

目錄

| | |
|-----------------|---------------|
| 简介 | 1.1 |
| Unity 手册 | 1.2 |
| 图形 | 1.3 |
| 图形概述 | 1.3.1 |
| 光照 | 1.3.1.1 |
| 光照概述 | 1.3.1.1.1 |
| 光源 | 1.3.1.1.2 |
| 灯光类型 | 1.3.1.1.2.1 |
| 灯光检视视图 | 1.3.1.1.2.2 |
| 使用灯光 | 1.3.1.1.2.3 |
| 灯光纹理 | 1.3.1.1.2.4 |
| 阴影 | 1.3.1.1.3 |
| 阴影 | 1.3.1.1.3.1 |
| 全局光照 | 1.3.1.1.4 |
| 摄像机 | 1.3.1.2 |
| 材质、着色器、纹理 | 1.3.1.3 |
| 创建和使用材质 | 1.3.1.3.1 |
| 标准着色器 | 1.3.1.3.2 |
| 内容和上下文 | 1.3.1.3.2.1 |
| 工作流程：金属 vs 镜面 | 1.3.1.3.2.2 |
| 材质参数 | 1.3.1.3.2.3 |
| 渲染模式 | 1.3.1.3.2.3.1 |
| 漫反射颜色和透明度 | 1.3.1.3.2.3.2 |
| 金属模式：金属参数 | 1.3.1.3.2.3.3 |
| 平滑度 | 1.3.1.3.2.3.4 |
| 法线贴图（凹凸贴图） | 1.3.1.3.2.3.5 |
| 高度图 | 1.3.1.3.2.3.6 |
| 散射贴图 | 1.3.1.3.2.3.7 |
| 自发光 | 1.3.1.3.2.3.8 |
| 辅助贴图（细节贴图）和细节蒙板 | 1.3.1.3.2.3.9 |

| | |
|---------------|----------------|
| 菲涅耳效应 | 1.3.1.3.2.3.10 |
| 材质图表 | 1.3.1.3.2.4 |
| 粒子系统 | 1.3.1.4 |
| 什么是粒子系统？ | 1.3.1.4.1 |
| 使用粒子系统 | 1.3.1.4.2 |
| 粒子系统入门 | 1.3.1.4.3 |
| 简单的爆炸 | 1.3.1.4.3.1 |
| 载具尾气 | 1.3.1.4.3.2 |
| 屏幕特效概述 | 1.3.1.5 |
| 图层 | 1.3.1.6 |
| 物理系统 | 1.4 |
| 物理系统概述 | 1.4.1 |
| 刚体概述 | 1.4.1.1 |
| 碰撞器 | 1.4.1.2 |
| 连接 | 1.4.1.3 |
| 角色控制器 | 1.4.1.4 |
| 脚本 | 1.5 |
| 脚本概述 | 1.5.1 |
| 创建和使用脚本 | 1.5.1.1 |
| 变量和检视视图 | 1.5.1.2 |
| 使用组件控制游戏对象 | 1.5.1.3 |
| 事件函数 | 1.5.1.4 |
| 时间和帧数控制 | 1.5.1.5 |
| 创建和销毁游戏对象 | 1.5.1.6 |
| 协同程序 | 1.5.1.7 |
| 特殊文件夹和脚本的编译顺序 | 1.5.1.8 |
| 事件函数执行顺序 | 1.5.1.9 |
| 理解自动内存管理 | 1.5.1.10 |
| 泛型函数 | 1.5.1.11 |
| 音频 | 1.6 |
| 音频概述 | 1.6.1 |
| 动画 | 1.7 |
| 动画系统概述 | 1.7.1 |
| 动画剪辑 | 1.7.2 |

| | |
|-----------|-----------|
| 动画视图指南 | 1.7.2.1 |
| 使用动画视图 | 1.7.2.1.1 |
| 创建动画剪辑 | 1.7.2.1.2 |
| 让游戏对象动起来 | 1.7.2.1.3 |
| 使用动画曲线 | 1.7.2.1.4 |
| 编辑动画曲线 | 1.7.2.1.5 |
| 动画控制器 | 1.7.3 |
| 动画控制器资源 | 1.7.3.1 |
| 动画控制器视图 | 1.7.3.2 |
| 动画状态机 | 1.7.3.3 |
| 用户界面 UI | 1.8 |
| 画布 Canvas | 1.8.1 |
| 基本布局 | 1.8.2 |
| 视觉组件 | 1.8.3 |
| 交互组件 | 1.8.4 |

Unity 5.5 手册（中文版）

<https://nuysoft.gitbooks.io/unity-manual/content/>

<https://github.com/nuysoft/Unity-Manual>



Unity 手册

Unity 编辑器允许你创建 2D 和 3D 游戏、应用程序。这本 Unity 手册将帮助你学习如何使用 Unity 编辑和相关的服务。你也可以从头到尾阅读手册，或者把它当作参考。

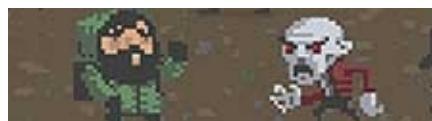
有关 5.5 最新特定的文档请阅读 [What's New in 5.5](#)。

有关从旧版本升级 Unity 工程的信息，请阅读 [Upgrade Guide](#)。

Unity 手册目录



[Unity 简介](#) 有关 Unity 软件的完整介绍。



[Unity 2D](#) Unity 2D 所有功能，包括运行、Sprite 和物理系统。



[图形系统](#) Unity 的图形部分，包括摄像机和灯光。



[物理引擎](#) Unity 的物理引擎，包括 3D 空间中刚体的使用和修改。



[联机](#) 如何实现多人联机。

```
13- void Update ()  
14- {  
15-     RaycastHit hit;  
16-     Ray landingRay = new Ray(transform.position, Vector3.  
17-     Debug.DrawRay(transform.position, Vector3.down * depth);  
18-     if(!deployed && !landed)  
19-     {  
20-         if(hit.distance <= depth)  
21-             landed = true;  
22-     }  
23- }
```

[脚本](#) 在 Unity 中使用脚本编程。



音频系统 Unity 的音频系统，包括剪辑、源文件、监听者、导入和声音设置。



动画系统 Unity 的动画系统。



用户界面 Unity 的用户界面系统



导航系统 Unity 的导航系统，包括人工只能和路径寻找。



Unity 服务 Unity 提供的服务，用来制作和改善游戏。



虚拟现实 VR 整合虚拟现实 VR。



为 Unity 贡献代码 对 Unity 源代码的修改建议。



平台规范 Web 平台和非桌面平台的规范信息。



遗留内容 用于维护老项目。

更多信息源

更多指南请查看：

- Unity Answers or Unity Forums - here you can ask questions and search answers.
- The Unity Knowledge Base - a collection of answers to questions posed to Unity's Support teams.
- Tutorials - step by step video and written guides to using the Unity Editor.
- Unity Ads Knowledge Base - a guide to including ads in your game.
- Everyplay documentation - a guide to the Everyplay mobile game replay platform.
- Asset Store help - help on Asset Store content sharing.

已知问题

某个功能是否不再支持？这可能是一个已存在的问题。请查阅 issuetracker.unity3d.com 的 **Issue Tracker**。

图形



Unity 提供了惊人的视觉保真度、渲染力和临场感。

Unity 为你的游戏提供符合直觉的实时全局光照和基于物理的着色器。从白天，到晚上霓虹灯的绚丽辉光；从光晕，到昏暗的午夜街道和阴暗的隧道 — 支持在任意平台上创建令人着迷和回味无穷的游戏。

本章将介绍光照、摄像机、材质、着色、纹理、粒子效果和视觉效果等内容。

另外，请参阅 [图形知识库](#)。

在教程部分还有许多有用的图形教程。

相关教程：[图形](#)

有关故障排查、提示和技巧的内容，请参阅 [图形知识库](#)。

图形概述

理解图形系统是深入游戏开发的关键。本章详细介绍 Unity 的图形特性，例如光照和渲染。

光照

本节详细介绍 Unity 提供的高级光照功能。

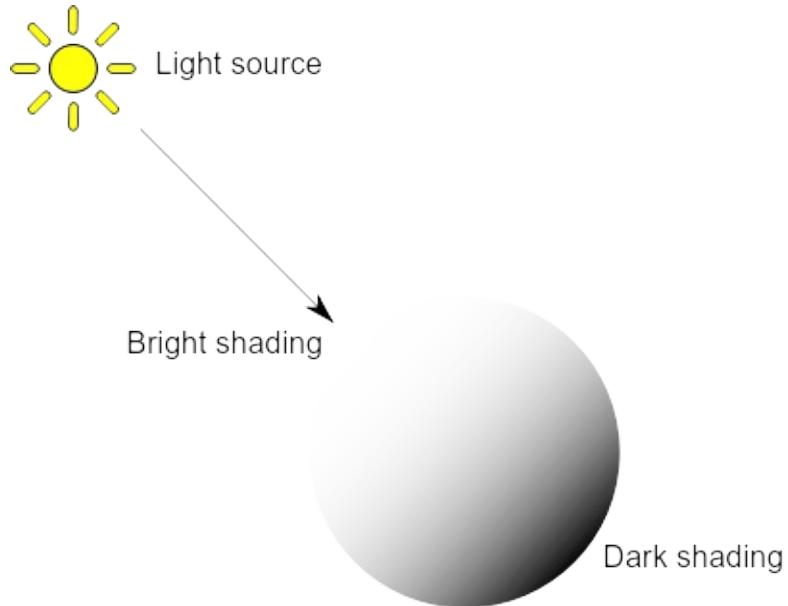
相关的介绍，请参阅 [灯光 手册](#) 和 [灯光组件](#) 的参考页。

教程部分还提供了 [光照教程](#)。

对于特殊问题，请尝试搜索 [知识库](#) 或在 [论坛](#) 提问。

光照概述

为了计算 3D 对象的着色，Unity 需要知道落在 3D 对象的光的强度、方向和颜色。



这些属性由场景中的灯光对象提供。不同类型的灯光以不同的方式发射为它们分配的颜色；某些光可以随着与光源的距离而衰减，并且接受的光线角度也有不同的规则。Unity 提供的各种光源在 [灯光类型](#) 中有详细说明。

Unity 以多种不同的方式计算复杂的、高级的光照效果，每种方式对应不同的场景。

选择光照方案

Unity 中的光照可以粗略地分为『实时』和『预算』，并且，两种光照可以组合使用，以创建沉浸式的场景光照。

在本节中，我们将简要概述不同技术的适用场景、相对优势，以及独特的性能因素。

实时光照

默认情况下，Unity 中的灯光——平行光、聚光灯和点光源——是实时的。这意味着，它们为场景提供直接光照，并且每帧更新。如果灯光和游戏对象在场景中移动，光照将立即更新。在场景视图和游戏视图中都可以观察这种变化。

注意，因为没有弹射光，所以阴影是完全黑色的。只有位于聚光灯椎体内的表面才会收到影响。

译注：原文有错误，这里应该有一张截图，展示一个聚光灯的实时效果。

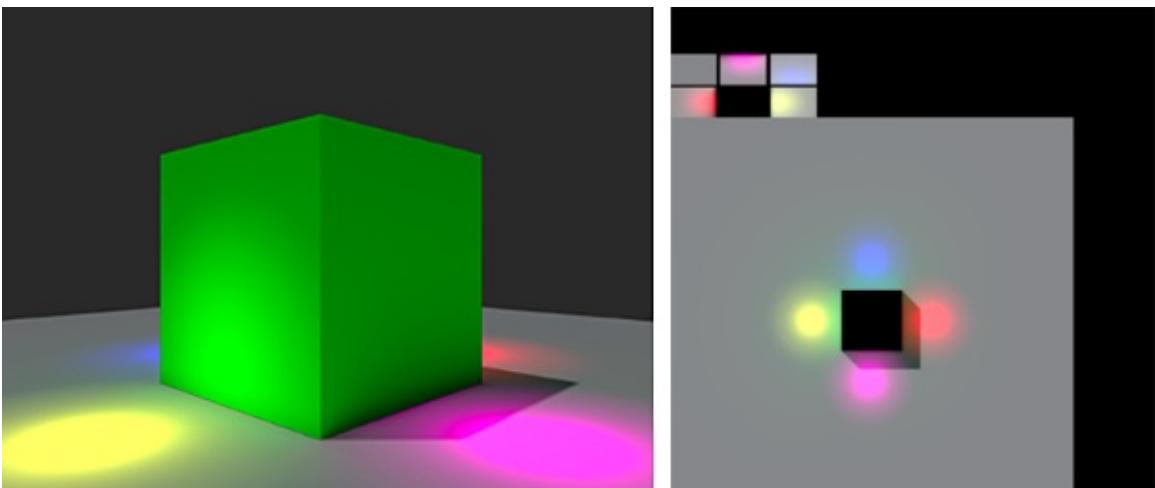
实时光照是照亮场景中物体的最基本方式，可以用于照亮角色或其他可移动的几何物体。

不幸的是，在 Unity 中，来自实时光的光线不会被弹射。为了使用诸如全局光照等技术，创建更加逼真的场景，我们需要使用 Unity 的预计算光照方案。

烘焙光照贴图

Unity 可以计算复杂的静态光照效果（使用称为全局光照或 GI 的技术），并将它们存储在一张称为光照贴图的纹理文件中。这个计算过程成为『烘焙』。

烘焙光照贴图时，场景中静态对象上的光源效果被计算，并将计算结果写入覆盖了场景几何顶点的纹理文件中，以产生光照效果。



左图：一个简单的光照贴图场景。右图：Unity 生成的光照贴图纹理。注意是如何捕获阴影和灯光信息的。

光照贴图可以包含撞击到表面的直接光和场景中其他对象或表面弹射的间接光。光照贴图可以和表面信息结合使用，例如对象的材质着色器指定的颜色（漫反射）和浮雕（法线）。

使用烘焙照明时，光照贴图在游戏期间无法改变，因为被称为是『静态』的。实时灯光可以叠加在光照贴图之上，但是无法改变光照贴图本身。

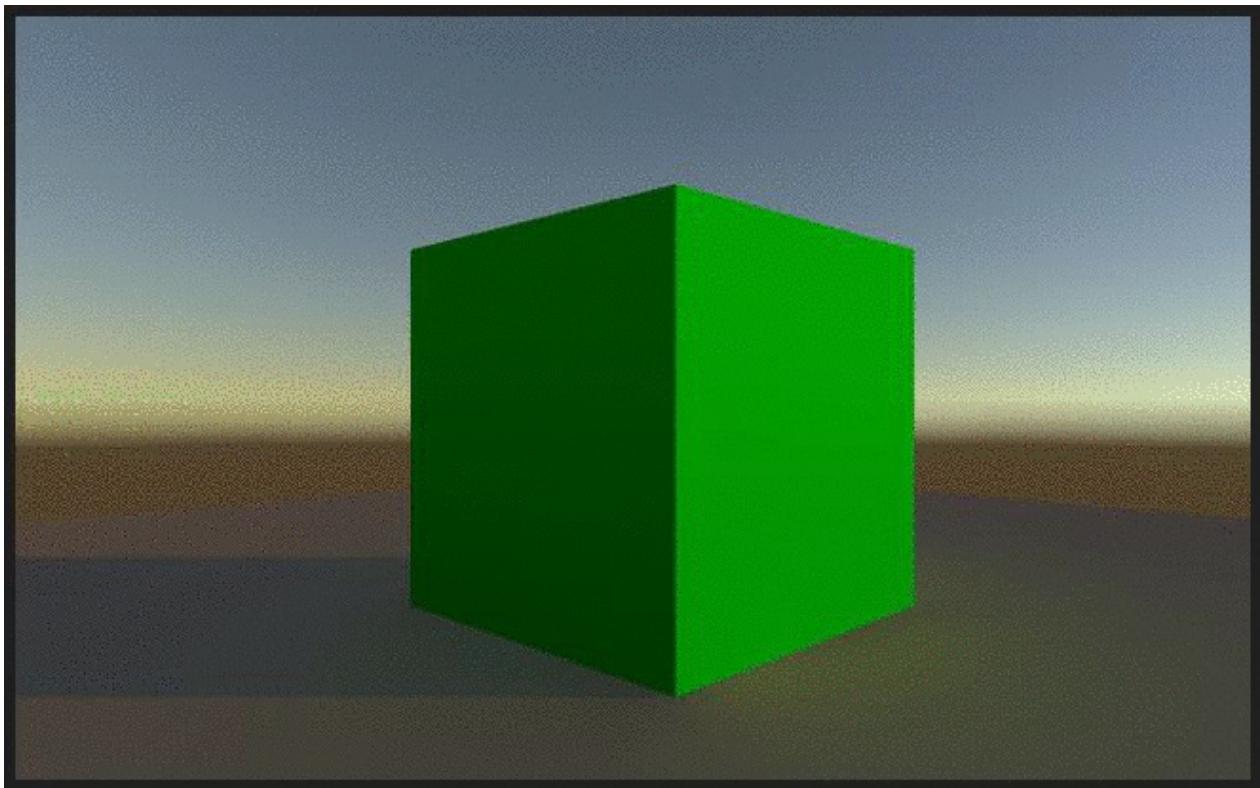
当在游戏过程中移动灯光时，使用这种方法可以潜在地提升性能，以适配性能不佳的硬件，例如移动平台。

更多信息请参阅 [光照视图参考页](#) 和 [使用预计算光照](#)。

预计算的实时 GI

虽然静态光照贴图不能对场景中光照条件的变化做出响应，但是，预计算实时 GI 技术可以支持交互式地更新复杂场景的光照。

使用这种技术，可以创建具有丰富全局光照的场景，并且可以实时地响应弹射光和光照变化。一个很好的例子是昼夜系统——光源的位置和颜色随着时间变化。而使用传统的烘培光照技术是不可能实现的。



简单的昼夜示例，使用了预计算实时 GI。

为了在可玩帧率下实现这些效果，我们需要将一些冗长的数字解调从实时处理转移到预计算过程中。

预计算把游戏中计算复杂光照明的过程，转移到不重要的时间段进行处理。我们称之为『离线』处理。

更多信息请参阅 [关照和渲染教程](#)。

利与弊

尽管可以同时使用烘培 GI 和预计算实时 GI，但要注意，同时渲染两个系统的性能成本恰好是它们的总和。我们不仅需要在显存中存储两种光照贴图，还需要支付着色器解码它们的成本。

选择和放弃哪种光照方案，取决于项目性质和硬件性能预期。例如，移动设备的显存和处理能力有限，烘培 GI 方案可能更加高效。在具有专用图形硬件的独立计算机上，或最新的游戏机上，很可能使用预算算实时 GI 方案，甚至同时使用两种方案。

关于采用哪种方案，必须基于特定项目的性质和目标平台进行评估。请记住，当定位于（需要支持）一系列不同的硬件时，通常性能最差的硬件能将决定需要哪种方案。

另请参阅 [灯光故障排除和性能](#)。

光源

Unity 中的光照主要由灯光对象提供。还有两种其他方式可以产生光（环境光和自发光材质），这取决于你选择的光照方案。

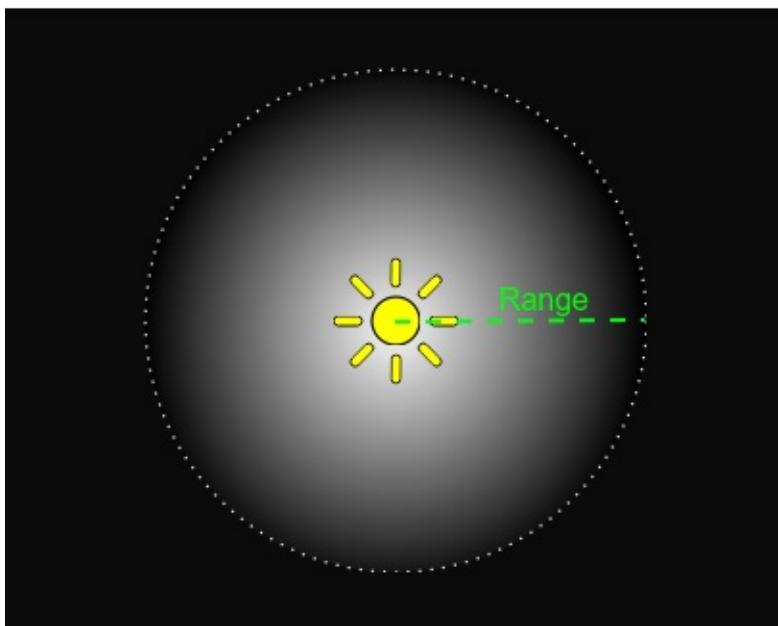
下面几节详细介绍了 Unity 中创建灯光的各种方法。

灯光类型

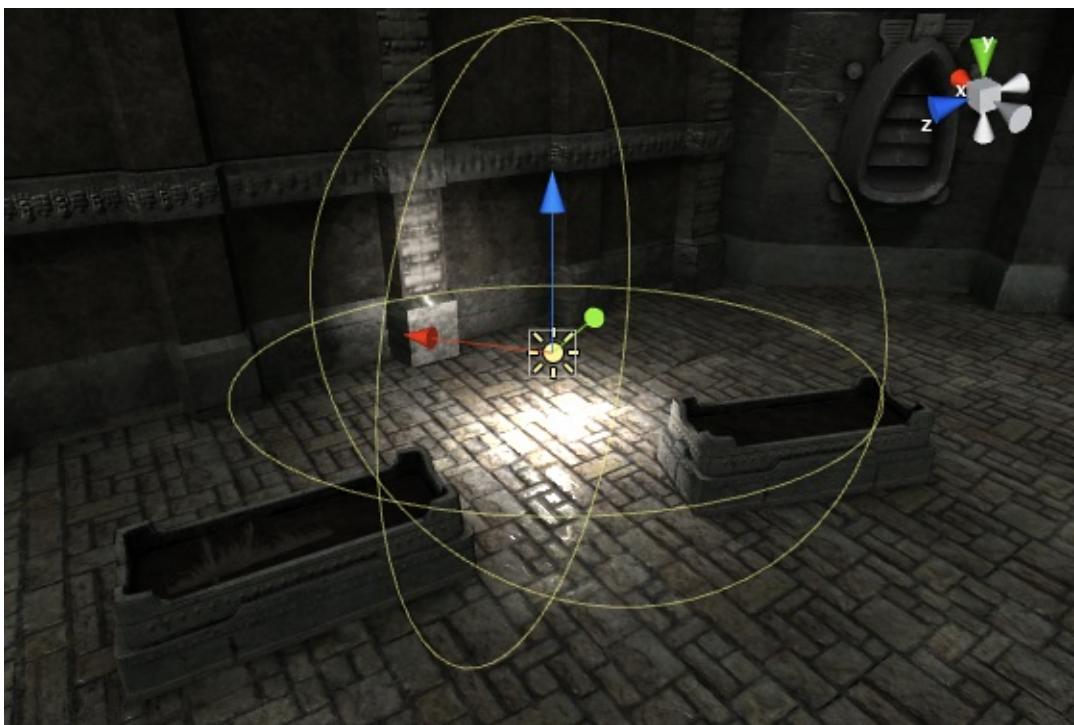
本节详细介绍 Unity 中创建灯光的多种不同方法。

点光源

点光源位于空间中的某个点，均匀地向所有方向发射光。光击中表面的方向是一条从灯光对象中心到接触点的直线。强度随着与灯光的距离而衰减，在指定范围衰减为 0。光的强度与目标对象到光源的距离的平方成反比。这被称为『平方反比定律』，类似于光在真实世界中的行为。



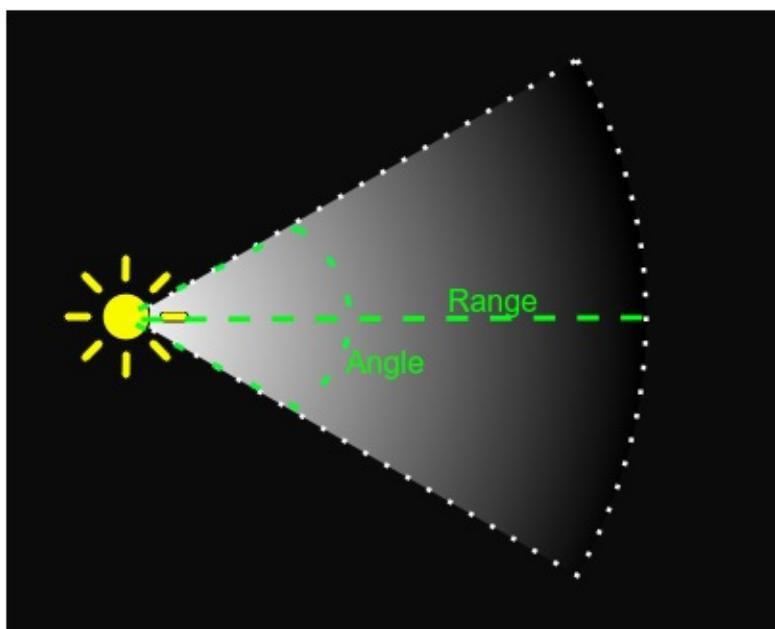
点光源可以用于模拟场景中的灯具或其他本地光源。可以使火花或爆炸以逼真的方式照亮周围的环境。



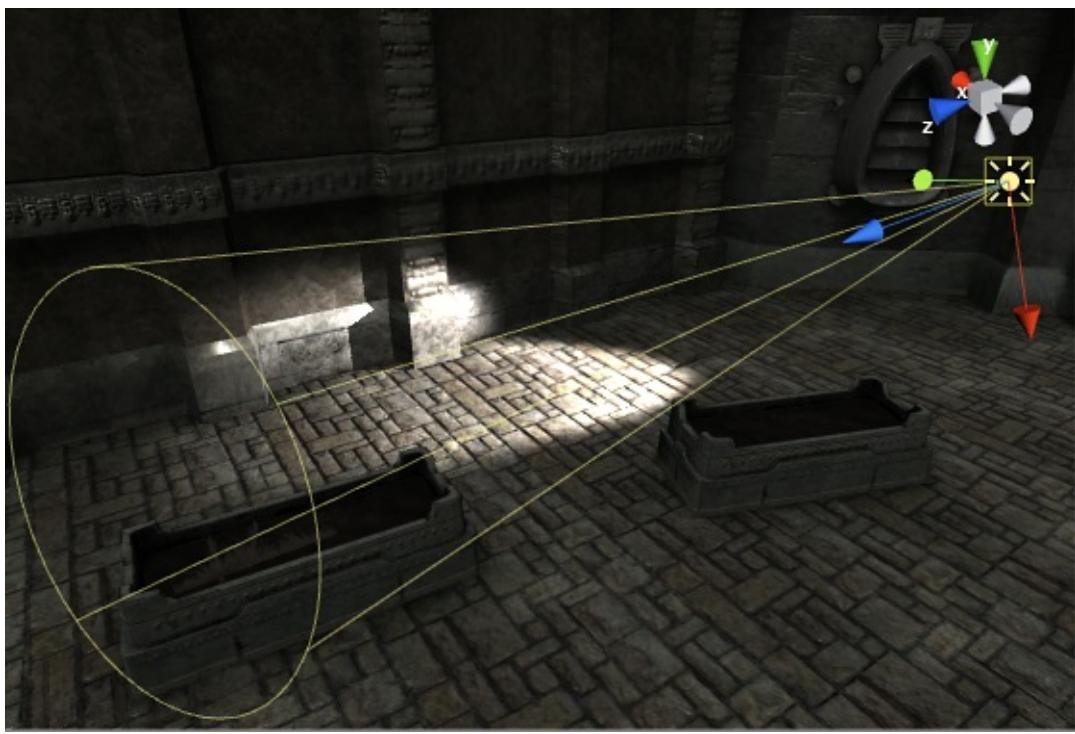
点光源在场景中的效果

聚光灯

聚光灯类似于点光源，具有特定的位置和方位，并且逐渐衰减。不过，聚光灯被限制在一定的角度范围内，产生锥形的照明区域。椎体的中心指向灯光对象的前方（Z轴）。光的强度沿着聚光灯椎体的边缘衰减。扩大角度会增加椎体的宽度，并增加渐变的幅度（即加剧衰减），被称为『半影锥』。



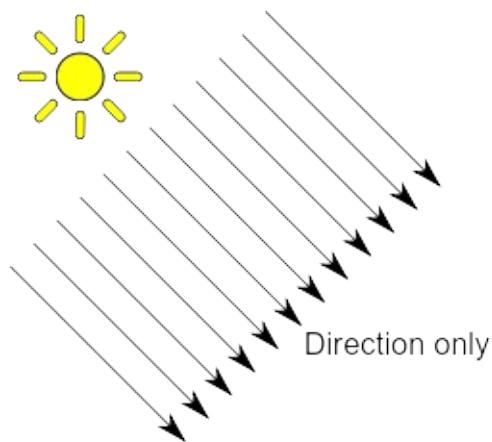
聚光灯通常用于人造光源，例如手电筒、汽车前灯和探照灯。通过用脚本或动画控制聚光灯的方向，一个移动的聚光灯将照亮场景的小块区域，产生戏剧性的照明效果。



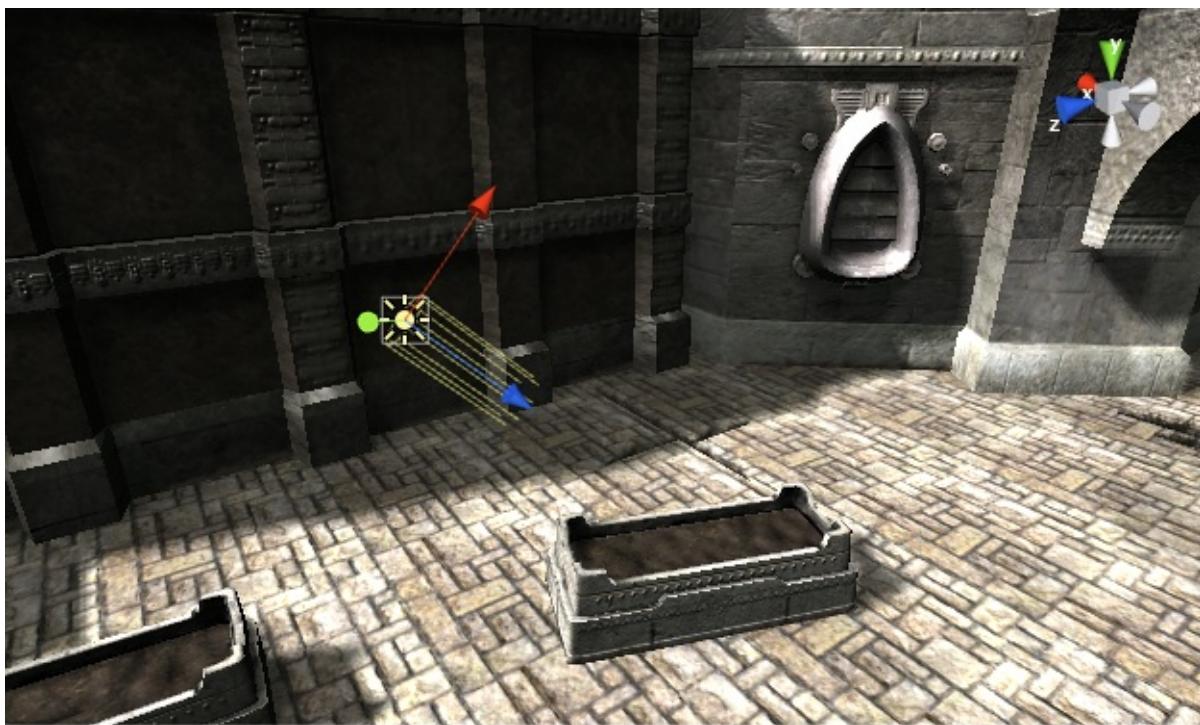
聚光灯在场景中的效果

平行光

平行光用于在场景中对于创建类似阳光的效果。在许多方面与太阳相似，可以把平行光认为是位于无限遥远距离的光源。平行光没有具体的来源，因此平行光对象可以放置在场景中的任意位置。场景中的所有对象都被照亮，就好像光总是来自相同的方向。因为无法定义从目标对象到灯光对象的距离，所以光的强度不会衰减。



平行光代表了游戏世界之外巨大而遥远的光源。在拟真场景中，它可以用来模拟太阳或月亮。在抽象的游戏世界中，它可以为对象添加逼真的阴影，而不需要精确指定光来自哪里。

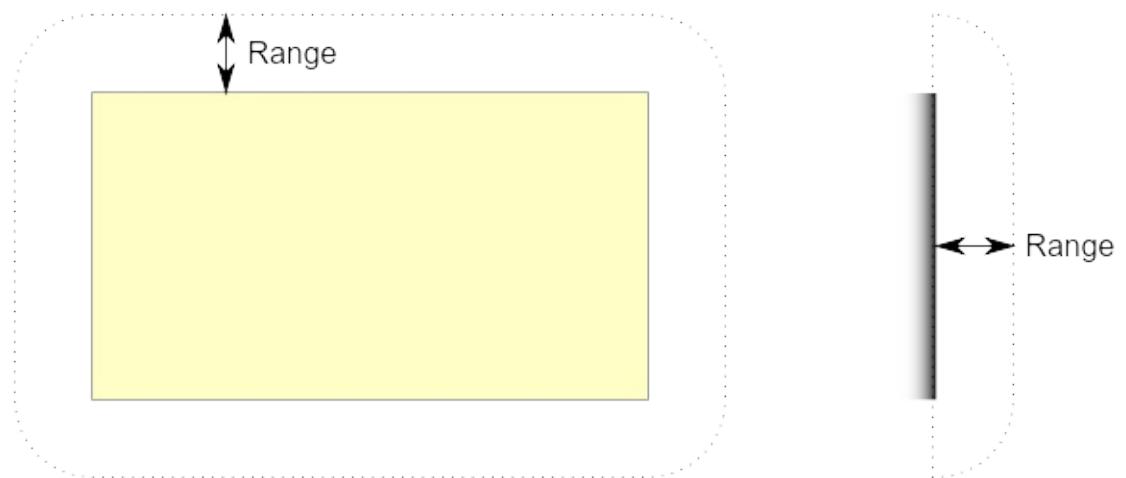


平行光在场景中的效果

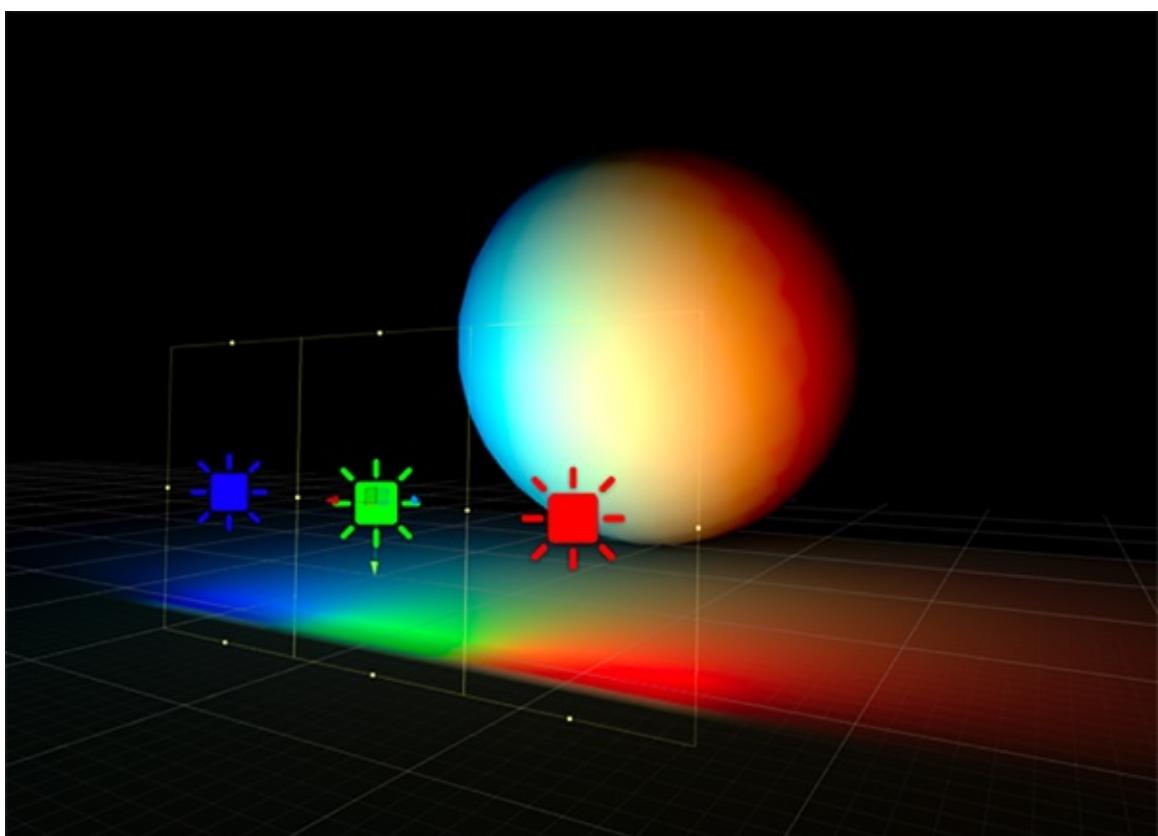
每个新建的 Unity 场景默认包含一个平行光。在 Unity 5 中，平行光与天空系统相关联，定义在光照面板的环境光照部分（Lighting>Scene>Skybox）。通过删除默认的平行光并新建一个平行光，或者为 Sun 参数（Lighting>Scene>Sun）简单地指定一个不同的游戏对象，你可以改变这种默认行为。旋转默认的平行光（或 Sun）将更新『天空盒』。调整平行光的角度，使之与地面平行，可以实现日落效果。将平行光指向上方，会使天空变黑，就像是在夜间一样。倾斜平行光，天空将类似于白天。如果天空盒被选择为环境光源，环境光照将按照平行光的颜色改变。

区域光

区域光由空间中的一个矩形定义。沿着它的表面，均匀地向所有方向发射光，但是只从矩形的一面发射。无法手动控制区域光的范围，不过，随着光线远离光源，强度将按照距离的平方衰减。因为光照计算需要大量的处理器资源，因此区域光在运行时是不可用的，只能烘培到光照贴图中。

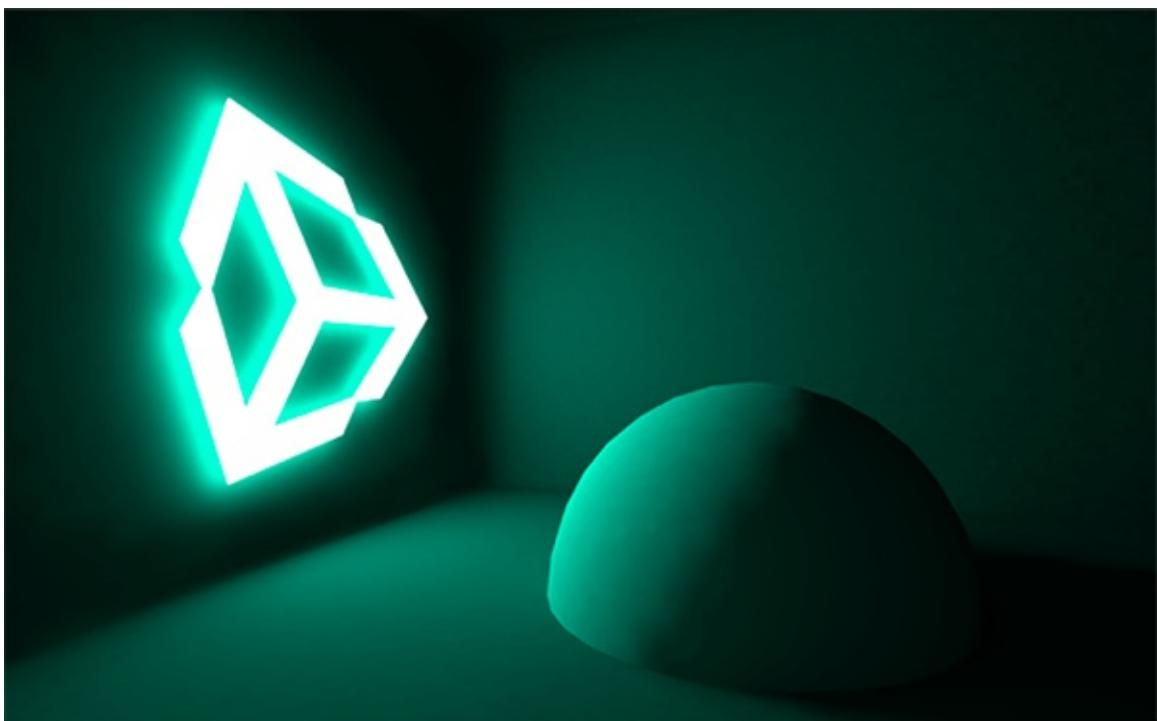


区域光同时从多个不同方向照亮物体，产生的阴影往往比其他类型的灯光更加柔和细腻。可以用它创建逼真的路灯或玩家附近的一排灯。小型区域光可以模拟更小的光源（例如室内照明），但是具有比点光源更真实的效果。



沿着区域光的表面发射光，产生具有柔和阴影的漫射光。

自发光材质



自发光材质像区域光一样沿着它的表面发射光。用于组成场景中的弹射光，并且可以在游戏中修改相关的属性，例如颜色和强度。尽管预算算实时 GI 不支持区域光，不过，使用自发光材质仍然可以实时渲染出类似的柔和光照效果。

`Emission` 是标准着色器的一个属性，允许场景中的静态对象发射光。`Emission` 默认被设置为 0。这意味着使用了该材质着色器的对象不会发射光。

自发光材质没有范围值，但是发射的光将以距离的平方衰减。发射的光只会被标记为『静态』或『静态光照贴图』的对象接收。类似地，如果自发光材质被应用到非静态对象或运动对象（例如角色）上，将不会影响环境光照。

在场景中，`Emission` 大于 0 的材质将明亮地发光，即使它并不影响环境光照。在检视视图中，为标准着色器的 `Global Illumination` 属性选择 `None`，也可以产生这样的效果。像这样的自发光材质可以用于创建诸如霓虹灯或其他可见光源等效果。

自发光材质只会直接影响场景中的静态对象。如果需要使运动的或非静态的对象（例如角色）也接收来自自发光材质的光，必须使用光照探针。

环境光

环境光是指存在于整个场景中的光，不来自于任何特定光源对象。它是场景整体外观和亮度的重要贡献者。

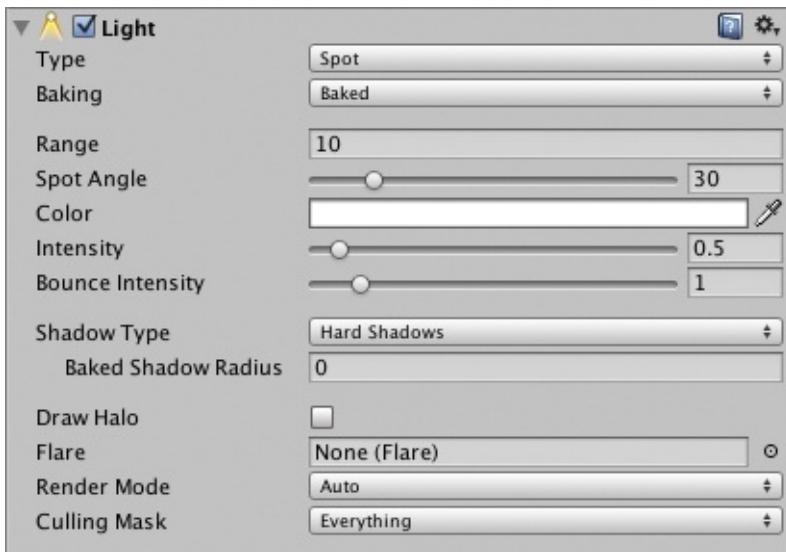
环境光在许多情况下是有用的，不过这取决于你选择的艺术风格。举个例子，明亮的、卡通风格的渲染可能不需要黑暗的阴影，或者光照可能被手绘成纹理。如果需要在不调整单个灯光的情况下增加场景的整体亮度，可以直接调整环境光。

环境光的设置可以在 [光照视图](#) 中找到。

灯光检视视图

[切换到脚本参考页](#)

灯光是图形渲染的基础组成部分，因为它们决定了对象的着色和投射的阴影。有关 Unity 中光照概念的更多详细介绍，请参阅本手册手册的 [灯光类型] 和 [全局照明] 部分。



属性

Type 灯光的当前类型。可选值有平行光 Directional、点光源 Point、聚光灯 Spot 和区域光 Area（这些类型的详细介绍请参阅 [光照](#)）。

Baking 当前灯光是否应该被烘培，如果开启了 Baked GI，选择 Mixed 仍然会烘培，并且仍然会在运行时为非静态对象提供直接光。选择 Realtime，既适用于预计算全局 GI，也适用于不使用 GI 的情况。有关光照贴图和烘培的更多信息请参阅本手册的 [全局光照](#) 部分。

Range 从灯光对象中心发出的光的传播距离（仅限点光源和聚光灯）。

Spot Angle 决定聚光灯椎体（光锥）的角度（以度为单位）。仅限聚光灯。

Color 被发射的光的颜色。

Intensity 光的亮度（强度）。对于点光源、聚光灯和区域光，默认值是 1，对于平行光，默认值是 0.5。

Bounce Intensity 改变非直接光（例如，从一个对象弹射向另一个对象的光）的强度。该值是 GI 系统计算的默认亮度的倍数；如果该值大于 1，那么弹射光将变亮，如果该值小于 1，则变暗。这个功能是有用的，例如，位于阴影中的暗表面（例如洞穴的内部）需要渲染的更

亮，以使细节可见。还有一种方案，如果想要使用预算算实时 GI，但是想要限制某个灯光为只提供直接光，你可以设置该值为 0。更多相关信息请参阅本手册的 [全局光照](#) 部分。

Shadow Type 决定灯光是否投射硬阴影（尖锐阴影）、软阴影（柔和阴影）或没有阴影。

Baked Shadow Radius 如果启用了阴影，这个属性会为点光源或聚光灯投射的阴影边缘添加一些虚化效果（理论上，源自某个点的光线投射出完美的锐利阴影，但是这种情况现在自然界中很少发生）。

Baked Shadow Angle 如果启动了阴影，这个属性会为平行光投射的阴影边缘添加一些虚化效果（理论上，来自真实平行光源的平行光线投射出完美的锐利阴影，但是自然光源的行为不严格遵循该规则）。

Draw Halo 如果选中，将绘制灯光的球形光晕，光晕半径等于 Range。另请参阅 [光晕](#) 组件。

Flare 可选。引用一个 [镜头光晕](#)，将在灯关所处位置渲染。

Render Mode 该灯光的重要性。可能会影响光照保真度和性能，请参阅下面的性能注意事项。可选值有 **Auto**（在运行时决定渲染模式，取决于附件灯关的亮度和当前的 [画质设置](#)），**Important**（该灯光始终以像素级别的质量渲染）和 **Not Important**（该灯光始终以更快的顶点光照模式渲染）。仅为最值得注意的视觉效果使用 **Important** 模式（例如，玩家所驾驶车辆的前灯）。

Culling Mask 剔除遮罩，用于有选择性地排除该灯光所能影响的对象分组；请参阅 [分层](#)。

详细介绍

你可以创建一张含有 alpha 通道的纹理，并分配给灯光的 **Cookie** 变量。这个 Cookie 将被灯光投射。Cookie 的 alpha 蒙板调节光量，在表面上创建明亮和阴暗斑点。可以非常有效地为场景添加复杂的氛围效果。

Unity 中的所有 [内置着色器](#) 都可以和任意类型的灯光无缝协作。不过，顶点光照着色器无法显示 Cookie 或阴影。

所有灯光可以选择性地投射 [阴影](#)。通过选择 **Shadow Type** 为 **Hard Shadows** 或 **Soft Shadows** 来完成。有关阴影的更多信息请阅读 [阴影](#) 页。

平行光阴影

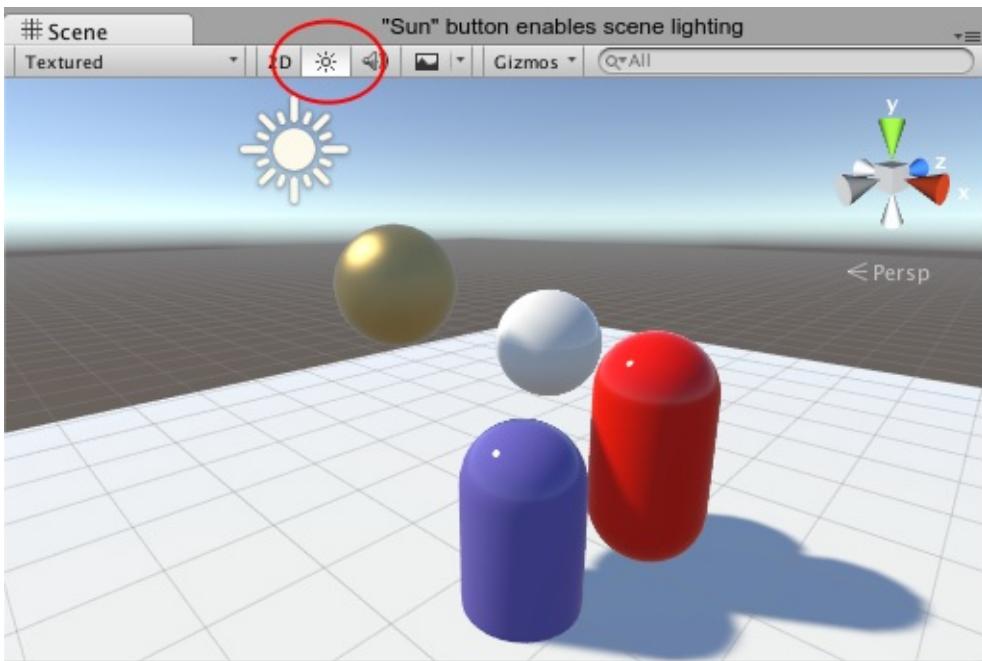
来自平行光的阴影在 [这个页面](#) 深入解释。注意，当采用正向渲染时，带有 Cookie 的平行光的阴影将被禁用。不过，在这种情况下，可以编写自定义着色器，使用 **fullforwardshadows** 标签来启用阴影；更多详情请参阅 [这个页面](#)。

提示

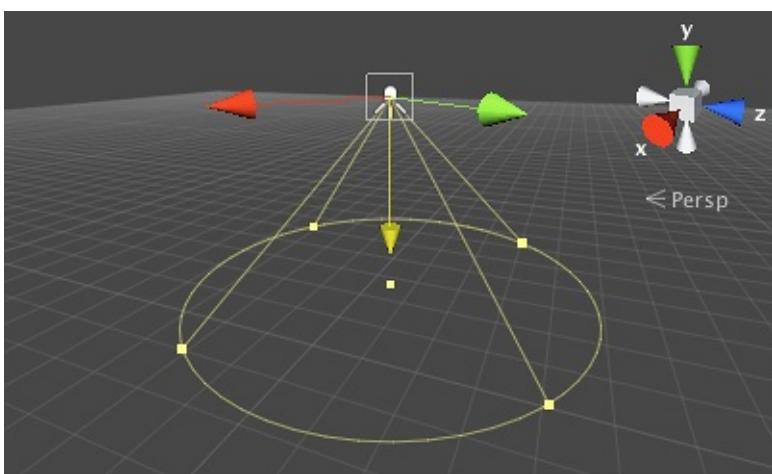
- 带有 **Cookie** 的聚光灯可以非常有效地创建窗户透射光。
- 低强度的点光源适合为场景提供深度。
- 为了获得最佳性能，请使用 [顶点光照](#) 着色器。这个着色器只会计算每个顶点的光照，为低端显卡提供更高的吞吐量。
- 重要性为 **Auto** 的灯光可以在使用了灯光贴图的对象上投射动态阴影，并且不会添加额外的光照。为了实现这点，当烘培光照贴图时，该灯光必须处于激活状态。否则，将作为实时灯光进行渲染。

使用灯光

Unity 中的灯光非常容易使用——你只需要创建一个所需类型的灯光（例如，通过菜单 **GameObject > Light > Point Light**），并将其放置在场景中合适的位置。如果开启了场景视图光照（工具栏上的『太阳』按钮），当移动灯光对象和设置它们的参数时，就可以预览光照的效果。



平行光通常可以放置在场景中的任意位置（除非使用了灯光纹理），并且使用 Z 轴作为它的方向。聚光灯也具有方向，但是范围有限，而且它的位置很重要。可以在检视视图中调整聚光灯、点光源和区域光的形状，或者，直接在场景视图中调整灯光的线框。



点光源的线框

灯具放置指南

平行光通常代表太阳，对场景的外观具有显著影响。平行光的方向应该稍微朝下，但是通常需要确保它与场景中的主要对象成微小的角度。例如，一个粗糙的立方体对象，如果光线不是正面直射到某个面上，立方体的阴影将更加有趣，在 3D 中看起来更加突出。

聚光灯和点光源通常代表人造光源，所以它们的位置通常由场景对象决定。一个常见陷阱是，当把这些灯光第一次添加到场景中时，它们似乎没有任何效果。当你调整灯光的范围以覆盖整个场景时，才会有有效果。在灯光范围的边界位置，灯光亮度衰减为 0。例如，如果把聚光灯的椎体底部设置在天花板上，那么这个灯光将很少有或没有效果，除非另一个物体从它下面经过。如果想要照亮水平物体，你需要扩大点光源和聚光灯，使它们的范围穿过墙壁和地板。

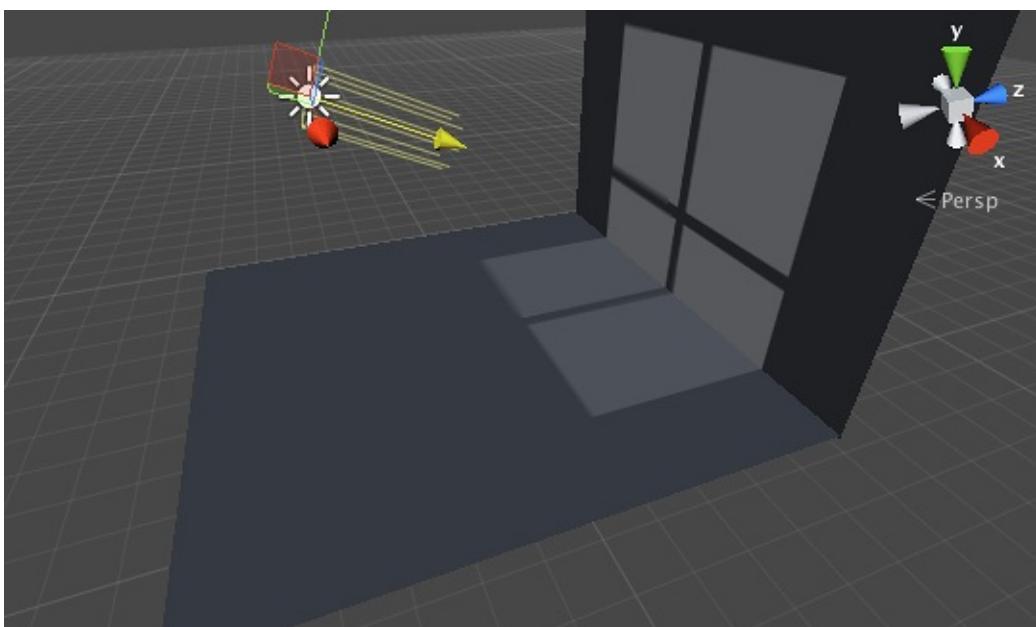
颜色和强度

灯光具有颜色和强度属性，可以在检视视图中设置。默认的强度和白色适用于普通照明，可以为对象着色，不过为了生成特殊效果，你可能需要改变这些属性。例如，一个发光的绿色力场可能非常明亮，让周围的对象沐浴在绿色光芒中；汽车前灯（特别是在旧车上）的光通常具有轻微的黄色，而不是亮白色。这些效果最常见于点光源和聚光灯，但是，如果游戏被设置在一个有红色太阳的遥远星球上，你可能会改变平行光的颜色。

灯光纹理

在戏剧和电影中，照明效果被长期用于表现不存在于场景中的物体。丛林探险家可能被覆盖在假想树冠的阴影中。监狱场景经常显示从铁栅栏窗透进来的光，即使窗户和墙壁并不真实存在于场景中。虽然非常有气氛，但是阴影的创建非常之简单，只需要在光源和目标对象之间放置某种形状的遮罩。这个遮罩简称为剪影或 Cookie。Unity 灯光支持以纹理的形式添加 Cookie，从而有效地增强场景气氛。

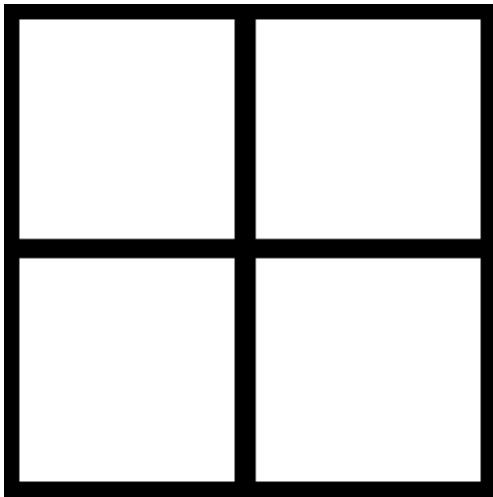
译注：剪影比 Cookie 更直观易懂，不过译者认为更合适的翻译是『灯光纹理』。因此，下文统一使用『灯光纹理』。



平行光纹理，模拟从窗户透进来的光

创建灯光纹理

灯光纹理只是一张普通的纹理，不过只和 alpha/transparency 通道有关系。当导入一张灯光纹理时，Unity 提供了将图像的亮度转换为 alpha 的选项，因此通常将灯光纹理简单地设计为一张灰度纹理。你可以使用任意图像编辑器生成一张灯光纹理，并保存到项目的 Assets 文件夹下。

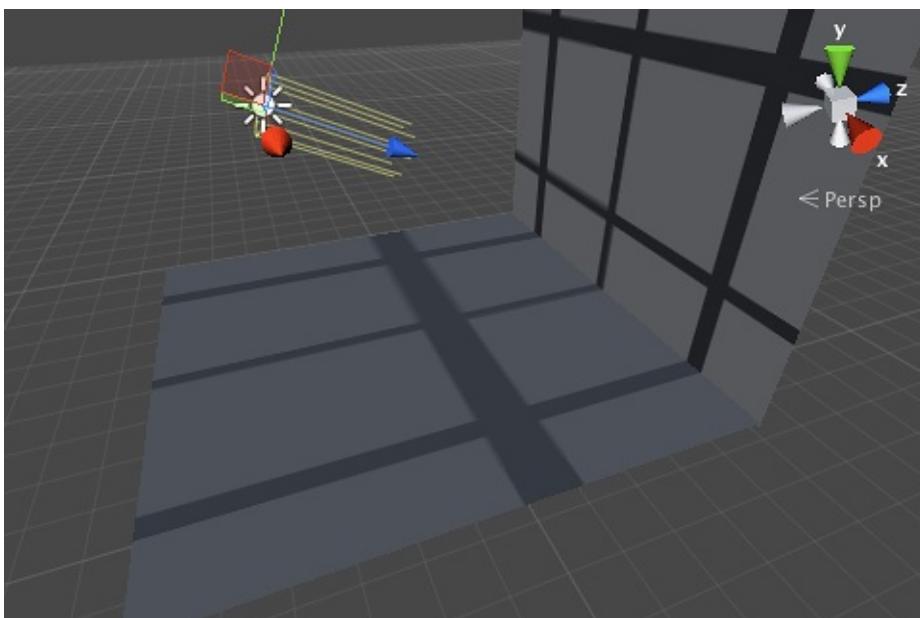


简单的窗户灯光纹理

导入灯光纹理到 Unity 中时，在项目视图中选中它，并在检视视图中把贴图类型设置为 Cookie。你还应该开启 Alpha From Grayscale，除非你已经设计好了图像的 alpha 通道。



选项 Light Type 会影响投射纹理的方式。因为点光源向所有方向发射光，所以纹理贴图必须是 Cubemap 类型。如上图所示，聚光灯使用的纹理类型应该被设置为 Spotlight，而平行光实际上可以使用 Spotlight 或 Directional 类型。使用了 Directional 纹理的平行光将在整个场景上重复平铺纹理。如果使用 Spotlight 类型，则只会在『光束』直线上显示一次；唯有在这种情况下，平行光的位置才显得如此重要。



在平行光模式下『平铺』窗口纹理

应用纹理到灯光

导入纹理后，在检视视图中，把它拖动到灯光的 **Cookie** 属性，以应用它。

聚光灯和点光源按照圆锥体或球体的尺寸简单地缩放灯光纹理。平行光额外提供了一个 **Cookie Size** 选项，允许你缩放纹理；类型为聚光灯和平行光的灯光纹理都支持缩放。

使用灯光纹理

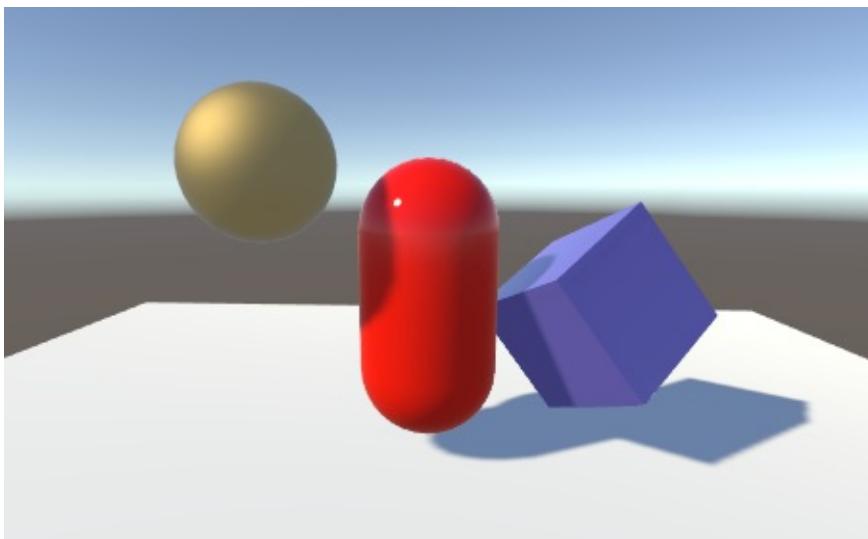
灯光纹理通常用于改变灯光的形状，使之与场景中绘制的某些细节相匹配。例如，黑暗的隧道可能具有沿着天花板的条形灯。如果使用标准的聚光灯模拟，光束将呈现为意想不到的圆形，但是可以使用灯光纹理约束光线，使之呈现为细长的矩形。正在使用监视器屏幕的角色，脸部可能会被投射绿色荧光，但是这种荧光应该被限制为小盒子形状。

注意，灯光纹理不需要完全是黑白的，它可以包含任意灰度级。这可以用于模拟光线路径上的尘埃和污垢。例如，如果游戏场景位于长期废弃的房间内，可以在窗户和灯光上使用带有噪点的、肮脏的灯光纹理，来营造气氛。类似的，汽车前灯玻璃通常含有脊线（不平整），产生还有较亮和较暗区域的焦散图案；使用灯光纹理可以很好地模拟这种效果。

译注：『焦散』是指当光线穿过一个透明物体时，由于对象表面的不平整，使得光线折射并没有平行发生，出现漫折射，投影表面出现光子分散。——[百度百科](#)

阴影

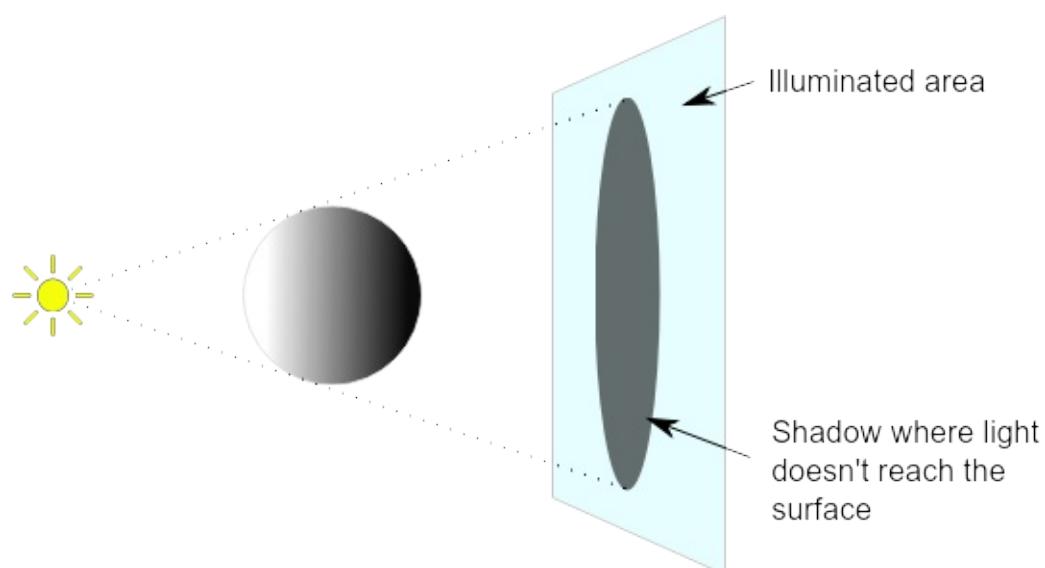
Unity 的灯光可以将 阴影 从一个游戏对象投射到自身的其他部分或是附近的其他游戏对象上。阴影以『扁平』的方式体现游戏对象的尺寸和位置，因此可以为场景添加一定程度的深度和真实感。



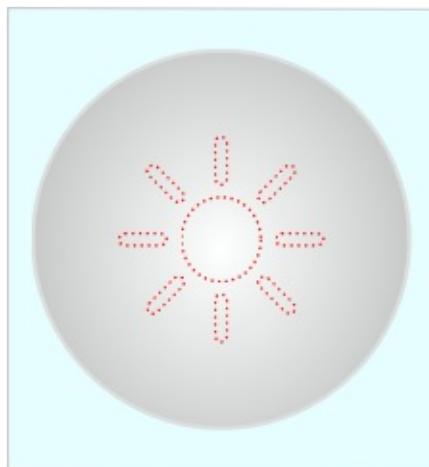
场景视图中的游戏对象正在投射阴影

阴影如何工作？

考虑一种最简单的情况，在场景中只有单个光源。光线从光源出发并沿着直线传播，最终可能会碰撞到场景中的游戏对象。一旦光线碰撞到某个游戏对象，光线将无法继续传播和照亮前方的任何其他游戏对象（例如，光线无法通过第一个游戏对象，并从该游戏对象上反弹离开）。游戏对象投射的阴影不过是一些不被照亮的区域，因为光线无法到达这些区域。



观察阴影的另一个方式是，想象在光源位置处有一个摄像机。场景中的阴影区域恰好是摄像机无法看到的区域。



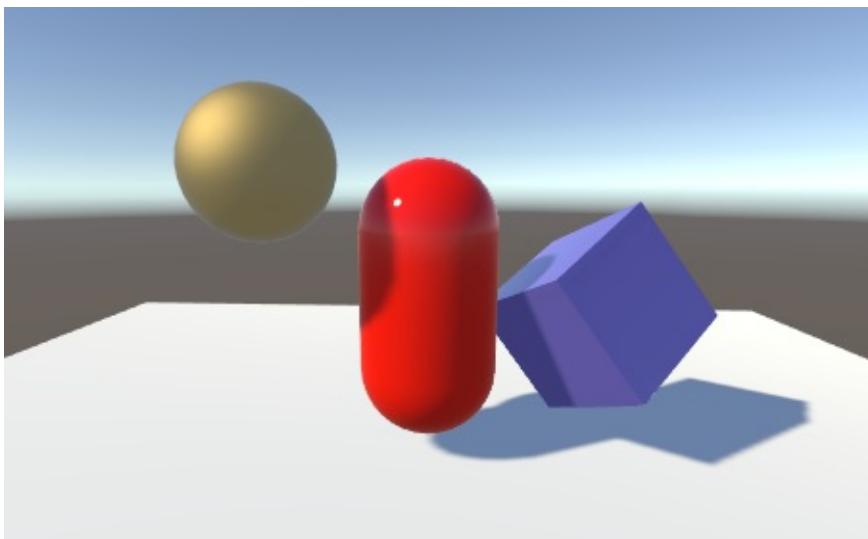
同一场景中，光源位置的眼睛所看到的视图

事实上，这正是 Unity 确定阴影位置的方式。光源使用与摄像机相同的原理，来『渲染』光源视角范围内的场景。光源像摄像机一样使用一个深度缓存系统，持续跟踪面向光源的表面；只有位于视线中的表面接受光照，其他所有表面则都在阴影中。这里的深度映射称为 阴影映射（你可以在阴影映射的 [维基百科页](#) 中找到更多信息）。

下面的章节将详细介绍 Unity 灯光对象投射阴影的细节。

阴影

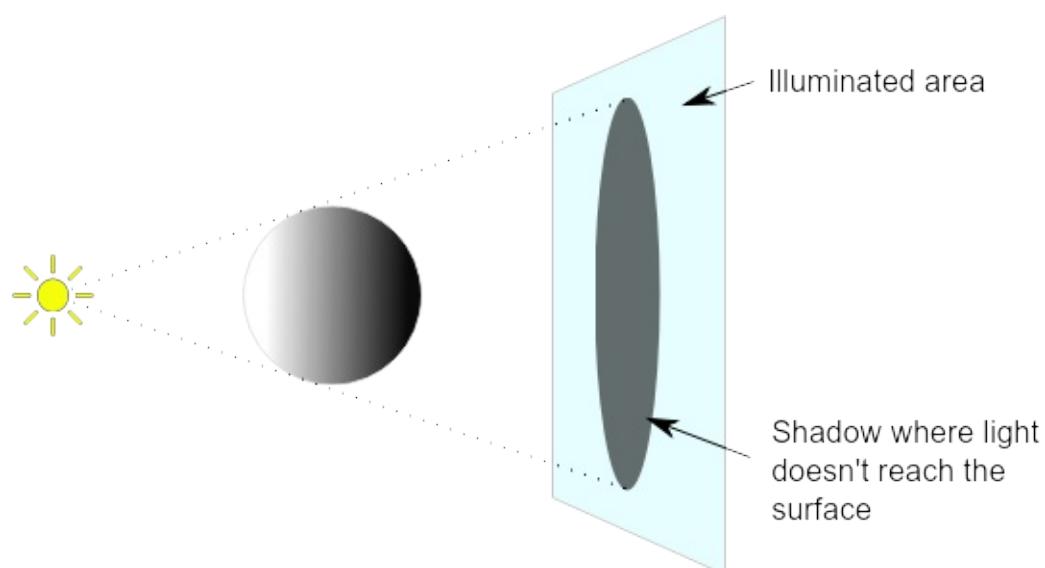
Unity 的灯光可以将 阴影 从一个游戏对象投射到自身的其他部分或是附近的其他游戏对象上。阴影以『扁平』的方式体现游戏对象的尺寸和位置，因此可以为场景添加一定程度的深度和真实感。



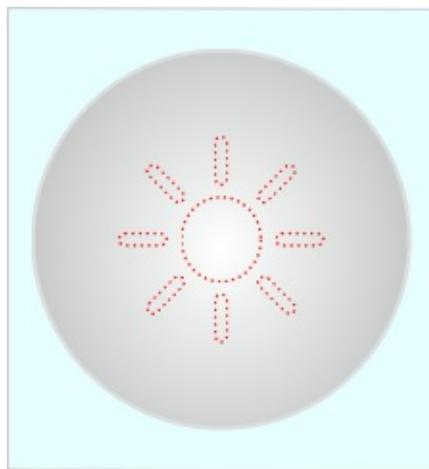
场景视图中的游戏对象正在投射阴影

阴影如何工作？

考虑一种最简单的情况，在场景中只有单个光源。光线从光源出发并沿着直线传播，最终可能会碰撞到场景中的游戏对象。一旦光线碰撞到某个游戏对象，光线将无法继续传播和照亮前方的任何其他游戏对象（例如，光线无法通过第一个游戏对象，并从该游戏对象上反弹离开）。游戏对象投射的阴影不过是一些不被照亮的区域，因为光线无法到达这些区域。



观察阴影的另一个方式是，想象在光源位置处有一个摄像机。场景中的阴影区域恰好是摄像机无法看到的区域。

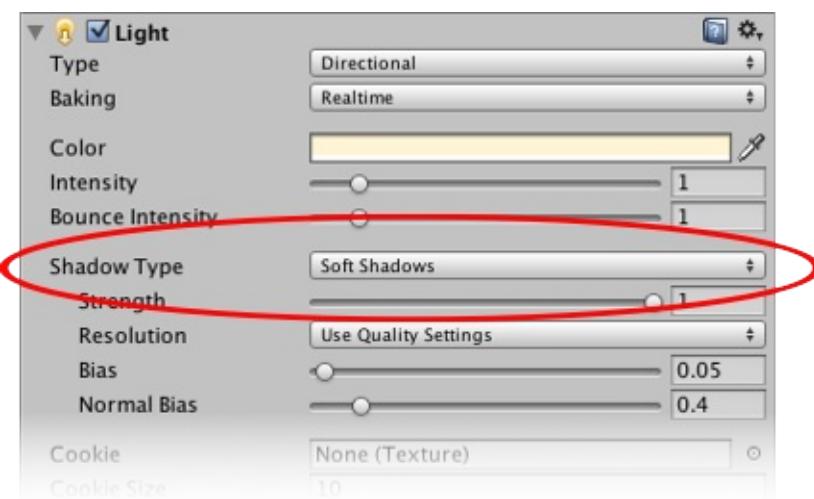


A “light’s eye view” of the same scene 同一场景中，光源位置的眼睛所看到的视图

事实上，这正是 Unity 确定阴影位置的方式。光源使用与摄像机相同的原理，来『渲染』光源视角范围内的场景。光源像摄像机一样使用一个深度缓存系统，持续跟踪面向光源的表面；只有位于视线中的表面接受光照，其他所有表面则都在阴影中。这里的深度映射称为 阴影映射（你可以在阴影映射的 [维基百科页](#) 中找到更多信息）。

开启阴影

使用检视视图中的 **Shadow Type** 属性来开启和定义来自单个光源的阴影。



Shadow Type 阴影类型

选项 **Hard Shadows** 产生具有尖锐边缘的阴影。与 **Soft Shadows** 相比，它不是特别逼真，但是只涉及较少的计算，并且对于大多数场合是可接受的。**Soft Shadows** 则可以减少阴影纹理的『块状』混叠效应。

Strength 阴影强度

决定阴影的暗度。一般来说，某些光线会被大气层散射，和被其他游戏对象反射，所以通常不希望阴影被设置为最大暗度。

Resolution 阴影分辨率

设置阴影纹理在『摄像机』上的渲染分辨率。如果阴影具有非常明显的边缘，那么可能需要增大该值。

Bias 阴影偏差

微调阴影的位置和定义。相关详细信息，请参阅下面的 [阴影映射和偏差属性 Bias](#)。

Normal Bias 阴影法线偏差

微调阴影的位置和定义。相关详细信息，请参阅下面的 [阴影映射和偏差属性 Bias](#)。

Shadow Near Plane 阴影近剪裁平面

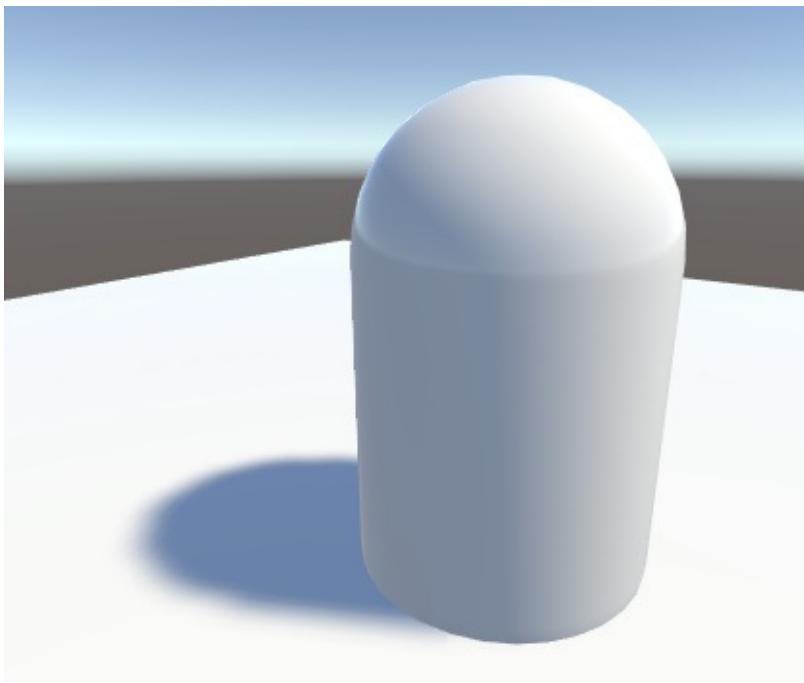
允许在渲染阴影时选择近剪裁平面的 **Near** 值。如果游戏对象到光源的距离小于该值，这不会投射任何阴影。

场景中的每个 [网格渲染器 Mesh Renderer](#) 还具有 **投射阴影 Cast Shadows** 和 **接收阴影 Receive Shadows** 属性，它们必须按需开启。

在 **投射阴影 Cast Shadows** 的下拉菜单中选择 **On** 或 **Off** 可以开启或禁用该网格投射阴影。或者选择 **Two Sided**，以允许表面的两面都投射阴影（此时，背面剔除被忽略）；或者选择 **Shadows Only**，以允许一个不可见的游戏对象投射阴影。

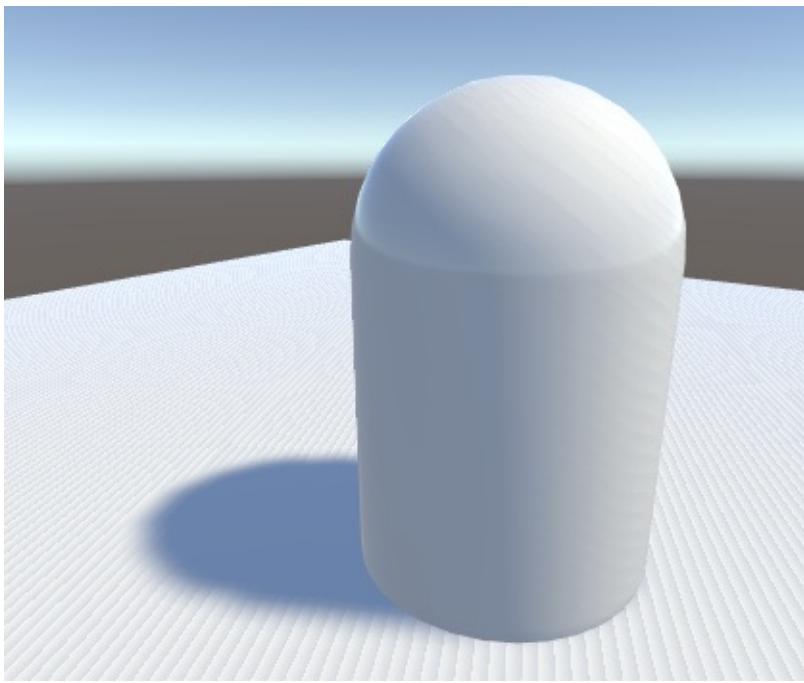
阴影映射和偏差属性 Bias

光源的阴影在渲染最终场景期间被确定。当场景被渲染到主摄像机视图时，视图中每个像素的位置被转换为光源坐标。比较表面像素和阴影纹理中像素距光源的距离。如果表面像素的距离更远，那么表面像素可能被它与光源之间的另一个游戏对象所遮挡，所以它不会被照亮。



正确的投影

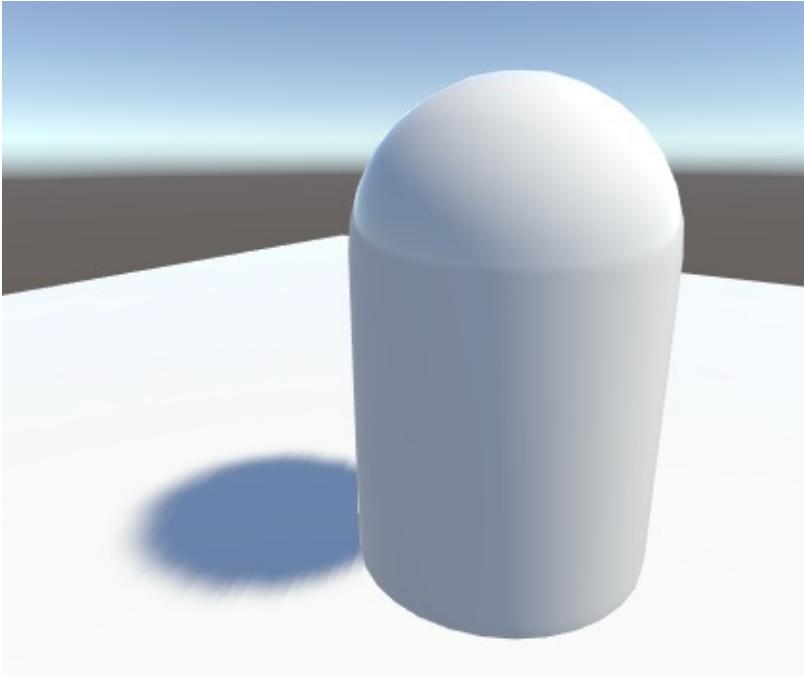
被光源直接照亮的表面，有时看起来部分位于阴影中。这是因为位于阴影中特定距离的像素，有时被计算为更远（这是使用了阴影过滤，或是为阴影纹理设置了低分辨率图像）。最终，位于阴影中的像素上的图纹，原本应该被真正照亮，却产生了成为了『阴影痤疮』的视觉效果。



错误的自投影产生了阴影痤疮

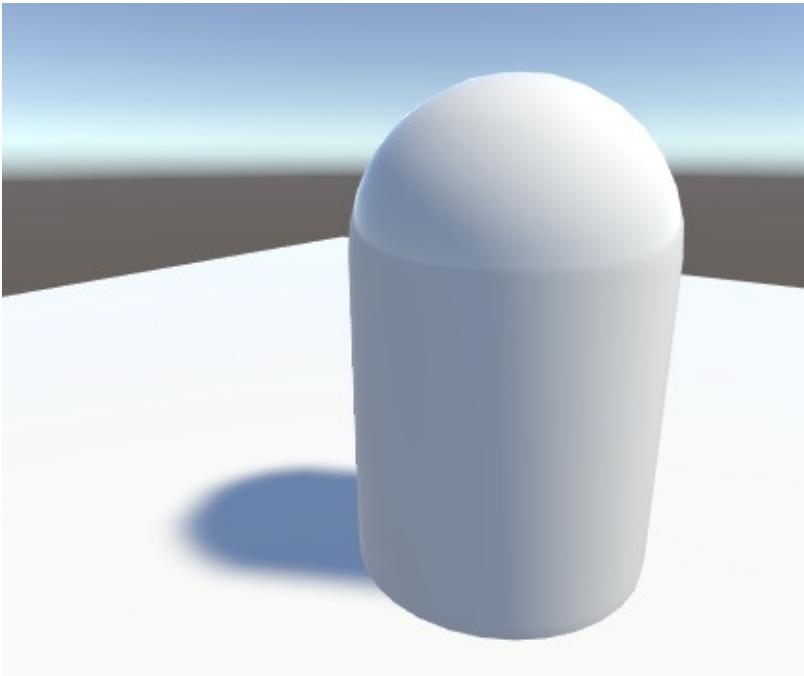
为了防止阴影痤疮，可以让阴影纹理的距离加上 **Bias** 值，以确保位于边界的像素通过距离比较，或者，在渲染阴影纹理时，稍微修正游戏对象的法线。当开启阴影时，这些值可以通过灯光检视视图中的 阴影偏差 **Bias** 和 阴影法线偏差 **Normal Bias** 属性设置。

不要将 阴影偏差 **Bias** 设置的太大，因为阴影周围靠近游戏对象的区域有时会被错误地照亮。从而导致阴影和游戏对象分离，使得游戏对象看起来好像是在地面上空飞行。



阴影偏差 **Bias** 过大，使得阴影与游戏对象分离。

同样地，阴影法线偏差 **Normal Bias** 过大，会使阴影相比游戏对象显得太窄：

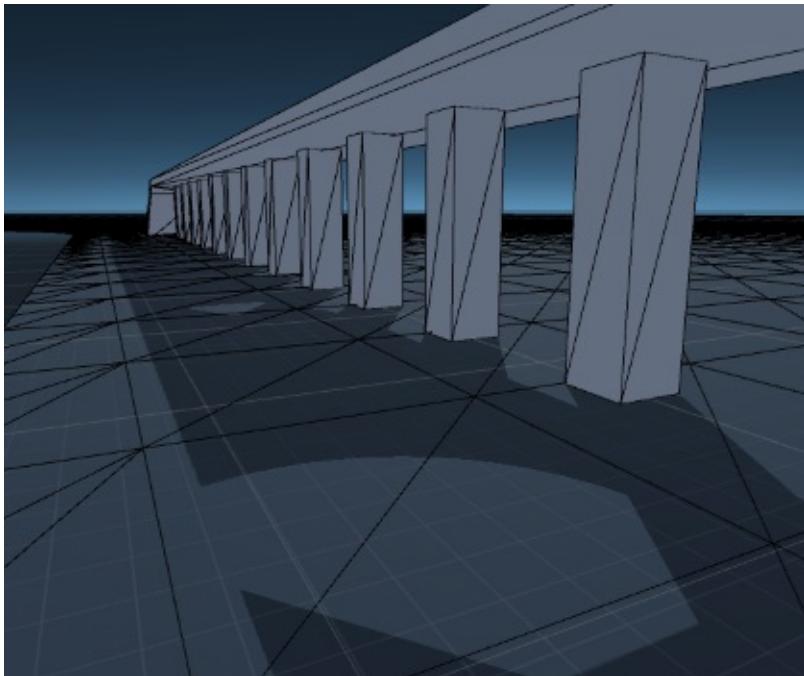


阴影法线偏差 **Normal Bias** 过大，使得阴影的形状太窄

在某些情况下，阴影法线偏差 **Normal Bias** 可能导致称为『光线泄漏』的意外效果，光线穿过相邻的几何体，泄漏到被遮盖的区域。一个可行的解决方案是，打开游戏对象的网格渲染器 **Mesh Renderer**，修改 **Cast Shadows** 属性为 **Two Sided**。但是这个方案可能需要更多的计算资源，增加渲染场景时的性能开销，尽管它在某些情况下确实有效。

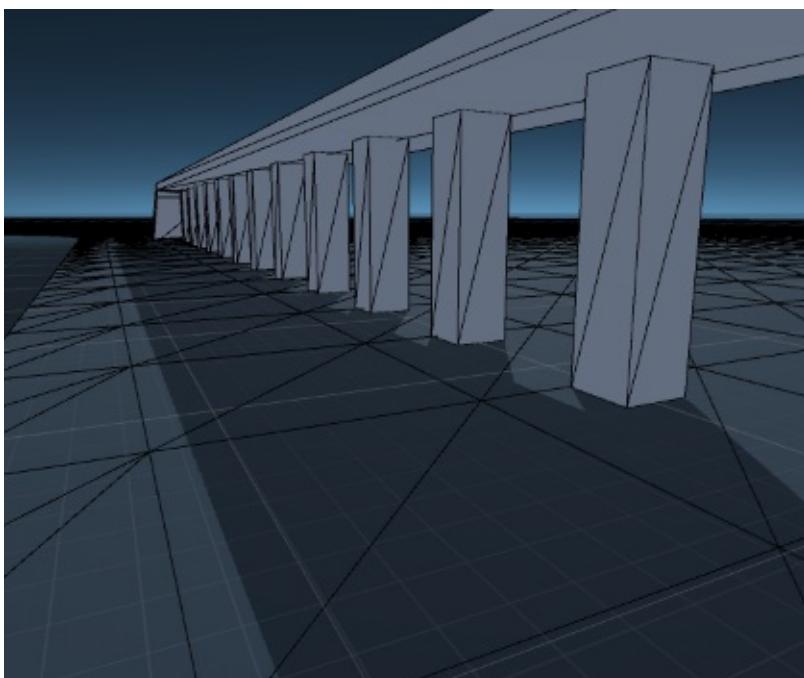
为了避免发生这种意外效果，光源的阴影偏差可能需要调整。通常，通过眼睛来判定正确的偏差值更容易些，而不是尝试通过计算获得。

为了进一步防止阴影座疮，我们使用了一种成为『阴影着陆』的技术（参见 [平行光阴影：阴影着陆](#)）。通常，这种技术是有效的，并且可以为非常大的三角形创建人为的视觉效果。



设置较低的 阴影近剪裁平面偏移 **Shadow near plane offset** 值，导致阴影中的孔状效果。

调整 阴影近剪裁平面偏移 **Shadow Near Plane Offset** 属性，可以解决这个问题。该值太大，也会导致阴影座疮。



正确的投影

全局光照

全局光照（Global Illumination, GI）是一套模拟系统，不仅模拟光线如何直接照射到表面（直接光），还可以模拟光线如何从一个表面弹射到其他表面（间接光）。对间接光的模拟使虚拟世界的效果看起来更加真实和连贯，因为对象彼此之间会相互影响外观。一个典型的例子是『色溢』（颜色溢出），例如，太阳光照射到红色沙发上，红色光将被弹射到后面的墙上。另一个例子是，当太阳光照射到洞穴口的地板上时，光线将在洞穴的内部弹射，所以洞穴的内部也被照亮。



场景视图中的全局光照。注意微妙的间接光照效果。

GI 概念

传统的视频游戏和其他实时图形应用程序仅限于直接光照，因为间接光照所需的计算太慢了，因此它们只能用于非实时情况，例如电影动画 CG。绕过这个限制的方法是，只为提前就知道不会移动（静态）的对象和表面计算间接光照。这样，慢速计算被提前进行计算，因为这些对象不会移动，所以预计算的间接光照在运行时仍然是正确的。Unity 支持这种技术，称为烘焙 GI（也被称为烘焙光照贴图），预先计算间接光照并存储的过程称为『烘焙』。烘焙 GI 利用大量计算时间，可以有效地生成来自区域光和间接光的柔和阴影，并且比通常用实时技术生成的更加逼真。

此外，Unity 5.0 增加了一种名为预计算实时 GI 的新技术。它仍然需要类似上述烘培的预计算阶段，并且仍然仅限于静态对象。不过，它不仅仅预计算光线如何在场景中弹射，而且预计算所有可能的光线弹射，并对这些信息进行编码，以便在运行时使用。所以，对于所有静态对象，它回答了『如果任意光线照射到表面，将向哪里弹射？』这一问题。然后，Unity 保存光线传播路径的信息，供以后使用。在运行时，通过反馈真实存在的灯光到之前计算的光线传播路径中，完成最终的光照。

这意味着，灯光的数量、类型、位置、方向和其他属性都可以改变，并且间接照明将相应地改变。类似地，可以改变对象的材质属性，例如颜色、反射光的数量或自发光的数量。

虽然预计算实时 GI 也可以产生柔和阴影，但是通常比烘培 GI 实现的更粗糙，除非场景非常小。还需要注意的是，尽管预计算实时 GI 在运行时完成最终光照，但是它需要在多个桢上迭代地执行，所以，如果光照场景中发生了大量改变，将需要更多的桢才能完全生效。虽然这对于实时应用程序来说已经足够快了，但是如果目标平台的计算资源非常有限，最好使用烘培 GI 来获得更好的运行时性能。

GI 的局限性

烘培 GI 和预计算实时 GI 都有其局限性，即只有静态对象才能被包含在烘培和预计算过程中——因此移动的对象不能弹射光线到其他对象上，反之亦然。不过，移动对象仍然可以接收使用了光照探针的静态对象的弹射光。在烘培和预计算过程中，会测量（探测）探针位置的光照，并且，在运行时的任意时刻，照射到非静态对象的间接光用最靠近它的探针的值近似模拟。因此，一个靠近白色墙壁的红球，不会溢出它的颜色到墙壁上，但是，一个靠近红色墙壁的白球，可以通过光照探针接收墙壁溢出的红光。

译注：球体是运动对象，墙壁是静态对象。

GI 效果示例

- 通过改变平行光的方向和颜色，以模拟太阳在天空中移动的效果。同时修改天空盒和平行光，可以逼真地创建在运行时更新的昼夜效果。（事实上，内置新增的过程式天空盒很容易做到这种效果。）
- 随着一天时间的流逝，穿过窗户的阳光在地板上移动，并且光线在房间内和天花板上逼真地弹射。当阳光到达红色沙发时，红光被弹射到它后面的墙壁上。把沙发的颜色从红色改为绿色，将导致它后面的墙壁也从红色变为绿色。
- 为霓虹灯材质的自发光属性添加动画，这样当它被打开时，将照亮周围的环境。

下面的章节详细介绍了如何使用全局光照功能。

摄像机

通过在三维空间中布置和移动游戏对象来创建 Unity 场景。而观察者的屏幕是二维的，因此需要通过某种方式来捕获视图，并使之平面化，才能显示在屏幕上。这个过程通过 **摄像机 Camera** 完成。

摄像机是一个游戏对象，定义了场景空间的观察视图。它的位置定义了观察点，它的向前轴 (Z) 和向上轴 (Y) 分别定义了观察方向和屏幕的垂直方向。**摄像机组件 Camera** 还定义了视椎体（落入观察视图的区域）的大小和形状。设置这些参数后，摄像机就可以把它当前『看到』的内容显示到屏幕上。当摄像机对象移动和旋转时，显示的视图也将相应地移动和旋转。

透视相机和正交相机



同一场景的透视模式（左）和正交模式（右）。

真实世界中的摄像机或人眼，观察世界的方式是，离观察点越远，物体看起来越小。这种众所周知的透视效果被广泛应用于艺术和计算机图形，并且对于创建一个真实的场景至关重要。Unity 支持透视摄像机，不过出于某些目的，你可能希望渲染的视图不具有这种效果。例如，创建地图或信息显示时，它们看起来不应该像真实世界中的物体那样。不随距离缩小物体的大小的摄像机称为 **正交摄像机**，Unity 摄像机为此提供了一个选项。以透视模式和正交模式观察场景被称为摄像机 **投影**。（上面的场景来自 [BITGEM](#)）。

视椎体的形状

透视摄像机和正交摄像机都对从当前位置能『看』多远有限制。该限制由一个垂直于摄像机向前轴的平面定义。该平面被称为远剪裁平面，因为大于这个距离的物体将被『剪裁』掉（即从渲染中剔除）。相应地，在摄像机附近还有一个近剪裁平面——这两个平面之间的距

离就是可见的距离范围。

如果没有透视效果，不管距离多远，物体的大小看起来都一样。这意味着，正交摄像机的视椎体是一个限制在两个剪裁平面之间的长方体。

使用透视效果时，随着与摄像机的距离增加，物体的大小看起来被减小了。这意味着，屏幕可视区域的宽度和高度随着距离的增加而增加。投射摄像机的视椎体不是一个长方体，而是一个金字塔形状，顶点位于摄像机位置，底部位于远剪裁平面。由于它的高度不是恒定的，截头椎体由其宽度和高度的比率（称为 纵横比、高宽比）和定点处上剪裁平面和下剪裁平面的夹角（称为 视场 *FOV*）来定义。相关详细说明请参阅 [了解截头视椎体](#)。

摄像机视图的背景

对于室内场景，例如建筑物、洞穴或其他结构，摄像机可以一直显示它们的内部结构。然后在室外场景时，物体之间将会有许多没有任何填充的空区域；这些背景区域通常代表了天空、太空或黑暗的水下场景。

摄像机不能没有背景，它必须用某些东西来填充空区域。最简单的方式是，在渲染场景之前把背景设置为单一的颜色。通过摄像机的 **背景 *Background*** 属性可以设置颜色，可以在检视视图中设置，也可以在脚本中设置。一个更复杂的方式是，为室外场景使用 [天空盒 *Skybox*](#)。正如其名字所暗示的，天空盒的行为像是一个内衬了天空图像的盒子。摄像机被放置在盒子的中心，可以从任何角度看到天空。随着摄像机的旋转，它可以看到不同的天空区域，但是一直位于盒子的中心（因此摄像机无法靠近天空）。天空盒被渲染在场景中所有物体的背后，所以它代表了无限距离处的视图。天空盒最常见的用法是表示室外场景的天空，而实际上，盒子完全包围住了摄像机，包括摄像机的下方。这意味着可以用天空盒来表示部分场景（例如，延伸到地平线之外的平原），或是太空中或水下的全景视图。

通过设置 [光照窗口](#)（菜单：**Window > Lighting**）的 **天空盒 *Skybox*** 属性，可以很容易地为场景添加天空盒。有关如何创建自定义天空盒的更多详细信息，请参阅 [这个页面](#)。

材质、着色器、纹理

在 Unity 中，渲染通过 材质、着色器 和 纹理 完成。

在 Unity 中，材质、着色器和纹理之间的关系非常紧密。

材质 定义了应该如何渲染表面，包含了对贴图的引用、拼接信息、颜色等等。材质的有效选项取决于所使用的的着色器。

着色器 是一些小脚本，包含了计算每个像素渲染颜色的数学计算和算法，基于光照输入和材质配置。

纹理 是一些位图图像。一个材质可以包含对多个纹理的引用，材质的着色器在计算物体表面颜色时可以使用这些纹理。相对于物体表面的基础颜色（漫反射），纹理可以表示更多的材质表面细节，例如反射率或粗糙度。

一个材质只能使用一个着色器，这个着色器决定了材质中的哪些选项是有效的。一个着色器指定一个或多个希望使用的纹理变量，在材质的检视视图中，你可以为这些纹理变量指定纹理资源。

对于大多数渲染——角色、场景、环境、固体、透明物体、坚硬外表、柔软外表等，[标准着色器](#) 往往是最佳选择。这是一种可高度定制的着色器，可以非常逼真地渲染很多种外表类型。

而在某些情况下，使用其他内置着色器，甚至是自定义着色器，可能更加合适——例如，液体、植物、玻璃折射、粒子效果、卡通化或者其他艺术效果，或者其他特技效果，例如，夜视仪、热成像仪或 X 光透视。

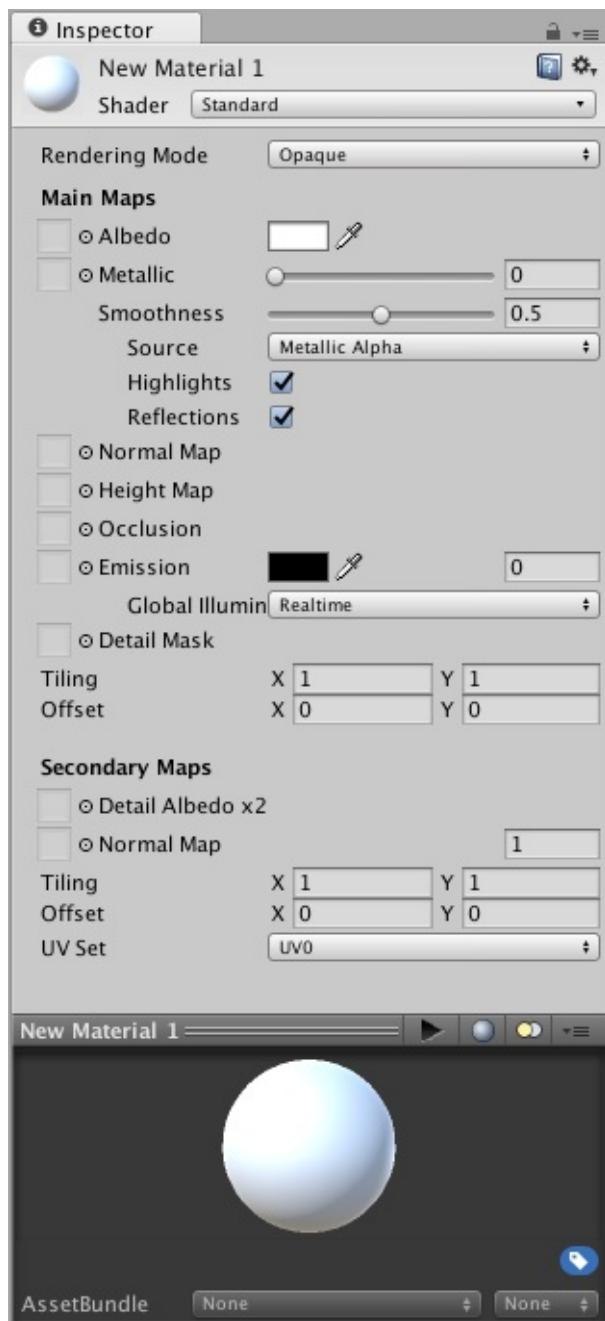
下面这些页面介绍了：

- [创建和使用材质](#)
- [内置标准着色器](#)
- [其他内置着色器](#)
- [编写着色器](#)

创建和使用材质

通过主菜单 **Assets->Create->Material** 或项目视图的上下文菜单，可以创建一个新材质。

默认情况下，新建材质会分配标准着色器，并且所有属性都空着，就像这样：



一旦材质被创建，你就可以把它应用到一个对象，并且在检视视图中修改它的所有属性。想把材质应用到一个对象上，只需要把材质从项目视图拖动到场景视图或层级视图中的任意对象上。

设置材质属性

你可以为材质选择任意着色器。操作非常简单，在检视视图中展开 着色器 下单框，选择一个新着色器。所选择的着色器决定了可改变的有效属性。属性可以是颜色、滑动器、数值或向量。如果你已经把材质应用到了 场景视图 中的某个激活对象上，那么，改变属性会实时应用到该对象上。

把纹理应用到某个属性有两种方式：

- 从项目视图拖动纹理资源到检视视图的纹理插槽上。
- 点击 选择 按钮，从弹出的下拉列表中选择纹理资源。

内置着色器

除了 [标准着色器](#)，还是许多其他类型的内置着色器，用于不同的用途：

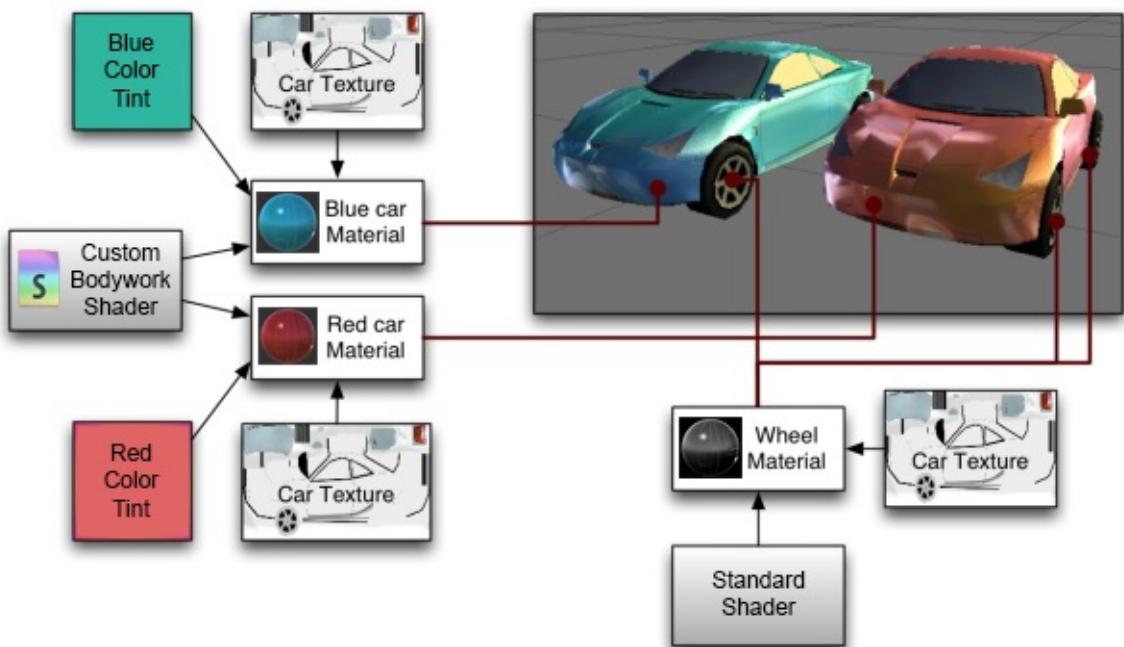
- **FX**：用于光照和玻璃效果。
- **GUI 和 UI**：用于用户图形界面。
- **Mobile**：用于移动设备的高性能着色器。
- **Nature**：用于树木和地形。
- **Particles**：用于粒子系统效果。
- **Skybox**：用于渲染位于所有几何图形之后的背景环境。
- **Sprites**：用于 2D 精灵系统。
- **Toon**：用于渲染卡通风格。
- **Unlit**：用于绕开所有光照和阴影的渲染。
- **Legacy**：大量老版本着色器的集合，已经被标准着色器取代。

着色器技术细节

着色器是一段包含了数学计算和算法的脚本，用于决定模型表层的外观。标准着色器会执行复杂而逼真的光照计算。其他着色器可能采用简化或不同的计算方式，从而现实不同的结果。任意一个着色器都带有一组可改变的属性。这些属性可能是数值、颜色或纹理，当你查看材质时，属性会显示在检视视图中。材质由绑定到游戏对象的 **Renderer** 组件所使用，用于渲染游戏对象的网格。

多个不同的材质可以引用同一个纹理。这些材质可以使用相同或不同的着色器，完全取决于需求。

下面是一套设置组合的示例，用到了 3 个材质、2 个着色器和 1 个纹理。



在上图中，有一辆红车和蓝车。两个模型的车身使用不同的材质，分别是红车材质和蓝车材质。

两个车身材质使用了同一个自定义着色器 — 车身着色器。之所以使用自定义着色器，是因为它可以为汽车提供额外的功能，例如渲染金属光斑，可能还会提供自定义的损伤变形功能。

两个车身材质都引用了汽车纹理，一张包含了车身所有细节的纹理图，但是其中没有指定绘制颜色。

车身着色器还可以接受一个色调，用于为红车和蓝车设置不同的色调，从而让两辆车拥有不同的外观，而我们实际上只使用了一张纹理。

车轮使用一个独立的材质，但是两辆车的车轮共享这个材质，因为两辆车的车轮并没有什么不同。车轮材质使用标准着色器，并再次引用汽车纹理。

需要注意的是，汽车纹理同时包含了车身和轮胎的细节 — 称为纹理集，纹理贴图的不同部分可以明确地映射到模型的不同部位。

尽管车身材质引用了一个包含车轮图像的纹理，但是车轮不会显示在车身上，因为车轮部分没有被映射到车身结构。

同样的，车轮材质也使用了同一个材质，其中包含了车身细节。但是车身细节不会显示在车轮上，因为只有贴图的车轮细节被映射到了车轮结构。

映射关系由外部 3D 应用程序制作，称为『UV 映射』。

更具体地说，一个着色器定义了：

- 渲染游戏对象的方法。包括代码和数学计算，数学计算包括光源角度、视角和其他所有相关的计算。着色器也可以基于终端用户的图形硬件采取不同的方式。

- 在材质检视图中可以自定义参数，例如纹理映射、色彩和数值。

一个材质定义了：

- 使用哪种着色器来渲染材质。
- 该着色器参数的具体值——例如使用哪个纹理映射、色彩值和数值。

自定义着色器由图形程序员编写。使用 **ShaderLab** 语言，非常简单。不过，要让一个着色器在各种各样的图形显卡上都正常运行，是一项复杂的工作，并且要求对显卡运行原理有相当全面的知识。

一些着色器直接内置在 Unity 中，更多的着色器在 [标准资源](#) 库中。

标准着色器

Unity 标准着色器是一个内置着色器，具有非常全面的功能。它可用于渲染『真实世界』的对象，例如，石头、木材、玻璃、塑料和金属，并支持各种各样的着色器类型和组合。通过使用或不使用材质编辑器中的各种纹理插槽和参数，可以很容易地启动或禁用其功能。

标准着色器还包括一个称为 **物理着色器** (**Physically Based Shading**, **PBS**) 的高级光照模型。物理着色器以模拟真实世界的方式模拟材质和光照之间的相互作用。物理着色器最近才在实时图形中变为可能。最适合物理着色器的场景是，光照和材质需要直观并真实地共存时。

物理着色器背后的目标是，在不同光照条件下，以用户友好的方式，实现一致、可信的视觉效果。它模拟了光照在真实世界中的行为，并且不需要使用其他特殊模型（不管特殊模型是否可以运行）。为此，物理着色器遵循物理学法则，包括能量守恒（对象反射的光永远不会超过接收的光）、菲涅尔反射（所有表面在掠射角时反射更多的光），和表面如何闭合自身（几何术语）。

标准着色器的设计考虑了硬表面（也成为『建筑材料』），可以处理大多数真实世界的材料，如石头、玻璃、陶瓷、黄铜或橡胶。它甚至还可以处理非硬材料，如皮肤、头发和衣服。



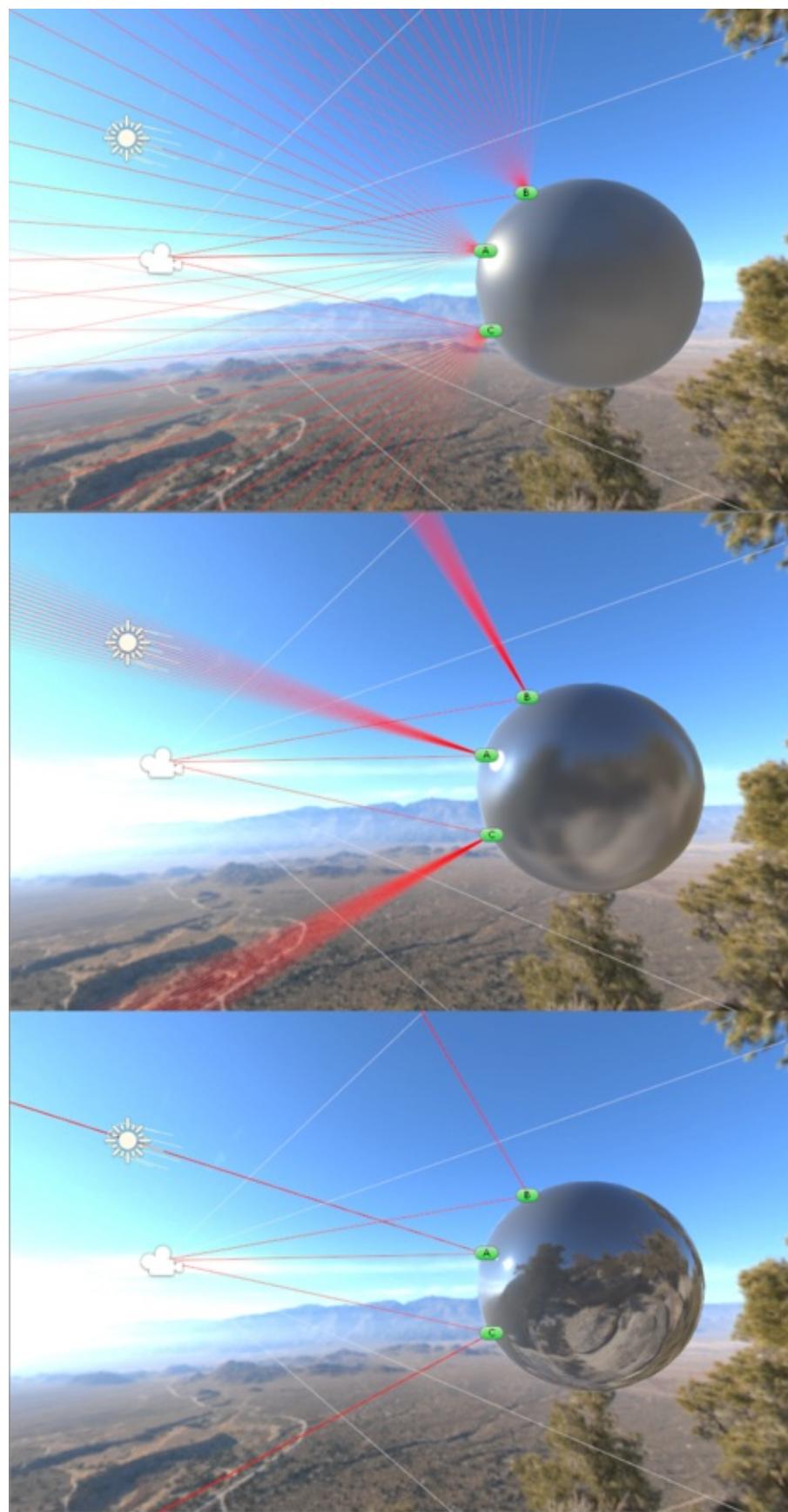
在所有模型上使用标准着色器渲染的场景

通过标准着色器，大量的着色器类型（例如，漫反射、镜面、凹凸镜面、反射）被合并为单个着色器，并应用于所有材质类型。这么做的好处是，在场景的所有区域都使用同样的光照计算，从而为使用标准着色器的所有模型提供真实、一致、可信的光照和阴影分布。

术语

在谈论 Unity 的物理着色器时，有许多概念非常有用。包括：

- 能量守恒 —— 这是一个物理概念，确保对象反射的光不会超过接收的光。材质越接近镜面，散射的光越少；表面越光滑，反射的光越强，反射的光束越小。



在表面每个点处渲染的光与从环境接收的光的数量相等。粗糙表面的细微颗粒受到来自更大区域的光的影响。越平滑的表面，反射的光越强，反射的光束越小。点 A 将光从光源反射到摄像机。点 B 从天空的环境光获得蓝色色调。点 C 接收环境光，并反射来自周围地面的光。

- 高动态范围图像（**High Dynamic Range**，**HDR**）——通常指 0 - 1 范围之外的光。例如，太阳光的亮度比蓝天高 10 倍。相关的深入讨论，请参阅 Unity 手册的 [HDR](#) 页。



一个使用高动态范围图像的场景。从车窗反射的太阳光比场景中的其他对象明亮的多，因为它已经用 **HDR** 处理过。

内容和上下文

当考虑 Unity 中的光照时，可以非常方便地把光照分成两个概念：内容 和 上下文。内容是指被渲染和照亮的物体，上下文是指存在于场景中的光照，会照亮（影响）物体。

上下文

当照亮一个物体时，重要的是要了解哪些光源正在在影响物体。在你的场景中，通常会有直接光源：那些你可能已经放在场景中的光源对象。还会有非直接光源，例如反射光和反弹光。这些都会影响物体的材质，最终生成摄像机在物体表面上所看到的结果。

这种划分并不是固定的，因为往往被认为是『内容』的物体，也可以是另一个物体的光照上下文。

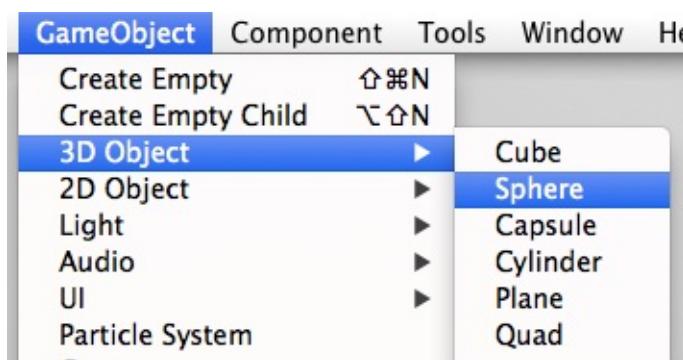
一个很好的例子是位于沙漠环境的建筑物。建筑物从天空盒获取光照信息，并且可能弹射周围地面的光照。

不过，可能有个角色正站在建筑物的外墙附近。对于这个角色，建筑物是光照上下文的一部分 — 建筑物可能投射阴影，可能把光照从墙壁弹射到角色上，或者角色身上的某个部位（镜面）可以直接反射建筑物。

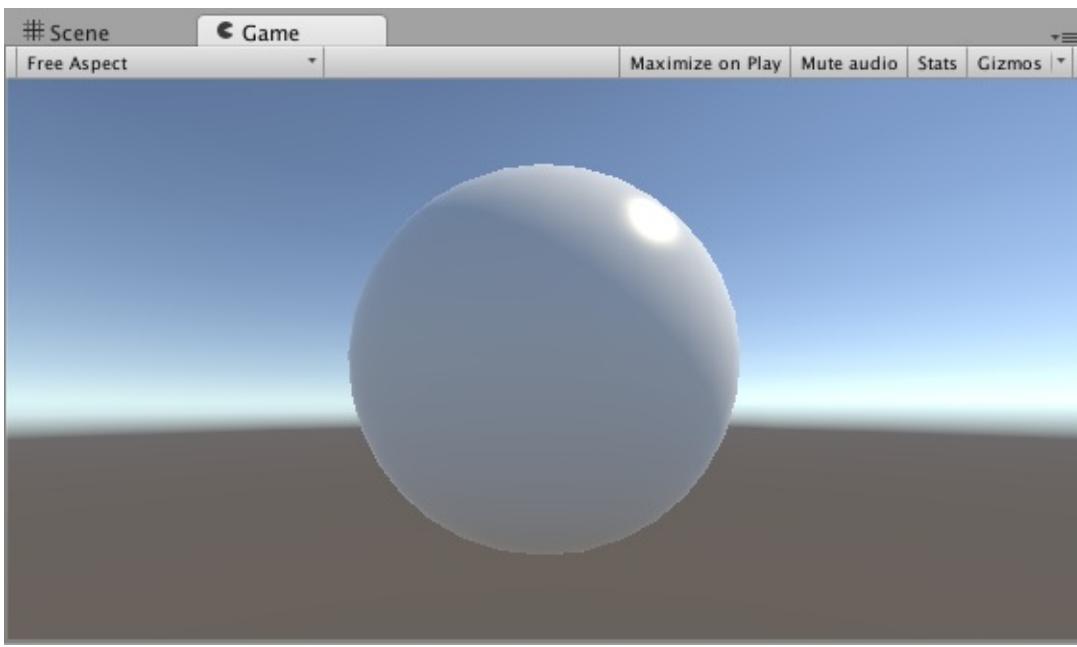
默认光照上下文

Unity 5 启动时会显示一个空场景。该场景中有一个默认的光照上下文：天幕反射和一个平行光。默认情况下，任何放入该场景中的物体，都应该拥有必需的光照，使整个场景看起来是正确的。

让我们向场景添加一个球体，看看默认光照上下文的效果。



默认情况下，添加的球体将使用标准着色器。将相机对焦在球体上会得到这样的东西：



请注意沿着球体边缘的反射光，以及细微的环境光，从棕色（底部）到天蓝色（顶部）。默认情况下，在一个空场景中，所有光照上下文都来自天空盒和一个平行光（默认被添加到场景中）。

当然这是默认设置，在某些情况下，单个光照和天空反射可能不够。你可以轻松地添加更多光照和反射探针：

要深入了解反射光和光照探针的工作原理，请参阅有关 [光照探针](#) 和 [反射探针](#)。

天空盒

天空盒是光照设置不可或缺的组成部分，无论是预先烘培还是程序实时生成。除了渲染天空外，它还可以用于控制环境光照和物体的反射光。程序实时生成的天空盒允许直接设置颜色，并创建一个太阳光盘，而不是使用一张位图——更多信息可以 [天空盒文档](#) 中找到。

对于场景中的许多物体来说，虽然反射天空盒是有用的，特别是户外场景，但是也有许多情况，需要你改变物体的反射设置——可能是户外场景的黑暗区域，例如小巷或茂密的森林，或者某些室内区域需要单独设置反射光照。

为了满足各种反射需求，Unity 提供了 [反射探针](#)，允许你在空间中的某个点，对场景环境进行采样，用该点附近的环境光和反射源，来代替默认的天空盒。反射探针可以放置在场景中的任意位置，在该位置，天空盒不能满足光照需求，或者不适合。

全局光照

全局光照概念是 Unity 5 不可或缺的部分。标准着色器和全局光照彼此协作。全局光照系统负责创建和跟踪反射光、来自发光材质的光，以及来自环境的光。你可以在 [这里](#) 找到更多细节。

上下文是整体图像的关键部分。在这个示例中，你可以看到，在内容和摄像机保持不变的情况下，场景是如何反射上下文的。

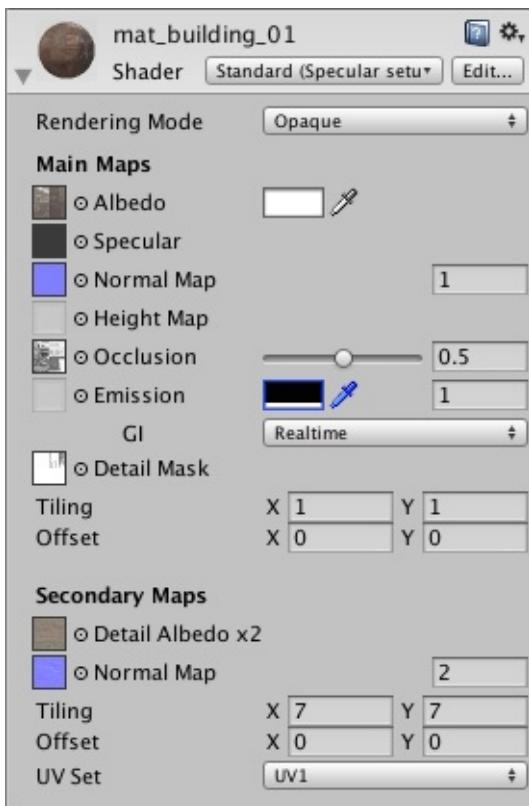


内容

内容是一个术语，用于描述场景中正在渲染的物体。它是光照上下文应用于物体材质之后的结果。

材质编辑器

在检视视图中查看使用了标准着色器的材质时，编辑器将显示材质的所有参数，包括纹理、混合模式、遮罩和辅助图。你可以一目了然地看到哪些功能被使用，并且实时预览材质。标准着色器是数据驱动的：Unity 将只使用用户已经设置过参数的着色器代码。换句话说，如果材质的某个功能或纹理插槽没有被用到，那么就不会消耗性能，着色器在幕后执行这一优化。



你可以按住 Ctrl 点击纹理缩略图，来查看它的大预览图，并且允许你检查颜色和 alpha 通道。

如何创建材料

为了支持各种各样的材质类型，标准着色器提供了许多配置。可以使用纹理贴图、颜色选择器和滑动器设置值。通常，UV 映射是必须的，用于描述网格和纹理贴图的对应关系。使用了标准着色器的材质，支持在同一个网格上设置多个的材质属性，例如反射贴图、平滑度贴图和金属贴图，并把它们结合起来渲染。换句话说，你可以在同一个网格上同时设置橡胶、金属和木材纹理；纹理的分辨率可以超过网格，从而支持平滑边角和材质过度，当然，这会让工作流程更复杂，不过，这将取决于材质的创建方式。

材质纹理通常有两种生成方式——在 2D 图像编辑器中绘制和合成，或者在 3D 软件中渲染和烘培，第二种方式可以使用更高分辨率的模型生成反射贴图、法线贴图和散射贴图等。工作流程取决于使用的外部软件。

通常，纹理贴图不应该包含固定光照（例如阴影、高光等）。PBS 的优点之一是，物体可以如你期望的那样响应光照，但是如果贴图已经包含了光照信息，就无法使用这一优点。

工作流程：金属 **vs** 镜面

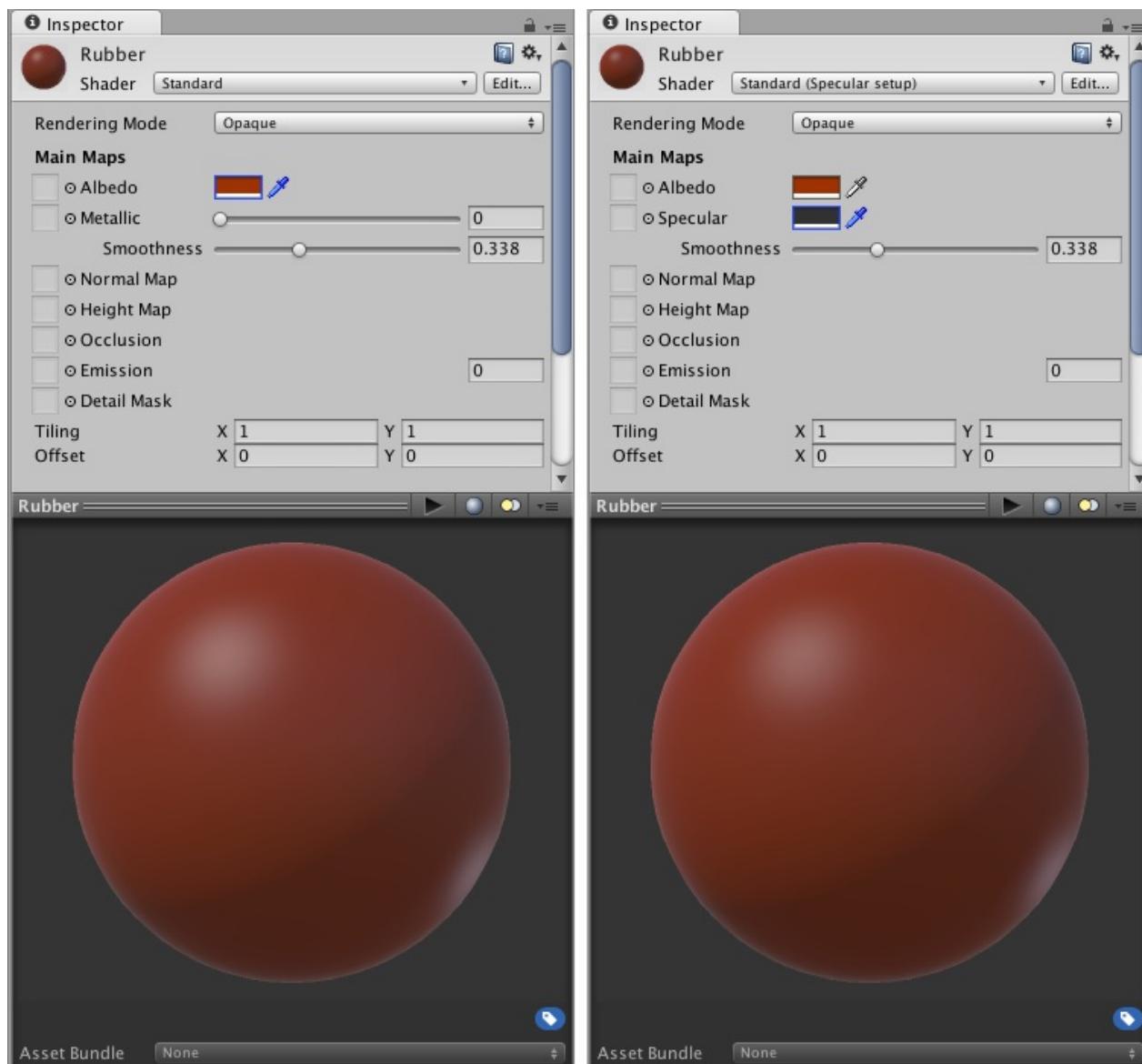
两种工作流程

使用标准着色器创建材质时，你可以选择『标准』和『标准（镜面）』两种类型之一。它们的数据不同，如下所述：

标准：这种着色器暴露一个金属值，用于设置材质是否是金属。在一个金属材质中，漫反射颜色将控制反射的颜色，并且高亮（亮斑）。非金属材质将反射和入射光相同的颜色，并且没有高亮（亮斑）。

标准（镜面）：经典着色器。镜面颜色用于控制反射的颜色和强度。可以反射与漫反射不同的颜色。

无论使用哪一种类型，通常都可以很好的呈现大多数材质类型，所以，在大多数情况下，选择哪种取决于个人偏好，以及是否适合你的美术制作流程。例如下面的橡胶塑料材质例子，分别用标准工作流程和标准（镜面）工作流程创建：



随着材质表面变得更光滑，观察者在掠射角处看到的菲涅耳效应越来越明显。

第一张图演示了金属工作流程，设置它的金属值为 0（表示非金属）。第二张图的设置与第一张几乎一样，但是把镜面颜色设置为接近黑色（所以不会得到类似金属镜面的反射）。

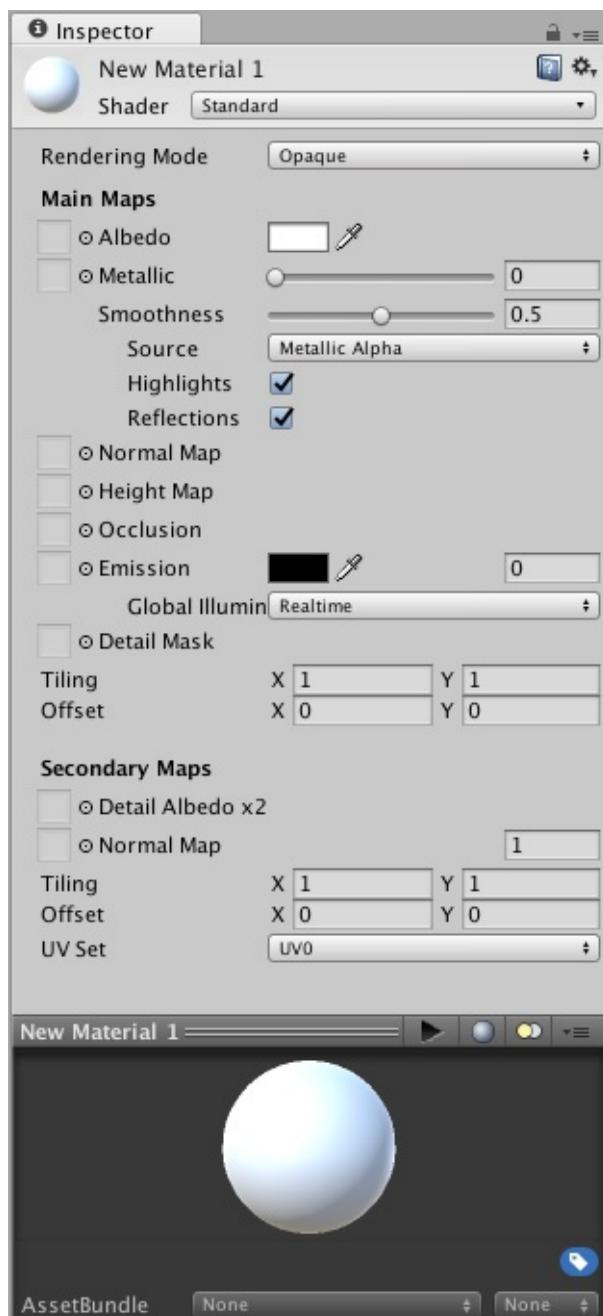
有人可能会问，这些值是从哪里得来的，接近黑色是什么颜色，是什么让草与铝完全不同的？在遵循物理定律的着色过程中，我们可以参考现实世界的材质。我们已经把其中一些材质编成一张快速参照图表，你可以用它来创建材质。

译注：光学现象整理

材质参数

标准着色器提供了材质参数列表。这些参数在金属模式和镜面模式下略有不同。不过大部分参数在两种模式下是一样的，本页涵盖了两种模式的所有参数。

这些参数一起使用可以重现几乎任何真实世界中的物体外观。

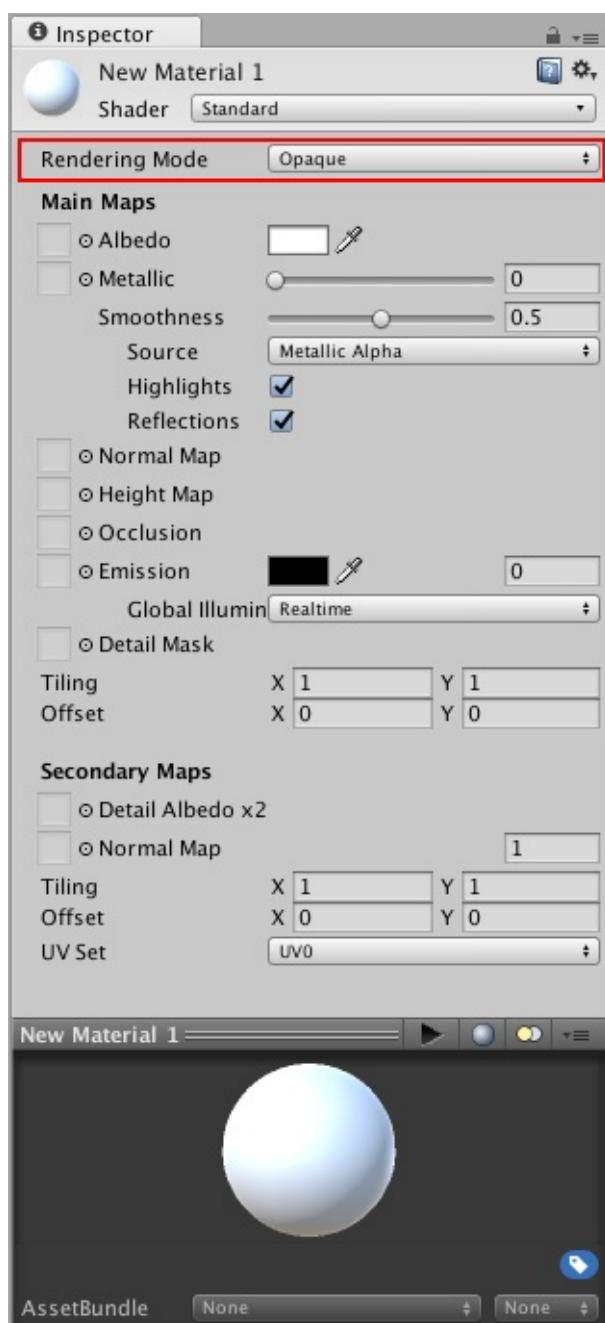


一个具有默认参数并且未分配值或纹理的标准着色器材质。

- 渲染模式
- 漫反射颜色和透明度
- 镜面模式：镜面

- 金属模式：金属
- 平滑度
- 法线贴图（凹凸贴图）
- 高度图（视察偏移贴图）
- 散射贴图
- 自发光
- 细节蒙板和贴图
- 菲涅耳效应

渲染模式

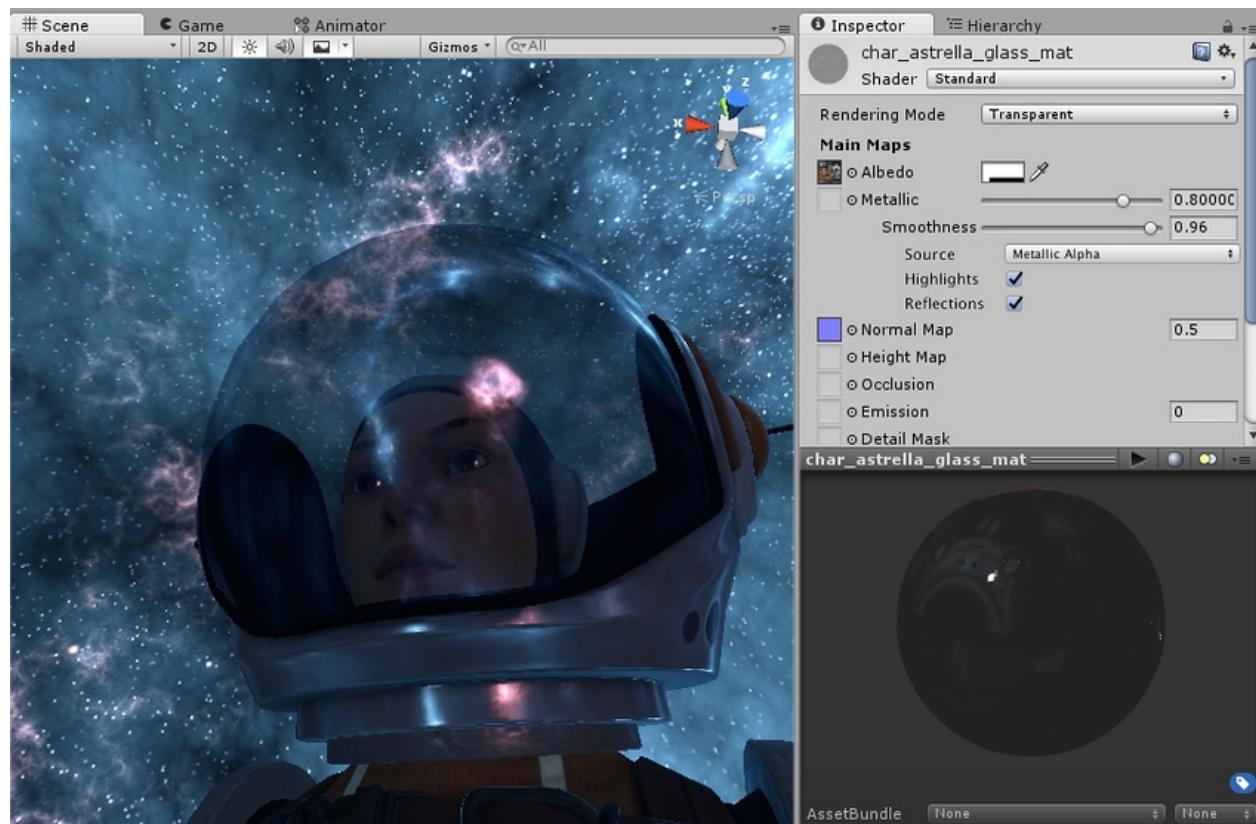


一个具有默认参数并且未分配值和纹理的标准着色器材质。参数『渲染模式』被突出显示。

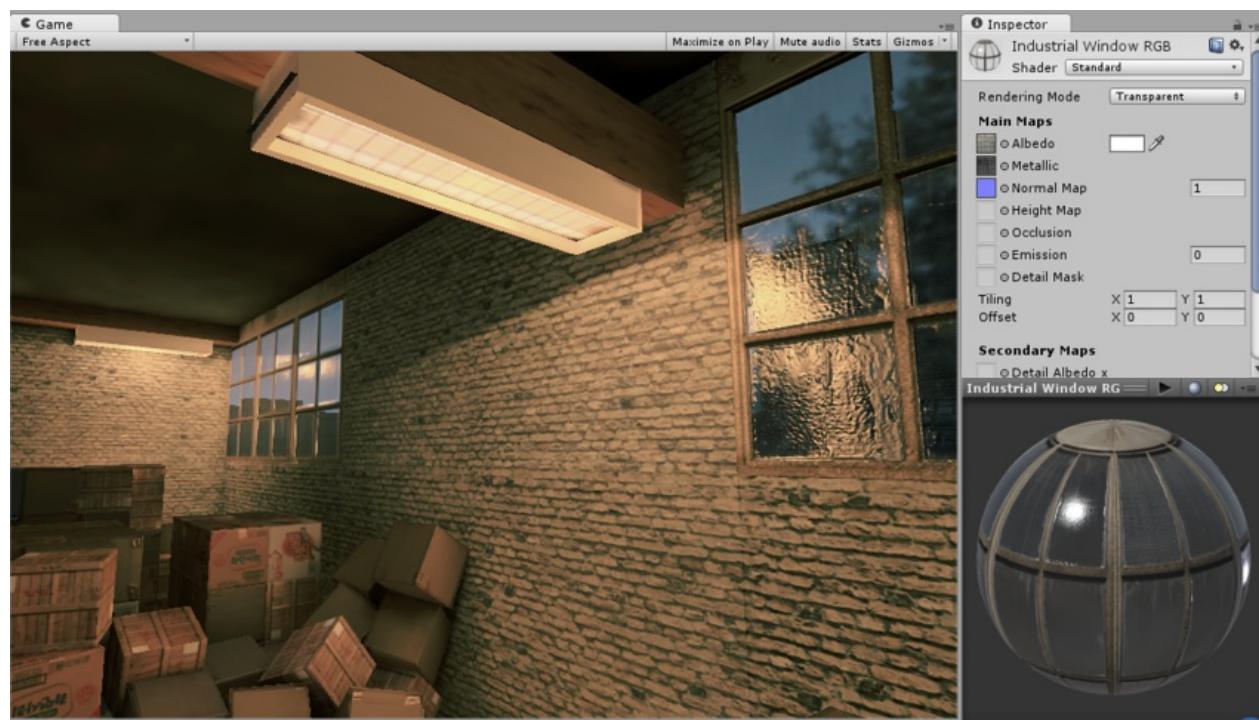
标准着色器的第一个参数是 渲染模式。该参数描述对象是否透明。

- 不透明 — 默认值，适用于没有透明区域的常规固态物体。
- 镂空 — 允许你创建在不透明和透明区域之间具有硬边缘的透明效果。在该模式下，没有半透明区域，纹理或者 100% 不透明，或者不可见。当创建带有透明区域的材质时，例如叶子、带孔衣服和碎布这类形状，使用该模式。

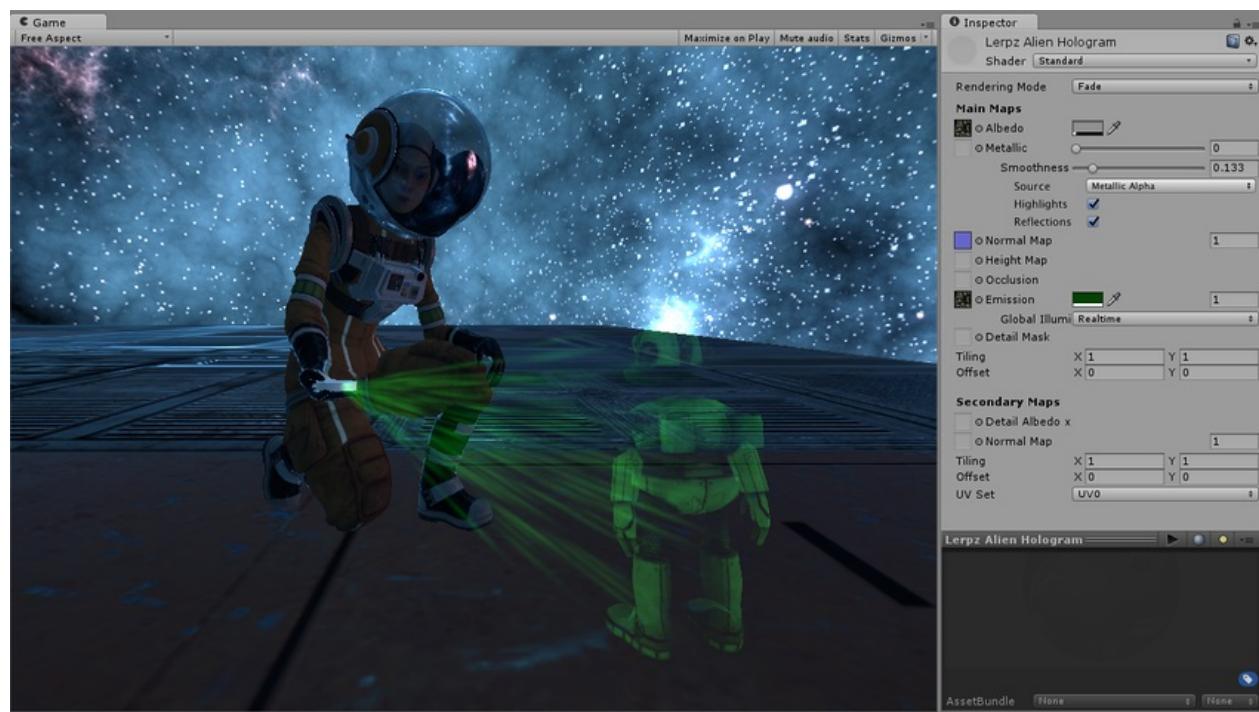
- 透明 — 适用于渲染逼真的透明材质，例如塑料或玻璃。在该模式下，材质自身具有透明度（基于纹理的 alpha 通道和色调的 alpha 值），并且，如同真实的透明材质一样，反射和高光将清晰可见。
- 渐变 — 允许物体逐渐消失，直至完全透明，包括镜面高光和光照反射。如果想创建一个让物体渐显或渐隐的动画，则该模式非常有用。该模式不适合渲染真实的透明材质，例如透明塑料或玻璃，因为光照反射和高光也会逐渐隐藏。



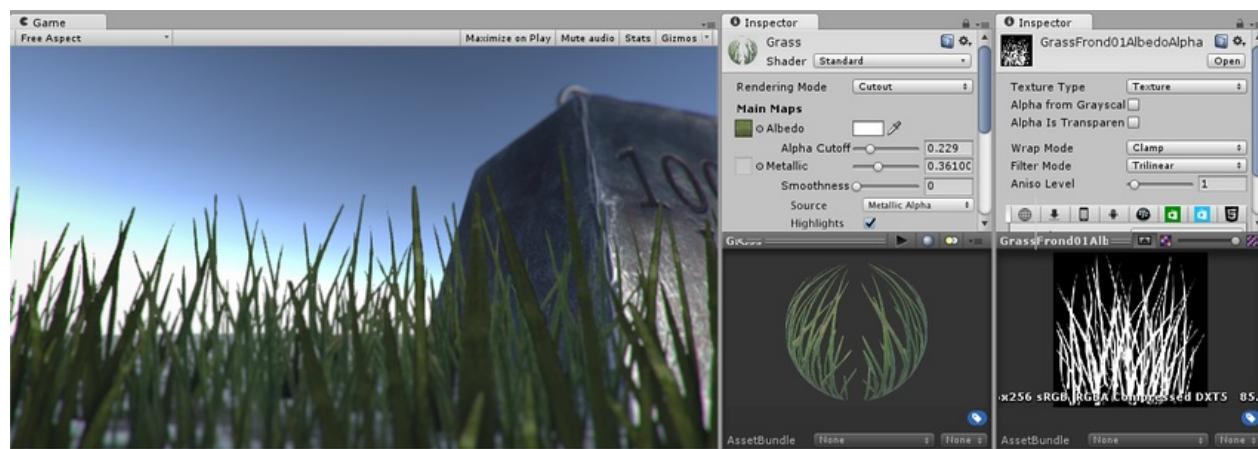
上图中的头盔面罩（密封偷窥观察窗）使用透明模式渲染，因为它可以表示具有透明度属性的真实物理对象。这里的面罩反射了场景中的天空盒。



这些窗户使用了透明模型，但是在纹理中定义一些不透明区域（窗框）。透明区域和不透明区域以镜面的方式反射来自光源的光照。

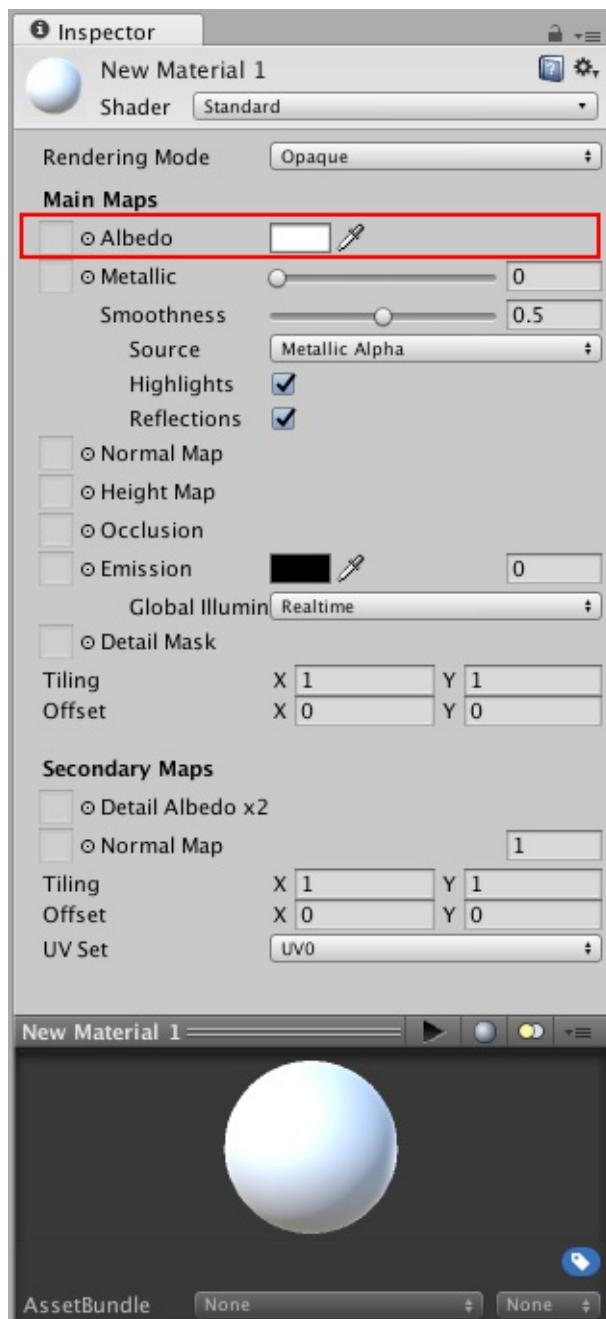


图中的全息图使用渐变模式渲染，因为它可以表示部分渐隐的不透明对象。



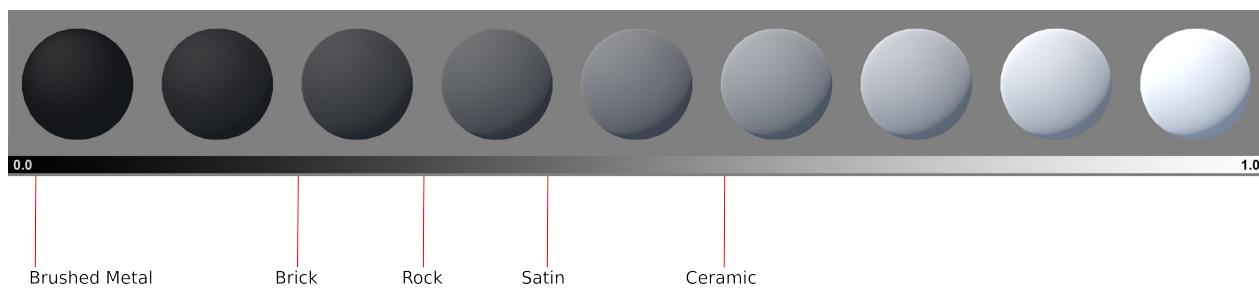
图中的草使用镂空模式渲染。通过定义镂空阀值可以显示清晰的锐利边缘。在图像中的 alpha 高于该阀值的部分是 100% 不透明的，低于阀值的所有部分是不可见的。在图像的右侧，你可以看到材质的设置，和所使用纹理的 alpha 通道。

漫反射颜色和透明度



一个具有默认参数并且未分配值和纹理的标准着色器材质。参数『漫反射颜色』被突出显示。

漫反射参数控制表面的基准颜色（底色）。



漫反射颜色从黑到白。译注：Brushed Metal 拉丝金属，Brick 砖块，Rock 岩石，Satin 绸缎，Ceramic 陶瓷。

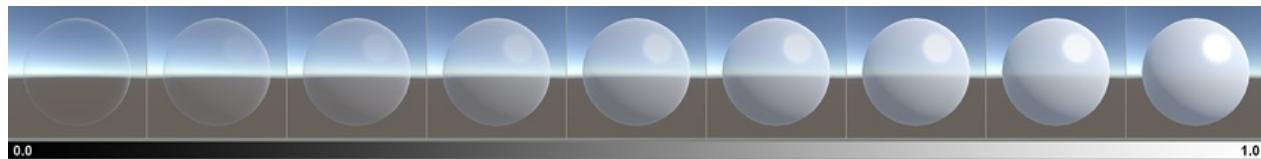
有些情况下，为漫反射指定一个颜色值是有用的，但是，更常见的情况是，为漫反射参数指定一张纹理贴图。漫反射应该表示物体表面的颜色。需要特别注意的是，漫反射纹理不应该包含任何光照信息，因为将基于上下文（环境）添加光照到物体的纹理上。



两个典型的漫反射问题贴图示例。左边是一个角色的纹理贴图，右边是一个木板箱的纹理贴图。注意，两者都不包含阴影和光照高光（镜面高光、光斑）。

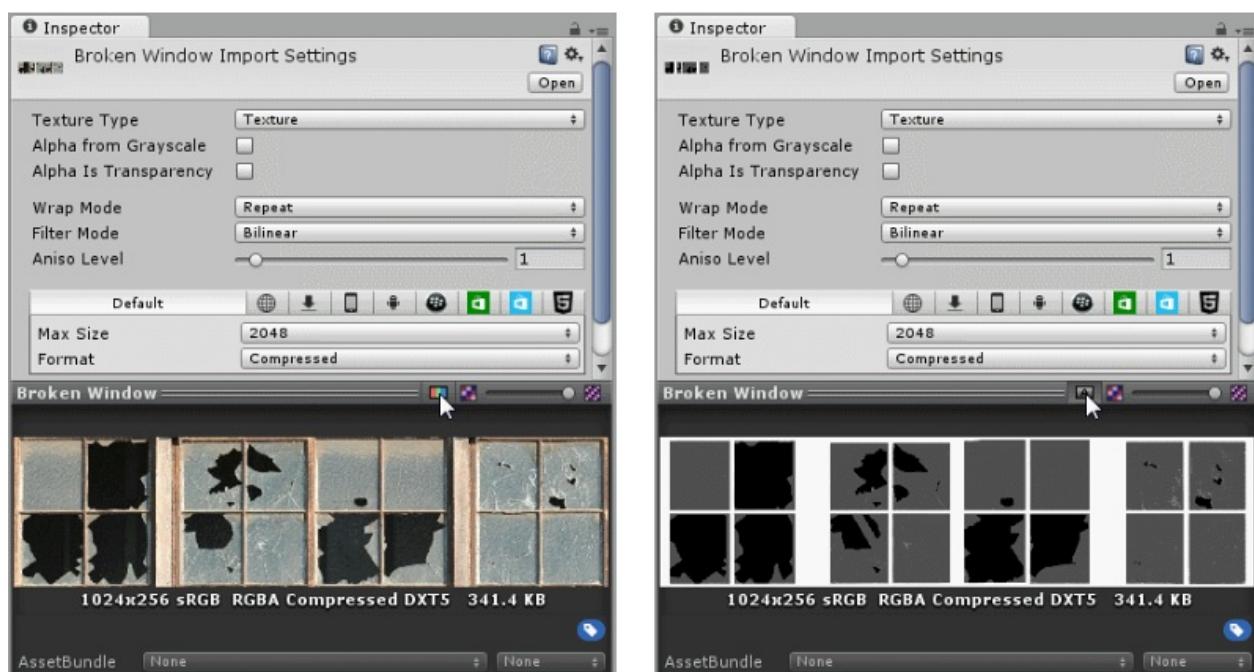
透明度

漫反射颜色的 **alpha** 值控制材质的透明程度。只有在材质的渲染模型被设置为透明模式之一（除了不透明模式意外的镂空、透明或渐变模式）时才会有效果。如上所述，选择正确的透明模式非常重要，因为它决定了是否可以完整地看到反射光照和镜面高光，以及是否可以随照透明度逐渐淡出。

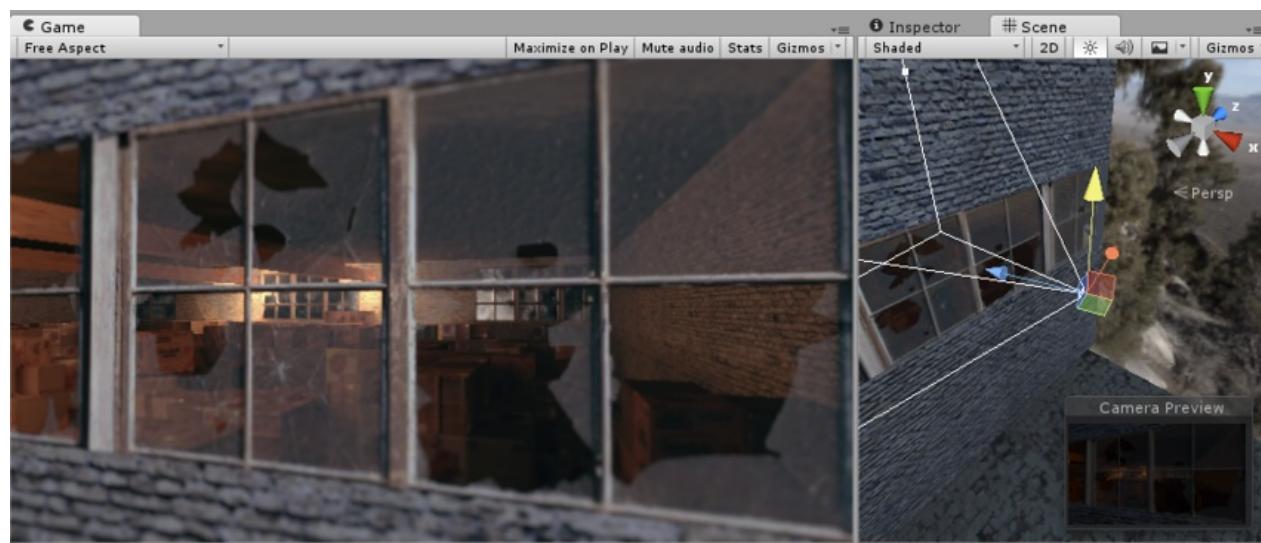


透明度的值从 0 到 1，使用透明模式，以适配真实的透明物体。

当使用分配给漫反射参数的纹理时，如果漫反射纹理贴图含有 **alpha** 通道，那么你可以控制材质的透明度。**alpha** 通道的值被映射为透明度，白色映射为完全不透明，黑色映射为完全透明。这将导致材质可能具有不同透明度的区域。

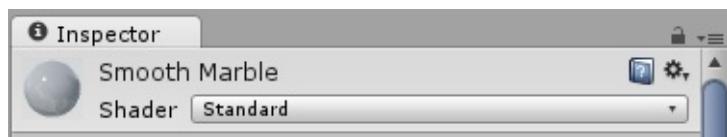


一个被导入纹理，带有 RGB 通道和 Alpha 通道。你可以点击 RGB/A 按钮，来切换预览图的通道。

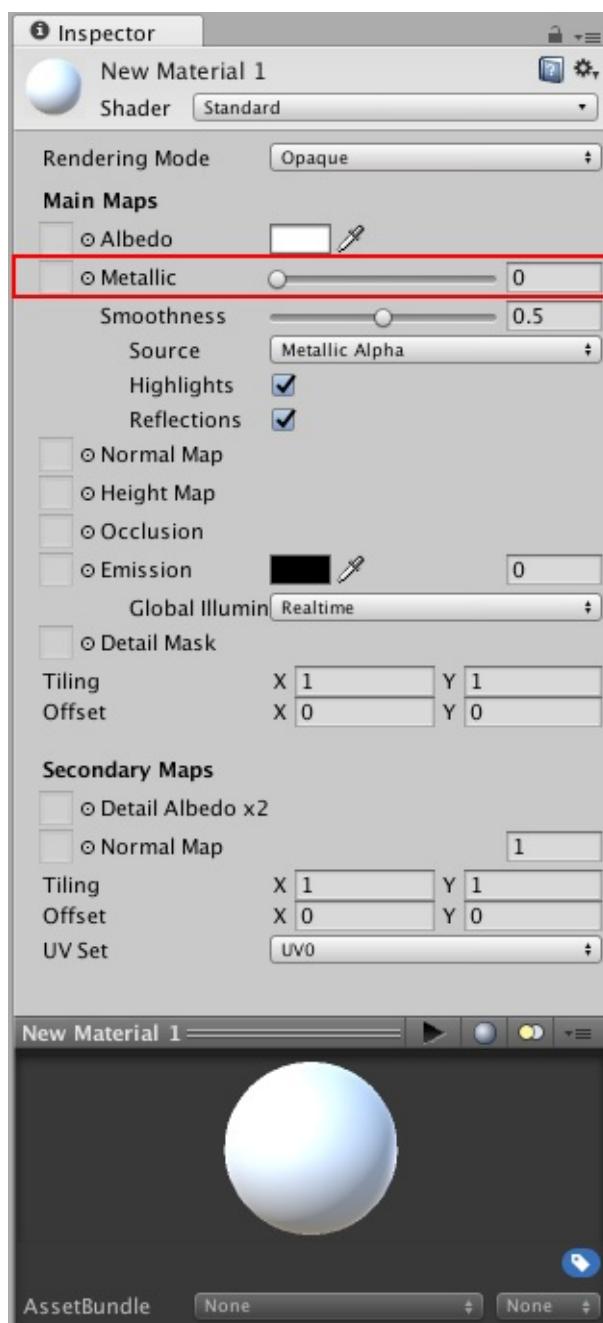


最终效果图，光线穿过破窗户进入建筑物。玻璃中间的分析是完全透明的，而玻璃随便是部分透明，窗框则完全不透明。

金属模式：金属参数



当使用 金属工作流程（相对于镜面工作流程）时，表面的发射率和光照通过金属度和平滑度修改。

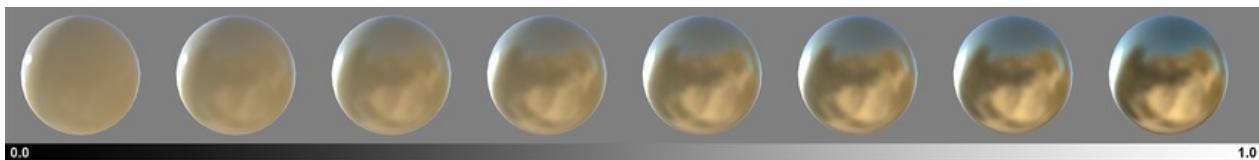


当使用这种流程时，仍然会生成镜面反射，但是看起来是否真实，取决于设置的金属度和平滑度，而不是一个明确定义的开关。

金属模式不仅仅适用于看起来有金属质感的材质！这种模式之所以称为金属模式，是因为你可以通过它控制物体表面的金属或非金属程度（译注：是否是金属或非金属，以及程度）。

金属参数

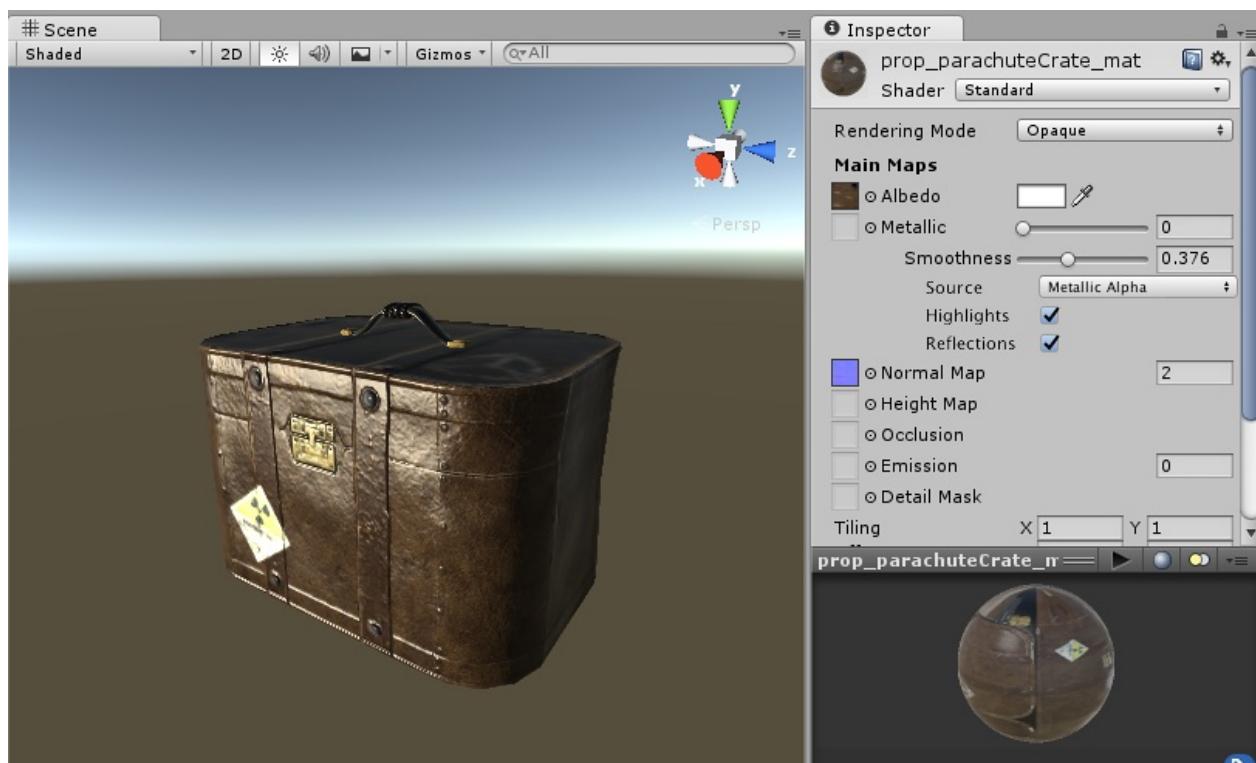
材质的金属参数决定了物体表面的金属度。当表面更像金属时，将反射更多的环境光照，散射颜色变得不可见。当表面是全属性时，表面颜色完全由环境光照驱动。当表面不像金属时，它的散射颜色更加清晰，并且，表面反射的光照更弱，不会使散射颜色模糊。



金属度的范围从 0 到 1（平滑度固定为 0.8）。

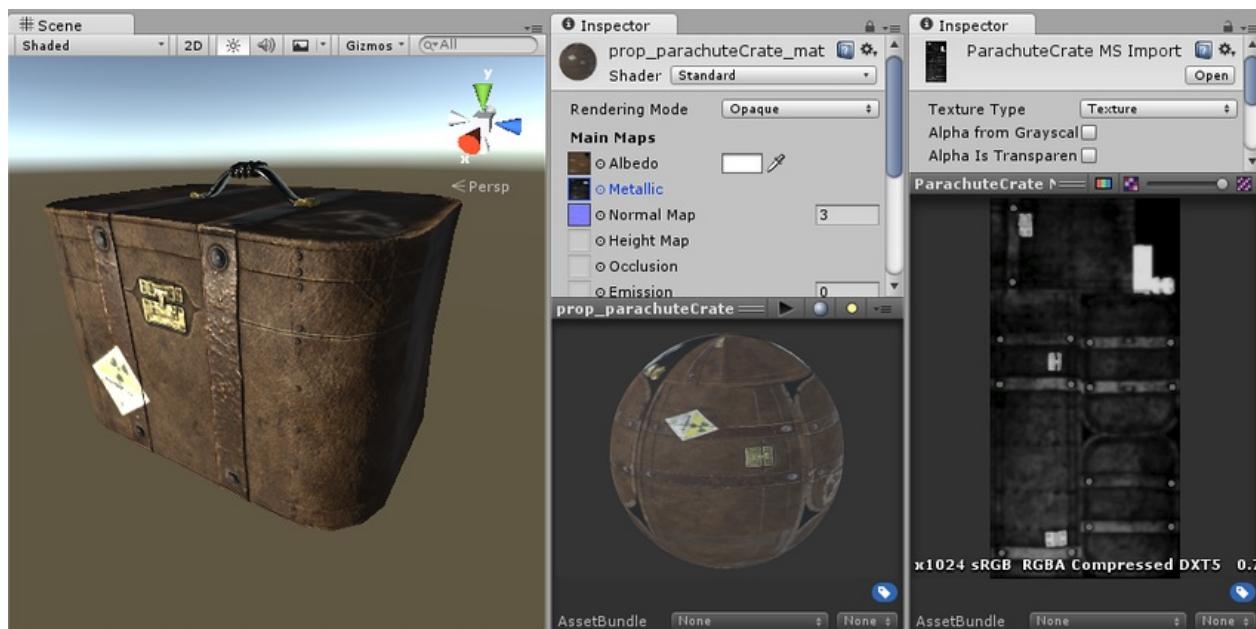
默认情况下是没有指定纹理的，金属度和平滑度参数由一个滑动器控制。这对于某些材质已经足够。但是，如果模型表面是由散射纹理中的多种表面类型混合而成，则可以使用一张纹理贴图来控制整个材质表面的金属度和光滑度。例如，一个角色服饰的散射纹理包含了金属纽扣和拉链。你会希望纽扣和拉链比服饰的布料具有更高金属度。为了实现这一点，不能使用滑动器的值，而是指定一张纹理贴图，其中，在纽扣和拉链区域包含更亮的像素颜色，在布料区域包含较暗的像素颜色。

指定一张纹理后，金属度和平滑度的滑动器将消失。此时，材质的金属度将由纹理的红色通道控制，材质的平滑度将由纹理的 Alpha 通道控制。（这意味着，绿色和蓝色通道被忽略。）这意味着，通过一张纹理，就可以定义粗糙区域、平滑区域、金属区域和非金属区域，当遇到纹理贴图覆盖的区域有不同的需求时 — 皮鞋，布衣，手和脸的皮肤，金属纽扣 — 这种方式非常有用。



这张图像演示了没有金属贴图的情况。

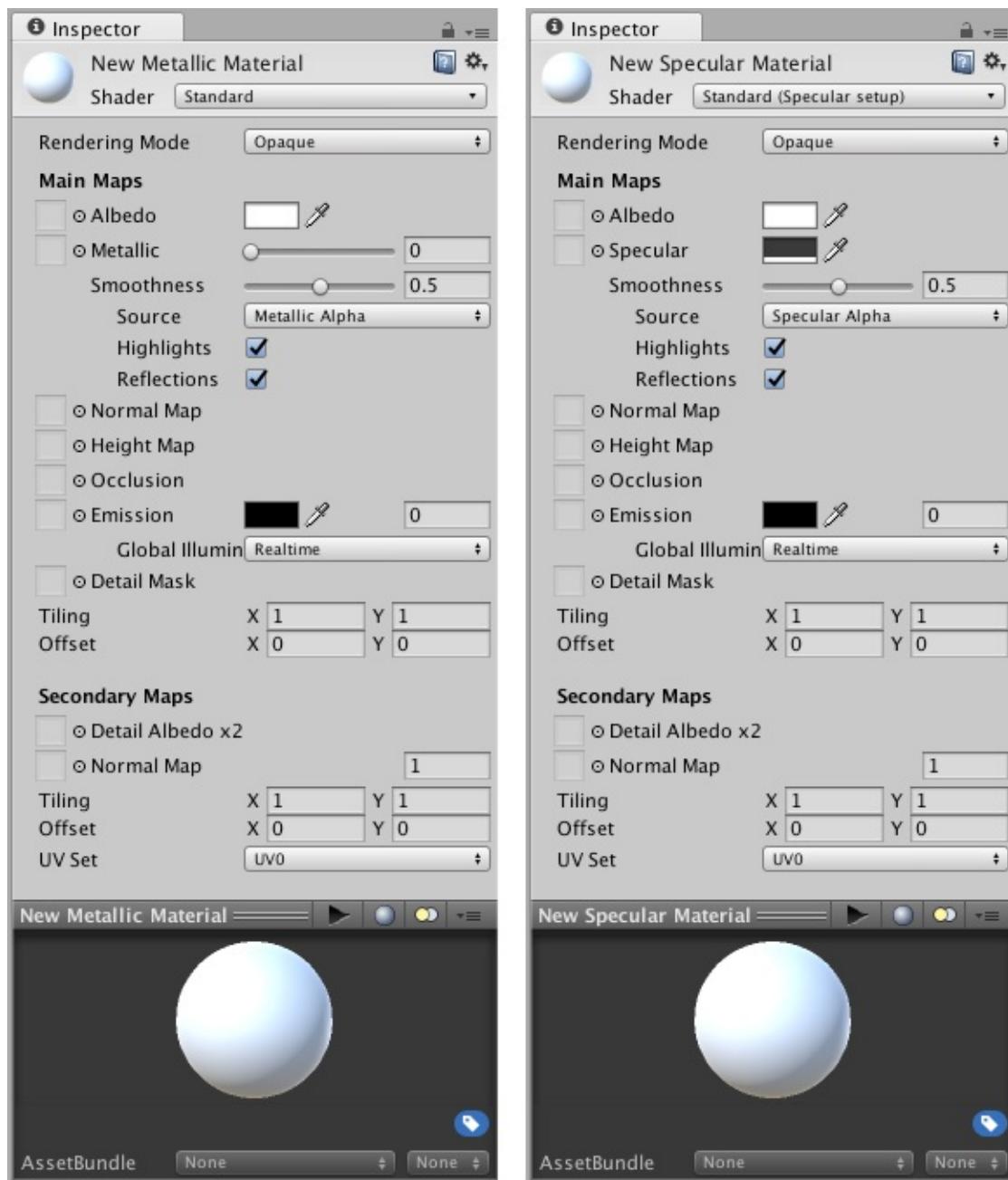
在上面的例子中，材质拥有一个散射贴图，但是没有金属贴图。这意味着，整个物体只有一种金属度和平滑度，显然不理想。皮带、金属带扣、帖子和提手应该呈现不同的表面特性。



这张图像演示了拥有金属贴图的情况。

在这个例子中，指定了一张金属度/平滑度贴图。带扣的金属度更高，响应的光照相应更亮。皮带比皮箱更光亮，尽管它们的金属度都比较低，但是仍然呈现出有光泽的非金属表面，最右侧贴图中的黑色和白色区域用于显示更亮的金属区域，低灰度区域用于显示皮革。

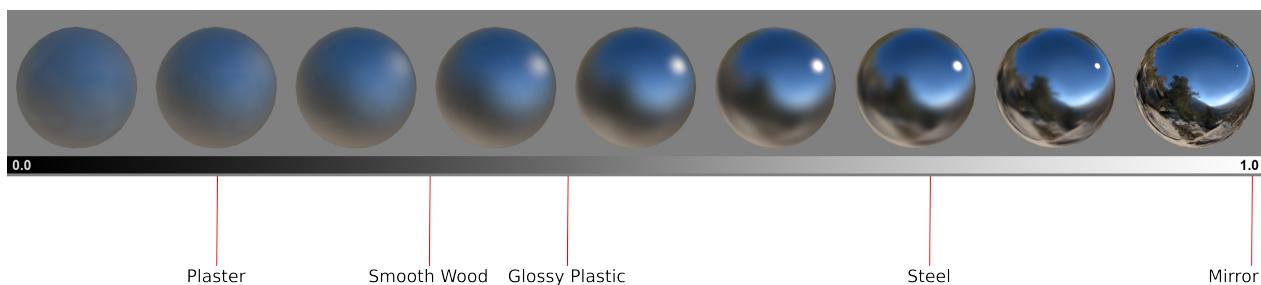
平滑度



平滑度参数，同时存在于标准着色器的金属模式和镜面模式中。

平滑度概念同时支持镜面模式和金属模式，并且工作方式也一样。默认情况下，没有指定金属或镜面纹理贴图，材质的平滑度通过一个滑动器控制。这个滑动器允许你控制『表面微观节』或整个表面的平滑度。

不管是哪种着色器模式，如果为金属属性或镜子属性使用了纹理贴图，平滑度的值将从该贴图中读取。更多细节会在本页的后面介绍。



平滑度的值范围为从 0 到 1。

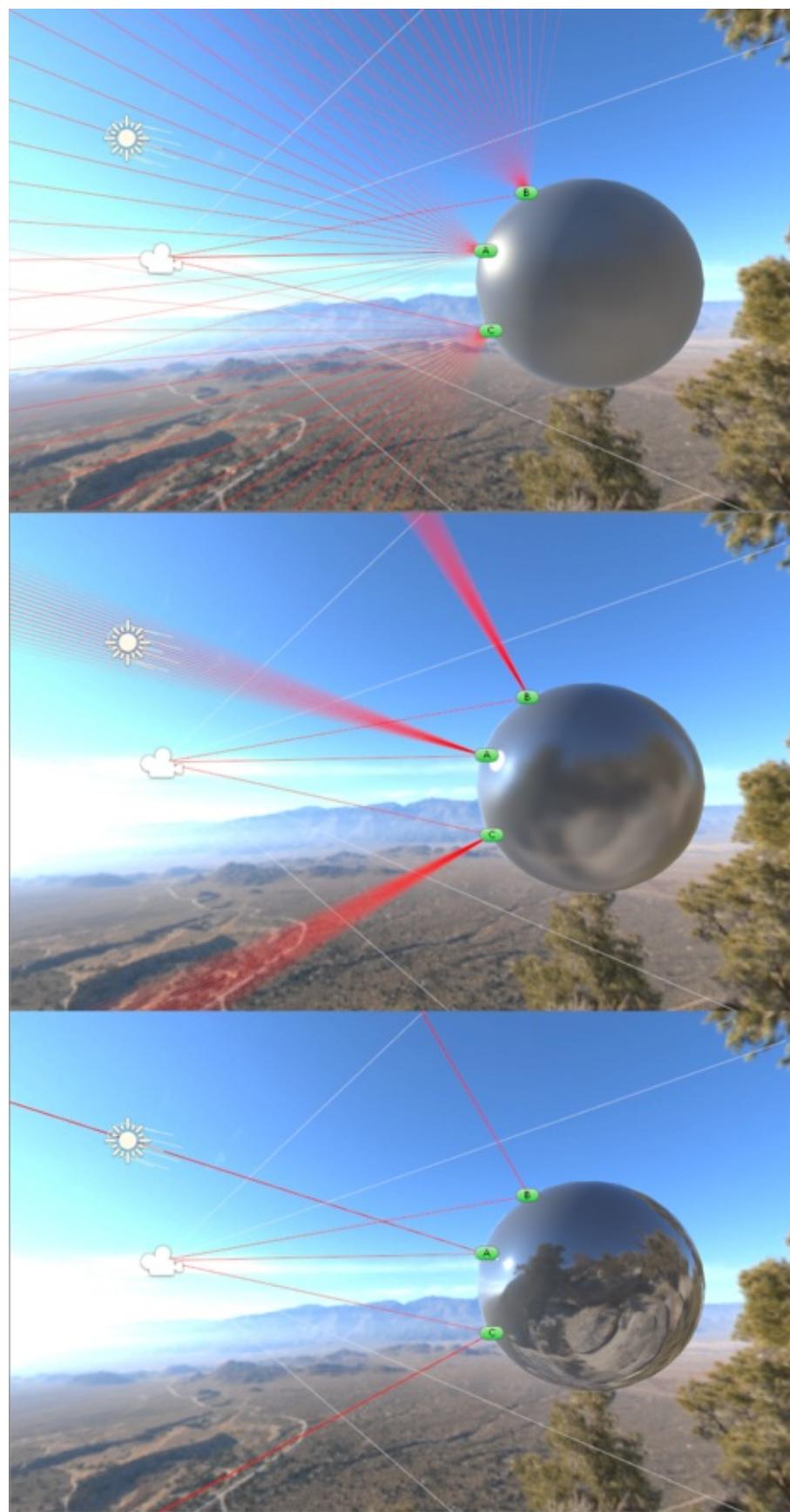
在 Unity 中，『表面微观细节』不是直接可见的东西。它是一个光照计算中使用的概念。尽管如此，当光从物体表面反射时，通过衡量其中漫反射光的数量，可以看到表面微观细节的效果。在光滑的表面上，所有光以可预测和一致的角度反射。极端情况下，一个绝对光滑的表面像镜子一样反射光。不太光滑的表面以更大的角度范围反射光（就像光撞击到微小的凸起），因此，反射光含有的信息更少，并且，以松散的方式散布在整个表面。

译注：漫反射，是投射在粗糙表面上的光向各个方向反射的现象。当一束平行的入射光线射到粗糙的表面时，表面会把光线向着四面八方反射，所以入射线虽然互相平行，由于各点的法线方向不一致，造成反射光线向不同的方向无规则地反射，这种反射称之为“漫反射”或“漫射”。这种反射的光称为漫射光。很多物体，如植物、墙壁、衣服等，其表面粗看起来似乎是平滑，但用放大镜仔细观察，就会看到其表面是凹凸不平的，所以本来是平行的太阳光被这些表面反射后，弥漫地射向不同方向。—来自百度百科



材质表面微观细节在低、中、高平滑度下的理论对比图（从左到右）。黄色线条表示在不同光滑度下，光线撞击表面和反射的角度。

光滑的表面具有非常低的表面微观细节，甚至根本没有，因此光以均匀的方式反弹，产生清晰的反射。粗糙的表面在微观程度上具有高峰和低谷，因此光在宽范围角度内反弹，平均下来，产生不清晰的漫反射颜色。



低、中、高平滑度的对比（从上往下）。

低平滑度时，物体表面每个点的反射光位于广泛的区域中，因为微观细节崎岖不平，光被分散开。高平滑度时，每个点的反射光位于狭窄聚焦的区域，对物体环境的反射更加清晰。

使用平滑度纹理贴图

与许多其他参数一样，你可以设置一张纹理贴图，来代替滑动器的值。这样可以更好地控制物体表面反射光的强度和颜色。

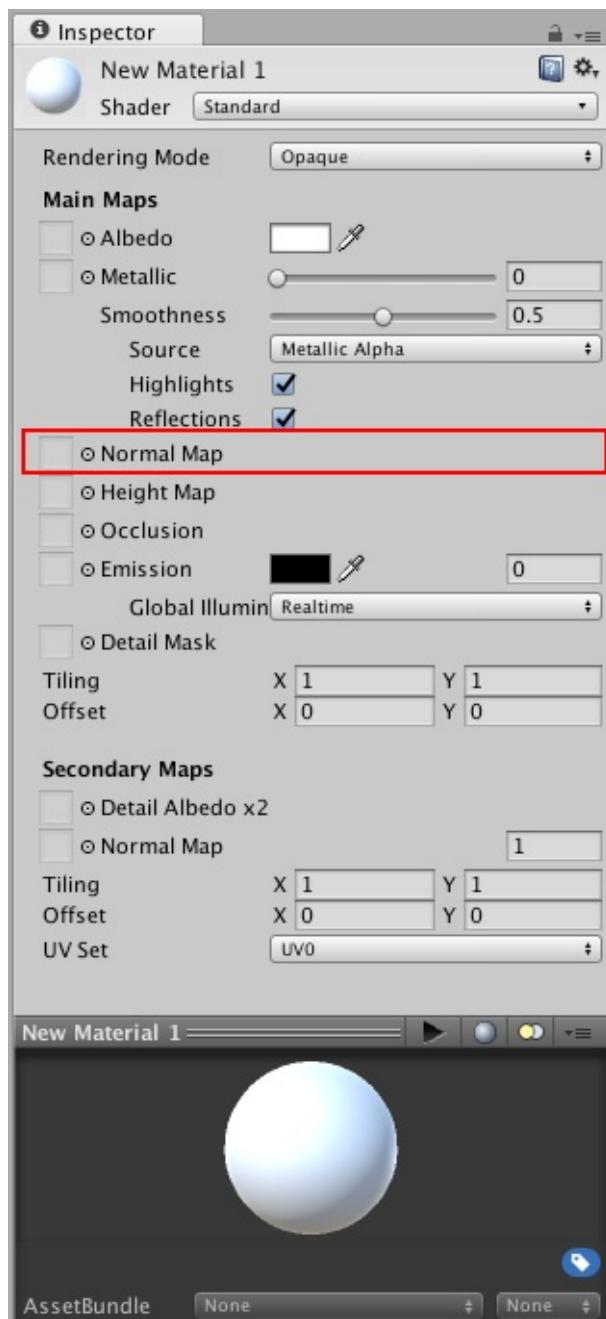
使用贴图代替滑动器，意味着你可以创建包含各种平滑度表面的材质（通常被设计为与漫反射纹理相同）。

属性和功能

- 平滑度来源 选择存储平滑度值的纹理通道。
 - 镜面/金属的 **Alpha** 通道 因为物体表面每个点的平滑度只需要一个值，所以只需要图像纹理的一个通道。因此，平滑度数据被预设存储在金属或镜面纹理贴图的 **Alpha** 通道中（取决于在这两种模式中你使用的是哪一种）。
 - 漫反射的 **Alpha** 通道 使用这个通道，可以减少纹理的总数，或者为镜面/金属的平滑度采用不同分辨率的纹理。
- 高光 选中此框可以禁用高光。这是针对移动设备的性能优化。它从标准着色器中移除高光计算。禁用后对外边的影响主要取决于镜面/金属和平滑度的值。
- 反射 选中此框可以禁用环境反射。这是针对移动设备的性能优化。它从标准着色器中移除高光计算。采用近似计算的方式，代替环境贴图采样。禁用后对外观的影响取决于平滑度。

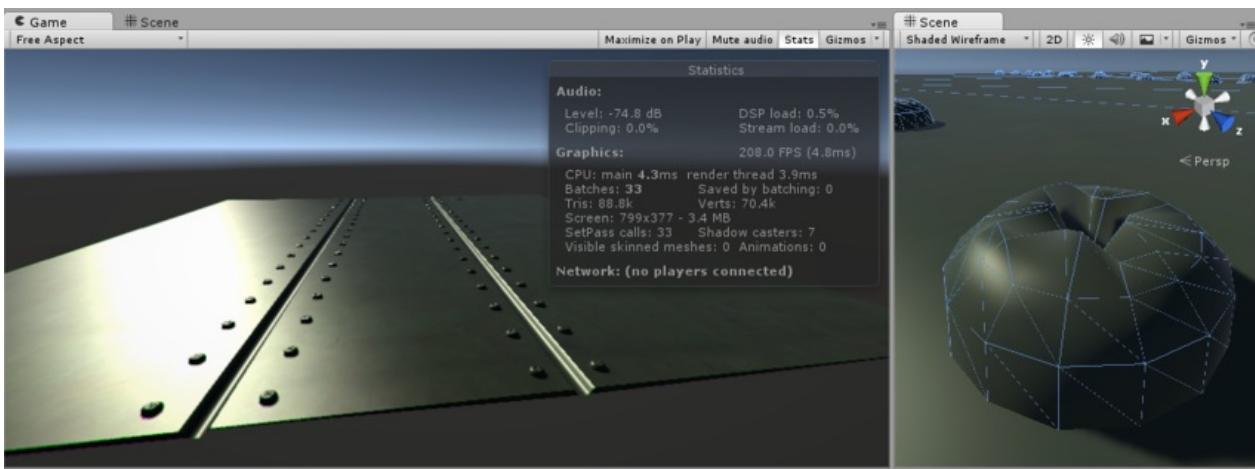
更平滑的表面更具反射性（反射率更高），具有更小、更聚焦的镜面高光。不太平滑的表面不能反射太多光照，所以镜面高光不太明显，并广泛地扩展在整个表面。调整反射率和平滑度贴图，使之与漫反射贴图相匹配，你就可以开始创建爱你非常逼真的问题。

法线贴图（凹凸贴图）



法线贴图是凹凸贴图的一种。凹凸贴图是一种特殊的纹理，允许为模型添加表面细节，例如凸起、凹槽和划痕光，这些细节将捕获光线，就像真实的几何体一样。

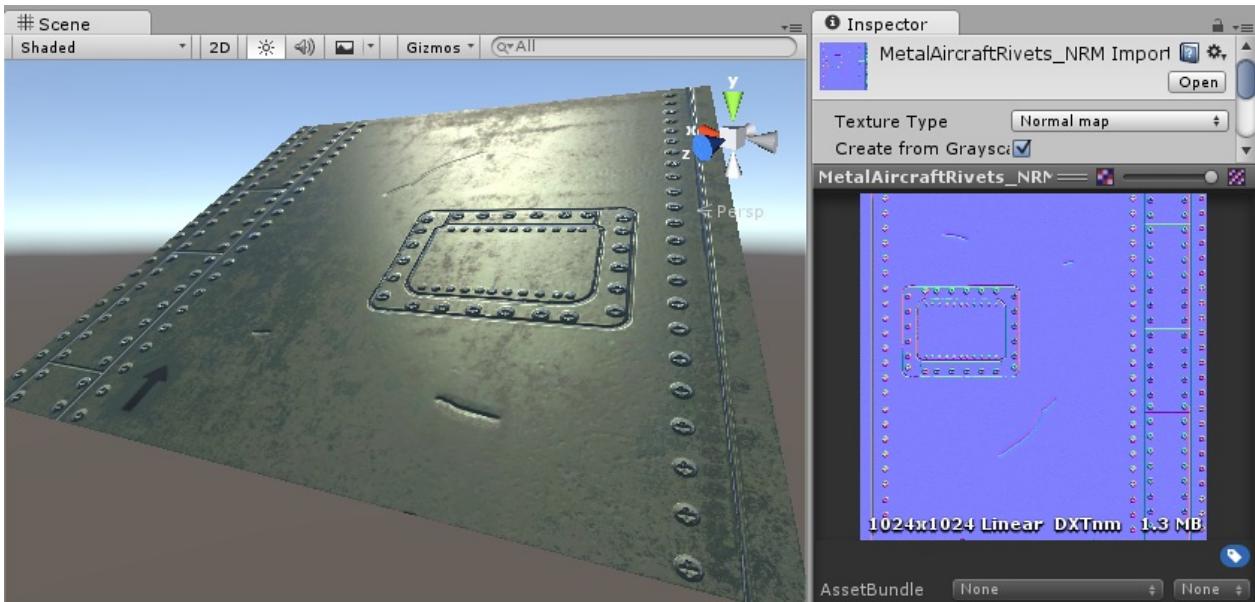
举个例子，你可能想要显示一个具有凹槽、螺丝钉或铆钉的表面，例如飞机机身。一种实现方式是为这些细节进行几何建模，如下所示。



一张飞机金属板，以真实的几何形状为细节建模。

以真实几何形状为如此细微的细节建模通常不是一个好主意。在上图右侧，你可以看到组成单个螺丝头细节所需的多边形。带有大量清晰细节的大型模型，将需要绘制非常多数量的多边形。为了避免这种情况，应该使用法线贴图来表示清晰的表面细节，较大形状的模型使用较低分辨率的多边形表面。

如果使用一张凹凸贴图来呈现细节，表面的几何形状可以变得更加简单，用细节纹理调制光从表面反射的方式。现代图形硬件可以非常快地执行这个过程。现在，金属表面是一个低面数（低多边形）的平面，螺丝、铆钉和划痕将捕获光线，并且因为纹理贴图，它们看起来似乎有了深度。



法线贴图中定义了螺钉、凹槽和划痕，修改了从低面数平面外表反射光线的方式，从而使这些细节呈现出 3D 效果。除了铆钉和螺丝，纹理支持更多的细节，例如微小的凸起和划痕。

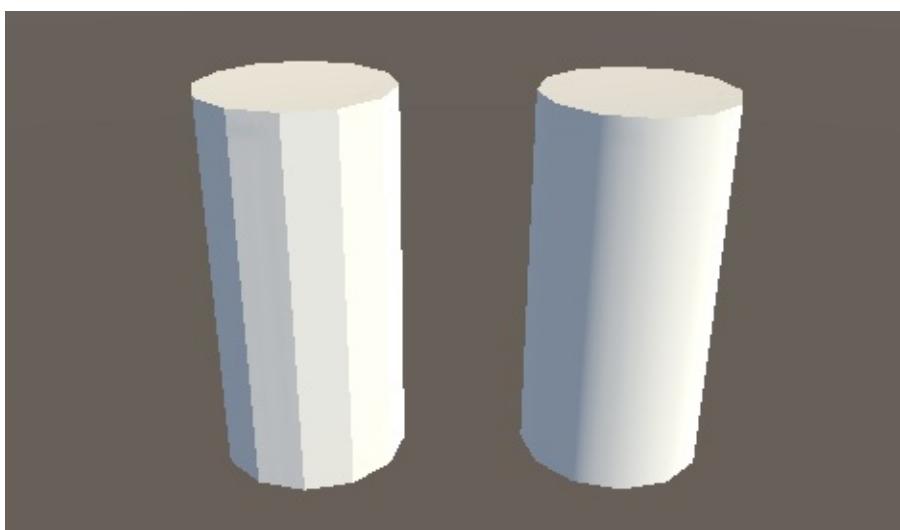
在现代游戏开发流程中，设计师将使用 3D 建模软件，基于非常高分辨率的原始模型生成法线贴图。然后，将法线贴图映射到较低分辨率的模型（游戏中实际使用的模型）上，这样，通过法线贴图，原始的高分辨率细节就被渲染了。

如何创建和使用凹凸贴图

凹凸贴图是一种相对古老的图形技术，但仍然是创建细节逼真的实时图形所需的核心方法之一。凹凸贴图通常也被称为 法线贴图 或 高度图，不过这些术语的含义略有不同，将会在下面解释。

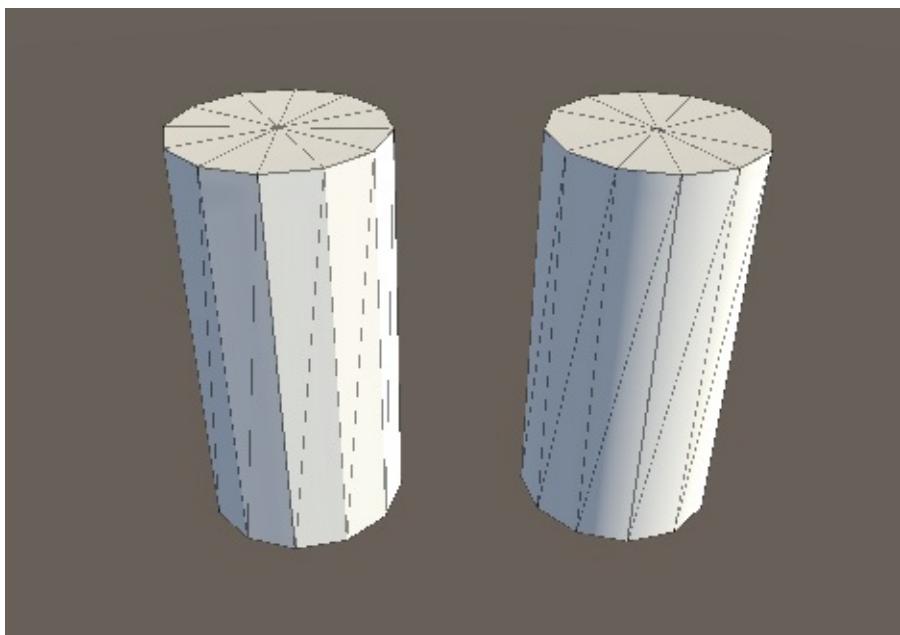
什么是表面法线？

为了真正理解法线贴图的工作原理，需要先清楚什么是 法线，以及如何在实时光照中使用法线。也许最基本的例子是，一个模型的多边形表面根据表面相对于光线的角度简单地被点亮。表面角度可以表示为一条垂直于表面并向上突出的线，是一个向量，这条线成为 表面法线，或者简单地称为 法线。



两个 12 面圆柱体，左边是平面着色，右边是平滑着色。

在上面的图像中，左边的圆柱体具有基本的平面着色，每个多边形按照其相对于光源的角度被着色。因为表面是平坦的，每个多边形区域大小一样，所以，每个多边形上的光照是恒定的。下面是这两个圆柱体的线框网格：



两个 12 面圆柱体，左边是平面着色，右边是平滑着色。

右边的模型具有与左边模型相同数量的多边形，但是着色看起来是平滑的 — 覆盖多边形的光照呈现为一张曲线表面。为什么会这样？原因是，沿着多边形，每个点用于反射光线的表面法线逐渐变化，因此，对于表面上的任意给定点，就好像它的表面真的弯曲了，而不是平坦的恒定多边形，

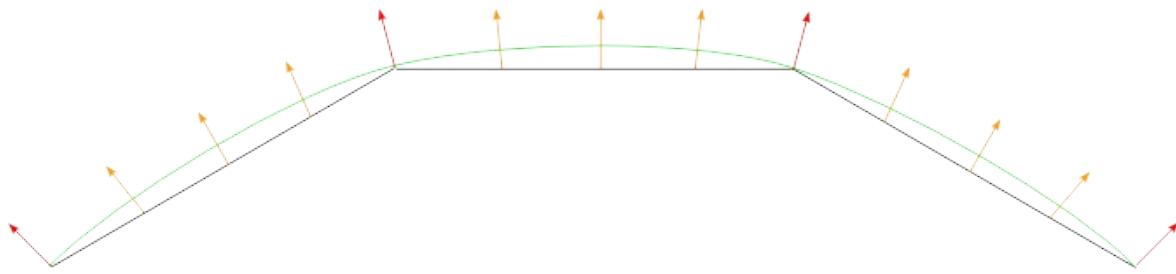
平面着色圆柱体表面上相邻 3 个多边形的 2D 图看起来像下面这样：



3 个平面着色多边形的 2D 图。

表面法线用橙色剪头表示。它们的值用于计算表面如何反射光线。在每个多边形上，因为表面法线的方向相同，所以反射同样的光线。所以左侧圆柱体呈现出平面着色，并且具有硬边缘。

然而，对于平滑着色圆柱体，表面法线沿着平面多边形逐渐变化，如下所示：



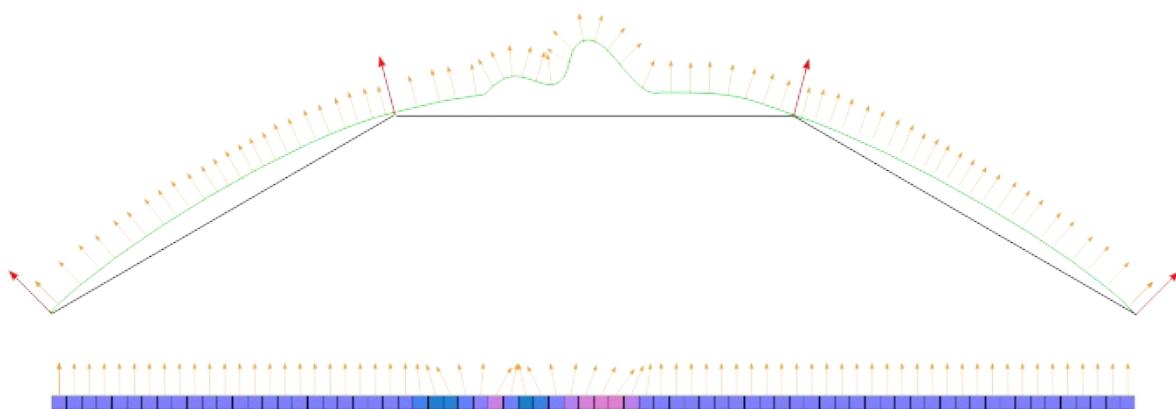
3个平滑着色多边形的2D图。

法线方向沿着平面多边形表面逐渐变化，所以看起来像是平滑曲面（用绿线表示）。这并不会影响网格的多边形特质，只会影响表面的光照计算。这种视觉上的曲线表面并不是真正存在，在掠射角观察时将显示平面多边形的本质，尽管如此，从大多数视角查看，这个圆柱体看起来具有平滑的曲线表面。

使用这种基本的平滑着色时，决定法线方向的数据实际上只存储每个定点，所以，沿着表面变化的法线数据其实是相邻两个定点之间的插值。在上图中，红色剪头表示每个顶点存储的法线方向，黄色剪头表示多边形区域的法线方向插值示例。

什么是法线贴图？

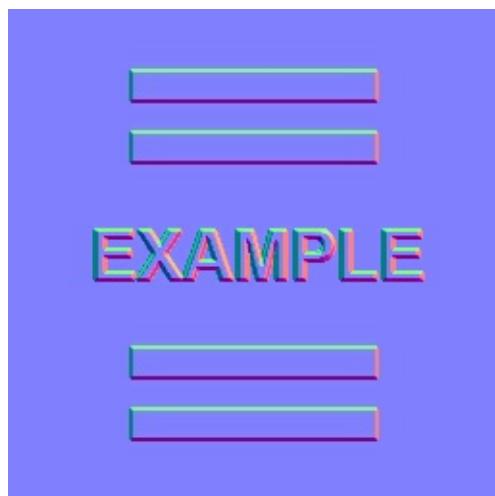
更进一步，用法线贴图表示表面法线的这种变化。法线贴图通过通过一张纹理来存储如何改变表面法线的信息。法线贴图是一张映射到模型表面的图像纹理。有些类似于常规的颜色纹理，不同的是，法线贴图中每个顶点（称为纹素）还表示了在表面法线方向上离开平面（或平滑）多边形真实表面的距离。



3个相邻多边形的法线贴图的2D图。

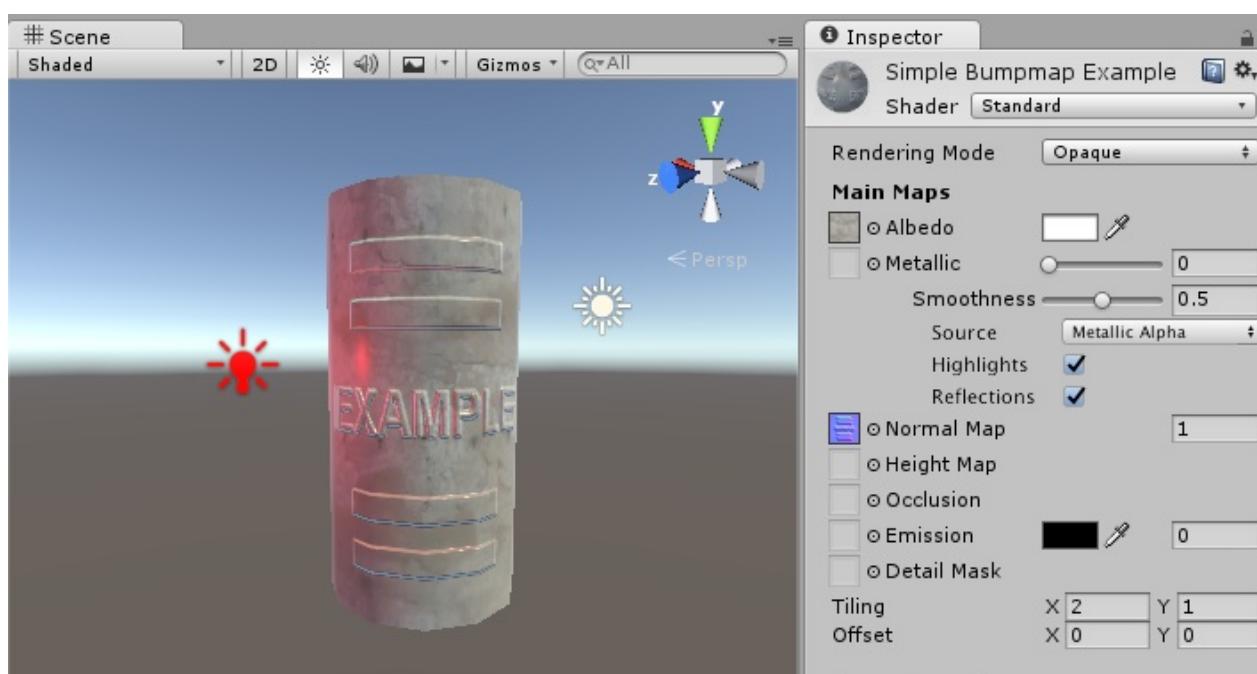
在这张 3D 模型表面的 3 个相邻多边形的 2D 示意图中，每个橙色剪头对应法线贴图中的一个像素。下面是法线贴图的每个像素切片。在中间位置，你可以看到法线被改变了，在多边形表面上呈现出多个凸起。只在当光照出现在表面上时，这些凸起才会显示，因为这些被改变的法线只用于光照计算。

原始法线贴图文件中的可见颜色呈现蓝色色调，并且不含有任何光照或阴影着色 — 这是因为这些颜色本身并不会显示。相反，每个纹素的 RGB 值表示的是一个方向矢量的 X、Y、Z 值，用于修改多边形表面平滑法线的插值。



一个法线贴图示例。

这是一张简单的法线贴图，包含了一些矩形和文本的凹凸信息。这张发现贴图可以导入 Unity，并放入标准着色器的法线贴图插槽。把它和颜色贴图（即漫反射贴图）一起应用在圆柱体网格的表面上时，效果如下所示：



把这张法线贴图应用到上面的圆柱体网格表面。

同样不会影响网格的多边形特质，只是用于计算表面的光照。这种视觉上的字符和形状凸起并不真正存在，在掠射角观察时讲现实平坦表面的本质，尽管如此，在大多数视角查看，这个圆柱体表面呈现出凸起的浮雕细节。

如何获取或制作法线贴图？

通常情况下，法线贴图是同 3D 或纹理设计师制作的模型或纹理一起生成的，通常反应了漫反射贴图的结构和内容。有时也会手工制作，有时由 3D 软件渲染生成。

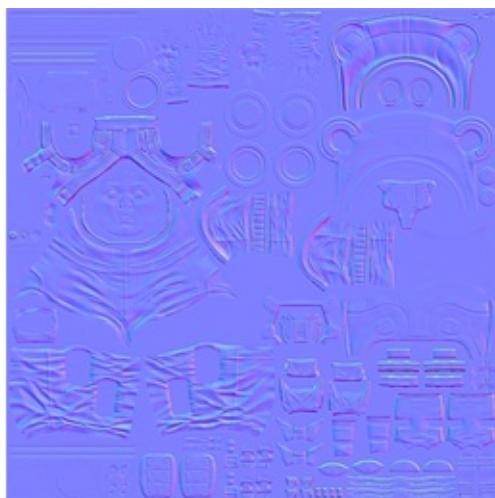
3D 软件如何渲染法线贴图超出了本文档的范围，但是基本的概念是，3D 设计师会为一个模型生成两个版本——一个包含了所有多边形细节的超高分辨率模型，和一个程序用的低分辨率模型。在游戏中，高分辨率模型含有太多细节（网格中有太多的三角形），以至于不适合在游戏中运行，但是用于在 3D 模型软件中生成法线贴图。低分辨率模型则可以运行在游戏中，并且忽略那些存储在法线贴图中的非常清晰的几何细节，几何细节则用法线贴图渲染。一个典型的用例是，在角色服饰上现实折痕、纽扣、带扣和接缝。

有些软件包可以分析普通摄影纹理中的光照信息，并从中提取法线贴图。工作原理是，假设一束平行光照射到原始纹理上，然后分析浅色和暗色区域，作为法线的角度。不过，在真正使用凹凸贴图时，你需要确保漫反射纹理中不含有任何方向的光照——理想情况下，它应该呈现完全无光情况下的表面颜色——因为光照信息由 Unity 根据光照角度、表面角度和凹凸贴图计算。

下面是两个示例，一个是简单重复的石墙问题以及对应的法线贴图，一个是角色的纹理图集以及对应的法线贴图。



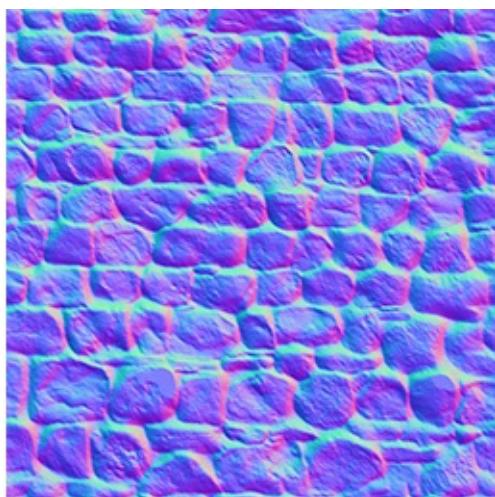
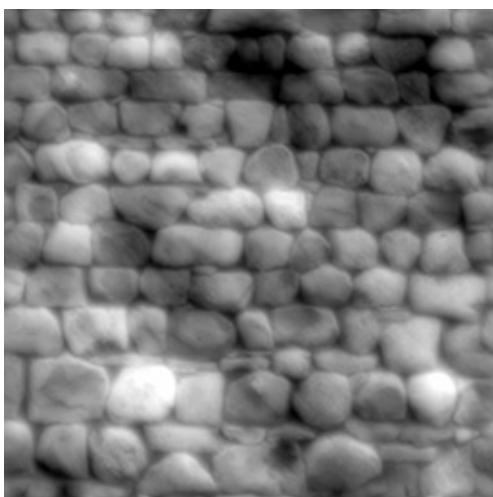
一个石墙纹理及其对应的法线贴图。



一个角色纹理图集及其对应的法线贴图集。

凹凸贴图、法线贴图和高度图有什么区别？

法线贴图和高度图都属于凹凸贴图类型。它们都含有用于简化多面性网格的表面视觉细节，但是以不同的方式存储数据。



左边是映射石墙凹凸信息的高度图，右边是映射石墙凹凸信息的法线贴图。

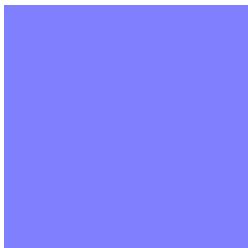
在左图中，可以看到一张映射石墙凹凸信息的高度图。高度图是一张简单的黑白纹理，每个像素代表了表面上该点应该凸起的高度，像素点颜色越白，该区域看起来更高。

法线贴图是一张 RGB 纹理，每个像素代表了表面相对于真实法线（垂直）的朝向（夹角）。这些纹理往往是蓝紫色调，因为矢量用 RGB 值存储。

现代实时 3D 图形硬件更信赖（依赖）法线贴图，因为法线贴图包含了表面应该如何弹射光线所需的矢量信息。Unity 的凹凸贴图也可以接受高度图，但导入必须转换为法线贴图才能使用。

为什么是蓝紫色？

对于使用法线贴图，理解为什么并不重要！跳过这一段也没关系。但是，如果你真的想知道的话：RGB 颜色值用于存储矢量的 X、Y、Z，并且 Z 轴向上（与 Unity 使用的 Y 轴向上惯例相关）。此外，纹理中的值被减半处理，并加了 0.5。这样可以存储所有方向的矢量。所以，要将 RGB 值转换为矢量方向，必须乘以 2 然后减 1。例如，RGB 值 (0.5, 0.5, 1) 或者十进制 #8080FF 转换后是 (0,0,1)，表示垂直的法线贴图 — 表示模型表面无变化。这是本页前面演示法线贴图平坦区域时的颜色。



一张只使用 #8080FF 的法线贴图，转换为矢量是 0,0,1，即垂直向上。这张贴图不会改变多边形的表面向量，因此光照不会变化。任何与该颜色不同的像素点，都会导致矢量指向不同的方向 — 改变后的角度用于计算该像素点上反射光的角度。

RGB 值 (0.43, 0.91, 0.80) 给出一个矢量 (-0.14, 0.82, 0.6)，表示表面非常的陡峭。可以在石墙法线贴图的亮青色区域看到这种颜色，对应于一些石头的边缘。相对于石头上较平坦的表面，这些边缘以非常不同的角度捕获光线。

译注：rgb(110,232,204)



这些石头的法线贴图中的亮青色区域显示了对每个石头顶部边缘的多边形表面法线的陡峭修改，使得它们以正确的角度捕获光线。在这些石头的法线图中，亮青色区域对每个石头顶部边缘的多边形表面法线进行了陡峭的修改，使得它们以正确的角度捕获光线。

法线贴图



一堵没有凹凸效果的石墙。岩石的边缘和表面不会捕捉场景中的平行太阳光。



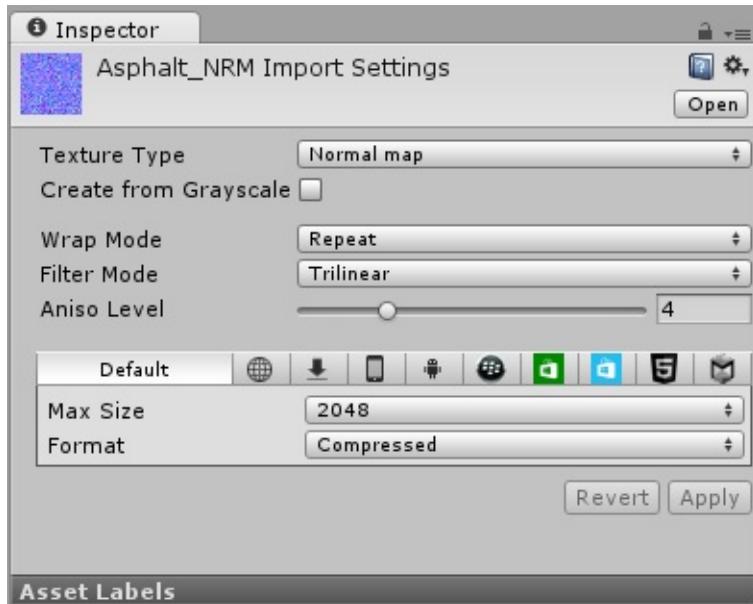
应用了凹凸贴图的同一堵石墙。岩石面向太阳的边缘反射的平行太阳光，与岩石的表面和不朝向太阳的边缘，非常不同。



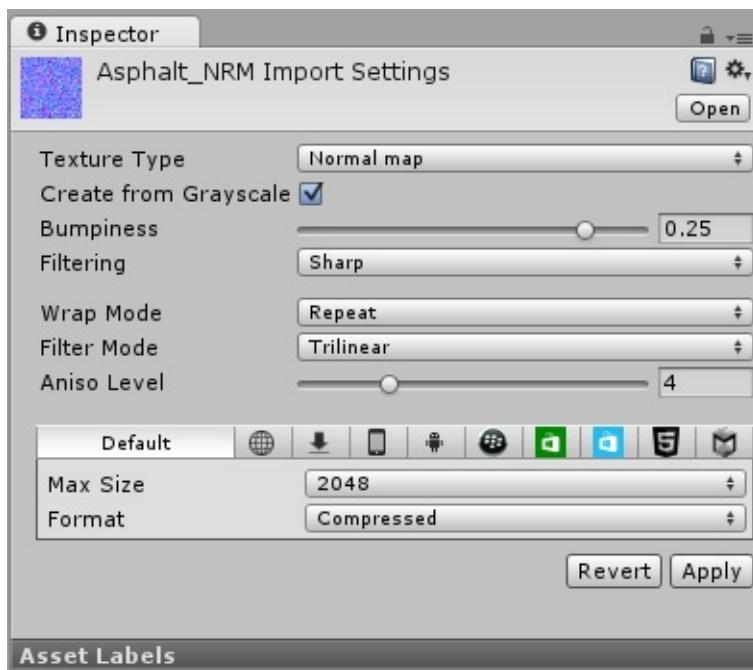
不同光照情况下，应用了凹凸贴图的同一堵石墙。一个火把点光源照亮了石头。光线撞击基础模型（多边形）的角度，被法线贴图中的矢量所调整，然后点亮石头的每个像素。因此，面向光源的像素是亮的，背光的像素较暗或在阴影中。

如何导入和使用法线贴图和高度图

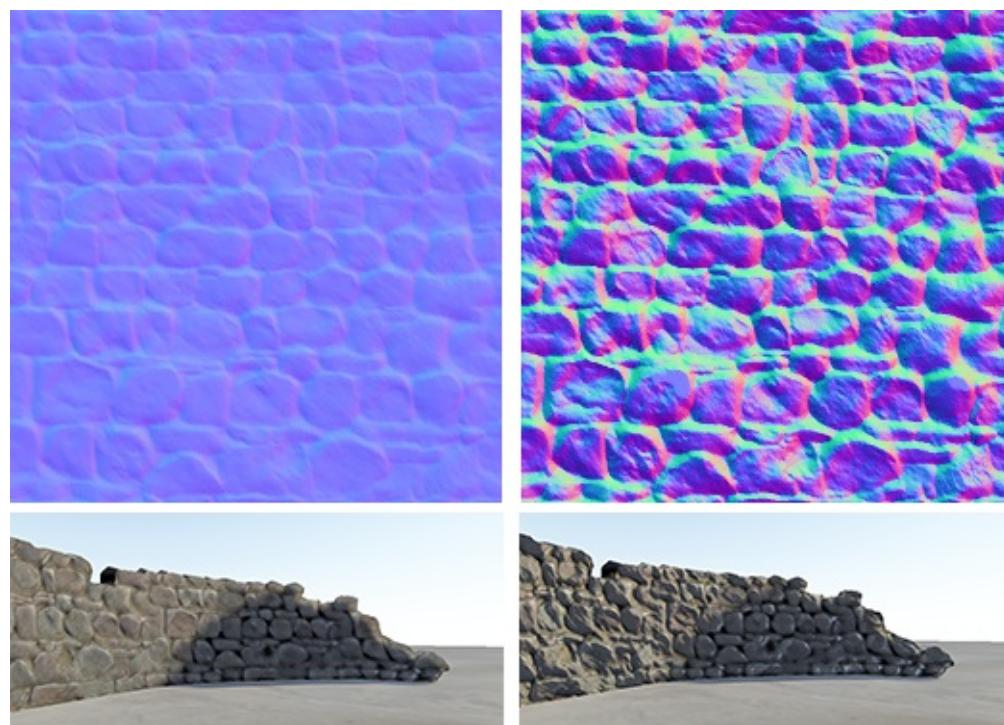
像平常一样，把纹理文件放入 Assets 文件夹，就可以导入法线贴图。不过，你需要告诉 Unity 这个纹理是一张法线贴图。可以在导入资源的检视视图中，把改变『纹理类型』为『法线贴图』。



把黑白高度图导入为法线贴图的过程，与直接导入法线贴图几乎完全相同，除了需要选中复选框『Create From Greyscale』。

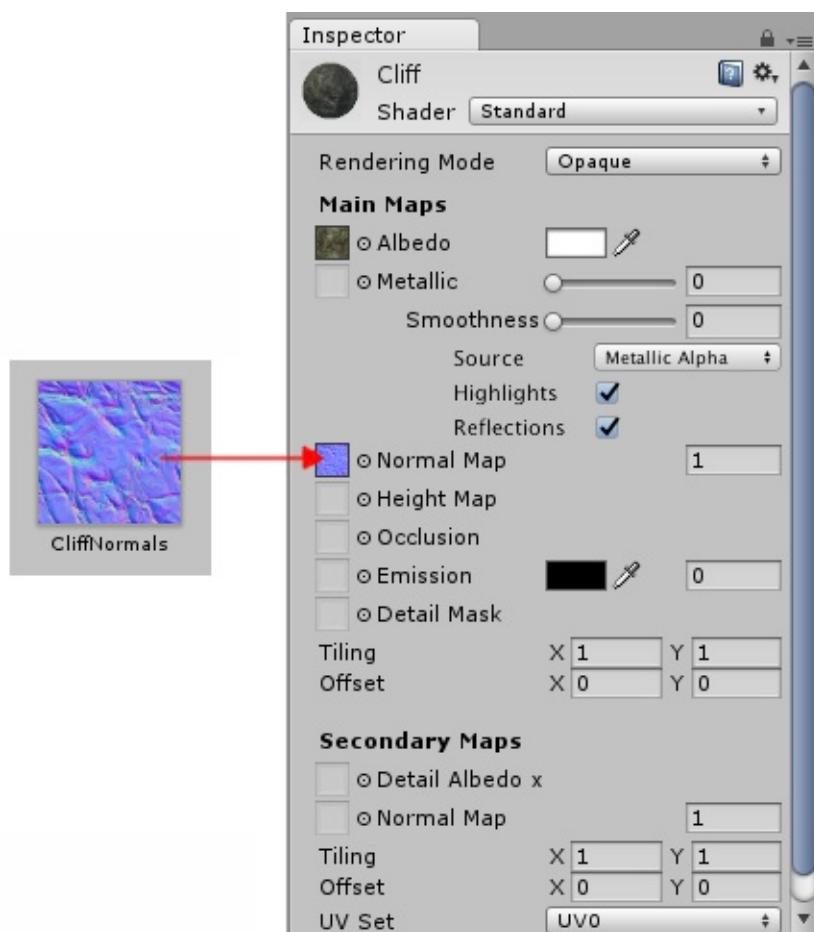


选中『Create From Greyscale』后，将出现一个凹凸度滑动器。你可以使用这个滑动器，来控制高度图的高度转换为法线贴图中的角度时的陡峭程度。低凹凸度将意味着，即使是高度图中的强烈对比，也将被转换为平缓的角度和凹凸。高凹凸度将创建夸张的凹凸，产生高对比度的光照响应。



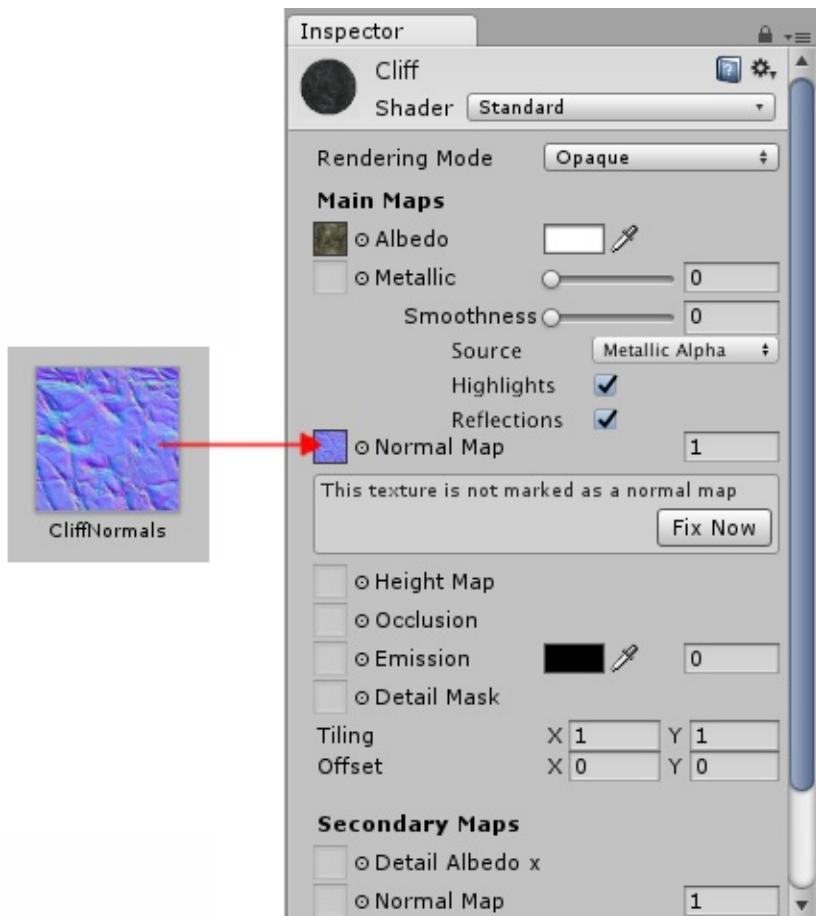
把高度图导入为法线贴图时，凹凸度高低对模型的影响。

Assets 文件夹中有了法线贴图后，就可以在检视视图中把它放入材质的法线贴图插槽。标准着色具有一个法线贴图插槽，许多旧的传统着色器也支持法线贴图。



把一张法线法线贴图放入标准着色器材质的正确插槽中。

如果导入了一张法线贴图或高度图，但是没有把它标记为法线贴图（选择纹理类型：法线贴图），材质的检视视图将提醒你修正它，像这样：



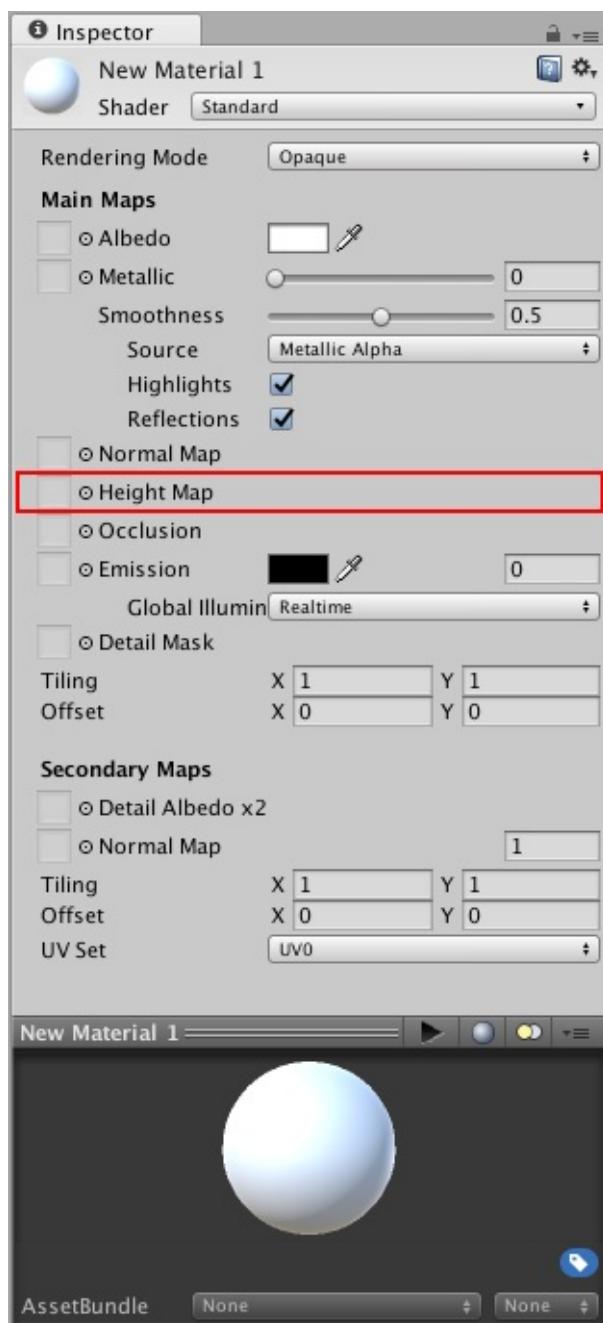
当尝试在法线贴图插槽上使用未标记为法线贴图的纹理时，将显示『立即修复』警告。

点击『立即修复』，与在纹理检视视图中选择纹理类型：法线贴图，具有相同的效果。如果纹理是一张法线贴图，这么做是有效的。但是，如果它其实是一张灰阶高度图，就无法自动检测到——所以对于高度图，你必须总是在纹理的检视视图中选中『Create from Greyscale』。

辅助法线贴图

你可能还注意到，在标准着色器材质的底部有一个辅助法线贴图插槽。它允许你再使用一张法线贴图来创建额外的细节。你可以把一张法线贴图添加到这个插槽中，就像常规法线贴图的插槽一样，但是，你应该使用不同的尺寸或平铺频率，这样，两张法线贴图在不同的尺度上，共同产生高级细节。例如，常规法线贴图可以定义墙壁或车辆上的镶板细节和镶板边缘的凹槽。辅助贴图可以为表面上划痕和磨损提供非常精细的凸起细节，平铺次数可能是基础法线贴图的 5 到 10 倍。这些细节可能非常惊喜，只有贴脸检查才能看到。为了在基础法线贴图上达到这个量级的细节，基础法线贴图需要非常大才行，尽管如此，通过两张不同尺寸但相对较小的法线贴图，可是实现高级细节。

高度图

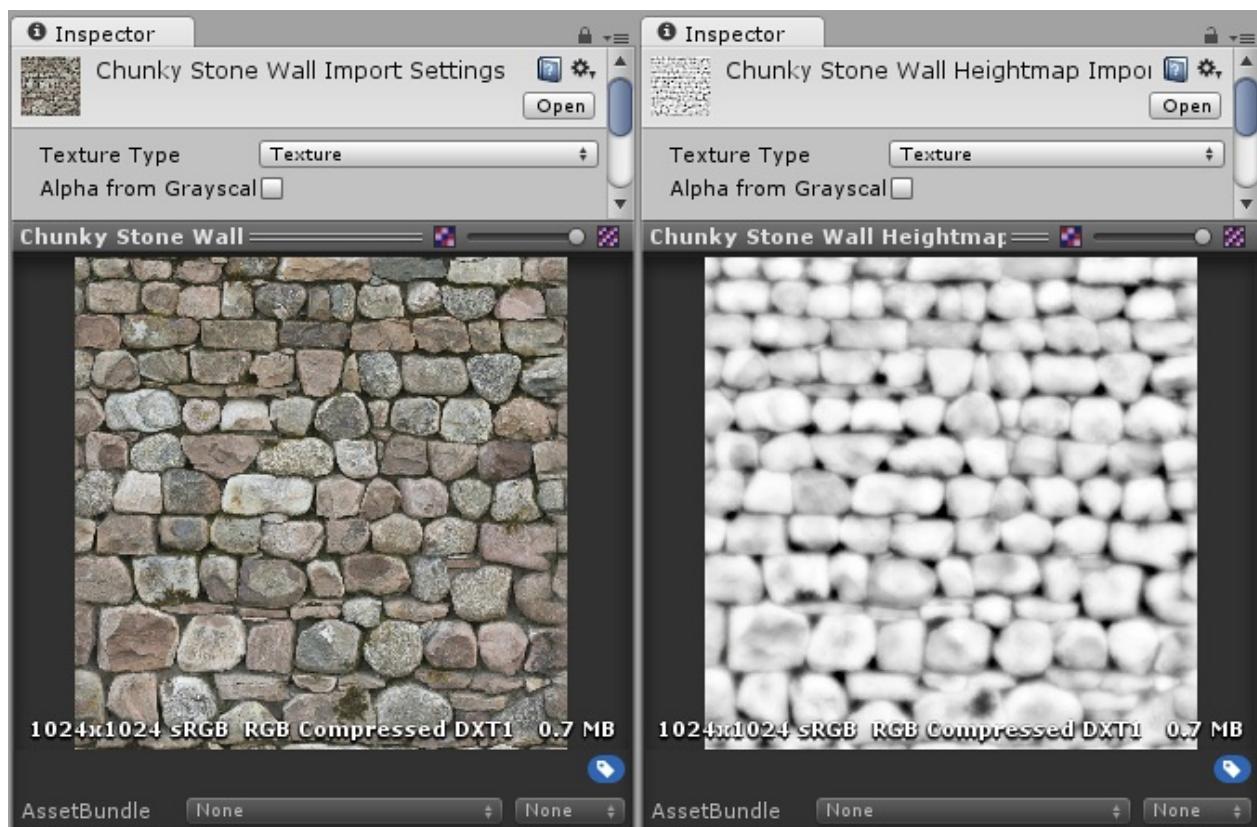


高度图（也称为视差贴图）是一个类似于法线贴图的概念，但是技术更复杂 — 因此更性能开销更大。高度图通常与法线贴图结合使用，负责定义和渲染表面额外的大型凸起。

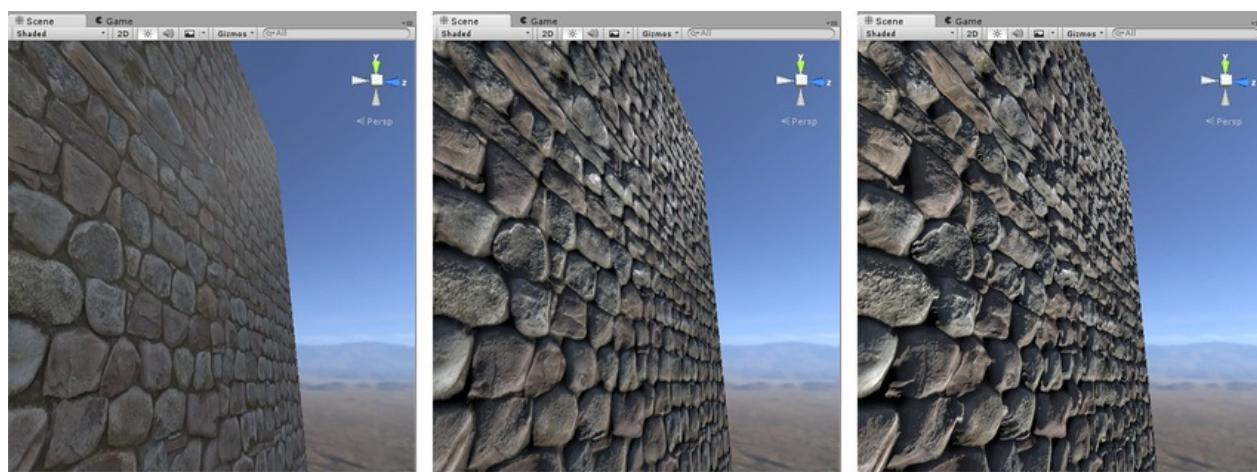
虽然法线贴图改变了纹理表面的光照，但是视差高度图更进一步，它会移动表面纹理的可见区域，从而实现表面遮挡效果。这意味着，明显的凸起将具有放大的正面（面向相机）和缩小的反面（背向相机），并且反面会被遮挡住。

虽然可以产生非常逼真的 3D 几何效果，但是仅限于物体网格的平面多边形表面。这意味着，表面凸起将会突出现实并且彼此遮挡，但是并不会改变模型的『轮廓』，因为最终效果是绘制在模型表面，并不会实际的几何形状。

一张高度图应该是一张灰阶图，白色区域表示纹理的高区域，黑色区域代表低区域。下面是一张典型的漫反射贴图和对应的高度图。



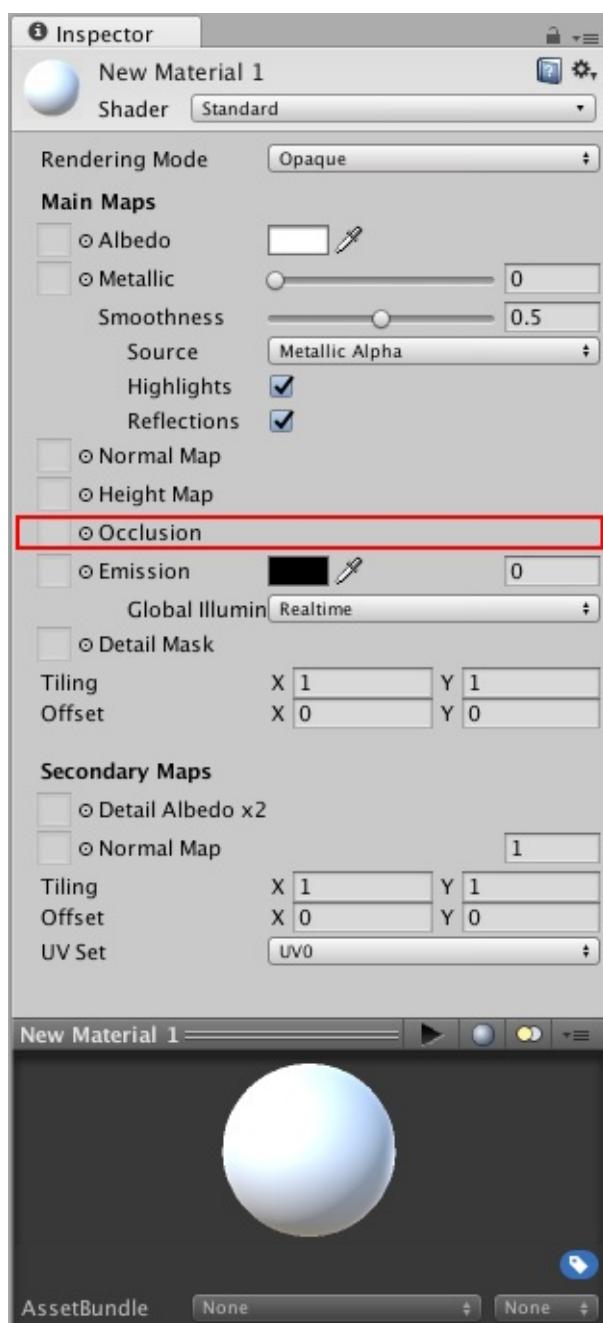
一张漫反射颜色贴图，和一张对应的高度图。



上图从左往右：1. 石墙材质被指定了漫反射贴图，但是没有法线贴图和高度图。2. 指定了法线贴图。表面光照被改变，但是岩石避免没有遮挡。3. 设置了法线贴图和高度图的最终效果。岩石看起来从表面凸起，近处岩石看起来遮挡了它们后面的岩石。

用于高度图的灰阶图，通常（但不总是）也可以很好地用于散射贴图。关于散射贴图请参阅下一节。

散射贴图



散射贴图用于提供模型某些区域应该接收的非直接光照的数量信息。非直接光照来自环境光和反射光，因此，模型的高陡度凹陷部分，例如裂缝或褶皱，不会接收太多的非直接光照。

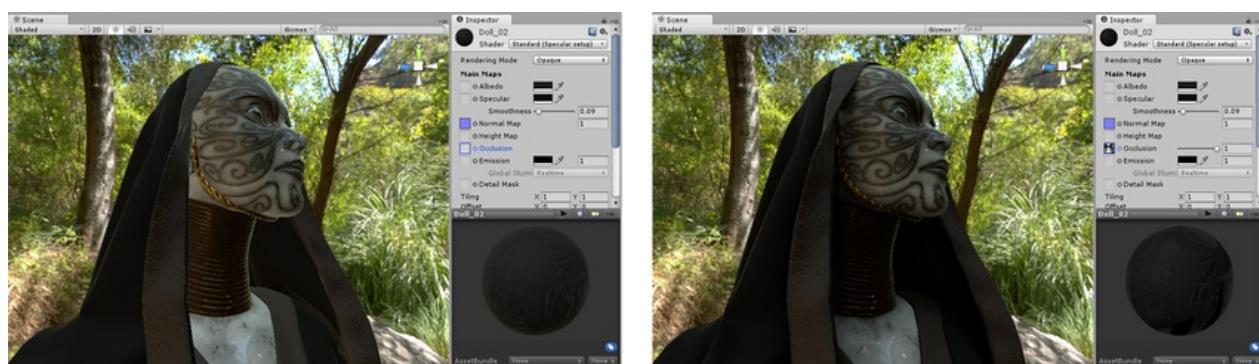
散射纹理贴图通常由 3D 应用程序计算，由建模器或第三方软件从 3D 模型中直接导出。

散射贴图是一张灰阶图，其中白色表示应该完全接受非直接光照，黑色表示没有非直接光照。对于简单表面，它像灰阶高度图一样简单（例如前面高度图示例中的不规则石墙纹理）。

在其他时候，要生成正确的散射贴图稍微有些复杂。例如，场景中的角色戴了头罩，头罩内边缘的非直接光照应该设置的非常低，甚至完全没有。在这些情况下，散射贴图经常由设计师使用 3D 应用程序基于模型自动生成。

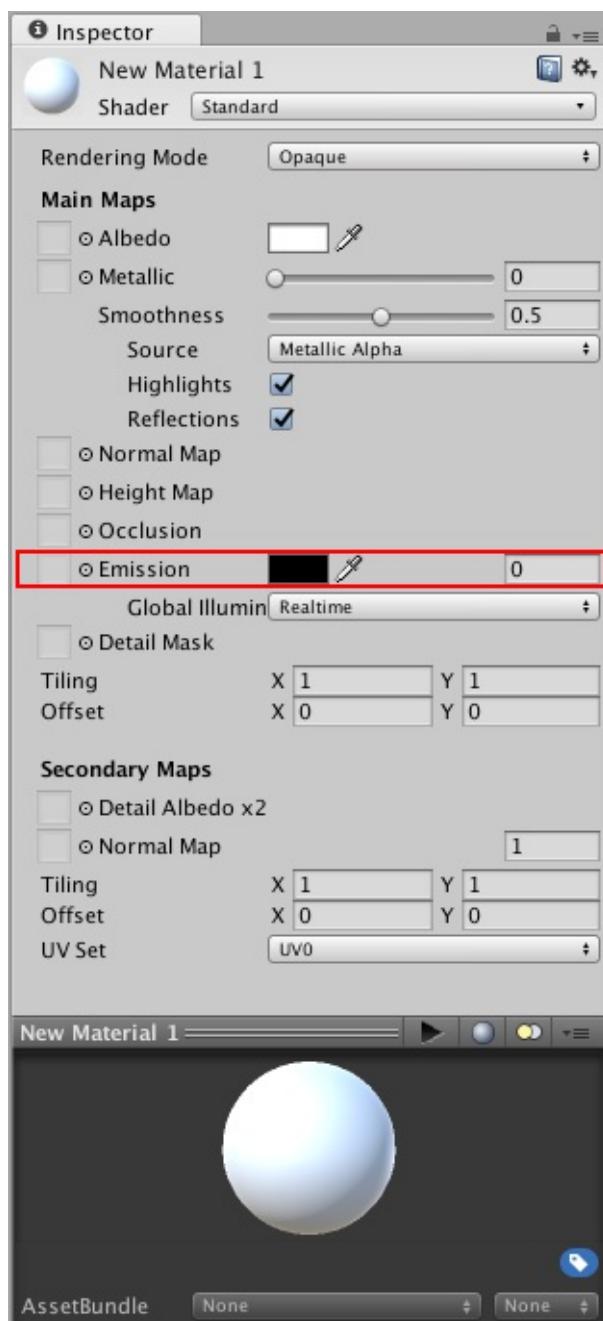


散射贴图标识了在环境光照下角色头罩下暴露或隐藏的区域。它用于下面所示的模型。



散射贴图应用前和应用后。在左图中，部分被遮挡的区域，特别是围绕颈部的衣服褶皱，被照的太亮了。指定散射贴图后，这些区域不再被来周围森林的绿色环境光所照亮。

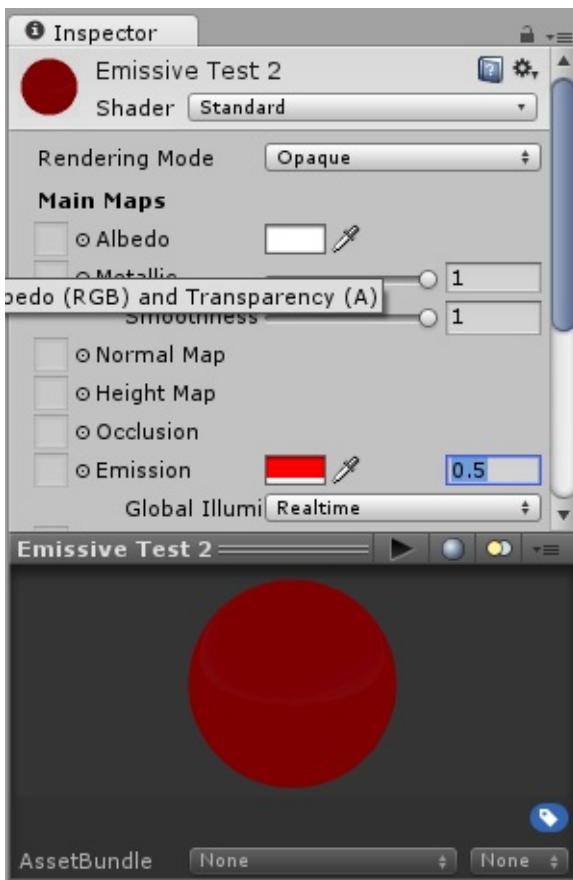
自发光



控制表面发射光的颜色和亮度。当场景中使用了自发光材质时，它看起来像一个可见光。物体将呈现『自发光』效果。

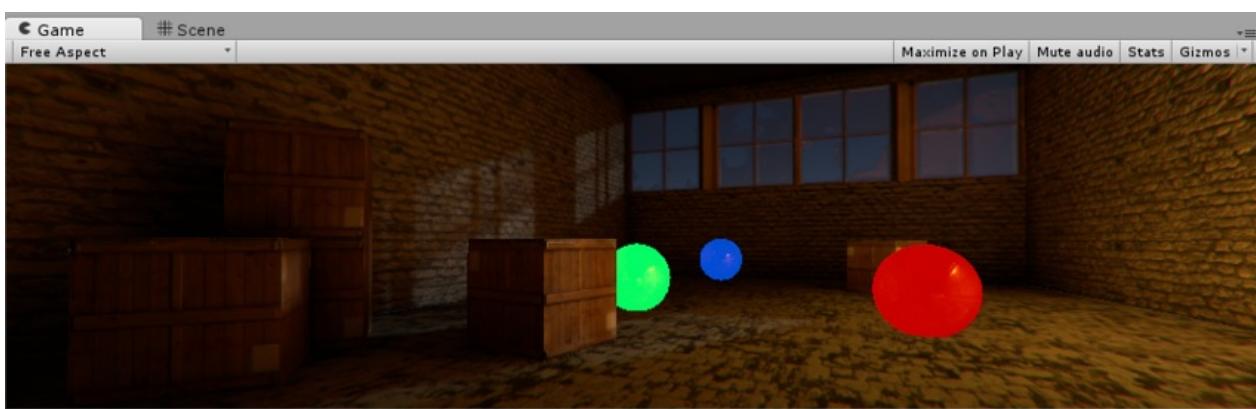
自发光材质通常用于某些部位应该从内部照亮的物体上，例如监视器屏幕、高速制动的汽车盘式制动器、控制面板上的发光按钮，或黑暗中仍然可见的怪物眼睛。

简单的自发光材质可以通过一个颜色和亮度来定义。如果设置为自发光亮度设置一个大于 0 的值，物体将呈现自发光颜色和亮度：



材质自身发射红色光，亮度为 0.5。

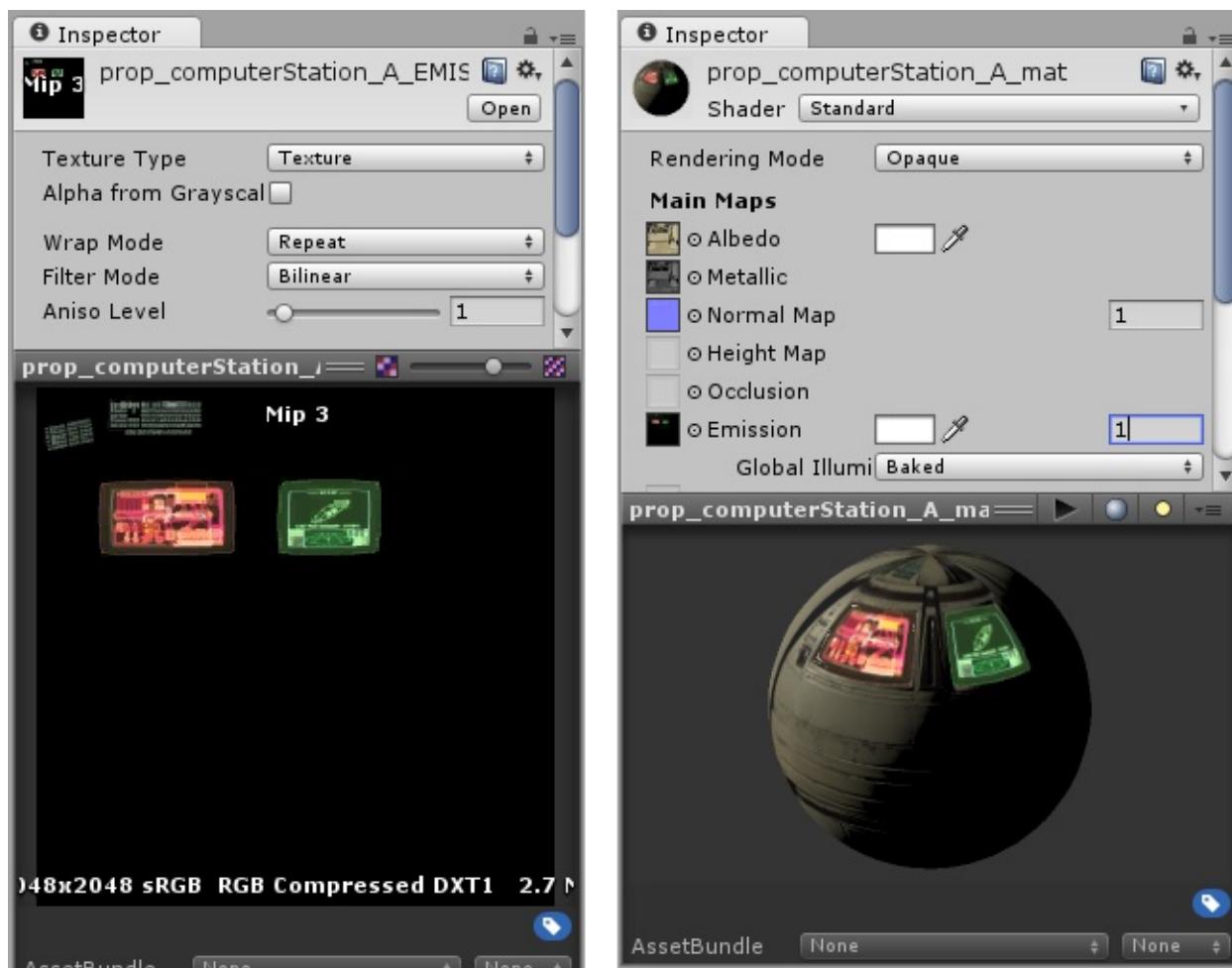
使用了自发光材质的物体，即使在场景的黑暗区域，也会保持亮度。



使用了自发光材质的红球、绿球和蓝球。即使它们位于一个黑暗的场景中，看起来仍然像是被内部光源照亮了。

除了使用单一颜色和亮度简单地控制自发光外，还可以为这个参数指定一张自发光贴图。与其他纹理贴图参数一样，这种方式可以更好地控制材质的发光区域。

如果指定了纹理贴图，纹理中的全部颜色值被用于定义自发光颜色和亮度。亮度数值输入框被保留了下来，可以把它作为一个乘数，提高或降低材质的整体亮度。



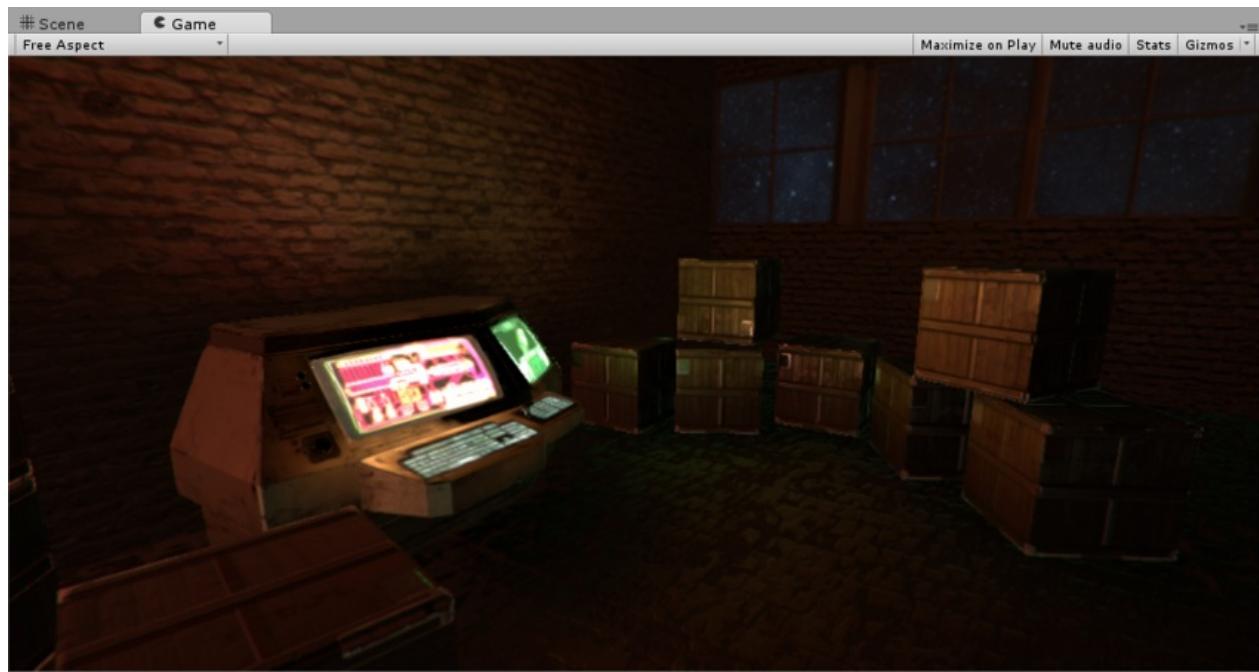
检视视图截图。左边：一张计算机终端的自发光贴图，含有两个发光屏幕和两个背光键盘。右边：使用这张自发光贴图的自发光材质。这个材质同时含有自发光和不发光区域。



在这张图中，存在了明亮和昏暗区域，以及投射在发光区域的阴影，完美呈现了自发光材质在不同光照条件下的视觉效果。

自发光参数带有一个全局光照设置，允许你指定该材质发射的光对附近物体的上下文光照的影响方式。提供了 3 个选项：

- **None** - 该物体将呈现为自发光，但是附近物体的光照不会受到影响。
- **Realtime** - 该材质的自发光将被添加到场景全局光照计算中，因此，附近物体甚至是移动物体将会受到自发光的影响。
- **Baked** - 该材质的自发光将被烘培到场景的静态光照贴图中，所以，附近物体将被该材质照亮，但是动态物体不会受到影响。



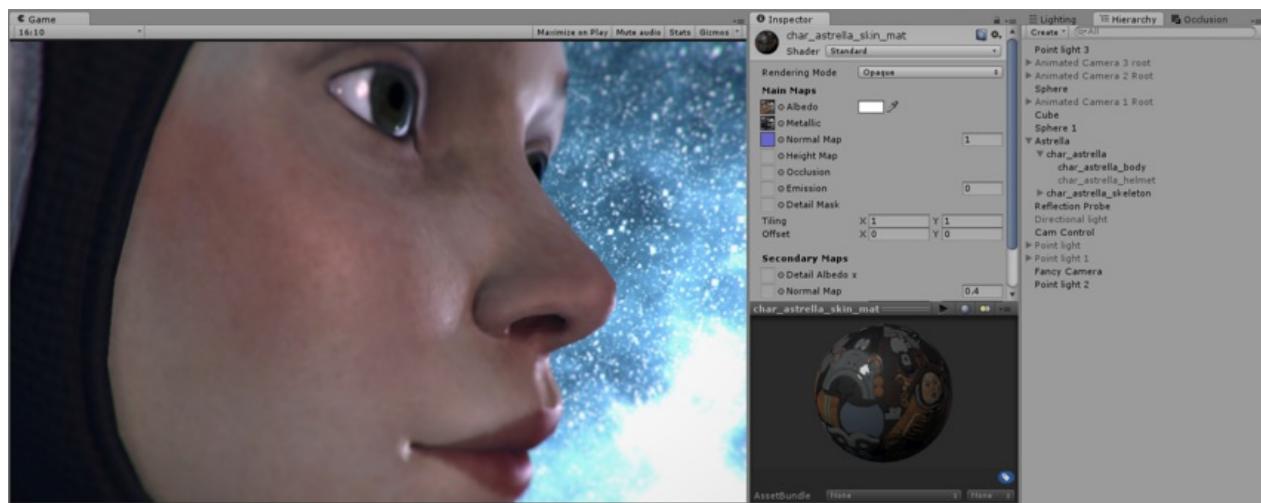
烘培计算终端的自发光贴图后，照亮了周围的黑暗区域。

辅助贴图（细节贴图）和细节蒙板

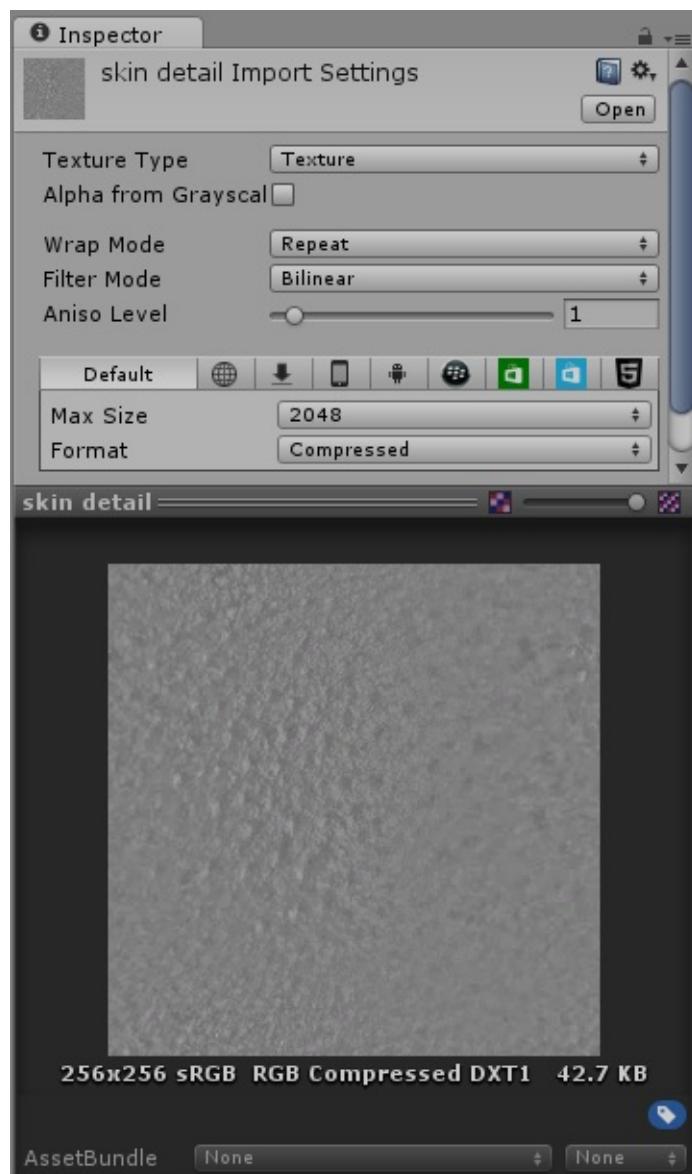
辅助贴图（或细节贴图）允许你在主纹理上叠加第二层纹理。你可以应用一张辅助漫反射贴图和辅助法线贴图。相对于主纹理，辅助贴图通常以非常小的尺寸在物体表面上重复多次。

设计辅助贴图的目的，是为了使材质在近距离观察时具有清晰的细节，在远距离观察时具有正常的细节，而不是必须使用一张极大的纹理来实现这两个目标。

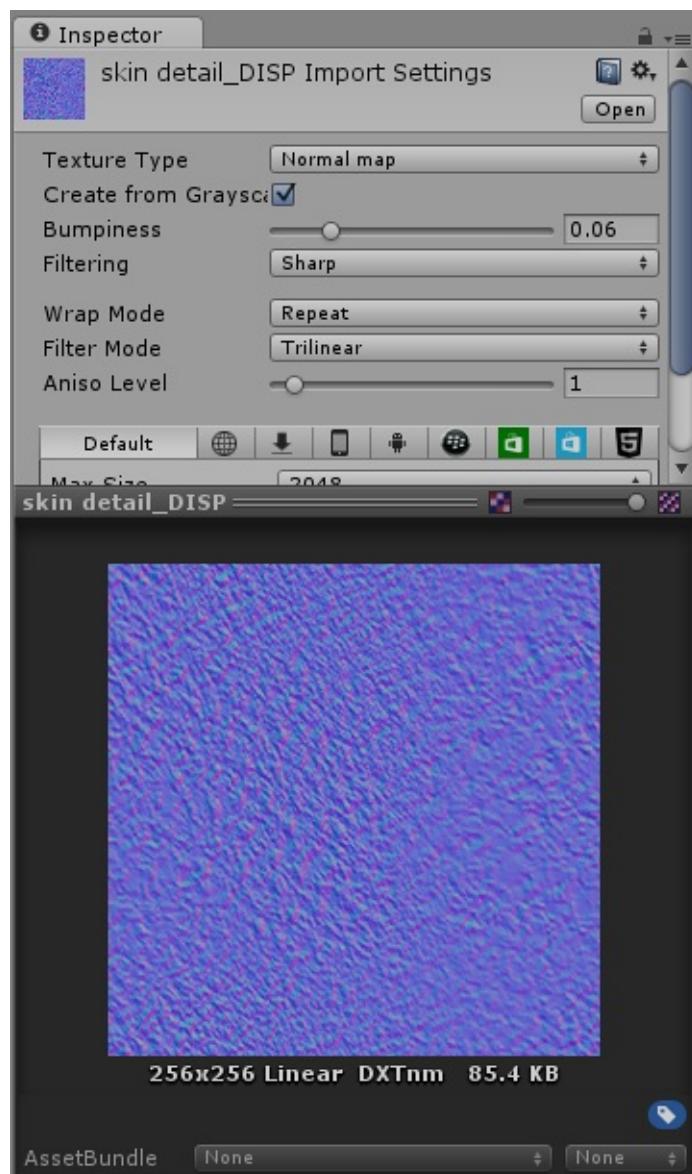
细节纹理的典型用途有：为角色的皮肤添加细节，例如毛孔和毛发；在砖墙上添加裂缝和地衣；为大型金属集装箱添加小划痕和磨损。



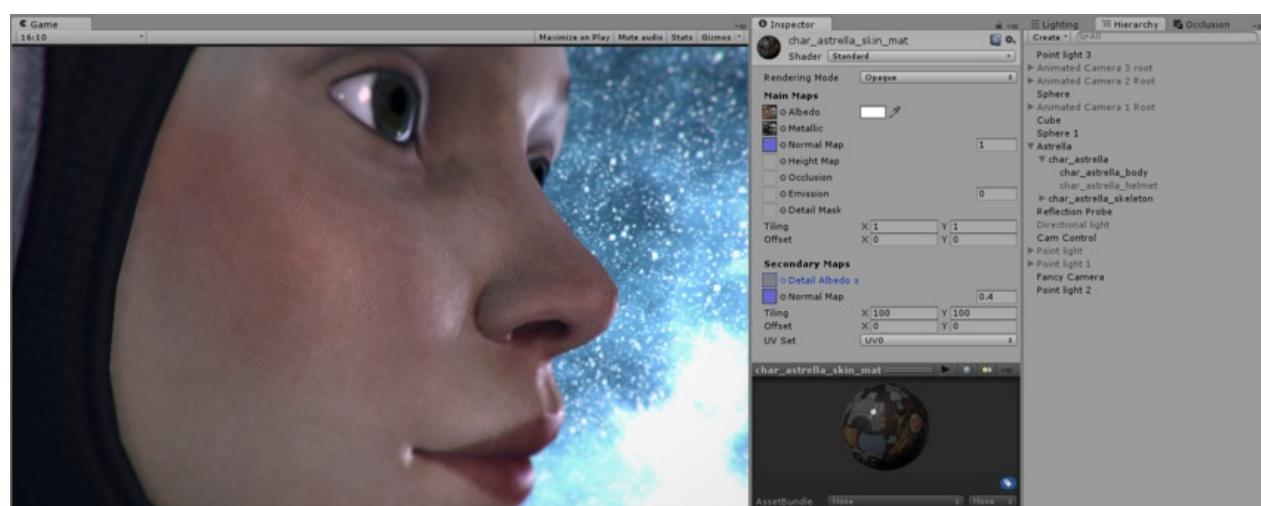
这个角色具有一张皮肤纹理贴图，但是没有细节纹理。我们将使用细节纹理增加皮肤毛孔。



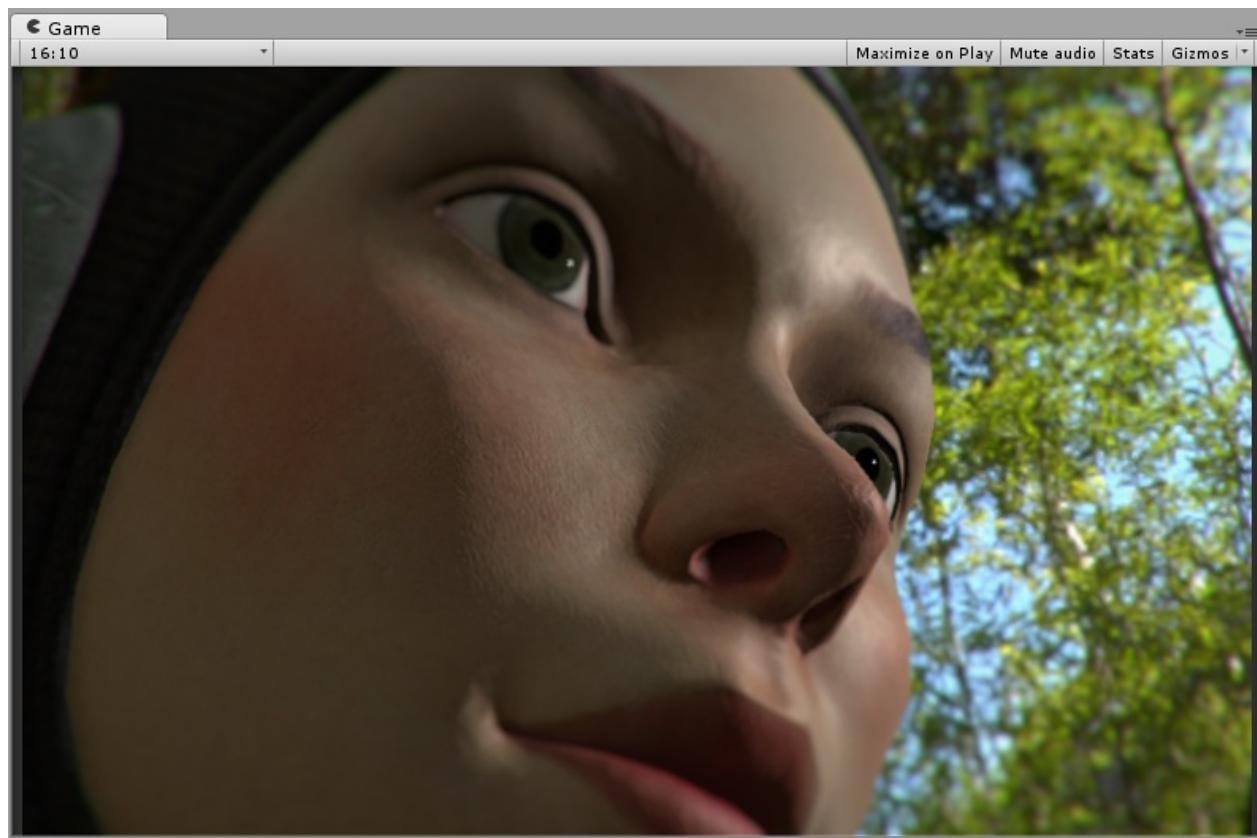
漫反射皮肤毛孔细节纹理。



皮肤毛孔细节的法线贴图。



最终效果，现在角色的皮肤上有细微的毛孔细节，分辨率比基础漫反射贴图或基础法线贴图层要高得多。



细节纹理可以为光照撞击表面的方式提供微妙而生动的效果。这是处于不同光照环境中同一个角色。

如果你只使用单个法线贴图，请务必把他插入主通道。因为，辅助法线贴图的性能开销比主法线贴图更高，但是效果相同。

细节蒙板

细节蒙板纹理允许对具有细节纹理的模型，遮盖某些特定区域。这意味着，你可以在特定区域显示细节问题，在其他区域隐藏细节纹理。在上面皮肤毛孔的例子中，你可能需要创建一个蒙板，以使毛孔不会出现在嘴唇和眉毛上。

菲涅耳效应

在现实世界中，物体的一个重要视觉暗示是，在掠射角处变得更具反射性（如下图所示）。这被称为菲涅耳效应。



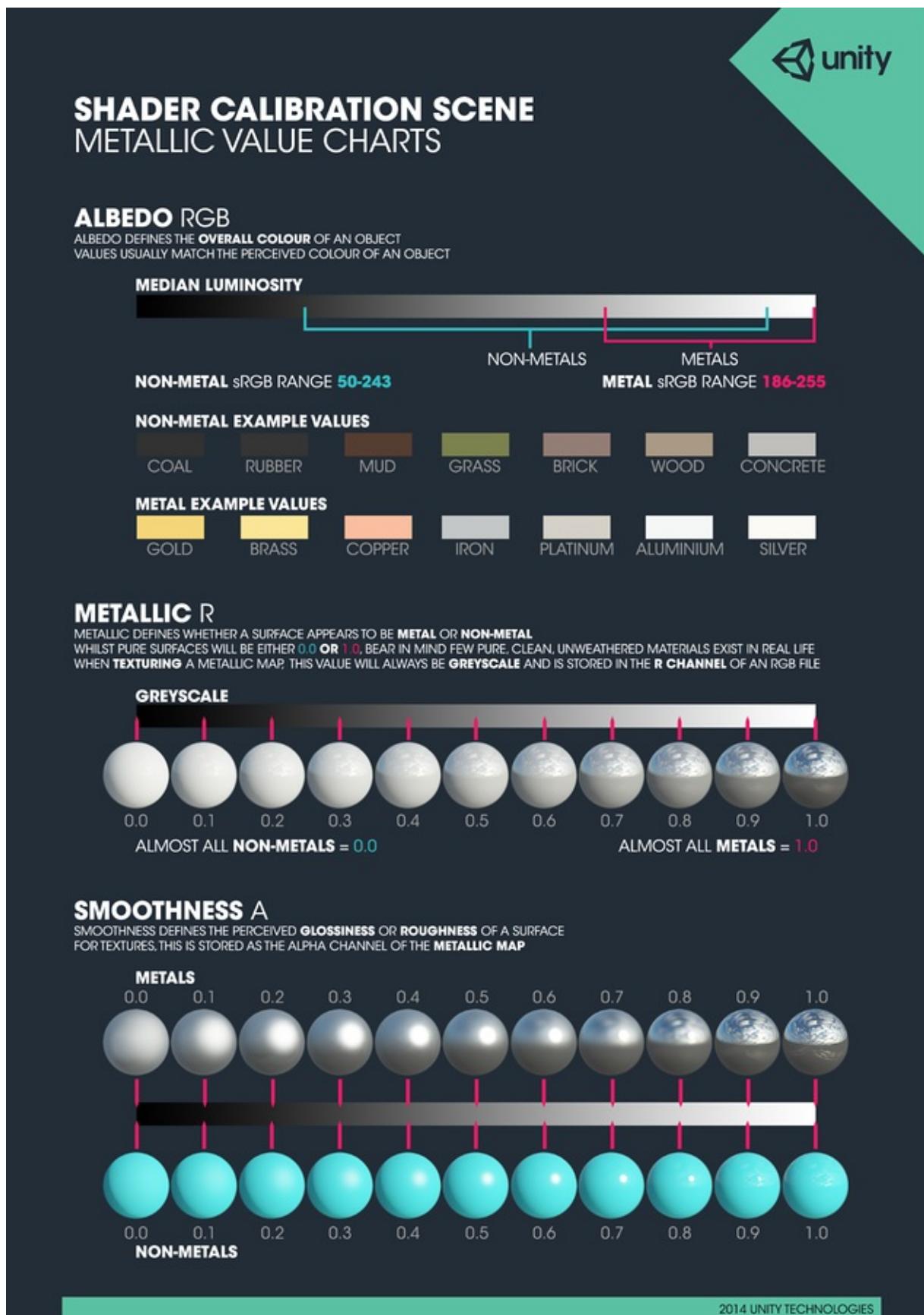
随着材质表面变得更加光滑，观察者在掠射角处看到的菲涅耳效应越来越明显。

在这个例子中有两点需要注意：首先，这些反射仅仅出现在球体的边缘（此时表面处于掠射角），并且，随着材质平滑度的提升，它们变得更加清晰和尖锐。

在标准着色器中没有对菲涅耳效应的直接控制。相反，需要通过材质的平滑度来间接控制。平滑的表面将呈现更强烈的菲涅耳效应，完全粗糙的表面不会有菲涅耳效应。

材质图表

使用这些图表作为实际设置的参考：



金属设置的参考图表。

译注：下面是图表中的中英文对照：

SHADER CALIBRATION SCENE

着色器校准场景

METALLIC VALUE CHARTS

金属度图表

ALBEDO RGB

漫反射颜色

ALBEDO DEFINES THE OVERALL COLOUR OF AN OBJECT

漫反射定义了物体的总体颜色

VALUES USUALLY MATCH THE PERCEIVED COLOR OF AN OBJECT

通常值与感知到物体颜色一致

MEDIAN LUMINOSITY

平均亮度

COAL 煤炭 RUBBER 橡胶 MUD 泥 GRASS 草 BRICK 砖 WOOD 木 CONCRETE 混凝土 GOLD 黄金 BRASS 黄铜 COPPER 铜 IRON 铁 PLATINUM 钯 ALUMINUM 铝 SLIVER 银

METALLIC R

金属度

METALLIC DEFINES WHETHER A SURFACE APPEARS TO BE METAL OR NON-METAL

金属度定义了表面是否呈现为金属或非金属

WHILST PURE SURFACES WILL BE EITHER 0.0 OR 1.0. BEAR IN MIND FEW PURE, CLEAN, UNWEATHERED MATERIALS EXIST IN REAL LIFE.

尽管纯粹的表面要么是 0.0 要么是 1.0。但是记住，现实生活中几乎不存在纯粹的、干净的材质。

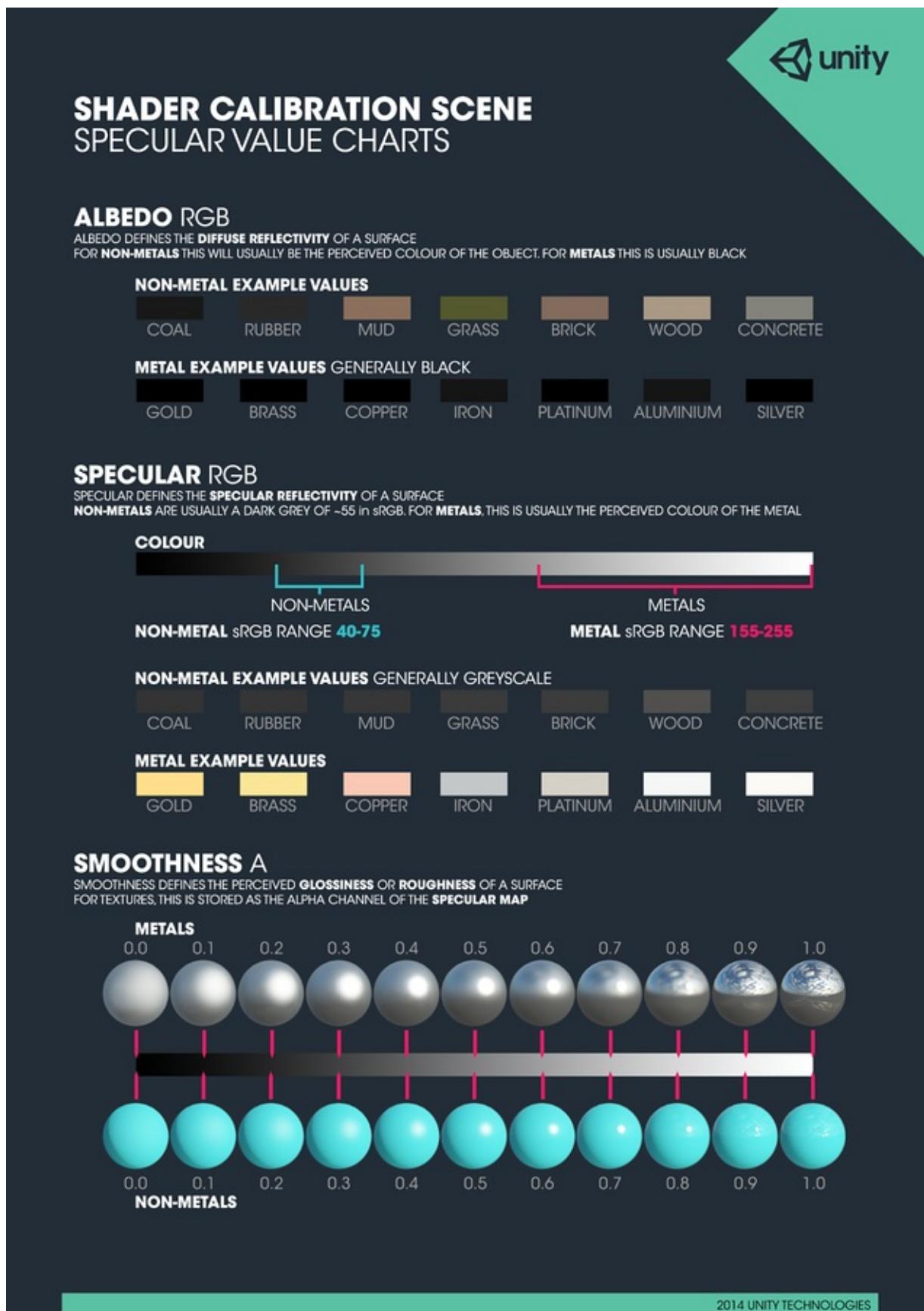
WHEN TEXTURING A METALLIC MAP. THIS VALUE WILL ALWAYS BE GREyscale AND IS STORED IN THE R CHANNEL OF AN RGB FILE.

设置金属贴图纹理时，该值总是灰色的，并且存储在 RGB 文件的 R 通道中。

SMOOTHNESS A 平滑度

SMOOTHNESS DEFINES THE PERCEIVED GLOSSINESS OR ROUGHNESS OF A SURFACE FOR TEXTURES. THIS IS STORED AS THE ALPHA CHANNEL OF THE METALLIC MAP

平滑度定义了表面纹理可感知的光泽度和粗糙度。该值存在金属贴图的 ALPHA 通道中。



镜面设置的参考图表。

译注：从前面的文档看，镜面模式使用起来不够友好，就不翻译了。

在这些图表中的，还有一些如何创建真实材料的提示。本质上，是选择一个工作流程（金属模式或镜面模式），然后为纹理或颜色选择器设置相应的值。举个例子，如果想要制作闪亮的白色塑料，我们需要设置漫反射为白色。因为它不是金属，我们需要一个非常低的 Metallic 值（或深色的 Specular），和一个非常高的平滑度。

粒子系统

在 3D 游戏中，大多数角色、道具和场景元素用 网格 表示，而在 2D 游戏中，则使用 精灵 表示。对于具有明确形状定义的实体对象，网格和精灵是描述它们的理想方式。不过游戏中还有其他实体，它们本质上是流动和无形的，因此很难用网格或精灵绘制。对于类似流动液体、烟雾、云、火焰和魔法等效果，可以使用一种称为 粒子系统 的图形方法，来实现内在流动性和能量效果。本章介绍 Unity 的粒子系统和使用场景。

什么是粒子系统？

粒子是小而简单的图像或网格，由粒子系统负责显示和剧烈移动。每个粒子代表了流体或无形实体的一小部分，所有粒子一起创建实体的完整外观。以烟雾为例，每个粒子是一张微小的烟雾纹理，像小块浮云一样。当许多这种微小浮云被一起布置在场景的某个区域时，整体效果是巨大的、体积填充的云朵。

系统动力学

每个粒子的生命周期是预定好的，通常是几秒钟，在此期间它可以经历各种变化。当粒子系统生成或射出一个粒子时，该粒子的生命便开始了。系统在特定空间区域（形状像球形、半球形、椎体、盒形或任意网格等）内的随机位置发射例子。粒子被一直显示，直到它的时间用完，然后它被从系统中删除。系统的发射率粗略地表示每秒钟发射的粒子数量，尽管发射的确切时间被略微随机化。发射率和粒子平均生命周期决定了『稳定』（即粒子的发射和死亡以相同速率发生）状态下粒子的粒子数量，以及系统达到这种状态所属额时间。

粒子动力学

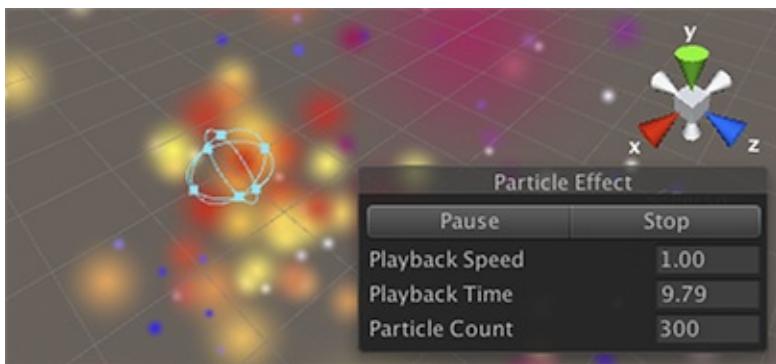
发射率和粒子生命周期会影响系统的整体行为，不过单个粒子也可以随时间改变。每个粒子具有一个速度向量，决定了粒子每帧移动的方向和距离。速度可以被系统自身施加的力和重力所改变，或者，粒子被地形上的区域风吹的到处乱飞。每个粒子的颜色、大小、旋转也可以随时间改变，或者与它当前的移动速度成比例地改变。颜色包含一个透明度组件，所以，粒子可以逐渐淡入淡出，而不是简单地出现和突然消失。

离子动力学可以用于相当逼真地模拟多重流体效果。例如，可以使用薄板发射形状来模拟瀑布，让水粒子在重力作用下下落，并逐渐加速。火产生的烟向上升起、膨胀并最终消散，所以系统应该在烟雾粒子上施加向上的力，并随着时间增加它们的尺寸和透明度。

使用粒子系统

Unity 使用一个组件实现粒子系统。在场景中放置粒子系统的常用方式是，添加一个预制的游戏对象（菜单：**GameObject > Create General > Particle System**），或者为一个现有的游戏对象添加粒子系统组件（菜单：**Component > Effects > Particle System**）。因为该组件相当复杂，所以检视视图被分割成数个可折叠的部分或模块，每个模块包含一组相关属性。另外，在检视视图中点击 **Open Window** 按钮，可以弹出单独的编辑窗口。相关信息请参阅 [粒子系统组件](#) 和 [粒子系统模块](#)。

当选中具有粒子系统的游戏对象时，场景视图将包含一个 **粒子效果** 小面板，其中包含一些简单的控件，用于可视化对系统设置的更改。



播放速度 **Playback Speed** 用于加速或减慢粒子模拟，以高级状态快速查看粒子效果。播放时间 **Playback Time** 表示自系统启动以来流逝的时间；这个时间可能比真实时间更快或更慢，取决于播放速度。粒子计数 **Particle Count** 表示当前系统中的粒子数量。通过点击播放时间 **Playback Time** 标签并左右拖动鼠标，可以向后和向前移动播放时间。面板顶部的按钮用于暂停和恢复模拟，或者停止并复位到初始状态。

随时间改变属性

粒子或粒子系统的许多数值型属性可以随时间变化。Unity 提供了几种不同的方法来指定变化如何发生：

- 常量 **Constant** 属性值在其生命周期中是固定的。
- 曲线 **Curve** 属性值由一条曲线或图形指定。
- 在两个常量之间随机 **Random Between Two Constants** 两个常量值分别定义属性值的上限和下限；真实值随着时间在这个范围内随机变化。
- 在两条曲线之间随机 **Random Between Two Curves** 两条曲线分别定义属性值在其生命周期内给定时间点上的上限和下限；当前值在这个范围内随机变化。

类似的，主模块中的 **起始颜色 Start Color** 属性具有以下选项：

- 颜色 **Color**：粒子的起始颜色在其生命周期中是固定的。
- 渐变 **Gradient**：以渐变指定的起始颜色发射粒子；渐变表示粒子系统的生命周期。
- 在两个颜色之间随机 **Random Between Two Colors**：粒子的起始颜色为两个给定颜色之间的随机线性插值。
- 在两个渐变之前随机 **Random Between Two Gradients**：在粒子系统的当前年龄点上，从给定的两个渐变上选取两个颜色；粒子的起始颜色为这两个颜色之间的随机线性插值。

对于其他颜色属性，例如 颜色随生命周期变化 **Color over Lifetime**，提供了两个单独的选项：

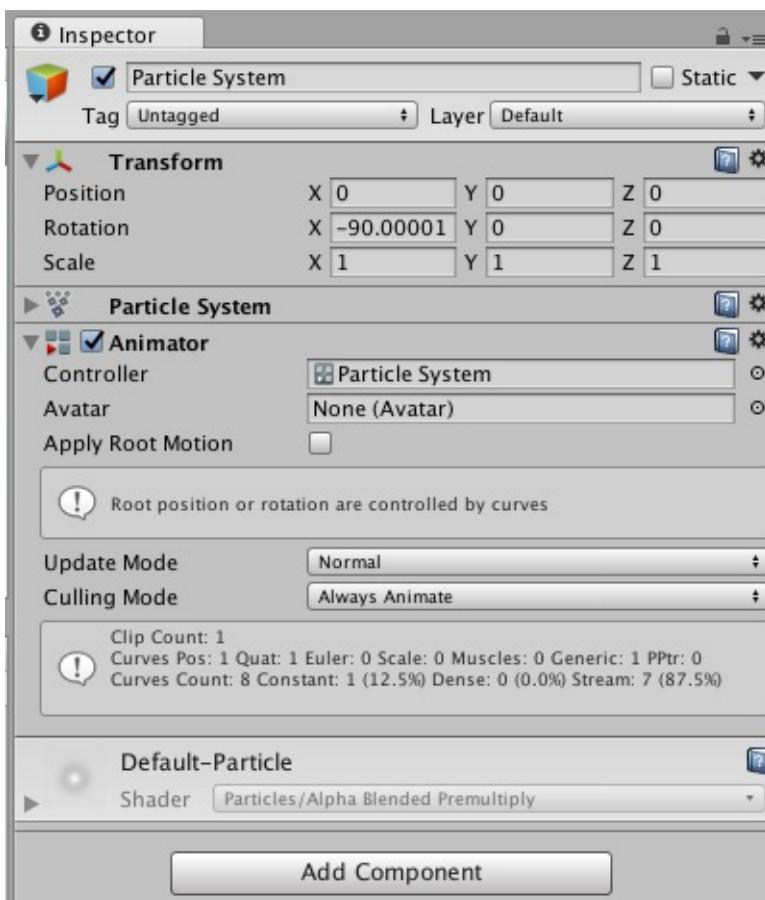
- 渐变 **Gradient**：以渐变指定的起始颜色发射粒子；渐变表示粒子系统的生命周期。
- 在两个渐变之前随机 **Random Between Two Gradients**：在粒子系统的当前年龄点上，从给定的两个渐变上选取两个颜色；粒子的起始颜色为这两个颜色之间的随机线性插值。

各模块中的颜色属性在每个通道上相互叠加，以计算出最终的粒子颜色。

绑定动画

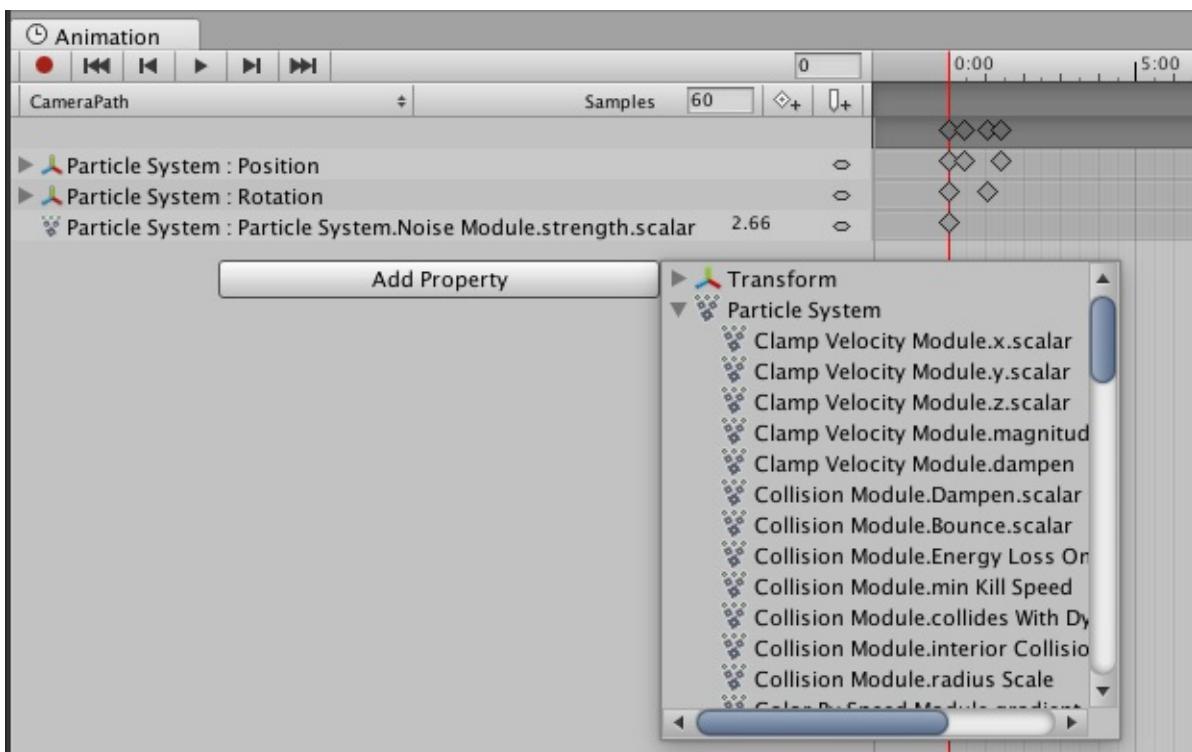
动画系统可以访问所有的粒子属性，这意味着可以为它们添加关键帧，并用动画控制它们。

在一个绑定了粒子系统的游戏对象上，为了访问粒子系统的属性，必须再绑定一个动画组件。还需要一个动画控制器和一个动画剪辑。



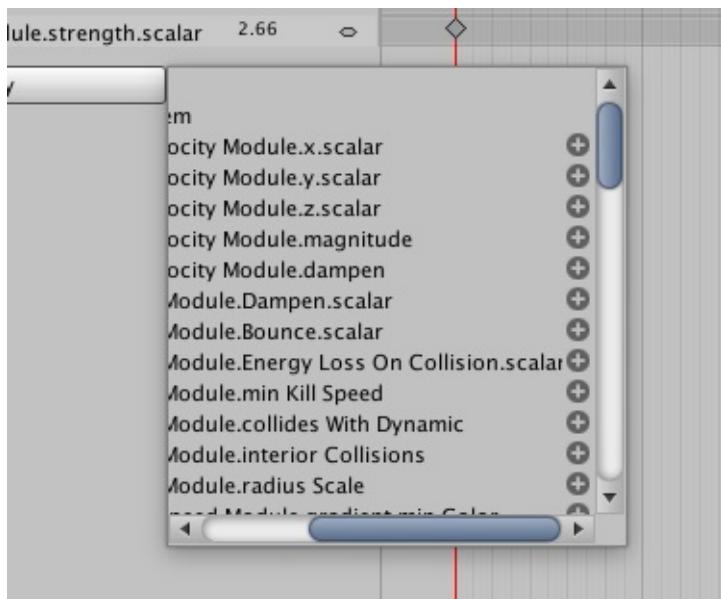
要为粒子系统添加动画，请添加一个动画组件，并分配一个带有动画剪辑的动画控制器。

为了粒子系统的属性添加动画，选中包含了动画控制器和粒子系统的游戏对象，然后打开动画视图 **Animation Window**。点击 添加属性 **Add Property**。



在动画视图 Animation Window 中，添加一个属性到动画剪辑中。

向右滚动以显示添加控件。



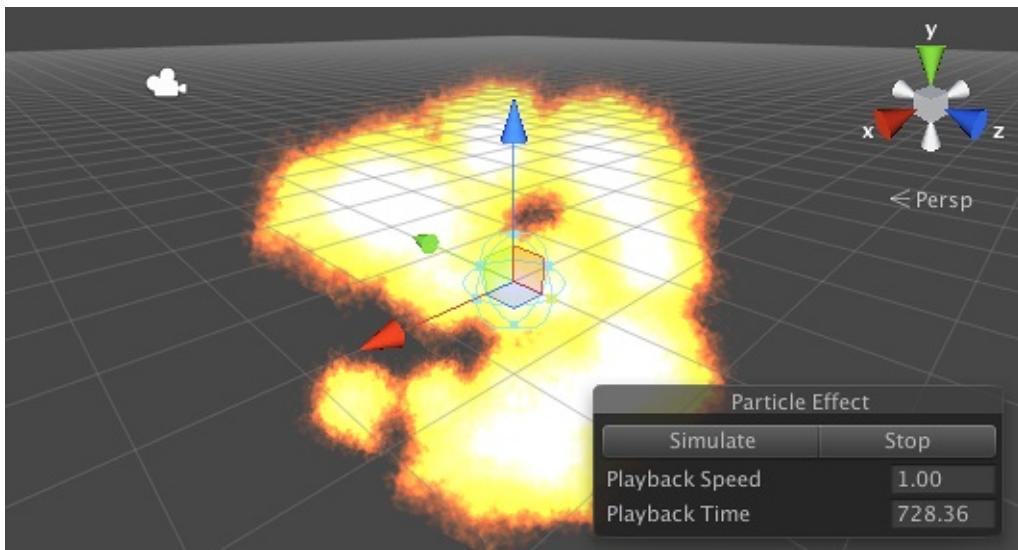
请注意，为曲线添加关键帧时，添加的是整条曲线的 倍增器，你可以在 检视视图 的曲线编辑器中看到。

粒子系统入门

本节介绍如何实现常见类型的粒子系统。你可以自由使用文档中所有代码，不受 Unity 的任何限制。

简单的爆炸

你可以使用粒子系统创建一个逼真的爆炸效果，但是其中的动力学可能有些复杂。本质上，爆炸不过是向外的粒子脉冲（爆发），不过你可以做一些简单的修改，让爆炸看起来更逼真。



开发阶段的粒子系统爆炸

粒子的时间轴

一个简单的爆炸会产生一个在所有方向上迅速向外膨胀的火焰球。爆炸的初始阶段具有巨大的能量，因此非常炽热（即明亮），并且火焰非常快速地移动。然后，能量快速消散，导致火焰的膨胀减慢和冷却（即变得不那么明亮）。最终，随着所有燃料被烧尽，火焰将减弱，并且很快完全消失。

爆炸粒子的生命周期通常很短，你可以在其生命周期中改变多个不同属性，以模拟这种效果。粒子的初始移动速度非常快，然后随着远离爆炸中心，它的速度将急剧降低。另外，颜色应该从明亮开始，然后变暗，最终逐渐变淡直至透明。最后，随着生命周期减小粒子的尺寸，将产生火焰因燃料耗尽而消散的效果。

实现

从默认的粒子系统对象开始（菜单：**GameObject > Create General > Particle System**），找到形状 **Shape** 模块，设置发射形状为一个小球 **Sphere**，并设置半径为 0.5 单位。粒子标准资源包中包含了一个名为 **Fire Add** 的材质，非常适合用于爆炸（菜单：**Assets > Import**

Package > Particles)。你可以使用 *Renderer* 模块为粒子系统设置材质。展开 *Renderer* 后，还应该禁用 投射阴影 *Cast Shadows* 和 接收阴影 *Receive Shadows*，因为爆炸的火焰应该发射光而不是接收光。

在这个阶段，粒子系统看起来像从中心点抛出了许多小火球。爆炸当然应该立即产生大量的粒子脉冲。在发射 *Emission* 模块，你可以设置速率 *Rate* 为 0，并在时间 0 点添加一个单独的粒子脉冲 *Burst*。该脉冲中的粒子数量取决于暴躁的大小和强度，不过，开始时最好设置为 50 个粒子。设置好这个脉冲后，现在粒子系统开始看起来更像是爆炸了，但是它相当缓慢，并且火焰看起来长时间炫富。在粒子系统模块中（与游戏对象同名，例如 *Explosion*），设置系统持续时间 *Duration* 和 粒子生命周期 *Start Lifetime* 为 2 秒。

你也可以使用 随生命周期改变大小 *Size Over Lifetime* 模块创建火焰耗尽燃料的效果。设置大小曲线为预设的『斜降 ramp down』（即，大小从 100% 开始，然后减小为 0）。为了使火焰变暗和变淡，开启 随生命周期改变颜色 *Color Over Lifetime* 模块，设置颜色渐变，从左侧的白色开始，到右侧的黑色结束。因为材质 *Fire Add* 材质使用了 *Additive* 着色器，所以颜色属性的明暗度也控制了粒子的透明度；当颜色渐变为黑色时，火焰将变得完全透明。并且，当绘制的粒子彼此重叠时，*Additive* 材质支持粒子亮度的叠加。这有助于增强爆炸开始时的明亮闪光，此时全部粒子紧靠在一起。

现在，爆炸正在形成，但是它看起来好像发生在太空中。粒子从被抛出到消失，以恒定的速度传播。如果游戏发生在太空中，那么这确实可能是你想要的效果。但是，发生在大气中的爆炸将被周围的空气阻碍并衰减。开启 随生命周期抑制速度 *Limit Velocity Over Lifetime*，设置速度 *Speed* 为约 3.0，设置抑制 *Dampen* 分数为约 0.4，你应该可以看到爆炸随着进度损失了一些强度。

最后要注意的是，随着粒子远离爆炸中心，它们各自的形状变得更加可辨认。特别是，可以看到所有粒子具有相同的大小和相同的旋转，显而易见，同样的图形被重复用于每个粒子。避免这种情况的一个简单方法是，在生成粒子时，为粒子的大小和大小增加一点随机变化。在检视视图顶部的 粒子系统 模块中，点击 初始大小 *Start Size* 和 初始旋转 *Start Rotation* 右侧的小箭头，把它们都设置为 在两个常量之间随机 *Random Between Two Constants*。对于旋转，将两个值设置为 0 和 360（即完全随机地旋转）。对于大小，设置为 0.5 和 1.5，以提供一些变化，并且不用担心会有太多巨大或微小的粒子。现在，你应该看到粒子图形的重复不太明显了。

用法

在测试期间，打开 *Looping* 属性是有用的，以便你可以反复看到爆炸，但是在完成的游戏 中，你应该关闭它，以使爆炸只发生一次。当为一个可能爆炸的对象（例如燃油箱）设置爆炸效果时，你需要为它添加粒子系统组件，并禁用 唤醒时播放 *Play On Awake* 属性。然后，你可以根据需要触发爆炸效果。

```

void Explode() {
    var exp = GetComponent<ParticleSystem>();
    exp.Play();
    Destroy(gameObject, exp.duration);
}

```

在某些情况下，爆炸发生在撞击点。如果爆炸来自某个对象（例如手榴弹），那么你可以在延迟一段时间后或当它与目标接触时，调用上面编写的 `Explode` 函数。

```

// Grenade explodes after a time delay.
public float fuseTime;

void Start() {
    Invoke("Explode", fuseTime);
}

// Grenade explodes on impact.
void OnCollisionEnter(Collision coll) {
    Explode();
}

```

译注：fuse time 引信时间。

当爆炸来自一个不可见的游戏对象时（例如，速度太快以至于无法看到的子弹），你可以在适当的位置实例化爆炸。例如，你可以用 [射线 raycast](#) 确定接触点。

```

// On the explosion object.
void Start() {
    var exp = GetComponent<ParticleSystem>();
    exp.Play();
    Destroy(gameObject, exp.duration);
}

// Possible projectile script.
public GameObject explosionPrefab;

void Update() {
    RaycastHit hit;

    if (Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition), out hit)) {
        Instantiate(explosionPrefab, hit.point, Quaternion.identity);
    }
}

```

更进一步

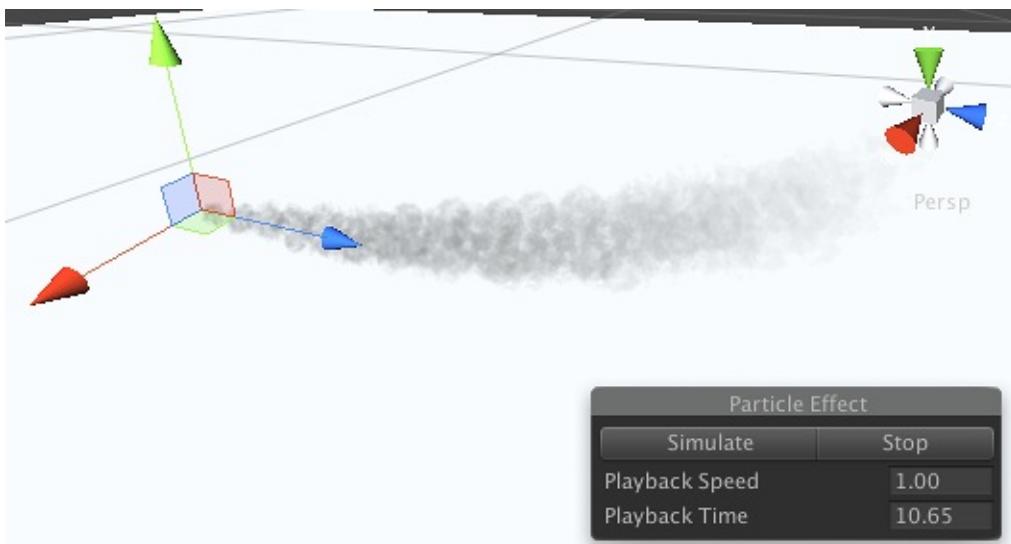
这里开发的爆炸非常基础，不过你可以修改它的各个模块，以获得想要的精确效果。

所使用的粒子图形将极大地影响玩家如何『理解』爆炸。许多微小的、单独可识别的火焰，暗示燃烧的碎片正在被抛出。更大的、完全不可移动的部分，看起来更像是由被破坏的燃料箱产生的火球。通常，你需要同时更改多个属性才能实现这种效果。例如，火球在消失前将持续更长的时间和较小的膨胀，而剧烈的爆炸可能在相当一段距离内散布燃烧的碎片。

爆炸示例中的几个属性被设置为随机值，但是其他许多其他属性都有一个在两个常量/曲线之间随机 *Random Between Two Constants/Curves* 选项，你可以使用这些属性添加各种变化。改变大小和旋转可以避免最明显的粒子重复问题，你也可以考虑为初始延迟 *Start Delay*、初始生命周期 *Start Lifetime* 和初始速度 *Start Speed* 添加一些随机性。轻微的变化有助于增强爆炸效果的自然性和不可预测性，而不是受控的机械式过程。较大的变化暗示一场『放射性』爆炸。例如，改变初始延迟 *Start Delay* 将产生不再突然、爆发更慢的爆炸，可能是因为车辆的燃油箱被单独点燃。

载具尾气

汽车和其他载具把燃油转换为动力时会排放废气。你可以使用粒子系统画龙点睛地为载具添加排气效果。



粒子系统产生的排气效果

粒子的时间轴

废气烟雾从管道中快速排出，随后在与大气接触时迅速减慢移动速度。随着减慢，它向四周散开，并变得更加模糊，然后快速消散在空气中。因为废气是热的，所以在穿过周围的冷空气时会轻微地上浮。

废气烟雾粒子的初始大小不能大于排气管的宽度，但是在它短短的生命周期内，它的大小将大大增加。它通常从半透明开始，随着它与空气混合，将淡出到完全透明。至于动力学，粒子将非常快速地发射，然后快速减慢，并略微上浮。

实现

在形状 **Shape** 模块中，选择圆锥体 **Cone** 形状，并设置它的角度 **Angle** 属性为 0；这里的『圆锥体』实际上是圆柱形的管道。管道的半径 **Radius** 取决于实际车辆的大小，不过，通常你可以在场景视图中，让半径线框与车辆模型相匹配（例如，汽车模型通常具有一个排气管，或是在后面有一个洞，从而你可以匹配排气管的大小）。半径实际上决定了很多属性的设置，例如粒子大小和发射率。在这个示例中，我们假设载具是一辆遵循 Unity 标准尺寸约定（1 单位等于 1 米）的汽车；因此设置半径为约 0.05 或 5 厘米。

标准资源提供的 *Smoke4* 材质很适合作为烟雾粒子的图形。如果尚未安装这些资源，那么从菜单中选择 **Assets > Import Package > Particles**。然后，进入粒子系统的渲染器 *Renderer* 模块，设置材质 *Material* 为 *Smoke4*。

对于汽车尾气，默认的 5 秒生命周期通常太长了，因此你应该打开 *Particle System* 模块（与游戏对象同名，例如『Exhaust』），设置初始生命周期 *Start Lifetime* 为 2.5 秒。还是在这个模块中，设置模拟空间 *Simulation Space* 为世界 *World*，设置重力修改器 *Gravity Modifier* 为一个小的负值，比如 -0.1。使用世界模拟空间，是为了使烟雾悬浮在其产生的位置，即使是在载具行驶时。负重力效应使烟雾粒子上浮，就像它们是由热气体构成的。还是一个点睛之笔是，点击初始渲染 *Start Rotation* 旁边的小菜单箭头，选择在两个常量之间随机 *Random Between Two Constants* 选项。将这两个值分别设置为 0 和 360，烟雾粒子将在发射时随机旋转。许多完全相同的粒子排列在一起太不自然了，不利于生成随机的、无形的烟雾轨迹。

在这个阶段，烟雾粒子开始看起是真实的，默认的发射率很好地产生了发动机的间歇性燃烧效果。不过，烟雾没有向外膨胀，也没有消失。开启随生命周期改变颜色 *Color Over Lifetime* 模块，点击渐变面板右端顶部的按钮（控制颜色 alpha 值的透明度）。设置 alpha 值为 0，你应该在场景视图中看到烟雾粒子消失不见。根据发动机的清洁程序，你可能想要降低开始时的渐变透明度；厚重的、黑色的颜色暗示了肮脏的、低效的燃烧。

除了淡出，烟雾还应该在逃逸过程中增加大小，使用随生命周期改变大小 *Size Over Lifetime* 模块可以很容易地创建这种效果。启用这个模块，选中曲线，滑动左端的曲线手柄，使粒子的初始大小为完整尺寸的一小部分。粒子的实际大小取决于排气管的大小，但是，比排气管稍大的值，可以创建更好的气体溢出效果。（如果粒子的初始大小与排气管相同，将暗示气体的形状被排气管所限制，但气体根本并没有固定的形状。）可以在场景视图中看到，粒子系统很好地模拟了烟雾的视觉效果。如果创建的烟雾扩散效果不符合预期，你可能还需要在粒子系统模块中增加初始大小 *Start Size*。

最后一个效果，烟雾将随着扩展而减慢速度。一个简单的实现方式是使用随时间改变作用力 *Force Over Lifetime* 模块。启用这个模块，设置空间 *Space* 选项为本地 *Local*，设置作用力组件的 Z 为一个负值，表示粒子被作用力回推（系统沿着对象本地空间的正 Z 方向发射粒子）。如果按照前文所述设置其他参数，那么设置为约 -0.75，粒子系统将很好地运行。

用法

你可以将尾气粒子系统添加到载具的某个子对象上来定位它。对于简单的游戏，你可以开启唤醒时播放 *Play On Awake* 和循环播放 *Looping*，让粒子系统开始运行。然而大多数情况下，你可能至少需要改变载具行驶时的发射率。这么做首先是为了真实性（例如，引擎在剧烈运转时将产生更多的烟雾），其次还可以避免烟雾粒子随着载具行驶而散开。一个快速移动的载具，如果发射率太低，将产生一股一股的烟雾，非常不真实。

通过脚本可以非常容易地改变发射率。如果脚本中的某个变量代表了引擎的转速或载具的速度，那么你可以简单地把它的值乘以一个常数，并把计算结果分配给粒子系统的 `emissionRate` 属性。

```
// C#
using UnityEngine;
using System.Collections;

public class PartScriptTestCS : MonoBehaviour {

    public float engineRevs;
    public float exhaustRate;

    ParticleSystem exhaust;

    void Start () {
        exhaust = GetComponent<ParticleSystem>();
    }

    void Update () {
        exhaust.emissionRate = engineRevs * exhaustRate;
    }
}
```

```
// JS

var engineRevs: float;
var exhaustRate: float;

var exhaust: ParticleSystem;

function Start() {
    exhaust = GetComponent.<ParticleSystem>();
}

function Update () {
    exhaust.emissionRate = engineRevs * exhaustRate;
}
```

更进一步

基本方案创建了相当逼真的尾气效果，不过你可能已经注意到，引擎的『特性』随着参数的改变而改变。一个缺乏维护的、低效率的引擎将不完全地燃烧燃料，产生浓厚的、黑暗的烟雾，并长时间存在于空气中。这种效果很适合老旧的农用拖拉机，但是不适合高性能的运动

跑车。对于清洁的发动机，你应该为粒子的生命周期、透明度和大小增量设置更小的值。对于『肮脏』的引擎，你应该增大这些值，可能还需要使用发射 *Emission* 模块的脉冲 *Bursts* 属性，来创建引擎噼啪响的效果。

屏幕特效概述

屏幕特效是 [标准资源](#) 的一部分，可以快速简单地改变游戏的视觉效果。最简单的理解方式是，把它们看作是图像编辑程序（例如 Photoshop 或 Instagram），为图像提供了各种过滤器。

Unity 中的屏幕特效为每帧提供额外的后处理通道，通过操纵每个像素来改变图像。因此屏幕特效是资源密集型的，添加它们时应该考虑到这一点。添加的屏幕特效越多，处理图像的任务就越繁重，可能导致帧率越低。

Unity 编辑器提供了许多屏幕特效；更多的屏幕特效在资源商店 Asset Store 和其他在线服务中。要了解 Unity 编辑中可用的屏幕特效，请参阅 [屏幕特效参考页](#)。

参考

- [屏幕特效参考页](#)
- [编写屏幕特效](#)
- [教程：屏幕特效](#)

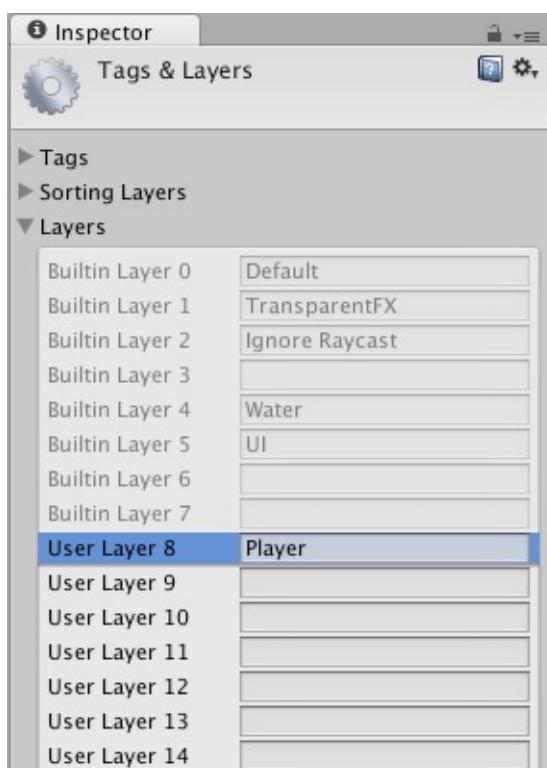
图层

图层 最常用于 摄像机 只渲染场景的一部分，和 光线 只照亮场景的一部分。此外，射线投射也可以用它们选择性地忽略碰撞器或创建 碰撞。

创建图层

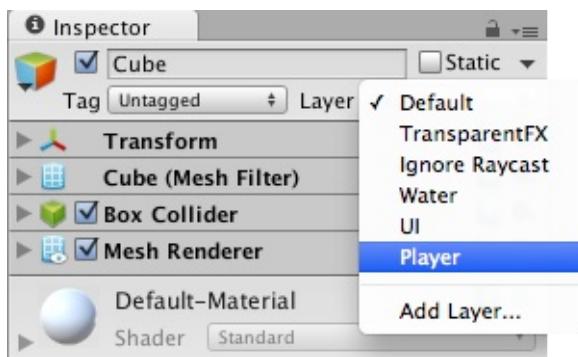
第一步是创建一个图层，稍后可以将其分配给一个 游戏对象 **GameObject**。要创建图层，请打开菜单并选择 **Project Settings->Tags and Layers**。

我们在空着的 **User Layers** 中新建一个图层。选择第 8 图层 **User Player 8**。



分配图层

现在，你已经新建了一个图层，必须将该图层分配给一个游戏对象。



在标签管理器中，指定图层 `Layer` 为 `Player`。

使用摄像机的剔除遮罩仅绘制部分场景

使用摄像机的剔除遮罩，你可以选择性地渲染特定图层中的对象。为此，请选中负责选择性渲染对象的摄像机。

在剔除遮罩 `culling mask` 属性中，通过选中或取消图层来修改。



选择性地投射射线

通过使用图层，你可以在投射射线时选择性地忽略特定图层中的碰撞器。例如，你可能想要投射只针对玩家的射线，而忽略所有其他碰撞器。

函数 `Physics.Raycast` 接受一个位掩码 `layerMask`，其中每一个比特决定了一个图形是否将被忽略。如果 `layerMask` 中的所有比特位都为 1，那么该射线将和所有碰撞器发生碰撞。如果 `layerMask` 等于 0，那么该射线永远不会和任意对象发生碰撞。

```
// JavaScript example.

// bit shift the index of the layer to get a bit mask
var layerMask = 1 << 8;
// Does the ray intersect any objects which are in the player layer.
if (Physics.Raycast (transform.position, Vector3.forward, Mathf.Infinity, layerMask))
    print ("The ray hit the player");
```

```
// C# example.

int layerMask = 1 << 8;

// Does the ray intersect any objects which are in the player layer.
if (Physics.Raycast(transform.position, Vector3.forward, Mathf.Infinity, layerMask))
    Debug.Log("The ray hit the player");
```

而在现实世界中，你想要做的恰好与之相反。我们想要向所有碰撞器投射射线，除了 Player 图层中的碰撞器。

```
// JavaScript example.

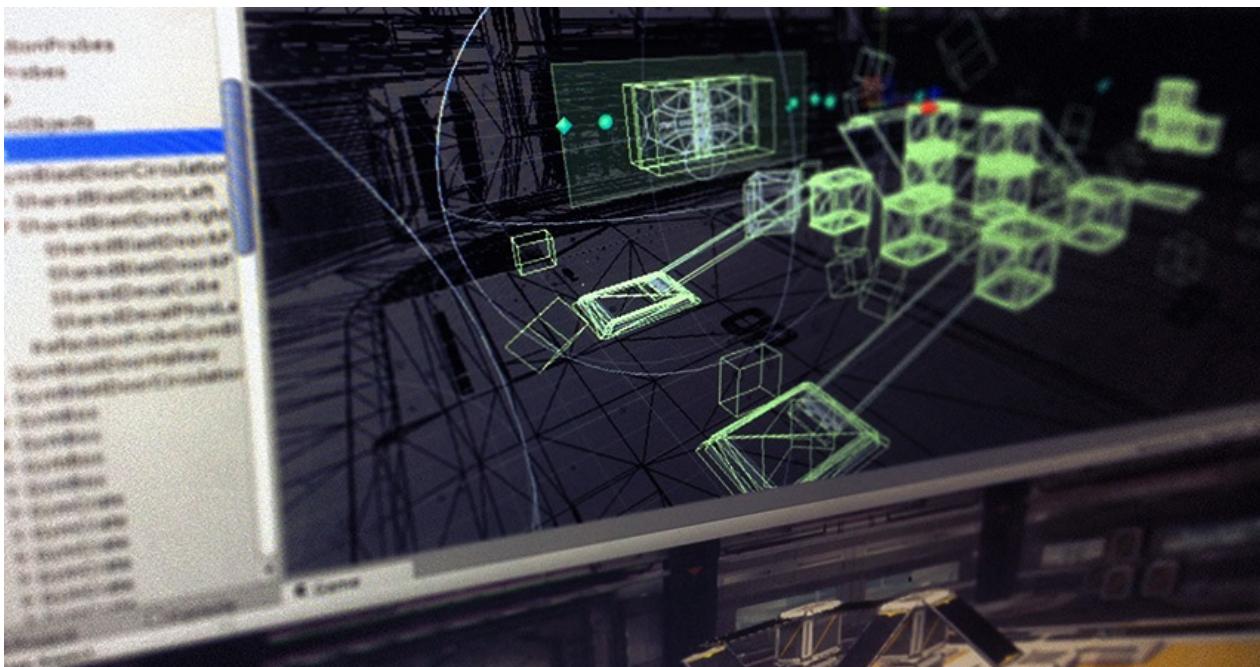
function Update () {
    // Bit shift the index of the layer (8) to get a bit mask
    var layerMask = 1 << 8;
    // This would cast rays only against colliders in layer 8.
    // But instead we want to collide against everything except layer 8. The ~ operator
    // does this, it inverts a bitmask.
    layerMask = ~layerMask;

    var hit : RaycastHit;
    // Does the ray intersect any objects excluding the player layer
    if (Physics.Raycast (transform.position, transform.TransformDirection (Vector3.forward), hit, Mathf.Infinity, layerMask)) {
        Debug.DrawRay (transform.position, transform.TransformDirection (Vector3.forward) *
        hit.distance, Color.yellow);
        print ("Did Hit");
    } else {
        Debug.DrawRay (transform.position, transform.TransformDirection (Vector3.forward) *
        1000, Color.white);
        print ("Did not Hit");
    }
}
```

```
// C# example.  
void Update () {  
    // Bit shift the index of the layer (8) to get a bit mask  
    int layerMask = 1 << 8;  
  
    // This would cast rays only against colliders in layer 8.  
    // But instead we want to collide against everything except layer 8. The ~ operator  
    // does this, it inverts a bitmask.  
    layerMask = ~layerMask;  
  
    RaycastHit hit;  
    // Does the ray intersect any objects excluding the player layer  
    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");  
    }  
}
```

如果调用 `Raycast` 函数时没有传入 `layerMask`，将只会忽略使用了 `IgnoreRaycast` 图层的碰撞器。这是投射射线时忽略某些碰撞器的最简单方式。

物理系统



为了实现逼真的物理行为，游戏中的对象必须被正确地加速，并且被碰撞、重力和其他力所影响。Unity 的内置物理引擎提供了处理物理模拟的组件。只需设置几个参数，就可以创建具有真实行为的对象（例如，对象被碰撞后将开始移动和掉落，但是它们不会自己移动）。通过脚本控制物理行为，你可以提供动态的车辆、机器，甚至是一片布料。本章概述了 Unity 中的主要物理组件，并提供扩展阅读的链接。

注意：Unity 中实际上有两个独立的物理引擎：一个用于 3D 物理，一个用于 2D 物理。两个引擎的主要概念是相同的（除了 3D 中的额外纬度），但是它们的实现使用了不同的组件。例如，刚体组件 `Rigidbody` 用于 3D 物理，与之类似的 `Rigidbody 2D` 则用于 2D 物理。

相关教程：[物理引擎](#)，[物理引擎最佳实践](#)

有关故障排除、提示和技巧，请参阅 [物理引擎知识库](#)。

物理系统概述

这些简单地介绍 Untiy 中的主要物理组件，并且详细介绍了它们的用法和扩展阅读链接。

刚体概述

刚体 **Rigidbody** 是为游戏对象赋予物理行为的主要组件。绑定组件后，游戏对象将立即响应重力。如果还添加了一个或多个碰撞器组件 **Collider**，游戏对象将被即将到来的碰撞所移动。

由于刚体组件接管了游戏对象的移动，所以不应该尝试通过修改 **Transform** 的位置或旋转属性来移动游戏对象。相反，你应该用作用力来推动物体，并让物理引擎来计算结果。

在某些情况下，你可能希望一个游戏对象具有刚体组件，但是不要物理引擎控制它的运动。例如，你可以想要通过脚本代码直接控制角色，同时允许触发器进行碰撞检测（请参见下面的触发器 **Triggers**）。这种由脚本产生的非物理运动称为 动力学运动。刚体组件有一个 **Is Kinematic** 属性，它可以把游戏对象从物理引擎中移除，并用脚本来移动。可以用脚本控制 **Is Kinematic** 的值，从而开始或关闭对象的物理行为，但这会带来性能开销，应该谨慎使用。

有关这些组件的设置和脚本选项的更多细节，请参阅 [Rigidbody](#) 和 [Rigidbody 2D](#) 的参考页。

休眠

当刚体以低于定义的最小线性速度或最小旋转角度运动时，物理引擎会认为该刚体已经停止。此时，游戏对象被设置为『休眠』模式，不再移动，直到它接收到碰撞或作用力。这个优化意味着，在刚体被再次『唤醒』（再次处于运动中）之前，不会消耗处理器时间来更新刚体。

当刚体移动速度低于定义的最小线性或旋转速度时，物理引擎假定它已经停止。当这种情况发生时，**GameObject** 不再移动，直到它接收到碰撞或力，因此它被设置为“睡眠”模式。此优化意味着在下一次“唤醒”（即，再次设置为运动）时，不更新刚体的处理器时间。

对于大多数场景，刚体组件的休眠和唤醒是显而易见的。但是，如果通过修改静态碰撞器（非刚体）的 **Transform** 位置，使进入或离开一个休眠中的游戏对象，该游戏对象可能无法唤醒。例如，当游戏对象下面的地板移走时，这可能导致游戏对象悬挂在空中。在这种情况下，可以使用 **WakeUp** 函数来显式地唤醒游戏对象。关于休眠的更多信息参阅 [Rigidbody](#) 和 [Rigidbody 2D](#) 组件页面。

碰撞器

碰撞器 组件定义了物体用于物理碰撞的形状。碰撞器是不可见的，并且不需要与物体网格的形状完全相同。事实上，在游戏中，粗略的近似值通常更加有效，并且微不可查。

最简单（也是最小性能开销）的碰撞器是所谓的 基本碰撞器。在 3D 中，包括 盒碰撞器、球形碰撞器、胶囊碰撞器。在 2D 中，包括 2D 盒碰撞器、2D 圆形碰撞器。可以为一个物体添加任意数量的基本碰撞器，从而创建复合碰撞器。

通过灵活地调整位置和尺寸，复合碰撞器通常能够很好地近似于物体的形状，同时保持较低的处理器开销。在子元素上添加额外的碰撞器可以获得更好的灵活性（例如，盒模型可以相对于父物体的本地坐标轴旋转）。创建这样的复合碰撞器时，应该是有且仅有一个刚体组建，并且放置在层级结构的根元素上。

注意，基本碰撞器不能正确运行在 剪切变换 上——这意味着，如果在变换层级中组合使用旋转和非均匀缩放，将导致它的形状不再匹配基本形状，基本形状将不能正确滴表示它。

但是在某些情况下，即使是复合碰撞器也不够精确。在 3D 中，你可以使用 网格碰撞器 来精确匹配物体的网格形状。在 2D 中，2D 多边形碰撞器 通常不能完美地匹配精灵图像的形状，但是可以把细节细化任意你喜欢的程度。这些碰撞器的性能开销比基本类型更高，因此请谨慎使用它们，以保持良好的性能。另外，一个网格碰撞器通常不能与另一个网格碰撞器发生碰撞（也就是说，当她们接触时不会发生任何事情）。在某些情况下，你可以把网格碰撞器标记为 凸状 来克服这个问题。此时，将会『凸包』状的碰撞器形状，类似于原始网格，但是没有任何填充。这样做的好处是，一个凸状网格碰撞器可以与其他网格碰撞器发生碰撞，所以，你可以在一个具有合适形状的运动角色上使用这个功能。不过，通常好的规则是，对场景中的几何物体使用网格碰撞器，对运动物体使用近似于它形状的复合碰撞器来。

可以在没有刚体组件的对象上添加碰撞器，从而在场景中创建地板、墙壁和其他静止元素。这些元素被称为静态碰撞器。一般来说，你不应该通过修改变换组件的位置来重新定位静态碰撞器，因为这会严重影响物理引擎的性能。具有刚体组件的物体上的碰撞器碰撞器称为 动态碰撞器。静态碰撞器可以和动态碰撞器交互，但是因为它没有刚体组件，所以在响应碰撞时不会移动。

前文中各种碰撞器类型的参考页链接提供了有关属性和用法的更多信息。

物理材质

当碰撞器交互时，它们的表面需要模拟材质特性。例如，一片冰将是滑的，而一个橡胶球将提供很大的摩擦力并且非常有弹性。尽管碰撞器的形状在碰撞过程中不会变形，但是可以通过 物理材质 来配置它们的摩擦力和弹性。可能需要一些试验和试错才能得到正确的参数，不

过可以参考常识，例如，冰面材质的摩擦力为 0（或非常低），橡胶材质具有非常高的摩擦力和近乎完美的弹性。有关可用参数的更多细节请参阅 [物理材质](#) 和 [2D 物理材质](#) 的参考页。请注意，由于历史原因，物理材质称为 **Physic Material**（没有 S），2D 物理材质称为 **Physics Material 2D**（有 S）。

触发器

脚本系统可以检测碰撞发生的时间，并且用 `OnCollisionEnter` 函数初始化响应行为。不过，你也可以在不产生碰撞的情况下，用物理引擎简单地检测一个碰撞器何时进入另一个碰撞器的空间。一个配置为触发器的碰撞器（使用 **Is Trigger** 属性）不会具有实体对象的行为，并且允许其他碰撞器穿过它。当一个碰撞器进入触发器的空间时，将会调用触发器上脚本的 `OnTriggerEnter` 函数。

碰撞时执行的脚本操作

当碰撞发生时，物理引擎将会在相关对象绑定的所有脚本上调用特定名称的函数。可以在这些函数中放置任意代码来响应碰撞事件。例如，当汽车撞到障碍物时，可能会播放一段碰撞音效。

在检测到碰撞的第一次物理更新中，`OnCollisionEnter` 函数被调用。在连接尚未断开的期间，`OnCollisionStay` 被调用，最后调用 `OnCollisionExit`，表示连接已经断开。触发器调用类似的 `OnTriggerEnter`、`OnTriggerStay` 和 `OnTriggerExit` 函数。请注意，对于 2D 物理，提供了等价的函数，但是函数名后附加了 **2D**，例如 `OnCollisionEnter2D`。这些函数的完整信息和代码示例可以在 [MonoBehaviour](#) 类的脚本参考页中找到。

对于非触发器碰撞，还有一个细节需要注意，至少一个相关对象必须具有非运动学刚体（即必须关闭 **Is Kinematic**）。如果两个物体都是运动学刚体，那么 `OnCollisionEnter` 等函数不会被调用。使用触发器碰撞时，则不受此限制，因为，当运动学刚体和非运动学刚体进入一个触发器时，将会调用 `OnTriggerEnter` 函数。

碰撞器交互

碰撞器之间的交互行为依赖于 [刚体组件](#) 的配置。有三种重要配置：静态碰撞器（没有附加刚体）、刚体碰撞器和运动学刚体碰撞器。

静态碰撞器

一个具有碰撞器但是没有刚体的游戏对象。静态碰撞器用于水平集合物体，它们总是停留在相同的位置，从不移动。刚体物体将会和静态碰撞器发生碰撞，但是不会移动静态碰撞器。

物理引擎假定静态碰撞器从不移动或改变，基于这一假设可以执行有效的优化。因此，在游戏中，不应该禁用、启动、移动或缩放静态碰撞器。如果改变了静态碰撞器，将会导致物理引擎进行额外的内部运算，从而导致性能大幅下降。更糟的是，有时还会导致静态碰撞器处于未知状态，产生错误的物理运算。例如，可能无法检测向被改静态碰撞器发射的射线，或者随机返回空间中的一个位置。此外，被正在移动的静态碰撞器碰撞的刚体不一定会被『唤醒』，并且静态碰撞器也不会施加任何摩擦力。由于这些原因，只有刚体碰撞器才能被改变。如果想要一个不受其他刚体影响、并且可以用脚本移动的碰撞器对象，那么，应该绑定一个运动学刚体组件，而不是不绑定刚体。

刚体碰撞器

一个具有碰撞器和非运动学刚体的游戏对象。刚体碰撞器完全由物理引擎模拟，并且可以响应碰撞和脚本施加的作用力。刚体碰撞器可以和其他对象（包括静态碰撞器）发生碰撞，是游戏中最常用的碰撞器配置。

运动学刚体碰撞器

一个具有碰撞器和运动学刚体（即启动了 `IsKinematic` 属性）的游戏对象。脚本可以修改变换主见，从而移动运动学刚体对象，但是不会响应碰撞和作用力，就像非运动学刚体一样。运动学刚体应该用于偶尔移动、禁用或启用的碰撞器，但是在其他情况下，它应该像静态碰撞器那样运行。例如一扇滑动门，通常情况下，它作为不可移动的物理障碍，但是需要时可以被打开。与静态碰撞器不同，一个移动的运动学刚体碰撞器将对其他对象施加摩擦力，并在碰撞时唤醒其他刚体对象。

即使在不移动时，运动学刚体碰撞器的行为也与静态碰撞器不同。例如，如果静态碰撞器被设置为是一个触发器，那么为了在脚本中接收触发事件，你还需要为它添加一个刚体组件。如果希望触发器不受到重力或其他物理现象的影响，那么你可以在刚体组件上启用 `IsKinematic` 属性。

通过 `IsKinematic` 属性，一个刚体组件可以随时在正常和运动学之间切换。

一个常见的例子是『布偶』效果，正常情况下，一个角色在动画的控制下移动，但是可以被爆炸和重击抛出。这个角色的四肢各自被赋予刚体组件，并且默认启用 `IsKinematic` 属性。在 `IsKinematic` 关闭前，四肢将在动画的控制下移动，在关闭 `IsKinematic` 后，四肢立即表现出物理对象的行为。这个时候，碰撞或爆炸力将击飞这个角色，并且，它的四肢以逼真的方式被抛出。

碰撞行为矩阵

当两个对象碰撞时，根据刚体的配置，可能发生许多不同的脚本事件。下面的图表列出了基于对象组件调用事件函数的详细信息。某些组合只会导致两个对象中的一个被碰撞所影响，但一般规则是，没有附加刚体组件的对象不会应用物理现象。

译注：矩阵表格见[原文](#)。

连接

通过 **连接 Joint** 组件，你可以把一个刚体对象附加到另一个对象上，从而建立固定连接。一般来说，你想要的是一个自由度有限的连接，因为 Unity 提供了不同的连接组件来实施不同的限制。

例如，一个 [链条连接 Hinge Joint](#) 允许围绕特定点和轴线旋转，而一个 [弹簧连接 Spring Joint](#) 则保持物体分离并稍微拉开距离。

2D 连接组件的名字的末尾带有 **2D**，例如 [Hinge Joint 2D](#)。2D 连接的概述和背景信息请参考 [Joints 2D](#)。

连接还有其他可以实现特定效果的选项。例如，可以为连接设置断开效果，当施加到连接的作用力超过某个特定阈值时，连接将断开。某些连接还允许相连物体之间产生驱动力，从而自动地运动。

有关连接属性的更多信息，请查看 **Joint** 类的参考页。

角色控制器

第一人称或第三人称游戏中的角色通常需要一些基于碰撞的物理特性，以使它不会穿过地板掉落下去或穿过墙壁。角色的加速度和运动通常不是物理真实的，它可以立即加速、制动和改变方向，而不会收到动量的影响。

在 3D 物理中，这种类型的行为可以使用 角色控制器 创建。这个组件给角色提供了简单的胶囊碰撞器，并且总是直立向上。控制器有着自己的特殊功能，可以用来设置对象的速度和方向，与真正的碰撞器不同的是，它不需要刚体，动量效果也不真实。

角色控制器无法穿过场景中的静态碰撞器，因为会沿着地板行走并被墙壁阻挡。当它移动的时候，会把刚体对象推向一旁，但是自身不受碰撞影响。也就是说，你可以用标准的 3D 碰撞器来创建场景，而角色控制器可以不受真实物理行为限制地自由行走。

你可以在参考页中找到有关角色控制器额更多信息。

脚本



```

50     vignette.blur = (1-health) * 2 + smokeEffect * 10 * Health;
51     vignette.blurDistance = (1-health) * 2 * smokeEffect * 10;
52     vignette.chromaticAberration = heatEffect * 50;
53 }
54
55
56 void OnTriggerStay(Collider c)
57 {
58     var fire = c.GetComponent<Fire>();
59     if (fire && fire.alive)
60     {
61         float dist = 1-((transform.position - fire.transform.position).magnitude);
62         NearHeat(dist);
63     }
64
65     var smoke = c.GetComponent<SmokeStartEffect>();
66     if (smoke && smoke.GetComponent<HealthBox>())
67     {
68         float dist = 1-((transform.position - smoke.transform.position).magnitude);
69         NearSmoke(dist);
70     }
71 }
72
73 void OnCollisionEnter(Collision c)
74 {
75     var healthBox = c.gameObject.GetComponent<HealthBox>();
76     if (healthBox)
77     {
78         if (c.gameObject.healthBox)
    
```

脚本是所有游戏的基本要素。即使是最简单的游戏也会用到脚本，例如，在游戏中响应玩家的输入，并分派符合预期的事件。除此之外，脚本可以用来创建图形效果、控制对象的物理行为，甚至是为游戏中人物实现一套自定义 AI 系统。

编写脚本是一门技能，需要投入一定的时间和精力去学习。但本节的目的不是教你如何从头开始编写脚本，而是解释 Unity 脚本的主要概念。

相关教程：[脚本](#)

有关故障排除、技巧和窍门的内容，请参阅 [脚本编译错误知识库](#)。

脚本概述

Unity 采用了标准的 Mono 运行时来提供脚本功能，并扩展了自主的实践和技术，来支持脚本访问引擎。本节介绍如何通过脚本控制在 Unity 编辑器中创建的对象，并详细介绍 Unity 游戏功能和 Mono 运行时之间的关系。

译注：Mono 是 ECMA 通用语言基础架构（ECMA Common Language Infrastructure，CLI）的实现。关于 Mono 是如何提供脚本功能的，请参阅这篇 [从游戏脚本语言说起，剖析 Mono 所搭建的脚本基础](#)。

创建和使用脚本

游戏对象的行为由绑定的组件所控制。尽管 Unity 内置的组件非常灵活多样，但是你很快就会发现它们提供的功能远远不够，为了实现你所要的游戏功能，你需要超越它们才行。Unity 支持通过脚本创建属于你自己的组件。在组件中，随着时间的推移，你可以触发游戏事件、修改组件属性，还可以以任何你喜欢的方式来响应用户输入。

Unity 内置支持两种编程语言：

- **C#** 一种工业标准语言，类似于 Java 或 C++；
- **UnityScript** 一种专为 Unity 设计的语言，模仿自 JavaScript；

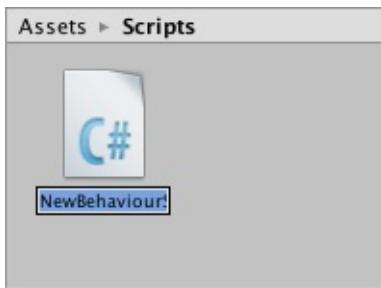
除此之外，许多其他可以编译为兼容 DLL 的语言也可以用于 Unity —— 更多细节请查阅 [这里](#)。

学习编程艺术和使用这些特定语言超出了本文的范围。不过，有许多书籍、指南和其他资源可以让你学习如何在 Unity 中编程。请参阅我们网站的 [学习部分](#) 了解更多细节。

创建脚本

与其他大多数资源文件不同的是，脚本通常是直接在 Unity 中创建。你可以通过 Project 面板左上角的 Create 菜单创建一个新脚本，或者从主菜单选择 **Assets > Create > C# Script** (或 JavaScript)。

新脚本将被创建在 Project 面板中选中的文件夹下。新脚本的文件名将被选中，提示你输入一个新的名称。



最好是在这个时候为新脚本输入名称，而不是在以后再修改。你输入的脚本名称将被用于初始化文件的内容，如下所述。

脚本文件剖析

当你在 Unity 中双击一个脚本文件时，它将被一个文件编辑器打开。默认情况下，Unity 会使用 MonoDevelop，你也可以在 Unity 偏好设置的 External Tools 面板中选择任何你喜欢的编辑器。

脚本文件的初始内容会是这个样子：

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

脚本通过继承内置类 **MonoBehaviour** 与 Unity 相连接。你可以把类当作是一种设计图，它可以创建一个新的组件，并绑定到游戏对象上。每当你把脚本组件绑定到游戏对象上时，会创建一个由设计图定义的对象实例。类的名称来自于文件创建时你提供的文件名。类名和文件名必须一致，这个脚本组件才能被绑定到游戏对象。

特别需要注意在类中定义的两个函数。函数 **Update** 用于放置处理游戏对象帧更新的代码。可能包括移动、触发动作和响应用户输入等，基本上包含了任何游戏运行时随着时间推移需要处理的事情。为了让 **Update** 函数能够工作起来，通常需要在任何游戏行为发生前，设置变量、读取配置和连接其他游戏对象。函数 **Start** 在游戏开始前被 Unity 调用（例如，第一次调用 **Update** 函数之前），是执行初始化的理想场所。

老司机请注意：你可能会惊讶于一个对象的初始化居然没有用到构造函数，这是因为，对象的构造过程是由编辑器处理的，而且不是发生在你所期望的游戏开始时。如果你试图为脚本组件定义一个构造函数，将妨碍 Unity 的正常运行，可能会引发严重问题。

UnityScript 脚本与 C# 脚本稍有不同：

```
#pragma strict

function Start () {

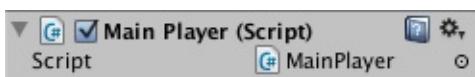
}

function Update () {
```

在这里，`Start` 和 `Update` 函数具有相同的含义，但是没有显示地声明类。脚本本身就被认为是对类的定义；它将隐式地继承 `MonoBehaviour`，并且把脚本文件名作为类名。

控制游戏对象

如上所述，脚本只是组件的设计图，在脚本实例被附加到游戏对象之前，它的代码不会被激活。你可以拖动脚本文件到层级视图的某个游戏对象上，或者拖到到当前选中的游戏对象的检视视图中。在 `Component` 菜单下有一个 `Scripts` 子菜单，其中包含了当前项目中的所有有效脚步，包括你创建的脚本。脚本实例看起来就像检视视图中的其他组件一样。



绑定之后，当你按下 `Play` 按钮运行游戏时，脚本就开始工作。你可以在 `Start` 函数中添加下面的代码来验证这一点：

```
// Use this for initialization
void Start () {
    Debug.Log("I am alive!");
}
```

`Debug.Log` 是一个简单的命令，只是向 `Unity` 的控制台输出打印一条消息。现在，如果你按下 `Play` 按钮，你会在 `Unity` 编辑器窗口底部和 `Console` 窗口（菜单：`Window > Console`）看到这条消息。

变量和检视视图

创建脚本的本质是创建新的组件类型，它可以像其他组件一样附加到游戏对象。

就像其他组件在检视视图中具有可编辑的属性一样，你也可以在检视视图中编辑脚本的属性值。

```
using UnityEngine;
using System.Collections;

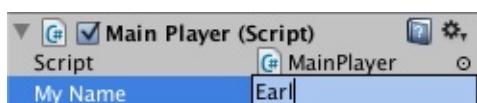
public class MainPlayer : MonoBehaviour {
    public string myName;

    // Use this for initialization
    void Start () {
        Debug.Log("I am alive and my name is " + myName);
    }

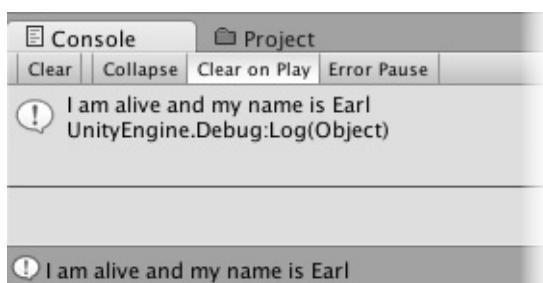
    // Update is called once per frame
    void Update () {

    }
}
```

这段代码在检视视图中创建了一个标签为『My Name』的可编辑字段，。



Unity 在检视视图中创建标签时，发现变量名中包含了一个大写字母，然后会在大写字母的位置插入一个空格。不过这纯粹是为了显示，你应该一直使用代码中的变量名。如果编辑这个名称，然后按下播放按钮，你看到日志信息中包含了你输入的文本。



在 C# 中，你必须把一个变量声明为公共变量，才能在检视视图中看到它。在 UnityScript 中，变量默认是公共的，除非被指定为私有变量。

```
#pragma strict

private var invisibleVar: int;

function Start () {

}
```

实际上，你可以在游戏运行时改变脚本变量的值。这点非常有用，无需停止和重新启动，就可以直接看到改变后的效果。当游戏结束时，变量的值将被重置为按下播放按钮之前的状况。你可以自由调整对象的设置，而无需担心造成任何永久性破坏。

使用组件控制游戏对象

在 Unity 编辑器中，你可以使用检视视图修改组件的属性。例如，改变游戏对象的变换组件的位置值，将会导致游戏对象的位置发生变化。类似地，修改渲染器材质的颜色或刚体的质量，将会对对象的外观或行为产生相应的影响。大多数情况下，脚本也可以修改组件属性，从而操纵游戏对象。不同的是，脚本可以随时间改变属性的值，或响应用户的输入。通过在正确的时间修改、创建和销毁对象，可以实现任何类型的游戏。

访问组件

最简单也最常见的情况是，脚本需要访问游戏对象上绑定的其他组件。正如简介部分所述，组件实际上是某个类的实例，所以第一步是拿到要访问的组件实例的引用。这一步通过 `GetComponent` 函数完成。通常，你需要把组件实例分配给一个变量。在 C# 中使用下面的语法完成：

```
void Start () {
    Rigidbody rb = GetComponent<Rigidbody>();
}
```

在 UnityScript 中，语法稍有不同：

```
function Start () {
    var rb = GetComponent.<Rigidbody>();
}
```

一旦有了组件实例的引用，就可以像在检视视图 Inspector 中一样，设置它的属性值：

```
void Start () {
    Rigidbody rb = GetComponent<Rigidbody>();

    // Change the mass of the object's Rigidbody.
    rb.mass = 10f;
}
```

一个额外特性是，脚本可以调用组件实例的函数，这在检视视图中是不可能的：

```

void Start () {
    Rigidbody rb = GetComponent<Rigidbody>();

    // Add a force to the Rigidbody.
    rb.AddForce(Vector3.up * 10f);
}

```

还需要注意到，同一个游戏对象没有理由不能绑定多个自定义脚本。如果需要从一个脚本访问另一个脚本，可以像平常一样使用 `GetComponent`，只需要用脚本类的名字（或文件名）指定所需的组件类型。

如果尝试检索一个实际上尚未添加到游戏对象的组件，`GetComponent` 函数将返回 `null`；如果尝试在一个 `null` 对象上改变任意值，会在运行时抛出一个空指针错误。

访问其他对象

尽管脚本有时是孤立地运行，但是脚本跟踪其他对象的状态也很常见。例如，一个追击的敌人可能需要知道被追击玩家的位置。Unity 提供了许多不同的方式来检索其他对象，每种方式对应特定的情况。

用变量连接对象

查找关联对象的最直接方式是，为脚本添加一个公共的游戏对象变量：

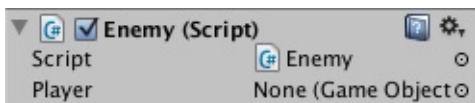
```

public class Enemy : MonoBehaviour {
    public GameObject player;

    // Other variables and functions...
}

```

这个变量将像其他变量一样显示在检视视图 Inspector 中。



现在，可以从场景视图或层级视图拖动一个对象到这个变量上进行分配。然后，就可以访问这个对象的 `GetComponent` 函数和组件变量。因此，你可以使用下面的代码：

```

public class Enemy : MonoBehaviour {
    public GameObject player;

    void Start() {
        // Start the enemy ten units behind the player character.
        transform.position = player.transform.position - Vector3.forward * 10f;
    }
}

```

另外，如果在脚本中声明了某个组件类型的公共变量，你可以拖动绑定了该组件的任意对象到该变量上。这时将直接访问组件，而不是游戏对象。

```
public Transform playerTransform;
```

当处理永久链接的独立对象时，用变量连接对象是最合适的方式。你可以用一个数组变量来连接多个同类型的对象，但是这种连接必须在 Unity 编辑器中建立，而不是在运行时。不过，在运行时定位对象也很方便，Unity 提供了两种基本方式来实现这点。

查找子对象

有时，一个游戏场景将使用多个相同类型的对象，例如敌人、路径和障碍物。可能需要一个特定脚本来跟踪、监督它们，或者还需要对它们进行响应（例如，用一个寻路脚本维护所有的路径）。虽然使用变量连接这些对象是一种可行的方式，但是如果每个新行路径都需要拖拽到某个脚本的变量上，将使设计过程变得非常乏味。同样，如果一个路径被删除，那么移除变量对无效对象的引用就会变成麻烦。在这样的情况下，通常最好是把它们都作为子对象放到一个父对象中，用一个对象集合来管理它们。子对象可以通过父对象的变换组件 `Transform` 来检索（因为所有游戏对象都隐含一个变换组件）：

```

using UnityEngine;

public class WaypointManager : MonoBehaviour {
    public Transform[] waypoints;

    void Start() {
        waypoints = new Transform[transform.childCount];
        int i = 0;

        foreach (Transform t in transform) {
            waypoints[i++] = t;
        }
    }
}

```

你还可以使用 `Transform.Find` 函数查找特定名称的子对象：

```
transform.Find("Gun");
```

在游戏过程中，如果父对象的子对象可以被添加和删除，这种方式可能很有用。一个很好的例子是，一把武器可以被捡起和放下。

按照名称或标签查找对象

只要场景中的游戏对象具有某些标识信息，那么，总是可以定位任意位置的游戏对象。可以用 [GameObject.Find](#) 函数按照名称检索单个对象：

```
GameObject player;

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

也可以使用 [GameObject.FindWithTag](#) 和 [GameObject.FindGameObjectsWithTag](#) 函数按照标签来定位对象或对象集合：

```
GameObject player;
GameObject[] enemies;

void Start() {
    player = GameObject.FindWithTag("Player");
    enemies = GameObject.FindGameObjectsWithTag("Enemy");
}
```

事件函数

传统编程的思路是，在一个循环结构中连续运行代码，直到完成任务。Unity 中的脚本则不同，Unity 通过间歇地调用脚本中声明的特定函数来控制脚本。一旦某个函数执行完成，控制权被交回 Unity。这些函数被称为事件函数，因为 Unity 通过激活它们来响应游戏中发生的各种事件。Unity 使用一套命名方案来唯一标识特定事件对应（调用）的函数。例如，你已经看到的 `Update` 函数（每帧更新之前被调用）和 `Start` 函数（游戏对象的第一帧更新前被调用）。Unity 提供了许多事件函数，可以在 `MonoBehaviour` 类的脚本参考页找到完整列表和用法的详细信息。下面是一些最常用和最重要的事件。

定期更新事件

一个游戏更像一段动画，因为动画帧是实时生成的。游戏编程中的一个关键概念是，在每一帧渲染之前，更改游戏中对象的位置、状态和行为。在 Unity 中，`Update` 函数是放置这类代码的主要地方。`Update` 在帧渲染和动画计算之前被调用。

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

物理引擎也以离散的时间步长进行更新，类似于帧渲染。每次物理更新之前，函数 `FixedUpdate` 被调用。因为物理更新和帧更新以不同的频率发生，所以，如果你把物理代码放入 `FixedUpdate` 而不是 `Update`，可以获得更准确的结果。

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

当场景中所有对象的 `Update` 和 `FixedUpdate` 函数调用完，并且所有动画计算完成之后，执行一些额外的更改可能会很有用。一个例子是摄像机跟踪某个目标对象时，对摄像机方位的调整必须发生在目标对象移动之后。另一个例子是脚本代码想要覆盖动画效果（例如，让角色头部面向场景中的某个目标）。`LateUpdate` 函数可以用于这类情况。

译注：既然 `Update` 和 `FixedUpdate` 的执行频率不同，那么怎么保证 `LateUpdate` 的顺序呢？看了 [事件函数执行顺序] 里的流程图还是不确定。

```
void LateUpdate() {
    Camera.main.transform.LookAt(target.transform);
}
```

初始化事件

在游戏期间发生任何更新之前，能够调用初始化代码通常很有用。在对象的第一帧或第一次物理更新之前，[Start](#) 函数被调用。在场景加载时，为场景中的每个对象调用 [Awake](#) 函数。请注意，尽管各种对象的 [Start](#) 和 [Awake](#) 函数以任意顺序被调用，但是在第一个 [Start](#) 函数被调用之前，所有的 [Awake](#) 函数都应该已经完成。这意味着，[Start](#) 函数中的代码可以使用之前 [Awake](#) 阶段执行的初始化（赋值）。

GUI 事件

Unity 提供了一套 GUI 控件渲染系统，用于控制场景中的主要操作和响应对控件的点击。这种代码与正常的帧更新有些不同，因为应该放入 [OnGUI](#) 函数，该函数会被周期性调用。

```
void OnGUI() {
    GUI.Label(labelRect, "Game Over");
}
```

你还可以检测场景中显示的游戏对象上发生的鼠标事件。这可以用于武器瞄准，或显示当前鼠标指针所指角色的信息。脚本使用一组 [OnMouseXXX](#) 事件函数（例如 [OnMouseOver](#)、[OnMouseDown](#)）响应用户鼠标的行为。例如，如果按下鼠标按钮，并且指针位于一个特定对象之上时，那么该对象上脚本的 [OnMouseDown](#) 函数将被调用（如果存在的话）。

物理事件

物理引擎将通过调用脚本的事件函数，来报告与对象的碰撞。当建立、保持和断开连接时（碰撞时两个物体连接在一起），[OnCollisionEnter](#)、[OnCollisionStay](#) 和 [OnCollisionExit](#) 函数将被调用。如果对象的碰撞器被配置为触发器（例如，简单地检测某个对象是否进入了碰撞器，而不是响应物理行为），相应的 [OnTriggerEnter](#)、[OnTriggerStay](#) 和 [OnTriggerExit](#) 函数将被调用。在物理更新期间，如果检测到多个连接，那么这些函数可能会连续多次被调用，因此，会传给函数一个含有碰撞信息的参数（位置、相关对象的标识等）。

```
void OnCollisionEnter(otherObj: Collision) {
    if (otherObj.tag == "Arrow") {
        ApplyDamage(10);
    }
}
```

时间和帧率控制

脚本中的 **Update** 函数允许你定期地监听输入和其他事件，并执行适当的响应。例如，当按下『向前』键时，你可以会移动一个角色。处理类似这样的基于时间的行为时，一件重要的事情是，游戏的帧率不是恒定的，两次 **Update** 函数调用之间的时间长度也不是恒定的。

举个例子，假设有一个逐渐向前移动对象的任务，每帧移动一次。开始时，你可能只是使对象每帧移动一个固定距离：

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    public float distancePerFrame;

    void Update() {
        transform.Translate(0, 0, distancePerFrame);
    }
}
```

```
//JS script example
var distancePerFrame: float;

function Update() {
    transform.Translate(0, 0, distancePerFrame);
}
```

但是，因为桢时间不是恒定的，所以对象看起来将以不规则的速度移动。如果桢时间是 10 毫秒，那么该对象每秒向前移动 100 步 **distancePerFrame**。但是，如果桢时间增加到 25 毫秒（例如由于 CPU 负载），那么只会每秒向前移动 40 步，因此移动距离更短。解决方案是按照桢时间适配移动距离，可以从 **Time.deltaTime** 属性读取桢时间：

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    public float distancePerSecond;

    void Update() {
        transform.Translate(0, 0, distancePerSecond * Time.deltaTime);
    }
}
```

```
//JS script example
var distancePerSecond: float;

function Update() {
    transform.Translate(0, 0, distancePerSecond * Time.deltaTime);
}
```

注意，现在用 `distancePerSecond` 表示移动距离而不是 `distancePerFrame`。当帧率改变时，移动步长将相应地改变，对象的速度将是恒定的。

固定时间步长

与主桢更新不同，Unity 的物理系统以固定的时间步长运行，这对模拟的精确性和一致性非常重要。当物理更新开始时，Unity 在时间轴上设置一个固定时间步长（后将消失）的『警告器』，用来表示物理更新的结束时间。然后，物理更新开始执行计算，直到『警告器』消失。

通过时间管理器可以更改固定时间步长的大小，在脚本中，可以使用 `Time.fixedDeltaTime` 属性读取它的值。注意，较小的时间步长将导致更频繁的物理更新和更精确的模拟，代价是更大的 CPU 负载。除非对物理引擎有很高的要求，否则可能不需要更改默认的固定时间步长。

译注：时间管理器 Time Manager 的菜单位置 **Edit > Project Settings > Time**。

最大时间步长

固定时间步长保证了物理模拟的实时精确，但是，当游戏使用了大量物理模拟时，可能会导致游戏帧率降低（由于有大量对象在运行）。频繁的物理更新会『挤压』主桢更新的时间，并且，如果有大量处理需要执行，那么一帧时间内可能发生多次物理更新。这时，当桢更新开始时，对象的位置和其他属性被冻结，图像可能与频繁的物理更新不同步。

尽管事实上只有有限的 CPU 功率可用，但是 Unity 提供了一个选项，可以让你有效地降低物理时间，从而让帧处理保持（恢复）同步。最大时间步长（在时间管理器中）限制了一帧中消耗在物理更新和 `FixedUpdate` 调用上的时间。如果帧更新消耗的时间超过了最大时间步长，物理引擎将暂停执行，从而让帧更新保持（赶上）同步。一旦帧更新完成，物理更新将恢复执行，就像自它停止后时间没有流逝一样。这样做的结果是，刚体将不会像通常那样完美地实时移动，而是稍微减慢。但是，物理时钟将会像正常移动一样跟踪它们。这种物理更新放慢通常不明显，是一种可接受的游戏性能平衡。

时间尺度

减慢游戏时间对于某些特效会很有用，例如『子弹时间』，可以使动画和脚本响应以降低后的速率运行。另外，有时可能想要完全冻结游戏时间，此时游戏被暂停。Unity 提供了一个时间尺寸 `Time Scale` 来控制游戏时间相对于真实时间的速度。如果时间尺寸设置为 1.0，那么游戏时间和真实时间一致。设置为 2.0，将使 Unity 中的游戏时间两倍于真实时间（例如，动作将加速）；而设置为 0.5，将使游戏速度降低一半。设置为 0 将使事件完全『停止』。注意，时间尺寸不会真正减慢执行过程，而是简单地改变了 `Time.deltaTime` 和 `Time.fixedDeltaTime` 返回给 `Update` 和 `FixedUpdate` 函数的时间步长。当游戏时间降低时， `Update` 函数的调用比正常情况更频繁，只是每帧返回的 `deltaTime` 被简单地降低了。其他脚本函数不会受到时间尺度的影响，例如，在游戏暂停时，显示可正常交互的 GUI。

译注：子弹时间（Bullet time）是一种使用在电影、电视广告或电脑游戏中，用计算机辅助的摄影技术模拟变速特效，例如强化的慢镜头、时间静止等效果。“子弹时间”效果因在好莱坞华纳兄弟电影公司出品的电影《骇客帝国》中大量使用名声大噪。其中男主角 Neo 仰身躲子弹的慢动作镜头堪称经典，“子弹时间”也因此得名。—— [百度百科](#)

时间管理器提供了一个可全局设置时间尺度的属性，不过，通常是在脚本中使用 `Time.timeScale` 属性设置时间尺度：

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    void Pause() {
        Time.timeScale = 0;
    }

    void Resume() {
        Time.timeScale = 1;
    }
}
```

```
//JS script example
function Pause() {
    Time.timeScale = 0;
}

function Resume() {
    Time.timeScale = 1;
}
```

捕获帧率

时间管理的一个特例是把游戏记录为视频。如果你尝试在正常的游戏过程中录制视，因为保存屏幕图像需要相当长的时间，游戏帧率通常会大大降低。这将导致视频不能真实反应游戏的性能。

幸运的是，Unity 提供了一个捕获帧率 `Capture Framerate` 属性来解决这个问题。当该属性的值被设置为非 0 时，游戏速度将降低，桢更新将以精确的时间间隔定期执行。两桢之间的间隔等于 `1 / Time.captureFramerate`，因此，如果该值被设置为 5.0，那么桢更新每 1/5 秒发生一次。随着帧率的有效降低，`Update` 函数有充足的时间来保存屏幕截图或执行其他行为。

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    // Capture frames as a screenshot sequence. Images are
    // stored as PNG files in a folder - these can be combined into
    // a movie using image utility software (eg, QuickTime Pro).
    // The folder to contain our screenshots.
    // If the folder exists we will append numbers to create an empty folder.
    string folder = "ScreenshotFolder";
    int frameRate = 25;

    void Start () {
        // Set the playback framerate (real time will not relate to game time after this).
        Time.captureFramerate = frameRate;

        // Create the folder
        System.IO.Directory.CreateDirectory(folder);
    }

    void Update () {
        // Append filename to folder name (format is '0005 shot.png')
        string name = string.Format("{0}/{1:D04} shot.png", folder, Time.frameCount );

        // Capture the screenshot to the specified file.
        Application.CaptureScreenshot(name);
    }
}
```

```
//JS script example

// Capture frames as a screenshot sequence. Images are
// stored as PNG files in a folder - these can be combined into
// a movie using image utility software (eg, QuickTime Pro).
// The folder to contain our screenshots.
// If the folder exists we will append numbers to create an empty folder.
var folder = "ScreenshotFolder";
var frameRate = 25;

function Start () {
    // Set the playback framerate (real time will not relate to game time after this).
    Time.captureFramerate = frameRate;

    // Create the folder
    System.IO.Directory.CreateDirectory(folder);
}

function Update () {
    // Append filename to folder name (format is '0005 shot.png')
    var name = String.Format("{0}/{1:D04} shot.png", folder, Time.frameCount );

    // Capture the screenshot to the specified file.
    Application.CaptureScreenshot(name);
}
```

使用这种技术录制的视频虽然通常看起来很不错，但是当游戏速度大幅降低时，游戏可能没法完了。为了保证充足的记录时间，并避免使播放器任务过度复杂化，你可能需要反复实验 `Time.captureFramerate` 的值。

创建和销毁游戏对象

某些游戏在场景中维护恒定数量的对象，但是在游戏过程中，创建和移除人物、物品以及其他对象也非常普遍。在 Unity 中，一个游戏对象可以通过 `Instantiate` 函数创建一个已有对象的新副本。

```
public GameObject enemy;

void Start() {
    for (int i = 0; i < 5; i++) {
        Instantiate(enemy);
    }
}
```

需要注意的是，被复制的原始对象不一定必须是场景中的对象。比较常见的是，将一个预制对象拖动到编辑器项目视图的一个公共变量上。而且，初始化一个游戏对象将复制原始对象上的所有组件。

还有一个 `Destroy` 函数用于在帧更新完成之后或者一段可选的短暂延迟之后销毁对象：

```
void OnCollisionEnter(Collision otherObj) {
    if (otherObj.gameObject.tag == "Missile") {
        Destroy(gameObject, .5f);
    }
}
```

请注意，`Destroy` 函数可以单独销毁某些组件而不影响游戏对象本身。下面是一个常见的错误：

```
Destroy(this);
```

上面这行代码实际上只是销毁被调用的脚本组件，而不是绑定了该脚本组件的游戏对象。

协同程序

当调用一个函数时，在它返回之前，会一直运行到完成。这意味着该函数中的任何动作都必须在一帧内完成；函数调用不能包含过程动画或一段时间内的事件序列。例如有这样一个任务，逐渐降低一个对象的 `alpha`（不透明度）值，直到它完全不可见。

```
void Fade() {
    for (float f = 1f; f >= 0; f -= 0.1f) {
        Color c = renderer.material.color;
        c.a = f;
        renderer.material.color = c;
    }
}
```

实际情况是，函数 `Fade` 不会实现你期望的效果。为了使渐变过程可见，`alpha` 必须随着桢序列降低，以渲染显示中间值。但是，该函数将在一帧内完整地执行。你将永远不会看到中间值，对象会立即消失。

可以把代码添加到 `update` 函数中，逐桢地执行淡出，来处理这种情况。不过，更方便的方式是使用协程（协同程序）执行这种任务。

协程就像一个函数，它能够暂停执行并将控制权返回给 Unity，但是在下一桢时，可以在暂停的位置继续执行。在 C# 中，可以像这样声明协程：

```
IEnumerator Fade() {
    for (float f = 1f; f >= 0; f -= 0.1f) {
        Color c = renderer.material.color;
        c.a = f;
        renderer.material.color = c;
        yield return null;
    }
}
```

协程本质上是一个返回类型被声明为 `IEnumerator` 的函数，并且在函数体的某处包含 `yield return` 语句。执行过程在 `yield return` 行暂停，并在下一桢恢复执行。要让协程运行起来，需要使用 `StartCoroutine` 函数：

```
void Update() {
    if (Input.GetKeyDown("f")) {
        StartCoroutine("Fade");
    }
}
```

在 UnityScript 中，事情稍微简单一些。任何含有 `yield` 语句的函数都被认为是一个协程，不需要显示声明返回类型 `IEnumerator`：

```
function Fade() {
    for (var f = 1.0; f >= 0; f -= 0.1) {
        var c = renderer.material.color;
        c.a = f;
        renderer.material.color = c;
        yield;
    }
}
```

此外，在 UnityScript 中，可以通过直接调用协程来启动它，就像它是一个普通的函数一样：

```
function Update() {
    if (Input.GetKeyDown("f")) {
        Fade();
    }
}
```

你将会注意到，在协程的生命周期内，`Fade` 函数中的循环计数器一直保持正确的值。实际上，`yield` 之间的任何变量或属性都将正确地保留。

默认情况下，协程在 `yield` 之后的帧中恢复，不过也可以使用 `WaitForSeconds` 延迟恢复：

```
IEnumerator Fade() {
    for (float f = 1f; f >= 0; f -= 0.1f) {
        Color c = renderer.material.color;
        c.a = f;
        renderer.material.color = c;
        yield return new WaitForSeconds(.1f);
    }
}
```

在 UnityScript 中：

```
function Fade() {
    for (var f = 1.0; f >= 0; f -= 0.1) {
        var c = renderer.material.color;
        c.a = f;
        renderer.material.color = c;
        yield WaitForSeconds(0.1);
    }
}
```

协程可以把某些效果分散在一段时间内，也可以有效地优化性能。游戏中的许多任务需要定期执行，最明显的方式是将它们包含在 `Update` 函数中执行。但是 `Update` 函数通常每秒调用多次。当任务不需要如此频繁地重复时，你可以把它放入协程定期更新，而不是每帧都更新。一个例子是在敌人靠近玩家时触发警告。代码看起来可能像这样：

```
function ProximityCheck() {
    for (int i = 0; i < enemies.Length; i++) {
        if (Vector3.Distance(transform.position, enemies[i].transform.position) < dangerDistance) {
            return true;
        }
    }

    return false;
}
```

如果有很多敌人，每帧都调用该函数可能会带来很大的开销。不过，你可以使用协程每秒调用该函数 10 次：

```
IEnumerator DoCheck() {
    for (;;) {
        ProximityCheck;
        yield return new WaitForSeconds(.1f);
    }
}
```

这将大大减少执行检测的次数，而且不会对游戏性产生任何显著影响。

特殊文件夹和脚本的编译顺序

大多数情况下，你可以为项目中的文件夹选择任意你喜欢的名称，但是 Unity 因为一些特殊原因而保留了一些名称。其中，有部分命名会影响到脚本的编译顺序。从本质上讲，脚本编辑过程分为 4 个独立的阶段，而一个脚本何时被编译，则取决于它所在的父文件夹。

当一个脚本必须引用其他脚本中的类时，这个问题变得尤为明显。基本规则是，先编译的脚本不能引用后编译的脚本。也就是说，处于同一编译阶段的所有脚本都是可引用的，所有更早编译的脚本也是可引用的。

还有一种特殊情况，一个脚本是用某种语言编写的，它必须引用另外一个脚本，而第二个脚本是用别的语言编写的（例如，一个 **UnityScript** 文件声明了定义在 C# 文件中的类的变量）。这种情况的规则是，被引用的类必须在早期阶段被编译。

4 个编译阶段如下：

第 1 阶段：文件夹 **Standard Assets**、**Pro Standard Assets** 和 **Plugins** 中的运行时脚本。

第 2 阶段：文件夹 **Standard Assets**、**Pro Standard Assets** 和 **Plugins** 下 **Editor** 中的编辑器扩展脚本。

第 3 阶段：不在文件夹 **Editor** 中的所有脚本文件。

第 4 阶段：所有剩余的文件（例如，文件夹 **Editor** 中的脚本文件）。

一个常见的例子是，一个 **UnityScript** 文件需要引用 C# 文件中的类。你可以把 C# 文件放入 **Plugins** 文件夹，把 **UnityScript** 放入普通文件夹。如果不这么处理，就会报错，提示找不到 C# 类。

注意：文件夹 **Standard Assets** 只在根目录 **Assets** 下起作用。

事件函数执行顺序

在 Unity 脚本中，有大量的事件函数以特定的顺序被执行。执行顺序描述如下：

编辑器

- **Reset**

当第一次把脚本绑定到对象上，或者使用了 **Reset** 命令时，该事件被触发，用以初始化脚本的属性。

加载第一个场景

当某个场景开始时，下面的事件被触发（场景中的每个对象都会执行一次）：

- **Awake**

该函数总是在所有 **Start** 函数之前被调用，并且只会在某个 **prefab** 被实例化之后才会被调用。（如果一个 **GameObject** 在启动时是非激活状态，那么 **Awake** 函数不会被调用，直到这个 **GameObject** 处于激活状态。）

- **OnEnable**

（只有对象处于激活状态才会被调用）该函数在对象处于可用状态之后被调用。当一个 **MonoBehaviour** 实例被创建时，就会调用该函数。例如关卡加载完成、某个带有脚本组件的 **GameObject** 被实例化后。

- **OnLevelWasLoaded**

该函数在某个新关卡加载完成后被执行，用来向游戏通知新关卡已经被载入。

请注意，对于场景 **scene** 中已有对象上附加的脚本组件，函数 **Awake** 和 **OnEnable** 将在所有 **Start**、**Update** 等函数之前被调用。当然，如果某个对象是游戏运行时实例化的，那么不遵循这条原则。

第一帧更新之前

- **Start**

该函数在第一帧更新之前被调用，前提是该脚本实例必须是可用的。

对于场景 `scene` 中已有对象上附加的脚本组件，函数 `Start` 在 `Update` 等函数之前被调用。当然，如果某个对象是游戏运行时实例化的，那么不遵循这条原则。

帧间

- **OnApplicationPause**

如果游戏处于暂停状态，该函数在某一个帧的末尾被调用，也就是说，是在正常帧（更新）之间被调用。当 `OnApplicationPause` 被调用后，一个特殊的帧被创建，这样游戏可以显示暂停状态的图形。

帧更新顺序

当你跟踪游戏的逻辑、交互、动画、摄像机位置等时，也有几个事件可供使用。通常我们是在 `Update` 函数中执行大部分任务，但也可以使用其他的函数。

- **FixedUpdate**

通常， `FixedUpdate` 比 `Update` 调用的更频繁。如果帧率很低，可以在每一帧上多次调用 `FixedUpdate`；如果帧率很高， `FixedUpdate` 根本不会被调用。调用 `FixedUpdate` 之后，所有的物理计划和更新会立即生效，如果在 `FixedUpdate` 中执行位移计算，你就不需要基于 `Time.deltaTime` 来计算。因为 `FixedUpdate` 基于一个可靠的计时器，与帧率无关。

- **Update**

每帧调用一次 `Update`。对于帧更新来说， `Update` 是主要的任务承载函数。

- **LateUpdate**

`LateUpdate` 在 `Update` 完成之后被调用，每帧调用一次。当 `LateUpdate` 开始执行时， `Update` 中执行的所有计算都已完成。如果你在 `Update` 中移动和旋转角色，那么你可以在 `LateUpdate` 中移动和旋转摄像机。这样，在摄像机跟随角色的位置之前，可以确保角色的移动已经完成。

渲染

- **OnPreCull**

在摄像机对场景进行 `Culling` 之前被调用。`Culling` 确定了哪些对象对于摄像机是可见的。

译注 Unity 中的优化技术

- **OnBecameVisible/OnBecameInvisible**

当某个对象对于任意摄像机变为可见或不可见时被调用。

- **OnWillRenderObject**

为每个摄像机调用一次，如果该对象是可见的。

- **OnPreRender**

在摄像机开始渲染场景之前被调用。

- **OnRenderObject**

在常规场景渲染完成之后被调用。此时，你可以使用类 [GL](#) 或 [Graphics.DrawMeshNow](#) 绘制自定义的几何体。

- **OnPostRender**

当某个摄像机完成渲染场景之后被调用。

- **OnRenderImage**

在场景渲染完成之后被调用，用于图像后处理，请查看 [ImageEffects](#)。

- **OnGUI**

用于响应 GUI 事件，每一帧会多次调用。首先处理 Layout 和 Repaint 事件，然后是 Layout，以及每次用户输入触发的 keyboard/mouse 事件。

- **OnDrawGizmos**

用于在场景视图中绘制 Gizmos，使之可视化。

协同程序

通常，协同更新在函数 `Update` 返回后运行。一个协同程序是一个可以暂停运行过程的函数，当给定的 `YieldInstruction` 完成时继续执行。协同程序的不同用法如下：

- **yield**

下一帧中的所有 `Update` 函数被调用后，协同程序将继续执行。

- **yield WaitForSeconds**

当前帧的所有 `Update` 函数被调用后，并且延迟给定的时间，协同程序将继续执行。

- **yield WaitForFixedUpdate**

所有脚本的 `FixedUpdate` 函数被调用后，协同程序将继续执行。

- **yield WWW**

网络下载完成后，协同程序将继续执行。

- **yield StartCoroutine**

将协同程序串联起来，等待 MyFunc 执行完成后，继续链式执行。

当对象被销毁时

- **OnDestroy**

在对象存在的最后一帧，当所有帧更新都完成后，该函数被调用（该对象可能因为调用了 Object.Destroy 而被销毁，也可能随着场景的关闭而销毁）。

当退出时

这些函数会在场景中的所有激活对象上调用。

- **OnApplicationQuit**

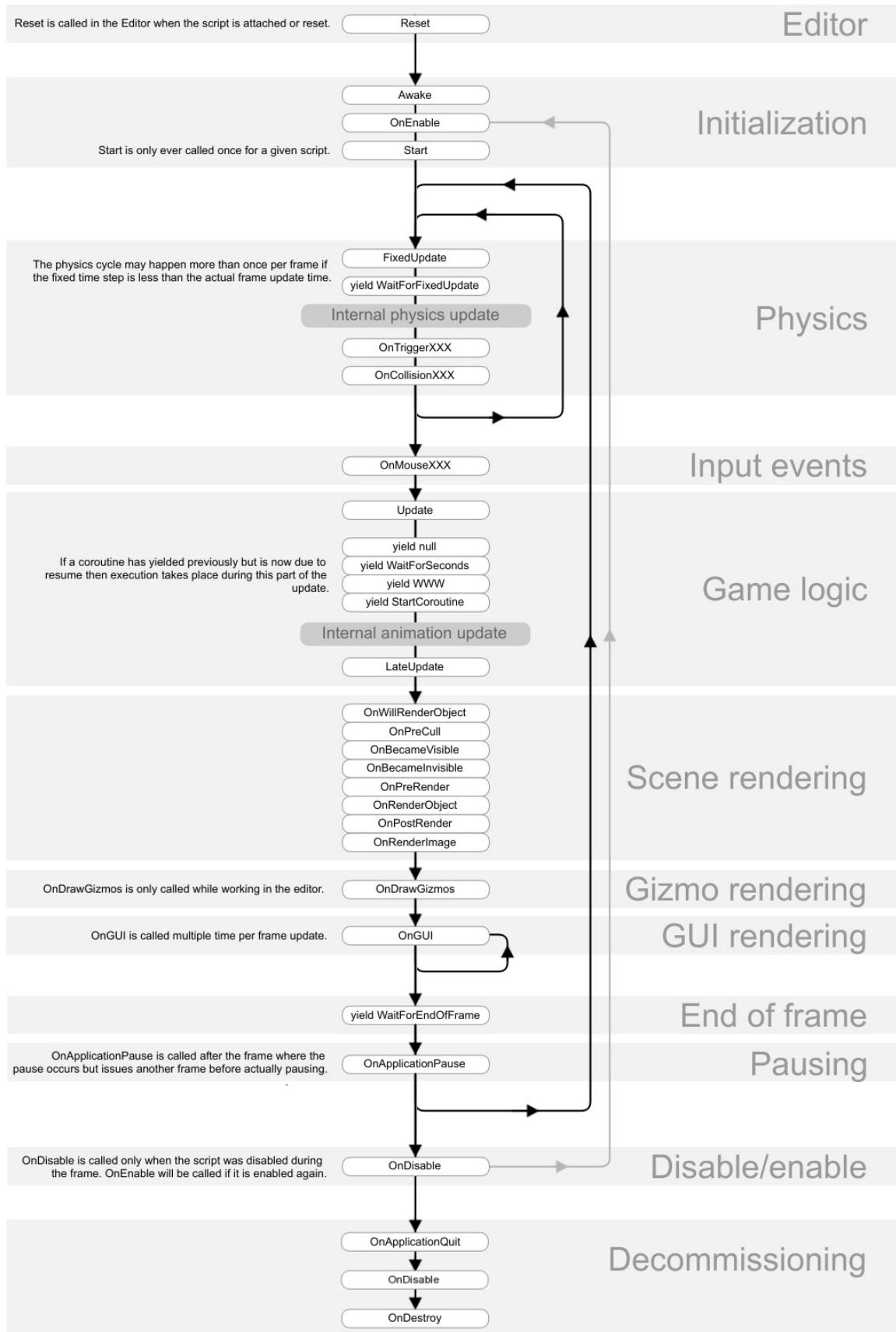
在应用程序退出前，在所有游戏对象调用该方法。如果是在编辑器中，那么当用户停止游戏模式时，该函数被调用。

- **OnDisable**

当游戏对象的行为变为禁用或不活动时，该函数被调用。

脚本生命周期流程图

下图总结了脚本生命周期中事件函数的执行顺序和重复周期。



理解自动内存管理

当创建一个对象、字符串或数组时，会从名为 堆 的中央池中分配一块内存，用来存储创建的值。当这些值不再被使用时，被占用的内存可以被回收，并用于存储其他的值。在过去，是由程序员显式地调用相应的函数分配和释放堆内存。如今，像 Unity Mono 引擎这样的运行时系统，可以自动地管理内容。相比显式地分配和释放内存，自动内存管理需要的编码工作更少，并且大大降低了发生内存泄露的可能性（例如，分配内存后一直不释放的情况）。

值和引用类型

当调用一个函数时，参数值被复制到一块专门用于本次调用的内存区。对于数据类型，它们只占用很少的字节，可以非常迅速和容易地复制。但是，常见的对象、字符串和数组则大得多，如果频繁地复制这些类型的数据，是非常低效的。幸运的是，没必要这么做；大型值的实际存储空间从堆分配，然后用一个小巧的『指针』值记录下它的存储位置。这样，在传递参数的过程中，只有这个指针被复制。既然运行时系统可以通过这个指针定位到实际的值，那么，在必要时可以使用它的副本。

在传递参数的过程中，直接存储和复制的类型称为『值类型』，包括整型、浮点型、布尔型和 Unity 的结构类型（例如 Color、Vector3）。在堆中存储、然后用一个指针访问的类型成为『引用类型』，因为存储在变量中的值只是『指向』了真实值。引用类型的例子包括对象、字符串和数组。

分配和垃圾回收

内存管理器会一直跟踪堆的状态，知道哪些区域是闲置的。当请求一块新的内存区域时（意味着一个新对象被创建），管理器从闲置区域中选择一块，并从闲置区域中移除它。后续的请求被执行同样的处理，直到闲置区域不足以满足请求的尺寸。所有堆内存都被使用的可能性极小。堆上的引用类型只能通过引用变量访问，如果对某块内存区域的引用全都消失了（例如，引用变量被重新赋值，或者它们只是局部变量并且离开了作用域），那么这块内存区域可以被安全地重新分配。

为了确定哪些区域不再被使用，内存管理器会遍历当前所有有效的引用变量，并把他们所引用的区域标记为『活动』。遍历结束后，未被标记为『活动』的区域都被内存管理器认为是闲置的，可以用于后续的分配。定位和释放内存的过程被直观地称为垃圾回收（简写为 GC）。

优化

垃圾回收运行在后台，因此对于程序员来说是自动的、不可见的，但实际上，回收过程需要耗费相当的 CPU 时间。如果使用得当，自动内存管理的整体性能通常与手动分配相当或者更好。然后，程序员要注意避免频繁地触发不必要的回收，从而导致执行过程暂停。

有一些臭名昭著的算法堪称是 GC 噩梦，即使它们初看似乎没什么问题。一个典型的例子是字符串重复拼接：

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    void ConcatExample(int[] intArray) {
        string line = intArray[0].ToString();

        for (i = 1; i < intArray.Length; i++) {
            line += ", " + intArray[i].ToString();
        }

        return line;
    }
}

//JS script example
function ConcatExample(intArray: int[]) {
    var line = intArray[0].ToString();

    for (i = 1; i < intArray.Length; i++) {
        line += ", " + intArray[i].ToString();
    }

    return line;
}
```

这里的关键细节是，新片段并没有被添加到已有的字符串之后。事情的真相是，每执行一次循环，变量 `line` 的旧内容被丢弃，一个全新的字符串被创建，用来包含旧内容和新增部分。随着变量 `i` 的增加，字符串变得越来越长，消耗的堆空间也随之增长；每当这个函数被调用，就会用掉成百上千个字节的闲置堆空间。如果你需要拼接许多字符串，更好的选择是使用 Mono 库的 `System.Text.StringBuilder` 类。

不过，字符串反复拼接并不会造成太大的麻烦，除非你频繁地调用，而在 Unity 中，字符串拼接通常是为了帧更新，就像这样：

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    public GUIText scoreBoard;
    public int score;

    void Update() {
        string scoreText = "Score: " + score.ToString();
        scoreBoard.text = scoreText;
    }
}

//JS script example
var scoreBoard: GUIText;
var score: int;

function Update() {
    var scoreText: String = "Score: " + score.ToString();
    scoreBoard.text = scoreText;
}
```

每次 `Update` 被调用，将分配一个新字符串，以恒定地速率产生新垃圾。通常我们可以这样优化这种情况：只有当比分更新时，才更新文本。

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    public GUIText scoreBoard;
    public string scoreText;
    public int score;
    public int oldScore;

    void Update() {
        if (score != oldScore) {
            scoreText = "Score: " + score.ToString();
            scoreBoard.text = scoreText;
            oldScore = score;
        }
    }
}

//JS script example
var scoreBoard: GUIText;
var scoreText: String;
var score: int;
var oldScore: int;

function Update() {
    if (score != oldScore) {
        scoreText = "Score: " + score.ToString();
        scoreBoard.text = scoreText;
        oldScore = score;
    }
}
```

当函数返回数组时，会引发另外一个潜在问题：

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    float[] RandomList(int numElements) {
        var result = new float[numElements];

        for (int i = 0; i < numElements; i++) {
            result[i] = Random.value;
        }

        return result;
    }
}
```

```
//JS script example
function RandomList(numElements: int) {
    var result = new float[numElements];

    for (i = 0; i < numElements; i++) {
        result[i] = Random.value;
    }

    return result;
}
```

这种函数创建了一个填满值的数组，看起来非常优雅和方便。但是，如果反复调用它，那么每次都会分配新的内存。因为数组可能非常大，所以闲置堆空间可能很快被用完，进而导致频繁的垃圾回收。避免这个问题的方式是，利用数组是引用类型这一事实。把数组作为参数传入函数，在函数内部修改这个数组，当函数返回后，数组中的值依然有效。上面的函数可以替换为下面这个：

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    void RandomList(float[] arrayToFill) {
        for (int i = 0; i < arrayToFill.Length; i++) {
            arrayToFill[i] = Random.value;
        }
    }
}

//JS script example
function RandomList(arrayToFill: float[]) {
    for (i = 0; i < arrayToFill.Length; i++) {
        arrayToFill[i] = Random.value;
    }
}
```

在上面的代码中，用新值替换了数组中的已有内容。尽管这种方式需要在调用函数的代码中完成数组的初始化分配（看起来不怎么优雅），但是这个函数被调用时将不再产生任何新的垃圾。

请求一个集合

如上所述，最好是尽可能地避免分配。但是，鉴于不可能完全消除分配的事实，有两种主要策略可以最小化分配对游戏的影响：

快节奏地分配小堆 + 频繁地内存回收

这一策略对于需要平稳帧率、长时间运行的游戏非常有效。这类游戏通常会频繁地分配小块内存，并且只是短暂地使用这些小块内存。在 iOS 上使用这种策略时，典型的堆大小是 200KB 左右，以 iPhone 3G 为例，内存回收大约耗时 5ms；如果堆大小增加到 1MB，内存回收将耗时约 7ms。因此这种策略是有效的，最理想的情况是，有时内存回收会发生在常规帧之间。尽管这种策略会导致更频繁的内存回收，但是回收非常快，最小化了对游戏的影响：

```
if (Time.frameCount % 30 == 0)
{
    System.GC.Collect();
}
```

不过，你应该谨慎地使用这项技术，检查性能统计数据，以确保真的降低了内存回收时间。

慢节奏地分配大堆 + 不频繁地内存回收

这种策略对于分配和回收相对不频繁、可以在游戏暂停期间处理的游戏非常有效。在分配尽可能大的堆后，有些操作系统会因为系统内存不足而杀死应用，这种策略对于不会杀死应用的操作系统非常有用。不过，Mono 在运行时会尽可能不自动去扩展堆大小。你可以在启动时通过预分配占位空间的方式，手动扩展堆大小（例如，初始化一个纯粹是为了分配内存空间的无用对象）：

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    void Start() {
        var tmp = new System.Object[1024];

        // make allocations in smaller blocks to avoid them to be treated in a special
        // way, which is designed for large blocks
        for (int i = 0; i < 1024; i++)
            tmp[i] = new byte[1024];

        // release reference
        tmp = null;
    }
}
```

```
//JS script example
function Start() {
    var tmp = new System.Object[1024];

    // make allocations in smaller blocks to avoid them to be treated in a special way
    // , which is designed for large blocks
    for (var i : int = 0; i < 1024; i++)
        tmp[i] = new byte[1024];

    // release reference
    tmp = null;
}
```

在游戏暂停之间，这个足够大的堆不应该被完全填满，因为会导致内存回收。当游戏暂停时，你可以明确地请求一次内存回收：

```
System.GC.Collect();
```

同样，你应该小心地使用这种策略，关注性能分析，而不仅仅是假设它有预期的效果。

可复用的对象池

在很多情况下，你可以简单地通过减少需要创建和销毁的对象数量来避免产生垃圾。游戏中某些类型的对象，例如射弹，它们可能在会反复出现，但是每次只会出现少数几个。在这种情况下，复用对象通常是可行的，而不是先销毁旧对象，然后创建新对象替换它们。

补充信息

内存管理是一个精细而复杂的课题，已经投入了大量学术上的努力。如果你有兴趣了解更多内容，memorymanagement.org 是一个很好的资源，上面列出了许多出版物和网络文章。关于对象池的更多信息，你可以在 [Wikipedia page](#) 和 [Sourcemarking.com](#) 上找到。

泛型函数

在脚本手册中，一些函数的名称后跟有一对尖括号，尖括号中是字符 T 或类型名称：

```
//C#
void FuncName<T>();
//JS
function FuncName.<T>(): T;
```

这些被称为是泛型函数。他们的意义在于指定参数类型和（或）返回类型。在 JavaScript 中，泛型函数可以避开动态类型的局限性：

```
// The type is correctly inferred since it is defined in the function call.
//In C#
var obj = GetComponent<Rigidbody>();
//In JS
var obj = GetComponent.<Rigidbody>();
```

在 C# 中，泛型函数可以节省大量的按键：

```
Rigidbody rb = go.GetComponent<Rigidbody>();

// ...as compared with:

Rigidbody rb = (Rigidbody) go.GetComponent(typeof(Rigidbody));
```

如果在脚本参考页看到了带有泛型声明的函数，那么就可以使用这种特殊的调用语法。

音频



Unity 的音频功能包括完整 3D 空间声音、实时混音和母带处理、混音层次结构、快照、预定义效果等等。

阅读本节以了解 Unity 中的音频，包括剪辑、声源、监听器、导入和声音设置。

相关教程：[音频](#)

相关的提示、技巧和故障排除，等参阅 [音频知识库](#) 部分。

音频概述

没有音频的游戏是不完整的，例如背景音乐或音响效果。Unity 的音频系统灵活而强大。它可以导入大多数标准音频文件格式，并且为播放 3D 空间中的声音提供了复杂的功能，以及可选的音响效果，例如回音和过滤。Unity 还可以记录来自用户机器上任意可用麦克风的音频，以便在游戏过程中使用，或者用于存储和传输。

基础理论

在现实生活中，声音由对象发出，并被听众听到。声音被感知的方式取决于许多因素。听众可以大致地判断声音来自的方向，并且可以通过音量和音质判断声音的距离。因为多普勒效应，快速移动的音源（例如下落的炸弹或路过的警车）将随着移动改变音高。此外，周围环境将影响声音反射的方式，所以，同样的声音，在洞穴内会有回音，在露天中则没有回音。



Audio Sources and Listener 音频源和监听器

为了模拟位置效果，Unity 要求为对象附加音频源 **Audio Source** 组件，并指定声音文件；并为另一个对象附加音频监听器 **Audio Listener** 组件，通常是主摄像机。这样，音频源发出的声音被音频监听器检测到。然后，Unity 可以模拟监听器到音源的距离和位置效果，并相应地向用户播放声音。也可以用音频源和监听器的相对速度来更加逼真地模拟多普勒效应。

Unity 不支持纯粹地通过场景几何计算回音，但是可以通过添加音频过滤器 **Audio Filters** 来模拟它们。例如，你可以为来自洞穴内部的声音应用回音过滤器 Echo。对于对象可以移入和移出的强回音场景，可以添加混响区域 **Reverb Zone**。例如，游戏可能会汽车穿过隧道。如果在隧道内放置一个混响区域，当汽车进入隧道时，引擎声音将开始产生回音，当汽车从隧道另一边出现时，回音将消失。

Unity 的音频混音器 **Audio Mixer** 支持混合各种音频源、添加效果和执行母带处理。

有关音响效果的选项和参数的更多信息，请参阅 [音频源](#)、[音频监听器](#)、[音频混音器](#)、[音频效果](#) 和 [混响区域](#) 的手册页。

使用音频资源

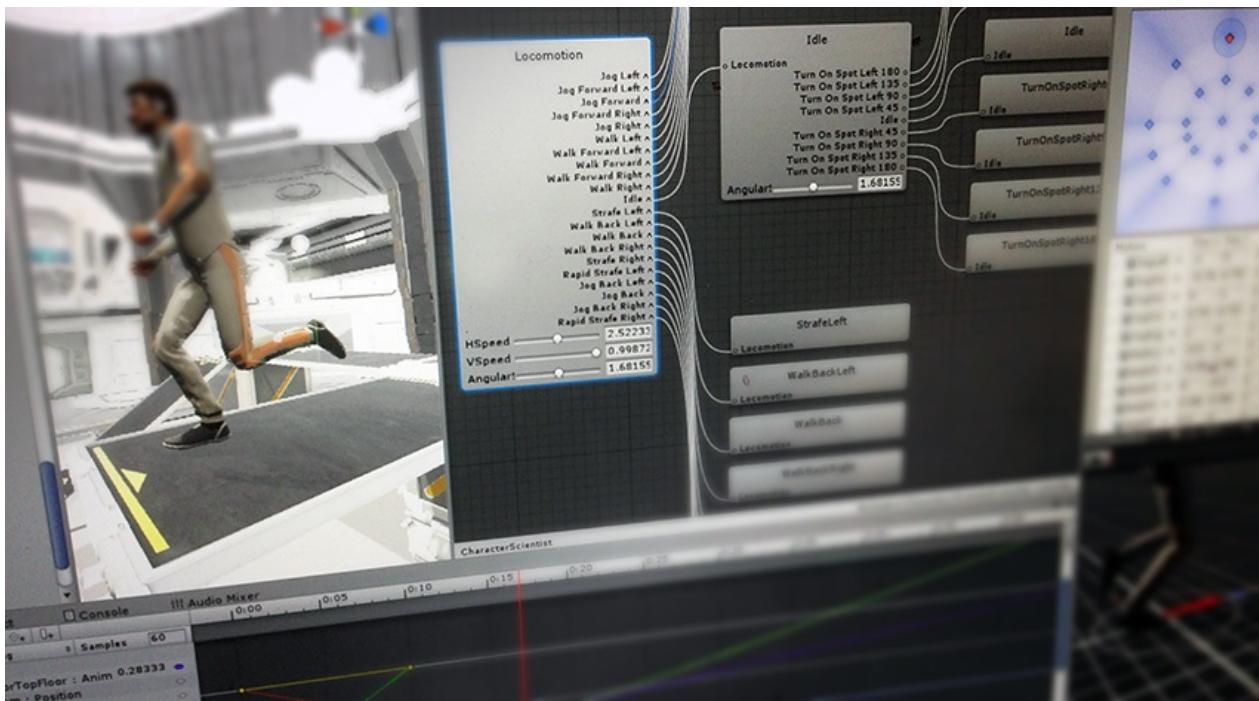
Unity 可以导入 **AIFF**、**WAV**、**MP3** 和 **Ogg** 格式的音频文件，导入方式与其他资源一样，只需简单地将文件拖入项目面板。导入一个音频文件会创建一个音频剪辑 **Audio Clip**，可以将其拖动到一个音频源 **Audio Source** 组件，或是从脚本中使用它。音频剪辑的参考页提供了导入音频文件时可用选项的更多详细信息。

对于音乐，Unity 还支持音轨模块，它采用短音频样本作为乐器，然后排列乐器播放音乐。音轨模块可以从 **.xm**、**.mod**、**.it** 和 **.s3m** 文件导入，使用方式与普通的音频剪辑大致相同。

音频录制

Unity 可以通过脚本访问计算机的麦克风，通过直接录制创建音频剪辑。麦克风 **Microphone** 类提供了直观的 API 来查找可用的麦克风，包括查询性能、开始和结束录制会话。[麦克风 Microphone](#) 的脚本引用页提供了音频录制的更多信息和代码示例。

动画



Unity 中的动画

Unity 的动画特性包含重定向动画、运行时动画权重控制、事件回掉、复杂的状态机分层和转换、混合面部动画着色器，等等。

这一章将介绍如何导入和使用动画，如何让游戏对象、外表动起来，以及 Unity 中的其他控制参数。

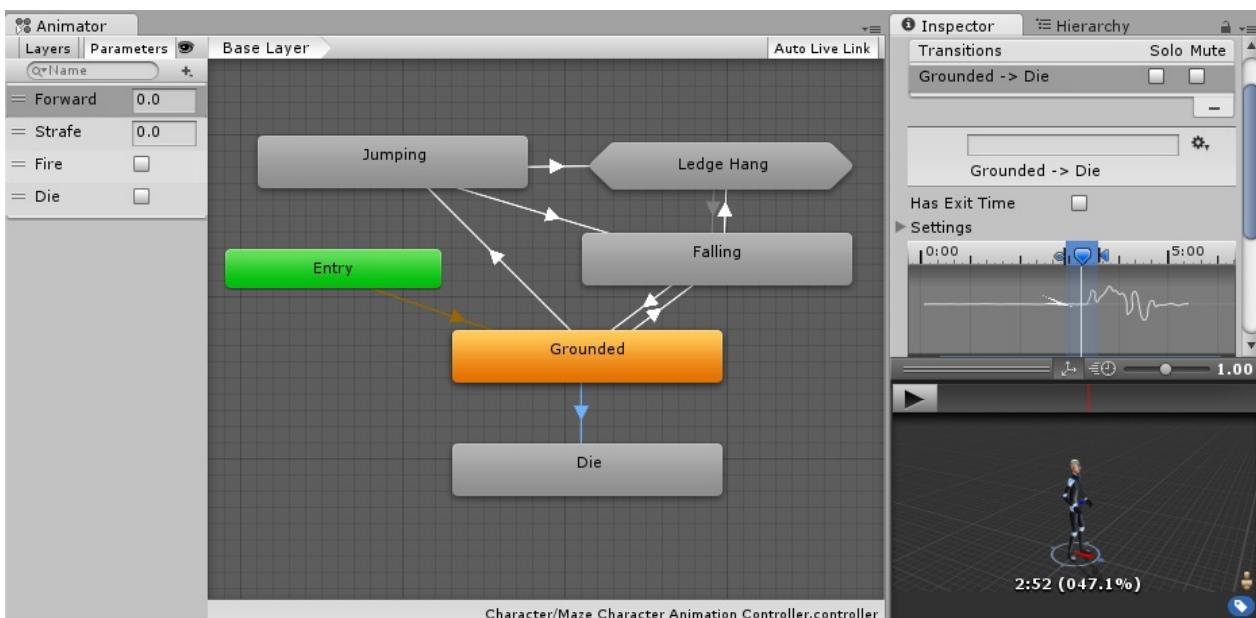
相关教程：[Animation](#)

一些技巧、常见问题，请查看 [动画基础知识](#)。

动画系统概览

Unity 拥有丰富和复杂的动画系统（有时称为 Mecanim），提供了以下功能：

- 为 Unity 中的对象、角色、道具等元素提供易用的动画工作流程和设置。
- 支持导入动画片断和用 Unity 创建的动画。
- 人形动画重定位 — 这一能力允许把一个角色模型的动画应用到另一个角色。
- 为调整动画剪辑提供简化了的工作流程。
- 为动画剪辑、转换、交互提供方便的预览。使得动画可以更独立于程序和原型运行，在嵌入代码之前，就可以预览动画。
- 用可视化的编程工具管理动画之间的负责交互。
- 可以为不同的身体部位添加不同的动画逻辑。
- 层和遮罩功能。



动画视图中的动画状态机

动画工作流程

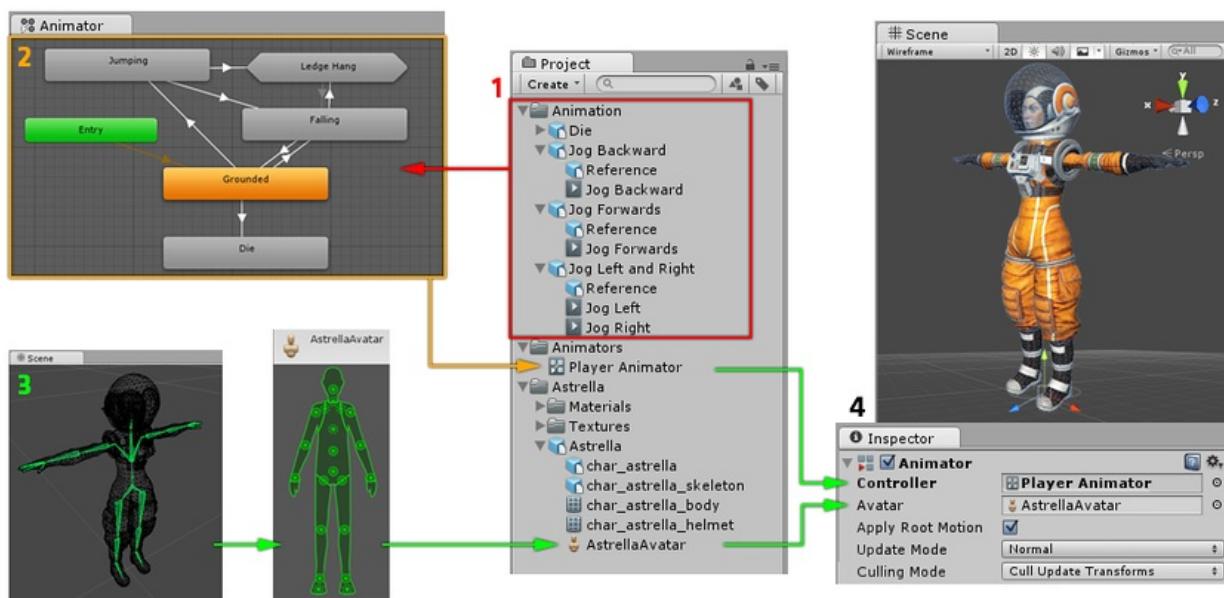
Unity 动画系统基于动画剪辑概念，包含了指定对象随着时间推移应该如何改变位置、旋转或其他属性的信息。每个剪辑可以认为是一段线性记录。外部导入的动画剪辑由第三方艺术或动画工具创建，例如 Max 和 Maya，或者来源于动作捕捉设置，或者其他来源。

动画剪辑 Animation Clip 被组织成类似流程图的系统，称为动画控制器 Animator Controller。动画控制器的行为像一台状态机，记录了当前应该播放的动画剪辑，以及什么时候应该切换或混合动画剪辑。

一个非常简单的动画控制器可能只包含一个或两个动画剪辑，例如控制一段旋转或弹跳动画，或者是在合适的时间打开或关闭一扇门。更高级的动画控制器可能包含许多人形动画，涵盖了主要角色的动作，并且当玩家在场景内移动时，可能同时混合多个动画剪辑来提供流畅的动作。

Unity 动画系统还为人形角色提供了一些特殊功能，使你可以重构任意来源的人形动画（例如，动作捕捉，资源商店，或其他第三方动画库）到你的角色模型上，并且可以调整肌肉定义。这些特殊功能由 Unity 的 Avatar 系统提供，可以把人形角色映射到一种通用的内部格式。

每个元素 — 动画剪辑、动画控制器、Avatar — 通过动画组件 Animator 组件合并到一个游戏对象上。这个组件引用了动画控制器和（如果需要）Avatar 系统，动画控制器又包含了对动画剪辑的引用。



动画系统的各部分是如何连接在一起的

上面的图形展示了：

1. 动画剪辑或者从外部资源导入，或者在 Unity 中创建。在这个例子中，是从动作捕捉导入的人形动画。
2. 动画剪辑被放入动画控制器。上图显示了动画视图中的动画控制器。这些状态（表示动画或嵌套的子状态机）以节点的形式展现，用线条连接。动画控制器作为一个资源显示在项目视图中。
3. 被控制的角色模型（在这里是宇航员 Astrella）拥有特殊的骨骼配置，被映射到 Unity 的通用 Avatar 格式。映射以 Avatar 资源的形式存储在角色模型下，做为被导入角色模型的一部分，显示在项目视图中。
4. 当赋予角色模型动画时，一个动画组件被添加。在上面的检视视图中，你可以看到动画组件被指定了动画控制器和 Avatar。动画组件同时使用两者来使模型动起来。只有人形角色才需要引用 Avatar。对于其他类型的动画，只需要一个动画控制器。

Unity 动画系统（称为 Mecanim）含有大量的概念和术语。在任何时候，如何你需要查找某个东西的含义，请访问 [动画术语](#)。

传统动画系统

尽管在大多数情况下推荐使用 Mecanim，但是 Unity 还保留了 Unity 4 之前的传统动画系统。如果使用 Unity 4 之前创建的老内容时，你可能需要用到它。有关传统动画系统的更多信息请查看 [这章](#)。

Unity 打算把传统动画系统的工作流程合并到 Mecanim，来逐步淘汰传统动画系统。

动画剪辑

动画剪辑是 Unity 动画系统的核心元素。Unity 不仅支持从外部源导入动画，而且支持在编辑器的动画视图中创建和编辑动画(剪辑)。

从外部源导入动画

从外部源导入的动画剪辑可能包括：

- 动作捕捉工作室捕捉的人形动画
- 设计师通过外部 3D 程序（例如 3D Max 或 Maya）创建的动画
- 第三方的动画集合库（例如来自 Unity Asset store）
- 导入的单个时间线等分切割为多个动画剪辑

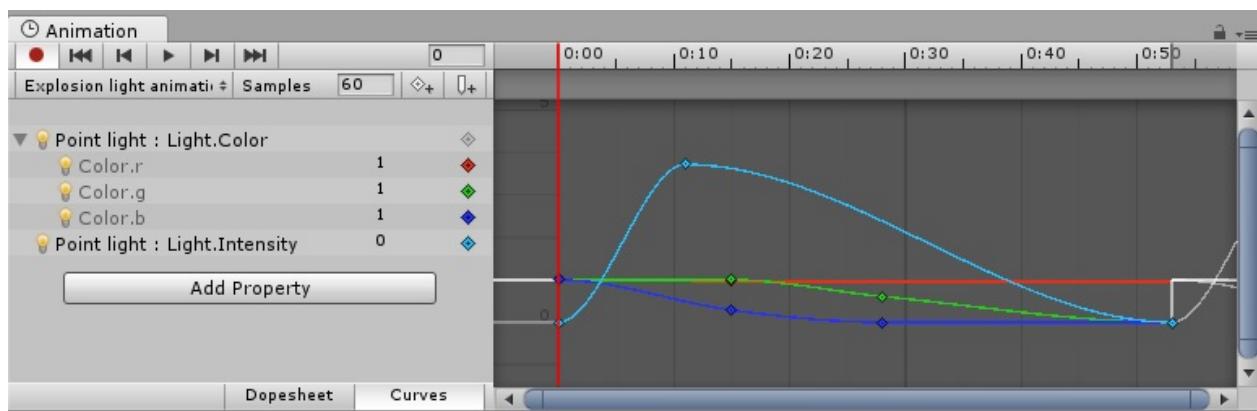


在 Unity 检视视图，一段导入的动画剪辑示例。

在 **Unity** 中创建和编辑动画

Unity 的动画视图允许创建和编辑动画剪辑。动画剪辑支持以下属性（即为属性附加动画）：

- 游戏对象的位置、旋转和尺寸。
- 组件属性，例如材质颜色、光照强度和声音音量。
- 脚本属性，包括浮点型、整型、矢量和布尔变量。
- 调用脚本方法的时机。



Unity 动画视图中，一个为组件属性添加动画的示例。在这个示例中，是一个点光源的光照强度和光照范围。

动画视图指南

在 Unity 中的动画视图中，你可以直接创建和修改动画剪辑。动画视图的功能强大、操作简单，被设计为外部 3D 动画软件的替代品。除了运动动画外，编辑器还可以为材质和组件添加动画，以及基于动画事件修改动画剪辑，动画事件是一些在特定时间点被调用的函数。

更多相关信息请阅读 [动画导入](#) 和 [动画脚本](#)。

本章接下来的内容将详细介绍动画视图的各个方面。

当前为 5.4 版本，稍后将升级到 5.5。在 5.5 中，本节内容被拆分成了多个小节。

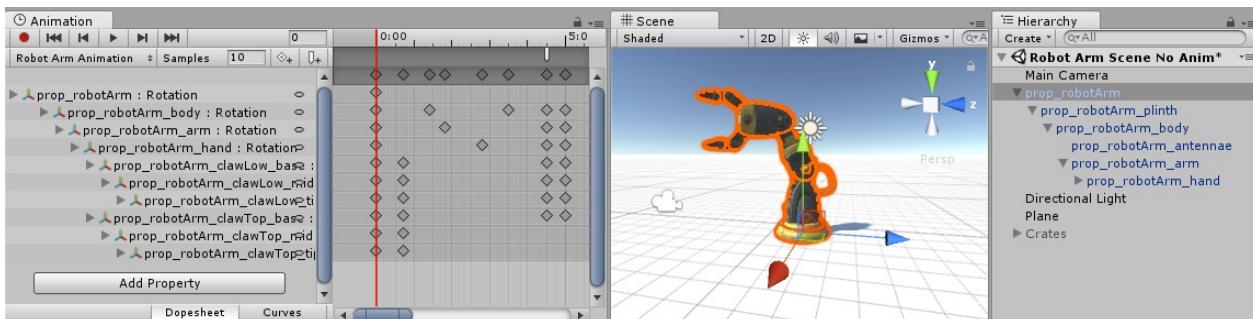
使用动画视图

在 Unity 中，动画视图用于预览和编辑游戏对象的动画剪辑。动画视图可以通过菜单 Window -> Animation 打开。

查看游戏对象上的动画

动画视图和层级视图、场景视图以及检视视图紧密耦合。类似于检视视图，动画视图将显示当前选中对象的动画的时间轴和关键帧。你也可以在层级视图或场景视图中选择一个游戏对象查看。（如果你在项目视图中选择了一个预制体 Prefab，同样可以查看动画时间线，但是只有先拖动预制体 Prefab 到场景视图中，才能编辑动画。）

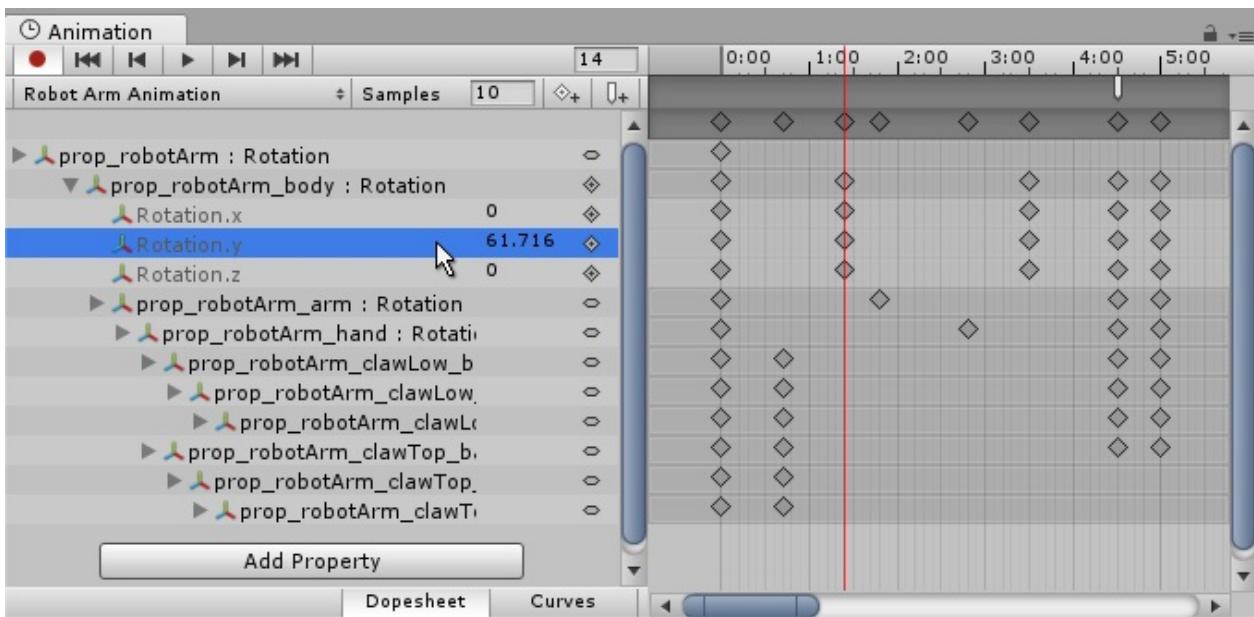
动画属性列表



动画视图（左侧）显示了当前选中的游戏对象所使用的动画，如果它的子对象也使用了动画，那么也会显示在这里。场景视图和层级视图显示在右侧，可以看到当前选中的游戏对象。

动画视图的左侧是动画属性列表。在一个新创建的动画剪辑中，这个列表为空，没有记录任何动画的。当你开始让各种属性动起来时，参与动画的属性将显示在这里。如果动画控制了多个子对象，这个列表还会以层级的方式显示子对象的动画属性。在上面的例子中，机械手臂的各个部分都由同一个动画剪辑驱动，每个被它控制的游戏对象，按照它们在附加了动画组件的根元素中的层级关系显示。

每个属性可以折叠和展开，从而显示每个关键帧记录的精确值。如果在关键帧之间移动磁头（红线），属性右侧的值域将显示补间值。值域可以直接编辑。如果磁头位于某个关键帧上，修改值域将直接修改该关键帧的值。如果磁头在关键帧之间（此时值域显示的是补间值），修改值域将在磁头位置创建一个新的关键帧，值域的值即为该关键帧的值。



在动画视图中展开属性列表，允许直接键入关键帧的值。在这张图中，因为磁头（红线）位于关键帧之间，所以显示的是补间值。如果在该位置输入一个新值，将创建一个新关键帧。

动画时间轴

动画视图的右侧是每个动画剪辑的时间轴。每个动画属性的关键帧显示在时间轴中。时间轴有两种模式：关键帧清单模式和曲线模式。通过点击动画属性列表底部相应的按钮，可以在两种模式之间切换：

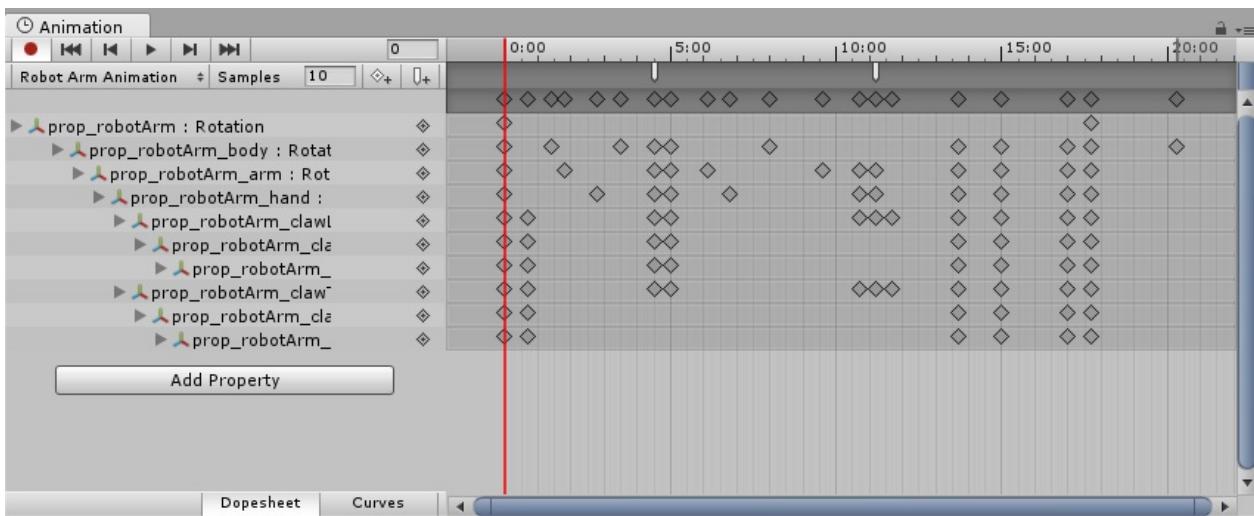


关键帧清单模式/曲线模式按钮

这两种模式可以彼此替换，都可以现实动画的时间轴和关键帧数据，但是各有优势。

关键帧清单模式

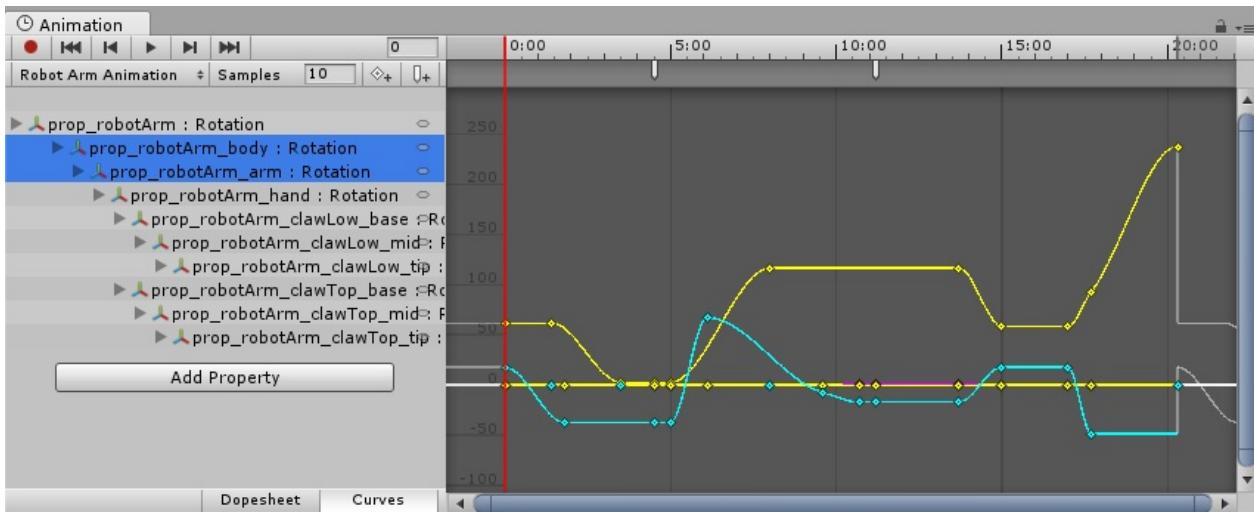
关键帧清单模式提供更紧凑的视图，允许你在水平轨道上独立地查看每个属性的关键帧序列。可以让你简单地描述多个属性或对象的关键帧时间设置。



这里的动画视图是关键帧清单模式，显示了动画剪辑中所有动画属性的关键帧位置。

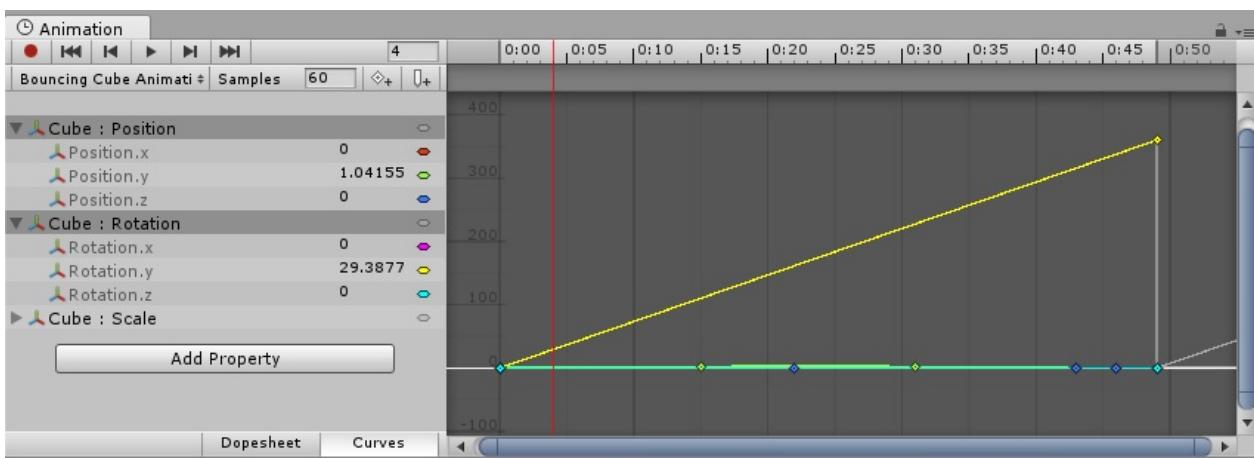
曲线模式

曲线模式以曲线图的方式显示每个动画属性的值是如何随时间变化的，曲线图可以调整。所有选中的属性叠加显示在同一张图中。这种模式为查看和编辑属性值、补间值，提供了强大的控制。



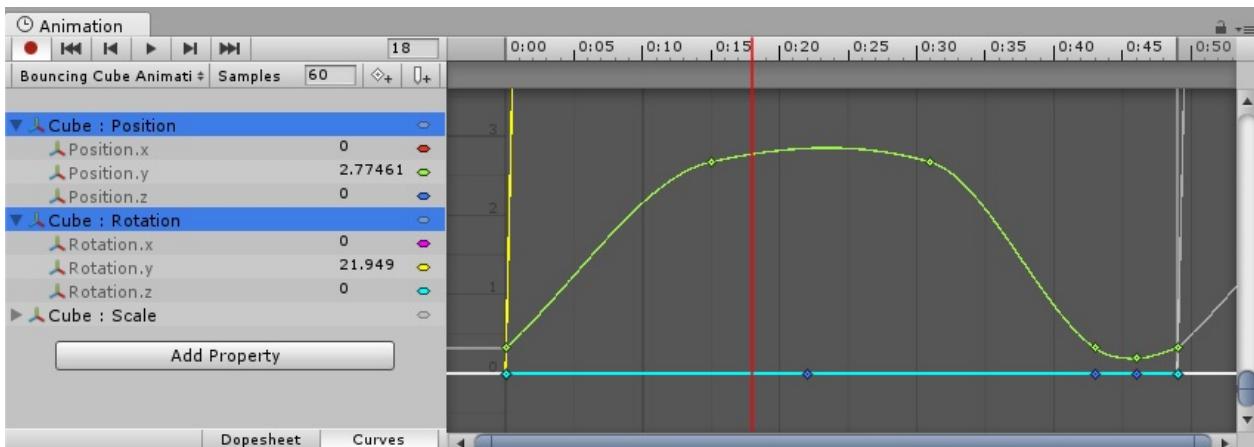
这里的动画视图显示了动画剪辑中 4 个选中对象的旋转曲线。

当使用曲线模式查看动画剪辑时，非常重要的一点是，每个属性值的范围有时会相差很大。举个例子，假如有一个同时旋转和弹跳的立方体。弹跳的 Y 坐标值可能介于 0 和 2 之间（也就是说，立方体在动画过程中弹跳 2 个单位），但是，旋转值介于 0 和 360 之间。同时查看这两条曲线时，位置动画曲线将非常难以辨别，因为整个视图被放大以显示 0 到 360。



同时选择立方体的位置动画和旋转动画，因为视图被放大以显示旋转曲线的 0 到 360，位置曲线的 Y 轴值不可辨识。

你可以点击列表中的单个动画属性，曲线视图会自动缩放，以匹配值范围；或者，你也可以通过拖动右侧和下方的滚动条，来手动缩放曲线视图。



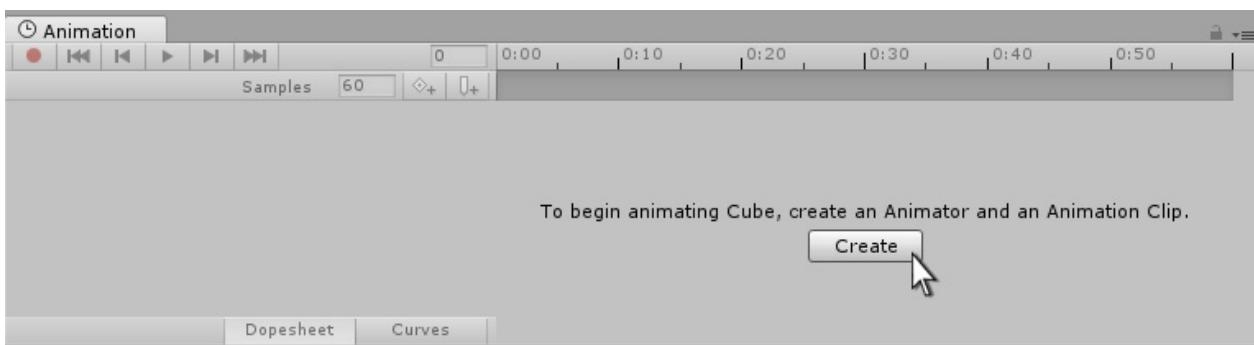
动画视图被放大，以显示位置动画的 Y 轴值。黄色的旋转动画曲线仍然可见，但只显示了一部分。

创建动画剪辑

在 Unity 中，为了让游戏对象动起来，需要附加一个动画组件。这个动画组件必须引用一个动画控制器，动画控制器再引用一个或多个动画剪辑。

在 Unity 中，当开始使用动画视图让游戏对象动起来时，这些元素将被自动创建和绑定。

点击动画视图左上角的下拉框，选择 **Create New Clip**，可以为选中的游戏对象创建一个新的动画剪辑。然后，会提示你把动画剪辑保存到 Assets 文件夹的某个位置。如果这个游戏对象已经附加了动画组件，并且指定了动画控制器，新的动画剪辑将作为一个状态，被添加到现有的动画控制器中。



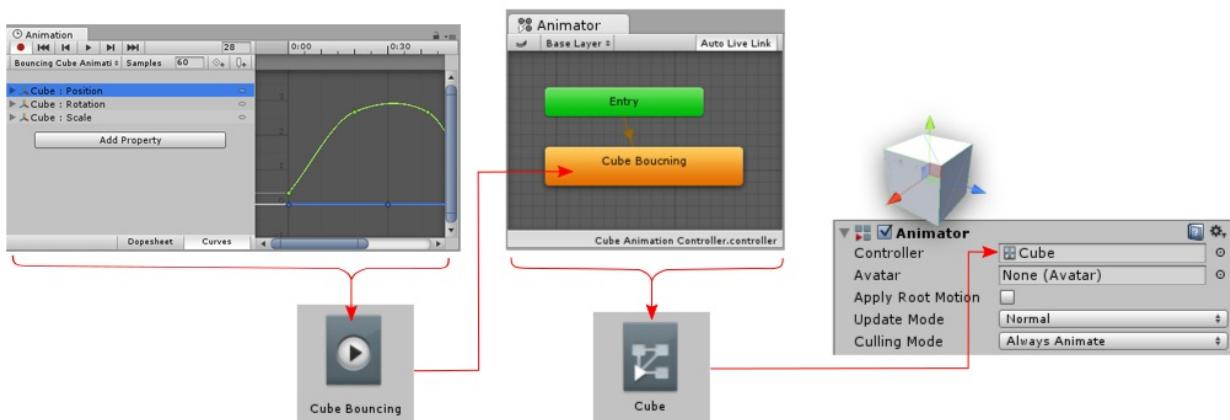
创建一个新的动画剪辑

如果这个游戏对象尚未绑定动画组件，那么将自动进行下面的操作：

- 一个新的动画控制器资源文件将被创建
- 新创建的动画剪辑将作为默认状态，被添加到该动画控制器
- 一个动画组件将被添加到该游戏对象
- 该动画组件将引用该动画控制器

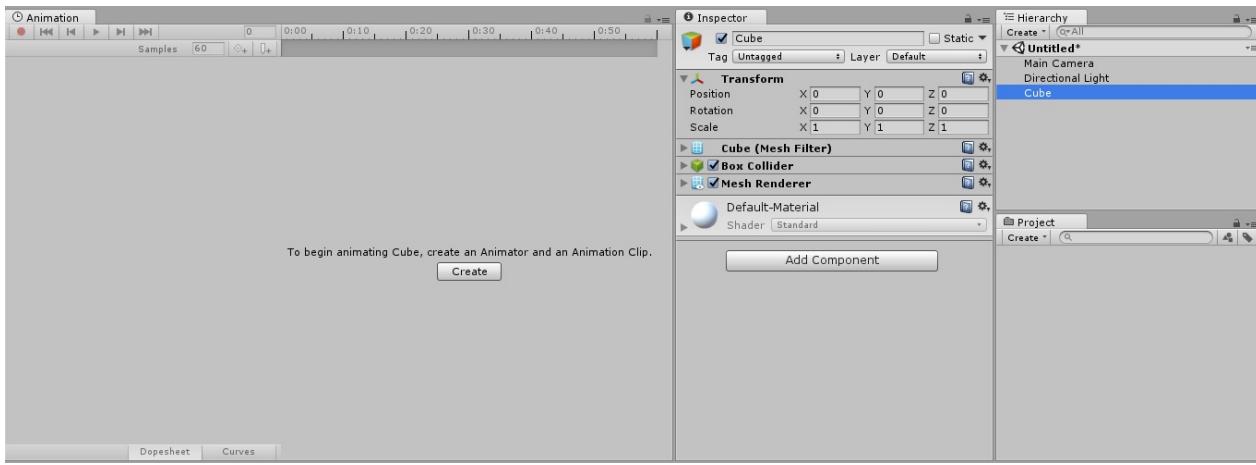
这样做的好处是，你只需要点击记录按钮或选择 **Create New Clip**，就可以开始构建动画，动画系统所需的所有元素已经被自动配置好。

下面的图演示了，在动画视图中，创建新的动画剪辑时，这些元素是如何分配的：



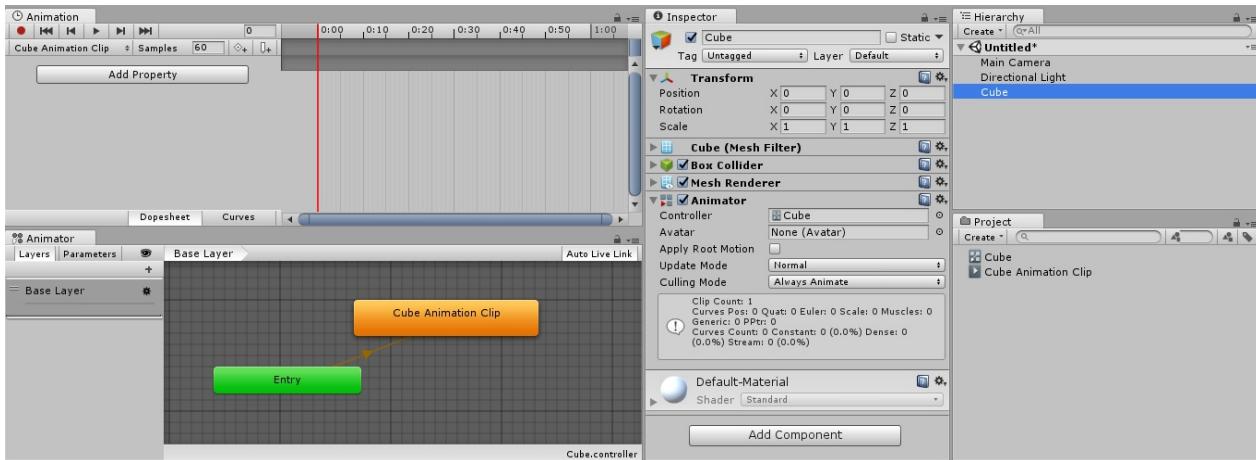
一个新的动画剪辑被创建，并保存为一个资源。该动画剪辑作为默认状态，被自动添加到一个新的动画控制器中，该动画控制器也被保存为一个资源。该动画控制器被分配给游戏对象上的动画组件。

在下面的图中，你可以看到一个没有动画的游戏对象。它只是一个简单的立方体，没有附加动画组件。动画视图、检视视图、层级视图和项目视图挨个排列如下。



之前：一个无动画的游戏对象（立方体）被选中。它没有动画组件，动画控制器也不存在。

通过按下（左侧）动画视图中的记录按钮，或选择动画视图中下拉框的 Create New Clip，就创建了一个新的动画剪辑。Unity 将会询问你该动画剪辑的名字和存储位置。Unity 还会创建一个与选中的游戏对象同名的动画控制器资源文件，并且添加一个动画组件到游戏对象上，并且合适地连接这些资源。



之后：创建一个新的动画剪辑后，你可以在项目视图中看到新的资源文件，在检视视图（右侧）中看到附加的动画组件。你还可以在动画控制器视图中看到该动画剪辑被当作默认状态。

让游戏对象动起来

一旦你保存了新动画剪辑，就可以开始为动画剪辑添加关键帧了。开始为选中的游戏对象编辑动画剪辑之前，需要先点击动画记录按钮，进入动画记录模式，此时，对该游戏对象的修改会被记录到动画剪辑。



记录按钮

任何时候，你都可以通过再次点击动画模式按钮，从而退出动画记录模式。这一操作将把该游戏对象的状态恢复到进入动画记录模式之前。

你对该游戏对象所做的修改将被记录为关键帧，红线所指示的时间即为关键帧的时间。

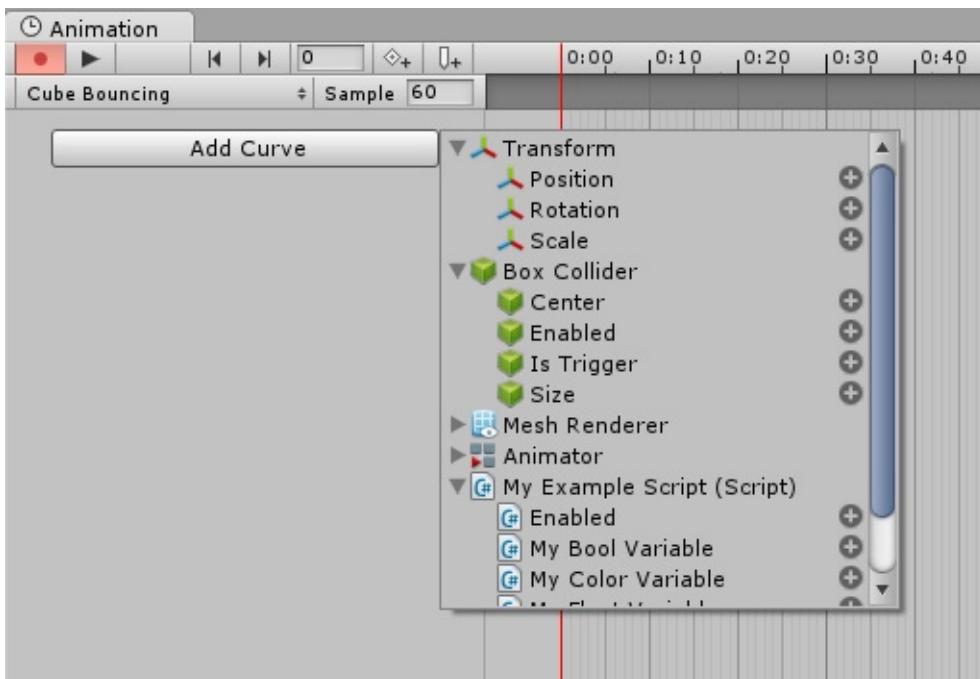
在动画记录模式中，通过修改游戏对象，你可以为任何属性添加动画。无论是移动、旋转或缩放游戏对象，对应的属性都将被添加到动画剪辑的相应关键帧中。在游戏对象的检视视图中修改值，也会添加关键帧。支持检视视图中的任意属性，例如，数值型、复选框、颜色等大部分属性。

游戏对象的动画属性被罗列在动画视图左侧的属性列表中。不参与动画的属性不会显示在视图中。新增加的动画属性，包括子对象上的属性，同样会被添加到属性列表中。

Transform 组件的属性有些特别，属性 **.x**、**.y** 和 **.z** 是连接在一起的，所以一次会添加 3 条曲线。

通过点击 **Add Curves** 按钮，你也可以浏览当前游戏对象（和它的子对象）上所有支持动画的属性。

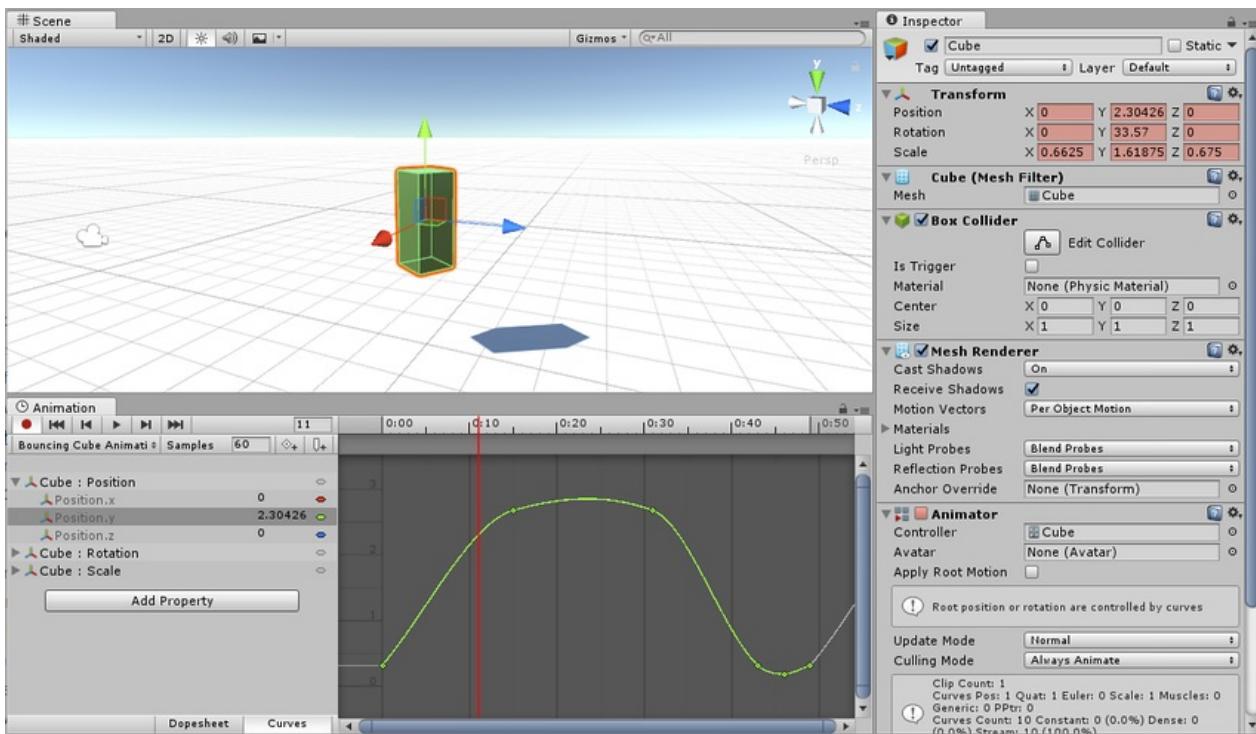
通过 **Add Curves** 按钮，你可以选择任意属性，为它添加动画。



添加曲线

在动画录制模式中，红色的垂直线表示当前预览的是动画剪辑的那一桢。检视视图和场景视图将显示游戏对象在该桢的状态。动画属性在该桢的值显示在属性名称的右侧。

在动画录制模式中，红色的垂直线表示当前预览的桢。



当前桢

时间轴

在动画剪辑中，你可以点击时间轴上的任意一点，来预览或修改该桢。时间轴上的数字是秒数和帧数，所以 1:30 的意思是 1 秒加 30 帧。



时间轴

桢导航

你可以使用下面的键盘快捷键在桢之间导航：

- 按下逗号 (,) 切换到前一桢。
- 按下句号 (.) 切换到下一桢。
- 同时按下 Alt 和逗号 (,) 切换到前一个关键桢。
- 同时按下 Alt 和句号 (.) 切换到下一个关键桢。



桢导航

动画模式

在动画模式下，你可以在场景视图中移动、旋转或缩放游戏对象。如果位置、旋转或缩放属性在动画剪辑中不存在的话，这些操作将自动创建动画曲线；并且，自动在当前预览的桢上创建关键帧，以存储改变后的 Transform 值。

你也可以在检视视图中修改游戏对象的任意动画属性。如果需要的话，这些操作也会自动创建动画曲线，并且在当前预览的动画曲线上创建关键帧，以存储改变后的 Transform 值。

创建关键帧

你可以使用 **添加关键桢按钮** 手动创建一个 关键帧。这个操作将会为 动画视图 中当前选中的所有属性创建一个关键帧。也就是说，你可以有选择性地只为特定属性添加关键帧。



添加关键桢按钮

播放

任意时候，在 动画视图 中点击 播放按钮，都可以播放&重放 动画剪辑。



The Play button 播放按钮

锁定动画视图

你可以锁定动画视图，这样，当你在层级视图或场景视图中切换选中的游戏对象时，动画视图不会自动切换。如果只关注某个特定的游戏对象，那么锁定动画视图会非常有用，此时你可以选择或编辑场景视图中的其他对象。



锁定按钮

更多资料

想要学习控制 曲线模式 的更多内容，请查阅 [使用动画曲线](#) 一节。

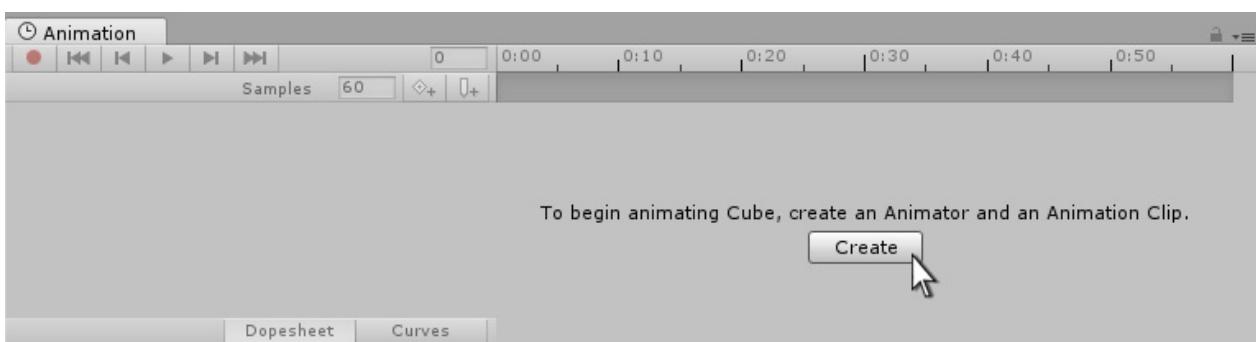
创建动画剪辑

在 Unity 中，为了让游戏对象动起来，需要附加一个 动画组件。这个动画组件必须引用一个 动画控制器，动画控制器再引用一个或多个 动画剪辑。

在 Unity 中，当开始使用动画视图让游戏对象动起来时，这些元素将被自动创建和绑定。

在为选中的游戏对象创建一个新动画剪辑前，需要先确保 动画视图 是打开的。

如果游戏对象尚未绑定任何动画剪辑，那么可以在动画视图的时间轴区域中看到『Create』按钮。点击该按钮，将会提示你把新创建的空动画剪辑保存到 **Assets** 文件的某个位置。



创建一个新动画剪辑

一旦保存了新创建的空动画剪辑，那么将自动执行下面的操作：

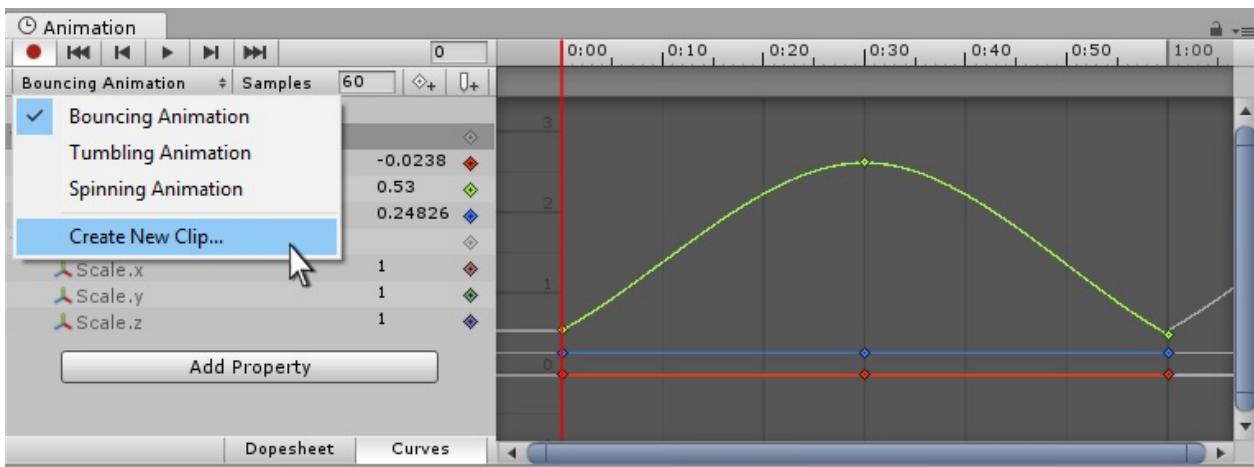
- 一个新的动画控制器资源文件将被创建
- 新创建的动画剪辑将作为默认状态，被添加到该动画控制器中
- 一个动画组件将被添加到该游戏对象
- 该动画组件将引用该动画控制器

上述操作（自动化序列）完成后，动画系统所需的所有元素已经就绪，可以开始让游戏对象动起来了。

再创建一个动画剪辑

如果这个游戏对象已经分配了一个或多个动画剪辑，则『Create』按钮将不可见。相反，其中一个剪辑将出现在动画视图中。你可以使用动画视图左上角的下拉菜单，就在播放控件下面，切换视图中显示的动画剪辑。

如果想要在一个已经含有动画的游戏对象上创建一个新的动画剪辑，则必须在下拉菜单中选择『Create New Clip』。你将被再次提示保存新建的空动画剪辑，然后才能使用它。



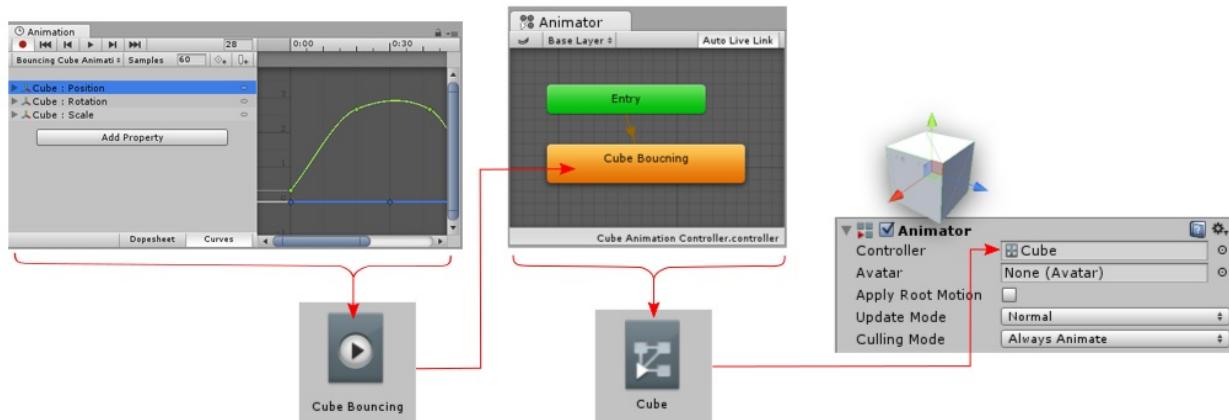
为已经含有动画剪辑的游戏对象添加一个新的动画剪辑。

如何整合在一起

虽然上述步骤自动设置相关的组建和引用，但是理解它们是如何整合在一起的也非常有用。

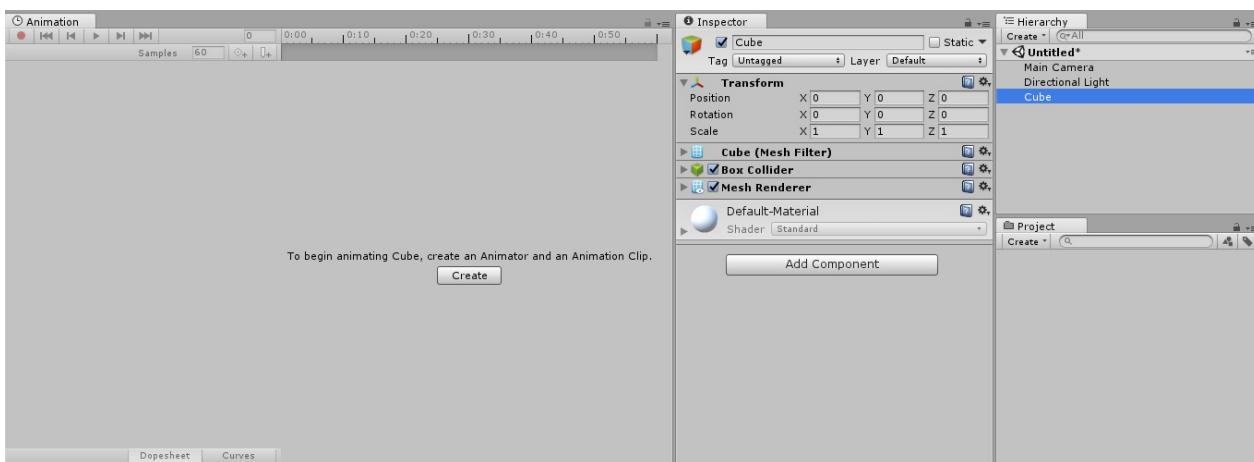
- 一个游戏对象必须有一个 动画组件
- 该游戏组件必须被分配一个 动画控制器 资源
- 该动画控制器资源必须被分配一个或多个动画剪辑

下图演示了，在动画视图中新建一个动画剪辑时，这些元素是如何分配的：



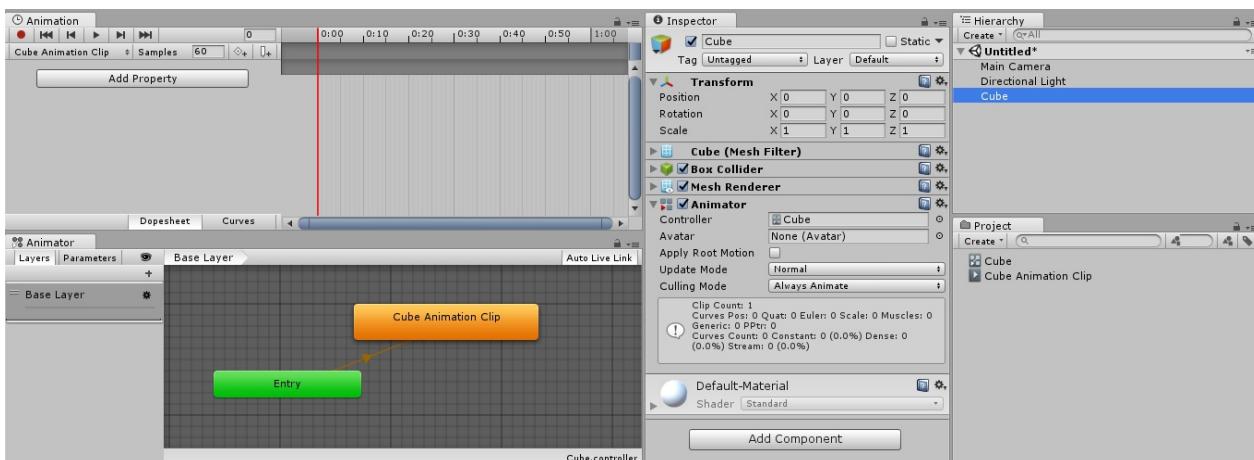
新建一个动画剪辑，并保存为一个资源。该动画剪辑作为默认状态，被自动添加到一个新的动画控制器中，该动画控制器也被保存一个资源。该动画控制器被分配给游戏对象上的动画组建。

在下面的图中，你可以看到一个没有动画的游戏对象。它只是一个简单的立方体，没有附加动画组件。动画视图、检视视图、层级视图和项目视图挨个排列如下。



之前：一个无动画的游戏对象（立方体）被选中。它没有动画组件，动画控制器也不存在。

通过按下动画视图中的创建按钮，就创建了一个新的动画剪辑。Unity 将会询问你该动画剪辑的名字和存储位置。Unity 还会创建一个与选中的游戏对象同名的动画控制器资源文件，并且添加一个动画组件到游戏对象上，并且恰当地连接这些资源。



之后：创建一个新的动画剪辑后，你可以在项目视图中看到新的资源文件，在检视视图（右侧）中看到附加的动画组件。你还可以在动画控制器视图中看到该动画剪辑被当作了默认状态。

在上面的视图中，你可以看到：

- 动画视图（左上）现在显示一条带有红色播放磁头的时间轴，准备开始记录新关键帧。该剪辑的名字出现在剪辑菜单中，就在播放空间下方
- 检视视图（中间）显示立方体游戏对象现在添加了一个动画组件，并且该组件的『Controller』域被分配了一个名为『Cube』的动画控制器资源。
- 项目视图（右下）现实两个新资源已经被创建——一个名为『Cube』的动画控制器资源和一个名称『Cube Animation Clip』的动画剪辑资源。
- 动画控制器视图（左下）显示了动画控制器的内容——你可以看到立方体动画剪辑已经被添加到了控制器中，并且是『默认状态』，用橙色颜色标出。后续添加到该控制器中的剪辑将是灰色，表示它们不是默认状态。

让游戏对象动起来

一旦你保存了新动画剪辑，就可以开始为动画剪辑添加关键帧了。开始为选中的游戏对象编辑动画剪辑之前，需要先点击动画记录按钮，进入动画记录模式，此时，对该游戏对象的修改会被记录到动画剪辑。



记录按钮

任何时候，你都可以通过再次点击 动画模式按钮，从而退出 动画记录模式。这一操作将把该游戏对象的状态恢复到进入动画记录模式之前。

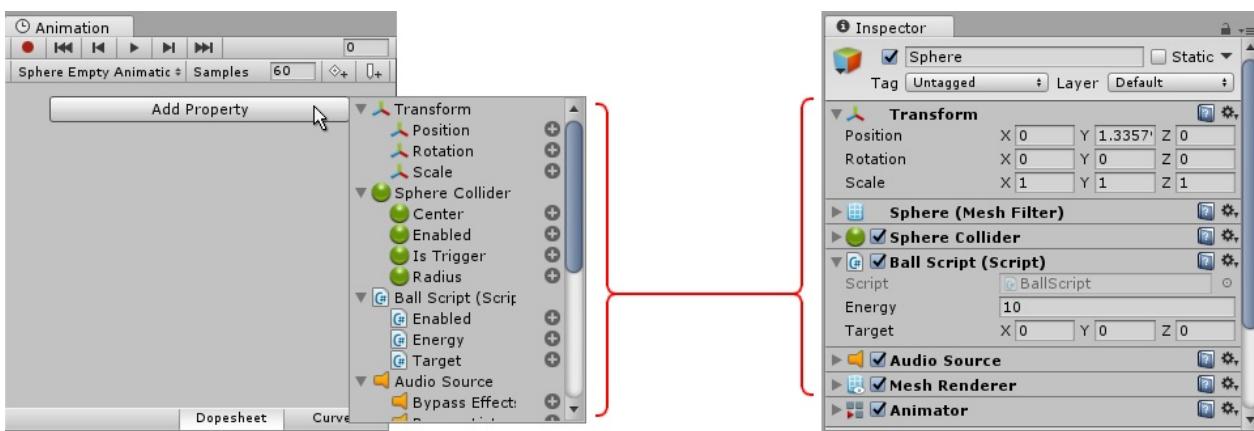
你对该游戏对象所做的修改将被记录为关键帧，红线所指示的时间即为关键帧的时间。

在动画记录模式中，通过修改游戏对象，你可以为任何属性添加动画。无论是移动、旋转或缩放游戏对象，对应的属性都将被添加到动画剪辑的相应关键帧中。在游戏对象的检视视图中修改值，也会添加关键帧。支持检视视图中的任意属性，例如，数值型、复选框、颜色等大部分属性。

游戏对象的动画属性被罗列在动画视图左侧的属性列表中。不参与动画的属性不会显示在视图中。新增加的动画属性，包括子对象上的属性，同样会被添加到属性列表中。

Transform 组件的属性有些特别，属性 **.x**、**.y** 和 **.z** 是连接在一起的，所以一次会添加 3 条曲线。

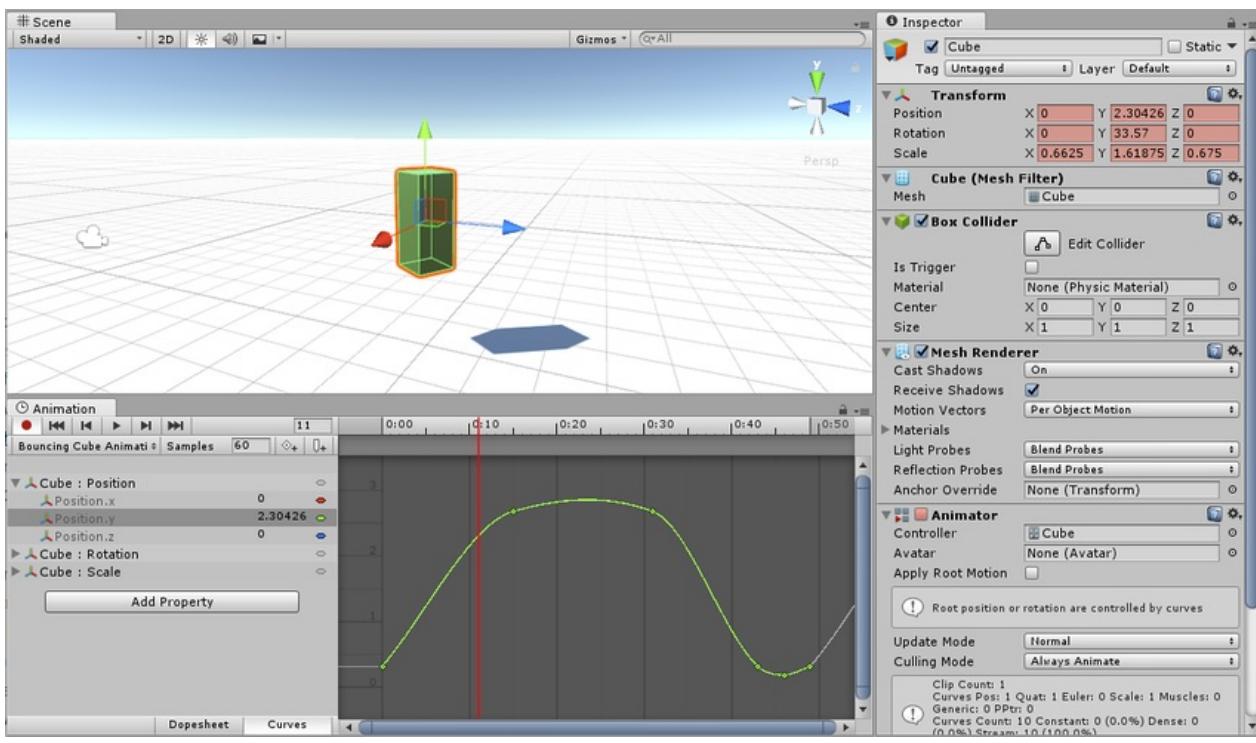
你也可以点击 添加属性按钮，为当前游戏对象（和它的子对象）添加动画属性。点击这个按钮，将弹出一个浮层，现实该游戏对象上所有支持动画的属性。这些属性与检视视图中列出的属性一一对应。



当点击 **Add Property** 按钮，游戏对象上可参与动画的属性被显示出来。

在动画录制模式中，红色的垂直线表示当前预览的是动画剪辑的哪一帧。检视视图 和 场景视图 将显示游戏对象在该帧的状态。动画属性在该帧的值显示在属性名称的右侧。

在动画录制模式中，红色的垂直线表示当前预览的帧。



当前帧

时间轴

在动画剪辑中，你可以点击时间轴上的任意一点，来预览或修改该帧。时间轴上的数字是秒数和帧数，所以 1:30 的意思是 1 秒加 30 帧。

0:00 | 0:30 | 1:00 | 1:30

时间轴

动画模式

在动画模式下，你可以在场景视图中移动、旋转或缩放游戏对象。如果位置、旋转或缩放属性在动画剪辑中不存在的话，这些操作将自动创建动画曲线；并且，自动在当前预览的帧上创建关键帧，以存储改变后的 **Transform** 值。

你也可以在检视视图中修改游戏对象的任意动画属性。如果需要的话，这些操作也会自动创建动画曲线，并且在当前预览的动画曲线上创建关键帧，以存储改变后的值。

创建关键帧

你可以使用添加关键帧按钮手动创建一个关键帧。这个操作将会为动画视图中当前选中的所有属性创建一个关键帧。也就是说，你可以有选择性地只为特定属性添加关键帧。



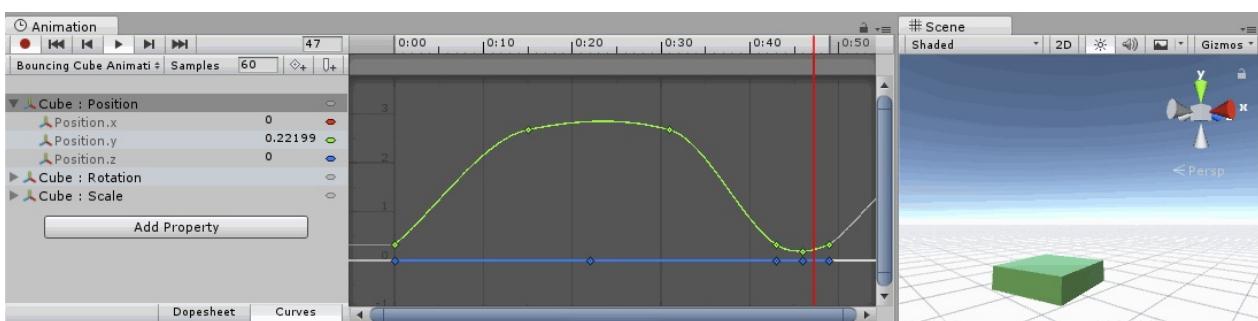
添加关键帧按钮

使用动画曲线

属性列表

在一段 动画剪辑 中，任意动画属性都可以拥有 动画曲线，也就是说，动画剪辑控制着属性如何随时间变化。在 动画视图（左侧）的属性列表区域，罗列出了当前所有参与动画的属性。当动画视图处于关键帧清单模式，每个属性的值是一个条直线轨道，但是在曲线模式下，正在改变的属性值被可视化为曲线图形。无论你用哪种模式查看，曲线一直存在 — 关键帧清单模式只是简单的关键帧数据预览。

在 曲线模式 下，动画曲线 带有颜色标记，每种颜色代表一个当前选中属性的值。关于如何为动画属性添加曲线，请阅读 [Using the Animation View](#) 一章。

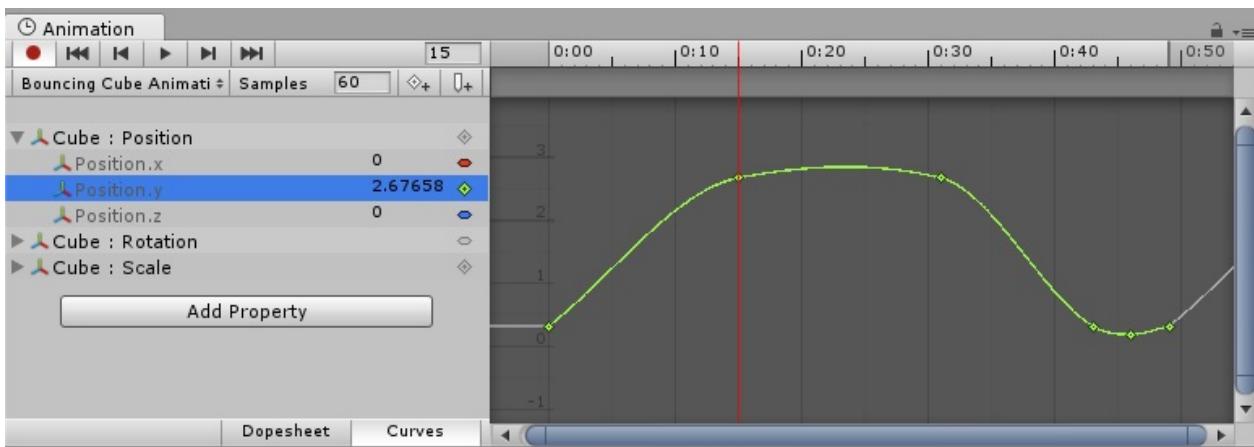


动画曲线带有可视化的颜色标记。在这个例子中，绿色线条表示弹跳立方体的 Y 轴动画。

理解曲线、关键点、关键帧

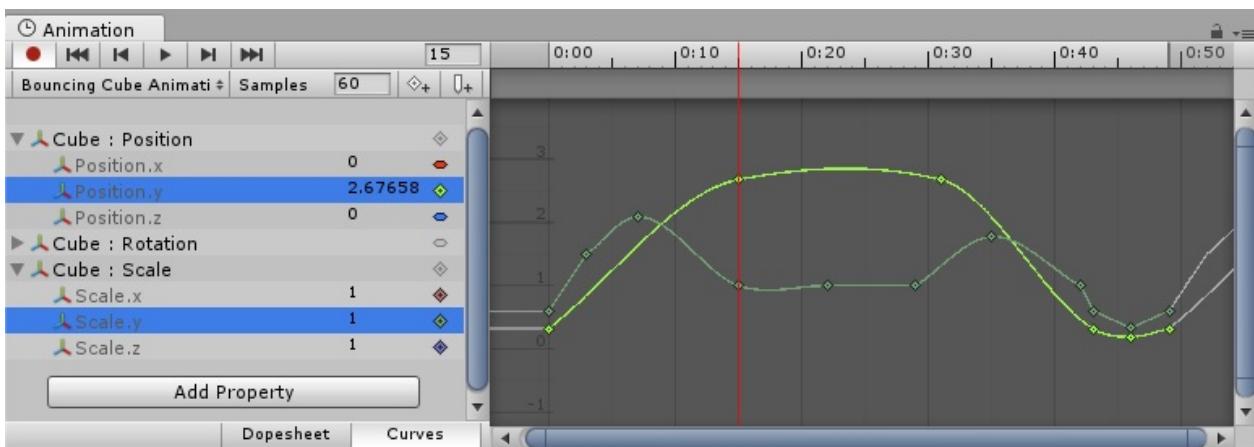
一条 动画曲线 含有（穿过）多个 关键点。在 曲线编辑器 中，关键点显示为曲线上的小菱形。含有一个或多个关键点的桢称为 关键帧。

如果某个属性在当前预览的桢上有一个 关键点，对应的曲线上也会有一个小菱形，在属性列表中也会有一个小菱形紧跟在值后面。



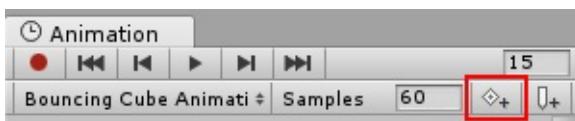
属性 **Rotation.y** 在当前预览帧上有一个关键点。

曲线编辑器 只会显示当前选中属性的曲线。如果在属性列表中选中了多个属性，曲线将叠加显示。



当多个属性被选中时，它们的曲线叠加显示在曲线编辑器中。

添加和移动关键帧



通过点击 关键帧按钮，你可以在当前预览的帧上添加一个 关键帧。

通过点击 关键帧按钮，你可以在当前预览的帧上添加一个 关键帧。这一操作将会为当前所有选中的曲线添加关键帧。或者，通过在想要新增 关键帧 的位置双击曲线，你可以为单条曲线添加关键帧。也可以右键点击 关键帧线，在上下文菜单中选择 添加关键帧，来添加 关键帧。一旦被添加，就可以用鼠标拖动 关键帧。也可以一次选中多个关键帧，然后一起拖动。通过先选中关键帧，然后按下删除键，可以删除关键帧；或者先右键点击关键帧，然后在上下文菜单中选择 删除关键帧。

支持动画的属性

动画视图 不仅仅支持 游戏对象 的位置、旋转和缩放。任何 组件 和 材质 的属性都可以被赋予动画 — 甚至是脚本组件的公共变量。通过添加 动画曲线，可以用可视化的方式，为相关属性制作复杂的动画和行为。

动画系统支持下面类型的属性：

- `Float`
- `Color`
- `Vector2`
- `Vector3`
- `Vector4`
- `Quaternion`
- `Boolean`

不支持数组，不在上面列表中的结构和对象也不支持。

对于布尔型属性，`0` 等价于 `False`，其他值等价于 `True`。

下面是一些可以用动画视图制作的示例：

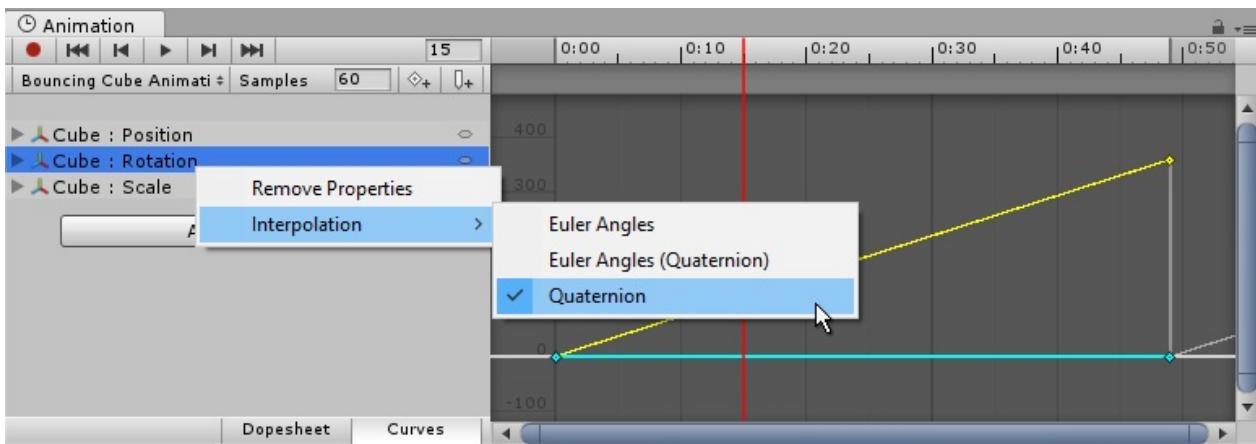
- 为灯光的颜色和强度制作动画，制作闪烁、摇曳、搏动效果。
- 为循环播放的声音源的音调和音量制作动画，制作生动的吹风、运行中的发动机或流水效果，同时保持音频文件最小化。
- 为材质球的贴图偏移制作动画，模拟移动履带、移动轨道、流水或特有的效果。
- 为多个椭圆体粒子发射器的发射状态和速度制作动画，制作壮丽烟火或喷泉效果。
- 为脚步组件的变量制作动画，制作随时间变化的行为。

当使用 动画曲线 控制游戏逻辑时，需要注意的是，在 Unity 中，动画是[重复播放和片段化](#)的。

旋转插值类型

在 Unity 内部，旋转角度是用 四元数 **Quaternions** 表示的。四元数包含 `.x`、`.y`、`.z` 和 `.w`，通常情况下，不应该手动修改它们的值，除非你明确知道它们的工作原理。相反，旋转角度通常用欧拉角 **Euler Angles** 表示，它包括 `.x`、`.y` 和 `.z`，分别表示围绕对应坐标轴旋转的角度。

在两个旋转角度之间插值时，既可以基于 四元数 **Quaternion**，也可以基于 欧拉角 **Euler Angles**。当为变换组件 **Transform** 的旋转制作动画时，动画视图 允许选择哪种插值类型。不过，无论使用哪种插值类型，旋转角度总是以 欧拉角 **Euler Angles** 格式显示。

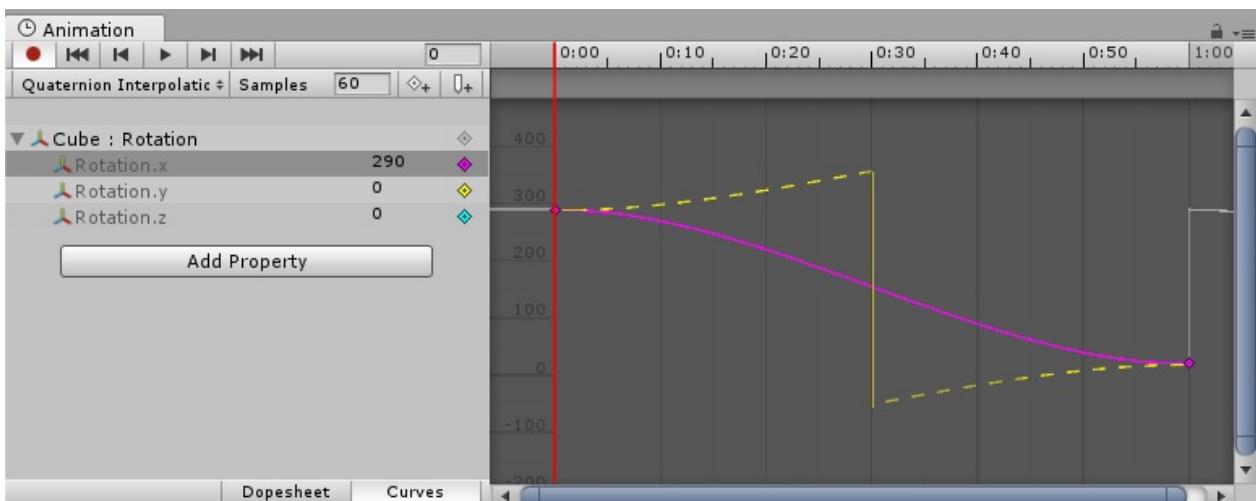


变换组件 Transform 的旋转角度可以使用欧拉角 **Euler Angles** 插值或四元数 **Quaternion** 插值。

四元数 **Quaternion** 插值

在旋转时，四元数 **Quaternion** 插值总是沿着两个旋转角度之间的最短路径生成平滑动画。从而避免不必要的旋转插值，例如万向节死锁 **Gimbal Lock** 这样的人造物体。不过，四元数 **Quaternion** 插值不能表示大于 180 度的旋转角度，因为它的行为总是寻找最短路径。（你可以设想，在球体的表面选取两个点 — 它们之间的最短路径永远不会大于球体的一半）。

如果你使用四元数 **Quaternion** 插值，并且设置大于 180 度的旋转角度，动画视图中绘制的曲线依然显示为大于 180 度，但实际上在旋转物体将采用最短路径。



使用四元数 **Quaternion** 插值，设置两个相差 270 度的关键点，将导致插值执行另一个路径，实际上只有 90 度。动画视图中现实的是洋红色曲线。在这张截图中，对象真正的插值用黄色点线表示，并不会真正显示在编辑器中。

当使用四元数 **Quaternion** 插值类型表示旋转角度时，改变关键点，或者 x、y、z 曲线之一的切线，可能会改变另外两条曲线的值，因为这 3 条曲线是由内部的四元数 **Quaternion** 创建的。当使用四元数 **Quaternion** 插值类型时，关键点之间是关联的，所以，在某个特定时间点，为 3 条曲线中的一条（x、y 或 z）创建关键点时，将同时为另外两条曲线创建关键点。

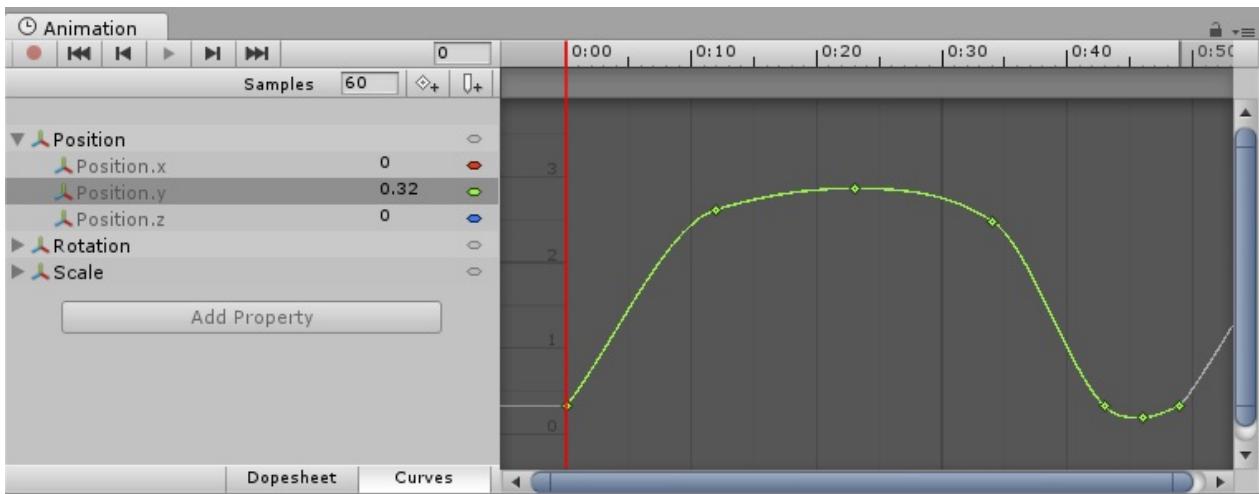
欧拉角 Euler Angles 插值

大多数人使用的其实是欧拉角 Euler Angles 插值。欧拉角 Euler Angles 可以表示任意大的旋转角度，并且，**.x**、**.y** 和 **.z** 曲线是彼此独立的。当同时围绕多个坐标轴旋转时，欧拉角 Euler Angles 插值可以基于某个物体，例如万向节死锁 Gimbal Lock，但是可以一次只围绕一条坐标轴旋转，更加简单和符合直觉。有点类似于从外部程序导入动画到 Unity 中。需要注意的是，当烘培曲线时可能添加额外的关键点，并且，在子桢中，常量类型的切线可能不完全精确。

编辑动画曲线

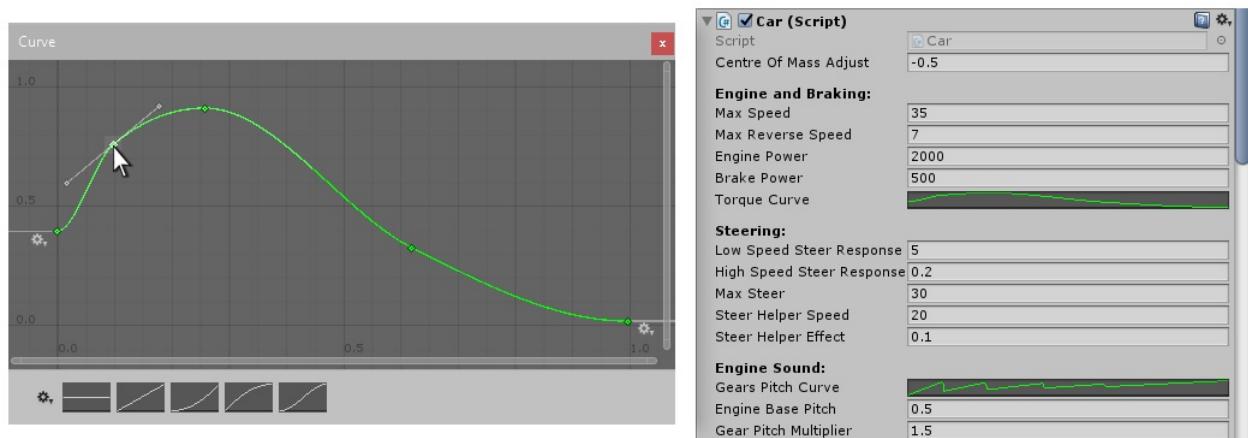
Unity 编辑器提供了几种不同的功能和窗口，用 曲线 来显示和编辑数据。这些用来查看和编辑的方法大体上相同，但是也有一些区别。

- 动画视图 使用曲线来显示和编辑 动画剪辑 中动画属性随时间变化的值。



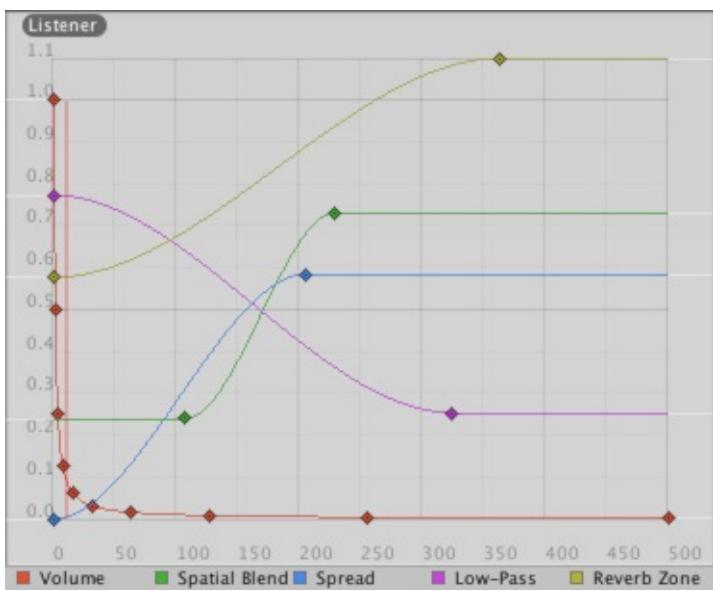
动画视图

- 脚步组件可以含有 动画曲线 类型的成员变量，可以用于各种对象。在检视视图中点击这些变量将弹出 曲线编辑器 。



曲线编辑器。

- 声源 组件使用曲线来控制声源的衰减量（随着距离变化）和其他属性。



在检视视图中，声源组件的距离函数曲线。

虽然这些控制方法有些微小差异，但是，所有曲线的编辑方式完全相同。本页将介绍如何在这些控制方法中导航和编辑曲线。

在曲线上添加和移动关键点

在想要放置关键点的位置双击曲线，可以在该点添加一个关键点。也可以右键点击曲线，并从上下文菜单中选择 **添加关键点**，来添加一个关键点。

添加的关键点可以用鼠标拖动：

- 点击关键点来选中它。用鼠标拖动选中的关键点。
- 拖动时，在 Mac 上按下 **Command** 键或在 Windows 上按下 **Control** 键，可以让关键点对齐到网格。

也可以一次选中多个关键点：

- 想要选择多个关键点，请在点击关键点时按住 **Shift** 键。
- 想要取消选择的关键点，请在点击关键点时按住 **Shift** 键。
- 想要选择矩形区域内的所有关键点，请点击空白处并拖动鼠标形成一个矩形选择区域。
- 也可以在矩形选择时按住 **Shift** 键，把矩形区域内的关键点添加到已选中的关键点中。

选中并按下 **Delete** 键，或者右键点击并从上下文菜单中选择 **Delete Key**，可以删除关键点。

编辑关键点

在曲线编辑器中直接编辑关键点的值，是 Unity 5.1 新增的一个功能。使用 **Enter/Return** 键或上下文菜单来开始编辑选中的关键点，使用 **Tab** 键在输入域之间切换，使用 **Enter/Return** 键来提交修改，使用 **Escape** 来取消修改。

浏览曲线视图

使用 动画视图 时，可以很容易地放大要处理的曲线细节，或缩小曲线以获得完整的图像。

你可以随时按下 **F** 键，完整地查看曲线，或者放大显示选中的关键点。

译注：没有选中关键点时，曲线被自动缩放至合适的尺寸，以完整显示整个曲线；选中关键点时，将放大曲线，视图中只显示选中的关键点。

缩放

使用鼠标滚轮或触控板的缩放功能，或者鼠标右键拖动时按住 **Alt** 键，可以缩放曲线视图。

你可以只在水平或垂直方向上缩放曲线视图。

- 在 Mac 中按住 **Command** 键，或者在 Windows 中按住 **Control** 键，只进行水平缩放。
- 按住 **Shift** 键，只进行垂直缩放。

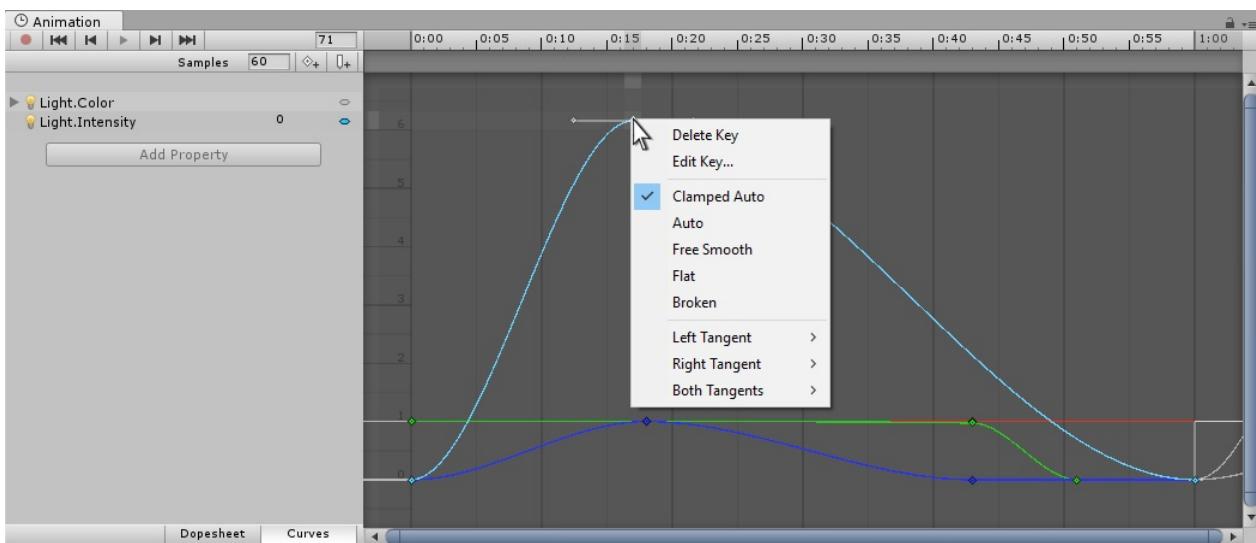
此外，你可以在曲线视图中拖动滚动条的两端来缩小或放大可见区域。

移动

按住鼠标滚轮并拖动，或鼠标左键拖动时按住 **Alt** 键，可以移动曲线视图。

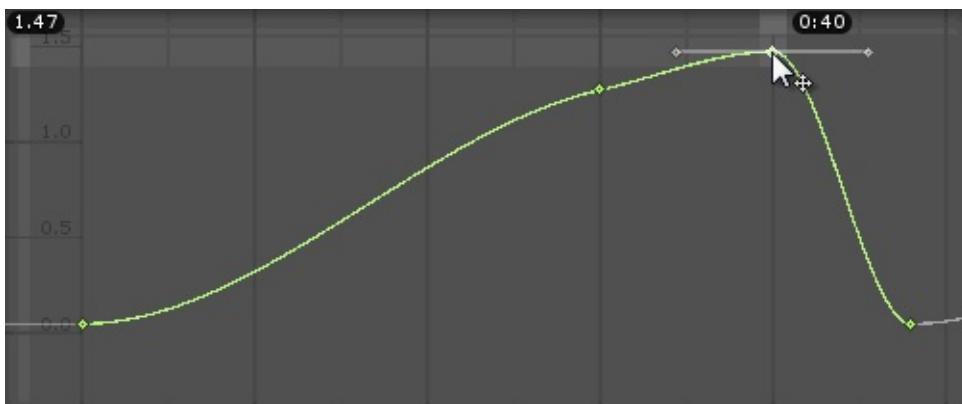
编辑切线

一个关键点拥有两条 切线 —— 左侧的逼近斜率和右侧的远离斜率。切线控制了关键点之间的曲线形状。你可以从多个不同的切线类型中选择，来控制曲线如何离开一个关键点和到达下一个关键点。右键单击某个关键点来选择该关键点的切线类型。

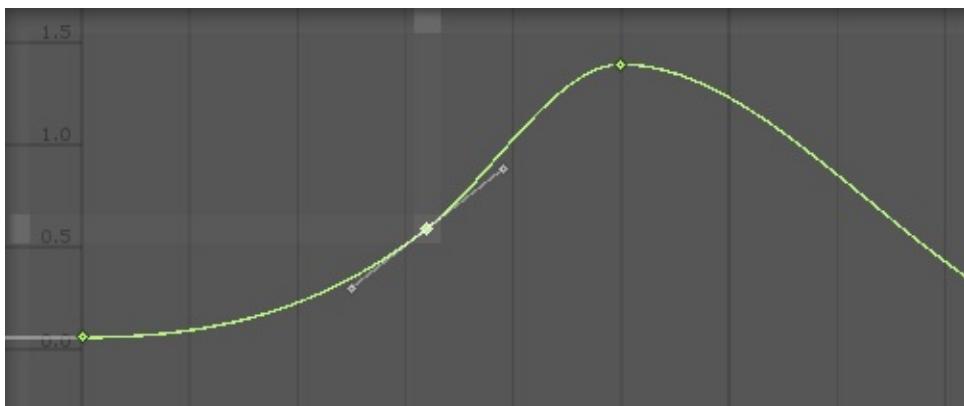


要使动画值在经过关键点时平滑变化，左侧切线和右侧切线必须是共线的（在同一条直线上）。下面的切线类型可以保证平滑度：

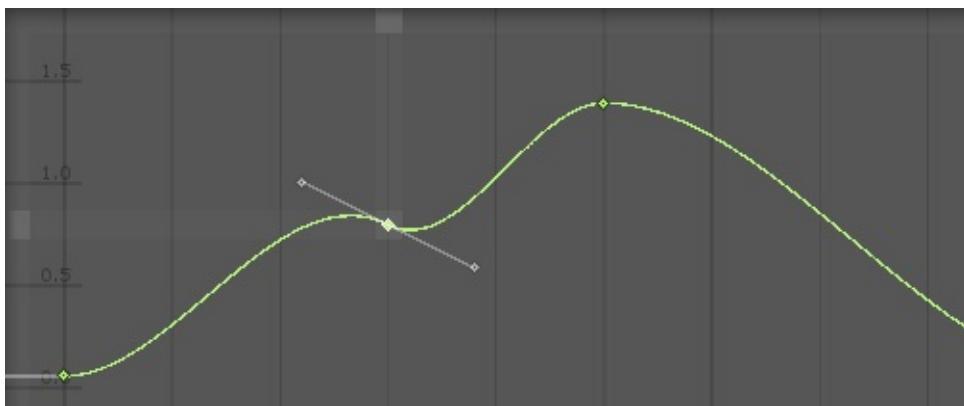
- 自动钳位 (**Clamped Auto**)：默认切线类型。自动设置切线，使曲线平滑地经过该关键点。当编辑该关键点的位置或时间时，自动调整切线，以避免曲线被过度调整。在自动钳位模式下，如果你手动调整了该关键点的切线，切线类型会被切换为自由平滑 (**Free Smooth**) 模式。在下面的例子中，当移动关键点时，切线的逼近斜率和远离斜率被自动调整：



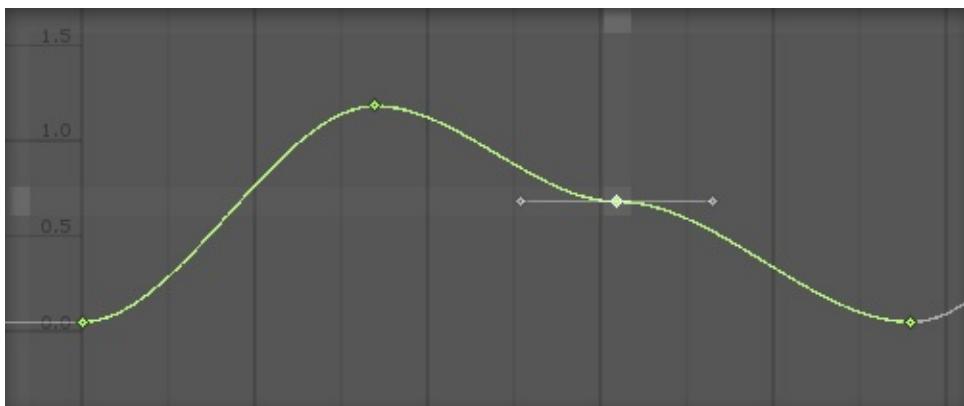
- 自动 (**Auto**)：这是一个遗留切线模式，保留该模式是为了向后兼容旧项目。除非你有特殊原因必须使用该模式，否则请使用默认的钳位自动 (**Clamped Auto**)。当关键点被设置为该模式时，切线被自动设置，使曲线平滑地经过该关键点。但是，相比钳位自动 (**Clamped Auto**) 模式，有两点差异：
 1. 当编辑该关键点的位置或时间时，切线不会自动调整；切线仅仅在该关键点被第一次设置为自动 (**Auto**) 模式（初始化）时才会进行调整。
 2. Unity 在计算切线时，不会考虑避免过度调整（即可能会导致过度调整）。



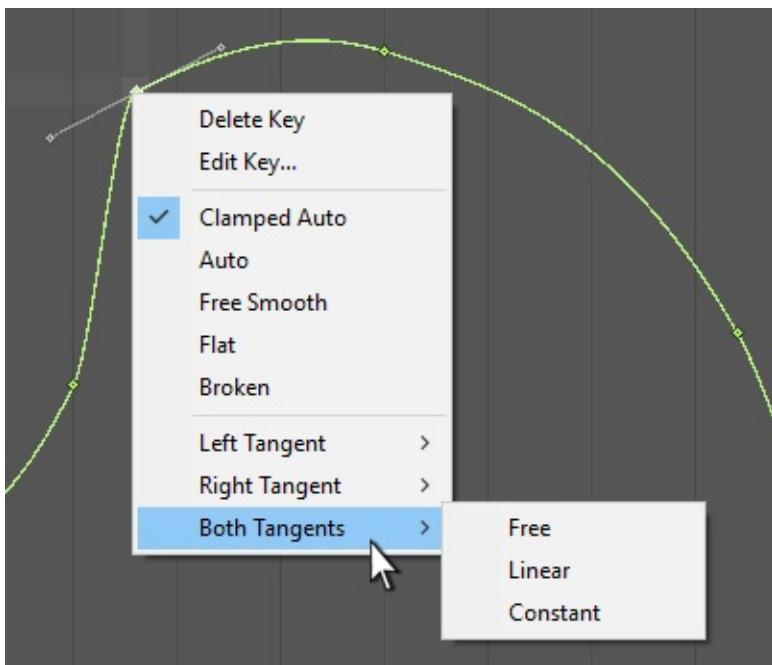
- **自由平滑（Free Smooth）**：拖动切线图柄来自由地设置切线。切线被锁定为共线，以确保平滑度。



- **水平（Flat）**：切线被设置为水平（自由平滑的一种特殊情况）。

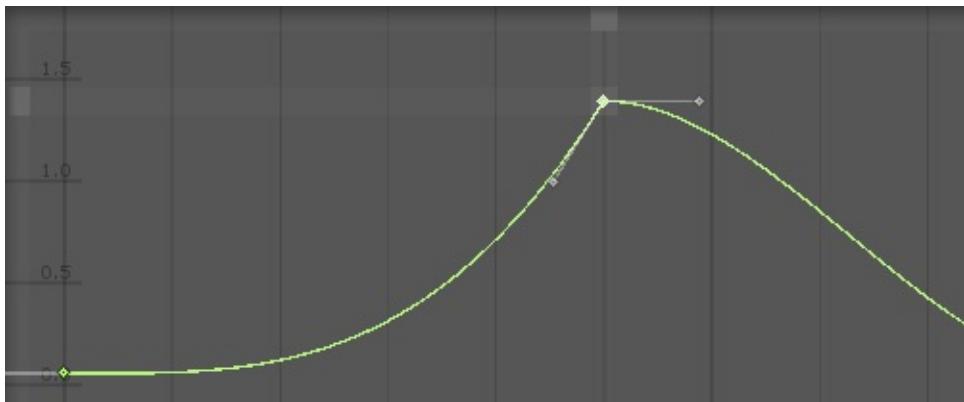


有时你可能不希望曲线平滑地通过某个关键点。想要在曲线中创建剧烈的变化，请选择折线切线模式。

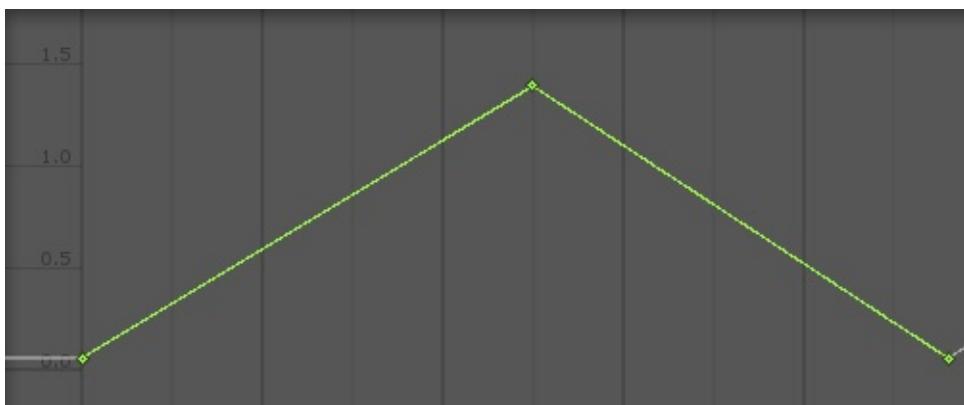


使用折线切线时，可以单独设置左右切线。左右切线可以设置为以下类型之一：

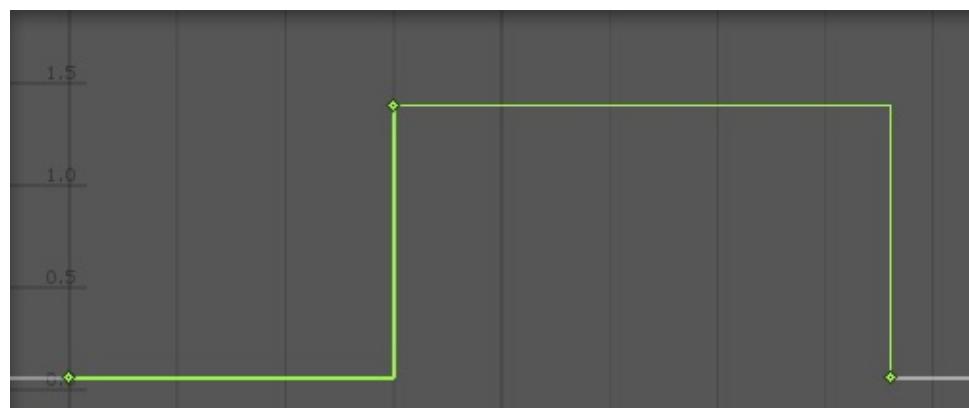
- 折线 - 自由（**Broken - Free**）：拖动切线图柄来自由地设置切线。



- 折线 - 直线（**Broken - Linear**）：切线指向相邻的关键点。要创建直线型的曲线片段，请将切换的两边都设置为 直线 **Linear**。在下面的例子总，3 个关键点都被设置为 折线 - 直线，**Broken - Linear**，以实现关键点到关键点的直线。



- 折线 - 常量（**Broken - Constant**）：曲线在两个关键点之间保持为一个常量值。左侧关键点的值决定了曲线片段的值。



动画控制器

动画控制器允许你为一个角色或其他游戏对象安排和维护一组动画。

控制器引用了动画剪辑，并且使用 状态机 来管理各种动画状态和它们之间的转换，可以把状态机认为是一种流程图，或者是一段在 Unity 中用可视化编程语言编写的简单程序。

下面的章节涵盖了 动画系统 **Mecanim** 提供的控制和序列化动画的主要功能。

动画控制器资源

准备好动画剪辑后，你还需要使用 动画控制 **Animator Controller** 把它们整合在一起。动画控制器资源由 Unity 创建，允许为一个角色或对象维护一组动画。

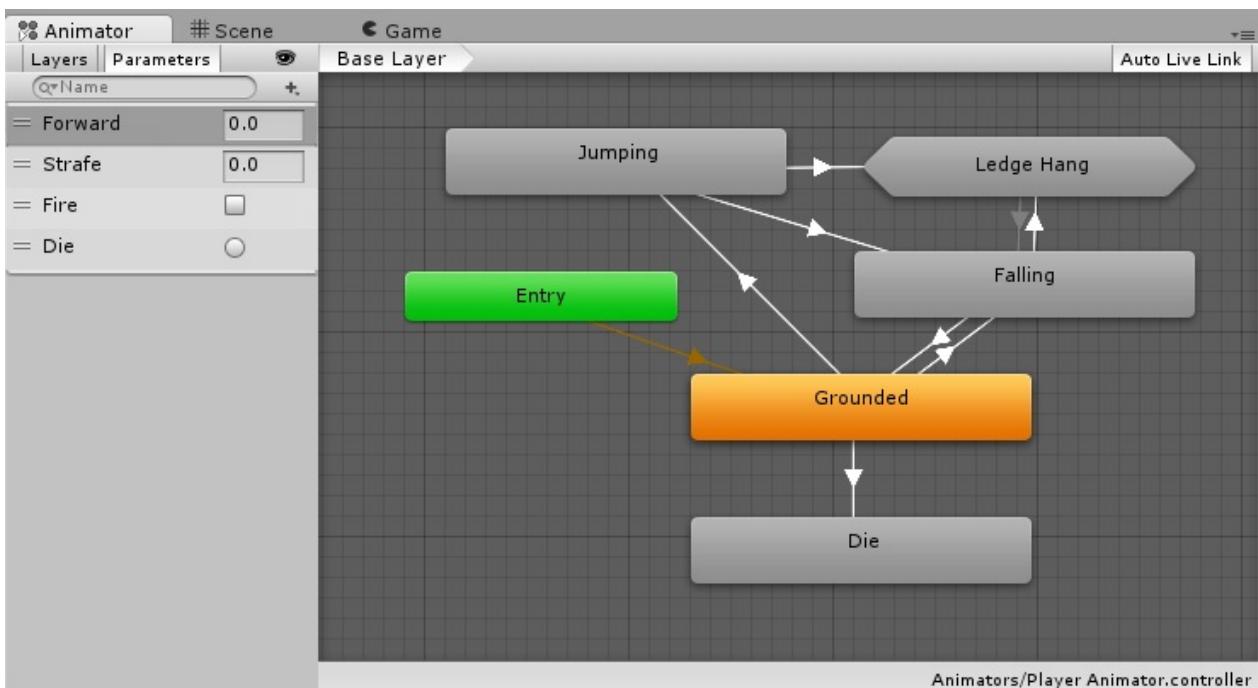


项目视图中的动画控制器资源。

可以从 **Assets** 菜单或项目视图的 **Create** 菜单创建动画控制器。

最常见的情况是，拥有多个动画剪辑，并在特定游戏条件发生时，在它们之间切换。例如，每当按下空格键时，可以从行走动画切换为跳跃动画。即使是只有一个动画剪辑，想要在游戏对象上使用它，也需要把它放入一个动画控制器中。

动画控制器使用所谓的 状态机 **State Machine** 来管理各种动画和它们之间的切换。可以把状态机认为是一种流程图，或者是一段在 Unity 中用可视化编程语言编写的简单程序。有关状态机的更多信息可以在 [这里](#) 找到。可以在 [动画控制器视图](#) 中创建、查看和修改动画控制器的结构。

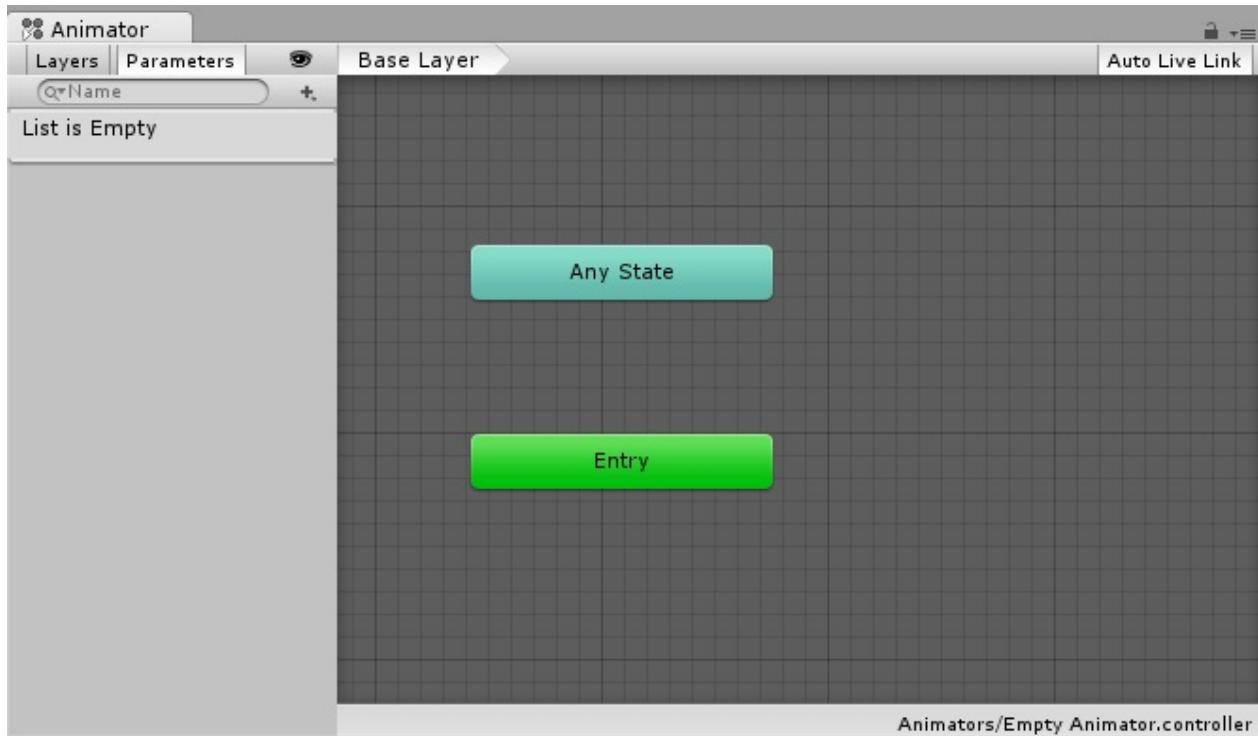


一个简单的动画控制器

最终，动画控制器通过 [动画组件](#) 来应用到游戏对象上，动画组件引用了动画控制器。更多详细信息请参阅 [动画组件](#) 和 [动画控制器](#) 的参考手册。

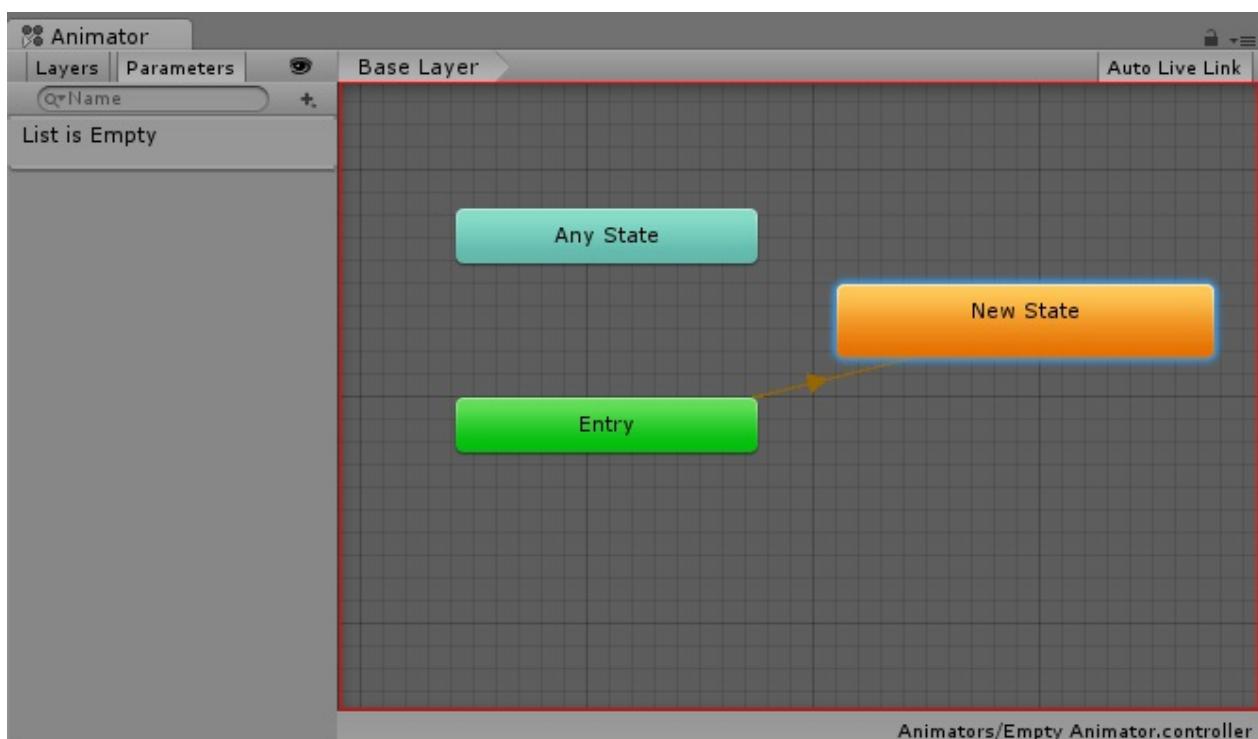
动画控制器视图

动画控制器视图允许你创建、查看和修改动画控制器资源。



动画控制器视图显示了一个新的空动画控制器资源。

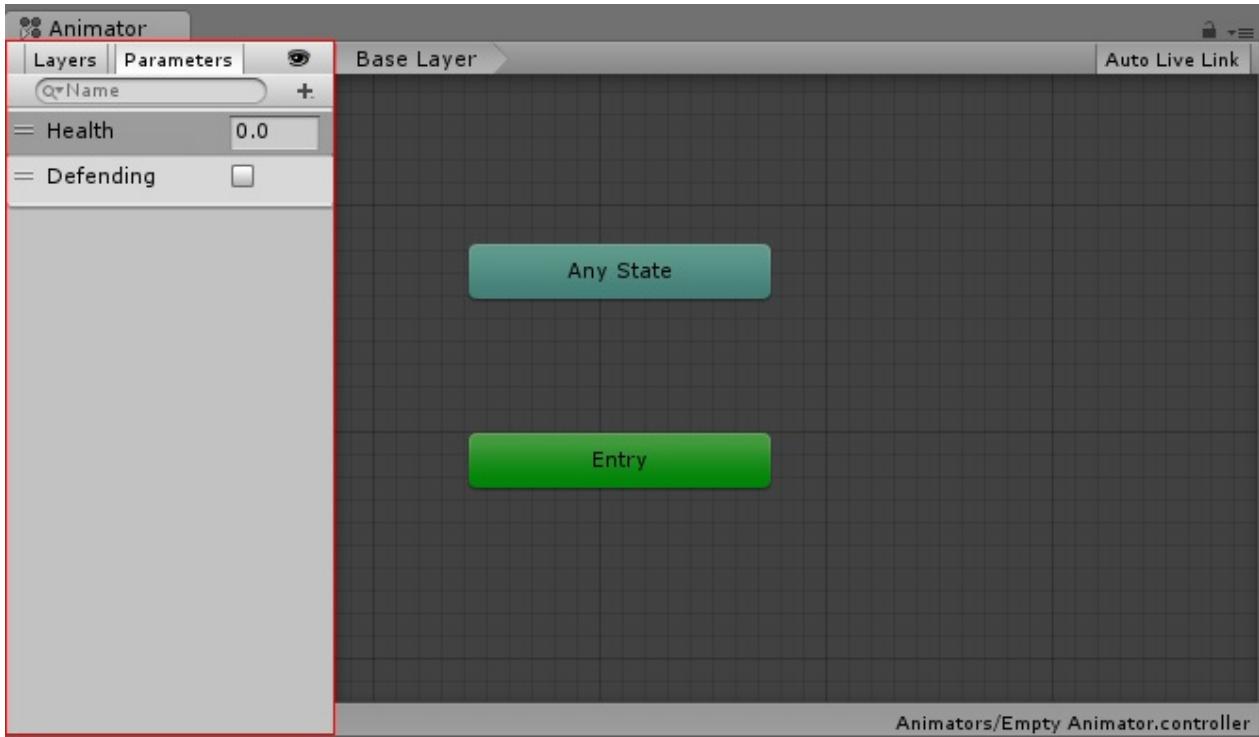
动画控制器视图主要有两部分：网格布局主体区域，左侧的分层和参数面板。



动画控制器视图的布局区域

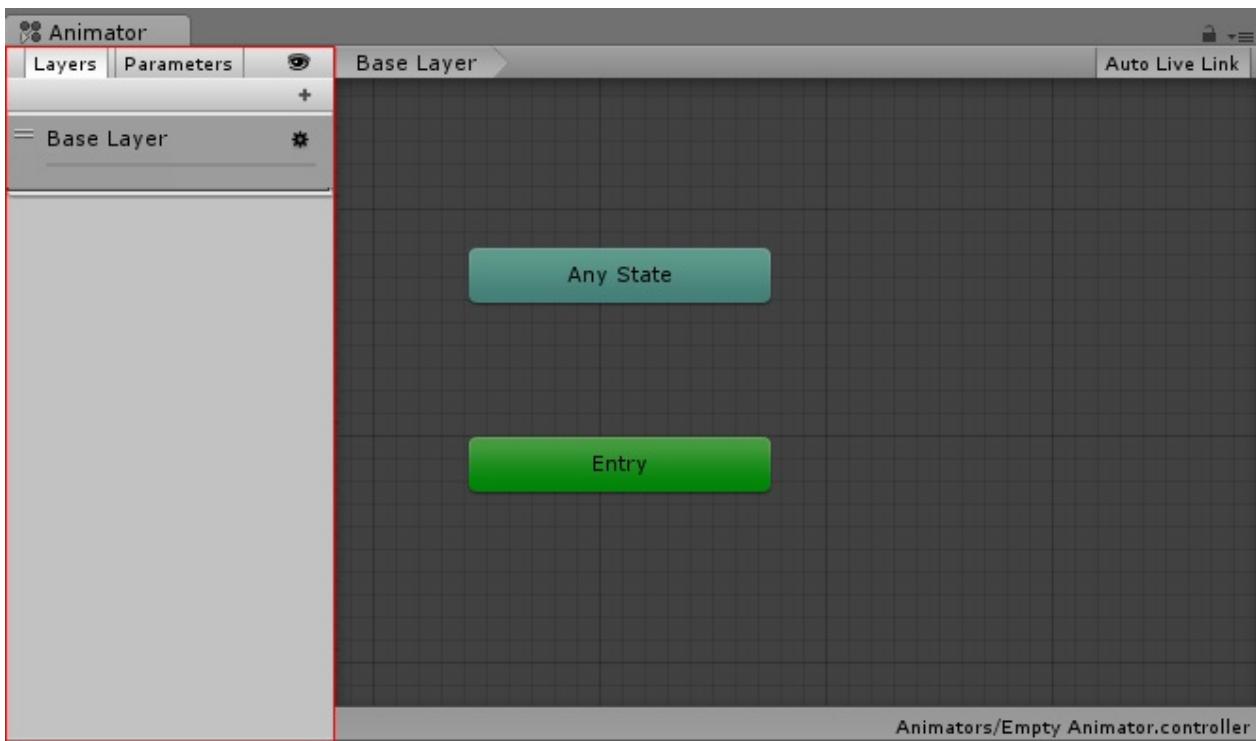
深灰色网格部分是主体布局区域。你可以在这里创建、排列和连接 动画控制器 的状态（即动画剪辑）。

可以在网格上右键点击创建一个新的状态节点。使用鼠标中键拖动，或拖动时按住 Alt/Option 键，可以移动视图。可以单击状态节点编辑它们，可以点击并拖动状态节点来重新排版状态机的布局。



参数视图，创建了两个示例参数。

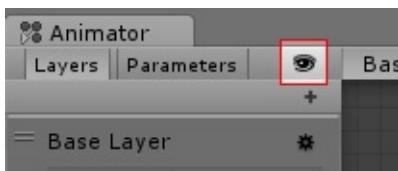
左侧的面板可以在参数视图和分层视图之间切换。参数视图允许你创建、查看和编辑 动画控制器参数。这里定义的变量用作状态机的输入。想要添加一个参数，请点击加号图标，然后从弹出的菜单中选择参数类型。想要删除一个参数，请在列表中选中参数，然后按下删除键。



分层视图

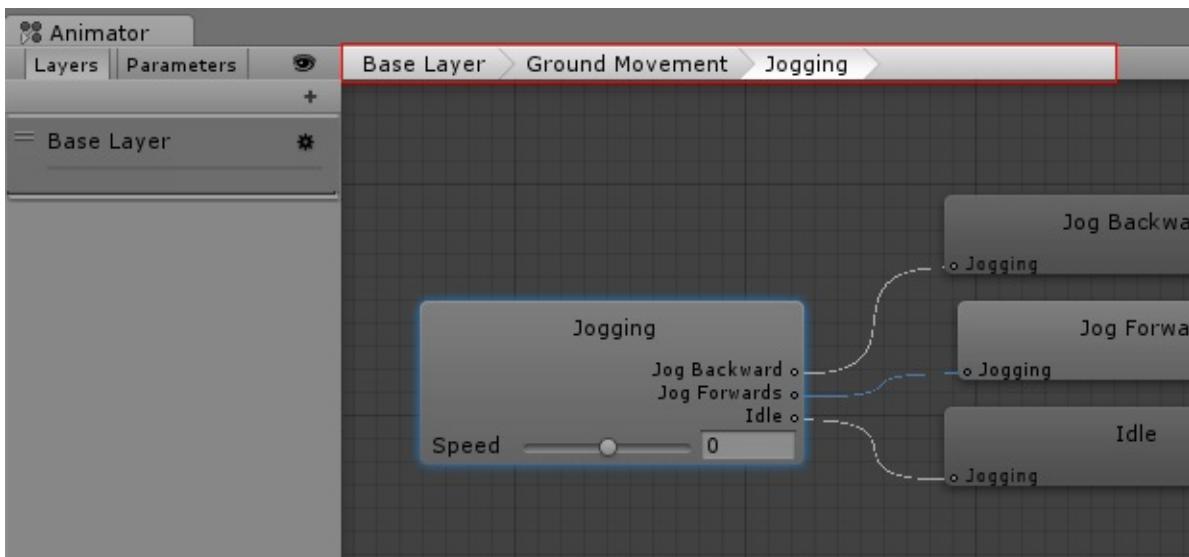
当左侧面板被切换为分层视图时，你可以创建、查看和编辑动画控制器的**分层**。允许在一个动画控制器中同时运行多个动画分层，每个分层由一个独立的状态机控制。一个常见的用例是，用一个基础分层控制角色的移动动画，在这之上，用一个独立分层控制上身动画。

想要增加一个分层，请点击加号图标。想要删除一个分层，请选中它，然后按下删除键。



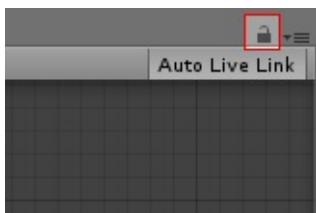
分层和参数的隐藏图标

点击眼睛图标，将显示或隐藏参数和分层边栏，这样可以有更多的空间来查看和编辑状态机。



层级面包屑定位

层级面包屑定位。状态可以包含 子状态 和 混合树，并且，这些结构可以重复嵌套。当向下钻入（进入）子状态时，这里会列出父级状态和当前（子级状态）。点击父级状态可以跳回父级状态，后者直接返回状态机的根状态。



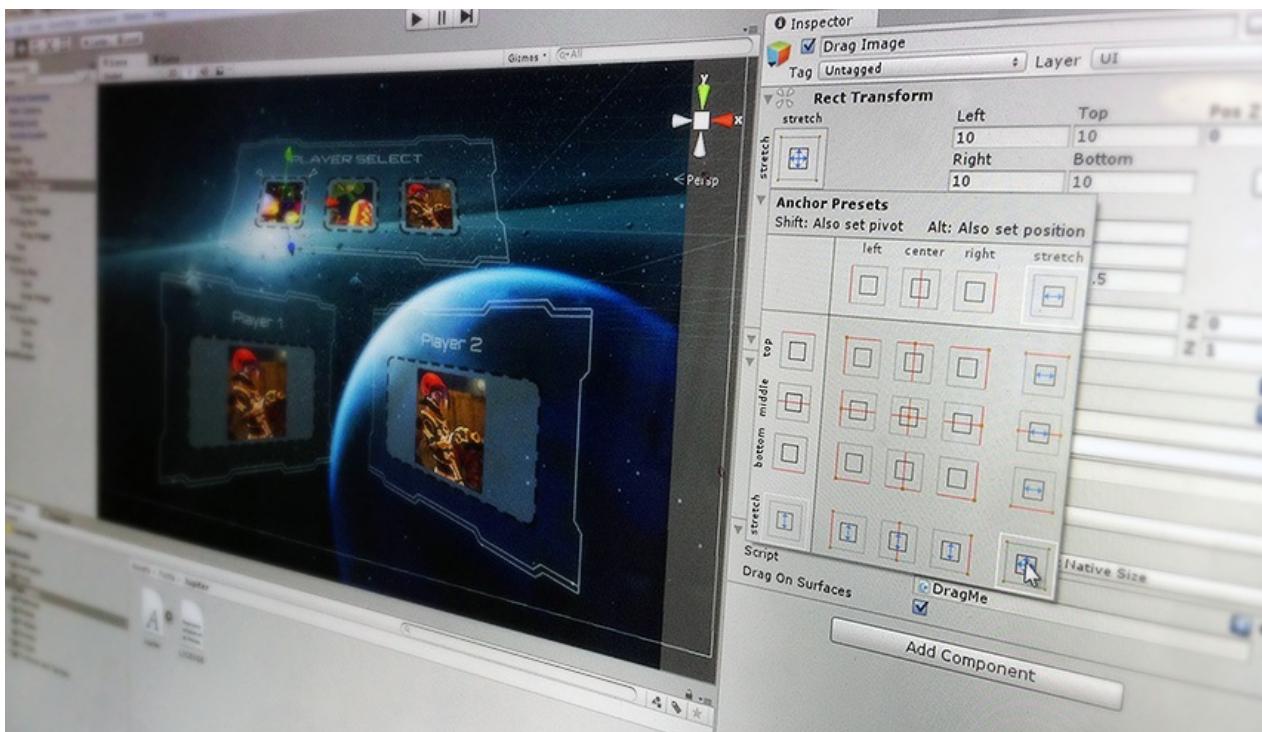
锁定图标

启动锁定图表将使动画控制器视图聚焦于当前状态机。当锁定图标关闭时，点击一个新的动画控制器资源或带有动画组件的游戏对象时，动画控制器视图将现实该元素的状态机。锁定视图，使动画控制器视图总是显示同一个状态机，无论你选择了其他哪个资源或游戏对象。

动画状态机

在游戏中，一个角色或其他动画游戏对象通常具有多个对应不同行为的动画剪辑。例如，一个角色在空闲时可能会呼吸或轻微摇摆，当收到命令时开始行走，从平台跌落时惊慌地抬起手臂。一扇门可以具有打开、关闭、卡住和被砸开动画。动画系统使用一种类似于流程图的可视系统来表示状态机，是使你可以控制和序列化想要应用在角色或游戏对象上的动画剪辑。本章提供了有关动画系统的状态机的更多内容，并介绍如何使用它们。

用户界面 UI



UI 系统允许你快速地、直观地创建用户界面。本节介绍 Unity UI 系统的主要功能。

相关教程：[用户界面 \(UI\)](#)

有关提示、技巧和错误排查，请搜索 [Unity 知识库](#)。

画布 Canvas

画布 **Canvas** 是一块包含了所有 UI 元素的区域。画布 Canvas 是一个带有 Canvas 组件的游戏对象，所有的 UI 元素都必须是 Canvas 对象的子对象。

新建一个 UI 元素，例如通过菜单 **GameObject > UI > Image** 创建一个图像，如果场景中不存在 Canvas 对象，将自动创建一个。UI 元素作为 Canvas 的子对象被创建。

Canvas 区域在场景视图中显示为一个矩形。这样可以很容易地定位 UI 元素，而不需要（不依赖）一直显示游戏视图。

画布 **Canvas** 使用 EventSystem 对象来通知 Messaging System。

元素绘制顺序

画布 Canvas 中的 UI 元素的绘制顺序与它们在层级视图中显示的顺序相同。首先绘制第一个子对象，然后是第二个子对象，以此类推。如果两个 UI 元素发生重叠，后一个将显示在前一个之上。

要使某个元素显示在其他元素之上，只需要在层级视图中通过拖动重新排序元素。也可以在脚本中通过 Transform 组件上的这些方法来控制顺

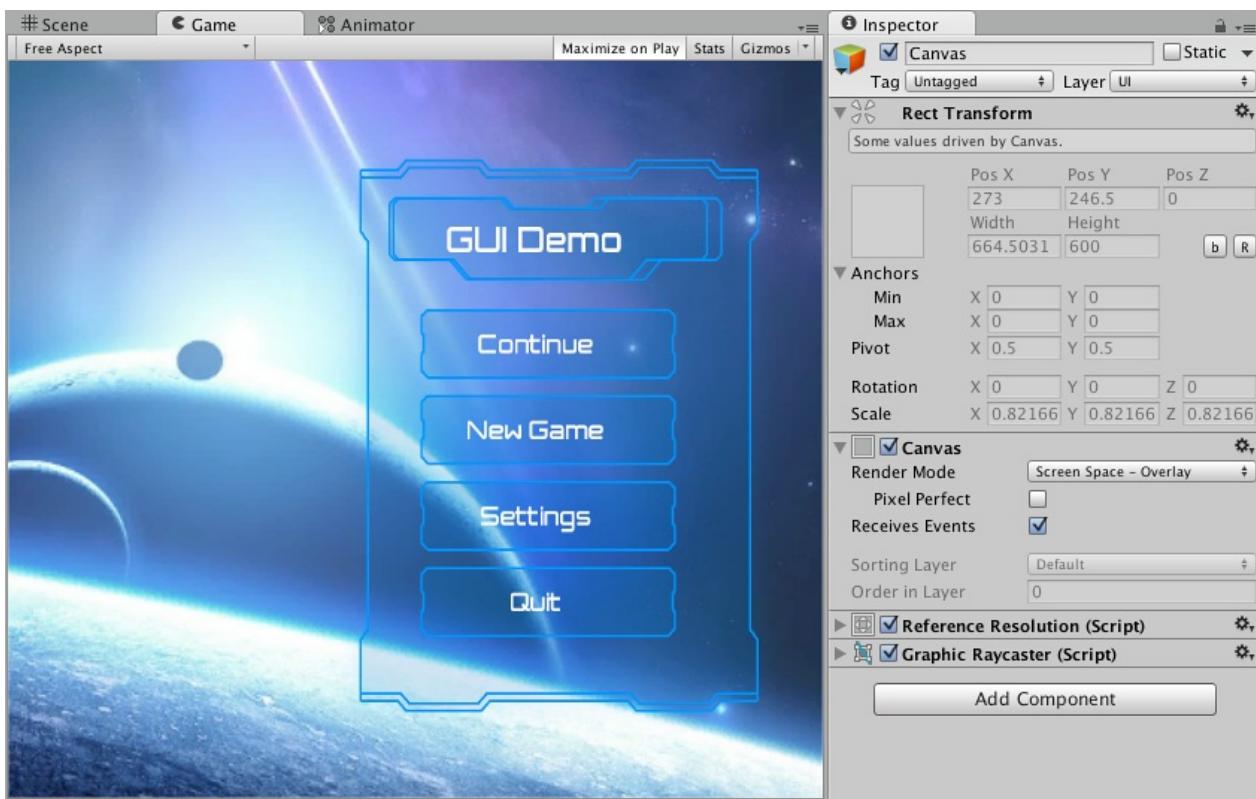
序：`SetAsFirstSibling`、`SetAsLastSibling` 和 `SetSiblingIndex`。

渲染模式

画布 Canvas 具有一个渲染模式 **Render Mode** 设置，可以用于控制是在场景空间还是世界空间中渲染。

场景空间 - 叠加

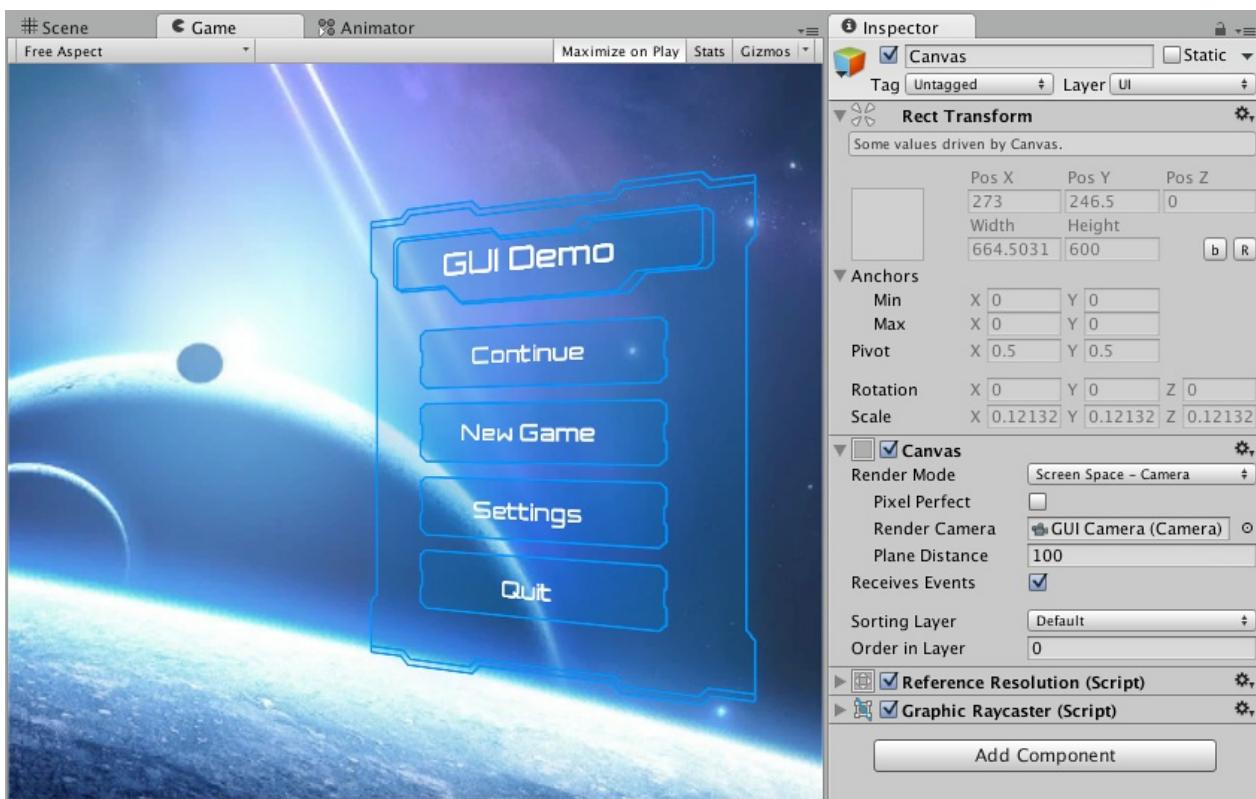
该渲染模式放置 UI 元素到屏幕上，位于场景的顶部。如果屏幕调整大小或改变分辨率，画布 Canvas 将自动改变大小，以匹配屏幕。



UI，场景空间与画布叠加

场景空间 - 摄像机

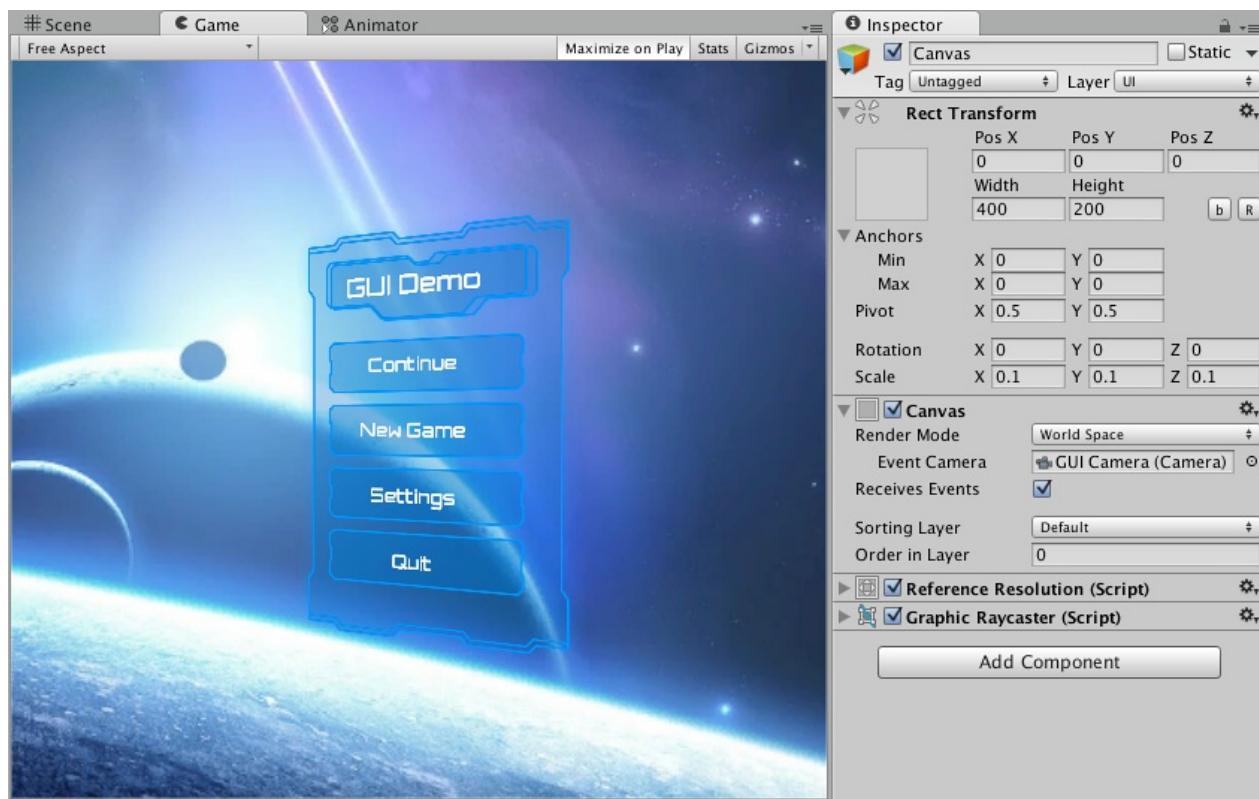
该渲染模式类似于 场景空间 - 叠加，但是该模式的画布 Canvas 被放置在特定 摄像机 Camera 前方给定距离的位置上，这意味着摄像机 Camera 的设置会影响 UI 的外观。如果摄像机 Camera 被设置为 透视 Perspective，UI 元素将以透视方式渲染，透视变形程度可以用摄像机 Camera 的 视野 Field of View 控制。如果屏幕调整大小、改变分辨率或者摄像机 Camera 视锥发生变化，画布 Canvas 也将自动改变大小，以匹配屏幕。



UI，场景空间 - 摄像机 - 画布

世界空间

在该渲染模式下，画布 **Canvas** 的行为与场景中的任意其他对象一样。通过矩形变换组件 **Rect Transform** 可以手动调整画布 **Canvas** 的大小，**UI** 元素将基于 3D 位置被渲染在场景中其他对象的后面。这意味着，**UI** 成为了世界的一部分。这种模式也被成为『剧情界面』。



UI，世界空间 - 画布

基本布局

在本节中，我们将看看如何定位 UI 元素，相对于画布 Canvas 或其他元素。如果你想边阅读边测试，可以使用菜单 **GameObject -> UI -> Image** 创建一个 Image。

矩形工具 Rect Tool

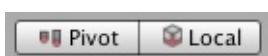
为了方便布局，每个 UI 元素都用一个矩形表示。在场景视图下，可以使用菜单栏中的 矩形工具 **Rect Tool** 操作矩形。矩形工具可以用于 Unity 的 2D 功能和 UI，事实上甚至还可以用于 3D 对象。



工具栏按钮，选中了矩形工具

矩形工具可以用于移动、调整大小和旋转 UI 元素。一旦选中了某个 UI 元素，你可以通过单击矩形内部的任意位置并拖动来移动矩形。你可以通过单击边缘或边角并拖动来调整矩形大小。你可以把鼠标光标移动到稍微远离边角的位置，直到光标变成一个旋转符号，然后单击并向任意方向拖动，来旋转矩形。

就像其他工具一样，矩形工具基于工具栏上设置的当前轴心和空间。当使用 UI 时，通常最好将它们设置为 **Pivot** 和 **Local**。



工具栏按钮，设置为 Pivot 和 Local

矩形变换 Rect Transform

矩形变换 **Rect Transform** 是一个新变换组件，用于所有 UI 元素，而不是常规的 变换 **Transform** 组件。



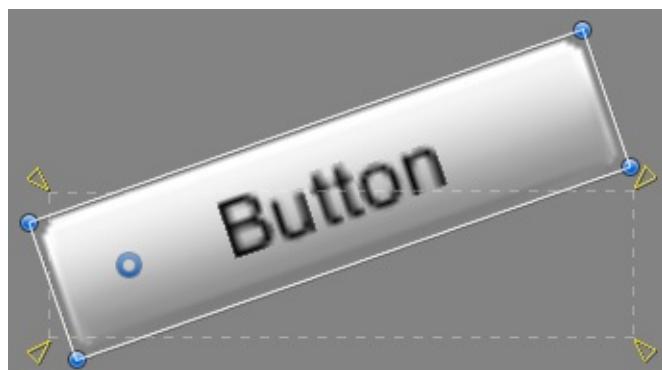
Rect Transform 具有位置、旋转和缩放属性，类似于常规的 Transform，但是前者还具有宽度和高度，用于指定矩形的尺寸。

调整大小 vs 缩放

当使用矩形工具改变对象的大小时，对于 2D 系统中的精灵 Sprite 和 3D 对象，将改变本地缩放比例 `scale`。但是，对于具有矩形变换 Rect Transform 的对象，将改变宽度和高度，保持本地缩放比例不变。并且，调整大小不会影响字体大小、切片图像的边框，等等。

轴心 Pivot

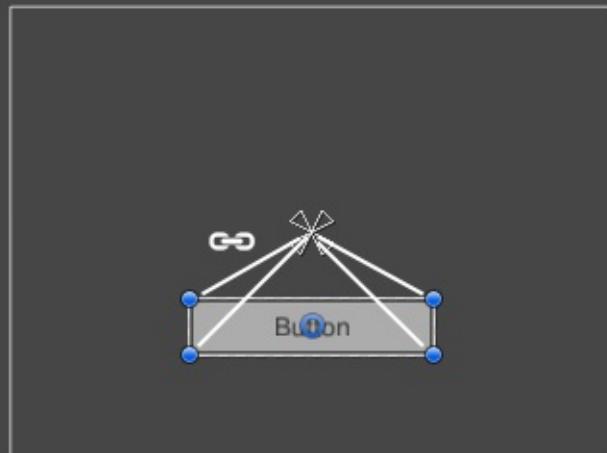
旋转、大小和缩放的变化围绕轴心进行，所以轴心的位置会影响旋转、调整大小或缩放的结果。当工具栏上的『轴心』按钮设置为 Pivot 模式时，可以在场景视图中移动矩形变换的轴心。



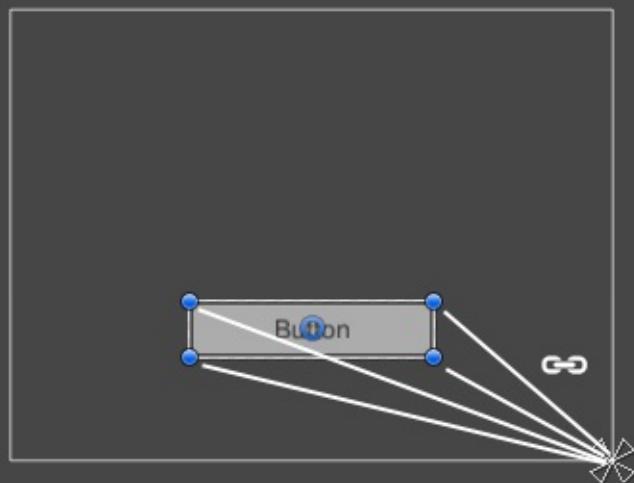
锚点 Anchors

矩形变换包含一个称为 锚点 **Anchors** 的布局概念。在场景视图中，锚点显示为 4 个小三角形手柄，锚点信息则显示在检视视图中。

如果某个矩形变换的父元素也是一个矩形变换，那么子矩形变换可以通过各种方式锚定到父矩形变换。例如，子矩形变换可以锚定到父矩形变换的中心或某个边角。

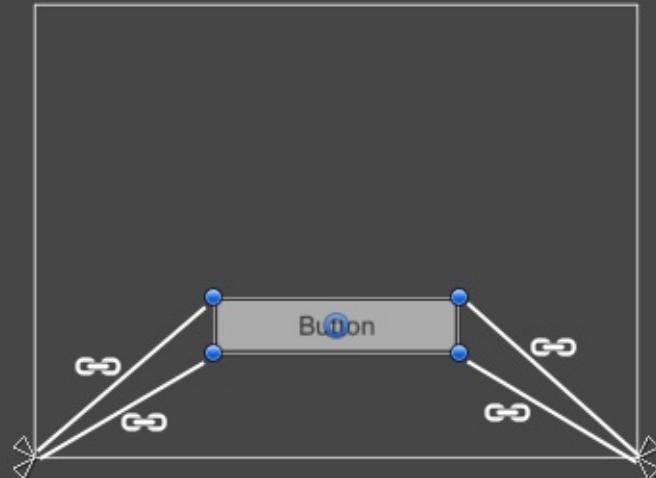


UI 元素锚定到父元素的中心。元素与中心保持固定的偏移。



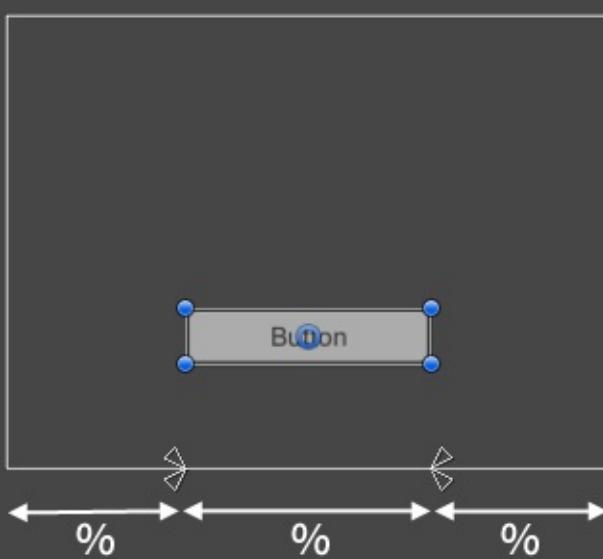
UI 元素锚定到父元素的右下角。元素与右下角保持固定的偏移。

锚定还允许子元素随着父元素的宽度或高度一起伸展。矩形的每个边角与相应的锚点具有固定的偏移，例如，矩形的左上角与左上锚点具有固定的偏移，以此类推。这样，矩形的不同边角可以锚定到父矩形的不同点。



UI 元素的左侧边角锚定到父矩形的左下角，右侧边角锚定到父矩形的右下角。该元素的边角与各自的锚点保持固定的偏移。

锚点的位置用父矩形宽度和高度的分数（或百分比）来定义。0.0（0%）对应左侧或底部，0.5（50%）对应中间，1.0（100%）对应右侧或顶部。但是锚点不限于边框和中间，子矩形可以锚定于父矩形内的任意点。



UI 元素的左侧边角锚定于距父元素左侧特定比例的点，右侧边角锚定于距父矩形右侧特定比例的点。

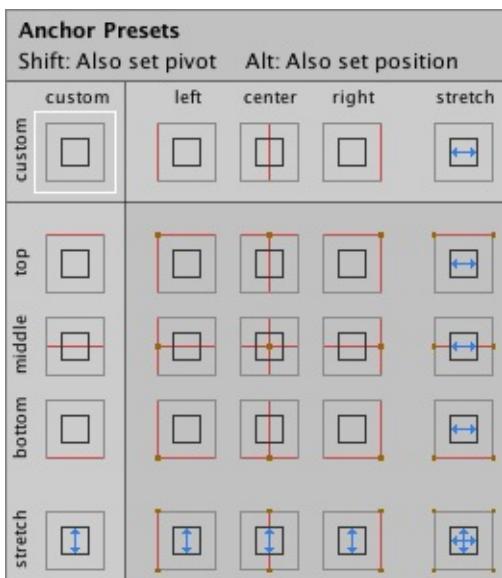
你可以单独地拖动每个锚点，也可以一次拖动多个。如果在拖动锚点时按下 **Shift** 键，对应的矩形边角将与锚点一起移动。

这段文档的前半句没看懂。

锚点手柄的一个很有用的功能是，它们自动对齐到兄弟矩形的锚点，从而可以精确定位。

预设锚点

在检视视图中，在矩形变换组件 **Rect Transform** 的左上角可以找到预设锚点按钮 **Anchor Preset**。点击该按钮将显示预设锚点下拉菜单。在这里，你可以快速选择一些最常见的锚点选项。你可以将 UI 元素锚定到父元素的边和中间，或者随父元素的大小伸展。水平和垂直锚定是独立的。



预设锚点按钮显示了当前选择的预设选项，如果有选的话。如果在水平轴或垂直轴上选择了与预设不同的位置，则会显示自定义选项。

检视视图中的锚点和位置属性

点击锚点 **Anchors** 的展开箭头，可以显示锚点的数值域（如果它们不可见的话）。**Anchor Min** 对应场景视图中左上角的锚点手柄，**Anchor Max** 对应右上角的手柄。

矩形位置属性的显示方式取决于锚点是聚集在一起（产生固定的宽度和高度）还是分离的（导致矩形随着父矩形伸展）。

当所有锚点手柄聚集在一起时，数值域显示为 **Pos X**、**Pos Y**、**Width** 和 **Height**。**Pos X** 和 **Pos Y** 的值指示了轴心相对于锚点的位置。

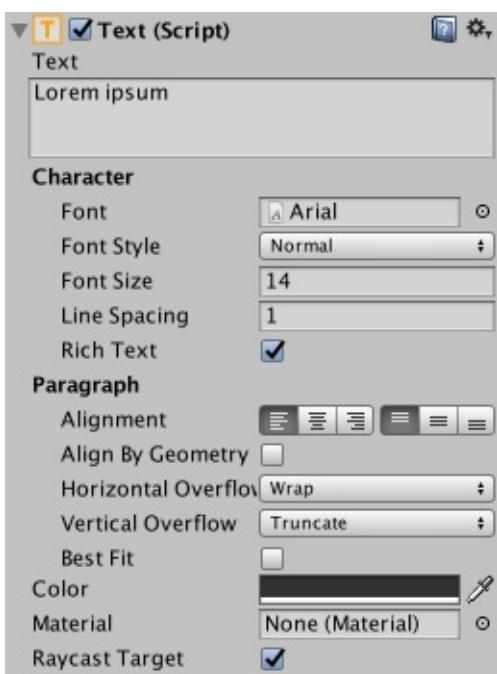
当锚点是分离的，数值域的部分或全部显示为 `Left`、`Right`、`Top` 和 `Bottom`。这些数值域定义了锚点矩形的内边距。如果锚点在水平方向上是分离的，则使用 `Left` 和 `Right` 域，如果在垂直方向上是分离的，则使用 `Top` 和 `Bottom` 域。

请注意，更改锚点域或轴心域的值通常会反过来调整位置域的值，以使矩形保持原位。如果不希望出现这种情况，可以在检视视图中开启 原始模式 **Raw Mode**，它是一个小按钮。在这种模式下，改变锚点和轴心的值不会改变任何其他值。这也可能导致矩形在视觉上移动或改变大小，因为矩形的位置和大小取决于锚点和轴心的值。

视觉组件

随着 UI 系统被引入，Unity 增加了一些新组件，用于帮助创建特定功能的 GUI。这节将介绍可以这些新组件的基础知识。

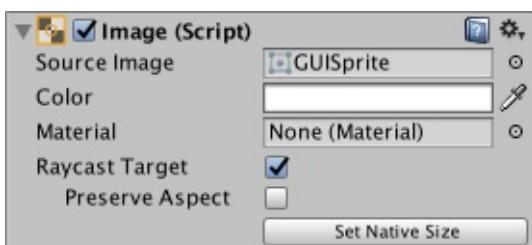
文本 Text



文本 **Text** 组件（也称为标签 Label）具有一个文本区域，用于输入将要显示的文本。可以设置字体、字体样式、字体大小，以及是否具有富文本功能。

提供了用于设置文本对齐的选项，用于水平或垂直溢出的设置（控制了文本大于矩形的宽度或高度时如何显示），和 **Best Fit** 选项（调整文本大小以适配可用空间）。

图像 Image



图像 **Image** 具有一个矩形变换组件 **Rect Transform** 和一个图像 **Image** 组件。图片精灵可以被应用于图像组件的 **Target Graphic** 域，颜色可以在 **Color** 域设置。材质也可以应用于图像组件。**Image Type** 域定义了图像精灵的显示方式，选项如下：

- 简单 **Simple** - 等比缩放整个图像精灵。
- 分割 **Sliced** - 使用 3×3 精灵分割，调整大小不会使边角变形，只会伸展中心部分。
- 平铺 **Tiled** - 类似于 **Sliced**，但是平铺（重复）中心部分，而不是伸展。对于没有边框的图像精灵，整个图像精灵被平铺。
- 填充 **Filled** - 以与 **Simple** 相同的方式显示图像精灵，但是从一个远点开发，以定义好的方向、方法和数量，填充图像精灵。

当选中 **Simple** 或 **Filled** 时，显示选项 **Set Native Size**，重置图像为原始大小。

导入图像时，为选项 **Texture Type** 选择 **Sprite(2D / UI)**，图像将作为 **UI** 精灵导入。相对于旧版的 **GUI** 精灵，**UI** 精灵具有额外的导入设置，最大的不同是增加了图像精灵编辑器。图像精灵编辑器提供了 **9-slicing** 选项，将图像分割成 9 个区域，当图像调整大小时，边角不会拉升或变形。

原始图像 **Raw Image**

图像组件 **Image** 接受一张图像精灵，而 原始图像 **Raw Image** 接受一张纹理（无边框等）。原始图像只应该在必要时使用，大多数情况下，图像更适合。

遮罩 **Mask**

遮罩 **Mask** 不是可见的 **UI** 控件，而是一种修改控件子元素外观的方法。遮罩限制（则遮盖）子元素为父元素形状。因此，如果子元素大于父元素，那么只有位于父元素中的部分将是可见的。

特效 **Effects**

视觉组件还可以具有各种简单特效，例如简单的阴影或轮廓。更多信息请参阅 [Effects](#) 参考页。

交互组件

本节介绍 UI 系统中处理交互的组件，例如鼠标或触摸事件，以及使用键盘或控制器进行的交互。

交互组件本身不可见，必须与一个或多个 [视觉组件](#) 组合，才能正确工作。

常用功能

大多数交互组件有一些共同点。它们是可选择的，这意味着它们内置支持可视化的状态转换（正常、高亮、按下、禁用），以及使用键盘或控制器导航其他可选项。这一内置功能的描述请参阅 [Selectable](#) 页。

按钮 **Button**

按钮具有 **OnClick** 事件，用于定义点击后执行的行为。



Button

有关 **Button** 组件的详细用法，请参阅 [Button](#) 页面。

开关 **Toggle**

开关 **Toggle** 含有一个复选框，用于确定 **Toggle** 当前是打开还是关闭的。当用户点击 **Toggle**，它的值被反转，并且相应的打开或关闭视觉标记。它还具有一个 **OnValueChanged** 事件，用于定义值被改变时执行的行为。



有关 **Toggle** 组件的详细用法，请参阅 [Toggle](#) 页面。

开关组 **Toggle Group**

开关组可用于对彼此互斥的一组 [开关 Toggles](#) 进行分组。同属一个组（属于同一组）的开关被约束为一次只能选中其中一个——选中其中一个自动取消所有其他开关。

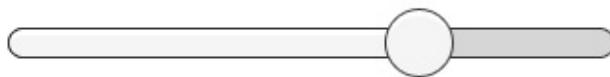
Choose a character

- Wizard
- Warrior
- Thief

有关 [Toggle Group](#) 组件的详细用法，请参阅 [Toggle Group](#) 页面。

滑块 Slider

滑块 Slider 具有一个小数 **Value**，用户可以在最小值和最大值之间拖动。它可以是水平或垂直的。它还有一个 **OnValueChanged** 事件，用于定义值被改变时执行的行为。



有关 [Slider](#) 组件的详细用法，请参阅 [Slider](#) 页。

滚动条 Scrollbar

滚动条 Scrollbar 具有一个小数 **Value**，介于 0 和 1 之间。当用户拖动滚动条时，值会相应地改变。

滚动条 Scrollbars 通常与 [滚动矩形 Scroll Rect](#) 和 [遮罩 Mask](#) 一起使用，来创建滚动视图。Scrollbar 具有一个 **Size** 值，介于 0 和 1 之间，决定了手柄相对于整个滚动条长度的大小。该值通常由其他组件控制，用于指定滚动视图中内容的可见比例。滚动矩形组件 [Scroll Rect](#) 可以自动执行该操作。

滚动条 Scrollbar 可以是水平或垂直的。它还有一个 **OnValueChanged** 事件，用于定义值被改变时执行的行为。



有关 [Scrollbar](#) 组件的详细用法，请参阅 [Scrollbar](#) 页。

下拉列表 **Dropdown**

下拉列表 **Dropdown** 包含一个可供选择的选项列表。可以为每个选项指定一段文本字符串和一张可选的图像，并且可以在检视视图中设置，或者通过代码动态设置。它有一个 **OnValueChanged** 事件，用于定义当前选中的选项被改变时执行的行为。



有关 **Dropdown** 组件的详细用法，请参阅 [Dropdown](#) 页。

输入域 **Input Field**

输入域 **Input Field** 使用用户可以编辑 [文本元素 Text Element](#) 的文本内容。它有一个 **OnValueChange** 事件，用于定义文本内容被改变时执行的行为，还有一个 **EndEdit** 事件，用于定义用户完成编辑后执行的行为。



有关 **Input Field** 组件的详细用法，请参阅 [Input Field](#) 页。

滚动矩形（滚动视图） **Scroll Rect**

当占据大量空间的内容需要显示在很小区域中时，可以使用滚动矩形 **Scroll Rect**。滚动矩形为内容提供了滚动功能。

通常，滚动矩形与 [遮罩 Mask](#) 一起使用，来创建滚动视图，只有位于滚动矩形内的滚动内容可见。它也可以结合一个或两个 [滚动条 Scrollbars](#)，通过从拖动滚动条来水平或垂直滚动内容。

ScrollView

A Scroll Rect is usually used to scroll a large image or panel of



有关 Scroll Rect 的详细用法，请参阅 [Scroll Rect](#) 页。