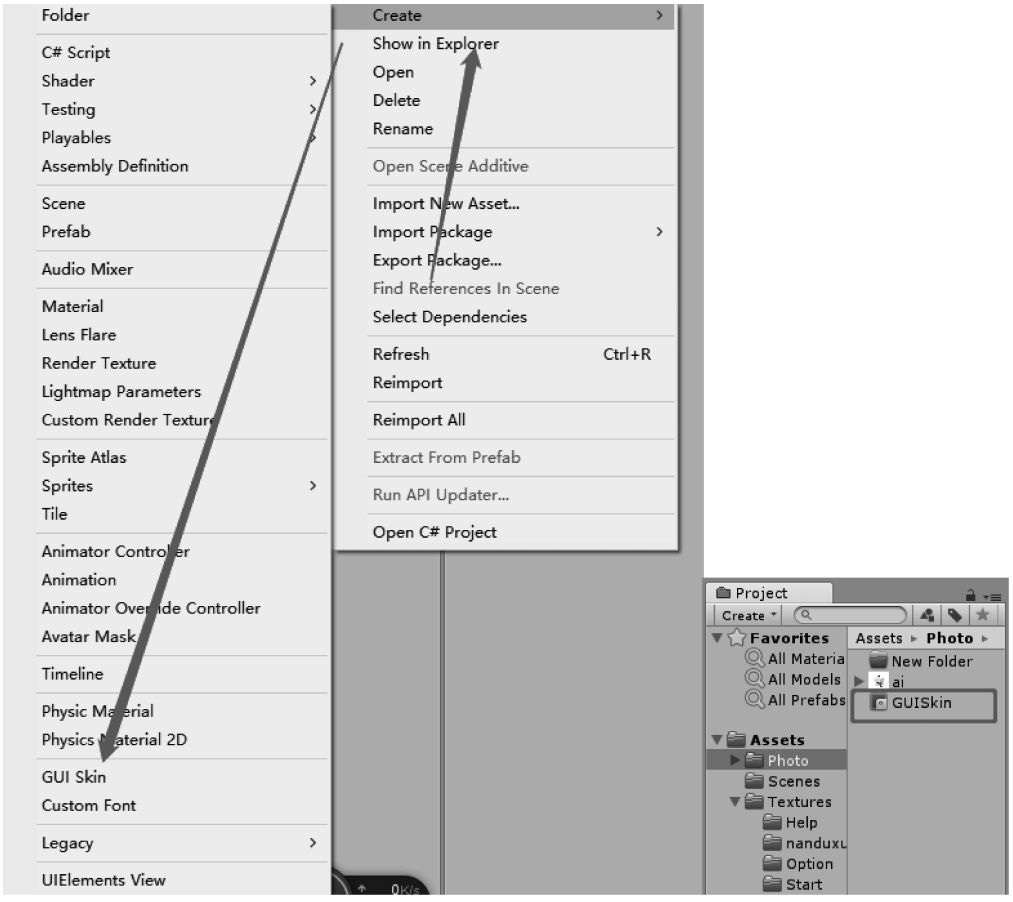


图 5.104　创建 GUISkin

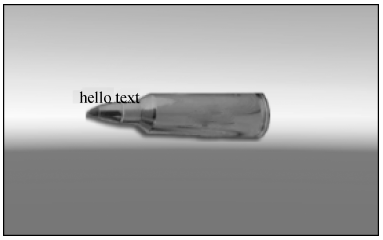


图 5.105　运行效果

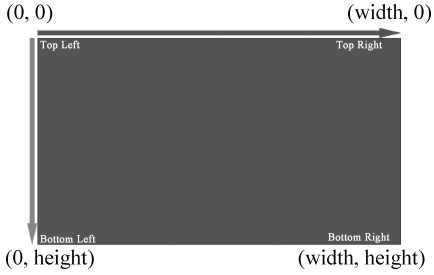


图 5.106　Screen.Width 和 Screen.Height 属性可以被用于检视当前屏幕的范围

5.6.3　放置控件

使用 GUI 类时，必须手动地摆放屏幕上的控件。控件使用 GUI 静态函数的 position 参数来摆放。为了在屏幕上摆放控件，必须将一个 Rect 结构作为第一个参数传递给 GUI 控件函数。Rect 结构为控件定义了 X、Y、Width、Height 属性，单位都是像素，如图 5.106 所示。

1. GUI 类

GUI 类是 Unity 用于将控件渲染到屏幕上的主要类。GUI 类使用手动摆放来决定屏幕上控件的位置,这意味着在渲染控件时必须显式地指定控件在屏幕上的位置。使用这种手动摆放控件的方法需要多做一些工作,但它可以精确地控制屏幕上的控件位置。如果不想手动指定 GUI 控件的位置,可使用 GUILayout 类。后面将详细阐述 GUILayout。

2. GUI 控件

在下面的章节中,将介绍在使用 GUI 和 GUILayout 时可利用的不同控件,这些类提供的默认控件是 Box, Button, Label, Window, Texture, ScrollBars, Sliders, TextField, TextArea, Toggle 和 Toolbar。

1) GUI. Button

最常用的控件之一为按钮,可以使用 GUI. Button()静态函数来创建一个按钮。此函数用于将按钮渲染到屏幕上,当松开按钮时函数返回 true。

值得一提的是 GUI. Button()函数,其只有当鼠标在按钮上按下并且在按钮上松开时才返回 true。如果用户按下按钮移动鼠标在按钮外面释放鼠标,则函数不会返回 true。同样地,如果用户按下了鼠标之后将光标移动到按钮上,然后释放鼠标该函数也不会返回 true。要使该函数返回 true,必须在按钮上按下并释放鼠标。

以下代码可用于使用按钮创建一个简单的等级选择屏幕(假定在 Build Settings 对话框中有多个场景文件要设置)。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Demo : MonoBehaviour {
    // 初始化
    void Start () {
    }
    // 更新
    void Update ()
    {
    }
    void OnGUI()
    {
        float groundWidth = 120;
        float groundHeight = 150;
        float screenWidth = Screen.width;
        float screenHeight = Screen.height;
        float groupx = (screenWidth - groundWidth) / 2;
        float groupy = (screenHeight - groundHeight) / 2;
        GUI.BeginGroup(new Rect(groupX, groupY, groundWidth, groundHeight));
        GUI.Box(new Rect(0, 0, groundWidth, groundHeight), "Option Select");
        if (GUI.Button(new Rect(10, 30, 100, 30), "Level 1"))
        {
            Application.LoadLevel(1);
        }
        if (GUI.Button(new Rect(10, 70, 100, 30), "Level 2"))
```

```
        {
            Application.LoadLevel(2);
        }
        if (GUI.Button(new Rect(10, 110, 100, 30), "Level 3"))
        {
            Application.LoadLevel(3);
        }
        GUI.EndGroup();
    }
}
```

运行效果如图 5.107 所示。

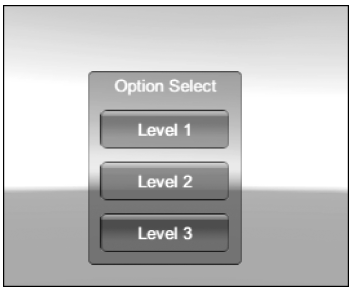


图 5.107　运行效果

2) GUI.Label()

GUI.Label()静态函数用于绘制一个标签。标签通常是在屏幕上指定位置绘制的文字。标签控件最常用的是在菜单屏幕中指定选项名称(比如文本框和文本域)。标签可包含文字、材质或者两者兼有(使用之前讲过的 GUIContent 结构)。

下面的例子在屏幕上显示绘制了两个选项。选项名称和滑块的值使用标签呈现。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Demo : MonoBehaviour {
    // 初始化
    void Start () {
    }

    // 更新
    void Update ()
    {
    }
    private float masterVolume = 1.0f;
    private float sfxVolume = 1.0f;
    void OnGUI()
    {
        float groupWidth = 380;
        float groupHeight = 110;
        float screenWidth = Screen.width;
        float screenHeight = Screen.height;
        float groupX = (screenWidth - groupWidth) / 2;
        float groupY = (screenHeight - groupHeight) / 2;
        GUI.BeginGroup(new Rect(groupX, groupY, groupWidth, groupHeight));
        GUI.Box(new Rect(0, 0, groupWidth, groupHeight), "Audio Settings");
        GUI.Label(new Rect(10, 30, 100, 30), "Master Volume");
        masterVolume = GUI.HorizontalSlider(new Rect(120, 35, 200, 30), masterVolume, 0.0f, 1.0f);
        GUI.Label(new Rect(330, 30, 50, 30), "(" + masterVolume.ToString("f2") + ")");
        GUI.Label(new Rect(10, 70, 100, 30), "Effect Volume");
```

```
        sfxVolume = GUI.HorizontalSlider(new Rect(120, 75, 200, 30), sfxVolume, 0.0f, 1.0f);
        GUI.Label(new Rect(330, 70, 50, 30), "(" + sfxVolume.ToString("f2") + ")");
        GUI.EndGroup();
    }
}
```

运行效果如图 5.108 所示。

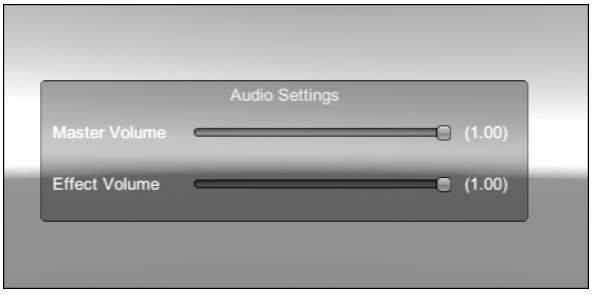


图 5.108　运行效果

3) GUI.HorizontalSlider()和 GUI.VerticalSlider()

GUI.HorizontalSlider()和 GUI.VerticalSlider()这两个静态函数可分别用于绘制水平和竖直滑块。滑块用于指定在一定范围内的一个数值。在上面的例子中,使用了两个水平滑块来指定主音量和音效,范围为 0～1。

Slider 函数接受当前滑块值、滑块最小值和滑块最大值。上面的例子展示了如何使用水平滑块,而竖直滑块使用的是同样的参数,只是滑块是竖直绘制而不是水平绘制。

下面的例子展示了使用竖直滑块来创建一个音频均衡器。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Demo : MonoBehaviour {
    // 初始化
    void Start () {
    }
    // 更新
    void Update ()
           {
    }
private float[] equalizerValues = new float[10];
void OnGUI()
    {
        float groupWidth = 320;
        float groupHeight = 260;
        float screenWidth = Screen.width;
        float screenHeight = Screen.height;
        float groupX = (screenWidth - groupWidth) / 2;
        float groupY = (screenHeight - groupHeight) / 2;
        GUI.BeginGroup(new Rect(groupX, groupY, groupWidth, groupHeight));
        GUI.Box(new Rect(0, 0, groupWidth, groupHeight), "Equalizer");
```

```
        for (var i = 0; i < equalizerValues.Length; i++)
        {
                equalizerValues[i] = GUI.VerticalSlider(new Rect(i * 30 + 20, 30, 20, 200),
                equalizerValues[i], 0.0f, 1.0f);
        }
        GUI.EndGroup();
    }
}
```

运行效果如图 5.109 所示。

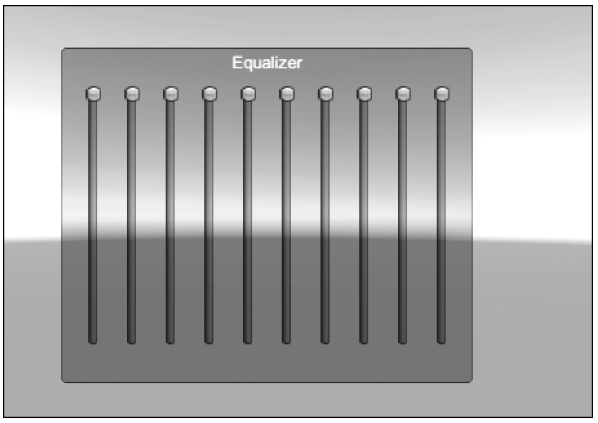


图 5.109　运行效果

当使用水平滑块时，最小值在滑块的左边，最大值在滑块的右边。当使用竖直滑块时，最小值在顶部，最大值在滑块底部。

4) GUI.Window()和 GUI.DragWindow()

GUI 类提供了在屏幕上绘制窗口的函数，窗口可以使用外部函数(除了 OnGUI)来渲染窗口的内容。如果在窗口的回调函数中使用 GUI.DragWindow()函数，那窗口将会是可拖动的。

下面的代码创建了一个简单而可拖动的窗口。

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Demo : MonoBehaviour {
    // 初始化
    void Start () {
            }
        // 更新
    void Update ()
           {
    }
    //窗口的初始位置以及大小
private Rect windowRect0 = new Rect(20, 20, 150, 0);
void OnGUI()
```

```
    {
        //渲染窗口 ID 为 0
        windowRect0 = GUILayout.Window(0, windowRect0, WindowFunction, "Draggable Window");
    }
public void WindowFunction(int id)
    {
        GUILayout.Label("This is a draggable window!");
        //窗口的拖条(drag - strip),坐标相对于窗口的左上角
GUI.DragWindow(new Rect(0, 0, 150, 20));
    }
}
```

运行效果如图 5.110 所示。

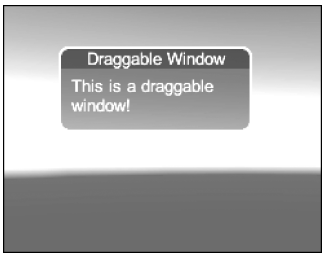


图 5.110　运行效果

如果在新场景中将脚本应用到 GameObject 上,就会看到窗口,单击并拖动窗口标题,就可以实现窗口在屏幕上的拖动。

在窗口中可以放置任意数量的控件,如果想要 Unity 在窗口中自动布局控件(类似例子中一样),应该使用 GUILayout.Window()函数而不是 GUI.Window()函数。当使用 GUILayout.Window()函数时,Unity 会自动修改窗口的高度以适应内容。

5.6.4　自动布局

前面展示的例子都是用 GUI 类来创建菜单的。GUI 类需要手动地在屏幕上放置控件。在某些情况下,手动摆放控件很有用,但如果想要 Unity 自动布局控件,则需要使用 GUILayout 类。这个类提供了许多像 GUI 一样的功能,但无须指定控件的大小。

默认情况下,当使用 GUILayout 函数时所有视图中的组件都会竖直排列。可以使用 GUILayout.BeginHorizontal 和 GUILayout.EndHorizontal 静态函数使控件相邻排放。每出现一次 GUILayout.BeginVertical 必须有相应的 GUILayout.EndVertical 与其对应;每出现一次 GUILayout.BeginHorizontal 则必须有相应的 GUILayoutHorizontal 与其对应。

下面的例子展示了如何使用竖直和水平布局来创建复杂的表格。

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Demo : MonoBehaviour {
    // 初始化
    void Start () {
    }
    // 更新
    void Update ()
    {
    }
    private string firstName = "First Name";
    private string lastName = "Last Name";
```

```
private uint age = 0;
private bool submitted = false;
private Rect windowRect0;
void OnGUI()
{
    var screenWidth = Screen.width;
    var screenHeight = Screen.height;
    var windowWidth = 300;
    var windowHeight = 180;
    var windowX = (screenWidth - windowWidth) / 2;
    var windowY = (screenHeight - windowHeight) / 2;
    //将窗口放置到屏幕中间
    windowRect0 = new Rect(windowX, windowY, windowWidth, windowHeight);
    GUILayout.Window(0, windowRect0, UserForm, "User information");
}
void UserForm(int id)
{
    GUILayout.BeginVertical();
    //姓
    GUILayout.BeginHorizontal();
    GUILayout.Label("First Name", GUILayout.Width(80));
    firstName = GUILayout.TextField(firstName);
    GUILayout.EndHorizontal();
    //名
    GUILayout.BeginHorizontal();
    GUILayout.Label("Last Name", GUILayout.Width(80));
    lastName = GUILayout.TextField(lastName);
    GUILayout.EndHorizontal();
    //年龄
    GUILayout.BeginHorizontal();
    GUILayout.Label("Age", GUILayout.Width(80));
    var ageText = GUILayout.TextField(age.ToString());
    uint newAge = age;
    if (uint.TryParse(ageText,out newAge))
    {
        age = newAge;
    }
    GUILayout.EndHorizontal();
    if (GUILayout.Button("Submit"))
    {
        submitted = true;
    }
    if (GUILayout.Button("Reset"))
    {
        firstName = "First Name";
        lastName = "Last Name";
        age = 0;
        submitted = false;
    }
    if (submitted)
    {
```

```
            GUILayout.Label("submitted!");
        }
        GUILayout.EndVertical();
    }
}
```


运行效果如图 5.111 所示。

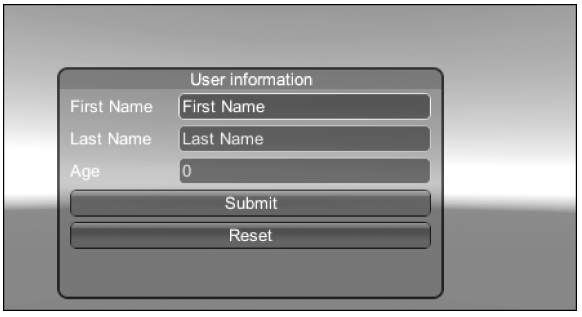


图 5.111　运行效果

所有此类函数都可以被用于创建组(或者可以自动布局的区域),这些组或者区域可以保证将在特定区域内的控件聚合起来。

5.6.5　样式和皮肤

Unity 为所有的 GUI 控件提供了默认的外观。对于一个快速解决方案,默认的样式足够使用了,但大部分人都不希望在将要面市的游戏中使用 Unity 的默认 GUI 样式。这时候就需要使用自定义的 GUISkin 更改按钮、标签、滑块和滚动条的外观。

GUISkin 是一个套件,可以从主菜单中选择 Assets→Create→GUISkin 命令创建它,如图 5.112 所示。

如果在项目视图里选择了 GUISkin,可以为你创建的多种控件编辑单独的样式设置。

若想将默认皮肤替换掉而使用自定义的皮肤,需在 GUI 脚本里面设置 GUI。Skin 属性为自定义的皮肤。将 GUI.Skin 属性设置为 null 将还原回默认的 GUISkin。

GUIStyle 是一系列 GUISkin 样式的集合。GUIStyle 定义了一个控件所有可变状态的样式。一个控件有以下几种状态。

(1) 正常状态:控件的默认状态。鼠标既没有悬停到控件上,控件也没有获得系统焦点。

(2) 悬停状态:鼠标当前悬停在控件上。

(3) 注视状态:当前正选择控件,选中的控件将会接受键盘输入。

(4) 活动状态:控件被单击,此状态对于按钮、滑块和滚动条都是合法的。

GUIStyle 可以在没有 GUISkin 的情况下使用以便改写控件的样式。若使用 GUIStyle,只要在脚本中创建一个 GUIStyle 类型的公开变量即可。

在目前的游戏市场上,手游依然是市场上的主力军。然而,只有快速上线,玩法系统完善的游戏才能在国内市场中占据份额。在手游开发过程中,搭建 UI 系统是非常基本且重要的技能。

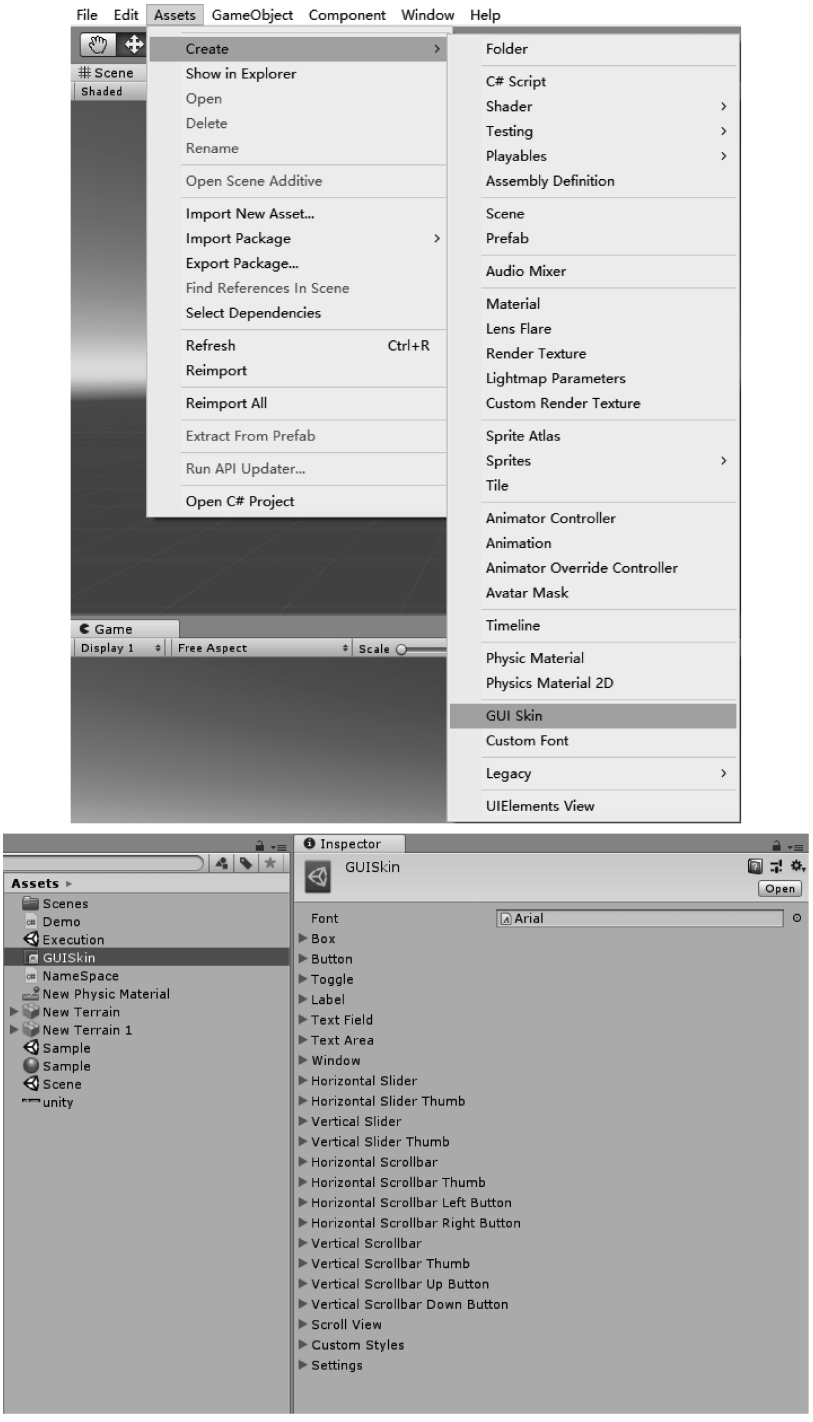


图 5.112　创建 GUISkin

最简单的事要做好也是有难度的。UI 这块的变动通常也是整个游戏中最烦琐的一块，如果没有一个合理的设计思路和管理方案，后期将会陷入无止境的调试优化之中。下面开始从 Unity 中的 UGUI 系统进行讲解。

步骤 1：创建一个 UI 画布

直接新建场景，右击 Hierarchy 窗口，选择 UI 选项，单击列表中出现的 Canvas（画布）选项，如图 5.113 所示。

Canvas：UI 的画布，UI 图片都会在这下面渲染。

EventSystem：UI 的事件系统。很多新手都会选择遗忘掉这个组件，结果导致制作的按钮无法单击，其原因就是误删了这个物体。

步骤 2：创建一个 Image 组件

在 Canvas 上右击，选择 UI 选项中的 Image 选项，如图 5.114 所示。

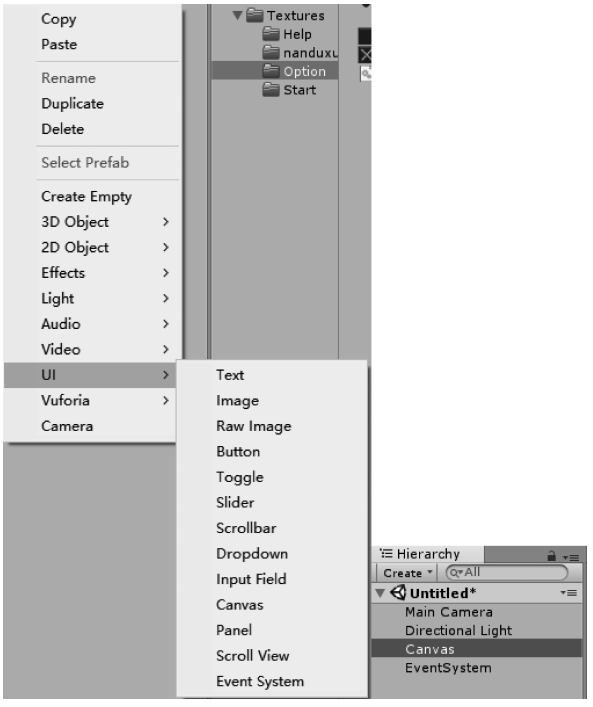


图 5.113　创建一个 UI 画布

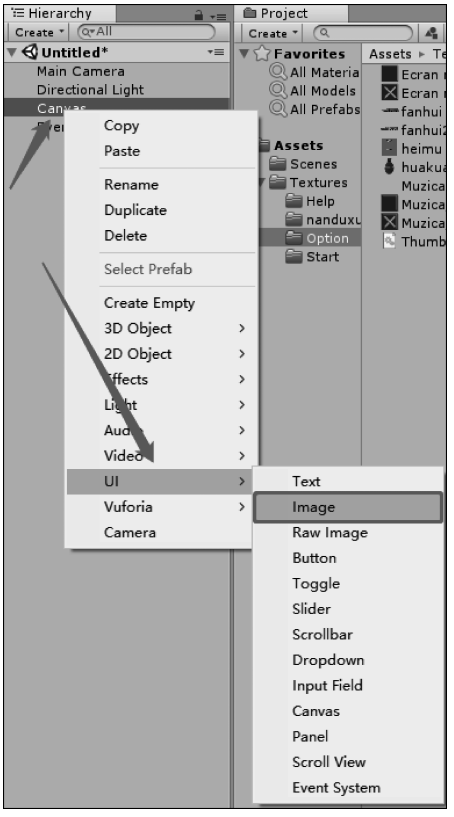


图 5.114　创建一个 Image 画布

此时，一个默认的 Image 图片出现在了游戏框之中，如图 5.115 所示。

注意：UI 的图片只会在 Canvas 下才能看得见。若将 Image 移出 Canvas，镜头内的图片将会消失。

UI 的 Rect Transform 组件中涵盖了位置、旋转、缩放、锚点等信息，如图 5.116 所示。对各项的介绍如下。

（1）Width 和 Height：一般 UI 里面放大和缩小图片的宽度和高度都是以此来控制的，而不是直接调整缩放值。

（2）Anchors：锚点位置。当屏幕的宽高变化时，若依旧希望 UI 按照预想的正常显示，就需要通过锚点来定位。

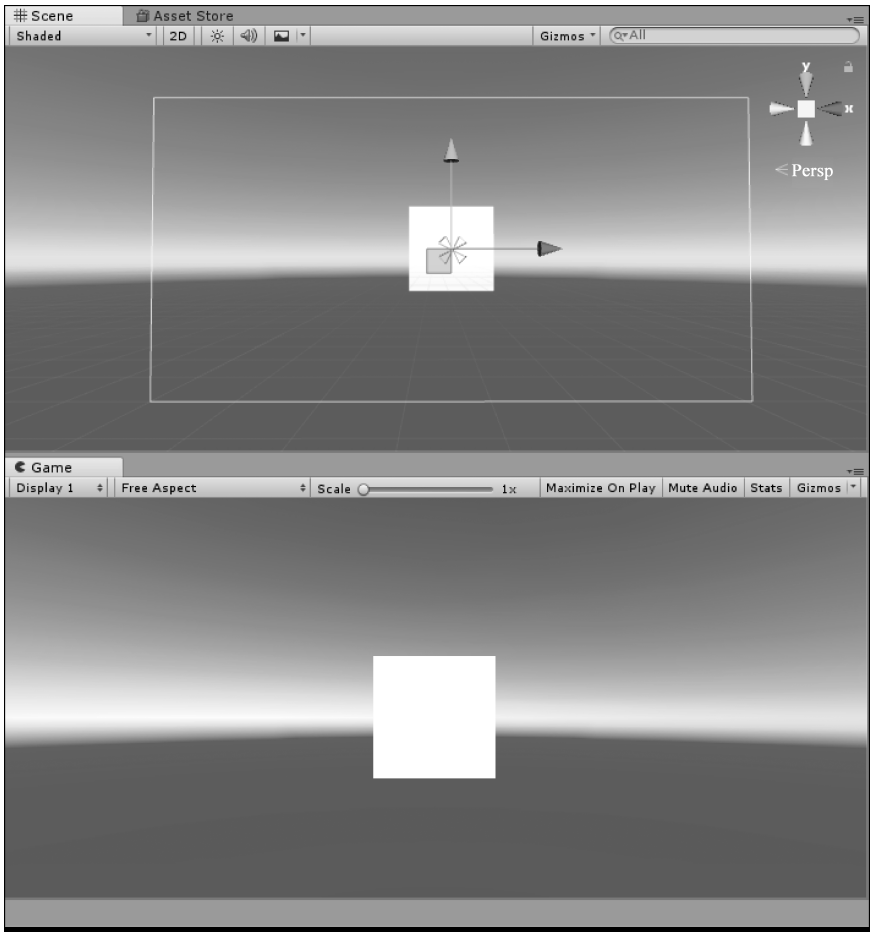


图 5.115　效果

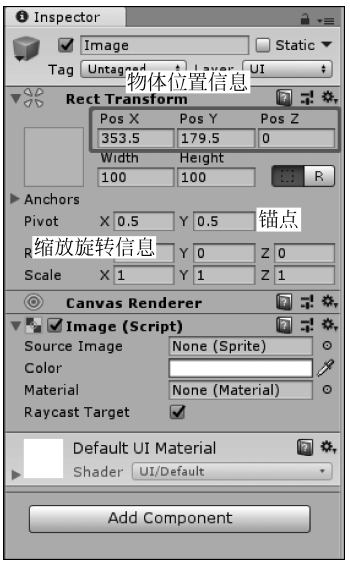


图 5.116　设置 Rect Transform 组件参数

(3) Pivot：中心点。该属性定义图片的中心点位置，(0.5,0.5)刚好为图片中心。若想左右拉长一个横条，想让它只在右边增长，修改中心点位置(0,0.5)，中心点位于最左边，调整 Width 就会只看到横条在右方向的长度变化。

5.6.6 Image 组件

Unity 中大多用于图片显示的 UI 组件都会有基础的 Image 组件。这个基础 Image 组件提供了如下属性。

(1) Source Image：UI 显示的图片资源，注意这里只能支持 Sprite 类型的图片，后面会介绍 Sprite 类型的图片的设置。

(2) Color：修改图片的颜色。

(3) Material：Unity 支持自定义图片材质实现复杂的效果，不填则默认只用 Unity 已经设置好的 UI 材质效果。在游戏设计中几乎不会修改此内容。

(4) Raycast Target：勾选此选项后，UI 将会响应射线单击，单击到这个 UI 物体的时候，事件管理器会知道单击了什么物体。该参数与 Button 组件配合，即可完成单击操作。

下面通过一个例子来看如何创建一个 UI 图片。

首先，导入一张图片，选择 Texture Type 的类型为 Sprite(2D and UI)后，单击 Apply 按钮。这时 Unity 会修改图片为 Sprite 类型的图片，只有这种类型才能放入 Image 组件中，如图 5.117 所示。

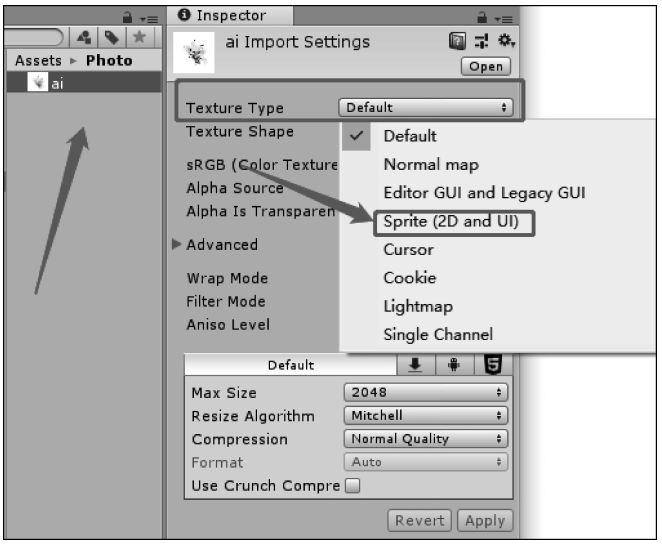


图 5.117　创建一个 UI 图片(1)

然后，直接将图片拖入 Image 的 Source Image 中，图片便渲染出来了，此时图片采用的是 100×100 像素。若单击 Image 新出来的按钮就可以设置为图片本身的像素尺寸，如图 5.118 所示。

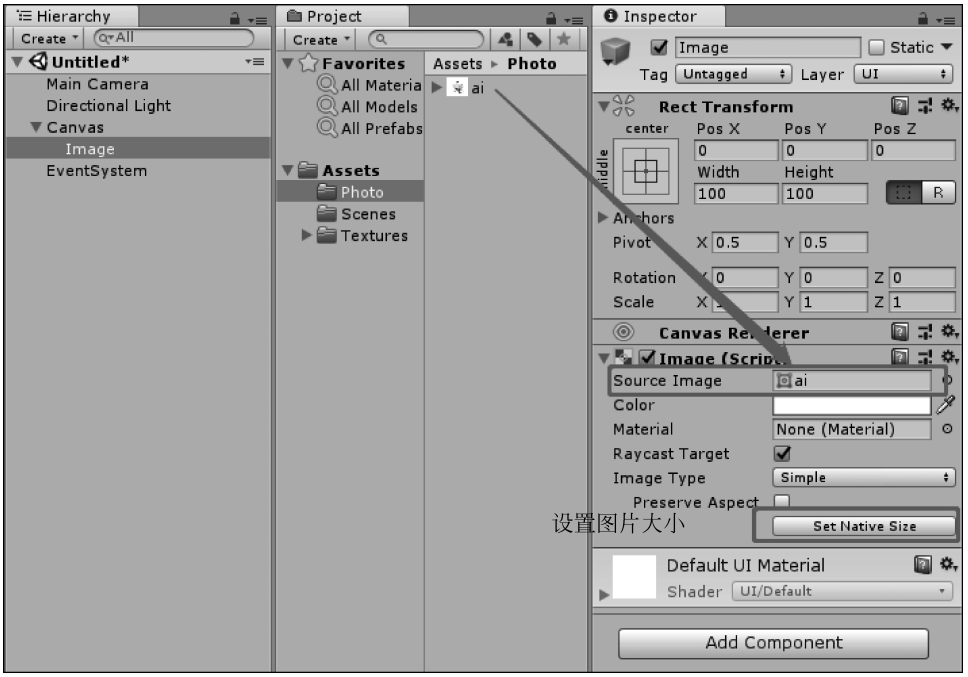


图 5.118　创建一个 UI 图片(2)

单击 Set Native Size 按钮,图片效果如图 5.119 所示。

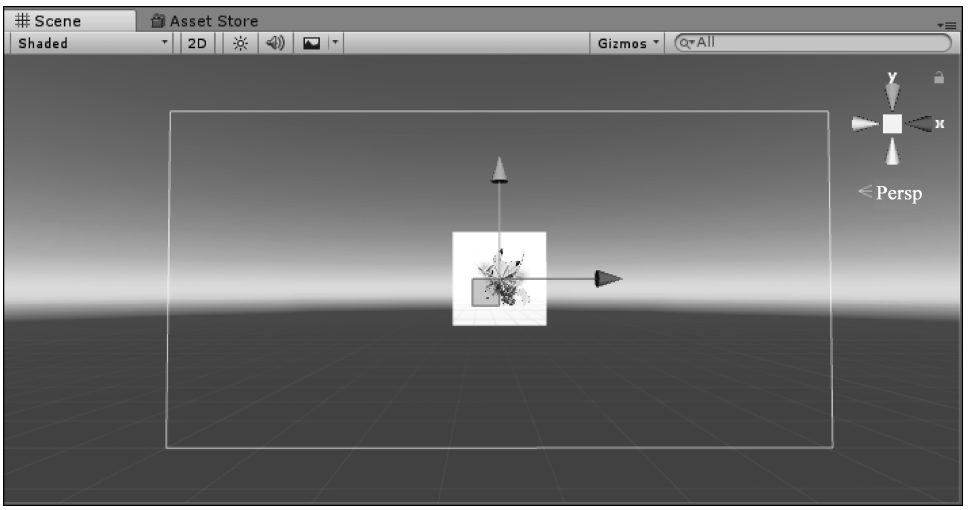


图 5.119　效果

如果要创建一个 Button 按钮,可以右击选择 UI 中的 Button 选项,如图 5.120 所示。

创建出来的 Button 只有 Button 和 Text 两个物体,Text 是 Unity 的文字显示组件,Button 的功能本身和 Text 没有任何关联,因此这里可以将 Text 删除掉(Unity 将 Text 和 Button 一起创建主要是因为按钮带文字更加常见)。

Button 物体上只有两个组件，一个组件是之前介绍过的 Image 组件，一个是按钮功能相关的 Button 组件。我们将一张新的图导入工程，修改图片格式为 Sprite 后拖到 Image 上，然后单击 Set Native Size 按钮修改 Rect Transform 中的宽度和高度，使其和原图片相同，如图 5.121 所示。

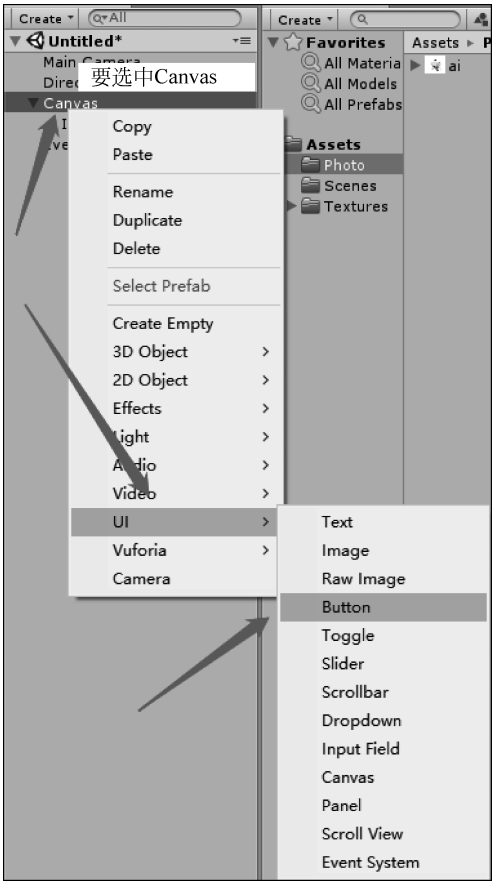


图 5.120　创建一个 Button 组件(1)

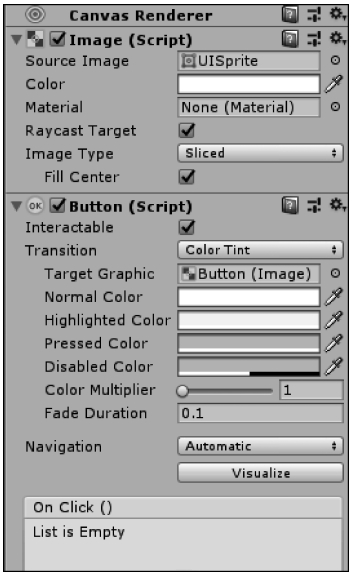


图 5.121　创建一个 Button 组件(2)

各个参数的含义如下。

(1) Normal Color(默认颜色)：初始状态的颜色。

(2) Highlighted Color(高亮颜色)：选中状态或是鼠标靠近会进入高亮状态。

(3) Pressed Color(按下颜色)：单击或是按钮处于选中状态时按下 Enter 键。

(4) Disabled Color(禁用颜色)：禁用时颜色。

(5) Color Multiplier(颜色切换系数)：颜色切换速度，越大则颜色在几种状态间变化速度越快。

(6) Color Tint(颜色改变过渡模式)：颜色变化的过渡模式。

(7) Fade Duration(衰落延时)：颜色变化的延时时间，越大则变化越不明显。

(8) Interactable(是否可用)：勾选，按钮可用；取消勾选，按钮不可用，并进入 Disabled 状态。

(9) Transition(过渡方式)：按钮在状态改变时自身的过渡方式：Color Tint(颜色改变)、Sprite Swap(图片切换)、Animation(执行动画)。

(10) Target Graphic(过渡效果作用目标)：可以是任一 Graphic 对象，如图 5.122 所示。

其中，通过设置 Navigation 来定义如何通过键盘、手柄来切换 Button 的焦点，使其进入下一个 Button。

图 5.122 中通过 On Click 设置鼠标被单击时触发的事件。也可以直接通过 Inspector 窗口设定鼠标被单击的事件，如图 5.123 所示。

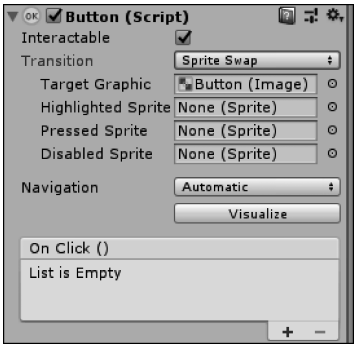


图 5.122　创建一个 Button 组件(3)

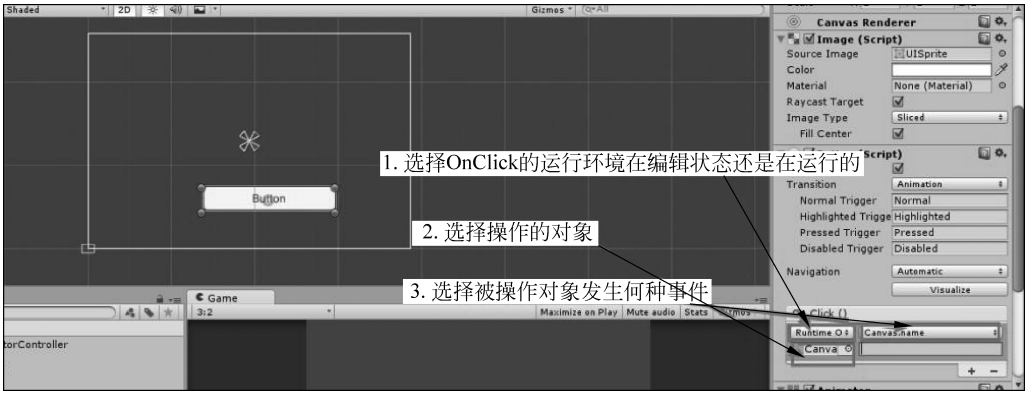


图 5.123　Inspector 窗口设定鼠标被单击的事件

5.6.7　Text 组件

Text 组件负责显示 Unity 中的文本信息。首先在 UI 中创建出 Text，如图 5.124 所示。

由于参数还未开始设置，因此上面的 Text 创建出来不明显。首先来看下 Text 组件的参数。

(1) Font：字体设置，Unity 默认字体是 Arial，可从计算机中选取其他字体替换，也可以从网上下载放在 Unity 中替换，如图 5.125 所示。

(2) Font Style：字体的加粗、倾斜等设置。

(3) Font Size：字体大小设置，注意如果字体设置过大，超过了 Rect Transform 设置的宽度或高度将不会显示字体(很多时候美术 PS 中的字体大小和 Unity 的字体大小有区别，应该统一使用像素单位)。

(4) Line Spacing：行间距，即当前字体大小的倍数。

(5) Rich Text：富文本选项。如果勾选该选项，可以通过加入颜色命令字符来修改字体颜色(如＜color＝＃525252＞变色的内容＜/color＞)。游戏公告的编辑就需要该功能。

(6) Alignment：设置文件上下左右居中等对齐效果。

(7) Align By Geometry：几何对齐，图文混排的时候需要该功能配合。

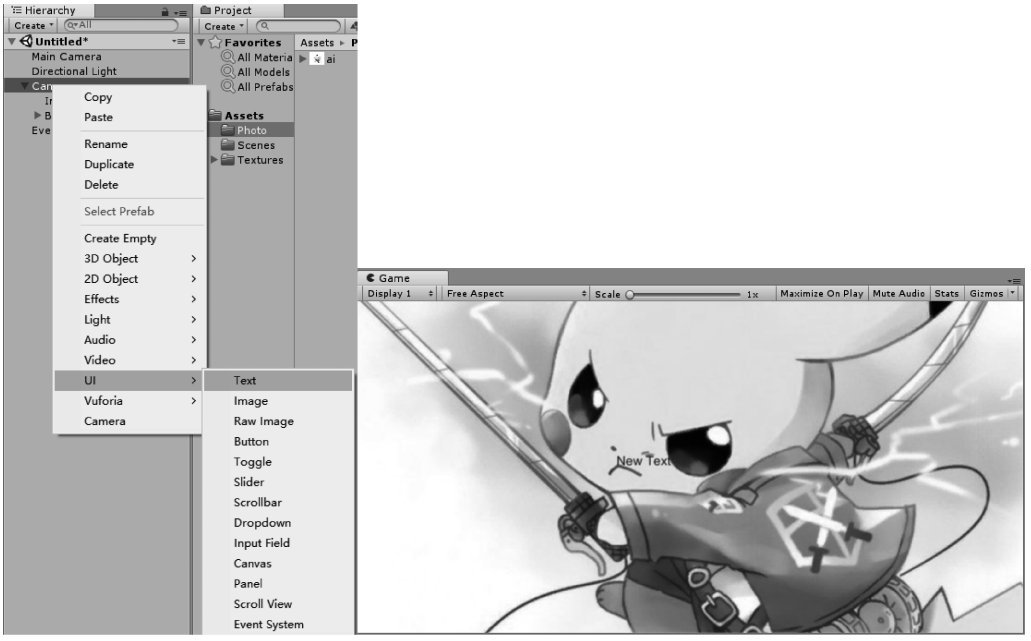


图 5.124　创建一个 Text 组件

图 5.125　设置 Text 组件的参数

(8) Horizontal Overflow 和 Vertical Overflow 分别为水平和竖直换行，如果选择 Wrap 和 Truncate 选项，内容将会束缚在规定宽度高度之内；如果选择 Overflow 选项，内容将会超出设定的边界。

(9) Best Fit：勾选此选项，字体将会以 Rect Transform 的宽度、高度为边界，动态修改字体大小让所有内容刚好填充满整个框。

(10) Color：字体颜色。若用了富文本修改颜色，则不会改变用到了富文本的字体颜色。

(11) Raycast Target：和 Image 一样，勾选该选项后，UI 会屏蔽射线，鼠标单击到这个字体的时候下面如果有按钮区域，响应将会被中止。

简单处理 UI 的遮挡关系：UGUI 中的层级是根据 Hierarchy 中物体的上下关系来决定的。Button 在 Image 的下面，所以游戏窗口中 Button 遮挡了 Image。

如图 5.126 所示对 Button 进行了修改。

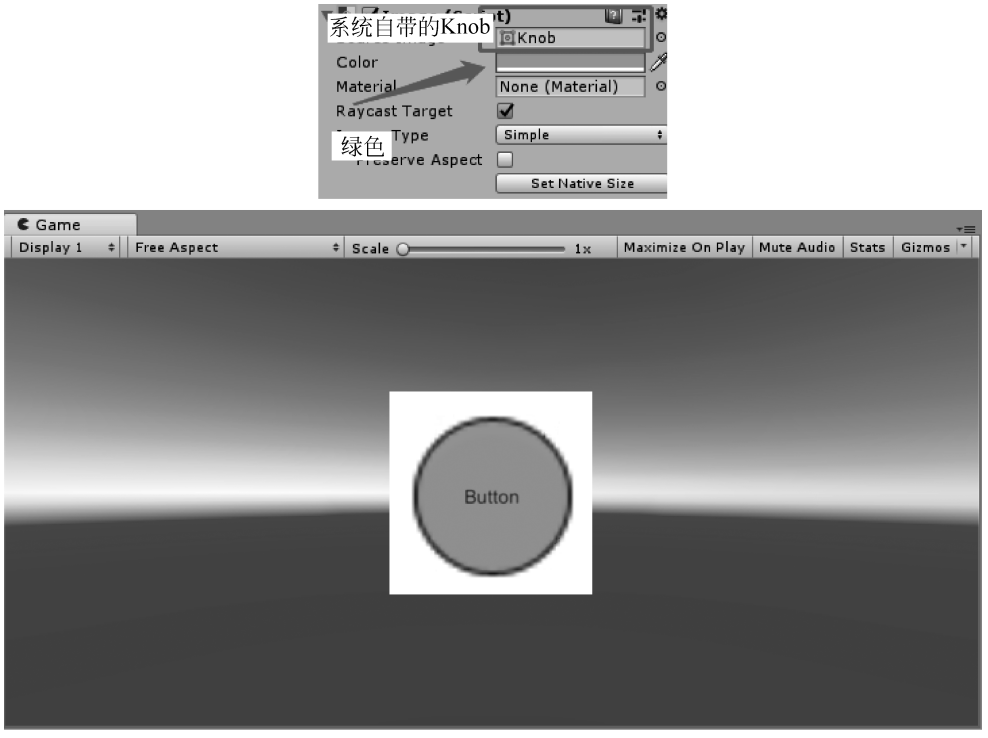


图 5.126　处理 UI 的遮挡关系

首先是修改 UGUI 的自适应。游戏中的分辨率自适应主要做两方面的工作：调整画布组件和调整锚点。

(1) 调整画布组件。

UGUI 中 Canvas Scaler 组件是调整整体缩放的，有三种模式，如图 5.127 所示。

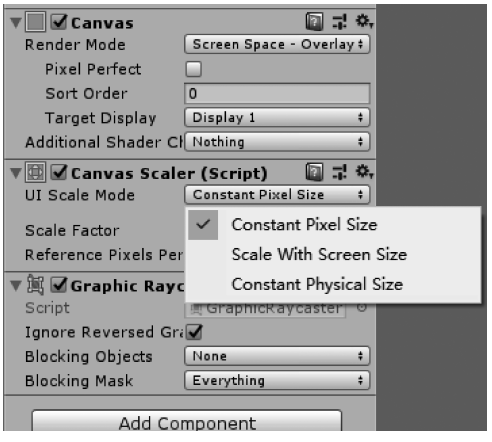


图 5.127　UGUI 中 Canvas Scaler 组件

Constant Pixel Size：固定像素尺寸。在任何分辨率下都不会进行缩放拉伸，只能通过改变 Scale Factor 才能进行(如果需要制作屏幕的分辨率自适应，不推荐使用)。

Constant Physical Size：保持物理上不变的方式，无论场景怎样变化，应用场景较少。

Scale With Screen Size：根据屏幕尺寸缩放，应用场景较多，主要应用在分辨率自适应上，下面是对其参数的详细讲解。

Reference Resolution：开发时分辨率，后续缩放的主要参考对象，一般使用主流分辨率，如 1920×1080、1136×640 等。

Screen Match Mode 的三种模式如下。

① Match Width Or Height：屏幕的宽度和高度对 UI 大小的影响。

② Expand：缩放不裁剪。当屏幕分辨率和设定不同时，选择变化较小的方向进行缩放。

③ Shrink：缩放裁剪。当屏幕分辨率和设定不同时，选择变化较大的方向进行缩放。一般默认选择就可以。

(2) 调整锚点。

每个 UI 都有自己的锚点，它们的锚点是由 4 个三角形表示，并且还有 4 个基准点(用来控制 UI 的大小)。

这时 Button 是子控件，Canvas 是主控件。当主控件被设置为自动拉伸时，子控件和锚点的距离(不是比例)将会永远保持不变。

经过总结得出锚点的设置规律如下。

① 当 4 个锚点在一起时，UI 不会因为窗口的改变而被压缩变形，但是它可能超出主控件。

② 当 4 个锚点全部分开时，UI 对象会随着父节点的改变而改变。

③ 当锚点左右两边分开时，UI 对象的高不会随着父节点的改变而改变，宽会随着父节点的改变而改变。

④ 当锚点上下两边分开时，UI 对象的宽不会随着父节点的改变而改变，高会随着父节点的改变而改变。

在做自适应屏幕时，可以根据自己的需要，合理地选择锚点的位置，如图 5.128 所示。

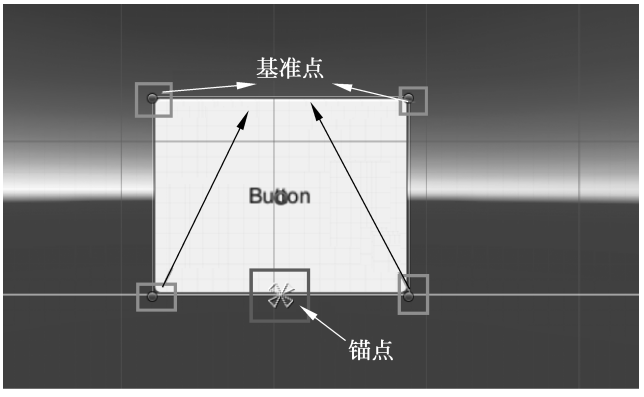


图 5.128　设置 UI 的锚点

5.6.8 创建一个界面

首先创建一个新场景，改名字为 UGui，如图 5.129 所示。

在 Hierarchy 场景中创建 Canvas，然后在 Canvas 里面添加 RawImage，最后在 RawImage 里面添加两个 Button。首先给 RawImage 添加图片，此处随便使用一张图片，效果如图 5.130 所示。

给 Button 添加图片，然后在 Button 下面的 Text 中写“百度”和“Unity”。最后用代码去连接两个 Button。

图 5.129　创建一个界面

图 5.130　添加图片的效果

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
public class TestOpen : MonoBehaviour {
    // 初始化
public Button[ ] myButton;
    void Start () {
myButton[0].onClick.AddListener(delegate { Application.OpenURL("https://www.baidu.com/"); });
myButton[1].onClick.AddListener(delegate { Application.OpenURL("https://store.unity.com/"); });
    }
    // 更新
    void Update () {
    }
}
```

单击 Button 后，效果如图 5.131 所示。

图 5.131　按钮的效果展示

单击左边的“百度”按钮直接弹出百度网页。单击右边的 Unity 按钮直接弹出 Unity Store 网页。

一个简单的用户界面就完成了。

5.7 Mecanim 动画系统

新版的 Unity 采用新的动画系统 Mecanim 取代旧系统 Animation。

5.7.1 基本知识

创建动画的一个基本步骤是设置一个从用户提供的骨架到 Mecanim 系统骨架的映射；在 Mecanim 的术语中，这个映射称为 Avatar，即骨骼到骨架的映射，如图 5.132 所示。

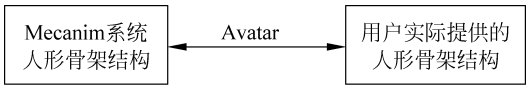


图 5.132 Avatar 映射

Avatar 主要用于类人骨骼模型，可以实现角色之间的 Retargeting。非类人模型认为骨架就是骨骼。

构建模型的基本步骤：Modelling(建模)→Rigging(构建骨架)→Skinning(蒙皮)。

步骤 1：Modelling 建模

(1) 遵循合理的拓扑结构，一个合理的标准是动画带动的网格变形是漂亮的。

(2) 注意网格的缩放比例。最好做一下各个建模软件模型的导入测试，从而设置好正确的缩放比例(不同建模软件导入比例不一样)。

(3) 安放角色，使得角色的脚站在坐标原点或者模型的“锚点”。角色通常是竖直地走在地面上，如果角色的锚点(也就是它的变换中心)在地面上会更容易控制。

(4) 如果是类人模型，则尽量使用 T 字姿态建模(Unity 为类人模型提供了许多功能和优化)。

(5) 整理模型，去掉垃圾。如果可能的话，覆盖孔洞，焊接顶点并且移除隐藏的面，这会对蒙皮有帮助，特别是自动蒙皮过程。

步骤 2：Rigging 构建骨架

Rigging 的目的是创建骨架上的关节控制模型的运动。

对于非类人模型，可以认为没有骨架，只有骨骼，骨骼直接控制动画；类人模型是骨架控制动画。上述模型已经有脚、手、头、武器等骨骼，还有受击骨骼等，这些可以用来控制模型或者悬挂额外物件。

步骤 3：Skinning 蒙皮

蒙皮的方法就是给骨架附加网格。

(1) 把网格中的顶点绑定到骨骼，包括硬绑定(一个顶点指定一个骨骼，不是一一对应，可能多个顶点指定的是一个骨骼)和软绑定(一个顶点指定多个骨骼，每个骨骼有一定权重)。

(2) 蒙皮的实操步骤：先自动蒙皮，接着用一个测试动画看看蒙皮效果并根据此效果慢慢改。

(3) 每个顶点最多绑定 4 个骨骼，这是 U3D 的上限。

5.7.2 动画应用

在 Animations 页面中可以给 Clip 加帧事件，即播到某帧时触发某个事件。

(1) 给 Clip 的某些帧上加 Event，Function 就是事件的名字，其他的是这个函数的参数。

(2) 定义一个脚本来接收这个事件，比如这个图需要定义一个脚本，并在脚本里定义了 void Ani(xxx)()函数。

(3) 参数的处理：根据脚本的函数定义格式传参数，比如 void ani(int a)，则传 int 格式参数，ani(Object a)则传 Object 格式参数，ani(float a, string b)则传 float 和 string 两个格式的参数，而 ani(AnimationEvent a)则传整个 Event(包括所有参数和当前 Event 对应的 Clip 的信息)。添加了 Event 的 Clip 的 Animator 物体必须挂上定义了该事件名 Function 的函数的脚本，否则会报错。

Unity Script 部分可查看 class AnimationEvent，主要是获得该 Event 所在 Clip 的一些信息，包括与此 Event 相关的 stateinfo、clipinfo，以及该 Event 本身的信息，如 Event 调用的函数名 functionName、函数调用的参数 float/int/string/object(采用哪个看函数定义式)、time(事件的触发时间)。

这个 Event 机制可以在播到某些帧时执行一些事件，例如在某个点播放某个特效，在某个点播放某个声音，在某个点进行一些画面特效，在某个点进行对敌人“击退”“击飞”“击浮空”，从而实现各种节奏效果。还可以加入技能打断机制：当播到某两个帧之间时可以被打断。可以在一个专门的脚本中定义各种接收事件的函数，并进行相应处理来使用此 Event 机制。

5.8 导航系统

本节学习的是 Unity 的导航系统。导航系统需要用到 Navigation 面板，首先来学习如何打开 Navigation 面板。

5.8.1 导航面板

单击 Windows 菜单下的 Navigation 选项，打开 Navigation 面板，如图 5.133 所示。

Navigation 面板中包括以下几个模块。

(1) Agents：可以添加多个 NavigationAgents，也可以用不同的 Agents。

(2) Areas：可以设置自动寻路烘焙的层。

(3) Bake：烘焙参数的设置。

(4) Object：设置去烘焙哪个对象，比如地形之类的，就是可以行走的范围路径。

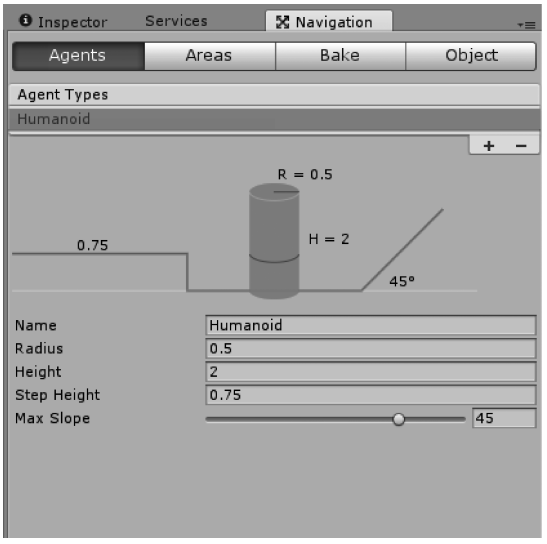


图 5.133　Navigation 面板

5.8.2　导航步骤

(1) 在场景中摆放各种模型,包括地板、斜坡、山体、扶梯等。

(2) 为所有的模型加上 Navigation Static 和 Off Mesh Link 组件(这个可根据需要,例如地板与斜坡相连,斜坡就不需要添加 Off Mesh Link)。

(3) 特殊处理扶梯,需要手动添加 Off Mesh Link,设置好开始点和结束点。

(4) 保存场景,烘焙场景。

5.8.3　上下斜坡

在用 Unity 的自动寻路系统的时候,如果人物不能按照规定到达目的地,很可能是因为烘焙寻路出现了问题,所以这是需要重视的地方。下面就是一开始烘焙的寻路,这里有个问题,就是在两个红圈的位置是没有烘焙上的,并且区域很大,当人物寻路到这里的时候很容易卡在这里,如图 5.134 所示。

设置烘焙的参数:将烘焙半径调小点儿就可以解决这个问题。将烘焙半径设置为 0.1,烘焙效果如图 5.135 所示。上坡和下边的地面连接处没有烘焙上的区域就很小了。

斜坡角度和连接问题。如果上坡的角度很大,人物也会卡在上坡中,现在设置的上坡角度是 40°。如果把角度设置为 30°或者以下,人物就可以很顺利地爬上斜坡。如果下坡的角度很大,人物就会直接跳下斜坡,现在设置的下坡角度是 50°。从图中可以看到人物是直接跳下来的。如果把角度设置为 40°或者以下,人物就可以很顺利地下斜坡了。

还有就是斜坡与地面和站台连接处的问题,它们的连接之间一定不能有空隙,否则人物也容易卡在空隙处。如图 5.135 所示,斜坡与站台没有完全连接上,有个很小的缝隙,即使寻路烘焙也没有问题,人物有时候也会卡在这个地方。

人物容易卡在寻路的边缘处。因为寻路就是解决人物查找最短的路径(在忽略消耗体力值前提下),并最终到达目的地的问题,所以在上下坡时也经常会遇到人物沿着斜坡运动,

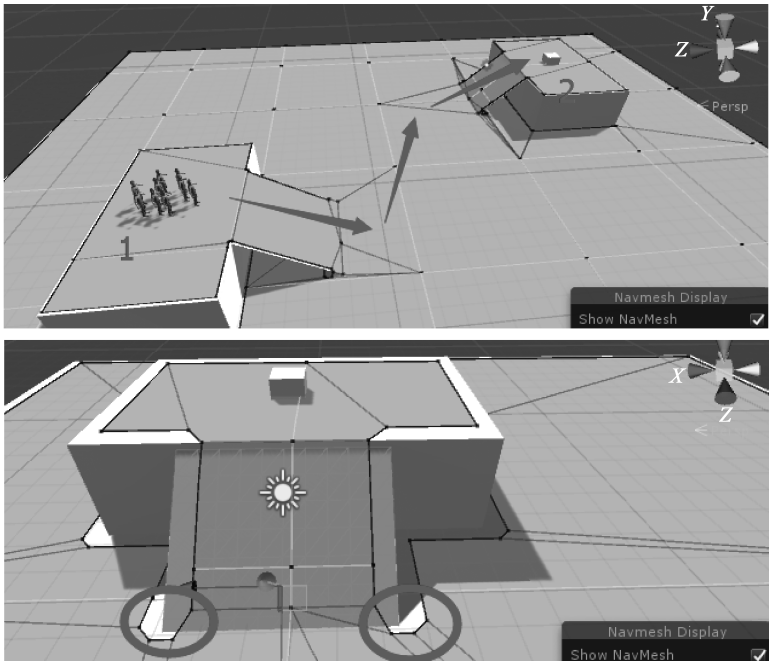


图 5.134　上下斜坡

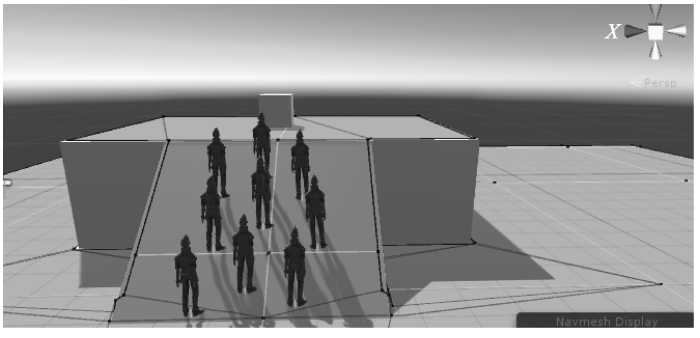


图 5.135　斜坡角度问题

这就可能使人物卡在烘焙好的寻路边缘处。解决办法是设置中间目标物，让其绕开寻路边缘，这就需要设置几个中间目标，当人物到达一个目标的时候，向着下一个目标运动。

代码如下：

```
using UnityEngine;
using UnityEngine.AI;

[RequireComponent(typeof(NavMeshAgent))]
public class NavigationTest : MonoBehaviour
{
    public Transform targetOne;
    public Transform targetTwo;
    public Transform targetThree;
    private NavMeshAgent navAgent;
```

```
    private float distanceOne;
    private float distanceTwo;
    // 初始化
void Start()
    {
navAgent = transform.GetComponent<NavMeshAgent>();
        navAgent.SetDestination(targetOne.position);
    }

    // 更新
void Update()
    {
CheckReachTarget();
    }

void CheckReachTarget()
    {
distanceOne = Vector3.Distance(transform.position, targetOne.position);
distanceTwo = Vector3.Distance(transform.position, targetTwo.position);

        if (distanceOne < 1f)
        {
            navAgent.SetDestination(targetTwo.position);
        }

        if (distanceTwo < 1f)
        {
            navAgent.SetDestination(targetThree.position);
        }
    }
}
```

5.8.4 自动寻路

本节用一个简单的例子来说明如何使用自动寻路。

(1) 在 Scene 视图中新建三个 Cube,摆放方式如图 5.136 所示。

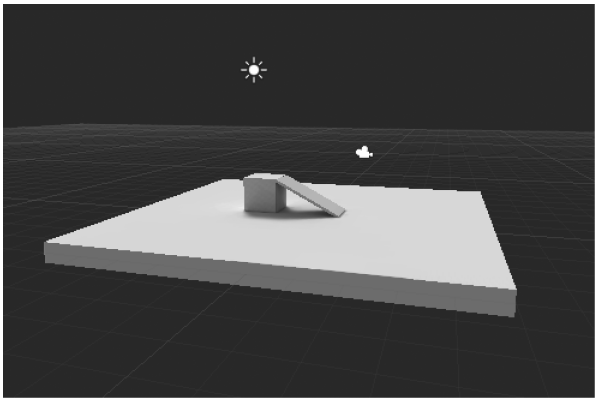


图 5.136 自动寻路(1)

(2) 选中图 5.136 中的三个 Cube,并在 Inspector 面板中选中静态(static)下拉选项的 Navigation Static,依次选择菜单栏中的 Windows→Navigation,打开后面板如图 5.137 所示。

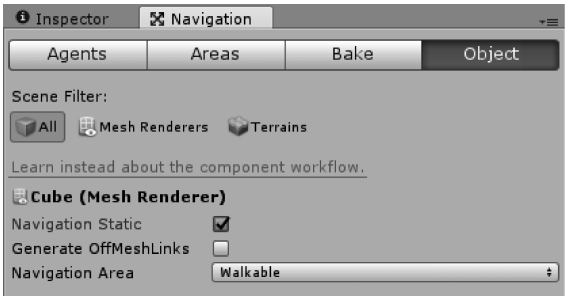


图 5.137　自动寻路(2)

单击该面板右下角的 Bake 按钮,即可生成导航网格。

(3) 下面就可以让一个运动体根据一个导航网格运动到目标位置。

首先新建一个 Cube 为目标位置,起名为 TargetCube;然后创建一个 Capsule(胶囊)运动体,为该胶囊挂载一个 Nav Mesh Agent(Component→Navigation→Nav Mesh Agent);最后写一个脚本就可以实现自动寻路。脚本如下:

```
using UnityEngine;
using System.Collections;
using UnityEngine.AI;
public class Run : MonoBehaviour
{
    public Transform TargetObject = null;

void Start()
    {
if (TargetObject != null)
        {
            GetComponent<NavMeshAgent>().destination = TargetObject.position;
        }
    }
void Update()
    {

    }
}
```

脚本新建完成后挂载到胶囊体上,然后将 TargetCube 赋予胶囊体的 Run 脚本,运行场景如图 5.138 所示,胶囊体会按照箭头的方向运动到 Cube 位置。

这样一个简单的自动寻路就完成了。如果要更精细地寻路,或要实现上坡、钻“桥洞”等,可根据下面介绍的相关参数进行调节。

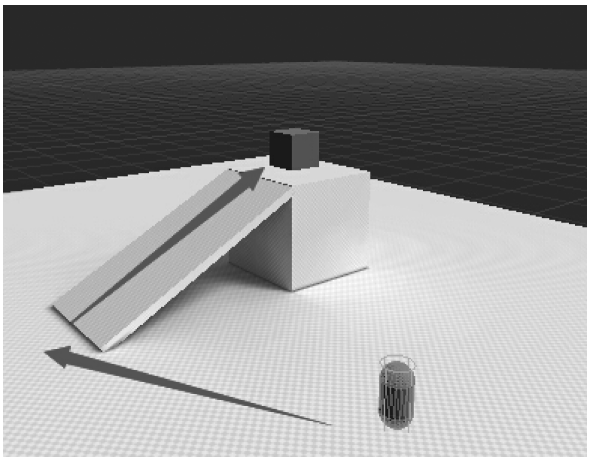


图 5.138　自动寻路(3)

5.8.5　导航组件

下面介绍 Navigation 组件和 Nav Mesh Agent 组件的相关参数。

1. Navigation 组件

Object：物体参数面板。

Navigation Static：勾选后表示该对象参与导航网格的烘焙。

Off Mesh Link Generation：勾选后可跳跃(Jump)导航网格和下落(Drop)。

Bake：烘焙参数面板。

Radius：具有代表性的物体半径，半径越小生成的网格面积越大。

Height：具有代表性的物体的高度。

Max Slope：斜坡的坡度。

Step Height：台阶高度。

Drop Height：允许最大的下落距离。

Jump Distance：允许最大的跳跃距离。

Min Region Area：网格面积小于该值则不生成导航网格。

Width Inaccuracy：允许最大宽度的误差。

Height Inaccuracy：允许最大高度的误差。

Height Mesh：勾选后会保存高度信息，同时会消耗一些性能和存储空间。

2. Nav Mesh Agent 组件

Radius：物体的半径。

Speed：物体的行进最大速度。

Acceleration：物体的行进加速度。

Angular Speed：行进过程中转向时的角速度。

Stopping Distance：离目标距离还有多远时停止。

Auto Traverse Off Mesh Link：是否采用默认方式通过链接路径。

Auto Repath：在行进中因某些原因中断后是否重新开始寻路。

Height：物体的高度。

Base Offset：碰撞模型和实体模型之间的垂直偏移量。

Obstacle Avoidance Type：障碍躲避的表现登记，None 选项为不躲避障碍。另外，等级越高，躲避效果越好，同时消耗的性能越多。

Avoidance Priority：躲避优先级。

Nav Mesh Walkable：该物体可以进行的网格层掩码。

下面通过一个例子来说明如何在 Navigation 中实现高低落差以及跳跃。

不管是爬楼梯还是跳跃，Nav Mesh 都是通过 Off Mesh Link 实现的。创建 Off Mesh Link 的方法有两种，接下来通过例子进行说明。为了实现这个例子，预先在场景里面准备了一些物体：摄像机是必需的，一个作为地面的 Plane，然后是 F1～F5 几个高低落差不一样的台阶，L1 和 L2 是楼梯模型，控制人物主体 man，还有移动的目标点 target。其中，man 身上必须带有 Nav Mesh Agent 组件。为了观察方便，在 target 身上带了 Light 组件。

按照上面所讲的，Plane 和 F1～F5 台阶在 Navigation 面板中勾选 Navigation Static 选项，然后 Bake，观察 Scene 视窗，会发现已经生成了所要的 Nav Mesh 网格，现在可以像上面那样在 Plane 上面给人物做寻路和移动了，但人物是不会爬楼梯的。

这时找到 L1 楼梯，在楼梯的开始和结束的位置放置两个点，这两个点只需要拾取它的位移，可以用 empty GameObject 来做。为了便于观察，此处就拿 Cube 来做。开始点命名为 startPoint，结束点命名为 endPoint，如图 5.139 所示。

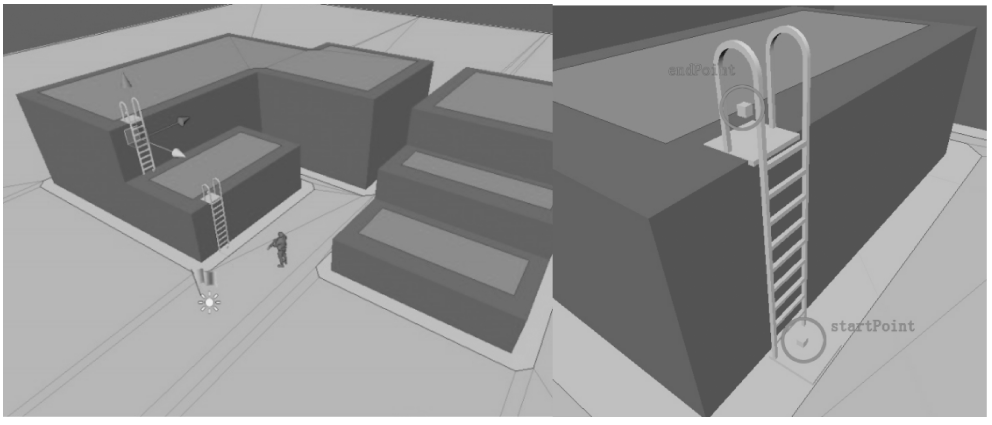


图 5.139　导航举例(1)

注意：startPoint 和 endPoint 的位置要比所在的平面稍微高一点点儿。

接下来介绍第一种生成 Off Mesh Link 的方法。选择 L1 楼梯，然后在 Component 下拉选项中选择 Navigation→Off Mesh Link。

选择后，Off Mesh Link 组件已经添加到了 L1 的身上，可以在 Inspector 面板看到。

把刚才放置在场景里面的 startPoint 和 endPoint 指定到 Off Mesh Link 组件的 Start 和 End 位置，其他选项默认不改变。

再次 Bake。现在，在 Scene 面板里，startPoint 和 endPoint 之间生成了一条线，而方向是从 startPoint 指向 endPoint 的。这时候可以通过移动目标点让角色开始爬楼梯，但爬上去之后角色暂时不能跳下来。如果把目标点移动到 Plane 上，角色会顺着楼梯爬下来。

使用同样的方法对 L2 生成 Off Mesh Link。这个时候，角色应该可以爬两层楼梯了。至此，第一个目标完成了。

接下来进行第二个目标的制作，首先来分析一下场景：我们希望人物能从 2.5m 的高度往下跳。若超过 2.5m，因太高会有危险，人物就不能跳。然后横向希望人物能跳过 2m 的沟。

根据这个设定，场景会是这样的情况：L1 和 L2 只能通过爬楼梯，L2 和 L3 之间可以跳跃，L3～L5 是可以往下跳的。

于是，在 Navigation 面板里面找到 Bake 栏，Drop Height（掉落高度）填 2.5，Jump Distance（跳跃距离）填 2，单位都是 m。

接下来介绍第二种生成 Off Mesh Link 的方法：选中 L1～L5 的物体，在 Navigation 面板的 Object 栏里把 Off Mesh Link Generation 选项勾选上。场景里面会出现很多新的 Off Mesh Link，这是 Unity 通过计算把可以跳跃或者下落的地方自动生成了 Off Mesh Link。这时应该已经可以通过移动目标点，让角色进行跳跃和下落了。进行到这里，第二个目标也完成了。

在制作过程中，假如没有这个大兵的模型，而是用一个胶囊体代替人物，其爬楼梯和跳跃的时候好像是在一瞬间完成的，没有大兵那个爬楼梯和跳跃动作的过程。

图 5.140 所示为导航举例示意图。

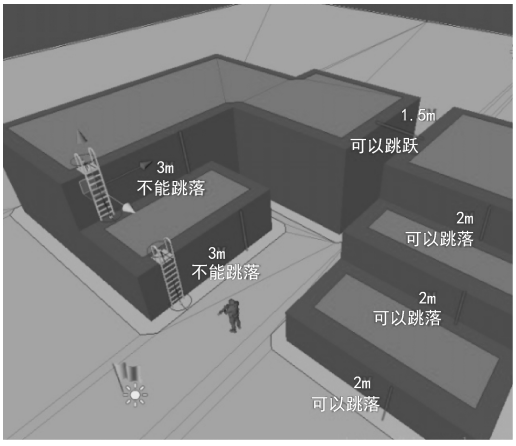


图 5.140　导航举例(2)

因为默认的 Nav Mesh Agent 组件中是勾选了 Auto Traverse Off Mesh Link（自动通过 Off Mesh Link）选项的，这意味着：人物只要到了 Off Mesh Link 的开始点，就会自动地移动到 Off Mesh Link 的结束点。

假如需要对越过 Off Mesh Link 时进行控制，则需要另外写脚本。这里简单地介绍一下方法。首先用状态来控制角色的概念，比如人物可以分为站立、走路、跑步、上下楼梯、横向跳跃和往下掉落几种状态。针对 NavMesh 来说，人物简单地分为站立、正常的 NavMesh 寻路和通过 Off Mesh Link 移动几种状态。首先把 Auto Traverse Off Mesh Link 选项取消。然后，通过人物在 Off Mesh Link 移动的状态（可以用 NavMeshAgent.isOnOffMeshLink 判断），获取到当前通过的 Off Mesh Link：OffMeshLinkData link＝NavMeshAgent.currentOffMeshLinkData；

这样就能获取到 link 的开始点和结束点的坐标(link. startPos 和 link. endPos),这时人物就可以用最简单的 Vector3. Lerp 来进行移动。当人物的位移到达了结束点的坐标,人物的 Off Mesh Link 移动状态就可以结束,又重新变回正常寻路或者站立的状态了。在这个 Vector3. Lerp 的过程中,可以随意地控制人物的爬行或者跳跃的动作。

5.9 音乐音效

本节学习的是 Unity 的音乐音效系统。随着游戏的普及,游戏音乐渐渐出现在了玩家的视野中,同时游戏音效也出现在大家的视野当中。音效,大到整个游戏的背景音乐,小到风吹衣服的声音等,在任何类型的游戏中都是不可或缺的一部分。在游戏中,一个好的音效能让游戏提升一个等级。例如,在青山绿水、优美的环境中,配上一曲优美的古曲,会让人有种身临其境的感觉,极大地提升游戏的快感。

在游戏中,一般存在两种音乐,一种是时间较长的背景音乐,另一种是时间较短的音效(比如按钮单击、开枪音效等)。

Unity 3D 支持下面几种音乐格式。

(1) AIFF: 适用于较短的音乐文件,可用作游戏打斗音效。

(2) WAV: 适用于较短的音乐文件,可用作游戏打斗音效。

(3) MP3: 适用于较长的音乐文件,可用作游戏背景音乐。

(4) OGG: 适用于较长的音乐文件,可用作游戏背景音乐。

5.9.1 音乐组件

Unity 3D 中对音乐进行了封装,总体来说,播放音乐需要 3 个基本的组件。下面分析这 3 个组件。

1. Audio Listener

在创建场景时,一般 Camera 上就会带有这个组件,该组件只有一个功能,就是监听当前场景下的所有音效的播放并将这些音效输出,如果没有这个组件,则不会发出任何声音。幸运的是,一般场景中只需要在任意的 GameObject 上添加一个该组件,无须创建多个该组件,但是要保证这个 GameObject 不被销毁,所以一般按照 Unity 的做法,在主摄像机中添加即可。

2. Audio Source

控制一个指定音乐播放的组件,可以通过属性设置来控制音乐的一些效果,详细内容可以查看官方的文档 http://docs. Unity 3D. com/Manual/class-AudioSource. html。

下面列出一些常用的属性。

(1) Audio Clip: 声音片段,还可以在代码中动态地截取音乐文件。

(2) Mute: 是否静音。

(3) Bypass Effects: 是否打开音频特效。

(4) Play On Awake: 开机自动播放。

(5) Loop: 循环播放。

(6) Volume: 声音大小,取值范围为 0.0～1.0。

(7) Pitch：播放速度，取值范围为－3～3，设置 1 为正常播放，小于 1 为减慢播放，大于 1 为加速播放。

3. Audio Clip

当我们把一个音乐导入 Unity 3D 中，这个音乐文件就会变成一个 Audio Clip 对象，即可以直接将其拖动到 Audio Source 的 Audio Clip 属性中，也可以通过 Resources 或 AssetBundle 进行加载，加载出来的对象类型就是 Audio Clip。Audio Clip 面板有很多参数，设置起来容易出错，如图 5.141 所示。

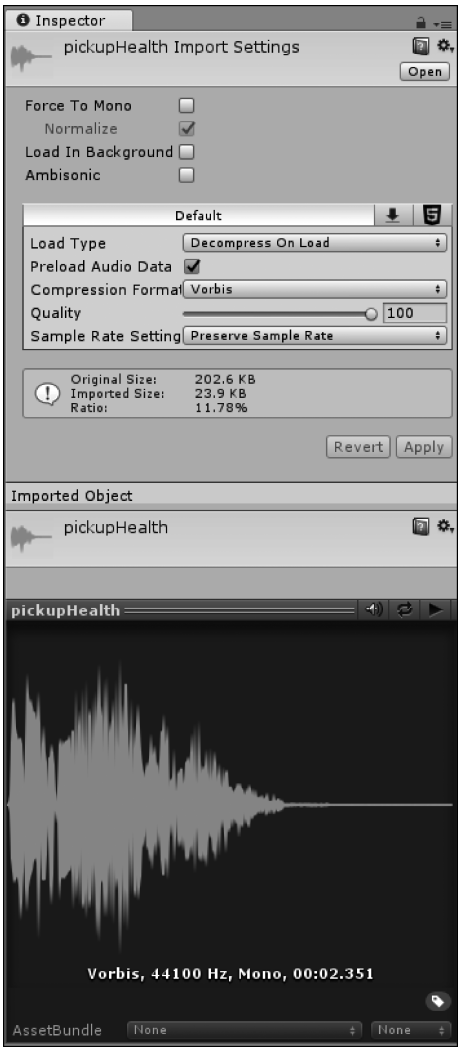


图 5.141　Audio Clip 面板

(1) Force To Mono：将多声道的声音合并成单声道，声音文件大小会小很多，在手机上推荐使用。合并声道之后，勾选 Normalize 复选框可以使声音听起来更优美一些。

(2) Load In Background：在后台加载，使得声音不阻塞主加载线程。它默认是关闭的，官方的说法是为了保证游戏运行时声音体验的一致性。个人觉得如果加载不会引起运行时卡顿，那么相对于提升加载时间和减少加载数量的优势，还是值得将其勾选的。

(3) Preload Audio Data：在进入场景时预加载音效，如果不勾选，直到第一次被使用时才加载。背景音乐无须勾选，但 UI 音效可以勾选，反正基本都是要加载的，这样还不会占用运行时间。

(4) Load Type-Decompress On Load：声音一旦被加载就会解压存储在内存中。这可以提供更好的声音响应，但会占用内存，尤其是 Vorbis 编码的声音，因此比较适合短小的声音。

(5) Load Type-Compressed In Memory：声音在内存中以压缩的形式存储，等播放时再解压。这种方式有轻微的效率消耗，但节省了内存，因此适合 Vorbis 形式的大文件。这部分消耗可以在 Profiler 中 Audio 面板的 DSP CPU 中查看。

(6) Load Type-Streaming：播放时解码。这种方式占用内存最小，却增加了磁盘读写和解压。这部分消耗可以在 Profiler 中 Audio 面板的 Streaming CPU 看到。基本上是大文件才会采用的设置。

(7) Compression Format-PCM：最高的质量，最大的文件。

(8) Compression Format-ADPCM：一些包含噪声，且会被多次播放的音频，可以采用这个格式，例如脚步、打击、武器等。它的 PCM 压缩了约 70%，CPU 消耗却比 Vorbis 小，是

高频、小声音的最佳选择。

(9) Compression Format-Vorbis：压缩更小的文件，但质量不过关。压缩率可以在Quality 面板中配置，可以边听边选，最后确定一个合适的压缩率。

(10) Compression Format-Quality：压缩比率，只对 Vorbis 类型有效。最终文件大小在 Inspector 面板中可以看到。

(11) Sample Rate Setting-Preserve Sample Rate：先前默认的值。

(12) Sample Rate Setting-Optimize Sample Rate：通过最高频率分析优化之后的值。

(13) Sample Rate Setting-Override Sample Rate：自定义的采样率的值，建议用默认的。

5.9.2 播放音乐的例子

Unity 允许通过简单的拖动、单击，并且不写一行代码即可实现音乐的播放。

新建一个场景，给 Main Camera 添加一个 Audio Source 组件，并将音乐文件拖动到 Audio Clip 属性上，勾选 Loop 使其可以进行循环播放，如图 5.142 所示。

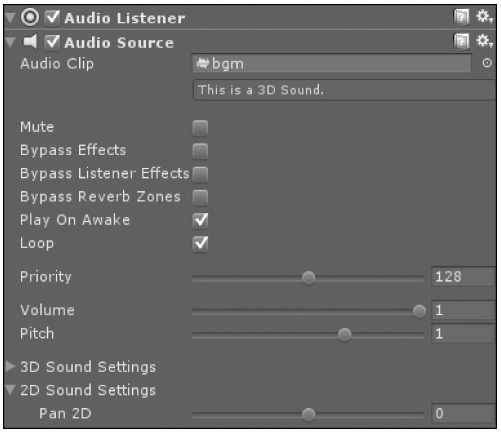


图 5.142 播放音乐

运行程序就可以听到声音。

5.9.3 三维音效

Unity 之所以把音乐播放拆分成 Audio Listener、Audio Source 和 Audio Clip 这 3 个组件，最重要的原因就是实现三维音效。

如果将 Audio Listener 看作一双耳朵，三维音效效果就能很好地被理解。Unity 会根据 Audio Listener 对象所在的 GameObject 和 Audio Source 所在的 GameObject 对距离和位置进行判断，从而模拟真实世界中音量近大远小的效果。

首先，导入所需的音乐文件，必须设置为三维音乐。如果是二维音乐，就不会有近大远小的效果。

新建一个场景，添加 3 个 GameObject，给第一个添加一个 Audio Listener 组件，其他两个添加 Audio Source 组件并赋予两个音乐文件。

移除 Main Camera 上的 Audio Listener 组件，按照下面的位置摆放这 3 个组件，如

图 5.143 所示。

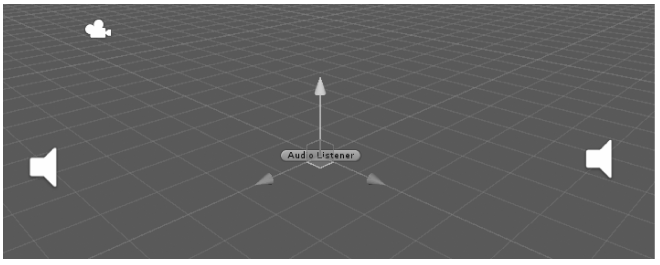


图 5.143　三维音效

运行游戏，返回 Scene 视窗，拖动 Audio Listener 组件的位置，就可以真切感受到类似在两个音响之间移动的效果。（对于每个 Audio Source 声音，可传递的距离可以通过拖动其球形的线条进行调整。）

5.10　VR 实例

5.10.1　飞机引擎拆装

飞机引擎的拆装要能在三维引擎中单击三维模型，并且使三维模型跟随鼠标移动。首先需要准备两套飞机模型，把飞机模型复制到 Unity 资源里。

把模型放入游戏场景中，如图 5.144 所示。

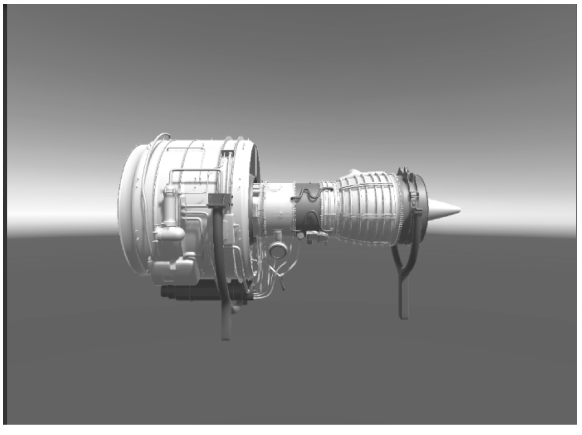


图 5.144　把模型放入游戏场景中

把飞机引擎的各个部分自动分离。

下面是分离各个引擎部分的方法。

```
if(Input.GetKeyDown(KeyCode.Space))
{
    transform.position = Vector3.MoveTowards(transform.position, new Vector3(3, 0, 0), 4);
}
```

先判断是否按下了空格键，如果按下了空格键，执行分离操作。

```
Input.GetKeyDown(KeyCode.Space)//是按下空格键的操作
transform.position = Vector3.MoveTowards(transform.position, new Vector3(3, 0, 0), 0.5f);
```

分离操作，从自身的位置 transform.position，移动到目标点 new Vector3(3，0，0)，0.5f 是每一帧移动的最大距离。

接下来把飞机引擎的各个部分组装起来，先在飞机引擎的各个部分添加下面的脚本。

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
public class MouseMove : MonoBehaviour
{
    //鼠标经过时改变物体颜色
    private Color mouseOverColor = Color.blue;       //声明变量为蓝色
    private Color originalColor;                     //声明变量来存储本来的颜色
    void Start()
    {
        originalColor = GetComponent<MeshRenderer>().sharedMaterial.color;//开始时得到
                                                                          //物体着色
    }
    void OnMouseEnter()
    {

        GetComponent<MeshRenderer>().material.color = mouseOverColor; //鼠标滑过时改变
                                                                      //物体颜色为蓝色
    }
    void OnMouseExit()
    {
        GetComponent<MeshRenderer>().material.color = originalColor;//鼠标滑出时恢复
                                                                    //物体本来的颜色
    }
    IEnumerator OnMouseDown()              //利用协同程序移动三维物体，鼠标单击时开始移动
    {
        Vector3 screenSpace = Camera.main.WorldToScreenPoint(transform.position);
                                                                  //三维物体坐标转为屏幕坐标
        //将鼠标屏幕坐标转为三维坐标，再计算物体位置与鼠标之间的距离
var offset = transform.position - Camera.main.ScreenToWorldPoint(new Vector3(Input.
mousePosition.x, Input.mousePosition.y, screenSpace.z));
        print("down");
        while (Input.GetMouseButton(0))//按下鼠标左键，一直让三维引擎的部分跟随鼠标移动
        {
            Vector3 curScreenSpace = new Vector3(Input.mousePosition.x, Input.mousePosition.y,
screenSpace.z);
            var curPosition = Camera.main.ScreenToWorldPoint(curScreenSpace) + offset;
                                                          //把鼠标的屏幕坐标转换成三维坐标
            transform.position = curPosition;//把当前转换后的鼠标的三维坐标赋值给当前的
                                             //游戏物体即是飞机引擎的部分组件
            yield return new WaitForFixedUpdate(); //等待 FixedUpdate()函数执行完毕
        }
    }
}
```

然后就可以用鼠标拖动某个飞机引擎的部分组件，如图 5.145 所示。

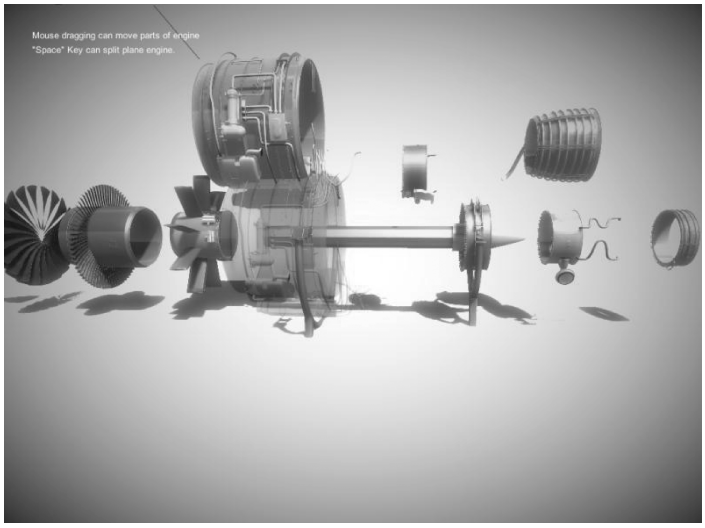


图 5.145　飞机引擎的拆装

经过上面的一系列操作，飞机引擎的拆装基本上完成。

5.10.2　VR 房地产项目讲解

VR 房地产要实现的功能是可以在室内漫游，可以从各个视角动态地观看房子的结构与构成。

首先需要把资源导入 Unity 中。我们需要在场景中添加一个第一人称角色控制器 CharacterController。在 Unity 中，CharacterController 组件是控制角色移动的。通过在一个胶囊体上添加一个 CharacterController 组件来控制胶囊体的移动，并且把摄像机作为胶囊体的子物体一起移动，相当于移动摄像机，如图 5.146 所示。

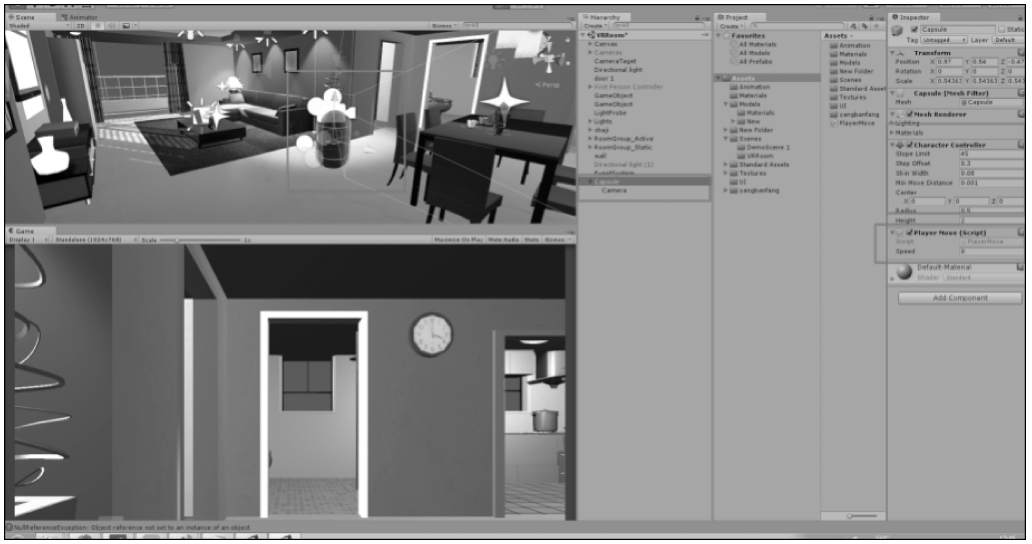


图 5.146　添加一个第一人称角色控制器

下面是 PlayerMove 脚本。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class PlayerMove : MonoBehaviour
{
    public float speed;
    CharacterController cc;
    void Start()
    {
        cc = GetComponent<CharacterController>();
    }
    float rotateY;
    float rotateX;
    void Update()
    {
        #region 移动的功能
        float h = Input.GetAxis("Horizontal");      //获取水平轴值
        float v = Input.GetAxis("Vertical");        //获取垂直轴值
        Vector3 direction = new Vector3(h, 0, v);   //要移动的方向
        direction = transform.TransformDirection(direction); //把相对坐标的位置转换成
                                                             //世界坐标的位置
        cc.Move(direction * Time.deltaTime * speed); //通过角色控制器来控制角色进行移动
                                                     //或者是在房间里面漫游
        #endregion
        #region 旋转的功能
        float x = Input.GetAxis("Mouse X");    //获取鼠标的 X 方向的轴值,即鼠标水平的轴值
        float y = Input.GetAxis("Mouse Y");    //获取鼠标的 Y 方向的轴值,即鼠标垂直的轴值
        rotateY = rotateY + x * Time.deltaTime * speed;  //角色在 Y 轴上的旋转增量
        rotateX = rotateX + y * Time.deltaTime * speed;  //角色在 X 轴上的旋转增量
        rotateX = Mathf.Clamp(rotateX, -20, 20);    //限制 X 轴方向的旋转量,范围为 -20°～20°
        transform.eulerAngles = new Vector3(rotateX, rotateY, 0);  //把旋转增量赋值给角色
                                                                   //的欧拉角
        #endregion
    }
}
```

习题

1. 用 Unity 3D 开发一个汽车组装演示系统。
2. 用 Unity 3D 开发一个校园漫游系统。

第6章　增强现实系统的标定

为了实现虚拟与真实场景更好的结合，计算机产生的虚拟添加信息在增强现实系统中需要通过三维跟踪注册算法和真实场景保持精确的对准关系，如图 6.1 所示。为了确保虚拟与真实场景的全方位对准，需要用到系统中以多种方式呈现的先验知识和信息。真实场景的基准点的三维空间坐标，摄像机内部和外部参数等，都是大部分增强现实系统需要的系统信息。

图 6.1　Unity 资源商店中的高通手机平台增强现实软件包

为了获取摄像机的内部和外部参数，增强现实系统需要在初始时刻进行标定。目前的增强现实系统可以分为光学透视式增强现实系统和视频透视式增强现实系统两种。虽然光学透视式增强现实系统与视频透视式增强现实系统的标定方式不相同，但是这两种系统在进行系统标定时，都涉及世界坐标系、摄像机坐标系、平面坐标系的坐标变换。下面将对各种坐标系以及如何建立坐标系之间的关系进行简单的介绍。

6.1　系统几何模型及坐标变换

在计算机视觉中，空间物体在像平面上的投影就是图像。利用所拍摄的图像可以计算出三维空间中被测物体的几何参数。摄像机通过成像透镜将三维空间投影到二维像平面，即成像模型。为了方便描述成像过程，定义四种坐标系，分别是世界坐标系、摄像机坐标系、图像坐标系和像素坐标系。

6.1.1　图像坐标系和像素坐标系

在数字图像中，经常定义两种坐标系：像素坐标系和图像坐标系。如图 6.2 所示，一般情况下，像素坐标系的原点 O_0 位于图像的左上角，U 轴和 V 轴平行于图像的行和列，坐标单位是像素(pixel)。在像素坐标系中，每一点的坐标(u,v)表示该像素的行数和列数。图