# 第2章 开始Unity游戏开发

Unity让游戏开发变得简单。使用Unity不需要你有多年的技术积累，也不需要你有任何艺术方面的技能。只需要学习和掌握一些基本的概念和工作流程，就可以使用Unity开发游戏了。当然，学习的过程少不了实践和练习，想要用好Unity也需要花费时间对引擎功能和相关技术有逐渐深入的理解。而且Unity作为一种游戏引擎，很多功能的使用还是离不开编写脚本，所以本章也会介绍脚本开发相关的内容。

学习Unity最大的好处是，不需要等完全学会了所有功能以后再尝试开发自己的游戏。你完全可以在掌握了最基本的概念和使用方法之后，就试着制作自己的游戏原型。刚开始你的原型可能会比较简陋，但随着学习的进展，你会发现游戏的功能和效果会越来越丰富，可以解决的问题也越来越多。通过这种方式，你会发现学习Unity是一件非常自然的事情，随着时间的推移和经验的积累，你的开发技术也会越来越成熟。

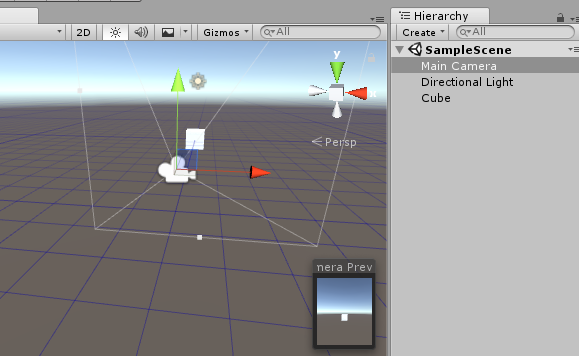
前一章里，我们学习了Unity的基础操作，对引擎界面有了大致的了解。本章开始我们正式学习Unity游戏开发，本章将从解释Unity的核心概念开始，逐步介绍游戏开发中必然用到的知识和基本操作，同时还会适时加入脚本基础的讲解。

本章作为整本书的核心，建议在学习时一边阅读一边练习。读者只要理解好本章，就可能提纲挈领、在较短时间内学会Unity大部分的使用方法，起到事半功倍的效果。

## 场景

### 场景的概念

场景包含了游戏环境、角色和UI元素。可以将每个场景看作一个独立的关卡。在每个场景中，都可以放置环境、障碍物和装饰（比如花、草、树木），在设计游戏时，总是可以将游戏划分为多个场景，分别实现。



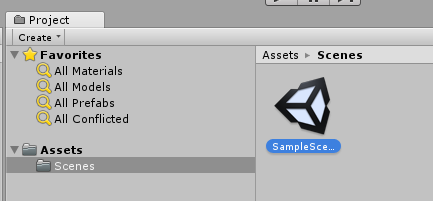
（一个新建的空白场景，默认带有一个摄像机Main Camera，和一个方向光源）

在创建新的Unity工程时，默认场景视图就打开了一个新的场景，这个场景没有名称也没有保存。虽然场景是空的，但是默认有一个摄像机“Main Camera”，和一个方向光源。

### 保存场景

要保存当前制作的场景，在主菜单中选择File > Save Scene，或者按下Ctrl+S快捷键即可（在MacOS下为Cmd+S键）。

Unity的场景会以.unity的后缀保存在Assets文件夹内，所以在工程窗口中你也可以看到保存的场景文件。



（已保存的场景文件）

### 打开场景

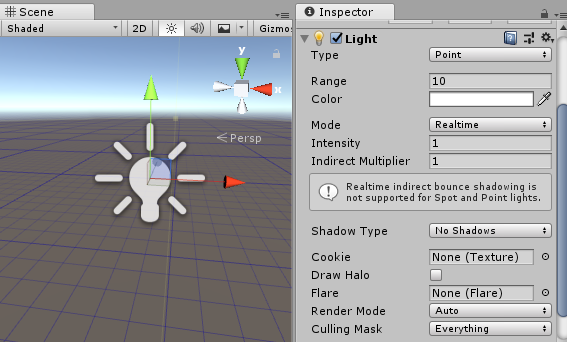
我们在Unity中做的操作总是属于某个场景的，所以需要先新建一个场景，或者打开一个已有的场景。要打开一个场景，双击工程窗口中的场景文件即可。如果当前场景还未保存，Unity会弹出提示窗口。

## 游戏物体（GameObject）

游戏物体（GameObject）是Unity中最基本的元素之一。

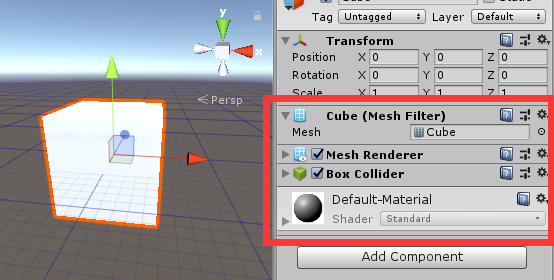
游戏物体本身并不直接实现具体的功能，它们的核心功能是作为组件（Component）的容器。也就是说，游戏物体包含一个或多个组件，由组件完成具体的功能。

例如，要制作一个光源，就创建一个带有灯光组件的物体：



（一个典型的光源，带有一个灯光（Light）组件）

再举个例子，一个方块实体，包含一个网格过滤器（Mesh Filter）组件和一个网格渲染器（Mesh Renderer）组件，这两个组件一起实现了方块的外形。另外方块还包含一个盒子碰撞体（Box Collider），以实现方块在物理层面上的体积和外形。



（一个基本的立方体，具有多种组件）

每个游戏物体总是有一个变换（Transform）组件，变换组件表示了物体的位置、旋转、缩放信息，所以Unity规定所有的游戏物体都必须包含变换组件且不可删除。变换组件与物体的父子关系有紧密的联系。

变换组件（Transform）必须有、且只能有一个。

其它组件通常都可以添加或删除，可以在编辑器中操作，也可以在游戏运行中通过脚本随时修改。有一些常用的基本游戏物体已经配置好了相关组件，比如方块、球体等。

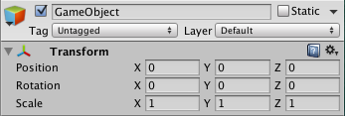
以上就是游戏物体最基本的概念。之后我们还会继续介绍游戏物体的一些细节问题，另外用脚本操作游戏物体也是重点内容。

## 组件（Component）

一个游戏物体可以包含多个组件。

之前说到过，变换组件（Transform）是最基本最重要的一种组件，它与游戏物体一一对应。

创建一个新的游戏物体，就可以在检视窗口中查看它的变换组件：



（检视窗口中，正在查看一个新建物体的变换组件）

从上图中可以看到，最简单的空白游戏物体（empty GameObject）具有名称——上图中为“GameObject”、标签（Tag）——上图中为Untagged，以及层级（Layer）——上图中为Default，以及变换组件。

### 变换组件

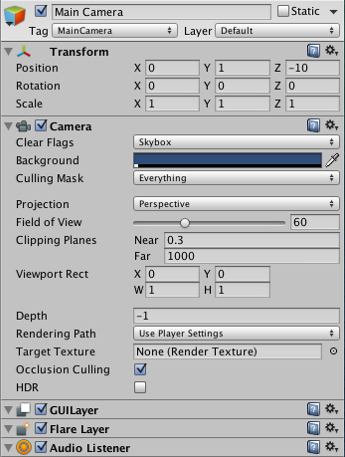
不可能创建一个不带有变换组件的物体，变换组件定义了物体在游戏世界中的位置、旋转角度以及缩放。

另外，变换组件实现了游戏物体的“父子关系”功能，父子关系也是Unity使用的重点，在之后会详细介绍。

### 其它组件

游戏物体还可以包含其它组件，每种组件具有不同的功能，会在使用中慢慢学习。且某些类型的组件和变换组件一样，每个物体只能有一个，另一些组件可以同时包含多个。

比如选中默认的游戏物体“Main Camera”，可以在检视窗口中看到以下详情：



（新建的场景默认有一个游戏物体——Main Camera，它已经预先添加了多个组件）

可以看到，主摄像机（Main Camera）已经包含了摄像机组件（Camera）、用户界面层组件（GUI Layer）、耀斑层组件（Flare Layer）以及音频侦听器组件（Audio Listener）。以上每种组件都实现一种特定的功能，共同实现了一个完整的主摄像机。

还有一些其它常用组件，比如刚体组件（Rigidbody），碰撞体组件（Collider），粒子系统组件（Particle System）以及音频组件（Audio），可以试着将它们添加到游戏物体上。

## 使用组件

组件是实现具体游戏功能的基石，多个组件共同发挥作用，组成了具有特定功能的游戏物体。将本节内容与之前关于游戏物体的介绍联系在一起，就可以更好地理解游戏物体与组件的关系。

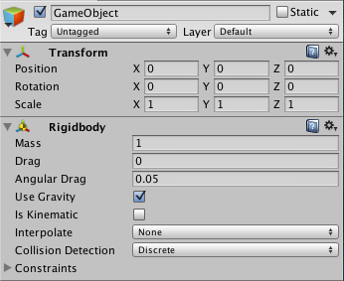
从功能的角度上看，游戏物体只是一个容器（类比现实中的容器比如锅、碗、盆子）。容器本身在功能性上讲是空白的，只有当容器内装载了具有具体功能的实体时，才具有了具体的功能，这种功能性的实体就是组件。所有游戏物体默认至少具有变换组件，因为物体至少要具有位置和方向信息，

在讲解每一个概念时，读者都可以随时查看检视窗口，为物体添加、删除或修改组件。

### 添加组件

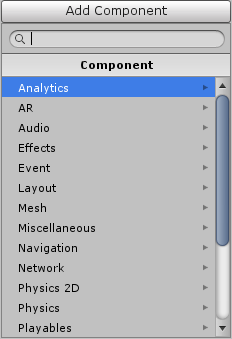
选中一个游戏物体比如一个方块或者球体，在检视窗口中点击“Add Component”按钮，之后选择Physics类 > Rigidbody，就为这个物体添加了刚体组件。这时你播放游戏，就会发现在游戏运行时，这个物体的Y坐标一直在减小。这是因为刚体是一种物理组件，默认会受到重力影响，所以物体是由于重力影响而下落的。

另外，从这个例子中也可以体会到在游戏停止时或运行时，选中和查看物体的方法是类似的，但是游戏运行时参数会有动态的变化。



（一个添加了刚体组件的游戏物体）

刚才在添加组件时，我们还看到了一个选择组件的菜单，可以称之为“组件浏览器”（Component Browser）。



（组件浏览器）

利用组件浏览器，我们可以分组查看到所有组件的名称，另外为了加快操作效率，组件浏览器还具有一个内置的搜索框，在你知道组件的大概名称时，就可以直接搜索组件名称的任何一部分快速查找到该组件。

理论上一个游戏物体可以挂载任意数量的组件，特别是某些组件往往需要和别的组件配合才能发挥作用，比如刚体组件想要模拟一个现实时间中的物体，就往往需要配合碰撞体组件，因为现实中的物体都具有一个物理外形。刚体组件内部利用了NVIDIA PhysX物理引擎来计算并更新物体的变换组件的信息（也就是位置、旋转和缩放），碰撞体组件赋予了这个物体与别的物体发生碰撞的能力。

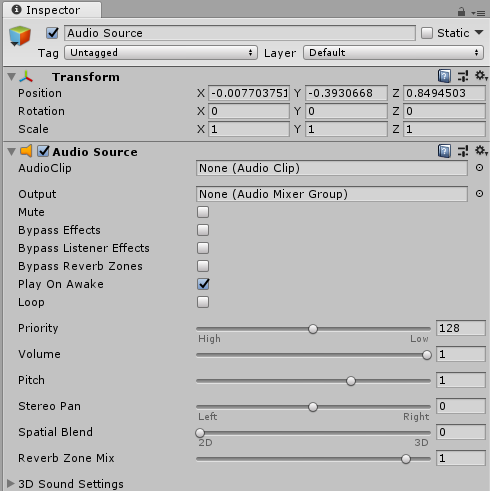
本书的其他章节中往往会针对某些刚体做完全展开的、详细的介绍，也就是说Unity许多复杂功能的实现，实际上背后是某几个特定的组件来支持的。

### 编辑组件

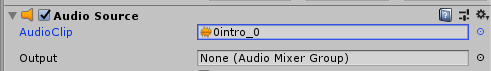
组件的编辑是十分便捷，这是它的优点之一。组件的主要参数都会很明确地显示到检视窗口中，可以随时编辑（甚至可以在游戏运行时编辑，虽然这样做不会保存），另外在游戏运行时，由脚本来动态修改组件的参数，是脚本发挥作用的重要途径。比方说，在游戏运行时，脚本一直慢慢增加变换组件的X轴位置的值，你就发现物体在慢慢地移动。

有两种基本的属性类型：值属性和引用属性。

看下面的音源（Audio Source）组件的截图，这是声音组件默认的设置：



这个组件包含两个引用属性——音频剪辑（AudioClip）和输出（Output），其他能够看到的属性都是值属性。音频剪辑可以引用一个文件，当声音开始播放时，这个组件会尝试播放音频剪辑属性中所指定的文件。如果没有指定或在运行时没有正确找到该剪辑，那么就会发生错误。想要正确引用到某个音频文件，只需要用拖拽的方式将音频文件拖到音频剪辑的编辑框中即可；另一种方法是点击右边的圆形小按钮，用“对象选择器”来指定具体文件。



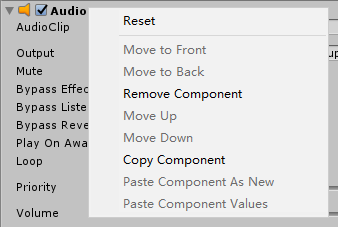
（指定了引用的声音文件为0intro\_0）

引用属性所引用的参数类型，根据属性本身的类型有所不同，可能性很多，比如引用组件、游戏物体、某种素材文件。

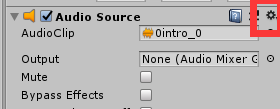
该组件其它的属性都是值类型的属性。值类型的属性很简单，就是说该属性由具体的数值（一个或多个具体的值）指定，这个数值直接在界面上调节或选择即可。值类型可以是勾选的（代表是或否，即布尔类型）、数值型的、下拉列表的（从多个选项中选择一个），也可以由输入文本、颜色、曲线或其他各种方式来指定。

### 组件选项菜单

右键点击组件名称处，就打开了组件常用菜单，里面有几项常用的命令。



在组件右上角的齿轮图标处，也可以打开同样的菜单。



**复位（Reset）**

将组件参数设置还原为默认的，也就是新添加组件时的默认值。

**删除组件（Remove Component）**

从游戏物体上删除这个组件。值得一提的是，某些组件之间具有依赖关系，比如物理关节组件（Joint）要求必须有刚体组件才能工作，当删除了被依赖的组件时，会弹出警告。

**上移/下移（Move Up/down）**

将组件向上或向下移动。这个功能不仅仅影响界面上的外观，也会真的影响组件挂载的顺序，这个顺序在某些情况下对运行结果有影响。

**复制/粘贴组件（Copy/Paste Component）**

复制组件会将组件的当前参数暂存下来。之后有两种粘贴的方式：一是粘贴为一个新的组件，即新建并粘贴参数（Paste Component As New）；二是将这些参数粘贴到另一个同类型的组件上，而不新建组件（Paste Component Values）。

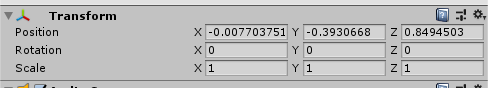
### 测试组件参数

当游戏进入运行模式后，依然可以随意修改组件的参数。例如，开发游戏时需要反复测试并调整跳跃的高度。如果在某个脚本组件中有一个公开的跳跃参数，那么就可以直接运行游戏，一边测试跳跃，一边修改参数的值。在游戏运行中，你可以反复修改这个参数以最终确定合适的值。注意在得到了合适的值以后，一旦退出播放模式，所有的参数就会重置到播放之前的状态，这个设计的好处是如果你在播放模式下改乱了多个参数，停止运行就会回到最初的状态；当然反过来说，这要求我们在找到了合适的参数以后，务必记录下参数或者拷贝参数的值，在停止运行以后，再修改记录的值。

在运行模式下实时修改参数并测试是一个非常强大的功能，不仅多年以前的游戏引擎不具备这种特性，甚至至今某些游戏引擎都没有实现这一功能。这个功能给游戏开发带来了巨大便利，务必要实际体验过才能理解。

## 最基本的组件——变换组件

变换组件（Transform）决定了物体的位置、朝向和放缩比例。每个游戏物体必定有且只有一个变换组件。



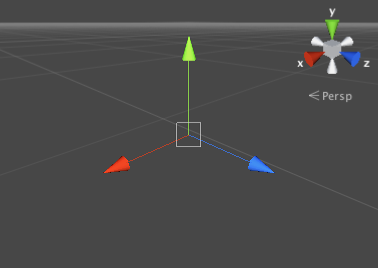
### 属性列表

| 属性 | 功能 |
| --- | --- |
| Position | 以X、Y、Z方式表示的物体坐标 |
| Rotation | 旋转（朝向），是以绕X、Y、Z轴旋转的角度表示的，这种旋转的表示方式也称之为欧拉角。 |
| Scale | 沿X、Y、Z轴的缩放比例，1表示原始比例，0.1表示缩小为原来的10%，10表示放大10倍。这个值甚至允许是负数，代表沿着该轴翻转。 |

以上的三个属性位置、旋转和缩放，都指的是相对父物体的位置，而不一定是世界坐标系中的位置。当一个物体没有父物体时，它的局部坐标系与世界坐标系是一致的。

### 编辑变换组件

在3D空间中，变换组件具有X、Y、Z三个轴的参数，而2D空间中就只有X轴和Y轴。Unity中约定X、Y、Z轴分别以红色、绿色和蓝色表示，无论表示旋转或位置都尽可能用同样的颜色来展示，在熟悉之后就会觉得很方便。



（移动物体时的图示，三个轴的颜色为红、绿、蓝）

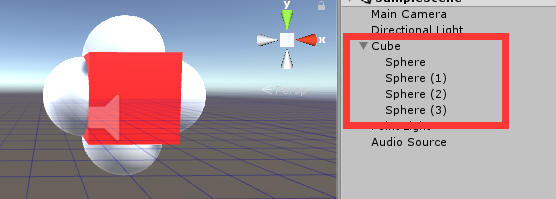
修改物体位置、旋转、缩放的操作，实际上就是修改变换组件的参数。具体的操作方法可以再回去参考第一章，有过详细的讲解。

### 父子关系

父子关系（Parenting）是Unity中最重要的基本概念之一。当一个物体是另一个物体的父物体时，子物体会严格地随着父物体一起移动、旋转、缩放。可以将父子关系理解为你的手臂与身体的关系，当身体移动时，手臂也一定会跟着一起移动。且手臂还可以有自己下一级的子物体，比如手掌就是手臂的子物体、手指是手掌的子物体等等。任何物体都可以有多个子物体，但是每个物体都只能有一个父物体。这种父子关系组成了一个树状的层级结构。最基层的那个物体是唯一不具有父物体的物体，它被称之为根节点（root）。

由于物体的移动、旋转、缩放与父子关系密切相关，所以在Unity中，游戏物体的层级结构完全可以理解为变换组件的层级结构。由于游戏物体和变换组件是一一对应的，所以这两种理解方式是等价的，在后面学习到脚本编写时，你会发现父子关系的操作在脚本中确实是在变换组件上进行的。

第一章中介绍过，可以在层级窗口中将一个物体拖拽到另一个物体上创建父子关系。



（父子关系的例子）

子物体的变换组件的参数，其实是相对父物体的值，这被称之为局部坐标系或是本地坐标系（Local coordinate）。再次考虑之前身体和手臂的例子，无论身体如何移动，手臂和身体的连接处是固定不变的。

在处理不同的问题时，有时使用局部坐标系更方便，而有时使用世界坐标系更方便。

例如在搭建场景时，我们更喜欢使用局部坐标系，比如移动一个房屋时，屋子里所有的东西都会跟着一起移动；而在编写游戏逻辑时，更多时候需要获得物体在空间中的实际位置，比如我们要将摄像机对准人物的眼睛，这时候眼睛和人物的相对坐标就没有太大价值，而应当要让相机对准眼睛在世界坐标系中的位置。所以在脚本系统中，变换组件的大部分操作都提供了两类操作方式，分别是世界坐标系的和局部坐标系的，可以根据需求使用。

### 非等比缩放的问题

非等比缩放即变换组件的X、Y、Z缩放值不相等，比如分别是2、4、1的情况。非等比缩放在某些情景下也是有用的，但是它会带来一些奇怪的问题，这些问题在等比缩放时就不会遇到，所以这里要特别说明一下。

某些组件不完全支持非等比例缩放，也就是说在非等比缩放的情况下可能会出现意想不到的结果。例如，碰撞体、角色控制器这些组件，具有一个球体或者胶囊体的外壳，这些外壳的大小是通过一个半径参数指定的，还有灯光、音源也有类似的情况。在物体或者父物体被拉伸或压扁的时候，这些组件的球体范围并不会跟着压扁成椭球体，它们实际上仍然是球体或胶囊体。所以当物体中具有这类组件时，由于组件形状和物体形状的不一致，可能导致穿透模型或者被意外阻挡等等情况。这些问题不算致命，但是会引起奇怪的BUG。

之前说过，子物体的旋转、缩放、位移会严格跟着父物体变化，但在父物体进行了非等比缩放时，这会带来一个很麻烦的问题。当父物体沿X轴和Z轴的缩放不一样时，子物体也会同样被拉伸，这时如果旋转父物体，那么子物体的缩放与旋转参数就会非常难以计算。由于性能原因，这种情况下Unity引擎不会立即更新子物体的实际缩放情况，所以这时可能会导致子物体信息没有及时改变，而又在稍后发生突然变化。这种情况在编写脚本时会更容易察觉到，就好像是子物体脱离了父物体一样。

### 关于缩放和物体大小的问题

变换组件的缩放参数，决定了Unity场景中的模型大小与原始模型大小相对的倍数。Unity中物体的大小非常重要，特别是在物理系统中是至关重要的。默认情况下，物理引擎规定所有场景中的单位都对应国际标准单位，比如空间中的1个单位就代表1米。如果一个物体的模型非常巨大，那么它的运动就显得非常缓慢。这在物理上是非常严谨的，举个例子，一个半米高的大楼模型，模型顶部的零件掉落到模型底部，只需要一瞬间；而一个真正的摩天大楼，楼顶上的一个石块掉落到底部可能需要数秒的时间。将这个原理放在Unity中，就可以明白为什么如果比例特别大的物体，在物理运动中会显得非常缓慢。

有三个因素会影响物体的实际大小：

1. 在三维软件中制作的模型的大小（一般三维软件导出模型文件时可以设置比例）。
2. 在导入模型时，可以设置模型的大小。详见讨论导入资源的章节。
3. 变换组件的缩放参数。

理论上来说，为了减少潜在的问题，不应当通过调节变换组件的缩放参数来调整模型的大小。最佳方案是在制作模型、导出模型时，就选择符合真实比例的大小，这样在后续工作中就更方便、更不容易带来麻烦。次一等的方案是在导入模型时，可以在导入设置中改变模型的比例。在改变模型比例时，引擎会根据模型大小做特定的优化，且在创建一个改变了比例的模型到场景中时，由于引擎会自动调整模型，所以会带来一点性能上的影响。

### 变换组件的其它注意事项

* 当为一个物体添加子物体时，可以考虑先将父物体的位置设置为原点，这样子物体的局部坐标系就和世界坐标系重合，方便我们去指定子物体的准确位置。
* 粒子系统不会受变换组件的缩放系数的影响。要该表一个粒子的整体比例，还是需要在粒子系统中适当改变相关参数。
* 在物理系统的刚体组件相关章节中，会提到刚体组件与缩放系数的问题。该问题在Unity官方文档中有更详细的描述，可以在Unity官网文档中，刚体组件（Rigidbody component）相关页面中查找。
* 本书中有一些描述编辑器中的颜色的地方，比如坐标轴的颜色默认为红、绿、蓝。这些颜色均可以在选项中修改。一般不推荐修改默认颜色，因为默认颜色是统一的，比较方便开发者之间互相交流。但是如果你的场景颜色比较特别，导致默认颜色看不清楚，也可以考虑修改默认的颜色。
* 修改物体缩放比例时不仅会直接影响子物体的比例，还会影响子物体的实际位置（因为要保证相对位置不变）。

## 脚本与组件操作

本节中，将介绍创建并编写一个脚本，用脚本调用Unity提供的功能，并将脚本作为组件挂载到游戏物体上发挥作用。

当我们将创建的脚本文件挂载到游戏物体上面时，脚本组件就会出现在检视窗口中，和内置组件完全一样。Unity就是以这种组件化的方式来扩展多种功能的。

从技术角度来说，每一个新的脚本都定义了一种新的组件类型，而不是说所有的脚本都是同一种类型的组件。所以每添加一个新的脚本都像是为Unity定制了一种新的组件。脚本中的公共成员也会显示在编辑器中，就像其它内置组件一样可以方便修改。

### 创建和使用脚本

游戏物体的实际功能是由组件实现的。虽然Unity的内置组件已经能完成各种各样的功能，但是在开发游戏时，你很快就会发现还是需要自己定制和游戏相关的具体功能。Unity扩展新的功能性组件的主要方法就是依靠脚本。脚本可以用来处理游戏事件、修改组件属性或是接受输入等等，可以实现所有必要的功能。

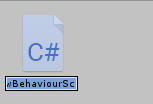
在Unity 2017以后的版本中，官方推荐使用的脚本语言简化为C#（C Sharp）这一种。Unity的历史版本中支持过UnityScript，由于用户较少，不再推荐使用。

但是，除C#以外，.Net平台支持的语言（例如F#、C++ CLI、VB.net等等）都可以编译为通用的DLL库，关于.Net的详细讨论超出了本书的范围，不做详述。

接下来逐步讲解脚本相关的知识。

脚本文件通常在Unity编辑器中直接创建即可。通常可以在工程窗口的某个目录中操作，在右键菜单中选择Craete > C# Script即可。

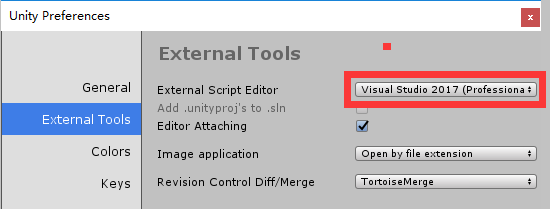
之后要为新建的文件指定名称，这个初始名称非常重要但又常常被初学者忽略而带来问题。



因为Unity规定脚本中的类名称必须与文件名完全一致，例如如果脚本名称为“Wall.cs”，那么在脚本内容中，就必须写着“class Wall”，包括大小写也要完全一致才可以，否则Unity就不允许将这个脚本挂载到游戏物体上。也就是说如果修改了脚本文件名，那么class名称也要跟着改变。

### 初识脚本

双击脚本文件，会弹出MonoDevelop编辑器或者Visual Studio开发环境（在第一章安装Unity时推荐勾选安装VS2017社区版），建议使用VS开发环境。可以随时在Edit > Preferences中修改脚本编辑器。



（在可以随时修改首选项改变默认的脚本编辑器）

新建的脚本内容如下：

|  |
| --- |
| using UnityEngine;  using System.Collections;  public class MainPlayer : MonoBehaviour {  // Use this for initialization  void Start () {  }    // Update is called once per frame  void Update () {  }  } |

自定义的脚本组件会和Unity引擎进行密切的交互，规定脚本中必须包含一个类，且该类继承了Unity内置的MonoBehaviour类。只有符合这个规范的脚本才会被看作一个组件，且可以被挂载到游戏物体上。当你挂载脚本到物体上时，实际上是创建了该脚本类的实例。另外前面说过，脚本文件名称也必须和类名称一致，例如对上面的脚本来说，文件名也必须是MainPlayer.cs，才能成为合法的组件。

接下来是两个重点：脚本中定义的两个函数。二者分别叫做Update函数和Start函数。

Update函数会在每一帧被调用，所以Update函数可以用来实现用户输入、角色移动和角色行动等功能，基本上大部分游戏逻辑都离不开Update函数。Unity提供了非常多的基本方法来读取输入、查找某个游戏物体、查找组件、修改组件信息等等，这些方法都可以在Update函数中使用，用来完成实际的游戏功能。

Start函数在物体刚刚被创建时调用，也就是说，在第一个Update之前，只调用一次。Start函数适合用来做一些初始化工作。

对有经验的程序编写者来说，要注意脚本组件通常不使用构造函数来做初始化，因为构造函数可控制性较差，会导致调用时机和预想的不一致。所以最好的方式是遵循Unity的设计惯例。

### 用脚本控制游戏物体

要让脚本被调用执行，就一定要将它挂载到物体上。要挂载脚本到某个物体上，只需要选中某个游戏物体，然后将脚本文件拖拽到检视窗口下方空白区域即可。成功的话可以看到新添了一个名称和类名一致的组件，和其它组件非常相似。

IMG_257

挂载好之后，点击运行游戏的按钮，游戏执行，脚本函数也会被调用。要检查脚本是否运行，可以修改Start函数如下：

|  |
| --- |
| void Start () {  Debug.Log("我是主角。");  } |

Debug.Log是我们学的第一个简单命令，它的效果是打印一段信息到Unity的控制台窗口（Console）。如果在修改脚本后运行游戏，就可以在控制台窗口看到“我是主角。”的信息了。如果找不到控制台窗口，可以通过主菜单的Window > Console选项打开控制台菜单。

### 变量与检视窗口

上面说到，自定义脚本就是定义了一种新的组件。内置的组件往往有很多参数可以调整，其实脚本也可以在编辑器中修改参数。关键是使用公共变量（public variables），如下面的脚本所示：

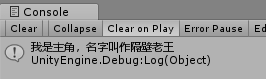
|  |
| --- |
| using UnityEngine;  using System.Collections;  public class MainPlayer : MonoBehaviour {  public string myName;    // Use this for initialization  void Start () {  Debug.Log("我是主角，名字叫作" + myName);  }    // Update is called once per frame  void Update () {  }  } |

以上脚本定义了一个公共变量“myName”，于是在检视窗口的脚本组件上，我们可以看到多了一个可编辑参数“My Name”。



可以发现，从变量名称到参数名称，有一种简单的转换规则。变量以小写字母开头，下一个单词的开头字母要大写，在转换为参数时，Unity会在两个单词之间加入空格，这样在编辑器中查看这个变量就非常清晰和标准。给变量命名时务必要遵守这种规则，才能编写出好用易懂的脚本。

加入了myName变量以后，在编辑器中将“My Name”修改为不同的值，就可以在游戏运行之后看到不同的打印信息了。



Unity甚至可以让你在游戏运行过程中修改变量的值。这种方法对于调试参数、查看参数对游戏的影响非常有用，而不需要反复停止和启动游戏。但是和其它参数一样，游戏停止运行时一切参数都会回到开始运行之前的值。所以你可以放心大胆地试验参数，但是要注意得到合适的参数后要及时复制或记录下来。

### 通过组件控制游戏物体

不仅在检视窗口中可以随时修改组件的参数，在脚本中修改组件的参数更为普遍。因为脚本可以自动化运行，所以可以实现持续地修改参数的值，比方说可以每帧都改变一点变换组件的坐标位置，这样就能让物体“连续”移动起来。

绝大部分组件参数都可以在脚本或编辑器中修改，例如刚体的质量、渲染器的材质，包括组件的开启或关闭都可以控制。通过巧妙设计逻辑条件，我们可以通过这些基本组件操作实现任意复杂的游戏机制。

#### 访问组件

最简单也最常用的一个情景，是在脚本中访问同一个游戏物体上的某个组件。上面说过，游戏物体上挂载的组件，就是一个组件的实例（component instance，或者可以叫做component object组件对象）。所以我们要做的就是获取另一个组件的实例。内置的GetComponent函数专门用来实现这个功能。通常我们把获得的组件保存到一个变量中，像这样：

|  |
| --- |
| void Start () {  // Rigidbody，刚体组件  Rigidbody rb = GetComponent<Rigidbody>();  } |

这样rb就是该物体上的刚体组件的引用变量，只要是刚体组件支持的操作，都可以在脚本中任意使用：

|  |
| --- |
| void Start () {  Rigidbody rb = GetComponent<Rigidbody>();    // 改变刚体的质量为10千克  rb.mass = 10f;  } |

另一个可用的操作是给刚体施加一个力，只需要调用rigidbody的AddForce方法即可。

|  |
| --- |
| void Start () {  Rigidbody rb = GetComponent<Rigidbody>();    // 施加一个向上的力，大小为10牛顿  rb.AddForce(Vector3.up \* 10f);  } |

对于一个物体上能挂多少个脚本，并没有任何限制，只要有必要可以挂上多个脚本分别完成一项功能。注意：脚本组件也是组件，所以也可以用GetComponent获得。要获得脚本组件，组件名就是脚本的类名，例如对于脚本mainPlayer.cs，类名为MainPlayer，那么通过GetComponent<MainPlayer>() 即可获得这个组件。

尝试获取一个不存在的组件，函数会返回null（空引用），如果去操作null就会引发运行时异常。

### 访问其它游戏物体

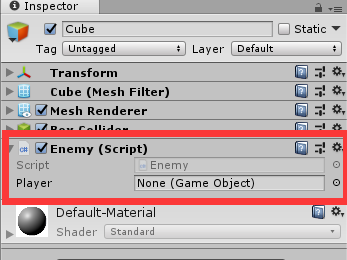
实际游戏开发中，一个脚本不仅会对当前挂载的物体进行操作，还可能会引用其它物体。例如，正在追逐玩家的敌人角色，会一直保留着玩家的引用，以便随时确定玩家的位置。访问其它游戏物体的方法多样，使用非常灵活，可以根据不同的情况采用不同的方式。（只要涉及到编程，解决问题的方法就总是多样的）。

#### 用变量引用游戏物体

Unity中获得其它物体最简单直接的方式，就是为脚本添加一个public GameObject变量，不需要设置初始值：

|  |
| --- |
| public class Enemy : MonoBehaviour {  public GameObject player;    // ……  } |

player变量会显示在检视窗口中，默认值为空：



现在就可以将任何物体或预制体，拖拽到Player变量的文本框中，就为Player赋予了初始值。之后在脚本中就可以随意使用player这个游戏物体了，如下所示：

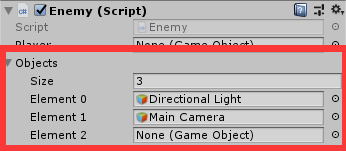
|  |
| --- |
| public class Enemy : MonoBehaviour {  public GameObject player;    void Start() {  // 读取player的位置，并设置本物体的位置在它后方  transform.position = player.transform.position - Vector3.forward \* 10f;  }  } |

另外，上面说的引用其它物体，变量类型不仅可以是GameObject，甚至可以是一个组件类型，依然可以将游戏物体拖到这个变量上，只要被拖拽的物体确实具有这个组件就可以：

|  |
| --- |
| public Transform playerTransform; |

Unity的组件机制用面向对象的方式理解会有一些费解，这里解释一下。简单来说，可以用任何一个组件来指代游戏物体本身。这是因为组件实体具有“被游戏物体挂载”这样的性质，所以通过一个游戏物体可以获得它上面的任何一个组件，通过任何一个组件也可以获得挂载该组件的游戏物体。这个对应关系是明确的，因此上面的变量类型可以是组件类型，而又可以将游戏物体直接拖拽上去。

用变量将物体联系起来的做法非常有用，特别是这种联系是持续存在、不易变化的情况下。还可以用一个数组或者列表来保存多个游戏物体，别忘了在编辑器中为每个游戏物体都给定初始值。



（Unity支持直接查看和修改List类型）

如果被引用的物体是游戏运行时才动态添加的，或者被引用的物体会随着游戏进行而变化，事先拖拽的方式就不可行了，需要动态指定物体。下面将详细说明。

#### 查找子物体

有时需要管理一系列同类型的游戏物体，例如一批敌人、一批寻路点、多个障碍物等等。如果这时候需要对这些物体进行统一的管理或操作，就需要在脚本中用数组或者容器来管理它们（比如说用来指引角色移动的一批路点，就需要统一管理，按顺序指引角色行动）。使用前文所述的方法，可以一个一个地将每个物体拖动到检视窗口中，但是着怎样做不仅低效，而且容易误操作，且在物体增加、减少时还需要再次手动操作。所以，这种情况下手动指定物体引用是不合适的，可以用查找子物体的方法来遍历所有子物体。具体实现时，要用父物体的变换（Transform）组件来查找子物体（物体的父子关系访问的属性，都在变换组件中，而不在GameObject对象中）。以下是遍历所有子物体的例子：

|  |
| --- |
| using UnityEngine;  public class WaypointManager : MonoBehaviour {  public Transform[] waypoints;    void Start() {  waypoints = new Transform[transform.childCount];  int i = 0;  // 用foreach循环访问到所有子物体  foreach (Transform t in transform) {  waypoints[i++] = t;  }  }  } |

同样可以指定查找到某一个子物体，使用transform.Find方法：

|  |
| --- |
| transform.Find("Gun"); |

在实践中这种管理子物体的方式非常有用。由于Find函数的效率不好估计，可能会遍历所有物体才能查找到指定物体，所以如果可以在Start函数中使用，就不要在Update函数中使用，毕竟Start函数只会被执行一次，而Update函数每帧都会执行。

#### 通过标签或名称查找物体

如果要查找的物体并非当前物体的子物体，那么也有办法在场景中查找某个特定的物体。查找物体当然需要特定的特征信息，例如名字、标签等等。通过GameObject.Find方法可以通过名称查找物体：

|  |
| --- |
| GameObject player;  void Start() {  player = GameObject.Find("MainHeroCharacter");  } |

如果要用标签查找物体，那么就要用到GameObject.FindWithTag或GameObject.FindGameObjectWithTag方法：

|  |
| --- |
| GameObject player;  GameObject[] enemies;  void Start() {  player = GameObject.FindWithTag("Player");  enemies = GameObject.FindGameObjectsWithTag("Enemy");  } |

### 常用的事件函数

Unity中的脚本组织，不像是传统的游戏循环，有一个持续进行的主循环并在循环体中处理游戏逻辑。相对的，Unity会在特定的事件发生时，调用脚本中特定的函数，然后执行逻辑的任务就交给了该脚本函数，函数执行完毕后，执行的权力就重新还给Unity。这些特定的函数通常被成为事件函数，因为是在特定事件发生时由引擎层调用的。例如我们已经看到过的Update函数就是最常用的事件函数之一，它在每帧一开始、渲染之前被调用；还有Start函数，在该物体出现的第一帧之前被调用。另外还有更多的事件，每个事件都有特定的函数名称和参数。接下来介绍一些比较常用、比较重要的事件。

#### 基本更新事件

游戏的进行特别像动画片，一帧一帧地进行，只不过对游戏来说未来的帧还没有画出来，需要在游戏进行的同时计算和准备。游戏中的一个基本概念是：在每一帧刚开始的时候（在渲染之前），对物体的位置、状态或行为进行计算，然后渲染（显示）出来。Update函数就是最常用的用来完成这个功能的事件函数。Update函数在每一帧刚刚开始时被调用：

|  |
| --- |
| void Update() {  float distance = speed \* Time.deltaTime \* Input.GetAxis("Horizontal");  transform.Translate(Vector3.right \* distance);  } |

物理引擎也会按照“物理帧”更新，机制和Update函数类似，但是更新的时机完全不同。物理更新的事件函数叫做FixedUpdate，它在每一次物理更新时被调用。要认识到，物理更新的频率和时机和Update是相对独立的。尽可能在FixedUpdate中进行物理相关的操作，在Update中进行其它操作，只有选择正确的函数才能让游戏效果尽可能准确。

|  |
| --- |
| void FixedUpdate() {  Vector3 force = transform.forward \* driveForce \* Input.GetAxis("Vertical");  rigidbody.AddForce(force);  } |

有时，不仅需要在每帧之前操作物体，还可能需要在所有的Update函数执行完毕之后再进行一些操作。我们有时需要获得在物体这一帧执行过后，最新的位置。例如，摄像机需要追随物体的位置，那么就需要在物体移动之后再更新摄像机的位置；还有当物体同时受脚本和动画影响时，我们需要在动画执行完毕后，再获得物体的位置。这时就要用到LateUpdate函数了：

|  |
| --- |
| void LateUpdate() {  // 在一帧的最后阶段，摄像机转向玩家位置。这样摄像机的旋转会更流畅  Camera.main.transform.LookAt(target.transform);  } |

#### 初始化事件

在物体第一帧执行之前，我们往往需要做一些初始化工作。Start函数会在第一次Update和FixedUpdate之前，物体被加载（或创建）出来时被调用。

Awake的调用时机会比Start更早，在场景加载时就会被调用。

注意，在Start函数执行之前，所有物体的Awake函数都已经执行完毕了，所以在Start函数中可以进一步访问物体在Awake时被修改的属性，二者是按顺序执行的。

### 时间和帧率

Update函数可以用来侦测输入或者检查其它事件，并做出相应的行为。例如，如果用户按住上键则角色向前移动。在编写时间相关的操作比如移动时，本节将介绍很重要的一个问题：帧率与速度控制。要知道，游戏的帧率（也是Update执行的频率）并不是一个固定的值，两次Update函数执行的间隔时间也不是一个固定的值。如果按照帧数来考虑物体的移动，一开始可能会写出如下代码：

|  |
| --- |
| //C# script example  using UnityEngine;  using System.Collections;  public class ExampleScript : MonoBehaviour {  public float distancePerFrame;    void Update() {  transform.Translate(0, 0, distancePerFrame);  }  } |

但是，由于每一帧所经历的时间并不是一个定值，所以物体移动的速度将不会是稳定的。为什么帧率会不固定呢？主要是因为硬件负载的原因，默认引擎会按照每秒60帧运行游戏，但是当负载增大时，帧率可能会有下降无法达到60帧。这时可能就只有30帧，帧率降低了一倍，每帧的时间也增加了一倍。

如果每帧移动0.01米，帧率为60帧，那么每秒移动0.6米；如果帧率降低到30帧，每秒就只能移动0.3米，物体的运动由于帧率降低而变慢了。实践中一般不允许这种情况的发生，解决方案是将两帧之间的间隔Time.deltaTime考虑进去：

|  |
| --- |
| using UnityEngine;  public class ExampleScript : MonoBehaviour {  public float distancePerSecond;    void Update() {  transform.Translate(0, 0, distancePerSecond \* Time.deltaTime);  }  } |

注意，通过乘以Time.deltaTime的运算，物体的移动不再以“每帧距离”为准，而变成了“每秒距离”。物体移动的距离将根据每帧时间长短而变化，从而在时间上看起来，移动是匀速的。

#### 物理更新间隔

与主更新函数Update不同，Unity的物理系统必须以固定的时间间隔工作，因为只有固定的时间间隔才能保证物理模拟的准确性。就算当前负载很高、Update帧率很低，Unity也会尽可能保证物理刷新的频率，因为如果物理刷新帧率无法保证，就可能出现不可预料的计算结果。

在主菜单，工程选项的TimeManager中可以修改物理更新的时间间隔。在脚本中使用Time.fixedDeltaTime可以获得物理更新间隔。较小的物理更新间隔，会带来更高的更新频率、更准确细腻的运算结果，但是也会极大增加硬件负担。fixedDeltaTime默认值为0.02，当确实对物理运算的准确性非常在意时，可以考虑适当减少这个值。

### 创建和销毁物体

除了个别游戏不会在运行中创建和销毁物体，大部分游戏都需要在游戏运行中实时生成角色、宝物、子弹等等物体，或者是删除它们。Unity提供了Instantiate函数专门用来创建一个新的物体，但是要提供一个预制体或者已经存在的游戏物体作为模版：

|  |
| --- |
| public GameObject enemy;  void Start() {  // 以enemy为模版，生成5个新的敌人  for (int i = 0; i < 5; i++) {  Instantiate(enemy);  }  } |

可以用已经存在的物体作为模版，更常见的方式是使用预制体作为模版。比如开开枪时会用一颗子弹的预制体来创建更多的子弹。创建的物体将会具有原物体一样的组件和参数。

另外可以用Destroy函数来销毁游戏物体或者组件，例如下面的代码会在导弹产生碰撞时，销毁该导弹，第二个参数0.5f表示在0.5秒之后才执行销毁动作：

|  |
| --- |
| void OnCollisionEnter(Collision otherObj) {  if (otherObj.gameObject.tag == "Missile") {  Destroy(gameObject, 0.5f);  }  } |

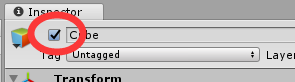
注意由于销毁游戏物体和销毁脚本都是使用Destroy函数，经常会出现误删除组件的情况，例如：

|  |
| --- |
| Destroy(this); |

由于this指代的是当前这个脚本实例，所以Destroy(this)会从物体上删除本脚本组件，而不是销毁物体。

### 使游戏物体或组件无效化

游戏物体可以被标记为不激活状态，这样就相当于临时从场景中删除了。可以使用脚本让物体无效化，或者取消勾选检视窗口最上方的激活勾选框，让物体无效化。



（游戏物体的激活勾选框，默认是激活状态）

与游戏物体同样的，每个组件也有一个是否激活的开关。

### 父物体无效化

当一个父物体被标记为不激活状态，那么它也会覆盖所有子物体的激活状态，这样以这个父物体为基础的整个分支都会变成不激活状态。注意到这个时候，子物体的激活选项并没有变化，所以只要父物体被激活，子物体立刻也会立即被重新激活。所以，不能通过在脚本中直接读取相关属性来判断物体是否被激活。相应的，Unity提供了activeInHierarchy属性来进行判断，这个属性考虑到了父物体的影响。

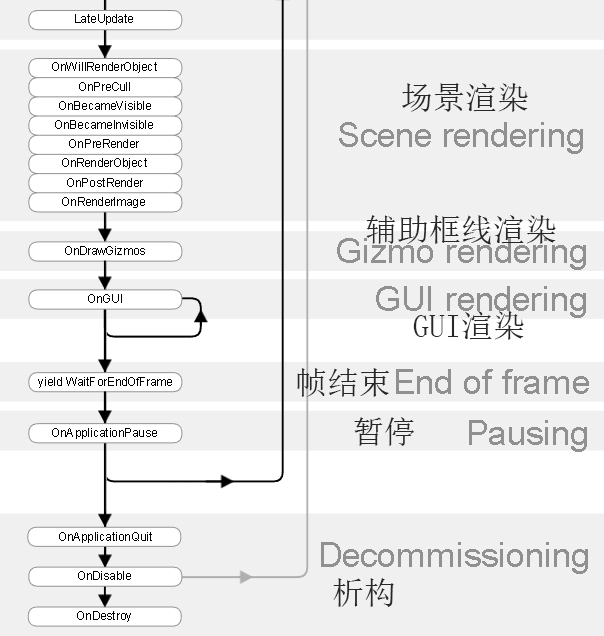
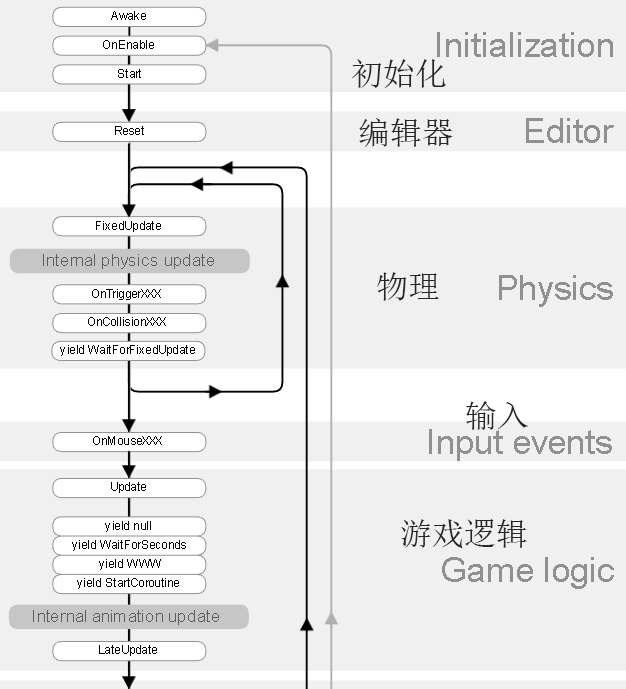
|  |
| --- |
| if (gameObject.activeInHierarchy)  {  // ....  } |

## 脚本组件生命期

脚本不一定要继承MonoBehavior，例如一个用于计算的class、一个只是定义了简单的struct的脚本，都不需要继承Unity。

但是如果脚本需要作为组件使用，能够被挂载到游戏物体上，且能够处理Unity的事件，那么这个脚本就必须继承MonoBehavior，被Unity当作一个标准的组件对待。

MonoBehavior有着严格的设计和规则，它的事件触发具有明确的先后顺序，这个顺序和引擎的处理方式有关，Unity官方资料明确解释了“MonoBehavior生命期”，如下图：



可以看到，除了常见的Start和Update以外，还有众多事件可能会被用到，这里简单罗列一下：

|  |  |
| --- | --- |
| **事件** | **说明** |
| **Update** | 每一帧被调用，按帧执行的逻辑都放在这里。 |
| **LateUpdate** | 每一帧游戏逻辑的最后，渲染之前调用 |
| **FixedUpdate** | 固定更新，专门用于物理系统，因为物理更新的频率必须保证稳定性。 |
| **Awake** | 当一个脚本实例被载入时被调用。 |
| **Start** | 在Update函数第一次被调用前调用。 |
| **Reset** | 重置为默认值时调用（在Unity编辑器中才有这种情况）。 |
| **OnMouseEnter** | 当鼠标进入到GUI元素或碰撞体中时调用。 |
| **OnMouseOver** | 当鼠标悬浮在GUI元素或碰撞体上时调用。 |
| **OnMouseExit** | 当鼠标移出GUI元素或碰撞体上时调用OnMouseExit。 |
| **OnMouseDown** | 当鼠标在GUI元素或碰撞体上点击时调用OnMouseDown。 |
| **OnMouseUp** | 当用户释放鼠标按钮时调用。 |
| **OnMouseUpAsButton** | 只有当鼠标在同一个GUI元素或碰撞体按下，在释放时调用。 |
| **OnMouseDrag** | 当用户鼠标拖拽GUI元素或碰撞体时调用。 |
| **OnTriggerEnter** | 当碰撞体进入触发器时调用。 |
| **OnTriggerExit** | 当碰撞体脱离触发时调用。 |
| **OnTriggerStay** | 当碰撞体接触触发器时，在每一帧被调用。 |
| **OnCollisionEnter** | 当此Collider/Rigidbody触发另一个Rigidbody/Collider时，将被调用。 |
| **OnCollisionExit** | 当此Collider/Rigidbody停止触发另一个Rigidbody/Collider时，将被调用。 |
| **OnCollisionStay** | 当此Collider/Rigidbody触发另一个Rigidbody/Collider时，将会在每一帧被调用。 |
| **OnControllerColliderHit** | 在移动的时，当Controller碰撞到Collider时OnControllerColliderHit被调用。 |
| **OnJOINTBREAK** | 当附在同一对象上的关节被断开时调用。 |
| **OnParticleCollision** | 当粒子碰到Collider时被调用。 |
| **OnBecameVisible** | 当Renderer(渲染器)在任何相机上可见时调用。 |
| **OnBecameInvisible** | 当Renderer(渲染器)在任何相机上都不可见时调用。 |
| **OnLevelWasLoaded** | 当一个新关卡被载入时此函数被调用。 |
| **OnEnable** | 当对象变为可用或激活状态时此函数被调用。 |
| **OnDisable** | 当对象变为不可用或非激活状态时此函数被调用。 |
| **OnDestroy** | 当MonoBehavior将被销毁时，这个函数被调用。 |
| **OnPrecull** | 在相机消隐场景之前被调用。 |
| **OnPreRender** | 在相机渲染场景之前被调用。 |
| **OnPostRender** | 在相机完成场景渲染之后被调用。 |
| **OnRenderObject** | 在相机场景渲染完成后被调用。 |
| **OnWillRenderObject** | 如果对象可见每个相机都会调用它。 |
| **OnGUI** | 渲染和处理GUI事件时调用。每帧调用一次 |
| **OnRenderImage** | 当完成所有渲染图片后被调用，用来渲染图片后期效果。 |
| **OnDrawGizmosSelected** | 如果你想在物体被选中时绘制Gizmos，执行这个函数。 |
| **OnDrawGizmos** | 如果你想绘制可被点选的Gizmos，执行这个函数。 |
| **OnApplicationPause** | 当玩家暂停时发送到所有的游戏物体。 |
| **OnApplicationFocus** | 当玩家获得或失去焦点时发送给所有游戏物体。 |
| **OnApplicationQuit** | 在应用退出之前发送给所有的游戏物体。 |
| **OnPlayerConnected** | 当一个新玩家成功连接时在服务器上被调用。 |
| **OnServerInitialized** | 当Network.InitializeServer被调用并完成时，在服务器上调用这个函数。 |
| **OnConnectedToServer** | 当你成功连接到服务器时，在客户端调用。 |
| **OnPlayerDisconnected** | 当一个玩家从服务器上断开时在服务器端调用。 |
| **OnDisconnectedFromServer** | 当失去连接或从服务器端断开时在客户端调用。 |
| **OnFailedToConnect** | 当一个连接因为某些原因失败时在客户端调用。 |
| **OnFailedToConnectToMasterServer** | 当报告事件来自主服务器时在客户端或服务器端调用。 |
| **OnMasterServerEvent** | 当报告事件来自主服务器时在客户端或服务器端调用。 |
| **OnNetworkInstantiate** | 当一个物体使用Network.Instantiate进行网络初始化时调用。 |
| **OnSerializeNetworkView** | 在一个网络视图脚本中，用于自定义变量同步。 |

## 标签（Tag）

标签是一个可以标记在游戏物体上的记号，它一般是一个简单的单词。比方说，你可以为玩家角色添加一个“Player”的标签，并为敌人角色添加一个“Enemy”的标签，还可以为地图上的道具添加一个“Collectable”标签。

在脚本中查找和指定物体时，使用标签是一种非常好的方法。这种方法可以避免总是采用公开某个变量的方式来指定游戏物体，那样还需要通过拖拽的操作才能给变量赋初值。采用标签来查找物体可以简化编辑工作，更具通用性。

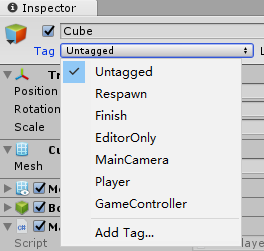
标签还特别适合于用在处理碰撞的时候，当玩家角色与其它物体发生碰撞时，你可以通过判断碰到的物体是敌人、道具或是其它东西，来进行下一步处理。

可以使用GameObject.FindWithTag()方法来通过标签查找物体。下面的例子使用了这个函数，在找到了“Respawn”标签的物体后，它将事先准备好的预置体respawnPrefab实例化成一个新的物体，并将其放置在原来带有Respawn标签的物体的位置。

|  |
| --- |
| using UnityEngine;  using System.Collections;  public class Example : MonoBehaviour {  public GameObject respawnPrefab;  public GameObject respawn;  void Start() {  if (respawn == null)  respawn = GameObject.FindWithTag("Respawn");    Instantiate(respawnPrefab, respawn.transform.position, respawn.transform.rotation) as GameObject;  }  } |

### 为物体设置标签

检视窗口的上方显示了标签（Tag）和层级（Layer）的下拉菜单



在标签的下拉菜单中点击任意一个标签名称，就可以为物体指定该标签了。物体的默认标签为Untagged，是“未指定标签”的意思。

### 创建新的标签

要创建一个新的标签，在标签下拉菜单中选择Add Tag，之后检视窗口会切换到标签与层级管理器（Tag and Layer Manager）。注意，标签一旦创建就不可以再被修改，只能删除并重新创建。

层级与标签类似，都用来标记物体，但是层级有一些非常灵活的用途，比如层级可以用来定义游戏物体在场景中如何被渲染，以及限定哪些碰撞会发生，哪些碰撞会被忽略。之后我们会再次用到层级的概念。

### 小提示

一个游戏物体只能被指定一个标签。

Unity预置了一些常用标签，在标签管理器中你不能修改这些预置的标签，它们是：

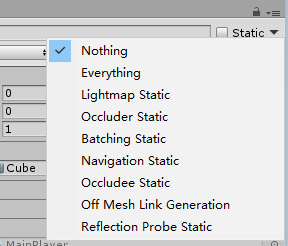
1. Untagged（没有标签）
2. Respawn（出生）
3. Finish（完成）
4. EditorOnly（编辑器专用）
5. MainCamera（主摄像机）
6. Player（玩家）
7. GameController（游戏控制器）

可以用任意一个单词为标签命名，甚至可以起一个很长的词组做为名称，但是那样的话可能会不太方便，比方说在界面中看不到完整的名字。

## 静态物体（Static GameObject）

如果引擎事先知道了某一个物体在游戏进行中是否会移动，那么就可以针对性地应用一系列优化策略。如果一个物体是静态的，即不会移动的，那么引擎就可以假定它不会收到任何其它物体或者事件的影响，从而预先计算好物体的信息。比如说，渲染器可以将场景中许多静态物体合并为一个整体，这样就可以通过一次渲染就将它们全部处理完毕，这种做法也称之为批量渲染。

在检视窗口中，每个游戏物体名称的右侧都有一个静态（Static）勾选框以及一个菜单，它用来指定物体是否为静态的，且可以进一步指定在某些子系统中是静态的。可以独立地设置游戏物体在每个子系统中是否为静态，这样就可以对物体进行非常细致的优化。



（静态标记菜单，可以针对性设置物体在每个子系统中是否是静态的）

### 静态设置

静态菜单中的Everything和Nothing选项，分别同时启用或禁用物体在所有子系统中的静态特性以便优化。这些子系统包含：

**[Lightmapping](https://docs.unity3d.com/Manual/GIIntro.html)**：场景中的高级光照特性。

**[Occluder and Occludee](https://docs.unity3d.com/Manual/OcclusionCulling.html)**：根据物体在特定摄像机下的可见性，进行渲染优化。

**[Batching](https://docs.unity3d.com/Manual/DrawCallBatching.html)**：将多个物体合并为一个整体进行渲染。

**[Navigation](https://docs.unity3d.com/Manual/Navigation.html)**：在寻路系统中，将此物体做为静态的障碍物。

**[Off-mesh Links](https://docs.unity3d.com/Manual/class-OffMeshLink.html)**：寻路系统中的网格链接。

**[Reflection Probe](https://docs.unity3d.com/Manual/class-ReflectionProbe.html)**：反射探针优化。

某些子系统与内部渲染方式有较大关联，可以在相关文档中阅读它们的细节。

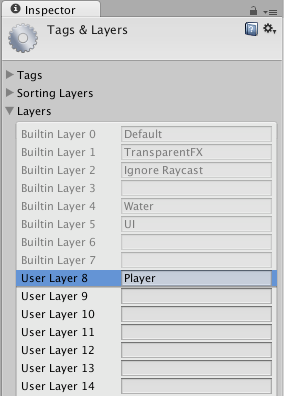
## 层级（Layer）

层级（Layer）和游戏物体、标签一样，都是Unity最基本的概念之一。层级最有用、最常用的地方是用来让摄像机仅渲染场景中一部分物体；还可以让让灯光只照亮一部分物体。除此以外，层级还能用来在碰撞检测、射线检测时，只让某些物体发生碰撞，另一些物体不发生碰撞。

### 新建层级

在为物体指定层级之前，我们先新建一个层级。点击主菜单的Edit > Project Settings > Tags and Layers选项，会在检视窗口中打开层级和标签窗口。

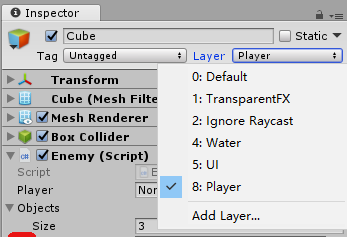
这步操作在之前介绍标签时的完全一样，不同的是，在学习标签时我们展开了标签菜单，这里我们要展开层级菜单。



如图，将新的层级User Layer 8起名为Player，就建立了一个新的Player层级，序号为8。

### 为物体指定层级

现在我们已经新建了一个层级，现在将物体指定为这个层级。

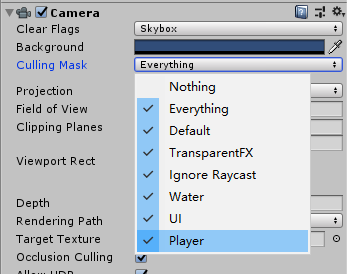


如图，只要选中物体，在检视窗口中点击Layer下拉菜单，并选择层级名称即可。

### 仅渲染场景的一部分

层级配合摄像机的剔除遮罩（Culling Mask）使用，就可以选择性地显示某些层级的物体，而另一些不渲染。要做到这一点，只需要在摄像机中选中需要渲染的层即可。

点击摄像机的Culling Mask下拉菜单，打勾的层就是要显示的层。



（摄像机的剔除遮罩设置）

注意Unity中有很多类似这样的下拉菜单，菜单的最上面两项是特殊的，当点击Nothing时，所有勾选都会取消，表示全部不选；而点击Everything时，则所有选项都会被勾选。使用这两个快捷选项有助于快速选中必要的层。比如说，要仅选中UI层和Player层，就可以先选择Nothing，然后再次分别单独勾选UI和Player层即可。

另外，UI系统如果采用屏幕空间画布，则不会受摄像机剔除遮罩的影响。UI系统的问题未来有单独的章节进行介绍。

### 选择性的射线检测

使用层级可以让射线检测忽略某些碰撞体，类似也是用“层级”和“遮罩”这两个概念来设置的。例如有时需要发射一条射线，仅和“Player”层级的物体发生碰撞，而忽略其它层。

方法Physics.Racast用来发射一条射线，它可以带有一个参数叫做layerMask，layerMask是一种利用位标记来作为遮罩（bitmask）的参数。

用位（bit）作为标记是二进制相关的一种方法，原理是说int型的变量由32个位组成，每个位的值只能是0或1，那么就可以根据某一个位为0还是为1来选中或者忽略某一层。

如果layerMask所有的位都是1，那么就会和所有的层发生碰撞。如果layerMask等于0，那么它就不会和任何层、任何物体发生碰撞。

|  |
| --- |
| // 将1左移8位  int layerMask = 1 << 8;  // 将layerMask设置为只有从右数第8位是1，其它位是0，则只会和Layer 8 Player层发生碰撞  if (Physics.Raycast(transform.position, Vector3.forward, Mathf.Infinity, layerMask))  Debug.Log("The ray hit the player"); |

如果是玩家射击发出的检测射线，就恰恰相反，需要和除了玩家的所有物体发生碰撞：

|  |
| --- |
| void Update () {  // 将1左移8位  int layerMask = 1 << 8;  // 按位取反，所有值为0的位变为1，值为1的位变为0  layerMask = ~layerMask;    RaycastHit hit;  // 发射射线，Debug.DrawRay用于画出辅助框线  if (Physics.Raycast(  transform.position, transform.TransformDirection (Vector3.forward), out hit, Mathf.Infinity, layerMask)) {  Debug.DrawRay(transform.position  , transform.TransformDirection (Vector3.forward)  \* hit.distance, Color.yellow);  Debug.Log("Did Hit");  } else {  Debug.DrawRay(transform.position  , transform.TransformDirection (Vector3.forward) \*1000  , Color.white);  Debug.Log("Did not Hit");  }  } |

如果在Raycast方法调用时省略layerMask参数，则默认除了“Ignore Raycast”这层不会被碰撞，其它层都可能发生碰撞。使用“Ignore Raycast”层可以专门指定某些物体不受射线检测碰撞，这种方法比较简单但是只是一个小技巧而已，当指定了layerMask时，这一层也和其它层一样并没有什么特别。

重要提示：序号为31的最后一层是特别的，编辑器内部会利用它来做预览使用。所以添加层时不要使用31层，可能会造成冲突。

## 预制体（Prefab）

在场景中创建物体、添加组件并设置合适参数的操作一开始会觉得方便，但是当场景中会用到大量同样的NPC、障碍物或机关时，创建以及设置属性的操作就会带来巨大的麻烦。单纯拷贝这些物体看似可以解决问题，但是由于这些物体都是独立的，所以还是需要一个一个单独修改它们。通常，我们希望所有这些物体会引用某一个模板物体，这样只要修改了模板物体或其中一个物体的实例，就可以同时修改所有相关的物体。

所以Unity提供了预置体（Prefab）这个概念，专门用来实现这一重要功能。它允许事先保存一个游戏物体，包括该物体上挂载的组件与设置的参数。这样预置体就可以成为一个模板，可以用这个模板在场景中创建物体。一方面，任何对预置体文件的修改可以立即影响所有相关联的物体上；另一方面，每个物体还可以重载（override）一些组件和参数，以实现与模版有所区别的设置。

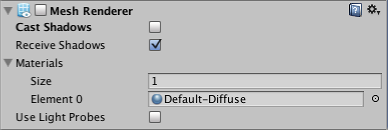
注意：当你拖拽一个资源文件到场景中时（比如一个模型），Unity会自动创建一个新的游戏物体，且当原始资源修改时，也会影响到这些相关的游戏物体。但是这种概念看起来像是预置体，但是和预置体是完全不同的，所以不适用下面介绍的预置体的特性。仅仅是这种“引用关系”有相似之处。

### 使用预置体

创建预置体有两种常用方法，一种是在工程窗口中的某个文件夹内点击右键，选择Create > Prefab 创建一个空白预置体，然后将场景中制作好的某个游戏物体拖拽到空白预置体上；另一种方法是直接将某个游戏物体从场景中拖拽到文件夹中。在创建好预置体以后，将另一个游戏物体拖拽到prefab文件上，会提示是否替换预置体。

预置体是一个后缀为.prefab的资源文件。层级视图中，所有与预置体关联的游戏物体，会以蓝色名称显示（普通物体的名称是黑色的）。

之前提到过，一方面修改预置体后可以影响所有相关的物体，另一方面物体又可以单独修改一些属性而不影响预置体。这个设计非常有用，在实际使用时，可以创建很多相似的NPC角色但是每个角色的参数又可以略有不同，满足游戏丰富性的要求。为了更清楚地显示哪些参数和预置体一致，哪些参数是独特的，在检视窗口中会将独特的参数以粗体显示。特别是当为物体加上一个新的组件时，整个新组件的文本都会以粗体显示。



（某个关联预置体的游戏物体修改了网格渲染器的产生阴影（Cast Shadows）选项）

同样可以在脚本中，以预置体为模板创建游戏物体，本节中会讲解具体的例子。

### 通过游戏物体实例修改预置体

与预置体关联的游戏物体，会在检视窗口中的上方多出三个按钮：选择（Select），回滚（Revert）与应用（Apply）。

选择按钮会选中物体相关联的那个预置体，点击后，在工程窗口中会高亮显示该预置体。这有助于迅速找到相关的预置体。

应用按钮可以将本物体上修改的那些组件和参数写回到原始的预置体中（但是变换组件的位置信息不会写回预置体，原因不难想到）。这个设计可以方便我们通过任何一个物体修改预置体，有时会非常方便，特别是在某些预置体只有一个实例的时候。

回滚按钮会将游戏物体修改过的组件和属性恢复到和预置体一致。这个功能用于试验性修改某些参数以后，将物体恢复到原始状态。

### 在运行时实例化预置体

到这里，读者应该已经对预置体的基本概念有了大体的掌握。预置体简单来说就是一个实现定义好的游戏物体，可以稍后在游戏中反复多次使用。

在游戏运行时，通过脚本创建游戏物体非常方便，无论游戏物体有多么复杂，操作都是一样是非常简洁的。下面通过多个例子说明具体的方法

#### 常见的情景

为了展示预置体的强大功能，我们先考虑一些非常常见的情况：

* 通过反复实例化单独的“砖块”预置体，来创建一堵完整的墙。
* 当击发一个火箭发射器时，会创建一个火箭的预置体。火箭的预置体包含了网格、刚体、碰撞体等组件，且还包含一个子物体用于播放粒子。
* 机器人爆炸为多个散落部件的场景。一个完整的机器人被摧毁并散落为多个碎片，多个碎片可以用预置体来实现，每个碎片都可以包含刚体与粒子组件，以模拟真实的效果。只需要通过简单的创建物体、替换预置体等操作就可以做出如此震撼的效果。

#### 建造一堵墙

新建一个C#脚本，命名为Wall.cs，确保class名称为Wall：

|  |
| --- |
| using UnityEngine;  public class Wall : MonoBehaviour {  public Transform brick;  void Start()  {  for (int y = 0; y < 5; y++)  {  for (int x = 0; x < 5; x++)  {  Instantiate(brick, new Vector3(x, y, 0), Quaternion.identity);  }  }  }  } |

创建一个空白物体命名为TestWall，并挂载好这个脚本。

这段代码不仅非常精简，且具有很好的复用性。整段代码与具体的砖块无关，所有必要的组件和参数只要事先在预置体中设置好即可。使用这段代码需要事先准备好预置体，下面是详细操作步骤：

1. 在场景中创建一个3D方块。右键点击层级窗口，选择3D Object > Cube。
2. 选中这个方块，并添加组件 Physics > Rigidbody。
3. 选中这个方块，将它拖拽到工程窗口中，就创建了一个预置体，将预置体起名为“Brick”。
4. 预置体成功创建后，就可以删除场景中的方块了。

创建好预置体Brick之后，我们要将它关联到脚本中。

1. 选中前面准备好的名为TestWall的物体，为它挂载好脚本以后，可以在检视窗口中看到公开的变量“brick”，该变量可以用一个预置体赋值。
2. 拖动Brick预置体文件拖动到brick变量上，就将脚本和预置体关联起来了。

这时运行游戏，如果顺利的话，就可以看到在TestWall物体的位置上，创建了一堵方块组成的墙。

以上的步骤就是一个典型的使用预置体的工作流程，复杂的大型工程中也会以类似的方式使用预置体。随着多次练习会更好地理解这个过程。

像这样使用预置体间接地创建游戏物体有什么好处呢？一个显然的好处是可以迅速调整预置体，整个墙体就会随之变化。比如说你可以任意改变砖块的刚体质量，换一种砖块的材质，修改墙体的表面阻力等等，只需要修改预置体的设置，一次操作即可完成。

#### 创建火箭弹与爆炸效果

下面来看看稍微复杂些的情景：

当玩家按下开火键时，火箭筒就会创建一个火箭弹的实例并发射。火箭弹的预置体包含了模型、刚体、碰撞体以及一个子物体用于表现拖尾粒子效果。

当火箭弹撞击到其它物体以后，就会创建一个爆炸的预置体。爆炸的预置体包含一个粒子组件、一个灯光组件以表现绚丽的爆炸效果，另外还要附加一个脚本用于在游戏逻辑中产生实质性的伤害。

一般我们不会完全用脚本一步一步组装好一个火箭弹物体，显然使用预置体是更好的方法。将复杂的、具体的创建火箭弹和粒子的步骤交给编辑器去完成，脚本里只需要单纯地用一句代码实例化它即可。而且在创建了实例以后，还可以接着在脚本中操作这个实例（游戏物体），比如修改它的速度等等。

在实现具体的效果之前，我们更关心在技术上如何使用这些预置体。所以可以先用一些简单的物体来代替真实的火箭筒、粒子，未来只需要修改预置体即可，不需要再次修改脚本代码。预置体的这种使用方法，将美术效果的实现和游戏逻辑的实现很好地分离开来，让一个复杂的问题变成多个简单的问题。

简化的代码如下所示：

|  |
| --- |
| using UnityEngine;  public class RocketLauncher : MonoBehaviour {  // 挂载脚本的物体必须包含刚体组件，否则就会在开枪时报错。  // 这种写法符合组件式编程的范例，rocket既是预置体本身，又是预置体上面的刚体组件  // 组件模型允许我们用物体上的任意一个组件指代这个物体  public Rigidbody rocket;  public float speed = 10f;  void FireRocket()  {  // rocketClone指代着创建出来的火箭弹，它本身又是火箭弹上的刚体组件  Rigidbody rocketClone = (Rigidbody)Instantiate(rocket, transform.position, transform.rotation);  // 接下来直接操作rocketClone实例  // 先修改刚体  rocketClone.velocity = transform.forward \* speed;  // 举例：通过一个组件可以查找到其它任何组件，比如脚本组件MyRocketScript  // rocketClone.GetComponent<MyRocketScript>().DoSomething();  }  // Fire1按键默认是鼠标左键或Ctrl键  void Update()  {  if (Input.GetButtonDown("Fire1"))  {  FireRocket();  }  }  } |

该脚本的使用方法与创建一堵墙的方法是类似的：

1. 先创建一个物体代表火箭筒，在火箭筒上挂载上述脚本RocketLauncher.cs。
2. 创建一个火箭弹预置体，火箭弹必须包含刚体组件。
3. 将火箭弹预置体拖拽到火箭筒脚本的rocket变量上。
4. 运行游戏，点击鼠标左键即可发射火箭弹。

#### 将一个物体替换为散落的零件

一般来说，要表现一个角色死亡，可以让该角色播放一段死亡动画，同时小心处理它所挂载的所有功能性脚本，以停止它的逻辑功能。

还有一种比较好的方法，就是在角色死亡的瞬间，创建另一个准备好的表示死亡的角色来替换它，这样做的好处是灵活性非常强，而且原始角色与死亡的角色可以做分别的调整和优化。你可以让死亡的角色拥有另一套材质，另一套死亡状态的脚本、配上一些散落的子物体来表现更好的效果。

进一步说，这种思路除了灵活性强，更大的优势在于可以为死亡的角色定制一套完全不同的模型，由于死亡状态和非死亡状态两套游戏物体的完全独立的，对于已经死亡的角色就可以用更少的多边形、更少量的子物体来表现，这样游戏的整体性能就会得到提高；反过来说，未死亡的角色也不需要带着死亡后才需要的模型，同样也会得到性能上的提升。只要灵活运用实例化预置体的方法，从技术上讲上述效果的实现会非常容易。

接下来用一个破碎的飞机来举例，我们准备制作一个场景，飞机在空中解体，各个部分冒着烟下坠。和之前一样，还是标准的实例化预置体的方法：

1. 准备一些表示表示飞机解体的各种部件。
2. 将这些零件的模型文件拖拽到场景中。
3. 同时选中这些零件，添加刚体组件Rigidbody。这种批量操作的方式很方便。
4. 批量添加盒子碰撞体（Box Collider）组件，方法同上。
5. 为了效果逼真，可以给每个零件添加一个冒烟的粒子。
6. 由于游戏物体比较多，可以创建一个空物体作为它们的父物体，方便管理。

这时基本素材都准备完毕，如果没有合适的资源，可以先随意创建几个模型作为替代。然后运行游戏，看看是否有零件分别下落的效果。

调整参数，基本满意后将空的父物体拖拽到工程窗口中，创建预置体。这样做相比创建许多预置体文件要清晰、方便很多。

接下来用脚本表现飞机破碎的过程。

|  |
| --- |
| using System.Collections;  using UnityEngine;  public class DestroyPlane : MonoBehaviour {  public GameObject wreck; // 预置体放在这个变量中  // 这种Start方法的写法比较特别。效果是3秒以后执行KillSelf方法  IEnumerator Start()  {  yield return new WaitForSeconds(3);  KillSelf();  }  void KillSelf()  {  // 实例化表示解体的预置体  GameObject wreckClone = (GameObject)Instantiate(wreck, transform.position, transform.rotation);  // 销毁飞机本身  Destroy(gameObject);  }  } |

注：将该脚本挂载到未损坏之前的飞机上，三秒后飞机就会破碎，用预置体代替。

#### 将多个物体按特定的形状摆放

接下来我们思考这样一个例子：将一些物体摆放到场景中，且等间距地围成一个圆圈。

如果是在场景中直接摆放这些物体，那么很难恰好摆成一个圆圈的形状，因为手工对齐坐标非常困难。而用脚本来实现就只需用到一点小技巧：

|  |
| --- |
| public GameObject prefab; // 预置体  public int numberOfObjects = 20; // 物体总数  public float radius = 5f; // 圆圈半径  void Start() {  for (int i = 0; i < numberOfObjects; i++) {  // 算出物体间隔角度  float angle = i \* Mathf.PI \* 2 / numberOfObjects;  // 利用三角函数求位置  Vector3 pos = new Vector3(Mathf.Cos(angle), 0, Mathf.Sin(angle)) \* radius;  Instantiate(prefab, pos, Quaternion.identity);  }  } |

再看另一种摆放的规则，矩形：

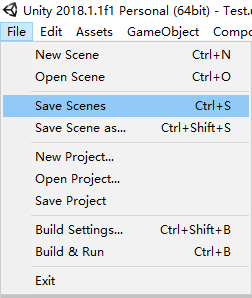
|  |
| --- |
| // 按照矩形实例化预置体  public GameObject prefab;  public float gridX = 5f; // X方向个数  public float gridY = 5f; // Y方向个数  public float spacing = 2f; // 物体之间间隔  void Start() {  for (int y = 0; y < gridY; y++) {  for (int x = 0; x < gridX; x++) {  Vector3 pos = new Vector3(x, 0, y) \* spacing;  Instantiate(prefab, pos, Quaternion.identity);  }  }  } |

## 保存工程的注意事项

一个Unity工程所包含的全部信息是非常复杂的，并不是说资源本身的数量多，而是不同的数据需要完全不同的保存方式。这样就带来了一个小问题：当保存工程时，不同的变动保存的时机和方法不同，需要解释一下，在实践中注意这一点可以避免因操作不当而丢失数据。

在实际开发中强烈建议使用版本控制工具（SVN、Git等）来监视工程的逐步变化，也方便在出现问题时回滚工程，减小损失。

### 保存当前场景

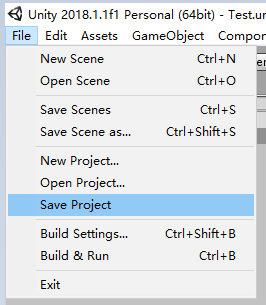


保存场景时，会保存所有物体在层级窗口中的任何变化。包括所有的添加、删除、移动物体操作，场景中的物体的组件的增加删除，以及场景中物体的组件参数的变化。

不仅可以在主菜单中选择保存场景（Save Scene），还可以使用快捷键Ctrl+S（Mac系统Cmd+S）。

实际上，保存场景时还会另外执行保存工程的操作（Save Prject），这意味着当保存场景时，不仅保存了场景信息，绝大部分数据包括工程也被保存了。接下来会讲解保存工程。

### 保存工程



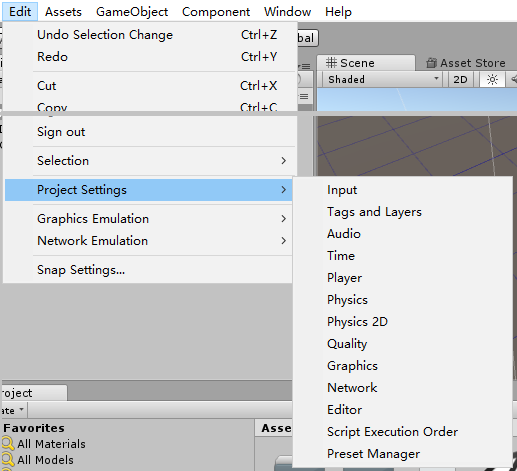
Unity工程中，有很多数据是不针对具体场景的，可以称之为工程数据。理论上来说，这些数据与场景的保存是分离的，可以在菜单中选择“保存工程”（Save Project）命令单独保存工程数据。

保存工程不会保存场景的改动，而只包含工程信息的改动。有时我们在场景中做了一些测试，不想将这些场景保存，但是还是要保存工程的改动，这时就要用到保存工程的功能了。

保存工程，包括以下内容，分节讲解：

#### 所有的工程设置（Project Settings）:

所有的“工程设置”菜单中的内容，例如自定义的输入轴、添加的标签和层级、修改过的物理系统参数都被包含在内。



（工程设置菜单项，其中包含的数据都是工程数据）

对这些选项的修改会被保存在工程的“Library”文件夹中：

输入（Input）：保存为InputManager.asset

标签和层级（Tags And Layers）：保存为 “TagManager.asset”

音频（Audio）：保存为 “AudioManager.asset”

时间（Time）：保存为 “TimeManager.asset”

播放器（Player）：保存为 “ProjectSettings.asset”

物理（Physics）：保存为 “DynamicsManager.asset”

2D物理（Physics 2D）：保存为 “Physics2DSettings.asset”

显示品质（Quality）：保存为 “QualitySettings.asset”

图形（Graphics）：保存为 “GraphicsSettings.asset”

网络（Network）：保存为 “NetworkManager.asset”

编辑器（Editor）：保存为 “EditorUserSettings.asset”

#### 发布设置

发布设置（Build Setting）也会被保存在Library文件夹中，文件名为“EditorBuildSettings.asset”。

Unity2018版，会在发布设置窗口中自动连接服务器，如果未登陆会出现警告框。

#### 资源的修改

在工程窗口中选中资源并修改它的设置时，某些资源不带有“应用”（Apply）按钮，这些改动也被算作工程数据，和工程一起保存。例如：

* 材质（Material parameters）
* 预置体（Prefabs）
* 动画控制器（Animator Controllers）
* 动画遮罩（Avatar Masks）
* 其他所有不带有“应用”按钮的资源

### 不需要保存的改动

有一些改动是立即存盘的，也就不需要单独执行保存命令。包含以下操作：

#### 所有带有“应用（apply）”按钮的资源

某些资源在修改参数时，在检视窗口中带有一个“Apply”按钮，当你修改过参数后，点击这个应用按钮就会立即生效并保存到磁盘中。本质上是由于这种资源被修改后，需要被重新加载才会生效，所以必须立即存盘。这类资源的例子比如：

* 贴图资源的参数
* 3D模型的导入参数
* 声音文件的压缩参数
* 其它所有带有“Apply”按钮的资源

#### 其它会被立即存盘的改动

还有一些改动也不需要单独保存，下面举例说明：

* 新建的资源文件，比如新建的材质或预置体
* 光照（Lighting）烘培的数据（烘培可以理解为运行前预先计算好）
* 寻路（navigation）烘培的数据
* 烘培的遮挡剔除（occlusion culling）的数据
* 脚本等其他直接写入磁盘的

## 输入

输入操作是游戏的基础之一。Unity不仅支持绝大部分传统的操作方式，例如手柄、鼠标、键盘等，而且还支持触屏操作、重力传感器、手势等等移动平台上的操作方式。另外Unity对新出现的VR和AR系统也有完善的支持，而且仍然在不断进步之中（实际上反过来说，主流的VR、AR设备都会很好地支持Unity，因为这样才能方便开发者制作出大量优秀的作品）。

另外，Unity还会利用手机或PC的麦克风、摄像头作为特殊的输入设备。

### 传统输入设备与虚拟输入轴

Unity支持键盘、手柄、鼠标和摇杆等传统输入设备。

为了支持此类设备，Unity设计了一些概念。第一个概念叫做“虚拟控制轴”（Virtual axes）。虚拟控制轴将不同的输入设备，比如键盘或摇杆的按键，都归纳到一个统一的虚拟控制系统中。比如键盘的W、S键以及手柄摇杆的上下运动，默认都统一映射到竖直（Vertical）输入轴上，这样就屏蔽了不同设备之间的差异，让开发者可以用同一套非常简单的输入逻辑，同时兼容多种输入设备。

再例如鼠标左键和键盘的Ctrl键默认都映射到“Fire1”这个虚拟轴上，这样无论用键盘和鼠标都可以实现开火操作了。而且所有这些设置都可以删除或者修改，也可以添加新的虚拟轴。

使用输入管理器（Input Manager）可以查看、修改或增删虚拟轴，而且操作方法非常容易理解。

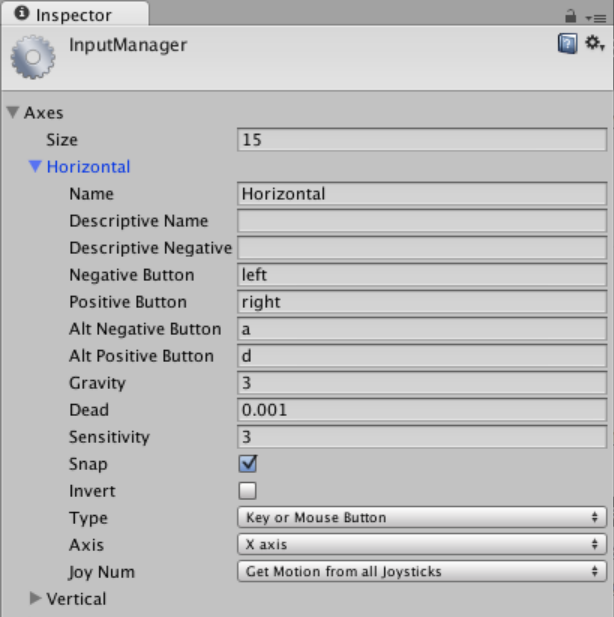
现代的游戏中往往允许玩家在游戏中自定义按键，所以使用Unity的输入管理器就更为必要了。通过一层虚拟轴间接操作，可以避免在代码中直接写死操作按钮，而且还能允许通过动态修改虚拟轴设置来实现改变键位的功能。

关于虚拟输入轴，还有一些需要知道的：

1. 脚本可以直接通过虚拟轴的名称读取那个轴的输入状态。
2. 创建Unity工程时，默认创建了以下虚拟轴：
   * 横向和纵向输入被映射在键盘的W、A、S、D键以及方向键上。
   * Fire1、Fire2、Fire3这三个按钮映射到了鼠标的左中右键以及Ctrl、Alt等键位上。
   * 鼠标移动可以模拟摇杆输入（和鼠标在屏幕上的位置无关），被映射在专门的鼠标偏移轴上。
   * 其它常用虚拟轴，例如跳跃（Jump）、确认（Submit）和取消（Cancel）。

#### 编辑和添加虚拟输入轴

要添加新的虚拟轴，只需要点击主菜单的Edit > Project Settings > Input选项。点击后会在检视窗口中显示一个输入管理器，在里面就可以修改或添加虚拟轴了。



注意轴具有正、负两个方向，英文记作Positive和Negative。某些相反的动作可以只用一个轴来表示。比方说如果摇杆向上为正，那么向下就是同一个轴的负方向。

每个虚拟轴可以映射两个按键，第二个按键作为备用，功能一样。备用的英文为Alternative。

| **属性** | **功能** |
| --- | --- |
| Name | 轴的名字。在脚本中用这个名字来访问这个轴 |
| Descriptive Name | 描述性信息，在某些窗口中显示出来方便查看。（正方向） |
| Descriptive Negative Name | 描述性信息，在某些窗口中显示出来方便查看。（负方向） |
| Negative Button | 该轴的负方向，绑定某个按键。 |
| Positive Button | 该轴的正方向，绑定某个按键。 |
| Alt Negative Button | 该轴的负方向，绑定某个备用按键。 |
| Alt Positive Button | 该轴的正方向，绑定某个备用按键。 |
| Gravity | 轴回中的力度，下文解释。 |
| Dead | 轴的死区，下文解释。 |
| Sensitivity | 敏感度。 |
| Snap | 保持式按键。比如输入方向键下，则一直保持下的状态，直到再次按上。 |
| Invert | 如果勾选，则正负方向交换。 |
| Type | 控制该虚拟轴的类型，比如手柄、键盘是两种不同的类型。 |
| Axis | 很多手柄的输入不是按钮式的，这时就不能配置到“Button”里面，而是要配置到这里。可以理解为实际的“操作轴”。 |
| Joy Num | 当有多个控制器时，要填写控制器的编号。 |

上图中关于Gravity、Dead等属性需要解释一下：

现代游戏的方向输入和早期游戏不太一样。早期游戏中，上、中、下都是离散的状态可以直接用1、0、-1来表示。而现代游戏输入往往具有中间状态，比如0、0.35、0.5、0.7、1，是带有多级梯度的，比如轻推摇杆代表走路，推到底就是跑步。所以现代游戏的输入默认都是采用多梯度的模式。

虽然键盘没有多级输入的功能，Unity依然会模拟这个功能，也就是说当你按住W键时，这个轴的值会以很快的速度逐渐从0增加到1。

所以上表中“Gravity”和“Sensitivity”的含义就不难理解了，它们影响着虚拟轴从1到0、从0到1的速度以及敏感度。具体调试方法略，建议使用默认值。

还有“Dead”死区需要单独说明。由于实体手柄、摇杆会有一些误差，比方说放着手柄不动时，某些手柄的输出值可能会在-0.05~0.08之间浮动。这个误差有必要在程序中排除。所以Unity设计了“死区”的功能，在该值范围内的抖动被忽略为0，这样就可以过滤掉输入设备的误差。

另外，鼠标放在这些属性上会有详细的提示，可以参考。

#### 在脚本中处理输入

读取输入轴的方法很简单：

float value = Input.GetAxis ("Horizontal");

得到的值的范围在-1到1之间，默认位置为0。这个读取虚拟轴的方法和具体控制器是键盘还是手柄无关。另外也有一些特例，比如如果用鼠标控制虚拟轴，就有可能由于移动过快超过-1到1的范围。

注意：可以创建多个同样名字的虚拟轴。Unity可以同时管理多个同名的轴，最终结果以变化最大的轴为准。这样做的原因是很多游戏可以同时用多种设备操作，比如PC游戏可以用键盘鼠标或手柄操作，手机游戏可以用重力感应器或手柄操作。这种设计有助于用户在多种操作设备之间切换，且在脚本中不用去关心这一点。

#### 按键名称

要映射按键到轴上，需要在正方向输入框或者负方向输入框中，输入正确的按键名称。按键名称在下面列举一下。

按键名称规则和举例：

1. 常规按键： “a”, “b”, “c” …
2. 数字键： “1”, “2”, “3”, …
3. 方向键： “up”, “down”, “left”, “right”
4. 小键盘键：“[1]”, “[2]”, “[3]”, “[+]”, “[equals]”
5. 修饰键：“right shift”, “left shift”, “right ctrl”, “left ctrl”, “right alt”, “left alt”, “right cmd”, “left cmd”
6. 鼠标按钮：“mouse 0”, “mouse 1”, “mouse 2”, …
7. 手柄按钮（不指定具体手柄）：“joystick button 0”, “joystick button 1”, “joystick button 2”, …
8. 手柄按钮（指定具体手柄序号）： “joystick 1 button 0”, “joystick 1 button 1”, “joystick 2 button 0”, …
9. 特殊键： “backspace”, “tab”, “return”, “escape”, “space”, “delete”, “enter”, “insert”, “home”, “end”, “page up”, “page down”
10. 功能键：“f1”, “f2”, “f3”, …

注：脚本中和编辑器中使用按键的名称是一致的。例如：

|  |
| --- |
| value = Input.GetKey ("a"); |

另外，可以使用KeyCode枚举类型来指定按键，与用字符串的效果是一样的。

### 移动设备的输入

对于移动设备来说，Input类还提供了触屏、加速度计以及访问地理位置的功能。

另外移动设备上还经常会用到虚拟键盘，即在屏幕上操作的键盘，Unity中也有相应的访问方法。

本节专门讨论移动设备特有的输入方式。

#### 多点触摸

iPhone、iPad等设备提供同时捕捉多个手指触摸操作，通常最多可以处理最多5根手指同时触摸屏幕的情况。通过访问Input.touches属性，可以以数组的方式处理多个手指当前的位置等信息。

安卓设备上多点触摸的规范相对灵活，不同的设备能捕捉的多点触摸数不尽相同。较老的设备上可能只支持1到2个点同时触摸，新的设备上可能会支持5个点的触摸操作。

每一个手指的触摸信息以Input.Touch结构体来表示。

| **属性** | **功能** |
| --- | --- |
| fingerId | 该触摸的序号 |
| position | 触摸到屏幕上的位置 |
| deltaPosition | 当前触摸位置和前一个位置的差距 |
| deltaTime | 最近两次改变触摸位置，之间的操作时间间隔 |
| tapCount | iPhone/iPad设备会记录用户短时间内点击屏幕的次数，它表示用户多次点击操作且没有将手拿开的次数。安卓设备没有这个功能，该值保持为1。 |
| phase | 触摸的阶段。可以用它来判断刚开始触摸、触摸时移动或者手指刚刚离开屏幕。 |

phase的取值可以为下列值之一：

|  |  |
| --- | --- |
| Began | 手指刚接触到屏幕 |
| Moved | 手指在屏幕上滑动 |
| Stationary | 手指接触屏幕但还未滑动 |
| Ended | 手指离开了屏幕。这个状态代表着一次触屏操作的结束。 |
| Canceled | 系统取消了这次触屏操作。例如当用户开始拿起手机进行通话，或者触摸点多于5个的时候，这次触屏操作就会被取消。这个状态也代表这次触屏操作结束。 |

以下脚本的功能是在用户点击屏幕时，发射一条射线，在射线碰触到物体以后，实例化一个预制体：

|  |
| --- |
| // Touch.cs  using UnityEngine;  public class Touch : MonoBehaviour {  public GameObject prefab;  void Update () {  foreach (var touch in Input.touches)  {  // 如果某个手指刚开始触摸  if (touch.phase == TouchPhase.Began)  {  // 常用方法：利用摄像机和屏幕上的点，可以确定出一条从手指到场景内的射线  var ray = Camera.main.ScreenPointToRay(touch.position);  // 常用方法：用物理引擎发射这条射线，如果碰到物体则返回true  RaycastHit hitInfo;  if (Physics.Raycast(ray, out hitInfo))  {  // 如果成功碰到物体，则碰撞信息保存在hitInfo中  // hitInfo.point代表碰撞点的位置  Instantiate(prefab, hitInfo.point, Quaternion.identity);  }  }  }  }  } |

该脚本的使用方法：

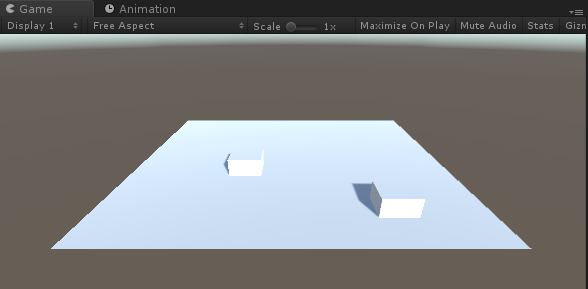
1. 该脚本名为Touch.cs，可以挂载到主摄像机Main Camera上面。
2. 在场景中的(0,0,0)点放置一个平面，以便让射线碰撞到具体的物体。
3. 创建一个预制体，拖拽到脚本的prefab变量上进行赋值。
4. 运行游戏，点击地面即可在点击的位置实例化预制体。

但是，如果在PC上测试，你会发现并没有产生效果。这是因为触摸操作只能在移动设备上起作用，要想测试代码，必须要发布到手机上才可以。

为了让读者理解以上代码的含义，我们可以将触摸控制修改为鼠标控制，只需要修改Touch相关的部分：

|  |
| --- |
| // Touch.cs  using UnityEngine;  public class Touch : MonoBehaviour {  public GameObject prefab;  void Update () {  if (Input.GetMouseButtonDown(0))  {  // 常用方法：利用摄像机和屏幕上的点，可以确定出一条从手指到场景内的射线  var ray = Camera.main.ScreenPointToRay(Input.mousePosition);  // 常用方法：用物理引擎发射这条射线，如果碰到物体则返回true  RaycastHit hitInfo;  if (Physics.Raycast(ray, out hitInfo))  {  // 如果成功碰到物体，则碰撞信息保存在hitInfo中  // hitInfo.point代表碰撞点的位置  Instantiate(prefab, hitInfo.point, Quaternion.identity);  }  }  }  } |

运行成功，效果如下：



（在鼠标点击到平面的位置，会实例化预制体）

#### 模拟鼠标操作

绝大部分移动设备可以用触屏模拟鼠标操作。比如使用Input.mousePosition属性不仅可以获得PC上的鼠标位置，也可以获得移动设备上面的触摸位置。这个功能的原理不难理解，毕竟触屏可以支持多点触摸，而鼠标则是单点操作，这个功能属于向下兼容。移动平台游戏的开发阶段可以暂时用鼠标操作代替触屏操作，但是稍后应当修改为触屏专用的方式，因为操作手感和功能会有很大区别。比如常见的游戏例如《王者荣耀》左手操作虚拟摇杆，右手还要同时释放技能。这时使用模拟鼠标的方式就不可能做到移动的同时释放技能。

#### 加速度计

当移动设备移动时，内置的加速度计会持续报告当前加速度的值，这个值是一个三维向量，因为物体的运动是任意方向的。这个数值和重力加速度的表示方法类似，在某个轴方向上1.0代表该轴具有+1.0g的加速度，而负值则是方向相反。正常竖直持手机时（Home键在下方时），X轴的正方向朝右，Y轴正方向朝上，Z轴正方向从手机指向用户。

通过Input.acceleration属性可以直接访问加速度计当前的数值。

下面是一个尽可能简单的，用加速度计控制物体移动的例子：

|  |
| --- |
| using UnityEngine;  // 用加速度传感器控制物体移动的简单例子  public class AccelerationControl : MonoBehaviour {  float speed = 10.0f;  void Update()  {  Vector3 dir = Vector3.zero;  // 假设设备横置，Home键在右手位置  // 注意转换坐标轴朝向  dir.x = -Input.acceleration.y;  dir.z = Input.acceleration.x;  // 将dir向量的范围限制在单位球体内  if (dir.sqrMagnitude > 1)  dir.Normalize();  // 常用方法，按帧计算  dir \*= Time.deltaTime;  // 移动物体  transform.Translate(dir \* speed);  }  } |

#### 防止加速度计抖动的方法

加速度计的瞬间数值抖动非常严重，会引起不太好的操作体验。接下来介绍一种低通滤波的方法，过滤掉高频数据（可以认为是快速抖动的部分），来让加速度计的数值变化尽可能平滑。

下面的脚本是一个尽可能简单的演示低通滤波的例子：

|  |
| --- |
| public class LowPassFilterAccelerometerValue : MonoBehaviour {  float AccelerometerUpdateInterval = 1.0f / 60.0f;  float LowPassKernelWidthInSeconds = 1.0f;  float LowPassFilterFactor;  private Vector3 lowPassValue = Vector3.zero;  void Start()  {  lowPassValue = Input.acceleration;  LowPassFilterFactor =  AccelerometerUpdateInterval / LowPassKernelWidthInSeconds;  }  Vector3 LowPassFilterAccelerometer()  {  lowPassValue = Vector3.Lerp(  lowPassValue, Input.acceleration, LowPassFilterFactor);  return lowPassValue;  }  } |

每次调用LowPassFilterAccelerometer方法时，都可以获得当前被处理过的加速度计的值。

#### 进一步提高加速度计准确度

从Input.acceleration属性获取的数值并不完全等于硬件采样的数值。简单来说，由于Unity默认每60帧采样一次加速度计的值，这个频率和加速度计更新的频率不完全匹配，导致最终结果的偏差。而加速度计更新的频率又很复杂，它不是一个确定的频率，而是和CPU当前负载相关。

以下代码考虑到了加速度计更新的时间间隔，可以获得尽可能准确的数值：

|  |
| --- |
| float period = 0.0f;  Vector3 acc = Vector3.zero;  foreach (iPhoneAccelerationEvent evt in iPhoneInput.accelerationEvents) {  acc += evt.acceleration \* evt.deltaTime;  period += evnt.deltaTime;  } |

有关加速度计更多的优化方法可以参考其它更详细的文档。

### VR输入概览

Unity支持多种VR设备的专用输入设备。不同的VR设备具有不同的开发插件，例如：

* Oculus OVR，支持Rift，Oculus Go以及Samsung Gear VR。
* Google VR，支持Google Daydream与Cardboard应用。
* Windows混合现实（Windows Mixed Reality），微软的混合现实技术。
* SteamVR，同时支持多种VR设备的开发套件，支持的多种设备中包含HTC Vive。

## 方向与旋转的表示方法

3D空间中的旋转通常有两种表示方法：四元数或者欧拉角。这二者各有利弊。Unity在引擎内部使用四元数表示旋转，但是在检视窗口中以欧拉角来显示物体的旋转角度，欧拉角的表示方法便于查看和编辑。

### 欧拉角

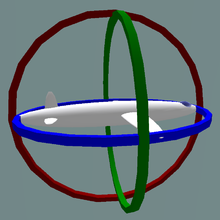
欧拉角表示旋转比较简单和直观，它具有三个值，分别是绕X、Y、Z轴旋转的角度。要将一个物体按照某个欧拉角旋转，需要按照某种顺序依次绕三个轴旋转。

优点1：便于阅读和编辑，因为三个数值与直观角度相对应。

优点2：欧拉角可以方便的表示超过180度的转向。

缺点：欧拉角表示旋转，会遇到万向节锁定问题（Gimbal Lock）。下面进行详细解释。

#### 万向节锁定



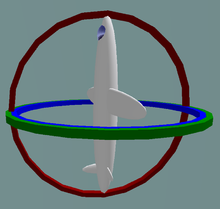
欧拉角表示物体旋转，可以完全对应现实世界的“陀螺仪”。其中，最外层的圈控制物体绕Y轴旋转，而中层的圈固定在外层圈上，内层圈又固定在中层圈上，物体最终固定在内层圈上，层层嵌套。

当旋转最外层圈时，所有的三个圈以及物体都会跟着全部旋转。

当旋转中层圈时，外圈不动，内圈和物体跟着旋转。

当旋转内层圈时，只有物体和内层圈旋转。

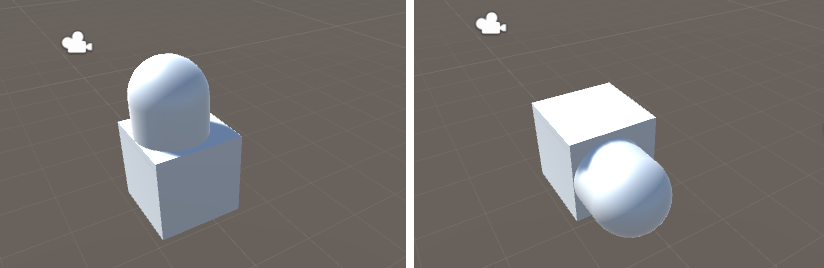
简单来说，主要问题发生在中层圈旋转90度的时候。



如图，中层圈旋转90度时，内层圈与中层圈重合了。这时候再旋转外层或内层，你会发现外层和内层控制的方向是相同或相反的（内层和外层都在控制同一个轴的旋转），也就是说，内外两个轴不再能独立控制物体的旋转。

#### Unity中的万向节锁定

Unity中欧拉角的设置和上图类似，X、Y、Z轴也存在顺序性。其中，Y轴是外层轴，X轴是中层，Z轴是内层。



如图，在Unity中，随意创建一个容易看清方向的物体。先将该物体X轴旋转改为90度，就进入了万向节锁定状态。这时在检视窗口中直接修改Y轴和Z轴的数值，会发现无论修改Y的值还是Z的值，旋转都是沿同一个轴进行的。

在场景中使用旋转工具就不会发生上述问题，因为Unity内部是使用四元数系统的，旋转工具不会有万向节锁定问题。

### 四元数

四元数可以用来表示物体的旋转和朝向。四元数内部包含了四个数字（通常用x、y、z、w来表示），但是，这四个数字并不代表直观上旋转的角度，我们在使用时也不应该直接读取或单独修改x、y、z和w的值。四元数有着完整的数学定义，我们用它来表示三维控件中的旋转和朝向时，只需要了解相应的使用方法即可。

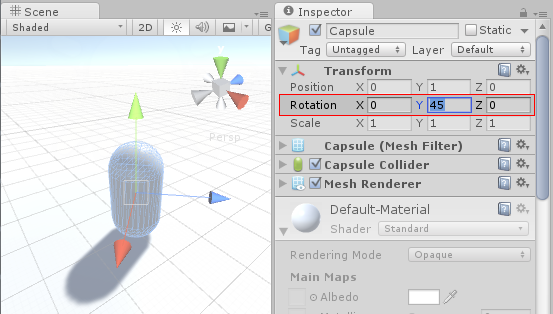
我们知道，向量可以用来表示位置和位移两种意义，四元数也同样能用来表示朝向和旋转（朝向的变化）。位置的零点是坐标轴原点，四元数的原点记作“Identity”。

优点：不存在万向节锁定问题。

缺点1：一个四元数无法表示超过180度的旋转。

缺点2：四元数的数值无法直观理解。

在Unity中，所有物体的旋转和朝向，在引擎内部都是以四元数表示的。因为相对来说四元数的优点更为重要。



（在检视窗口中，我们还是以欧拉角表示物体的旋转，因为它更容易理解和编辑。但是这个欧拉角在引擎内部还是会转换为四元数保存。）

Unity的这种设计有一个副作用，如果在检视窗口中输入一个大于180度的旋转值，例如：将某个物体的旋转值设置为(0, 365, 0)，这时运行游戏，这个值会自动变成(0, 5, 0)，这是因为四元数无法表示一个“先转360度再转5度”的朝向，而是会简单表示“5度”这个结果。

### 直接使用四元数

在脚本中使用四元数时，应当使用Quaternion类以及它提供的众多方法来创建和修改四元数。某些方法会使用欧拉角作为参数（欧拉角通常用Vector3表示），但是，应当尽可能在程序中以四元数来记录和运算旋转相关的信息，这可以最大程度上避免欧拉角带来的问题。欧拉角可以通过Quaternion.Eular方法转换为四元数，。

Unity’s Quaternion class has a number of functions which allow you to create and manipulate rotations without needing to use Euler angles at all. For example:

Quaternion类包含许多方法，这些方法可以实现创建四元数、运算、转换等功能。例如：

创建四元数：

* Quaternion.LookRotation
* Quaternion.AngleAxis
* Quaternion.FromToRotation

相关操作：

* Quaternion.Slerp
* Quaternion.Inverse
* Quaternion.RotateTowards
* Transform.Rotate

当确实使用欧拉角比较方便时，也可以在局部使用欧拉角。但是要避免将代表一个物体朝向的四元数转为欧拉角，然后修改后再转换回去。类似这种不必要的操作会带来各种问题。最常见的使用欧拉角的例子，比如将一个物体绕Y轴转30度：

|  |
| --- |
| transform.Rotate(0,30,0); |

下面再展示一些错误的例子。该脚本的目的是让物体沿X轴每秒转10度，我们来看看常见的错误写法。

|  |
| --- |
| // 常见错误1：不应当直接修改四元数的值，rot.x的值并不是绕x旋转的角度  void Update () {    var rot = transform.rotation;  rot.x += Time.deltaTime \* 10;  transform.rotation = rot;  }  // 常见错误2：将朝向转为欧拉角，修改后再转换回去。  // 将四元数转为欧拉角时，并不像直观看上去那么简单。差异不大的朝向，所转换出的欧拉角可能会有很大差异。且会在转换中遇到万向节锁定的问题、  void Update () {  var angles = transform.rotation.eulerAngles;  angles.x += Time.deltaTime \* 10;  transform.rotation = Quaternion.Euler(angles);  } |

接下来给出一个相对正确的写法：

|  |
| --- |
| // 以下代码避免读取当前四元数的值，直接通过连续指定最终角度来让物体旋转。  float x;  void Update () {  x += Time.deltaTime \* 10;  transform.rotation = Quaternion.Euler(x,0,0);  } |

### 在动画中表示旋转

Unity的动画窗口（Animation Window）中，允许你使用欧拉角来制作旋转动画。

动画的旋转用欧拉角表示的话，很容易超出180度的范围。例如，假设一个物体原地旋转720度，用欧拉角表示就是(0, 720, 0)，而用四元数表示就非常困难。

#### 动画窗口中的操作

Unity的动画窗口提供了一个选项，可以在四元数或者欧拉角之间切换。四元数或者欧拉角在描述动画时有非常大的差异。使用欧拉角，表示要完全按照指定的角度来进行旋转；而使用四元数，则表示只关心最终旋转到的角度，旋转的过程是按照最短路径来运动的。

本书的动画相关章节会详细讨论动画窗口中相关的操作。

#### 外部的动画资源

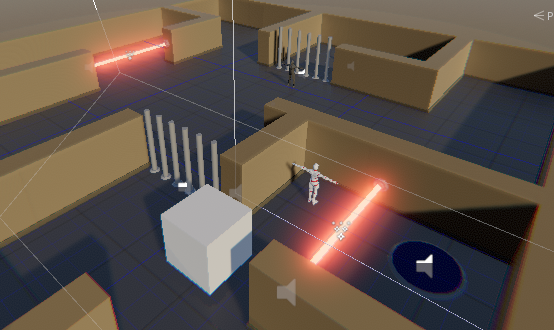
当从外部资源导入做好的动画时，这些动画通常都带有用欧拉角表示的旋转关键帧。Unity默认会重新计算这个动画，在必要时插入一些新的以四元数表示的关键帧，以尽量防止出现超出四元数旋转范围的情况。

例如，考虑这样一个动画：首尾是两个关键帧，中间间隔6帧。首帧指定旋转为0度，尾帧指定旋转为270度。如果直接导入这样一个动画而不做处理，那么结果是反向旋转90度，因为反向旋转90度才是最近的一个旋转路线。但是动画系统为了避免这个问题，会自动添加很多关键帧，以保证能按照动画最初的设计，正向旋转270度。

当Unity自动调整导入的动画时，可能会出现一些精度方面的问题。所以Unity提供了关闭自动调整的方法。这样就可以继续用欧拉角表示原始的动画了。

## 灯光

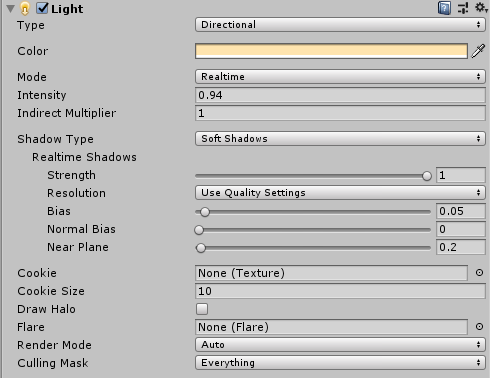
灯光奠定了每一个场景的基调。在游戏中，模型和贴图定义了场景的骨架和外表，灯光则定义了场景的色调和情感。很多时候我们会用到多于一个的灯光，同时调节多个灯光、让它们协同工作可能需要练习和反复尝试，但是最终可以得到一个非常棒的效果。



（用侧光源表现出的场景）

添加灯光的方法和创建方块类似，在层级窗口中添加Light > Directional Light即可，灯光也分很多类型，Directional Light是方向光源，最适合作为室外的整体光源。灯光本质上是一个组件，所以灯光的移动、旋转等操作方法和其它物体并没有区别。甚至还可以把灯光组件（Light Component）直接添加到游戏物体上，灯光组件位于Rendering分类中。

下面介绍灯光组件的设置参数。



只要稍微改动灯光的颜色，就可以得到完全不同的场景氛围。偏黄、红色的光源显得温暖，暗绿色的光源显得潮湿阴暗。

### 渲染路径

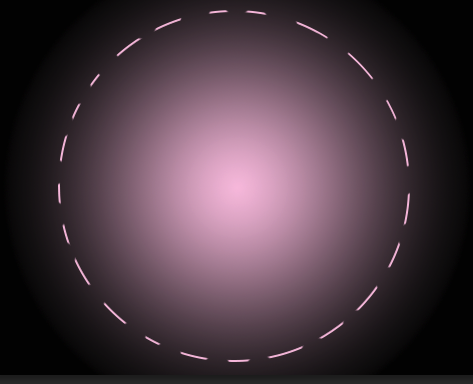
Unity支持不同的渲染路径（Rendering Paths）。不同的渲染路径主要影响的是光照和阴影，所以选择哪种渲染路径主要取决于所要做的游戏本身的需求，选择合适的渲染路径有助于提高游戏的性能。更多相关信息请参考相关章节。

### 灯光的种类

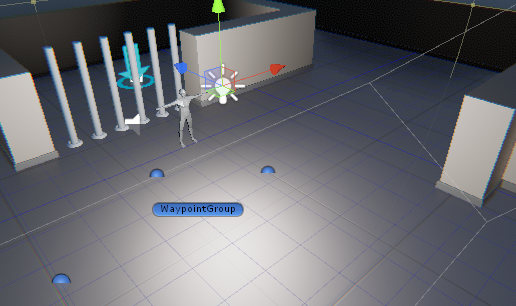
本节将介绍各种类型的灯光，在合适的地方使用合适的灯光，再配合阴影效果，可以极大提升游戏的表现力。

#### 点光源（Point Light）

点光源是空间中的一个点，向每个方向都发射同样强度的光。在只有一个点光源的场景中，直接照射到某个物体的一条光线，一定是从点光源中心发射到被照射位置的。光线的强度会随着距离的增加而减弱，到了某个距离就会减小为0。光照强度与距离的平方成反比，这被称为平方反比定律，与真实世界中的光照规律相符合。



点光源非常适合用来模拟场景中的灯泡或蜡烛等具有特定位置的光源。而且还能用于模拟枪械发射时的照亮的效果，这点具体来说：一般开枪时枪口闪光的效果是用粒子实现的，但是枪口的火焰会在瞬间照亮周围的环境，这时就可以用一个短时间出现的点光源来模拟这个效果，让开枪的效果更为逼真。



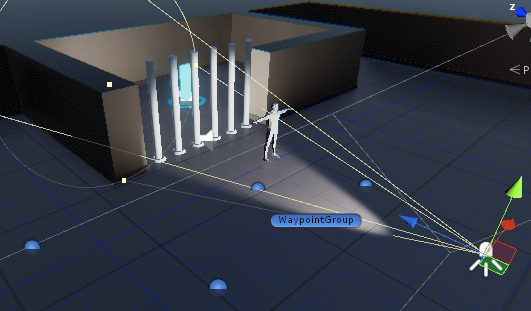
（点光源用在场景中的效果）

#### 探照灯（Spot Light）

探照灯可以类比点光源，它也具有固定的位置以及强度根据距离减弱的规律。最主要的区别是，探照灯的发射角度是被限制在一个固定的角度内的，最终形成了一个锥形（蛋筒状）的照射区域。锥形的开口默认指向该光源所在游戏物体的Z轴方向（前方）。

探照灯发射的光线会在锥形侧边缘处截止。扩大发射角度可以增大锥形的范围，但是会影响到光线的汇聚效果，和现实中的探照灯或手电筒是一致的。

探照灯通常用来表现一些特定的人造光源，例如手电筒、汽车大灯或者直升机上的探照灯。在脚本中控制物体的旋转，就可以控制探照灯的方向。试想：在一个黑暗的环境中，有一只探照灯一边左右查看一边慢慢前进。这样就可以营造出一种引人入胜的效果，或是一种恐怖的感觉。



（探照灯在场景中的效果）

#### 方向光（Directional Light）

方向光非常常用，在默认的场景中就有一个方向光源。绝大多数场景都需要阳光来提供基本的亮度，就算是夜晚的场景也需要一个类似月光的照明效果。和现实中的太阳光非常相似，方向光并没有发射源位置，也就是说在场景中，方向光所在的位置并不会对效果有任何影响。所以方向光可以放在任何位置，但是它的旋转角度非常重要。对方向光来说，所有的物体都会被一致方向的光照射到，光照的强度也不会减弱，与距离完全无关。

由于太阳离我们非常非常遥远，所以太阳照到地面上的光线，可以认为就是平行的。这就是方向光模拟太阳光的原理。

可以认为，方向光代表着遥远而又巨大的光源对当前场景的影响，这个光源非常遥远以至于可以认为它在游戏世界之外，就像是太阳和月亮。在游戏世界中，使用方向光可以带来非常有说服力的阴影效果，虽然没有指定光线具体从哪里来，但看起来和现实世界非常符合。

每个新建的场景默认都有一个方向光。在Unity 5.0之后的版本中，这个默认的方向光会和天空盒有关联，相关设置在全局灯光窗口中（Lighting > Scene > Skybox）。天空盒的颜色以及默认的“太阳”贴图位置都会和方向光绑定，实现非常逼真的场景。这些设置，包括太严的素材、绑定的光源、天空盒都是可以修改的。

通过倾斜方向光源，可以让方向光接近平行于地面，营造出一种日出或日落的效果。如果让方向光向斜上方照射，不仅整体环境会暗下来，天空盒也会暗下来，就和晚上一样。而当方向光向下照射时，天空盒也会变得明亮，就像又回到了白天。通过修改天空盒的设置，或者方向光的颜色，可以给整体环境笼罩上不同的色彩。

#### 区域光源（Area Light）

区域光源在空间中是一个矩形。光线从矩形的表面均匀向四周发射，但是只光线只会来自矩形的一面，而不会出现在另一面。区域光源不提供光照范围的选项，但是因为光线强度是受平方反比定律约束的，最终光照范围还是会被光照强度所控制。由于区域光源所带来的计算量比较大，引擎不允许实时计算区域光源，而只允许烘焙到光照贴图中。

和点光源不同，区域光源会从一个面上发射光线到物体上，也就是说照射到物体的光线同时来自许许多多不同的点、不同的方向，所以得到的光照效果会非常柔和。用区域光源可以营造出一条非常漂亮的充满灯光的街道，或是以柔和的光线照亮游戏世界。使用一个较小的区域光照，可以用来照亮一个较小的区域（例如一个房间），但是得到的效果相比点光源，更接近真实世界的效果。

#### 发光材质（Emissive Materials）

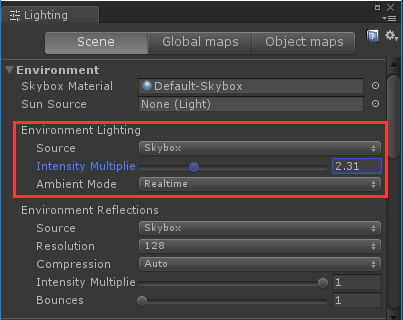
与区域光源类似，发光材质也可以从物体的表面发射出光线。它们会发射出散射式的光线到场景中，引起场景中其它物体颜色和亮度的变化。前面说到区域光照不支持实时渲染，相对的，发光材质支持实时计算。

在默认的着色器（Standard Shader）中，有一项“Emission”可以用来设置发光材质。默认此项不勾选，也就是该材质不会发光。勾选此项后，就可以指定发光的贴图、颜色、发光强度等等。注意：这种方式发射的光线只会影响到场景中的静态物体。

#### 环境光（Ambient Light）

环境光是一种特殊的“光源”，它会对整个场景带来照明，但这个光照不来自于任何一个具体的光源。它为整个场景增加基础的亮度，影响整体的效果。在很多情况下环境光都是必要的，具体来说和场景的美术风格相关。一个典型的例子是明亮的卡通风格的场景，这种场景中要避免浓重的阴影，甚至很多影子也是手绘到场景中的，所以用环境光来代替普通的灯光会更合适。有时我们需要整体提高场景的亮度包括阴影处的亮度，也可以用环境光来实现。

和其它类型的灯光不同，环境光不属于组件。它可以在光照窗口（Lighting）的Scene > Environment Lighting一栏中进行调节。



（灯光窗口。默认环境光是以天空盒作为基础，并可以在此基础上调节亮度）

### 灯光设置详解

灯光决定了物体的着色效果，以及物体的阴影效果。因此，灯光和摄像机一样都是图像渲染中非常基础的部分。

#### 属性介绍

| **属性** | **功能** |
| --- | --- |
| Type | 灯光类型，可能的类型有方向光、点光源、探照灯和区域光源。 |
| Range | 指定光线照射的最远距离。只有某些光源有这个属性。 |
| Spot Angle | 探照灯光源的照射角度大小。 |
| Color | 指定光的基本颜色。 |
| Mode | 灯光的渲染模式。有Realtime、Mixed和Baked三种选项，分别是实时光照、混合光照和预先烘培。 |
| Intensity | 调节光照的强度。 |
| Indirect Multiplier | 反射系数。反射（间接光照）就是从物体表面反射的光线，反射系数会影响反射光衰减的比例，一般这个值小于1，代表随着反射次数增加，光线强度越来越低。但也可以取大于1的值，让每次反射光线都会变强，这种方法用于一些特殊的情况，例如需要看清阴影处的细节的时候。另外可以将此值设置为0，只有直射光没有反射光，表现一些非常特殊的环境（例如恐怖氛围）。 |
| Shadow Type | 设置光线产生硬的阴影（Hard Shadows）、软的阴影（Soft Shadows）或是没有阴影（No Shadows）。 |
| Baked Shadow Angle | 当对方向光源选择软的阴影时，这个选项用来柔化阴影的边界以获得更自然的效果。 |
| Baked Shadow Radius | 当对点光源或探照灯光源选择软阴影时，这个选项用来柔化阴影的边界以获得更自然的效果。 |
| Realtime Shadows | 这选择硬阴影或软阴影时，这一项的几个属性用来控制实时阴影的效果。 |
| Strength | 用滑动条控制阴影的黑暗程度，取值范围在0~1之间，默认为1。 |
| Resolution | 控制阴影的解析度，较高的解析度让阴影边缘更准确，但是需要消耗更多GPU和内存资源。 |
| Bias | 用滑动条来调整阴影离开物体的偏移量，取值范围从0~2，默认为0.5。这个选项常用来避免错误的自阴影（self-shadowing）问题。 |
| Normal Bias | 用滑动条来让产生阴影的表面沿法线方向收缩。取值范围为0~3。这个选项也用来避免错误的自阴影（self-shadowing）问题。 |
| Near Plane | 这个选项用来调节最近的产生阴影的平面，取值范围在0.1到10之间。它的值和灯光的距离相关，是一个比例。默认值为0.2。 |
| Cookie | 指定一张贴图，来模拟灯光投射过网格状物体的效果。（例如灯光投射过格子状的窗户以后，呈现出窗格的阴影） |
| Draw Halo | 灯光光晕，由于灯光附近灰尘、气体的影响而让光源附近出现一个团状区域。  另外Unity还提供了专门的光晕组件，可以和灯光的光晕同时使用。 |
| Flare | 镜头光晕，和灯光光晕不同，是模拟摄像机镜头内光线反射而出现的一种效果。这个选项中可以指定镜头光晕的贴图。 |
| Render Mode | 使用下拉菜单设置本灯光渲染的优先级，这会影响到灯光的真实度和性能。 |
| Auto | 运行时自动确定。和品质设置（Quality Settings）有关。 |
| Important | 总是以像素级精度渲染，由于性能消耗更大，适用于屏幕中特别显眼的地方。 |
| Not Important | 总是以较快的方式渲染。 |
| Culling Mask | 剔除遮罩。用来指定只有某些层级会被这个灯光所影响。 |

#### 细节提示

如果使用带有透明通道的材质用来作为灯光的Cookie，那么该Cookie的透明度会影响光线的亮度，可以让亮度连续变化。这可以很好地增加场景的复杂度与气氛。

绝大部分内置的着色器都可以与每种光源协同工作。但是顶点光照着色器无法实现Cookie与阴影。

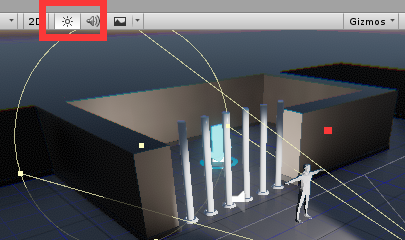
所有的灯光都可以选择性地产生阴影。要做到这点可以通过调节每种灯光的阴影选项来实现。

#### 提示

* 带有Cookie的探照灯，特别适合用来表现一束探照灯穿过窗户照进房间的效果。
* 低强度的点光源适合用来表现场景的深度、层次感。
* 使用顶点光照着色器可以极致提升性能。这种着色器只为每个顶点计算光照，在低端设备上可以实现非常好的性能。

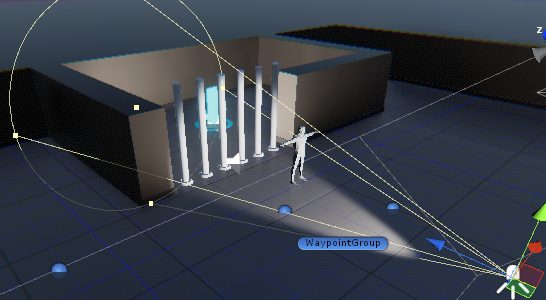
### 使用灯光

创建并放置光源的方法，和创建一个立方体并没有什么区别。例如可以通过在检视窗口中点击右键 Create > Light > Directional Light即可创建一个方向光源。在选中一个灯光物体时，可以看到它的辅助框线，不同的灯光有不同的框线。



（在场景视图窗口中可以开启和关闭光照效果，图中的太阳图标即是开关按钮）

前面介绍过，方向光的位置不重要（除非使用了Cookie的情况），角度很重要；修改点光源、探照灯光源的位置和方向都可以在场景中立即看到效果。另外这些光源的辅助框线也清晰地展示了光源影响的范围。



（探照灯以及辅助框线，注意圆锥形的黄色线条）

#### 放置光源的技巧

方向光源通常用来表现日光下的效果。一般日光的方向是斜向下方的，如果用垂直地面照射的光，就会显得很死板。例如一个角色跳入场景的时候，如果方向光源是正射而不是有一定角度的话，立体感和表现力就会差很多。

探照灯和点光源通常用来表现人造光源。我们在使用时往往会发现刚开始将它们加入场景时，往往看不到什么效果，只有将光线的范围调整到合适的比例时，才能看到明显变化。当探照灯只是射向地面时，只能感受到一个锥形的照亮范围。只有当探照灯前有一个角色或者物体经过时，才会体会到探照灯特有的效果。

灯光具有默认的光照强度和颜色（白色），适用于大多数正常的场景。但是当你想要个性化的场景氛围时，调整它们可以立即得到完全不同的效果。例如一个闪耀着绿光的力场，将周围的物体照亮成绿色；汽车的大灯带有一些黄色而不是纯白。在更有想象力的场景中需要更有想象力的灯光——例如在遥远的另一个星球上，有着红色的太阳。

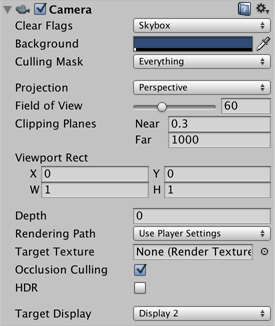
## 摄像机

摄像机本身是拍摄电影的工具。3D游戏中引入了摄像机的概念，作为场景空间与最终屏幕展示之间的媒介。游戏运行时，场景中需要至少有一个摄像机，也可以有多个摄像机。多个摄像机可以用来同时显示场景中两个不同的部分（比如双人分屏游戏），也可以用来制作一些高级的游戏效果。

在Unity中，摄像机作为一种组件，是被挂载到游戏物体上的，所以甚至可以用动画或者物理来控制摄像机。实际上，如何操作摄像机只受限于你的想象力，只要能够提高游戏的表现力，传统或者创新的方式都是值得尝试的。

### 属性介绍

摄像机用来“拍摄”场景，并将它展现给玩家。通过调节摄像机的属性，改变摄像机的运动方式，只凭这点就足以实现非常独特的效果。Unity对于摄像机的数量不做限制，且可以为这些摄像机设定不同的优先级、拍摄场景中不同的位置。甚至还可以用来实现某些极为特殊的功能，例如透视、小地图、无人机、双人同屏等等。



（摄像机组件的截图）

| 属性 | 功能 |
| --- | --- |
| Clear Flags | 清除标记。用来指定屏幕中未绘制部分如何处理。当使用多个摄像机时这个选项非常必要，下面会详细解释。 |
| Background | 场景中没有物体、也没有天空盒的区域，会以这项指定的背景颜色显示。 |
| Culling Mask | 剔除遮罩。可以用来指定某些层不被渲染。 |
| Projection | 切换透视摄像机或正交摄像机。 |
| Perspective | 默认的透视摄像机模式。 |
| Orthographic | 正交摄像机模式。注意：延迟渲染（Deferred rendering）在正交模式下不可用，此模式下总是使用前向渲染（Forward rendering）。 |
| Size | 当使用正交模式时，指定摄像机拍摄的范围大小。 |
| Field of view | 当使用透视摄像机时，指定视野的角度大小。 |
| Clipping Planes | 剪切面。指定摄像机渲染的距离范围，过近或过远的物体都不被渲染。 |
| Near | 靠近摄像机的那个剪切面的距离。 |
| Far | 远离摄像机的那个剪切面的距离。 |
| Viewport Rect | 视图矩形。四个值用来确定摄像机拍摄到的画面显示到屏幕的哪个位置、以及显示的大小。这四个值是标准化的值，取值范围从0到1，按比例计算。 |
| X | 摄像机画面输出到屏幕上的起始点X轴坐标。 |
| Y | 摄像机画面输出到屏幕上的起始点Y轴坐标。 |
| W (Width) | 摄像机画面输出到屏幕上的宽度。 |
| H (Height) | 摄像机画面输出到屏幕上的高度。 |
| Depth | 深度。决定了该摄像机在绘制顺序中的序号。较大深度的摄像机画面会稍后绘制，所以会覆盖较低深度的。 |
| Rendering Path | 渲染路径。Unity提供了多种不同的渲染路径，影响到了光照、阴影等渲染问题。 |
| Use Player Settings | 使用播放器设置。渲染路径以播放器中的设置为准。 |
| Vertex Lit | 顶点光照方式。 |
| Forward | 前向渲染方式。每个物体的每个材质所受光照的影响，都会被计算一遍。 |
| Deferred Lighting | 延迟渲染。先在不考虑光照的前提下渲染每一个物体。然后再统一计算每个像素的光照。 |
| Target Texture | 目标贴图。摄像机默认渲染到屏幕上，但是设置此选项以后，就会渲染到一张贴图上去。这在制作小地图等特殊效果时非常有用。 |
| HDR | 让这个摄像机开启/关闭高动态范围度功能。 |
| Target Display | 显示目标。指定该摄像机渲染到哪个外部设备上，可选从1至8。 |

### 细节

摄像机是将游戏画面呈现给玩家的基础。摄像机可以被设置、调整，也可以用脚本来控制，甚至还可以作为子物体被挂载到其它父物体下面，用法非常灵活。对于桌面类游戏来说，可能只要一个静态的全景摄像机就足够了。而对于主视角游戏来说，最简单的方法是将摄像机挂载到玩家角色身上，高度设置为眼睛的高度。对于赛车游戏来说，你可能希望摄像机保持在车辆的后面某个位置。

可以创建多个摄像机，并将它们设置为不同的深度（Depth）。摄像机画面会从较低深度开始，逐步向高一级绘制。举个例子，深度值为2的摄像机画面会覆盖在深度值为1的摄像机上面。

可以通过设置视图矩形（View Port）来设定摄像机画面显示到屏幕上的位置和大小。这样就可以创建多个“画中画”的效果，例如小地图、无人机拍摄画面、后视镜等等。

### 渲染路径

Unity支持几种不同的渲染路径，可以根据游戏类型与目标发布平台进行选择。不同的渲染路径会带来不同的渲染效果，以及不同的性能损耗，它们影响的主要是场景中灯光与阴影的表现。默认渲染路径是在播放器设置中统一配置的，也可以为每个摄像机设定不同的渲染路径。

### 清除标记

每个摄像机保存着各自的颜色和深度信息。没有物体可渲染的部分就是空白区域，默认空白区域会渲染为天空盒。当使用多个摄像机的时候，每个摄像机都保存着自己的颜色与深度信息，这些信息是可以重叠的。可以为每一个相机设置不同的清除标记，可以达到同时显示两层画面的效果，具体的设置有如下四种：

### 天空盒（Skybox）

天空盒是摄像机清除标记的默认设置。空的区域（没有东西可显示的区域），会显示为天空盒。默认这个天空盒以“光照窗口”（Lighting Window，在主菜单的Window > Lighting选项中打开）中指定的天空盒为准。

可以为每个摄像机添加不同的天空盒，可以尝试在摄像机上添加专门的天空盒组件（Skybox component）。

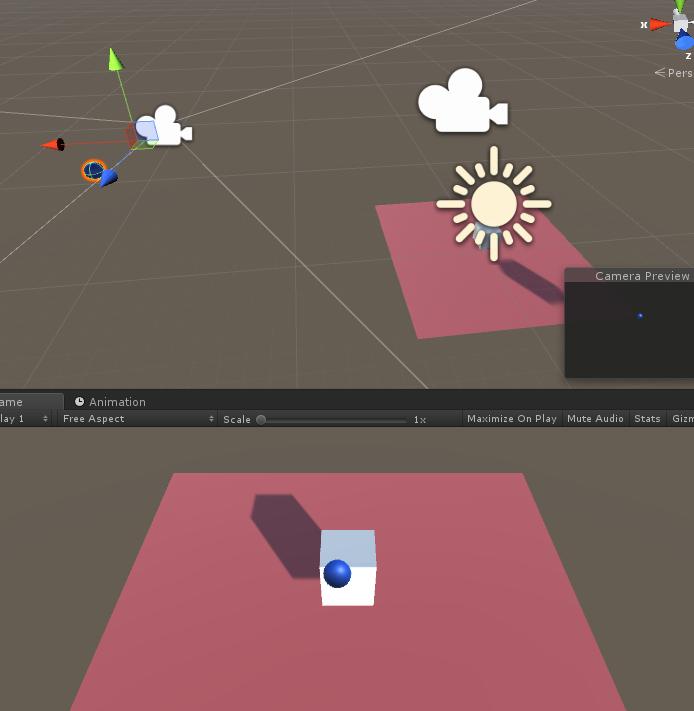
### 纯色（Solid Color）

空白区域以纯色方式显示，该颜色在摄像机的背景色（Background Color）中指定。

### 仅深度（Depth Only）

仅深度（Depth Only）这一选项可以用于混合两个摄像机所看到的画面。由于摄像机的深度为多个摄像机区分了先后绘制顺序，那么后一个摄像机在绘制时，就可以保留前一个摄像机的画面，但却清除之前所有的深度信息。这样一来，后一个摄像机所拍摄的画面就会叠加到之前的画面上，但是绝对不会被遮挡（因为之前的深度信息已经清除了）。

这个功能看似高级，但却经常被使用——它被用来制作主视角射击游戏中主角手持的枪。FPS的主角持枪靠近墙的时候，枪的模型很容易穿进墙面（常被称作模型穿墙问题）。而这时如果把枪放在远离场景的位置，然后用另一个摄像机单独渲染枪支，再把枪支叠加到游戏画面中，这样做枪支就绝对不会被任何物体遮挡住了。

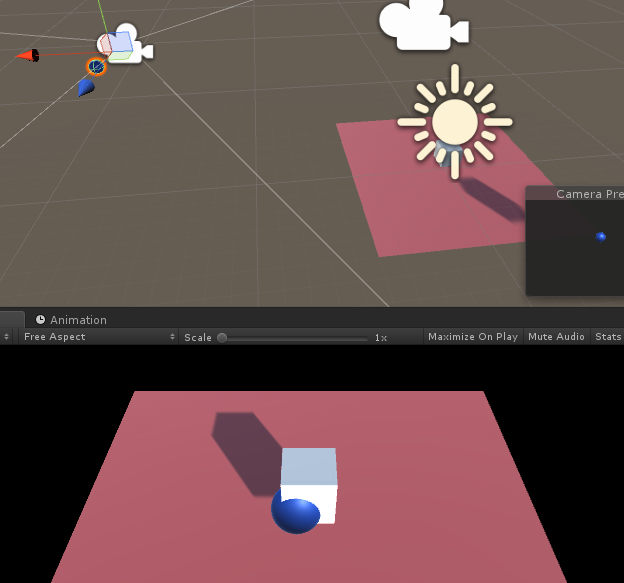


如上图中，场景中只有红色的地面与白色的方块。蓝色的球放在很远的地方，用一个单独的摄像机渲染这个球。单独的摄像机深度值较大，且设置为“仅深度”。这样蓝色的球就会叠加到场景中，且蓝色的球无论远近，都不会被遮挡住。

### 不清除（Don't Clear）

这个模式既不清除之前渲染的画面，也不清除深度信息。结果每个摄像机看到的画面都被直接混合起来。结果会造成一种比较混乱的显示方式。这种模式在游戏中很少使用，在自定义着色器的情况下可能有用武之地。

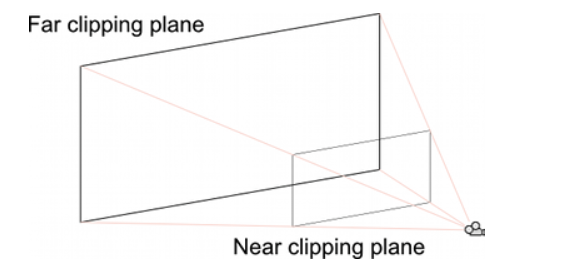
注意在某些GPU硬件上（特别是很多移动平台的GPU），这个模式可能会带来不可预计的后果，可能会导致两个摄像机画面混乱叠加，也可能会出现随机颜色的像素。



如上图，场景的布置和之前一样，将“仅深度”改为“不清除”，则两个摄像机的深度信息会叠加起来。两个摄像机所看到的不同物体，在叠加时仍然会有前后遮挡关系。

### 剪切面

摄像机所拍摄的范围，实际上是一个金字塔的形状，被称为“视椎体（Camera Frustum）”。由于我们不需要渲染特别远处的物体，所以实际上需要拍摄的物体可以被限制在一个很有限的范围。



如图，这个范围可以简单用两个平面来表示，离摄像机较近的平面称之为近剪切面（Near clipping plane），较远的叫做远剪切面（Far Clipping plane）。这两个平面截取了视椎体的一部分，我们只需要“拍摄”和渲染中间的这一部分物体即可。将远剪切面移到更远处，就可以看到更多远处的细节；拉近远剪切面，就可以减少渲染的工作，有助于提高游戏性能。

完全位于视椎体之外的物体不需要被渲染，这一特性被称为“视椎体裁剪”（Frustum Culling），视椎体裁剪是3D游戏引擎的最基本功能之一，不需要被关闭也不需要配置。注意不要和其它功能搞混。

更进一步，为了深度优化游戏，可能希望不同层级的物体具有不同的裁剪距离，例如很小的物体只有很近处才能看到，大型的建筑在很远的地方也会被看到。Unity提供了相关的功能，但是只能在脚本中进行操作，方法名为Camera.layerCullDistance，详见相关文档。

### 剔除遮罩（Culling Mask）

剔除遮罩用来指定只渲染某一个层级的物体。

摄像机的这个选项是和游戏物体的层级（Layer）配合使用的，详见本书中讲解游戏物体的层级的章节。

### 视图矩形（Viewport Rectangle）

标准化视图矩形（Normalized Viewport Rectangle），用来指定摄像机所拍摄的内容固定显示在屏幕的某一个矩形的范围内。例如可以将一个小地图视图放在屏幕右下角的位置，或者一个无人机视图放在左上角位置。通过配置视图矩形，可以实现非常特别的界面效果。

用视图矩形来实现双人分屏游戏也非常简单。步骤为：

1. 创建两个摄像机，分别显示两个玩家各自的画面。
2. 两个摄像机视图矩形的值都设置为0.5。
3. 第一个摄像机的视图矩形值设置为0，第二个值为0.5。

只需要简单几步操作，就让两个摄像机分别显示到屏幕中不同的区域，实现了分屏效果。



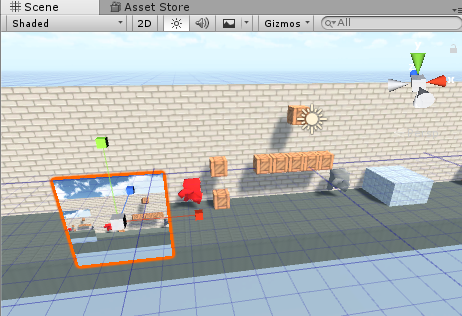
（用Unity很容易做到这种分屏效果，同样的方法还可以做成四分屏）

### 正交摄像机（Orthographic）

关于正交摄像机的例子和图片，已经在本书的第一章中有过详细介绍，此处略。

### 渲染贴图（Render Texture）

这个选项可以让摄像机所拍摄到的画面渲染到一张贴图上，这张贴图可以别应用在另一个物体上。这个功能让我们很容易做出游戏中的镜子、显示屏、小地图、监视器等等。发挥想象力还能做出其它非常有趣的效果（比如可以让玩家跳进屏幕里）。



（使用摄像机配合Render Texture，很容易实现画中画效果。）

### 显示目标（Target Display）

现代的个人电脑支持多显示器，某些游戏可以在三个屏幕上拼接显示游戏内容。摄像机可以指定所要渲染的目标的序号，可以最多选择8个显示设备其中之一。这个功能只在PC、Mac和Linux这些桌面系统上有效。

### 其它提示

* 摄像机所在的游戏物体可以被实例化、被作为子物体，也可以用脚本控制，和其它游戏物体一样。
* 使用较大或较小的视野范围，可以用于表现不同的场景。
* 如果在摄像机物体添加刚体，也可以让摄像机受物理引擎的控制。
* 场景中摄像机的数量不受限制，只需要考虑性能。
* 正交摄像机非常适合用来表示3D用户界面。
* 如果由于两个物体表面非常接近而产生显示问题，试着尽可能加大近剪切面。
* 摄像机无法同时渲染到屏幕上和一张贴图上，一次只能选择一个。
* Unity官方提供了一个控制摄像机运动的专业插件——Cinemachine，可以用于实现各种各样的游戏类型。详见官方文档。

## 开始做游戏吧

到目前为止，你已经学习了Unity的界面和基本操作，以及如何使用资源、怎样创建场景等等，最后还有如何发布游戏。实际上，现在已经没有什么能阻碍你做出自己想要的游戏了，一路上我们学习了很多东西，也许很多概念理解的不深，随着实践的深入，你可以再回头看看这些基本概念，参考更多的资料，以增强自己的游戏开发技能。

切记，在了解了最基本的概念以后，我们随时都可以开发游戏。千万不要等一切准备就绪才开始，那一天可能永远不会到来。