

# ARM® Unity 开发者指南

版本 3.2

优化移动游戏图形

**ARM**

## ARM® Unity 开发者指南

### 优化移动游戏图形

版权所有 © 2014-2016 ARM。保留所有权利。

#### 版本信息

#### 文档历史

发行号	日期	机密性	变更
0100-00	2014 年 9 月 5 日	非机密	首次发布版本 1.0
0200-00	2015 年 6 月 23 日	非机密	首次发布版本 2.0
0201-00	2015 年 7 月 28 日	非机密	首次发布版本 2.1
0300-00	2015 年 9 月 18 日	非机密	首次发布版本 3.0
0300-01	2015 年 11 月 5 日	非机密	第二次发布版本 3.0
0301-00	2016 年 4 月 07 日	非机密	首次发布版本 3.1
0301-01	2016 年 4 月 25 日	非机密	第二次发布版本 3.1
0301-01	2016 年 4 月 25 日	非机密	第三次发布版本 3.1
0302-00	2016 年 5 月 31 日	非机密	首次发布版本 3.2

#### 非机密性所有权通告

本文受版权和其他相关权利的保护，并且本文所含信息的运用或实施可能受一个或多个专利或待审批专利应用的保护。未经 ARM 明确书面许可，不得以任何形式或目的复制本文信息。**除非特别声明，否则本文未以禁止反言或其他方式明示或默示地授予任何知识产权许可。**

您必须事先确保不会使用或允许他人使用本文信息来确认实施是否侵犯任何第三方专利，才能查看本文信息。

本文按“原样”提供。就本文档而言，ARM 不做任何明示、默示或法定的陈述和保证，包括但不限于对适销性、质量满意度、非侵权性或特定用途适用性的默示保证。为避免疑义，ARM 对有关确认或理解第三方专利、版权、商业机密或其他权利的范围和内容不做任何陈述或进行任何分析。

本文档可能包含技术性错误或印刷错误。

在不违反法律的情况下，对于因使用本文档而造成的损害，包括但不限于直接、间接、特殊、意外、惩罚性或附带损害，ARM 概不负责，无论以何种方式导致并且基于任何责任理论，即使 ARM 曾被告知此类损害的可能性。

本文只包含商业项目。您应该负责确保本文的任何使用、复制或披露完全遵循任何相关出口法律法规，从而确保本文或本文的任何部分，不会违反此类出口法律，而直接或者间接的出口。使用“合作伙伴”一词表示 ARM 客户不能创建或指代与任何其他公司的合作伙伴关系。ARM 可随时更改本文档，无需另行通知。

如果上述条款包含的任何规定违反了与 ARM 书面签署的协议（含本文档）的规定，则以书面签署的协议为准，包括与这些条款相冲突的规定。为了方便起见，可能会将本文档翻译成其他语言。如果本文档的英文版本与翻译版本之间出现任何冲突，应以英文版本的协议规定的条款为准。

标有 ® 或 ™ 的词语和徽标是 ARM Limited 或其子公司在欧盟和/其他国家/地区的注册商标或商标。保留所有权利。本文档提及的其他品牌和名称可能是其各自所有者的商标。请遵循 ARM 商标使用指南，其网址为 <http://www.arm.com/about/trademark-usage-guidelines.php>

版权所有 © [2014-2016]，ARM Limited 或其子公司。保留所有权利。

ARM Limited。本公司在英国注册，注册号为 02557590。

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

**保密状态**

本文档是非机密文档。根据 ARM 与本文档的接收方之间协议的条款，使用、复制及披露本文档的权利受到许可限制。

不受限访问属于 ARM 内部分类。

**产品状态**

本文档中的信息是最终版，即用于开发完成的产品。

**网址**

<http://www.arm.com>

# 目录

## ARM® Unity 开发者移动游戏图形优化指南

<b>第 1 章</b>	<b>前言</b>	
	关于本书 .....	7
	反馈 .....	9
<b>第 1 章</b>	<b>简介</b>	
	1.1 关于 Unity .....	1-11
	1.2 关于 Mali™ GPU .....	1-12
	1.3 关于 Unity 优化 .....	1-13
	1.4 关于冰穴演示 .....	1-14
<b>第 2 章</b>	<b>优化应用程序</b>	
	2.1 优化流程 .....	2-16
	2.2 Unity 质量设置 .....	2-17
<b>第 3 章</b>	<b>分析您的应用程序</b>	
	3.1 关于分析 .....	3-21
	3.2 使用 Unity 分析器分析 .....	3-22
	3.3 Unity Frame Debugger .....	3-24
<b>第 4 章</b>	<b>优化列表</b>	
	4.1 应用处理器优化 .....	4-27
	4.2 GPU 优化 .....	4-33
	4.3 资源优化 .....	4-53
	4.4 使用 Mali™ Offline Shader Compiler 优化 .....	4-55
<b>第 5 章</b>	<b>Unity 中利用 Enlighten 的全局照明</b>	
	5.1 关于 Enlighten .....	5-61
	5.2 Enlighten 的结构 .....	5-62

5.3 使用 <i>Enlighten</i> 设置场景.....	5-71
5.4 示例：冰穴演示的 <i>Enlighten</i> 设置.....	5-73
5.5 在自定义着色器中使用 <i>Enlighten</i> .....	5-80

## 第6章

<b>高级图形技术</b>	
6.1 自定义着色器 .....	6-85
6.2 使用局部立方体贴图实施反射 .....	6-98
6.3 组合反射 .....	6-114
6.4 基于局部立方体贴图的动态软阴影 .....	6-120
6.5 基于局部立方体贴图的折射 .....	6-128
6.6 冰穴演示中的镜面反射效果 .....	6-134
6.7 使用 Early-z .....	6-137
6.8 脏镜头光晕效果 .....	6-138
6.9 光柱 .....	6-141
6.10 雾化效果 .....	6-145
6.11 高光溢出 .....	6-152
6.12 冰墙效果 .....	6-159
6.13 过程天空盒 .....	6-165
6.14 萤火虫 .....	6-173
6.15 正切空间至世界空间法线转换工具 .....	6-177

## 第7章

<b>虚拟现实</b>	
7.1 Unity 虚拟现实硬件支持 .....	7-185
7.2 Unity VR 移植流程 .....	7-186
7.3 移植到 VR 时需要考虑的问题 .....	7-189
7.4 VR 中的反射 .....	7-191
7.5 结果 .....	7-196

## 附录A

<b>修订</b>	
A.1 修订 .....	Appx-A-198

# 前言

此前言介绍了《*ARM® Unity 开发者移动游戏图形优化指南*》。

它包含下列内容：

- [关于本书](#)（第 7 页）。
- [反馈](#)（第 9 页）。

## 关于本书

本书旨在帮助您创建能够在移动平台（尤其是采用 Mali™ GPU 的移动平台）上最大程度利用 Unity 的应用程序和内容。

## 产品修改状态

*rmpn* 标识符指示本书所述产品的修改状态，例如 r1p2，其中：

*rm* 表示产品的大修改，例如 r1。

*pn* 表示产品的小修改，例如 p2。

## 目标受众

本书适合初学者至中级开发人员。

## 使用本书

本书分为以下几个章节：

### 第 1 章 简介

本章节介绍了 ARM® Unity 指南：优化您的移动游戏

### 第 2 章 优化应用程序

本章节描述了如何优化 Unity 中的应用程序。

### 第 3 章 分析您的应用程序

本章节描述了如何分析您的应用程序。

### 第 4 章 优化列表

本章节列出了许多 Unity 应用程序优化实例。

### 第 5 章 Unity 中利用 Enlighten 的全局照明

本章节描述了 Unity 中利用 Enlighten 的全局照明。

### 第 6 章 高级图形技术

本章节列出了您可以利用的多种高级图形技术。

### 第 7 章 虚拟现实

本章节描述了调整应用程序或游戏以便在虚拟现实硬件上运行的流程，同时介绍了虚拟现实中实施反射的一些区别。

### 附录 A 修订

此附录介绍本书发布的版本之间的变化。

## 词汇表

ARM 词汇表列出了在 ARM 文档中使用的一系列术语及其定义。ARM 词汇表不包含行业标准术语，除非 ARM 指示的含义不同于一般公认的含义。

有关更多信息，请参阅 [ARM 词汇表](#)。

## 书写规范

### 斜体

介绍特殊术语，表示交叉参考和引用。

### 黑体

突出显示界面元素，如菜单名称。表示符号名称。还可用于描述性列表中的词汇（如适用）。

**等宽字体**

表示可通过键盘输入的文本，例如命令、文件名、程序名以及源代码。

**等宽字体**

表示命令或选项的缩写。您可用下划线文本代替完整命令或选项名称。

**等宽斜体**

表示等宽文本的参数，其中的参数将由特定值替换。

**等宽黑体**

表示在示例代码外使用时的语言关键词。

**<and>**

封闭汇编程序语法的可替换词语，这些词语出现在代码或代码片段中。例如：

`MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`

**小型大写字母**

在正文本中用于表示具有特定技术含义的一些术语，已在 **ARM** 词汇表中定义。例如，**IMPLEMENTATION DEFINED**、**IMPLEMENTATION SPECIFIC**、**UNKNOWN** 和 **UNPREDICTABLE**。

## 补充阅读资料

**ARM** 和第三方发布的信息。

请访问 <http://infocenter.arm.com>，获取 **ARM** 文档。

### ARM 出版物

本书包含的信息仅适用于本产品。请参阅下列文档，获取其他相关信息：  
无。

开发人员资源：

<http://malideveloper.arm.com>.

### 其他出版物

第三方出版的相关文档：

*OpenGL ES 1.1 规范*，网址为 <http://www.khronos.org>。

*OpenGL ES 2.0 规范*，网址为 <http://www.khronos.org>。

*OpenGL ES 3.0 规范*，网址为 <http://www.khronos.org>。

*OpenGL ES 3.1 规范*，网址为 <http://www.khronos.org>。

*Unity 脚本参考 (Unity)*。

*GPU 精粹：实时图形编程的技术、技巧和技艺*，作者：Randima Fernando（丛书编辑）。

*GPU Pro：高级渲染技术*，作者：Wolfgang Engel（编辑）。

<https://developer.oculus.com/osig>.

## 反馈

### 产品反馈

如果您对本产品有任何意见或建议, 请联系您的供应商并提供:

- 产品名称。
- 产品修改状态或版本。
- 解释说明 (应包含尽可能多的信息)。包括症状和诊断程序 (如果适用)。

### 内容反馈

如果您对本文内容有任何意见, 请发送电子邮件到 [errata@arm.com](mailto:errata@arm.com)。提供:

- 书名《ARM® Unity 开发者移动游戏图形优化指南》。
- 编号 ARM 100140\_0302\_00\_en。
- 您有意见的页码 (如果适用)。
- 您意见的简单说明。

ARM 还欢迎您对新增和改进之处提出一般建议。

#### ——备注——

ARM 仅在 Adobe Acrobat 和 Acrobat Reader 中测试了本 PDF, 无法保证使用任何其他 PDF 阅读器时显示文档的质量。

# 第 1 章

## 简介

本章节介绍了 ARM® Unity 指南：优化您的移动游戏

它包含下列部分：

- [1.1 关于 Unity \(第 1-11 页\)](#)。
- [1.2 关于 Mali™ GPU \(第 1-12 页\)](#)。
- [1.3 关于 Unity 优化 \(第 1-13 页\)](#)。
- [1.4 关于冰穴演示 \(第 1-14 页\)](#)。

## 1.1 关于 Unity

Unity 是一个软件平台，能够让您创建和发布 2D 游戏、3D 游戏以及其他应用程序。

本书旨在帮助您创建能够在移动平台（尤其是采用 Mali™ GPU 的移动平台）上最大程度利用 Unity 的应用程序和内容。它描述了可用于提高应用程序性能的技术和最佳实践。

### 备注

除非另有说明，否则本书所述技术还可在其他平台上使用。

## 1.2 关于 Mali™ GPU

Mali GPU 专为移动或嵌入式设备而设计。Mali GPU 分为以下系列：

### Mali Utgard GPU 系列

Mali Utgard GPU 系列拥有一个顶点处理器和一个或多个片段处理器。它们仅用于使用 OpenGL ES

1.1 和 2.0 的图形应用程序。Mali Utgard 系列包含下列 Mali GPU:

- Mali-300。
- Mali-400 MP。
- Mali-450 MP。

### Mali Midgard GPU 系列

Mali Midgard GPU 系列拥有执行顶点、片段和计算处理的统一着色器内核。它们用于使用 OpenGL ES 1.1 至 OpenGL ES 3.1 以及 OpenCL 1.1 Full Profile 的图形和计算应用程序。

Mali Midgard 系列包含下列 Mali GPU:

- Mali-T604。
- Mali-T622。
- Mali-T624。
- Mali-T628。
- Mali-T720。
- Mali-T760。
- Mali-T820。
- Mali-T830。
- Mali-T860。
- Mali-T880。

## 1.3 关于Unity优化

图形可以使事物更美观。优化是指用最少的计算量使事物更美观。对于通过限制计算功率和内存带宽来保持较低功耗的移动设备而言，这尤为重要。

## 1.4 关于冰穴演示

冰穴演示是由 ARM 创建的一款演示应用程序，它利用多种优化技术生成适用于移动设备的高质量视觉内容。

本文描述了在冰穴演示中使用的图形技术，以及针对项目开发过程中所遇问题的解决方案。

冰穴演示面向包含下列组件的移动设备：

ARM Cortex<sup>®</sup>-A57 MP4 处理器

ARM Cortex-A53 MP4 处理器

ARM Mali-T760 MP8 GPU

## 第 2 章

# 优化应用程序

本章节描述了如何优化 Unity 中的应用程序。

它包含下列部分：

- [2.1 优化流程（第 2-16 页）。](#)
- [2.2 Unity 质量设置（第 2-17 页）。](#)

## 2.1 优化流程

优化是指选择一个应用程序并使其更有效率的过程。对于图形应用程序，这通常是指对应用程序进行修改，使其运行更快。

例如，低帧率的游戏意味着其画面十分跳跃。这会给用户留下不好的印象，并且使游戏很难玩下去。您可以利用优化提高游戏的帧率，带来更好、更流畅的体验。

若要优化您的代码，请使用优化流程。此流程为迭代流程，可指导您找出性能问题并予以解决。

优化流程包含下列步骤：

1. 使用分析器测量应用程序。
2. 分析数据，找到瓶颈。
3. 确定要运用的相关优化。
4. 验证优化是否成功。
5. 如果性能不可接受，请返回至第 1 步，然后重复该流程。

以下是优化流程的一个示例：

1. 如果一款游戏未达到您需要的性能，可以使用 **Unity** 分析器进行测量。
2. 使用 **Unity** 分析器分析测量结果，以便您可以隔离并确认性能问题的来源。
3. 比如，游戏的问题在于渲染过多的顶点。
4. 减少您的代码中的顶点数量。
5. 再次运行游戏，确保优化发挥了作用。

如果执行完此操作后游戏仍未按照预期运行，请重新开始执行此流程，方法是再次分析应用程序以找出引发此问题的其他原因。

最好重复执行此流程多次。优化是一项迭代流程，您可以通过该流程找到大量不同区域出现的性能问题。

## 2.2 Unity 质量设置

了解 Unity 质量设置非常有用，可以确保您为应用程序选择了正确的设置。

Unity 包含许多可以改变游戏图像质量的选项。部分选项计算量较高，会对游戏性能产生不利影响。

下图显示了检视面板中的质量设置：

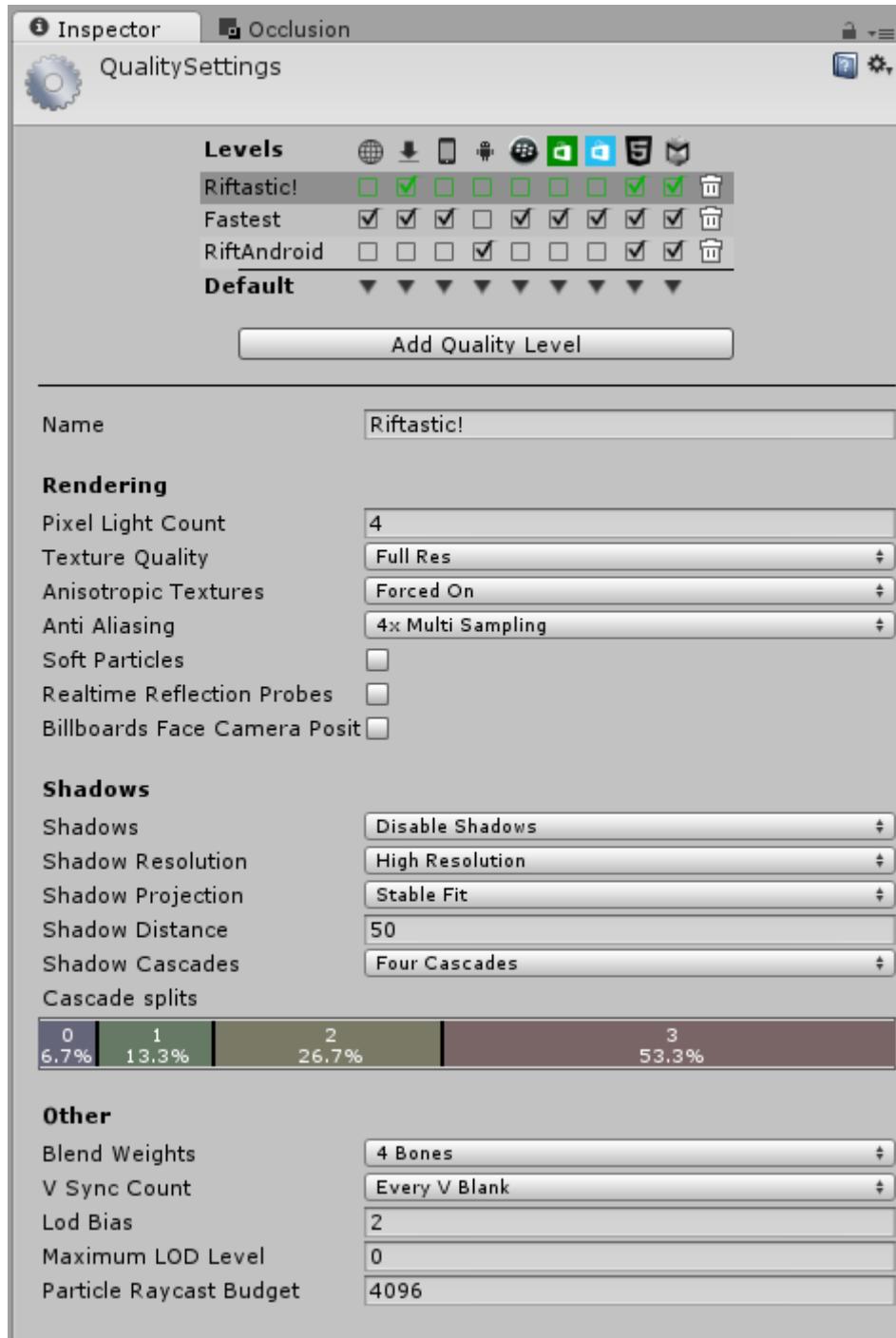


图 2-1 质量设置

有许多选项可以提高游戏图像质量，代价只是极小幅度地降低性能。例如，如果游戏的帧率较低，GPU 可能正在处理过多执行复杂图形效果的信息。您可以执行较简单版本的图形效果，例如阴影或光照。相比而言，这对图形质量产生的影响较小。更简单的效果可以显著降低 GPU 负载，从而提供更高的帧率。

光照默认设置有时对于移动设备而言过于复杂，因此针对移动平台编写的部分游戏应避免复杂的技术，或者根据每个游戏使用相应的技术。这可能涉及将光照预烘焙为光照贴图的技术，或者用投影纹理代替投射阴影的技术。

在**项目设置 > 质量**中，有许多可以对游戏性能产生巨大影响的选项：

#### 像素光源数量

**像素光源数量**是指可以影响给定像素的光源数量。较高的像素光源数量需要执行大量计算。大部分游戏能够在使用极少量动态实时光源的同时，最大限度降低对质量的影响。如果光照引发了性能问题，请考虑在游戏中使用光照贴图和投影纹理等技术。

#### 纹理质量

**纹理质量**可以给 GPU 带来负载，但通常不会引发性能问题。降低纹理质量会对游戏的视觉质量产生不良影响，所以请只在必要的情况下降低纹理质量。在冰穴演示中，**纹理质量**设为全分辨率。

如果纹理引发了性能问题，可尝试使用纹理映射。纹理映射可降低计算和带宽要求，同时不会影响图像质量。

#### 抗锯齿

**抗锯齿**是一项边缘平滑技术，该技术混合了三角形边周围的像素。这显著提高了游戏的视觉质量。有多种抗锯齿方式，但是此种情况下采用的是**多重采样抗锯齿 (MSAA)**。**4x MSAA** 在 Mali GPU 上的运算量较低，应尽可能使用。

#### 软粒子

**软粒子**需要渲染到深度纹理或在延迟模式下渲染。这会提高 GPU 负载，但可以获得逼真的粒子效果，因此值得采用。在移动平台上，渲染到深度纹理和从深度纹理读取会消耗掉宝贵的带宽，并且使用延迟路径进行渲染意味着您不能使用 MASS。请考虑软粒子是否足够重要到需要在游戏中使用。

#### 各向异性纹理

**各向异性纹理**技术可消除在高梯度下绘制的纹理的失真。这项技术可提高图像质量，但是非常消耗资源。除非失真特别明显，否则请避免使用此技术。

#### 阴影

**阴影**具有高质量时，计算量较大。如果阴影引发了性能问题，请尝试采用简单的阴影或关闭阴影。如果阴影对于您的游戏非常重要，请考虑使用简单的动态阴影技术，例如投影纹理。

#### 实时反射探测器

**实时反射探测器**选项对运行时性能存在显著的负面影响。

在渲染反射探测器时，立方体贴图的每一面都由探测器原点处的摄像机进行单独渲染。如果考虑相互反射，此过程会在每个反射反弹级别进行一次。在光泽反射情形中，立方体贴图纹理映射也用于应用模糊处理。

下列因素影响立方体贴图的渲染：

#### 立方体贴图分辨率

立方体贴图分辨率越高，渲染时间就越长。为您需要的质量，尽可能使用最低分辨率的立方体贴图。

### 剔除遮罩

在渲染立方体贴图时使用剔除遮罩，从而避免对反射中任何无关几何体进行渲染。

### 立方体贴图更新

**刷新模式**选项定义立方体贴图的更新频率：

- **每帧**选项针对每一帧渲染立方体贴图。这是计算成本最高的选项，所以非必要时请勿使用。
- **唤醒时**选项在场景启动时进行一次运行时立方体贴图渲染。
- **借助脚本**选项让您能控制立方体贴图更新的时间。使用此选项时，您可以通过指定发生更新的条件来限制对运行时资源的使用。

# 第 3 章

## 分析您的应用程序

本章节描述了如何分析您的应用程序。

它包含下列部分：

- [3.1 关于分析 \(第 3-21 页\)](#)。
- [3.2 使用 Unity 分析器分析 \(第 3-22 页\)](#)。
- [3.3 Unity Frame Debugger \(第 3-24 页\)](#)。

## 3.1 关于分析

分析应用程序可找出性能瓶颈。当您确定这些瓶颈后，针对这些区域进行优化可提高应用程序性能。

您可以使用下列工具分析 Unity 应用程序：

- Unity 分析器。
- Unity Frame Debugger。
- ARM Mali Graphics Debugger。
- ARM DS-5 Streamline。

## 3.2 使用 Unity 分析器分析

Unity 分析器以一系列图表的形式提供详细的每帧性能数据，帮助您查找游戏中的瓶颈。

如果您点击一个图表，就会看到垂直切片，并且选择了某个单帧。您可以在屏幕底部的显示面板中读取该帧的信息。如果您在没有修改所选帧的情况下点击另一图表，面板将显示您已选择的分析器的数据。

下图显示了 Unity 分析器：

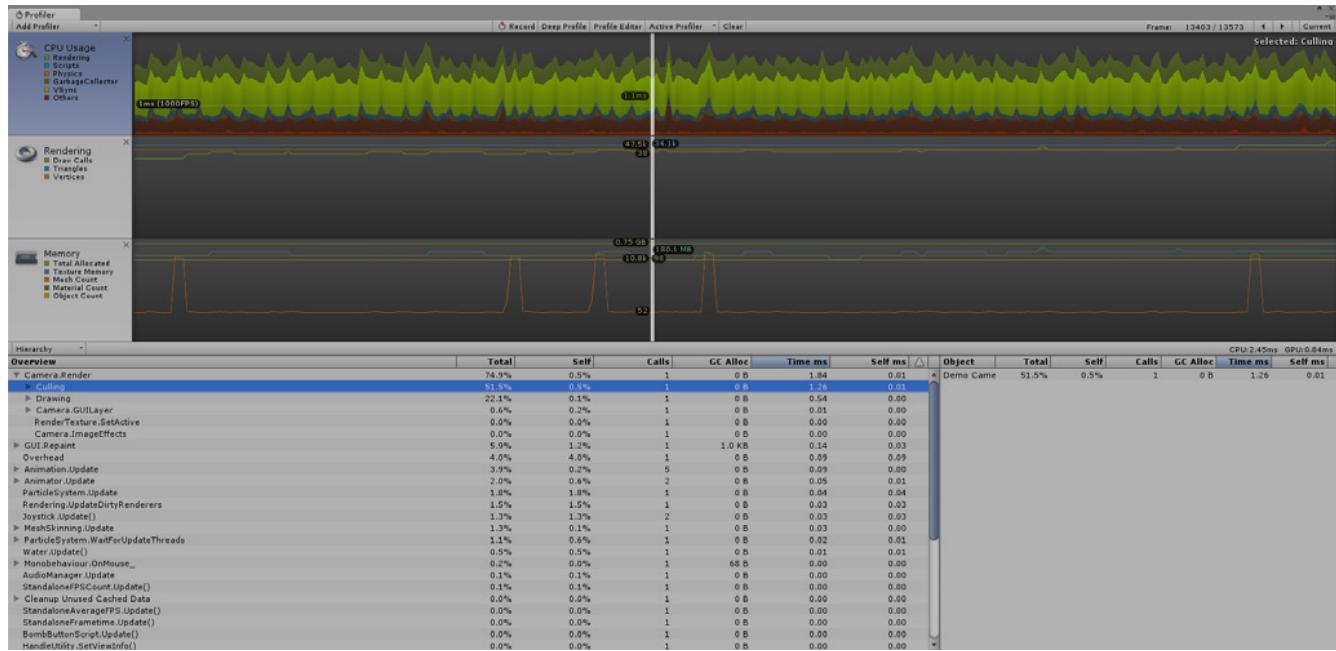


图 3-1 Unity 分析器

Unity 分析器提供下列功能：

### CPU 使用率分析器

CPU 使用率分析器图表显示了 CPU 使用率的细分情况，重点突出不同组件（如渲染、脚本或物理）的 CPU 使用情况。如果您在图表上选择了某帧，则面板将显示对该帧影响最大的功能的耗费时间、调用数量或内存分配。

请重点关注耗费更多执行时间或分配过多内存的功能。

### 备注

在多处理器系统中，这些值为平均值。

### 渲染分析器

渲染分析器图表显示了绘制调用、三角形以及在场景中渲染的顶点的数量。在图表上选择某帧将显示更多有关批处理、纹理和内存消耗的信息。

请仔细查看绘制调用、三角形以及场景渲染的顶点的数量。这些是移动平台上最为重要的数字。

### 内存分析器

内存分析器图表显示了分配的内存数量以及游戏所用资源（如网格或材质）的数量。在图表上选择某帧后，将显示资源、图形和音频子系统或分析器数据本身的内存消耗情况。

移动平台上的内存有限，因此您必须在其生存期内监控游戏需求，并检查占用资源的数量。某些技术如果运用不恰当，会创建大量新对象。例如，不恰当地运用纹理贴图集会创建大量新材质对象。

### 添加分析器功能

**添加分析器**选项位于分析器窗口左上角的下拉菜单中。使用该选项能够向分析器窗口添加更多图表，例如 CPU 使用率、渲染或内存。

### Profiler.BeginSample() 和 Profiler.EndSample() 方法

Unity 分析器可让您采用 Profiler.BeginSample() 和 Profiler.EndSample() 方法。您可以在脚本中标记一个区域，然后附上自定义标签，此区域将作为单独的条目出现在分析器层级中。通过执行此操作，您可以获取特定代码的信息，而无需采用深度分析选项，从而节省计算和内存开销。

```
void Update()
{
    Profiler.BeginSample("ProfiledSection");
    [...]
    Profiler.EndSample();
}
```

下图显示了被分析部分：

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
WaitForTargetFPS	95.7%	95.7%	1	0 B	15.06	15.06
▼ ProfiledSectionTest.Update()	2.1%	0.0%	1	0 B	0.33	0.00
ProfiledSection	2.1%	2.1%	1	0 B	0.33	0.33

图 3-2 被分析部分

### 3.3 Unity Frame Debugger

Frame Debugger 是一款分析工具，您可以在每一帧基础上追踪绘制调用。

Frame Debugger 可以从窗口菜单中选用。

左窗格中显示该帧中发出的绘制调用树。

右窗格中显示与选定绘制调用相关的其他信息，如几何体详情以及绘制它的着色器等。

下图显示了 Frame Debugger 中的 Phoenix 对象：

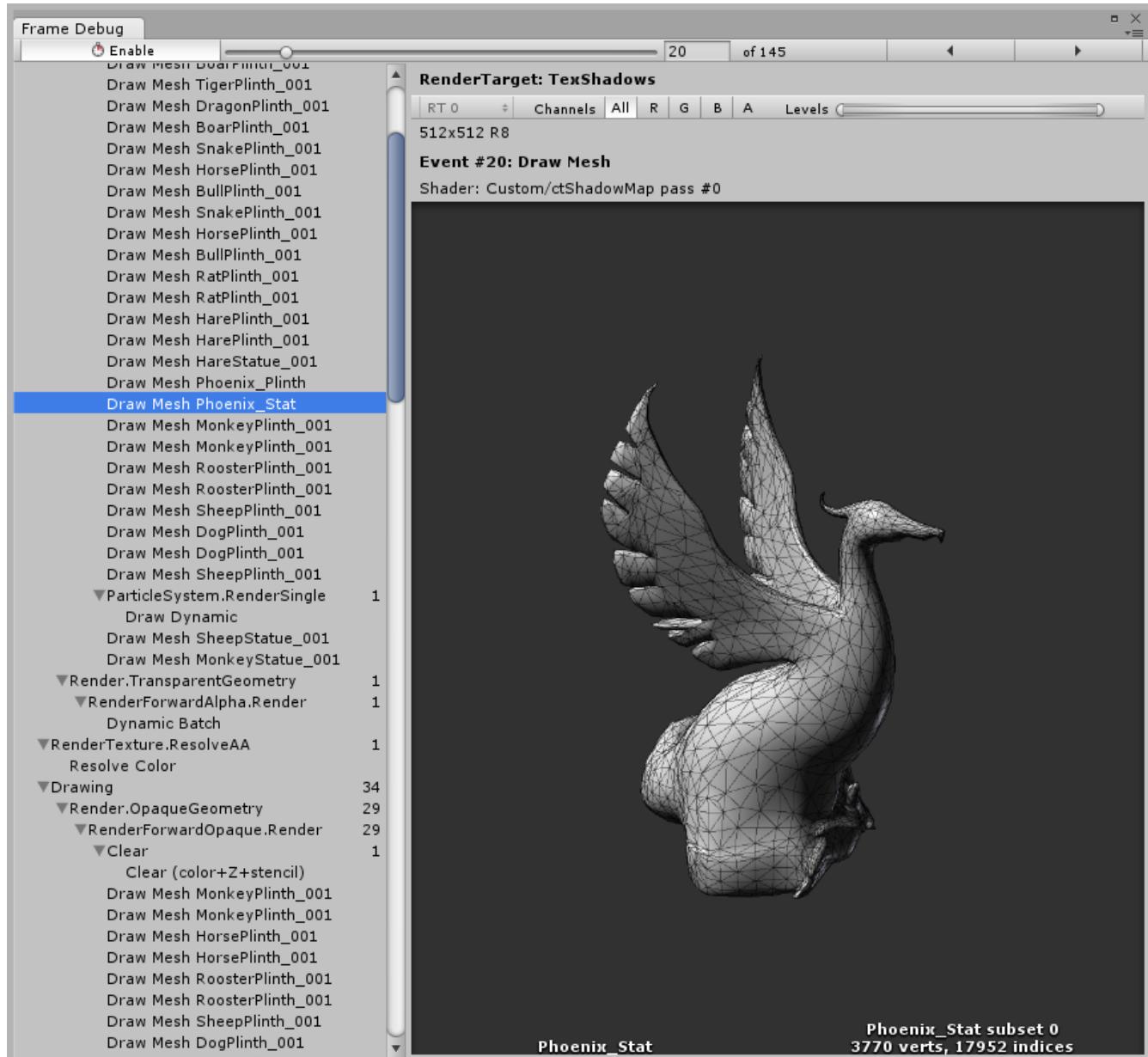


图 3-3 Frame Debugger

如果摄像机渲染到所选绘制调用的目标，您可以在游戏视图中查看被渲染纹理的视觉呈现。

下图显示了游戏视图中的 Phoenix 对象：

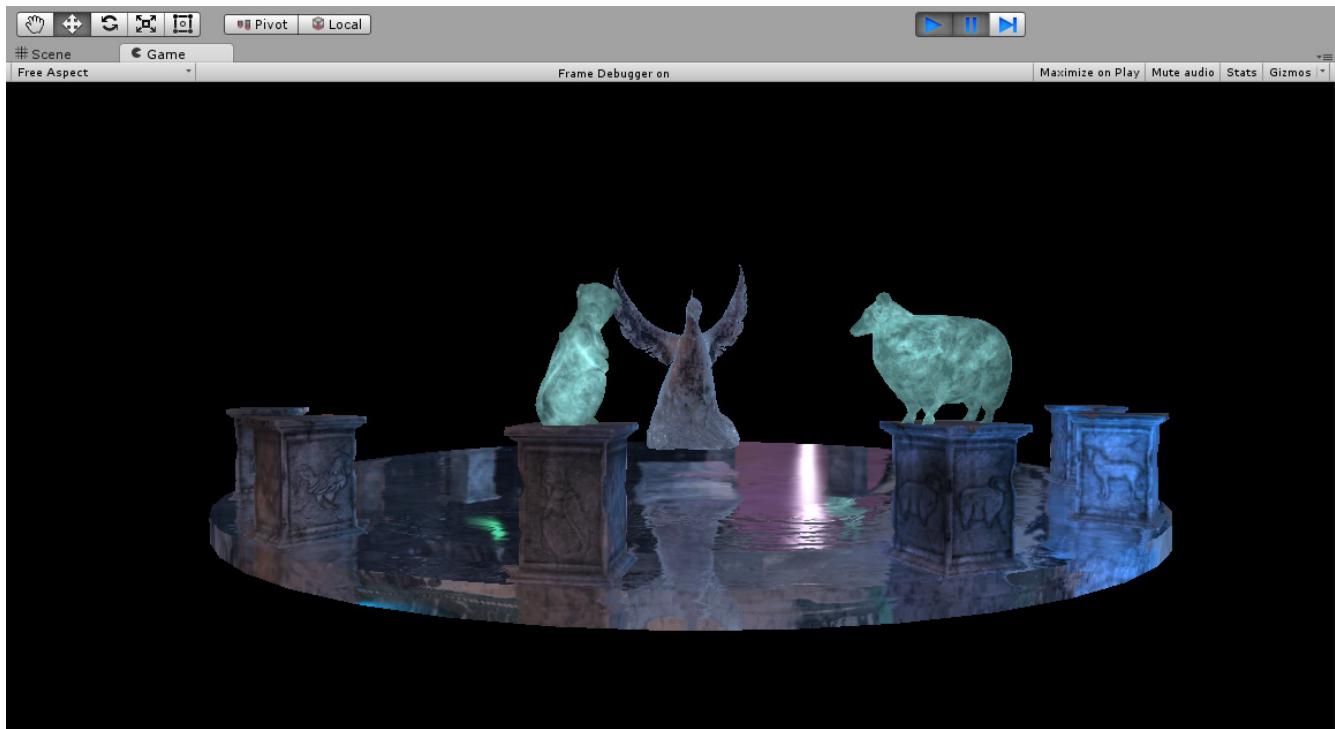


图 3-4 Frame Debugger 游戏视图

如需关于如何利用 Frame Debugger 帮助优化对象渲染顺序以提升游戏性能的说明，请参见 [4.2.9 指定渲染顺序 \(第 4-50 页\)](#)。

# 第 4 章

## 优化列表

本章节列出了许多 Unity 应用程序优化实例。

它包含下列部分：

- [4.1 应用处理器优化（第 4-27 页）。](#)
- [4.2 GPU 优化（第 4-33 页）。](#)
- [4.3 资源优化（第 4-53 页）。](#)
- [4.4 使用 Mali™ Offline Shader Compiler 优化（第 4-55 页）。](#)

## 4.1 应用处理器优化

下表描述了应用处理器优化：

### 使用协程代替 Invoke()

`Monobehaviour.Invoke()` 方法可快速便捷地在时间延迟的情况下调用类中的方法，但存在以下局限性：

- 它使用 C# 反射查找调用方法，这比直接调用方法的速度更慢。
- 没有针对方法签名的编译时检查。
- 您无法提供附加参数。

下列代码显示了 `Invoke()` 函数：

```
public void Function()
{
    [...]
}
Invoke("Function", 3.0f);
```

替代方法是使用协程。协程是 `IEnumerator` 类型的函数，可以使用特殊 `yield return` 语句将控制权归还给 Unity。您可以稍后再次调用函数，它将从之前中断的位置恢复运行。

您可以通过 `MonoBehaviour.StartCoroutine()` 方法调用协程：

```
public IEnumerator Function(float delay)
{
    yield return new WaitForSeconds(delay);
    [...]
}
StartCoroutine(Function(3.0f));
```

从 `Monobehaviour.Invoke()` 方法转为使用协程，可以在传递至处理动画状态的函数的参数上提供更高的灵活性。

### 使用协程轻松进行更新

如果您的游戏需要每隔一段特定时间执行操作，请尝试在 `MonoBehaviour.Start()` 回调函数中启动协程，以此代替 `MonoBehaviour.Update()` 回调函数（每帧都要执行操作）。例如：

```
void Update()
{
    // Perform an action every frame
}

IEnumerator Start()
{
    while(true)
    {
        // Do something every quarter of second
        yield return new WaitForSeconds(0.25f);
    }
}
```

#### 备注

此技术的另一可选用途是以不规则的间隔产生敌人。利用协程中的无线循环大量产生敌人，并生成随机数字。将随机数传递到 `WaitForSeconds()` 函数。

### 使标记避免硬编码字符串

由于标记的硬编码值会限制游戏的可扩展性和鲁棒性，因此应避免使用。例如，如果您直接通过字符串引用名称，则无法轻松修改标记名称，并且很可能会出现拼写错误。例如：

```
if(gameObject.CompareTag("Player"))
{
    [...]
}
```

您可以为显示公共常量字符串的标记执行一个特殊类，从而改善这一情况。例如：

```
public class Tags
{
    public const string Player = "Player";
    [...]
}

if(gameObject.CompareTag(Tags.Player))
{
    [...]
}
```

#### ——备注——

您可以采用带有公共常量字符串的标记类，以一致且可扩展的方式添加新标记。

### 通过更改固定时间步长减少物理计算量

您可以通过更改固定时间步长降低物理计算的计算量。通常，大部分物理计算发生在固定时间步长内，您可以增加或减小此步长。

增加时间步长会减小应用程序处理器上的负载，但是会降低物理计算的准确性。

您可以从主菜单访问时间管理器：**编辑 > 项目设置 > 时间**。

下图显示了时间管理器：

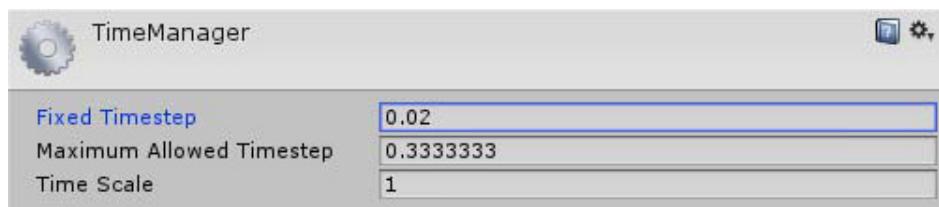


图 4-1 固定时间步长设置

### 删除空回调函数

如果您的代码包含 `Awake()`、`Start()` 或 `Update()` 等函数的空定义，则将其删除。这会产生不必要的开销，因为引擎在函数为空时仍会尝试访问它们。例如：

```
// Remove the following empty definition

void Awake()
{
}
```

### 避免在每帧中都使用 `GameObject.Find()`。

`GameObject.Find()` 函数用于循环访问场景中的每个对象。如果它在代码中使用的位置不正确，则会导致主线程大小显著增加。例如：

```
void Update()
{
    GameObject playerGO = GameObject.Find("Player");
    playerGO.transform.Translate(Vector3.forward * Time.deltaTime);
}
```

更好的做法是在启动时调用 `GameObject.Find()` 并缓存结果，例如将结果缓存在 `Start()` 或 `Awake()` 函数中：

```
private GameObject _playerGO = null ;

void Start()
{
    _playerGO = GameObject.Find("Player");
}

void Update()
{
    _playerGO.transform.Translate(Vector3.forward * Time.deltaTime);
}
```

另一种替代方法是使用 `GameObject.FindWithTag()`：

```
void Update()
{
    GameObject playerGO = GameObject.FindWithTag("Player");
    playerGO.transform.Translate(Vector3.forward * Time.deltaTime);
}
```

#### 备注

使用称为 `LocatorManager` 的专用类，它可以在场景完成加载后立即执行所有对象检索。其他类可以使用它作为服务，以便使对象不会被检索多次。

### 使用 `StringBuilder` 类连接字符串

连接复杂字符串时,请使用 `System.Text.StringBuilder` 类。其速度远快于 `string.Format()` 方法,并且使用的内存少于通过加号运算符进行连接时所用的内存:

```
// Concatenation with the plus operator
string str = "foo" + "bar";

// String.Format() method
string str = string.Format("{1}{2}", "foo", "bar");
```

`System.Text.StringBuilder` 类:

```
// StringBuilder class
using System.Text;

StringBuilder strBld = new StringBuilder();
strBld.Append("foo");
strBld.Append("bar");
string str = strBld.ToString();
```

下图显示了字符串连接:

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ StringConcatenationTest.Start()	99.8%	0.0%	1	1.12 GB	2488.86	0.33
▼ StringConcatenationTest.StringFormatMethod()	51.0%	0.2%	1	0.56 GB	1273.01	6.78
► String.Format()	50.8%	0.3%	10000	0.56 GB	1266.22	7.51
▼ StringConcatenationTest.PlusConcatenation()	48.4%	0.2%	1	0.56 GB	1208.21	6.53
► String.Concat()	48.2%	0.5%	10000	0.56 GB	1201.68	13.15
▼ StringConcatenationTest.StringBuilderClass()	0.2%	0.2%	1	256.2 KB	7.31	6.97
► StringBuilder.Append()	0.0%	0.0%	10000	256.1 KB	0.32	0.12
► StringBuilder..ctor()	0.0%	0.0%	1	0 B	0.00	0.00
► StringBuilder.ToString()	0.0%	0.0%	1	0 B	0.01	0.00

图 4-2 字符串连接

### 使用 `CompareTag()` 方法代替标记属性

使用 `GameObject.CompareTag()` 方法代替 `GameObject.tag` 属性。`CompareTag()` 方法的速度更快,并且不会分配额外的内存:

```
GameObject mainCamera = GameObject.Find("Main Camera");

// GameObject.tag property
if(mainCamera.tag == "MainCamera")
{
    // Perform an action
}

// GameObject.CompareTag() method
if(mainCamera.CompareTag("MainCamera"))
{
    // Perform an action
}
```

下图显示了 `CompareTag()` 的用法:

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ TagComparisonTest.Start()	97.2%	0.1%	1	3.6 MB	127.48	0.26
▼ TagComparisonTest.TagProperty()	68.0%	59.6%	1	3.6 MB	89.27	78.14
▼ GameObject.get_tag()	7.9%	1.0%	100000	3.6 MB	10.44	1.32
GC.Collect	6.9%	6.9%	4	0 B	9.12	9.12
▼ String.op_Equality()	0.5%	0.3%	100000	0 B	0.69	0.50
String.Equals()	0.1%	0.1%	100000	0 B	0.18	0.18
► TagComparisonTest.CompareTagMethod()	28.9%	28.4%	1	0 B	37.94	37.27
GameObject.Find()	0.0%	0.0%	1	0 B	0.00	0.00

图 4-3 比较标记

## 使用对象池

如果您的游戏有许多同类对象在运行时创建和破坏，则可以使用设计模式对象池。该设计模式可避免在动态分配和释放众多对象时产生性能损失。

如果您知道所需对象的总数，则可以直接创建所有对象，并禁用暂时不需要的对象。需要新对象时，请搜索首个未用对象的池并启用该对象。

不再需要某个对象时，您可以将其放回至池中。这意味着将对象重置为默认开始状态并禁用该对象。此技术可以与敌人、抛射物和粒子等对象结合使用。如果您不知道所需对象的准确数目，则进行测试，找出使用的对象数并创建一个数目稍大于此数的对象池。

### 备注

将对象池用于敌人和炸弹。这使得这些对象不会被分配至游戏的加载阶段。

## 缓存组件检索

缓存 `GameObject.GetComponent<Type>()` 返回的组件实例。涉及的函数调用非常消耗性能。`GameObject.camera`、`GameObject.renderer` 或 `GameObject.transform` 等属性是对应 `GameObject.GetComponent<Camera>()`、`GameObject.GetComponent<Renderer>()` 和 `GameObject.GetComponent<Transform>()` 的快捷方式：

```
private Transform _transform = null;  
  
void Start()  
{  
    _transform = GameObject.GetComponent<Transform>();  
}  
  
void Update()  
{  
    _transform.Translate(Vector3.forward * Time.deltaTime);  
}
```

请考虑缓存 `Transform.position` 的返回值。即便它是 C# getter 属性，也会在用于计算全局位置的转换层级上产生与迭代相关的开销。

### 备注

Unity 5 及更高版本会自动缓存转换组件。

## 使用 `OnBecameVisible()` 和 `OnBecameInvisible()` 回调函数

如果与 `MonoBehaviour.OnBecameVisible()` 和 `MonoBehaviour.OnBecameInvisible()` 等回调函数相关的游戏对象出现在或消失在屏幕上，则这些回调函数将通知您的脚本。

如果某一游戏对象未在屏幕上渲染，这些调用能够让您禁用大量计算性代码例程或特效。

### 使用 `sqrMagnitude` 比较向量幅度

如果您的应用程序需要比较向量幅度，请使用 `Vector3.sqrMagnitude` 代替 `Vector3.Distance()` 或 `Vector3.magnitude`。

虽然 `Vector3.sqrMagnitude` 在不计算根的情况下计算正方形组件的总和，但是这对于比较非常有帮助。其他调用使用计算量非常大的平方根。

下列代码显示了比较空间中两个位置采用的三种不同方法：

```
// Vector3.sqrMagnitude property
if (_transform.position - targetPos).sqrMagnitude < maxDistance * maxDistance)
{
    // Perform an action
}

// Vector3.Distance() method
if (Vector3.Distance(transform.position, targetPos) < maxDistance)
{
    // Perform an action
}

// Vector3.magnitude property
if (_transform.position - targetPos).magnitude < maxDistance)
{
    // Perform an action
}
```

### 使用内置数组

如果您事先知道数组大小，则使用内置数组。

`ArrayList` 和 `List` 类的灵活性更高，它们会随插入元素的增加而增大，但是速度要比内置数组慢。

### 使用平面作为碰撞目标

如果场景只需要与平面物体（如地板或墙壁）进行粒子碰撞，则可将粒子系统碰撞模式更改为 `Planes`。将设置更改为使用平面可降低所需的计算量。在此模式下，您可以为 Unity 提供一系列空的游戏对象作为碰撞器平面。

下图显示了碰撞设置：

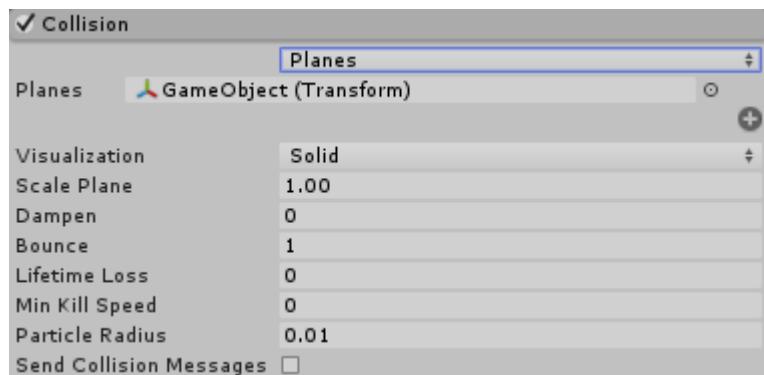


图 4-4 碰撞设置

### 使用复合的原始碰撞器（而非网格碰撞器）

网格碰撞器以对象的实际几何数据为基础。它们在碰撞检测方面具备极高的准确性，但是计算量非常大。

您可以将盒子、舱体或球体等形状合并为模拟原始网格形状的复合碰撞器。这可使您获得相似的结果，同时还可大幅降低计算开销。

## 4.2 GPU 优化

本节列出了 GPU 优化。

本节包含以下小节：

- 4.2.1 杂项 GPU 优化（第 4-33 页）。
- 4.2.2 光照贴图和灯光探测器（第 4-34 页）。
- 4.2.3 ASTC 纹理压缩（第 4-43 页）。
- 4.2.4 纹理映射（第 4-47 页）。
- 4.2.5 天空盒（第 4-48 页）。
- 4.2.6 阴影（第 4-48 页）。
- 4.2.7 遮挡剔除（第 4-49 页）。
- 4.2.8 使用 `OnBecameVisible()` 和 `OnBecomeInvisible()` 回调函数（第 4-50 页）。
- 4.2.9 指定渲染顺序（第 4-50 页）。
- 4.2.10 使用深度预传值（第 4-52 页）。

### 4.2.1 杂项 GPU 优化

下表描述了杂项 GPU 优化：

#### 使用静态批处理

静态批处理是一种常见的优化技术，可以减少绘制调用数量，从而降低应用程序处理器的使用率。

动态批处理可由 Unity 以透明的方式执行，但是无法运用至大量顶点组成的对象，因为计算开销过大。静态批处理可在大量顶点组成的对象上运行，但是经过批处理的对象在渲染过程中不得移动、旋转或扩大。

若要使 Unity 能够集合要进行静态批处理的对象，请在“检视面板”中将它们标记为静态。

下图显示了静态批处理设置：

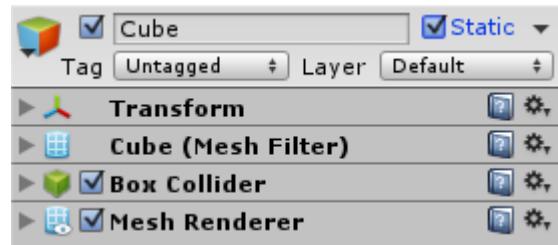


图 4-5 静态批处理设置

#### 使用 4x MSAA

ARM Mali GPU 能够以极低的计算开销执行 4x 多重采样抗锯齿。您可以在“Unity 质量设置”中启用 4x MSAA。

下图显示了 MSAA 设置：

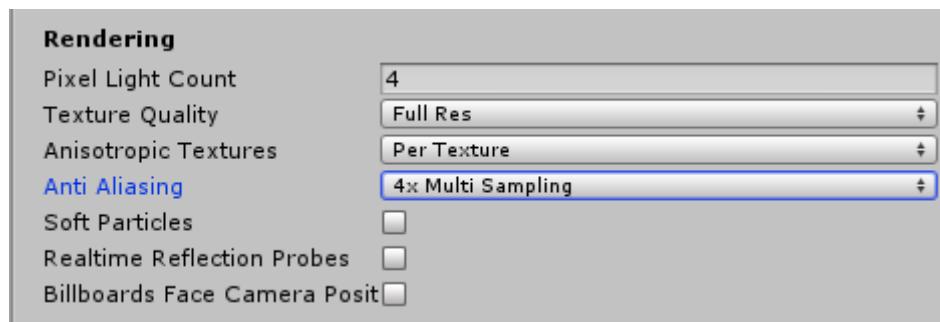


图 4-6 MSAA 设置

### 使用细节层次

Unity 引擎可以使用 **细节层次 (LOD)** 技术根据与摄像机的距离渲染同一对象的不同网格。

当对象更接近摄像机时，几何结构更为清晰。随着对象远离摄像机，细节层次降低。当处于最远距离时，您可以使用平面公告板。

您必须恰当地设置 LOD 组，以管理要使用的网格以及相应的距离范围。

要访问 LOD 组的设置，请选择：**添加组件 > 渲染 > LOD 组**。

在 Unity 5 中，您可以设置渐变模式使每个 LOD 层次混合为连续的 LOD。这可以平滑它们之间的过渡。Unity 可以根据对象的屏幕大小计算混合因子，并将它传递到着色器以进行混合。您必须在着色器中实施几何体混合。

下图显示了 LOD 组设置：

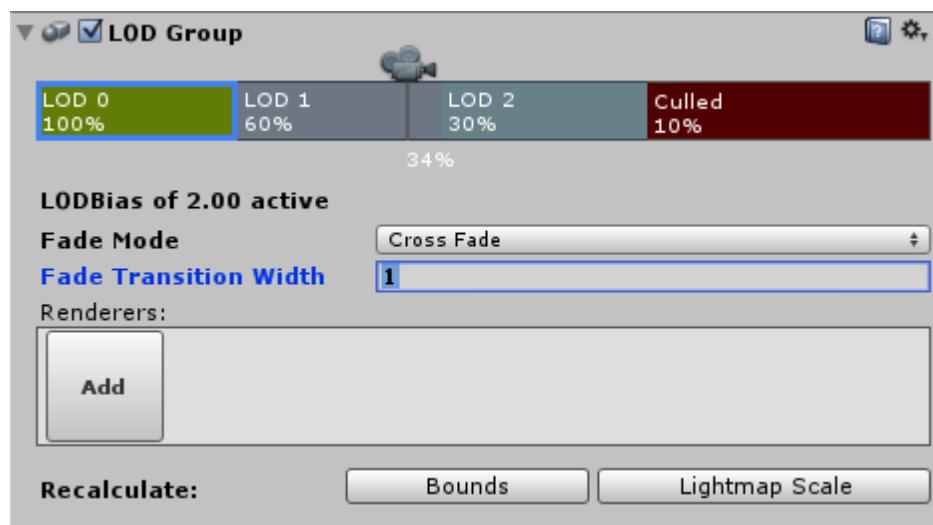


图 4-7 LOD 组设置

### 避免在自定义着色器中使用数学函数

在编写自定义着色器时，请避免使用计算量大的内置数学函数，例如：

- `pow()`.
- `exp()`.
- `log()`.
- `cos()`.
- `sin()`.
- `tan()`.

## 4.2.2 光照贴图和灯光探测器

运行时光照计算的计算成本很高。一项用于降低计算要求的常见技巧称为光照贴图，它预先进行光照计算并将它们烘焙为名为光照贴图的纹理。

这意味着您会丧失完全动态光照环境的灵活性，但您可以生成质量非常高的图像，而不会影响性能。

在静态光照贴图中烘焙生成的光照

- 将接收光照的几何体设置为**静态**。
- 将灯光中的**烘焙**选项设为**已烘焙**，而不是**运行时**。
- 在光照贴图窗口的“场景”选项卡中，选中**已烘焙 GI** 选项。

查看生成的光照贴图：

- 选择几何体。
- 选择窗口 > 光照，以打开光照窗口。
- 按对象按钮。
- 选择预览选项中的烘焙强度光照贴图。

如果选择了持续烘焙选项，Unity 将烘焙该光照贴图，并在几秒后更新编辑器中的场景。

若要快速检查光照贴图设置是否正确，可在编辑器中运行游戏，并禁用光源。如果光照仍在，则光照贴图已经正确创建并在使用中。

下图显示了光照选项卡中的强度光照贴图。

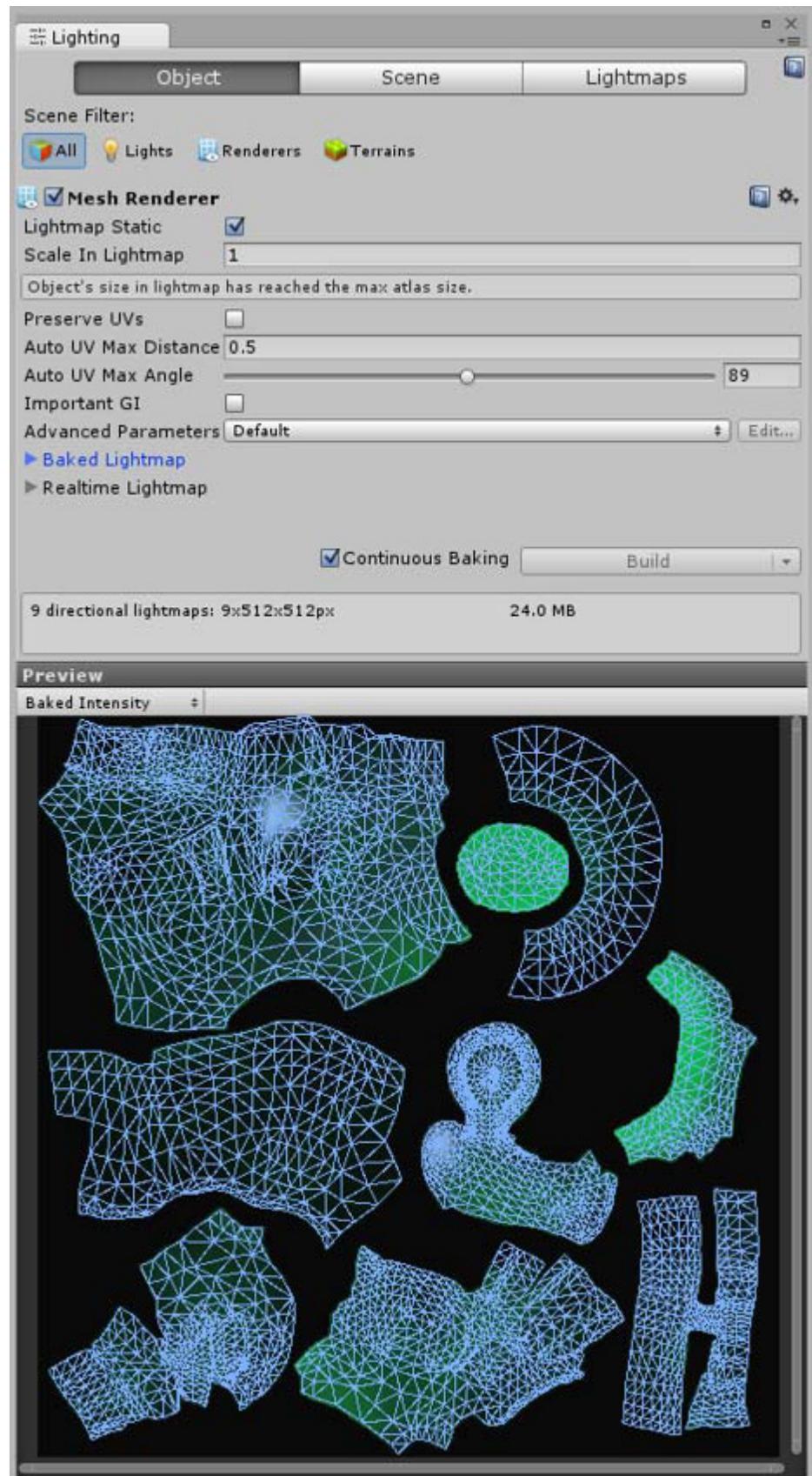


图 4-8 强度光照贴图

下图显示了编辑器中显示来自洞穴尽头绿色光源的光照。此光照使用静态光照贴图生成。

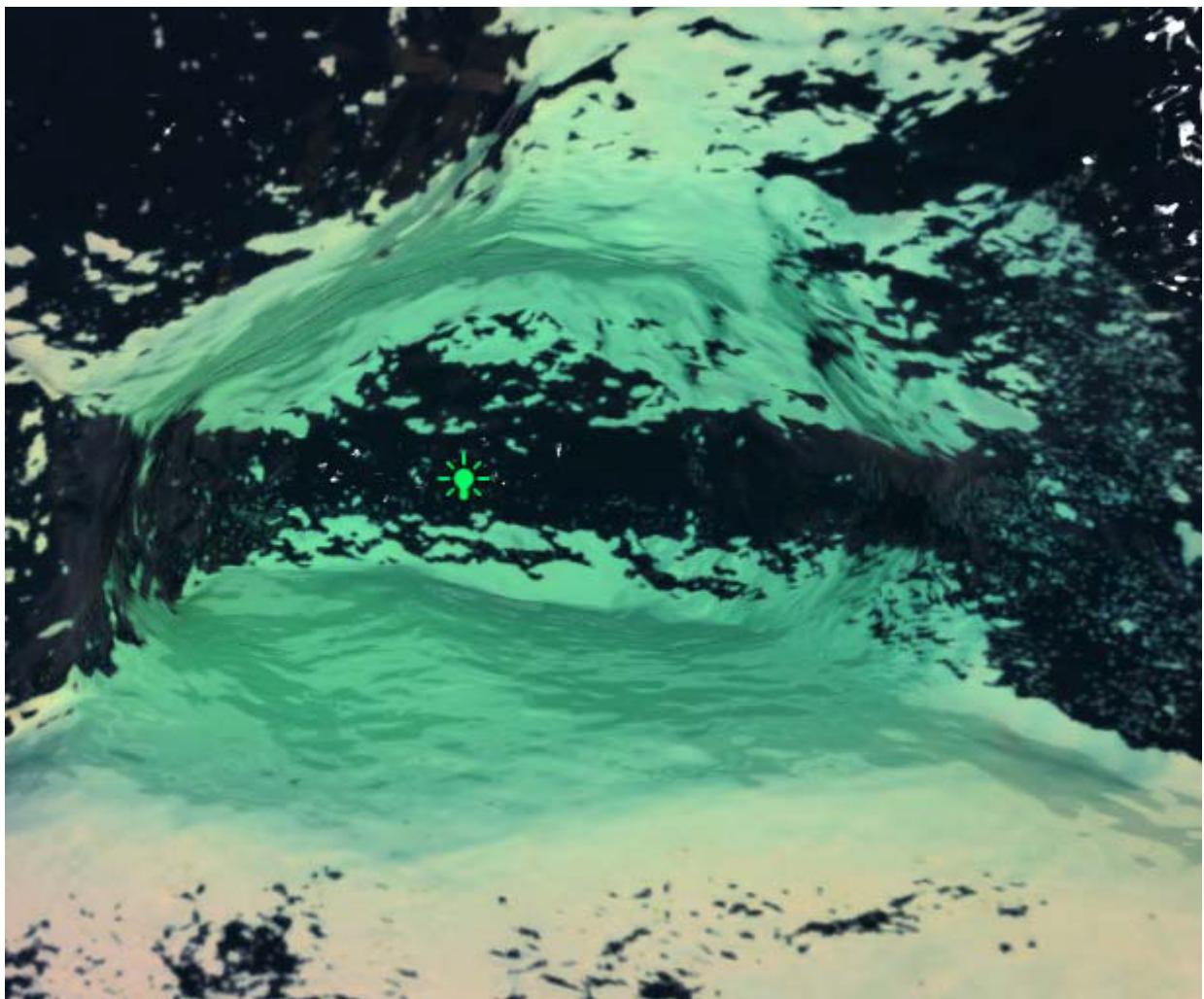


图 4-9 添加灯光以烘焙静态光照贴图

下图显示了冰穴演示中静态光照贴图的结果。



图 4-10 经过光照贴图的洞穴

### 构建光照贴图

若要为光照贴图准备对象，您必须拥有：

- 带有光照贴图 UV 的场景模型。
- 模型必须标记为静态光照贴图。
- 模型范围内必须存在光源。
- 光源的烘焙类型必须设为“已烘焙”。

#### 备注

只有场景内的静态对象已经过光照贴图。它们的效果可能不是最佳，因此请通过试验了解哪种设置最适合您的游戏。

未标记为静态的对象不会置于光照贴图中。选择渲染器可以为您提供众多设置，您可以设置光照贴图是否为静态。

从编辑器窗口的主菜单中打开光照窗口，再选择窗口和光照贴图。共有三个按钮：

- 对象。
- 场景。
- 光照贴图。

下图显示了光照贴图选项：



图 4-11 光照贴图选项

## 对象

单击**对象**按钮可更改与您在层级中所选对象的光照贴图相关的设置。借此可修改影响光照贴图流程的对象设置。选择一个光源，便可更改诸多选项：

- **仅烘焙**: 可在烘焙时启用光源并在运行时禁用光源。
- **已烘焙**: 如果选中了**已烘焙 GI**，将烘焙该光源。
- **实时**: 光源可用于预算计算的实时 GI，也可在没有 GI 时使用。
- **仅实时**: 可在烘焙时禁用光源并在运行时启用光源。
- **混合**: 光源已烘焙，但它在运行时仍然存在，为非静态对象提供直接光照。

将大部分光源设为**已烘焙**可确保运行时的计算量相对较低。

## 场景

**场景**选项卡包含应用于整个场景的设置。您可以在此选项卡中启用和禁用**预算计算实时 GI** 和**已烘焙 GI** 功能。

**环境光照**部分中提供了诸多选项，您可用于定义影响环境光照的多个因素，如天空盒、环境光源类型以及环境光强度等：

- **反射反弹次数**选项是性能方面最重要的选项。**反射反弹次数**定义反射对象之前相互反射的次数，即视野覆盖这些对象的探测器的烘焙次数。如果反射探测器在运行时更新，此选项对性能有很大的负面影响。只有反射对象在探测器中清晰可见时，才可将反射次数设为高于一的值。
- 在**预算计算实时 GI**选项卡中，**CPU 使用率**选项定义在运行时评估 GI 上花费的处理器时间量。较高的**CPU 使用率**值可以加快光照的反应速度，但可能会影响帧率。在多处理器系统中，对性能的影响比较小。
- **已烘焙 GI**选项卡中的一个选项可以设置要压缩的光照贴图纹理。压缩光照贴图纹理可以降低对存储空间和带宽的要求，但压缩处理可增加纹理的失真。
- 在**常规 GI**选项卡中，请谨慎使用**定向模式**选项。如果您无法针对双重光照贴图使用延迟光照，则另外一种方法是使用定向光照贴图。这能够使您在没有实时光照的情况下使用法线贴图和镜面反射光照。如果必须保存法线贴图但未提供双重光照贴图，则可使用定向光照贴图。移动设备通常就属于这种情况。当**定向模式**设为**定向**时，将创建一个额外的光照贴图，以存储射入光线的主要方向。因此，这一模式要求大约两倍的存储空间。
- 在**定向镜面**模式中，会为镜面反射和法线贴图存储额外的数据。这时存储要求将提高四倍。
- **光照贴图**选项卡可用于设置和查找供场景使用的光照贴图资源文件。要访问光照贴图快照方框，必须取消选中**持续烘焙**选项。

下图显示了**光照**选项卡中的光照贴图：

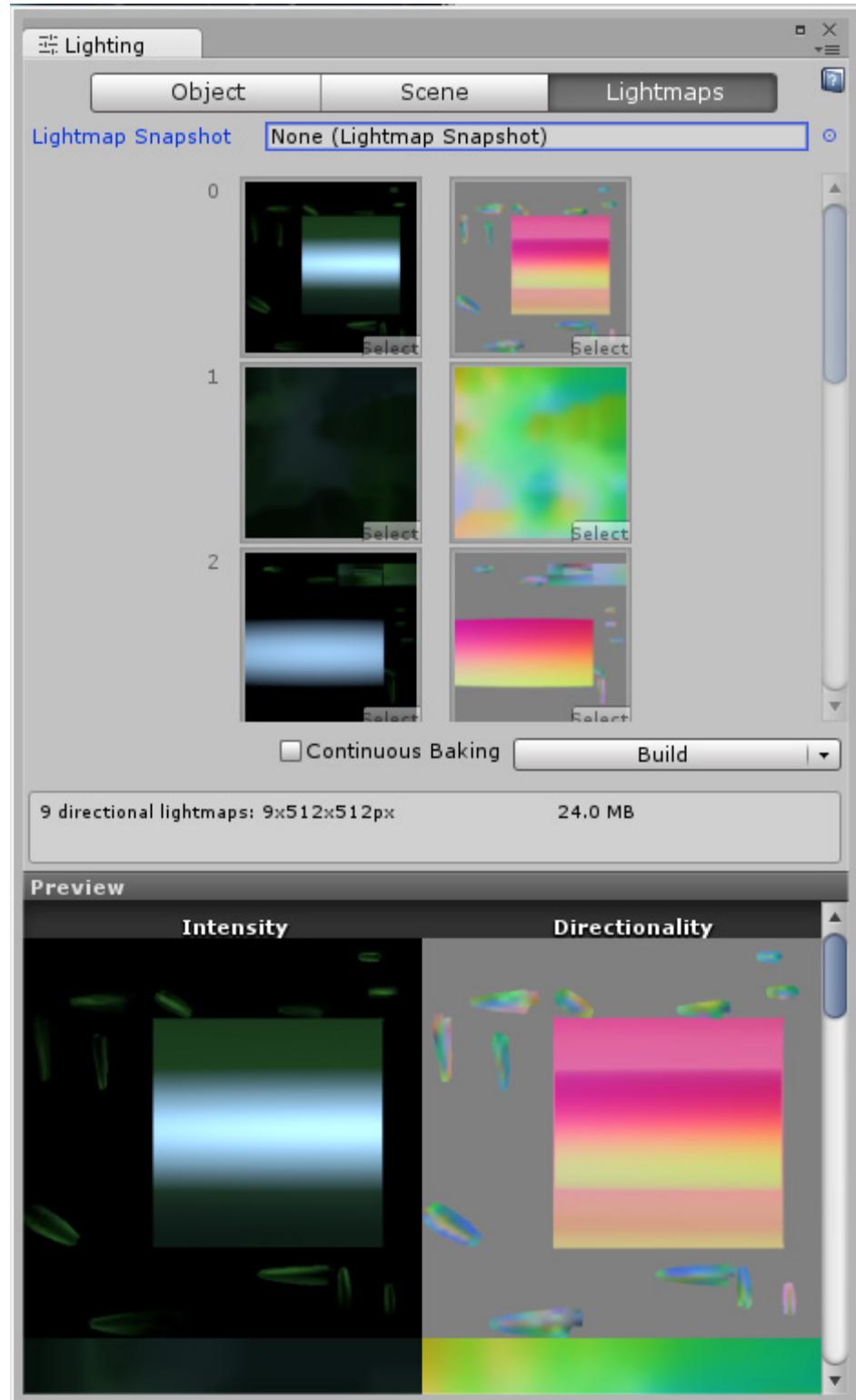


图 4-12 光照选项卡中的光照贴图

#### 使用定向光照贴图

如果您无法针对双重光照贴图使用延迟光照，则另外一种方法是使用定向光照贴图。这能够使您在没有实时光照的情况下使用法线贴图和镜面反射光照。

如果必须保存法线贴图但未提供双重光照贴图，则可使用定向光照贴图。移动设备通常就属于这种情况。

——备注——

此方法需要更多的视频内存，因为它需要计算第二组光照贴图以存储定向信息。

**针对游戏中的动态对象使用灯光探测器**

您可以使用灯光探测器向进行光照贴图的场景添加一些动态光照。

下图显示了灯光探测器设置：

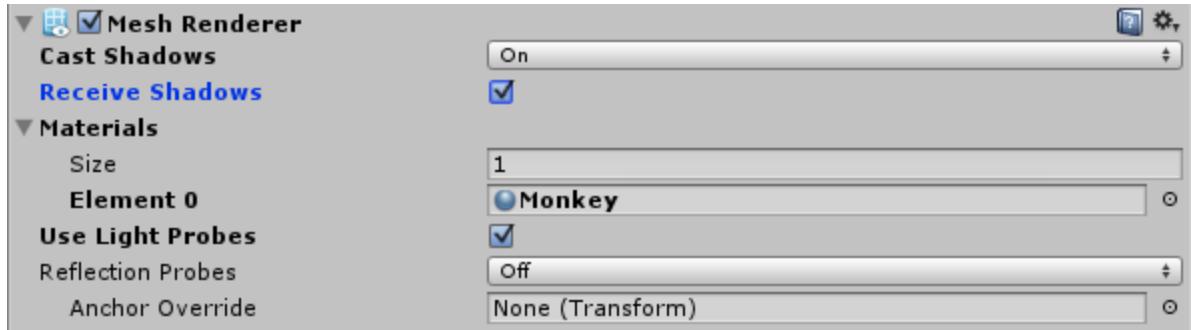


图 4-13 灯光探测器设置

灯光探测器可采集样本或探测某一区域中的光照。如果探测器形成整体或单元，则光照将根据其在单元中的位置插入这些探测器之间。

下图显示了灯光探测器：

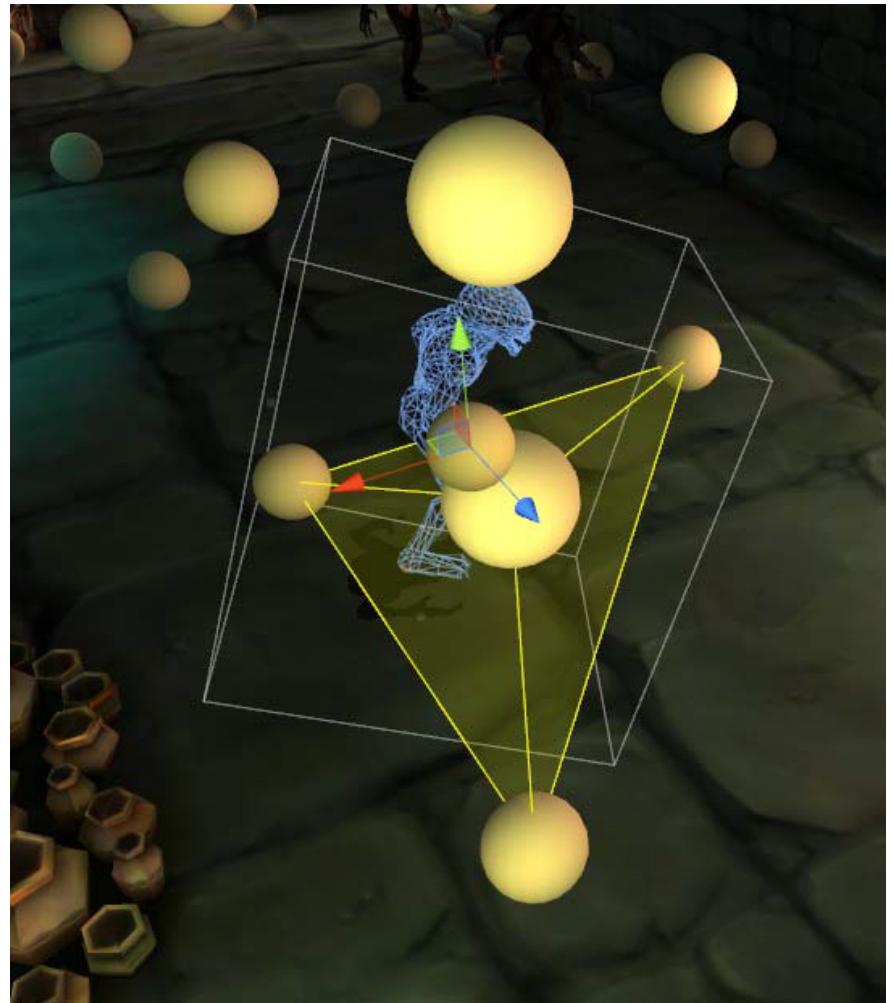


图 4-14 灯光探测器

使用的探测器越多，光照就越准确。通常，您不需要太多的灯光探测器，因为探测器之间存在插值。在灯光颜色或亮度出现较大差异的区域，您需要更多的灯光探测器。

然后便可根据最近的探测器采集的样本之间的插值，估计任意位置的光照。

小心放置灯光探测器，并用使用**灯光探测器选项**标记您希望会受其影响的网格。

下图显示了多个灯光探测器：

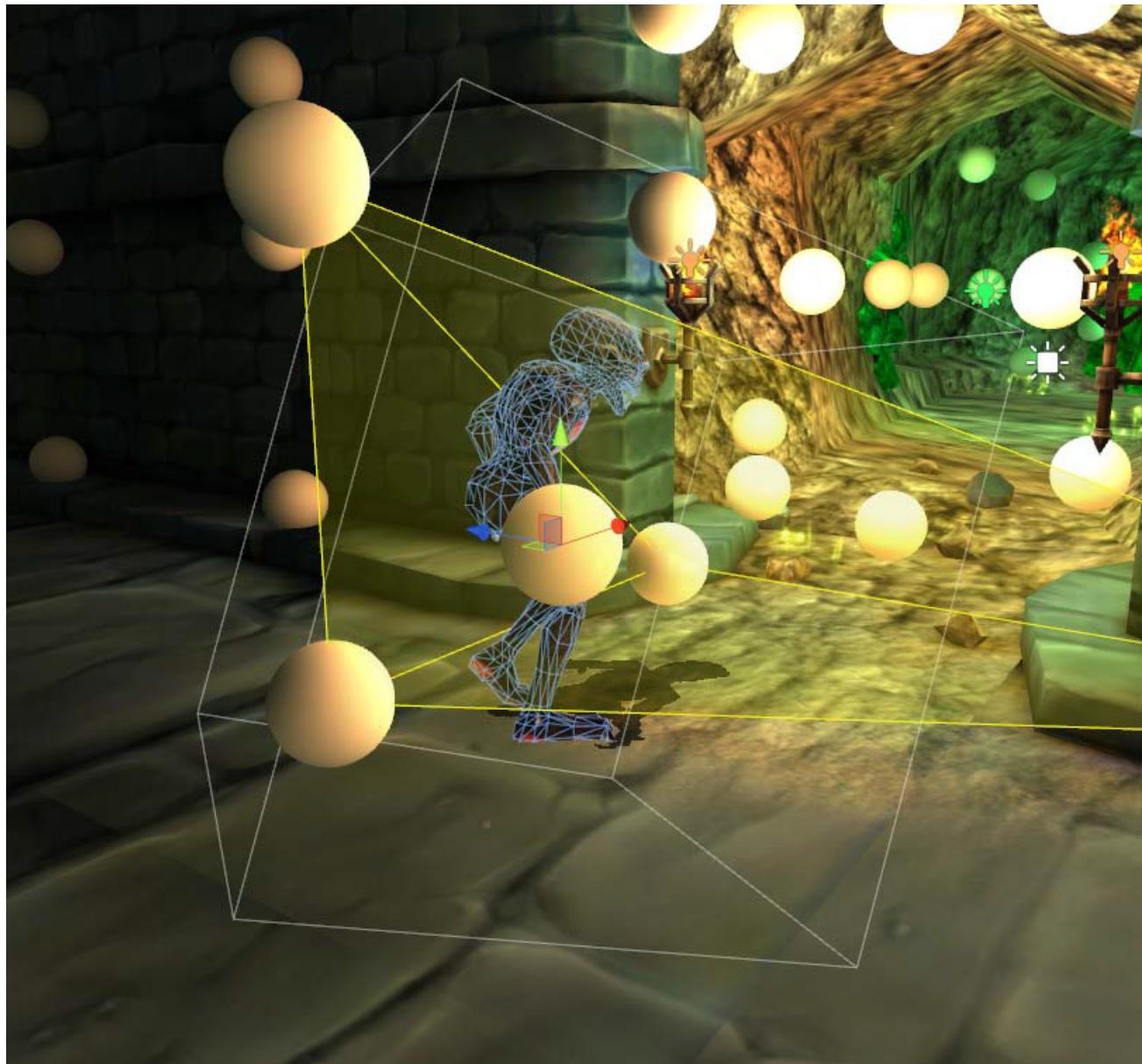


图 4-15 多个灯光探测器

#### 4.2.3 ASTC 纹理压缩

ASTC 纹理压缩是 OpenGL 和 OpenGL ES 图形 API 的官方扩展。  
ASTC 可以减小应用程序所需的内存以及 GPU 需要的内存带宽。

ASTC 提供的纹理压缩质量高、比特率低，而且控制选项也很多。它具有下列特性：

- 比特率从 8 位每像素 (bpp) 到小于 1 bpp 不等。这可使您微调文件大小与质量的平衡值。
- 支持 1 至 4 个颜色通道。
- 同时支持低动态范围 (LDR) 和高动态范围 (HDR) 图像。
- 支持 2D 和 3D 图像。
- 支持选择不同的特性组合。

下图显示了 ASTC 设置窗口：

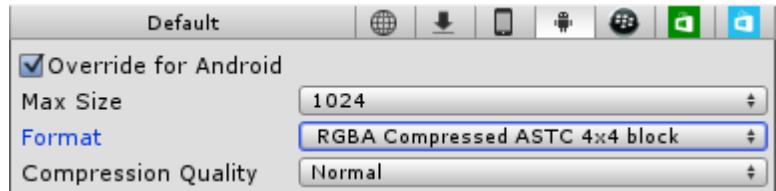


图 4-16 ASTC 设置

ASTC 设置窗口中有多个块大小选项。您可以从中选用，选择与资源最匹配的块大小。块大小越大，提供的压缩率越高。为显示细节度不高的纹理选择较大的块大小，如距离摄像机较远的对象。为显示细节度较高的纹理选择较小的块大小，如距离摄像机较近的对象。

#### 备注

- 如果您的设备支持 ASTC，请用它来压缩 3D 内容中的纹理。如果您的设备不支持 ASTC，请尝试使用 ETC2。
- 您必须区分 3D 内容所用纹理和 GUI 元件所用纹理。在有些情况下，最好使 GUI 纹理保持未压缩状态，以避免不必要的失真。

下图显示了可供不同纹理压缩格式使用的块大小：

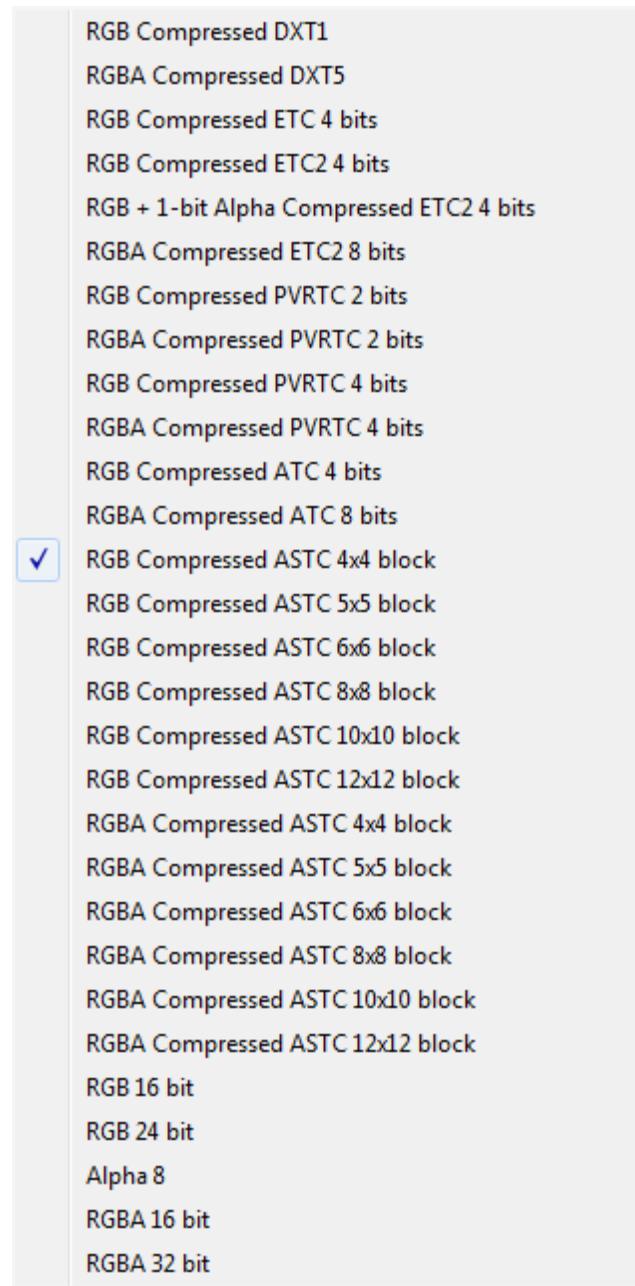


图 4-17 纹理压缩块大小

#### 为 ASTC 纹理选择恰当的格式

压缩 ASTC 纹理时，有诸多选项可供选择。

纹理压缩算法具有不同的通道格式，通常为 RGB 和 RGBA。ASTC 支持多种其他格式，但是这些格式不会出现在 Unity 中。通常，每种纹理用于不同的用途，例如：标准纹理、法线贴图、镜面反射、HDR、alpha 和查找纹理。为获得尽可能好的结果，所有这些纹理类型都需要不同的压缩格式。

下图显示了纹理设置：

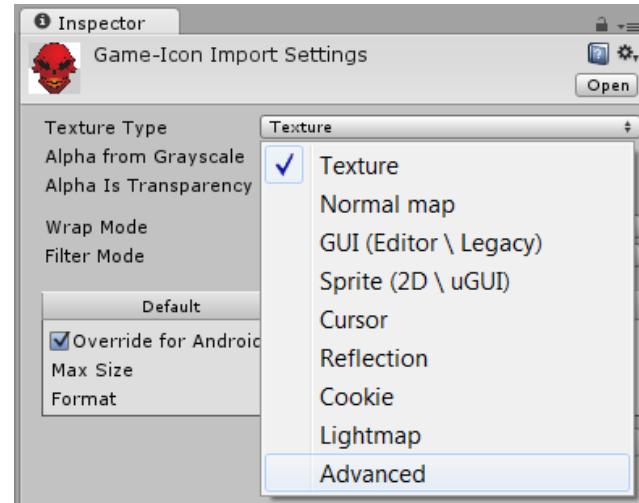


图 4-18 纹理设置

请勿在**构建设置**中以一种格式压缩所有纹理。使纹理压缩保持为**请勿覆盖**。

在项目层级中查找纹理并使其显示在检视面板。**Unity**通常会以**纹理**类型导入您的纹理。该类型仅为压缩提供一定数量的选项。将类型设为**高级**可显示更多选项。

下图显示了具有一定透明度的**GUI**纹理的设置。此纹理适用于**GUI**，因此**sRGB**和**纹理映射**已被禁用。若要包含透明度，您需要**alpha**通道。为此，请选中**Alpha**透明方框和**覆盖Android**方框。

下图显示了高级纹理设置：

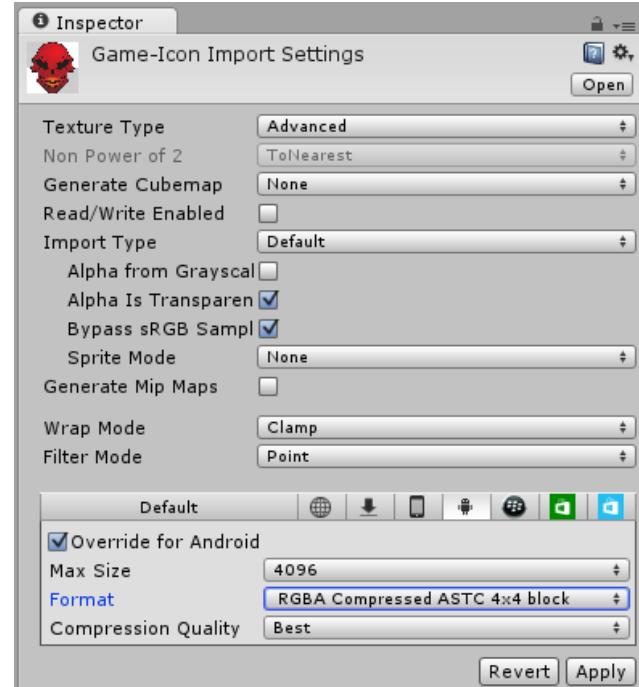


图 4-19 高级纹理设置

有一个用于选择格式和块大小的选项。**RGBA**包含**alpha**通道，**4x4**是您可以选择的最小块大小。将**最大纹理尺寸**设为最大值并设置**压缩质量**，此设置可规定寻找准确的压缩方法花费的时间。

针对所有纹理选择相应的设置可提高项目的视觉质量，并避免在压缩时出现不必要的纹理数据。

下表显示了，针对于一幅大小为 4M，像素分辨率为 1024x1024，格式为 RGBA（8 位每通道）的纹理而言，Unity 中所有可用的 ASTC 块大小所对应的压缩比率。

表 4-1 适用于 Unity 中可用 ASTC 块大小的压缩率

ASTC 块大小	大小	压缩比率
4x4	1 MB	4.00
5x5	655 KB	6.25
6x6	455 KB	9.00
8x8	256 KB	16.00
10x10	164 KB	24.97
12x12	144 KB	35.93

#### 4.2.4 纹理映射

纹理映射技术与能够同时提高游戏视觉质量和性能的纹理相关。

纹理映射是不同大小的纹理的预算版本。每个生成的纹理称为一个层级，它们的宽度和高度是前一个纹理的一半。Unity 能够自动生成完整的层级，从原始尺寸的第一层级到 1x1 像素版本。

若要生成纹理映射，请执行以下操作：

1. 在项目窗口中选择某一纹理。
2. 将纹理类型更改为高级。
3. 在检视面板中启用生成纹理映射选项。

下图显示了纹理映射设置：

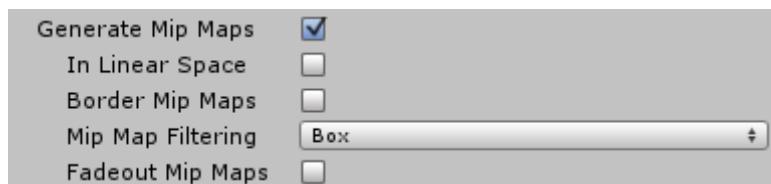


图 4-20 纹理映射设置

如果纹理没有纹理映射层级，当具有纹理的表面覆盖的区域（以像素表示）小于纹理尺寸时，GPU 会将纹理缩小至适合更小的区域。但是，此过程中会丧失部分准确性，即便使用滤波器插值像素颜色。

如果纹理具有纹理映射层级，GPU 将从最接近对象大小的层级中提取像素数据以渲染纹理。这可提高图像质量并降低带宽，因为 GPU 为获得更高质量已经离线扩展了层级，并且 GPU 仅从恰当的层级中提取纹理数据。多级渐进纹理 (mipmapping) 的劣势在于它额外需要 33% 的内存来存储纹理数据。

##### 纹理映射和 GUI 纹理

您通常不需要对 2D UI 中所用的纹理进行纹理映射。UI 纹理通常不需要扩大即可在屏幕上渲染，它们仅使用纹理映射链中的第一层级。

若要更改此设置，请在项目窗口中选择某一纹理，然后进入检视面板并查看纹理类型。将类型设置为编辑器 GUI 和传统 GUI，或者将类型设置为高级并禁用生成纹理映射选项。

## 4.2.5 天空盒

天空盒经常用于游戏和其他应用程序中，有多种方法可运行天空盒。

您可以通过使用单个立方体贴图渲染摄像机的背景来绘制天空盒。

这需要一个立方体贴图纹理和一个绘制调用函数。与其他方法相比，该方法使用的内存、内存带宽和绘制调用函数较少。

若要构建天空盒，请使用此方法：

1. 选择摄像机。
2. 确保清除标记设为天空盒。
3. 选择或添加天空盒组件。

天空盒组件对每个材质都拥有一个点。该材质是 Unity 在每帧开始时用于绘制摄像机背景的材质。

您使用的材质是包含全部所需信息的材质。使用 **移动 > 天空盒** 者色器创建材质，再使用该材质填充天空盒的六张图像。材质预览将显示一张图像。

当您完成此材质后，将其拖入天空盒组件中。您的天空盒将在后台进行正确渲染，不会出现明显的裂缝或不必要的绘制调用。

## 4.2.6 阴影

阴影有助于增加场景的透视度和真实感。没有阴影，有时很难告知对象的深度，尤其是它们与周围对象相似时。

阴影算法可能会非常复杂，渲染高分辨率的精确阴影时尤为如此。请确保在游戏中为阴影选择了相应级别的复杂度和分辨率。

Unity 支持转换反馈，实现了对实时阴影的计算。

### 备注

冰穴演示中实施的是自定义阴影。基于局部立方体贴图的阴影与运行时渲染的阴影相结合。

Unity 的 **编辑 > 项目设置 > 质量** 下包含多个阴影选项，它们可影响游戏的性能：

#### 硬阴影/硬加软阴影

软阴影看起来更为真实，但是计算时间较长。

#### 阴影距离

**阴影距离** 选项可定义与出现阴影的摄像机的距离。增大阴影距离会增加可见阴影的数量，从而加大计算量。增大阴影距离还会增加用于阴影贴图中阴影的像素元数量，从而被动地提高阴影分辨率。

您可以使用阴影距离较小且分辨率较高的硬阴影。这会在距离摄像机的适当范围内产生不是很复杂且质量较高的阴影。

进行光照贴图的对象不会产生实时阴影，您在场景中烘焙的静态阴影越多，GPU 执行的实时计算就越少。

下图显示了具有阴影的外星人：



图 4-21 具有阴影的外星人

#### 谨慎使用实时阴影

实时阴影可大幅提高场景的真实度，但是它们的计算量非常大。

在移动平台上，请尝试限制仅包含实时阴影的光源数量，并尝试使用光照贴图。

请考虑使用场景对象的网格渲染器组件。如果您不想使用它们投射或接收阴影，请相应地禁用**投射阴影**和**接收阴影**选项。这可降低渲染阴影的计算量。

您可以在**质量设置**部分找到更多阴影设置，例如：

- **阴影分辨率**: 可让您选择质量和处理时间之间的平衡值。
- **阴影距离**: 可让您限制距离摄像机较近的对象生成的阴影。
- **阴影级联**: 可让您选择质量和处理时间之间的平衡值。您可以将它设置为零、二或四。级联阴影贴图用于定向光源，可获得非常好的阴影质量，尤其是视距较远时。级联数越高，质量越好，但会增加处理开销。

#### 4.2.7 遮挡剔除

遮挡剔除流程可在从摄像机角度看对象被遮挡时禁用对象渲染。这使得渲染的对象较少，从而节省 GPU 处理时间。

但是，当对象完全退出摄像机视锥体时，Unity 将自动执行遮挡剔除，根据您的应用程序风格，可能仍存在不可见和无须渲染的其他对象。

Unity 包含称之为 Umbra 的遮挡剔除系统。有关 Umbra 的更多信息，请参阅 Unity 文档中的遮挡剔除章节。

用于遮挡剔除的设置取决于您的游戏风格。在设置不恰当的场景中使用遮挡剔除会降低性能，所以请务必谨慎选择设置。

#### 4.2.8 使用 `OnBecameVisible()` 和 `OnBecameInvisible()` 回调函数

如果您使用 `MonoBehaviour.OnBecameVisible()` 和 `MonoBehaviour.OnBecameInvisible()` 回调函数，则它们关联的游戏对象移入或移出摄像机视锥体时 Unity 会通知您的脚本。然后您的应用程序便可作出相应的反应。

您可以使用 `OnBecameVisible()` 和 `OnBecameInvisible()` 优化渲染流程，例如用第二个摄像机和渲染对象对池上的反射做渲染。

这需要在渲染到最终屏幕表面前渲染几何体并合并屏幕外的纹理。此技术比较耗费资源，因此仅在必要时使用。您只需要在反射可见时进行渲染，即在以下情况下渲染：

- 反射面位于摄像机视锥体内。
- 反射面前方无任何不透明物体。

上述条件可使用 `OnBecameVisible()` 和 `OnBecameInvisible()` 回调函数从反射面进行检查：

```
void OnBecameVisible()
{
    enabled = true;
}

void OnBecameInvisible()
{
    enabled = false;
}
```

即使在恰当的位置进行了检查，也仍然存在反射在屏幕外渲染的情况，即便反射在屏幕上不可见。为避免这种情况，您可以添加其他条件：

例如，摄像机必须位于反射面的空间内：

```
void OnBecameVisible()
{
    if (inside == false)
    {
        return;
    }
    enabled = true;
}

void OnBecameInvisible()
{
    if (inside == false)
    {
        return;
    }
    enabled = false;
}

void OnTriggerEnter()
{
    inside = true;
}

void OnTriggerExit()
{
    inside = false;
}
```

上述条件可将反射渲染限制在游戏的特定区域。这意味着您可以在计算量较小的其他游戏区域添加效果。

#### 4.2.9 指定渲染顺序

在一个场景中，对象渲染顺序对性能非常重要。

如果按照任意顺序渲染多个对象，则可能会出现一个对象在渲染之后被其前面的另一对象遮挡。也就是说，渲染被遮挡对象需要的计算全都浪费。

有各种各样的软件和硬件技术来减少被遮挡对象造成的计算浪费；但是，您可以引导这一流程，因为您清楚玩家如何探索这一场景。

自 Mali-T600 系列起，ARM Mali GPU 上用于减少计算浪费的硬件技术中包含了 Early-Z。从您的视角而言，Early-Z 是完全透明的系统，它在片段着色器被实际处理之前执行 Z 测试。如果 GPU 无法启用 Early-Z 优化，则在片段着色器之后执行深度测试。其计算成本可能会很高，而且片段被遮挡时也属浪费。Early-Z 系统检查被处理的像素的深度是否没有被更近的像素占用。若已被占用，它就不执行该片段着色器。此系统有益于性能，但在一些情形中会自动禁用。例如：如果片段着色器通过写入 gl\_FragDepth 变量修改深度，片段着色器将调用 `discard`；或者，如果为透明对象等对象启用了混合或 `alpha` 测试。为帮助此系统达到最高效率，请确保从前往后渲染不透明对象。这有助于减少仅含不透明对象的场景中的过度绘制因素。

按照从前往后的顺序渲染每一帧成本可能会很昂贵，如果同一通道中还渲染透明对象，这也可能会不正确。自 T620 起，ARM Mali GPU 提供了一种称为前像素终止 (*Pixel Forward Kill, PFK*) 的机制。Mali GPU 已流水化，可以为同一像素同时执行多个线程。当某一线程完成了其执行时，如果当前线程遮挡了该像素的所有其他线程时，PFK 系统会将它们全部停止。这起到了减少计算浪费的效用。

Unity 为您提供着色器内或材质内的排队选项，让您能够指定渲染的顺序。这可以在着色器中设置，这样具有使用该着色器的材质的对象可以一起渲染。在这一渲染组中，渲染的顺序是随机的，但透明等一些情形中除外。

默认情况下，Unity 提供了一些标准的组，它们按照下列顺序从头到尾进行渲染：

表 4-2 指定渲染顺序的队列值

名称	值	备注
背景	1000	-
几何体	2000	默认值，用于不透明几何体。
Alpha 测试	3000	这在所有不透明对象后绘制。例如，植物。
透明	4000	此组也按照从后往前的顺序渲染，以提供正确的结果。
叠加	5000	用户界面、镜头光晕和脏镜头等叠加效果。

可以使用整数值，而不用其字符串名称。这些不是唯一可用的值。您可以使用介于上表中所示的整数值来指定其他队列。值越高，渲染越靠后。

例如，您可以使用下列指令之一在 `Geometry` 队列之后、`AlphaTest` 队列之前渲染某一着色器：

```
Tags { "Queue" = "Geometry+1" }
Tags { "Queue" = "2001" }
```

### 使用渲染顺序提升性能

在冰穴演示中，洞穴覆盖了大部分屏幕，其着色器成本高昂。尽可能避免渲染它的组成部分可以提升性能。

利用 **Unity Frame Debugger** 和 **ARM Mali Graphics Debugger** 等其他工具查看帧缓冲区组成后，您会发现它已包含了渲染顺序优化。这些工具可帮助您查看渲染顺序。

若要打开 **Unity Frame Debugger**，请选择菜单选项 **窗口 > Frame Debugger**。这很有用处，因为一些东西在编辑器模式中似乎正确，但在您执行它们时却可能不能正确工作。例如，如果您有仅运行时设置或您要渲染来自另一摄像机的纹理，就可能会出现这种情形。以播放模式启动演示并定位摄像机后，您可以启用 **Frame Debugger**，获取由 **Unity** 执行的绘制序列。

在冰穴演示中，向下滚动绘制调用可显示洞穴率先得到渲染。然后，对象渲染到场景中，将已渲染的部分洞穴挡住。再例如，一些场景中的反光水晶被洞穴遮挡。在这些情形中，设置较高的渲染顺序值可以减少计算量，因为不会为被遮挡的水晶执行片段着色器。

### 4.2.10 使用深度预通道

通过为对象设置渲染顺序来避免过度绘制很有用处，但并不总是能够为每个对象指定渲染顺序。

例如，如果有一组具有计算成本高昂的着色器的对象，而且摄像机可以绕着它们自由旋转，那么某些之前位于后面的对象有可能会移到前面。这时，如果为这些对象设定了静态渲染顺序，一些对象即便被遮挡也会最后绘制。如果对象挡住了自身的一部分，也会出现这种问题。

在这样的情形中，您可以使用深度预通道来减少过度绘制。深度预通道渲染几何体，但不在帧缓冲区中写入颜色。这将使用最近可见对象的深度来初始化各个像素的深度缓冲区。在此预通道后，几何体被正常渲染，但会使用 **Early-Z** 技术，只有参与构成最终场景的对象才会被实际渲染。此技术需要额外的顶点着色器计算，因为要为每个对象计算顶点着色器两次，一次用于填充深度缓冲区，另一次用于实际的渲染。如果您的游戏受片段约束，并且您的顶点着色器中有额外容量，此技术很有用。

在 **Unity** 中，若要为带有自定义着色器的对象执行渲染预通道，请为着色器添加额外的通道：

```
// extra pass that renders to depth buffer only
Pass {
    ZWrite On
    ColorMask 0
}
```

在添加这一通道后，**Frame Debugger** 将显示对象被渲染了两次。第一次渲染时，颜色缓冲区中没有变化。

#### 备注

您可以在 **Frame Debugger** 左上角菜单中选择深度缓冲区进行查看。

## 4.3 资源优化

下表描述了资源优化：

### 禁用静态纹理读取/写入

如果您不动态修改纹理，请确保检视面板中的**读取/写入已启用**选项已被禁用。

### 合并网格以减少绘制调用数量

为减少渲染所需的绘制调用数量，您可以使用 `Mesh.CombineMeshes()` 方法将多个网格合并为一个网格。如果所有网格的材质相同，请将 `mergeSubMeshes` 参数设置为 `true`，以便它可以根据合并组中的每个网格生成单一子网格。

把多个网格合并为单个较大的网格将帮助您：

- 创建更高效的遮挡器。
- 将多个基于图块的资源转变为大型、无缝、实心的单一资源。

网格合并脚本对性能优化很有帮助，但是这取决于您的场景构成。较大网格在视图中的停留时间长于较小网格，请进行试验，以获取正确的网格大小。

应用此技术的一种方法是在层级中创建空的游戏对象，使其成为您想要合并的所有网格的父网格，然后为其附加一个脚本。

有关网格合并脚本的更多信息，请参阅 **Unity** 文档：

<http://unity3d.com>.

### 请勿在非动画 FBX 网格模型上导入动画数据

当导入不包含任何动画数据的 **FBX** 网格时，您可以在导入设置的“装备”选项卡中将**动画类型**设置为**无**。这样设置后，将网格置于层级时 **Unity** 不会生成未使用的动画组件。

### 避免读取/写入网格

如果运行时修改了模型，则 **Unity** 在保留原始数据的同时会在内存中另外保存一份网格数据副本。

如果运行时未修改模型（即使准备缩放），也请在导入设置的模型选项卡中禁用**读取/写入已启用**选项。这样便无需另外保存一份副本，因而可节省内存。

### 使用纹理贴图集

您可以使用纹理贴图集减少一组对象所需的绘制调用数量。

纹理贴图集是指合并成一个大型纹理的纹理组。多个对象可通过一组合适的坐标重复使用此纹理。这有助于 Unity 对共享相同材质的对象采用自动批处理。

设置对象的 UV 纹理坐标时，请避免更改其材质的 mainTextureScale 和 mainTextureOffset 属性。这会创建新的独特材质，该材质无法与批处理一同运行。请通过 MeshFilter 组件获取网格数据并使用 Mesh.uv 属性更改每个像素元的坐标。

下图显示了纹理贴图集：

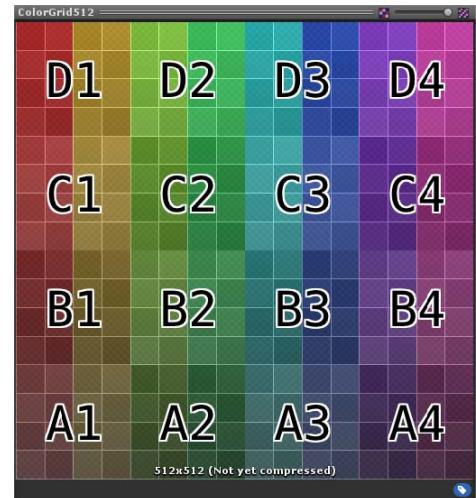


图 4-22 纹理贴图集

## 4.4 使用 Mali™ Offline Shader Compiler 优化

借助 Mali Offline Shader Compiler 这款工具，您可以将顶点、片段和计算着色器编译为二进制形式。该编译器也可用作性能分析工具。

本节包含以下小节：

- [4.4.1 关于 Mali™ Offline Shader Compiler \(第 4-55 页\)](#)。
- [4.4.2 衡量 Unity 着色器 \(第 4-55 页\)](#)。
- [4.4.3 分析统计信息 \(第 4-56 页\)](#)。
- [4.4.4 针对 Mali™ GPU 流水线进行优化 \(第 4-56 页\)](#)。
- [4.4.5 减少流水线周期的其他技巧 \(第 4-57 页\)](#)。

### 4.4.1 关于 Mali™ Offline Shader Compiler

应用程序中的着色器在 GPU 上运行。这要求 GPU 花费时间计算着色器的最终结果，如顶点位置或像素颜色。

Mali Offline Shader Compiler 提供关于着色器在 Mali GPU 每一流水线中所需执行的周期数的信息。

生成的周期数是为某一 GPU 定制的。您要通过命令行中的选项选择 GPU。务必选择与应用程序的目标设备范围对应的 GPU。这可确保您从工具中获得的统计信息切实可靠，符合您的典型用例情景。

### 4.4.2 衡量 Unity 着色器

Unity 着色器使用编程语言 *C for Graphics* (Cg) 编写。Cg 以 C 编程语言为基础，但进行了一些修改，因而更适用于 GPU 编程。

在构建过程中，Unity 将 Cg 转换为 OpenGL、OpenGL ES 或 DirectX。

检索 OpenGL ES 着色器代码：

1. 选择您要在 Unity 中分析的着色器。
2. 选择 **OpenGLES30** 或 **OpenGLES20** 作为您要为之生成程序的自定义平台。
3. 单击**编译并显示**按钮。

其结果将显示在您的开发环境中。

#### 备注

- Mali Offline Shader Compiler 仅支持 OpenGL ES 着色器。
- 如果您的构建平台设为 Android，Unity 默认认为生成 **OpenGLES30** 着色器。

顶点和片段会话通常由 `#ifdef VERTEX` 或 `#ifdef FRAGMENT` 分隔。如果使用 `#pragma multi_compile <FEATURE_OFF> <FEATURE_ON>` 等选项，文件中将生成多个着色器变体。

通常会存在多个 VERTEX 和 FRAGMENT 部分。Unity 启动您的应用程序时，会单独编译各个变体。启用一项功能时，会选中相关的变体。

由于代码已转换为 OpenGL ES，您可以将顶点和片段着色器代码复制到两个独立的文件，并使用 Mali Offline Shader Compiler 进行编译。

使用下列选项之一编译着色器：

- `-v` 用于顶点着色器。
- `-f` 用于片段着色器。

下图显示了 Mali Offline Shader Compiler 的输出：

```
ARM Mali Offline Compiler v4.6.0
(C) Copyright 2007-2015 ARM Limited.
All rights reserved.

No driver specified, using "offline_api" as default.

No core specified, using "Mali-T880" as default.

No core revision specified, using "r2p0" as default.

8 work registers used, 13 uniform registers used, spilling not used.

          A      L/S      T      Total      Bound
Cycles:    43      11      13      67      A
Shortest Path: 16      11      13      40      A
Longest Path: 16      11      13      40      A

Note: The cycles counts do not include possible stalls due to cache misses.

Compilation succeeded.
```

图 4-23 Mali Offline Shader Compiler 输出

#### 4.4.3 分析统计信息

Mali Offline Shader Compiler 产生的统计信息提供着色器所要求的每一顶点或像素的周期数的测量结果。

结果分成三行：

- 合计。
- 最短路径。
- 最长路径。

最短和最长路径通过检查代码中是否采取分叉的效应来衡量。这提供对执行周期数最小和最大值的估计。

从算术而言，第一行中的测量结果除以算术流水线的数目。该数值为一、二或三，具体取决于 Mali GPU。

第二和第三行用于加载/存储和纹理流水线。它们没有考虑缓存遗漏，所以最好将这些数字乘以 1.5，从而获得更加实际的估计。

#### 4.4.4 针对 Mali™ GPU 流水线进行优化

Mali Offline Shader Compiler 提供每一流水线中使用的周期数。着色器在流水线中速度最慢，其周期数最高。将流水线中速度最慢的作为优化目标，优化您的着色器。

Mali GPU 包含三种类型的处理流水线：

- 算术流水线。
- 加载/存储流水线。
- 纹理流水线。

这些流水线全部并行运行。着色器通常使用所有这三种流水线。

##### 算术流水线

所有算术运算消耗算术流水线中的周期。

下面列出了您可以降低算术流水线使用量的多种方法：

- 避免使用复杂算术，例如：
  - 矩阵求逆函数。
  - 取模运算符。
  - 除法。
  - 行列式。
  - 正弦。
  - 余弦。

- 对于整数操作数，使用移位等运算来计算除法、取模和乘法。
- 对正交矩阵使用移项，而不要求逆。
- 为避免计算移项，如果其中一个矩阵已移项，更改矩阵-向量或矩阵-矩阵乘法中操作数的顺序。例如：

```
Transpose(A)*Vector == Vector * A.
```

也可以将负载移到其他流水线中，从而降低对算术管道的负载：

- 将矩阵作为统一变量传递，而不要计算它们。这可使用加载/存储流水线。
- 使用纹理来存储代表正弦或余弦等函数的一组预算值。  
这可将负载移到纹理流水线。

### 加载/存储流水线

加载/存储流水线用于读取统一变量、写入变量，以及访问着色器中的缓冲区，如统一缓冲区对象或着色器存储缓冲区对象。

如果应用程序属于加载/存储流水线束缚型，可尝试下列技巧：

- 使用纹理而非缓冲区对象来读取着色器中的数据。
- 使用算术运算计算数据。
- 压缩或减少统一变量和变量。

### 纹理流水线

纹理访问使用纹理流水线中的周期，也会使用内存带宽。使用大纹理可能是不利的，因为缓存遗漏几率更高，而且这样可能导致多个线程因为等待数据而停滞。

要提高纹理流水线的性能，可进行如下尝试：

#### 使用纹理映射

纹理映射可提高缓冲命中率，因为它根据纹理坐标变化选用分辨率最佳的纹理。

#### 使用纹理压缩

这也有益于降低内存带宽并提高缓存命中率。每一压缩的块包含多个像素元，所以其访问变得更容易缓存。

#### 避免三线性或各向异性滤波

三线性和各向异性滤波将增加获取像素元所需的运算数。若非绝对必要，请避免使用这些技术。

## 4.4.5 减少流水线周期的其他技巧

还有多种技巧可用于减少每一流水线中使用的周期数。

### 避免寄存器溢出

Mali Offline Shader Compiler 可指示您的着色器是否溢满寄存器。造成寄存器溢出的原因通常是线程中包含大量变量，它们无法完整装入寄存器集中。

寄存器溢出通常由包含大量以下对象的线程造成：

- 输入统一变量 (uniform)。
- 变量 (varying)。
- 临时变量。

如果变量使用高精度，也可能会发生寄存器溢出。

寄存器溢出会强制 Mali GPU 从内存读取一些统一变量，这可加重加载/存储单元的负载并降低性能。要解决此问题，可尝试减少您向着色器提供的统一变量数量并降低其精度。

在冰穴演示中，部分着色器遭遇了寄存器溢出状况，例如：

```
8 work registers used, 16 uniform registers used, spilling used.
```

图 4-24 具有寄存器溢出的着色器。

减少允许的统一变量数量可解决此问题，因而能提高性能。例如：

```
8 work registers used, 13 uniform registers used, spilling not used.
```

图 4-25 没有寄存器溢出的着色器。

### 降低变量和统一变量的精度

编写自定义着色器时，您可以指定利用 32 位浮点数或 16 位半浮点数指定统一变量和变量的浮点精度。精度确定了最小值和最大值，以及变量可以表示的数值的粒度。

使用半浮点数有几个好处：

- 带宽用量减少。
- 算术流水线中使用的周期数减少，因为着色器编译器可以优化您的代码，使用更多并行化。
- 要求的统一寄存器数量减少，这反过来又降低了寄存器溢出风险。

下列代码提供了冰穴演示中简单片段着色器变体的示例。该着色器使用 Mali Offline Shader Compiler 编译了两次。

第一个代码示例使用浮点数编译：

```
$ malisc -f -V Compiled-CaveMaliStandardFloat.frag
ARM Mali Offline Compiler v4.6.0
(C) Copyright 2007-2015 ARM Limited.
All rights reserved.

No driver specified, using "offline_api" as default.

No core specified, using "Mali-T880" as default.

No core revision specified, using "r2p0" as default.

8 work registers used, 13 uniform registers used, spilling used.

          A      L/S      T      Total    Bound
Cycles:   47      15      10      72      A
Shortest Path: 15      13       9      37      A
Longest Path: 16      15      10      41      A

Note: The cycles counts do not include possible stalls due to cache misses.
Compilation succeeded.
```

图 4-26 使用浮点数编译的着色器

第二个代码示例使用半浮点数编译：

```
$ malisc -f -V Compiled-CaveMaliStandardHalf.frag
ARM Mali Offline Compiler v4.6.0
(C) Copyright 2007-2015 ARM Limited.
All rights reserved.

No driver specified, using "offline_api" as default.

No core specified, using "Mali-T880" as default.

No core revision specified, using "r2p0" as default.

7 work registers used, 7 uniform registers used, spilling not used.

Cycles:      A      L/S      T      Total      Bound
Shortest Path: 15        9        9        33        A
Longest Path: 15       11       10       36        A

Note: The cycles counts do not include possible stalls due to cache misses.

Compilation succeeded.
```

图 4-27 使用半浮点数编译的着色器

在半浮点数版本中，加载/存储指令的数量减少了。使用的工作和统一寄存器数量变少，并且没有寄存器溢出。使用半浮点数时生成的代码大小也小于使用浮点数生成的代码。这可提高 Mali GPU 上的缓存命中率，从而提升了性能。

### 将世界空间法线贴图用于静态对象

您可以使用正切空间法线贴图来提高模型的细致度，而不必增加几何细节。您可以将正切空间法线贴图用于动画对象而不必修改它们，因为它们具有对网格中每一三角形的局部性。

但遗憾的是，这需要在着色器中进行更多算术运算，从而获得正确的结果。对于静态对象，这些计算通常是不必要的。

您可以选择使用局部空间法线贴图或世界空间法线贴图。使用局部空间法线贴图可减少着色器中执行的计算数量，但是必须对采样的法线应用模型上的变换。世界空间法线贴图不需要任何变换，但它们是静态的，并且对象无法移动。在冰穴演示中，洞穴和其他高质量对象是静态的；使用世界空间法线贴图可以大幅减少着色器需要的 ALU 运算数量。大多数常见的 3D 建模工具可以创建世界空间法线贴图，或者您可以通过在离线进程中的代码来生成它们。

# 第 5 章

## Unity 中利用 **Enlighten** 的全局照明

本章节描述了 Unity 中利用 **Enlighten** 的全局照明。

它包含下列部分：

- [5.1 关于 \*\*Enlighten\*\* \(第 5-61 页\)](#)。
- [5.2 \*\*Enlighten\*\* 的结构 \(第 5-62 页\)](#)。
- [5.3 使用 \*\*Enlighten\*\* 设置场景 \(第 5-71 页\)](#)。
- [5.4 示例：冰穴演示的 \*\*Enlighten\*\* 设置 \(第 5-73 页\)](#)。
- [5.5 在自定义着色器中使用 \*\*Enlighten\*\* \(第 5-80 页\)](#)。

## 5.1

### 关于 *Enlighten*

Unity 5 中包含了 *Enlighten*，它是 Geomerics 出品的实时全局光照解决方案。Geomerics 是 ARM 旗下公司。

Unity 5 使用 *Enlighten* 模拟实时非直接光照。非直接光照是从物体表面反弹回场景中的光线。

*Enlighten* 的核心是一个实时光能传递引擎。该引擎生成仅包含非直接光照的光照贴图和灯光探测器，非直接光照重新投影到场景中。在 Unity 5 编辑器和 Unity 所支持的众多平台上的游戏中，光照贴图和灯光探测器可得到实时更新。这些平台包括 Windows、OS X、iOS 和 Android。此外，Unity 使用 *Enlighten* 产生烘焙的光照贴图。

当 *Enlighten* 计算场景中的非直接光照时，其结果存储在光照贴图和灯光探测器中，它们应用到着色器代码中。标准的 Unity 材质充分利用了 *Enlighten*，包括自定义材质着色器。着色器和 Unity 渲染引擎确保直接光照得到正确渲染，使得最终游戏外观与最终组成相符。

下图显示了一个场景示例，其分别为只有直接光照、只有 *Enlighten* 非直接光照，以及组合了直接光照和非直接光照的效果。



图 5-1 走廊

左图中的场景只有直接光照照明，没有反弹光线和额外的环境光照。阴影中的所有表面均为黑色。

中图显示了相同的场景，但光照设置只有 *Enlighten* 非直接光照，没有应用反射纹理。这是 *Enlighten* 实时生成的光照输出。

右图显示了最终的组成，它包含直接光照和 *Enlighten* 非直接光照。

## 5.2 **Enlighten** 的结构

**Enlighten** 包含一系列组件，它们可在 **Unity** 中进行配置。了解这些组件以及如何搭配使用它们将很有益处。

本节包含以下小节：

- [5.2.1 概览（第 5-62 页）。](#)
- [5.2.2 预计算（第 5-62 页）。](#)
- [5.2.3 实时求解器（第 5-69 页）。](#)
- [5.2.4 烘焙光照贴图（第 5-70 页）。](#)

### 5.2.1 概览

**Enlighten** 的组件包括：

#### 预算算

它预先计算场景中的光线传输。预算算仅基于静态几何体，不依赖材质或光照。

#### 实时求解器

它获取预算算的数据并将它与材质和光照信息组合，实时生成光照贴图和灯光探测器。

#### 光照贴图烘焙器

它为直接和非直接光照以及环境遮挡生成已烘焙光照贴图。光照贴图烘焙器依赖于预算算数据。

下列章节将更加详细地介绍这些组件。

### 5.2.2 预计算

预算算组件根据场景的几何体计算数据。在运行时，**Enlighten** 使用预算算数据为所有光照设置实时生成非直接光照。

在“光照”窗口中按“构建”按钮即可启动预算算；或者，如果选择了自动构建选项，则更改场景时也会启动。

启动了预算算时，**Unity** 底部的进度条将显示运行中预算算任务的状态。

下图显示了进度条和当前的作业。

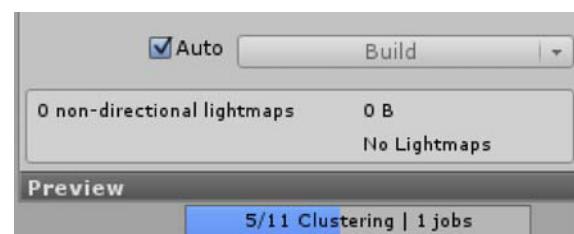


图 5-2 进度

#### 备注

进度条中显示的预算算步骤要多于本节中所述，但这些不太重要。

预算算执行下列步骤：

1. 封装。
2. 集群。
3. 计算光线传输。

为帮助理解各个步骤以及影响它们的设置，您可以思考下图中的场景。该场景包含地板、带有墙角的墙壁，以及悬挂于墙上的电视机。

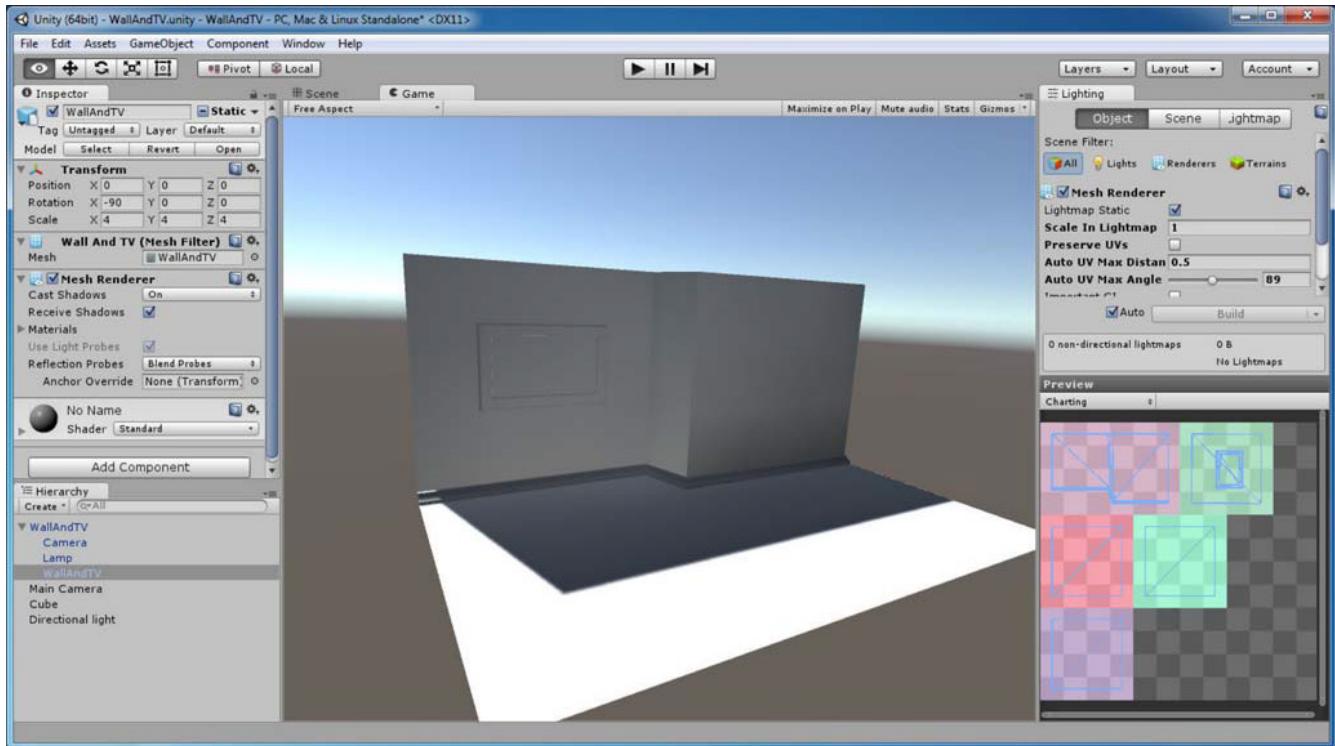


图 5-3 墙壁与电视机场景

### 封装

封装确定 UV 中的图表，再将这些图表封装为用于实时光照的光照贴图。

封装使用现有的 UV，或者使用您在“导入”设置中选择“生成光照贴图 UV”时 Unity 为您创建的 UV。

图表是 UV 中共享顶点的三角形组。确定了图表后，Enlighten 重新封装图表，确保图表之间没有漏光，并且 UV 尽可能紧密封装在一起。您可以在 **UV 图表渲染模式** 中查看封装结果，也可在制图预览中查看。本节中的下列几张图显示了 **UV 图表渲染模式** 中的图表，其制图预览显示在窗口的右下方。

预算算、运行时内存用量和 Enlighten 运行时性能都取决于光照贴图像素的数量。网格越复杂，生成具有较少图表的 UV 解包的难度越高。您可以通过利用 Enlighten 生成可移除网格中较小细节的简化 UV 来优化图表数量。生成简化 UV 默认认为启用，但如果已仔细制作了自己的 UV，您可以通过勾选 **保留 UV** 来禁用此项。您可以在 **UV 图表模式** 中查看结果。

下图显示了 **UV 图表模式**，以及启用了 **保留 UV** 的制图预览。每一图表以不同的颜色显示。

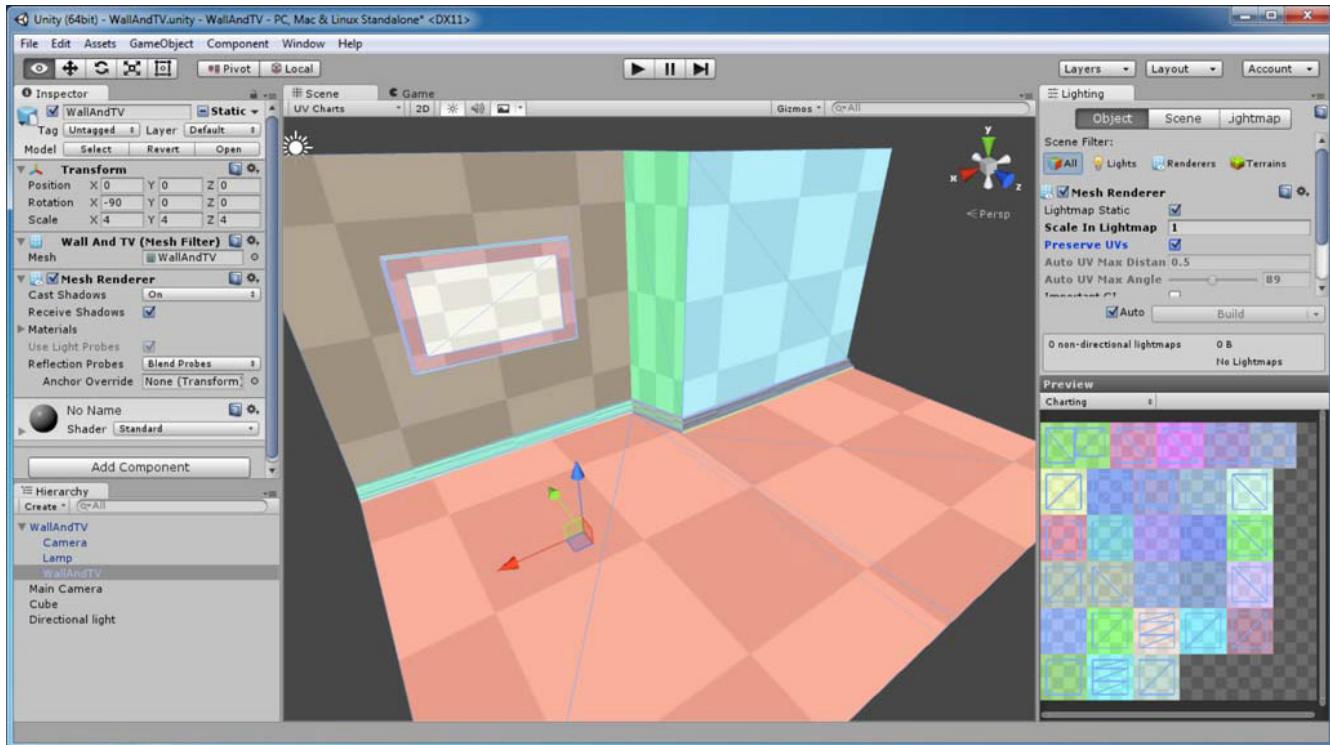


图 5-4 启用了保留 UV 的墙壁与电视机场景

如图中所示，电视机、墙壁和裙板的一部分都由多个图表组成。

只有您已仔细制作了要在 Unity 中保留的 UV 时，才应启用**保留 UV**。通常而言，最好不要启用此标记。

影响简化 UV 生成的最重要设置是**自动 UV 最大距离**设置。**自动 UV 最大距离**告诉 Enlighten 可以从照明中省略什么细节。Enlighten 尝试通过为组合图表创建 UV 布局来合并图表，但它不会分割输入图表。通过将识别的所有图表的顶点投影到世界空间中的一个优化平面上，创建新的 UV 布局。只有其世界空间顶点在平面上方给定距离内的图表才会考虑合并。

当**自动 UV 最大距离**设为较小的值（如 0.01）并且**保留 UV** 已禁用时，封装结果将如下图所示。

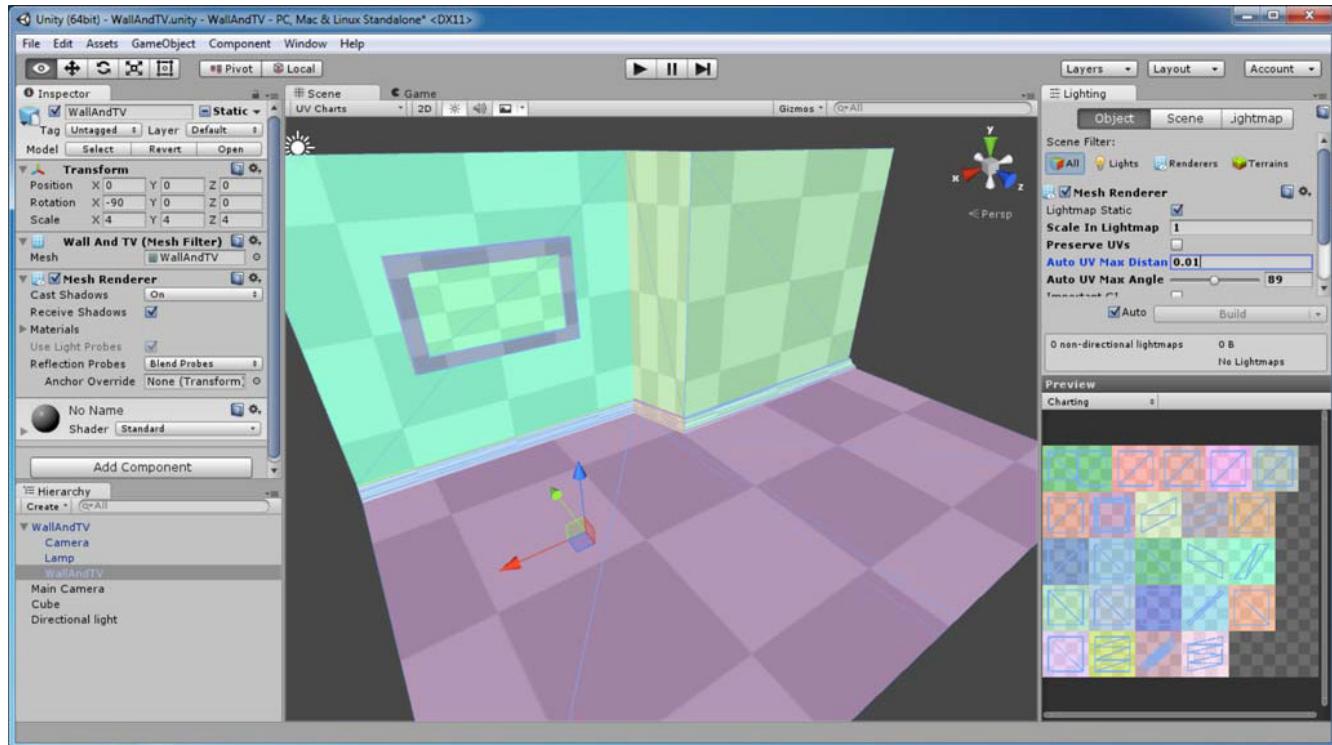


图 5-5 自动 UV 最大距离为 0.01 时的墙壁与电视机场景

除了视觉化的颜色外，[图 5-5 自动 UV 最大距离为 0.01 时的墙壁与电视机场景（第 5-65 页）](#)中显示的封装与启用了保留 UV 的[图 5-4 启用了保留 UV 的墙壁与电视机场景（第 5-64 页）](#)相似。它具有数量大致相同的图表，但解包稍有差异。

对于选定的光照贴图分辨率，只有少量光照贴图像素被用于墙壁和地板。这足以捕捉粗糙的场景全局光照。电视机和裙板非常小，它们几乎只需要一个像素。但是，它们被分配了许多像素，因为它们有多个图表组成，而图表的最小大小为  $4 \times 4$  像素。

#### 备注

如果最小图表大小的值设为 2，则图表可具有较小的  $2 \times 2$  像素分辨率。不过，这将禁用拼接，可能会造成可见的裂缝。

下图显示了自动 UV 最大距离的值设为 0.5 时的结果。

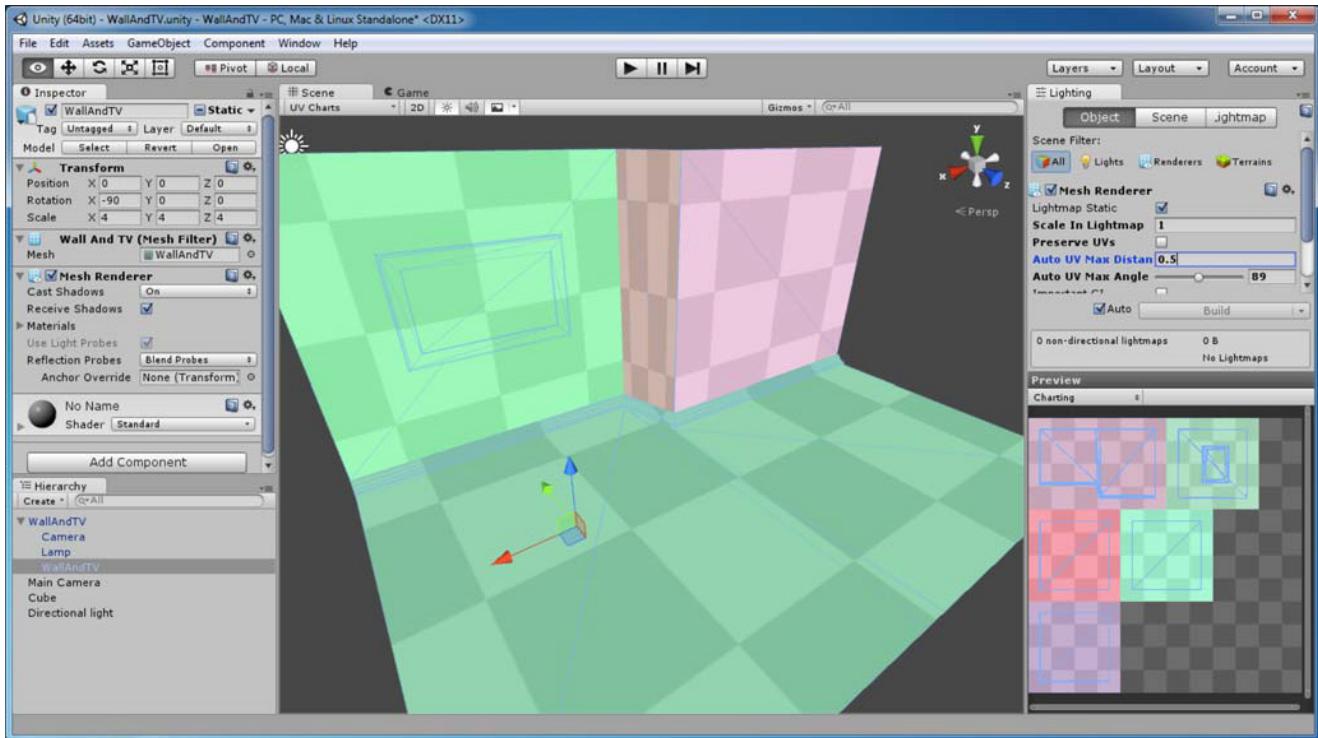


图 5-6 自动 UV 最大距离为 0.5 时的墙壁与电视机场景

当自动 UV 最大距离的值设为 0.5 时，Enlighten 分别将电视机和裙板的 UV 投影到墙壁和地板上。因此，它们不需要在光照贴图中增加额外的像素。它显示在了制图预览中。增大自动 UV 最大距离的值将导致更多细节被忽略。

下图显示了自动 UV 最大距离的值设为 2 时的结果。

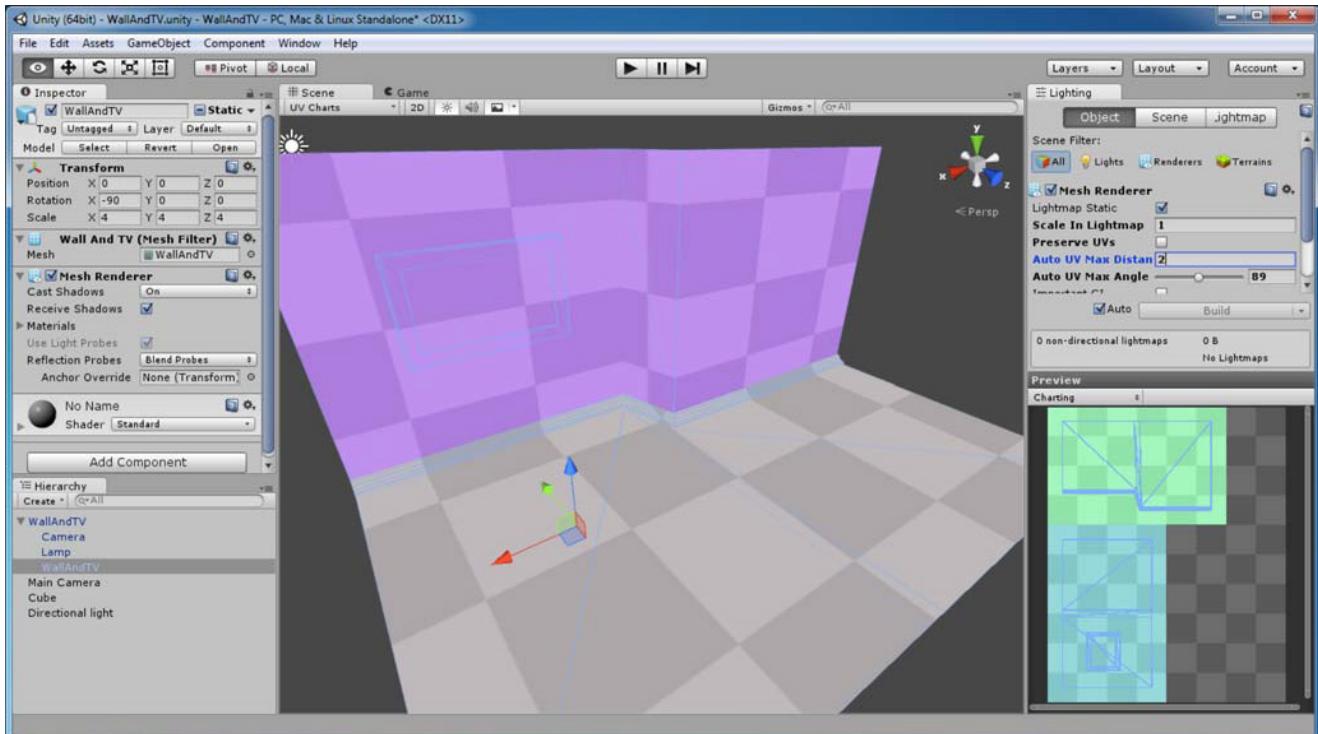


图 5-7 自动 UV 最大距离为 2 时的墙壁与电视机场景

当自动 UV 最大距离的值设为 2 时，Enlighten 将包括墙角在内的墙壁不同部分考虑为一个区段，其像素跨越不同的部分。因此，其光照贴图比较小。增大距离将导致更多几何体投影到一个平面中，造成细节丢失。

与自动 UV 最大距离类似，您可以根据三角形之间的角度定义 Enlighten 考虑进行合并的图块。您可以设置自动 UV 最大角度的值来定义最大角度。默认值为 89 度，这将确保 Enlighten 不合并在世界空间中设为互成直角的图表。

### 集群

为了对场景中的光线传输建模，Enlighten 将场景几何体分割为群集。

群集与光照贴图像素独立，仅从三角形的位置和朝向生成。不使用材质或 UV 坐标。

下图显示了墙壁与电视机场景示例中的群集。

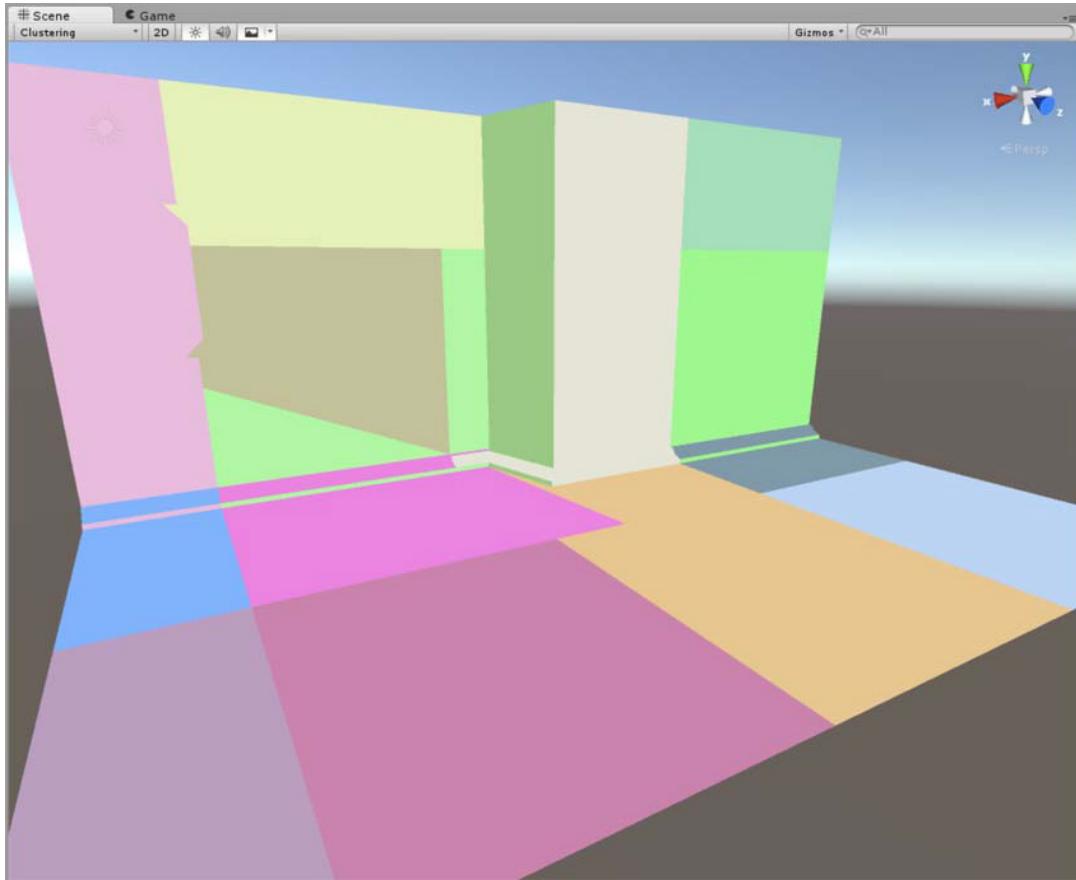


图 5-8 墙壁与电视机群集

### 计算光线传输

每一群集从放入场景中的光源接收入射光线。群集也会发出光线。

对于光照贴图中的每一个像素，以及每一个灯光探测器，Enlighten 在其半球的所有方向或探测器的所有方向上投射光线。它从像素或探测器计算群集的可见性。Enlighten 假定场景仅包含漫射表面，所以其可见性与像素从群集收到的光照直接成比例。其可见性在计算机图形中通常称为形状因子。形状因子在计算后将被压缩，以降低内存使用和性能的要求。

下图显示了此流程的一个简化视图，它使用了带有地板、墙壁和立方体的示例场景。图中显示了像素 X，它可以看到群集 A、B、C 和 D，其可见性值为 0.05、0.1 和 0.2。在运行时，Enlighten 可以通过估算下列表达式，评估此像素收到的光照：

$$0.05*A+0.1*B+0.05*C+0.2*D$$

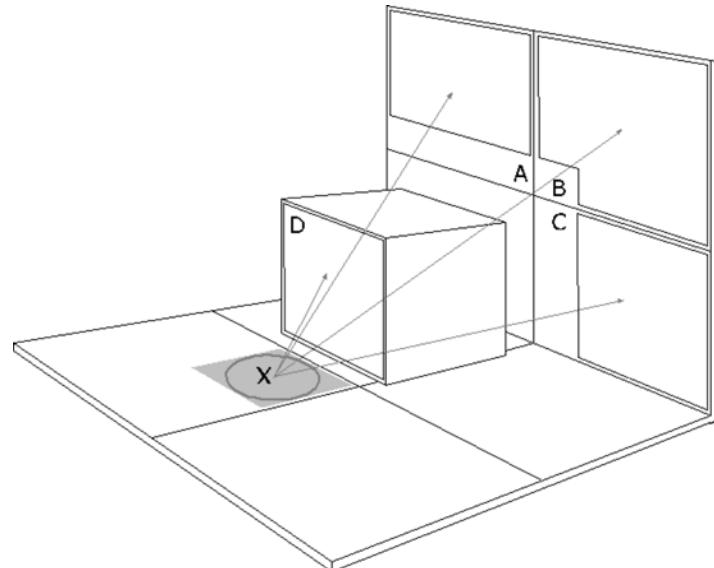


图 5-9 形状因子

下图显示了一个光照设置，群集 A 上亮红色光，群集 B 上绿色光，群集 C 上蓝色光，群集 D 上则没有光线。由于群集 B 的权重大于群集 A 和 C 的权重，像素 X 的最终输出值具有绿色色调。

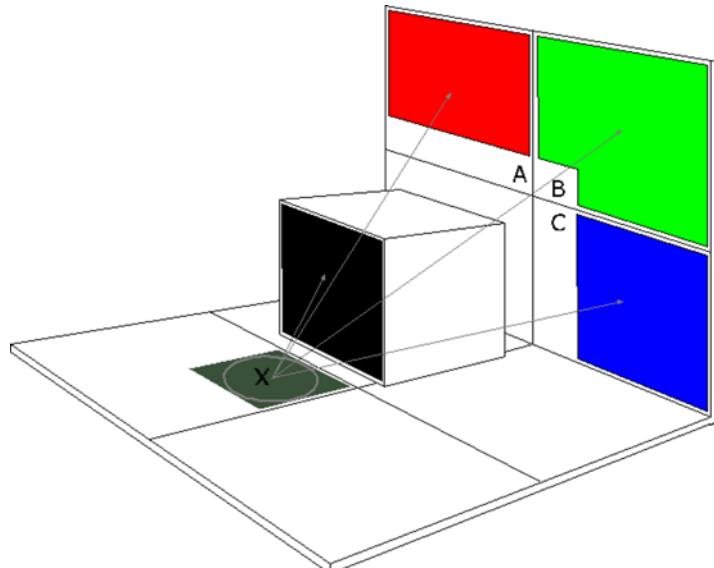


图 5-10 带有光照的形状因子

每当几何体更改时，必须更新预计算的数据，因为群集依赖场景几何体。Unity 可以在需要时自动触发预计算；或者，您可以关闭连续构建，仅在完成了场景变化时触发预计算。您必须预计算场景，然后才能使用光照或烘焙光源。

Unity 提供了**群集分辨率**设置，它控制群集相对于光照贴图分辨率的大小。默认值为 0.5，这表示群集是光照贴图像素大小的两倍。通常没有原因会导致需要更改**群集分辨率**设置。

光线传输计算形状因子。**Enlighten** 具有影响此步骤的两个重要参数：

- **辐照度预算**是 **Enlighten** 存储的形状因子数量。**Enlighten** 根据需要合并形状因子，使其保持在给定预算范围内，但预算过小会造成照明过于简化。另一方面，预算较大可占用较多的运行时内存，运行时的求解成本也会增高。通常而言，默认值 128 便是良好的选择，但 64 这样的低值也可提供不错的结果。提高这个值不会增加光线传输时间，也不会增加预算计算时间。
- **辐照度质量**是每个像素投射的光线数量。加大这个值可延长预算计算时间。默认值 8192 一般能产生不错的结果，但如果您选择了较高的光照贴图分辨率，ARM 建议您增加这个值，因为非直接光照可能会包含噪点。此值不影响运行时内存或性能。

为加快开发速度，可将此选项设为默认值，或者设为能在开发游戏时获得合理结果的值。在出货之前提高该值，从而获得最佳的质量。

### 5.2.3 实时求解器

实时求解器组件存在于编辑器中，也存在于游戏中。它为非直接光照的直接反馈生成光照贴图和探测器。

实时求解器包含下列阶段：

#### 入射光照阶段

入射光照阶段计算光源如何照亮群集，其包含动态光源。所有光源的光值加到一起，其最终值再乘以群集的反射率。而这基于组成群集的几何体的材质。对光源的逼真渲染（包括入射光照）始终需要阴影。定向光源可以正确处理阴影，即，处于光源阴影区域中的群集不接收任何光线。**Enlighten** 支持阴影聚光灯和点光源，但这在 Unity 5 中没有实施。因此，需要为聚光灯和点光源设置适当的范围，以避免非直接光源漏光。

#### 求解阶段

求解阶段将群集值乘以存储的形状因子的值进行求和，并将结果存储在光照贴图中。因此，其运行时性能与每个像素或探测器存储的形状因子数量直接关联。

#### 反弹阶段

反弹阶段从光照贴图读回值，再将光值弹回到群集中。这样，**Enlighten** 可模拟多重光线反弹。光线反弹次数限制为 **Enlighten** 中光照贴图更新速率。光照贴图更新速率通常由渲染更新速率决定。

光源、材质和预算算数据是此组件的输入值。光源和材质在运行时都可能会改变。这样，您便可以利用即时反馈细调场景的光照和外观。

您可以使用 **CPU 使用率**设置，控制运行求解器所需的线程数。默认值为 Low。如果要提高非直接光照的更新速率但不能优化您的场景（如设置适当的简化 UV 设置），您可以加大这个值。

下图显示了 **CPU 使用率**设置的下拉菜单。

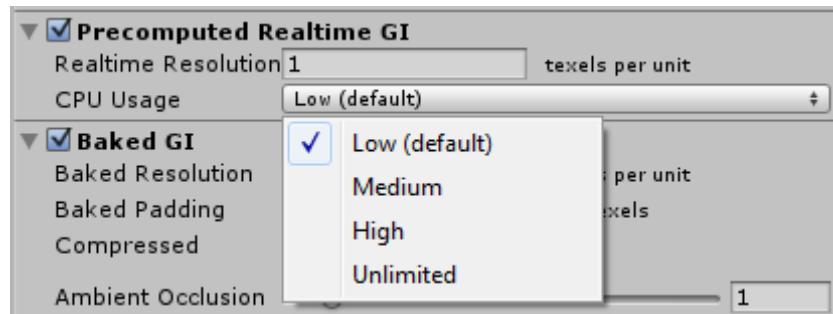


图 5-11 CPU 使用率设置

#### 5.2.4 烘焙光照贴图

Enlighten 可以生成包含直接光照、非直接光照和环境遮挡的烘焙光照贴图。非直接光照贴图基于实时结果生成，它们经过上采样和滤波生成高质量输出。

您也可以启用将烘焙光照贴图用作输入的最终收集阶段，Enlighten 将计算另一个最终光线反弹，其具有通过烘焙像素分辨率限定的精确度。这一步将标准的烘焙光照贴图用作输入，所以只有在您满意场景中的光照后，再考虑切换到最终收集。

实时光照和烘焙光照均可组合。在 Unity 中，这显示于光源级别的设置。两种类型的光照可以混合在一起，加入到非直接光照中。

下图显示了光照贴图烘焙设置：

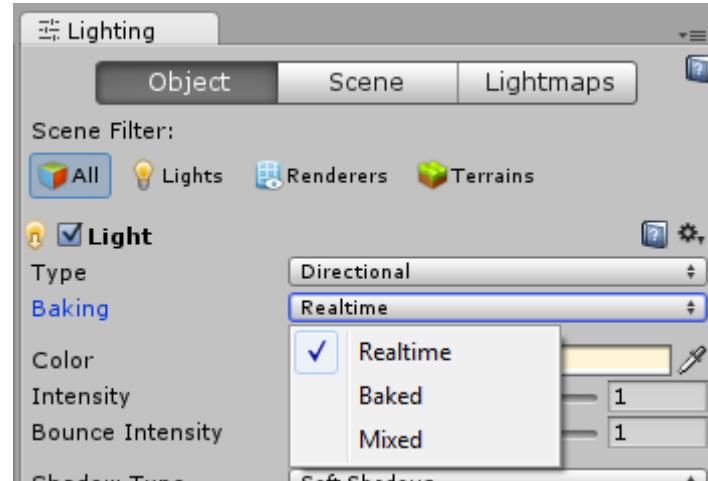


图 5-12 光照贴图烘焙设置

## 5.3 使用 *Enlighten* 设置场景

本节为如何以最佳方式在 Unity 中使用 *Enlighten* 设置场景提供一些通用的诀窍。

### 关闭光照贴图烘焙

为了获得光照的实时反馈，并且能够实现更加快速的迭代时间，请像仅使用实时全局光照那样设置您的场景。即使您仅计划使用烘焙光照贴图，也应这么做。为仅实时全局光照设置场景的一个优势是，您可以针对较强大的目标平台启用实时光照，对较不强大的目标平台使用烘焙光照贴图。

当您对自己的光照满意时，可为所有需要的光源打开光照贴图烘焙。*Enlighten* 同时负责实时光照和光照贴图烘焙，所以烘焙的光照贴图与实时渲染的结果匹配，其差别仅体现在软阴影和区域光源上。

### 设置 *Enlighten* 光照贴图分辨率

要优化的最重要指标是 *Enlighten* 光照贴图分辨率。在模拟的真实世界场景中，对于人类大小的角色，您通常将纹理分辨率设置为每米一个像素。较小尺度的细节最好通过屏幕空间环境遮挡处理。

### 优化预算算

在 Unity 中为 *Enlighten* 设置场景时，**UV 图块**视觉化模式通常是最佳的视图模式。当启动了预算算时，此模式中仅渲染网格。不过，封装是预算算的第一个阶段，而且 Unity 会在一个阶段完成时更新结果。

封装阶段可以很快完成计算。如果需要较长的时间后网格才会显示在 **UV 图表**渲染模式中，则其分辨率可能过高。要测试分辨率是否太高，可降低分辨率，再重新运行预算算。预算算将自动触发，除非您已将它停用。

按照要求调整了分辨率后，请等待预算算完成。如果注意到漏光等光照失真，请再次使用 **UV 图表**渲染模式。*Enlighten* 不会分割输入图表，穿过墙壁的图表可能会漏光。典型的示例是跨越多个房间但只有一个图表的地板网格。在此类情形中，可考虑分割并分离输入 UV 来创建较小的图表。

### 调整场景元素

完成预算算时，您可以添加光源，并获得总体场景外观的即时反馈。这不限于光源及其位置。

您也可以更改材质属性，如表面颜色、纹理或发光设置。

利用 *Enlighten* 时，任何表面可以设置为发光，因此可以转变为区域光源。这些区域光源具有无相关渲染成本的优点，因为所有光照都是由 *Enlighten* 完成的。对于动态光源数量非常有限的低端移动设备而言，这很有用处。针对发光表面使用对象面板中的**重要 GI**复选框，特别是它们比较小时，因为这可确保这些表面的群集和其他群集合并。

### 利用灯光探测器照明较小对象

为场景中较小的对象提供照明时，请考虑使用灯光探测器而非光照贴图。要使用灯光探测器，请勿选中**光照贴图静态**复选框。

通常而言，较小对象对全局光照的作用不大，把它们设置为通过灯光探测器照明，可以将它们从预算算阶段剔除，从而加快预算算的速度。

较小对象通常也比较难以为其生成 UV。您也可以将较小对象的网格与较大对象合并，例如 [5.2.2 预计算](#)（第 5-62 页）所用示例中的电视机和墙壁。

## 5.4 示例：冰穴演示的 Enlighten 设置

Unity 具有多种全局光照视觉化模式。本小节以冰穴场景为例，介绍这些模式。

下图显示了 Unity 全局光照视觉化菜单中列出的不同视觉化模式。

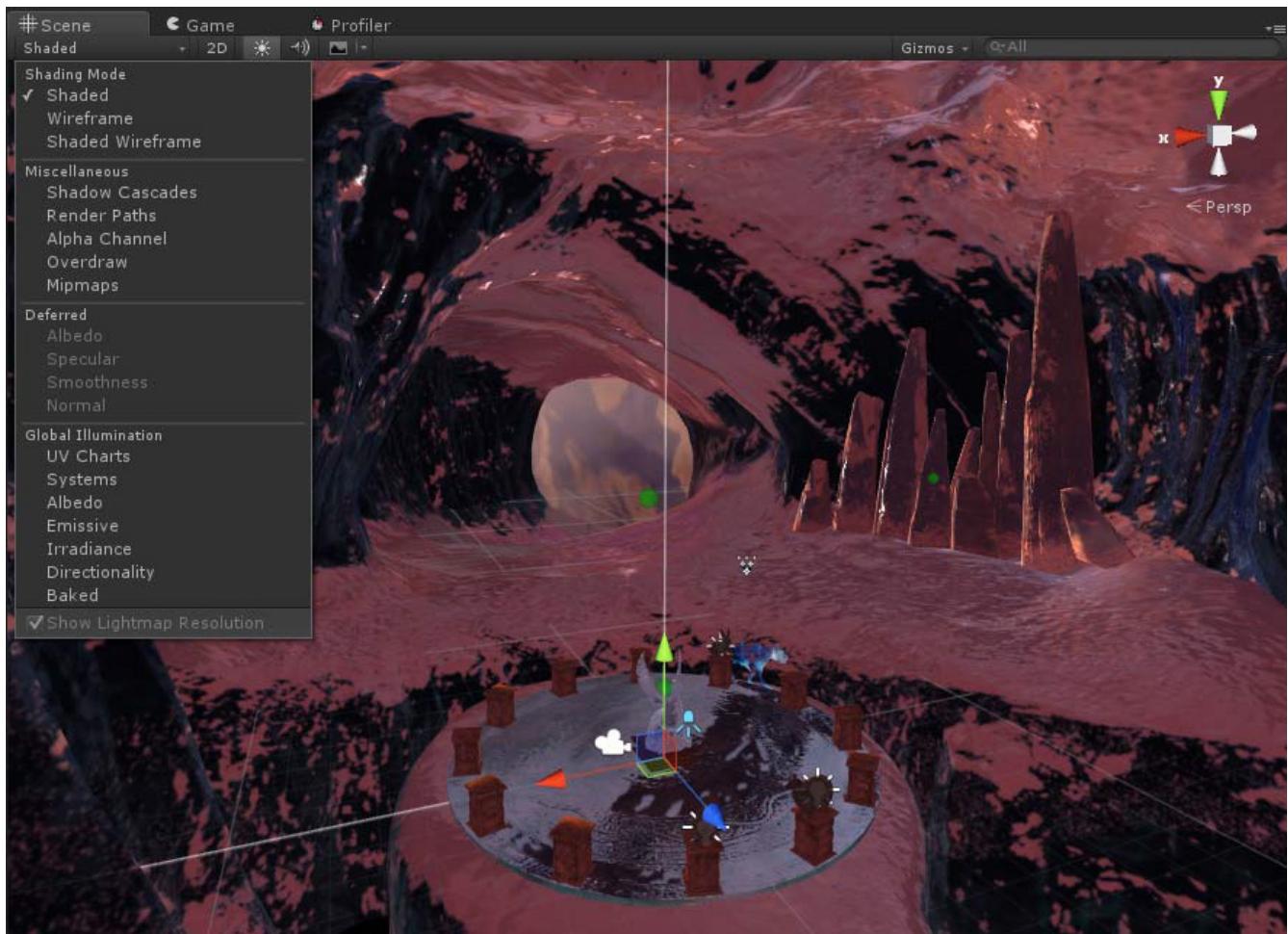


图 5-13 全局光照视觉化菜单

### UV 图表视图

建议将 **UV 图表** 视图用于验证您的像素大小和 UV 图表。与前面的示例场景相比，冰穴场景更加复杂，它在 **UV 图表** 中显示的网格具有多个图表和锐利边界。例如，可参见洞穴顶部。为避免最终渲染的图像中出现任何可见的裂缝，Enlighten 尝试自动拼接这些边界。

#### 备注

拼接的结果可以在**辐照度**和**定向**视图中查看。不过，最终图像中裂缝的可见度比较低，因为应用了材质纹理。因此，不要高估这两个视图中的可见裂缝。理想状态下，您可以通过将它们移到网格中可见性较低的部分来隐藏裂缝。

下图显示了 **UV 图块** 视图中的冰穴 UV 部分。

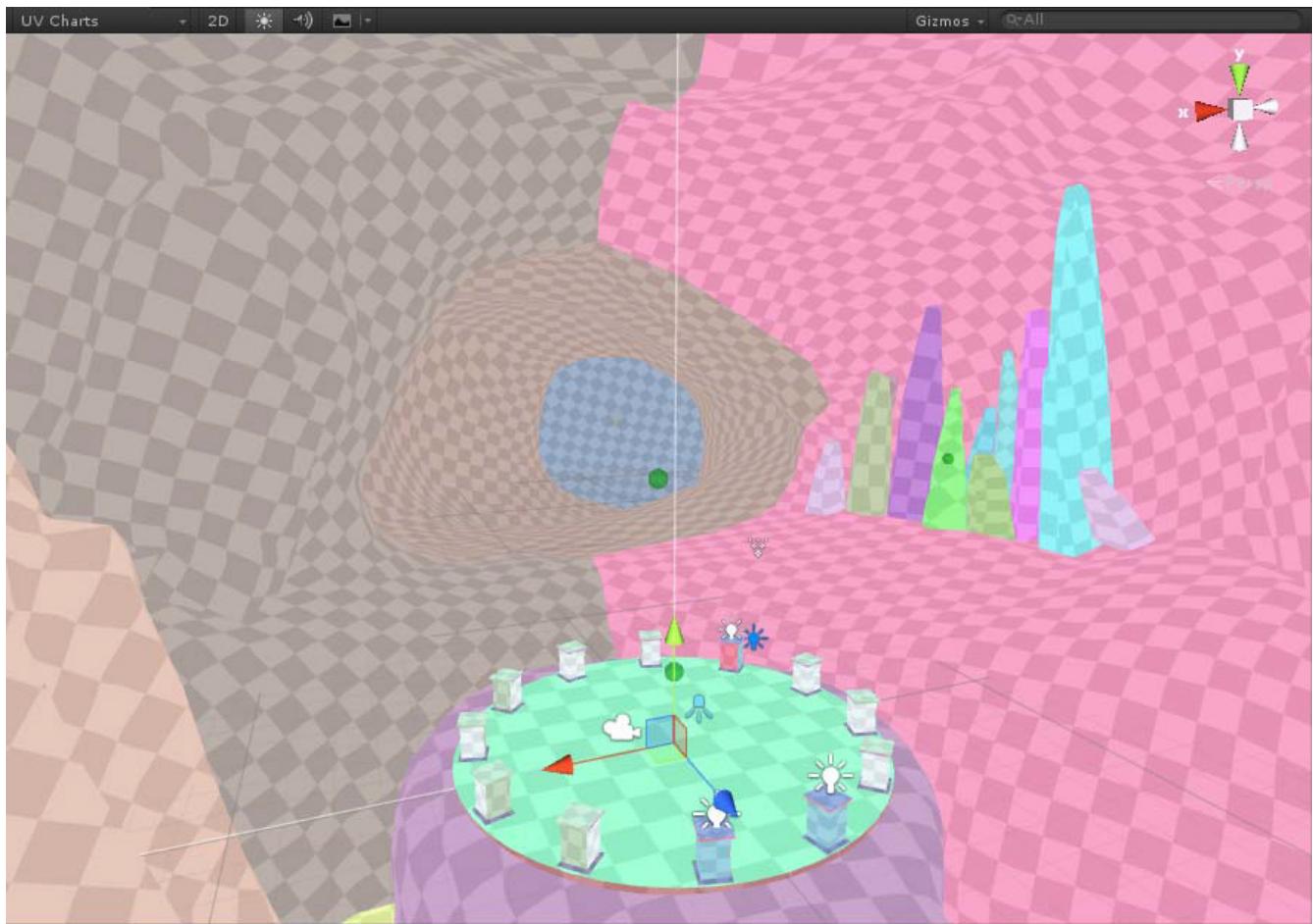


图 5-14 冰穴 UV 图块

### 系统视图

系统视图显示了由 Enlighten 自动生成的系统。多个系统可以实现并行执行预算算和运行时。通常而言，自动生成效果不错，不需要更改任何设置。

在下图中，每一种颜色代表一个系统。每个系统可以包含多个对象。

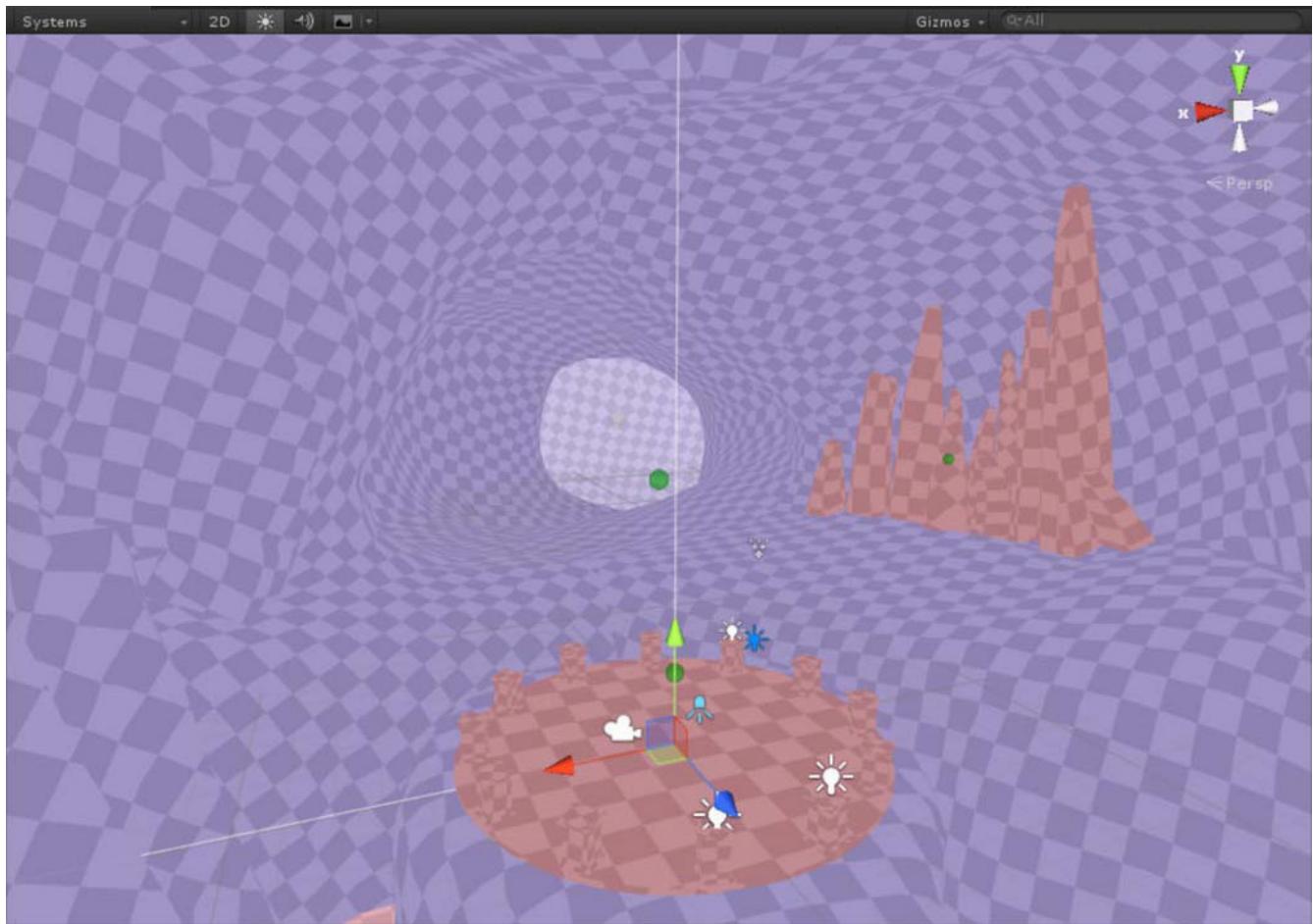


图 5-15 冰穴系统

### 反射视图

反射视图显示用漫射颜色渲染的光照贴图像素，Enlighten 将这种漫射颜色用于场景中反弹或发出的光线。它有效地显示了 Enlighten 使用来自您的场景的颜色信息的细致程度。

下图演示了反射是反射贴图和反射颜色的平均值。

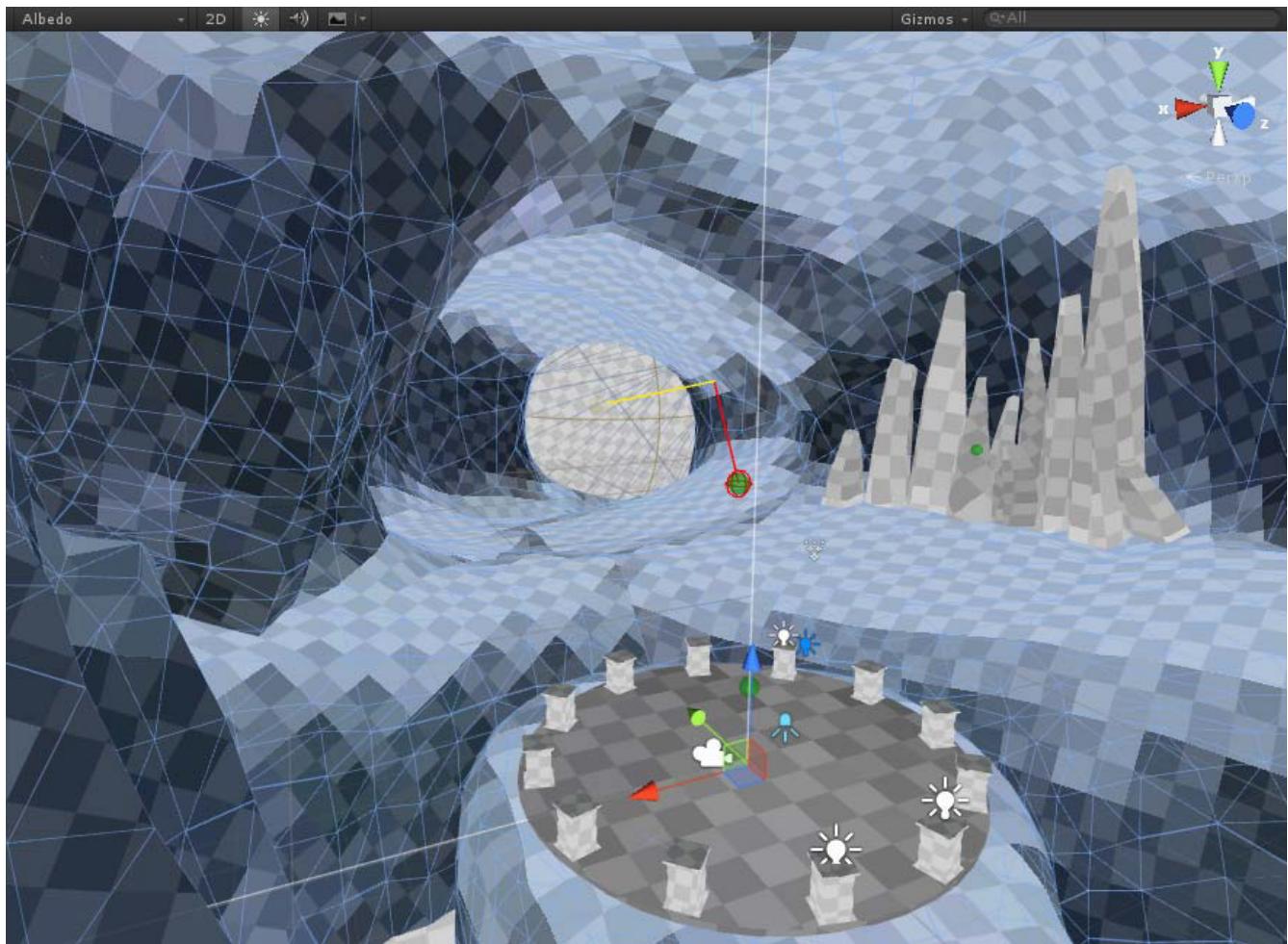


图 5-16 冰穴反射

### 发光视图

发光视图类似于反射视图，但它显示的是从发光对象发出的光线的颜色和强度。您可以借助 Enlighten 使用发光对象创建无渲染成本的区域光源。

下图演示了发光值是发光贴图和发光颜色的平均值。例如，蓝色晶体和绿色平台是发光的。

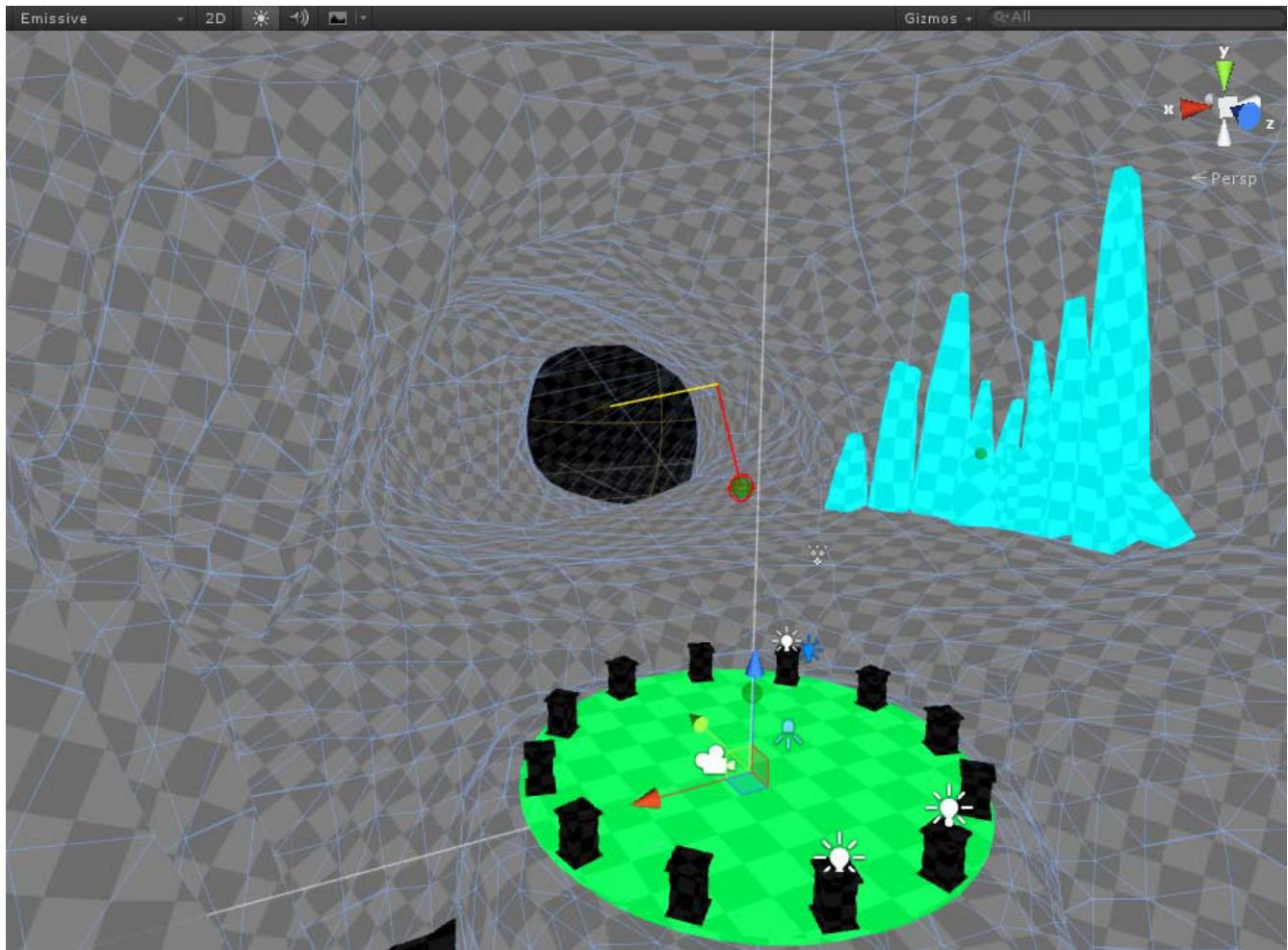


图 5-17 冰穴发光

#### 辐照度视图

辐照度视图仅显示从利用光照贴图的表面收到的辐照度。此视图有助于确定光照或场景设置问题，因为通常应用了纹理的标准渲染视图难以识别确切的问题。

下图显示的辐照度视图具有来自萤火虫和晶体背后光源的光照。

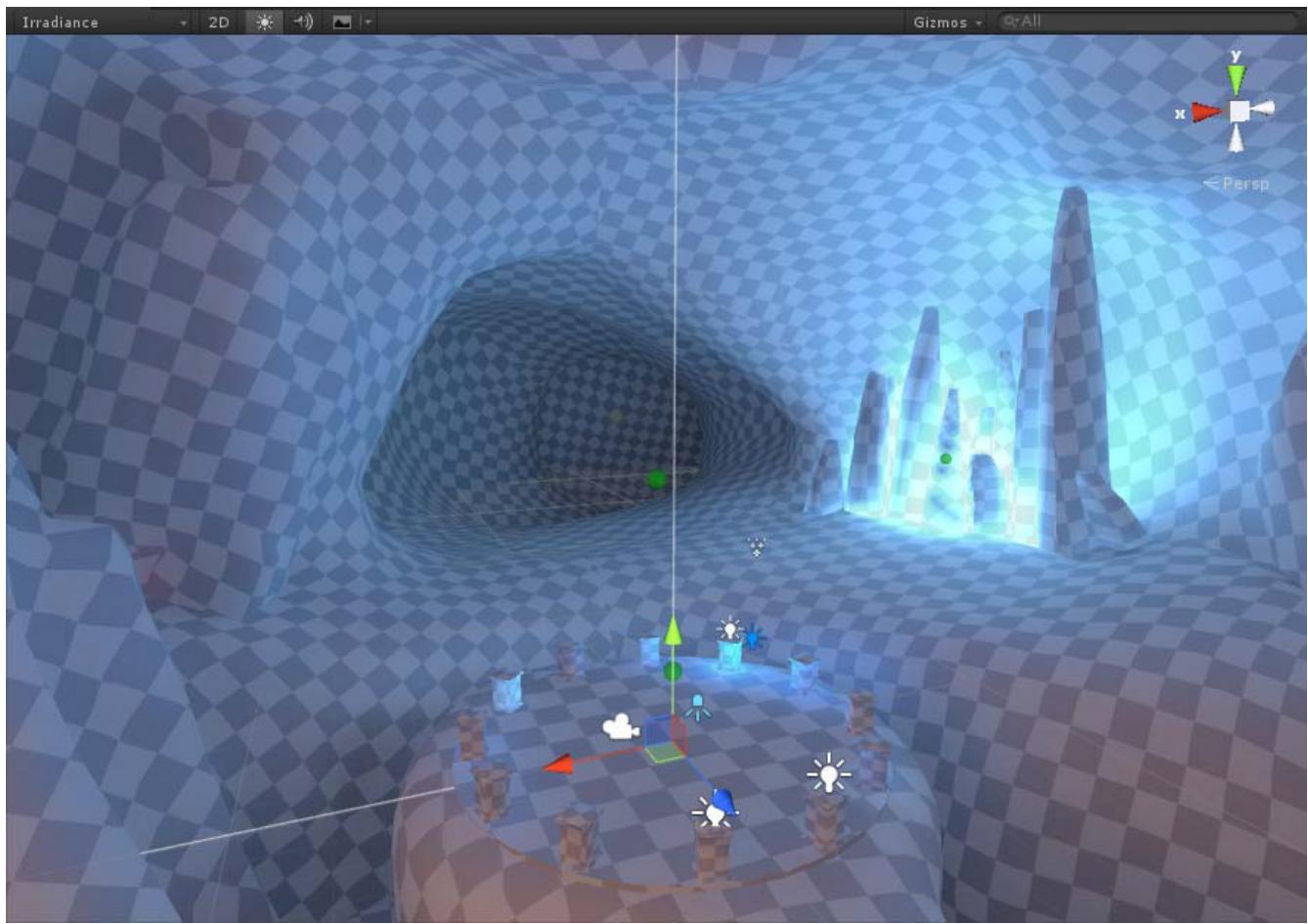


图 5-18 冰穴辐照度

### 定向视图

定向视图显示各个像素的主要光源方向。若选中了定向或定向镜面选项，将使用此视图。与辐照度视图一样，更改光源位置将改变此贴图。

在下图中，场景的底部主要接收来自洞穴上方太阳的光线。绿色表示这在 Y 轴上是正值。定向在晶体附近有所变化，因为其主要光源位于晶体背后。

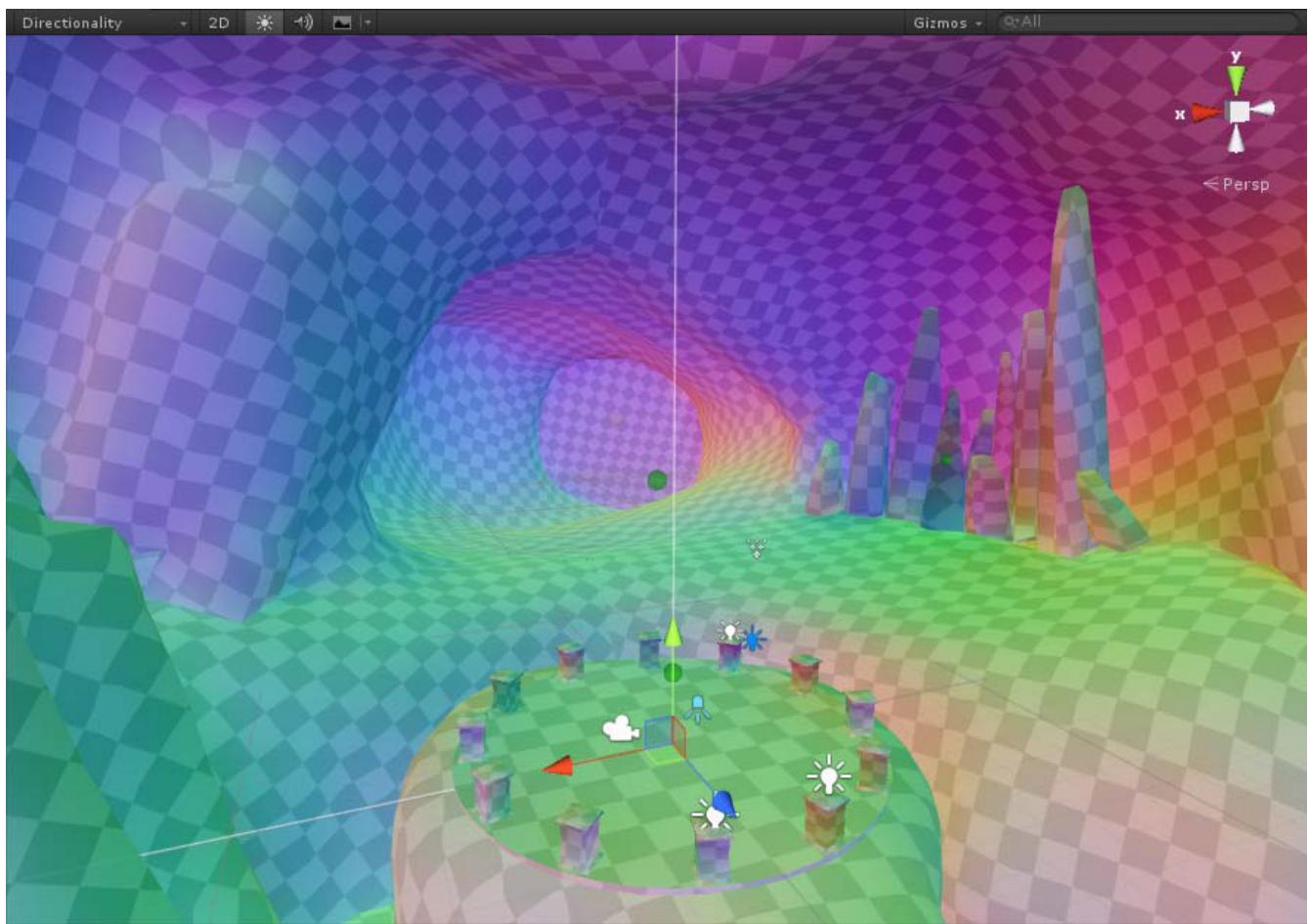


图 5-19 冰穴定向

您也可以将这些贴图视觉化为每个对象的 2D 纹理。选择一个对象，然后在“光照”选项卡中选择“对象”开关。在选项卡底部的“预览”窗口中，您可以看到已选中了该对象所指的相同图表。

## 5.5 在自定义着色器中使用 *Enlighten*

本节介绍在自定义着色器中使用 *Enlighten*。此为高级主题，只有您希望完全替换着色器并且依然要利用 *Enlighten* 光照贴图和灯光探测器输出时才需要阅读。

本节包含以下小节：

- [5.5.1 全局光照代码流程（第 5-80 页）](#)。
- [5.5.2 自定义着色器代码（第 5-81 页）](#)。
- [5.5.3 \*Enlighten.cginc\*（第 5-82 页）](#)。

### 5.5.1 全局光照代码流程

本节介绍由标准着色器前向渲染器执行的代码流程。

#### 顶点着色器

在顶点着色器中，您必须设置光照贴图的纹理坐标，以便能访问全局光照数据。

使用 `vertForwardBase()` 函数设置纹理坐标，此函数位于 `UnityStandardCore.cginc` 文件中。它计算静态和动态光照贴图的纹理坐标，并基于球谐函数计算各顶点环境颜色。

#### 片段着色器

计算全局光照的主要函数是 `UnityGlobalIllumination()`。`UnityGlobalIllumination()` 在 `UnityGlobalIllumination.cginc` 文件中。

下列代码演示了 Unity 标准着色器使用前向渲染时执行的调用序列：

```
Pass Forward:Standard.shader ->
    fragForwardBase (UnityStandardCore.cginc) ->
        FragmentGI (UnityStandardCore.cginc)->
            UnityGlobalIllumination (UnityGlobalIllumination.cginc)
```

`fragForwardBase()` 是了解元素组合方式的起点。

函数 `fragForwardBase()` 初始化 `FragmentCommonData` 结构。此结构包含法线、位置、眼睛向量和其他值的正确值。这些值基于材质的设置，如视差贴图和法线贴图。

`fragForwardBase()` 函数计算阴影衰减因子，以及基于阴影和遮挡贴图的遮挡因子。用于进行此计算的值基于构建平台的着色器模型。

`fragForwardBase()` 函数使用下列函数访问全局光照数据：

```
UnityGI gi = FragmentGI (s.posWorld, occlusion, i.ambientOrLightmapUV, atten,
s.oneMinusRoughness, s.normalWorld, s.eyeVec, mainLight);
```

`FragmentGI()` 函数初始化：

- 具有作为参数传递的信息的 `UnityGIInput` 结构。
- 用于访问光照贴图的 UV 坐标。另外，如果不使用光照贴图，它将初始化环境颜色。
- 访问反射立方体贴图所需的数据。

`FragmentGI()` 函数调用 `UnityGlobalIllumination()` 函数。

`UnityGlobalIllumination()` 函数检查当前的对象，若有必要，它将从灯光探测器采样非直接光照。

探测器将数据存储为球谐函数。着色器模型定义所使用的最大球面谐波阶。阶越高，质量越好，但计算成本也更高。

探测器通常用于使用非直接光源为动态对象照明。

其结果存储为非直接漫射颜色。对于从灯光探测器采样的对象，静态或动态光照贴图将被忽略。

如果对象不使用任何光照贴图，则代码存储对象所接收的直接光照的颜色。

如果对象使用静态光照贴图，它们将被采样。光照贴图的类型取决于全局光照的类型：

- 非定向。
- 定向。
- 定向镜面。

这些类型的全局光照也可用于动态光照贴图。这些光照贴图在运行时由 *Enlighten* 更新。

静态和动态光照贴图都修改此函数返回的非直接漫射颜色的值。

`UnityGlobalIllumination()` 函数从单一镜面立方体贴图采样反射，或者插值来自两个镜面立方体贴图的反射，从而设置非直接镜面颜色。

`fragForwardBase` 代码使用 `UNITY_BRDF_PBS` 计算像素位置上的颜色。

此宏由依赖于所用着色器模型的各种函数定义。

此宏在计算像素颜色时会依据材质属性、直接光照，以及在 `fragForwardBase()` 函数中计算的非直接光照。

`UNITY_BRDF_GI()` 函数计算来自静态和动态光照贴图的光的作用量。

`UNITY_BRDF_GI()` 函数读取 `UnityGI` 结构的 `gi.light2` 和 `gi.light3` 字段。这些字段包含关于从定向和定向镜面全局光照的光照贴图获取的光的位置和强度的信息。

`fragForwardBase()` 函数将发光因子用应用到像素。

`fragForwardBase()` 函数应用雾颜色。

## 5.5.2 自定义着色器代码

您可以修改 `Unity` 标准着色器，使其包含定向全局光照。

本节介绍修改为包含定向全局光照的 `Unity` 标准着色器示例。示例着色器不包含使用球谐函数的动态对象的镜面光照或全局光照。它基于 `Unity 5.0`。

### Enlighten 纹理坐标

*Enlighten* 和 Unity 为所有标记为静态的对象生成一组额外的 UV 坐标，用于访问它们更新的光照贴图。

它将这些坐标作为 TEXCOORD1（静态光照贴图）和 TEXCOORD2（动态光照贴图）传递。

您必须将它们从顶点着色器拷贝到片段着色器。

下列代码演示了如何将这一输入添加到您的自定义着色器：

```
struct vertexInput {
    half4 vertex : POSITION;
    half2 texcoord : TEXCOORD0;
    half2 uv1 : TEXCOORD1; //Static Lightmap texture coordinates
    half2 uv2 : TEXCOORD2; //Dynamic Lightmap texture coordinates
    half3 normal : NORMAL;
};
```

### 使用变量的正确名称

*Enlighten* 构建运行时使用的反射和发光贴图。当 *Enlighten* 进行此操作时，它访问所用材质的属性。

这些属性必须正确命名。否则，所构建的贴图的颜色会出错。

确保您的自定义着色器属性的名称与标准着色器中的名称匹配。

若要下载标准着色器文件，请访问：<https://unity3d.com/get-unity/download/archive>.

下载与您的操作系统和 Unity 版本对应的内置着色器文件。

标准着色器名为 *Standard.shader*

下列代码显示了标准着色器中声明的纹理名称。此纹理名称供 *Enlighten* 用于使用材质反射信息更新其光照贴图：

```
_Color("Color", Color) = (1, 1, 1, 1)
_MainTex("Albedo", 2D) = "white" {}
```

### 插入正确的宏

在运行时，Unity 定义 *multi\_compile* 宏。这些宏指定对象是否使用 *Enlighten*，以及如何采样生成的数据。例如，它们可以指定动态对象访问灯光探测器，而不访问 *Enlighten* 数据。

下列代码演示了宏的定义：

```
//Those multi_compile options are set automatically by unity.
#pragma multi_compile LIGHTMAP_OFF LIGHTMAP_ON
#pragma multi_compile DIRLIGHTMAP_OFF DIRLIGHTMAP_COMBINED DIRLIGHTMAP_SEPARATE
#pragma multi_compile DYNAMICLIGHTMAP_OFF DYNAMICLIGHTMAP_ON
```

### 顶点着色器

顶点着色器使用一个实用程序函数检索数据：

```
output.enlightenData = GetEnlightenInput(output.vertexInWorld, output.normalInWorld,
                                         input.uv1, input.uv2);
```

此函数在代码中描述，请参见 [5.5.3 \*Enlighten.cginc\* \(第 5-83 页\)](#)。

### 片段着色器

片段着色器采样非直接光照作用量，并将它用作环境颜色：

```
//If the shader uses enlighten, the ambient term is embedded in the lightmap. Do not
//add it again
ambient = texColorTerm * SampleIndirectContribution(input.enlightenData,
normalInWorld, 1.0);
```

## 5.5.3 *Enlighten.cginc*

*Enlighten.cginc* 是一个实用程序文件，它将全局光照与自定义着色器相集成。

下列代码定义了您可在着色器中使用的 GetEnlightenInput() 和 SampleIndirectContribution() 函数：

```
#ifndef ENLIGHTEN_INCLUDED
#define ENLIGHTEN_INCLUDED

#include "UnityShaderVariables.cginc"
#include "UnityStandardBRDF.cginc"
#include "UnityPBSLighting.cginc"

struct enlightenInput
{
    half4 posWorld;
    half4 ambientOrLightmapUV;
    half3 eyeVec;
    half3 normalWorld;
};

enlightenInput GetEnlightenInput(half4 posWorld, half3 normalWorld, half2 uvCoord1,
half2 uvCoord2)
{
    enlightenInput o;

    // Static lightmap texture coordinates
    o.ambientOrLightmapUV.xy = uvCoord1.xy * unity_LightmapST.xy + unity_LightmapST.zw;

    // Dynamic lightmap texture coordinates
    o.ambientOrLightmapUV.zw = uvCoord2.xy * unity_DynamicLightmapST.xy +
unity_DynamicLightmapST.zw;

    o.posWorld = posWorld;
    o.eyeVec = normalize(posWorld.xyz - _WorldSpaceCameraPos);
    o.normalWorld = normalWorld;

    return o;
}

half3 SampleIndirectContribution(enlightenInput i, half3 normalWorld, half
shadowAttenuation)
{
    half3 output = half3(0.0,0.0,1.0);
    #ifdef ENLIGHTEN_ON
    //First Light

    // Do not consider any attenuation or occlusion
    half atten = 0.0;
    half occlusion = 1.0;
    UnityGI gi = SimplifiedFragmentGI (i.posWorld, occlusion, i.ambientOrLightmapUV,
atten, normalWorld, i.eyeVec);

    return half3(gi.indirect.diffuse);
    #elif defined (ENLIGHTEN_OFF)
        return output;
    #else
        #error You must include "#pragma multi_compile ENLIGHTEN_OFF ENLIGHTENS_ON" in your
shader to allow projectors to be enabled and disabled
    #endif
    #endif
}
```

SimplifiedFragmentGI() 是 UnityStandardCore.cginc 文件内的简化版 FragmentGI() 函数。

此版本不使用立方体贴图和灯光探测器所需的数据，因此这些部分已被去除。在这一函数中，对 UnityStandardGlobalIllumination() 的调用被修改为指向它的简化版本。UnityStandardGlobalIllumination() 包含在 UnityPBSLighting.cginc 中，它含有用于采样各种光照贴图、灯光探测器和立方体贴图的代码。

# 第 6 章

## 高级图形技术

本章节列出了您可以利用的多种高级图形技术。

它包含下列部分：

- 6.1 自定义着色器（第 6-85 页）。
- 6.2 使用局部立方体贴图实现反射（第 6-98 页）。
- 6.3 组合反射（第 6-114 页）。
- 6.4 基于局部立方体贴图的动态软阴影（第 6-120 页）。
- 6.5 基于局部立方体贴图的折射（第 6-128 页）。
- 6.6 冰穴演示中的镜面反射效果（第 6-134 页）。
- 6.7 使用 Early-z（第 6-137 页）。
- 6.8 脏镜头光晕效果（第 6-138 页）。
- 6.9 光柱（第 6-141 页）。
- 6.10 雾化效果（第 6-145 页）。
- 6.11 高光溢出（第 6-152 页）。
- 6.12 冰墙效果（第 6-159 页）。
- 6.13 过程天空盒（第 6-165 页）。
- 6.14 萤火虫（第 6-173 页）。
- 6.15 正切空间至世界空间法线转换工具（第 6-177 页）。

## 6.1 自定义着色器

本节介绍自定义着色器。

本节包含以下小节：

- [6.1.1 关于自定义着色器（第 6-85 页）。](#)
- [6.1.2 着色器结构（第 6-86 页）。](#)
- [6.1.3 编译指令（第 6-88 页）。](#)
- [6.1.4 `include` 语句（第 6-88 页）。](#)
- [6.1.5 OpenGL ES 3.0 图形流水线（第 6-89 页）。](#)
- [6.1.6 顶点着色器（第 6-90 页）。](#)
- [6.1.7 顶点着色器输入（第 6-91 页）。](#)
- [6.1.8 顶点着色器输出和变量（第 6-91 页）。](#)
- [6.1.9 片段着色器（第 6-93 页）。](#)
- [6.1.10 向着色器提供数据（第 6-93 页）。](#)
- [6.1.11 调试 Unity 中的着色器（第 6-95 页）。](#)

### 6.1.1 关于自定义着色器

Unity 5 和更高版本包含基于物理的着色 (**PBS**) 模型，它模拟材质与光线之间的交互。这可提供很高度的真实感，并且能够在不同光照条件下获得一致的外观。

PBS 可以轻松搭配标准着色器使用。如果自行创建材质，它会被自动分配标准着色器。

您可以轻松访问标准着色器。如果自行创建材质，它将被分配标准着色器。有许多对初学者很有帮助的内置着色器。您可以在检视面板中单击**着色器**下拉菜单，查看所有按系列划分的可用内置着色器。

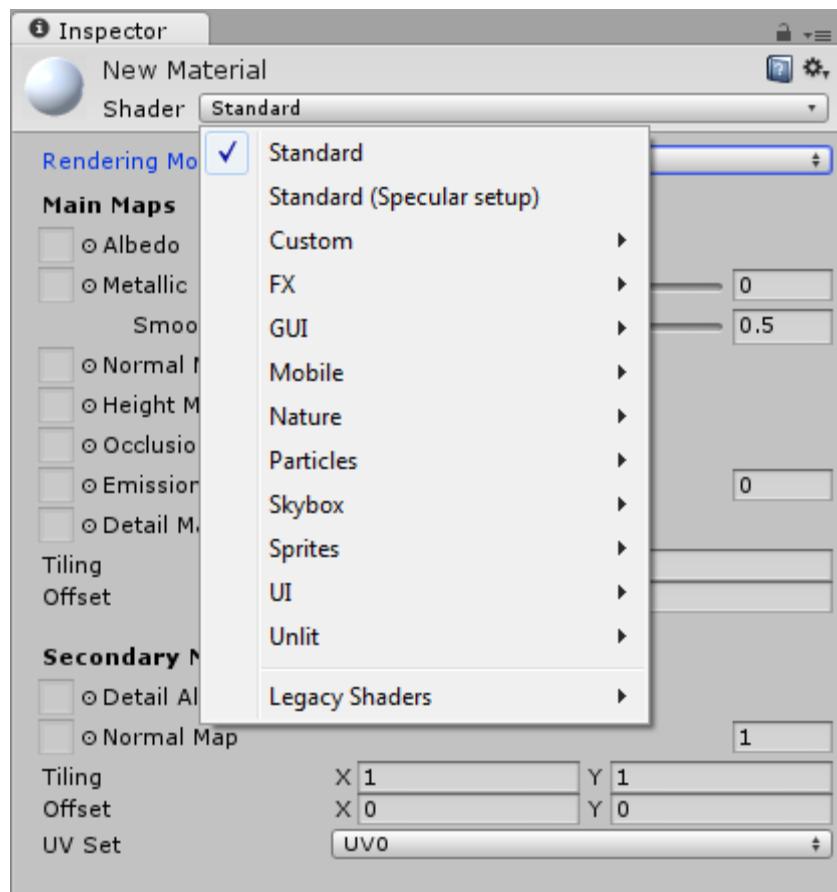


图 6-1 Unity 内置着色器

内置着色器的源代码可在 Unity 下载存档中找到，其网址为 <http://unity3d.com/>，该存档涵盖了 120 多个着色器。您可以通过阅读并尝试理解这些着色器代码来了解更多信息。

除这些以外，许多效果无法通过使用现有着色器实现。例如，根据局部立方体贴图实现反射的着色器。有关更多信息，请参见 [6.2 使用局部立方体贴图实现反射（第 6-98 页）](#)。

在 Unity 中，编写着色器的方法通常有两种：

#### 表面着色器

着色器受到光线和阴影影响时，通常使用表面着色器。Unity 为您执行与光照模型相关的作业，能够让你编写更紧凑的着色器。

#### 顶点和片段着色器

顶点和片段着色器最为灵活，但是您必须执行所有事项。Unity **ShaderLab** 比顶点和片段着色器的功能更多，但它们是图形流水线的主要编程部分，您需要在图形流水线中完成所有着色。因此，了解如何编写自定义顶点和片段着色器非常重要。

## 6.1.2 着色器结构

下列代码显示了一个非常简单的顶点和片段着色器，它包含顶点或片段着色器所需的大部分元素。

着色器示例以 Cg 编写。Unity 还支持适用于着色器片段的 HLSL 语言。

```
Shader "Custom/ctTextured"
{
    Properties
```

```

    _AmbientColor ("Ambient Color", Color) = (0.2,0.2,0.2,1.0)
    _MainTex ("Base (RGB)", 2D) = "white" {}

}

SubShader
{
    Pass
    {
        CGPROGRAM
        #pragma target 3.0
        #pragma glsl
        #pragma vertex vert
        #pragma fragment frag

        #include "UnityCG.cginc"

        // User-specified uniforms
        uniform float4 _AmbientColor;
        uniform sampler2D _MainTex;

        struct vertexInput
        {
            float4 vertex : POSITION;
            float4 texCoord : TEXCOORD0;
        };
        struct vertexOutput
        {
            float4 pos : SV_POSITION;
            float4 tex : TEXCOORD0;
        };

        // Vertex shader.
        vertexOutput vert(vertexInput input)
        {
            vertexOutput output;

            output.tex = input.texCoord;
            output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
            return output;
        }

        // Fragment shader.
        float4 frag(vertexOutput input) : COLOR
        {
            float4 texColor = tex2D(_MainTex, float2(input.tex));
            return _AmbientColor + texColor;
        }
    ENDCG
}
}

Fallback "Diffuse"
}

```

首个关键字是着色器路径/名称前的“**Shader**”。当您设置材质时，路径用于定义下拉菜单中显示着色器的类别。此示例中的着色器显示在下列菜单中的**自定义**着色器类别下方。

**Properties {}** 块列出了在检视面板中可见的着色器参数，以及您可以进行交互的参数。

Unity 中的每个着色器均包含一系列子着色器。当 Unity 渲染网格时，它将查找要使用的着色器，并选择可在图形卡上运行的第一个子着色器。通过这种方式，可以在支持不同着色器型号的不同图形卡上正确运行着色器。此特性非常重要，因为 GPU 硬件和 API 正在不断发展。例如，您可以 Mali Midgard GPU 为目标编写主着色器，从而利用 OpenGL ES 3.0 的最新特性，并同时在单独的子着色器中为支持 OpenGL ES 2.0 及更低版本的图形卡编写替代着色器。

**Pass** 块会使对象的几何结构一次性渲染。着色器可包含一个或多个 **pass** 参数。您可在旧硬件上使用多个 **pass** 参数，或者用其实现各种特效。

如果 Unity 无法在能够正确渲染几何结构的着色器主体中找到子着色器，则它将回滚到另一个在 **Fallback** 语句后定义的着色器。在此示例中，着色器为“漫反射”内嵌着色器。

Cg 程序片段在 CGPROGRAM 和 ENDCG 之间编写。

### 6.1.3 编译指令

您将编译指令作为 #pragma 语句传递。编译指令可指示待编译的着色器功能。

每个编译指令必须至少包含一个用于编译顶点和片段着色器的指令：#pragma vertex name, #pragma fragment name。

默认情况下，Unity 将着色器编译为着色器模型 2.0。#pragma target 指令可将着色器编译为其他能力级别。如果着色器变得较大，您将得到下列类型的错误：

```
Shader error in 'Custom/MyShader': Arithmetic instruction limit of 64 exceeded; 83 arithmetic instructions needed to compile program;
```

如果出现这种情况，您必须添加 #pragma target 3.0 语句，将着色器模型 2.0 更改为着色器模型 3.0。着色器模型 3.0 拥有更高的指令限制。

如果将多个变量从顶点着色器传递至片段着色器，可能会得到下列错误：

```
Shader error in 'Custom/MyShader': Too many interpolators used (maybe you want #pragma glsl?) at line 75.
```

如果出现这种情况，请添加编译指令 #pragma glsl。该指令可将 Cg 或 HLSL 代码转换成 GLSL。

```
#pragma only_renderers
```

Unity 支持多种渲染平台，如 gles、gles3、opengl、d3d11、d3d11\_9x、xbox360、ps3 和 flash。默认情况下，着色器可在所有上述平台上编译，除非您使用 #pragma only\_renderers 指令明确限制此数字，该指令后跟有渲染器 API 且两者之间留有空格。

如果您以移动设备为目标，则只需将着色器编译限制为 gles 和 gles3。您还必须添加 Unity 编辑器使用的 opengl 和 d3d11 渲染器：

```
#pragma only_renderers gles gles3 [opengl, d3d11]
```

### 6.1.4 include 语句

可以在着色器中添加 **Include** 文件以利用 Unity 预定义的变量和辅助函数。

您可以在 C:\Program Files\Unity\Editor\Data\CGIncludes 中找到可用 **include** 语句。例如，在 **include** 语句 UnityCG.cginc 中，您可找到若干个用于许多标准着色器的辅助函数和宏。若要使用它们，请在着色器中声明 **include**。

有许多 Unity 内置变量可用于着色器。它们位于 **include** 文件 UnityShaderVariables.cginc 中。您不需要将此文件包含在着色器中，因为 Unity 会自动执行此操作。多个有用的转换矩阵和幅度可直接用于着色器。必须了解上述内容，以避免重复工作。例如，在考虑如何将矩阵传递至着色器、摄像机位置、投射参数或灯光参数前，请先检查 **include** 语句是否已提供矩阵。

为提高性能，有时更偏向于在 CPU 中进行运算并将结果传递至 GPU，而非在顶点着色器中对每个顶点进行运算。例如，矩阵统一变量乘法运算就是这种情况。这就是为什么 Unity 可用作统一多个复合矩阵的内置工具。下表显示了一些重要的 Unity 着色器内置值：

表 6-1 重要的 Unity 着色器内置值

内置统一变量	说明
UNITY_MATRIX_V	当前视图矩阵
UNITY_MATRIX_P	当前投影矩阵
Object2World	当前模型矩阵
_World2Object	当前世界矩阵的逆矩阵
UNITY_MATRIX_VP	当前视图 * 投影矩阵
UNITY_MATRIX_MV	当前模型 * 视图矩阵
UNITY_MATRIX_MVP	当前模型 * 视图 * 投影矩阵
UNITY_MATRIX_IT_MV	当前模型 * 视图矩阵的逆转置矩阵
_WorldSpaceCameraPos	世界坐标空间的摄像机位置
_ProjectionParams	作为向量分量的近平面和远平面以及 $1/farPlane$
_Time	向量中的当前时间和分段 ( $t/20$ 、 $t$ 、 $t^2$ 和 $t^3$ )

### 6.1.5 OpenGL ES 3.0 图形流水线

知道可编程顶点着色器和片段着色器在图形流水线中的位置十分重要。

下图显示了 OpenGL ES 3.0 图形流水线流程的示意图。OpenGL ES 3.0 是嵌入式图形演变的重大突破，在 OpenGL 3.3 规范的基础上衍生而来。

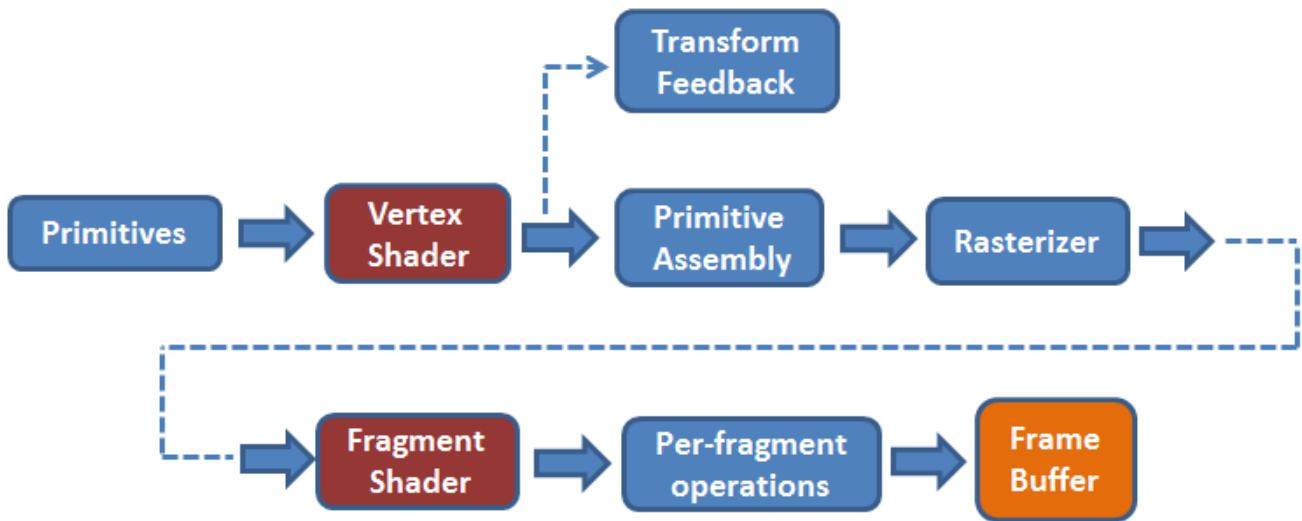


图 6-2 OpenGL ES 3.0 可编程流水线

#### 原语

在原语阶段，流水线在通过顶点、点、线和多边形描述的几何原语上运行。

#### 顶点着色器

顶点着色器采用通用编程方法在顶点上进行运算。顶点着色器转换并照亮顶点。

### 原语汇编

在原语汇编期间，顶点将汇编至几何原语中。产生的原语将剪切至裁剪区域并发送至光栅化程序。

### 光栅化

针对每个生成的片段计算顶点着色器的输出值。该流程称为插值。光栅化过程中，原语将转换为一组二维片段，这些片段将被发送至片段着色器。

### 转换反馈

转换反馈能够将所选编写内容写入顶点着色器输出的输出缓冲区，然后再发送至顶点着色器。**Unity** 不会显示此特性，但是它通常用于在内部优化人物外观。

### 片段着色器

片段着色器采用通用编程方法在片段被发送至下一阶段前对其进行运算。

### 逐片段操作

在逐片段操作中，可以在每个片段上应用多项功能和测试：像素所有权测试、裁剪测试、模板和深度测试以及混合和抖动。因此，在逐片段阶段，片段将被弃置或者片段颜色、模板或深度值将写入屏幕坐标中的帧缓冲区。

## 6.1.6 顶点着色器

顶点着色器示例针对几何体的每个顶点运行一次。顶点着色器旨在将对象局部坐标中给定的每个顶点的 3D 位置转换为屏幕空间中的投影 2D 位置，并计算 Z 缓冲区的深度值。

有关顶点着色器示例代码，请参阅 [6.1.2 着色器结构（第 6-86 页）](#)。

转换位置预期出现在顶点着色器的输出中。如果顶点着色器未返回值，控制台将显示下列错误：

```
Shader error in 'Custom/ctTextured': '' : function does not return a value: vert  
at line 36
```

在此示例中，顶点着色器接收局部空间中的顶点坐标和纹理坐标以作为输入。顶点坐标通过作为 **Unity** 内置值的模型视图投影矩阵 `UNITY_MATRIX_MVP` 从局部空间转换至屏幕空间：

```
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
```

纹理坐标作为变量传递至片段着色器，但是这并不意味着它们未被转换。

法线将以其他方式从对象空间转换到世界空间。为保证在执行不均匀缩放操作后，法线仍为三角形的法线，必须乘以转换矩阵逆矩阵的转置。若要应用转置操作，请在乘法算数中颠倒乘数因子的顺序。局部矩阵到世界矩阵的逆矩阵是内置 `World2Object` **Unity** 矩阵。它是  $4 \times 4$  矩阵，因此您必须通过 3 个分量的法线输入向量构建 4 个分量。

```
float4 normalWorld = mul(float4(input.normal, 0.0), _World2Object);
```

构建四个分量时，您要添加零作为第四个分量。这对于在第四个维度空间中正确处理矢量转换必不可少；而对于坐标而言，第四个分量必须是一个单位。

如果世界坐标中已提供法线，则可以跳过法线转换流程。这将节省顶点着色器的工作量。如果对象网格可能由 **Unity** 内置着色器进行处理，则避免此提示，因为此种情况下，法线将出现在对象坐标中。

大多数图形效果在片段着色器中运行，但是您也可以在顶点着色器中运行一些效果。顶点置换贴图也称为置换贴图，是一项众所周知的技术，能够让您使用纹理对多边形网格进行变形，从而增加表面细节，例如在地形生成过程中使用高度贴图。若要在顶点着色器中获取此纹理，也称之为置换贴图，您必须添加编译指令 `#pragma target 3.0`，因为它仅位于着色器模型 3.0 中。根据着色器模型 3.0，至少必须在顶点着色器内访问 4 个纹理单元。如果您强制编辑器使用 OpenGL 着色器，则还必须添加 `#pragma glsl` 指令。如果您未声明此指令，产生的错误消息会建议执行此操作：

```
Shader error in 'Custom/ctTextured': function "tex2D" not supported in this profile  
(maybe you want #pragma glsl?) at line 57
```

在顶点着色器中，您还可以使用“过程动画”技术将顶点做成功画。您可将支持您修改顶点坐标的着色器中的时间变量用作时间函数。网格蒙皮是另一种在顶点着色器中实现的功能。Unity 使用此技术，将与人物骨骼相关的网格的顶点做成功画。

### 6.1.7 顶点着色器输入

顶点着色器的输入和输出借助结构定义。在此示例的输入结构中，您仅发布了顶点属性位置和纹理坐标。

您可以使用下列语义定义更多的属性作为输入，例如另一组纹理坐标、对象坐标中的法线、颜色和切线。

```
struct vertexInput  
{  
    float4 vertex : POSITION;  
    float4 tangent : TANGENT;  
    float3 normal : NORMAL;  
    float4 texcoord : TEXCOORD0;  
    float4 texcoord1 : TEXCOORD1;  
    fixed4 color : COLOR;  
};
```

语义是着色器输入或输出随附的字符串，提供有关参数使用的信息。您必须为在着色器阶段之间传递的所有变量指定语义。

如果您使用了不正确的语义，如 float3 tangent2 :TANGENTIAL，则会得到下列类型的错误：

```
Shader error in 'Custom/ctTextured': unknown semantics "TANGENTIAL" specified for  
"tangent2" at line 32
```

为了提高性能，请仅在您确实需要的输入结构中指定参数。Unity 拥有一些适用于最常用输入参数组合的预定语义输入结构：appdata\_base、appdata\_tan 和 appdata\_full。UnityCG.cginc include 文件中对这些结构进行了说明。前述顶点输入结构示例与 appdata\_full 对应。在这种情况下，您无需声明结构，只需声明 include 文件。

### 6.1.8 顶点着色器输出和变量

顶点着色器输出在必须包含顶点转换坐标的输出结构中定义。在下列示例中，输出结构非常简单，但是您可以添加其他量度。

下列代码列出了受 Unity 支持的语义：

```
struct vertexOutput  
{  
    float4 pos : SV_POSITION;  
    float4 tex : TEXCOORD0;  
    float4 texSpecular : TEXCOORD1;  
    float3 vertexInWorld : TEXCOORD2;  
    float3 viewDirInWorld : TEXCOORD3;  
    float3 normalInWorld : TEXCOORD4;  
    float3 vertexToLightInWorld : TEXCOORD5;  
    float4 vertexInScreenCoords : TEXCOORD6;  
    float4 shadowsVertexInScreenCoords : TEXCOORD7;  
};
```

转换顶点坐标使用语义 `SV_POSITION` 进行定义。两个纹理、多个向量以及在不同空间中调用语义 `TEXCOORDn` 的坐标也会传递至片段着色器。

通常情况下，`TEXCOORD0` 为 `UV` 保留，而 `TEXCOORD1` 为光照贴图 `UV` 保留。但是从技术角度而言，你可以在 `TEXCOORD0` 到 `TEXCOORD7` 中存入任何内容，传递给片段着色器使用。必须注意，每个插值器（即每个语义）只能处理最多 4 个浮点。将较大的变量（如矩阵）放入多个插值器中。这意味着，如果您将待传递的矩阵定义为变量：`float4x4 myMatrix : TEXCOORD2`，Unity 将使用 `TEXCOORD2` 至 `TEXCOORD5` 的插值器。

默认情况下，Unity 会对从顶点着色器发送到片段着色器的所有内容进行线性插值。对于通过顶点  $V_1$ 、 $V_2$  和  $V_3$  定义的三角形中的每个像素，位于顶点着色器和片段着色器之间的图形流水线中的光栅化程序将使用重心坐标  $\lambda_1$ 、 $\lambda_2$  和  $\lambda_3$  计算像素坐标，以作为顶点坐标的线性插值。

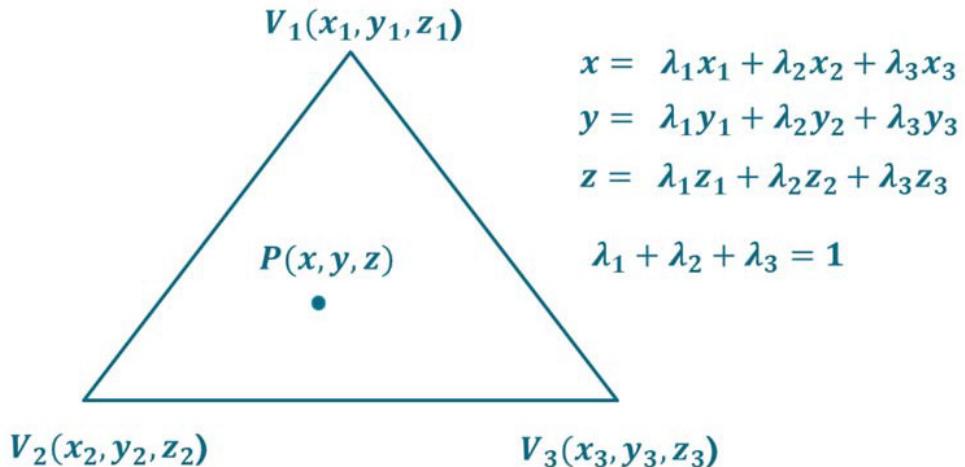


图 6-3 使用重心坐标的线性插值

下图显示了使用顶点颜色红色、绿色和蓝色得出的三角形颜色插值结果。

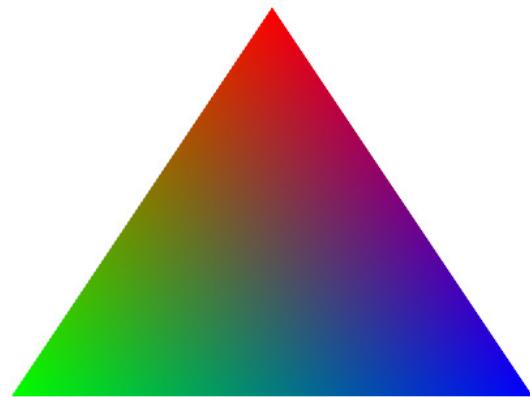


图 6-4 颜色插值

相同的插值适用于从顶点着色器传递至片段着色器的所有变量。这是一款功能非常强大的工具，因为它配备硬件线性插值器。例如，如果您拥有一个平面，并且想要将颜色作为与中心 C 距离的函数，请将中心 C 的坐标传递至顶点着色器，计算从顶点到 C 之间的平方距离，然后将该量度传递至片段着色器。距离值将自动内插至每个三角形的每个像素中。

由于值可以线性内插，因此可以执行逐顶点计算并在片段着色器中重复利用这些计算，也就是说，可在片段着色器中线性插值的值可以在顶点着色器中进行计算。这能够大幅提高性能，因为顶点着色器运行的数据集远小于片段着色器运行的数据集。

您必须谨慎使用变量，尤其是在移动平台上，因为性能和内存带宽消耗对众多游戏的成功至关重要。变量越多，顶点获取的带宽以及片段着色器变量读取的带宽也就越多。使用变量时，请寻求一个合理的平衡。

### 6.1.9 片段着色器

片段着色器是原语光栅化后的图形流水线阶段。

对于原语覆盖的像素的每个示例，都会生成一个片段。系统将针对每个生成的片段执行片段着色器代码。由于片段的数量多于顶点数量，因此您必须注意在片段着色器中执行的运算量。

在片段着色器中，您除了可以获取窗口空间的片段坐标外，您还可以获取从顶点着色器中得到的每个顶点输出值的插值。

在 [6.1.2 着色器结构 \(第 6-86 页\)](#) 中的着色器示例中，片段着色器接收来自顶点着色器的内插纹理坐标，并执行纹理查找以获取在这些坐标处的颜色。它将此颜色和环境颜色合并在一起，从而产生最终的输出颜色。通过声明片段着色器 `float4 frag (vertexOutput input) :COLOR`，很明显可以产生片段颜色。您可以在片段着色器中执行此操作，以达到预期效果。最终步骤是将正确颜色分配至片段。

### 6.1.10 向着色器提供数据

任何在 `Pass` 块中被声明为统一变量的数据均可用于顶点着色器和片段着色器。

由于统一变量无法在着色器中进行修改，因此可以将统一变量视作一种全局常数变量。

您可以通过以下方式向着色器提供此统一变量：

- 使用 `Properties` 块。
- 通过脚本编程。

`Properties` 块能够让您在 [检视面板](#) 中以交互方式定义统一变量。任何在 `Properties` 块中声明的变量都会出现在材质检视面板中，并且会带有变量名称。

下列代码显示了与材质 `ctSphereMat` 相关的着色器示例 `Properties` 块：

```
Properties
{
    _AmbientColor ("Ambient Color", Color) = (0.2,0.2,0.2,1.0)
    _MainTex ("Base (RGB)", 2D) = "white" {}
}
```

在 `Properties` 块中声明的名称为 `Ambient Color` 和 `Base (RGB)` 的变量 `_AmbientColor` 和 `_MainTex` 分别出现在带有这些名称的[材质检视面板](#)中。

下图显示了材质检视面板中的属性：

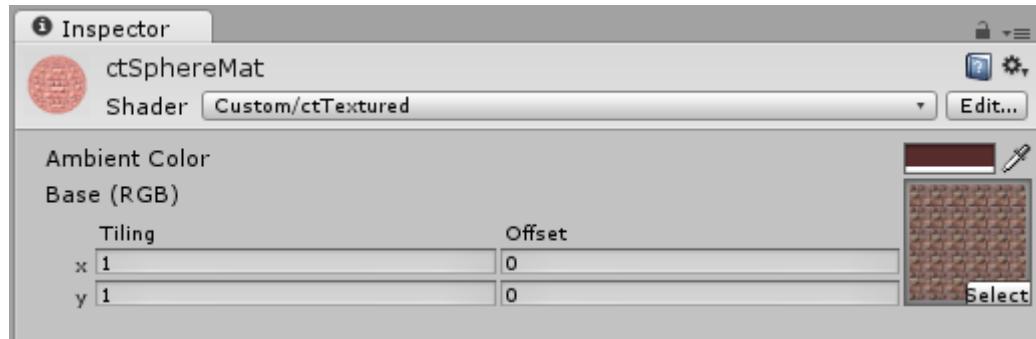


图 6-5 材质检视面板中的属性

当您处于着色器开发阶段时，使用 `Properties` 块将数据传递至着色器尤为有用，因为您可以在运行时以交互方式更改数据并查看效果。

您可以将下列类型的变量放入 `Properties` 块：

- 浮点数。
- 颜色。
- 纹理 2D。
- 立方体贴图。
- 长方形。
- 向量。

如果在之前计算中需要使用数据，或者需要在指定时间点传递数据，则 `Properties` 块不是一种非常有用的数据传递方法。

向着色器传递数据的另一种方法是根据脚本编程。

材质类显示了多种将材质相关数据传递至着色器的方法。下表列举了最常用的方法：

表 6-2 将材质相关数据传递至着色器的常用方法

方法
<code>SetColor (propertyName: string, color: Color);</code>
<code>SetFloat (propertyName: string, value: float);</code>
<code>SetInt (propertyName: string, value: int);</code>
<code>SetMatrix (propertyName: string, matrix: Matrix4x4);</code>
<code>SetVector (propertyName: string, vector: Vector4);</code>
<code>SetTexture (propertyName: string, texture: Texture);</code>

在下列代码中，主摄像机渲染完场景前，辅助摄像机 `shwCam` 会渲染阴影到将与主摄像机渲染通道合并的纹理。

对于阴影纹理投影流程，必须以方便的方式转换顶点。阴影摄像机投影矩阵 (`shwCam.projectionMatrix`)、世界到局部转换矩阵 (`shwCam.transform.worldToLocalMatrix`) 和渲染的阴影纹理 (`m_ShadowsTexture`) 将传递至着色器。

这些值可在着色器中用作为统一变量，其名称为 `_ShwCamProjMat`、`_ShwCamViewMat` 和 `m_ShadowsTexture`。

下列代码显示了如何借助 shwMats 列表包含的材质将矩阵和纹理发送至着色器。

```
// Called before object is rendered.  
  
public void OnWillRenderObject()  
{  
    // Perform different checks.  
    ...  
    CreateShadowsTexture();  
    // Set up shadows camera shwCam.  
    ...  
    // Pass matrices to the shader  
    for(int i = 0; i < shwMats.Count; i++)  
    {  
        shwMats[i].SetMatrix("_ShwCamProjMat", shwCam.projectionMatrix);  
        shwMats[i].SetMatrix("_ShwCamViewMat", shwCam.transform.worldToLocalMatrix);  
    }  
    // Render shadows texture  
    shwCam.Render();  
    for(int i = 0; i < shwMats.Count; i++)  
    {  
        shwMats[i].SetTexture( "_ShadowsTex", m_ShadowsTexture );  
    }  
    s_RenderingShadows = false;  
}
```

### 6.1.11 调试 Unity 中的着色器

在 Unity 中，无法以处理传统代码的方式调试着色器。但是，您可以用片段着色器的输出使待调试的值形象化。然后，您必须解析产生的图像。

下图显示了应用于 [6.2 使用局部立方体贴图实现反射（第 6-98 页）](#) 提供的地板反射面的着色器 ctReflLocalCubemap.shader 的输出：



图 6-6 带有反射的棋牌室

在下列片段着色器中，输出颜色已替换为标准化的局部修正反射向量：

```
return float4(normalize(localCorrReflDirWS), 1.0);
```

它使标准化为颜色的反射向量分量（而非反射图像）得以形象化。

地板的红色区域指示反射向量拥有很强的 X 分量，也就是说，它大部分都指向 X 轴方向。红色区域显示来自带窗墙壁的反射。

蓝色区域表示指向 Z 轴的反射向量（即来自右侧墙壁的反射）最为显著。

在黑色区域中，向量主要指向 -Z，但是颜色只能拥有正值分量，因为负值分量已固定为 0。

下图显示了将片段输出颜色替换为标准化的局部修正反射向量的结果：

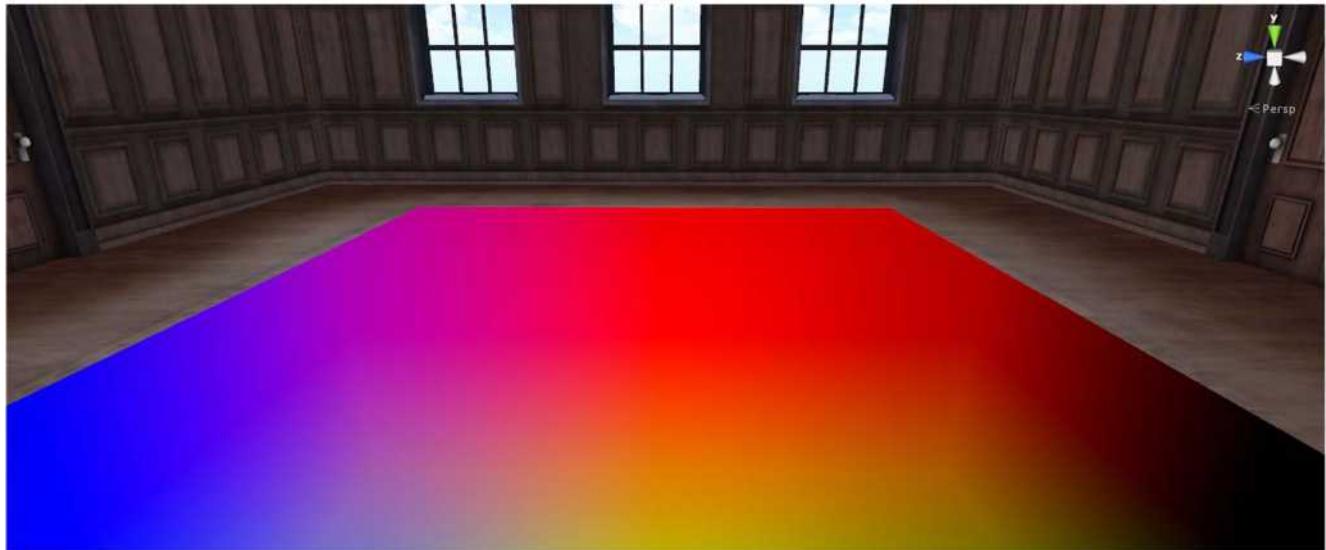


图 6-7 使用多种颜色调试着色器

最初可能很难在调试时解析各颜色的含义，因此请尝试专注于单一颜色分量。例如，您只能返回标准化局部修正反射向量的 Y 分量：

```
float3 normLocalCorrReflDirWS = normalize(localCorrReflDirWS);
return float4(0, normLocalCorrReflDirWS.y, 0, 1);
```

在这种情况下，输出只是主要来自摄像机上方屋顶的反射。也就是说，房间中指向 Y 轴的部分。房间墙壁的反射来自 X、Z 和 -Z 方向，所以它们以黑色渲染。

下图显示了使用单一颜色的着色器调试：



图 6-8 使用单一颜色调试着色器

检查使用颜色调试的幅度是否介于 0 和 1 之间，因为所有其他值已自动固定。所有负值均指定为 0，而所有大于 1 的值均指定为 1。

## 6.2 使用局部立方体贴图实施反射

对于在移动设备上渲染反射，基于局部立方体贴图的反射是很有用的技巧。

Unity 版本 5 和更高版本将基于局部立方体贴图的反射实施为反射探测器。您可以将它们与其他类型的反射组合，如运行时在您的自定义着色器中渲染的反射。

本节包含以下小节：

- 6.2.1 反射实施的历程（第 6-98 页）。
- 6.2.2 使用局部立方体贴图生成正确的反射（第 6-100 页）。
- 6.2.3 着色器实施（第 6-102 页）。
- 6.2.4 过滤立方体贴图（第 6-104 页）。
- 6.2.5 射线与盒求交算法（第 6-108 页）。
- 6.2.6 用于使编辑器脚本生成立方体贴图的源代码（第 6-111 页）。

### 6.2.1 反射实施的历程

图形开发人员一直在尝试寻找计算量较小的反射实现方法。

第一批解决方案中的其中一个便是球形贴图。此方案无需通过计算量大的射线跟踪或光照计算来模拟对象的反射或光照。

下图显示了球体上的环境贴图：

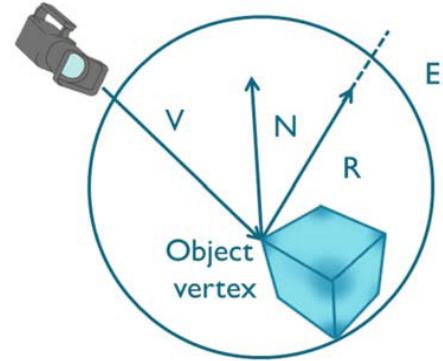


图 6-9 球体的环境贴图

下图显示了将球面映射为两个维度的方程：

The spherical surface is mapped into 2D:

$$u = \frac{R_x}{m} + \frac{1}{2}$$

$$v = \frac{R_y}{m} + \frac{1}{2}$$

$$m = 2 \cdot \sqrt{R_x^2 + R_y^2 + (R_z + 1)^2}$$

图 6-10 球面 2D 贴图方程

此方法有多种劣势，但是主要问题在于，将图片贴图至球体时会出现失真。1999 年，立方体贴图与硬件加速可结合使用。立方体贴图解决了与球形贴图相关的图像失真、视角依赖性以及低效计算等问题。

下图显示了展开的立方体：

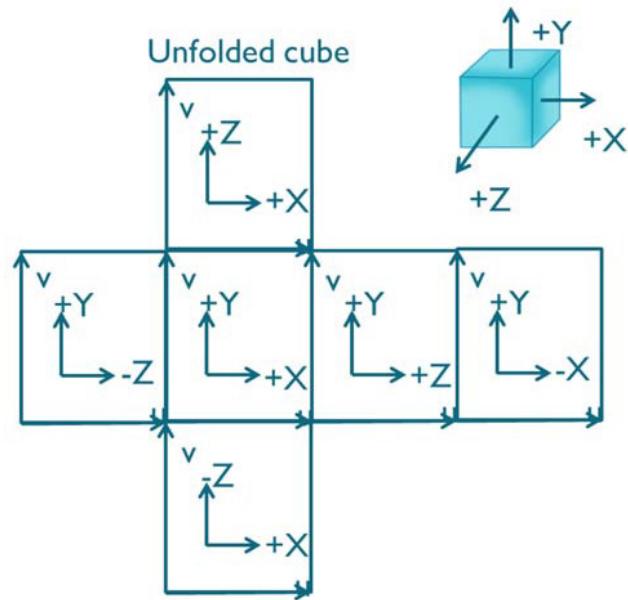


图 6-11 展开的立方体

立方体贴图使用立方体的六个面作为贴图形状。环境投射到立方体的每个面并存储为六个正方纹理，或展开为单个纹理的六个区域。可以使用六个不同的摄像机方位从指定位置渲染场景，以生成立方体贴图，90 度的视锥体代表每个立方体表面。源图像将被直接采样。对中间环境贴图重新采样不会产生任何失真。

下图显示了无限反射：

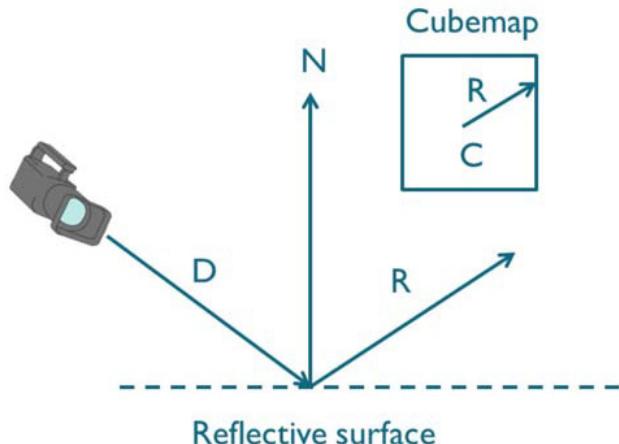


图 6-12 无限反射

若要实现基于立方体贴图的反射，请评估反射向量  $R$  并用该向量从立方体贴图  $_Cubemap$  中获取像素元，方法是使用可用纹理查找函数  $\text{texCUBE}()$ ：

```
float4 color = texCUBE(_Cubemap, R);
```

法线 N 和视线向量 D 可从顶点着色器传输至片段着色器。片段着色器从立方体贴图中获取纹理颜色：

```
float3 R = reflect(D, N);
float4 color = texCUBE(_Cubemap, R);
```

此方法只能从立方体贴图位置不重要的远距离环境中复制反射。这种简单有效的技术主要用于室外光照，例如增加天空的反射效果。

下图显示了不正确的反射：



图 6-13 不正确的反射

如果在局部环境中使用此技术，则会产生不正确的反射。反射不正确的原因在于，在表达式 `float4 color = texCUBE(_Cubemap, R);` 中，没有与局部几何结构绑定。例如，如果您走过反射地板并从同一角度看它，您将始终看到相同的反射。反射向量始终相同，并且该表达式始终得出相同的结果。这是因为视线向量的方向未发生变化。在真实的世界中，反射取决于观察角度和位置。

## 6.2.2 使用局部立方体贴图生成正确的反射

此问题的解决方案涉及在程序中与局部几何结构绑定以计算反射。

此解决方案在《GPU 精粹：实时图形编程的技术、技巧和技艺》（作者：Randima Fernando（丛书编辑））进行了介绍。

下图显示了使用包围球体进行局部修正：

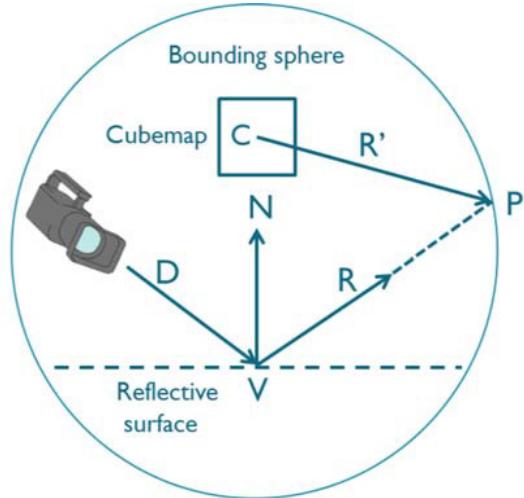


图 6-14 使用包围球体进行局部修正

包围球体用作界定待反射场景的代理区域。它不使用反射向量  $R$  从立方体贴图中获取纹理，而是使用新向量  $R'$ 。为构建此新向量，您需要在反射向量  $R$  方向的局部点  $V$  中找到射线包围球体的相交点  $P$ 。再根据生成立方体贴图的立方体贴图  $C$  的中心创建相交点  $P$  的新向量  $R'$ 。然后使用此向量获取立方体贴图的纹理。

```
float3 R = reflect(D, N);
Find intersection point P
Find vector R' = CP
float4 col = texCUBE(_Cubemap, R');
```

此方法将在近球形的对象表面中产生较好的结果，但是平面反射表面的反射将会失真。此方法的另一缺点在于，用于计算与包围球体的相交点的算法需要解一元二次方程式，此过程非常复杂。

2010 年，一位开发人员在论坛上提出了一个更好的解决方案，论坛网址为：<http://www.gamedev.net>。此方法使用盒代替之前的包围球体，解决了前述方法所产生的失真和复杂性问题。有关更多详情，请参阅：<http://www.gamedev.net/topic/568829-box-projected-cubemap-environment-mapping/?p=4637262>.

下图显示了使用包围盒进行局部修正：

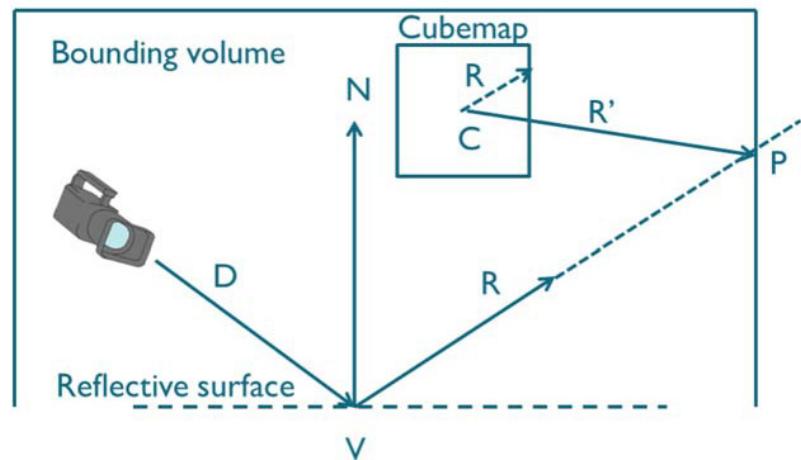


图 6-15 使用包围盒进行局部修正

Sebastien Lagarde 于 2012 年发布的一件新作品使用这种新方法模拟了更复杂的环境镜面反射光照（使用多个立方体贴图），并使用一种算法评估了每个立方体贴图的作用并将其有效融合在 GPU 上。请参阅 <http://seblagarde.wordpress.com>

表 6-3 无限和局部立方体贴图之间的差异

无限立方体贴图	本地立方体贴图
• 主要用于室外，代表远距离环境的光照。	• 代表有限局部环境的光照。
• 立方体贴图位置不重要。	• 立方体贴图位置重要。
	• 这些立方体贴图的光照仅在立方体贴图的创建位置才正确。
	• 必须运用局部修正来调整立方体贴图的固有无限性，以使其适应局部环境。

下图显示了带有用局部立方体贴图所生成正确反射的场景。



图 6-16 正确反射

### 6.2.3 着色器实施

本节介绍使用局部立方体贴图实施反射的着色器。

顶点着色器用于计算作为内插值传输至片段着色器的三个幅度：

- 顶点位置。
- 观察方向。
- 法线。

这些值位于世界坐标中。

下列代码显示了使用局部立方体贴图的反射的着色器实施，它适用于 Unity。

```
vertexOutput vert(vertexInput input)
{
    vertexOutput output;
    output.tex = input.texcoord;
    // Transform vertex coordinates from local to world.
    float4 vertexWorld = mul(_Object2World, input.vertex);
    // Transform normal to world coordinates.
    float4 normalWorld = mul(float4(input.normal, 0.0), _World2Object);
```

```
// Final vertex output position. output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
// ----- Local correction -----
output.vertexInWorld = vertexWorld.xyz;
output.viewDirInWorld = vertexWorld.xyz - _WorldSpaceCameraPos;
output.normalInWorld = normalWorld.xyz;
return output;
}
```

区域盒中的相交点以及反射向量在片段着色器中计算。您将构建新的局部修正反射向量并用该向量从局部立方体贴图中获取反射纹理。然后，您将结合纹理和反射生成输出颜色：

```
float4 frag(vertexOutput input) : COLOR
{
    float4 reflColor = float4(1, 1, 0, 0);
    // Find reflected vector in WS.
    float3 viewDirWS = normalize(input.viewDirInWorld);
    float3 normalWS = normalize(input.normalInWorld);
    float3 reflDirWS = reflect(viewDirWS, normalWS);
    // Working in World Coordinate System.
    float3 localPosWS = input.vertexInWorld;
    float3 intersectMaxPointPlanes = (_BBoxMax - localPosWS) / reflDirWS;
    float3 intersectMinPointPlanes = (_BBoxMin - localPosWS) / reflDirWS;
    // Looking only for intersections in the forward direction of the ray.
    float3 largestParams = max(intersectMaxPointPlanes, intersectMinPointPlanes);
    // Smallest value of the ray parameters gives us the intersection.
    float distToIntersect = min(min(largestParams.x, largestParams.y), largestParams.z);
    // Find the position of the intersection point.
    float3 intersectPositionWS = localPosWS + reflDirWS * distToIntersect;
    // Get local corrected reflection vector.
    float3 localCorrRefDirWS = intersectPositionWS - _EnviCubeMapPos;
    // Lookup the environment reflection texture with the right vector.
    reflColor = texCUBE(_Cube, localCorrRefDirWS);
    // Lookup the texture color.
    float4 texColor = tex2D(_MainTex, float2(input.tex));
    return _AmbientColor + texColor * _ReflAmount * reflColor;
}
```

在前述片段着色器代码中，幅度 `_BBoxMax` 和 `_BBoxMin` 是包围区域的最大点和最小点。变量 `_EnviCubeMapPos` 是立方体贴图的创建位置。请通过下列脚本将这些值传输至着色器：

```
[ExecuteInEditMode]
public class InfoToReflMaterial : MonoBehaviour
{
    // The proxy volume used for local reflection calculations.
    public GameObject boundingBox;

    void Start()
    {
        Vector3 bboxLength = boundingBox.transform.localScale;
        Vector3 centerBBox = boundingBox.transform.position;

        // Min and max BBox points in world coordinates
        Vector3 BMin = centerBBox - bboxLength/2;
        Vector3 BMax = centerBBox + bboxLength/2;

        // Pass the values to the material.
        gameObject.renderer.sharedMaterial.SetVector("_BBoxMin", BMin);
        gameObject.renderer.sharedMaterial.SetVector("_BBoxMax", BMax);
        gameObject.renderer.sharedMaterial.SetVector("_EnviCubeMapPos", centerBBox);
    }
}
```

将 `_AmbientColor`、`_ReflAmount`、主纹理以及立方体贴图纹理的值传输至着色器，作为属性块的统一变量：

```
Shader "Custom/ctReflLocalCubemap"
{
    Properties
    {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _Cube("Reflection Map", Cube) = "" {}
        _AmbientColor("Ambient Color", Color) = (1, 1, 1, 1)
        _ReflAmount("Reflection Amount", Float) = 0.5
    }
}
```

```
SubShader
{
    Pass
    {
        CGPROGRAM
        #pragma glsl
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"

        // User-specified uniforms
        uniform sampler2D _MainTex;
        uniform samplerCUBE _Cube;
        uniform float4 _AmbientColor;
        uniform float _ReflAmount;
        uniform float _ToggleLocalCorrection;
        // -----Passed from script InfoRoReflmaterial.cs -----
        uniform float3 _BBoxMin;
        uniform float3 _BBoxMax;
        uniform float3 _EnviCubeMapPos;

        struct vertexInput
        {
            float4 vertex : POSITION;
            float3 normal : NORMAL;
            float4 texcoord : TEXCOORD0;
        };
        struct vertexOutput
        {
            float4 pos : SV_POSITION;
            float4 tex : TEXCOORD0;
            float3 vertexInWorld : TEXCOORD1;
            float3 viewDirInWorld : TEXCOORD2;
            float3 normalInWorld : TEXCOORD3;
        };
        Vertex shader      { }
        Fragment shader   { }

        ENDCG
    }
}
}
```

计算包围区域相交点的算法基于使用参数表示局部位置或片段的反射射线。有关射线与盒求交算法的说明，请参阅 [6.2.5 射线与盒求交算法（第 6-108 页）](#)。

#### 6.2.4 过滤立方体贴图

使用局部立方体贴图实现反射的其中一项优势就是立方体贴图为静态。也就是说，它在开发时生成，而非在运行时生成。这样便能够对立方体贴图图像应用任何过滤来实现某一效果。

CubeMapGen 是 AMD 提供的用于对立方体贴图应用过滤的工具。您可从 AMD 开发人员网站上获取 CubeMapGen，网址为：<http://developer.amd.com>。

若要将 Unity 中的立方体贴图图像导出至 CubeMapGen，您必须单独保存每张立方体贴图图像。有关图像保存工具的源代码，请参阅 [6.2.6 用于使编辑器脚本生成立方体贴图的源代码（第 6-111 页）](#)。此工具不仅能够创建立方体贴图，而且还可选择性地单独保存每张立方体贴图图像。

您必须将此工具的脚本存储于 **Asset** 目录的 **Editor** 文件夹中。

如何使用立方体贴图编辑器工具：

1. 创建立方体贴图。
2. 从“游戏对象”菜单中启动烘焙立方体贴图工具。
3. 提供立方体贴图和摄像机渲染位置。
4. 如果您计划对立方体贴图应用过滤，请选择性地保存每张图像。

下图显示了烘焙立方体贴图工具界面：

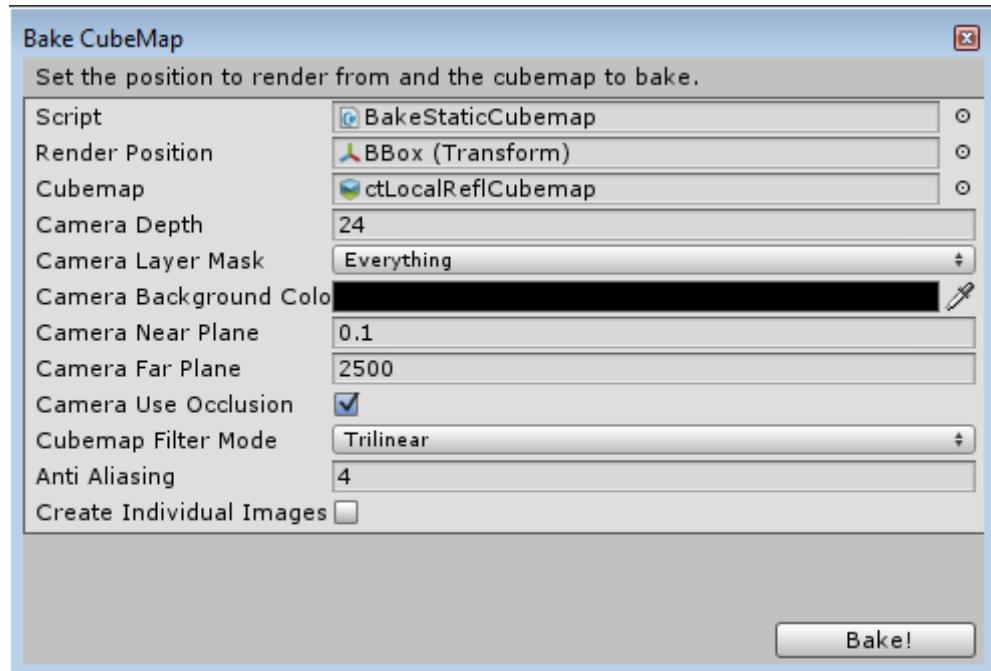


图 6-17 烘焙立方体贴图工具界面

您可以使用 CubeMapGen 为立方体贴图单独加载每张图像。

从选择立方体表面下拉菜单中选择要加载的表面，然后按下加载立方体表面按钮。加载完所有表面后，可旋转立方体贴图并检查其是否正确。

CubeMapGen 在过滤类型下拉菜单中拥有众多不同的过滤选项。选择您需要的过滤设置，然后按下过滤立方体贴图，应用过滤器。过滤过程可能会花费数分钟，具体取决于立方体贴图的大小。由于没有撤消选项，因此在应用任何过滤之前，请将立方体贴图保存为独立图像。如果过滤结果与您期待的不一样，您可以重新加载立方体贴图并尝试调整参数。

按照下列步骤将立方体贴图图像导入 CubeMapGen：

1. 选中复选框，在烘焙立方体贴图时保存独立图像。
2. 启动 CubeMapGen 工具，并按照下表所示关系加载立方体贴图图像。
3. 将立方体贴图保存为单个 dds 或立方体交叉图像。由于无法撤消，因此这使您能够在使用过滤器进行试验的情况下重新加载立方体贴图。
4. 根据需要对立方体贴图应用过滤器，直到结果满意为止。
5. 将立方体贴图保存为独立图像。

下表显示了 CubeMapGen 和 Unity 间立方体贴图表面指数的等效情况。

表 6-4 CubeMapGen 和 Unity 间立方体贴图表面指数的等效情况

AMD CubeMapGen	Unity
X+	-X
X-	+X
Y+	+Y
Y-	-Y
Z+	+Z
Z-	-Z

下图显示了加载六张立方体贴图图像后的 CubeMapGen:



图 6-18 CubeMapGen

下图显示了在应用高斯滤波以获得霜冻效果后的 CubeMapGen:



图 6-19 显示霜冻效果的 CubeMapGen

下表显示了与高斯过滤器结合使用以达到霜冻效果的过滤器参数。

表 6-5 CubeMapGen 中用于使反射产生霜冻效果的参数。

过滤设置	值
类型	高斯
基本过滤角度	8
Mip 初始过滤角度	5
Mip 过滤角度比例	2.0
边缘修复	已选中
边缘修复宽度	4

下图显示了通过带有霜冻效果的立方体贴图生成的反射：



图 6-20 带有霜冻效果的反射

下图汇总了使用 CubeMapGen 工具对 Unity 立方体贴图应用过滤的工作流程。

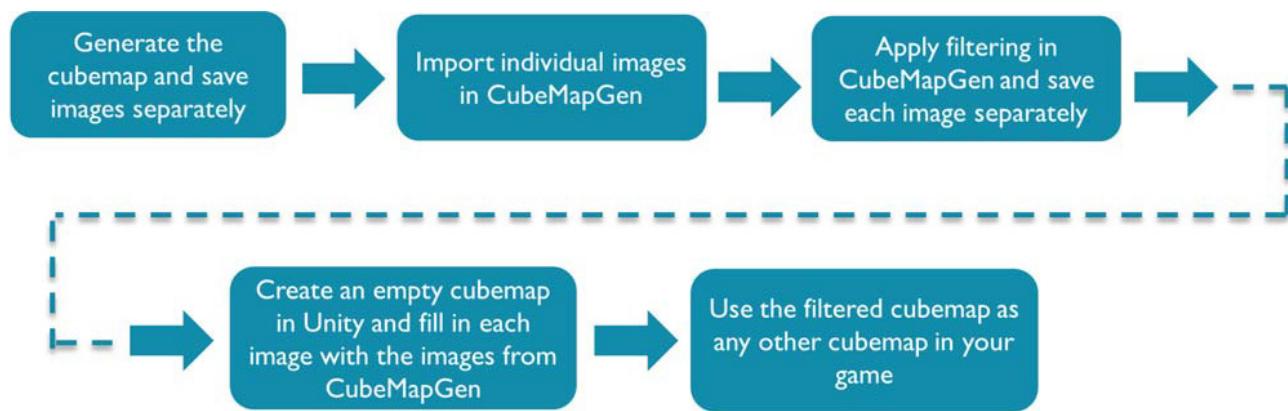


图 6-21 立方体贴图过滤工作流程

## 6.2.5 射线与盒求交算法

本节介绍射线与盒求交算法。

下图显示了含直线方程的图形：

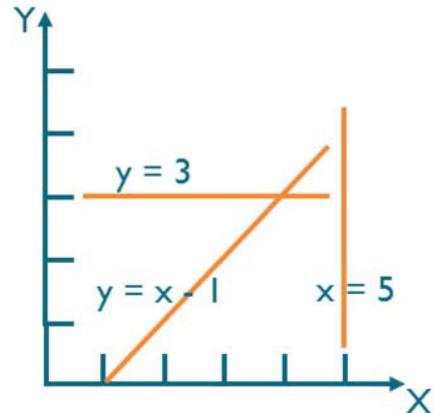


图 6-22 含直线方程的图形

直线方程

$$y = mx + b$$

该方程的向量形式为:

$$\mathbf{r} = \mathbf{o} + t * \mathbf{d}$$

其中:

o 代表原点

d 代表方向矢量

t 代表参数

下图显示了一个轴对齐包围盒:

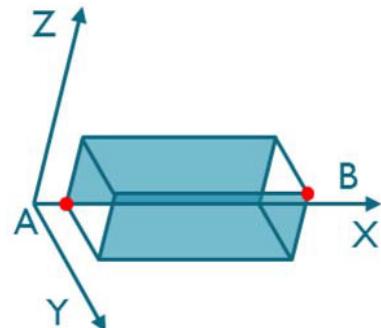


图 6-23 轴对齐包围盒

可通过最小点 A 和最大点 B 定义轴对齐包围盒 AABB。

AABB 定义了一组与坐标轴平行的直线。可使用下列方程定义每条直线:

$$\begin{aligned}x &= A_x; y = A_y; z = A_z \\x &= B_x; y = B_y; z = B_z\end{aligned}$$

若要找到射线与其中一条直线的相交点，只需使这两个方程相等。例如:

$$\mathbf{o}_x + t_x * \mathbf{d}_x = A_x$$

您可以将解答方程写成：

$$t_{Ax} = (Ax - 0_x) / D_x$$

以相同的方式求出这两个相交点的所有可能解：

$$\begin{aligned} t_{Ax} &= (Ax - 0_x) / D_x \\ t_{Ay} &= (Ay - 0_y) / D_y \\ t_{Az} &= (Az - 0_z) / D_z \\ t_{Bx} &= (Bx - 0_x) / D_x \\ t_{By} &= (By - 0_y) / D_y \\ t_{Bz} &= (Bz - 0_z) / D_z \end{aligned}$$

向量形式的方程为：

$$\begin{aligned} t_A &= (A - 0) / D \\ t_B &= (B - 0) / D \end{aligned}$$

这可找出直线与立方体表面定义的平面相交的位置，但是它无法保证相交点位于立方体上。

下图显示了射线与盒相交点的 2D 表示：

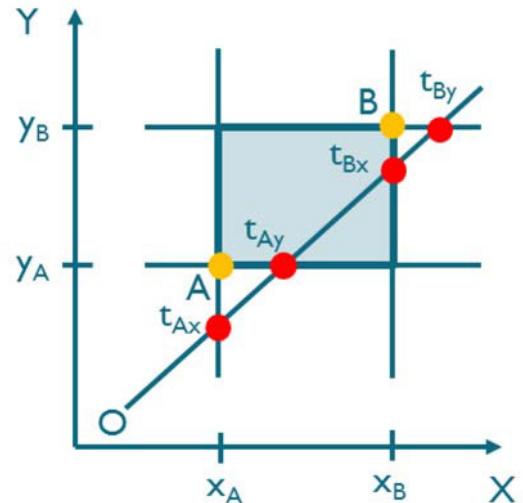


图 6-24 射线与盒相交点的 2D 表示

若要找到哪种答案确实是与盒的相交点，您需要用参数  $t$  中较大的值来确认最小平面上的相交点。

$$t_{\min} = (t_{Ax} > t_{Ay}) ? t_{Ax} : t_{Ay}$$

您需要用参数  $t$  中较小的值来确认最大平面上的相交点。

$$t_{\max} = (t_{Ax} > t_{Ay}) ? t_{Ax} : t_{Ay}$$

如果您未得到任何相交点，也必须考虑这些情况。

下图显示了无相交点的射线与盒：

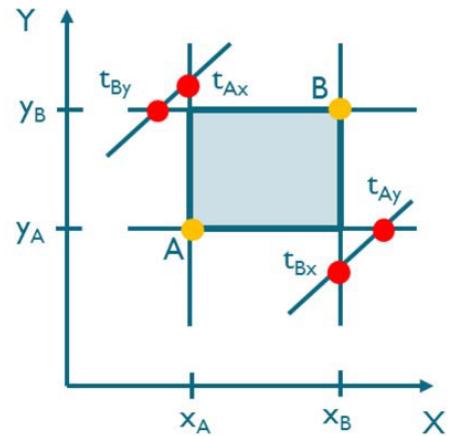


图 6-25 无相交点的射线与盒

如果您保证反射面已被 BBox 包围，即反射线的来源位于 BBox 中，则始终存在两个与该盒相交的相交点，并且处理不同情况的流程也会简化

下图显示了 BBox 中的射线与盒相交点：

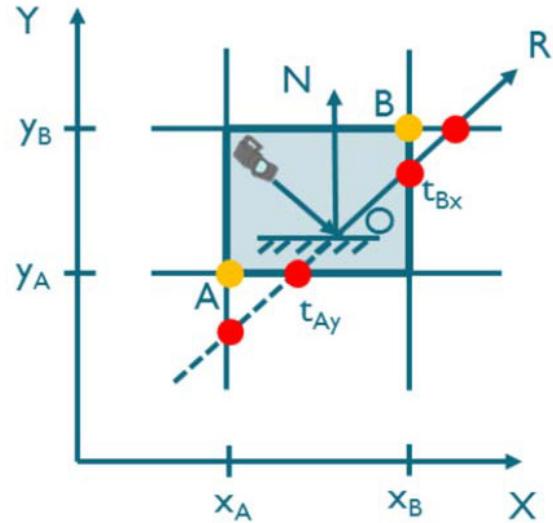


图 6-26 BBox 中的射线与盒相交点

## 6.2.6 用于使编辑器脚本生成立方体贴图的源代码

本节提供的源代码可以供编辑器脚本生成立方体贴图。

```
/*
 * This confidential and proprietary software may be used only as
 * authorised by a licensing agreement from ARM Limited
 * (C) COPYRIGHT 2014 ARM Limited
 * ALL RIGHTS RESERVED
 * The entire notice above must be reproduced on all authorised
 * copies and copies may only be made to the extent permitted
 * by a licensing agreement from ARM Limited.
 */

using UnityEngine;
using UnityEditor;
using System.IO;
/**
```

```
* This script must be placed in the Editor folder.  
* The script renders the scene into a cubemap and optionally  
* saves each cubemap image individually.  
* The script is available in the Editor mode from the  
* Game Object menu as "Bake Cubemap" option.  
* Be sure the camera far plane is enough to render the scene.  
*/  
  
public class BakeStaticCubemap : ScriptableWizard  
{  
  
    public Transform renderPosition;  
    public Cubemap cubemap;  
    // Camera settings.  
    public int cameraDepth = 24;  
    public LayerMask cameraLayerMask = -1;  
    public Color cameraBackgroundColor;  
    public float cameraNearPlane = 0.1f;  
    public float cameraFarPlane = 2500.0f;  
    public bool cameraUseOcclusion = true;  
    // Cubemap settings.  
    public FilterMode cubemapFilterMode = FilterMode.Trilinear;  
    // Quality settings.  
    public int antiAliasing = 4;  
  
    public bool createIndividualImages = false;  
  
    // The folder where individual cubemap images will be saved  
    static string imageDirectory = "Assets/CubemapImages";  
    static string[] cubemapImage  
        = new string[]{"front+Z", "right+X", "back-Z", "left-X", "top+Y", "bottom-Y"};  
    static Vector3[] eulerAngles = new Vector3[]{new Vector3(0.0f,0.0f,0.0f),  
        new Vector3(0.0f,-90.0f,0.0f), new Vector3(0.0f,180.0f,0.0f),  
        new Vector3(0.0f,90.0f,0.0f), new Vector3(-90.0f,0.0f,0.0f),  
        new Vector3(90.0f,0.0f,0.0f)};  
  
    void OnWizardUpdate()  
    {  
        helpString = "Set the position to render from and the cubemap to bake.";  
        if(renderPosition != null && cubemap != null)  
        {  
            isValid = true;  
        }  
        else  
        {  
            isValid = false;  
        }  
    }  
  
    void OnWizardCreate ()  
    {  
  
        // Create temporary camera for rendering.  
        GameObject go = new GameObject( "CubemapCam", typeof(Camera) );  
        // Camera settings.  
        go.camera.depth = cameraDepth;  
        go.camera.backgroundColor = cameraBackgroundColor;  
        go.camera.cullingMask = cameraLayerMask;  
        go.camera.nearClipPlane = cameraNearPlane;  
        go.camera.farClipPlane = cameraFarPlane;  
        go.camera.useOcclusionCulling = cameraUseOcclusion;  
        // Cubemap settings  
        cubemap.filterMode = cubemapFilterMode;  
        // Set antialiasing  
        QualitySettings.antiAliasing = antiAliasing;  
  
        // Place the camera on the render position.  
        go.transform.position = renderPosition.position;  
        go.transform.rotation = Quaternion.identity;  
  
        // Bake the cubemap  
        go.camera.RenderToCubemap(cubemap);  
  
        // Rendering individual images  
        if(createIndividualImages)  
        {  
            if (!Directory.Exists(imageDirectory))  
            {  
                Directory.CreateDirectory(imageDirectory);  
            }  
  
            RenderIndividualCubemapImages(go);  
        }  
    }  
}
```

```
// Destroy the camera after rendering.  
DestroyImmediate(go);  
}  
  
void RenderIndividualCubemapImages(GameObject go)  
{  
    go.camera.backgroundColor = Color.black;  
    go.camera.clearFlags = CameraClearFlags.Skybox;  
    go.camera.fieldOfView = 90;  
    go.camera.aspect = 1.0f;  
  
    go.transform.rotation = Quaternion.identity;  
  
    //Render individual images  
    for (int camOrientation = 0; camOrientation < eulerAngles.Length ; camOrientation++)  
    {  
        string imageName = Path.Combine(imageDirectory, cubemap.name + "_"  
            + cubemapImage[camOrientation] + ".png");  
        go.camera.transform.eulerAngles = eulerAngles[camOrientation];  
        RenderTexture renderTex = new RenderTexture(cubemap.height,  
            cubemap.height, cameraDepth);  
        go.camera.targetTexture = renderTex;  
        go.camera.Render();  
        RenderTexture.active = renderTex;  
  
        Texture2D img = new Texture2D(cubemap.height, cubemap.height,  
            TextureFormat.RGB24, false);  
        img.ReadPixels(new Rect(0, 0, cubemap.height, cubemap.height), 0, 0);  
  
        RenderTexture.active = null;  
        GameObject.DestroyImmediate(renderTex);  
  
        byte[] imgBytes = img.EncodeToPNG();  
        File.WriteAllBytes(imageName, imgBytes);  
  
        AssetDatabase.ImportAsset(imageName, ImportAssetOptions.ForceUpdate);  
    }  
    AssetDatabase.Refresh();  
}  
  
[MenuItem("GameObject/Bake Cubemap")]  
static void RenderCubemap ()  
{  
    ScriptableWizard.DisplayWizard("Bake CubeMap", typeof(BakeStaticCubemap), "Bake!");  
}
```

## 6.3 组合反射

本节介绍组合反射。

本节包含以下小节：

- [6.3.1 关于组合反射 \(第 6-114 页\)](#)。
- [6.3.2 组合反射着色器实施 \(第 6-116 页\)](#)。
- [6.3.3 组合远距离环境的反射 \(第 6-118 页\)](#)。

### 6.3.1 关于组合反射

基于局部立方体贴图的反射技巧实现了基于静态局部立方体贴图渲染高质量、高效率的反射。但是，如果对象是动态的，静态局部立方体贴图便不再有效，该技巧也无法发挥作用。

您可以将静态反射与动态生成的反射相组合，从而解决此问题。

下图显示了组合来自静态和动态几何体的反射：

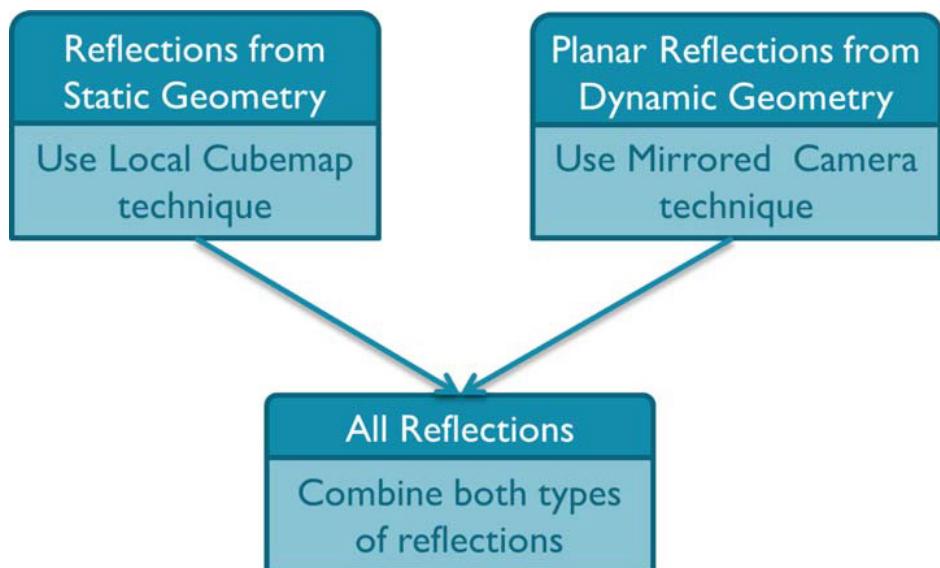


图 6-27 组合来自静态和动态几何体的反射

如果反射表面是平面，您可以使用镜像摄像机生成动态反射。

要创建镜像摄像机，可计算在运行时渲染反射的主摄像机的位置和指向。

相对于反射平面，镜像主摄像机的位置和指向。

下图显示了镜像摄像机技巧：

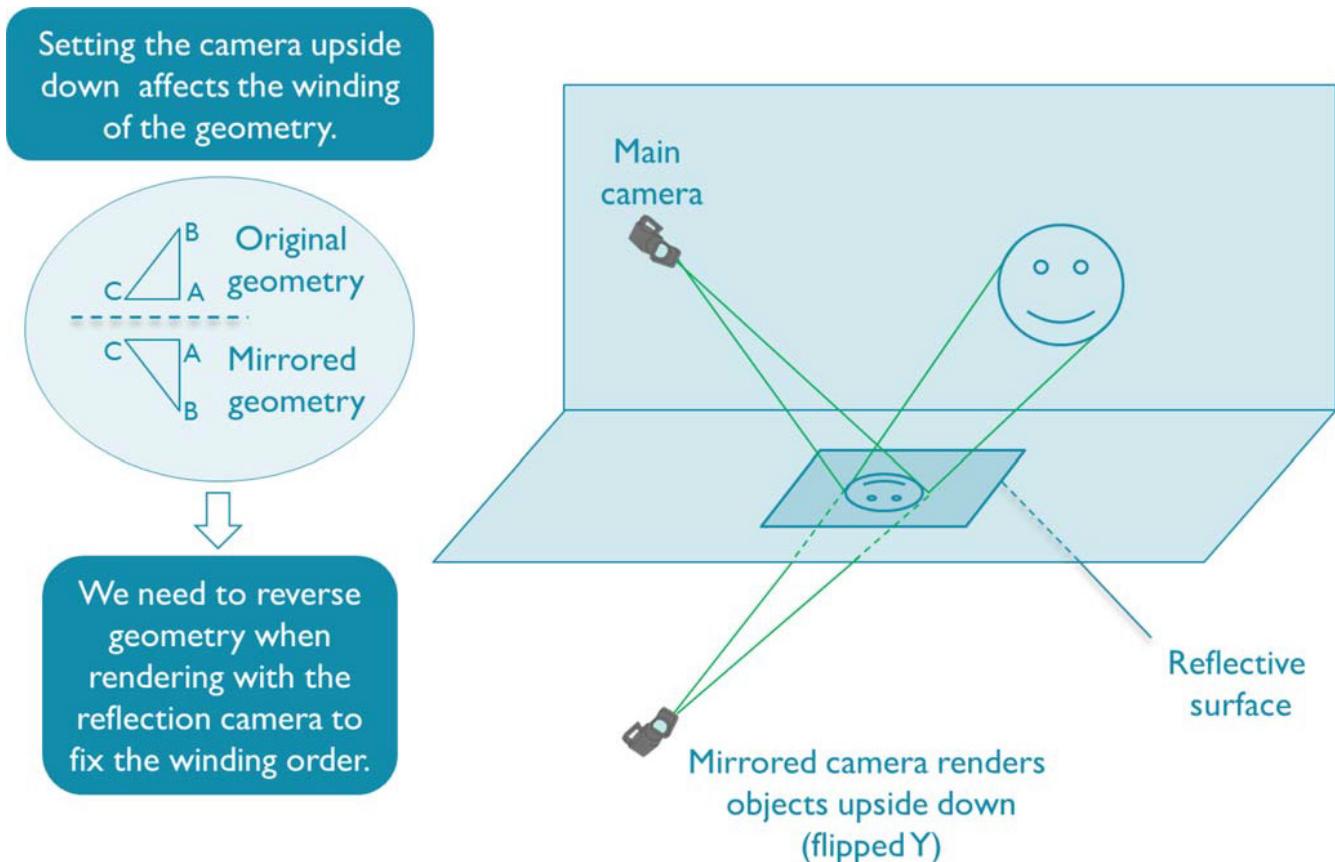


图 6-28 用于渲染平面反射的镜像摄像机技巧

在镜像过程中，新的反射摄像机最终表现为其轴在相对的指向。与现实中的镜子一样，左右的反射被颠倒。这意味着反射摄像机使用相反的卷绕渲染几何体。

要正确渲染几何体，您必须逆转几何体卷绕，然后再渲染反射。完成了反射渲染时，恢复原始卷绕。

下图显示了设置镜像摄像机和渲染反射所需的步骤序列：

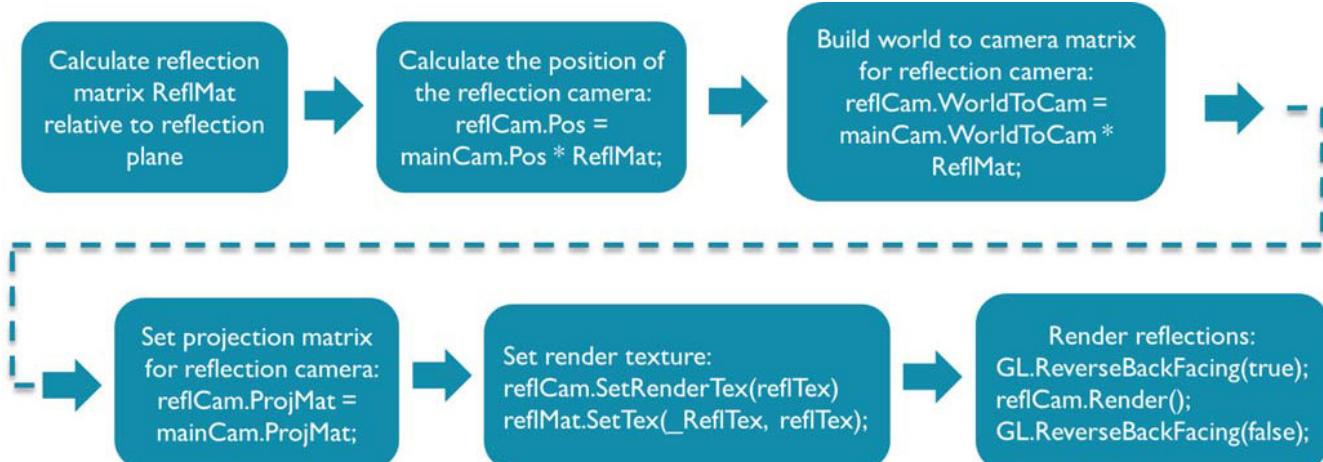


图 6-29 设置镜像摄像机和渲染反射的主要步骤

构建镜像反射变换矩阵。使用此矩阵计算反射摄像机的位置和世界至摄像机变换矩阵。

下图显示了镜像反射变换矩阵：

$$R = \begin{bmatrix} 1 - 2n_x^2 & -2n_xn_y & -2n_xn_z & -2n_xn_w \\ -2n_xn_y & 1 - 2n_y^2 & -2n_yn_z & -2n_yn_w \\ -2n_xn_z & -2n_yn_z & 1 - 2n_z^2 & -2n_zn_w \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} n_x &= planeNormal.x \\ n_y &= planeNormal.y \\ n_z &= planeNormal.z \\ n_w &= -dot(planeNormal, planePos) \end{aligned}$$

图 6-30 镜像反射变换矩阵

将反射矩阵变换应用到主摄像机的位置和世界至摄像机矩阵。这将为您提供反射摄像机的位置和世界至摄像机矩阵。

反射摄像机的投影矩阵必须和主摄像机的投影矩阵相同。

反射摄像机将反射渲染到纹理。

为获得良好的结果，在渲染之前必须先正确设置此纹理：

- 使用纹理映射。
- 将过滤模式设置为三线性。
- 使用多重采样。

确保纹理大小与反射表面的面积成正比。纹理越大，反射的像素化程度越低。

您可以从以下网址找到镜像摄像机的脚本示例：<http://wiki.unity3d.com/index.php/File:MirrorReflection.png>。

### 6.3.2 组合反射着色器实施

您可以组合着色器中的静态环境反射和动态平面反射。

要组合着色器中的反射，您必须修改 [6.2.3 着色器实施（第 6-102 页）](#) 中提供的着色器代码。

着色器必须融合平面反射，在运行时通过反射摄像机渲染。要达到此目的，来自反射摄像机的纹理 `_ReflectionTex` 作为统一变量传递到片段着色器，再使用 `lerp()` 函数与平面反射结果组合。

除了与局部修正相关的数据外，顶点着色器还要使用内置函数 `ComputeScreenPos()` 计算顶点的屏幕坐标。它将这些坐标传递给片段着色器：

```
vertexOutput vert(vertexInput input)
{
    vertexOutput output;
    output.tex = input.texcoord;

    // Transform vertex coordinates from local to world.
```

```

float4 vertexWorld = mul(_Object2World, input.vertex);

// Transform normal to world coordinates.
float4 normalWorld = mul(float4(input.normal, 0.0), _World2Object);

// Final vertex output position.
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);

// ----- Local correction -----
output.vertexInWorld = vertexWorld.xyz;
output.viewDirInWorld = vertexWorld.xyz - _WorldSpaceCameraPos;
output.normalInWorld = normalWorld.xyz;

// ----- Planar reflections -----
output.vertexInScreenCoords = ComputeScreenPos(output.pos);
return output;
}

```

平面反射渲染至纹理，让片段着色器能够访问片段的屏幕坐标。为此，需要将顶点屏幕坐标作为变量传递到片段着色器。

在片段着色器中：

- 向反射向量应用局部修正。
- 从局部立方体贴图检索环境反射的颜色 `staticReflColor`。

下列代码显示了如何将使用局部立方体贴图技巧的静态环境反射与运行时使用镜像摄像机技巧渲染的动态平面反射相组合：

```

float4 frag(vertexOutput input) : COLOR
{
    float4 staticReflColor = float4(1, 1, 1, 1);

    // Find reflected vector in WS.
    float3 viewDirWS = normalize(input.viewDirInWorld);
    float3 normalWS = normalize(input.normalInWorld);
    float3 reflDirWS = reflect(viewDirWS, normalWS);

    // Working in World Coordinate System.
    float3 localPosWS = input.vertexInWorld;
    float3 intersectMaxPointPlanes = (_BBoxMax - localPosWS) / reflDirWS;
    float3 intersectMinPointPlanes = (_BBoxMin - localPosWS) / reflDirWS;

    // Look only for intersections in the forward direction of the ray.
    float3 largestParams = max(intersectMaxPointPlanes, intersectMinPointPlanes);

    // Smallest value of the ray parameters gives us the intersection.
    float distToIntersect = min(min(largestParams.x, largestParams.y), largestParams.z);

    // Find the position of the intersection point.
    float3 intersectPositionWS = localPosWS + reflDirWS * distToIntersect;

    // Get local corrected reflection vector.
    float3 localCorrReflDirWS = intersectPositionWS - _EnvCubeMapPos;

    // Lookup the environment reflection texture with the right vector.
    float4 staticReflColor = texCUBE(_Cube, localCorrReflDirWS);

    // Lookup the planar runtime texture
    float4 dynReflColor = tex2Dproj(_ReflectionTex,
        UNITY_PROJ_COORD(input.vertexInScreenCoords));

    // Revert the blending with the background color of the reflection camera
    dynReflColor.rgb /= (dynReflColor.a < 0.00392)?1:dynReflColor.a;

    // Combine static environment reflections with dynamic planar reflections
    float4 combinedRefl = lerp(staticReflColor.rgb, dynReflColor.rgb, dynReflColor.a);

    // Lookup the texture color.
    float4 texColor = tex2D(_MainTex, float2(input.tex));
    return _AmbientColor + texColor * _ReflAmount * combinedRefl;
}

```

从平面运行时反射纹理 `_ReflectionTex` 提取纹理颜色 `dynReflColor`。

在着色器中将 `_ReflectionTex` 声明为统一变量。

在 **Property** 块中声明 `_ReflectionTex`。这可使您能够查看它在运行时的外观，从而帮助您在开发游戏期间进行调试。

为查找纹理，可投射纹理坐标，即：将纹理坐标除以坐标向量的最后一个分量。您可以使用 **Unity** 内置函数 `UNITY_PROJ_COORD()` 进行此操作。

使用 `lerp()` 函数组合静态环境反射和动态平面反射。组合以下所列：

- 反射颜色。
- 反射表面的纹理颜色。
- 环境颜色分量。

### 6.3.3 组合远距离环境的反射

在渲染静态和动态对象的反射时，您可能也必须考虑来自远距离环境的反射。例如，通过局部环境中某一窗户可见的天空反射。

在这种情形中，您必须组合三种不同的反射：

- 来自静态环境的反射，它使用的是局部立方体贴图技巧。
- 来自动态对象的平面反射，它使用的是镜像摄像机技巧。
- 来自天空盒的反射，它使用的是标准立方体贴图技巧。反射向量在从立方体贴图获取纹理前不需要修正。

要整合来自天空盒的反射，可使用反射向量 `reflDirWS` 从天空盒立方体贴图获取像素元。将天空盒立方体贴图纹理作为统一变量传递到着色器。

#### 备注

请勿应用局部修正。

为确保天空盒仅可从窗户可见，请在为反射烘焙静态立方体贴图时在 **alpha** 通道渲染场景的透明度。

如果是不透明几何体，分配值一；如果没有几何体或几何体全部透明，分配值零。例如，在 **alpha** 通道中使用零渲染与窗户对应的像素。

将天空盒立方体贴图 `_Skybox` 作为统一变量传递到着色器。

在 [6.3.2 组合反射着色器实施（第 6-116 页）](#) 内的片段着色器代码中，查找下面这条注释：

```
// Lookup the planar runtime texture
```

在该注释前插入以下几行：

```
float4 skyboxReflColor = texCUBE(_Skybox, reflDirWS);
staticReflColor = lerp(skyboxReflColor.rgb, staticReflColor.rgb, staticReflColor.a);
```

此代码将静态反射与来自天空盒的反射组合。

下图显示了组合不同类型的反射：



图 6-31 组合不同类型的反射

## 6.4 基于局部立方体贴图的动态软阴影

此技巧使用局部立方体贴图保存代表静态环境透明度的纹理。此技巧在生成高质量软阴影时效率非常高。

本节包含以下小节：

- [6.4.1 关于基于局部立方体贴图的动态软阴影（第 6-120 页）](#)。
- [6.4.2 生成阴影立方体贴图（第 6-120 页）](#)。
- [6.4.3 渲染阴影（第 6-121 页）](#)。
- [6.4.4 组合立方体贴图阴影与阴影贴图（第 6-124 页）](#)。
- [6.4.5 立方体贴图阴影技巧的结果（第 6-125 页）](#)。

### 6.4.1 关于基于局部立方体贴图的动态软阴影

在您的场景中，既存在移动的对象，也有房间等静态环境。通过使用此技巧，您不必对每一帧渲染静态几何体到阴影贴图。这可让您通过纹理来表示阴影。

立方体贴图可以很好地逼近许多种类的静态局部环境，例如冰穴演示中的洞穴等不规则形状。**Alpha** 通道也可表示进入房间的光线量。

运动的对象通常是除了房间外的一切。这些对象包括：

- 太阳。
- 摄像机。
- 动态对象。

通过用立方体贴图来表示整个房间，您可以在一个片段着色器内访问环境的任意像素元。例如，这意味着太阳可以在任意位置上，您也可以根据从立方体贴图获取的值计算照射到片段上的光线量。

**Alpha** 通道或透明度可以表示进入房间的光线量。在您的场景中，将立方体贴图纹理附加到片段着色器上，这些着色器渲染您要为其添加阴影的静态和动态对象。

### 6.4.2 生成阴影立方体贴图

首先着手于您要向其应用来自环境外部光源的阴影的局部环境。例如，房间、洞穴或笼子。

此技巧类似于基于局部立方体贴图的反射。有关更多信息，请参见 [6.2 使用局部立方体贴图实现反射（第 6-98 页）](#)。

创建阴影立方体贴图的方式与创建反射立方体贴图相同，但您还必须添加 **alpha** 通道。**Alpha** 通道或透明度可以表示进入房间的光线量。

算出您要从中渲染立方体贴图六个面的位置。在大多数情形中，这是局部环境包围盒的中心。您需要此位置来生成立方体贴图。还必须将此位置传递到着色器，从而计算局部修正向量，以便从立方体贴图获取正确的像素元。

决定了立方体贴图中心的位置后，您可以将所有面渲染到立方体贴图纹理，再记录局部环境的透明度或 **alpha** 通道。一个区域的透明度越高，进入环境中的光线越多。如果没有几何体，则完全透明。必要时，您可以使用 **RGB** 通道存储彩色阴影的环境颜色，如有色玻璃、反射或折射。

### 6.4.3 渲染阴影

在世界空间中构建从顶点或片段到一个/多个光源的向量  $P_i L$ ，然后使用此向量获取立方体贴图阴影。

在获取每个像素元前，必须对  $P_i L$  向量应用局部修正。ARM 建议在片段着色器中进行局部修正，从而获得更加准确的阴影。

要计算局部修正，您必须计算片段至光源向量与环境包围盒的交叉点。使用此交叉点构建从立方体贴图原点位置  $C$  到交叉点  $P$  的另一向量。这可为您提供用于获取像素元的最终向量  $C P$ 。

您需要下列输入参数来计算局部修正：

- $_EnvCubeMapPos$  立方体贴图原点位置。
- $_BboxMax$  环境包围盒的最大点。
- $_BboxMin$  环境包围盒的最小点。
- $P_i$  世界空间中的片段位置。
- $P_i L$  世界空间中正规化片段至光源向量。

计算输出值  $C P$ 。这是修正后的片段至光源向量，您要用它从阴影立方体贴图获取像素元。

下图显示了片段至光源向量的局部修正。

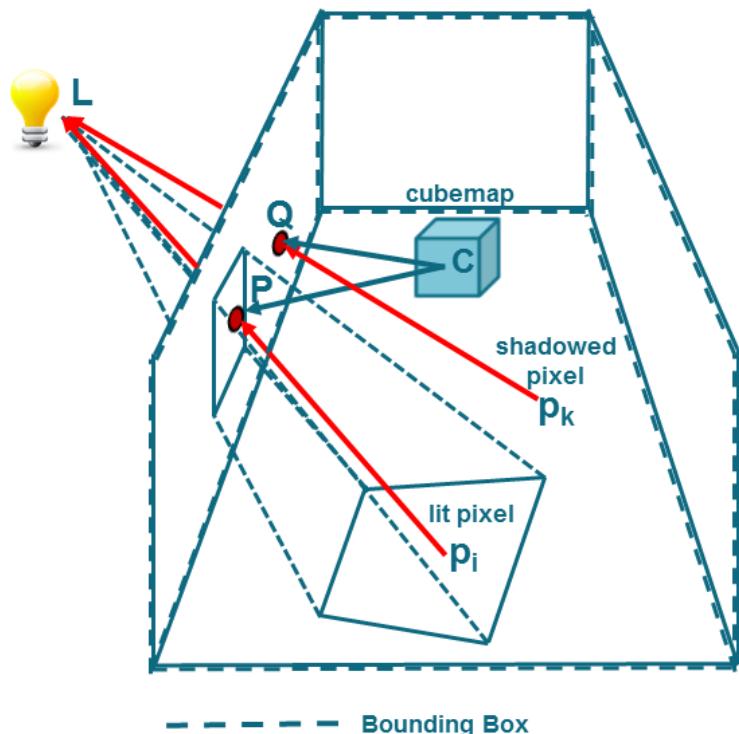


图 6-32 片段至光源向量的局部修正

下列示例代码演示了如何正确计算  $CP$  向量：

```
// Working in World Coordinate System.
vec3 intersectMaxPointPlanes = (_BBoxMax - Pi) / Pil;
vec3 intersectMinPointPlanes = (_BBoxMin - Pi) / Pil;

// Looking only for intersections in the forward direction of the ray.
vec3 largestRayParams = max(intersectMaxPointPlanes, intersectMinPointPlanes);

// Smallest value of the ray parameters gives us the intersection.
```

```

float dist = min(min(largestRayParams.x, largestRayParams.y), largestRayParams.z);

// Find the position of the intersection point.
vec3 intersectPositionWS = Pi + Pil * dist;

// Get the local corrected vector.
CP = intersectPositionWS - _EnviCubeMapPos;

```

使用 CP 向量从立方体贴图获取像素元。像素元的 **alpha** 通道提供关于必须向片段应用多少光线或阴影的信息：

```
float shadow = texCUBE(cubemap, CP).a;
```

下图显示了具有硬阴影的棋牌室：



图 6-33 具有硬阴影的棋牌室

此技巧可以在场景中生成有效的阴影，但您可以通过两个额外的步骤来提高阴影的质量：

- 阴影中的背面
- 平滑

### 阴影中的背面

立方体贴图阴影技巧不使用深度信息来应用阴影。这意味着某些面在应当要处于阴影中时会得到错误的照明。

只有表面朝向光源相反的方向时才会出现此问题。要解决此问题，可检查法线向量与片段至光源向量  $P_iL$  之间的角度。如果角度数超出 -90 到 90 度范围，该表面处于阴影中。

下列代码片段进行此检查：

```

if (dot(PiL, N) < 0)
shadow = 0.0;

```

以上代码导致每个三角形从亮硬切换至暗。若要获得平滑过渡，可使用下列公式：

```
shadow *= max(dot(PiL, N), 0.0);
```

其中：

- shadow 是从阴影立方体贴图获取的 alpha 值。
- $P_iL$  是世界空间中的正规化片段至光源向量。
- N 是表面在世界空间中的法线向量。

下图显示了具有阴影中背面的棋牌室：



图 6-34 具有阴影中背面的棋牌室

### 平滑

此阴影技巧可以在您的场景中提供逼真的软阴影。

1. 生成纹理映射，并为立方体贴图纹理设置三线性过滤。
2. 测量片段至交叉点向量的长度。
3. 将该长度乘以一个系数。

该系数是环境中最大距离与纹理映射层级数的正规化子。您可以使用包围区域和纹理映射层级数自动计算它。您必须按照自己的场景自定义该系数。这可让您调整相关的设置，使它适合您的环境，从而改善视觉质量。例如，冰穴项目中使用的系数为 0.08。

您可以重新利用为局部修正所进行的计算的结果。重新利用代码片段中局部修正的 dist，作为从片段位置到片段至光源向量与包围盒交叉点的线段的长度：

```
float texLod = dist;
```

将 texLod 乘以距离系数：

```
texLod *= distanceCoefficient;
```

要实施柔度，使用 Cg 函数 texCUBElod() 或 GLSL 函数 textureLod() 获取纹理的正确纹理映射层级。

构造一个 vec4，其中 XYZ 代表方向矢量，w 分量则代表 LOD。

```
CP.w = texLod;
shadow = texCUBElod(cubemap, CP).a;
```

此技巧可为您的场景提供高质量的平滑阴影。

下图显示了具有平滑阴影的棋牌室：



图 6-35 平滑阴影

#### 6.4.4 组合立方体贴图阴影与阴影贴图

要利用动态内容完善阴影，必须组合使用立方体贴图阴影和传统的阴影贴图技巧。这需要额外的工作，但值得一试，因为您只需要将动态对象渲染到阴影贴图。

下图显示了仅具有平滑阴影的棋牌室：



图 6-36 平滑阴影

下图显示了组合了平滑阴影和动态阴影的棋牌室：



图 6-37 平滑阴影与动态阴影组合

#### 6.4.5 立方体贴图阴影技巧的结果

使用传统的技巧时，渲染阴影的成本比较高，因为它涉及从每个阴影投射光源的视角渲染整个场景。此处介绍的立方体贴图阴影技巧可以提高性能，因为它大部分是预烘焙的。

此技巧还独立于输出分辨率。它在 1080p、720p 和其他分辨率上产生相同的视觉质量。

柔和度过滤是在硬件中计算的，所以平滑几乎没有计算上的成本。阴影越平滑，此技巧的效用越佳。这是因为较小的纹理映射层级数产生的数据要少于传统的阴影贴图技巧。传统的技巧需要较大的内核才能使阴影足够平滑，从而在视觉上更吸引人。这需要很高的内存带宽，所以会降低性能。

通过立方体贴图阴影技巧获得的质量可能会超越您的预期。它提供逼真的柔和度，阴影稳定，没有闪光的边缘。由于光栅化和锯齿效应，使用传统的阴影贴图技巧时可能会看到闪光的边缘。不过，所有抗锯齿算法都不能完全修复此问题。

立方体贴图阴影技巧没有闪光问题。边缘稳定，即使您使用的分辨率远低于渲染目标所用的分辨率。您可以使用比输出低四倍的分辨率，而且没有失真或多余的闪光。使用低四倍的分辨率也可节省内存带宽，因而能提升性能。

此技巧可用于市面上支持着色器的任何设备，比如支持 OpenGL ES 2.0 或更高版本的设备。如果您已清楚使用基于局部立方体贴图技术的反射的位置和时机，您就可以轻松在实施中应用此阴影技巧。

#### 备注

该技巧无法用于场景中的一切对象。例如，动态对象从立方体贴图接收阴影，但它们无法预烘焙到立方体贴图纹理。对于动态对象，请使用阴影贴图来生成阴影，并与立方体贴图阴影技巧搭配使用。

下图显示了具有阴影的冰穴：



图 6-38 具有阴影的冰穴

下图显示了具有平滑阴影的冰穴：



图 6-39 具有平滑阴影的冰穴

下图显示了具有平滑阴影的冰穴：



图 6-40 具有平滑阴影的冰穴

## 6.5 基于局部立方体贴图的折射

您可以使用局部立方体贴图实施高质量折射，也可以在运行时将它们与反射组合。

本节包含以下小节：

- [6.5.1 关于折射（第 6-128 页）。](#)
- [6.5.2 折射实施（第 6-128 页）。](#)
- [6.5.3 关于基于局部立方体贴图的折射（第 6-129 页）。](#)
- [6.5.4 准备立方体贴图（第 6-129 页）。](#)
- [6.5.5 着色器实施（第 6-131 页）。](#)

### 6.5.1 关于折射

游戏开发人员经常寻觅高效的方法，在游戏中实施夺人眼球的视觉特效。以移动平台为目标开发时，这尤为重要，因为您必须仔细权衡各种资源，从而获得最高的性能。

折射是光波因为它所穿透的介质的变化而改变方向。如果希望提高半透明几何体的逼真度，折射是您要考虑的一个重要效果。

折射率决定了光线进入一种物质时弯折或折射的程度。折射定义为光线从折射率为  $n_1$  的一种介质穿入折射率为  $n_2$  的另一介质时的弯折。

您可以使用斯涅耳定律计算折射率和入射与折射角度正弦之间的关系。

下图显示了斯涅耳定律和折射：

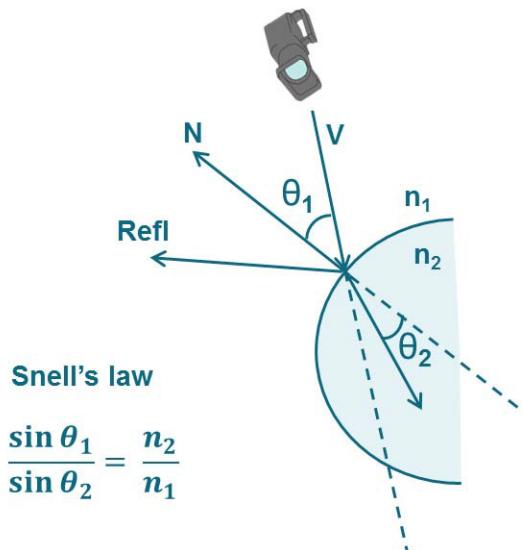


图 6-41 光线穿过一种介质射入另一种介质时的折射

### 6.5.2 折射实施

开发人员自开始渲染反射时就已尝试渲染折射，因为任何半透明表面上都会同时发生这些过程。渲染反射的技巧多种，但渲染折射的却不多。

运行时实施折射的现有方法根据具体的折射类型而不同。大部分技巧是运行时将折射对象后的场景渲染到纹理，然后在第二通道中应用纹理失真，从而获得折射的外观。根据纹理失真，您可以使用这种方法来渲染不同的折射效果，如水、热雾、玻璃和其他效果。

其中一些技巧可以获得不错的成绩，但纹理失真不是以物理学为基础，因此结果不一定正确。例如，如果您从折射摄像机的视角渲染纹理，可能会有一些区域不直接对该摄像机可见，但在基于物理的折射中可见。

使用渲染至纹理方法的主要局限在于质量。当摄像机移动时，经常会出现像素闪光或像素不稳定的现象。

### 6.5.3 关于基于局部立方体贴图的折射

局部立方体贴图是一种优良的反射渲染技巧，开发人员自它们可用时便开始将静态立方体贴图同时用于实施反射和折射。

不过，如果您在局部环境中使用静态立方体贴图实施反射或折射，倘若不应用局部修正，结果就会不正确。

此处描述的技巧中，通过应用局部修正来确保正确的结果。此技巧已经过高度优化。它对于移动设备特别有用，因为运行时资源有限，所以必须仔细均衡。

### 6.5.4 准备立方体贴图

您必须准备立方体贴图，以便在折射实施中使用：

若要准备立方体贴图，请执行以下操作：

1. 将摄像机置于折射几何体的中心。
2. 隐藏折射对象，并将六个方向上周围静态环境渲染到立方体贴图。您可以将这一立方体贴图同时用于实施折射和反射。
3. 将围绕折射对象的环境烘焙到静态立方体贴图。
4. 确定折射向量的方向，再找到它与局部环境包围盒的相交处。
5. 按照与 [6.4 基于局部立方体贴图的动态软阴影（第 6-120 页）](#) 中相同的方式，应用局部修正。
6. 生成从立方体贴图生成的点到交叉点的一个新向量。使用这一最终向量从立方体贴图获取像素元，渲染折射对象背后的内容。

我们不使用折射向量  $R_{rf}$  从立方体贴图获取像素元，而是找到折射向量与包围盒相交的点  $P$ ，再构建一个从立方体贴图中心  $C$  到交叉点  $P$  的新向量  $R'_{rf}$ 。使用这一新向量从立方体贴图获取纹理颜色。

```
float eta=n2/n1;  
float3 Rrf = refract(D,N,eta);  
找到交叉点 P  
找到向量 R'rf = CP;  
Float4 col = texCube(Cubemap, R'rf);
```

下图显示了带有立方体贴图和折射向量的场景：

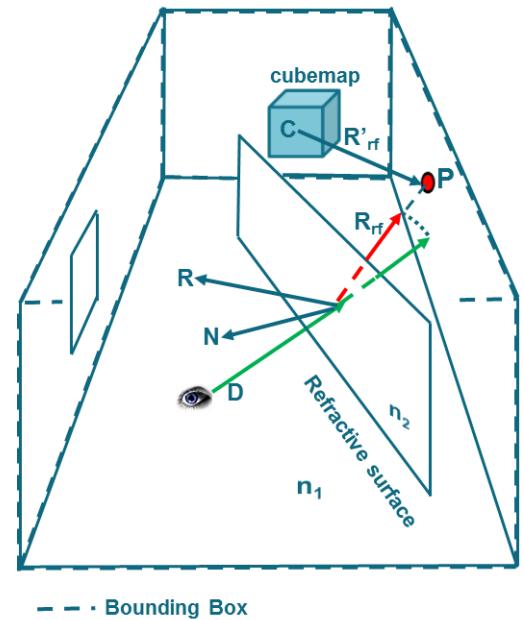


图 6-42 折射向量局部修正

此技巧生成的折射在物理学上是准确的，因为折射向量方向是通过斯涅耳定律计算的。

您还可以在着色器中使用一个内置函数，找到严格遵循斯涅耳定律的折射向量 R：

```
R = refract( I, N, eta);
```

其中：

- I 是正规化视角或入射向量。
- N 是正规化法线向量。
- eta 是折射率的比率  $n_1/n_2$ 。

下图显示了基于局部立方体贴图的折射着色器实施流程：

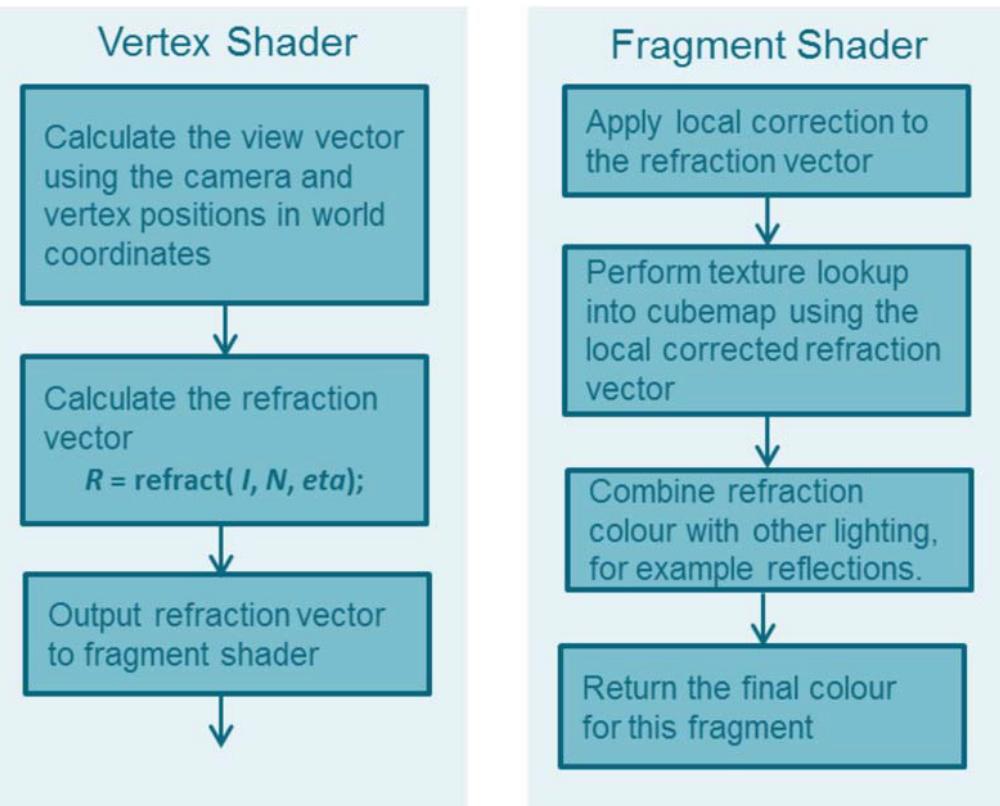


图 6-43 基于局部立方体贴图的折射着色器实施

### 6.5.5 着色器实施

在获取与局部修正折射方向对应的像素元时，您可能要将折射颜色与其他光照组合。例如，与折射同时发生的反射。

要将折射颜色与其他光照组合，您必须传递一个额外的视角向量到片段着色器，再向它应用局部修正。使用其结果从同一立方体贴图获取折射颜色。

下列代码片段演示了如何组合反射和折射来生成最终的输出颜色：

```

// ----- Environment reflections -----
float3 newReflDirWS = LocalCorrect(input.reflDirWS, _BBoxMin, _BBoxMax, input.posWorld,
_EnviCubeMapPos);
float4 staticReflColor = texCUBE(_EnviCubeMap, newReflDirWS);
// ----- Environment refractions -----
float3 newRefractDirWS = LocalCorrect(RefractDirWS, _BBoxMin, _BBoxMax, input.posWorld,
_EnviCubeMapPos);
float4 staticRefractColor = texCUBE(_EnviCubeMap, newRefractDirWS);
// ----- Combined reflections and refractions -----
float4 combinedReflRefract = lerp(staticReflColor, staticRefractColor, _ReflAmount);

float4 finalColor = _AmbientColor + combinedReflRefract;

```

系数 `_ReflAmount` 作为统一变量传递到片段着色器。利用此系数调整反射与折射占比之间的均衡。您可以手动调整 `_ReflAmount` 获得自己想要的视觉效果。

您可以在以下反射博客中找到 `LocalCorrect` 函数的实施：

<http://community.arm.com/groups/arm-mali-graphics/blog/2014/08/07/reflections-based-on-local-cubemaps>.

当折射几何体是中空物体时，折射和反射会在正面和背面同时发生。

下图显示了玻璃象棋子上基于立方体贴图的折射：



图 6-44 玻璃象棋子上基于立方体贴图的折射

左图显示了第一通道，它使用局部折射和反射仅渲染了背面。

右图显示了第二通道，它使用局部折射和反射以及与第一通道的 **alpha** 混合渲染了正面。

- 在第一通道中，与您渲染不透明几何体时一样渲染半透明对象。打开正面剔除选项，最后渲染该对象，即仅渲染其背面。您不想要遮挡其他对象，所以请勿写入深度缓冲区。  
其背面颜色的获得方式为混合根据对象本身反射、折射和漫射颜色计算的颜色。
- 在第二通道中，使用背面剔除渲染正面，此工作放在渲染队列的末尾执行。确保深度写入为关闭。将折射和反射纹理与漫射颜色混合，从而获得正面颜色。第二通道中的折射为最终的渲染增加了真实度。如果背面的折射已足以突出效果，您可以跳过此步骤。
- 在最终通道中，您要将生成的颜色与第一通道进行 **alpha** 混合。

下图显示了冰穴演示中对半透明凤凰实施基于局部立方体贴图的折射的结果。

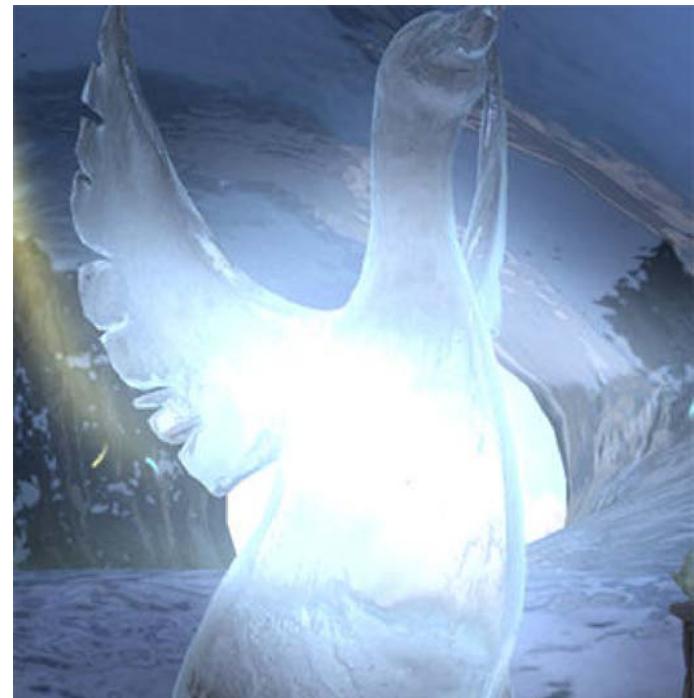


图 6-45 半透明凤凰折射

下图显示了半透明凤凰翅膀：



图 6-46 半透明凤凰翅膀

## 6.6 冰穴演示中的镜面反射效果

冰穴演示中的镜面反射效果是利用 Blinn 技巧实施的。此技巧非常高效，可以产生很好的结果。

下列代码演示了如何使用 Blinn 技巧实施镜面反射效果：

```
// Returns intensity of a specular effect without taking into account shadows
float SpecularBlinn(float3 vert2Light, float3 viewDir, float3 normalVec, float4 power)
{
    float3 floatDir = normalize(vert2Light - viewDir);
    float specAngle = max(dot(floatDir, normalVec), 0.0);
    return pow(specAngle, power);
}
```

Blinn 技巧的一个缺点在于它在某些条件下可能会产生不正确的结果。例如，阴影中的区域可能不会有镜面反射效果。

下图显示了没有镜面反射效果的阴影区域示例：



图 6-47 阴影区域

下图显示了阴影区域中出现镜面反射效果的示例。这些是错误的：



图 6-48 带有错误镜面反射效果的阴影区域

下图显示了光亮区域中出现镜面反射效果的示例。这是正确的：



图 6-49 带有正确镜面反射效果的光亮区域

阴影应当会使得镜面反射效果的强度变强或变弱，具体取决于到达镜面表面的光线。冰穴演示中已经具备了修正镜面反射效果强度的所有信息，所以修复此问题相对容易。用于反射和阴影效果的环境立方体贴图纹理包含两种信息。RGB 通道包含用于反射的环境颜色。Alpha 通道包含用于阴影的不透明度。您可以使用 alpha 通道来确定镜面反射强度，因为 alpha 通道代表了洞穴中让光线进入环境的孔洞。Alpha 通道因此可用于确保镜面反射效果仅应用到被光线照射到的表面。

为此，需要在片段着色器中计算修正的反射向量来生成反射效果。有关创建修正后反射向量的更多信息，请参见 [6.2 使用局部立方体贴图实现反射（第 6-98 页）](#)。使用此向量从立方体贴图纹理获取 RGBA 像素元：

```
// Locally corrected static reflections
const half4 reflColor = SampleCubemapWithLocalCorrection(
    ReflDirectionWS,
    _ReflBBoxMinWorld,
    _ReflBBoxMaxWorld,
    input.vertexInWorld,
    _ReflCubePosWorld,
    _ReflCube);
```

有关 `SampleCubemapWithLocalCorrection()` 函数定义的更多信息，请参见 [6.2 使用局部立方体贴图实现反射（第 6-98 页）](#)。

`reflColor` 是 RGBA 格式，其中 RGB 分量包含用于反射的颜色数据，而 alpha 通道则包含镜面反射效果的强度。在冰穴演示中，alpha 通道基于镜面反射颜色翻倍，即使用 Blinn 技巧进行计算：

```
half3 specular = _SpecularColor.rgb *
    SpecularBlinn(
        input.vertexToLight01InWorld,
        viewDirInWorld,
        normalInWorld,
        _SpecularPower) *
    reflColor.a;
```

`specular` 值代表最终的镜面反射颜色。您可以将此添加到光照模型中。

## 6.7 使用 Early-z

Mali GPU 包含了执行 Early-Z 算法的功能。Early-Z 可通过去除过度绘制的片段来提升性能。

Mali GPU 通常对大部分内容执行 Early-Z 算法，但在一些情形中并不执行，以达到确保正确度的目的。这可能比较难以从 Unity 内部加以控制，因为它依赖于 Unity 引擎以及编译器生成的代码。不过，您可以查看一些迹象。

针对移动平台编译您的着色器，再查看其代码。确保您的着色器不会落入下列类别之一：

### 着色器有副作用

这意味着着色器线程在执行期间修改了全局状态，因此二次执行该着色器可能会产生不同的结果。通常，这表示您的着色器写入到共享的读取/写入内存缓冲区，如着色器存储缓冲区对象或图像。例如，如果您创建通过递增计数器来测量性能的着色器，它就有副作用。

如下所列不归类为副作用：

- 只读内存访问。
- 写入到仅写入缓冲区。
- 纯粹的局部内存访问。

### 着色器调用 `discard()`

如果片段着色器在执行期间可以调用 `discard()`，那么 Mali GPU 无法启用 Early-Z。这是因为，片段着色器可以丢弃当前的片段，但深度值之前被 Early-Z 测试修改，而这是无法逆转的。

### 启用了 Alpha-to-coverage

如果启用了 Alpha-to-coverage，则片段着色器会计算稍后为定义 `alpha` 而要访问的数据。

例如，在渲染一棵树的树叶时，它们通常会表示为平面，其纹理定义树叶的哪个区域是透明还是不透明。如果启用了 Early-Z，您会获得不正确的结果，因为场景的一部分可能会被该平面的透明部分遮挡。

### 深度源不固定的函数

用于深度测试的深度值不来自顶点着色器。如果您的片段着色器写入到 `gl_FragDepth`，Mali GPU 无法执行 Early-Z 测试。

## 6.8 脏镜头光晕效果

您可以使用脏镜头光晕效果来实现戏剧感。它通常与镜头光晕效果一起使用。

您可以通过非常轻松和简单的方式来实施脏镜头光晕效果，这种方式很适合移动设备。

在冰穴演示中，脏镜头效果是在一个功能最少的着色器中实施的，该着色器在场景基础上渲染一个强度可变的全屏四边形。四边形的强度通过一个脚本传递。

此着色器在所有透明几何体都渲染后的最终阶段，使用附加的 `alpha` 混合渲染该四边形。

本节包含以下小节：

- [6.8.1 着色器实施（第 6-138 页）。](#)
- [6.8.2 脚本实施（第 6-140 页）。](#)
- [6.8.3 脏镜头着色器代码（第 6-140 页）。](#)

### 6.8.1 着色器实施

本节介绍脏镜头着色器。

如需脏镜头着色器的完整源代码，请参见 [6.8.3 脏镜头着色器代码（第 6-140 页）](#)。

下列子着色器标记指示全屏四边形在所有不透明几何体和九个其他透明对象之后渲染：

```
Tags {"Queue" = "Transparent+10"}
```

着色器使用以下命令停用深度缓冲区写入。这可以防止四边形遮挡它后面的几何体：

```
ZWrite Off
```

在混合阶段，片段着色器的输出与帧缓冲区中已有像素颜色混合。

着色器指定了附加混合类型 `Blend One One`。在这一混合类型中，来源和目标因子都是 `float4 (1.0, 1.0, 1.0, 1.0)`。

这种混合类型通常用于粒子系统来表示火焰等透明并且发光的效果。

着色器停用了剔除和 `ZTest`，以确保始终渲染脏镜头光晕效果。

四边形的顶点在视口坐标中定义，所以顶点着色器中不会发生顶点变换。

片段着色器仅从纹理获取像素元，再应用与效果强度成比例的因子。

下图显示了用作脏镜头光晕效果使用的全屏四边形纹理的图像：

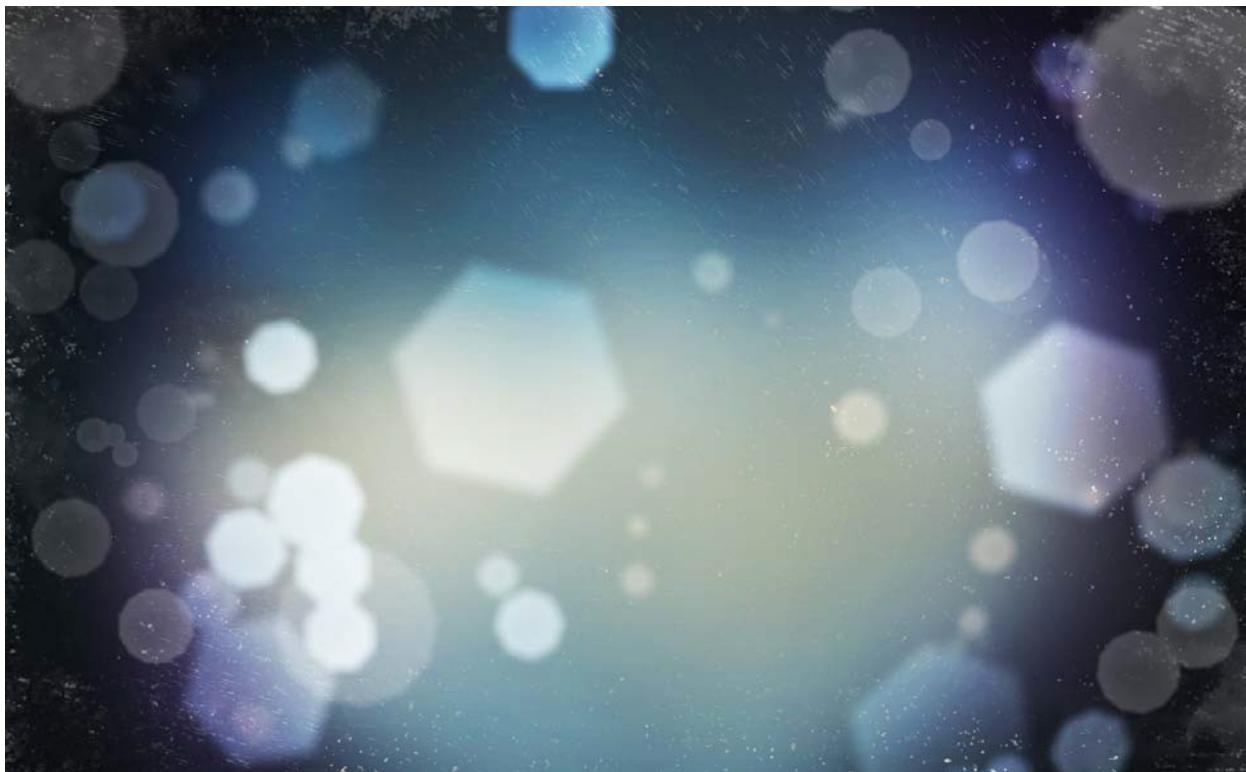


图 6-50 冰穴演示中脏镜头光晕效果使用的纹理

下图显示了冰穴演示中脏镜头光晕效果呈现的样貌，图中所示为摄像机朝向来自洞穴入口的阳光：



图 6-51 冰穴演示中所实施的脏镜头效果

## 6.8.2 脚本实施

一个简单脚本实施全屏四边形，并且计算传递到片段着色器的强度因子。

下列函数创建了 Start 函数中的四边形网格：

```
void CreateQuadMesh()
{
    Mesh mesh = GetComponent<MeshFilter>().mesh;
    mesh.Clear();
    mesh.vertices = new Vector3[] {new Vector3(-1, -1, 0), new Vector3(1, -1, 0),
        new Vector3(1, 1, 0), new Vector3(-1, 1, 0)};
    mesh.uv = new Vector2[] {new Vector2(0, 0), new Vector2(1, 0),
        new Vector2(1, 1), new Vector2(0, 1)};
    mesh.triangles = new int[] {0, 2, 1, 0, 3, 2};
    mesh.RecalculateNormals();

    //Increase bounds to avoid frustum clipping.
    bigBounds.SetMinMax(new Vector3(-100, -100, -100), new Vector3(100, 100, 100));
    mesh.bounds = bigBounds;
}
```

创建该网格时，其边界将递增，以确保视锥体绝不会被裁剪。根据您场景的尺寸来设置边界的大小。

该脚本计算强度因子，并将它传递到片段着色器。这基于摄像机至太阳向量和摄像机前向向量的相对朝向。当摄像机正对太阳时，该效果的强度达到最大值。

下列代码演示了强度因子的计算：

```
Vector3 cameraSunVec = sun.transform.position - Camera.main.transform.position;
cameraSunVec.Normalize();
float dotProd = Vector3.Dot(Camera.main.transform.forward, cameraSunVec);
float intensityFactor = Mathf.Clamp(dotProd, 0.0f, 1.0f);
```

## 6.8.3 脏镜头着色器代码

这是 DirtyLensEffect.shader 的代码：

```
half3 normalInWorld = half3(0.0,0.0,0.0);
half3 bumpNormal = UnpackNormal(tex2D(_BumpMapGlobal, input.tc));
half3x3 local2WorldTranspose = half3x3(input.tangentWorld,
    input.bitangentWorld,
    input.normalInWorld);
normalInWorld = normalize(mul(bumpNormal, local2WorldTranspose));
normalInWorld = normalInWorld*0.5 + 0.5;
return half4(normalInWorld,1.0);
```

## 6.9 光柱

光柱模拟云隙光、大气散射或阴影的效果。它们用于为场景增加深度和真实感。

本节包含以下小节：

- [6.9.1 关于光柱 \(第 6-141 页\)](#)。
- [6.9.2 淡化圆锥体边缘 \(第 6-143 页\)](#)。

### 6.9.1 关于光柱

在冰穴演示中，光柱模拟了从洞穴顶部开口处射入洞内的阳光散射。

下图显示了冰穴演示中的光柱：

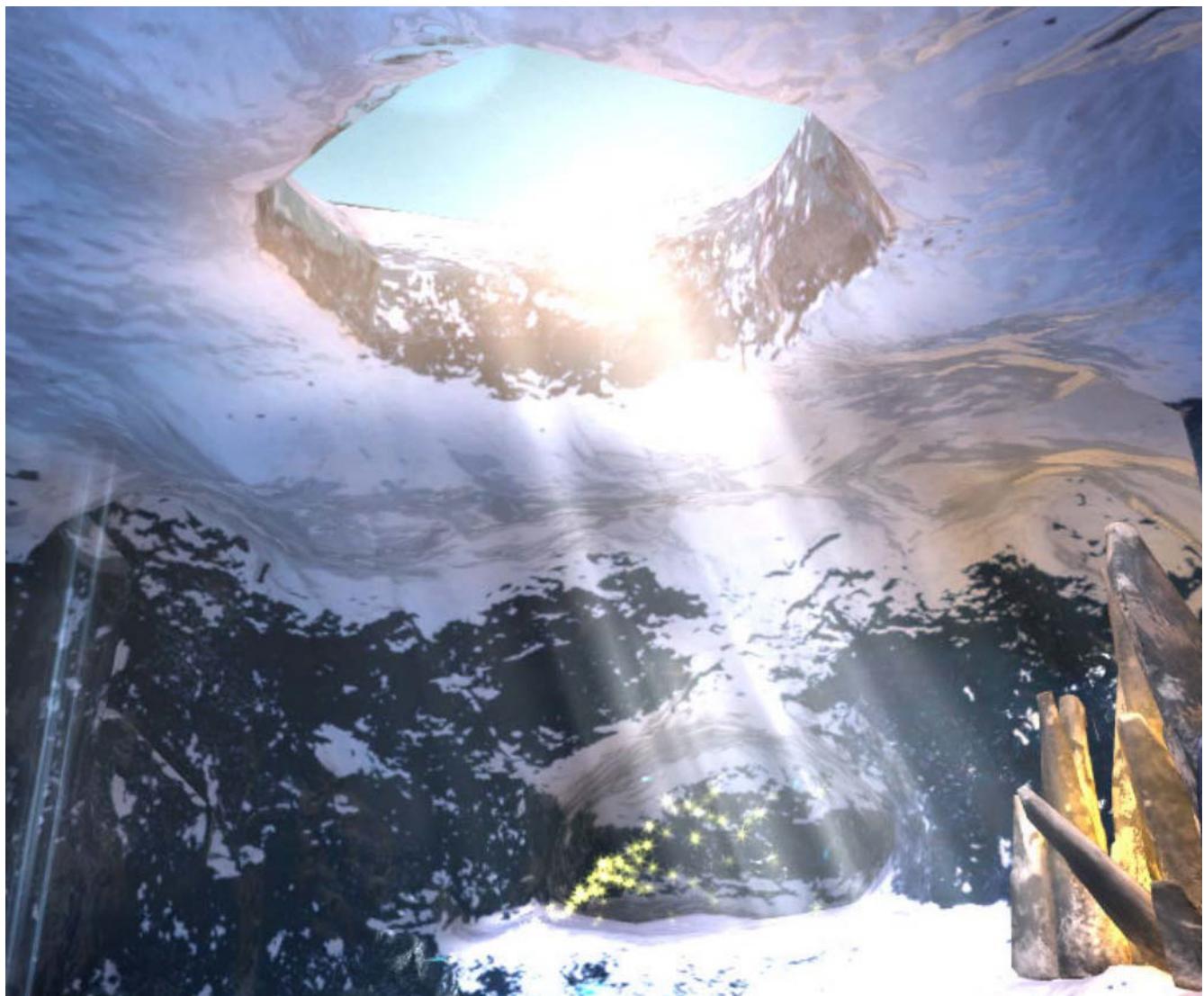


图 6-52 冰穴演示中的光柱

光柱的基础是一个截断的圆锥体。该圆锥体跟随光线的方向，其方式可确保圆锥体的上部始终固定。

下图显示了圆锥体的几何，其中：

- a 显示光柱的基本几何体是具有顶部和底部两个截面的圆柱体。
- b 显示其下截面已经扩展，根据您定义的角度  $\theta$  获得一个被截断的圆锥几何体。
- c 显示其上截面保持固定，下截面按照太阳光线的方向水平移位。

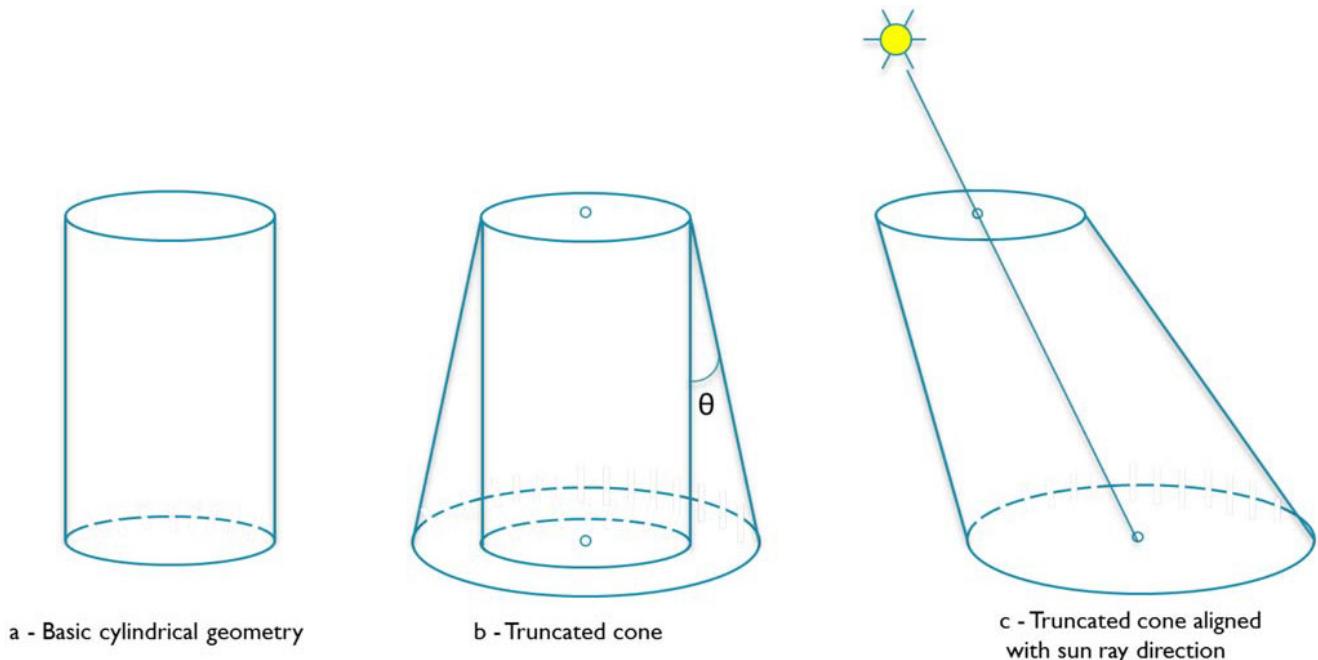


图 6-53 光柱几何体

一个脚本利用太阳的位置计算下列值：

- 圆锥体下截面扩展的幅度，它基于作为输入值的角度  $\theta$ 。
- 截面移位的方向和幅度。

顶点着色器基于此数据应用变换。在光柱局部坐标中，该变换应用到圆柱形几何体的原始顶点。

在渲染光柱时，请避免渲染显露出其几何体的任何硬边缘。要达到这一目的，您可以使用纹理遮罩来平滑淡化其顶部和底部。

下图显示了光柱纹理：



图 6-54 光柱纹理。左侧为遮罩纹理。右侧为光束纹理。

### 6.9.2 淡化圆锥体边缘

在与截面平行的平面中，根据摄像机与顶点相对朝向，淡化光柱强度。

在顶点着色器中，将摄像机位置投影到截面上。

构建一个从截面中心到投影的新向量，然后将它正规化。

计算此向量与顶点法线的点积，然后将结果提升为幂指数。

将结果作为变量传递到片段着色器。这用于调节光柱的强度。

顶点着色器在局部坐标系统(LCS)中进行这些计算。

以下代码显示顶点着色器：

```
// Project camera position onto cross section
float3 axisY = float3(0, 1, 0);
float dotWithYAxis = dot(camPosInLCS, axisY);
float3 projOnCrossSection = camPosInLCS - (axisY * dotWithYAxis);
projOnCrossSection = normalize(projOnCrossSection);

// Dot product to fade the geometry at the edge of the cross section
float dotProd = abs(dot(projOnCrossSection, input.normal));
output.overallIntensity = pow(dotProd, _FadingEdgePower) * _CurrLightShaftIntensity;
```

您可以通过系数 `_FadingEdgePower` 细调光柱边缘的淡化。

脚本传递系数 `_CurrLightShaftIntensity`。这使得光柱在摄像机靠近它时淡出。

下列代码演示了一个添加至光柱的最终润饰，它通过脚本缓慢向下滚动纹理：

```
void Update()
{
    float localOffset = (Time.time * speed) + offset;
    localOffset = localOffset % 1.0f;
    GetComponent<Renderer>().material.SetTextureOffset("_MainTex", new Vector2(0,
localOffset));
}
```

片段着色器获取光束和遮罩纹理，然后使用强度因子将它们组合：

```
float4 frag(vertexOutput input) : COLOR
{
    float4 textureColor = tex2D(_MainTex, input.tex.xy);
    float textureMask = tex2D(_MaskTex, input.tex.zw).a;
    textureColor *= input.overallIntensity * textureMask;
    textureColor.rgb = clamp(textureColor.a, 0, 1);
    return textureColor;
}
```

光柱几何体继所有不透明几何体之后，在透明队列中渲染。

它使用附加混合将片段颜色与帧缓冲区中的对应像素组合。

该着色器也禁用剔除和深度缓冲区写入，以便不遮挡其他对象。

此通道的设置为：

```
Blend One One
Cull Off
ZWrite Off
```

## 6.10 雾化效果

雾化效果可为场景增添气氛。您不需要通过高级实施来生成雾化，简单的雾化效果即可奏效。

本节包含以下小节：

- [6.10.1 关于雾化效果（第 6-145 页）。](#)
- [6.10.2 过程线性雾（第 6-145 页）。](#)
- [6.10.3 具有高度的线性雾（第 6-146 页）。](#)
- [6.10.4 非均匀雾（第 6-147 页）。](#)
- [6.10.5 预烘焙雾（第 6-147 页）。](#)
- [6.10.6 使用粒子的体积雾（第 6-147 页）。](#)

### 6.10.1 关于雾化效果

在现实世界中，您越往远看，被淡化的颜色就越多。不一定需要雾天才能看到这种效果，阳光灿烂时您也可以看到，特别是观赏高山风景时。

这在现实生活中特别常见，所以在游戏中添加此效果可以为场景增添真实感。

此部分介绍两种版本的雾化效果：

- 过程线性雾。
- 基于粒子的雾。

您可以同时应用这两种效果。冰穴演示中同时使用了这两种技巧。

### 6.10.2 过程线性雾

确保对象距离越远，其颜色更多地淡化为定义的雾颜色。要在片段着色器中实现这一目标，您可以在片段颜色和雾颜色之间使用简单线性插值。

下方示例代码演示了如何基于和摄像机的距离在顶点着色器中计算雾颜色。此颜色作为变量传递到片段着色器。

```
output.fogColor = _FogColor * clamp(vertexDistance * _FogDistanceScale, 0.0, 1.0);
```

其中：

- `vertexDistance` 是顶点至摄像机距离。
- `_FogDistanceScale` 是作为统一变量传递至着色器的因子。
- `_FogColor` 是您定义的基准雾颜色，作为统一变量传递。

在片段着色器中，插值的 `input.fogColor` 与片段颜色 `output.Color` 组合。

```
outputColor = lerp(outputColor, input.fogColor.rgb, input.fogColor.a);
```

下图显示了您在场景中获得的结果：



图 6-55 基于距离的线性雾

——备注——

在着色器中计算雾意味着您不需要执行任何额外的后处理来生成雾化效果。

设法将尽可能多的效果合并到一个着色器中。但是，务必要检查性能，因为超出缓存时性能可能会降低。您可能必须将着色器拆分为两个或更多通道。

您可以在顶点着色器或片段着色器中进行雾颜色计算。在片段着色器中计算更加准确，但需要更多计算性能，因为它是针对每个片段计算的。

顶点着色器计算精度较低，但性能也较高，因为它仅针对每一顶点计算一次。

### 6.10.3 具有高度的线性雾

雾在场景中均匀地应用。您可以按照高度更改密度，让它更具真实感。

提高低处的雾气浓度，降低高处的雾气浓度。您可以显示高度值，以进行手动调整。

下图显示了基于距离和高度的线性雾：



图 6-56 基于距离和高度的线性雾

#### 6.10.4 非均匀雾

雾不一定是均匀的。您可以引入一些噪点，让雾的视觉效果更加引人。

您可以应用噪点纹理创建非均匀雾。

若要获得更加复杂的效果，还可以应用多个噪点纹理，让它们以不同的速度滑动。例如，距离较远的噪点纹理的滑动速度慢于离摄像机较近的纹理。

您可以在一个着色器中的一个通道中应用多个纹理，只需根据距离混合噪点纹理。

#### 6.10.5 预烘焙雾

如果知道摄像机不会靠近它们，您可以将雾预烘焙到纹理中。这可以降低所需的计算功率。

#### 6.10.6 使用粒子的体积雾

您可以使用粒子模拟体积雾。这可以产生高质量的结果。

将粒子数量保持到最小值。粒子会增加过度绘制，因为每个片段要执行更多的着色器。尝试使用较大的粒子，而不要创建更多个粒子。

在冰穴演示中，一次使用了最多 15 个粒子。

##### 几何体取代公告板渲染

将粒子系统的渲染模式设置为网格，不要使用公告板。这是因为，要获得体积效果，各个粒子必须随机旋转。

下图显示了没有纹理的粒子几何体：



图 6-57 没有纹理的粒子

下图显示了带有纹理的粒子几何体：



图 6-58 带有纹理的粒子

下图显示了应用至各个粒子的纹理：

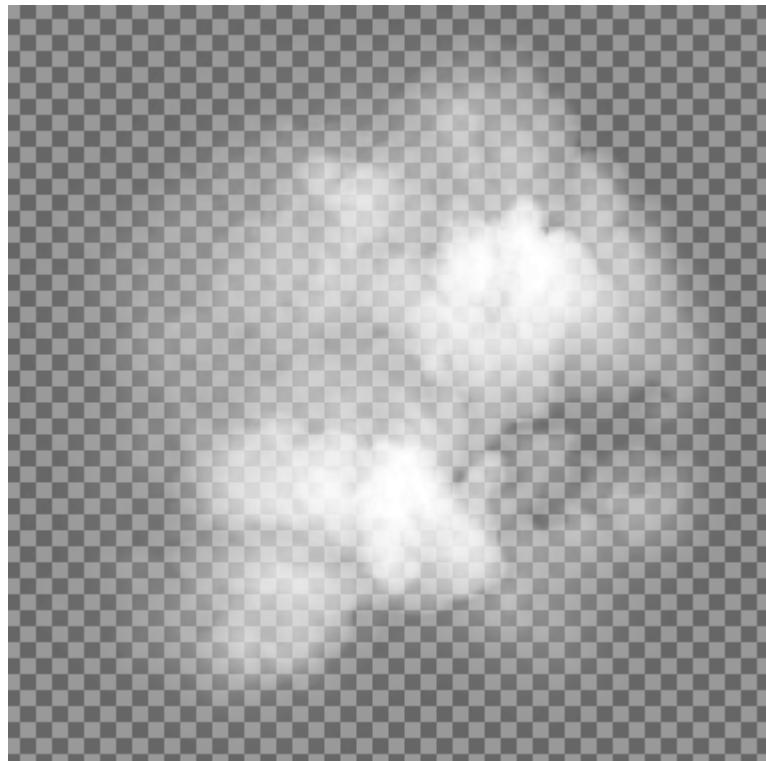


图 6-59 个别粒子的纹理

### 角度淡化效果

按照粒子相对于摄像机位置的朝向，淡入和淡出各个粒子。如果您不淡入和淡出粒子，粒子就会显现锐利边缘。下列示例代码演示了执行淡化的顶点着色器：

```
half4 vertexInWorld = mul(_Object2World, input.vertex);
half3 normalInWorld = (mul(half4(input.normal, 0.0), _World2Object).xyz);
const half3 viewDirInWorld = normalize(vertexInWorld - _WorldSpaceCameraPos);
output.visibility = abs(dot(-normalInWorld, viewDirInWorld));
output.visibility *= output.visibility; // instead of power of 2
```

可变参数 `output.visibility` 在粒子多边形上插值。在片段着色器中读取此值，再应用一定数量的透明度。下列代码演示了它的实现方式：

```
half4 diffuseTex = _Color * tex2D(_MainTex, half2(input.texCoord));
diffuseTex *= input.visibility;
return diffuseTex;
```

### 渲染粒子

要渲染粒子，可使用下列步骤：

1. 将粒子渲染为帧内最后的原语。

```
Tags { "Queue" = "Transparent+10" }
```

本例中出现了 +10，因为冰穴演示中在粒子前面渲染了 9 个其他透明对象。

2. 设置适当的混合模式。

在着色器通道的开头，添加下面这一行：

```
Blend SrcAlpha One
```

3. 禁用写入到 z 缓冲区。

添加下面这一行：

```
ZWrite Off
```

### 粒子系统设置

下图显示了冰穴演示中用于粒子系统的设置：

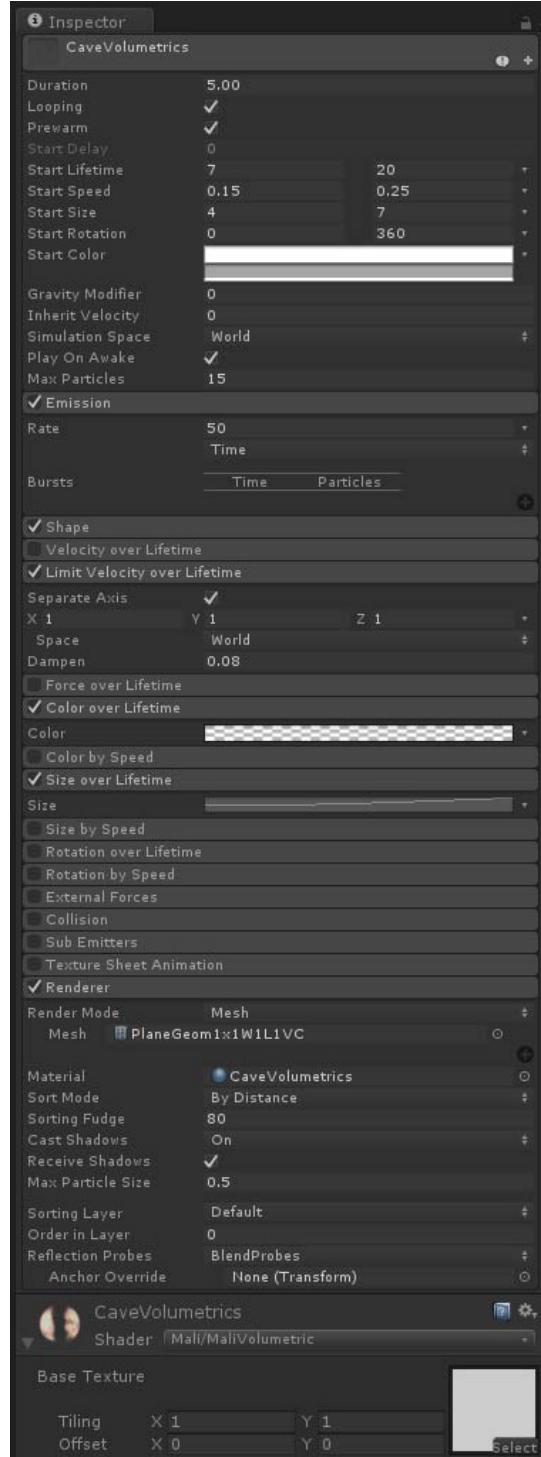


图 6-60 来自冰穴演示的粒子系统参数

下图显示了生成粒子的区域。这通过图像中的方框表示。该方框通过 Unity 粒子系统中内置的**形状**选项定义：



图 6-61 生成粒子的粒子方框定义

## 6.11 高光溢出

高光溢出用于再现真实摄像机在明亮环境中拍摄图片时出现的效果。高光溢出模拟从明亮区域边界延伸的光线条纹，创造出明亮光线压倒摄像机的假象。

下图显示了冰穴演示中洞穴入口处的高光溢出：

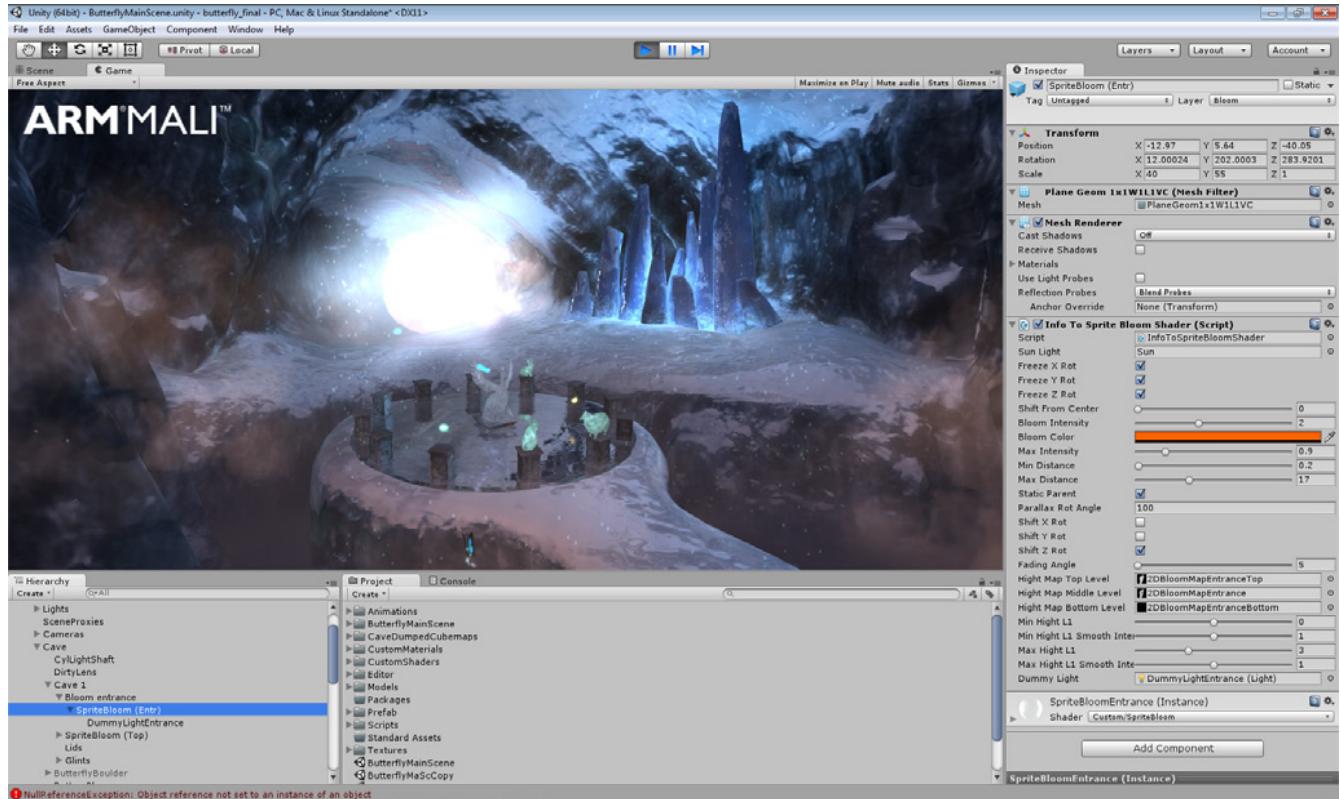


图 6-62 冰穴演示中所实施的洞穴入口处高光溢出

高光溢出效果发生于真实镜头，因为它们无法完美对焦。当光线穿过摄像机的光圈时，一些光线出现衍射，在图像周围创造出一个明亮的圆盘。此效果在大部分情形中通常不明显，但在强光照明下可见。

本节包含以下小节：

- [6.11.1 实施高光溢出（第 6-152 页）。](#)
- [6.11.2 创建高光溢出调节因子脚本（第 6-153 页）。](#)
- [6.11.3 顶点着色器（第 6-153 页）。](#)
- [6.11.4 片段着色器（第 6-154 页）。](#)
- [6.11.5 修正高光溢出效果（第 6-154 页）。](#)
- [6.11.6 遮挡贴图间插值（第 6-157 页）。](#)

### 6.11.1 实施高光溢出

高光溢出效果通常作为后处理效果实施。使用这种方式生成效果可能会占用大量的计算功率，因此不适合运用到移动游戏。如果您的游戏使用大量类似于冰穴演示中的复杂效果，这一点显得尤其突出。

一种替代方法是使用简单平面。将这一平面放在摄像机和光源之间。该平面的朝向必须是其法线指向摄像机应在其中移动的场景部分。

要以这样的方式创建高光溢出效果，可将一个平面放置在您要产生高光溢出效果的位置上。利用一个因子调节效果的强度，该因子基于视角向量与平面法线和光源的对齐。冰穴演示中实施了这种方法。

下图显示了利用平面实施的高光溢出效果：



图 6-63 平面位于实施高光溢出效果的洞口

### 6.11.2 创建高光溢出调节因子脚本

对齐因子可以使用脚本进行计算。此因子基于摄像机查看该效果的角度改变高光溢出效果。

例如，下列脚本计算了平面所附着的对齐因子：

```
// Light-plane
normal-camera alignmentVector3 planeToCamVec = Camera.main.transform.position
    - gameObject.transform.position;
planeToCamVec.Normalize();
Vector3 sunLightToPlainVec = origPlainPos - sunLight.transform.position;
sunLightToPlainVec.Normalize();
float sunLightPlainCameraAlignment = Vector3.Dot(plainToCamVec, sunLightToPlainVec);
sunLightPlainCameraAlignment = Mathf.Clamp(sunLightPlainCameraAlignment, 0.0f, 1.0f);
```

对齐因子 sunLightPlaneCameraAlignment 传递到着色器，以调节所渲染的颜色的强度。

### 6.11.3 顶点着色器

顶点着色器接收 sunLightPlainCameraAlignment 因子，并使用它调节所渲染的高光溢出颜色的强度。

顶点着色器应用 MVP 矩阵，将纹理坐标传递到片段着色器，再输出顶点坐标。

下列代码演示了它的实现方式：

```
vertexOutput vert(vertexInput input)
{
    vertexOutput output;
    output.tex = input.texcoord;
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}
```

#### 6.11.4 片段着色器

片段着色器获取纹理颜色并使用 CurrBloomColor 浅色进行递增，后者作为统一变量进行传递。对齐因子先调节颜色，然后再使用。

下列代码演示了此流程的执行方式：

```
half4 frag(vertexOutput input) : COLOR
{
    half4 textureColor = tex2D(_MainTex, input.tex.xy);
    textureColor += textureColor * _CurrBloomColor;
    return textureColor * _AlignmentFactor;
}
```

高光溢出平面继所有不透明几何体之后，在透明队列中渲染。在着色器中，使用下列队列标记设置渲染顺序：

```
Tags { "Queue" = "Transparent" + 1}
```

高光溢出平面利用附加的一对一混合来应用，将片段颜色和已存储在帧缓冲区中的对应像素相组合。该着色器也使用指令 ZWrite Off 禁用对深度缓冲区的写入，使得现有的对象不会被遮挡。

下图显示了冰穴演示用于其高光溢出效果的纹理，以及它所创造的结果：



图 6-64 显示摄像机进入不透明底座后遮挡区域的序列

#### 6.11.5 修正高光溢出效果

使用平面产生高光溢出效果非常简单，也适合移动设备。然而，当产生高光溢出的光源和摄像机之间存在不透明对象时，它就无法产生预期的结果。您可以使用遮挡贴图进行修正。

不透明对象后方的高光溢出效果外观出错的原因是该效果是在透明队列中渲染的。因此，混合发生于高光溢出平面前面的任何对象上。

下图中的示例显示产生的不正确高光溢出效果尚未得到修正：



图 6-65 不透明底座上显示的不正确高光溢出效果

为防止出现这些错误，您可以修改计算对齐因子的脚本，使它处理一个额外的遮挡贴图。这一遮挡贴图描述了摄像机通常不当应用高光溢出效果的区域。这些冲突区域分配到的遮挡贴图值为零。此因子与对齐因子结合，将这些位置上该因子的强度设为零。此更改将高光溢出设为零，使它不再渲染必须遮挡它的对象。下列代码实施了这一调整：

```
IntensityFactor = alignmentFactor * occlusionMapFactor
```

由于冰穴演示摄像机可以在三个维度上自由移动，因此使用了三个灰度贴图，各自覆盖不同的高度。黑色表示零，所以遮挡贴图因子乘以对齐因子使得最终强度为零，高光溢出被遮挡。白色表示一，所以强度等于对齐因子，高光溢出不被遮挡。

下图显示了地平面的 occlusionMapFactor:

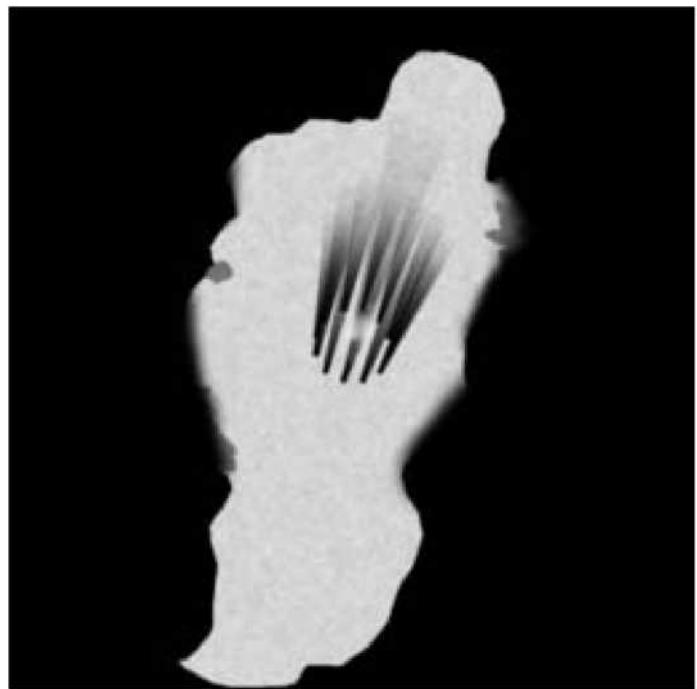


图 6-66 地平面高光溢出遮挡贴图

————备注————

贴图中间的黑色辐射区域是不透明底座背后的区域，它们遮挡了来自洞口的高光溢出效果。

地平面上方  $H_{max}$  高度上没有遮挡对象。这表示地平面上方的高光溢出遮挡贴图为白色。下图显示了地平面上方的 occlusionMapFactor:

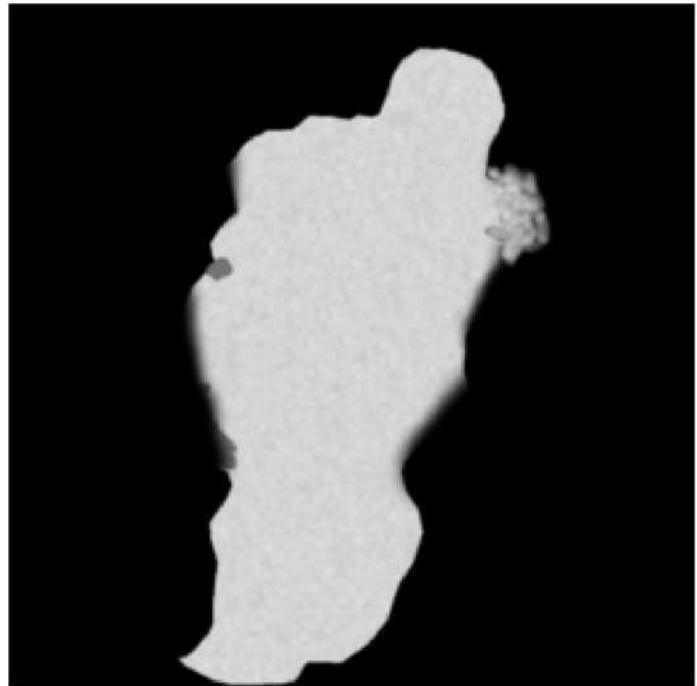


图 6-67 地平面上方的高光溢出遮挡贴图

低于高度  $H_{min}$  时，即地平面下方，高光溢出效果被完全遮挡。这表示地平面下方的高光溢出遮挡贴图为全黑。添加了白色轮廓线，使得该贴图可见。下图显示了地平面下方的 `occlusionMapFactor`：



图 6-68 地平面下方的高光溢出遮挡贴图

#### 6.11.6 遮挡贴图间插值

在冰穴演示中，脚本计算每一帧中摄像机位置在 2D 洞穴贴图上的 XZ 投影，并将结果正规化到零和一之间。如果摄像机高度低于  $H_{min}$  或高于  $H_{max}$ ，则使用正规化后的坐标从单一贴图获取颜色值。如果摄像机高度在  $H_{min}$  和  $H_{max}$  之间，则从两个贴图获取颜色并进行插值。

此插值可以为不同高度上的效果创造平滑的过渡。

冰穴演示使用 `GetPixelBilinear()` 函数从贴图获取颜色。此函数利用下列代码返回滤波颜色值：

```
float groundOcclusionFactor = groundOcclusionMap.GetPixelBilinear(camPosXZNormalized.x,  
camPosXZNormalized.y)).r;
```

使用高光溢出遮挡贴图可以防止在遮挡的不透明对象上混合高光溢出。下图显示了产生的效果。当摄像机进入和离开时，遮挡的黑色在不透明柱子的后方，从而防止出现不正确的高光溢出。

下图显示摄像机进入不透明底座后遮挡区域的序列：



图 6-69 进入不透明底座后的遮挡区域

## 6.12 冰墙效果

在冰穴演示中，洞穴的冰墙上使用了细微的反射效果。此效果很细微，但为场景增添额外的真实感和气氛。

本节包含以下小节：

- [6.12.1 关于冰墙效果（第 6-159 页）。](#)
- [6.12.2 修改和组合法线贴图以影响反射（第 6-160 页）。](#)
- [6.12.3 从不同的法线贴图创建反射（第 6-162 页）。](#)
- [6.12.4 向反射应用局部修正（第 6-164 页）。](#)

### 6.12.1 关于冰墙效果

冰是一种难以复制的材质，因为光线会以不同的方式从它散射开来，具体取决于其表面的细微细节。反射可以是完全清晰的、完全失真的，或介于两者之间。冰穴演示显示了此效果，并包含一个带来更高真实感的视差效果。

下图显示了展示此效果的冰穴演示：



图 6-70 冰穴演示

下图显示了此效果的特写：

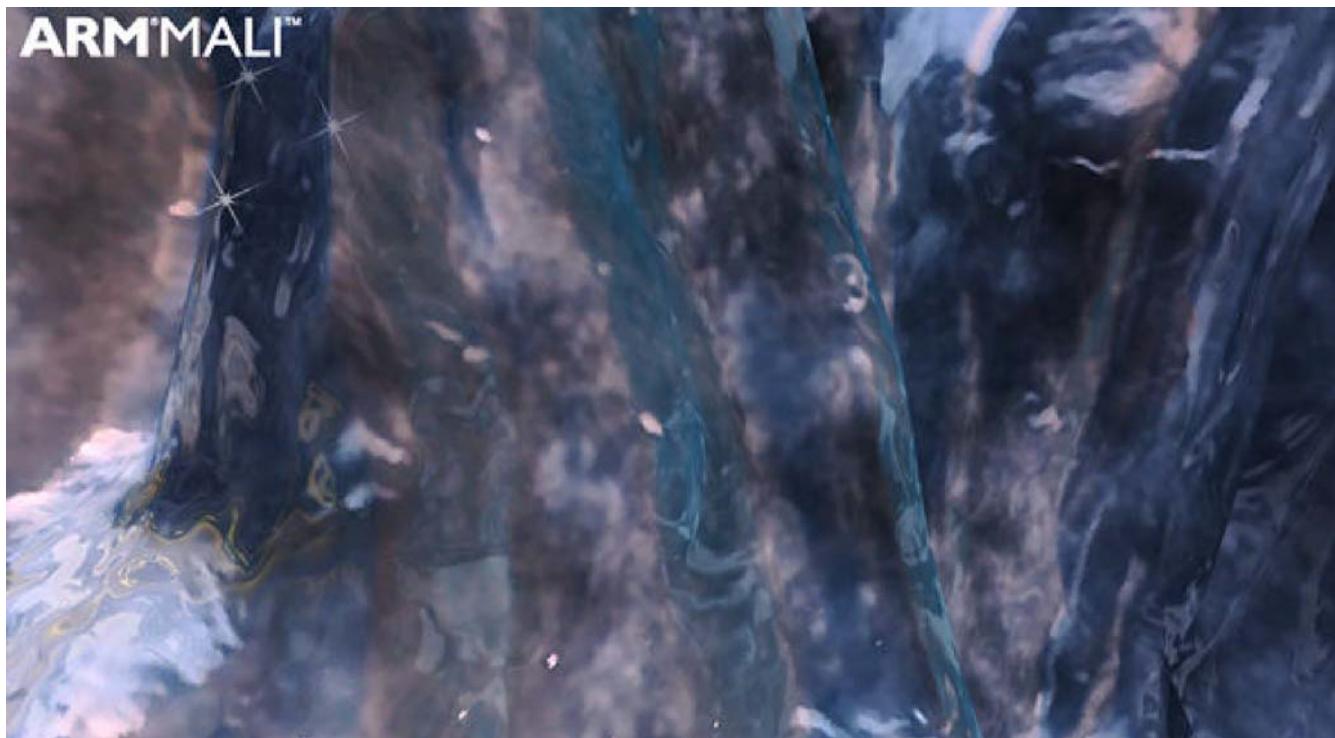


图 6-71 反射冰墙的特写

### 6.12.2 修改和组合法线贴图以影响反射

冰穴演示中的反射效果使用了正切空间法线贴图和计算而来的灰度虚构法线贴图。将这两种贴图与一些修改器组合，创造了演示中的效果。

灰度虚构法线贴图是正切空间法线贴图的灰度版本。在冰穴演示中，灰度贴图中的大部分值处于 0.3-0.8 范围内。下图显示了冰穴演示所使用的正切空间法线贴图和灰度虚构法线贴图：

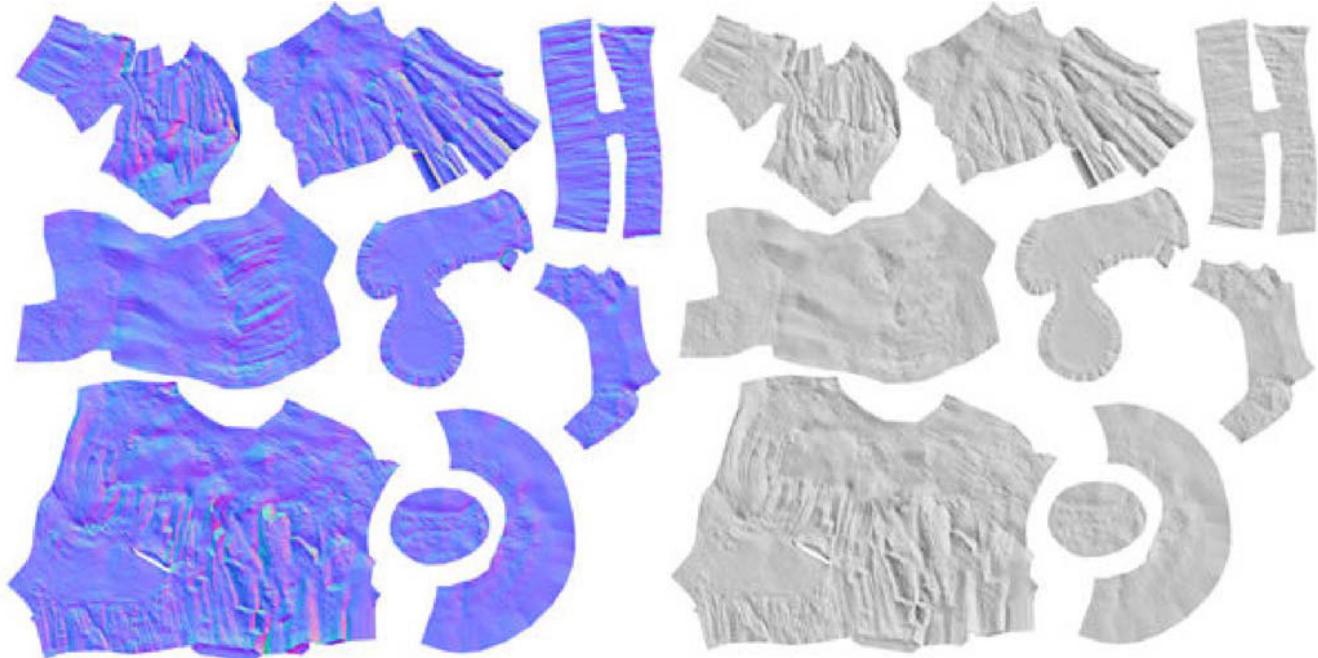


图 6-72 正切空间法线贴图和灰度虚构法线贴图

### 为何使用正切空间法线贴图

冰穴演示中使用了正切空间法线贴图，因为它们具有生成的灰度中的一小范围的值。这意味着，正切空间法线贴图在此渲染流程的后续阶段中发挥效果。

另一个选择是使用对象空间法线贴图。这些贴图显示与正切空间法线贴图相同的细节，但也同时显示光线照射到的位置。因此，生成的对象空间法线贴图灰度中的值范围太大，无法在渲染流程的后续阶段中发挥效果。下图显示了此效果：

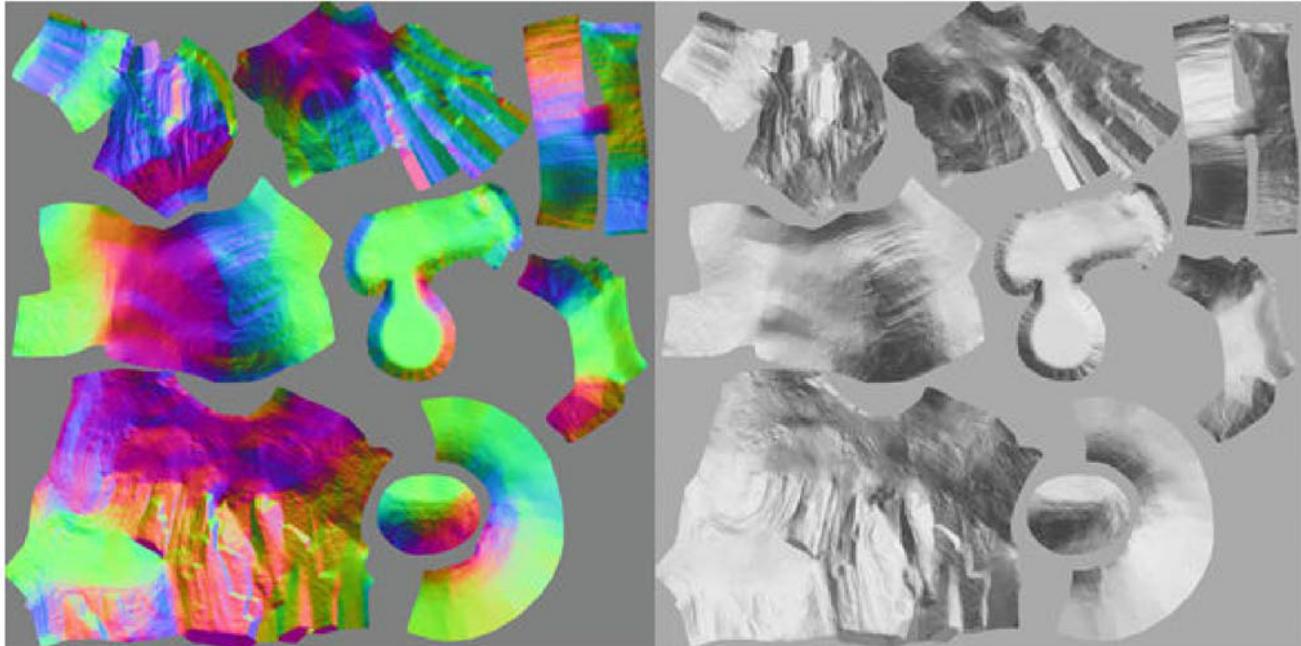


图 6-73 对象空间法线贴图和灰度效果

### 向法线贴图的部件应用透明

灰度虚构法线贴图仅应用到没有雪的区域。为防止它们被应用到有雪的区域，可通过对相同的表面使用漫射纹理贴图来修改灰度虚构法线贴图。此修改可在雪出现于纹理贴图中的位置上增大灰度虚构法线贴图的 **alpha** 分量。

下图显示了用于冰穴演示静态表面的漫射纹理贴图，以及它们应用到灰度虚构法线贴图时的结果：

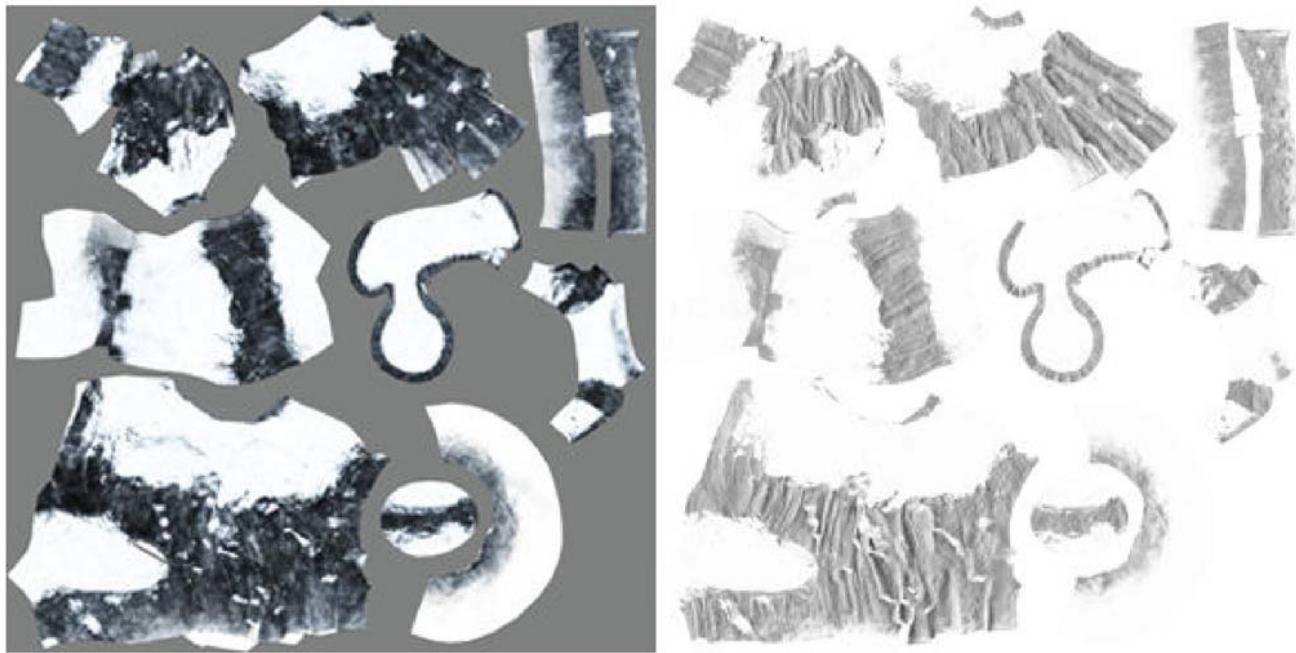


图 6-74 漫射纹理和带有透明区域的最终虚构法线贴图

### 6.12.3 从不同的法线贴图创建反射

将调整了透明度的灰度虚构法线贴图和真正法线贴图相组合，可创造出冰穴演示中展示的反射效果。

调整了透明度的灰度虚构法线贴图 `bumpFake` 和真正法线贴图 `bumpNorm` 按比例组合。该组合使用了以下函数：

```
half4 bumpNormalFake = lerp(bumpNorm, bumpFake, amountFakeNormalMap);
```

此代码意味着，在洞穴的昏暗部分中，主要的反射组成来自于灰度虚构法线。在洞穴的有雪区域，该效果来自于对象空间法线。

要应用使用灰度虚构法线贴图的效果，灰度必须转换为法线向量。要开始此流程，需要将法线向量的三个分量设置为和灰度值相等。在冰穴演示中，这表示分量向量介于  $(0.3, 0.3, 0.3)$  到  $(0.8, 0.8, 0.8)$  范围内。因为所有分量都设置为相同的值，所有法线向量指向同一个方向。

着色器向法线分量应用一个变换。它使用的变换通常用于将  $0\text{-}1$  范围中的值变换为  $-1$  到  $1$  范围中的值。执行此更改的等式为，结果  $= 2 * \text{值} - 1$ 。此等式更改了法线向量，使得它们或者指向与之前相同的方向，或者指向相反的方向。例如，如果原先具有分量  $(0.3, 0.3, 0.3)$ ，则生成的法线为  $(-0.4, -0.4, -0.4)$ 。如果原先具有分量  $(0.8, 0.8, 0.8)$ ，则生成的法线为  $(0.6, 0.6, 0.6)$ 。

变换到  $-1$  到  $1$  范围后，向量馈入到 `reflect()` 函数。此函数设计为使用正规化法线向量工作，但在这一情形中，非正规化法线被传递到该函数。下列代码演示了着色器内置函数 `reflect()` 的工作方式：

```
R = reflect(I, N) = I - 2 * dot(I, N) * N
```

根据反射定律，将此函数与长度小于一的非正规化输入法线搭配使用时，会导致反射向量偏离法线的程度超过预期。

当法线向量分量的值低于  $0.5$  时，反射向量将切换到相反的方向。此时，将读取立方体贴图中的另一个部分。这种在立方体贴图中不同部分间的切换，创造了在立方体贴图岩石部分反射的旁边反射立方体贴图中白色部分的不均匀点效果。由于灰度虚构法线贴图中导致在正向和负向法线之间切换的区域也是产生被反射向量最为失真的角度的区域，这创造了一种视觉引人的漩涡效果。下图显示了此效果：



图 6-75 冰雪反射中的漩涡效果

如果着色器被编程为无非正规化固定阶段的虚构法线值，其结果则将具有对角条纹。这是显著的差别，因此这些阶段非常重要。下图显示了没有固定阶段时生成的结果：

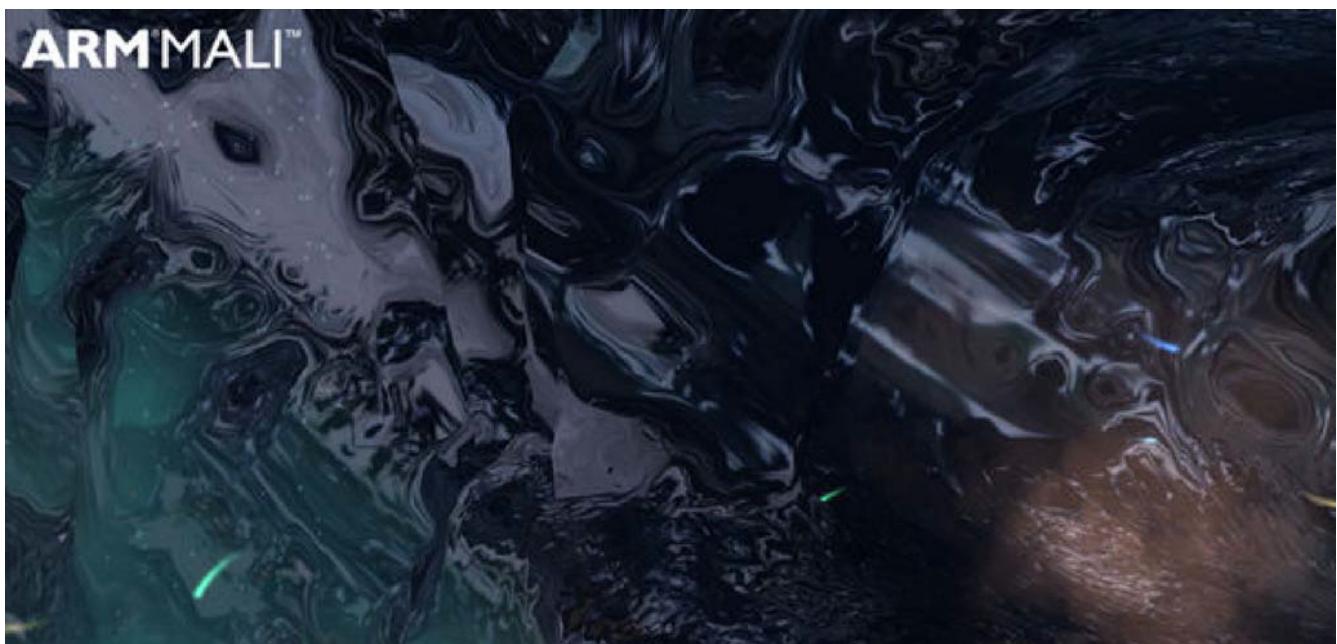


图 6-76 无固定阶段的冰雪反射

#### 6.12.4 向反射应用局部修正

向反射向量应用局部修正可提高反射效果的真实感。如果没有此修正，反射就不会像现实中一样随着摄像机位置的变化而改变。对于摄像机侧向移动而言，这一点显得尤其突出。

##### 相关信息

[6.2.2 使用局部立方体贴图生成正确的反射（第 6-100 页）。](#)

## 6.13 过程天空盒

冰穴演示中使用一个时间系统来展示您可以通过局部立方体贴图实现的动态阴影效果。

本节包含以下小节：

- [6.13.1 关于过程天空盒（第 6-165 页）。](#)
- [6.13.2 管理一天中的时间（第 6-166 页）。](#)
- [6.13.3 渲染太阳（第 6-167 页）。](#)
- [6.13.4 淡化群山后的太阳（第 6-168 页）。](#)
- [6.13.5 子表面散射（第 6-170 页）。](#)

### 6.13.1 关于过程天空盒

要获得动态时间效果，需要组合下列元素：

- 过程性生成的太阳。
- 代表昼夜更替的一系列淡化天空盒背景立方体贴图。
- 天空盒云朵立方体贴图。

过程太阳和单独的云朵纹理也可创建计算成本低廉的子表面散射效果。

下图显示了天空盒的一个视图：



图 6-77 冰穴演示中具有子表面散射的太阳渲染

冰穴演示使用了视角方向从立方体贴图采样。这使得该演示可以避免渲染天空盒的一个半球。相反，它在洞穴的孔洞附近使用平面。

相较于渲染之后大体被其他几何体遮挡的半球，这样做可以提升性能。

下图显示了利用平面渲染的天空盒。

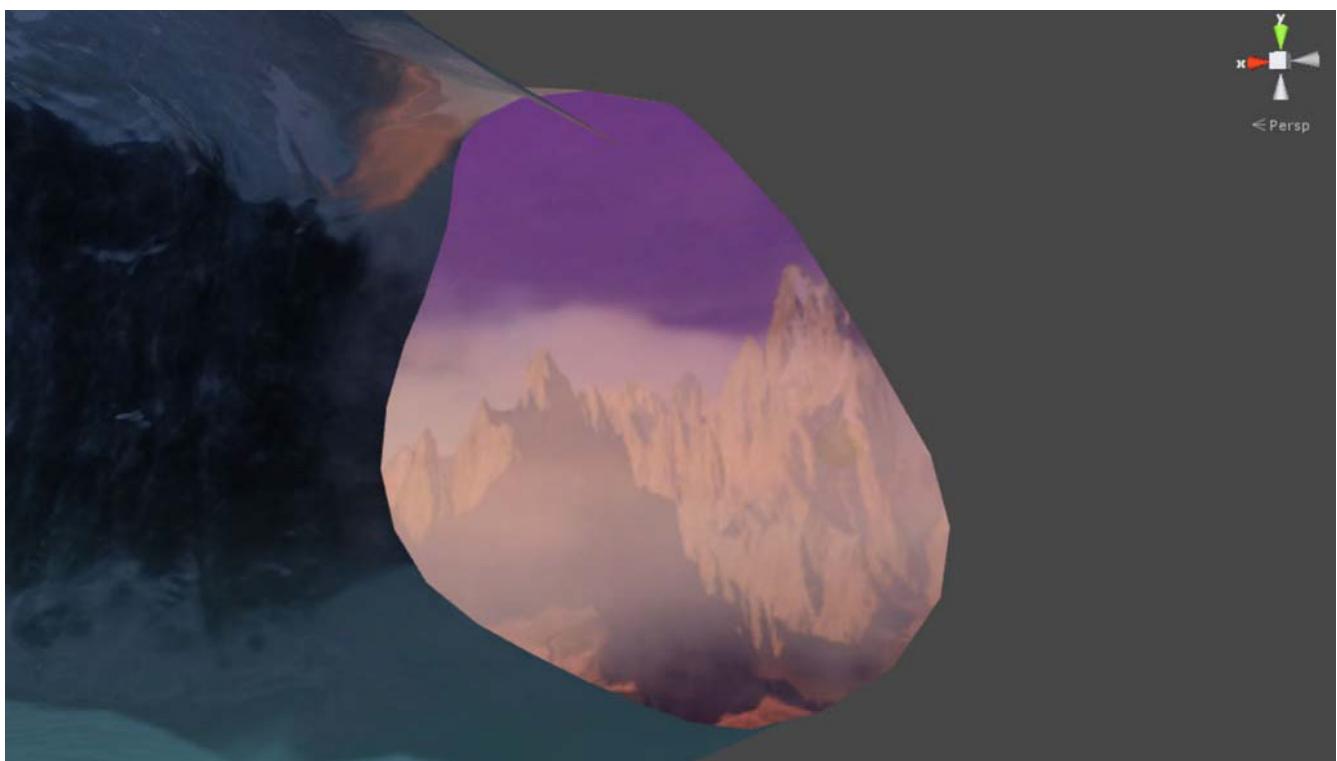


图 6-78 利用平面渲染的天空盒

### 6.13.2 管理一天中的时间

此效果使用 C# 脚本管理一天中的时间的数学计算，以及昼夜更替动画。一个着色器而后组合太阳和天空贴图。

您必须为该脚本指定下列值：

- 淡化天空盒背景相位的数量。
- 昼夜更替的最长时长。

对于每一个帧，该脚本选择天空盒立方体贴图并将它们混合在一起。选定的天空盒被设置为着色器的纹理，它在渲染时将这些纹理混合在一起。

为设置纹理，冰穴演示使用了 Unity 着色器全局功能。这样，您可以在一个位置设置纹理，而后供应用程序的所有着色器使用。下列代码对此进行了演示：

```
Shader.SetGlobalTexture (ShaderCubemap1, _phasesCubemaps [idx1]);
Shader.SetGlobalTexture (ShaderCubemap2, _phasesCubemaps [idx2]);
Shader.SetFloat (ShaderAlpha, blendAlpha);
Shader.SetGlobalVector (ShaderSunPosition, normalizedSunPosition);
Shader.SetGlobalVector (ShaderSunParameters, _sunParameters);
Shader.SetGlobalVector (ShaderSunColor, _sunColor);
Shader.SetGlobalTexture (ShaderCloudsCubemap, _CloudsCubemap);
```

#### 备注

使用的采样器名称不得与着色器在本地定义的名称冲突。

脚本代码中进行了如下设置：

- 插值了两个立方体贴图。值 idx1 和 idx2 根据消逝的时间计算。
- blendAlpha 因子，在着色器中用于混合两个立方体贴图。
- 正规化太阳位置，用于渲染太阳球体。

- 太阳的多个参数。
- 太阳的颜色。
- 云朵立方体贴图。ShaderCubemap1 和 ShaderCubemap2 是包含唯一取样器名称的两个字符串，本例中为 \_SkyboxCubemap1 和 \_SkyboxCubemap2。

为了在着色器中访问这些纹理，您必须使用下列代码声明它们：

```
samplerCUBE _SkyboxCubemap1;  
samplerCUBE _SkyboxCubemap2;
```

该脚本根据您指定的列表选择太阳颜色和环境颜色。它们针对每一个相位进行插值。

太阳颜色传递到着色器，从而使用正确的颜色渲染太阳。

环境颜色用于动态设置 Unity 变量 RenderSettings.ambientSkyColor：

```
RenderSettings.ambientSkyColor = _ambientColor;
```

设置此变量可使所有材质获得正确的环境颜色，同时也能让 Enlighten 在更新光照贴图时获得正确的环境颜色。

在冰穴演示中，此效果导致场景根据一天中所处的时段出现总体颜色的渐进变化。

### 6.13.3 渲染太阳

要渲染太阳，您必须在片段着色器中针对天空盒的每一像素检查它是否处于太阳圆周的内部。

为此，着色器必须计算所渲染像素的世界坐标中正规化太阳位置向量和正规化视角方向向量的点积。

下图显示了太阳的渲染。正规化视角向量和太阳位置的点积用于确定片段是渲染为天空 (P2) 还是渲染为太阳 (P1)。

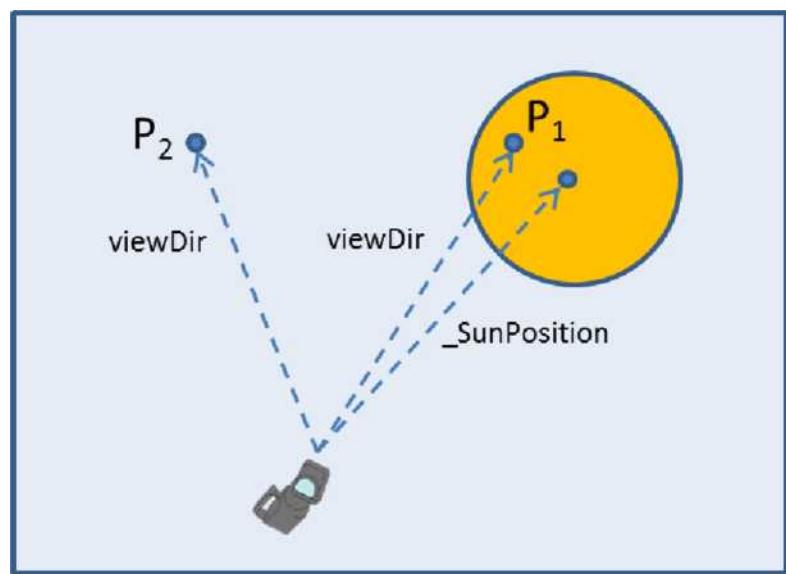


图 6-79 渲染太阳

通过 C# 脚本，将正规化太阳位置向量传递到着色器。

- 如果值大于指定的阈值，像素着色为太阳的颜色。
- 如果值小于指定的阈值，像素着色为天空的颜色。

点积也可朝着太阳边缘创造出淡化效果：

```
half _sunContribution = dot(viewDir,_SunPosition);
```

下图显示了晴空中的太阳视图:



图 6-80 晴朗天空条件下过程太阳渲染

#### 6.13.4 淡化群山后的太阳

如果太阳在天空中的高度较低，会存在一个问题。在现实中，它会在群山背后消失。

为创造这一效果，立方体贴图的 **alpha** 通道用于存储值 0（如果像素元代表天空）和值 1（如果像素元代表大山）。

在渲染太阳时，纹理被采样，其 **alpha** 用于使太阳在群山背后淡化。此采样过程几乎自由，因为纹理已被采样用于渲染群山。

您也可以渐进淡化边缘附近或山上有雪覆盖区域的 **alpha**。这可以产生阳光从白雪反弹的效果，而且几乎不需要计算工作。

类似的技巧也可用于为云朵创建计算代价低廉的子表面散射效果。

原始的相位立方体贴图分入两个独立小组。

- 一组立方体贴图包含天空和群山。其 **alpha** 值设为 0（天空）和 1（群山）。
- 另一组立方体贴图包含云朵。其 **alpha** 值在无云区域中设为 0，然后随着云朵变密而逐渐增大到 1。

下图显示了群山纹理。群山的 **alpha** 值为一，而天空的值为零：

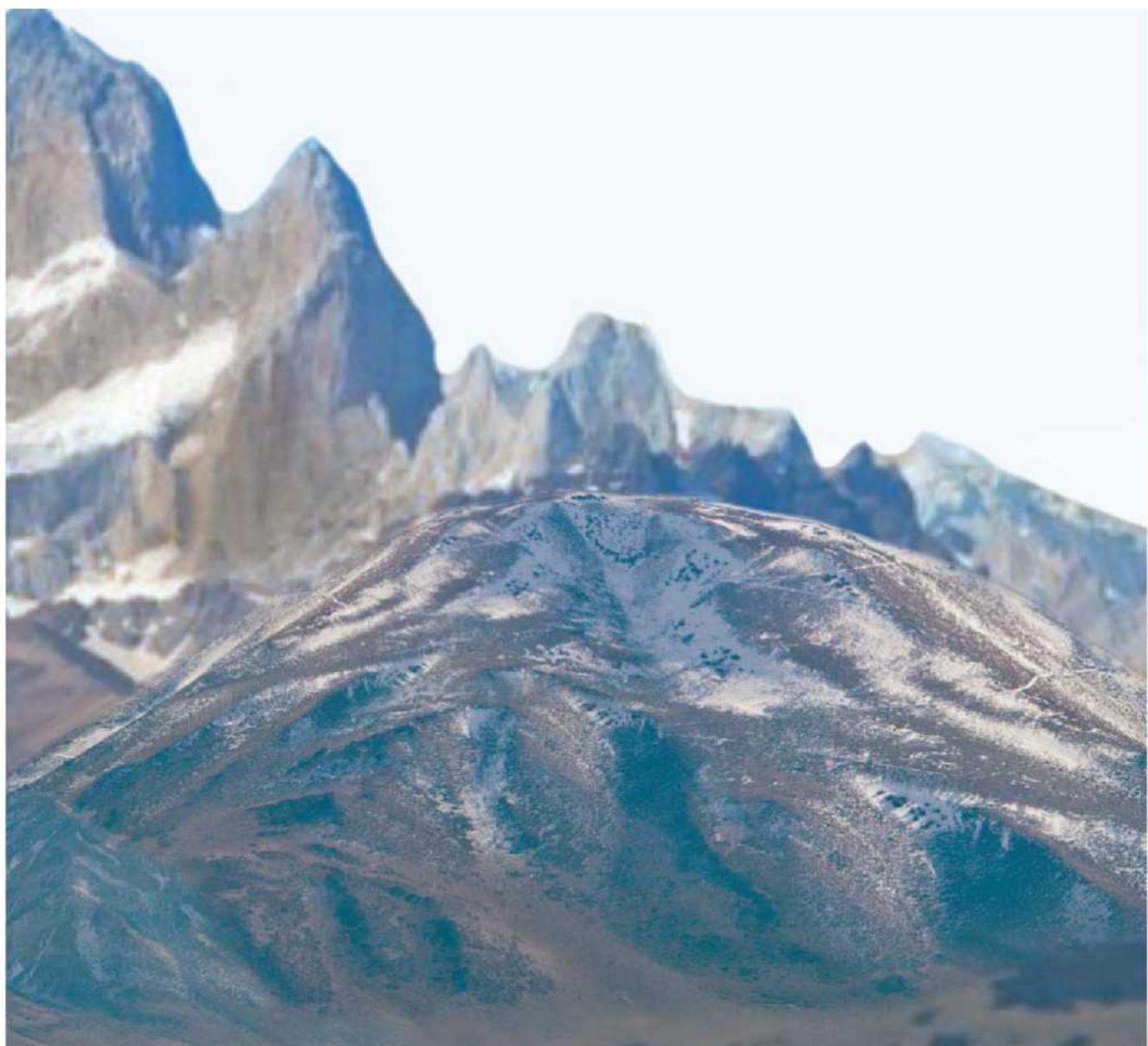


图 6-81 群山纹理

在云朵天空盒中，空白区域 **alpha** 值为 0，有云朵覆盖区域的值渐变为 1。其 **alpha** 通道平滑淡化，确保云朵的外观不像是人为置于天空中那样。下图显示了具有 **alpha** 的云朵纹理：

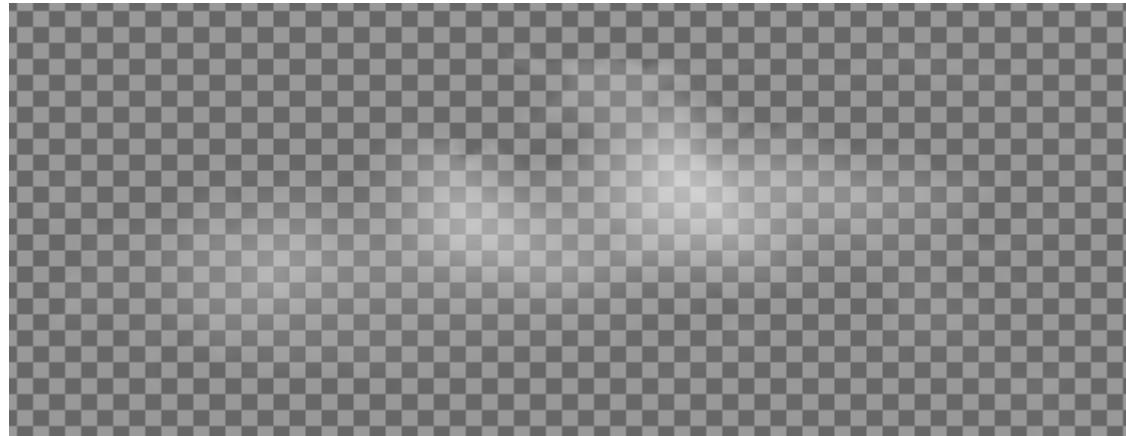


图 6-82 云朵纹理

着色器执行如下工作：

1. 对代表一天中当前时段的两个天空盒进行采样。
2. 根据由 C# 脚本计算的混合因子混合两种颜色。
3. 对云朵天空盒进行采样。
4. 使用云朵 alpha 将点 2 的颜色与云朵的颜色混合。
5. 将云朵 alpha 和天空盒 alpha 加到一起。
6. 调用一个函数，计算太阳颜色对当前像素的作用。
7. 将点 6 的结果和前面混合的天空盒与云朵颜色加到一起。

#### 备注

您可以将天空、群山和云朵放入一个天空盒内，优化这一序列。它们在冰穴演示中是分开的，以便美术师可以轻松地分别修改天空盒和云朵。

### 6.13.5 子表面散射

您可以添加子表面散射效果，此效果使用 alpha 信息来增加太阳的半径。

在现实中，太阳位于晴朗天空中时，其大小会显得相对较小，因为没有云层能够偏移或散射照向您的眼睛的光线。您看到的是直接从太阳射来的光线，仅受到空气的些许漫射。

当太阳被云朵遮挡时，但又没有完全遮蔽时，部分光线会在云朵周围散射。此光线可从与太阳稍有距离的方向进入您的眼睛。这可以使太阳显得比实际大。

冰穴演示中使用下列代码实现这一效果：

```
half4 sampleSun(half3 viewDir, half alpha);
{
    half _sunContribution = dot(viewDir, _SunPosition);
    half _sunDistanceFade = smoothstep(_SunParameters.z -(0.025*alpha), 1.0, _sunContribution);
    half _sunOcclusionFade = clamp( 0.9-alpha, 0.0, 1.0);

    half3 _sunColorResult = _sunDistanceFade * _SunColor * _sunOcclusionFade;
    return half4( _sunColorResult.xyz, 1.0 );
}
```

函数的参数为 viewDirection 以及计算的 alpha，即云朵 alpha 和天空盒 alpha 加到一起后。

太阳位置和视角方向的点积用于计算一个缩放因子，它用于表示当前像素与太阳中心的距离。

`_sunDistanceFade` 计算使用 `smoothstep()` 函数提供边缘附近从太阳中心到天空的更加平缓的渐变。这可以使得太阳的半径在靠近云朵时加大，从而模拟子表面散射效果。

此函数具有一个基于 `alpha` 的变量域，晴朗天空中时其 `alpha` 为 0，范围则在 `_SunParameters.z` 到 1.0 内。在此情形中，`_SunParameters.z` 在 C# 脚本中初始化为 0.995，它对应于直径为 5 度的太阳， $\cos(5 \text{ degrees}) = 0.995$ 。

如果被处理的像素包含云朵，则太阳的半径提高到 13 度，实现在靠近云朵时的加长散射效果。

`_sunOcclusionFade` 因子用于根据从群山和云朵获得的遮挡，将太阳的作用值逐渐变小。

下图显示了未被云朵遮挡的太阳：



图 6-83 太阳未被云朵遮挡

下图显示了被云朵遮挡的太阳：



图 6-84 太阳被云朵遮挡

## 6.14 萤火虫

萤火虫是一种发光的飞虫，在冰穴演示中用来增加动态感，并且演示将 **Enlighten** 用于实时全局照明的优点。

本节包含以下小节：

- [6.14.1 关于萤火虫（第 6-173 页）。](#)
- [6.14.2 萤火虫生成器预制件（第 6-175 页）。](#)

### 6.14.1 关于萤火虫

萤火虫由下列组件组成：

- 在运行时实例化的预制对象。
- 用于限制萤火虫飞行区域的盒碰撞器。

这两个组件通过 C# 脚本组合在一起，该脚本管理萤火虫的运动并且定义它们遵循的路径。

下图显示了萤火虫：



图 6-85 萤火虫

萤火虫预制件使用 Unity 标准粒子系统生成萤火虫的轨迹。

下图显示了供萤火虫使用的 Unity 粒子系统设置：

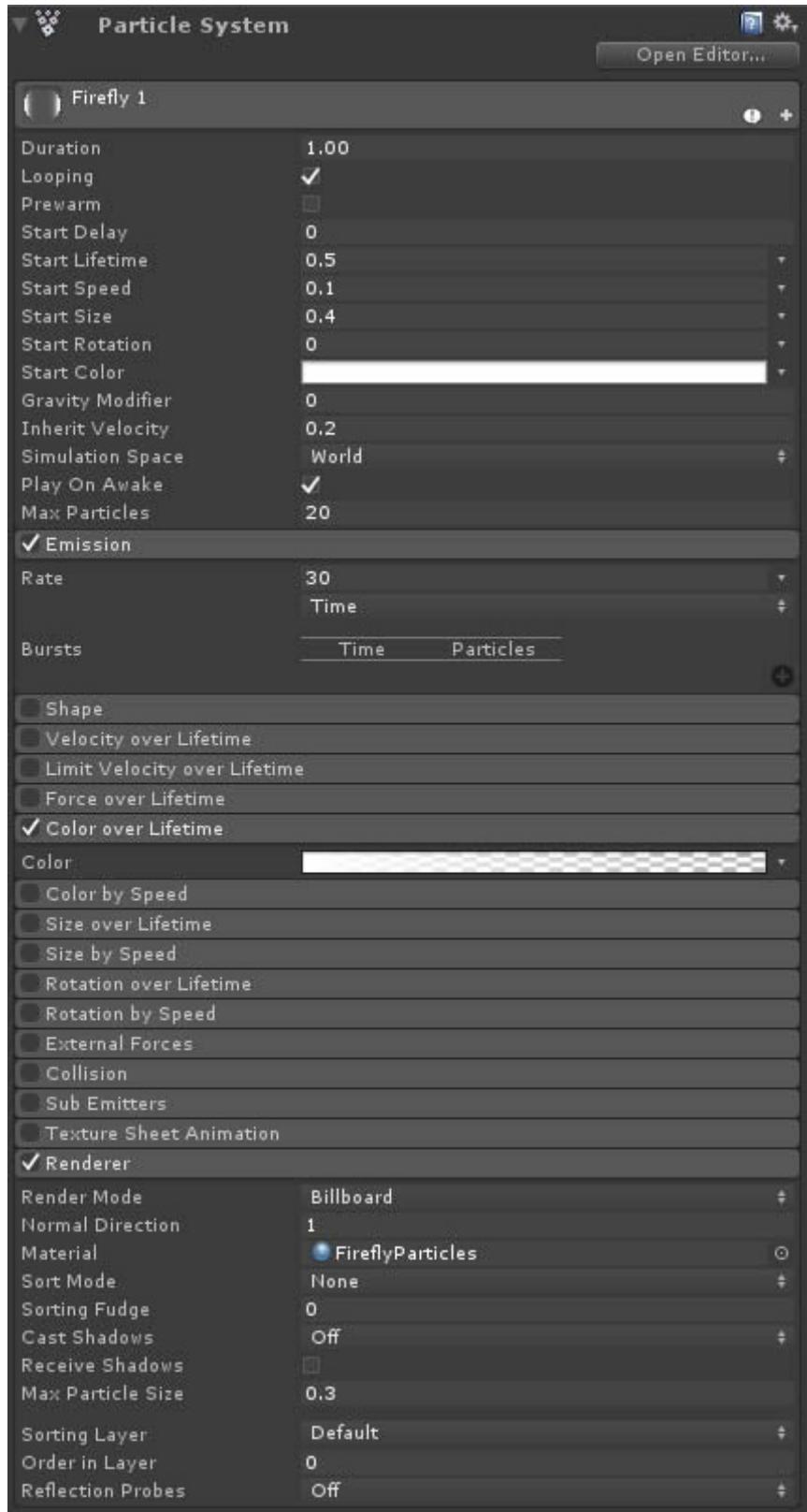


图 6-86 Unity 粒子系统设置

根据您要获得的效果，您可以使用 Unity Trail Renderer 来提供更为连贯的外观。

Trail Renderer 为每一道轨迹生成大量的三角形。您可以通过修改 Trail Renderer 的**最小顶点距离**设置来更改三角形的数量，但较大的值可能会在来源移动速度太快时造成轨迹运动不连贯。

**最小顶点距离**选项定义形成轨迹的顶点之间的最小距离。较高的数字对直线轨迹而言不错，但对曲线轨迹则会出现外观不平滑。

生成的轨迹始终朝向摄像机；因此，来源运动的任何断续可能会造成轨迹与自身重叠。这将导致因混合形成轨迹的三角形而产生失真。

下图显示了由重叠轨迹导致的失真：



图 6-87 重叠轨迹失真

添加到预制件中的最终分量是点光源，它一边移动一边在场景中投射光线。

此光源影响提供光线反弹效果的 Enlighten 全局照明，尤其是在场景的狭窄部分中，这些位置上由于光线分散较少而使得光线反弹可见性更高。

#### 6.14.2 萤火虫生成器预制件

萤火虫生成器预制件管理萤火虫的创建，并在每一帧上更新它们。它包含用于更新的 C# 脚本，以及封闭每个萤火虫可以在其中运动的体积的盒碰撞器。

该脚本将要生成的萤火虫数量取为参数，并利用预制件初始化萤火虫对象。由于萤火虫在包围盒内随机运动，它将变化限制在离开萤火虫运动方向的一个特定范围内。这可确保萤火虫不会突然改变方向。

为生成随机运动，使用一个分段三次埃尔米特插值来创建控制点。埃尔米特插值提供了一个平滑、连贯的功能，即使不同的路径连接在一起也能正确运行。端点的第一次衍生也是连贯的，所以没有突然的速度变化。

这种插值需要起点和终点各一个控制点，以及每个控制点两条切线。由于它们是随机生成的，该脚本可以存储三个控制点和两条切线。它使用第一和第二控制点的位置来定义第一点切线，使用第二和第三控制点来定义第二控制点切线。

在加载时，该脚本为每一个萤火虫生成以下几项：

- 初始位置。
- 初始方向，利用 Unity 函数 Random.onUnitSphere() 生成。

下列代码演示了如何初始化控制点：

```
_fireflySpline[i*_controlPoints] = initialPosition;
Vector3 randDirection = Random.onUnitSphere;
_fireflySpline[i*_controlPoints+1] = initialPosition + randDirection;
_fireflySpline[i*_controlPoints+2] = initialPosition + randDirection * 2.0f;
```

初始控制点在一条直线上。切线从这些的控制点生成：

```
//The tangent for the first point is in the same direction as the initial direction vector
_fireflyTangents[i*_controlPoints] = randDirection;

//This code computes the tangent from the control point positions. It is shown here for
// reference because it can be set to randDirection at initialization.
_fireflyTangents[i*_controlPoints+1] = (_fireflySpline[i*_controlPoints+2] -
_fireflySpline[i*_controlPoints+1])/2 + (_fireflySpline[i*_controlPoints+1] -
_fireflySpline[i*_controlPoints])/2;
```

为完成萤火虫初始化，您必须设置萤火虫的颜色，以及当前路径间隔的长度。

在每一帧上，该脚本使用下列代码计算埃尔米特插值，更新每个萤火虫的位置：

```
// t is the parameter that defines where in the curve the firefly is placed. It represents
// the ratio of the time the firefly has traveled along the path to the total time.
float t = _fireflyLifetime[i].y / _fireflyLifetime[i].x;

//Hermite interpolation parameters
Vector3 A = _fireflySpline[i*_controlPoints];
Vector3 B = _fireflySpline[i*_controlPoints+1];
float h00 = 2*Mathf.Pow(t,3) - 3*Mathf.Pow(t,2) + 1;
float h10 = Mathf.Pow(t,3) - 2*Mathf.Pow(t,2) + t;
float h01 = -2*Mathf.Pow(t,3) + 3*Mathf.Pow(t,2);
float h11 = Mathf.Pow(t,3) - Mathf.Pow(t,2);
//Firefly updated position
_fireflyObjects[i].transform.position = h00 * A + h10 * _fireflyTangents[i*_controlPoints]
+ h01 * B + h11 * _fireflyTangents[i*_controlPoints+1];
```

如果萤火虫完成了随机生成的整段路径，脚本从当前路径的终点开始创建一段新的随机路径：

```
//t > 1.0 indicates the end of the current path
if( t >= 1.0 )
{
    //Update the new position
    //Shift the second point to the first as well as the tangent
    _fireflySpline[i*_controlPoints] = _fireflySpline[i*_controlPoints+1];
    _fireflyTangents[i*_controlPoints] = _fireflyTangents[i*_controlPoints+1];

    //Shift the third point to the second, this point doesn't have a tangent
    _fireflySpline[i*_controlPoints+1] = _fireflySpline[i*_controlPoints+2];

    //Get new random control point within a certain angle from the current fly direction
    _fireflySpline[i*_controlPoints+2] = GetNewRandomControlPoint();

    //Compute the tangent for the central point
    _fireflyTangents[i*_controlPoints+1] = (_fireflySpline[i*_controlPoints+2] -
    _fireflySpline[i*_controlPoints+1])/2 + (_fireflySpline[i*_controlPoints+1] -
    _fireflySpline[i*_controlPoints])/2;

    //Set how long should take to navigate this part of path
    _fireflyLifetime[i].x = _fireflyMinLifetime;

    //Timer used to check how much we traveled along the path
    _fireflyLifetime[i].y = 0.0f;
}
```

## 6.15 正切空间至世界空间法线转换工具

正切空间至世界空间法线转换工具由 C# 脚本和着色器组成。该工具在 Unity 编辑器中离线运行，不会影响您的游戏的运行时性能。

本节包含以下小节：

- [6.15.1 关于正切空间至世界空间法线转换工具（第 6-177 页）。](#)
- [6.15.2 C# 脚本（第 6-177 页）。](#)
- [6.15.3 WorldSpaceNormalCreator 着色器（第 6-180 页）。](#)
- [6.15.4 WorldSpaceNormalsCreators C# 脚本（第 6-181 页）。](#)
- [6.15.5 WorldSpaceNormalCreator 着色器代码（第 6-183 页）。](#)

### 6.15.1 关于正切空间至世界空间转换工具

对冰穴演示的分析表明，其算术流水线中存在一个瓶颈。为降低负载，冰穴演示将世界空间法线贴图用于静态几何体，而不是正切空间法线贴图。

正切空间法线贴图可用于动画和动态对象，但需要额外的计算来正确定向取样的法线。

由于冰穴演示中大部分几何体是静态的，法线贴图转换成世界空间法线贴图。这可确保为纹理取样的法线已在世界空间中正确定向。此更改可以实现，因为冰穴演示光照是在一个自定义着色器中计算的，而 Unity 标准着色器使用正切空间法线贴图。

转换工具由以下内容组成：

- C# 脚本，用于在编辑器中添加新选项。
- 着色器，执行转换。

该工具在 Unity 编辑器中离线运行，不会影响您的游戏的运行时性能。

### 6.15.2 C# 脚本

您必须将 C# 脚本放在 Unity Assets/Editor 目录中。这可使脚本向 Unity 编辑器中的 **GameObject** 菜单添加新的选项。如果不存在此目录，请进行创建。

下列代码演示了如何在 Unity 编辑器中添加新选项：

```
[MenuItem("GameObject/World Space Normals Creator")]
static void CreateWorldSpaceNormals ()
{
    ScriptableWizard.DisplayWizard("Create World Space Normal",
        typeof(WorldSpaceNormalsCreator), "Create");
}
```

下图显示了脚本所添加的 **GameObject** 菜单选项。

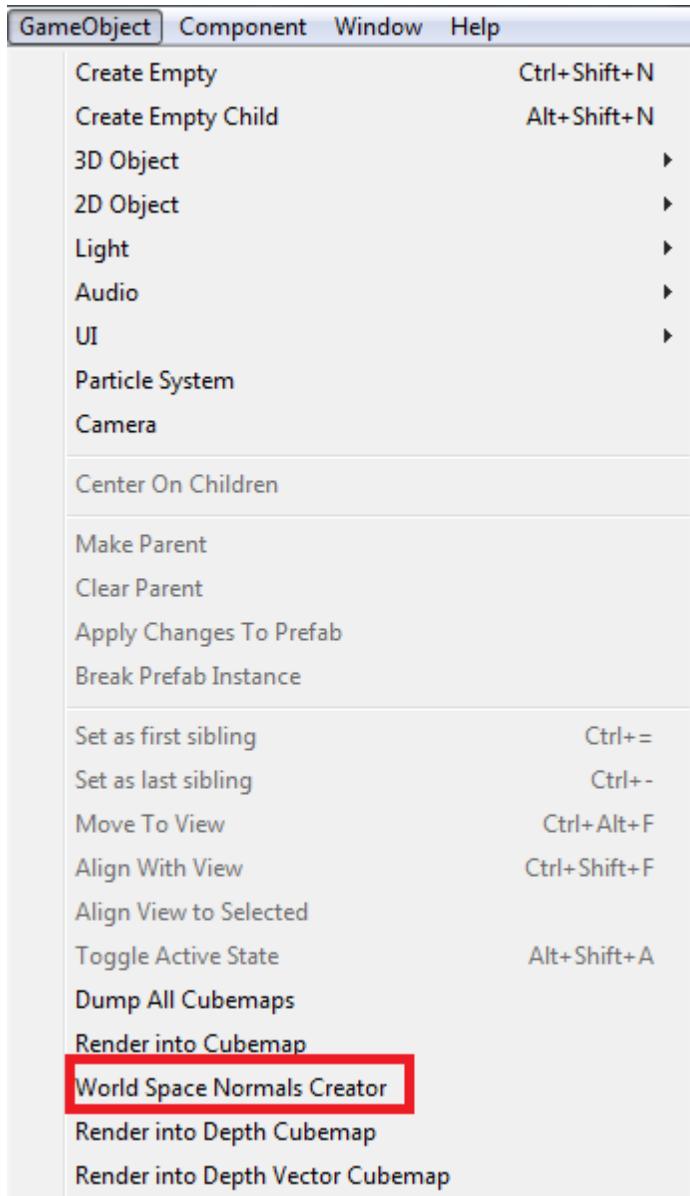


图 6-88 脚本添加的 GameObject 菜单项

C# 脚本中定义的类派生自 `Unity ScriptableWizard` 类，并且有权访问它的一些成员。从此类派生，生成编辑器向导。编辑器向导通常通过菜单项打开。

在 `OnWizardUpdate` 代码中，`helpString` 变量存放向导所创建的窗口中显示的帮助消息。

`isValid` 成员用于定义何时选中了所有正确的参数，以及何时可使用 **创建** 按钮。在本例中，`_currentObj` 成员已被选中，确保它指向有效的对象。

向导窗口的栏位是该类的公共成员。在本例中，只有 `_currentObj` 是公共的，因此向导窗口只有一个栏位。

下图显示了自定义向导窗口：

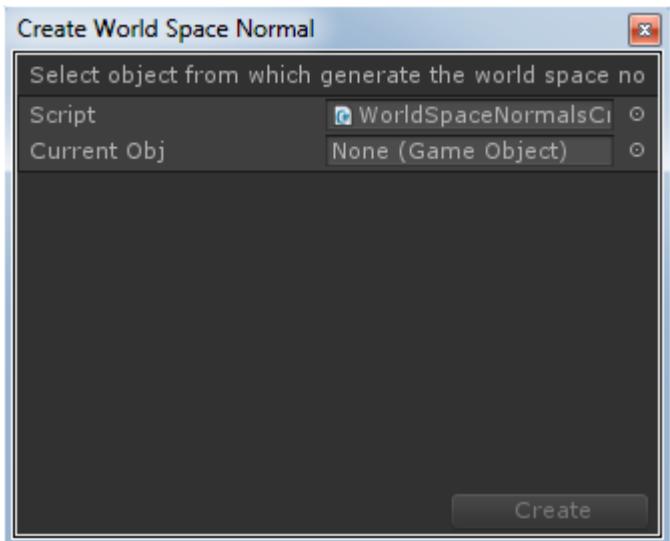


图 6-89 自定义向导窗口

当选中了对象并且单击了创建按钮时，系统将调用 `OnWizardCreate()` 函数。

`OnWizardCreate()` 函数执行转换的主要工作。

为转换法线，该工具创建一个临时摄像机，将新的世界空间法线渲染到 `RenderTexture`。为此，摄像机被设置为正交模式，对象的图层更改为未使用的层级。这意味着，它可以自行渲染该对象，即使它已经是场景的一个部分。

下列代码演示了如何设置此摄像机：

```
// Set antialiasing
QualitySettings.antiAliasing = 4;
Shader wns = Shader.Find ("Custom/WorldSpaceNormalCreator");
GameObject go = new GameObject( "WorldSpaceNormalsCam", typeof(Camera) );
_renderCamera = go.GetComponent<Camera> ();
_renderCamera.orthographic = true;
_renderCamera.nearClipPlane = 0.0f;
_renderCamera.farClipPlane = 10f;
_renderCamera.orthographicSize = 1.0f;
int prevObjLayer = _currentObj.layer;
_currentObj.layer = 30; //0x40000000
```

脚本设置了执行转换的替换着色器：

```
_renderCamera.SetReplacementShader (wns,null);
_renderCamera.useOcclusionCulling = false;
```

摄像机被指向对象。这可防止对象在视锥体剔除期间被移除：

```
_renderCamera.transform.rotation = Quaternion.LookRotation (_currentObj.transform.position -
_renderCamera.transform.position);
```

对于分配至对象的每一材质，脚本查找 `_BumpMap` 纹理。此纹理设置为利用着色器全局函数的替换着色器的来源纹理。

清色设置为 `(0.5,0.5,0.5)`，因为必须表示指向负方向的法线。

```
foreach (Material m in materials)
{
    Texture t = m.GetTexture("_BumpMap");
    if( t == null )
    {
        Debug.LogError("the material has no texture assigned named Bump Map");
        continue;
    }
    Shader.SetGlobalTexture ("_BumpMapGlobal", t);
    RenderTexture rt = new RenderTexture(t.width,t.height,1);
```

```
_renderCamera.targetTexture = rt;
_renderCamera.pixelRect = new Rect(0,0,t.width,t.height);
_renderCamera.backgroundColor = new Color( 0.5f, 0.5f, 0.5f);
_renderCamera.clearFlags = CameraClearFlags.Color;
_renderCamera.cullingMask = 0x40000000;
_renderCamera.Render();
Shader.SetGlobalTexture ("_BumpMapGlobal", null);
```

摄像机渲染了场景后，像素被读回，并保存为 PNG 图像。

```
Texture2D outTex = new Texture2D(t.width,t.height);
RenderTexture.active = rt;
outTex.ReadPixels(new Rect(0,0,t.width,t.height), 0, 0);
outTex.Apply();
RenderTexture.active = null;
byte[] _pixels = outTex.EncodeToPNG();
System.IO.File.WriteAllBytes("Assets/Textures/GeneratedWorldSpaceNormals/" + t.name
+ "_WorldSpace.png", _pixels);
```

摄像机剔除遮罩使用二进制遮罩（用十六进制格式表示）指定要渲染的图层。

本例中使用了图层 30：

```
_currentObj.layer = 30;
```

其十六进制格式为 0x40000000，因为它的第三十位被设为 1。

### 6.15.3 WorldSpaceNormalCreator 着色器

实施转换的着色器代码非常简单明了。它不使用实际的顶点位置，而将顶点的纹理坐标用作其位置。这使得对象投影到一个 2D 平面上，与纹理化时相同。

为使 OpenGL 流水线正确运作，UV 坐标从标准的 [0, 1] 范围移到 [-1, 1] 范围，并逆转 Y 坐标。未使用 Z 坐标，所以它可以设置为 0 或在近处和远处剪切平面内的任何值：

```
output.pos = half4(input.tex.x*2.0 - 1.0, ((1.0 - input.tex.y)*2.0 - 1.0), 0.0, 1.0);
output.tc = input.tex;
```

法线、切线和双切线在顶点着色器中计算，并传递到片段着色器以执行转换：

```
output.normalInWorld = normalize(mul(half4(input.normal, 0.0), _World2Object).xyz);
output.tangentWorld = normalize(mul(_Object2World, half4(input.tangent.xyz, 0.0)).xyz);
output.bitangentWorld = normalize(cross(output.normalInWorld, output.tangentWorld)
* input.tangent.w);
```

片段着色器：

1. 将法线从正切空间转换为世界空间。
2. 将法线缩放到 [0, 1] 范围。
3. 将法线输出到新纹理。

下列代码对此进行了演示：

```
half3 normalInWorld = half3(0.0,0.0,0.0);
half3 bumpNormal = UnpackNormal(tex2D(_BumpMapGlobal, input.tc));
half3x3 local2WorldTranspose = half3x3( input.tangentWorld,
                                         input.bitangentWorld,
                                         input.normalInWorld);
normalInWorld = normalize(mul(bumpNormal, local2WorldTranspose));
normalInWorld = normalInWorld*0.5 + 0.5;
return half4(normalInWorld,1.0);
```

下图显示了处理之前的正切空间法线贴图：



图 6-90 原始正切空间法线贴图

下图显示了该工具生成的世界空间法线贴图：

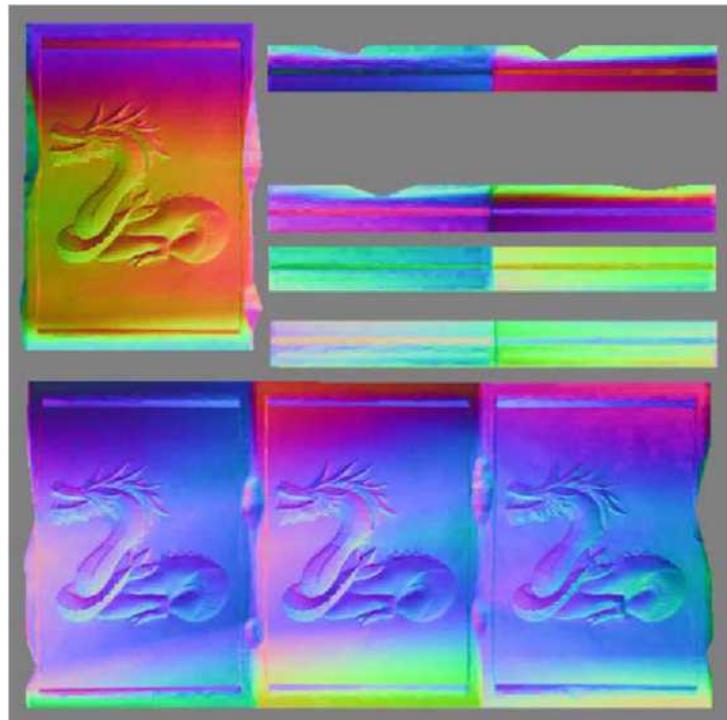


图 6-91 生成的世界空间法线贴图

#### 6.15.4 WorldSpaceNormalsCreators C# 脚本

以下是 WorldSpaceNormalsCreators C# 脚本的代码：

```
using UnityEngine;
using UnityEditor;
using System.Collections;

public class WorldSpaceNormalsCreator : ScriptableWizard
```

```

    public GameObject _currentObj;
    private Camera _renderCamera;
    void OnWizardUpdate()
    {
        helpString = "Select object from which generate the world space normals";
        if(_currentObj != null)
        {
            isValid = true;
        }
        else
        {
            isValid = false;
        }
    }
    void OnWizardCreate ()
    {
        // Set antialiasing
        QualitySettings.antiAliasing = 4;
        Shader wns = Shader.Find ("Custom/WorldSpaceNormalCreator");
        GameObject go = new GameObject( "WorldSpaceNormalsCam", typeof(Camera) );
        //Set the new camera to perform orthographic projection
        _renderCamera = go.GetComponent<Camera> ();
        _renderCamera.orthographic = true;
        _renderCamera.nearClipPlane = 0.0f;
        _renderCamera.farClipPlane = 10f;
        _renderCamera.orthographicSize = 1.0f;
        //Save the current object layer and set it to a unused one
        int prevObjLayer = _currentObj.layer;
        _currentObj.layer = 30; //0x40000000
        //Set the replacement shader for the camera
        _renderCamera.SetReplacementShader (wns,null);
        _renderCamera.useOcclusionCulling = false;
        //Rotate the camera to look at the object to avoid frustum culling
        _renderCamera.transform.rotation = Quaternion.LookRotation
        (_currentObj.transform.position - _renderCamera.transform.position);
        MeshRenderer mr = _currentObj.GetComponent<MeshRenderer> ();
        Material[] materials = mr.sharedMaterials;
        foreach (Material m in materials)
        {
            Texture t = m.GetTexture("_BumpMap");
            if( t == null )
            {
                Debug.LogError("the material has no texture assigned named Bump Map");
                continue;
            }
            //Render the world space normal maps to a texture
            Shader.SetGlobalTexture ("_BumpMapGlobal", t);
            RenderTexture rt = new RenderTexture(t.width,t.height,1);
            _renderCamera.targetTexture = rt;
            _renderCamera.pixelRect = new Rect(0,0,t.width,t.height);
            _renderCamera.backgroundColor = new Color( 0.5f, 0.5f, 0.5f);
            _renderCamera.clearFlags = CameraClearFlags.Color;
            _renderCamera.cullingMask = 0x40000000;
            _renderCamera.Render();
            Shader.SetGlobalTexture ("_BumpMapGlobal", null);
            Texture2D outTex = new Texture2D(t.width,t.height);
            RenderTexture.active = rt;
            outTex.ReadPixels(new Rect(0,0,t.width,t.height), 0, 0);
            outTex.Apply();
            RenderTexture.active = null;
            //Save it to PNG
            byte[] _pixels = outTex.EncodeToPNG();
            System.IO.File.WriteAllBytes("Assets/Textures/GeneratedWorldSpaceNormals/"
            +t.name+"_WorldSpace.png",_pixels);
        }
        _currentObj.layer = prevObjLayer;
        DestroyImmediate(go);
    }
    [MenuItem("GameObject/World Space Normals Creator")]
    static void CreateWorldSpaceNormals ()
}

```

```

    {
        ScriptableWizard.DisplayWizard("Create World Space Normal",
            typeof(WorldSpaceNormalsCreator), "Create");
    }
}

```

### 6.15.5 WorldSpaceNormalCreator 着色器代码

以下是 WorldSpaceNormalCreator 着色器的代码:

```

Shader "Custom/WorldSpaceNormalCreator" {
    Properties {
    }
    SubShader {
        Cull off
        Pass {
            CGPROGRAM
            #pragma target 3.0
            #pragma glsl
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"
            uniform sampler2D _BumpMapGlobal;
            struct vin
            {
                half4 tex : TEXCOORD0;
                half3 normal : NORMAL;
                half4 tangent : TANGENT;
            };
            struct vout
            {
                half4 pos : POSITION;
                half2 tc : TEXCOORD0;
                half3 normalInWorld : TEXCOORD1;
                half3 tangentWorld : TEXCOORD2;
                half3 bitangentWorld : TEXCOORD3;
            };
            vout vert (vin input )
            {
                vout output;
                output.pos = half4(input.tex.x*2.0 - 1.0,((1.0-input.tex.y)*2.0 - 1.0),
                    0.0, 1.0);
                output.tc = input.tex;
                output.normalInWorld = normalize(mul(half4(input.normal, 0.0),
                    _World2Object).xyz);
                output.tangentWorld = normalize(mul(_Object2World,
                    half4(input.tangent.xyz, 0.0)).xyz);
                output.bitangentWorld = normalize(cross(output.normalInWorld,
                    output.tangentWorld) * input.tangent.w);
                return output;
            }
            float4 frag( vout input ) : COLOR
            {
                half3 normalInWorld = half3(0.0,0.0,0.0);
                half3 bumpNormal = UnpackNormal(tex2D(_BumpMapGlobal, input.tc));
                half3x3 local2WorldTranspose = half3x3(
                    input.tangentWorld,
                    input.bitangentWorld,
                    input.normalInWorld);
                normalInWorld = normalize(mul(bumpNormal, local2WorldTranspose));
                normalInWorld = normalInWorld*0.5 + 0.5;
                return half4(normalInWorld,1.0);
            }
        ENDCG
    }
}

```

# 第 7 章

# 虚拟现实

本章节描述了调整应用程序或游戏以便在虚拟现实硬件上运行的流程，同时介绍了虚拟现实中实施反射的一些区别。

它包含下列部分：

- [7.1 Unity 虚拟现实硬件支持（第 7-185 页）](#)。
- [7.2 Unity VR 移植流程（第 7-186 页）](#)。
- [7.3 移植到 VR 时需要考虑的问题（第 7-189 页）](#)。
- [7.4 VR 中的反射（第 7-191 页）](#)。
- [7.5 结果（第 7-196 页）](#)。

## 7.1 Unity 虚拟现实硬件支持

有多种类型的**虚拟现实(VR)**硬件具有**Unity**支持。**Unity**原生支持一些设备。也可通过插件实现对一些其他设备的支持。

如果设备具有原生**Unity VR**支持,它实现的性能要高于使用插件支持的设备。这是因为**Unity**为具有原生**Unity VR**支持的设备实施了内部优化。

下列设备具有原生**Unity VR**支持:

- Oculus Rift。
- Samsung Gear VR。
- PlayStation VR。
- Microsoft Hololens。

下列设备可通过插件支持**Unity VR**:

- Google Cardboard。
- Moverio。
- HTC Vive 以及 Steam VR 平台支持的其他设备。

多种其他设备可借助面向**Unity**的**开源虚拟现实(OSVR)**插件受到支持。

## 7.2 Unity VR 移植流程

应用程序或游戏移植到原生 Unity VR 是一个多阶段流程。

应用程序移植到 Unity VR 时需要的阶段有：

1. 安装 Unity 5.1 或更高版本。Unity 5.1 和更高版本原生支持 VR。
2. 若有必要，可从适当的网站获取您的设备的签名文件，并将它放入设备上指定的文件夹中。

对于运行冰穴演示的 Samsung Gear VR，则应访问 Oculus 开发者网站  
<https://developer.oculus.com/osig>。Samsung 设备的签名文件必须放入 Plugins/Android/assets 文件夹。

3. 在 Unity 中，选择打开文件 > 构建设置 > 播放器设置。此时显示“播放器设置”窗口。在“其他设置”部分中，打开支持虚拟现实选项。

下图显示了此窗口的屏幕快照。

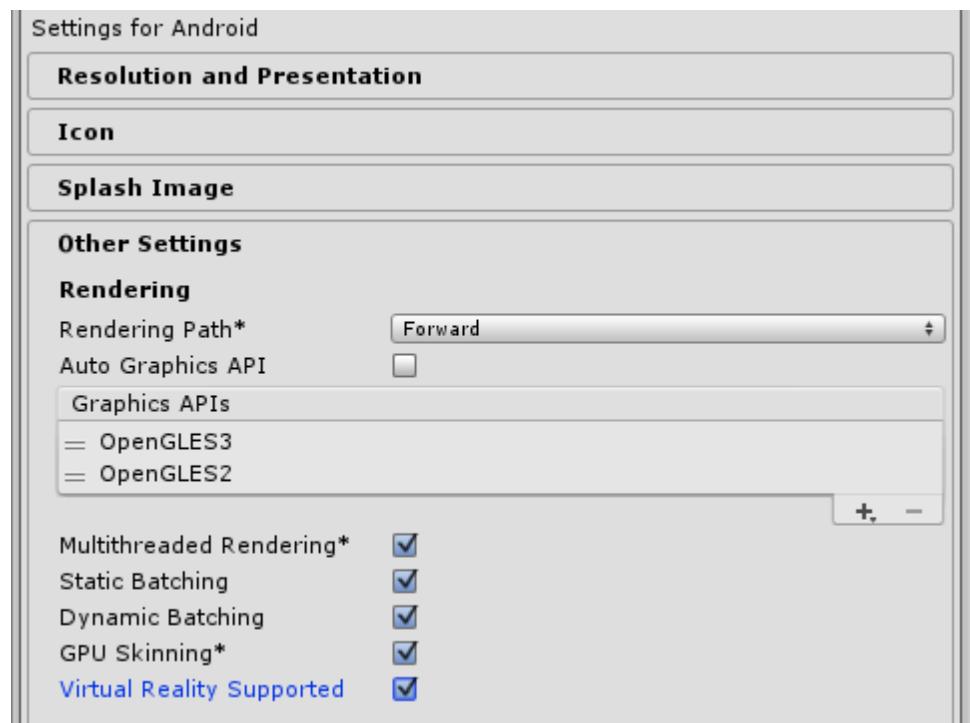


图 7-1 播放器设置窗口

4. 设置摄像机的父级。所有摄像机控制必须设置相对于此摄像机父级的位置和朝向。
5. 根据需要，将摄像机控制与 VR 耳机触控板关联。
6. 对于 Android 设备，启用开发者选项并打开 USB 调试。

下图显示了开发者选项菜单。

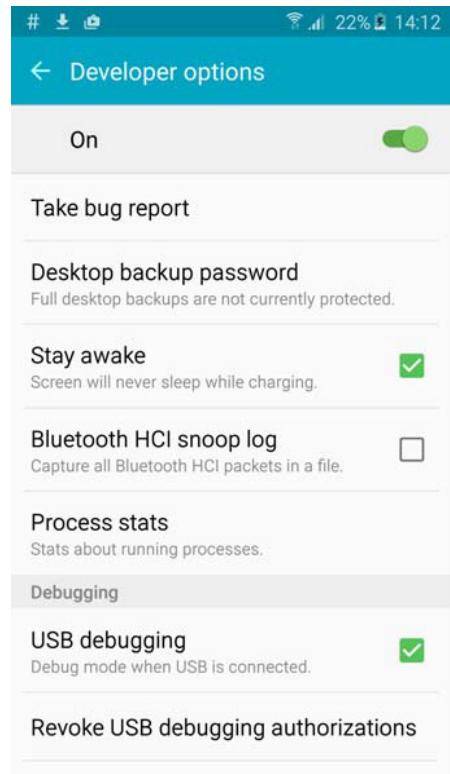


图 7-2 开发者选项菜单

7. 生成应用程序，并将它安装到设备上。
8. 启动应用程序。

在 Samsung 设备上启动应用程序时，它将提示您将设备插入耳机。如果设备尚未做好 VR 准备，它将提示您连接网络下载 Samsung VR 软件。

本节包含以下小节：

- [7.2.1 启用 Samsung Gear VR 开发者模式（第 7-187 页）。](#)

## 7.2.1 启用 Samsung Gear VR 开发者模式

开发者模式可以帮助您视觉化呈现运行的 VR 应用程序，而不必将设备插入 VR 耳机。

如果之前安装过已签名 VR 应用程序，则可启用 Samsung Gear VR 开发者模式。如果之前未安装过已签名 VR 应用程序，请安装一个已签名 VR 应用程序，以便能启用此模式。

启用开发者模式：

1. 在 Samsung 设备上，选择设置 > 应用管理器 > Gear VR 服务。
2. 选择管理存储。
3. 点击 VR 服务版本六次。
4. 等待扫描流程完成。此时显示开发者模式开关。

### 备注

使用开发者模式将缩短手机的电池续航时间，因为它将覆盖未使用时关闭耳机的所有设置。

下图显示了来自 Samsung Gear VR 开发者模式视图的冰穴屏幕快照示例。



图 7-3 在 Samsung Gear VR 开发者模式中运行的 VR 应用程序的屏幕快照示例

### 7.3 移植到 VR 时需要考虑的问题

VR 创造了与其他应用程序类型大为不同的用户体验。这意味着一些适用于非 VR 应用程序或游戏的项目对 VR 无效。

针对几位不同的用户测试应用程序，判断其舒适程度并调整代码，让他们找到舒适的体验。

在非 VR 游戏中悦目的摄像机动画可能会在游戏的 VR 版本中令人不适。例如，非 VR 的冰穴演示具有针对摄像机的动画模式。部分用户感觉这令人不适，并产生运动病症，尤其是当摄像机向后移动时。移除此模式可以防止这种让人不安的体验。

非 VR 应用程序可以通过手机的触控屏幕或倾斜手机来进行控制，而在 VR 应用程序中可能无法实现这些控制机制。例如，原始版本的冰穴演示利用两个虚拟手柄控制摄像机，而这在 VR 设备上不可工作，因为无法访问触控屏幕。冰穴 VR 演示设计为可以在 Samsung Gear VR 上运行，其耳机一侧配有触控板。使用触控板而非触控屏幕，即可解决此问题。

下图显示了 Samsung Gear VR 耳机上的触控板。



图 7-4 Samsung Gear VR 耳机触控板位置

用户可能会发现，一些在非 VR 应用程序中正常的视觉特效在 VR 应用程序中出现错误。例如，非 VR 版冰穴演示使用了一个脏镜头光晕效果，它会根据摄像机与太阳的对齐情况改变强度。在 VR 中测试此效果的用户发现它显示错误，所以它已被去除。

本节包含以下小节：

- [7.3.1 摄像机的外部设备控制（第 7-189 页）](#)。

#### 7.3.1 摄像机的外部设备控制

VR 可以从通常不与手机和移动设备关联的控制方法获益。其中一些方法值得考虑，具体取决于应用程序的目标受众。

可以使用控制器，通过蓝牙将它们与 VR 设备连接。为实现这一点，冰穴演示使用了一个扩展 Unity 功能的自定义插件，使得它能够解读 Android 蓝牙事件。这些事件触发摄像机的运动。

下图显示了可用于冰穴演示的蓝牙控制器。



图 7-5 控制冰穴演示的蓝牙控制器

## 7.4 VR 中的反射

反射在任何 VR 应用程序中都很重要。现实世界包含许多反射，所以游戏或应用程序中缺乏它们，人们会注意到。

VR 中的反射可以使用传统游戏所用的相同技巧，但必须加以修改，使它能够配合用户所看到的立体视觉输出。

良好实施反射可以让应用程序或游戏变得更加真实、更有沉浸感。然而，在 VR 中实施反射时您必须考虑一些额外的问题。

本节包含以下小节：

- [7.4.1 使用局部立方体贴图的反射（第 7-191 页）。](#)
- [7.4.2 组合不同类型的反射（第 7-191 页）。](#)
- [7.4.3 立体反射（第 7-191 页）。](#)

### 7.4.1 使用局部立方体贴图的反射

您可以使用局部立方体贴图创建反射。这种方法避免了每一帧创建反射纹理，而是从预先渲染的立方体贴图获取反射纹理。此方法根据立方体贴图生成的位置和场景包围盒，对反射向量应用局部修正，并使用该信息来获取正确的纹理。

使用局部立方体贴图生成的反射不会出现像素不稳定或像素闪光，而在运行时为每一帧生成反射时可能会有这样的问题。

有关使用局部立方体贴图的反射的更多信息，请参见 [6.2 使用局部立方体贴图实现反射（第 6-98 页）](#)。

### 7.4.2 组合不同类型的反射

需要使用不同的反射生成技巧，从而获得最佳的性能和效果。具体需要哪一种技巧取决于反射表面的形状，以及反射表面和被反射对象是静态还是动态。来自不同反射生成技巧的结果必须进行组合，从而产生用户所见到的结果。

有关组合不同反射类型的信息，请参见 [6.3 组合反射（第 6-114 页）](#)。

### 7.4.3 立体反射

在非 VR 游戏中，只有一个摄像机视点。而在 VR 中，每只眼睛都有一个摄像机视点。这意味着，必须为每只眼睛单独计算反射。

如果向两只眼睛呈现相同的反射，那么用户很快就会注意到反射中没有深度。这与他们的预期不符，可能会破坏他们的沉浸感，给 VR 体验的质量造成负面影响。

为修正此问题，必须计算并显示两个反射，同时根据用户在游戏中观察时每只眼睛的位置加以正确调整。

为了在冰穴演示中实施这些反射，它将两个反射纹理用于来自动态对象的平面反射，并使用两个不同的局部修正反射向量从一个供静态对象反射使用的单一局部立方体贴图获取纹理。

反射可能是动态或静态对象。每一种反射需要进行一组不同的更改，才能在 VR 中工作。

#### 在 Unity VR 中实施立体平面反射

在 VR 游戏中实施立体反射需要对非 VR 游戏代码进行一些调整。

在开始之前, 请先确保在 Unity 中启用了虚拟现实支持。为此, 请选择**构建设置 > 播放器设置 > 其他设置**, 再选中**支持虚拟现实**复选框。

### 动态立体平面反射

动态反射要求进行一些更改, 从而为两只眼睛生成正确的结果。

您必须创建两个摄像机, 并且为每个摄像机创建要渲染到的目标纹理。禁用两个摄像机, 使它们的渲染通过程序执行。然后, 为两者附加下列脚本。

```
void OnPreRender(){
    SetUpReflectionCamera();
    // Invert winding
    GL.invertCulling = true;
}
void OnPostRender(){
    // Restore winding
    GL.invertCulling = false;
}
```

此脚本使用主摄像机的位置和朝向设定反射摄像机的位置和朝向。为此, 它将调用 `SetUpReflectionCamera()` 函数, 紧接在左右反射摄像机渲染之前。下列代码演示了此函数的实施方式。

```
public GameObject reflCam;
public float clipPlaneOffset ;
...
private void SetUpReflectionCamera(){
    // Find out the reflection plane: position and normal in world space
    Vector3 pos = gameObject.transform.position;

    // Reflection plane normal in the direction of Y axis
    Vector3 normal = Vector3.up;
    float d = -Vector3.Dot(normal, pos) - clipPlaneOffset;
    Vector4 reflPlane = new Vector4(normal.x, normal.y, normal.z, d);
    Matrix4x4 reflection = Matrix4x4.zero;
    CalculateReflectionMatrix(ref reflection, reflPlane);

    // Update reflection camera considering main camera position and orientation
    // Set view matrix
    Matrix4x4 m = Camera.main.worldToCameraMatrix * reflection;
    reflCam.GetComponent<Camera>().worldToCameraMatrix = m;

    // Set projection matrix
    reflCam.GetComponent<Camera>().projectionMatrix = Camera.main.projectionMatrix;
}
```

此函数计算反射摄像机的视图和投影矩阵。它决定了反射变换, 以应用到主摄像机的视图矩阵 `worldToCameraMatrix`。

为设置每只眼睛的摄像机位置, 可将下列代码添加到这一行后面: `Matrix4x4 m = Camera.main.worldToCameraMatrix * reflection;`:

#### 左眼

```
m[12] += stereoSeparation;
```

#### 右眼

```
m[12] -= stereoSeparation;
```

移位值 `stereoSeparation` 为 0.011。`stereoSeparation` 值是眼睛分离值的一半。

将另一脚本附加到主摄像机, 以控制左右两个反射摄像机的渲染。下列代码演示了此脚本的冰穴实施。

```
public class RenderStereoReflections : MonoBehaviour
{
    public GameObject reflectiveObj;
    public GameObject leftReflCamera;
    public GameObject rightReflCamera;
    int eyeIndex = 0;

    void OnPreRender(){
```

```
if (eyeIndex == 0){  
    // Render Left camera  
    leftReflCamera.GetComponent<Camera>().Render();  
    reflectiveObj.GetComponent<Renderer>().material.SetTexture(  
        "_DynReflTex", leftReflCamera.GetComponent<Camera>().targetTexture);  
}  
else{  
    // Render right camera  
    rightReflCamera.GetComponent<Camera>().Render();  
    reflectiveObj.GetComponent<Renderer>().material.SetTexture(  
        "_DynReflTex", rightReflCamera.GetComponent<Camera>().targetTexture);  
}  
eyeIndex = 1 - eyeIndex;  
}
```

此脚本在主摄像机的 `OnPreRender()` 回调函数中处理左右反射摄像机的渲染。先对左眼调用此脚本一次，而后对右眼调用一次。`eyeIndex` 变量为各个反射摄像机分配正确的渲染顺序，并将正确的反射应用到主摄像机的每个眼睛。第一次调用该回调函数时，将假定其针对的是左眼。这是 Unity 调用 `OnPreRender()` 方法的顺序。

#### 检查为各个眼睛使用不同的纹理

检查脚本是否为各个眼睛正确生成不同的渲染纹理，这一点很重要。测试是否为每个眼睛显示了正确纹理：

#### 步骤

1. 更改脚本，使它将 `eyeIndex` 值作为统一变量传递到着色器。
2. 对反射纹理使用两种颜色，分别用于各个 `eyeIndex` 值。

如果您的脚本正常工作，其输出将类似于下图，其中显示了两个不同的稳定反射处于可见状态的屏幕快照。

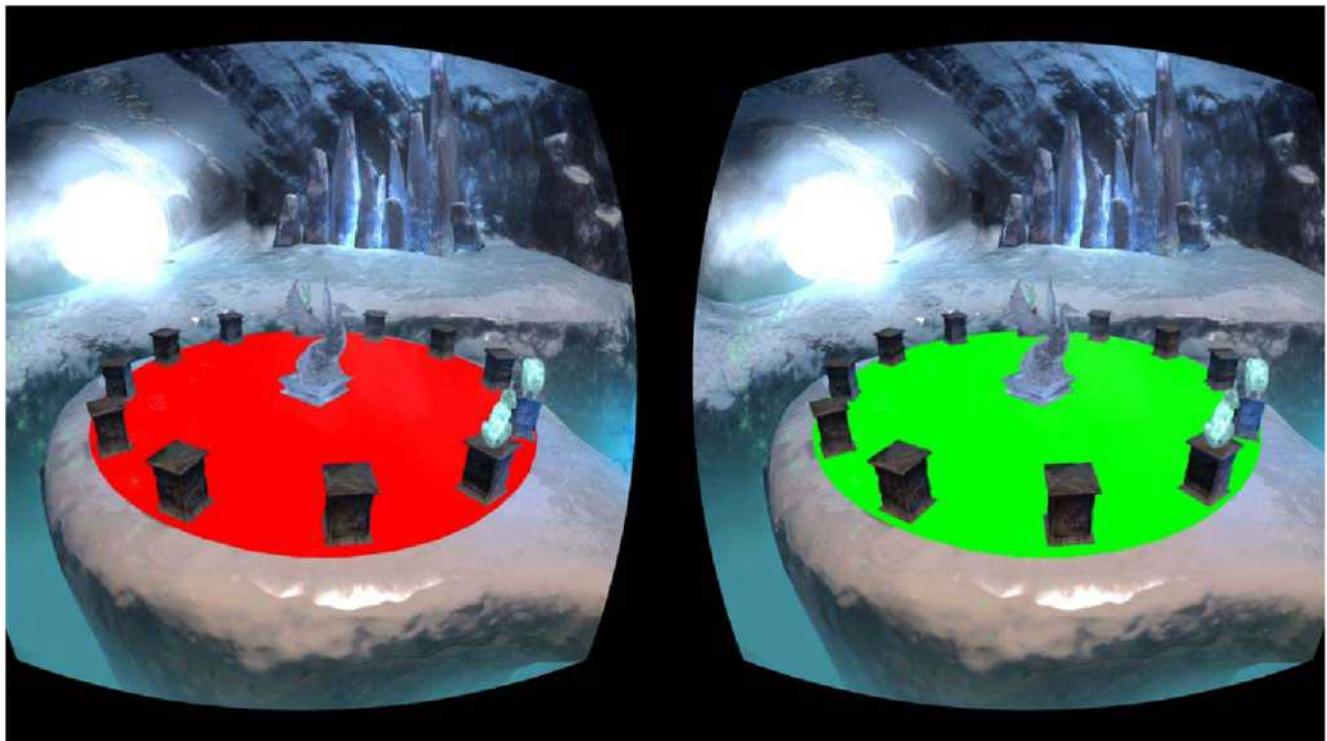


图 7-6 正确反射纹理输出检查的示例

## 静态立体反射

您可以使用立方体贴图，高效地创建来自静态物体的立体反射。唯一的差别在于，您必须使用两个反射向量从立方体贴图获取像素元，分别用于每只眼睛。

Unity 提供了一个内置的值，用于在着色器中访问世界坐标表示的摄像机位置：

```
_WorldSpaceCameraPos.
```

但在 VR 中，需要左、右两个摄像机的位置。`_WorldSpaceCameraPos` 无法提供左、右两个摄像机的位置。因此，您必须使用脚本计算左、右两个摄像机的位置，并将结果作为统一变量传递到着色器。

在可以传递摄像机位置信息的着色器中声明新的统一变量：

```
uniform float3 _StereoCamPosWorld;
```

计算左、右两个摄像机位置的最佳场所是附加到主摄像机的脚本中，因为这可以轻松访问主摄像机视图矩阵。下列代码演示可如何对 `eyeIndex = 0` 情形进行此操作。

代码修改了主摄像机的视图矩阵，以设置用局部坐标表示的左眼位置。左眼位置需要在世界坐标中，所以要查找逆矩阵。左眼位置通过统一变量 `_StereoCamPosWorld` 传递到着色器。

```
Matrix4x4 mWorldToCamera = gameObject.GetComponent<Camera>().worldToCameraMatrix;
mWorldToCamera[12] += stereoSeparation;
Matrix4x4 mCameraToWorld = mWorldToCamera.inverse;
Vector3 mainStereoCamPos = new Vector3(mCameraToWorld[12], mCameraToWorld[13],
                                         mCameraToWorld[14]);
reflectiveObj.GetComponent<Renderer>().material.SetVector("_StereoCamPosWorld",
    new Vector3 (mainStereoCamPos.x, mainStereoCamPos.y, mainStereoCamPos.z));
```

代码对右眼相同，除了立体分离是从 `mWorldToCamera[12]` 相减而来，而不是相加而得。

在顶点着色器中，您必须找到下面这一行，它负责计算视图向量：

```
output.viewDirInWorld = vertexWorld.xyz - _WorldSpaceCameraPos;
```

将它替换为下面这一行，以使用通过世界坐标表示的新左、右眼摄像机位置：

```
output.viewDirInWorld = vertexWorld.xyz - _StereoCamPosWorld;
```

实施了立体反射时，它在应用程序以编辑器模式运行时可见，因为反射纹理会由于它不断从左眼改为右眼而闪烁。这种闪烁在 VR 设备上不可见，因为对每只眼睛使用了不同的纹理。

## 优化立体反射

如果不作进一步优化，立体反射实施将始终运行。这意味着，反射不可见时其处理时间就会浪费。

在对反射本身执行任何工作之前，插入检查反射表面是否可见的代码。为此，可将类似下例所示的代码附加到反射对象上。

```
public class IsReflectiveObjectVisible : MonoBehaviour
{
    public bool reflObjIsVisible;

    void Start(){
        reflObjIsVisible = false;
    }

    void OnBecameVisible(){
        reflObjIsVisible = true;
    }

    void OnBecameInvisible(){}
```

```
        reflObjIsVisible = false;  
    }  
}
```

在定义了这个类后，在附加到主摄像机的脚本中使用以下 `if` 语句，使得仅在反射对象可见时才执行立体反射的计算。

```
void OnPreRender(){  
    if (reflectiveObjetc.GetComponent<IsReflectiveObjectVisible>().reflObjIsVisible){  
        ..  
    }  
}
```

其余的代码放入到此 `if` 语句内。此 `if` 语句使用 `IsReflectiveObjectVisible` 类检查反射对象是否可见。若不可见，则不计算反射。

## 7.5 结果

此工作创建了您的游戏的 VR 版本，它实施的立体反射有益于加强沉浸感，提高整体的 VR 用户体验。更改为立体反射可以大幅提升用户体验；这是因为，如果不实施，人们会注意到反射缺少深度。

下图显示了冰穴演示在开发者模式中运行时的屏幕照例，它展示了所实施的立体反射。

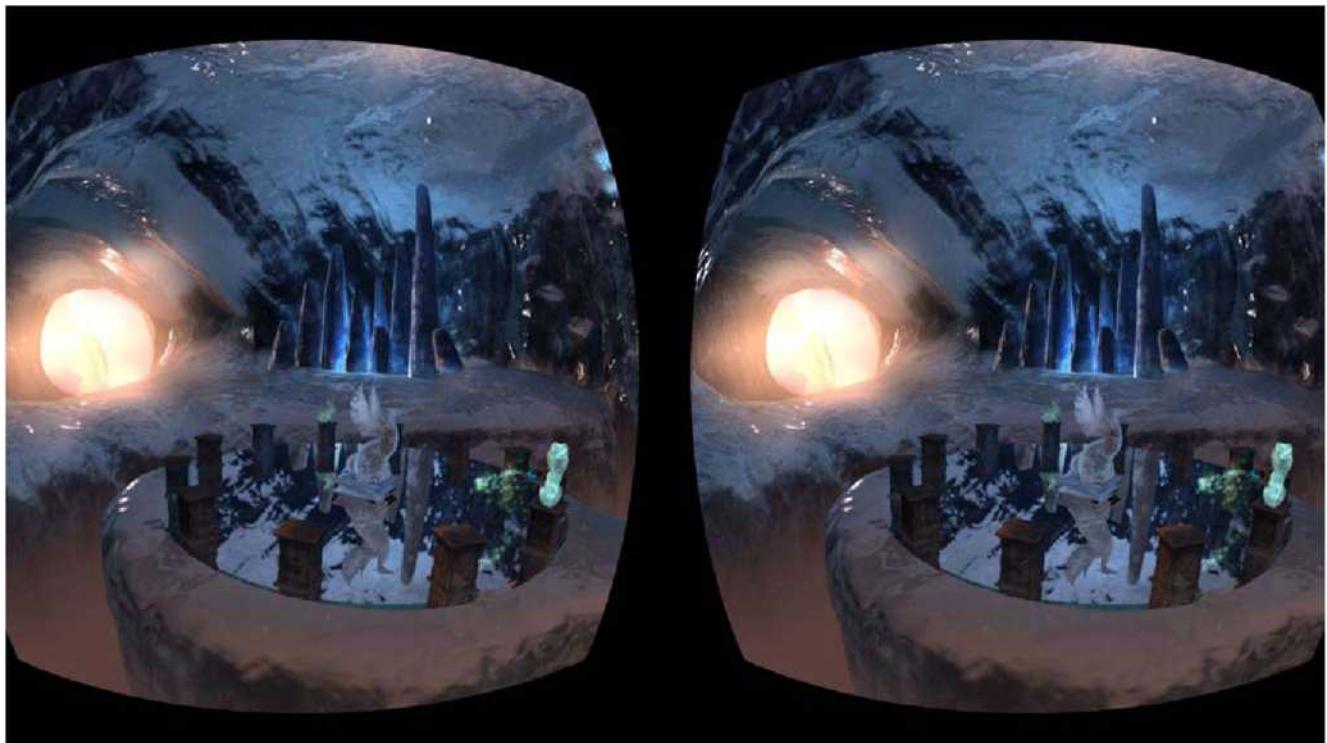


图 7-7 冰穴演示屏幕快照

# 附录 A

## 修订

此附录介绍本书发布的版本之间的变化。

它包含下列部分：

- [A.1 修订](#) (第 Appx-A-198 页)。

## A.1 修订

此附录介绍本书发布之后的内容增补。

表 A-1 版本 3.0 中的增补

增补	位置	影响
在自定义着色器中使用 <i>Enlighten</i>	<a href="#">5.5 在自定义着色器中使用 <i>Enlighten</i> (第 5-80 页)</a>	版本 3.0
组合反射	<a href="#">6.3 组合反射 (第 6-114 页)</a>	版本 3.0
使用 <i>Early-z</i>	<a href="#">6.7 使用 <i>Early-z</i> (第 6-137 页)</a>	版本 3.0
脏镜头光晕效果	<a href="#">6.8 脏镜头光晕效果 (第 6-138 页)</a>	版本 3.0
光柱	<a href="#">6.9 光柱 (第 6-141 页)</a>	版本 3.0
雾化效果	<a href="#">6.10 雾化效果 (第 6-145 页)</a>	版本 3.0
高光溢出	<a href="#">6.11 高光溢出 (第 6-152 页)</a>	版本 3.0
冰墙效果	<a href="#">6.12 冰墙效果 (第 6-159 页)</a>	版本 3.0
过程天空盒	<a href="#">6.13 过程天空盒 (第 6-165 页)</a>	版本 3.0
萤火虫	<a href="#">6.14 萤火虫 (第 6-173 页)</a>	版本 3.0
正切空间至世界空间的转换工具。	<a href="#">6.15 正切空间至世界空间法线转换工具 (第 6-177 页)</a>	版本 3.0

表 A-2 版本 3.0\_01 中的变更

变更	位置	影响
从“使用 <i>Early-z</i> ”中删除了一个列表条目	<a href="#">6.7 使用 <i>Early-z</i> (第 6-137 页)</a>	版本 3.0

表 A-3 版本 3.1 中的变更

变更	位置	影响
添加了关于虚拟现实的一个章节	<a href="#">第 7 章 虚拟现实 (第 7-184 页)</a>	版本 3.0

表 A-4 版本 3.2 中的变更

变更	位置	影响
更新了 <i>Enlighten</i> 章节的信息和结构	<a href="#">第 5 章 Unity 中利用 <i>Enlighten</i> 的全局照明 (第 5-60 页)</a>	版本 3.2 首次发布