Arm® 开发者指南

版本 4.1

优化移动游戏图形

arm

100140_0401_00_zh

Arm® 开发者指南

优化移动游戏图形

版本信息

文档历史

发行号	日期	机密性	变更
0100-00	05 九月 2014	非机密	首次发布版本 1.0
0200-00	23 六月 2015	非机密	首次发布版本 2.0
0201-00	28 七月 2015	非机密	首次发布版本 2.1
0300-00	18 九月 2015	非机密	首次发布版本 3.0
0300-01	05 十一月 2015	非机密	第二次发布版本 3.0
0301-00	07 四月 2016	非机密	首次发布版本 3.1
0301-01	25 四月 2016	非机密	第二次发布版本 3.1
0302-00	31 五月 2016	非机密	首次发布版本 3.2
0303-00	17 三月 2017	非机密	首次发布版本 3.3
0303-01	01 六月 2017	非机密	第二次发布版本 3.3
0400-00	19 九月 2019	非机密	首次发布版本 4.0
0401-00	28 二月 2020	非机密	首次发布版本 4.1

相关参考资料

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at *http://www.arm.com/company/policies/trademarks*.

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

非机密专有权声明

Arm Limited。英格兰注册公司，注册号：02557590。

110 Fulbourn Road, Cambridge, England CB1 9NJ。

LES-PRE-20349

相关参考资料
保密状态

本文档是非机密文档。根据 Arm 与从 Arm 处收取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。

相关参考资料
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。

相关参考资料

网址

www.arm.com

相关参考资料

内容
Arm® 开发者指南 优化移动游戏图形

前言

此前言介绍了 *Arm® 开发者指南 优化移动游戏图形*.

它包含以下:

- *关于本手册* on page 9.
- *反馈* on page 11.

关于本手册

本手册旨在帮助你充分利用 Unity 来为移动端创作应用程序和内容，尤其是配备 Mali™ GPU 的移动设备。

产品修改情况

r*m*p*n* 标识符指示本书所述产品的修改情况，例如 r*1*p*2*，其中：

r*m*　表示产品经过重大修改，例如 r1。

p*n*　表示产品经过略微修改，例如 p2。

相关参考资料

目标受众

This book is for beginner to intermediate developers.

相关参考资料

使用本手册

本手册由以下章节组成：

章 1 简介
本章介绍了《 Arm 开发者指南：优化移动游戏图形》。

章 2 优化应用程序
本章介绍如何在 Unity 中优化应用程序。

章 3 分析你的应用程序
本章介绍如何分析你的应用程序。

章 4 优化列表
本章列出了 Unity 应用程序的多种优化。

章 5 实时 3D 美术最佳实践：几何图形
本章重点介绍 3D 资源的一些关键几何优化方法。通过几何优化，游戏会更加高效，同时实现让游戏在移动平台上表现更好的总体目标。

章 6 实时 3D 美术最佳实践：纹理
本章介绍了多种纹理优化方法，使你的游戏更美观，运行更顺利。

章 7 实时 3D 美术最佳实践：材质和着色器
本章介绍了多种纹理和着色器优化方法，使你的游戏运行更高效，画面更美观。

章 8 高级图形技术
本章列出了你可以利用的多种高级图形技术。

章 9 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现反射的一些区别。

章 10 高级 VR 图形技巧
本章介绍了用于提高虚拟现实应用程序图形性能的各种技术。

章 11 Vulkan
本章将介绍 Vulkan 及其启用方法。

章 12 Arm Mobile Studio
本章将介绍图形分析器和 Streamline 工具。

附录 A 修订
本附录将介绍本书发行版本之间的变化。

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。你可输入带下划线的文本用于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字。

<and>
封闭汇编器语法的可替换项，这些项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。

其他读物

This document contains information that is specific to this product.See the following documents for other relevant information.See *https://developer.arm.com* for additional Arm documents.

Arm 出版物

无。

开发人员资源：

- *https://developer.arm.com/graphics/*.

其他出版物

- *OpenGL ES 1.1 规范*，位于 *http://www.khronos.org*。
- *OpenGL ES 2.0 规范*，位于 *http://www.khronos.org*。
- *OpenGL ES 3.0 规范*，位于 *http://www.khronos.org*。
- *OpenGL ES 3.1 规范*，位于 *http://www.khronos.org*。
- *Unity 脚本参考*，位于 *Unity*。
- *GPU 精粹：实时图形编程的技术、技巧和技艺* Randima Fernando（丛书编辑）著。
- *GPU 专家：高级渲染技巧* Wolfgang Engel（编辑）著。
- *https://developer.oculus.com/osig*.

相关参考资料

反馈

关于本产品的反馈

如果你对本产品有任何意见或建议，请与你的供应商联系并提供以下信息：

- 产品名称。
- 产品修订号或版本。
- 尽可能详细的提供信息加以解释。注明症状和诊断程序（如果有）。

相关参考资料

关于内容的反馈

如果您对内容有任何意见，请发送电子邮件至 *errata@arm.com* 。提供：

- 主题： *Arm 开发者指南 优化移动游戏图形*。
- 文档号：100140_0401_00_zh。
- 您有任何反馈意见的相关页码（如适用）。
- 您的意见的简单说明。

Arm 还欢迎您对需要增加和改进之处提出概括性的建议。

说明

Arm 仅在 Adobe Acrobat 和 Acrobat Reader 中进行 PDF 的测试，与任何其他 PDF 阅读器一起使用时，不能保证所示文档的质量。

相关参考资料

第 1 章
简介

本章介绍了《 Arm 开发者指南：优化移动游戏图形》。

它包含以下部分:

1.1 关于 Unity

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

本书旨在帮助你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。描述了可用于提高应用程序性能的技术和最佳实践。

说明

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 Arm® Mali™ GPU

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

Bifrost GPU

Bifrost GPU 具有统一的着色器核心，可执行顶点、片段、几何体、细分曲面和计算处理。它们用于配备 Vulkan、OpenGL ES 1.1 至 OpenGL ES 3.2 和 OpenCL 1.2 Full Profile 的图形与计算应用程序。

Midgard GPU

Bifrost GPU 具有统一的着色器核心，可执行顶点、片段、几何体、细分曲面和计算处理。它们用于配备 Vulkan、OpenGL ES 1.1 至 OpenGL ES 3.2 和 OpenCL 1.2 Full Profile 的图形与计算应用程序。

Utgard GPU

Utgard GPU 系列拥有一个顶点处理器和一个或多个片段处理器。它们用于配备 OpenGL ES 1.1 和 2.0 的仅图形应用程序。

1.3 关于优化

图形可以使画面更美观。优化是指用最少的计算量使画面更美观。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优质画面内容。

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® -A57 MP4 处理器和 Arm Mali -T760 MP8 GPU。

第 2 章
优化应用程序

本章介绍如何在 Unity 中优化应用程序。

它包含以下部分:

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。你可以利用优化提高游戏的帧率，带来更好、更流畅的体验。

若要优化你的代码，请使用优化流程。此流程为迭代流程，可指导你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，找到性能瓶颈。
3. 确定要运用的相关优化。
4. 验证优化是否达到预期。
5. 如果性能未达到预期，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你需要的性能，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，以便你可以找出并确认性能问题的根源。
3. 比如，游戏的问题在于渲染过多的顶点。
4. 减少代码中的顶点数量。
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，你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，可以确保你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。部分选项将导致运算量增加，会对游戏性能产生不利影响。

下图显示了**检视器**中的质量设置：

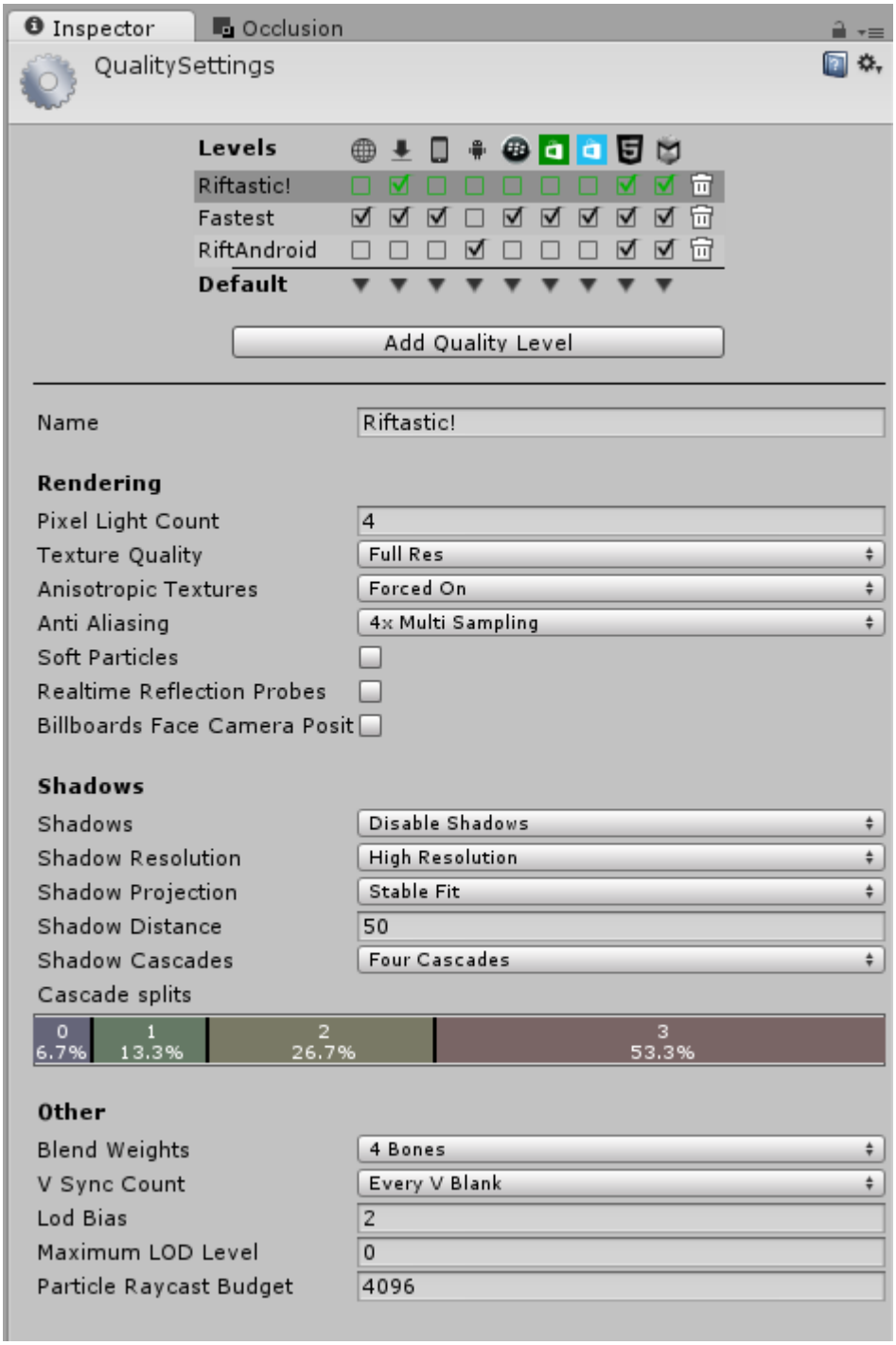

图 2-1 质量设置

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 可能正在大量处理渲染复杂图形效果的信息。你可以降低渲染图形效果的复杂度，例

如阴影和光照。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，有许多选项可以对游戏性能产生巨大影响：

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大量的计算。大部分游戏即使使用极少量的动态实时光源，对图像质量的影响也微乎其微。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

纹理质量会给 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画面质量产生不良影响，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 Mipmap 贴图。Mipmap 贴图可降低计算和带宽要求，同时不会影响图像质量。

反锯齿

反锯齿是一项边缘平滑技术，该技术会混合三角形边周围的像素。这显著提高了游戏的画面质量。有多种反锯齿方式，但是本例中采用的是*多重采样反锯齿* (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你不能使用 MSAA。如果软粒子并非十分重要，则尽量不要在游戏中使用。

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。如果阴影对于你的游戏非常重要，请考虑使用简单的动态阴影技术，例如投影纹理。

实时反射探针

实时反射探针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镜头进行单独渲染。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 Mipmap 贴图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。尽可能使用最低分辨率的立方体贴图来达到你需要的质量。

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**选项会逐帧渲染立方体贴图。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**选项让你能控制立方体贴图更新的时间。使用此选项时，你可以指定更新条件，以此限制运行时的资源使用。

第 3 章
分析你的应用程序

本章介绍如何分析你的应用程序。

它包含以下部分:

3.1 关于分析

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

你可以使用下列工具分析 Unity 应用程序：

- Unity 分析器。
- Unity Frame Debugger。
- 图形分析器。
- Streamline。

更多有关图形分析器的信息，请参阅 *12.2 关于图形分析器* on page 12-251。

有关 Streamline 的更多信息，请参阅 *12.3 关于 Streamline* on page 12-259。

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，帮助你查找游戏中的瓶颈。

如果你点击一个图表，就会看到垂直切片，并同时选择某个单帧。你可以在屏幕底部的显示面板中读取该帧的信息。如果你在没有修改所选帧的情况下点击另一图表，面板将显示你已选择的分析器的数据。

下图显示了 Unity 分析器：

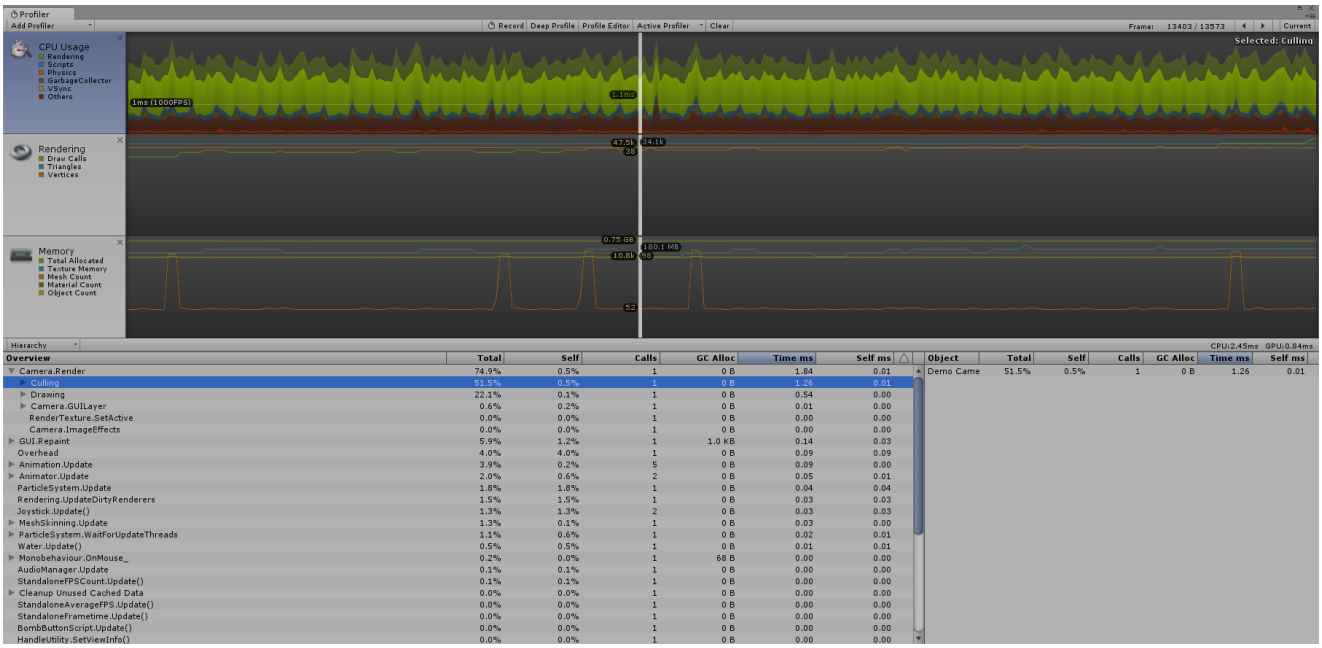

图 3-1 Unity 分析器

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。如果你在图表上选择了某帧，则面板将显示对该帧影响最大的函数的执行时间、调用次数或内存分配。
请重点关注耗费过多时间或分配过多内存的函数。

说明

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，因此你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，不恰当地运用纹理图谱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 分析器可让你采用 `Profiler.BeginSample()` 和 `Profiler.EndSample()` 方法。你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，你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消耗。

```
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 分析器细项

3.3 Unity Frame Debugger

Frame Debugger 是一款分析工具，你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：

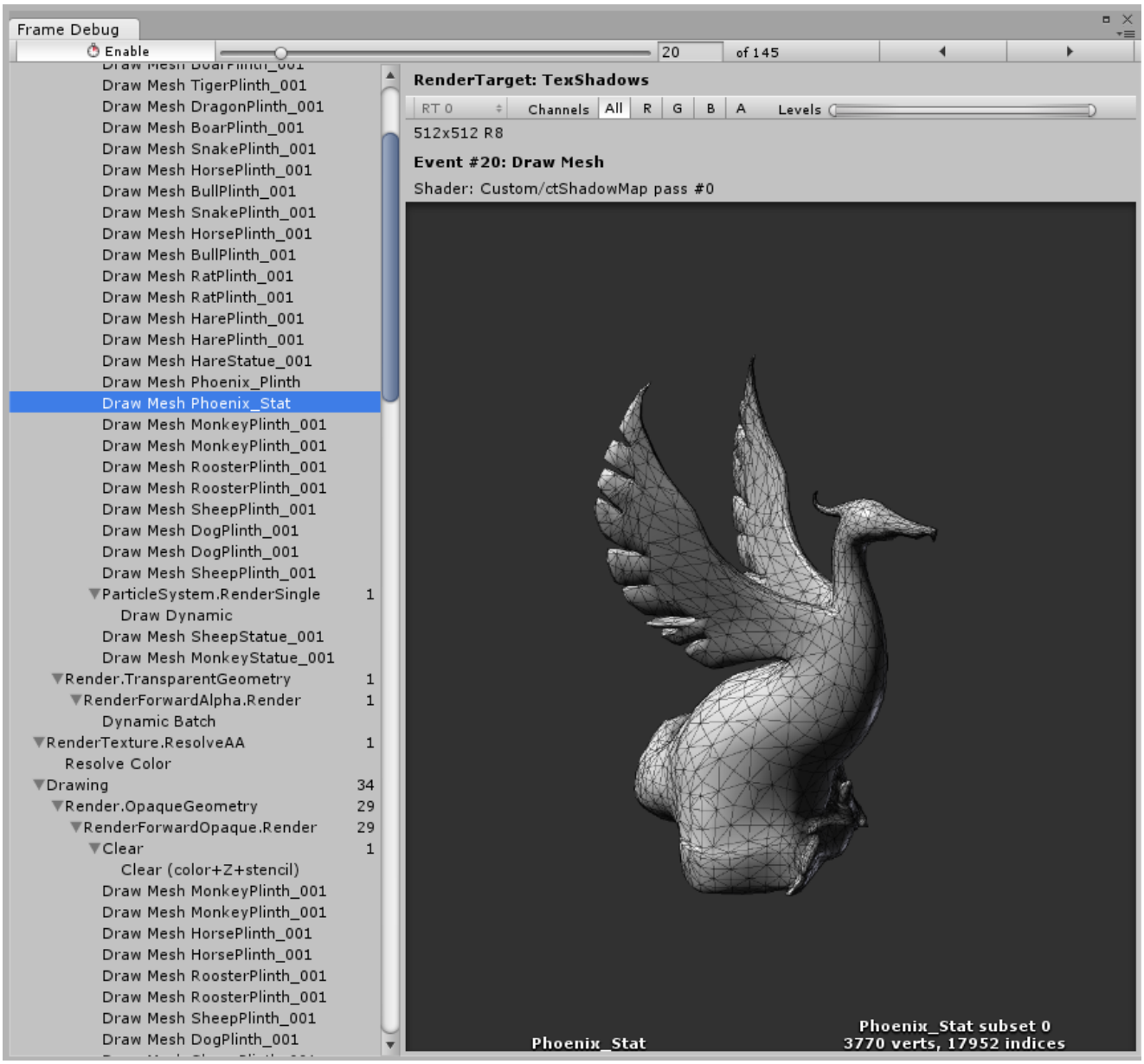

图 3-3 Frame Debugger

如果镜头渲染到所选绘制调用的目标，你可以在**游戏视图**中查看被渲染纹理的视觉效果。

下图显示了**游戏视图**中的 Phoenix 对象：

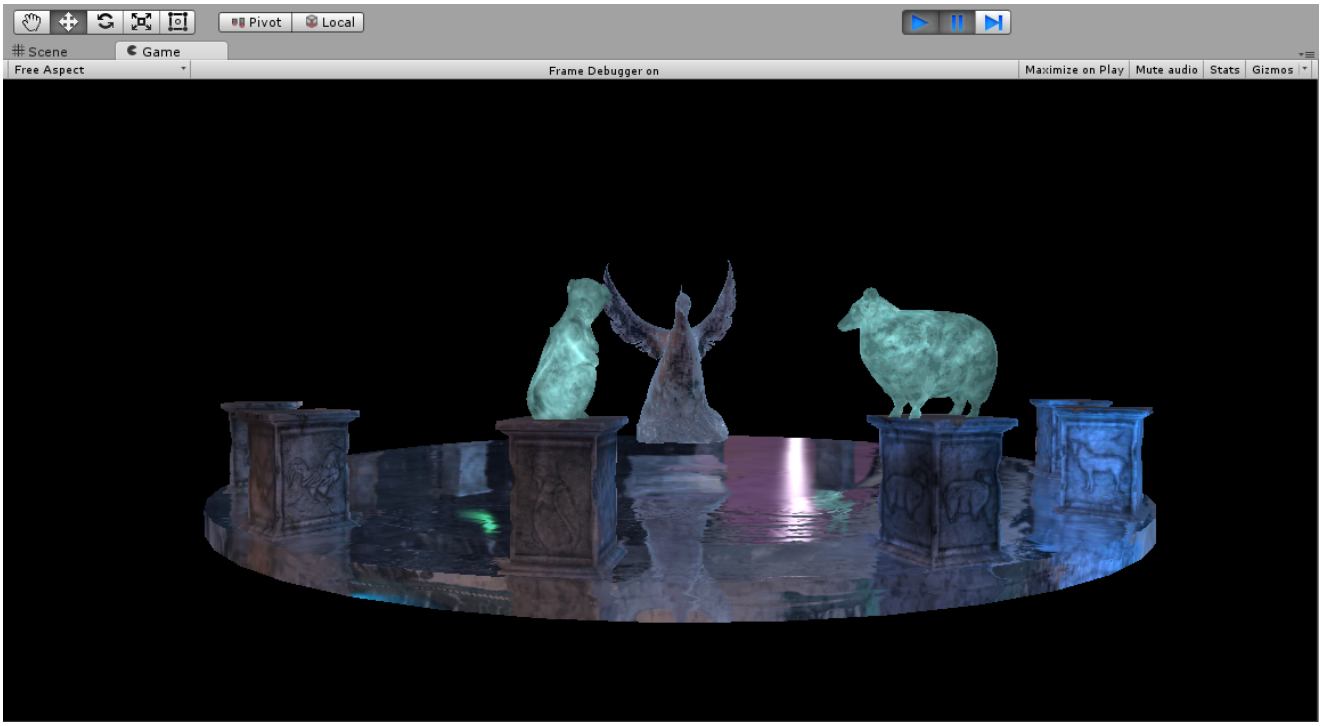

图 3-4 Frame Debugger 游戏视图

如需查看说明，了解如何利用 Frame Debugger 来优化对象渲染顺序以提升游戏性能，请参阅 *4.2.9 指定渲染顺序* on page 4-52。

第 4 章
优化列表

本章列出了 Unity 应用程序的多种优化。

它包含以下部分:

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。
- 你无法提供附加参数。

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。你可以稍后再次调用函数，它将从之前中断的位置恢复运行。
你可以通过 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

如果你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);
    }
}
```

—————— 说明 ——————

此技术还可以用来不定期地生成敌人。利用协程中的无限循环大量产生敌人，并生成随机数字。将随机数字传递到 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，如果你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"))
{
    [...]
}
```

你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))
{
    [...]
}
```

说明

你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

你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，你可以增加或减小此步长。
增加时间步长会减小应用处理器上的负载，但是会降低物理计算的准确性。
你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：

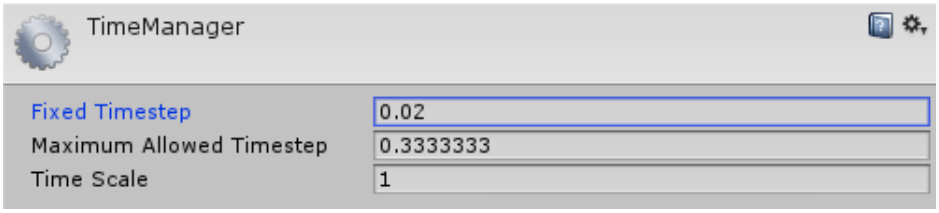

图 4-1 固定时间步长设置

删除空回调函数

如果你的代码包含 `Awake()`、`Start()` 或 `Update()` 等函数的空定义，则将其删除。这会产生不必要的消耗，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);
}
```

说明

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 Compare tag

使用对象池

如果你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。
如果你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，你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。如果你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。

—————— 说明 ——————

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消耗。

—————— 说明 ——————

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

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你的脚本。

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

使用 sqrMagnitude 比较向量幅度

如果你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

如果你事先知道数组大小，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，你可以为 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：

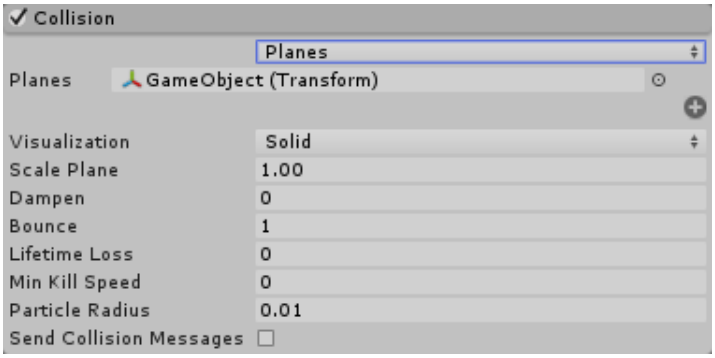

图 4-4 碰撞设置

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。

你可以将盒子、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。这可使你获得相似的结果，同时还可大幅降低计算消耗。

4.2 GPU 优化

本节列出了 GPU 优化。

本部分包含以下子部分:

4.2.1 几项 GPU 优化

下面列出了几项 GPU 相关的优化:

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。
静态批处理可用于大量顶点组成的对象，但是经过批处理的对象在渲染过程中不得移动、旋转或放大。
若要使 Unity 能够集合要进行静态批处理的对象，请在“检视器”中将它们标记为静态。
下图显示了静态批处理设置:

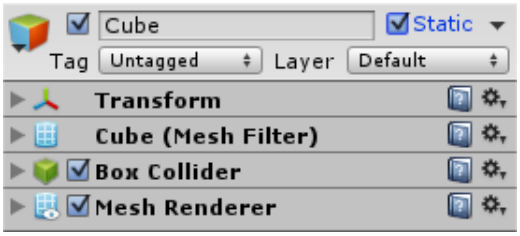

图 4-5 静态批设置

使用 4x MSAA

Arm Mali GPU 能够以极低的计算开销执行 4x 多重采样反锯齿。
你可以在 Unity 质量设置中启用 4x MSAA。
下图显示了 MSAA 设置:

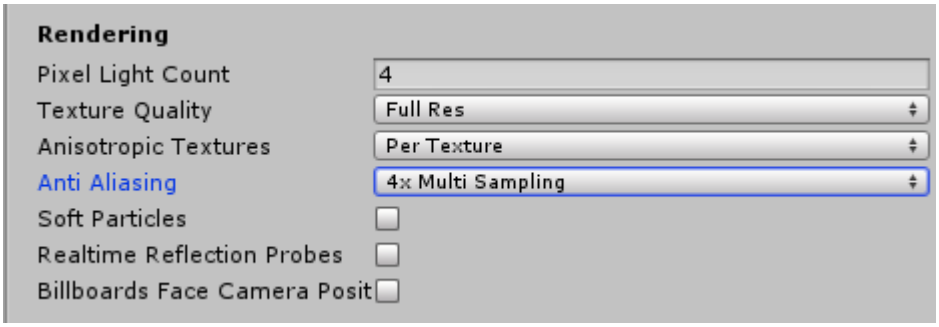

图 4-6 MSAA 设置

使用细节层次

Unity 引擎可以使用*细节层次* (LOD) 技术对于同一物体的渲染根据距离镜头的远近去调整网格。
当对象更接近镜头时，几何结构更为清晰。随着对象远离镜头，细节层次降低。当处于最远距离时，你可以使用平面公告板。
你必须恰当地设置 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，你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 可以根据对象的显示大小计算混合因子，并将它传递到着色器以进行混合。你必须在着色器中采用几何体混合。
下图显示了 LOD 组设置：

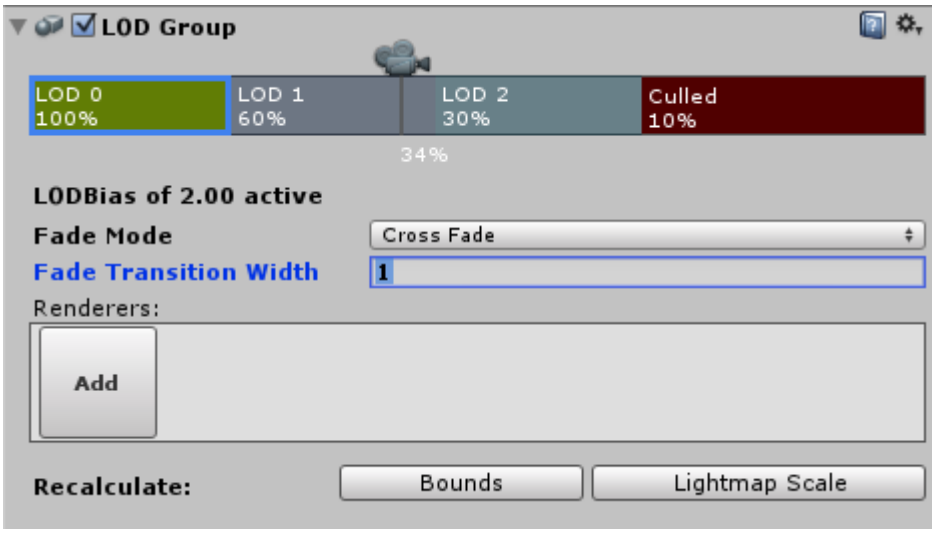

图 4-7 LOD 组设置

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

实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。

这意味着你会丧失完全动态光照环境的灵活性，但你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**实时**。
- 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，则光照贴图已经创建正确并运作正常。

下图显示了**光照**选项卡中的强度光照贴图。

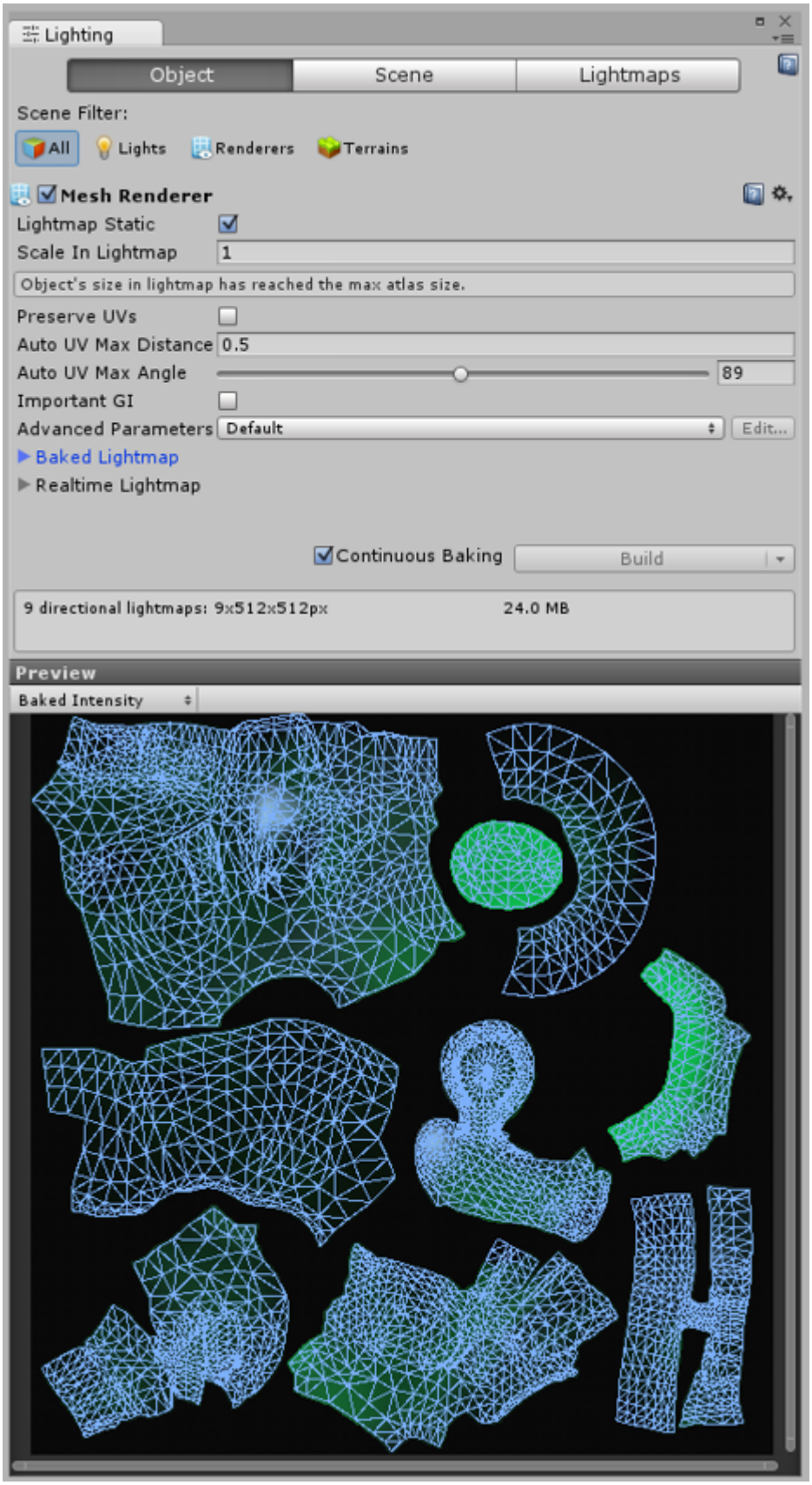

图 4-8 强度光照贴图

下图显示了来自洞穴尽头绿色光源的光照在编辑器中的显示画面。此光照使用静态光照贴图生成。

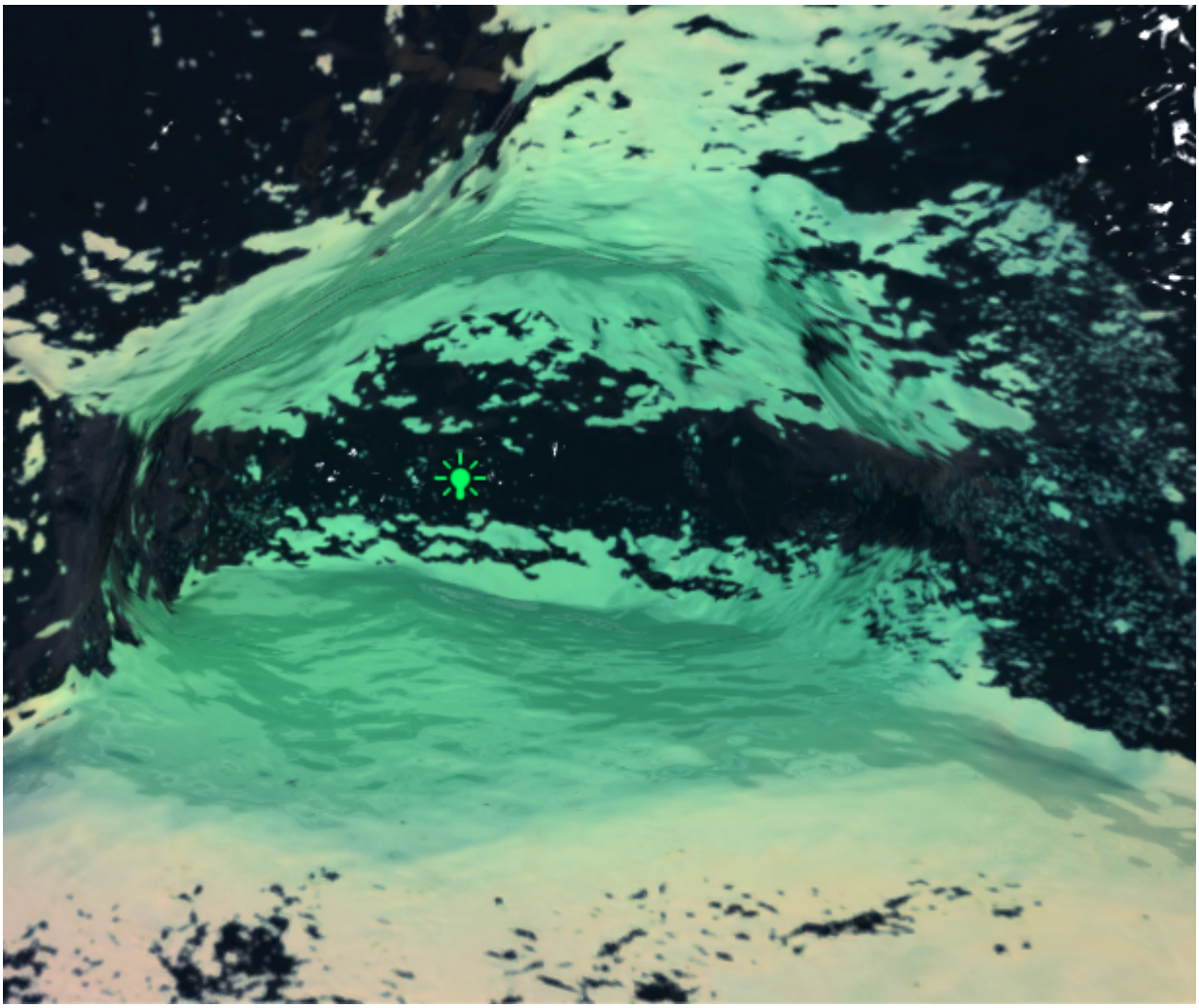

图 4-9 添加灯光以烘焙静态光照贴图

下图显示了冰穴演示中静态光照贴图的结果。

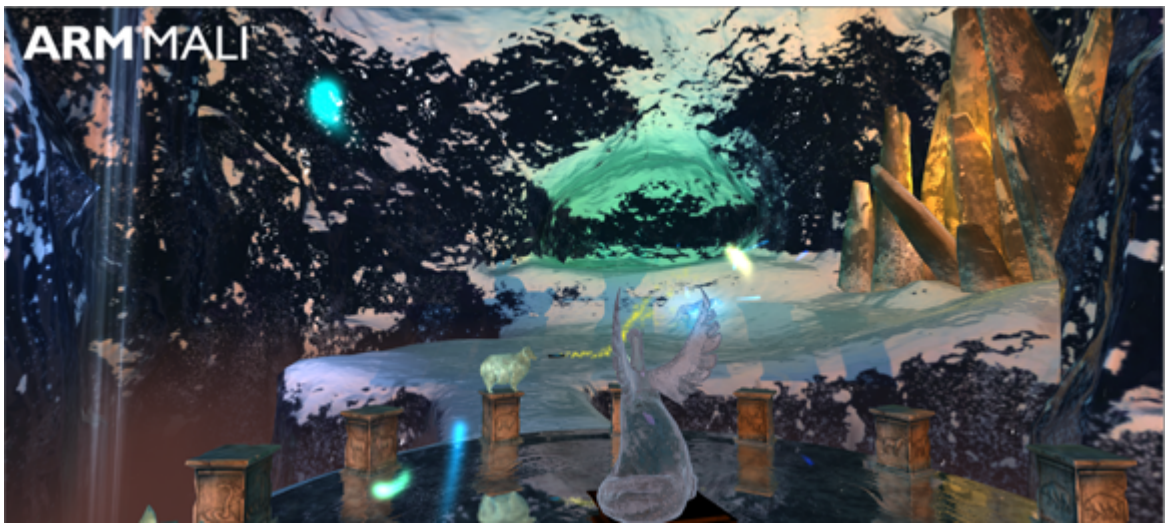

图 4-10 采用光照贴图的洞穴

构建光照贴图

若要为光照贴图准备对象，你必须做到以下几条：

- 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**必须设置为已烘焙。

——————— **说明** ———————

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你的游戏。

未标记为静态的对象不会置于光照贴图中。选择渲染器可以为你提供众多设置，你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**按钮可更改与你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。你可以在此选项卡中启用和禁用**预计算实时 GI**和**已烘焙 GI** 功能。

环境光照部分中提供了诸多选项，你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**定义反射对象之间相互反射的次数，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。如果你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。这能够使你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。要选择光照贴图快照方框，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：

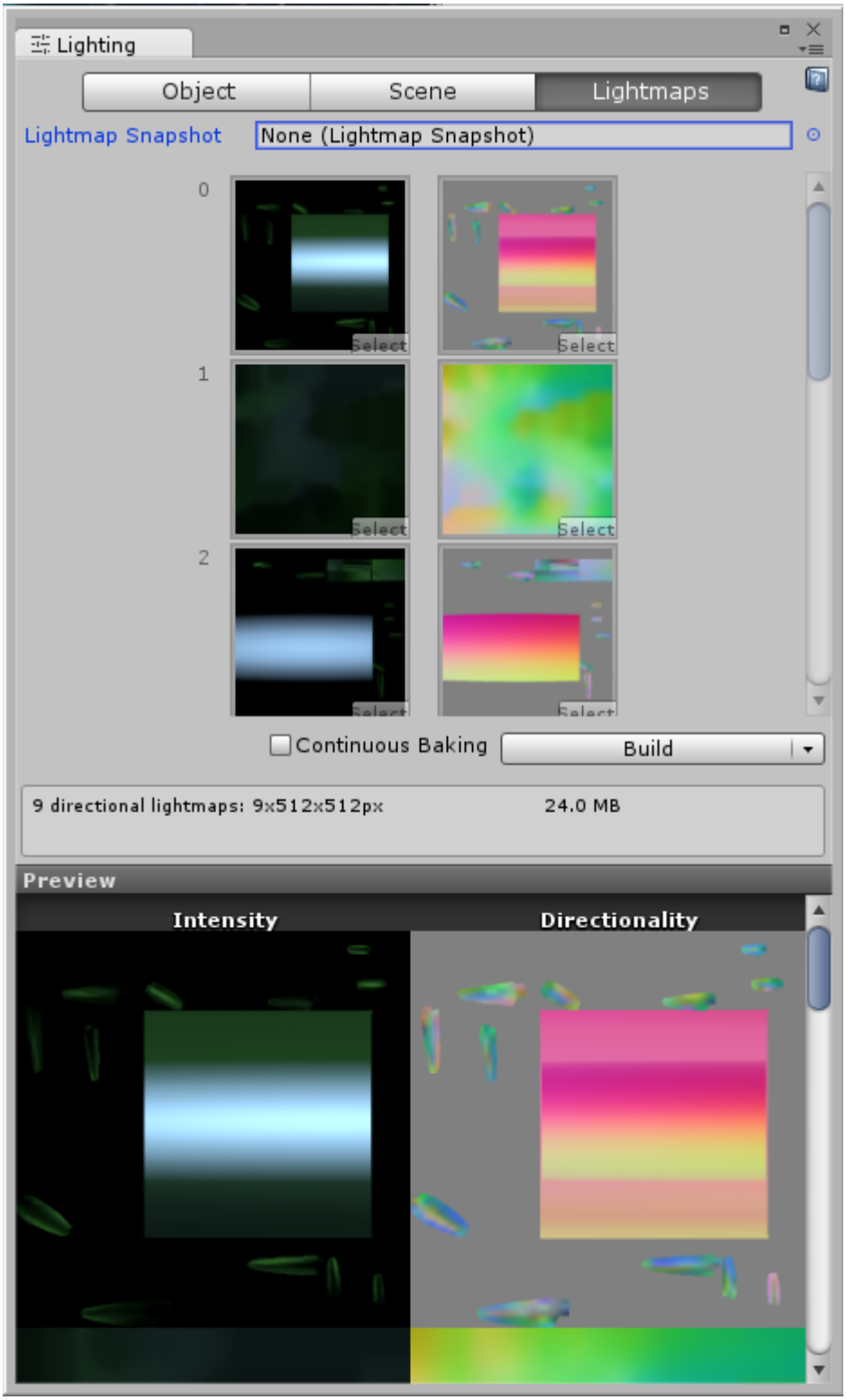

图 4-12 光照选项卡中的光照贴图

使用定向光照贴图

如果你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。这能够使你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。

说明

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

你可以使用灯光探测器为采用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的设置：

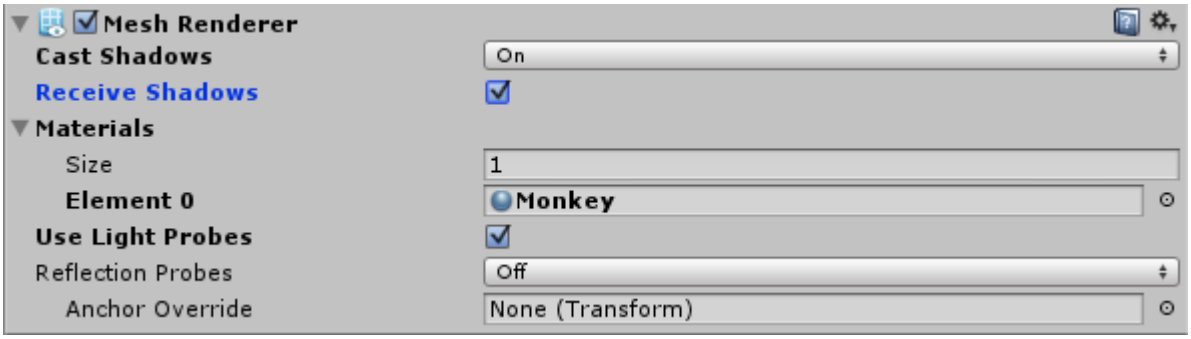

图 4-13 灯光探测器设置

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：

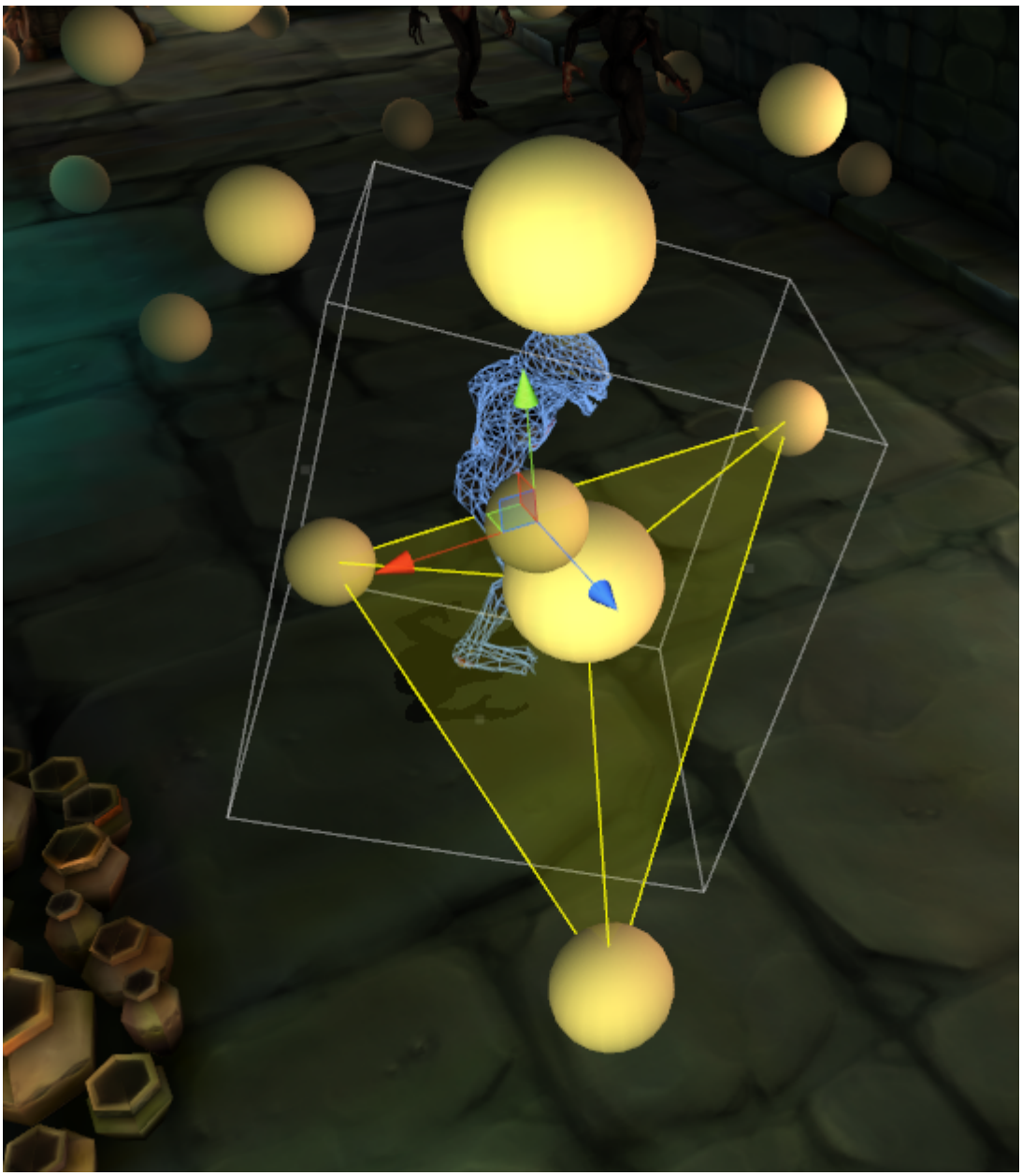

图 4-14 光照探针

使用的探测器越多，光照就越准确。通常，你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，你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，并通过使用灯光探测器选项标记你希望会受其影响的网格。

下图显示了多个灯光探测器：

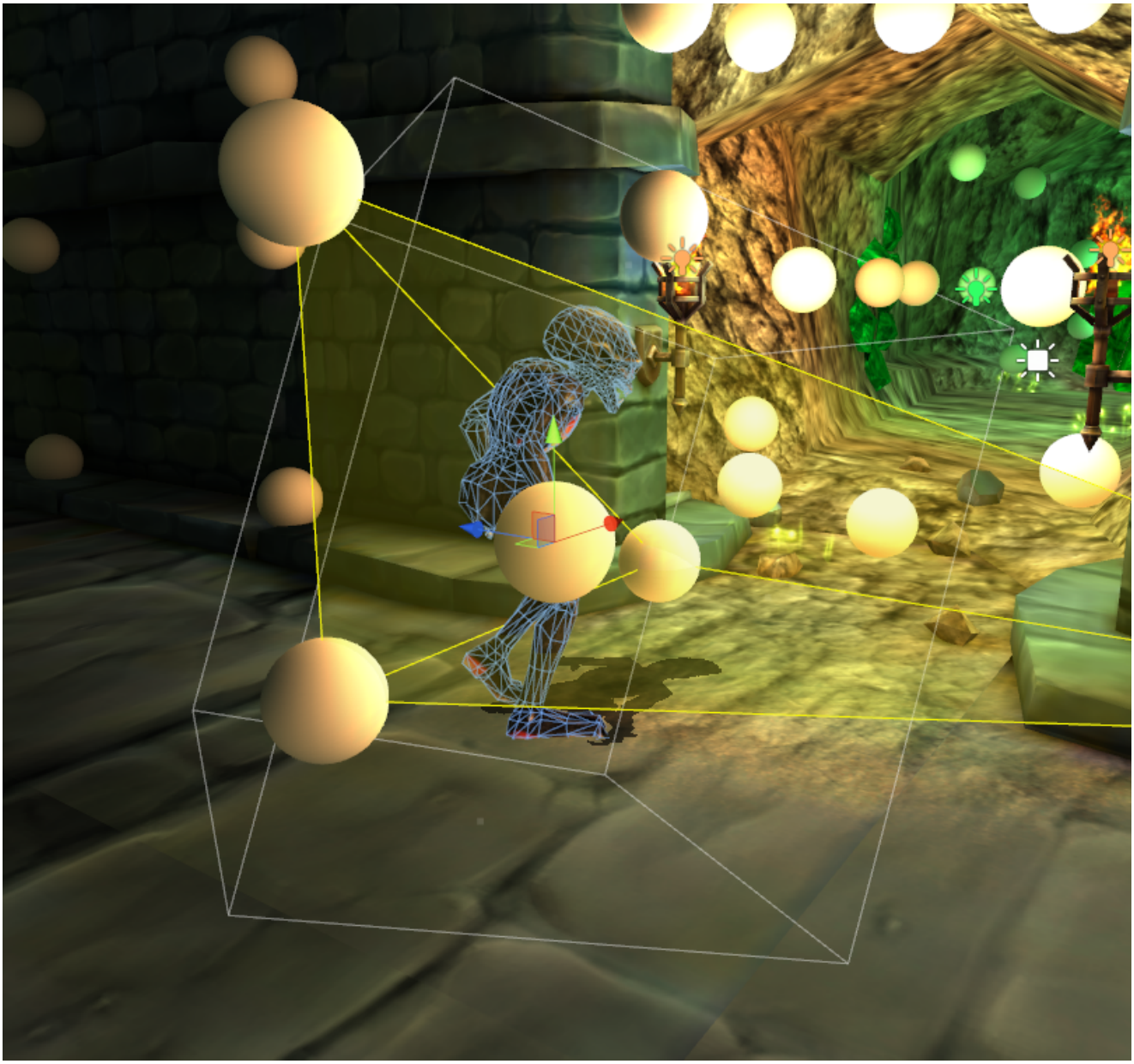

图 4-15 多个光照探针

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。这可使你微调文件大小与质量，以在两者间取得平衡。
- 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。你可以从中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，如距离镜头较远的对象。为显示细节度较高的纹理选择较小的块大小，如距离镜头较近的对象。

说明

- 如果你的设备支持 ASTC，请用它来压缩 3D 内容中的纹理。如果你的设备不支持 ASTC，请尝试使用 ETC2。
- 你必须区分 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。

下图显示了适合不同纹理压缩格式的块大小：

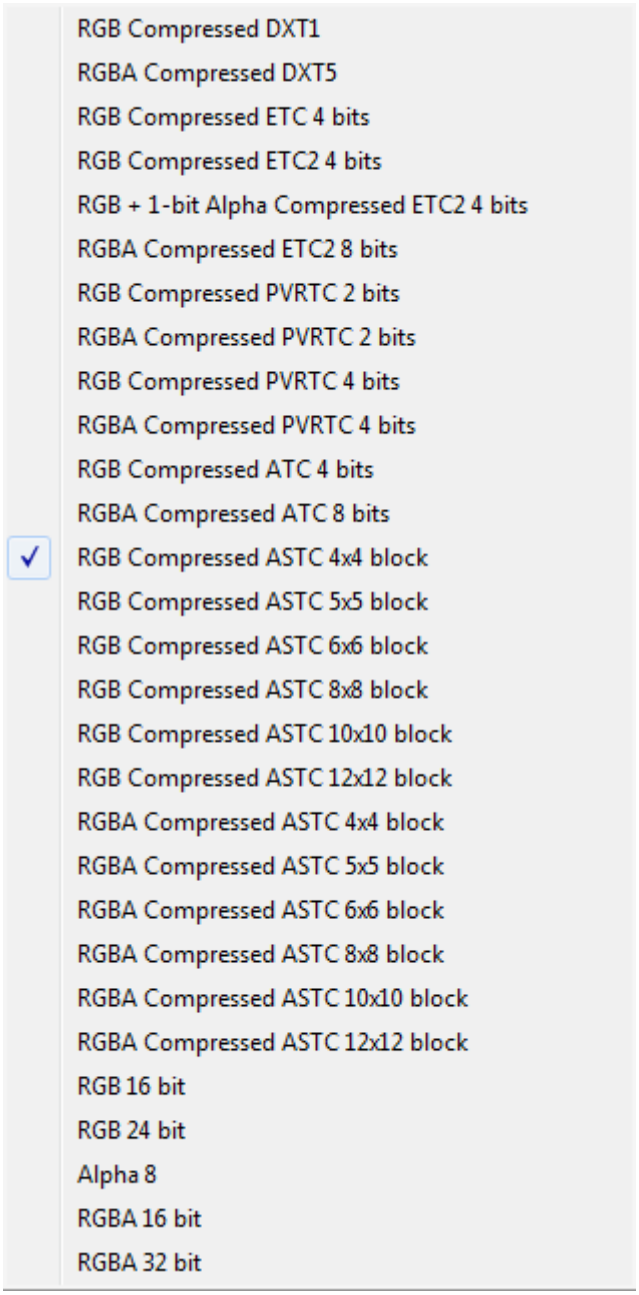

图 4-17 纹理压缩块大小

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：

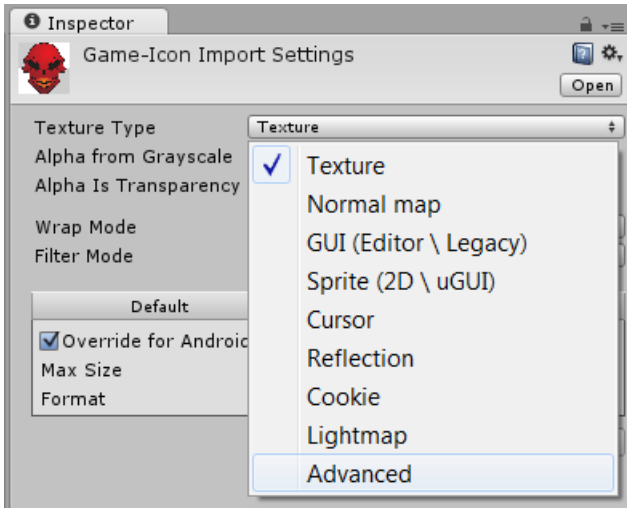

图 4-18 纹理设置

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

在项目层级中查找纹理并使其显示在**检视器**中。Unity 通常会以**纹理**类型导入你的纹理。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 **Mipmap 贴图**已被禁用。若要包含透明度，你需要 Alpha 通道。为此，请选择 **Alpha 透明**方框和**覆盖 Android** 方框。

下图显示了高级纹理设置：

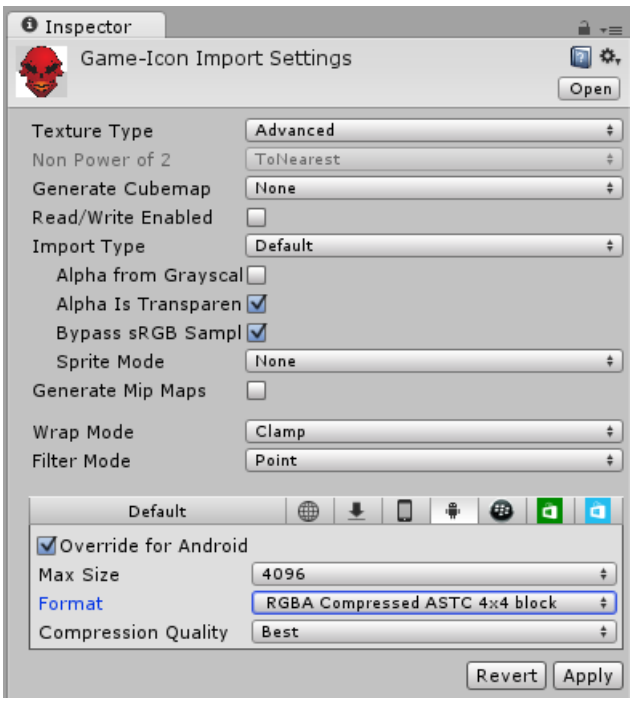

图 4-19 高级纹理设置

有一个用于选择格式和块大小的选项。RGBA 包含 Alpha 通道，4x4 是你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画面质量，并避免在压缩时出现不必要的纹理数据。

下表显示了，针对一幅大小为 4 MB，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 Mipmap 贴图

Mipmap 贴图是一项能够同时提高游戏画质和性能的纹理技术。

Mipmap 贴图是不同大小的纹理的预计算版本。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 Mipmap 贴图，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**检视器**中启用**生成 Mipmap 贴图**选项。

下图显示了 Mipmap 贴图设置：

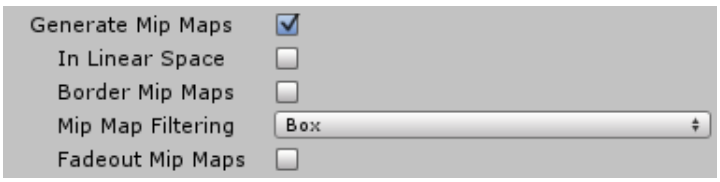

图 4-20 Mipmap 贴图设置

如果纹理没有 Mipmap 贴图层级，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，此过程中会丢失部分准确性，即便使用滤波器插值像素颜色。

如果纹理添加了 Mipmap 贴图层级，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。Mipmap 贴图的劣势在于它额外需要 33% 的内存来存储纹理数据。

Mipmap 贴图和 GUI 纹理

你通常不需要对 2D UI 中所用的纹理添加 Mipmap 贴图。UI 纹理通常不需要放大即可在屏幕上渲染，它们仅使用 Mipmap 贴图链中的第一层级。

若要更改此设置，请在**项目**窗口中选择某一纹理，然后进入**检视器**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 Mipmap 贴图**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。

你可以使用单个立方体贴图来渲染镜头的背景，以此绘制天空盒。

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. 选择镜头。
2. 确保**清除标记**设置为**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镜头背景的材质。

你使用的材质是包含全部所需信息的材质。使用**移动** > **天空盒**着色器创建材质，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。

当你完成此材质后，将其拖入天空盒组件中。你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 支持变换反馈，可计算实时阴影。

——— **说明** ———

冰穴演示中采用的是自定义阴影。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镜头的距离。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纹素数量，从而被动地提高阴影分辨率。

你可以使用阴影距离较小且分辨率较高的硬阴影。这会在距离镜头较远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，你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 带阴影的外星人

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，要尽量限制仅包含实时阴影的光源数量，并尽量使用光照贴图。

请考虑使用场景对象的网格渲染器组件。如果你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。

你可以在**质量设置**部分找到更多阴影设置，例如：

- **阴影分辨率**：你可通过选择来平衡质量和处理时间。
- **阴影距离**：可让你限制距离镜头较近的对象生成的阴影。
- **阴影级联**：可让你选择质量和处理时间之间的平衡值。你可以将它设置为零、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镜头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，当对象完全退出镜头视锥体时，Unity 将自动执行遮挡剔除，根据你的应用程序风格，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你的游戏风格。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 OnBecomeInvisible() 回调函数

如果你使用 MonoBehaviour.OnBecameVisible() 和 MonoBehaviour.OnBecameInvisible() 回调函数，则它们关联的游戏对象移入或移出镜头视锥体时 Unity 会通知你的脚本。然后你的应用程序便可作出相应的反应。

你可以使用 OnBecameVisible() 和 OnBecameInvisible() 优化渲染流程，例如用第二个镜头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。你只需要在反射可见时进行渲染，即在以下情况下渲染：

- 反射面位于镜头视锥体内。
- 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 OnBecomeVisible()
{
    enabled = true;
}

void OnBecomeInvisible()
{
    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，你可以添加其他条件：

例如，镜头必须位于反射面的空间内：

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

void OnBecomeInvisible()
{
    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。这意味着你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，你可以引导这一流程，因为你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就包含了 Early-Z。从你的视角而言，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，如果为透明物体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物体。这有助于在仅含不透明物体的场景中减少会导致过度绘制的因素。

按照从前往后的顺序渲染每一帧成本可能会很高昂，如果同一通道中还渲染透明物体，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* (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 为你提供着色器内或材质内的**排队**选项，让你能够指定渲染的顺序。这可以在着色器中设置，这样如果对象的材质使用该着色器，则所有此类对象都可以一起渲染。在这一渲染组中，渲染的顺序是随机的，但透明等情形除外。

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	这在所有不透明物体后绘制。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。你可以使用介于上表所示整数值之间的整数来指定其他队列。值越高，渲染越靠后。

例如，你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，洞穴占据了大部分屏幕画面，其着色器会消耗大量算力。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 Graphics Debugger 等其他工具查看帧缓冲区组成后，你会发现它已包含了渲染顺序优化。这些工具可帮助你查看渲染顺序。

若要打开 Unity **Frame Debugger**，请选择菜单选项 **窗口** > **Frame Debugger**。这很有用处，因为有些东西虽然在编辑器模式中看起来没问题，但在真正执行时却可能无法正常运行。例如，如果你具有仅运行时设置或你要渲染另一镜头下的纹理，就可能会出现这种情形。以播放模式启动演示并定位镜头后，你可以启用 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，将已渲染的洞穴部分挡住。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，如果有一组对象的着色器计算成本高昂，而且镜头可以绕着它们自由旋转，那么某些本来位于后方的对象有可能会在前方显示。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，你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。如果你的游戏受片段约束，并且你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。

说明

你可以在 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
如果你不动态修改纹理，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，你可以使用 `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你：

- 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，但是这取决于你的场景构成。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，使其成为你想要合并的所有网格的母网格，然后附加到一个脚本里。

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，你可以在导入设置的**装置**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
如果实时修改了模型，Unity 会在保留原始数据的同时在内存中另外保存一份复制的网格数据对其进行修改。
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。这样不需要另外保存一份待修改的副本以节省内存。

使用纹理图谱

你可以使用纹理图谱减少一组对象所需的绘制调用数量。

纹理图谱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。

下图显示了纹理图谱：

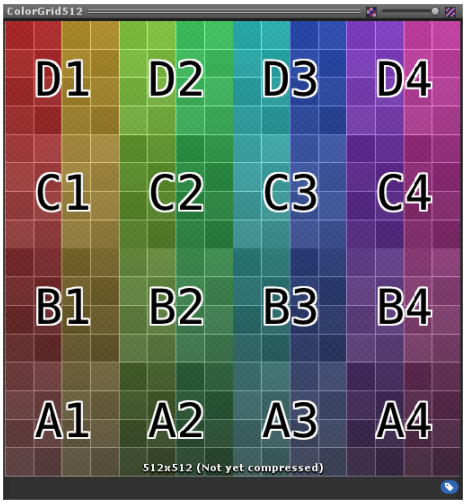

图 4-22 纹理图谱

4.4 使用 Mali™ 离线着色编译器优化

借助 Mali 离线着色编译器这款工具，你可以将顶点、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。

本部分包含以下子部分：

4.4.1 关于 Mali™ 离线着色编译器

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 离线着色编译器提供关于着色器在 Mali GPU 每一流水线中所需执行的周期数的信息。

生成的周期数针对一个特定 GPU。你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。这可确保你从工具中获得的统计信息切实可靠，符合你的常见用例情景。

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

其结果将显示在你的开发环境中。

说明

· Mali 离线着色编译器仅支持 OpenGL ES 着色器。
· 如果你的构建平台设为 Android，Unity 默认为生成 **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 启动你的应用程序时，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，你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 离线着色编译器进行编译。

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 离线着色编译器输出的数据：

```
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 离线着色编译器输出数据

4.4.3 分析统计信息

Mali 离线着色编译器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。

分别查看代码中采取/不采取分支的效果之后衡量得出最短和最长路径。这样可以估计执行周期数的最小和最大值。

对于算数操作，第一排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。

第二列和第三列用于加载/存储和纹理流水线。它们没有考虑缓存未命中情况，所以最好将这些数字乘以 1.5，从而获得更加真实的估计结果。

4.4.4 针对 Mali™ GPU 流水线进行优化

Mali 离线着色编译器提供每一流水线中使用的周期数。测量出的周期数最高的流水线其速度最慢。应将流水线中速度最慢的作为优化目标，优化你的着色器。

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。

下面列出了你可以降低算术流水线使用量的多种方法:

- 避免使用复杂算法，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-矢量或矩阵-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计算值。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。

如果发现应用处于加载/存储流水线 bound 的状态，可以尝试下列方法改善：
- 着色器中通过使用纹理而非缓冲区对象来读取数据。
- 使用算术运算计算数据。
- 压缩或减少统一缓冲区和变量的使用。

纹理流水线

访问纹理时会使用纹理流水线中的周期，也会使用内存带宽。使用大纹理可能有害，因为缓存未命中的几率更高，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 Mipmap 贴图
Mipmap 贴图可提高缓存命中率，因为它根据纹理坐标变化选用纹理的最佳分辨率。

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。每个压缩的块都包含至少一个以上的纹素，所以其访问变得更容易缓存。

避免三线性或各向异性过滤
三线性和各向异性过滤将增加获取纹素所需的操作数。若非绝对必要，请避免使用这些方法。

4.4.5 减少流水线周期的其他方法

还有多种方法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 离线着色编译器可指示你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。
- 变量。
- 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，这会加重加载/存储单元的负载并降低性能。要解决此问题，可尝试减少你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 发生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，你可以指定利用 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你的代码以提高并行化程度。
- 要求的统一变量寄存器数量减少，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的一个简单片段着色器变体的示例。该着色器使用 Mali 离线着色编译器编译了两次。

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.

                A       L/S     T       Total   Bound
Cycles:         47      11      10      68      A
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。使用的工作寄存器和统一变量寄存器数量变少，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

你可以使用切线空间法线贴图来提高模型的细致度，而不必增加几何细节。可以将切线空间法线贴图用于动画对象而不必修改它们，因为它们位于网格中每个三角形上。

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

你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，或者你可以在离线过程中通过代码来生成它们。

第 5 章
实时 3D 美术最佳实践：几何图形

本章重点介绍 3D 资源的一些关键几何优化方法。通过几何优化，游戏会更加高效，同时实现让游戏在移动平台上表现更好的总体目标。

它包含以下部分:

5.1 几何图形是什么

几何图形又叫做多边形网格，是一组构成 3D 对象形状的顶点、边和面的集合。3D 对象可以是汽车、武器、环境、人物或电子游戏中使用的任何类型的资源。

构成 3D 对象几何图形的三个元素如下：

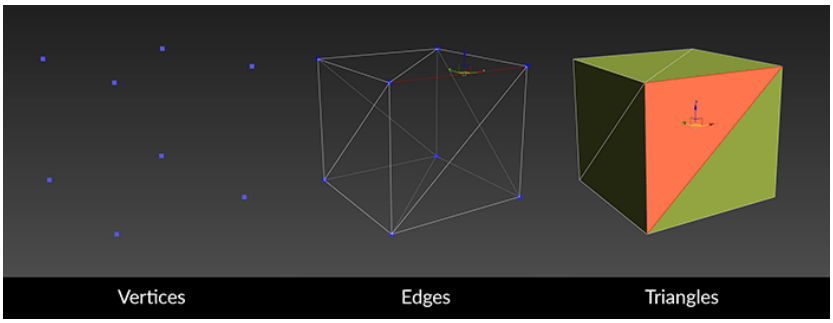

图 5-1 构成 3D 对象几何图形的三个元素

顶点

顶点是构成 3D 对象表面的点。

边

用直线连接两个顶点时，得到的就是一条边。

三角形

三角形由三个顶点组成，这三个顶点通过三条边相互连接，有时也称为多边形或面。Max、Maya 或 Blender 等 3D 软件中，通常使用四边形，在 3D 程序中四边形更容易修改和使用，但在屏幕上渲染时，所有多边形都显示为三角形。

5.2 三角形和多边形的使用

为了优化游戏性能，请始终跟踪屏幕上的三角形数量。

尽量使用更少的三角形，使各 3D 对象或模型达到平衡的预期画质，并实现一致的性能，这一点非常重要。

建议你尝试以下技巧：

- 降低三角形数量，提高性能。
  - 为移动平台创建内容时，牢记三角形计数是非常重要的。
  - 三角形越少，GPU 必须处理的顶点就越少。
  - 处理顶点的计算消耗很高。因此，处理的顶点数量越少，整体性能越好。
- 使用的三角形越少，就可以在越多的设备上发布游戏。而不仅限于拥有强大 GPU 的新款设备。

下图比较了两个 3D 对象。一个使用 584 个三角形，另一个使用 704 个三角形。两个对象在着色模式下看起来相同。因此，请移除任何与轮廓无关的边：

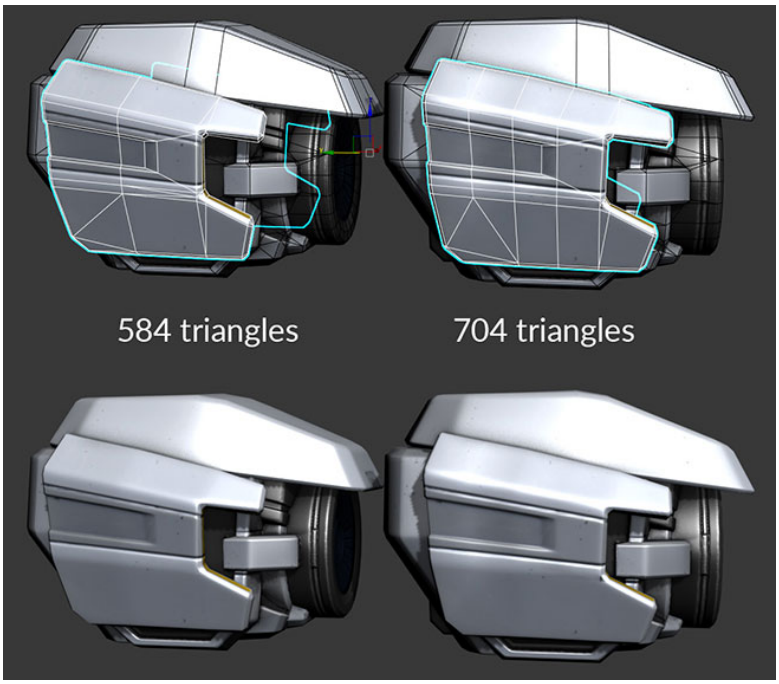

图 5-2 使用最小数量的三角形获得预期结果

在移动设备上，每个 3D 对象或网格最多可以使用 65535 个顶点。因此，使用的数量必须小于这个数。使用 Mali-400 等一些早期 GPU 的 Android 设备只能支持这个数量的顶点。这些设备不会渲染具有更多顶点的 3D 对象。

必须在尽可能多的目标设备上查看或测试游戏。仅仅在电脑屏幕上测试游戏并不能给你提供优化所需的信息。

请记住，移动设备的屏幕比普通电脑显示器要小。因此，使用大量三角形创建的细节在移动设备上有可能看不见。

在前景即离镜头更近的 3D 对象上使用更多的三角形，而在背景即更远的 3D 对象上使用更少的三角形。这一技巧对使用静态镜头*视角* (POV) 的游戏更有益处。

下图中，前景使用了高质量的 3D 模型，而远处低质量的 3D 模型直接被烘焙到了 2D 背景中：

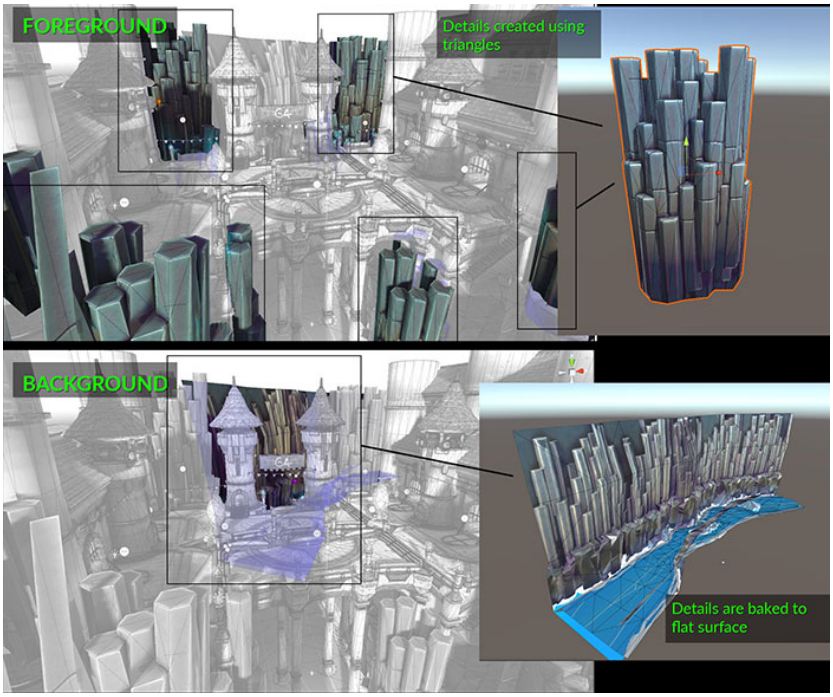

图 5-3 前景和背景对象上三角形用法示例

虽然没有规定 3D 对象可以使用的最大三角形数量，但是同时出现在屏幕上的 3D 对象越多，每个对象可以使用的三角形就越少。当然，屏幕上显示的 3D 对象越少，那么你可以使用的三角形就越多。

目标设备也很重要。较新的手机，例如最新的三星 Galaxy S 系列，相比旧手机能够处理更复杂的几何体。

以下示例显示了两个演示中的人物。电路 VR 演示只有一个机器人角色，因此可以使用多边形数更高的模型。而军队演示一帧中有数百名士兵，所以多边形数量必须很少。

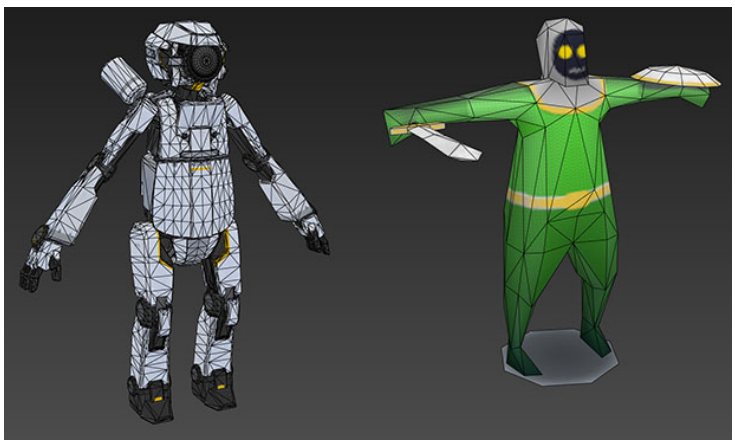

图 5-4 三角形数量差异示例

军队演示是一个 64 位的移动设备技术演示，它是在 Unity 中构建的，相机是静态的，有很多动画人物。每帧总共渲染了大约 21 万个三角形，从而使得演示速度约为 30 *帧每秒* (FPS)。

说明

要使用的三角形数量取决于正在创建的游戏类型和目标设备的规格。

以下示例显示了技术演示中使用的三角形总数：

图 5-5 示例场景中使用的三角形总数

场景中最大的对象是炮塔，大约有 3000 个三角形，它们占据了屏幕的很大一部分。

人物只使用了大约 360 个三角形。原因是他们太多了，要从远处才看的到。所以，从镜头视角来看，人物在屏幕上看起来很美观。

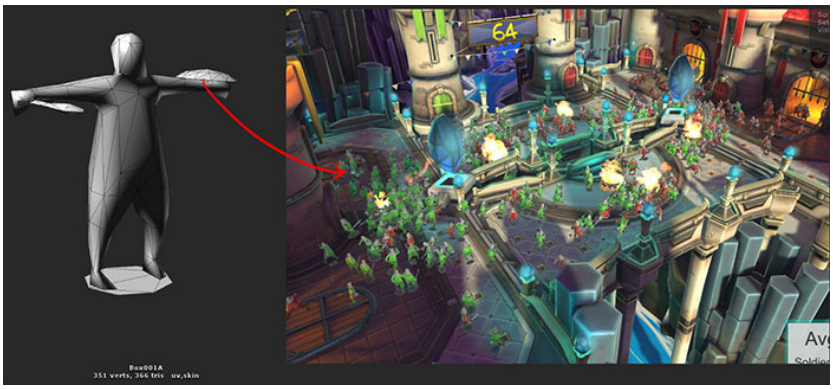

图 5-6 士兵上的小三角形数量

本部分包含以下子部分:

- *5.2.1 在重要区域分配三角形* on page 5-66.
- *5.2.2 为什么微型三角形不好* on page 5-67.
- *5.2.3 为什么细长的三角形不好* on page 5-69.

5.2.1 在重要区域分配三角形

在移动平台上，多边形和顶点都会消耗大量算力。要是不想浪费算力资源，应该将它们布置于能真正提升游戏画面质量的区域中。

由于大多数设备的屏幕尺寸较小，并且这些 3D 对象一般放置于游戏关卡中，因此它们的小三角形细节在游戏中大多并不可见。由此得出，应重点关注能影响到对象外轮廓的较大形状与部分，而非细节。

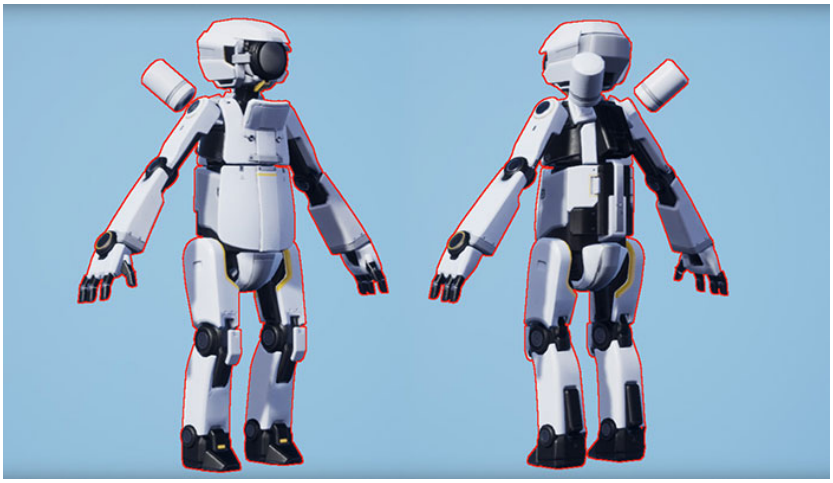

图 5-7 重点关注较大形状和外轮廓

对于不经常显示于屏幕的区域，使用更少数量的三角形。比如说，一辆汽车的底部，或是一个衣柜的背面。

为细节部分建模时，应避免布置密集的三角形网格。相反，精细部位应使用纹理和法线贴图。

说明

法线贴图是一种纹理贴图，它保存了每个像素表面的法线向量。

下面几张图展示了在同一个网格中，使用/未使用法线贴图的情形：

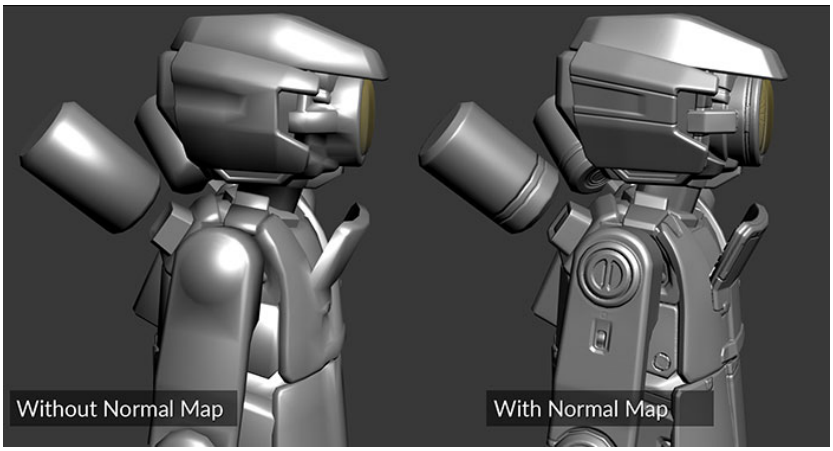

图 5-8 小细节应使用法线贴图和纹理

考虑删除镜头视角看不到的部位，比如对象的背面、底部或是其他部位。

但这是应该慎用的技巧，因为它会使这个场景难以重复使用。例如，你认为最终用户不会看见一张桌子的底部，便删除了那里的网格。但这么做也意味着，你将再也不能倒置这个模型，或是在别处重复使用它。

5.2.2 为什么微型三角形不好

微型三角形是微小的三角形，对对象或场景的最终显示效果没有多大贡献。

具有大量多边形的 3D 对象远离镜头移动时，就会出现微型三角形问题。微型三角形通常是指设备上大小在 1 到 10 像素之间的三角形。

微型三角形不太好用，因为 GPU 必须处理所有这些三角形。但是，由于它们太小而看不到，它们对最终图像的贡献不大。请记住，顶点的计算量非常大，并且同时在屏幕上显示许多小三角形意味着要处理更多的顶点。

以下两种方法会造成微型三角形问题：
- 细节太过精细，且由许多三角形组成。
- 对象具有许多距镜头较远的三角形。

下图显示了当 3D 对象靠近和远离镜头时使用的三角形数量。左图使用的三角形较少，而右边的示例使用法线贴图而非三角形来显示同等精细的细节：

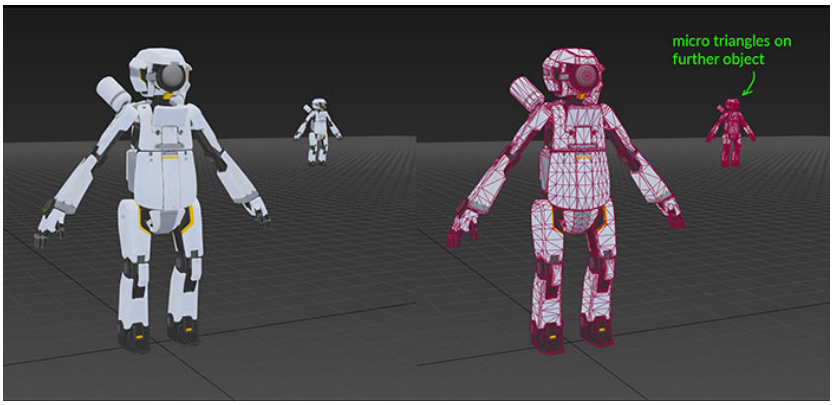

图 5-9 距离镜头较远的对象上的微型三角形

下图中，高亮区域内的大部分三角形都太小，在移动设备上看不见。因此对最终显示效果的贡献不大：

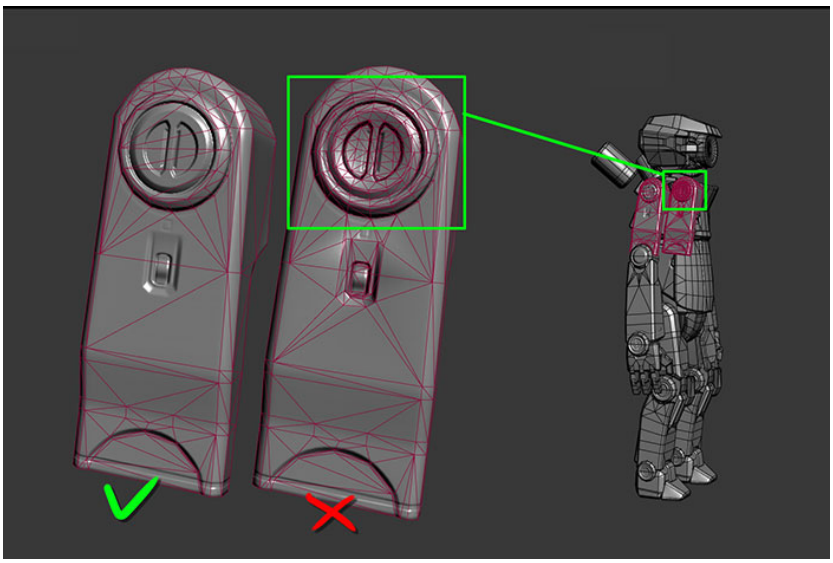

图 5-10 近距离内的微型三角形

如何最小化这个问题

你可以采取以下几个步骤来缓解这一问题：
- 对于会改变与镜头距离的对象，请使用*细节层次*(LOD)。使用正确的 LOD，可以简化距离较远的对象，并降低三角形的使用数量。
- 在背景对象上使用较少的三角形。
- 避免使用多边形创建更精细的细节。相反，应使用纹理和法线贴图。
- 合并所有太小而无法在屏幕上看到的顶点或三角形，或者对于最终画质没有太大价值的顶点或三角形。
- 尝试将区域内的三角形尺寸保持在 10 像素以上。

为什么尽量少用微型三角形很重要

尽量少用微型三角形很重要，有多个原因。分别为：

- GPU 必须处理所有的三角形和顶点。即使它们对屏幕上的最终场景效果贡献不大。
- 它们会将更多数据发送至 GPU 处理，由此不利于内存带宽。
- 需要的处理量直接影响移动设备的电池寿命。因此，GPU 必须处理的数据越少，电池寿命越长。

5.2.3 为什么细长的三角形不好

细长的三角形由那些在最终图像中渲染时，小于 10 个像素，并沿屏幕延伸的顶点构成。细长三角形之所以不好是因为它们要比普通三角形消耗更多的 GPU 算力。

下方截屏显示了使用细长三角形的示例：截屏突出显示了从远处看时支柱上的斜面。当然，近距离观察时，斜面并不是问题：

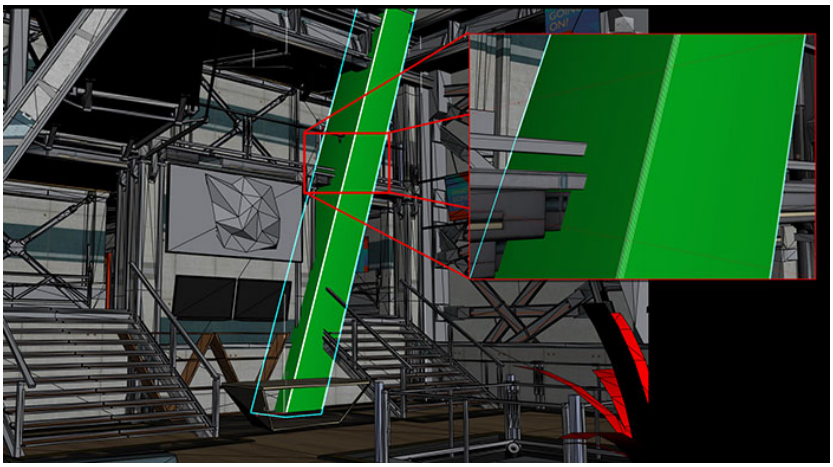

图 5-11 细长三角形示例

如何最小化这个问题

你可以采取以下几个步骤来缓解这一问题：

- 尽可能从所有对象中移除任何细长的三角形。虽然在某些情况下这是不可能的，但最好的解决方法是完全去除细长的三角形。
- 避免在有细长三角形的对象上使用有光泽的材质，因为这会导致对象闪烁。
- 在对象离屏幕较远时，使用*细节层次* (LOD) 并去除细长三角形。
- 从技术上讲，保持三角形接近等边更好。确保三角形内部相对于边而言具有更大的面积。
- 有关此问题的更多技术解释，请访问：*http://www.humus.name/index.php?page=News&ID=228*。

5.3 细节层次

细节层次 (LOD) 是一种随着对象远离观察者而降低网格复杂度的技术。

LOD 减少了必须处理的顶点数量。LOD 还避免了微三角形问题，并且通常使位于场景中更远处的对象看起来更美观。因此，ARM 建议你为与镜头的距离有显著变化的每个 3D 对象优化 LOD。

下图显示了如何利用 LOD 管理来降低 3D 模型的复杂度并且同时在模型远离镜头时仍然保持适当的细节层次：

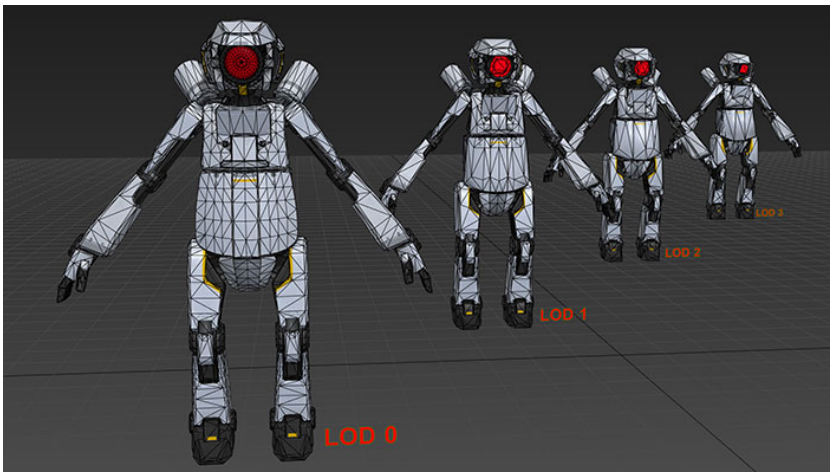

图 5-12 四级 LOD 示例

以下是关于你考虑如何优化 LOD 的使用时的一些其他提示：

- 减少 LOD 的三角形数量时，注意三角形对对象轮廓的影响。
- LOD 也可以应用于着色器复杂度。着色器和材质可以针对距离较远的 3D 对象进行优化。例如，通过减少使用的纹理数量进行优化。
- 区域越平坦，去除越多的多边形。
- 使用纹理映射作为纹理的 LOD。

何时不使用 LOD

不一定每种情况都适合使用 LOD。例如，请勿在镜头视图和对象都是静态的游戏中使用 LOD。也不要在对象使用的多边形数量本身就低的情况下使用 LOD。

LOD 会造成内存开销，并增加文件大小。所有 LOD 网格数据必须保存在内存中，方便实时利用。

下图显示了一个未使用 LOD 的静态场景示例。相反，你可以使用其他优化技巧，例如去除永远不会对玩家可见的多边形：

图 5-13 未使用 LOD 的静态场景示例

为什么使用 LOD?

对象离镜头越远，你所看到的对象细节就越少。从 20 米以外，很难看出一个有 200 个三角形的对象和该对象的另一个有 2,000 个三角形的版本有何区别。因此，没有必要使用大量多余并且不会增加场景价值的三角形。

其他一些主要优势包括：

- 由于需要处理的三角形更少，所以性能得到提升。
- LOD 有助于减轻微三角形引起的问题。

远距离的对象即使多边形数量不同，但看起来是一样的，如下图所示：

图 5-14 远距离模型 LOD 示例

为每个 LOD 使用适当数量的三角形

确定每个 LOD 中的三角形数量时，需要考虑如下关键点：

- 各个 LOD 层级之间值得按照 50% 的幅度递减三角形数量。
- 请勿在较低的 LOD 上密集使用三角形。这些只有当对象离得更远时才能被看到。
- 在 LOD 应当要被看到的相对于镜头的正确距离上，检查 LOD 的呈现状态。较低的 LOD 近距离看起来很糟糕，这没关系，因为它们本就不适合近景查看。

下图显示了当一个对象使用的多边形逐个层级减少 50% 时所呈现的外观：

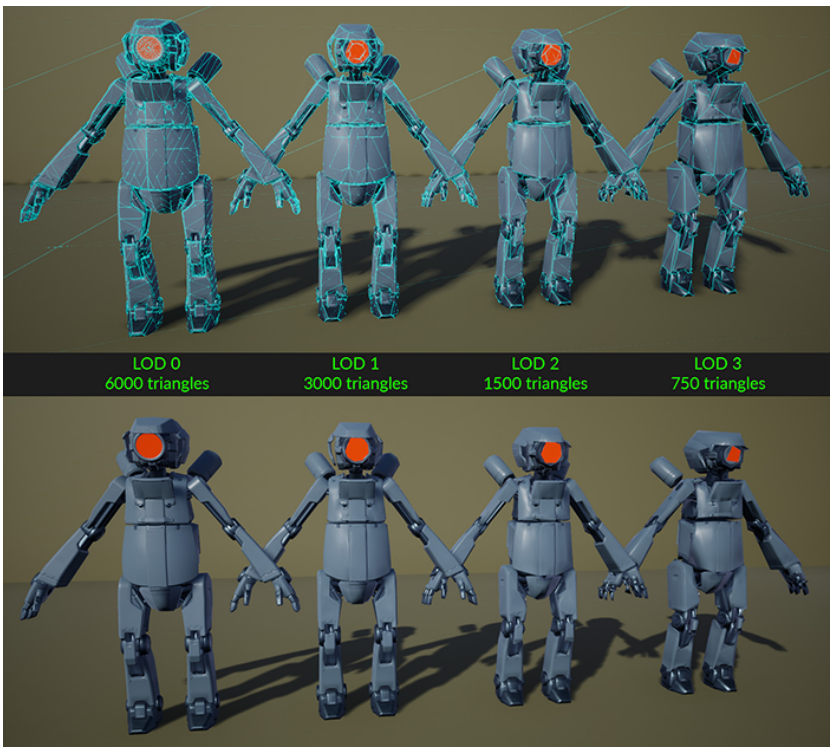

图 5-15 多边形数量 LOD 示例

下图示例显示了具有较少多边形的对象必须距离屏幕有多远。它还突出显示了较低 LOD 在距离屏幕太近时的糟糕画面效果：

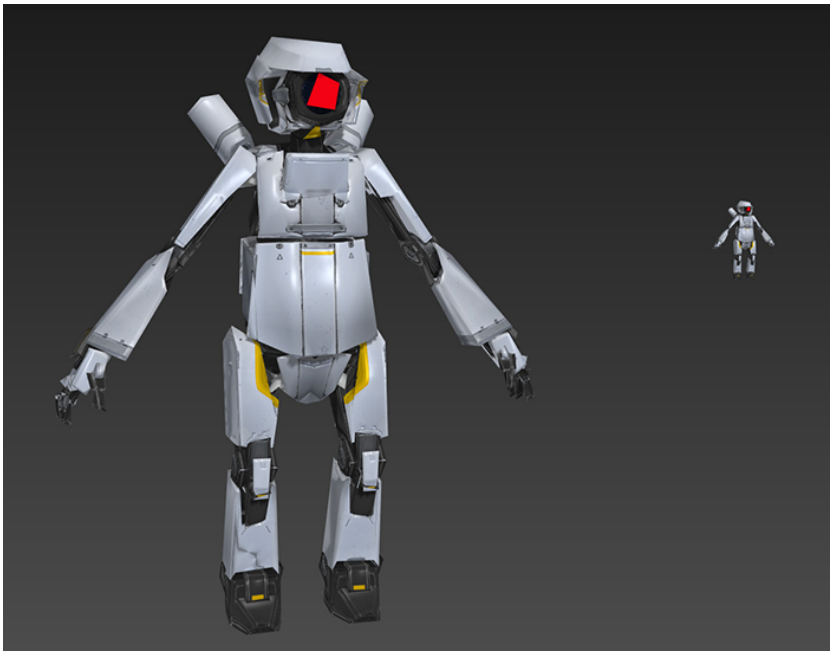

图 5-16 LOD 距离示例

LOD 三角形数量重要是因为，如果不充分减少低 LOD 对象的多边形数量，游戏的性能就会受到负面影响。这是因为 CPU 要处理的顶点多于必要的数量。

如果你过于激进地减少低 LOD 对象的多边形数量，那么这些项目的细节就会实时地突然出现或突然消失。这种细节突然出现或突然消失的效果对用户来说是明显的，可能会破坏他们在游戏中的沉浸感。

在一个对象上使用多少 LOD 层级是合理的?

关于一个对象能具有多少 LOD，没有固定的数字。这取决于对象的大小和重要性。例如，相比于树和易拉罐等小背景对象，动作类游戏中的角色或者竞速类游戏中的汽车能够更多地受益于使用更多的 LOD 层级。

如果使用的 LOD 层级太少，效果会是:

- 如果 LOD 层级之间的多边形减少幅度不够大，则性能增益降低。
- 如果多边形减少方面有太大的跳跃，则在 LOD 切换时出现的细节突然消失或突然出现效果会更明显。

如果使用的 LOD 层级太多，效果会是:

- 增加了 CPU 的负荷，因为在决定显示哪个 LOD 时需要进行高于必要数量的处理。
- 增加内存使用，因为你必须存储这些额外的网格，因此会增加文件大小。
- 然而，最大的成本在于创建和验证这些 LOD 模型所需要的时间。尤其是在美术师手工创作的情况下。

如何创建 LOD 网格

在 3D 软件中手动创建 LOD 网格时，剔除边缘环或减少 3D 对象上的顶点数。虽然这会给予美术师更多的控制权，但可能花费更长的时间。

自动创建 LOD 网格时，使用内置修改器或单独的 LOD 生成软件。作为内置修改器的一个示例，在 3DS Max 中，使用 Maya 中的 **ProOptimizer** 功能或 **Generate LOD Meshes** 功能。

在 Unity 中实现 LOD

关于如何在 Unity 中实现 LOD 的指导，请单击以下链接：*在 Unity 中实现细节层次 on page 10-224*。

5.4 其他几何最佳实践

你还可以尝试更多技巧和窍门，进一步优化游戏性能。

平滑组

使用平滑组或自定义顶点法线，定义边缘的硬度并改变模型的外观。当美术设计有意使用少量多边形时，平滑组有助于进行更好的着色。

平滑组会影响模型的 UV island，还会影响烘焙时法线贴图的质量，因此需要格外小心。更多信息，请参阅*纹理最佳实践章节* on page 6-76。

在 3D 模型上进行平滑处理时，必须从 3D 软件中导出并导入到引擎中。

下图为平滑处理应用于对象的示例：

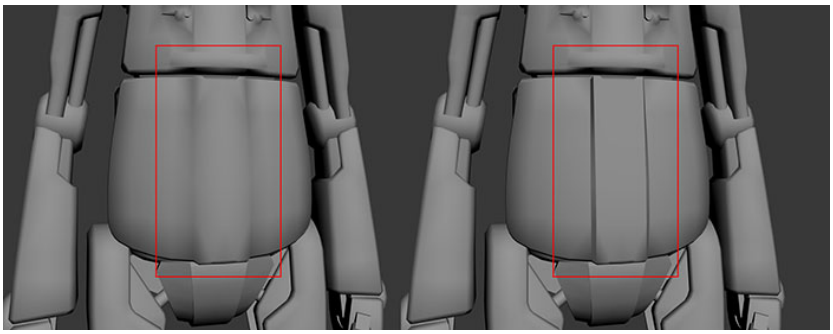

图 5-17 平滑组应用示例

网格拓扑

使用网格拓扑时，请记住以下几点：

- 创建 3D 资源时，请务必使用适度整洁的拓扑。
- 规整的拓扑对于变形或动画化的人物或其他对象至关重要。
- 不要沉迷于在 3D 模型上创建完美的拓扑。虽然不是所有对象都需要完美的边缘线条，但是要尽量保持模型整洁。
  - 玩家或最终用户看不到 3D 模型的线框。
  - 网格的纹理和材质对 3D 模型的外观影响更大。

下图为使用简单几何图形和拓扑的岩石悬崖网格线框示例。使用这种材质后，岩石悬崖看起来更加美观。这样，就不存在什么拓扑问题了：

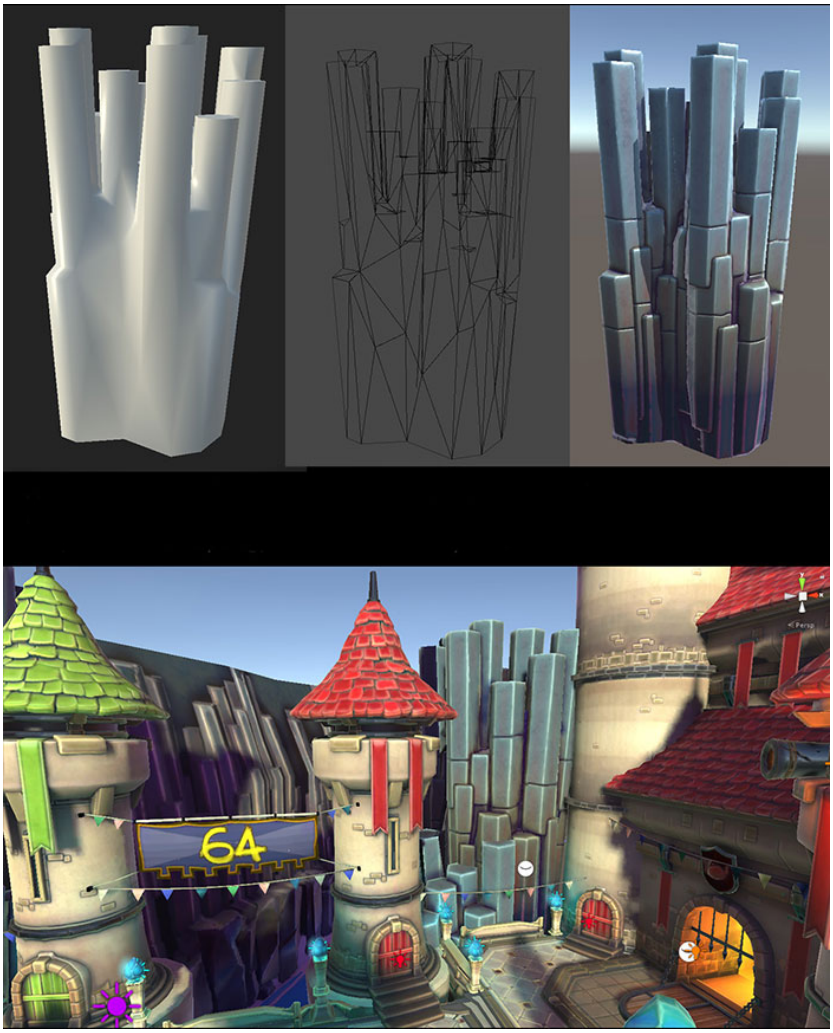

图 5-18 网格拓扑示例

形状夸张

形状夸张是一种将某些部分和形状做得比正常情况下大，使之更加清晰可辨。然而，是否进行夸张化处理取决于你正在创建的游戏的类型和风格。

例如：

- 移动设备屏幕很小，有时很难捕捉到某些微小的形状。夸大形状有助于克服这个问题。例如，将人物的手放大，这样在小屏幕上就更容易看到。

下图显示了夸大手的效果。手、剑和身体部分都以不同比例进行了强调。其目的是为了提高可见性，同时考虑到所使用的少量多边形：

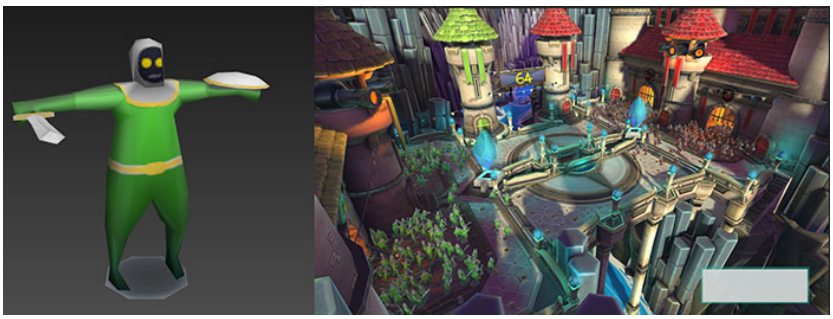

图 5-19 形状夸张示例

第 6 章
实时 3D 美术最佳实践：纹理

本章介绍了多种纹理优化方法，使你的游戏更美观，运行更顺利。

它包含以下部分:

6.1 纹理图谱、过滤和 Mipmap 贴图纹理图谱

纹理图谱是一幅图像，包含了几幅打包在一起的较小图像的数据。不是一个网格一个纹理，而是几个网格共享一个更大的纹理。

可以在制作资源之前创建纹理图谱，这意味着资源是根据纹理图谱进行 UV 展开的。这就需要在创建纹理时进行早期规划。

也可以在资源创建完成后，通过在绘图软件中合并纹理来创建纹理图谱。然而，这也意味着必须根据纹理重新排列 UV island。

说明

UV island 是纹理贴图中一组相连的多边形。

下图突出显示了哪些 3D 对象使用同一个纹理集：

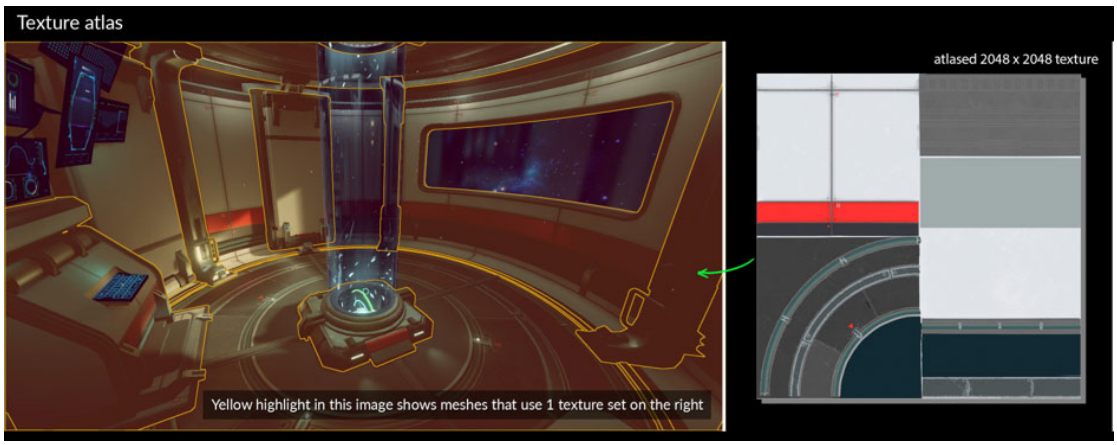

图 6-1 纹理图谱示例

为什么要使用纹理图谱

纹理图谱支持对共享该纹理图谱和相同材质的多个静态对象进行批处理。批处理减少了绘制调用的次数，当游戏受 CPU 限制时，绘制调用越少，性能越好。

对象被标记为静态时，Unity 游戏引擎具有批处理功能。你无需手动合并对象。有关更多信息，请参阅 *Unity 官网*。

纹理图谱在游戏内所需要的纹理数也更少，因为纹理被打包在一起。压缩有助于降低纹理的内存成本。

6.2 纹理过滤

纹理过滤是一种用于改善场景中纹理质量的方法。一般来说，如果不进行纹理过滤，失真（例如走样）会看起来更糟。Unity 针对纹理过滤提供了几个选项。

纹理过滤会使纹理的视效更好，像素块更少。这通常能提升游戏画质。

但是，纹理过滤也会降低性能，因为更高的画质通常需要更多的算力。纹理过滤最高可占据 GPU 一半的能耗。因此，选用更简单的纹理过滤器可以减少一款应用所需的能量。

最近/点过滤

当近距离观察时，近点过滤会让纹理显示为像素块。这是最简单，也是成本最低的纹理过滤。

双线性过滤

双线性过滤会模糊近处的纹理。GPU 会对四个最接近的纹素进行采样，然后取平均值为主像素上色。与近点过滤不同，在双线性过滤下，像素的渐变更平滑，像素块更少。

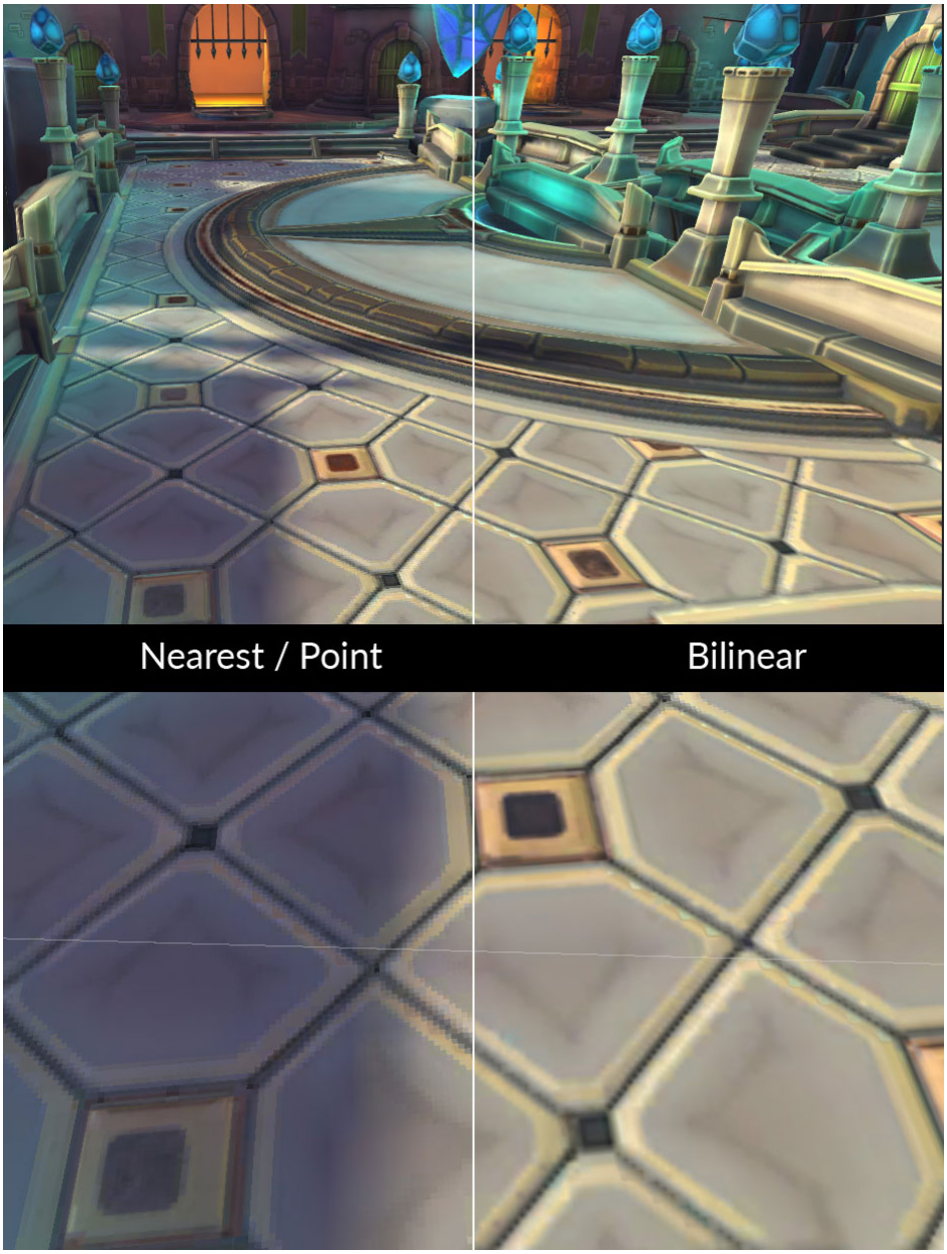

图 6-2 近点和双线性纹理过滤示例

三线性过滤

三线性过滤类似于双线性过滤，但会在两个 Mipmap 贴图层级之间添加混合效果。通过在 Mipmap 贴图之间实现平滑过渡，三线性过滤可以抹除贴图之间明显的界限。

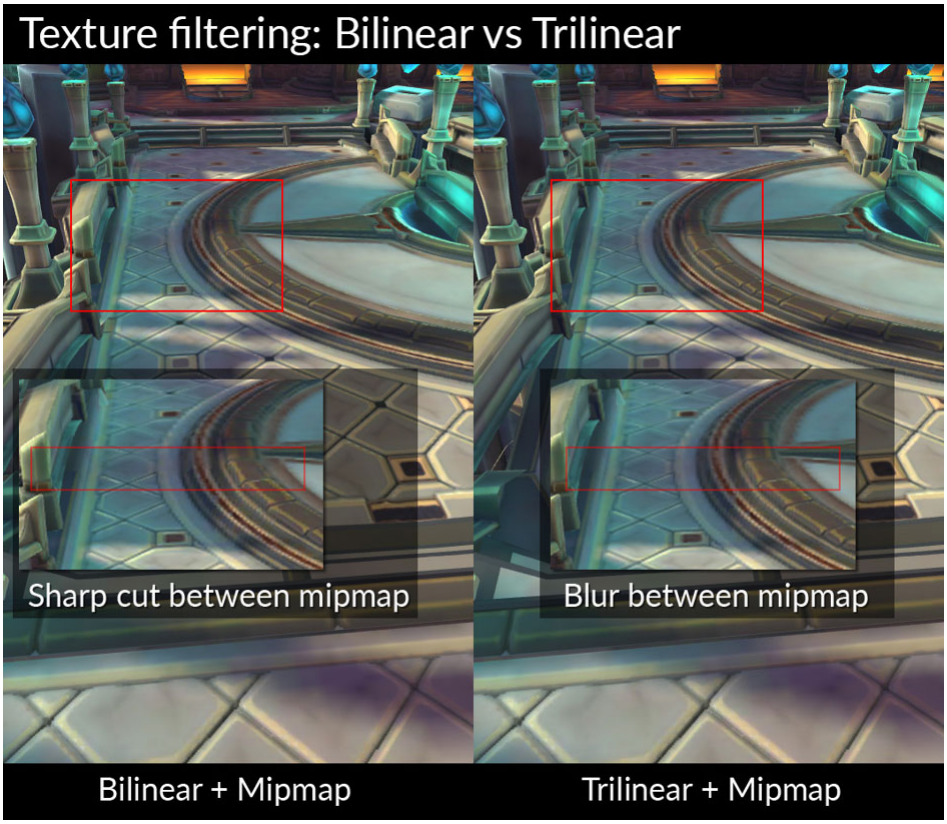

图 6-3 双线性和三线性纹理过滤对比示例

说明

双线性和三线性过滤需要采样更多的像素点，因此会耗费更多算力。

各向异性过滤

各向异性过滤能够提高倾斜纹理的画面质量。一个典型用例便是地平面纹理。

图 6-4 各向异性纹理过滤示例

纹理过滤最佳实践

以下是 Arm 所推荐的纹理过滤小技巧：

- 若要在性能和画面质量之间寻求平衡点，可以使用双线性过滤。
- 有选择性地使用三线性过滤，因为相较于双线性过滤，它占据的内存带宽更多。
- 与其使用三线性/1x 各向异性过滤的组合，不妨使用双线性/2x 各向异性过滤的组合，这样能同时兼顾视效和性能。
- 始终采用较低的各向异性值。仅对游戏中的关键素材使用高于 2 的值。

6.3 Mipmap 贴图

Mipmap 贴图包含原始纹理及其较低分辨率副本，因此你可以认为 Mipmap 贴图类似于*细节层次 (LOD)*。

我们可以根据一个片段所占据的纹理空间大小，选取适当的采样分辨率水平。当一个对象远离镜头时，更低分辨率的纹理便可以派上用场了。而当一个对象贴近镜头时，则使用更高分辨率的纹理。

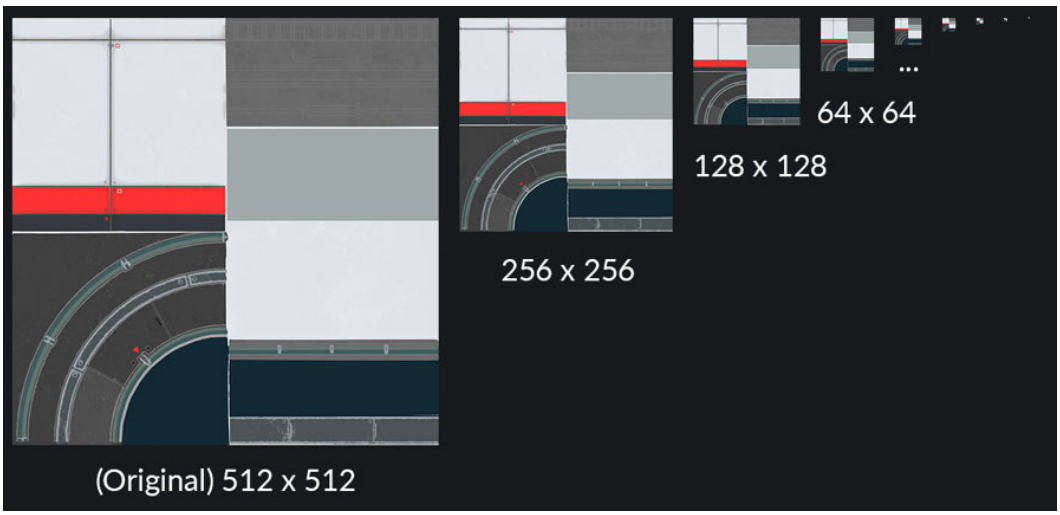

图 6-5 Mipmap 贴图链示例

最佳实践

Mipmap 贴图不仅能提升游戏性能，还能改善画质，务必要善加利用。Mipmap 贴图能提升 GPU 的性能，因为对于一个远离镜头的对象，GPU 不必渲染最高分辨率下的纹理。

Mipmap 贴图还能减少纹理锯齿，改善图像质量。当区域远离镜头时，纹理锯齿会有闪烁的现象。

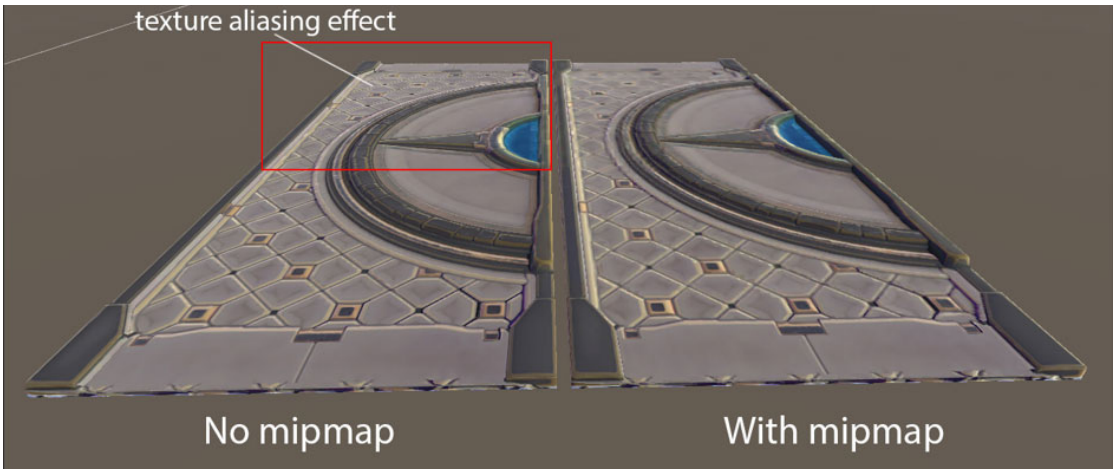

图 6-6 Mipmap 贴图对比示例

Unity 会在导入时根据需要自动创建 Mipmap 贴图，并重新缩放非 2 次方幂的纹理。相关详细信息，请访问 *Unity 的网站*。

6.4 纹理大小、颜色空间与压缩

只创建足够满足质量要求大小的纹理，但不要太大。最好是使用包含多个网格共享纹理的大纹理图谱。

纹理大小

纹理有各种大小。减少某些需要较少细节的纹理的大小有助于降低带宽级别。例如，可以将漫反射纹理设置为 1024x1024，将粗糙或金属贴图设为 512x512。

选择性地缩小纹理大小，然后检查视觉效果是否变差。

纹理颜色空间

Adobe Photoshop 或 Substance Painter 等大多数纹理软件使用 sRGB 颜色空间进行工作和数据导出。

推荐你：

- 在 sRGB 颜色空间使用漫反射材质纹理。
- 请勿将不作为颜色处理的纹理放在 sRGB 颜色空间。包括但不限于，金属、粗糙度和法线贴图。
  - 原因是这类贴图表示的不是物体的颜色。
  - 在这些贴图中使用 sRGB 会导致材质外观或显示上的错误。

说明

在**检视**窗口 **sRGB（颜色纹理）**设置不得用于粗糙度、镜面反射、法线贴图或类似贴图。

以下屏幕快照展示了当 sRGB 被错误地应用于上述纹理时会发生什么：

图 6-7 应用 sRGB 的纹理对比示例

纹理压缩

纹理压缩是一种图像压缩，用于减少纹理数据的大小，同时尽可能地避免降低画面质量。在开发中，我们使用通用格式导出纹理，例如 TGA 或者 PNG。这些格式使用起来更方便，主要的图像软件程序也支持。

与专门的图像格式相比，这些格式的访问和采样速度都比较慢，因此不能在最终渲染中使用。安卓系统具有多个选项，例如 *Adaptive Scalable Texture Compression* (ASTC)、*Ericsson Texture Compression* (ETC) 1 或 ETC2。

纹理压缩最佳实践

我们建议你使用 ARM 开发的 ASTC 技术。ASTC 具有以下优势：

- 内存大小相同时，ASTC 比 ETC 质量更高。
- 相应的，ASTC 能够用更少的内存达到和 ETC 相同的效果。
- ASTC 的编码时间比 ETC 长，而且会增加游戏打包的时间。如果你在意它的话，那么最好只在最终游戏打包时使用 ASTC。
- ASTC 允许通过设置块的大小来对质量进行更多的控制。虽然块大小没有一个理想的默认值，但一开始最好将块大小设置为 5x5 或 6x6。

有时候，如果你必须在设备上快速部署游戏，使用 ETC 进行开发可能会更快。你可以使用具有快速压缩设置的 ASTC 来解决部署时间延长的问题。编码时，可以在速度和质量以及大小之间进行权衡。就平衡画面质量和文件大小而言，ASTC 是最终构建的理想选择。

你打包游戏时，游戏引擎进行纹理压缩。但是你可以选择使用哪一个。你必须选择要使用的格式，因此很难跳过这一步。

以下屏幕快照显示了在 Unity 中构建安卓包时选择 ASTC 的位置：

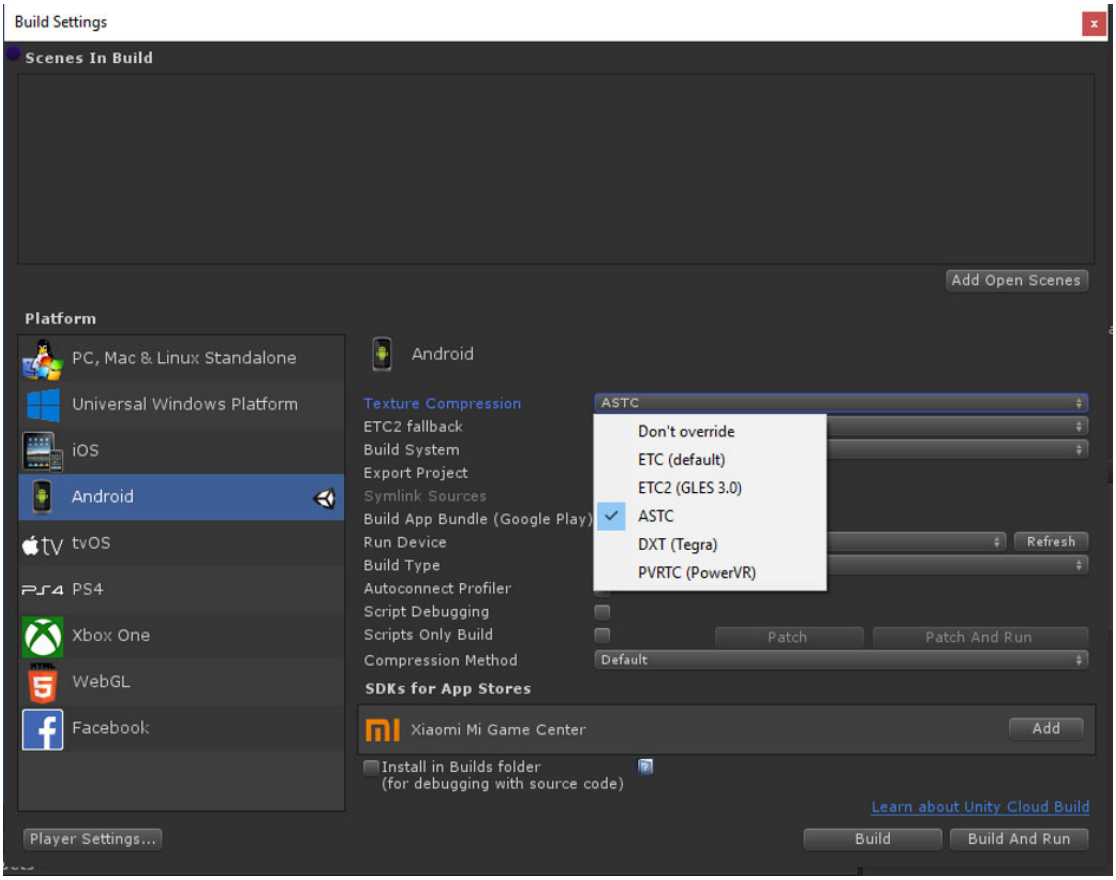

图 6-8 在 Unity 中为安卓启用 ASTC 纹理压缩

下图显示了 ETC 和 ASTC 压缩之间的质量差异和各自的文件大小：

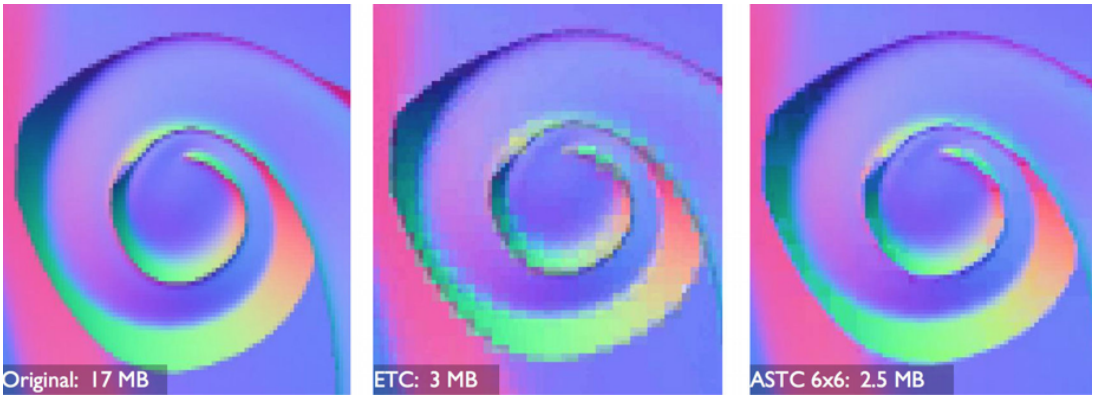

图 6-9 ETC 与 ASTC 纹理压缩比较

6.5 UV 展开、视觉冲击和纹理通道打包

UV 贴图将 2D 纹理投影到 3D 模型的表面。UV 展开是创建 UV 贴图的过程。

UV 展开

最佳做法是保持 UV island 尽可能直。

说明

UV island 是纹理贴图中一组相连的多边形。

原因如下：

- 使得 UV island 更容易打包，浪费的空间更少。
- 直的 UV 有助降低纹理上发生的阶梯效应。
- 在移动平台上，纹理空间是有限的，因为纹理大小通常比游戏主机或电脑上显示的要小。良好的 UV 打包可确保纹理获得更高分辨率。
- 为保持 UV 笔直从而获得整体质量更好的纹理，让 UV 出现稍许扭曲也是值得的。

将 UV 接缝放在不太显眼的地方。这样做有利于视效，因为纹理接缝在模型上看起来可能不太美观。因此，在边缘清晰处分离 UV island，之间稍微保持一点距离。这有助于后续通过烘焙过程创建更好的法线贴图。

下图显示了如何利用 UV 展开最大化纹理空间的示例：

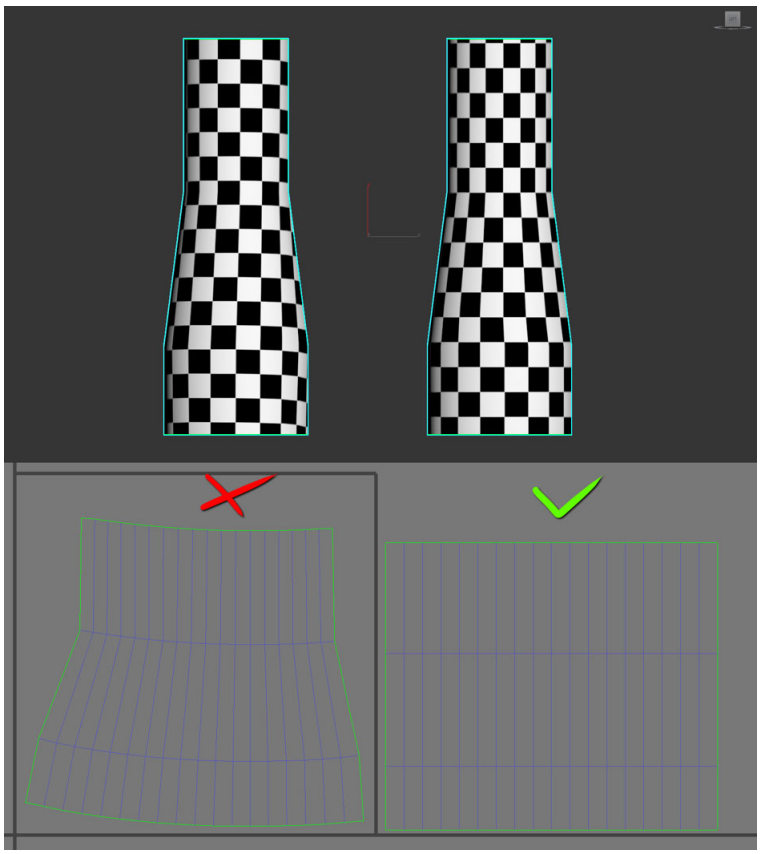

图 6-10 利用直的 UV 展开最大化纹理空间的示例

视觉冲击

确保你只在看得见的地方创建细节。手机屏幕小，因此精细的细节是看不见的。创建纹理时请考虑这一点。例如，对于一把在房间角落几乎看不见的椅子，就不需要有很多细节的 4K 纹理。

下方截屏显示了仅有所需细节的士兵身上的小纹理示例：

图 6-11 精细程度合适的小纹理示例

有些情况下，你需要着重突出显示边缘和阴影，让形状更清晰可辨。请记住，移动平台通常使用较小的纹理。因此，可能很难在小纹理内展现出所需的所有细节。

使用较少的纹理，然后将其他细节烘焙到同一个纹理中。这很重要，因为：

- 手机屏幕很小，最好将某些细节烘焙到漫反射纹理本身上，以确保这些细节看得见。
- 可以烘焙环境遮挡和小镜面反射高光之类的元素，然后将其添加到漫反射纹理中。

利用这种方法，你不必太依赖着色器和引擎功能进行镜面发射和环境光遮挡。

下方截屏显示了如何将细节烘焙到纹理中的示例：

图 6-12 烘焙到纹理中的细节示例

如果可能，请使用允许在着色器中着色的灰度纹理。这样可以节省纹理内存，但你需要创建自定义着色器执行着色。

此方法的显示效果并非适合所有对象，因此请选择性地使用。将其应用于具有均匀或相似颜色的对象会更容易。

你也可以使用 RGB 遮罩，然后应用基于遮罩颜色范围的纹理来执行此操作。

下图显示了应用了灰度纹理的有色石柱的示例：

图 6-13 灰度纹理示例

纹理通道打包

使用纹理通道将多个纹理打包为一个纹理。我们还推荐：

- 打包有助于节省纹理内存，因为你可以使用此技术将三个贴图合并为一个纹理。这样就可以使用更少的纹理采样器。
- 纹理打包技术通常用于将粗糙度或平滑度和金属打包到一个纹理中。但它可以用于任何纹理遮罩。

使用绿色通道存储更重要的遮罩。绿色通道通常具有更多位。这是由于人们的眼睛对绿色更敏感，而对蓝色不那么敏感。我们还推荐：

- 粗糙度或平滑度贴图通常比金属贴图具有更多细节，并且可以放置在绿色通道中。
- 为这些纹理设置为线性或 RGB 颜色空间，而不是 sRGB 颜色空间。

下图显示了纹理打包示例：

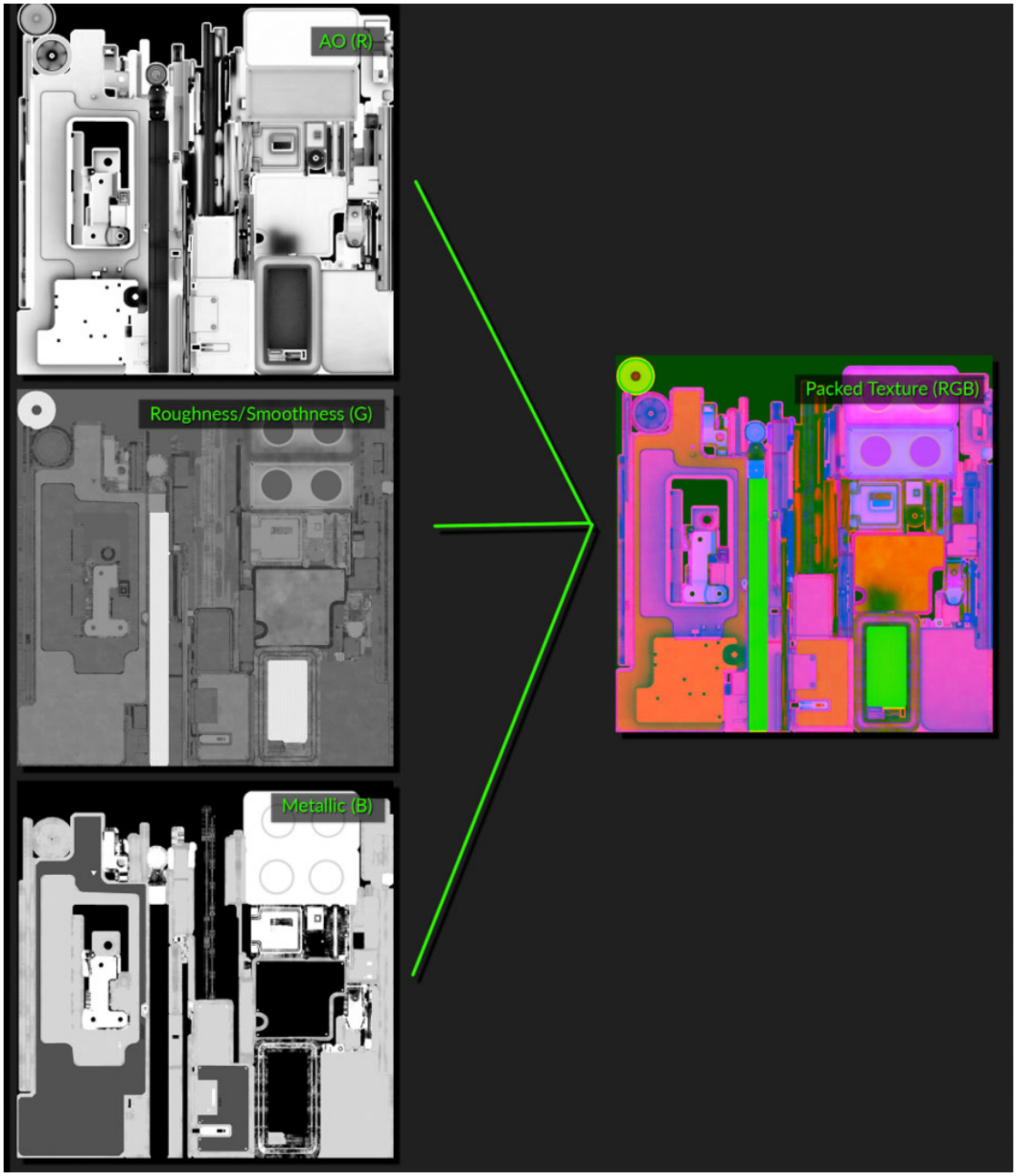

图 6-14 纹理通道打包示例

有关对色彩敏感的视锥细胞的更多信息，*请点击这里*。

6.6 Alpha 通道和法线贴图最佳实践

如果你想在游戏中使用 Alpha 通道或法线贴图，推荐你尝试如下最佳实践。

使用 Alpha 通道

要选择性地向纹理添加 Alpha 通道。增加透明度，会将纹理转换为 32 位格式，从而增加纹理文件的大小，因此会提高整体的内存占有率。

另一种存储 Alpha 通道的方法是在粗糙或金属纹理中使用额外的通道。Unity 中，这种纹理有时使用三个通道中的两个，即粗糙通道 (G) 和金属通道 (B)，剩下的 (R) 通道可以自由使用。

使用自由通道存储 Alpha 掩码，你可以将漫反射纹理保持在 16 位，从而将文件大小减半。环境遮挡贴图通常可以在漫反射贴图中进行烘焙。

下图为如何在红色通道中存储不透明度贴图的示例：

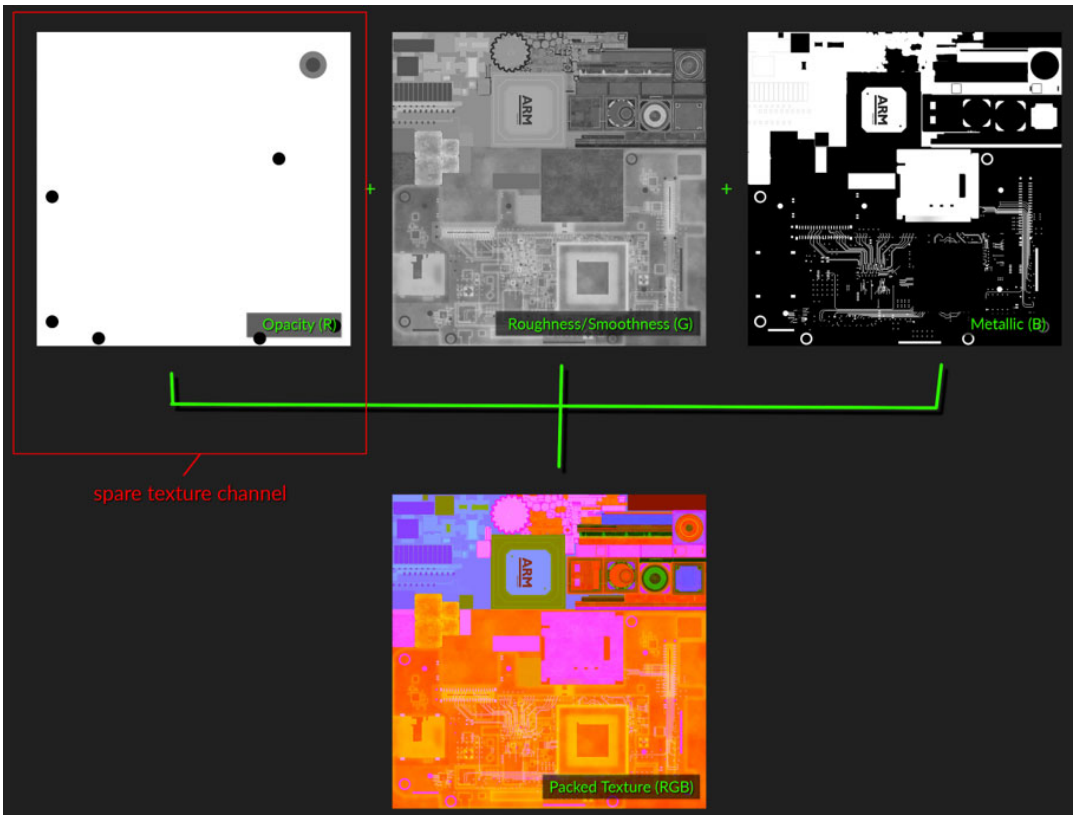

图 6-15 在红色通道存储不透明贴图

法线贴图最佳实践

法线贴图可以使 3D 对象呈现出更多细节。该方法适用于添加更小的细节，如皱纹、螺栓和其他需要大量三角形建模的细节。是否使用法线贴图取决于游戏的类型和美术设计。

我们在大多数内部项目使用法线贴图，但没有明显的性能下降。由于我们大部分演示都针对高端设备，低端设备结果可能会不同。

虽然算力消耗很低，但法线贴图确实会消耗 GPU 的运算量。请记住：

- 法线贴图是一个额外的纹理，意味着要提取更多纹理，因此会使用更多的带宽。
- 请谨慎在低端设备上使用法线贴图。

然而，如果使用法线贴图后几何图形具有较少的三角形，则可以提高性能。

下图为如何使用法线贴图和纹理处理较小细节的示例：

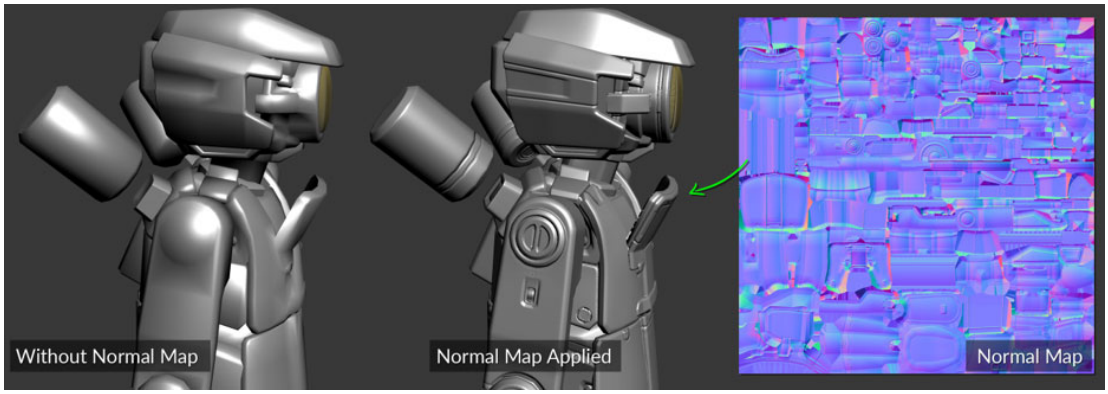

图 6-16 法线贴图和纹理示例

法线贴图烘焙最佳实践

无论你烘焙的是何种类型的表面，使用笼子是获得高质量法线贴图的好方法。大多数法线贴图软件会自动制作笼子。必要时，你可以从低多边形模型的副本制作法线贴图，然后稍微增加其大小。

该程序使用笼子来改变烘焙时用于计算法线的方向。这样可大大改善分裂法线和硬边缘的效果。

笼子是低多边形数模型的较大版本，因此该模型看起来像被推出来了。笼子还必须在物理上覆盖高多边形计数模型，才能顺利进行烘焙。

6.7 法线贴图烘焙最佳实践

无论你烘焙的是何种类型的表面，使用笼子是获得高质量法线贴图的好方法。大部分法线贴图软件可自动制作笼子，但是你可以从低多边形模型的副本制作法线贴图，然后稍微增加其大小。

使用笼子可帮助程序改变烘焙时用于计算法线的方向。如下图所示，这样可以改善分裂法线和硬边缘的效果：

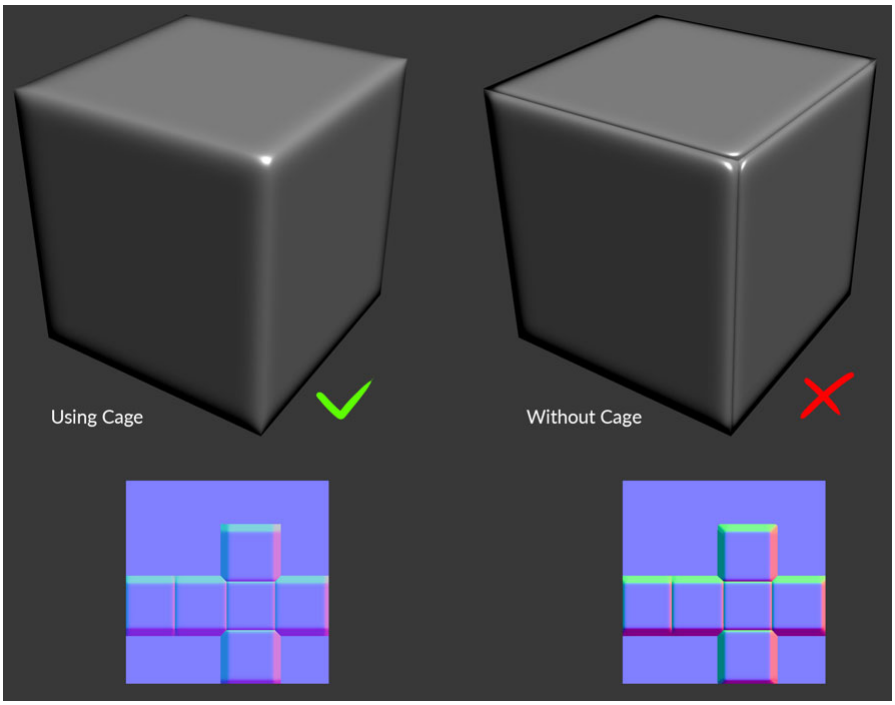

图 6-17 笼子使用比较

笼子基本上是低多边形数模型的较大版本或推出版本。必须覆盖高多边形计数模型，才能顺利进行烘焙。

网格笼用于限制法线贴图烘焙过程中使用的射线投射距离。如下图所示，笼子还可以解决法线贴图上分裂法线的接缝问题：

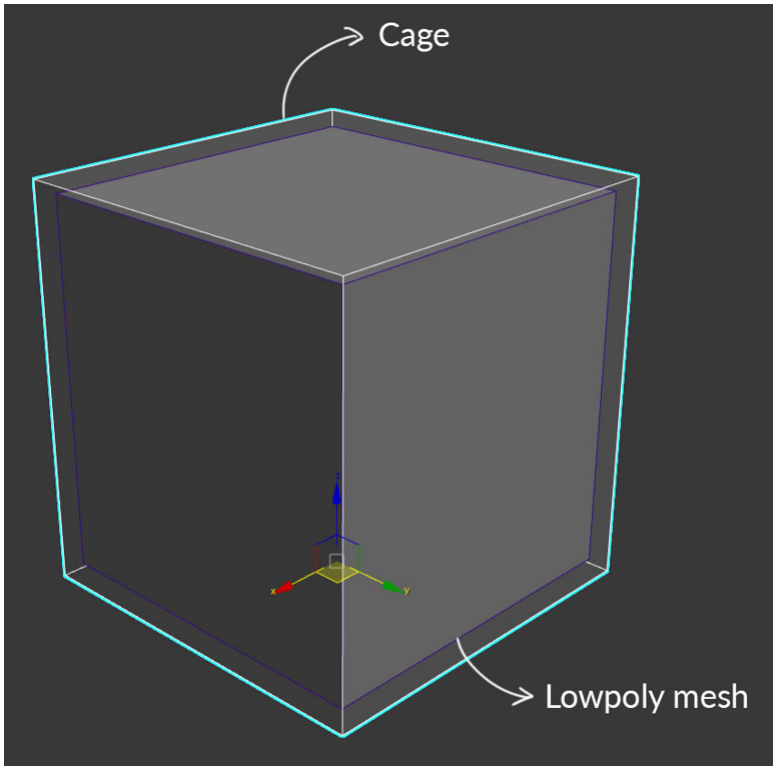

图 6-18 网格笼示例

烘焙软件支持时，请通过匹配网格名称进行烘焙。这样可以缓解创建的法线贴图投影错误的问题。对象彼此靠得太近时，会意外地将法线贴图投影到错误的面上。该方法可以确保只烘焙正确的、名称匹配的表面。

有关按名称匹配网格的更多信息，*请参阅 substance3d 网站*以及 *Marmoset Toolbag 教程*。

如果无法匹配要烘焙的网格名称，请切割网格。切割网格意味着将零件彼此移开，以便法线贴图不会投影到不需要的表面上。这也有助于避免不正确的法线贴图投影。

下图显示了被切割的网格示例：

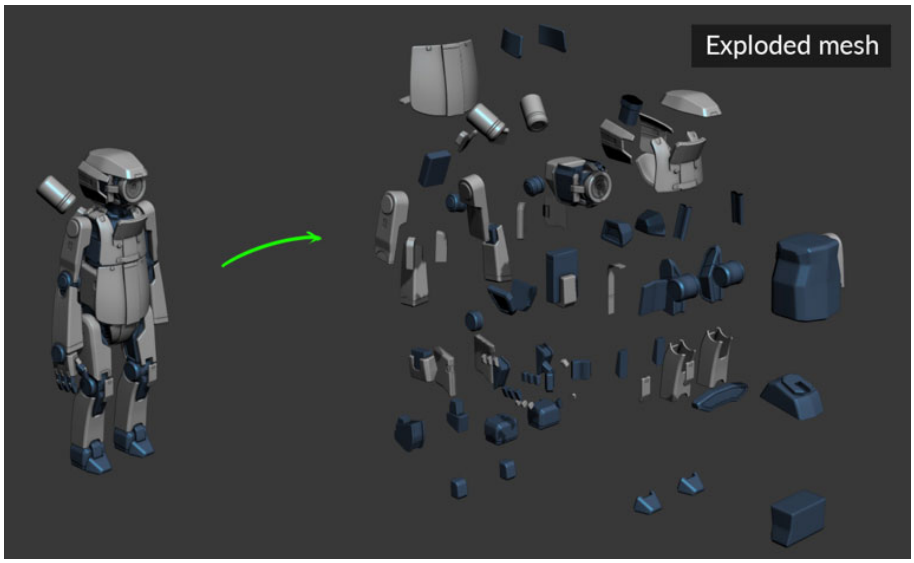

图 6-19 被切割的用于法线烘焙的网格示例

该解决方案有时可能需要对环境遮挡进行单独烘焙。因此，在硬边上拆分 UV，因为在硬边上连续 UV 会导致明显的接缝。一般规则是将角度保持在 90 度以下，或将其设置为其他平滑组。在三角形上让具有不同平滑组的 UV 接缝重合。

下图显示了分裂 UV 在硬边缘上的视效示例：

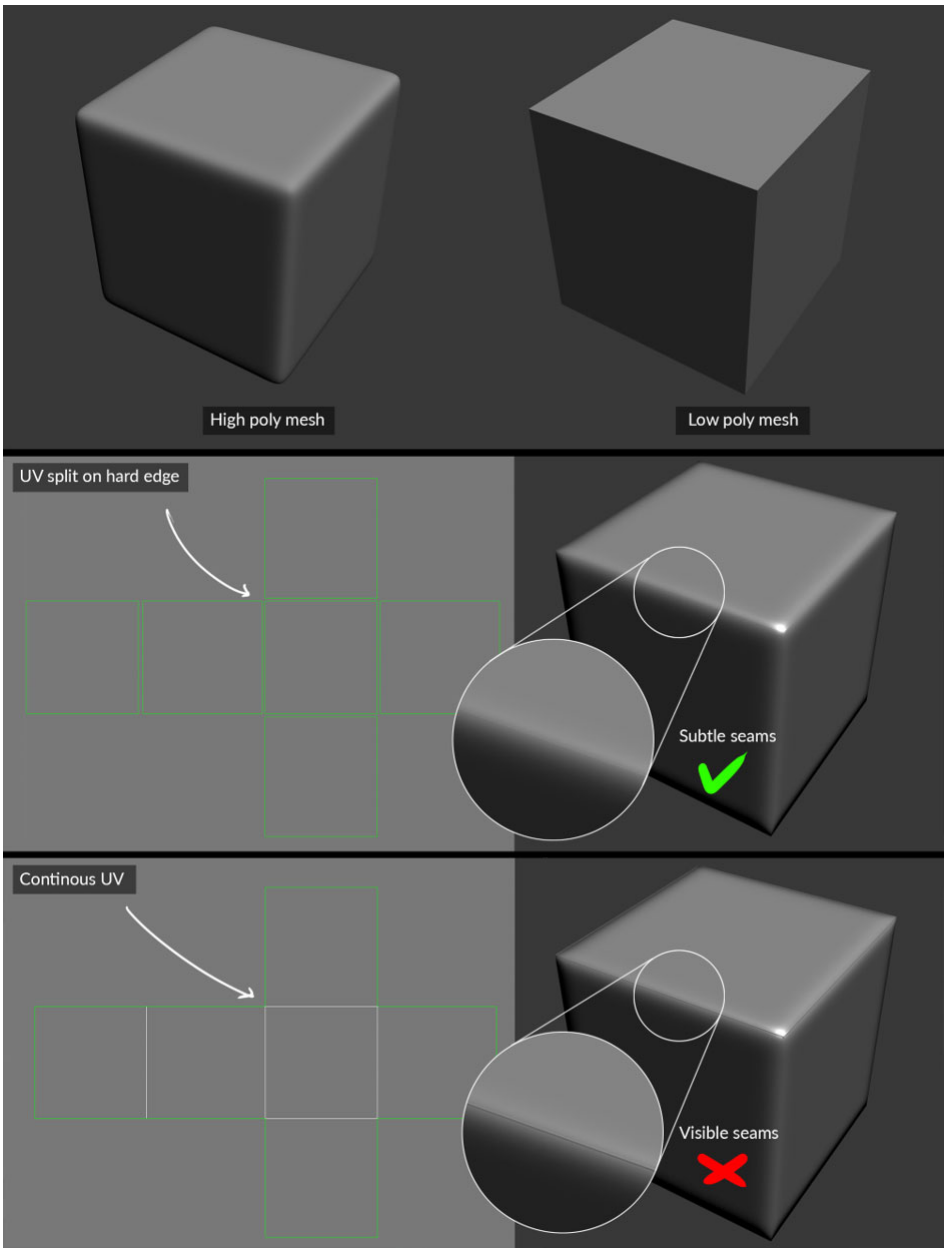

图 6-20 硬边缘上分裂 UV 示例

6.8 编辑纹理设置

Unity 让你容易地编辑纹理设置。选择你想编辑的纹理，以打开显示纹理设置的检视窗口。

编辑纹理设置时，请记住以下几点：

- **纹理类型**用于控制纹理的类型，从而使纹理在引擎中以不同的方式使用。
- **纹理形状**允许你为某些类型的纹理选择立方体贴图而非 2D 贴图。
- 根据选择的**纹理类型**，出现如下两个部分：
  - **纹理设置**具有可以根据需要工作的纹理的特定控件。
  - **高级设置**包括 **sRGB**、**Alpha 源**、**Alpha 透明**、**读写已启用**和**生成 Mipmap 贴图**。
- **循环模式**控制纹理循环 UV 的方式。可用选项包括：
  - **重复**平铺纹理，并用于重复图案。
  - **强制拉伸**将纹理锁定到边缘的最后一个像素。
  - **镜像**的工作方式与**重复**类似，但会对所有重复的纹理进行镜像。
  - **镜像一次**只将纹理镜像一次，然后将纹理锁定到边缘像素。
- **滤波模式**用于控制在纹理上使用哪种纹理滤波方式。
- 纹理压缩盒用于控制：**最大尺寸**、**Resize 算法**、**格式**、**压缩**和**使用 Crunch 压缩**。

第 7 章
实时 3D 美术最佳实践：材质和着色器

本章介绍了多种纹理和着色器优化方法，使你的游戏运行更高效，画面更美观。

它包含以下部分:

7.1 着色器和材质介绍

材质和着色器决定了 3D 对象的外观，因此了解它们各自的功能以及如何进行优化非常重要。

着色器是一个小程序，它告诉 GPU 如何在屏幕上绘制对象，以及必须对该对象进行的每个计算。着色器只能在附着到材质时使用。

对于编写着色器，你可使用两种常用的脚本语言：*高级着色语言* (HLSL) 和 *OpenGL 着色语言* (GLSL)。

材质可以应用到对象或网格上，用来决定对象的显示效果。使用着色器后即可设置一系列参数，而材质即用于将这些参数设为特定的值。例如，颜色、纹理和数值。

以下两个截屏展示了着色器由不同材质创建并使用的两种方式：

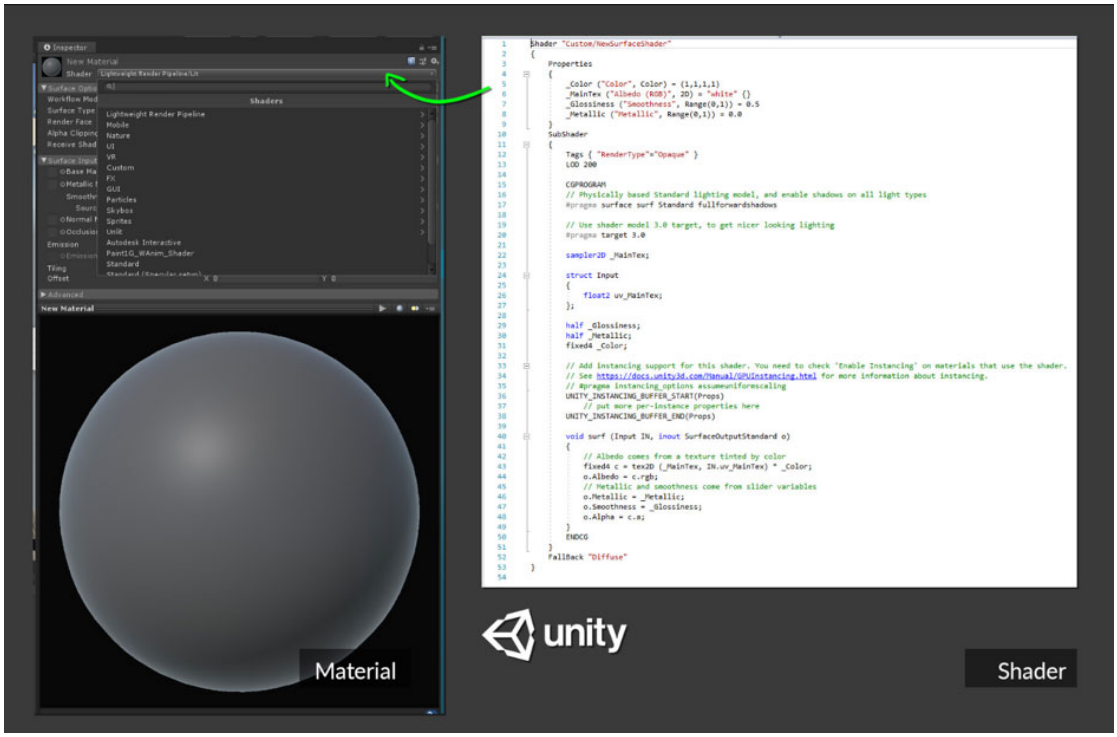

图 7-1 着色器和材质

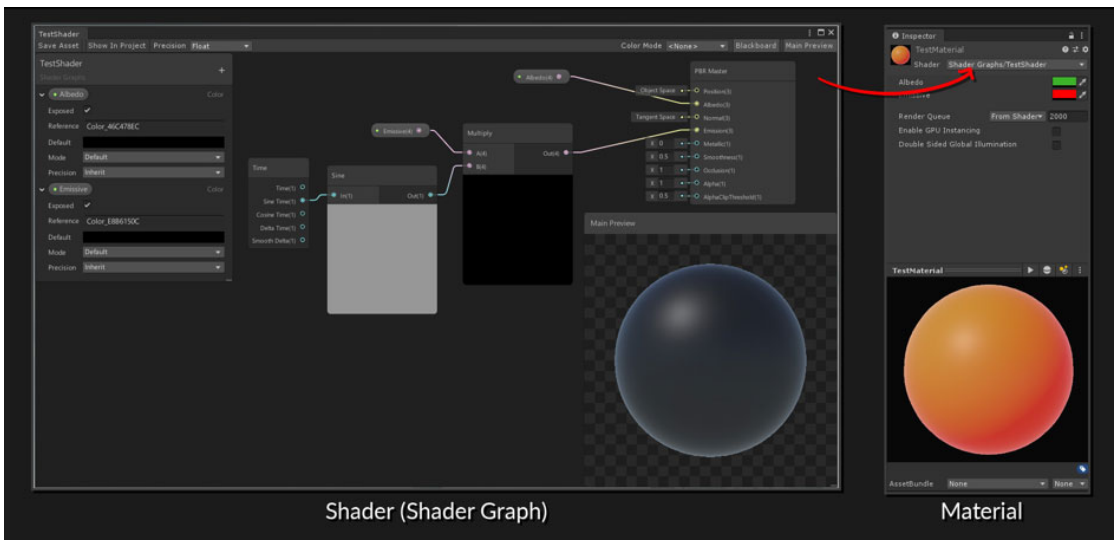

图 7-2 着色器图形和材质菜单

7.2 使用针对移动平台进行优化的着色器

Unity 提供一系列针对移动平台进行优化的着色器。这些着色器位于 Unity 的移动类别中。

优化后的着色器功能更为简单，但它们能够在移动平台上获得更好的性能。当然，与标准的非移动着色器相比，可用的功能较少。例如，没有“color tinting”功能。

以下屏幕快照显示了你可以找到移动专用着色器的位置：

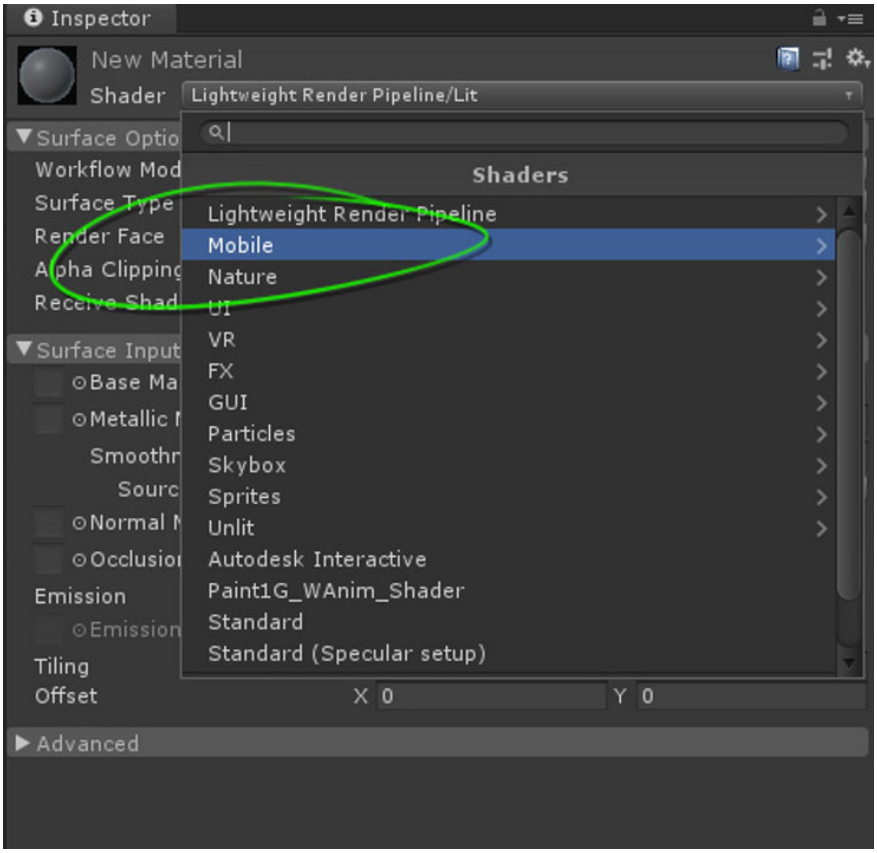

图 7-3 Unity 移动着色器的位置

请仅使用需要的功能。根据材质中选择和配置的着色器，Unity 创建优化的运行版本。降低着色器复杂度有助于提高其在移动平台的性能。

7.3 优化你的纹理

在移动平台上尽可能少用纹理。因为纹理越多，纹理提取就越多。而纹理提取越多，使用的带宽就越多，从而影响设备的电池寿命。

保存在内存中的纹理越多，应用程序的大小也会增加。你可以将粗糙纹理和金属纹理打包到单个纹理的通道中，而不是使用多个独立的纹理。这种技术称为纹理打包，有助于减少纹理的使用数量。

下图为如何将三个单独的纹理打包到一个通道中以节省带宽的示例：

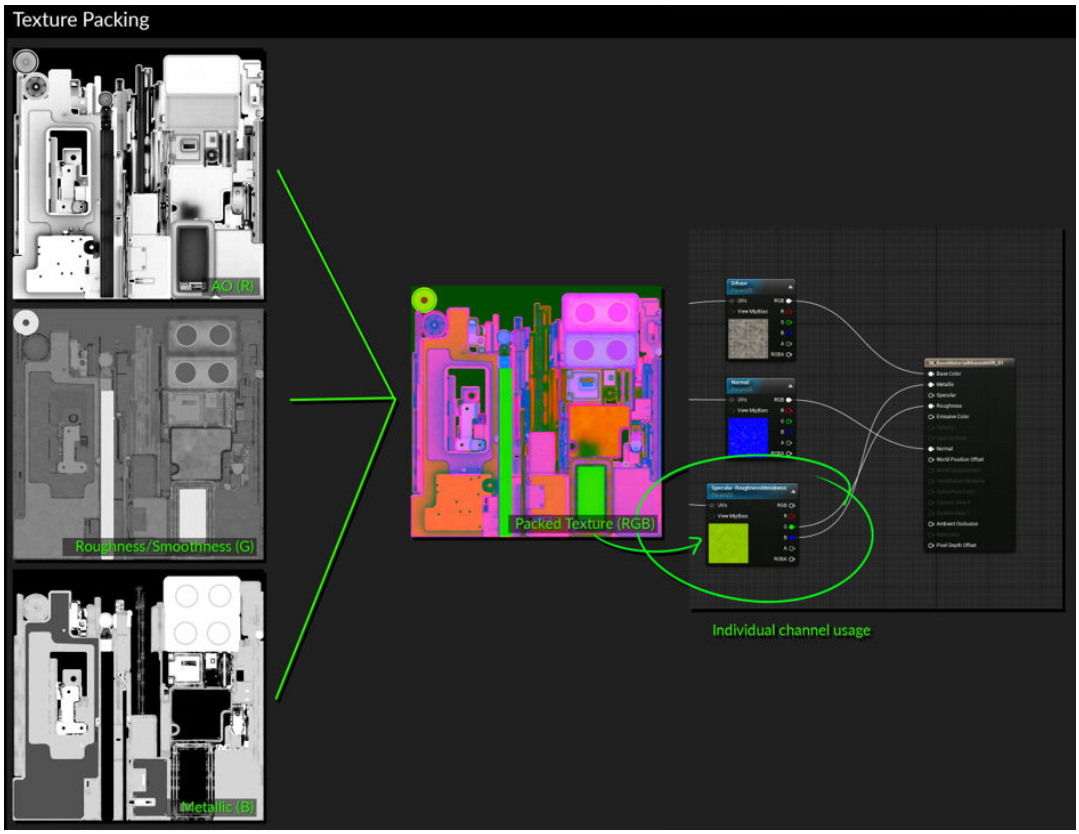

图 7-4 Unity 中的纹理打包

对于某些参数，如金属、粗糙度或平滑度，你也可以使用数值而非纹理。尽可能这样做，并观察它是否影响画面质量。使用数值会进一步减少使用的纹理数量。

说明

在 Unity 中，必须在着色器中添加一个值。

由于光照不会影响材质，也可以使用“unlit”着色器来减少使用的纹理数量。这时不需要粗糙或金属纹理。

7.4 对比“unlit”和“lit”着色器

创建着色器时，你可以确定材质对光线的反应。移动游戏中使用的大多数着色器分为“lit”着色器和“unlit”着色器。

Unlit

“unlit”着色器是速度最快且最实惠的着色模型。这类着色器适用于低端设备。需要考虑的几个要点包括：

- 照明不影响“unlit”的阴影模型，只输出发光的颜色。
- 因此很多计算都不需要，例如镜面反射。从而降低渲染成本或提高渲染速度。
- 在“unlit”时进行着色，对卡通等风格化的艺术设计也很好用。在移动平台上制作游戏时，这种艺术风格是值得考虑的。

Lit

与“unlit”着色器相比，“lit”着色器消耗更多的算力。但是：

- 光线会影响“lit”着色器，使表面产生镜面反射。
- 这可能是当今手机游戏中最常用的阴影模型。

下图对比了“lit”对象与“unlit”对象：

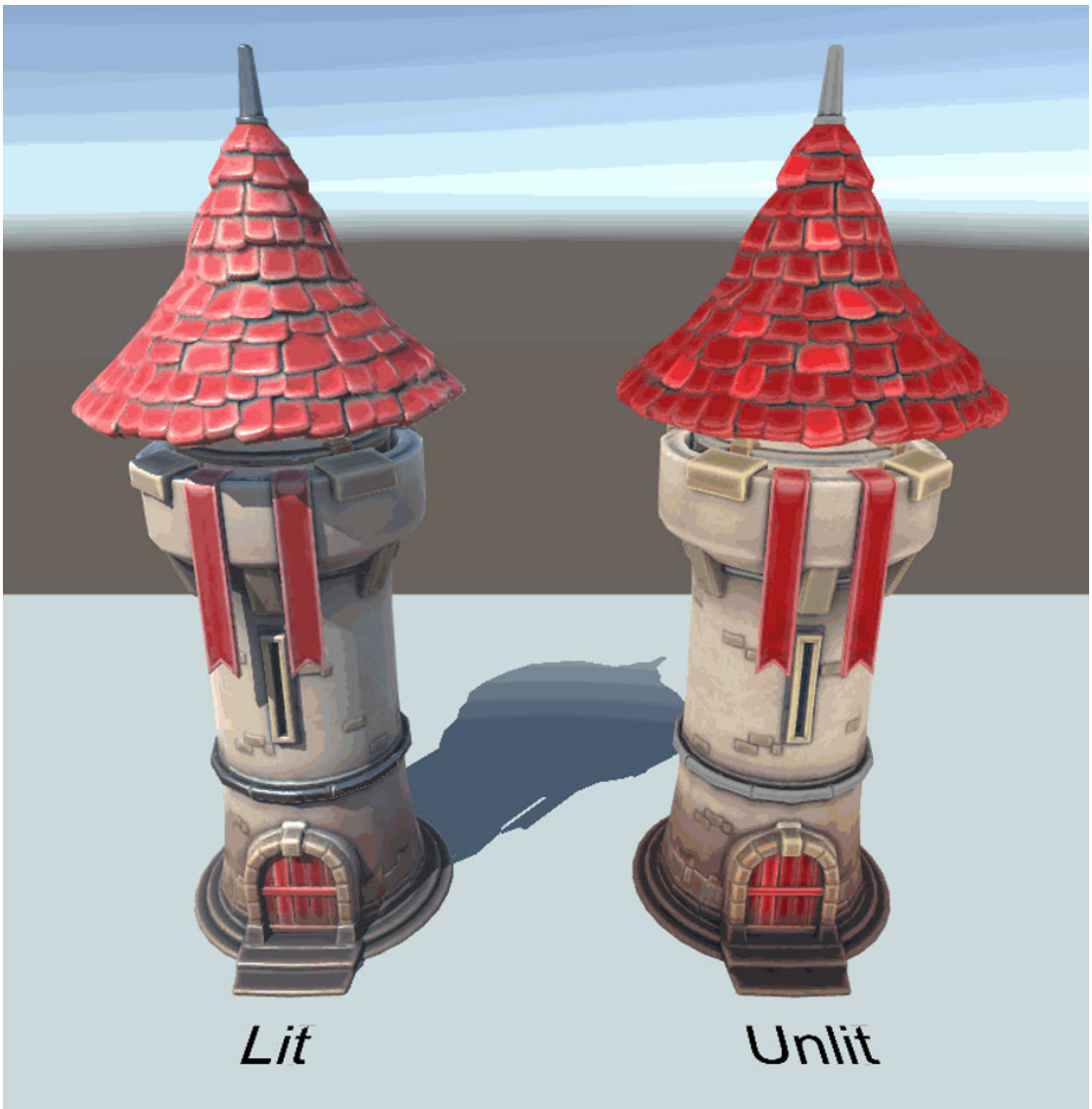

图 7-5 对比“unlit”和“lit”着色器

塔网格和纹理的设置相同，但是使用的着色器不同。光线对“unlit”着色器没有影响，因此需要的计算量较少。因此游戏性能更佳，特别是在低端设备上。

7.5 谨慎使用透明效果

尽可能使用不透明的材质。除非必要，在移动平台上最好避免使用透明材质。

渲染透明物体总是比渲染不透明物体使用更多的 GPU 资源。使用许多透明物体会影响移动平台的性能。尤其是当透明物体被多次叠加渲染时。

同一个像素在屏幕上绘制多次，就是所谓的过度绘制。过度绘制是个问题，因为透明层次越多，渲染的成本就越高。对于移动平台，过度绘制会严重影响性能。

下方截屏显示了蓝光如何产生透明效果，但是即便使用不透明材质，画面效果也并不差：

图 7-6 使用不透明材质

因此，构建层次时，将过度绘制降至最低。Unity 提供一种模式，可以用来查看场景中的过度绘制量。下方截屏中高亮显示了该模式：

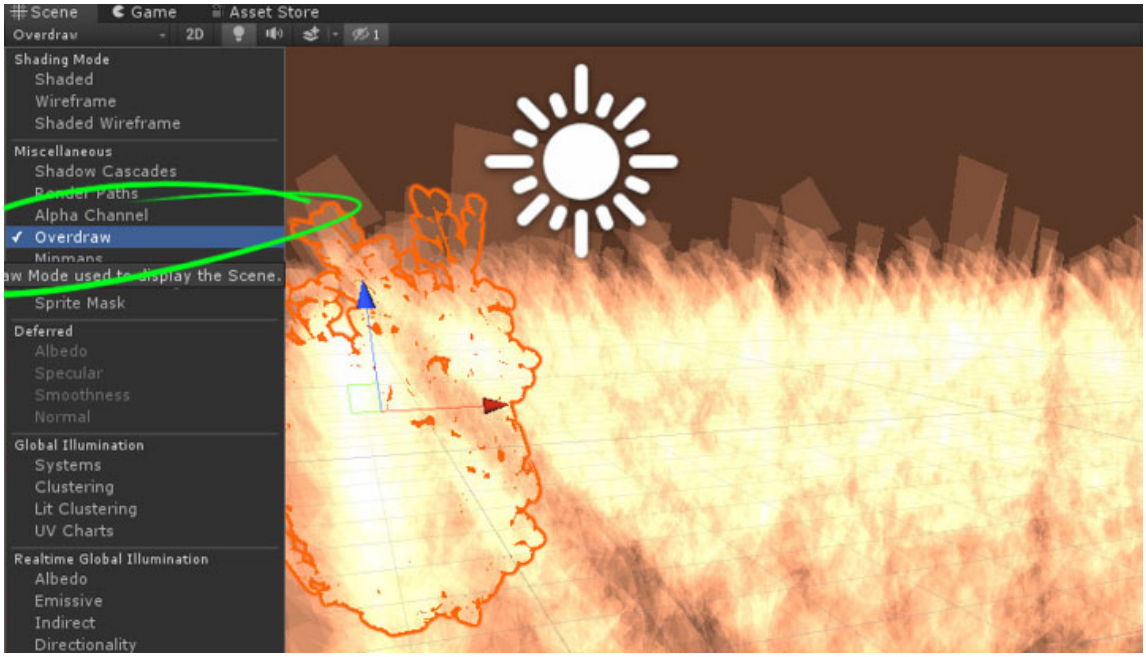

图 7-7 Unity 中的过度绘制可视化

7.6 分析和比较透明实现方式

在着色器中有不同的透明方式，最常用的是 Alpha 测试和 Alpha 混合。但是，使用情形不同，各种方法的效果也会有所不同。建议你始终分析两者之间的性能差异。

在移动平台上，你可使用 ARM 工具 *Streamline* 收集设备的性能数据。

Alpha 测试或裁剪

Alpha 测试让对象材质看起来要么 100% 不透明，要么 100% 透明。你可以为遮罩设置裁剪阈值。在 Unity 中，这种类型的透明方式称为**裁剪**。

Alpha 混合

视觉上，Alpha 混合允许材质有一定的透明度，使对象看起来部分透明，而非完全不透明或完全透明。Unity 将这种混合模式称为**透明**。

Alpha 混合允许部分透明，而 Alpha 测试会产生非黑即白的裁剪效果。下图对比了 Alpha 混合和 Alpha 测试：

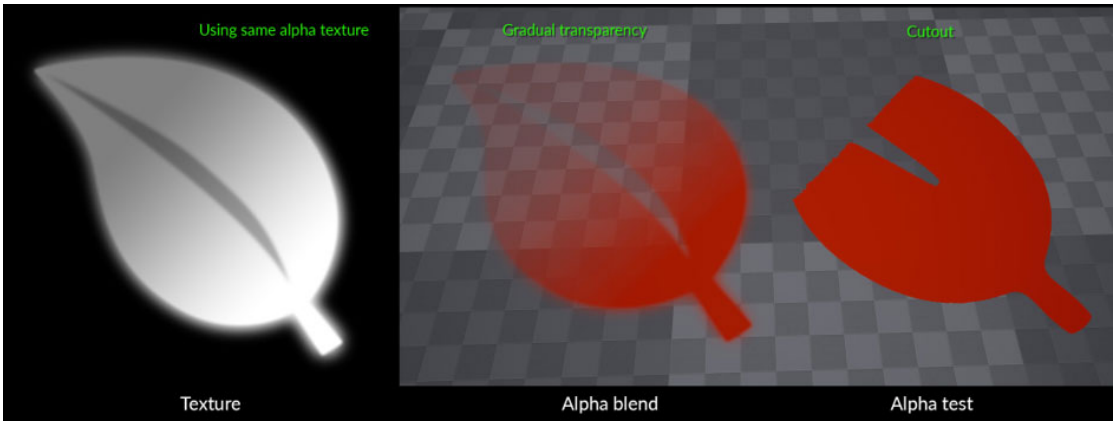

图 7-8 Alpha 混合和 Alpha 测试对比

Alpha 测试

如果你想在游戏中使用 Alpha 测试，请记住以下几点：

- 除非需要，最好避免使用 Alpha 测试或裁剪。
- Alpha 测试意味着材质要么 100% 不透明，要么 100% 透明。
- Alpha 测试禁用了 GPU 的某些优化功能，因此我们建议你在目标移动平台上测试该方式是否可用。此外，请勿忘记分析和比较该方式与 Alpha 混合之间的性能差异。

Alpha 混合

如果你想在游戏中使用 Alpha 混合，请记住以下几点：

- 对于移动平台，Unity *建议你使用 Alpha 混合，而非 Alpha 测试*。在实际使用中，你应该分析并比较 Alpha 测试和 Alpha 混合的表现，因为这取决于具体内容，因此需要测量。
- 通常在移动平台上应避免使用 Alpha 混合来实现透明。
- 需要进行 Alpha 混合时，尝试缩小混合区域的覆盖范围。

在树叶上使用 Alpha 测试实现透明

当显示静止状态下的树叶时，由于柔化的边缘，Alpha 混合的效果看上去更好。与 Alpha 测试一刀切式裁剪效果的对比如下图所示：

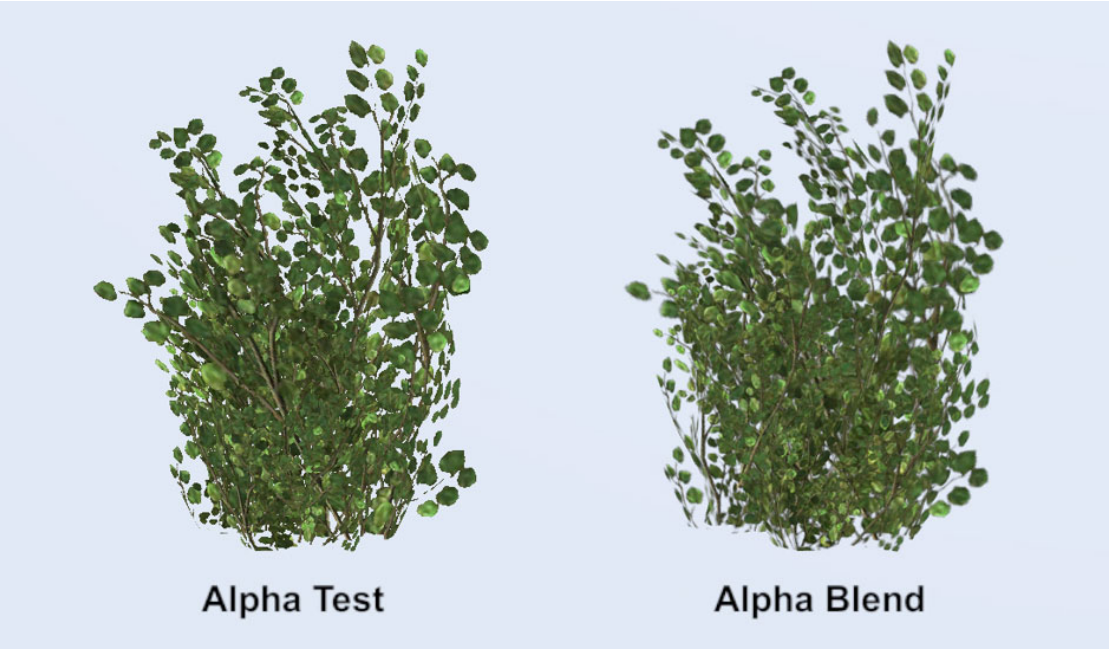

图 7-9 Alpha 混合和 Alpha 测试树叶示例

然而，在运动中应用 Alpha 混合后视效是错误的，且没有按正确的顺序渲染树叶。Alpha 测试可以更好地处理叶子的透明效果和渲染顺序。但是与 Alpha 混合相比，边缘将更粗糙/更有锯齿感。

锯齿边缘在运动中并不明显，因此 Alpha 测试的视效质量通常是可以接受的。但是当树叶和树枝因为使用 Alpha 混合而前后跳动时，视觉假象就被打破了。

7.7 其他材质和着色器最佳实践

Arm 建议你在游戏开发过程中牢记以下其他最佳实践技巧。

在过度绘制不可避免的情况下，请尽可能简化着色器。请牢记以下原则：

- 使用尽可能简单的着色器（例如“unlit”着色器），并避免使用不必要的功能。
- Unity 有一个专为粒子设计的着色器，可作为很好的起步实现方案。
- 尽可能减少过度绘制。减少例子的数量和/或大小有助于减少过度绘制。

分析着色器复杂度

添加更多纹理采样器、透明度和其他功能会使着色器更加复杂，并可能影响渲染。因此，我们建议你经常分析着色器。

为此，Arm 提供了一些工具，例如 *Mali Offline Shader Compiler* 和 *Streamline*。但是，这些工具需要你具备更高水平的图形知识，以便进一步了解 GPU 的工作原理。如果要编辑着色器，那么这是一个值得深入研究的领域。

在顶点着色器中执行尽可能多的运算

在项目中，通常是通过顶点着色器和像素（或片段）着色器的组合来实现特定画面效果的。顶点着色器用于每个顶点，而像素着色器则用于每个像素。

通常，渲染的像素多于屏幕上的顶点。这意味着像素着色器比顶点着色器的运行频率更高。我们建议你尽可能将计算从像素着色器移到顶点着色器。

将运算移到顶点着色器通常意味着需要通过变量将处理后的数据移到像素着色器。因此，即使通常情况下这是一个好方法，但是也必须注意一下“tiler”，以防使它成为瓶颈。与往常一样，在进行优化之后，必须进行进一步的性能分析。

尽可能避免使用复杂的数学运算

你可以在着色器中使用数学运算来自定义其外观和行为。这些数学运算的一些示例包括乘法、加法、乘方、底数、对数和平方根。

这些数学运算在性能成本方面并不相同。因此，必须注意它们的用法。一些较复杂的运算包括：sin、pow、cos、divide 和 noise。

基本运算（例如加法和乘法）的处理速度更快，因此，请尽量减少较慢的数学运算的数量。在较旧的设备（例如使用 GLES 2.0 的设备）上，必须保持较低的复杂数学运算量。

始终进行性能分析

若要了解应用程序中发生真正瓶颈的位置，请分析性能。为比较优化使用前后的效果，也建议进行性能分析。

第 8 章
高级图形技术

本章列出了你可以利用的多种高级图形技术。

它包含以下部分:

8.1 自定义着色器

本节介绍自定义着色器。

本部分包含以下子部分:

8.1.1 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，模拟材质与光线之间的交互。这可提供很高程度的真实感，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。

你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。你可以在**检视器**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。

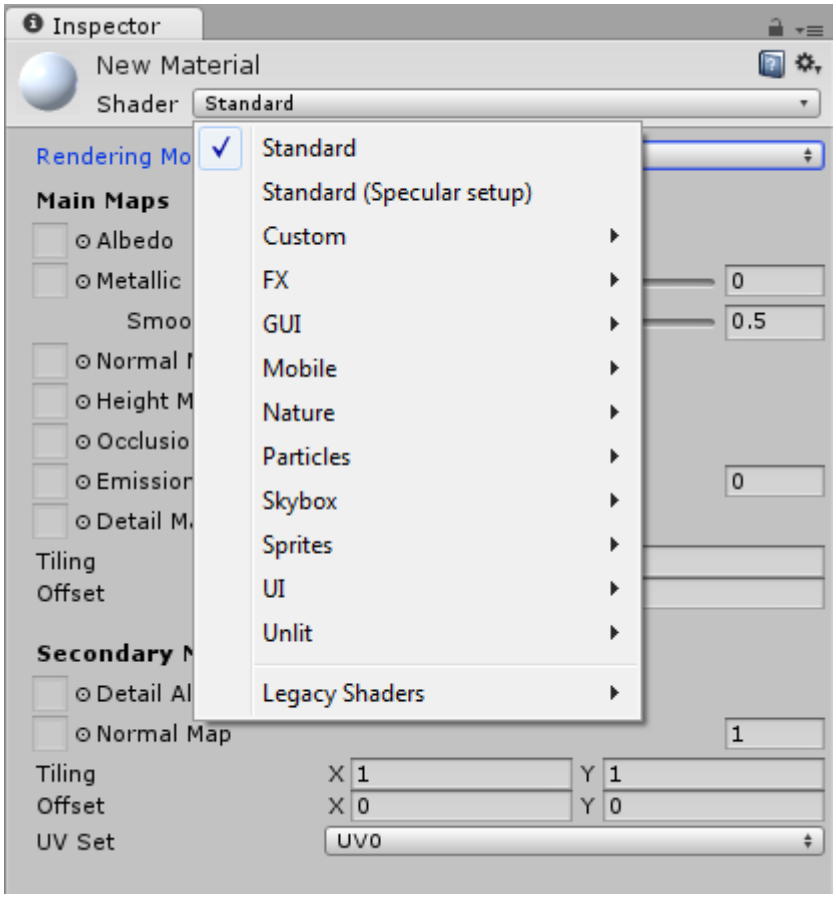

图 8-1 Unity 内置着色器

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。你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，请参阅 *8.2 使用局部立方体贴图实现反射* on page 8-121。

在 Unity 中，编写着色器的方法通常有两种：

表面着色器
当处理光线和阴影时，通常使用表面着色器。Unity 会为你执行与光照模型相关的处理工作，以便你编写出更紧凑的着色器。

顶点和片段着色器
顶点和片段着色器最为灵活，但是你必须自己处理所有事项。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，你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。

8.1.2 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 `Shader`，是指着色器所在的 *路径/名称* 。当你设置材质时，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拉菜单中**自定义**着色器类别的下方。

`Properties{}` 块列出了在检视器中可见的着色器参数，以及你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。Unity 渲染网格时，它将查找要使用的着色器，并选择可在显卡上运行的第一个子着色器。因为不同的显卡支持不同的着色器模型，这样可以确保着色器正确执行。此特性非常重要，因为 GPU 硬件和 API 正在不断发展。例如，你可以 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，另外，可以在单独的子着色器中编写替代着色器来适配那些只支持 OpenGL ES 2.0 及更低版本的显卡。

每经过一个 `Pass` 相当于对物体渲染一次。着色器可以包含一个或者多个 pass。你可以在旧硬件上使用多个 pass，来实现各种特效。

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”内置着色器。

Cg 程序片段在 `CGPROGRAM` 和 `ENDCG` 之间编写。

8.1.3 编译指令

通过 `#prama` 声明来传递编译指令。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` 用来设置着色器版本。如果着色器变得过大，将得到下面这种类型的错误：

```
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，你必须添加 `#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` 指令。

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，除非你使用 `#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。

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

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

8.1.4 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。

你可以在 `C:\Program Files \Unity\Editor\Data\CGIncludes` 中找到可用的 include 语句。例如，在 include 语句 `UnityCG.cginc` 中，你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。你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维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前，请先检查镜头位置、投射参数或灯光参数是否已经包含在 include 的矩阵信息里面。

为提高性能，有时更适合在 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 为我们提供一些内置统一变量的复合矩阵的原因。下表显示了一些重要的 Unity 着色器内置值：

表 8-1 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

表 8-1 重要的 Unity 着色器内置值 (续)

内置统一变量	说明
UNITY_MATRIX_IT_MV	当前模型 * 视图矩阵的逆转置矩阵
_WorldSpaceCameraPos	世界坐标空间的镜头位置
_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）

8.1.5 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。

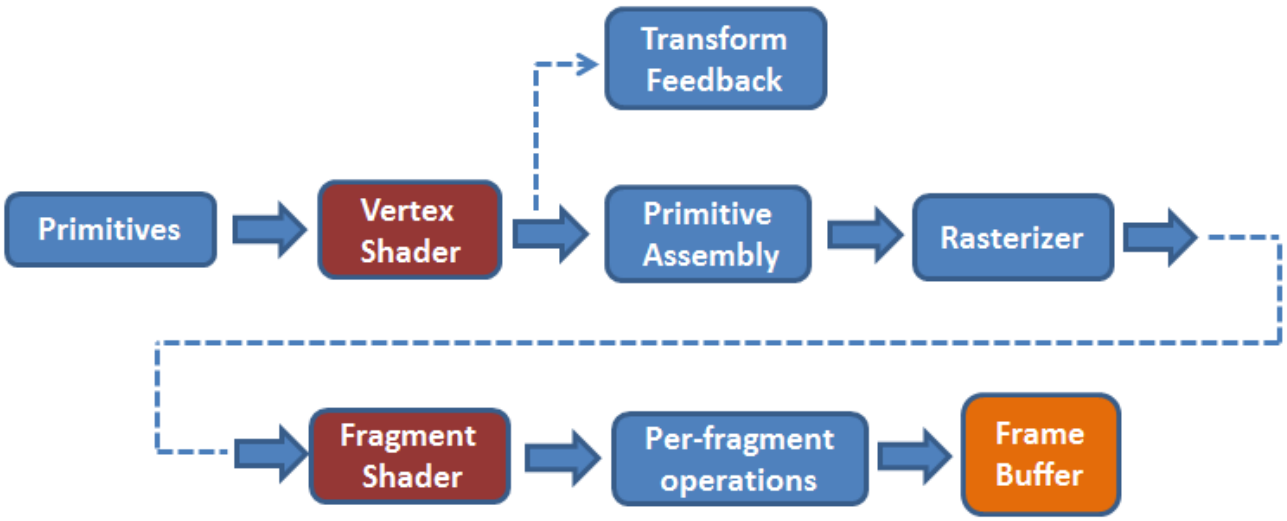

图 8-2 OpenGL ES 3.0 可编程管道

图元

在图元阶段，流水线在通过顶点、点、线和多边形描述的几何图元上运行。

顶点着色器

作为一个通用的可编程方案，顶点着色器用来处理顶点数据。顶点着色器用来处理顶点转换和光线信息等。

图元装配

在图元装配期间，顶点将装配至几何图元中。产生的图元将剪切至裁剪区域并送到光栅化阶段。

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，图元将转换为一组二维片段，这些片段将被发送至片段着色器。

变换反馈

变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函数和测试：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。

8.1.6 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 *8.1.2 着色器结构* on page 8-110。

如果情况顺利，顶点着色器可以输出转换后的位置。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，局部空间中的顶点坐标和纹理坐标作为输入片段输入到顶点着色器中。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 4x4 矩阵，因此你必须通过 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，你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处理，但是你也可以在顶点着色器中处理一些效果。顶点置换贴图也称为置换贴图，是一项众所周知的技术，能够让你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，你必须添加编译指令 `#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。如果你强制编辑器使用 OpenGL 着色器，还必须添加 `#pragma glsl` 指令。如果你未声明此指令，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，你还可以使用“过程动画”技术将顶点做成动画。你可将支持你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动画。

8.1.7 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，你仅声明了顶点属性位置和纹理坐标。

你可以使用下列语义定义更多的属性用作输入，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。你必须为在着色器阶段之间传递的所有变量指定语义。

如果你使用了不正确的语义，如 `float3 tangent2 : TANGENTIAL`，则会出现下列类型的错误：

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

为了提高性能，请仅在你确实需要的输入结构中指定参数。Unity 拥有一些预定义的输入结构，适用于最常用的输入参数组合：`appdata_base`、`appdata_tan` 和 `appdata_full`。`UnityCG.cginc inlcude` 文件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，你无需声明结构，只需声明 include 文件。

8.1.8 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，但是你可以添加其他量度。

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` 保留用于 UV，而 `TEXCOORD1` 保留用于光照贴图 UV。但是理论上来说，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，如果你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 `V1`、`V2` 和 `V3` 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 λ1、λ2 和 λ3 计算像素坐标，以作为顶点坐标的线性插值。

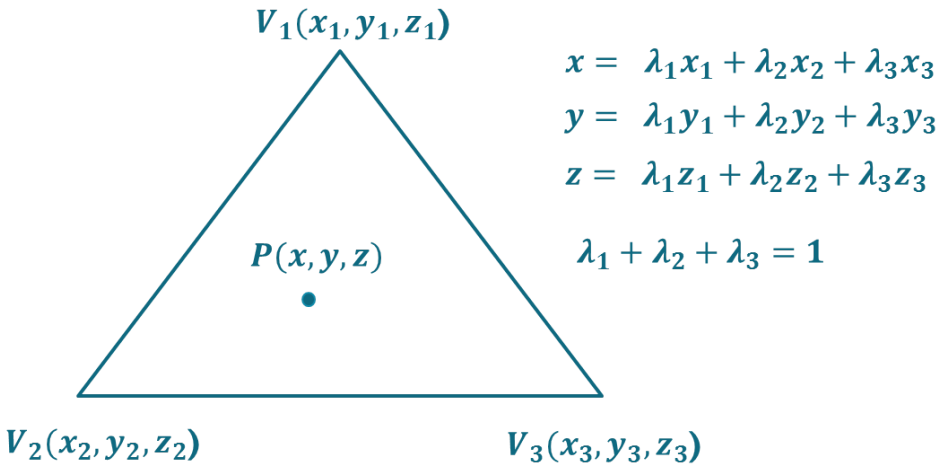

图 8-3 使用重心坐标进行线性内插

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。

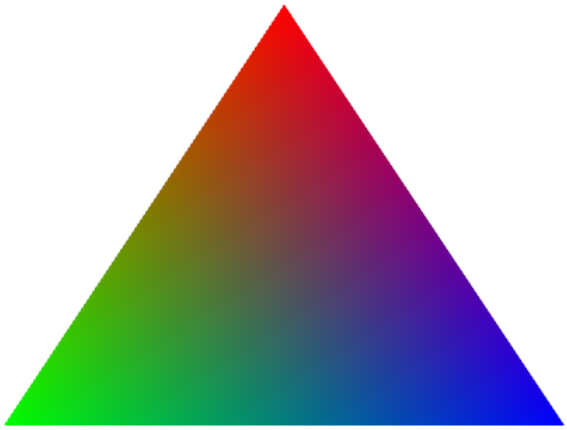

图 8-4 色彩内插

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，如果你拥有一个平面，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。

你必须谨慎使用变量，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。

8.1.9 片段着色器

片段着色器是图元光栅化后的图形流水线阶段。

对于图元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，因此你必须注意在片段着色器中执行的运算量。

在片段着色器中，你除了可以获取窗口空间的片段坐标外，你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 *8.1.2 着色器结构* on page 8-110 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。你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。

8.1.10 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。

你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` 块能够让你在**检视器**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器中，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**材质检视器**中。

下图显示了材质检视器中的属性：

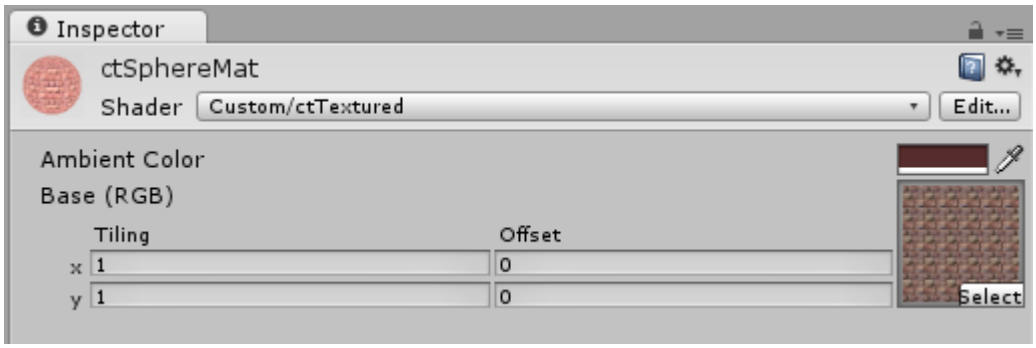

图 8-5 材质检查器中的属性

当你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，因为你可以在运行时以交互方式更改数据并查看效果。

你可以将下列类型的变量放入 `Properties` 块中：

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

如果在需要之前计算的数据，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 8-2 将材质相关数据传递至着色器的常用方法

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

在下列代码中，主镜头渲染完场景前，辅助镜头 shwCam 会渲染阴影到将与主镜头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。阴影镜头投影矩阵 (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;
}
```

8.1.11 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，你可以用片段着色器的输出使待调试的值形象化。然后，你必须解析产生的图像。

下图显示了应用于 *8.2 使用局部立方体贴图实现反射* on page 8-121 中地板反射面的着色器 ctReflLocalCubemap.shader 的输出效果：

图 8-6 带有反射的棋牌室

在下列片段着色器中，输出颜色已替换为归一化的局部修正反射向量：

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

它使归一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归一化的局部修正反射向量的结果：

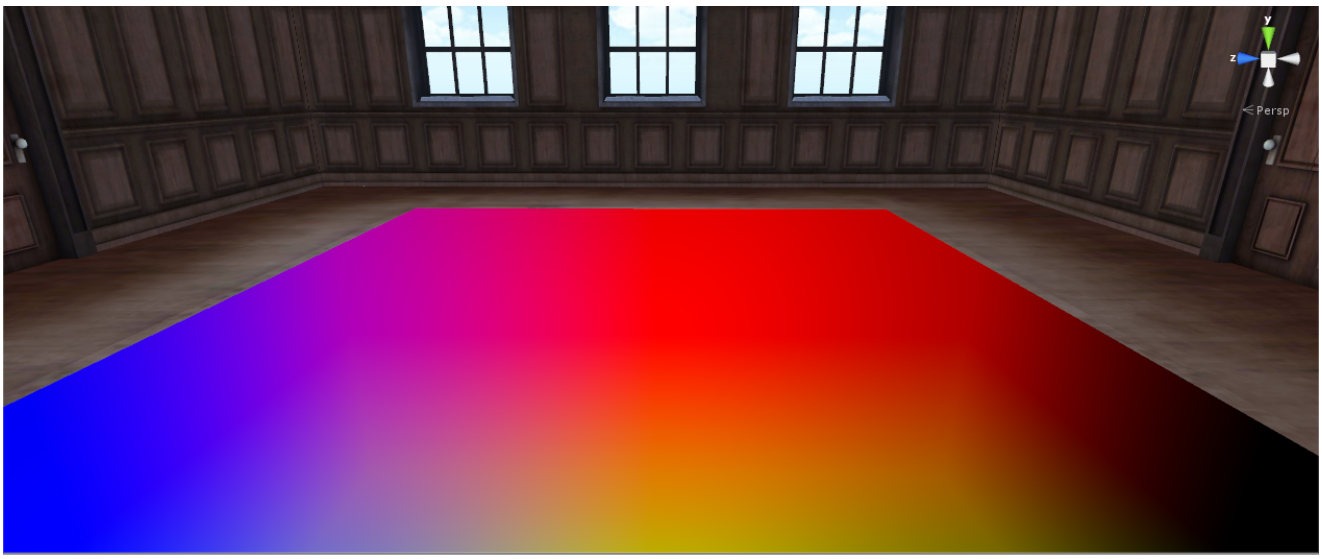

图 8-7 使用多种色彩调试着色器

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，你只能返回归一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，输出只有主要来自镜头上方屋顶的反射。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 8-8 使用单一色彩调试着色器

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。

8.2 使用局部立方体贴图实现反射

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

本部分包含以下子部分：

8.2.1 反射实现方法的演变

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。

率先出现的解决方案便是球形映射。此方法模拟对象上的反射或光照时，无需使用会消耗大量算力的射线跟踪或光照计算。

下图显示了球体上的环境贴图：

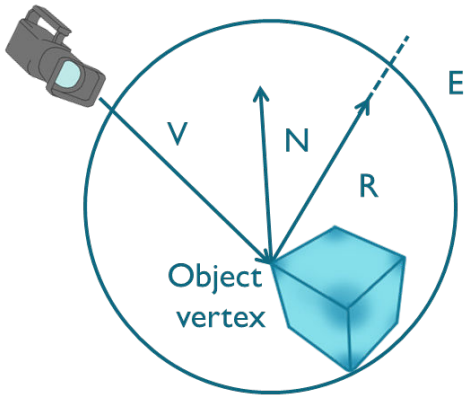

图 8-9 球体上的环境贴图

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 8-10 球面二维贴图方程

此方法有多种劣势，但是主要问题在于，将图片映射至球体时会出现失真。1999 年，实现了立方体贴图与硬件加速的结合使用。立方体贴图解决了球形映射的图像失真、视角依赖性以及计算效率低等问题。

下图显示了展开的立方体：

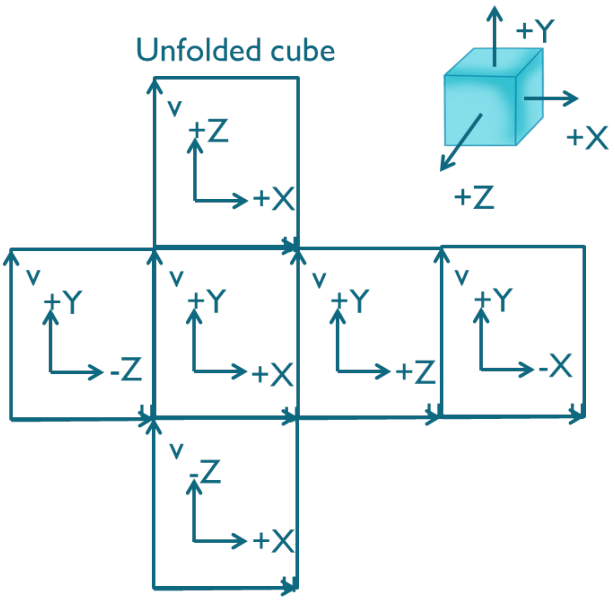

图 8-11 展开的立方体

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形纹理，或展开为单个纹理的六个区域。可以使用六个不同的镜头方位从指定位置渲染场景，以生成立方体贴图，每个立方体表面以一个 90 度的视锥体来展现。源图像将被直接采样。然后重新采样到过渡环境贴图中，这样不会引入失真。

下图显示了无限反射：

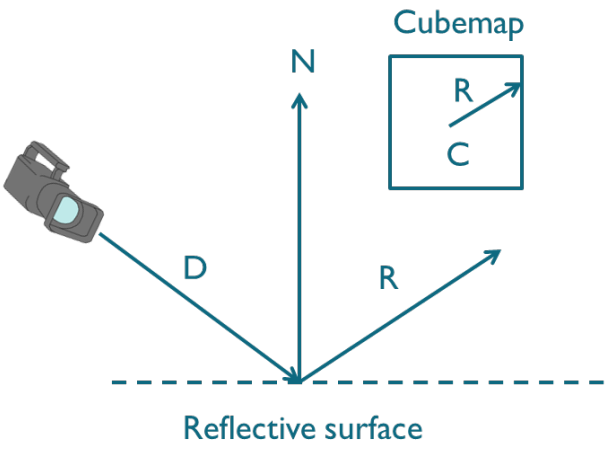

图 8-12 无限反射

若要使用立方体贴图来实现反射，请评估反射矢量 `R` 并用该矢量从立方体贴图 `_Cubemap` 中获取纹素，方法是使用可用纹理查找函数 `texCUBE()`：

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

法线 N 和视线矢量 D 可从顶点着色器传递至片段着色器。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处环境中正确复制反射。这种简单有效的方法主要用于室外光照，例如增加天空的反射效果。

下图显示了不正确的反射：

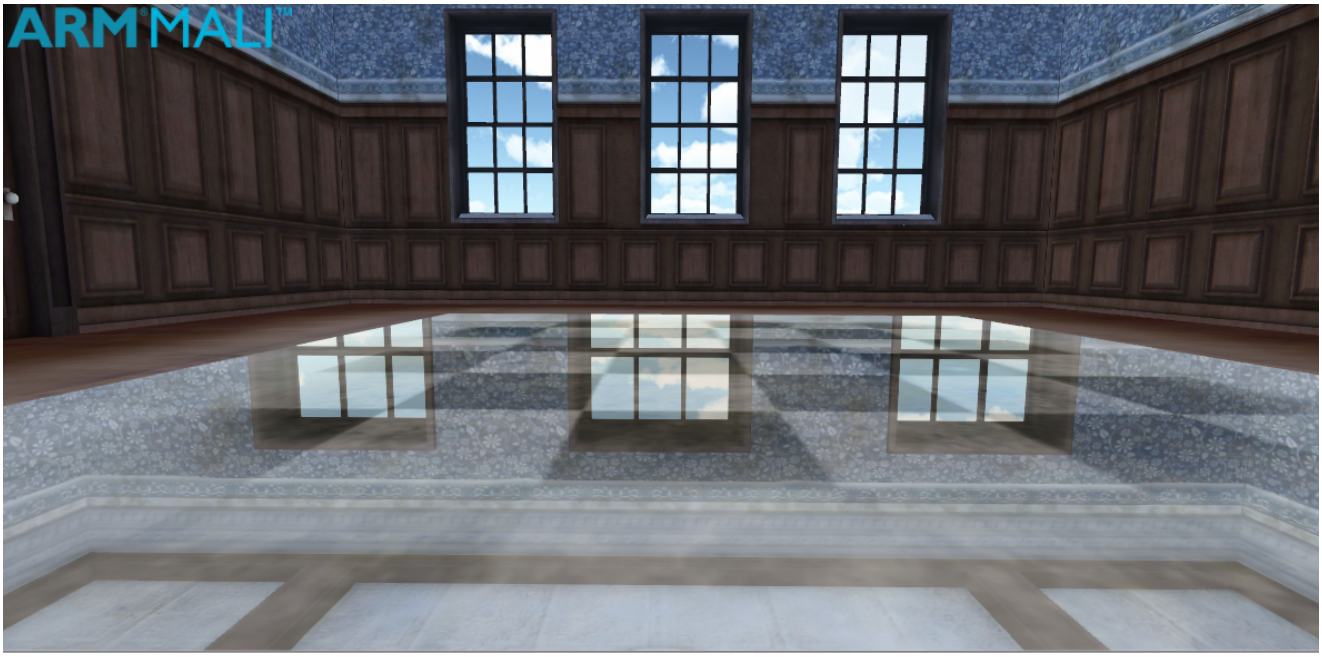

图 8-13 不正确的反射

如果在局部环境中使用此方法，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，如果你从同一角度看一块反光的地板，即便你在走动也将始终看到相同的反射。反射矢量始终相同，并且该表达式始终得出相同的结果。这是因为视线矢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。

8.2.2 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 Randima Fernando（丛书编辑）撰写的*《GPU 精粹：实时图形编程方法、提示和技巧》*中进行了介绍。

下图显示了使用包围球体进行局部修正：

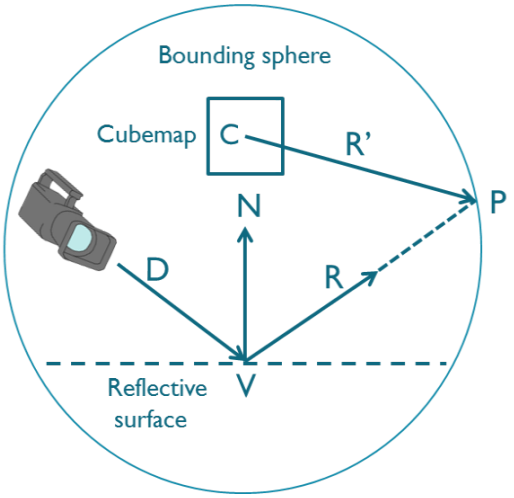

图 8-14 使用包围球进行局部修正

包围球体用作界定待反射场景的代理。它不使用反射矢量 R 从立方体贴图中获取纹理，而是使用新矢量 R’。为构建此新矢量，你需要在反射矢量 R 方向上从局部点 V 发出的射线的包围球体中找到相交点 P。从生成立方体贴图的立方体贴图中心 C 到相交点 P，创建一个新矢量 R’。然后使用此矢量从立方体贴图获取纹理。

```
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，此过程比较复杂。

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*。此方法使用盒体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。有关更多信息，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：

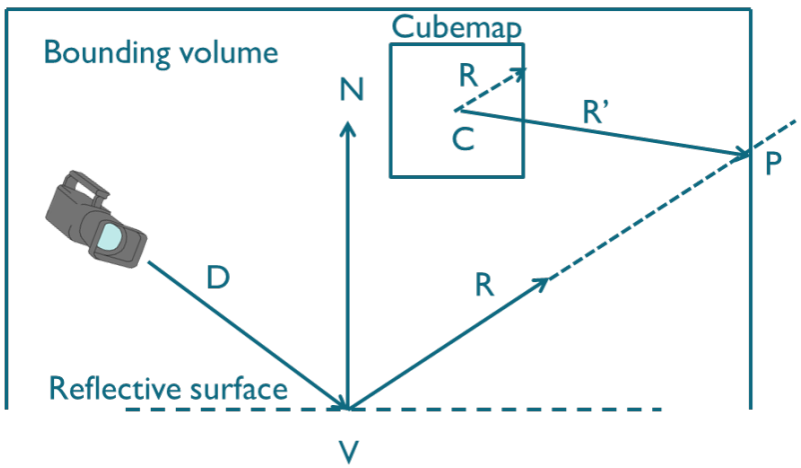

图 8-15 使用包围盒进行局部修正

Sebastien Lagarde 于 2012 年发布的一项新作业即使用这种新方法，使用多个立方体贴图和一种算法评估了每个立方体贴图的作用并将其在 GPU 上有效混合，从而模拟了更复杂的环境镜面反射光照。请参阅 *http://seblagarde.wordpress.com*

表 8-3 无限和局部立方体贴图之间的差异

无限立方体贴图	局部立方体贴图
· 主要用于室外，代表远距离环境的光照。 · 立方体贴图位置不重要。	· 代表有限局部环境的光照。 · 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会正确显示。 · 立方体贴图本质上是无限的，必须运用局部修正来对此加以调整，才能使其适应局部环境。

下图显示了用局部立方体贴图生成正确反射的场景。

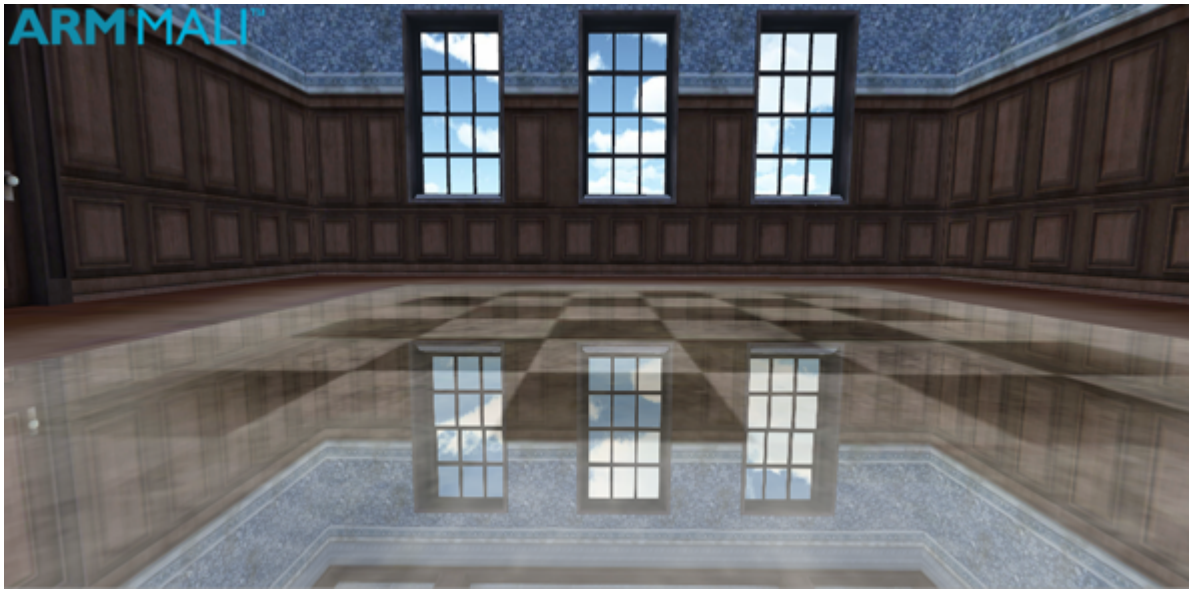

图 8-16 正确反射

8.2.3 着色器实现

本节介绍使用局部立方体贴图实现反射的着色器。

顶点着色器计算三个维度，然后作为内插值传递至片段着色器：

· 顶点位置。
· 观察方向。
· 法线。

这些值位于世界坐标中。

下列代码显示了着色器使用局部立方体贴图实现的反射，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;
}
```

体积盒中的相交点以及反射矢量在片段着色器中计算。你将构建新的局部修正反射矢量并用该矢量从局部立方体贴图中获取反射纹理。然后，你将结合纹理和反射来生成输出颜色：

```
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 localCorrReflDirWS = intersectPositionWS - _EnviCubeMapPos;
    // Lookup the environment reflection texture with the right vector.
    reflColor = texCUBE(_Cube, localCorrReflDirWS);
    // 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，属性 `_BBoxMax` 和 `_BBoxMin` 是包围体的最大点和最小点。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递至着色器：

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

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

        // Min and max BBox points in world coordinates
        Vector3 BMin = centerBBox - bboxLenght/2;
        Vector3 BMax = centerBBox + bboxLenght/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递至着色器，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
        }
    }
}
```

计算包围体相交点的算法必须使用参数，该参数用来表示从局部位置或片段反射的射线。有关射线与盒体相交算法的说明，请参阅 *8.2.5 射线与盒相交算法* on page 8-131。

8.2.4 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的。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。你可从 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，你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 *8.2.6 用于使编辑器脚本生成立方体贴图的源代码* on page 8-134。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。

你必须将此工具的脚本放在 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”菜单中启动立方体贴图制作工具。
3. 提供立方体贴图和镜头渲染位置。
4. 如果你计划对立方体贴图应用过滤，请选择性地保存每张图像。

下图显示了立方体贴图制作工具交互界面：

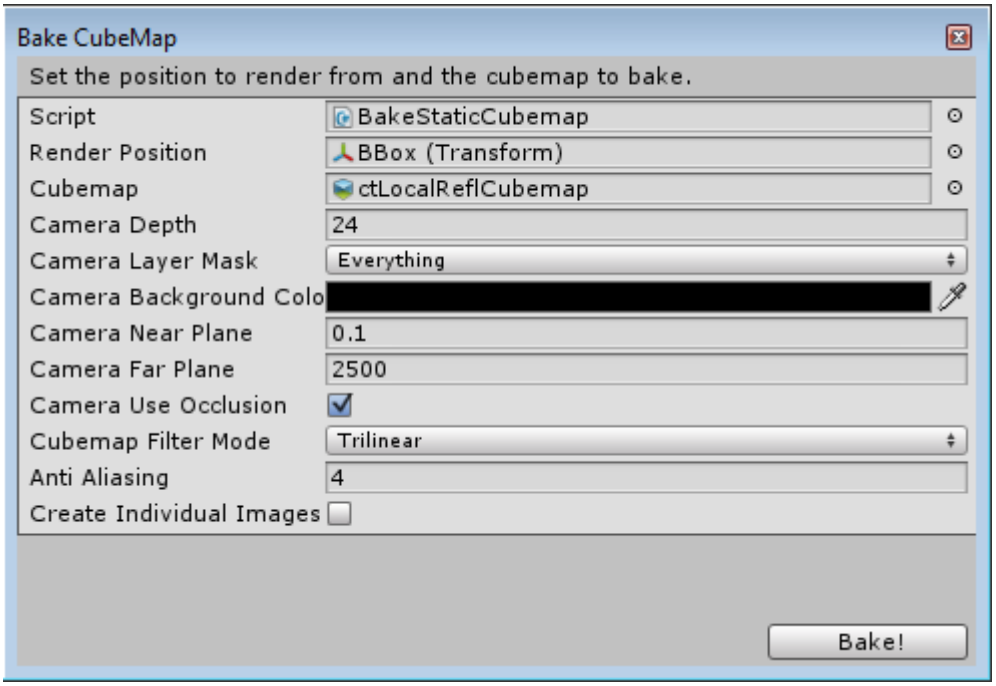

图 8-17 烘焙 CubeMap 工具界面

你可以使用 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**下拉菜单中有许多不同的过滤选项。选择你需要的过滤设置，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。如果过滤结果不符合你的设想，你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，在立方体贴图制作时保存独立图像。
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。撤销操作时不被允许的，而这可以让你在使用过滤器进行试验时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 立方体贴图表面指数间的对应关系。

表 8-4 CubeMapGen 和 Unity 立方体贴图表面指数间的对应关系

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 8-18 CubeMapGen

下图显示了在应用高斯滤波以获得*霜冻*效果后的 CubeMapGen：

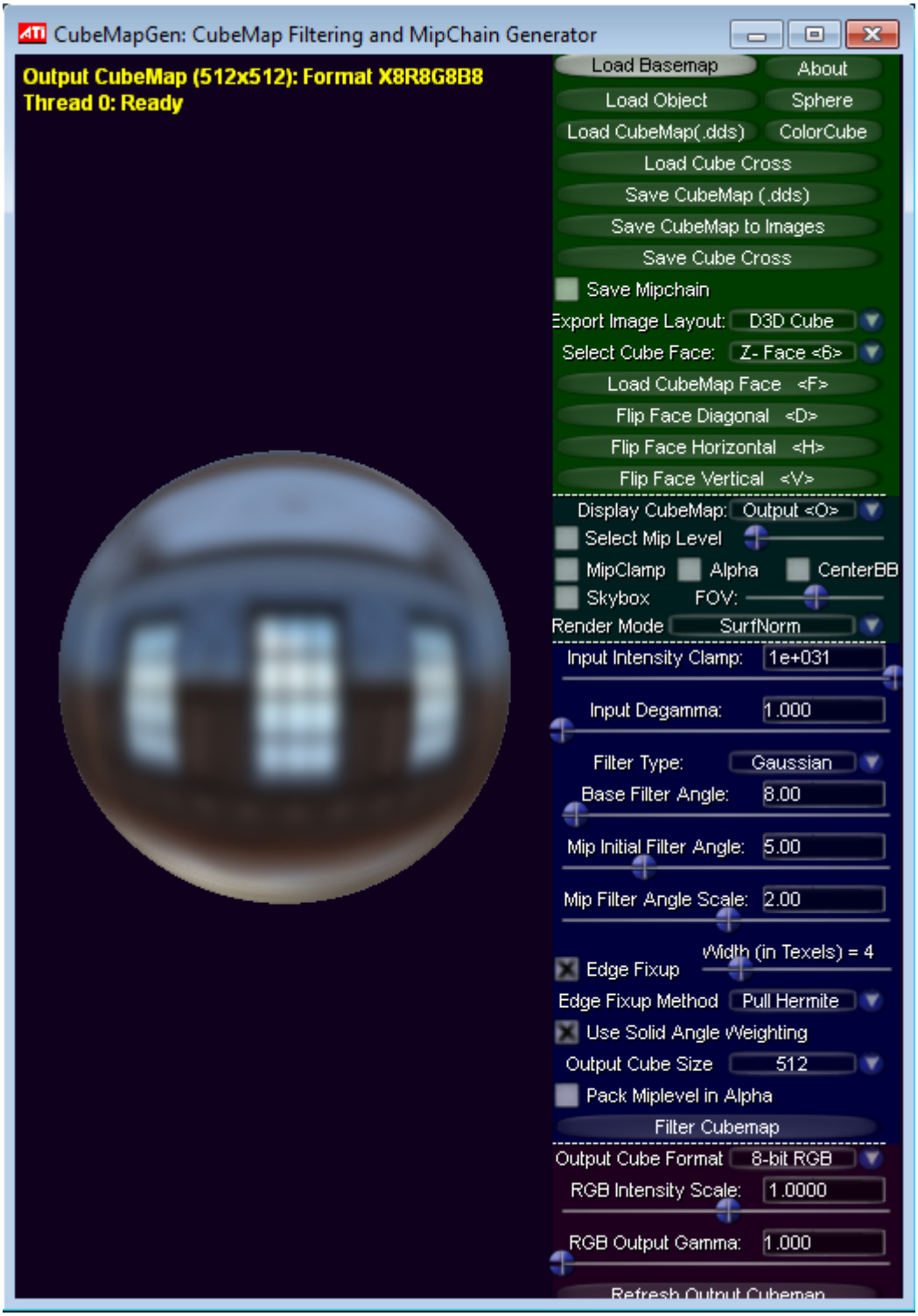

图 8-19 CubeMapGen 演示霜冻效果

下表显示了与高斯滤波器结合使用以达到霜冻效果的过滤器参数。

表 8-5 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 8-20 霜冻视效的反射

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。

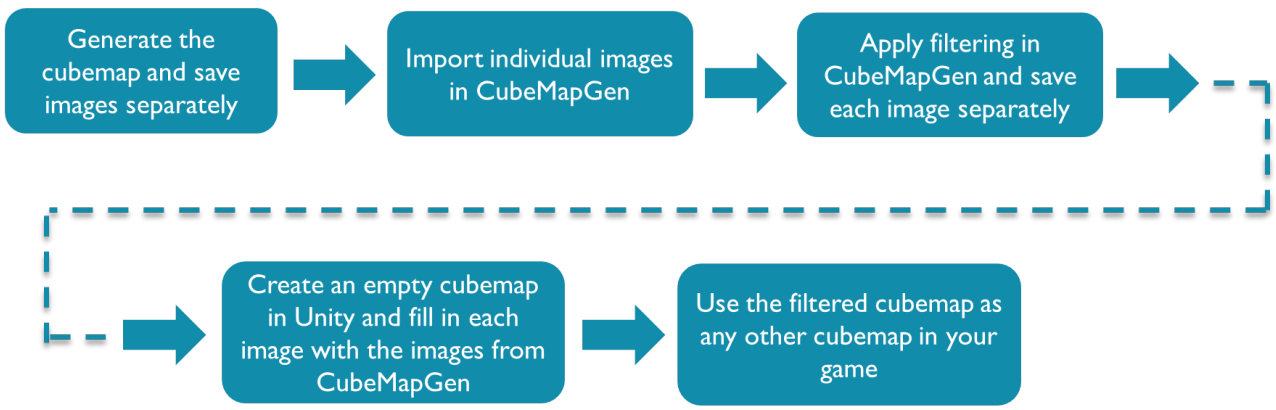

图 8-21 立方体贴图过滤工作流程

8.2.5 射线与盒相交算法

本节介绍射线与盒相交算法。

下图显示了含直线方程的图形：

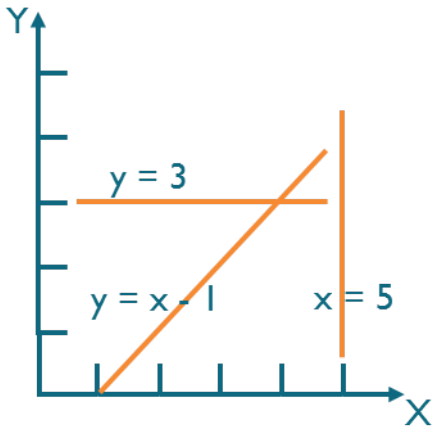

图 8-22 直线方程图

直线方程

```
y = mx + b
```

该方程的向量形式为：

```
r = O + t*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：

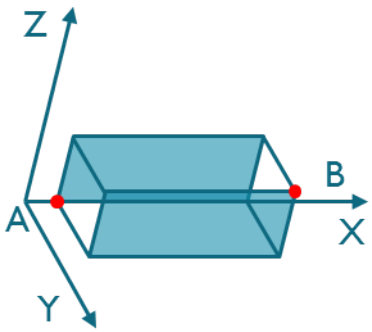

图 8-23 轴向包围盒

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：

```
x = Ax; y = Ay; z = AzX = Bx; y = By; z = Bz
```

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：

```
Ox + tx*Dx = Ax
```

你可以将解答方程写成：

```
tAx = (Ax – Ox) / Dx
```

以相同的方式求出这两个相交点的所有可能解答方程：

$$t_{Ax} = (Ax - Ox) / Dx$$
$$t_{Ay} = (Ay - Oy) / Dy$$
$$t_{Az} = (Az - Oz) / Dz$$
$$t_{Bx} = (Bx - Ox) / Dx$$
$$t_{By} = (By - Oy) / Dy$$
$$t_{Bz} = (Bz - Oz) / Dz$$

向量形式的方程为：

$$t_A = (A - O) / D$$
$$t_B = (B - O) / D$$

这可找出直线与平面（由立方体表面定义）相交的位置，但是它无法保证相交点位于立方体上。

下图以 2D 形式显示了射线与盒的相交点：

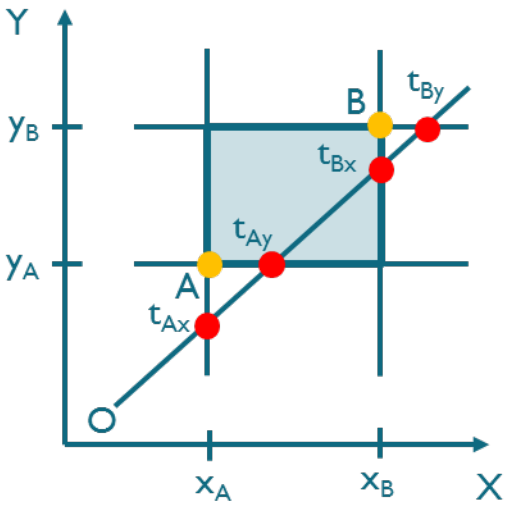

图 8-24 射线方盒相交二维表达

要找到哪个是与盒的相交点，你需要用参数 `t` 中较大的值来确认最小平面上的相交点。

```
tmin = (tAx> tAy) ? tAx: tAy
```

你需要用参数 `t` 中较小的值来确认最大平面上的相交点。

```
tmin = (tAx> tAy) ? tAx: tAy
```

如果你未得到任何相交点，也必须考虑这些情况。

下图显示了无相交点的射线与盒：

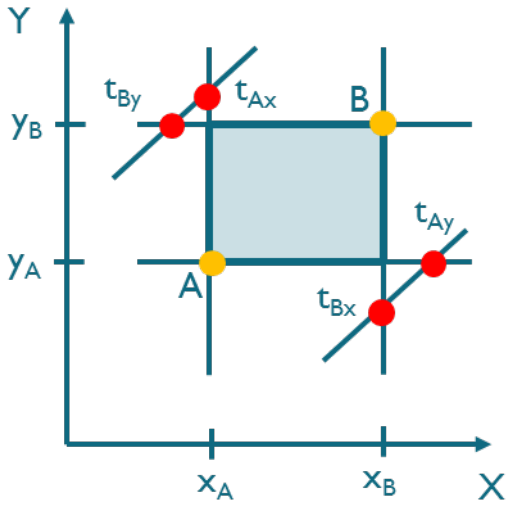

图 8-25 射线方盒不存在相交

如果你保证反射面已被 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：

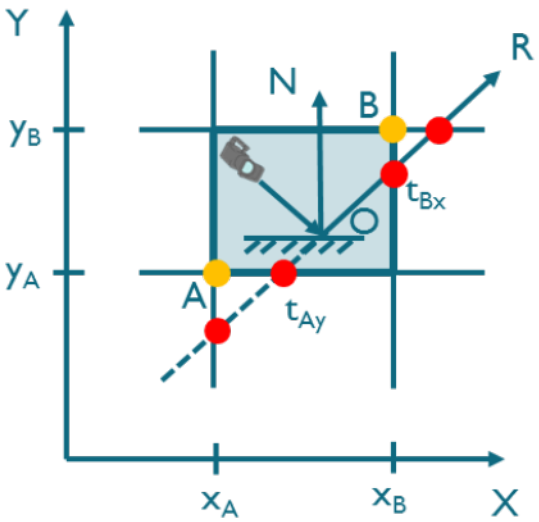

图 8-26 射线方盒在包围盒内相交

8.2.6 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!");
    }
}
```

8.3 组合反射

本节介绍组合反射。

本部分包含以下子部分:

8.3.1 关于组合反射

基于局部立方体贴图的反射方法可以使用静态局部立方体贴图来高效渲染高质量反射。但是，如果对象是动态的，静态局部立方体贴图便不再有效，该方法也无法发挥作用。

你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。

下图显示了静态和动态几何体的反射组合:

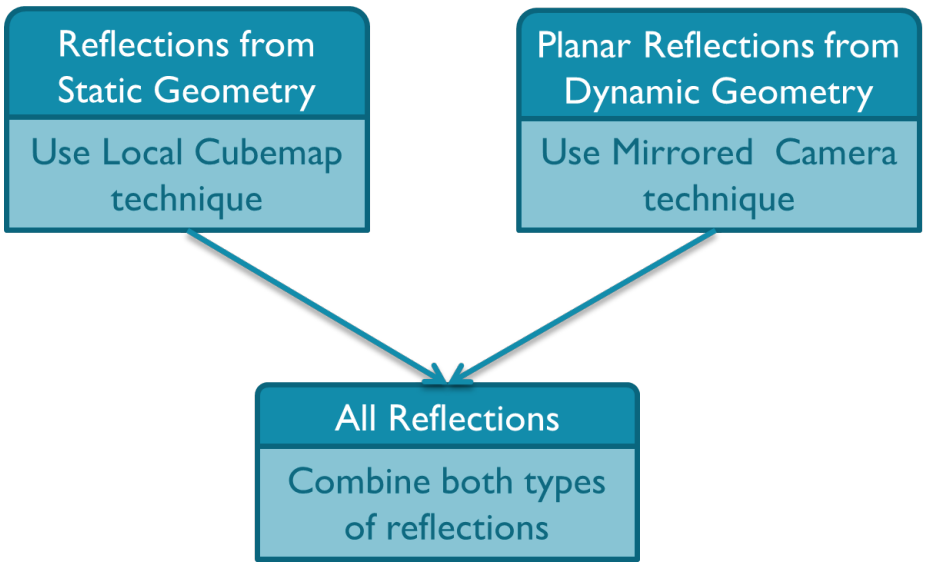

图 8-27 静态和动态几何体的反射组合

如果反射表面是平面，你可以使用镜像镜头生成动态反射。

要创建镜像镜头，可计算在运行时渲染反射的主镜头的位置和方向。

相对于反射平面，对主镜头的位置和方向进行镜像。

下图显示了镜像镜头方法:

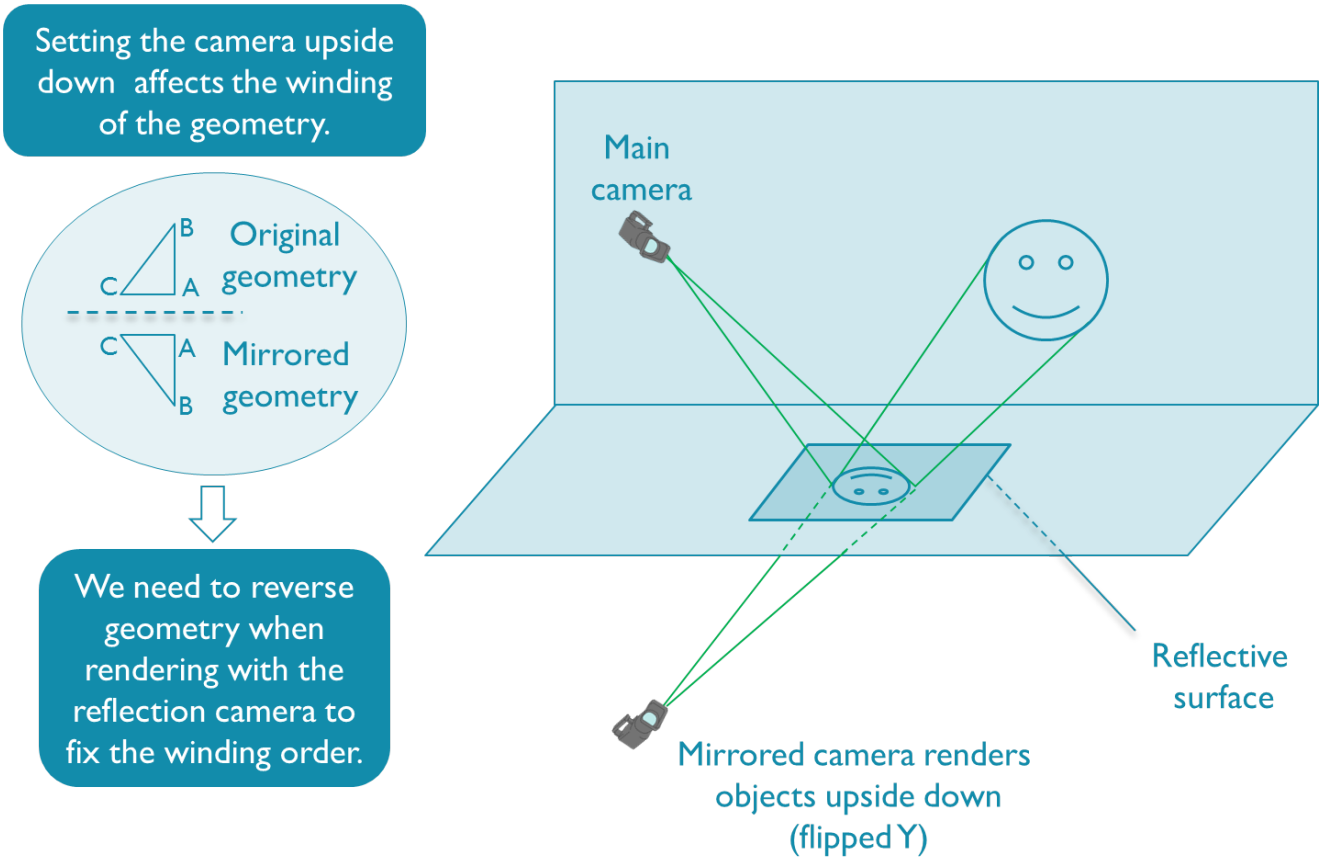

图 8-28 用于渲染平面反射的镜像镜头方法

在镜像过程中，新反射镜头的轴最终会处于相反的方向。与现实中的镜子一样，左右的反射被颠倒。这意味着反射镜头使用相反的卷绕方式渲染几何体。

要正确渲染几何体，你必须逆转几何体卷绕方式，然后再渲染反射。完成反射渲染后，应恢复原始卷绕。

下图显示了设置镜像镜头和渲染反射所需的步骤序列：

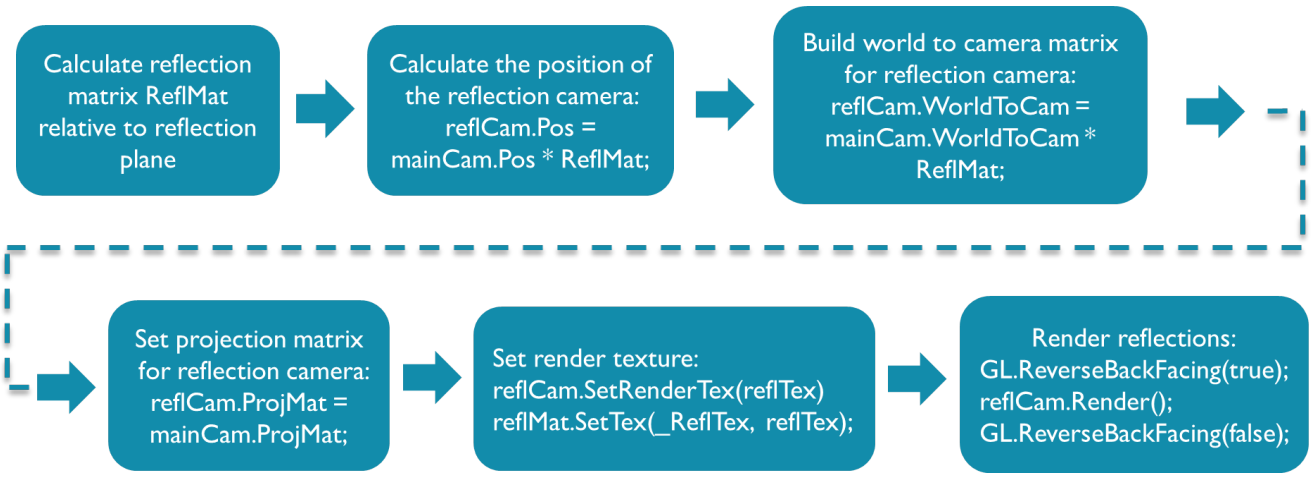

图 8-29 设置镜像镜头和渲染反射的主要步骤

构建镜像反射变换矩阵。使用此矩阵可以计算反射镜头的位置以及世界至镜头变换矩阵。

下图显示了镜像反射变换矩阵：

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

$$n_x = planeNormal.x$$
$$n_y = planeNormal.y$$
$$n_z = planeNormal.z$$
$$n_w = -dot(planeNormal, planePos)$$

图 8-30 镜像反射变换矩阵

将反射矩阵变换应用到主镜头的位置和世界至镜头矩阵。这将为你提供反射镜头的位置和世界至镜头矩阵。

反射镜头的投影矩阵必须和主镜头的投影矩阵相同。

反射镜头将反射渲染到纹理。

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 Mipmap 贴图。
- 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。

你可以从以下网址找到镜像镜头的脚本示例：*http://wiki.unity3d.com/index.php/File:MirrorReflection.png*。

8.3.2 组合反射着色器的实现

你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，你必须修改 *8.2.3 着色器实现* on page 8-125 中提供的着色器代码。

着色器必须融合在运行时通过反射镜头渲染的平面反射。要达到此目的，来自反射镜头的纹理 `_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建函数 `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：

- 向反射矢量应用局部修正。
- 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组合使用静态环境反射（局部立方体贴图方法）与动态平面反射（运行时使用镜像镜头方法渲染）：

```
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 - _EnviCubeMapPos;

    // 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。

在属性块中声明 `_ReflectionTex`。这使你能够查看它在运行时的显示效果，从而帮助你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矢量的最后一个分量。你可以使用 Unity 内建函数 `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。组合以下几项：

- 反射颜色。
- 反射表面的纹理颜色。
- 环境颜色分量。

8.3.3 组合远距离环境的反射

在渲染静态和动态对象的反射时，可能还必须考虑来自远距离环境的反射。例如，局部环境中某扇窗户外的天空造成的反射。

在这种情形中，你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方法。
- 来自动态对象的平面反射，它使用的是镜像镜头方法。
- 来自天空盒的反射，它使用的是标准立方体贴图方法。反射矢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，可使用反射矢量 `reflDirWS` 从天空盒立方体贴图获取纹素。将天空盒立方体贴图纹理作为统一变量传递到着色器。

说明

请勿应用局部修正。

为确保天空盒仅通过窗户显示，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 1；如果没有几何体或几何体完全透明，分配值 0。例如，在 alpha 通道中使用 0 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 *8.3.2 组合反射着色器的实现* on page 8-139 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。

下图显示了不同类型的反射组合：

图 8-31 不同类型的反射组合

8.4 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。

本部分包含以下子部分:

8.4.1 关于基于局部立方体贴图的动态软阴影

在你的场景中，既存在移动的对象，也有房间等静态环境。通过使用此方法，你不必在每一帧将静态几何体渲染到阴影贴图。这可让你通过纹理来表示阴影。

立方体贴图可以相当真实地展现许多种静态局部环境，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。
· 镜头。
· 动态对象。

通过用立方体纹理来展现整个房间，你可以在一个片段着色器内访问环境的任意纹素。例如，这意味着太阳可以在任意位置上，你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局部环境的光线量。在你的场景中，将立方体贴图纹理附加到片段着色器上，这些着色器渲染你要为其添加阴影的静态和动态对象。

8.4.2 生成阴影立方体贴图

如果你想要将来自环境外部光源的阴影应用于某个局部环境，则应从此处着手开始。例如，房间、洞穴或笼子。

此方法类似于基于局部立方体贴图的反射。有关更多信息，请参阅 *8.2 使用局部立方体贴图实现反射* on page 8-121。

创建阴影立方体贴图的方式与创建反射立方体贴图相同，但你还必须添加 alpha 通道。Alpha 通道或透明度可以表示进入局部环境的光线量。

算出你要从哪个位置来渲染立方体贴图的六个面。在大多数情形中，这是局部环境包围盒的中心。你需要通过此位置生成立方体贴图。还必须将此位置传递到着色器，从而计算局部修正矢量，以便从立方体贴图获取正确的纹素。

决定了立方体贴图中心的位置后，你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，你可以使用 RGB 通道存储环境颜色用于彩色阴影，如有色玻璃、反射或折射。

8.4.3 渲染阴影

在世界空间中构建从顶点或片段到一个/多个光源的矢量 P_iL，然后使用此矢量获取立方体贴图阴影。

在获取每个纹素前，必须对 P_iL 矢量应用局部修正。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，你必须计算片段至光源矢量与环境包围盒的相交点。使用此相交点构建从立方体贴图原点位置 C 到相交点 P 的另一矢量。这可为你提供用于获取纹素的最终矢量 CP。

你需要下列输入参数来计算局部修正：

- _EnviCubeMapPos - 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。
- Pi - 世界空间中的片段位置。
- PiL - 世界空间中归一化的片段至光源矢量。

计算输出值 CP。这是修正后的片段至光源矢量，你要用它从阴影立方体贴图获取纹素。

下图显示了片段至光源矢量的局部修正。

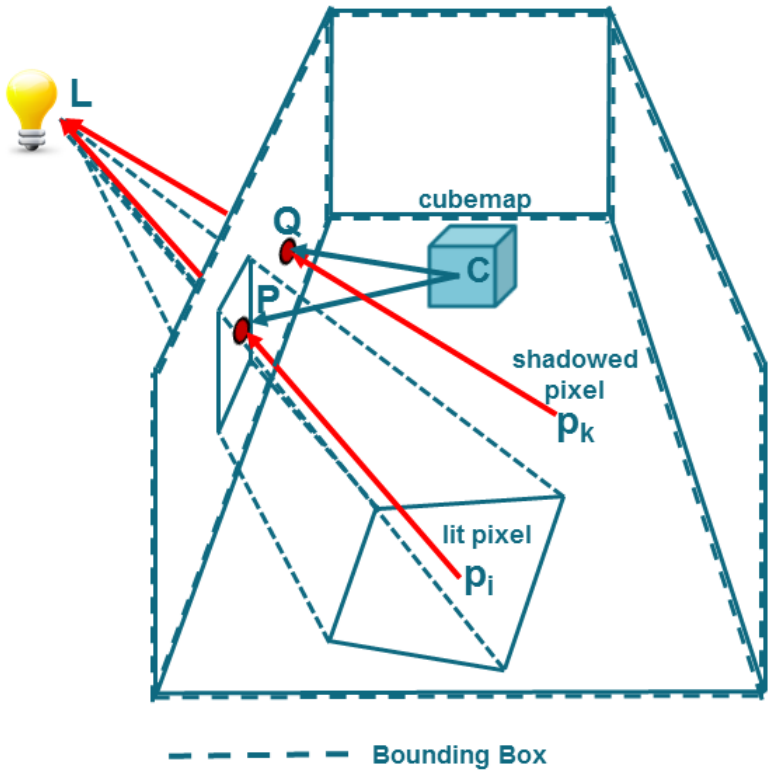

图 8-32 片段至光源矢量的局部修正

下列示例代码演示了如何正确计算 CP 矢量：

```
// 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 矢量从立方体贴图获取纹素。纹素的 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象棋室：

图 8-33 具有硬阴影的象棋室

此方法可以在场景中生成有效的阴影，但你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

立方体贴图阴影方法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被错误地照亮。

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，可检查法线矢量与片段至光源矢量 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。

下面这段代码可以进行检查：

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

以上代码导致每个三角形从照亮生硬地切换至阴影。若要平滑过渡，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 是世界空间中归一化的片段至光源矢量。
- N 是表面在世界空间中的法线矢量。

下图显示了背面处于阴影中的象棋室：

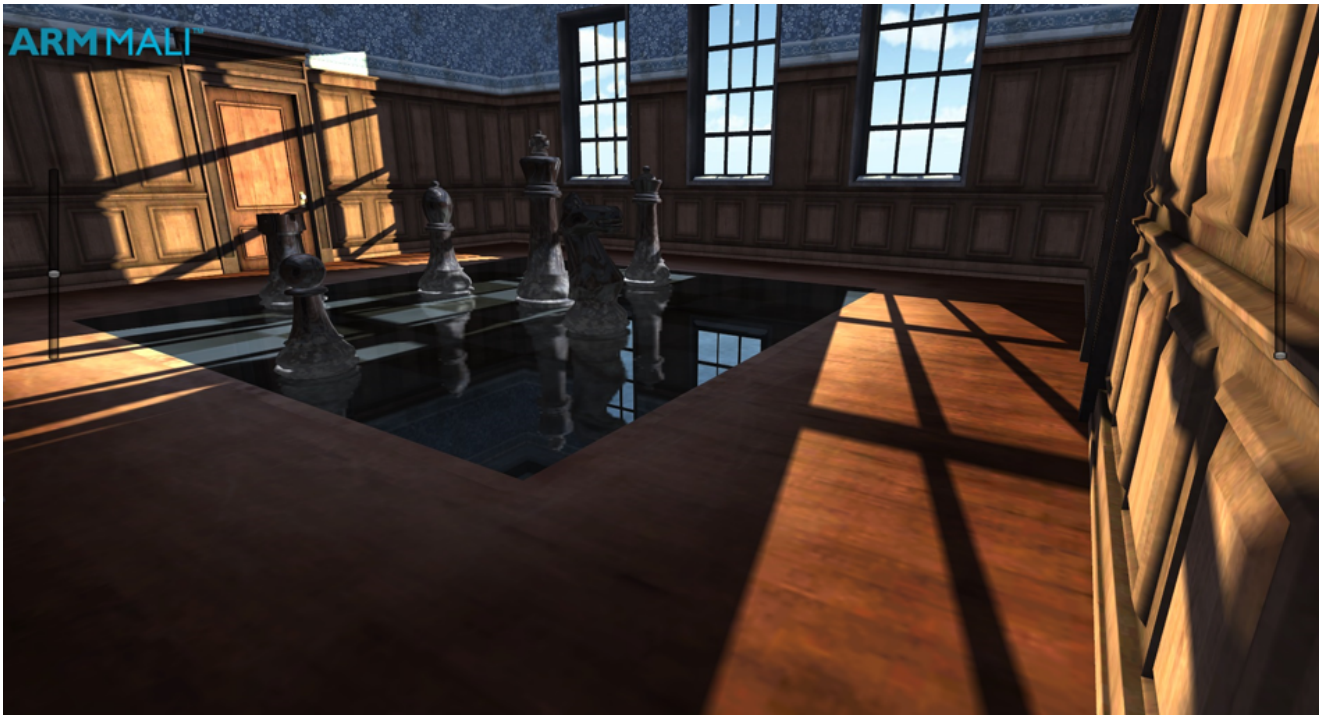

图 8-34 背面处于阴影中的象棋室

平滑

此阴影方法可以在你的场景中提供逼真的软阴影。

1. 生成 Mipmap 贴图，并为立方体贴图纹理设置三线性过滤。
2. 测量片段至相交点矢量的长度。
3. 将该长度乘以一个系数。

该系数是环境中到 Mipmap 贴图层级数的最大距离的归一化值。你可以根据包围体和 Mipmap 贴图层级数自动计算它。必须按照具体的场景自定义该系数。这可让你调整相关的设置，使其适合你的环境，从而改善画面质量。例如，冰穴项目中使用的系数为 `0.08`。

你可以重新利用为局部修正进行计算时得出的结果。重新利用局部修正代码片段中的 `dist`，作为从片段位置到片段至光源矢量与包围盒相交点的线段的长度：

```
float texLod = dist;
```

将 `texLod` 乘以距离系数：

```
texLod *= distanceCoefficient;
```

要实现柔和度，请使用 Cg 函数 `texCUBElod()` 或 GLSL 函数 `textureLod()` 获取纹理的正确 Mipmap 贴图层级。

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;
```

此方法可为你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象棋室：

图 8-35 平滑阴影

8.4.4 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方法。这需要额外的工作，但值得一试，因为你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象棋室：

图 8-36 平滑阴影

下图显示了组合了平滑阴影和动态阴影的象棋室：

图 8-37 平滑阴影与动态阴影组合

8.4.5 立方体贴图阴影方法的结果

使用传统的方法时，渲染阴影的成本比较高，因为它需要从每个阴影投射光源的视角渲染一整个场景。此处介绍的立方体贴图阴影方法可以提高性能，因为它大部分是预烘焙的。

此方法还无需依赖输出分辨率。它在 1080p、720p 和其他分辨率下产生的画面质量相同。

柔和度过滤是在硬件中计算的，所以平滑几乎没有计算成本。阴影越平滑，此方法的效用越佳。这是因为较小的 Mipmap 贴图层级数产生的数据要少于传统的阴影贴图方法。传统的方法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方法获得的质量可能会超越你的预期。它可以提供逼真的柔和度，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方法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。

立方体贴图阴影方法没有闪光问题。即使你使用的分辨率远低于渲染目标所用的分辨率，也能得到稳定的边缘。你可以使用只有输出结果四分之一的分辨率，而且不会产生失真或多余的闪光。使用四分之一的分辨率还可节省内存带宽，因而能提升性能。

此方法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。如果你已清楚应在何处以及何时使用基于局部立方体贴图方法的反射，你就可以轻松在实现中实现此阴影方法。

—————— 说明 ——————

该方法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方法搭配使用。

————————————

下图显示了具有阴影的冰穴：

图 8-38 具有阴影的冰穴

下图显示了具有平滑阴影的冰穴：

图 8-39 具有平滑阴影的冰穴

下图显示了具有平滑阴影的冰穴：

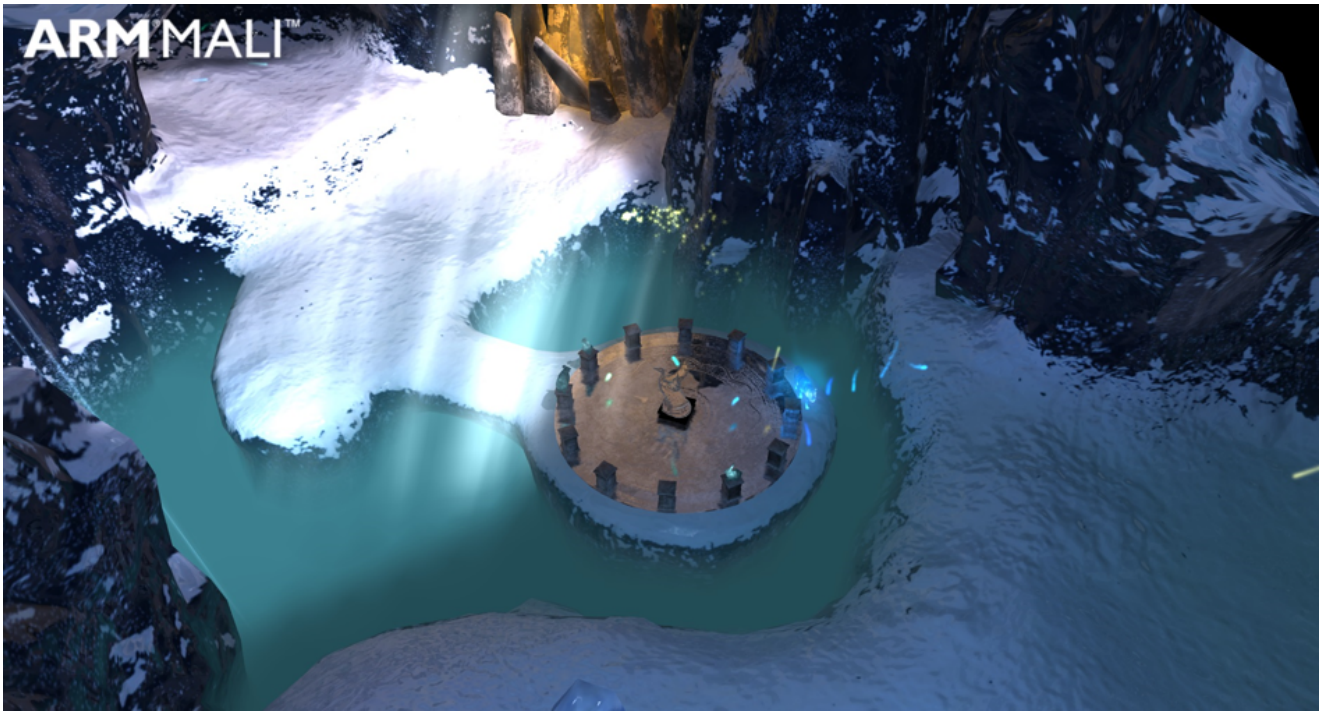

图 8-40 具有平滑阴影的冰穴

8.5 基于局部立方体贴图的折射

你可以使用局部立方体贴图实现高质量折射。也可以在运行时将它们与反射组合。

本部分包含以下子部分:

8.5.1 关于折射

游戏开发人员经常在寻求一些高效的方法，以期在游戏中实现惊艳的视觉效果。当你的目标平台是移动端时，这一点尤其重要，因为你必须仔细平衡资源才能实现理想性能。

折射是由于光线所穿过的介质发生变化而导致的光波方向上的变化。如果想做出格外逼真的半透明几何体，折射是必须要考虑到的重点特效。

折射率决定了光线进入材料后会发生多大程度的弯曲，或者说折射。折射的定义就是光线从一种介质（其折射率记为 n_1）进入另外一种介质（其折射率记为 n_2）时所发生的弯曲。

斯涅尔定律可用来计算折射率和入射角正弦与折射角正弦之间的关系。

下图显示了斯涅尔定律和折射现象:

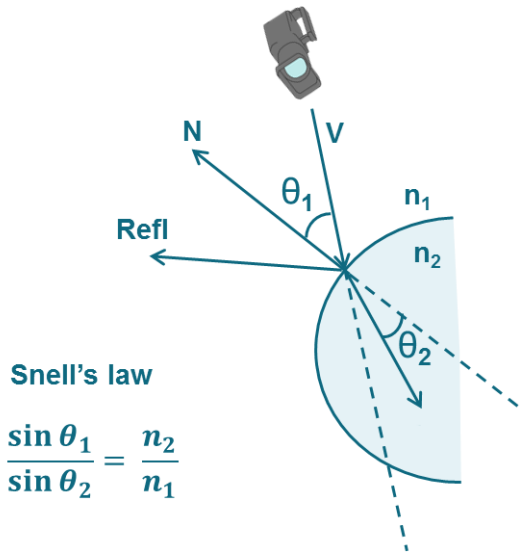

图 8-41 当光线穿过一个介质进入另一个介质时，就发生了折射

8.5.2 实现折射

自从开发人员着手渲染反射的那一刻起，他们就一直在尝试渲染折射，因为这两者在每一个半透明的表面上都是共同发生的。我们不缺少渲染反射的技术，但渲染折射的方法就很少了。

根据特定的折射类型，现如今能实现折射的方法在运行时也有所不同。实际运行时，大多数技术会将折射对象后面的场景渲染为纹理，然后在第二轮运行中使纹理扭曲，显示出折射的效果。按照不同的扭曲纹理，此方法可以用来渲染折射效果，例如水体、热浪、玻璃和其他效果。

这些技术中有一些确实能做到不错的视效，但是这样的扭曲纹理并非物理演算的结果，所以不见得每次都能获得理想的成果。比如说，如果从折射镜头的角度渲染一处纹理，可能在镜头中会有一些区域虽然无法直接看见，但按照物理原理，应该在发生折射后看见。

使用“纹理渲染法”的最大局限是画质。当镜头移动时，经常会出现明显的像素闪烁或像素不稳定。

8.5.3 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，开发人员自该技巧问世以来便将静态立方体贴图同时用于实现反射和折射。

不过，如果你在局部环境中使用静态立方体贴图实现反射或折射，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，所以必须仔细取舍。

8.5.4 准备立方体贴图

你必须准备立方体贴图，以便在折射实现中使用：

若要准备立方体贴图，请执行以下操作：

1. 将镜头置于折射几何体的中心。
2. 隐藏折射对象，并将六个方向上的周围静态环境渲染到立方体贴图。你可以将这一立方体贴图同时用于实现折射和反射。
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 *8.4 基于局部立方体贴图的动态软阴影* on page 8-143 中相同的方式，应用局部修正。
6. 生成一个新向量，该向量自立方体贴图生成的点到交叉点。使用这一最终向量从立方体贴图获取纹素，渲染折射对象背后的内容。

我们不使用折射向量 R_{rf} 从立方体贴图获取纹素，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;
```

```
float3R_rf = 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：

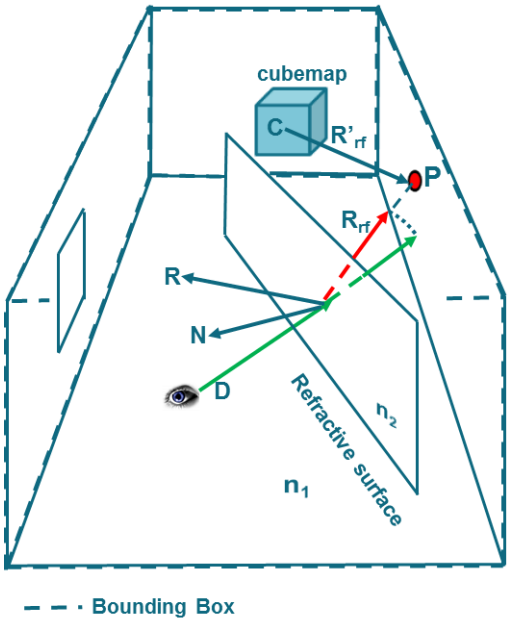

图 8-42 折射向量局部修正

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。

你还可以在着色器中使用一个内建函数，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` 是归一化视角或入射向量。
- `N` 是归一化法线向量。
- `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现流程：

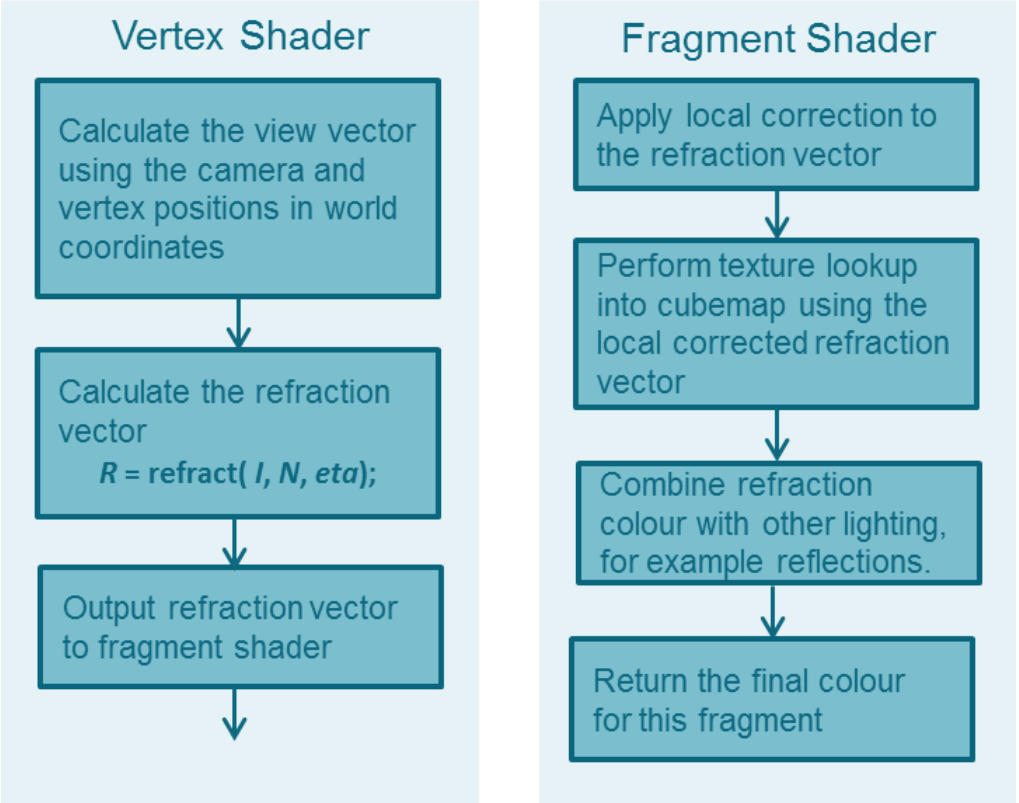

图 8-43 基于局部立方体贴图的折射着色器实现

8.5.5 着色器实现

在获取与局部修正折射方向对应的纹素时，你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，你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。你可以手动调整 `_ReflAmount` 获得自己想要的视觉效果。

你可以在下面这个博客中找到 `LocalCorrect` 函数在反射上的实现：*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：

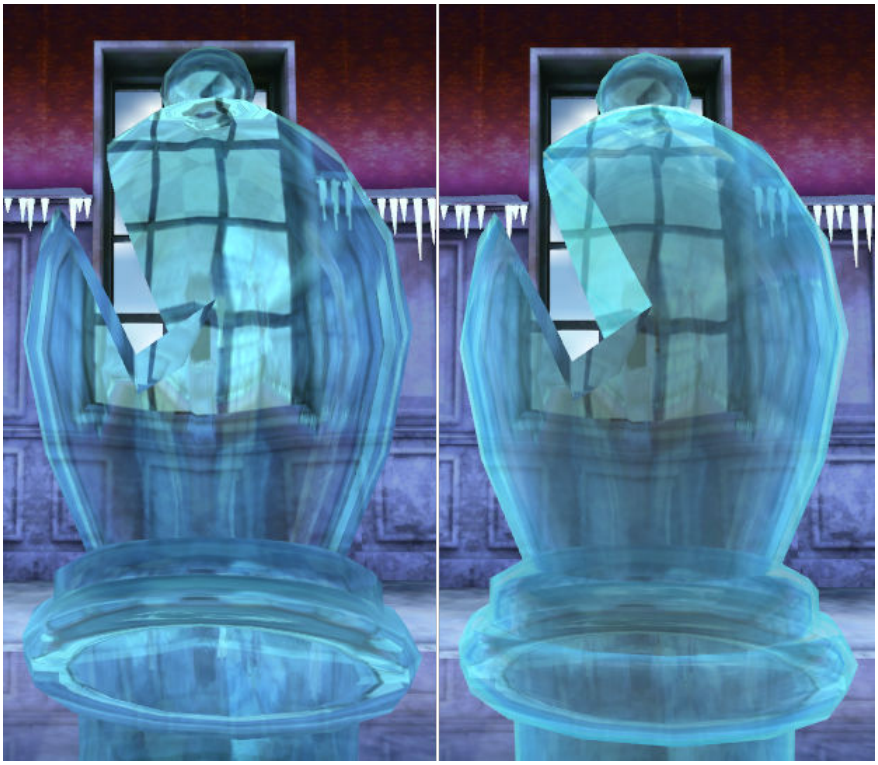

图 8-44 玻璃象棋子上基于立方体贴图的折射

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再加上第一通道的 Alpha 混合，仅渲染了正面。

- 在第一通道中，与你渲染不透明几何体时一样渲染半透明物体。打开正面剔除选项，然后渲染该对象，即仅渲染其背面。你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，根据对象本身反射、折射和漫反射颜色计算出颜色，然后将它们进行混合。
- 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反射颜色混合，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，你可以跳过此步骤。
- 在最终通道中，你要将生成的颜色与第一通道进行 Alpha 混合。

冰穴演示中对半透明凤凰实现了基于局部立方体贴图的折射，下图显示了其显示效果。

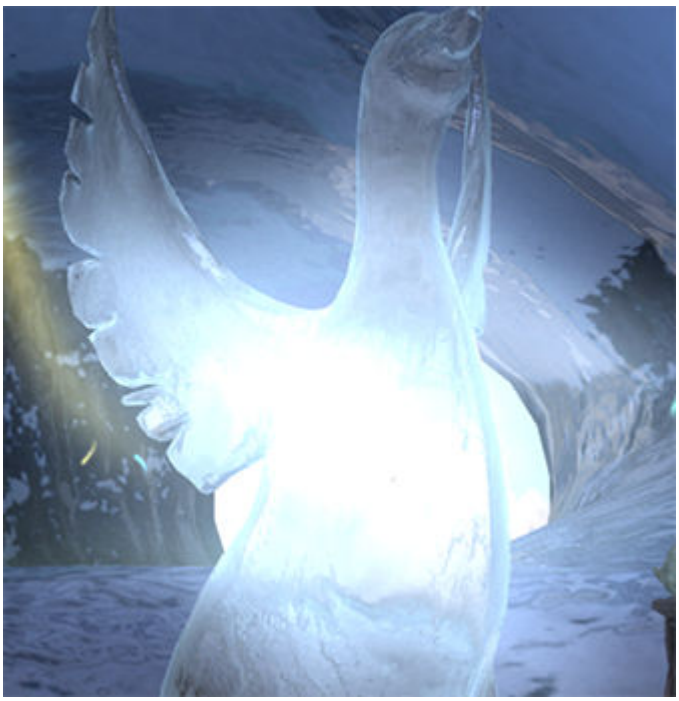

图 8-45 半透明凤凰折射

下图显示了半透明凤凰翅膀：

图 8-46 半透明凤凰翅膀

8.6 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 技巧实现的。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 技巧实现镜面反射效果：

```
// 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：

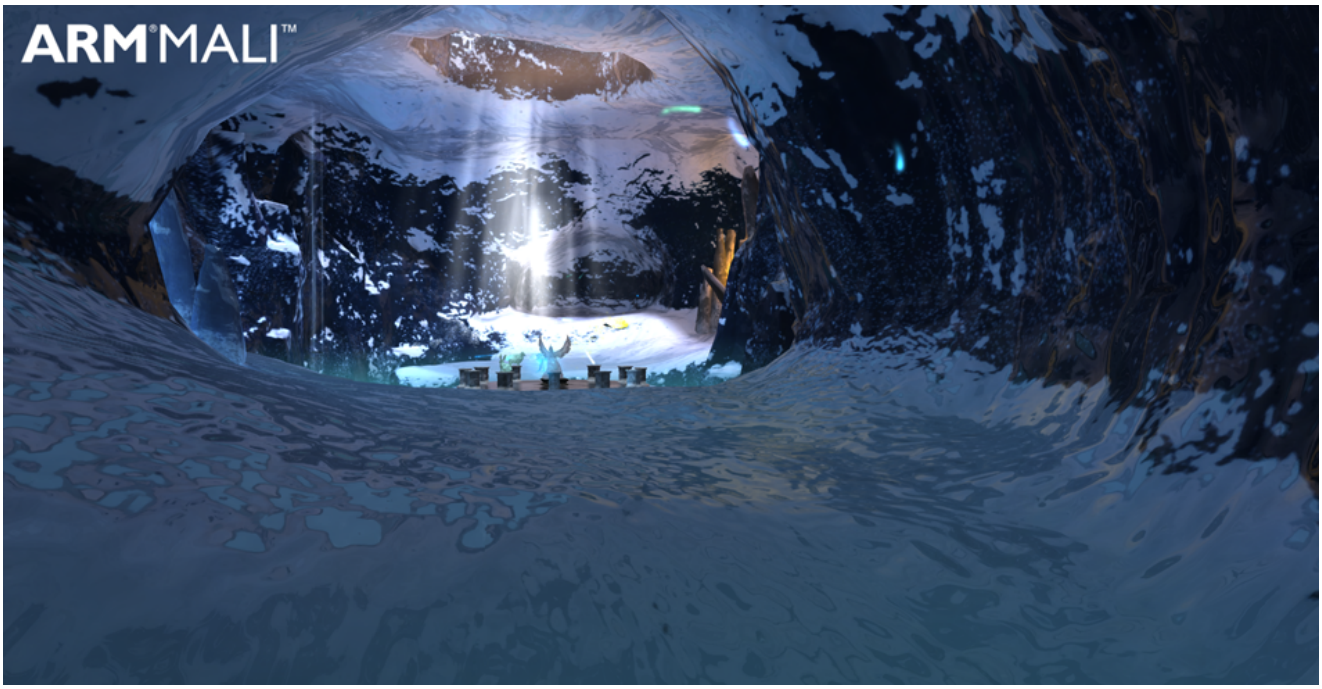

图 8-47 阴影区域

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：

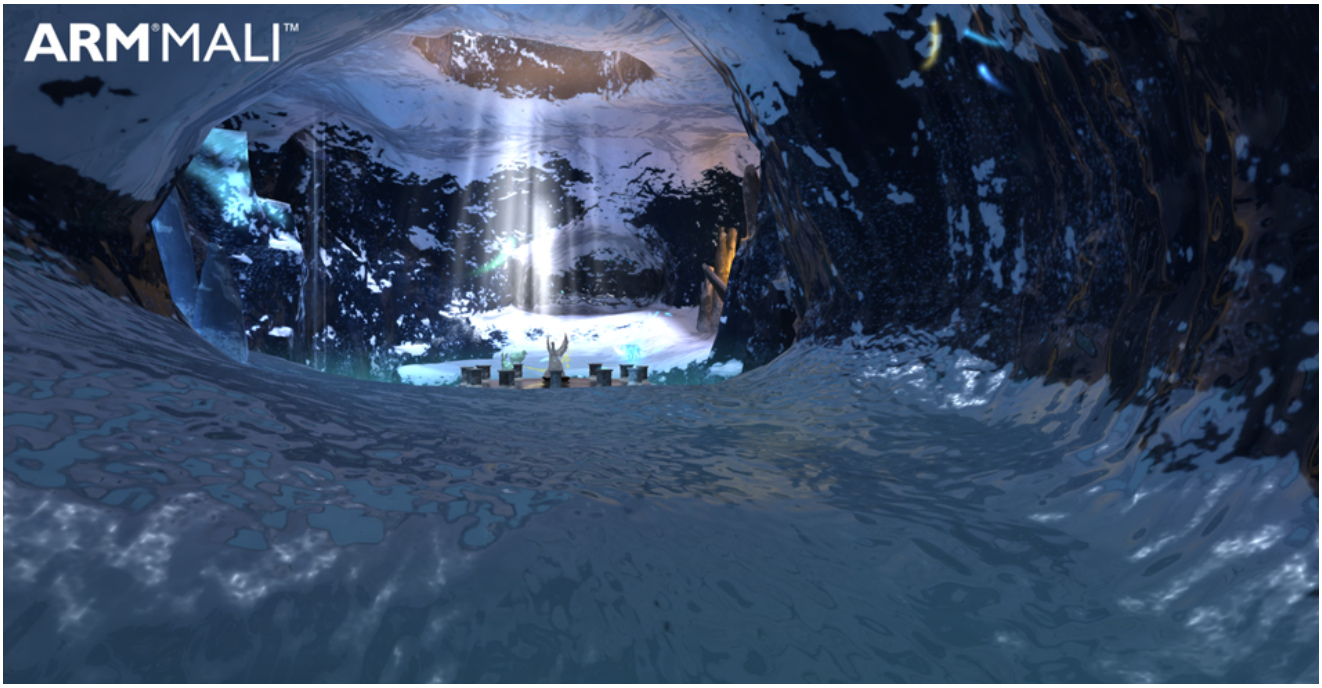

图 8-48 带有错误镜面反射效果的阴影区域

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：

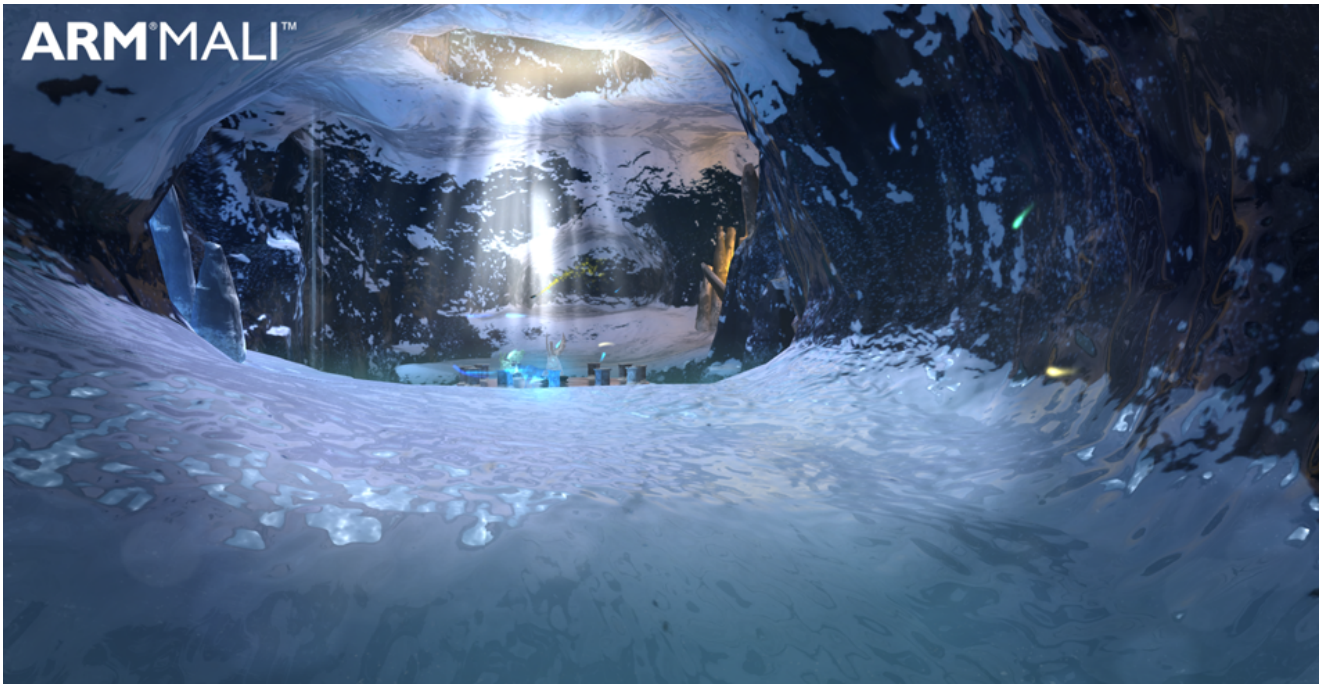

图 8-49 带有正确镜面反射效果的光亮区域

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。你可以使用 Alpha 通道来确定镜面反射强度，因为 Alpha 通道代表了洞穴中的透光孔。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，请参阅 *8.2 使用局部立方体贴图实现反射* on page 8-121。使用此向量从立方体贴图纹理获取 RGBA 纹素：

```
// 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，请参阅 *8.2 使用局部立方体贴图实现反射* on page 8-121。

`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，即使用 Blinn 技巧进行计算，Alpha 通道基于镜面反射颜色翻倍：

```
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。你可以将此添加到光照模型中。

8.7 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，但在一些情形中为了保持正确性，不执行该算法。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，你可以通过一些迹象来加以了解。

针对移动平台编译你的着色器，再查看其代码。确保你的着色器不会出现下列问题：

着色器有副作用

也就是说，着色器线程在执行期间会修改全局状态，因此二次执行该着色器可能会产生不同的结果。这通常是由于你的着色器会写入到共享的读取/写入内存缓冲区，如着色器存储缓冲区对象或图像。例如，如果你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。
- 写入到只写缓冲区。
- 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，它们通常会展示为平面，并由纹理指定树叶的某个区域是透明还是不透明。如果启用了 Early-Z，你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是来自顶点着色器。如果你的片段着色器写入到 `gl_FragDepth`， Mali GPU 无法执行 Early-Z 测试。

8.8 脏镜头效果

你可以使用脏镜头效果来制造戏剧氛围。它通常与镜头光晕效果一起使用。

你可以通过非常轻松和简单的方式来实现脏镜头效果，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现的，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给该着色器。

该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 混合方案渲染该四边形。

本部分包含以下子部分：

8.8.1 着色器实现

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 *8.8.3 脏镜头着色器代码* on page 8-163。

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物体之后渲染：

```
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纹素，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 8-50 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，图中所示为镜头朝向来自洞穴入口的阳光：

图 8-51 冰穴演示中所实现的脏镜头效果

8.8.2 脚本实现

一个简单脚本实现全屏四边形，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。根据你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。这基于镜头至太阳向量和镜头前向向量的相对朝向。当镜头正对太阳时，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);
```

8.8.3 脏镜头着色器代码

以下是 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);
```

8.9 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。

本部分包含以下子部分:

8.9.1 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并散开的太阳光线。

下图显示了冰穴演示中的光柱:

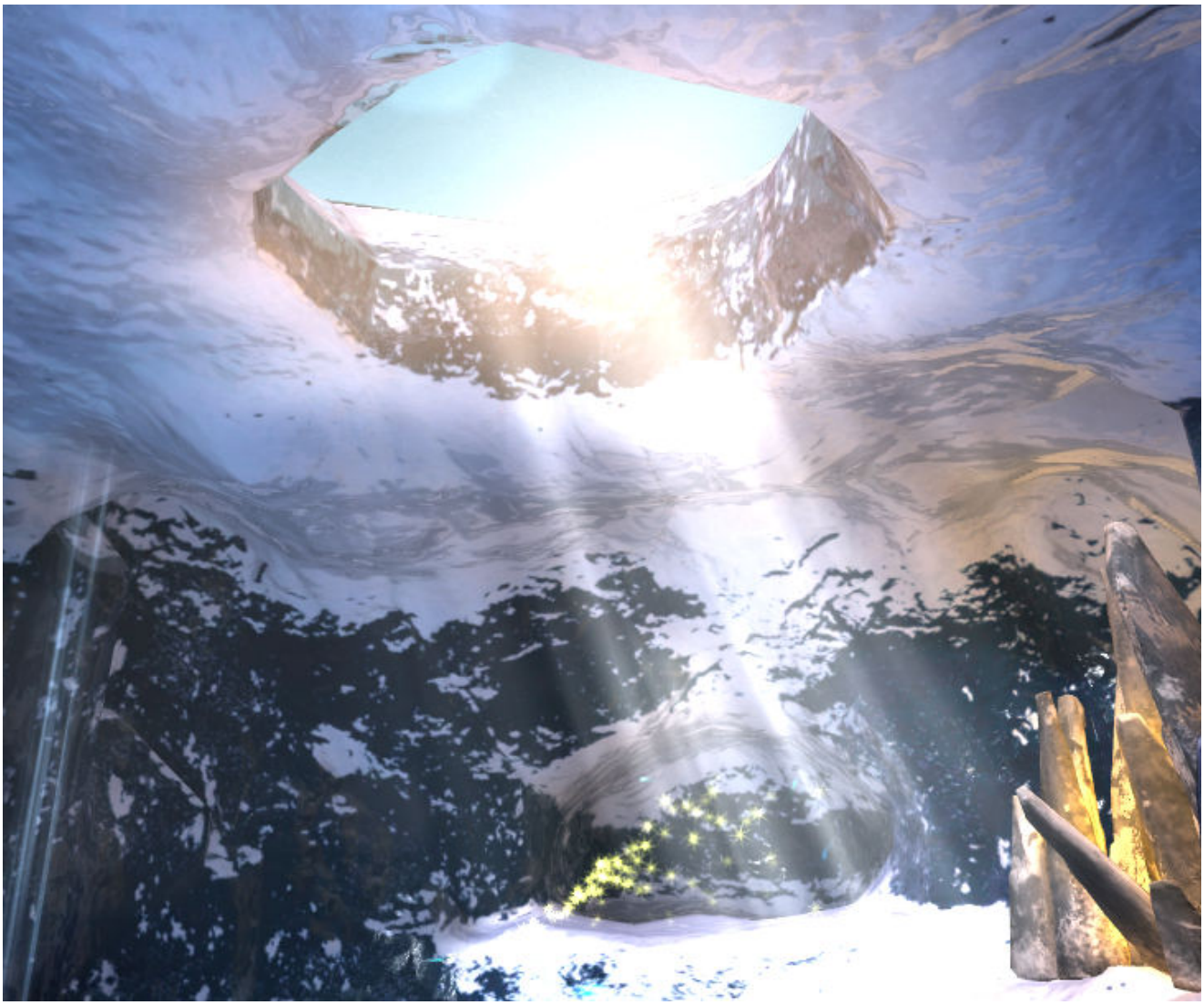

图 8-52 冰穴演示中的光柱

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形状，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，根据你定义的角度 θ 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。

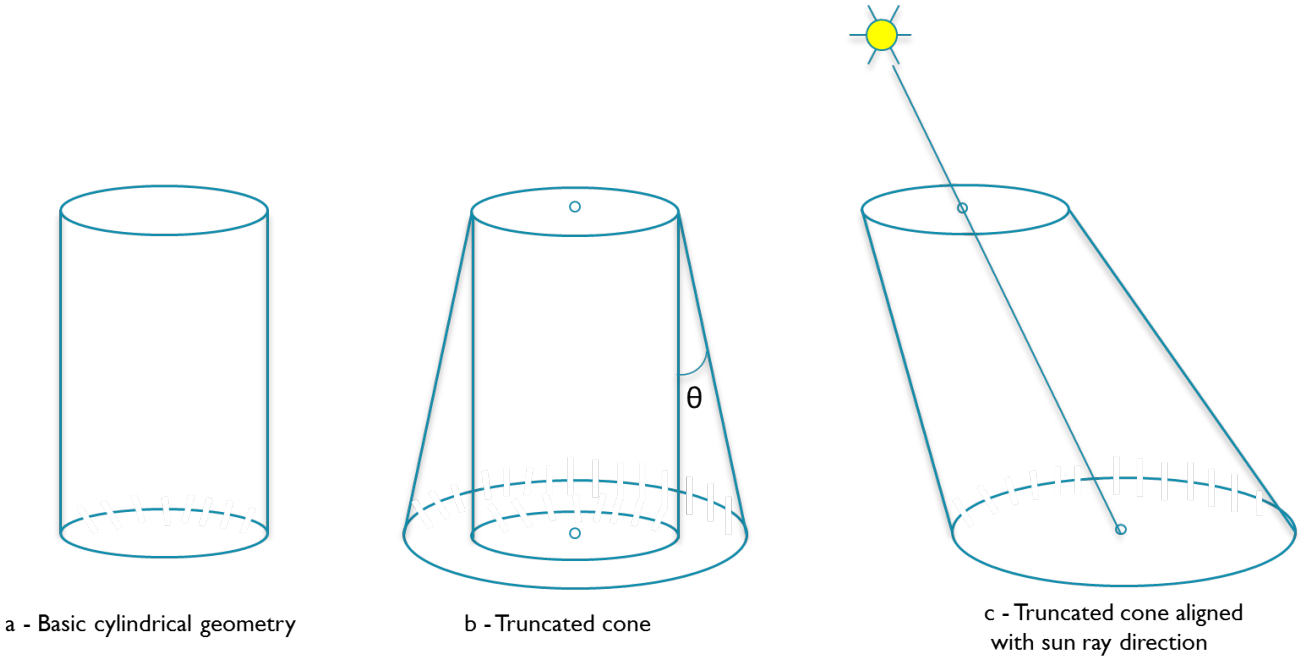

图 8-53 光柱几何体

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 θ。
- 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，请避免渲染会暴露出其几何体的任何硬边缘。要达到这一目的，你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 8-54 光柱纹理。左侧为遮罩纹理。右侧为光束纹理

8.9.2 淡化圆锥体边缘

在与截面平行的平面中，根据镜头与顶点相对朝向，淡化光柱强度。

在顶点着色器中，将镜头位置投影到截面上。

构建一个从截面中心到投影的新向量，然后将它归一化。

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;
```

你可以通过系数 `_FadingEdgePower` 细调光柱边缘的淡化。

脚本传递系数 `_CurrLightShaftIntensity`。这使得光柱在镜头靠近它时淡出。

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
```

8.10 雾化效果

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

本部分包含以下子部分:

8.10.1 关于雾化效果

在现实世界中，你越往远看，被淡化的颜色就越多。不一定需要雾天才能看到这种效果，阳光灿烂时你也可以看到，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。

本节介绍两种版本的雾化效果:

· 过程线性雾。
· 基于粒子的雾。

你可以同时应用这两种效果。冰穴演示中同时使用了这两种方法。

8.10.2 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，你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镜头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` 是顶点至镜头距离。
· `_FogDistanceScale` 是作为统一变量传递至着色器的因子。
· `_FogColor` 是你定义的基准雾颜色，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);
```

下图显示了你在场景中获得的结果：

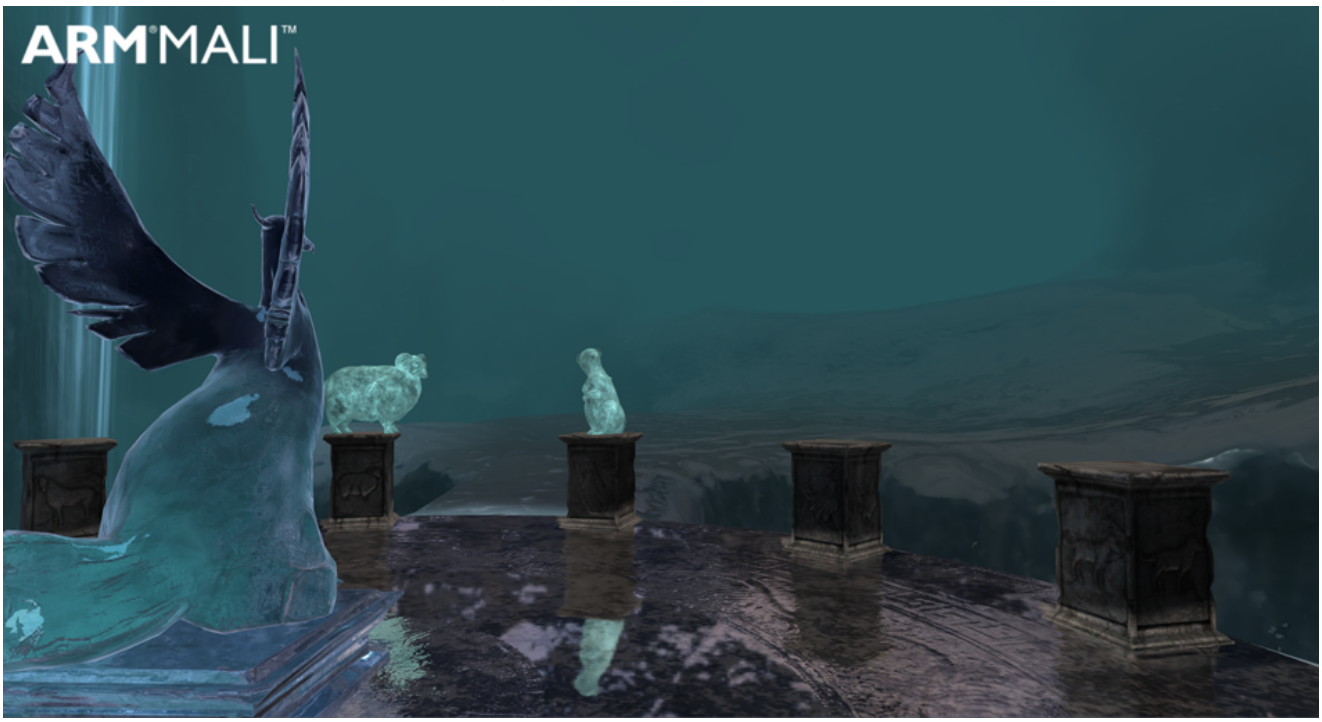

图 8-55 基于距离的线性雾

说明

在着色器中计算雾意味着你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。你可能必须将着色器拆分为两个或更多通道。

你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。

8.10.3 具有高度的线性雾

雾在场景中均匀地应用。你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。你可以显示高度值，以进行手动调整。

下图显示了基于距离和高度的线性雾：

图 8-56 基于距离和高度的线性雾

8.10.4 非均匀雾

雾不一定是均匀的。你可以引入一些噪点，让雾的视觉效果更加引人。

你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镜头较近的纹理。

你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。

8.10.5 预烘焙雾

如果知道镜头不会靠近它们，你可以将雾预烘焙到纹理中。这可以降低所需的计算功率。

8.10.6 使用粒子的体积雾

你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：

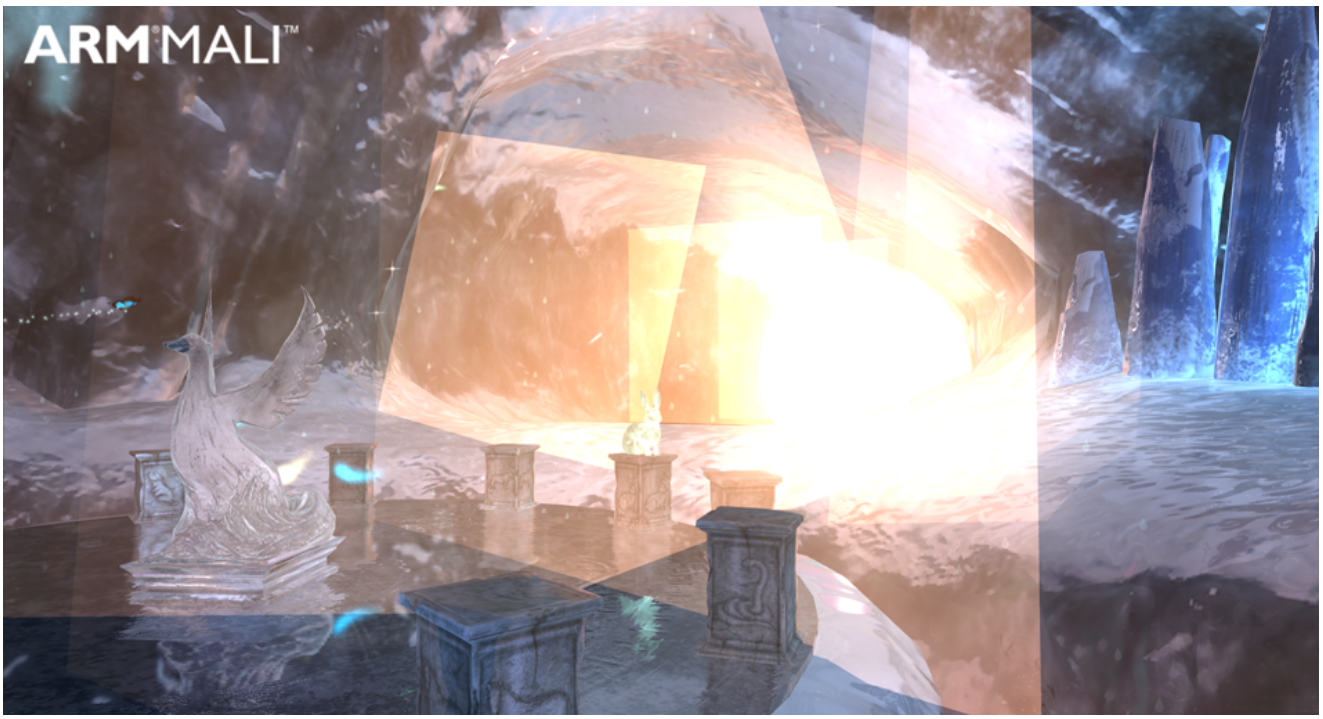

图 8-57 没有纹理的粒子

下图显示了带有纹理的粒子几何体：

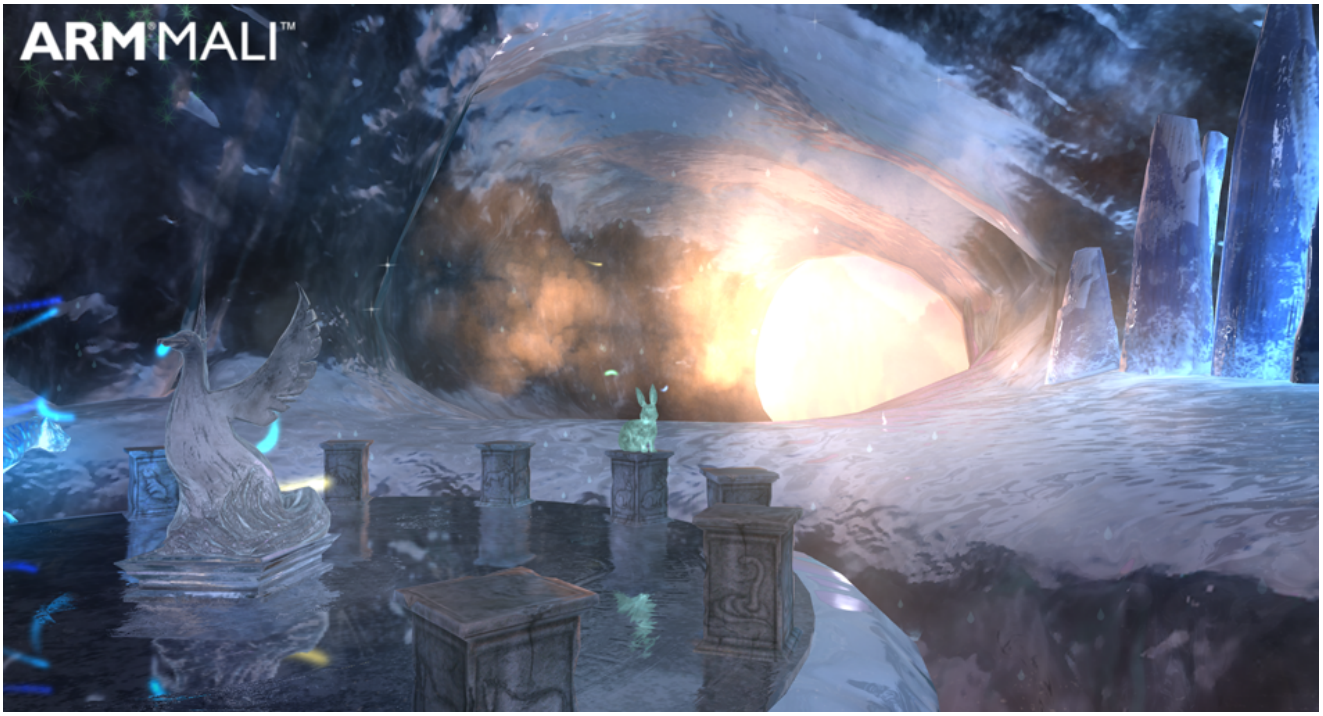

图 8-58 带有纹理的粒子

下图显示了应用至各个粒子的纹理：

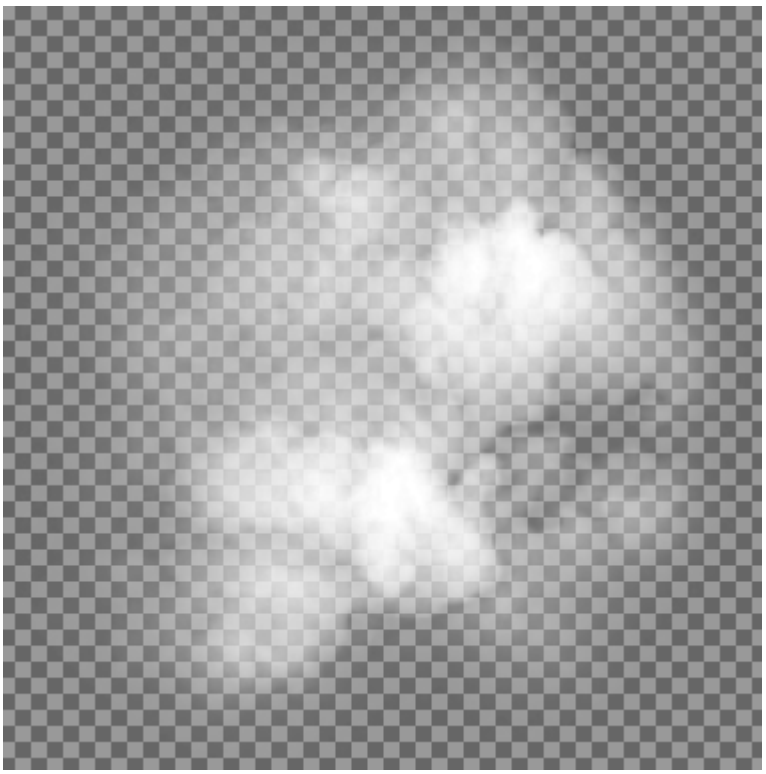

图 8-59 个别粒子的纹理

角度淡化效果

按照粒子相对于镜头位置的朝向，淡入和淡出各个粒子。如果你不淡入和淡出粒子，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图元。

   ```
   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 个其他透明物体。
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：

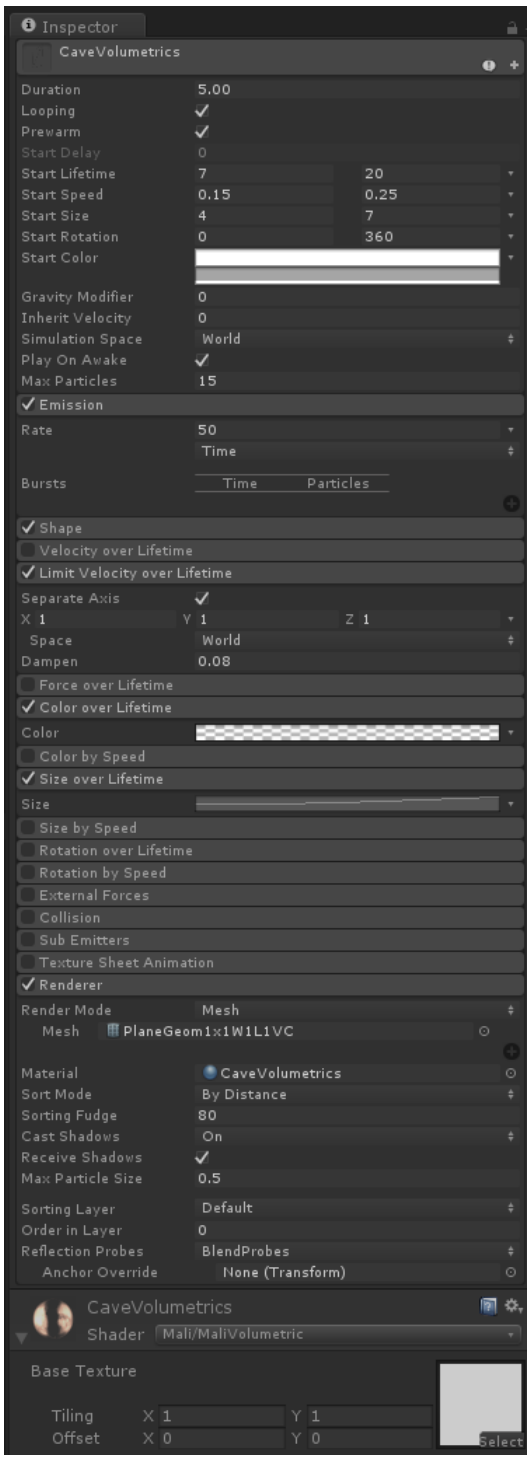

图 8-60 来自冰穴演示的粒子系统参数

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 8-61 生成粒子的粒子方框定义

8.11 高光溢出

高光溢出用于再现真实镜头在明亮环境中拍摄图片时出现的效果。高光溢出效果模拟从明亮区域周边散发出的光线，营造出一种镜头前的强光照的效果。

下图展示了在冰穴动画示例中洞穴入口处的高光溢出：

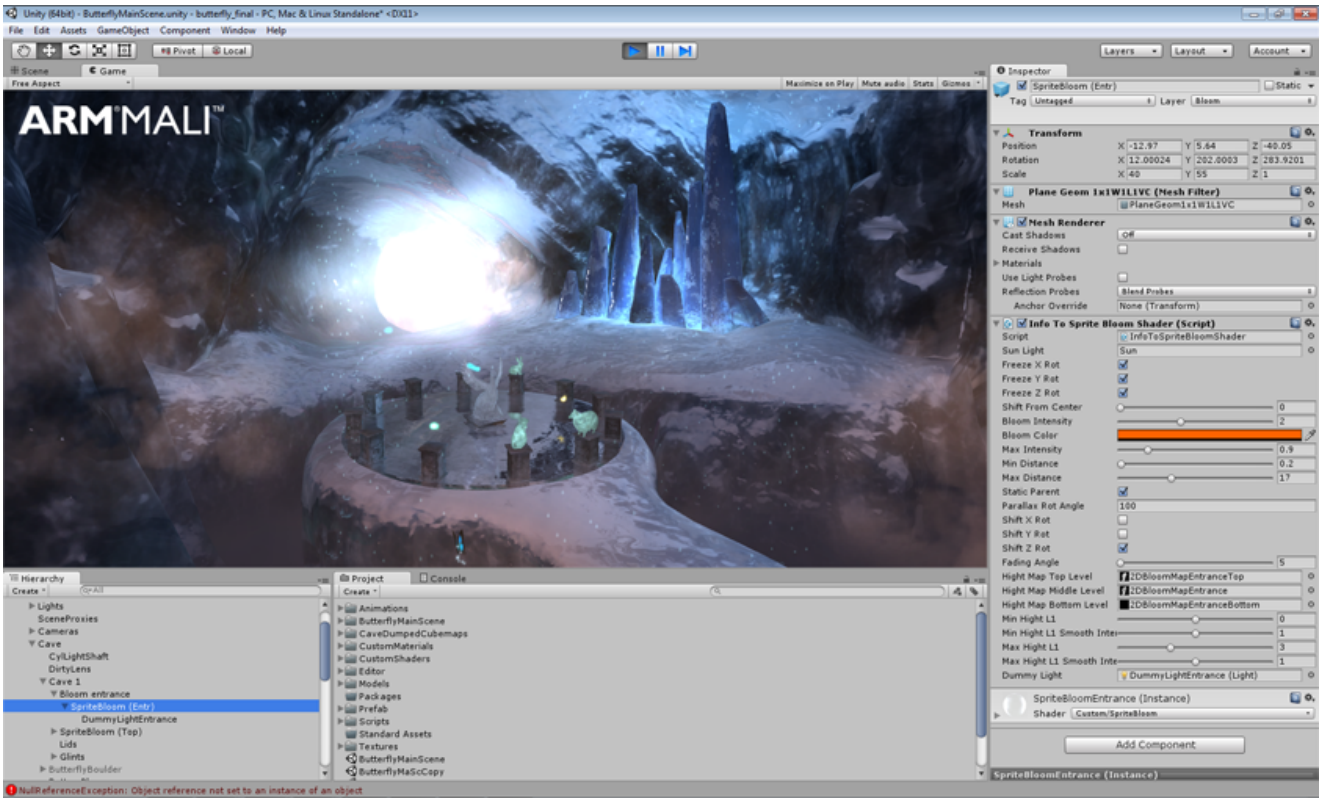

图 8-62 冰穴演示中所实现的洞穴入口处的高光溢出

高光溢出效果常见于真实镜头场景，因为它们无法完全对焦。当光线穿过镜头的光圈时，一些光线发生衍射，在图像周围形成一个明亮的光环。此效果在大部分情况下通常不明显，但在强光照射下会出现。

本部分包含以下子部分：

8.11.1 实现高光溢出效果

通常，高光溢出效果被实现为一种后处理效果。以这种方式生成特效可能会消耗大量的算力，因此这并不适用于手机游戏。如果你的游戏像冰穴演示动画一样使用了许多复杂的特效，则更是如此。

有一种替代办法是使用一个简单的平面。在镜头与光源之间放置一个平面。该平面必须以法线为基准，该法线指向镜头移动时所围绕的场景部分。

如果以这种方式制造高光溢出效果，平面应放置在预期产生高光溢出效果的位置。如果要调节效果强度，可以基于与平面法线和光源对齐的视线矢量来创建一个因子。冰穴演示动画中便应用了这一方法。

下图展示了使用一个平面所实现的高光溢出效果：

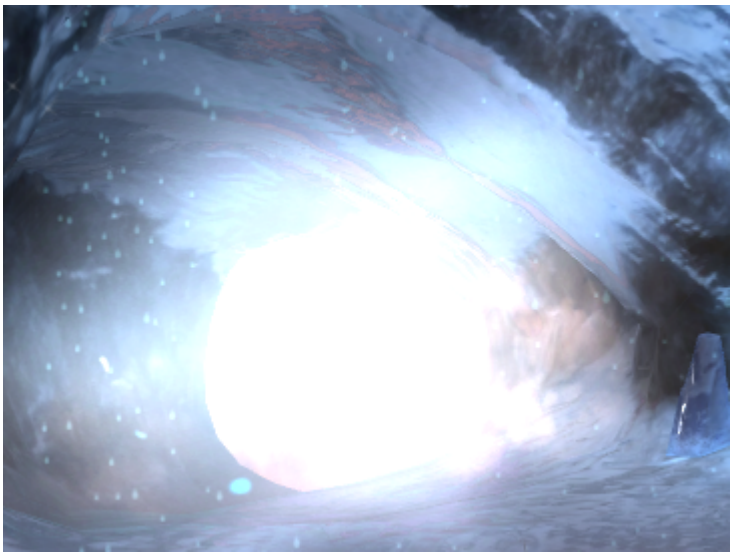

图 8-63 平面位于添加高光溢出效果的洞口

8.11.2 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。此因子基于镜头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。

8.11.3 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;
}
```

8.11.4 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 * _AligmentFactor;
}
```

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中使用了高光溢出效果的纹理，及其渲染结果：

图 8-64 显示镜头进入不透明柱子后遮挡区域的序列

8.11.5 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镜头之间存在不透明物体时，它就无法产生预期的结果。你可以使用遮挡贴图进行修正。

不透明物体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 8-65 显示在不透明的基座上的错误高光溢出效果

为防止出现这些错误，你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。这一遮挡贴图即为镜头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。下列代码实现了这一调整：

```
IntensityFactor = alignmentFactor * occlusionMapFactor
```

由于冰穴演示镜头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`：

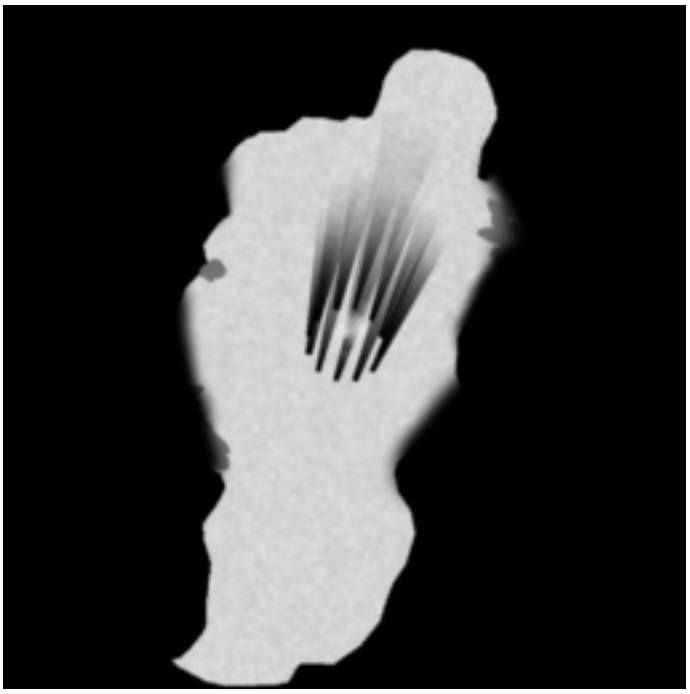

图 8-66 地平面高光溢出遮挡贴图

说明

贴图中间的黑色辐射区域是不透明柱子背后的区域，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`：

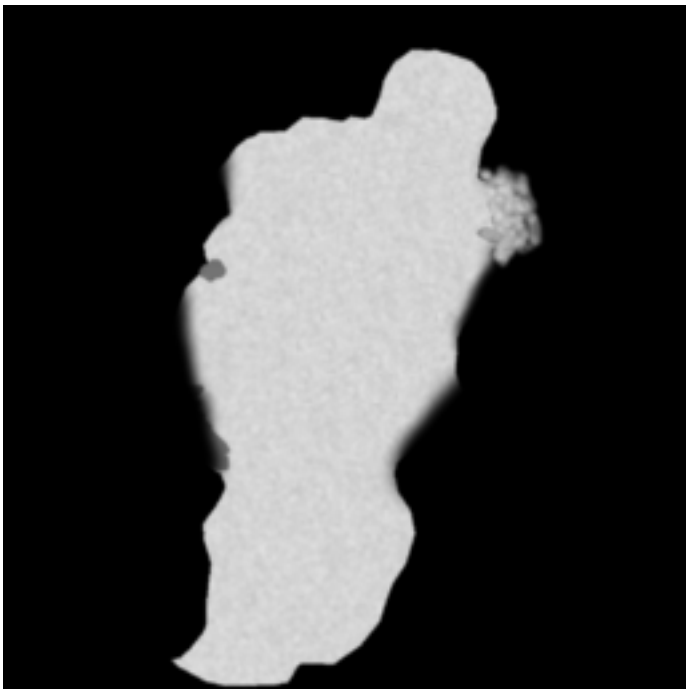

图 8-67 地平面以上的高光溢出遮挡贴图

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 8-68 地平面以下的高光溢出遮挡贴图

8.11.6 遮挡贴图间插值

在冰穴演示中，脚本计算每一帧中镜头位置在 2D 洞穴贴图上的 XZ 投影，并将结果归一化到零和一之间。

如果镜头高度低于 H_{min} 或高于 H_{max}，则使用归一化后的坐标从单一贴图获取颜色值。如果镜头高度在 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物体上混合高光溢出。下图显示了产生的效果。当镜头进入和离开时，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。

下图显示镜头进入不透明柱子后遮挡区域的序列：

图 8-69 进入不透明柱子后面的遮挡区域

8.12 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。

本部分包含以下子部分：

8.12.1 关于冰墙效果

冰是一种难以复现的材质，因为光线从不同方向发散开来，取决于冰表面细节。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，并引入了一个视差效果来增加真实感。

下图显示了引入此效果的冰穴动画演示：

图 8-70 冰穴演示动画

下图显示了此效果的特写：

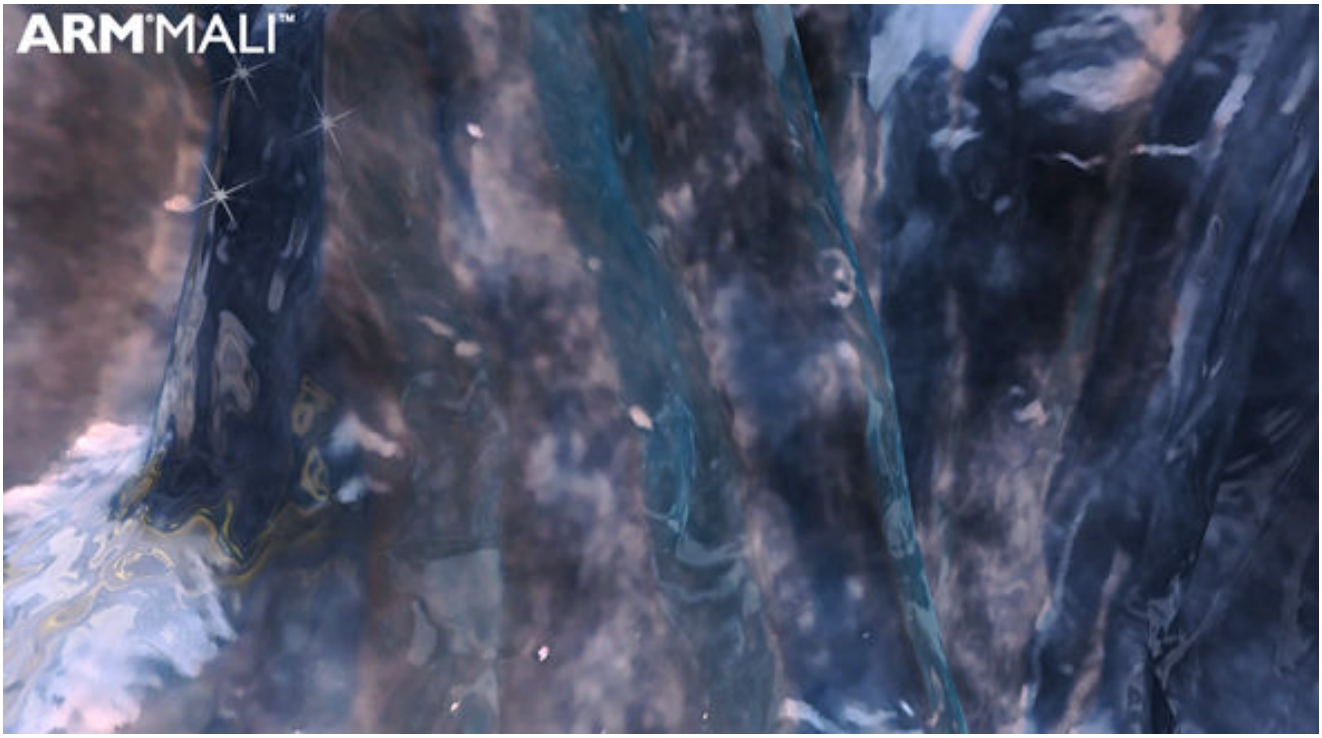

图 8-71 有反射效果的冰墙近景

8.12.2 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一种灰度正切法线贴图。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：

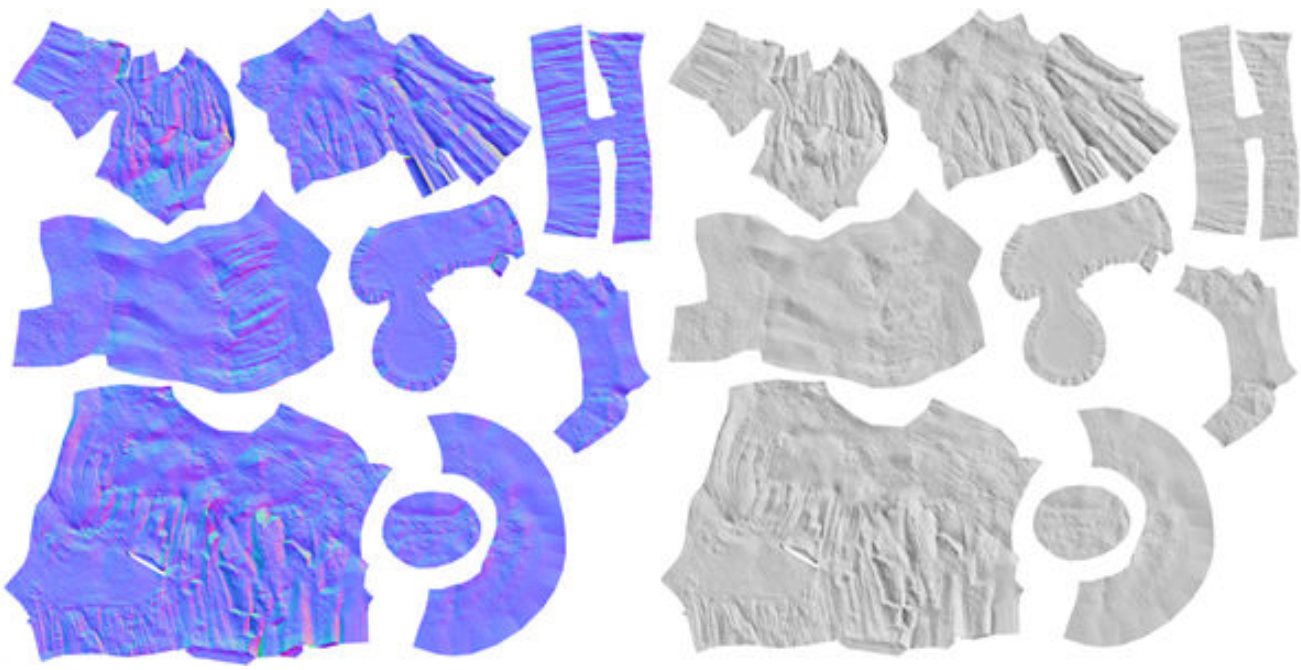

图 8-72 切线空间法线贴图和灰度虚构法线贴图

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，因为它们生成的灰度值范围跨度很小。这意味着，可以在渲染后期很好的使用正切空间法线贴图。

另一个做法是使用物体空间法线贴图。这些贴图具有与正切空间法线贴图相同的细节，同时还显示了光照位置。因此，生成的物体空间法线贴图灰度中的值范围太大，无法在渲染后期发挥很好的作用。下图显示了这种效果：

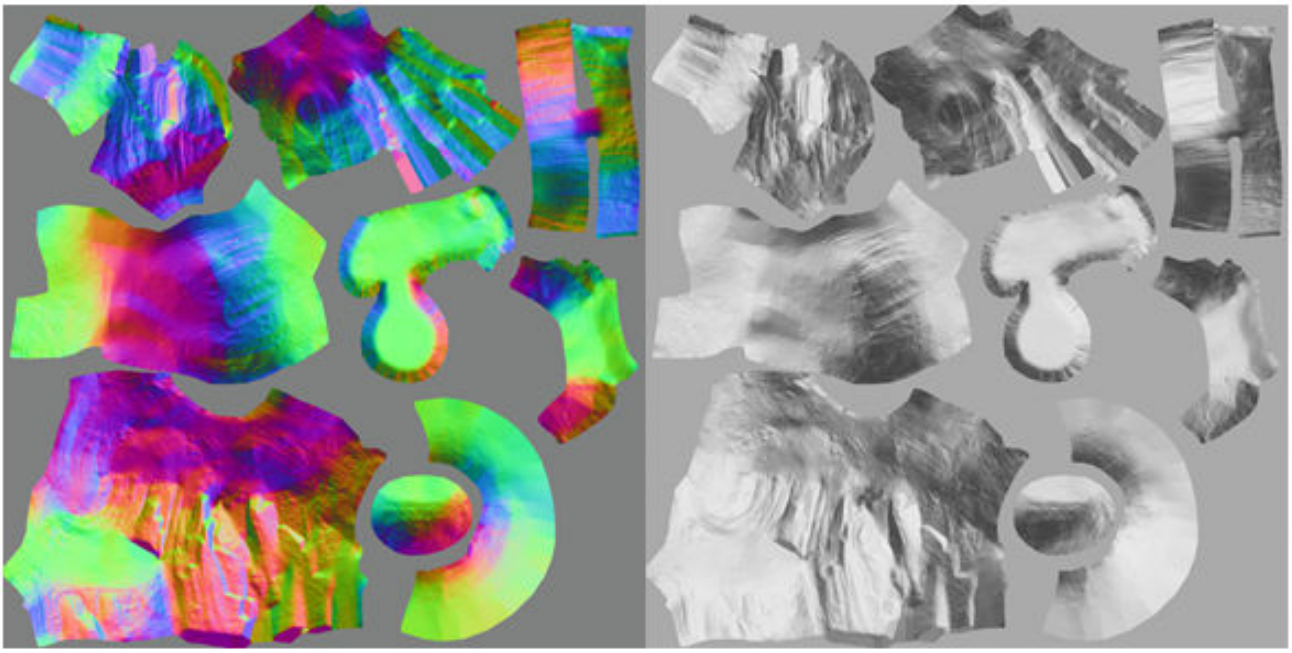

图 8-73 物体空间法线贴图和灰度效果

在部分法线贴图中使用透明度

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反射纹理贴图来修改灰度虚构法线贴图。此修改可增加纹理贴图中出现雪的地方的灰度虚构法线贴图的 alpha 分量值。

下图显示了用于冰穴演示静态表面的漫反射纹理贴图，以及它们应用到灰度虚构法线贴图时的结果：

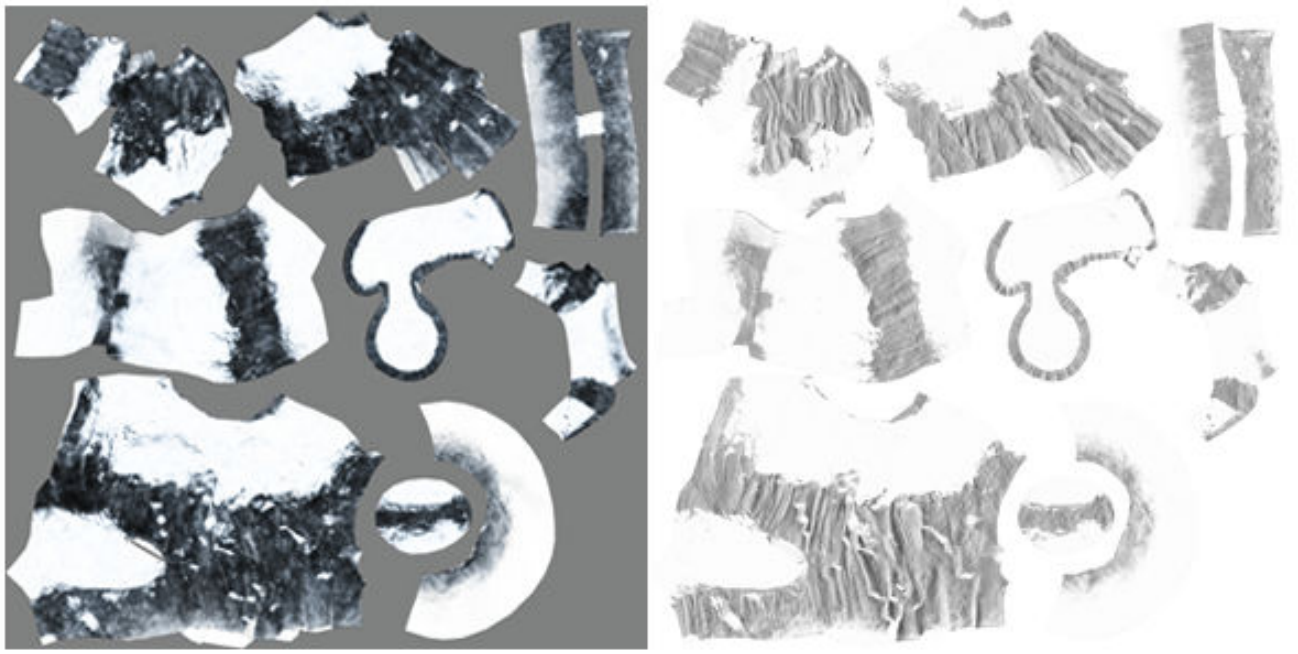

图 8-74 透明区域的漫反射纹理贴图和最终的虚构法线贴图

8.12.3 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。该组合由如下函数实现：

```
half4 bumpNormalFake = lerp(bumpNorm, bumpFake, amountFakeNormalMap);
```

此代码意味着，在洞穴的昏暗部分中，反射主要来自于灰度虚构法线。在洞穴的有雪区域，该效果来自于物体空间法线。

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。处理前，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。

着色器对法线分量进行变换。它使用的变换就是通常用于将 0 到 1 范围内的值变换到 -1 到 1 范围的变换。公式为，结果 = 2 * 值 - 1。此公式更改了法线向量，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，向量传入给 `reflect()` 函数。此函数设计为使用归一化法线向量工作，但在这个例子中，非归一化法线向量传入该函数。下列代码演示了着色器内建函数 `reflect()` 的工作方式：

```
R = reflect(I, N) = I – 2 * dot(I, N) * N
```

根据反射定律，当此函数与长度小于一的非归一化输入法线向量搭配使用时，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，在立方体贴图岩石区域倒影旁边的白色区域，营造出一些不均匀的白点效果。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，创造了一种独具吸引的漩涡效果。下图显示了这种效果：

图 8-75 冰面反射上的漩涡效果

如果着色器程序设计为在没有非归一化 clamp 阶段里使用虚构法线的话，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 clamp 阶段时生成的结果：

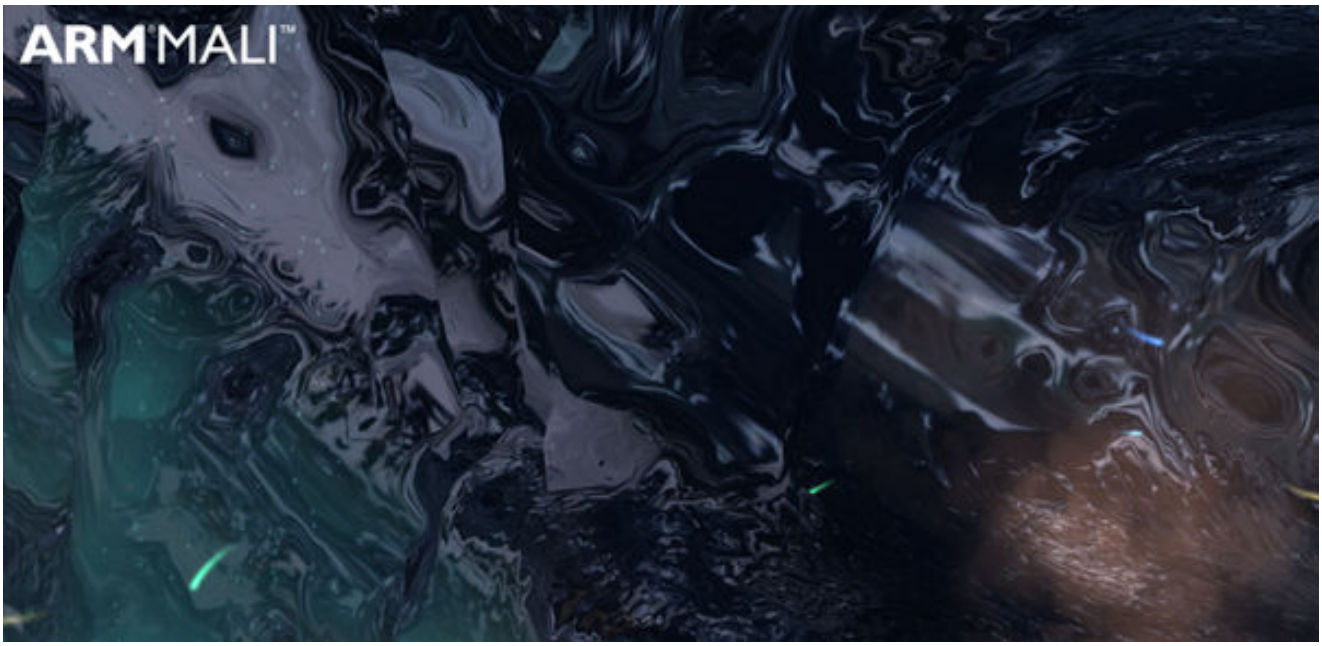

图 8-76 没有设置 clamp 阶段的冰面反射

8.12.4 在反射中使用局部修正

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镜头位置的变化而改变。对于镜头侧向移动而言，则更是如此。

相关信息

8.13 过程天空盒

冰穴演示中，以天作为时间单位，来展示可由局部立方体贴图实现的动态阴影效果。

本部分包含以下子部分：

8.13.1 关于过程天空盒

要获得动态时间效果，需要组合下列元素：

- 一个程序生成的太阳。
- 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。

过程太阳和独立的云朵纹理也可创建低计算成本的子表面散射效果。

下图显示了天空盒的一个视图：

图 8-77 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。这可以避免用一个半球去渲染天空盒子。相反，它只在洞穴的孔洞附近使用平面。

相比用半球去渲染，随后会被大量几何体遮挡的做法而言，性能会大大提升。

下图显示了利用平面渲染的天空盒。

图 8-78 利用平面渲染的天空盒

8.13.2 管理昼夜

此效果使用 C# 脚本管理一天时间变换的数学计算，以及昼夜更替动画。一个着色随后将太阳和天空贴图组合在一起。

你必须为该脚本指定下列值：

- 淡化天空盒背景阶段的数量。
- 昼夜更替的最长时长。

对于每一帧，该脚本选择天空盒立方体贴图进行混合。选定的天空盒被设置为着色器的纹理，然后渲染器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，你可以在一处设置纹理，而后供应用程序中的所有着色器使用。下列代码对此进行了演示：

```
Shader.SetGlobalTexture (ShaderCubemap1, _phasesCubemaps [idx1]);
Shader.SetGlobalTexture (ShaderCubemap2, _phasesCubemaps [idx2]);
Shader.SetGlobalFloat (ShaderAlpha, blendAlpha);
Shader.SetGlobalVector (ShaderSunPosition, normalizedSunPosition);
Shader.SetGlobalVector (ShaderSunParameters, _sunParameters);
Shader.SetGlobalVector (ShaderSunColor, _sunColor);
Shader.SetGlobalTexture (ShaderCloudsCubemap, _CloudsCubemap);
```

说明

使用的采样器变量名不得与着色器本地变量名冲突。

脚本代码中进行了如下设置：

- 两个立方体贴图经过插值。值 `idx1` 和 `idx2` 由经过的时间计算得到。
- 一个 `blendAlpha` 因子，在着色器中用于混合两个立方体贴图。
- 一个归一化的太阳位置坐标，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 中分别包含各自的采样器变量名，本例中，各自的采样器变量名为 _SkyboxCubemap1 和 _SkyboxCubemap2。

为了在着色器中访问这些纹理，你必须使用下列代码声明它们：

```
samplerCUBE _SkyboxCubemap1;
samplerCUBE _SkyboxCubemap2;
```

该脚本根据你指定的列表选择太阳颜色和环境颜色。它们针对每一个阶段进行插值。

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。在冰穴演示中，此效果使场景根据一天中所处的时间阶段出现总体颜色的渐进变化。

8.13.3 渲染太阳

要渲染太阳，你必须检查片段着色器中天空盒的每一个像素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在世界坐标中的归一化太阳位置向量与归一化视角方向矢量的点积。

下图显示了太阳的渲染。归一化视角矢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)。

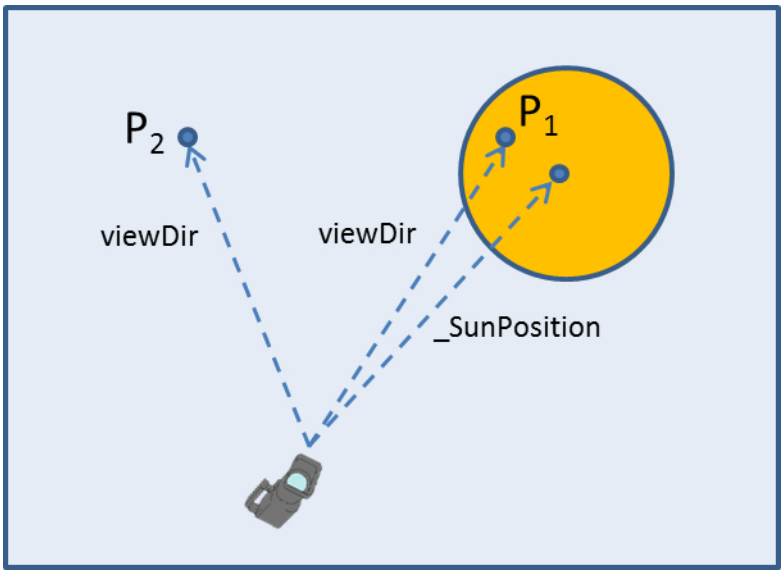

图 8-79 渲染太阳

通过 C# 脚本，将归一化太阳位置矢量传递到着色器。

- 如果结果值大于特定阈值，像素设置为太阳的颜色。
- 如果结果值小于该阈值，像素设置为天空的颜色。

点积也可创造出沿太阳边缘淡化的结果：

```
half _sunContribution = dot(viewDir,_SunPosition);
```

下图显示了晴空中的太阳视图：

图 8-80 晴朗天空条件下的过程太阳渲染

8.13.4 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`（如果纹素代表天空）和值 `1`（如果纹素代表大山）。

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。此采样过程不会产生大量计算，因为纹理已被采样用于渲染群山。

你也可以渐进淡化边缘附近或山上有雪覆盖区域的 alpha。这可以产生阳光在雪面反射的效果，而且几乎不需要计算工作。

类似的方法也可用于为云朵创建计算成本低廉的子表面散射效果。

原始的阶段立方体贴图分为两个独立小组。

- 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 1，而天空的值为 0：

图 8-81 群山纹理

在云朵天空盒中，空白区域的 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，确保天空中的云朵显得自然不生硬。下图显示了具有 alpha 的云朵纹理：

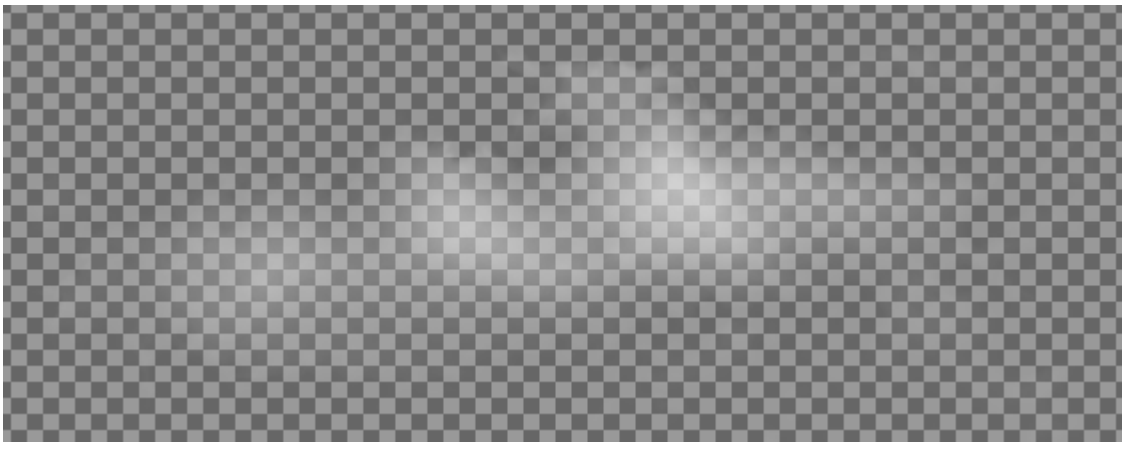

图 8-82 云朵纹理

着色器执行如下工作：

1. 对代表一天中当前时间阶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 将第 2 点中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. 调用一个函数来计算太阳颜色对当前像素的作用。
7. 将第 6 点的结果和前面混合的天空盒与云朵颜色加到一起。

——————— **说明** ———————

你可以将天空、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。

———————————————

8.13.5 子表面散射

你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你眼睛的光线。你看到的是直接从太阳射出的光线，仅受到空气的些许漫反射。

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你的眼睛。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 加到一起后得到的值）。

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` 计算中使用 `smoothstep()` 函数来提供从太阳中心到天空边缘附近的更加平缓的渐变。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，即 `cos(5 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 8-83 太阳未被云朵遮挡

下图显示了被云朵遮挡的太阳：

图 8-84 太阳被云朵遮挡

8.14 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。

本部分包含以下子部分：

8.14.1 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件对象。
· 用于限制萤火虫飞行区域的盒体碰撞器。

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。

下图显示了一个萤火虫：

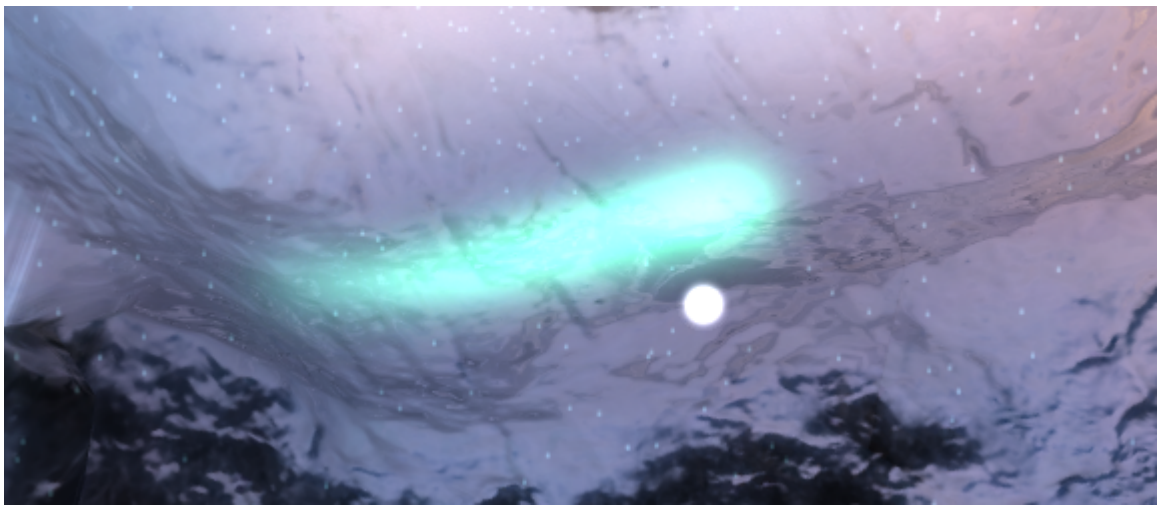

图 8-85 萤火虫

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：

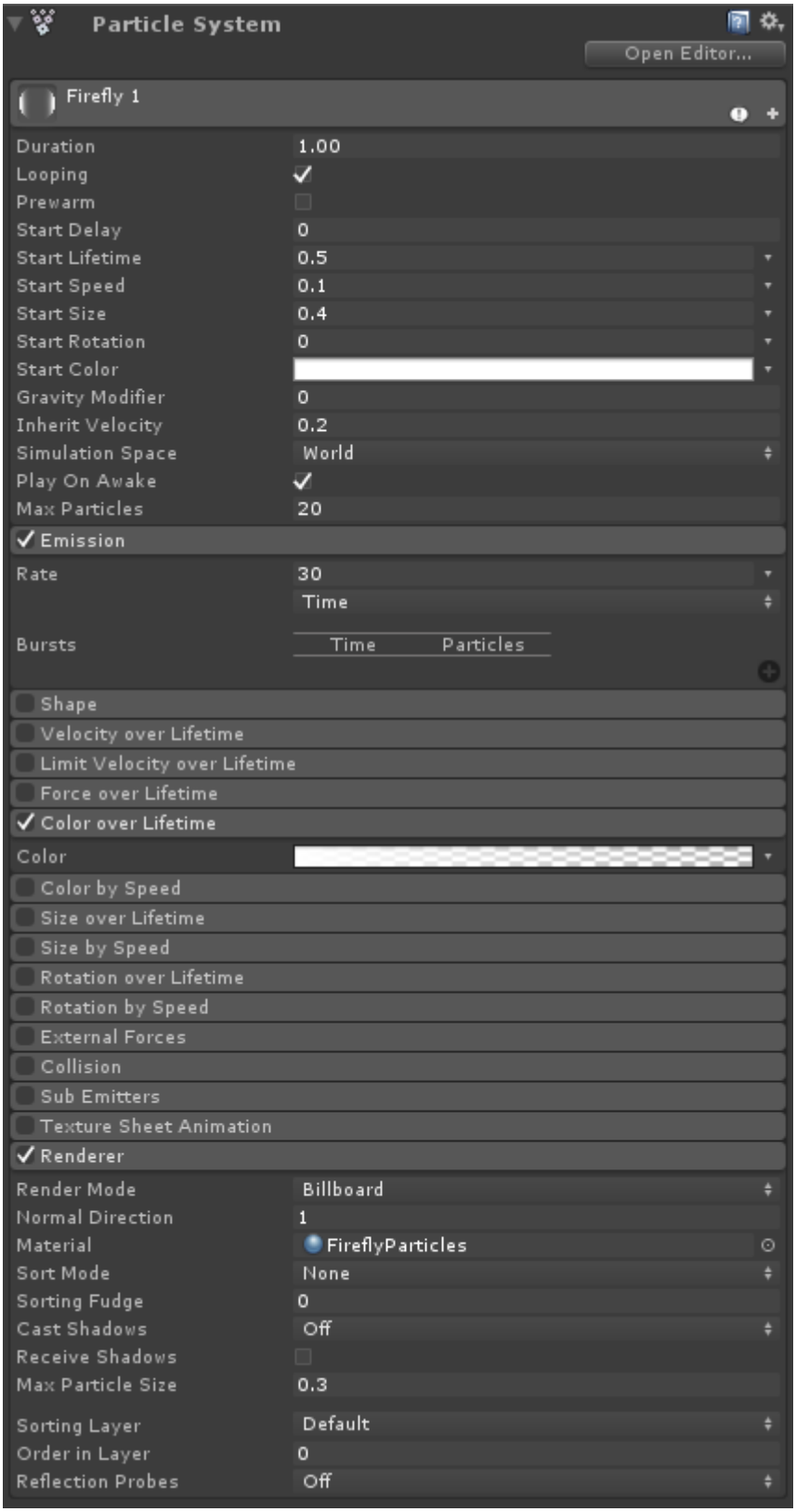

图 8-86 Unity 粒子系统设置

根据你想要获得的效果，你可以使用 Unity Trail Renderer 来提供更为连贯的显示效果。

Trail Renderer 为每一道轨迹生成大量的三角形。你可以通过修改 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镜头；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：

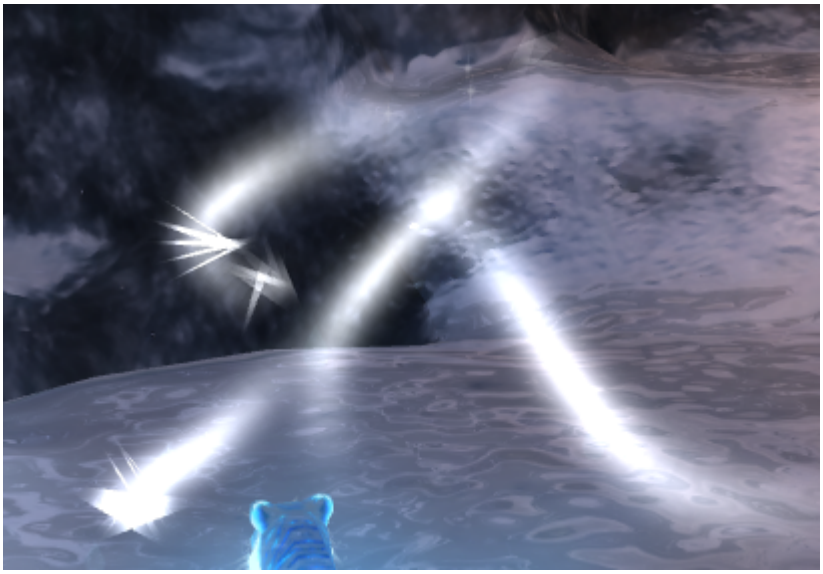

图 8-87 重叠轨迹失真

添加到预制件中的最后一个组件是点光源，它一边移动一边在场景中投射光线。

8.14.2 萤火虫生成器预制件

萤火虫生成器预制件可以管理萤火虫的创建，并在每一帧上更新它们。它包含用于更新的 C# 脚本，以及用于限制每个萤火虫运动空间的盒体碰撞器。

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。作为参数，由于萤火虫在碰撞盒内随机运动，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、连贯的函数，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。

这种插值需要在起点和终点各有一个控制点，并且每个控制点对应有两条切线。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控制点切线，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，你必须设置萤火虫的颜色，以及当前路径间隔的持续时间。

在每一帧上，该脚本使用下列代码计算埃尔米特插值，然后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;
}
```

8.15 切线空间至世界空间法线转换工具

切线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，不会影响你的游戏的运行时性能。

本部分包含以下子部分:

8.15.1 关于切线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。为减轻负载，冰穴演示将世界空间法线贴图用于静态几何体，而不是切线空间法线贴图。

切线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采样的法线。

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。这可确保为纹理采样的法线已在世界空间中正确定向。此更改之所以可以实现，是因为冰穴演示光照是在一个自定义着色器中计算的，而 Unity 标准着色器使用切线空间法线贴图。

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，用于执行转换。

该工具在 Unity 编辑器中离线运行，不会影响你的游戏的运行时性能。

8.15.2 C# 脚本

你必须将 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，请自行创建。

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。

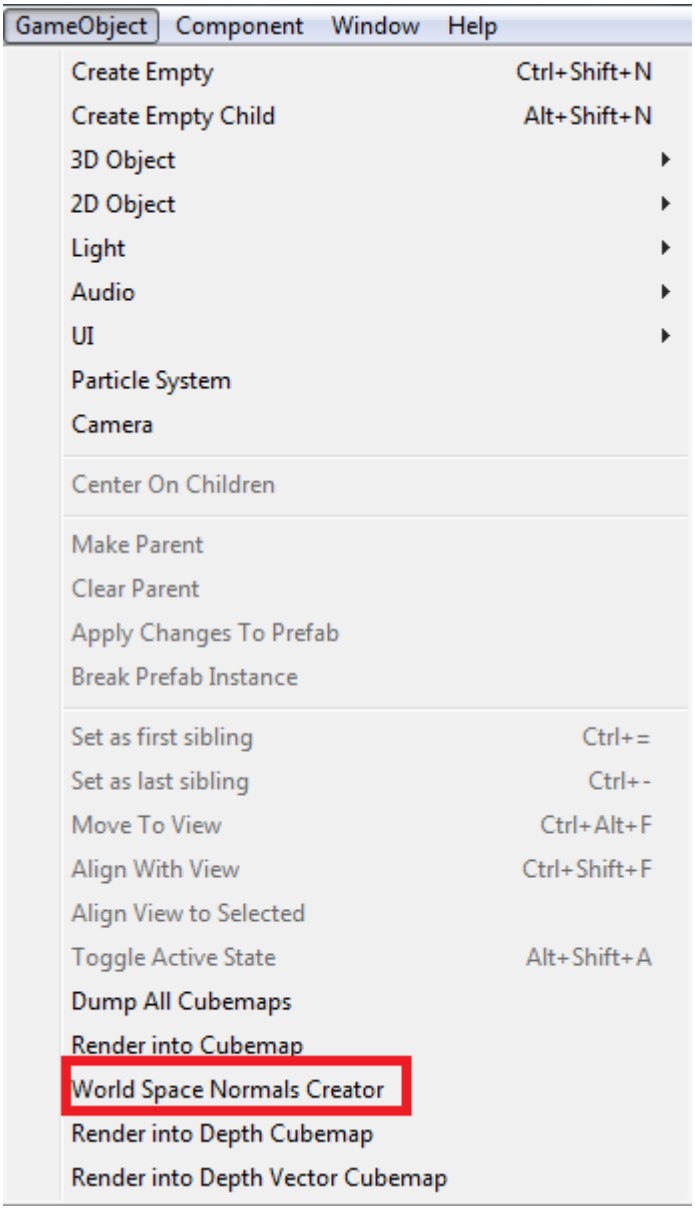

图 8-88 脚本添加的 GameObject 菜单选项

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。从此类派生可以生成编辑器向导。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。

向导窗口的字段是该类的公共成员。在本例中，只有 `_currentObj` 是公共的，因此向导窗口只有一个字段。

下图显示了自定义向导窗口：

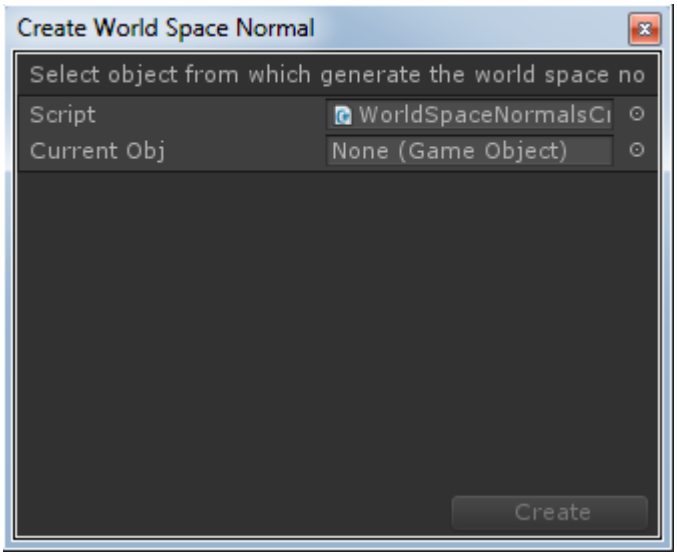

图 8-89 自定义向导窗口

选中了对象并且单击了**创建**按钮后，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镜头，此镜头将新的世界空间法线渲染到 `RenderTexture`。为此，镜头被设置为正交模式，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镜头：

```
// 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;
```

镜头被指向对象。这可防止对象在视锥体剔除期间被移除而不会被渲染：

```
_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。

清屏颜色设置为 `(0.5,0.5,0.5)`，因为必须指出法线的负方向。

```
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);
```

镜头渲染了场景后，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);
    }
```

镜头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 30 位被设为 1。

8.15.3 WorldSpaceNormalCreator 着色器

实现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，并取反 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. 将法线从切线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切线空间法线贴图：

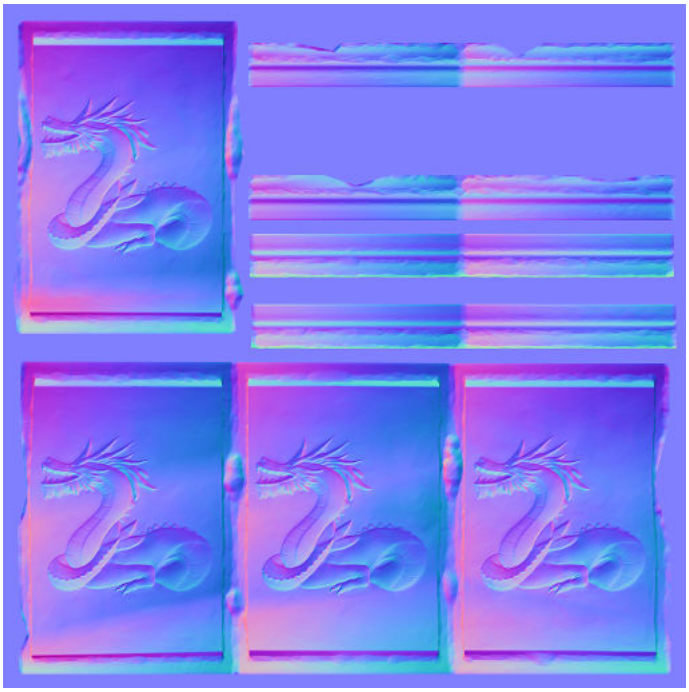

图 8-90 原始切线空间法线贴图

下图显示了该工具生成的世界空间法线贴图：

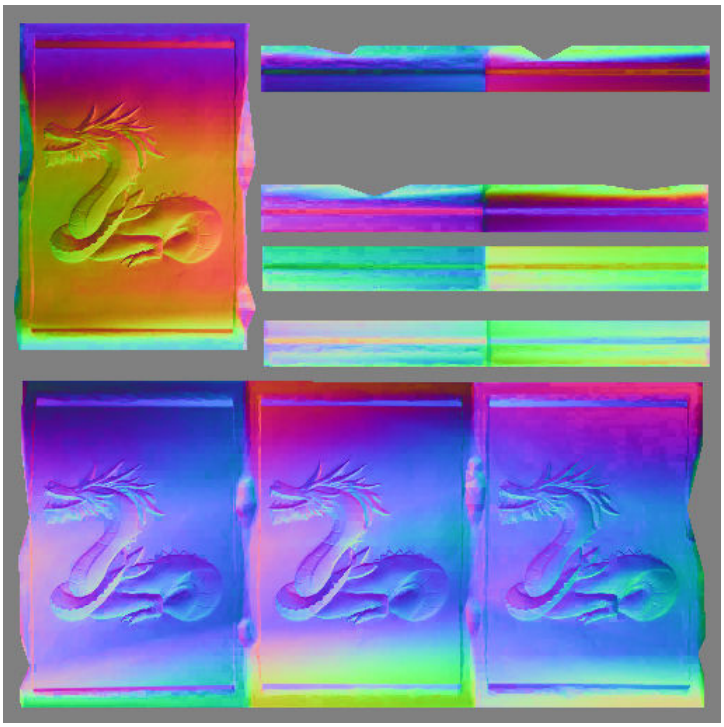

图 8-91 生成的世界空间法线贴图

8.15.4 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");
    }

}
```

8.15.5 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 9 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现反射的一些区别。

它包含以下部分:

9.1 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 支持的设备实现了内部优化。

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 SteamVR 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。

9.2 Unity VR 移植流程

移植应用程序或游戏到原生 Unity VR 是一个多阶段流程。

移植应用程序到 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你的设备的签名文件，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截屏。

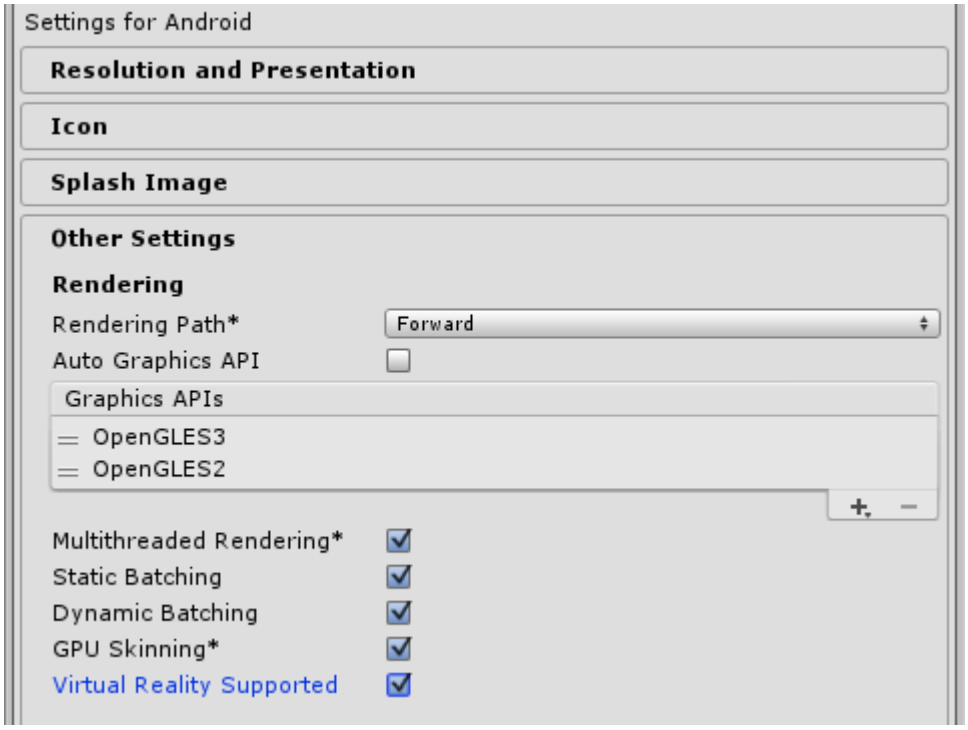

图 9-1 播放器设置窗口

4. 设置镜头的父级。所有镜头控制必须设置相对于此镜头父级的位置和朝向。
5. 根据需要，将镜头控制与 VR 头戴装置触控板关联。
6. 对于 Android 设备，在启用**开发者选项**菜单并打开 **USB 调试**。

   下图显示了**开发者选项**菜单。

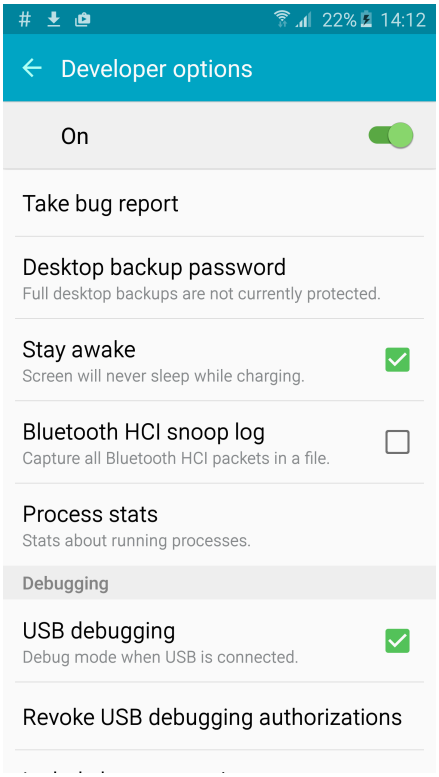

图 9-2 开发者选项菜单

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，它会提示你将设备插入头戴装置。如果设备尚未做好 VR 准备，它将提示你连接网络下载 Samsung VR 软件。

本部分包含以下子部分：

· *9.2.1 启用 Samsung Gear VR 开发者模式* on page 9-208.

9.2.1 启用 Samsung Gear VR 开发者模式

开发者模式可以帮助你视觉化呈现运行的 VR 应用程序，而不必将设备插入 VR 头戴装置。

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。

—————— 说明 ——————

使用开发者模式将缩短手机的电池续航时间，因为它会覆盖其它头戴装置在不使用时关机的设置。

————————————

下图显示了来自 Samsung Gear VR 开发者模式视图的冰穴截屏示例。

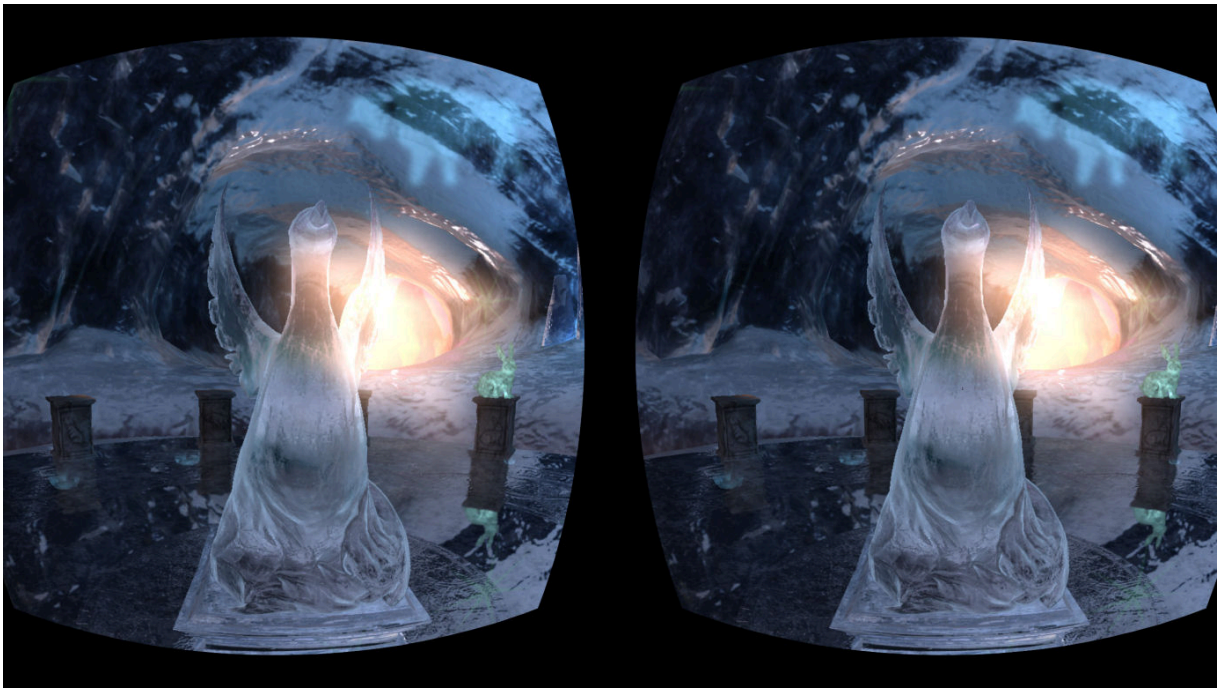

图 9-3 在 Samsung Gear VR 开发者模式中运行的 VR 应用程序的截屏示例

9.3 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，确保用户体验舒适。

在非 VR 游戏中悦目的镜头动画可能会同一游戏的 VR 版本中令人不适。例如，非 VR 的冰穴演示具有针对镜头的动画模式。部分用户无法接受这种不适感，并会发生晕动病，尤其是当镜头向后移动时。去除此模式可以避免造成这种让人不舒服的体验。

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镜头，而这在 VR 设备上却不可行，因为用户无法触碰触控屏幕。冰穴 VR 演示设计为可以在 Samsung Gear VR 上运行，其头戴装置一侧配有触控板。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 头戴装置上的触控板。

图 9-4 Samsung Gear VR 头戴装置触控板位置

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，它会根据镜头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。

本部分包含以下子部分:

· *9.3.1 镜头的外部设备控制* on page 9-210.

9.3.1 镜头的外部设备控制

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。这些事件触发镜头的运动。

下图显示了可用于冰穴演示的蓝牙控制器。

图 9-5 控制冰穴演示的蓝牙控制器

9.4 VR 中的反射

反射在任何 VR 应用程序中都很重要。现实世界中有许多反射现象，所以游戏或应用程序中也必须要有，否则人们会感到不对劲。

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。

合理实现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 中实现反射时你必须考虑一些额外的问题。

本部分包含以下子部分:

9.4.1 使用局部立方体贴图的反射

你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 *8.2 使用局部立方体贴图实现反射* on page 8-121。

9.4.2 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。必须将不同反射技巧生成的结果组合在一起，才能呈现出用户所见到的结果。

有关组合不同反射类型的信息，请参见 *8.3 组合反射* on page 8-137。

9.4.3 立体反射

在非 VR 游戏中，只有一个镜头视点。而在 VR 中，每只眼睛都有一个镜头视点。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现这些反射，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 中实现立体平面反射

在 VR 游戏中实现立体反射需要对非 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。

你必须创建两个镜头，并且为每个镜头创建要渲染到的目标纹理。禁用两个镜头，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;
}
```

此脚本使用主镜头的位置和朝向设定反射镜头的位置和朝向。为此，它将调用 `SetUpReflectionCamera()` 函数，紧接在左右反射镜头渲染之前。下列代码演示了此函数的实现方式。

```
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;
}
```

此函数计算反射镜头的视图和投影矩阵。它决定了反射变换，以应用到主镜头的视图矩阵 `worldToCameraMatrix`。

为设置每只眼睛的镜头位置，可将下列代码添加到 `Matrix4x4 m = Camera.main.worldToCameraMatrix * reflection;` 这一行后面：

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镜头，以控制左右两个反射镜头的渲染。下列代码演示了此脚本的冰穴实现。

```
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;
    }
}
```

此脚本在主镜头的 `OnPreRender()` 回调函数中处理左右反射镜头的渲染。先对左眼调用此脚本一次，而后对右眼调用一次。`eyeIndex` 变量为各个反射镜头分配正确的渲染顺序，并将正确的反射应用到主镜头的每个眼睛。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的不同纹理

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。

如果你的脚本正常工作，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截屏。

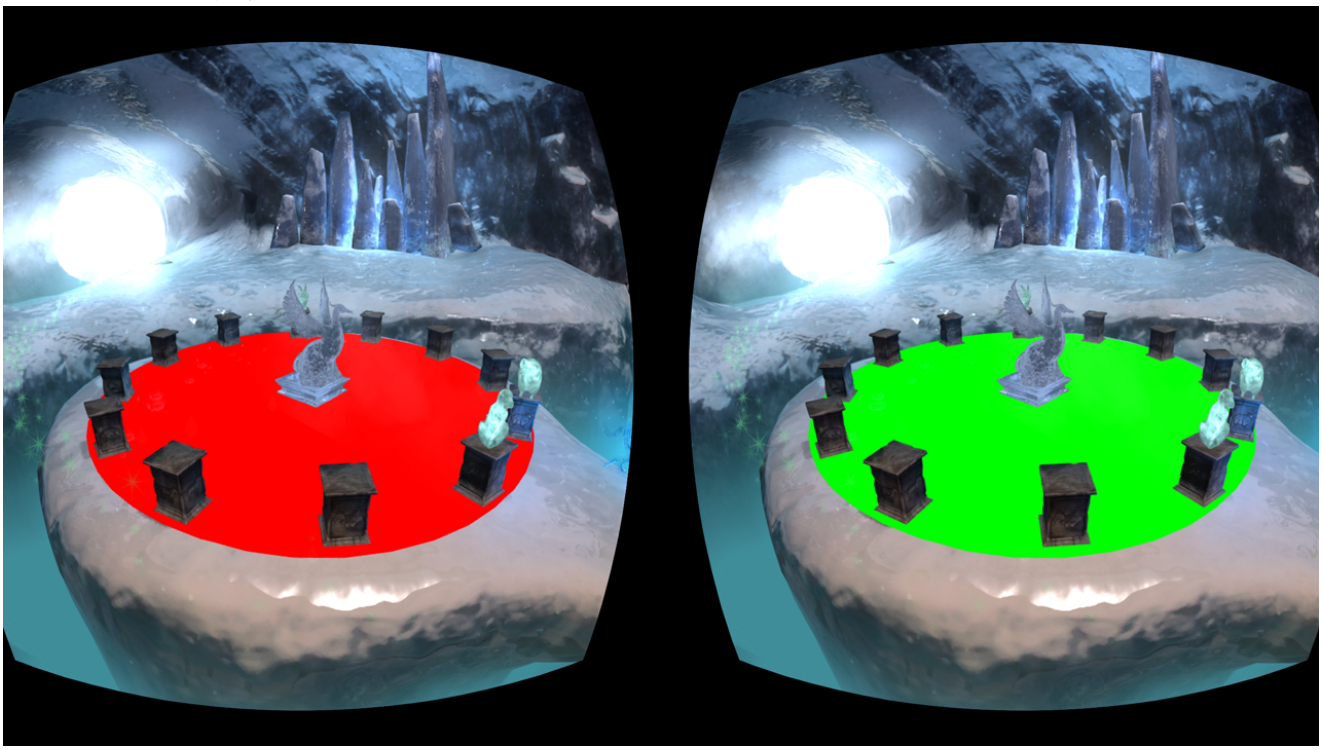

图 9-6 正确反射纹理输出检查的示例

静态立体反射

你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，你必须使用两个反射向量从立方体贴图获取纹素，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镜头位置：

`_WorldSpaceCameraPos`.

但在 VR 中，需要左、右两个镜头的位置。`_WorldSpaceCameraPos` 无法提供左、右两个镜头的位置。因此，你必须使用脚本计算左、右两个镜头的位置，并将结果作为统一变量传递到着色器。

在可以传递镜头位置信息的着色器中声明新的统一变量：

```
uniform float3 _StereoCamPosWorld;
```

计算左、右两个镜头位置的最佳场所是附加到主镜头的脚本中，因为这可以轻松访问主镜头视图矩阵。下列代码演示可如何对 `eyeIndex = 0` 情形进行此操作。

代码修改了主镜头的视图矩阵，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。左眼镜头位置通过统一变量 `_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，你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、右眼镜头位置：

```
output.viewDirInWorld = vertexWorld.xyz - _ StereoCamPosWorld;
```

实现了立体反射时，它会在应用程序以编辑器模式运行时显示，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，立体反射实现将始终运行。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，在附加到主镜头的脚本中使用以下 `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。

9.5 结果

此工作创建了你的游戏的 VR 版本，它实现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，如果不实现，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截屏示例，它展示了所实现的立体反射。

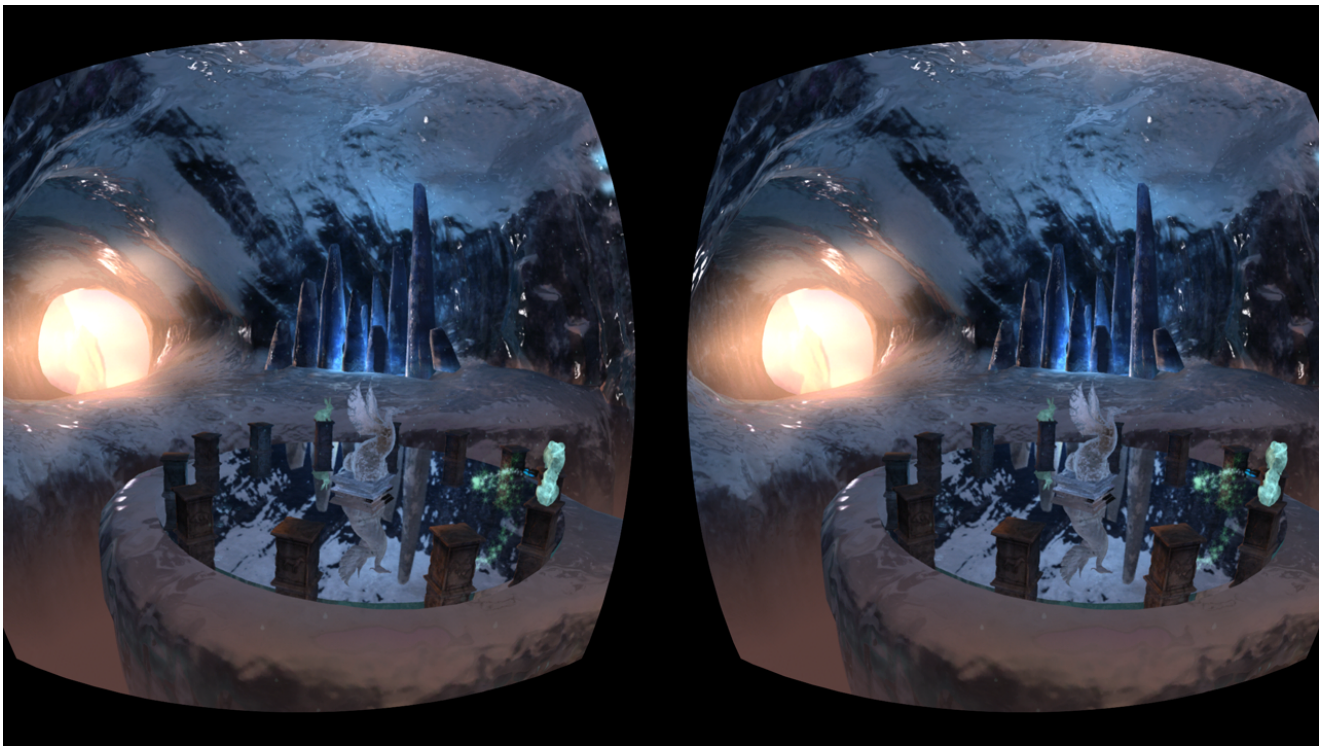

图 9-7 冰穴演示动画截图

第 10 章
高级 VR 图形技巧

本章介绍了用于提高虚拟现实应用程序图形性能的各种技术。

它包含以下部分:

10.1 锯齿

在真实重建物体时，获得的信息如果没有足够的采样，就会出现锯齿现象。在大多数实际应用中，锯齿是信号处理中一个不可忽视的方面。

在音频和视频中，对声音或颜色的原始数据进行采样以将其转换为离散的数字信号时会出现锯齿。在较高的音频频率下和更精细的视频细节中，锯齿现象更加明显。

对原始数据采样不够频繁时，就会发生锯齿。以低于内容最高频率一半的采样率进行采样时将发生锯齿。对于图形，在光栅化过程中，如果运动中的镜头与形成显示结果的离散像素网格相背离，则显示该镜头中的对象时，锯齿现象最明显。

在图形光栅化过程中，如果运动中的镜头与形成显示结果的离散像素网格相背离，则在显示从该镜头中观察到的对象时，将直接看到锯齿现象。

在 VR 环境中，有两种特别明显的锯齿变体，分别是几何锯齿和镜面反射锯齿。

当场景在颜色空间中包含高频输入信号，而这些信号显示在相对较低频率的像素样本上时，例如在两种对比色之间快速过渡时，几何锯齿最明显。

在移动镜头时，这种锯齿现象在直线上特别明显，使直线看起来就像在爬行。在下图中各个形状的红色边缘上可以看到这种几何锯齿的示例：

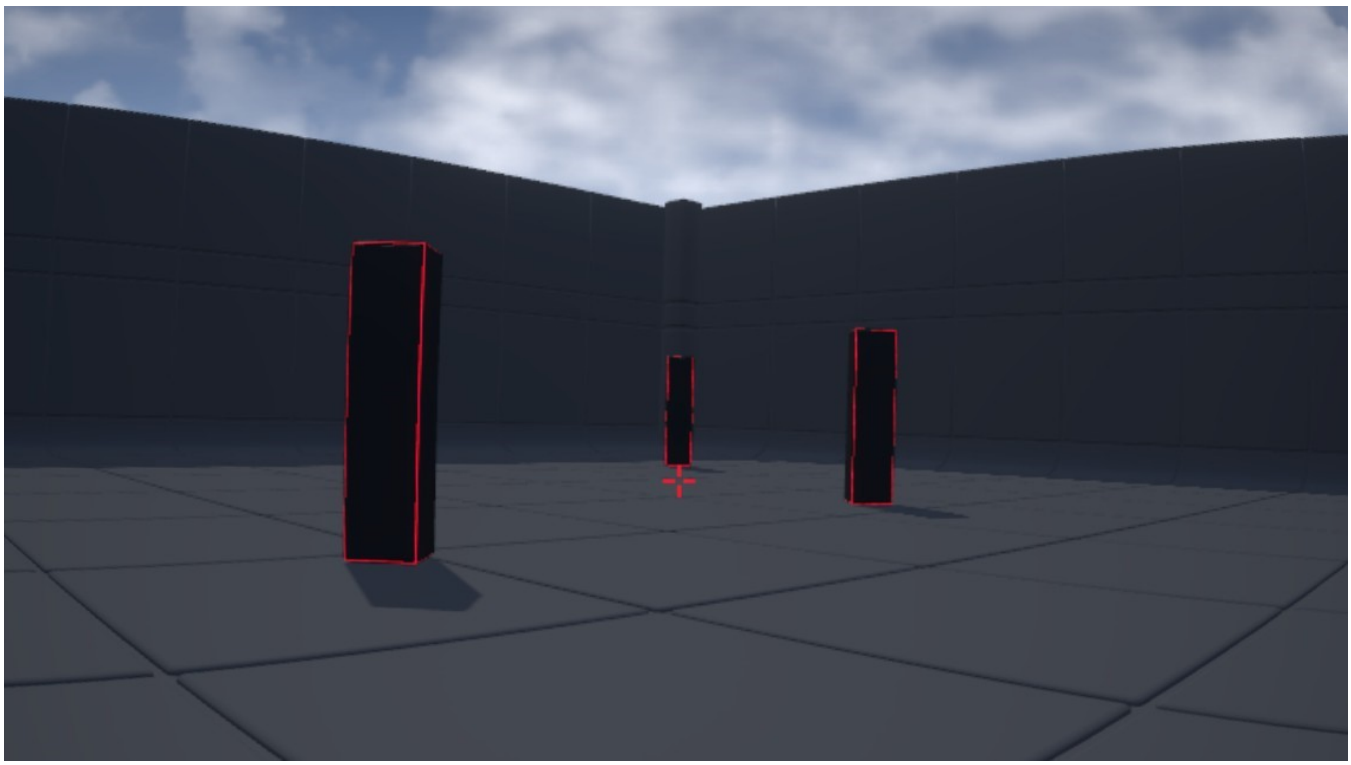

图 10-1 几何锯齿的示例

将多边形数很多的网格渲染到低分辨率显示器上时，由于宽度为几个像素的多边形数量庞大，导致细节过多，因此这种情况下也会发生几何锯齿。大量的多边形意味着大量的边缘，并且几乎没有像素可渲染这些多边形，这会导致更多的锯齿。

同样，三角形密度过高会导致在 GPU 中进行光栅化的效率不高，这是因为虽然 GPU 在几何体上花费了很多周期，但几何体太小，无法对最终图片产生重大影响。此外，GPU 在光栅化几何体上花费过多的周期意味着可用于执行片段着色器的周期数减少。这是因为 GPU 必须花费很多周期来光栅化几何体，而不是执行片段着色器。

锐利高光对象在镜头或对象移动时会突然“弹入”或“弹出”视野，因此会产生镜面反射锯齿。这种锯齿本身看起来是在不同帧之间闪烁的像素，即在一帧中出现镜面反射效果而在下一帧中不

出现镜面反射效果。这种闪烁效果让用户感到烦恼，因为它会分散他们的注意力，进而影响沉浸感并降低用户体验质量。下图的墙壁中的凸脊部分显示了镜面反射锯齿。

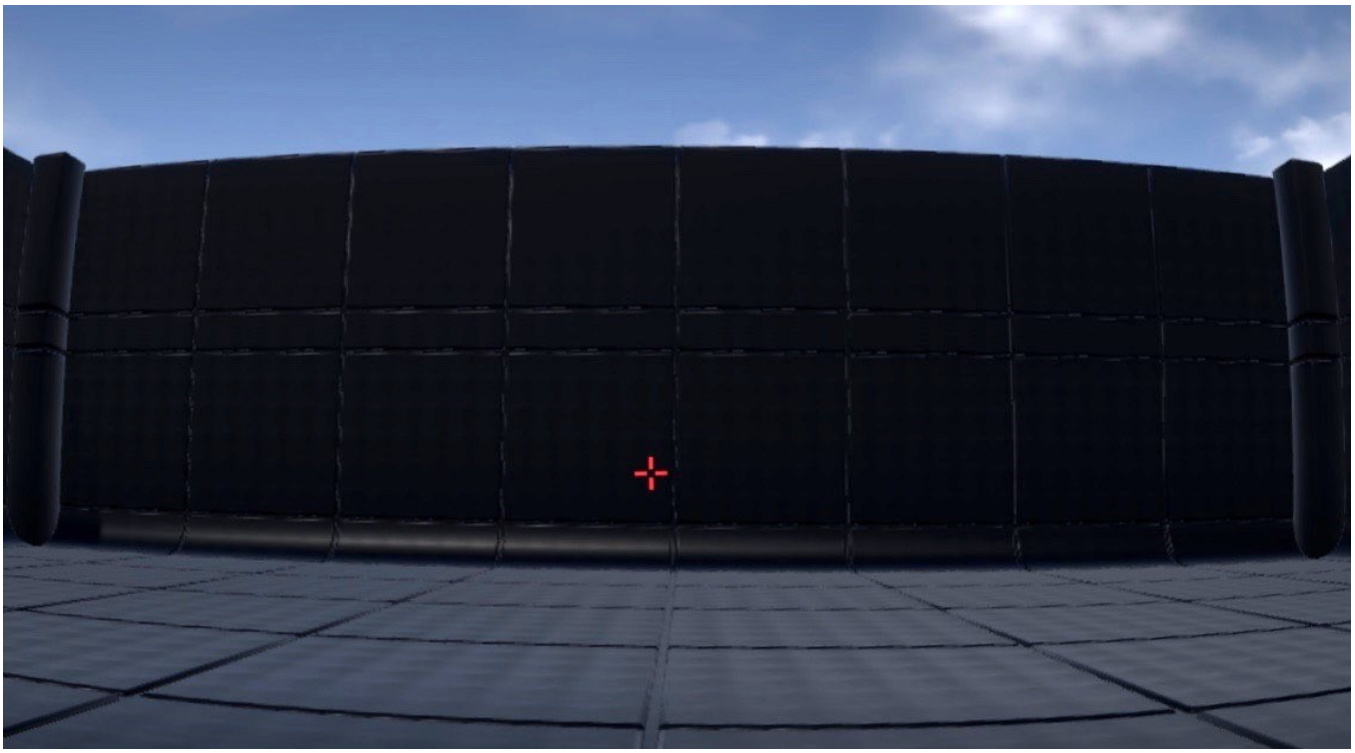

图 10-2 镜面反射锯齿的示例

常见的抗锯齿技术（例如超级采样）由于计算复杂度很高，因此在移动平台上不可行；另外，尽管*多重采样抗锯齿* (MSAA) 既有效又高效，但仅对几何锯齿起作用，因此对于在着色器内发生的欠采样失真（例如镜面反射锯齿）无效。

由于计算机图形（尤其是 VR）中存在多种类型的锯齿，因此需要多种方法来缓解每种不同的锯齿。

10.2 多重采样反锯齿

多重采样反锯齿 (MSAA) 是一种比超级采样更高效的抗锯齿技术。超级采样通过在较高分辨率下对每个像素进行片段着色，在缩小到显示分辨率之前，以较高的内部分辨率渲染图像。

MSAA 正常进行顶点着色，但随后将每个像素分成子样本，使用子样本逐位覆盖掩码进行测试。如果任何子样本通过了覆盖测试，则会进行片段着色，将片段着色的结果存储在通过覆盖测试的每个子样本中。

尽管 MSAA 只减轻了两个三角形相交处的几何锯齿现象，但它对每个像素只进行一次片段着色，因此比超级采样更高效。

4x MSAA 减少了三角形边缘的“锯齿”，其为虚拟现实应用程序带来的画质提升远远超过了其消耗的算力，应该尽可能使用。

使用 4x MSAA 时，Mali GPU 设计为全片段吞吐量，因此使用时沿三角形边缘生成的额外片段造成的性能影响很小。

本部分包含以下子部分:

· *10.2.1 在 Unity 中实现多重采样反锯齿* on page 10-220.

10.2.1 在 Unity 中实现多重采样反锯齿

在 Unity 中，如果使用 URP，则在*通用渲染流水线* (URP) 设置中设置*多重采样反锯齿* (MSAA)。否则，在**项目质量设置**面板中设置。

使用 URP 面板启用 MSAA:

1. 进入**资产**窗口。
2. 在**检视**窗口中，进入“质量”下拉菜单。
3. 从下拉菜单中选择所需的 MSAA 层级。

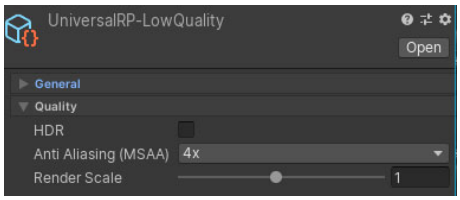

图 10-3 URP·MSAA 设置面板

在**项目质量设置**面板中启用 MSAA:

1. 进入**编辑** > **项目设置**。
2. 选择**质量**，即可打开**项目质量设置**面板。
3. 选择**反锯齿**下拉菜单，然后选择适当的设置。如果可能，最好选择 4x 多重采样。

——— **说明** ———

确保为所有质量层级设置反锯齿值。

———————————

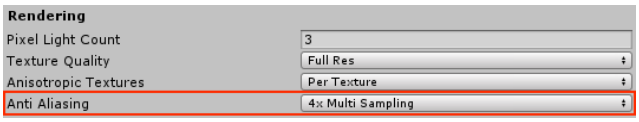

图 10-4 MSAA 设置

10.3 Mipmap 贴图

Mipmap 贴图用于对高分辨率纹理进行逐级缩小和过滤，因此每个后续的 Mip 层级都是前一个层级面积的四分之一。这意味着纹理及其所有生成的 Mip 所需的大小不超过原始纹理大小的 1.5 倍。

Mipmap 贴图可由美术师手动生成，也可由计算机生成，并会上传到 GPU。然后，GPU 为正在执行的采样选择最适合的 Mip。从较小的 Mip 层级进行采样有助于最大限度减少纹理锯齿，保持表面上纹理的清晰度，并防止在远处表面上形成波纹图案。

图 10-5 512x512 纹理具有 9 个低至 1x1 的 Mip 层级

在单个纹理图谱中存储多个连续的 Mipmap 贴图层级时，必须谨慎，因为使用注视点渲染时可能会出现视觉问题。

注视点渲染使用眼动追踪以更高的质量显示用户当前注视的图形。在注视点之外的区域，图像质量会大大降低。

当以原生分辨率渲染一个图块，而以四分之一分辨率渲染相邻图块时，会出现问题。这意味着相邻的图块会从低两个层级的 Mipmap 贴图采样纹理。

采样期间会出现一个问题。例如，当纹理过滤期间某个纹素与其相邻纹素模糊到一起时，可能会错误地从纹理图谱中的相邻层级渗色。

结果，位于两个不同注视区域的不同图块中的两个相邻像素会表现出色差。通过扩展纹理的边缘以便在贴图集的进入区域之间创建隔离间隙可以解决此问题。

本部分包含以下子部分:

· *10.3.1 在 Unity 中进行 Mipmap 贴图* on page 10-222.

10.3.1 在 Unity 中进行 Mipmap 贴图

Mipmap 贴图自动生成功能在**检视**窗口中启用。

若要生成 Mipmap 贴图，请执行以下操作：

1. 在项目窗口的**资源**部分选择一个纹理，打开**纹理检视**窗口。
2. 启用**生成 Mipmap 贴图**选项。

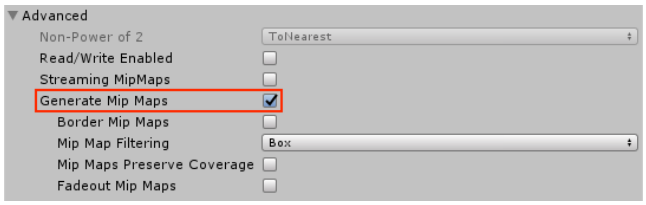

图 10-6 生成 Mipmap 贴图设置

10.4 细节层次

细节层次 (LOD) 是一种随着对象与观察者之间的距离增加而减少该对象的细节和复杂度的技术。

通过使用 LOD，对象会在观察者靠近时呈现高度的几何细节。因此，对象看起来精细而准确。当对象远离屏幕时，会自动切换到低层级细节的模型，向玩家显示精细程度较低的模型。

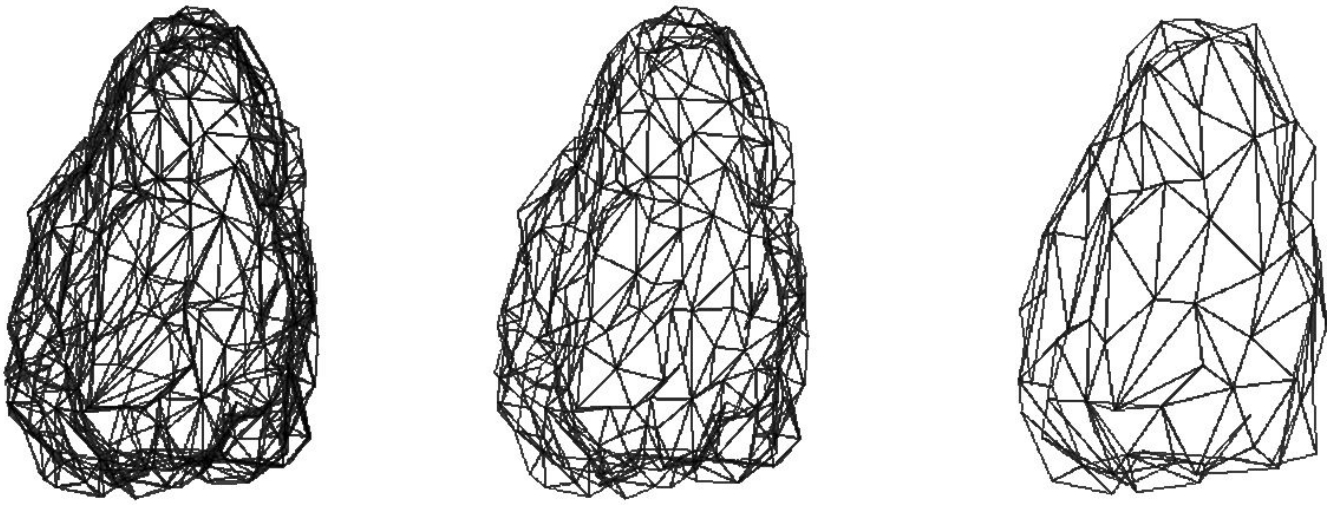

图 10-7 单个网格的三种 LOD

降低模型的细节层次 (LOD) 具有两大好处：

- 降低几何锯齿发生的可能性，从而减少失真。
- 降低需要着色的顶点数量，从而提高性能。

由于几何图形很小，创建 LOD 时须确保每个 LOD 之间的顶点数有显著差异，从而防止不必要的过采样。

有关 LOD 的其他信息，请参阅*几何最佳实践章节* on page 5-70 中的 LOD 部分。

演示 LOD 效果的画面示例

以下图像和视频展示了物体靠近或远离摄像机时，LOD 是如何工作的：

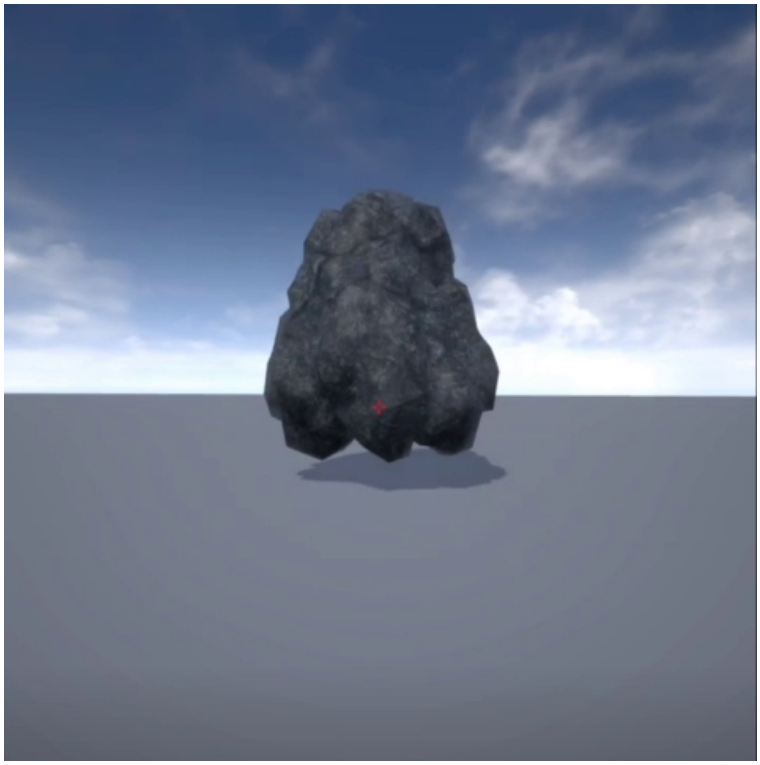

图 10-8 最终 LOD 渲染示例

以下网址链接到一个视频，展示摄像机靠近物体时，单个网格通过三个独立 LOD 的示例。视频显示，摄像机离物体越近，就越能看到从远处看不到的细微细节。而靠近细微细节有助于提高物体的真实感。

https://developer.arm.com/graphics/videos/level-of-detail-example

本部分包含以下子部分:

· *10.4.1 在 Unity 中采用细节层次* on page 10-224.

10.4.1 在 Unity 中采用细节层次

你可使用**组件**菜单选项在 Unity 中采用*细节层次*(LOD)。

要为模型启用 LOD，请执行以下操作:

1. 选择所需模型。
2. 进入**对象检视面板**中的**组件**选项。
3. 选择**渲染**。
4. 然后选择 **LOD 组**。

图 10-9 为对象启用 LOD 组

选择 **LOD 组**后，显示**对象检视面板**中的 LOD 组窗口。可以选择不同的 LOD 层次以显示**渲染器面板**，在此你可以选择具有所需复杂度的网格。你可以调整可用 LOD 层级的大小，具体取决于你希望应用程序在何处转换 LOD。

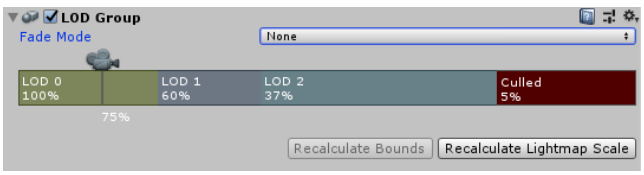

图 10-10 LOD 设置

必要时，可将**渐变模式**设置为 **Cross Fade** 或 **SpeedTree**。根据对象的应用程序，**渐变模式**显示可在着色器中访问的混合因子。混合因子可实现 LOD 之间的平滑混合。

Unity 还支持着色器 LOD，支持根据 `Shader.globalMaximumLOD` 或 `Shader.maximumLOD` 的当前值应用不同的着色器或着色器通道。

利用着色器 LOD 可以应用更高效的着色器，但同时这些着色器的硬件需求也非常高。

10.5 颜色空间

颜色空间规定了分配给每种颜色的数值。颜色空间还定义了每种颜色在空间中分配的面积，以及在该颜色中有多少变化。

最初是在伽马颜色空间中进行渲染的。其他情况下，因为显示器设计为显示伽马颜色空间图像，在成图出现在显示器上之前，需要对其进行伽马校正。

随着*基于物理的渲染* (PBR) 的出现，已经转变为在线性颜色空间中进行渲染。这样有助于使多个光源的数值准确地在着色器内累积。在伽马颜色空间中，由于伽马颜色空间中渲染时的固有曲线，这种添加在物理上是不准确的。

在线性颜色空间中渲染有助于减少镜面反射锯齿，因此具有更多好处。在伽马颜色空间中渲染时，增加场景中的亮度会导致对象快速变成白色，从而导致镜面反射锯齿效果，因此在线性颜色空间中渲染更有优势。在线性颜色空间中，对象会线性变亮，以阻止对象快速变白，因而降低镜面锯齿的风险。

本部分包含以下子部分:

· *10.5.1 如何在 Unity 中选择颜色空间* on page 10-226.

10.5.1 如何在 Unity 中选择颜色空间

在 Unity 中，需要在**项目播放器设置**中选择颜色空间。

要启用线性颜色空间，请执行以下步骤:

1. 进入**编辑** > **项目设置**。
2. 选择**播放器**以打开**项目播放器设置**面板。
3. 点击**播放器设置** > **其他设置**选项，打开**颜色空间**下拉选项。
4. 在**渲染**区域，从下拉菜单中选择**线性**选项。

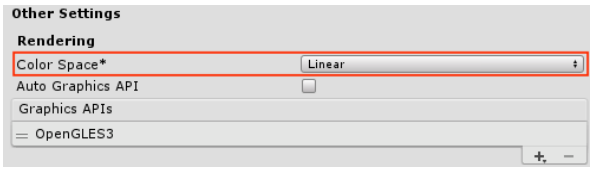

图 10-11 颜色空间设置

说明

应用程序设置为线性颜色空间时，必须将图形 API 设置为 OpenGLES 3.0。要选择正确的 API，清除**自动图形 API** 选项，然后从已命名的图形 API 列表中移除 **OpenGLES2**。**识别**部分中的**最小 API 层级**必须至少设置为 Android 4.3“Jelly Bean”、API 层级 18。

10.6 纹理过滤

纹理过滤是一种用于减轻纹理采样过程中发生的锯齿的方法。

要将纹理映射到渲染对象上时，如果你在屏幕上渲染的像素不完全位于纹理内的像素网格上，就会发生锯齿。相反，由于纹理被映射到的对象对于观察者而言处于任意距离和方向，像素会在一定程度上发生偏移。

将纹理像素（简称纹素）映射到屏幕像素时，可能会出现两种问题。一种问题是纹素大于屏幕像素，在这种情况下必须将纹素缩小以适应屏幕像素。另一种可能发生的情况是纹素小于屏幕像素，在这种情况下必须组合多个纹素以适应屏幕像素。

纹理过滤在很大程度上依赖于 Mipmap 贴图。这是因为在放大期间，要获取的纹素数量永远不会超过四个。但是，在缩小过程中，随着纹理对象向远处移动，整个纹理可以整合到一个像素中。在这种情况下，必须获取每个纹素，然后将其合并以获得准确的结果，但这对于 GPU 而言实现成本太高了。

取而代之的做法是选择四个样本，但这会导致不可预测的数据欠采样。Mipmap 贴图可以克服这一问题，其方法是以不同大小对纹理进行预过滤，因此在对象移开时，GPU 会调用较小的纹理尺寸。

行业通用的过滤方法包括双线性、三线性和各向异性过滤，它们之间的区别是采样了多少个纹素、如何组合这些纹素以及在过滤过程中是否使用了 Mipmap 贴图。

每种过滤方法的性能成本各不相同。因此，选择要使用的过滤类型必须根据具体情况而定，在此过程中，应权衡过滤方法的性能成本与其所提供的视觉优势。

例如，三线性过滤的成本是双线性过滤的两倍，但视觉优势并不总是显而易见的，特别是在将纹理应用于远处的对象时。相反，建议使用 2x 各向异性过滤，因为这种过滤方法通常可以提供更好的图像质量和更高的性能。

使用各向异性过滤时，必须设置最大样本数。但是，由于对移动设备的性能有影响，建议你在使用更多样本数量时，尤其是在八个或更多样本时，应谨慎。此方法最适合于倾斜表面的纹理，例如远处的地面。

以下一组图像显示了不同类型的纹理过滤之间的差异。特别要注意的是，启用 2x 样本各向异性过滤后，双线性和三线性过滤之间的区别是无法区分的。但是使用三线性过滤时会消耗更多的资源。

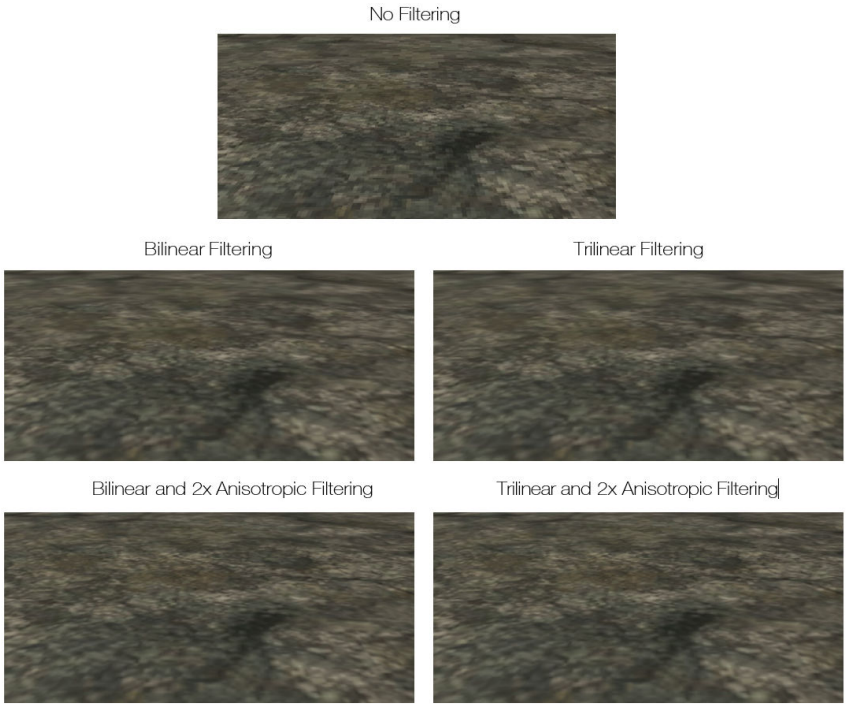

图 10-12 不同纹理过滤方法的比较

有关纹理过滤的其他信息，请参阅*纹理最佳实践章节* on page 6-78。

本部分包含以下子部分:

- *10.6.1 在 Unity 中采用纹理滤波* on page 10-228.

10.6.1 在 Unity 中采用纹理滤波

在 Unity 中，你可在**项目质量设置**面板中设置纹理滤波。

要在 Unity 中为纹理配置纹理滤波设置，请执行以下步骤:

1. 进入**编辑**。
2. 选择**项目设置**。
3. 选择**质量**，以打开**项目质量设置**面板。
4. 从该窗口下拉菜单中选择**各向异性纹理**。
5. 最后选择**每个纹理**选项。

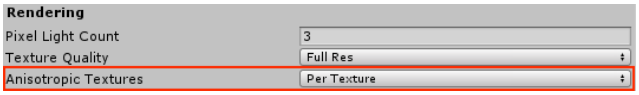

图 10-13 为每个纹理配置纹理滤波设置

然后，在**项目**窗口的**资源**部分选择每个纹理，从而打开**纹理检视**窗口。窗口打开后，为该纹理设置**滤波模式**和**各向异性等级**。

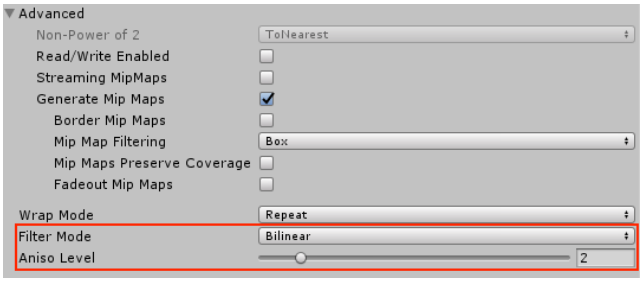

图 10-14 为纹理选择纹理滤波设置

10.7 Alpha 合成

Alpha 合成是一种将图像与背景图像相结合，以形成具有透明外观的合成图像的技术。

Alpha 测试是 Alpha 合成的一种广泛应用方式 。但是，因为 Alpha 通道是逐位进行的，会在对象边缘产生严重的锯齿效果，所以边缘之间没有混合。

而着色器对每个像素仅运行一次，因此多重采样对此没有影响。每个子样本返回相同的 Alpha 值，造成边缘锯齿。

Alpha 混合是一种替代方案。但是，如果没有进行多边形排序，混合就会失败，导致对象渲染错误。然而，启用排序成本较高且会降低性能。

Alpha to Coverage (ATOC) 是一种不同的 Alpha 合成方法，有助于减少锯齿。ATOC 将片段着色器的 Alpha 组件输出转换为覆盖掩码，并将其与多种采样掩码相结合。然后，ATOC 使用“与”操作符，只渲染经过“与”操作的像素。

演示 Alpha 测试和 Alpha 覆盖的图像示例

以下链接中的图像和视频展示了左侧基础 Alpha 测试实现方式和右侧 Alpha 覆盖实现方式之间的区别：

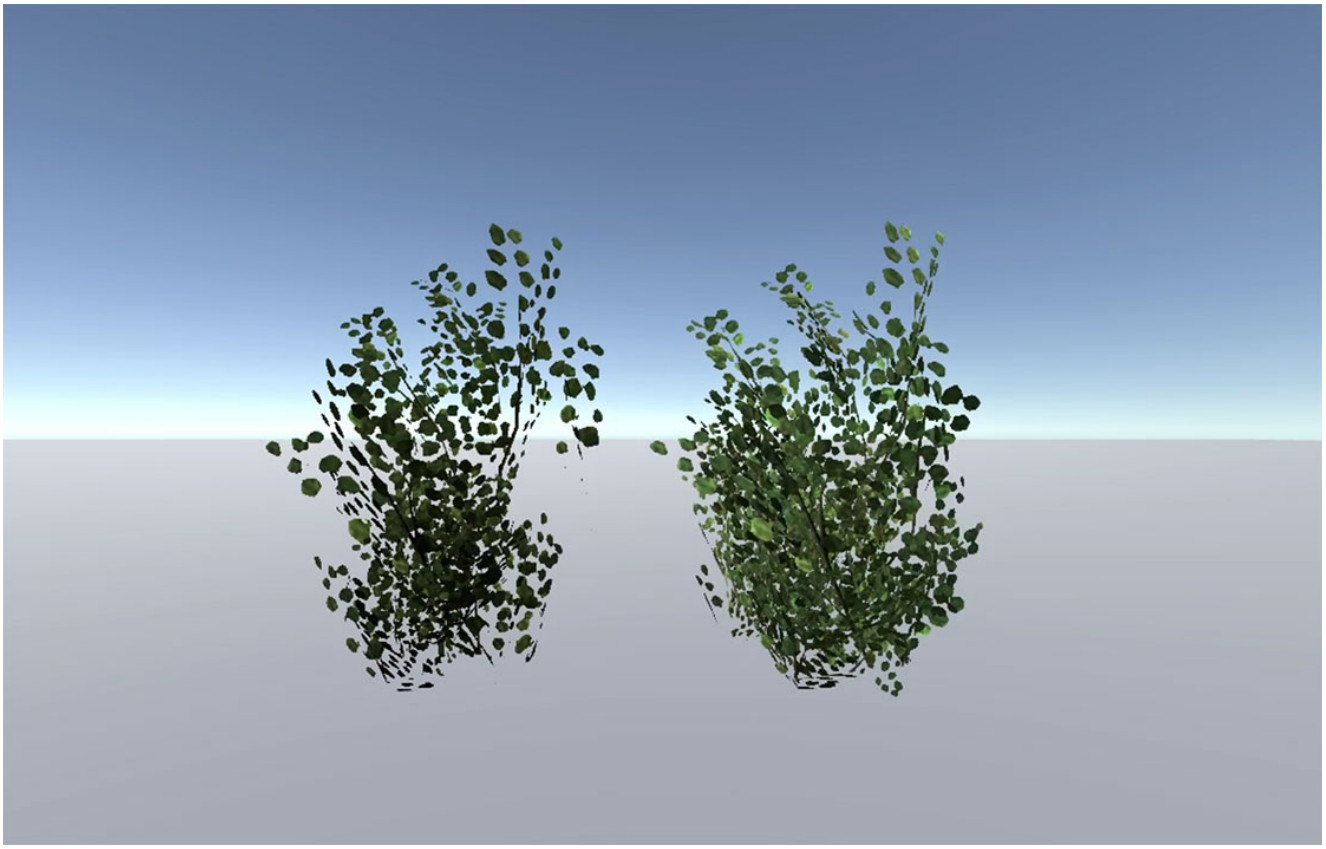

图 10-15 Alpha 合成示例

https://developer.arm.com/graphics/videos/alpha-compositing-example

使用 ATOC 可以大大降低锯齿和闪烁现象。

本部分包含以下子部分：

· *10.7.1 在 Unity 中进行 Alpha 合成* on page 10-230.

10.7.1 在 Unity 中进行 Alpha 合成

ATOC 应用于 Unity 着色器。

Mipmap 贴图保留覆盖选项 - 自 Unity 2017 起

Mipmap 贴图保留覆盖选项取消了早期版本中所需的 Mipmap 贴图层级计算。你可在**纹理检视**中启用该选项。

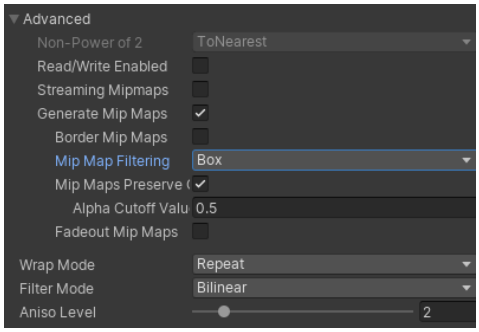

图 10-16 用于 Unity 2017 及之后版本的 Mipmap 贴图保留覆盖选项

要在 Unity 2017 或更高版本中启用 ATOC，你必须首先在“项目”窗口中创建一个新的着色器，然后插入以下着色器代码：

—————— 说明 ——————

将该着色器应用于材质，并将该材质设置为需要 Alpha 合成的对象。

```
AlphaToMask On
                struct frag_in
                {
                float4 pos : SV_POSITION;
                float2 uv : TEXCOORD0;
                half3 worldNormal : NORMAL;
                };

                fixed4 frag(frag_in i, fixed facing : VFACE) : SV_Target
                {
                /* Sample diffuse texture */
                fixed4 col = tex2D(_MainTex, i.uv);

                /* Sharpen texture alpha to the width of a pixel */
                col.a = (col.a - 0.5) / max(fwidth(col.a), 0.0001) + 0.5;
                clip(col.a - 0.5);

                /* Lighting calculations */
                half3 worldNormal = normalize(i.worldNormal * facing);
                fixed ndotl = saturate(dot(worldNormal,
normalize(_WorldSpaceLightPos0.xyz)));
                col.rgb *= ndotl * _LightColor0;

                return col;
                }
```

用于 Unity 2017 之前的版本

要在 Unity 2017 之前的版本中启用 ATOC，你还需要在着色器代码中添加一个 Mipmap 贴图层级计算。就像在 Unity 2017 及更高版本中一样，你必须首先在项目窗口中创建一个新的着色器。然后插入以下着色器代码：

—————— 说明 ——————

将该着色器应用于材质，并将该材质设置为需要 Alpha 合成的对象。

```
Shader "Custom/Alpha To Coverage"
{
    Properties
    {
```

```
        MainTex("Texture", 2D) = "white" {}
    }
        SubShader
    {
        Tags { "Queue" = "AlphaTest" "RenderType" = "TransparentCutout" } Cull Off

        Pass
        {
            Tags { "LightMode" = "ForwardBase" }

            CGPROGRAM
            #pragma vertex vert #pragma fragment frag

            #include "UnityCG.cginc"
            #include "Lighting.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
                half3 normal : NORMAL;
            };

            struct v2f
            {
                float4 pos : SV_POSITION;
                float2 uv : TEXCOORD0;
                half3 worldNormal : NORMAL;
            };

            sampler2D _MainTex; float4 _MainTex_ST;
            float4 _MainTex_TexelSize;

            v2f vert(appdata v)
            {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                o.worldNormal = UnityObjectToWorldNormal(v.normal);
                return o;
            }

            fixed4 frag(v2f i, fixed facing : VFACE) : SV_Target
            {

                fixed4 col = tex2D(_MainTex, i.uv);

                float2 texture_coord = i.uv * _MainTex_TexelSize.zw; float2 dx =
ddx(texture_coord);
                float2 dy = ddy(texture_coord);
                float MipLevel = max(0.0, 0.5 * log2(max(dot(dx, dx), dot(dy, dy))));

                col.a *= 1 + max(0, MipLevel) * 0.25; clip(col.a - 0.5);

                half3 worldNormal = normalize(i.worldNormal * facing);

                fixed ndotl = saturate(dot(worldNormal,
normalize(_WorldSpaceLightPos0.xyz)));
                fixed3 lighting = ndotl * _LightColor0; lighting +=
ShadeSH9(half4(worldNormal, 1.0));
                col.rgb *= lighting; return col;
             }
             ENDCG
         }
     }
}
```

10.8 层次设计

尽管有各种技术可以用来尽量减少锯齿量，但仔细的层次设计对于帮助减少锯齿失真仍然至关重要。层次设计中的错误选择会使每种技术解决方案都变得多余，因此层次设计中需要明智决策，才能发挥作用更大限度地减少锯齿。

为场景创建几何图形时，必须小心对网格边缘进行建模，以避免尖锐或突兀的边缘。例如，如果观看者旋转视角，则楼梯上每级边缘平坦的台阶都会产生锯齿。

然而，当从远处看楼梯时，单级台阶本身也可能是产生锯齿的原因。随着观察者移动，从远处看，每级台阶都变成了闪烁的薄薄的对象。

下图为使用楼梯和坡道时产生的不同锯齿级别的示例：

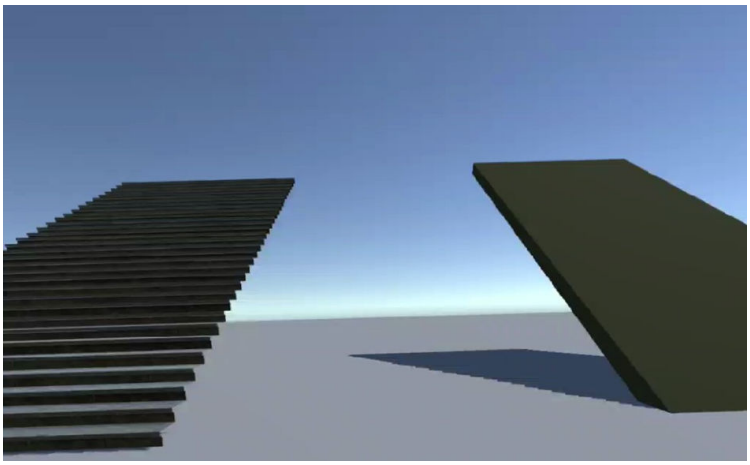

图 10-17 楼梯和坡道锯齿现象对比

对比楼梯和坡道锯齿现象的视频

下图为使用楼梯和坡道时产生的不同锯齿级别的视频示例：

https://developer.arm.com/graphics/videos/ramp-vs-stairs-aliasing-example

尽可能使用光滑的圆形代替有硬边或斜面的形状。当从远处看时，电线或电缆等较细物体更容易造成明显的锯齿现象。这是因为较细的物体被渲染为线条，造成锯齿现象。

必须谨慎使用金属材质。应尽量减少金属的使用，因为金属物体被照亮时会产生镜面效果，会随着观察者的移动而闪烁，从而导致锯齿现象。因此，最好使用哑光材质。

下图比较了金属材质和哑光材质造成的锯齿现象。图的上半部分为金属材质，下半部分为等效的哑光材质。

图 10-18 金属和哑光材质对比

谨慎使用场景照明，因为明亮的灯光会在照亮的对象上产生镜面效果。这样做可能会导致非常明显的镜面反射锯齿。镜面效果在帧之间出现和消失时，注意力将被像素的“闪烁”所吸引。

为了避免高耗能的实时照明计算，并减少照明失真（如带状）的出现，你可以在构建场景时预先烘焙照明。要实现这个目的，场景必须特别设计，使其进行预烘时只需要很少的移动物体且无需实时昼夜循环。

利用灯光探测器有助于减少所需的预烘培次数，这在 6 自由度虚拟现实中尤其具有优势。

与更常见的用例相比，需要更加小心虚拟现实中的反射。屏幕空间反射等技术对于虚拟现实来说计算成本太高。因此，必须采用其他反射技术。对于需要高质量反射的对象，可采用反射探测器和立方体贴图技术，因为反射可以预先渲染以便在运行时使用。

可以采用雾或烟雾等粒子效果隐藏锯齿。这些长期以来一直用于隐藏短渲染距离的技术，也可以用于覆盖由远处的对象引起的锯齿。

下图中，右图演示了添加粒子效果后锯齿的减少情况：

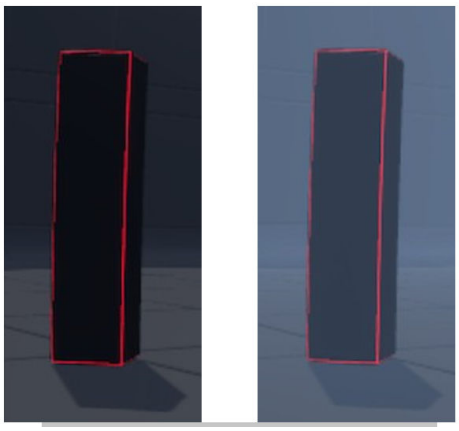

图 10-19 添加粒子效果前后对比

10.9 色带

无法在给定的每个像素位数内准确表示所需的颜色时，就会产生色带。在虚拟现实中，色带表现为不同的颜色带，每个颜色带之间都有明显的变化。

由于用户沉浸在场景中，这些色带在虚拟现实中最明显。因此，用户视线会适应场景中的光线水平。视线调整好后，色带就更加明显了。如果存在很多色带，那么眼睛就需要不断适应不停变化的光线水平，这也会让玩家感到疲劳。

缓解技术 - 抖动

抖动是将噪声引入色带材质或观察者的过程。通过抖动，明显的色带被分解和破坏，变得不再明显。

可以引入许多不同种类的抖动，各个种类要么应用不同形式的噪声，要么采用不同的方法收集所应用的噪声。例如，有些种类会实时生成噪声，而其他种类则从包含随机噪声的纹理中进行噪声采样。

缓解技术 - 色调映射

色调映射是颜色分级的一个子集，将*高动态范围* (HDR) 颜色转换为适合在不支持 HDR 屏幕上显示的*低动态范围* (LDR) 颜色。色调映射使用*查找表* (LUT) 将每种颜色对应转换为新色调的颜色。

将色调映射应用到场景，低光照级别或纹理中的尖锐梯度等会导致色带的因素都能得到减少。

本部分包含以下子部分:

- *10.9.1 在 Unity 中采用抖动* on page 10-234.
- *10.9.2 在 Unity 中采用色调映射* on page 10-235.

10.9.1 在 Unity 中采用抖动

Unity 中的抖动通常是通过后处理包实现的，因此必须添加和设置该包。或者，也可以使用**通用渲染流水线**的镜头中内置的效果。

要启用抖动，请执行以下步骤:

1. 在**窗口**中选择**包管理器**，搜索“后处理”，然后安装该包。确保是你正在使用的版本 2.x。
2. 在项目窗口中创建**后处理配置文件**，方法是在**项目窗口**中单击右键。
3. 在**创建**下，选择**后处理配置文件**。
4. 从镜头的**检视器**中，为场景中需要抖动的每个镜头添加一个**后处理体积**组件和一个**后处理层**组件。
5. 最后，从**后处理体积**中，将创建的**后处理配置文件**设置为“配置文件”。

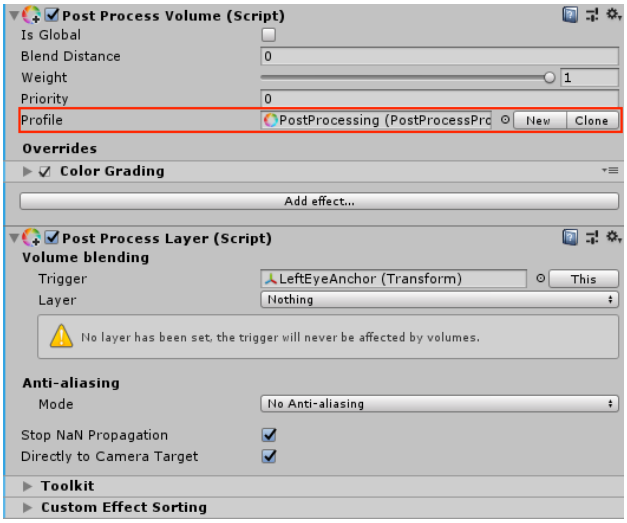

图 10-20 设置后处理配置文件

如以下屏幕快照所示，在 Unity 的**通用渲染流水线**中，你可以启用镜头抖动：

1. 在镜头的**检视器**窗口中的抖动部分，勾选**后处理**复选框。
2. 然后，勾选其下方的**抖动**复选框。

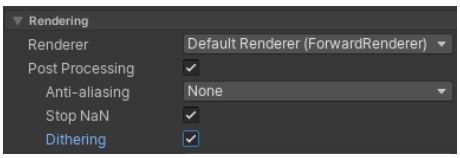

图 10-21 如何启用镜头抖动

10.9.2 在 Unity 中采用色调映射

在 Unity 中进行色调映射需要安装和设置后处理包。

要启用色调映射，请执行以下操作：

1. 在**窗口**中选择**包管理器**，搜索**后处理**，然后安装该包。确保是你正在使用的版本 2.x。
2. 接下来，在**项目窗口**中创建**后处理配置文件**，方法是在**项目窗口**中单击右键。
3. 在**创建**下，选择**后处理配置文件**。
4. 现在选择该**后处理配置文件**。
5. 在**检视器**中，选择**添加效果**…。
6. 在 **Unity** 下，选择 **Unity 颜色分级**，然后将**模式**选项设置为**高清晰度范围**。
7. 勾选色调映射**模式**旁边的复选框，然后在组合框中选择 **ACES**。
8. 从镜头部分的**检视器**中，为场景中需要色调映射的每个镜头添加一个**后处理体积**组件和一个**后处理层**组件。
9. 最后，从**后处理体积**中，将预先创建的**后处理配置文件**设置为**配置文件**。

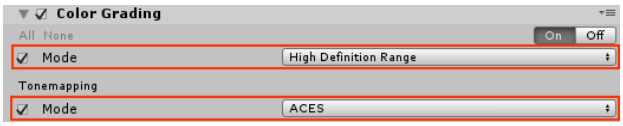

图 10-22 设置色调映射

10.10 凹凸贴图

凹凸贴图是一种用于减少网格顶点数的技术，涉及模拟对象表面的更精细的细节，如凹凸。在用于照明计算之前，通过操纵对象的法线来执行所需的模拟。

虽然法线贴图是一种适合大多数应用的技术，但在虚拟现实中，因为玩家可以轻易改变法线贴图纹理的视角，因此并不太有效。法线贴图技术没有考虑视角的改变，因此深度错觉被打破。

此外，因为法线仅从一个视点生成，法线贴图无法支持虚拟现实头戴装置中使用的立体镜头。这样，每只眼睛都接受同样的法线，但在人眼看来这是不正确的。

对比不同凹凸贴图方法的视频

以下视频显示了法线贴图和视差遮挡贴图的示例：

https://developer.arm.com/graphics/videos/bump-mapping-example

缓解技术 - 法线贴图

法线贴图是凹凸贴图最常见的实现，涉及建模过程中创建网格的高多边形版本和低多边形版本。然后，从高多边形版本导出法线贴图，更精细细节的法线随后被存储在法线贴图纹理中。

渲染时，片段着色器从法线贴图采样，根据采样值生成法线。然后将生成的法线与低多边形版本的表面法线组合，随后在计算照明时使用。接着，照明显示更精细的表面细节，而不需要渲染这些细节的各个顶点。

虽然法线贴图在虚拟现实中通常不那么有效，但是法线贴图仍然比平面材质更有效。尤其是在照明场景中需仔细考虑法线贴图的位置的情况下。

缓解技术 - 视差遮挡贴图

视差遮挡贴图与法线贴图相似。然而，移动纹理坐标时，视差遮挡贴图考虑了观察者相对于表面法线的角度。

因此，视角更陡时，纹理坐标偏移程度更高。这样可保持深度错觉。

视差遮挡贴图计算消耗很大。所以只在观察者可以接近的较小材质上使用这种技术。因为视角不会发生很大变化，距离较远的纹理从视差遮挡贴图中获得的好处很少。

本部分包含以下子部分：

- *10.10.1 在 Unity 中采用法线贴图* on page 10-236.
- *10.10.2 在 Unity 中进行视差遮挡贴图* on page 10-237.

10.10.1 在 Unity 中采用法线贴图

在 Unity 的许多内置着色器中都可以使用法线贴图。

要为材质添加法线贴图，请执行以下步骤：

1. 在**项目窗口**中选择材质。
2. 打开**材质检视器面板**，并选择一个支持法线贴图的着色器，例如**标准**、**通用渲染流水线**或**凹凸漫反射**（适用于移动平台）。
3. 最后设置所需的**法线贴图**纹理。

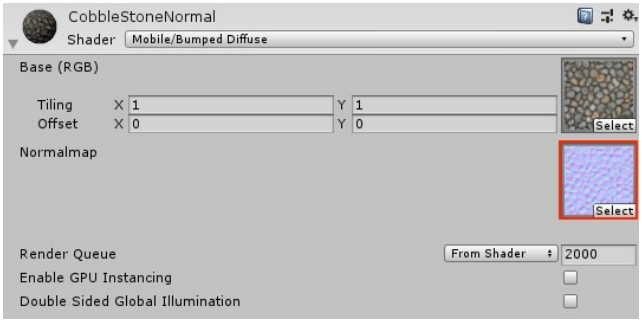

图 10-23 添加法线贴图纹理

10.10.2 在 Unity 中进行视差遮挡贴图

你可使用 Unity 的内置着色器进行视差遮挡贴图。

要为材质添加视差遮挡贴图，请执行以下步骤：

1. 在**项目窗口**中选择材质。
2. 打开**材质检视面板**，并选择一个支持视差漫反射贴图的着色器。例如，**标准**着色器。
3. 最后设置所需的**反照率**、**法线贴图**以及**高度贴图**纹理。

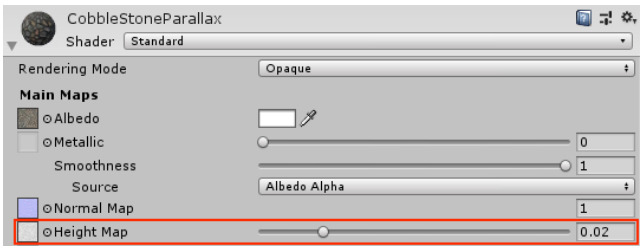

图 10-24 为材质添加视差遮挡贴图

10.11 阴影

传统的阴影映射方法在移动设备上计算量很大。阴影贴图需要额外的帧缓冲和渲染通道，从而导致性能严重下降。

因此，移动设备上的阴影缓冲区通常分辨率较低，并且不进行任何过滤。因而，阴影通常会引入大量锯齿，由于阴影瑕疵和硬阴影等失真现象，会破坏画面的真实性。

下方截屏展示了左边典型阴影和右边 Blob 阴影之间的比较：

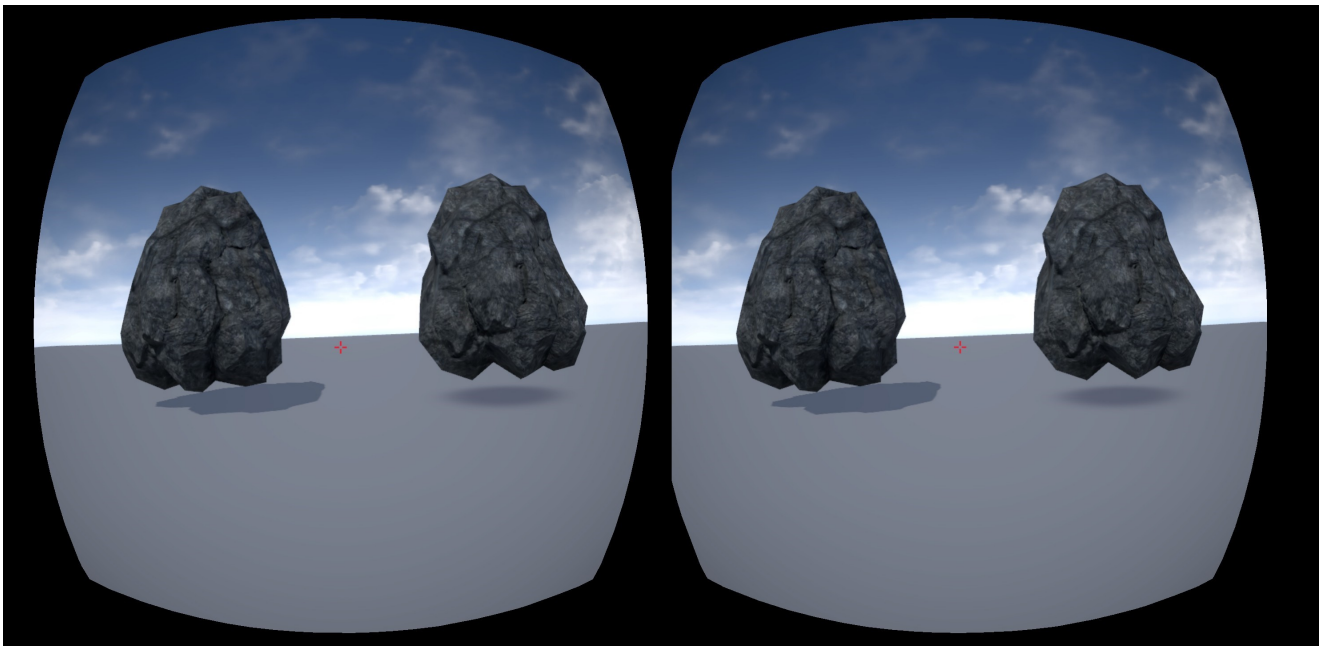

图 10-25 阴影贴图和 Blob 阴影的比较

另一个值得考虑的技术是使用烘焙阴影。与阴影贴图不同，阴影烘焙不需要额外的渲染通道。相反，高质量的阴影被烘焙到他们投射阴影的纹理中。

缓解技术 — Blob 阴影

VR 中推荐的做法是尽可能避免渲染阴影。但是，给玩家脚下添加阴影等需要阴影的场景中，可以在对象下面渲染 Blob 阴影。与典型的阴影贴图技术相比，Blob 阴影大幅度降低锯齿，同时提高了性能。

本部分包含以下子部分：

· *10.11.1 在 Unity 中采用 Blob 阴影* on page 10-238.

10.11.1 在 Unity 中采用 Blob 阴影

Blob 阴影是 Unity 中的场景组件。你可从**资源商店**中免费获取。

采用 Blob 阴影的步骤如下：

1. 从 Unity 资源商店导入 **Unity 标准资源**。
2. 进入 **BlobShadowProject** 预制件，其路径为**资源** > **标准资源** > **效果项目** > **预制件**。
3. 将 **BlobShadowProject** 预制件拖到**层级**中，作为需要 Blob 阴影对象的子对象。

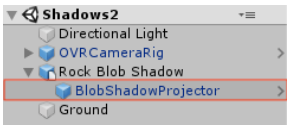

图 10-26 设置子对象

设置完成后，就会显示 Blob 阴影。完成设置 Blob 阴影的步骤如下：

1. 编辑**预制件**选项。请确保设置了 Blob 阴影与投射该阴影的对象的相对位置。
2. 选择投射阴影的对象，然后将**投射阴影**选项设为**关**。

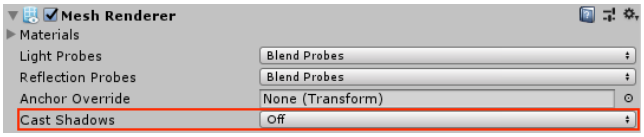

图 10-27 为对象禁用投射阴影

第 11 章
Vulkan

本章将介绍 Vulkan 及其启用方法。

Vulkan 是 Khronos Group 提供的一个跨平台图形和计算 API，与 OpenGL 和 OpenGL ES 相比，它能提供许多优势。这些优势包括：

- 为移动设备、桌面设备、控制台、服务器和嵌入式系统提供一个统一的 API 框架。
- 具备各种功能，为硬件提供支持。
- 尽量降低驱动程序消耗，帮助 Arm Mali GPU 硬件实现高性能。
- 应用程序可获取更多对 GPU 和计算资源的低级别访问权限。
- 减少应用处理器瓶颈。
- 支持多线程和多处理。
- 高效使用多个应用处理器。
- 使用适用于着色器的 SPIR-V 中间语言，减少运行时内核编译时间。
- 无需附带着色器源代码。
- 应用处理器消耗更低，驱动程序进行了简化，并使用了更多片上存储器，因此能耗更低。

它包含以下部分：

11.1 关于 Vulkan

Vulkan 是 Khronos Group 的跨平台图形和计算 API。

与较旧的 OpenGL 和 OpenGL ES 标准相比，Vulkan 拥有很多优势：

统一且可移植

Vulkan 可为移动设备、桌面设备、游戏机和嵌入式系统提供一个统一的 API 框架。
它可在多种实现间进行移植，并可用于多种应用程序。

驱动程序更简单

Vulkan 使用的驱动程序更简单，可最大限度地减少驱动程序开销。更低的延迟和更高的效率意味着，与使用 OpenGL ES 3.1 的应用程序相比，使用 Vulkan 的应用程序性能更高。
更简单的驱动程序可减少应用处理器瓶颈。
你的应用程序执行资源管理，并可直接对 GPU 进行低级别控制。

多线程和多处理

Vulkan 支持跨多个应用处理器实施多线程。通过这个功能，你可以高效使用多个应用处理器，从而降低处理负载和能耗。
你的应用程序控制线程管理和同步。

命令缓冲区

你可以使用多线程以并行方式为命令缓冲区创建命令。此外，你还可以使用单独的提交线程将命令缓冲区放入命令队列。
你可以添加图形、计算和 DMA 命令缓冲区。
不同的图形、DMA 和计算队列让你可以灵活分配作业。
借助多线程命令创建，你的代码能够跨多个应用处理器内核运行，从而提高性能。使用多个以较低时钟频率运行的应用处理器（而不是一个时钟频率更高的处理器）也可降低能耗。

SPIR-V

Vulkan 使用 SPIR-V 中间语言，它支持你使用通用语言前端。
SPIR-V 是一种适用于并行计算和图形的多 API 中间语言。它包括流控制、图形和并行计算结构。
它可以在本机上表示 Vulkan 着色器和 OpenCL 内核源语言。
多个平台可以使用同一 SPIR-V 前端编译程序生成预编译着色器。
使用 SPIR-V 意味着 Vulkan 驱动程序中没有前端编译程序，所以驱动程序更简单，着色器编译速度也更快。
使用中间语言意味着你的应用程序无需附带着色器源代码。此外，它还提供了在将来使用不同着色语言的灵活性。

可加载的层

借助 Vulkan，你可以在开发过程中加载软件层以进行测试和调试。
你可以移除用于生产的其他软件层，所以你的上市产品不会产生测试开销。

多通道渲染

多通道渲染是一种在渲染通道中提前声明所有内容的技术。你可以为每个子通道指定不同的输出，并将它们链接在一起。
借助多通道，一个子通道中的像素可以访问同一像素位置上前一子通道的结果，从而使驱动程序可以进行优化。通过这种方式，可以将数据置入快速片上存储器中，从而节省带宽和能耗。此流程在基于图块的 GPU（如 Mali GPU）更高效。
多通道渲染的使用情形示例包括：

- 延迟渲染
- 软粒子
- 色调映射

下图显示了 Vulkan 的结构。

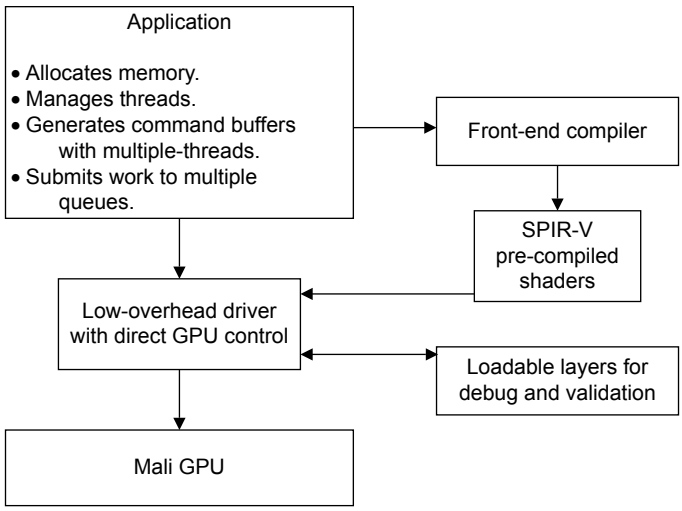

图 11-1 Vulkan 结构

11.2 关于 Unity 中的 Vulkan

在 Unity 中使用 Vulkan 的优势。

Vulkan 的许多优势都源自这样一个事实，即 Vulkan 的驱动程序比 OpenGL 和 OpenGL ES 更加简单。

但是，这也存在一些风险，即大部分管理工作都必须在应用程序中进行处理，而这种底层访问让应用程序的编写更加复杂。

Unity 会为你处理大部分这类工作，所以，你的应用程序将自动受益于 Vulkan 的增强性能。

在 Unity 中，你需要做的就是将 Vulkan 添加到 Unity 可用于构建应用程序的图形 API 列表中。

Vulkan 比 OpenGL ES 更高效，它可以降低能耗并延长电池寿命，因此，它通常是更好的选择。如有以下情况，请考虑使用 Vulkan：

- 你希望应用程序拥有理想性能。
- 你的应用程序受应用处理器的限制。
- 你怀疑 OpenGL 或 OpenGL ES 驱动程序引发了问题。

11.3 在 Unity 中启用 Vulkan

以下介绍在 Unity 5.6 中启用 Vulkan 的过程。

启用 Vulkan 的步骤：

1. 选择**文件** > **生成设置** …。
2. 在新窗口对话框中，按下**播放器设置** … 按钮。
3. 在**播放器设置**窗口中，找到**其他设置**部分。
4. 确保未选中**自动图形 API** 复选框。这让你能够手动选择 API。
5. 要将 Vulkan 添加为 API，按下 +，以将新 API 添加到列表，并添加 Vulkan。Vulkan 将添加为最后一个选项。
6. 要将 Vulkan 设置为主要 API，请选中 **Vulkan**，并将其移动到列表顶端。

现在，Vulkan 是播放器的主要 API。如下图所示。

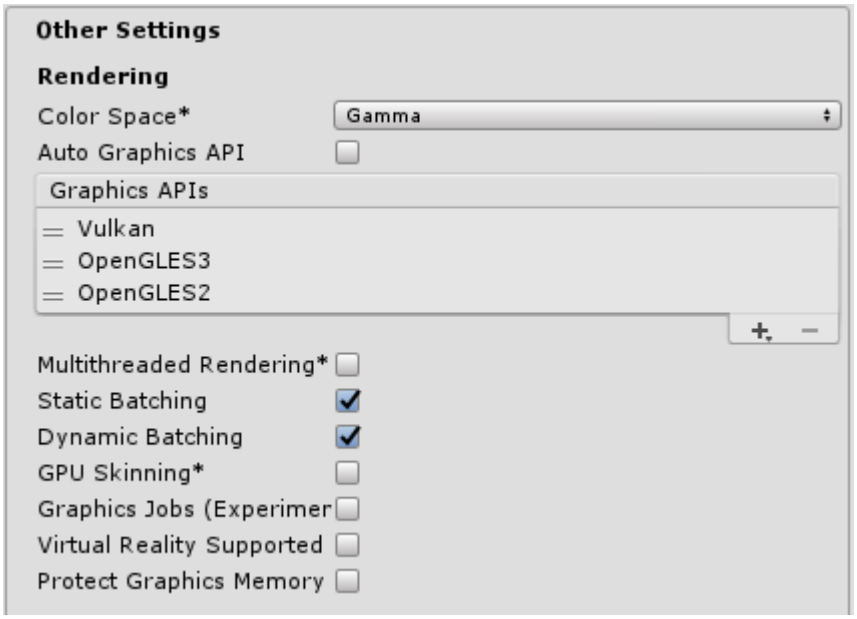

图 11-2 Unity API 选择

你可以移除其他两个 API，让 Vulkan 成为唯一的 API。

为此，请选择一个 API，并按下 – 将其移除。

11.4 Vulkan 案例研究

游戏开发公司 Infinite Dreams 如何使用 Vulkan 提高性能。

移动游戏开发公司 Infinite Dreams 开发了流行移动游戏 Sky Force Reloaded。Sky Force Reloaded 采用丰富的图形环境，设计有激烈的动作，对应用处理器和 GPU 的要求都很高。Sky Force Reloaded 最初通过采用 OpenGL ES 的 Unity 构建。

图 11-3 Sky Force Reloaded

Sky Force Reloaded 优化

游戏开发在一开始初步创造出所需的丰富图形。下一阶段就是优化。

该游戏在图形方面非常复杂，屏幕上会同时显示很多内容，因此团队认为最应该进行优化的方面就是填充率。填充率方面的问题通常可以通过降低帧缓冲区的分辨率来解决。

但是，在他们尝试使用低分辨率渲染游戏后，仍然会出现性能问题。即使在高端设备上，也不可能始终将*每秒帧数* (FPS) 维持在 60。团队发现该游戏会进行大量绘制调用。有时，每帧的绘制调用多达 1000 个。

每个绘制调用都会花费计算开销，因此进行大量绘制调用需要花费巨大的计算开销。这将导致 OpenGL ES 驱动程序使应用处理器在长时间内一直忙于为 GPU 准备数据。因此，即使采用高端移动设备，也会面临运行速度减慢的情况。

为了进行优化，团队需要最大限度地减少绘制调用的数量，或对其进行修改，以便游戏引擎对其进行批处理。但是，这样迟早是会影响游戏质量的。

测试 Vulkan

该团队听说了 Vulkan，他们决定看看 Vulkan 能否提高游戏性能。

在 Unity 中，Vulkan 只是一个渲染 API，而 Unity 负责处理所有繁重的工作。该团队只需在 Unity 中激活 Vulkan，即可使用后者。

为了测试 OpenGL ES 与 Vulkan 之间的差异，该团队分析了此款游戏。他们发现，在游戏中最慢的部分之一，OpenGL 无法实现 60 FPS。为了测量 OpenGL ES 与 Vulkan 之间的性能差异，开发团队根据这部分游戏内容创建了一个合成基准。

该基准让 API 渲染重复场景，为两种 API 进行一致的测试。

在下图中，你会发现 Vulkan 能在大部分时间保持 60 FPS，而 OpenGL ES 很难做到这一点。与 OpenGL ES 相比，Vulkan 的总体性能提升了 15%。考虑到 Vulkan 只是一个图形 API，该团队认为这个结果非常理想。

下图显示了对比 OpenGL ES 和 Vulkan 的基准测试结果：

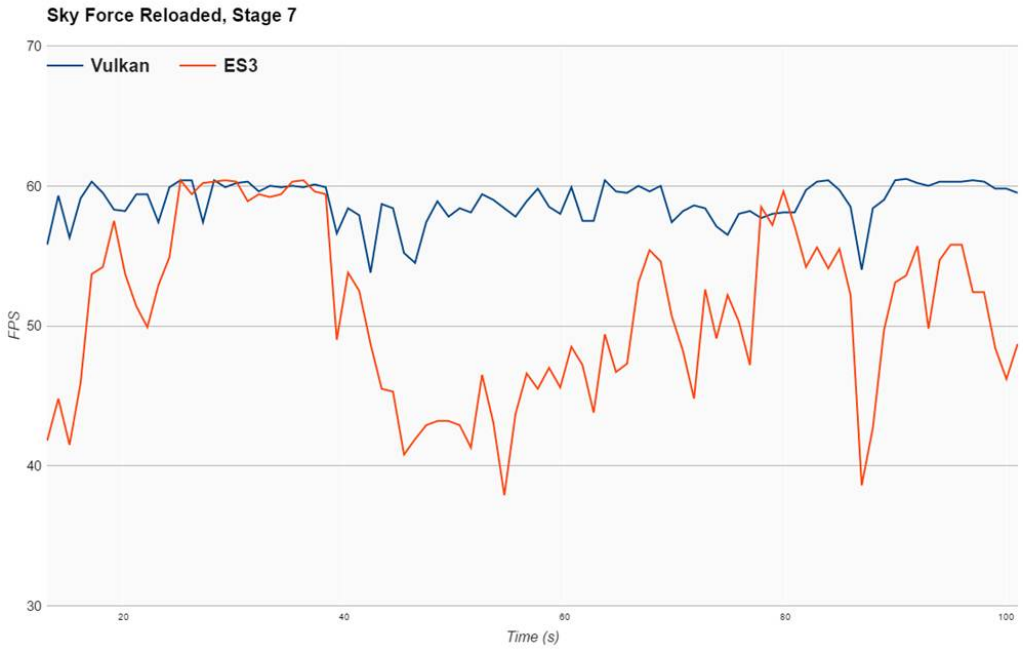

图 11-4 对比 OpenGL ES 和 Vulkan 的 Sky Force Reloaded 基准测试

测试 Vulkan 的其他性能

认识到 Vulkan 能够在大部分时间保持 60 FPS 的运行速度后，该团队开始好奇 Vulkan 可以将性能提高多少。

开发人员开始将额外的一些对象添加到基准平面，直到 Vulkan 也无法实现 60 FPS，这导致 OpenGL ES 与 Vulkan 之间的差距进一步扩大了。平均而言，Vulkan 的速度比 OpenGL ES 快 32%。

下图显示了对比 OpenGL ES 和 Vulkan 的基准测试结果，此时的场景中添加了一些对象：

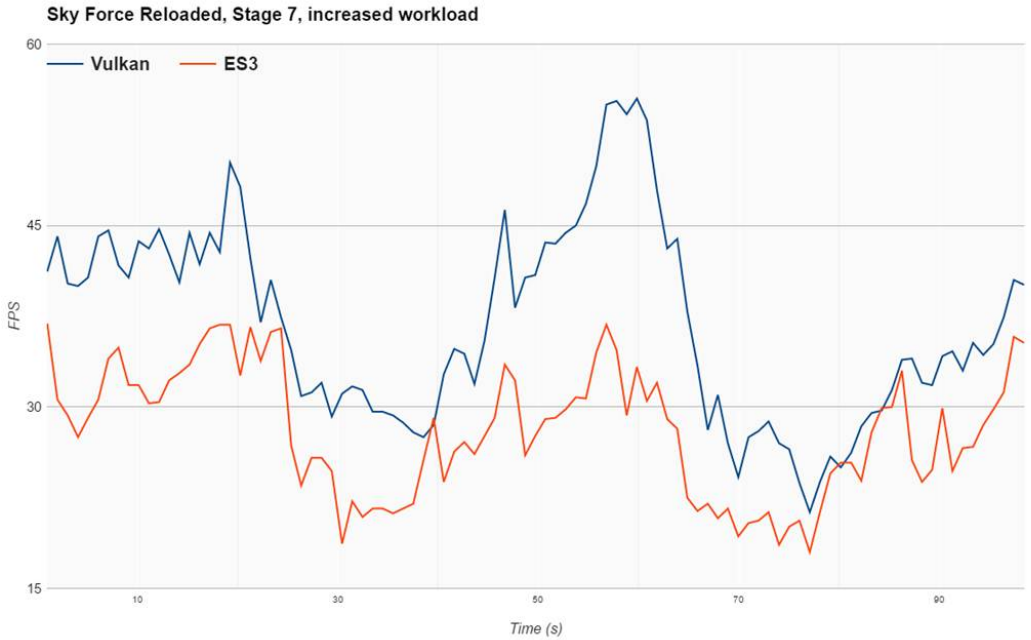

图 11-5 添加对象后的 Sky Force Reloaded 基准测试

确实有一些额外的性能可以加以利用，因此该团队开始考虑如何在游戏中利用这些额外的算力。他们添加了更多图形、粒子、对象和动画。该团队发现他们能让游戏画面看起来更丰富，同时仍能保持 60 FPS。不用大费周折就能获得如此好的效率，全然得益于有 Vulkan。

Vulkan 视频对比

有关视频对比，请访问：*https://www.youtube.com/watch?v=VCSkp-QZ37M*。

该视频并排显示对比了 Vulkan 版本与 OpenGL ES 版本的性能。左侧的游戏使用的是 OpenGL ES。右侧的游戏使用的是 Vulkan。两个版本的运行速度都是每秒 60 帧。

在同样的帧速率下，Vulkan 能渲染的星星数量比 OpenGL ES 版本多六倍，子弹数量多一倍。使用 OpenGL ES 时，渲染相同数量的对象会导致在复杂的场景下帧速率大幅下降。在同样的帧速率下，与 OpenGL ES 相比，Vulkan 支持你向屏幕添加更多几何图形。

下图显示了对比视频中的帧：

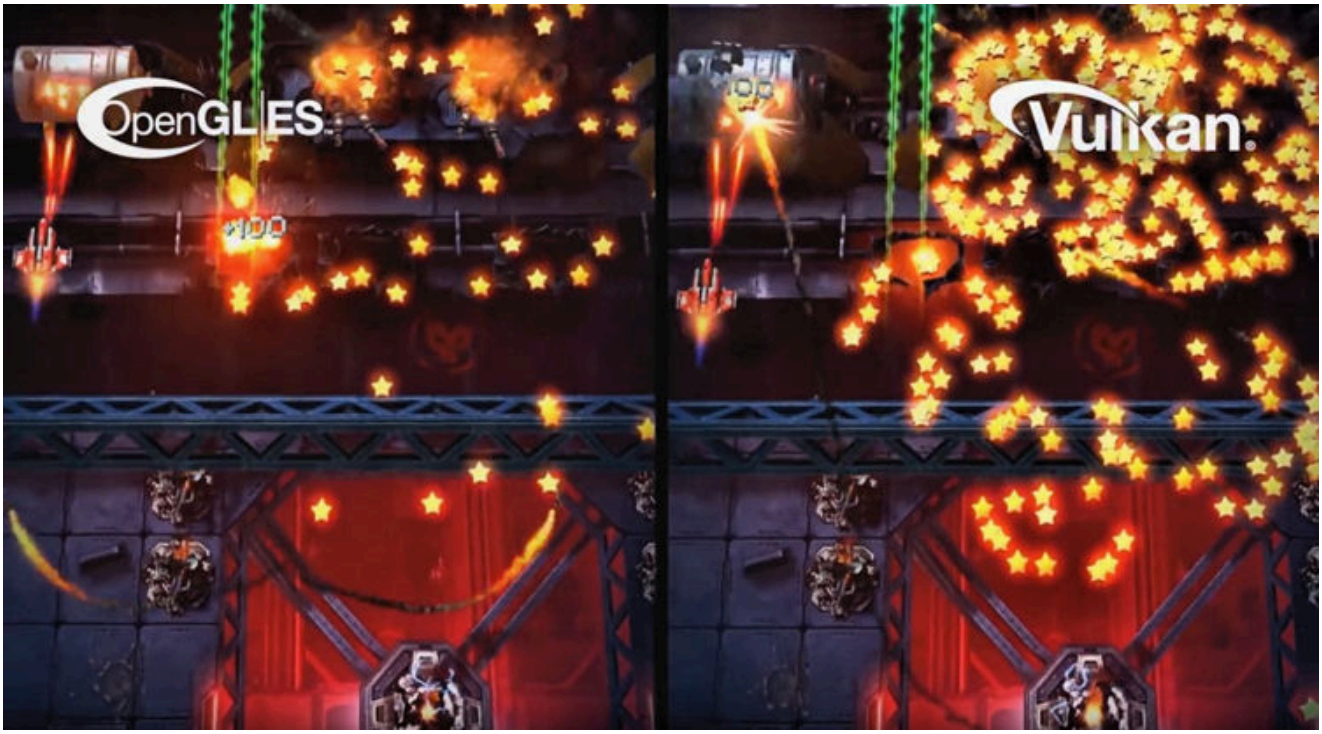

图 11-6 对比 OpenGL ES 和 Vulkan 的 Sky Force Reloaded 帧

Vulkan 能耗

Sky Force Reloaded 是一款应用处理器和 GPU 密集型游戏。有些玩家甚至抱怨过游戏耗电速度太快。该团队不希望游戏机级的质量受到丝毫影响，所以他们反复查验，想看看 Vulkan 能否提供帮助。他们使用 Vulkan 进行了测试，以查看它能否降低能耗，从而延长电池寿命。在 Unity 中启用 Vulkan 能够降低 10-12% 的能耗，从而延长 10-12% 分钟的游戏时间。

有关视频对比，请访问：*https://www.youtube.com/watch?v=WI7nXq8oozw*

第 12 章
Arm Mobile Studio

本章将介绍图形分析器和 Streamline 工具。

它包含以下部分:

12.1 Arm Mobile Studio 的优势

Arm Mobile Studio 是一个软件套件，让你能够直观地查看安卓游戏或应用程序的性能数据。先用 Arm Streamline 识别瓶颈，然后使用图形分析器的图形应用程序跟踪功能确定渲染缺陷发生的确切位置。

无需 root 即可轻松设置

Arm Mobile Studio 自带的所有工具都设计为可在没有 root 过的安卓设备上工作。这样一来，你无需修改设备即可分析和调试游戏。

轻松检测瓶颈

使用 Arm Streamline 直观地查看系统中的性能数据。易于使用的模板可帮助你为目标设备选择最合适的一组计数器，从而获得优化应用程序所需的信息，并过滤掉剩余的数据。移动设备（包括 64 位设备）优化速度更快，使用直观易懂的图表快速确定你的性能瓶颈是否与 CPU 或 GPU 有关。如果性能问题与 CPU 有关，你可以从线程和函数调用着手深入，直至逐行分析源代码，找出代码中最大的问题区域。

识别图形问题

使用图形分析器查看应用程序中的所有图形 API 的调用情况。轻松跟踪 OpenGL ES 和 Vulkan API 调用，并逐帧了解其对应用程序性能产生的效果。该工具捕获整个应用程序的状态变化，因此你可以轻松地识别渲染缺陷，并跟踪导致问题发生的 API 函数调用。通过逐个绘制调用来绘制场景，以查看应用程序中的帧是如何组成的。帮助你轻松查明应用程序中的任何图形缺陷，并发现任何低效或冗余的绘制调用。

获取 Arm Mobile Studio

你可从如下链接下载 Arm Mobile Studio 并找到 Unity 学习资源：*https://developer.arm.com/unity/*。更多有关 Arm Mobile Studio 的信息，请访问 *https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio*。

12.2 关于图形分析器

图形分析器是一种适用于 OpenGL ES、Vulkan 和 OpenCL 的 API 跟踪和调试工具。它可以帮助你识别应用程序可能存在的问题。

图形分析器为你提供了一组工具，让你能够监控并分析应用程序的运作。

它可以显示不同类型的信息和统计数据，从而告诉你有关应用程序的数据访问和处理情况。你可以借助此信息调试应用程序或识别性能瓶颈。

下图即为图形分析器：

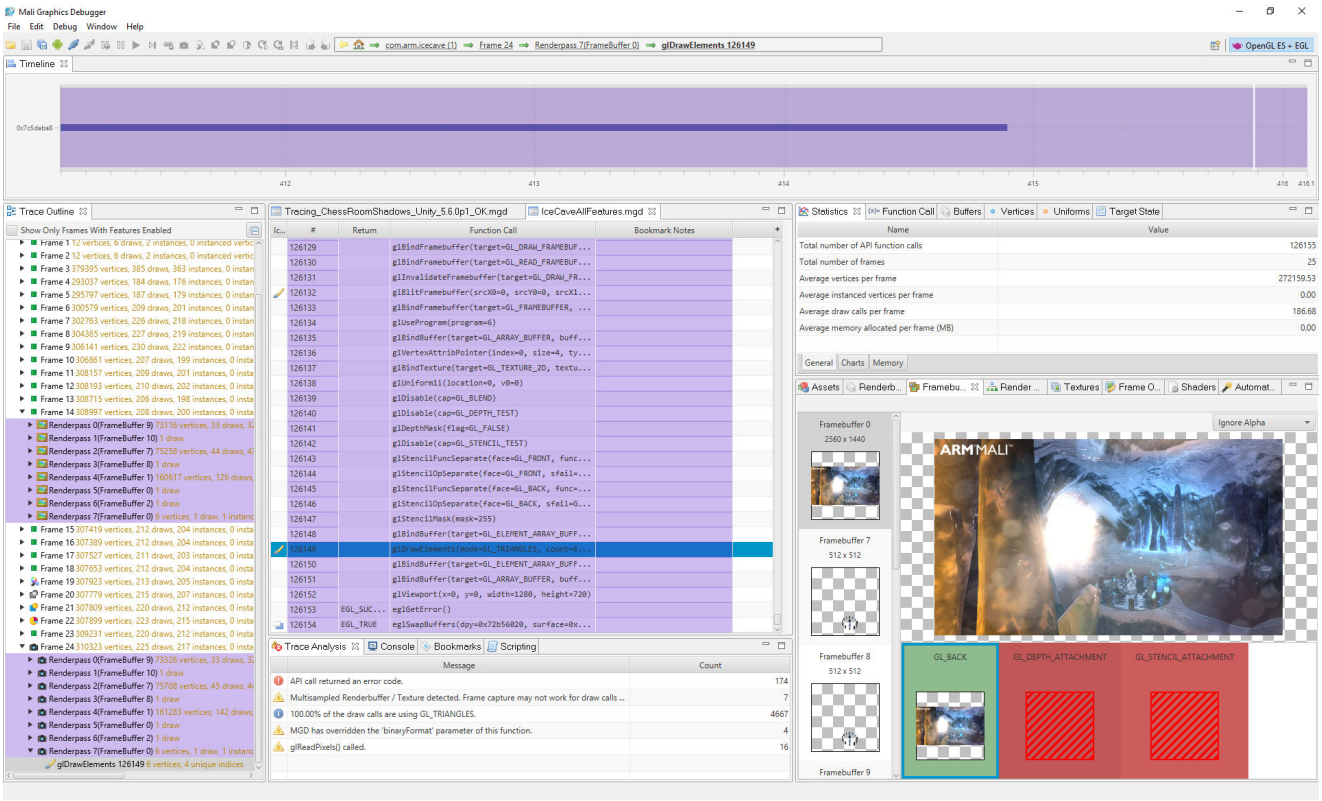

图 12-1 图形分析器

有关更多信息，请参阅 *https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/components/graphics-analyzer*。

本部分包含以下子部分：

- *12.2.1 图形分析器功能* on page 12-251.
- *12.2.2 启用图形分析器* on page 12-253.
- *12.2.3 在使用 Vulkan 的环境中启用图形分析器* on page 12-256.

12.2.1 图形分析器功能

图形分析器提供多个功能可帮助你识别应用程序可能存在的问题。

借助图形分析器，你能够跟踪 API 调用，并了解其在应用程序中的逐帧效果。此外，它还可以突出显示常见的 API 问题，并提供性能改进建议。

图形分析器具有以下功能和优势：

跟踪 API 调用

逐个绘制调用步进有助于你发现与绘制调用相关的问题、多余的绘制调用及其他优化机会。

你可以通过参数、时间、进程 ID 和线程 ID 跟踪 OpenGL ES、Vulkan、EGL 和 OpenCL API 调用。

它从一开始就可以进行全方位跟踪，且会自动检查错误。它具有搜索功能，还包含多进程支持。

自动跟踪视图使你能够在遇到特定帧时自动运行一系列标准图形分析器命令。

大纲视图

大纲视图支持你在帧级别进行核查，以查找哪些绘制调用具有更大的几何影响。它可以让你快速访问重要的图形事件，如帧、渲染目标和绘制调用。

目标状态调试

目标状态调试可以全方位展示跟踪过程中任何一点的图形状态和状态更改。此外，你还可以使用它发现状态更改的时间和方式。

每次在跟踪中选定新 API 调用时，状态就会更新。这有助于调试问题并了解出现性能问题的原因。

你可以在大纲视图和跟踪视图之间顺畅地跳转，有助于你进行调查。

跟踪统计和分析

你可以分析跟踪，以查找常见问题（如性能警告）。

图形分析器提供有关跟踪的每帧以及每绘制调用高级别统计数据。

此外，图形分析器还会在跟踪视图中突出显示有问题的 API 调用，以此显示 API 误用。

统一和顶点属性数据视图

在你选择绘制调用后，图形分析器会显示所有相关的统一属性数据和顶点属性数据。通过顶点属性和统一视图，你可以查看顶点和索引缓冲区数据。你可以对它们进行分析，以查看其在应用程序中的使用情况。

统一视图显示统一值，如采样器、矩阵和数组。

顶点属性视图显示顶点属性，如顶点名称和顶点位置。

缓冲区和纹理访问

纹理视图让你可以查看应用程序中的纹理使用情况，帮助你发现压缩纹理或更改格式的机会。

只要客户端和服务器端缓冲区发生更改，系统就会检测到。这让你能够了解各个 API 调用是如何影响缓冲区的。

上传的所有纹理都以原始分辨率捕捉，让你可以检查纹理的尺寸、格式和类型。此外，你还可以访问所有大型资源（包括数据缓冲区和纹理），以进行调试。

着色器报告和统计数据

着色器统计数据和循环计数可帮助你了解哪些顶点和片段着色器的计算消耗最大。

系统将测量应用程序使用的所有着色器。对于各绘制调用，图形分析器会计算着色器执行的次数，并计算总体统计数据。

各着色器也会通过 Mali Offline Compiler 进行编译。这样可以通过静态方式分析着色器，以显示各 GPU 管线的指令数以及工作寄存器和统一寄存器的数量。

帧捕捉和分析

图形分析器可以捕捉帧，以便你分析各绘制调用的效果。

每次进行绘制调用后，系统都会捕捉各帧缓冲区的原始分辨率快照。快照捕捉在目标设备上进行，因此你可以调查与目标相关的故障或精度问题。

此外，你还可以导出捕捉图像，进行单独分析。

替代绘制模式

图形分析器包含多种可以帮助你分析帧的专用绘制模式。你可以将它们用于实时渲染和帧捕捉：

原生模式
在原生模式下，帧会通过原始着色器进行渲染。

过度绘制模式
过度绘制模式会突出显示出现过度绘制的位置。你可以通过查看过度绘制收集信息和柱状图来分析过度绘制像素对每个帧缓冲区的影响。

着色器图模式
在着色器图模式下，原生着色器会替换为不同的纯色。这可以向你显示在一个帧中使用不同着色器的位置。

片段计数模式
在片段计数模式下，系统会计算为每个帧处理的所有片段。这可以向你显示一个帧的不同部分使用的片段数量。

12.2.2 启用图形分析器

介绍如何在 Unity 中启用图形分析器以分析 OpenGL ES 应用程序。

图形分析器库加载已集成到 Unity 中。你可以通过将图形分析器和 Unity 配合使用来分析 OpenGL ES 应用程序。

要启用图形分析器，你需要满足以下条件：

- 拥有 Unity 5.6 或更高版本。
- 你的 Android 设备必须采用 Android 4.2 或更高版本。
- 拥有最新版 Arm Mobile Studio，可从以下位置获取：*https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio*。
- 拥有 Android *软件开发工具包* (SDK)。
- 你的 `PATH` 必须包括 adb 二进制文件的路径。
- 你必须与目标设备建立有效的 ADB 连接。ADB 必须能返回设备的 ID，且没有权限错误，同时你必须能执行应用程序。有关更多信息，请参阅你的 Android 设备文档。
- 你的目标设备必须允许在端口 5002 上进行 TCP/IP 通信。

要为 Unity 应用程序添加图形分析器支持，请遵循以下步骤：

1. 找到图形分析器安装文件夹。找到包含图形分析器 Android 应用程序 `AGA.apk` 的子文件夹 `target\android\arm\`。

   要将 AGA apk 安装到你需要分析 Unity 游戏或应用程序的 Android 设备上，请键入此命令：

   ```
   adb install -r AGA.apk
   ```

2. 在图形分析器安装文件夹中的 `target\android\arm\unrooted\armeabi-v7a\` 下找到 `libMGD.so` 库。

   在 Unity 项目中，创建子文件夹 `Assets\Plugins\Android\`。将 `libMGD.so` 库放置在此处。

图 12-2 将 `libMGD.so` 复制到 Unity 项目

3. 在 Unity 生成设置对话框窗口中，选中**开发生成**选项。这样做会指示 Unity 将 `libMGD.so` 库打包到 apk 中，并在主机设备上运行时进行加载。

   生成 apk。

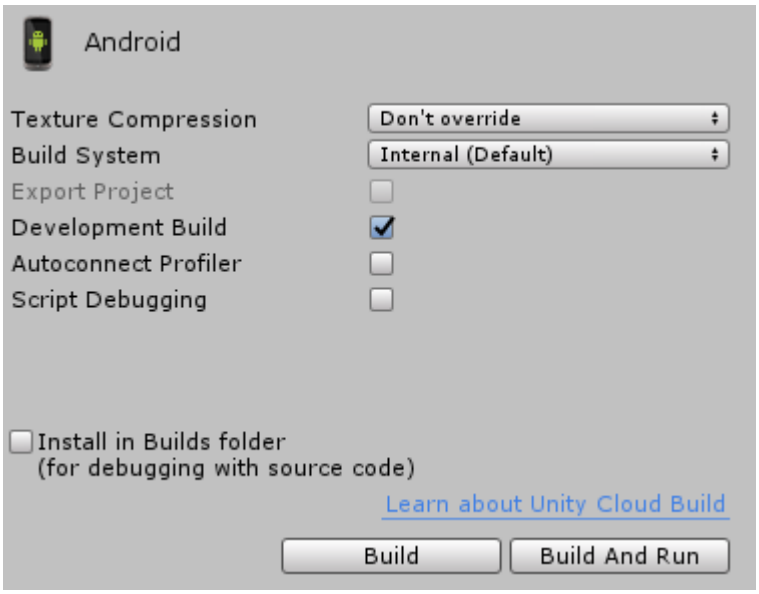

图 12-3 选中“开发生成”选项

4. 要在 Android 设备上安装支持图形分析器的 Unity 应用程序，请键入：

   ```
   adb install -r YourApplication.apk
   ```

5. 要使图形分析器通过 USB 连接到守护程序，请在主机上运行以下命令：

   ```
   adb forward tcp:5002 tcp:5002
   ```

6. 在主机上启动图形分析器 Android 应用程序，并通过滑动按钮启用*图形分析器守护程序* (GAD)。链接到 `libMGD.so` 库的 Unity 应用程序将列出。

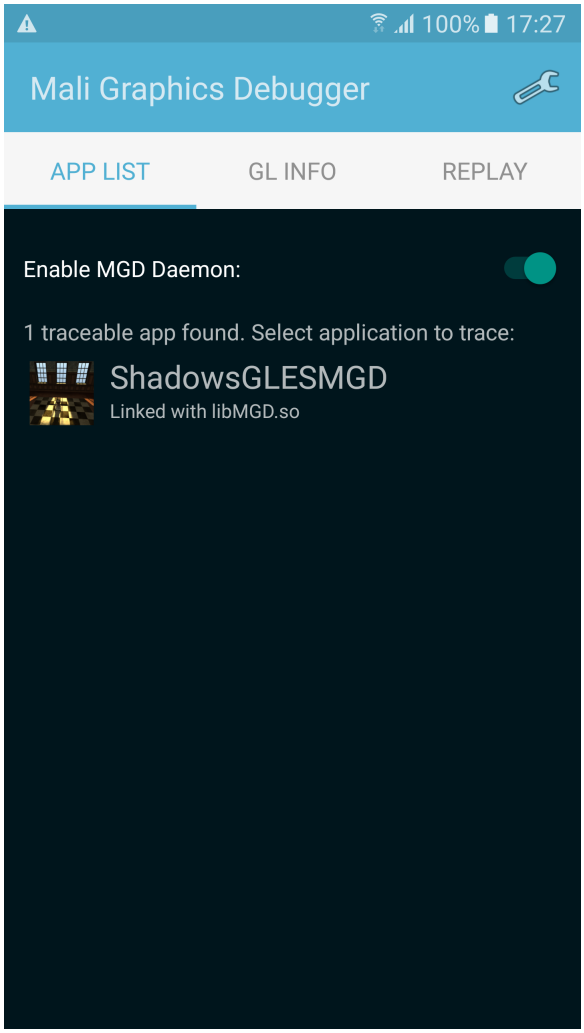

图 12-4 启用图形分析器守护程序

7. 在桌面设备上启动图形分析器。单击  图标，以连接到目标并开始跟踪。

   图形分析器会等待跟踪输入。
8. 设备上的图形分析器 Android 应用程序会列出你需要点击的应用程序。AGA 拦截器将应用程序发出的所有 OpenGL ES 和 EGL 调用发送到图形分析器主机应用程序。

下图显示了图形分析器跟踪采用 OpenGL ES 的 Unity 的过程：

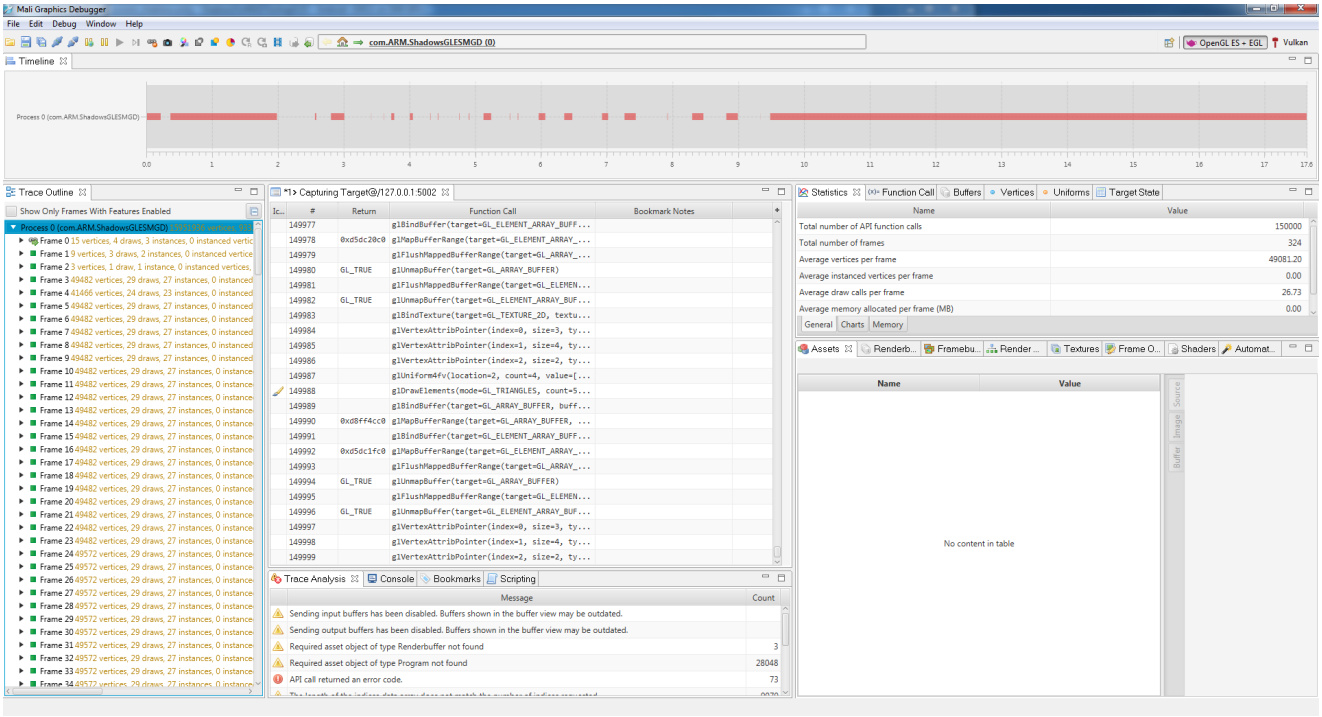

图 12-5 图形分析器跟踪采用 OpenGL ES 的 Unity

12.2.3 在使用 Vulkan 的环境中启用图形分析器

介绍如何在采用 Vulkan 的 Unity 中启用图形分析器。

图形分析器支持对使用高性能 Vulkan API 开发的图形应用程序进行调试。

要在 Vulkan 中调试应用程序，请使用更多能够在运行时插入系统的层。图形分析器包括可以用作 Vulkan 层的拦截器库。

要将层添加到基于 Vulkan 的 Unity Android 应用程序，你必须将层库复制到项目文件夹 `Assets/Plugins/Android/libs/armeabi-v7a`。

要构建支持图形分析器的 Vulkan 应用程序，你还必须将图形分析器库复制到同一文件夹中。

要为采用 Vulkan 的 Unity 启用图形分析器，你需要满足以下条件：

- 拥有 Unity 5.6 或更高版本。
- 你的 Android 设备必须采用 Android 4.2 或更高版本。
- 拥有最新版图形分析器，可从以下位置获取：*https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio*。
- 拥有 Android *软件开发工具包* (SDK)。
- 你的 `PATH` 必须包括 adb 二进制文件的路径。
- 你必须与目标设备建立有效的 ADB 连接。ADB 必须能返回设备的 ID，且没有权限错误，同时你必须能执行应用程序。有关更多信息，请参阅你的 Android 设备文档。
- 你的目标设备必须允许在端口 5002 上进行 TCP/IP 通信。

要为使用 Vulkan 开发的 Unity 应用程序添加图形分析器支持功能，请遵循以下步骤：

1. 找到图形分析器安装文件夹。

   找到子文件夹 `target\android\arm\`。

   此文件夹包含图形分析器 Android 应用程序 `AGA.apk`。

   要将此 apk 安装到你需要分析 Unity 游戏或应用程序的 Android 设备上，请键入此命令：

```
adb install -r AGA.apk
```

2. 在图形分析器安装文件夹中的 `target\android\arm\rooted\armeabi-v7a\` 文件夹下找到图形分析器库 `libMGD.so`。

   复制此库，并将其重命名为 `libVkLayerMGD32.so`。

   将其复制到 Unity 项目文件夹 `Assets\Plugins\Android\libs\armeabi-v7a` 中。

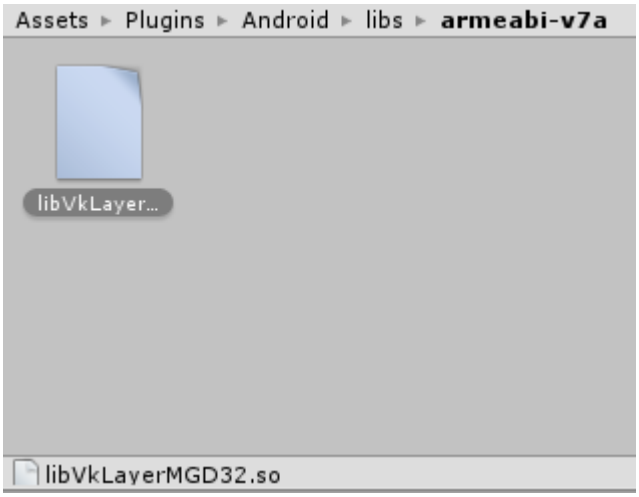

图 12-6 将 `libVkLayerMGD32.so` 复制到 Unity 项目

3. 在 Unity 生成设置对话框窗口中，选中**开发生成**选项。这会指示 Unity 在运行时启用 Vulkan 层机制。`libVkLayerMGD32.so` 库会被打包成与“开发生成”选项状态无关的 apk。

   生成 apk。
4. 通过运行以下命令安装应用程序：

   ```
   adb install -r YourApplication.apk
   ```
5. 要使图形分析器通过 USB 连接到守护程序，请在主机上运行以下命令：

   ```
   adb forward tcp:5002 tcp:5002
   ```
6. 在主机上启动图形分析器 Android 应用程序，并通过滑动按钮启用图形分析器守护程序。
7. 要启用图形分析器 Vulkan 层，请键入以下命令：

   ```
   adb shell

   setprop debug.vulkan.layers VK_LAYER_ARM_MGD
   ```

   此命令将指示 Vulkan 加载程序加载名为 `VK_LAYER_ARM_MGD` 的层。

8. 在桌面设备上启动图形分析器。单击  图标，以连接到目标并开始跟踪。按下图形分析器窗口右上角的透视图标，并选择“Vulkan”。

   图形分析器会等待跟踪输入。
9. 点击你设备上的应用程序。图形分析器 Vulkan 层将 Vulkan 调用发送给图形分析器，图形分析器会显示这些调用。

下图显示了图形分析器跟踪采用 Vulkan 的 Unity 的过程：

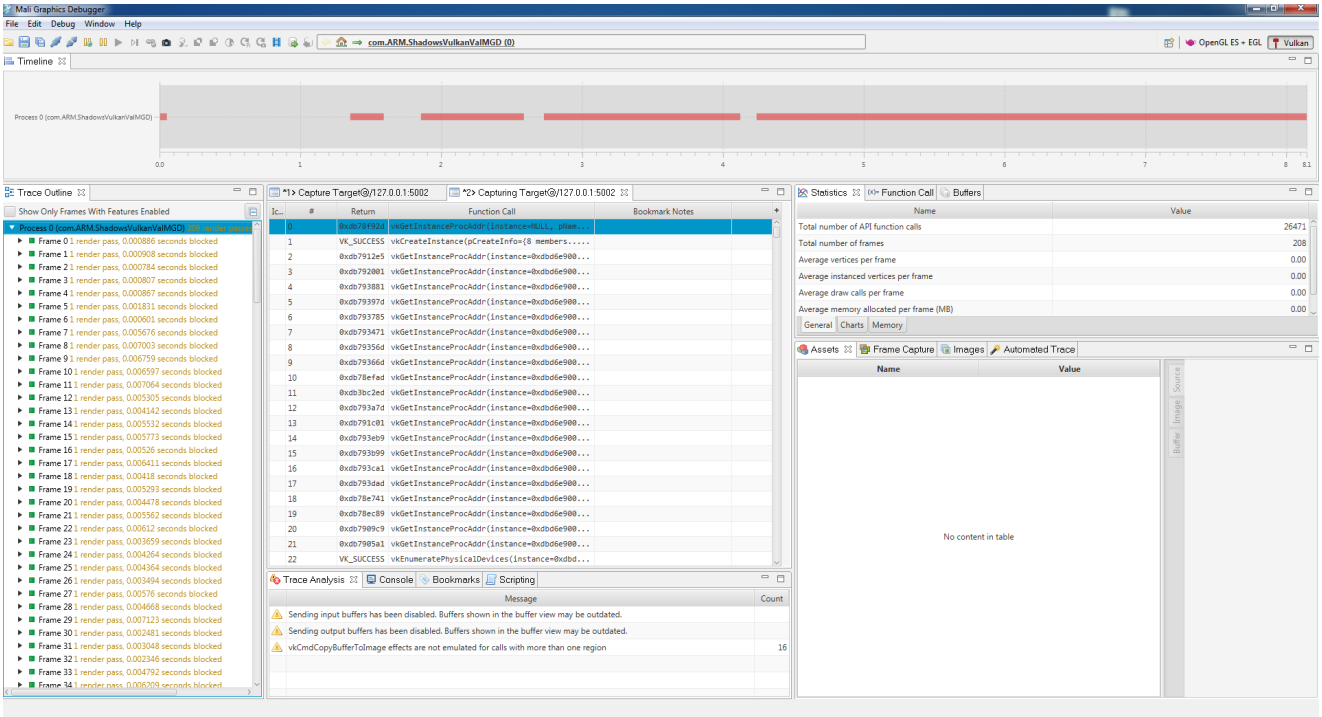

图 12-7 图形分析器跟踪采用 Vulkan 的 Unity

12.3 关于 Streamline

Streamline 是一个多用途性能分析工具，是 *Arm Mobile Studio* (AMS) 优化和调试软件工具套件的一部分。

Streamline 从安卓设备的多个来源收集样本和基于事件的性能数据。时间轴等不同视图显示数据整合的结果。时间轴视图上半部分显示收集的系统性能计数器数据，下半部分显示同一时间轴上不同类型的信息。

以下屏幕快照显示了时间轴视图，视图下半部分显示选中的热图：

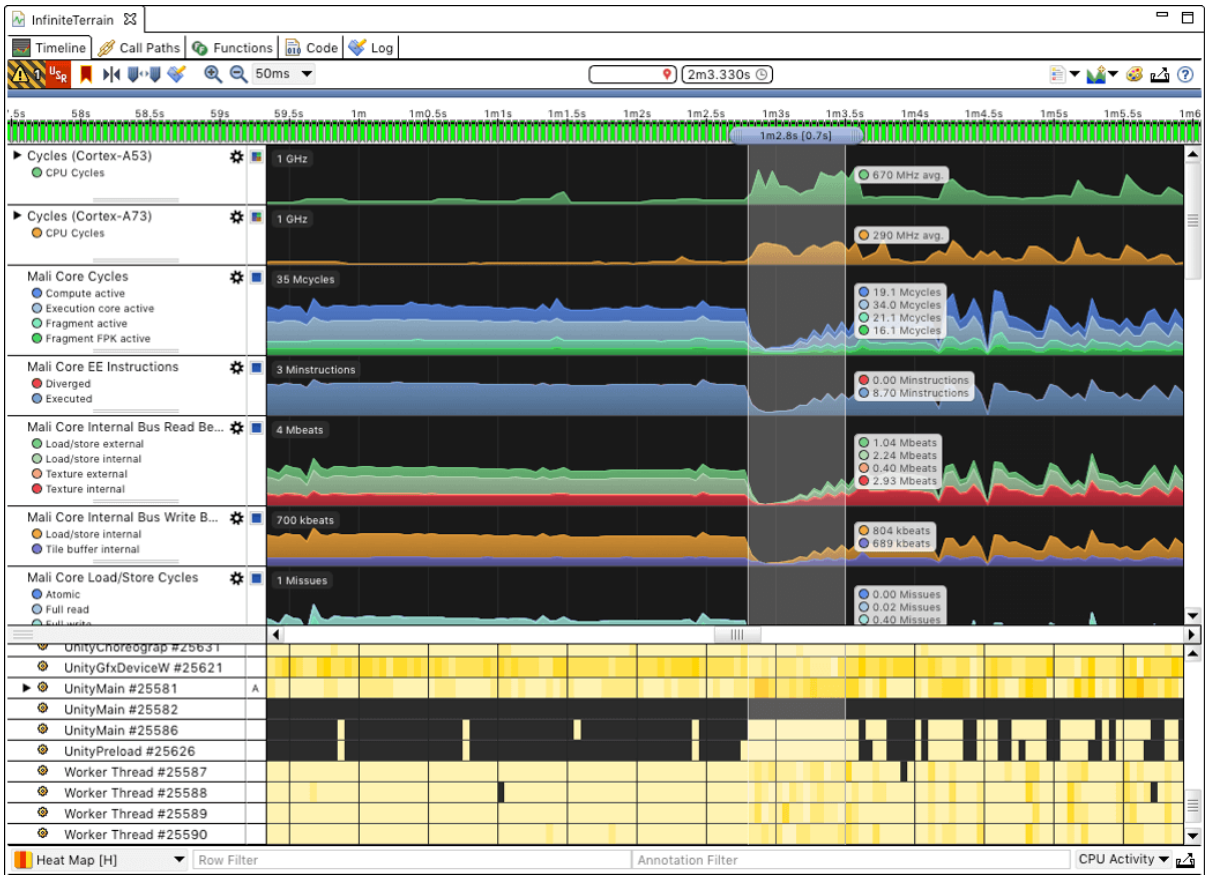

图 12-8 Streamline 时间轴热图视图

本部分包含以下子部分：

12.3.1 分析示例场景

以下分析程序生成的地形，即随摄像机旋转形成的动态地形。

场景填满时，离摄像机过远的图块被删除。删除图块可以使场景的复杂性随着时间的推移大致保持不变。摄像机移动缓慢时，新地形的生成速度也变慢。摄像机加速移动时，地形生成速度须随之加快。

下方屏幕快照中，每个图块边缘都以较暗的颜色渲染，显示每个图块的大小：

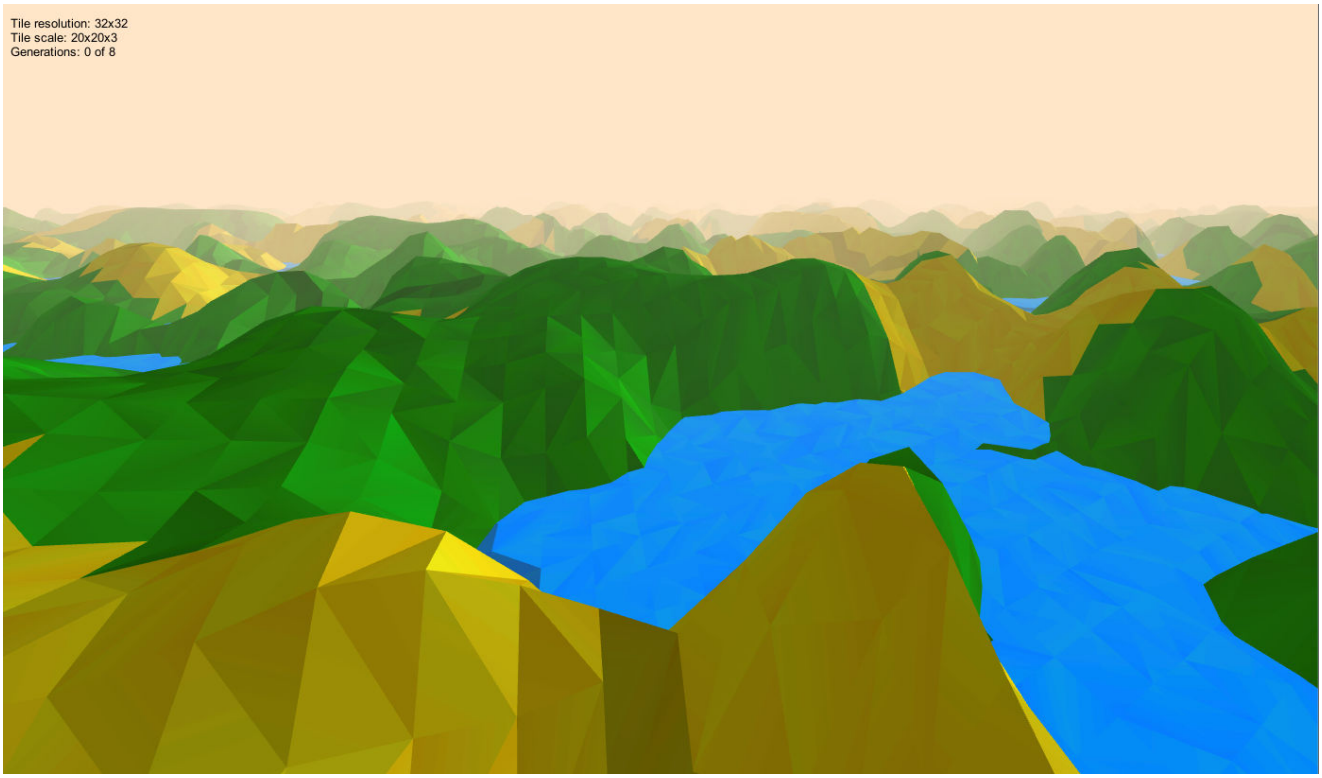

图 12-9 渲染的图块

动态生成地形图块的计算成本很高。为了更大限度地降低计算成本，采用 *Unity Job Scheduler* (UJS) 用于分派不影响 Unity 主线程完成的后台线程。

这样可以确保用户体验到稳定的帧率，避免每次生成新地形时都出现卡顿。

示例演示中运行四个不同的场景，这四个场景视觉效果相同，但是生成的图块是不同的。

如下图所示，地形由许多固定大小的地形块组成。每个地形块由几个固定分辨率的网格组成，一个用于绿色地形，一个用于黄色地形，一个用于水域。渲染距离控制玩家周围生成的图块数量。

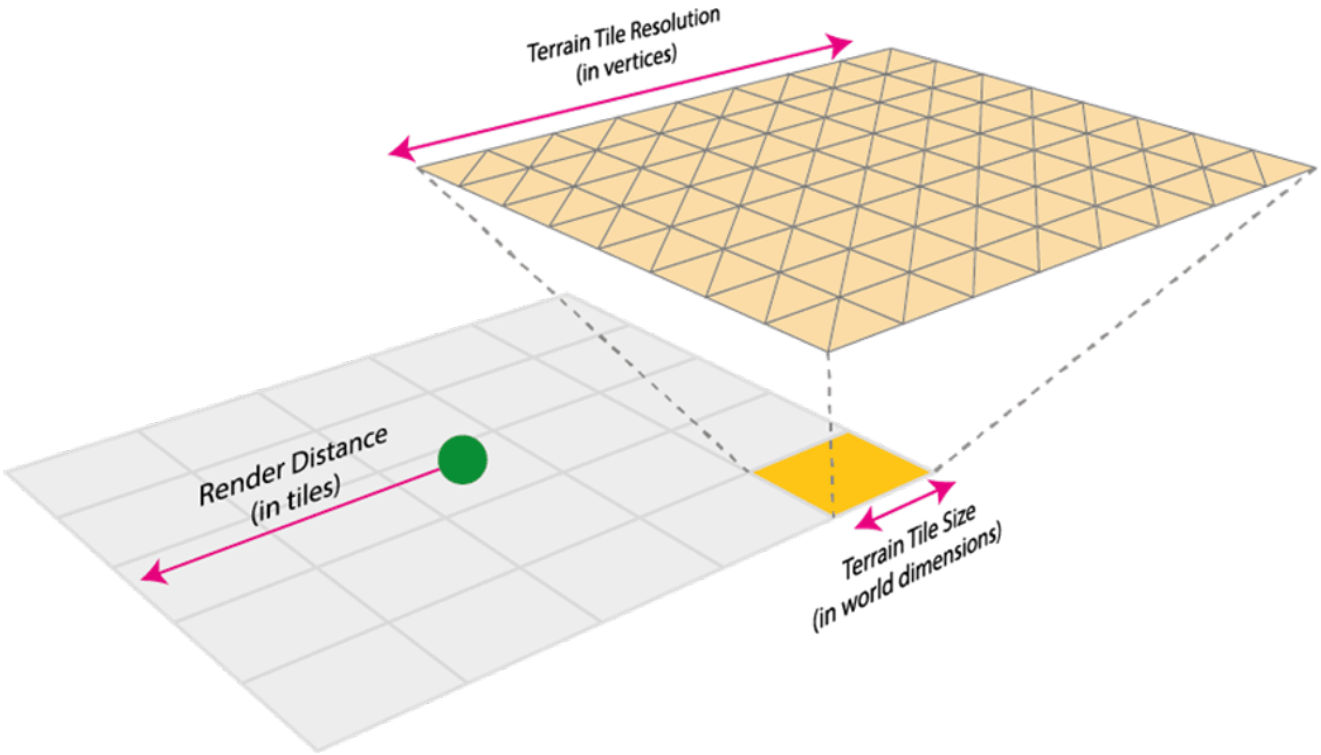

图 12-10 地形图块分辨率

四个场景配置如下：

表 12-1 示例场景的配置设置

场景	渲染距离	地形图块大小	地形分辨率	同时生成的地形数量
1	3	20x20	32x32	8
2	3	20x20	32x32	1
3	6	10x10	16x16	8
4	6	10x10	16x16	1

在 2017 版华为 P10 手机上进行分析。华为 P10 携带海思麒麟 960 芯片，包含：
- 4 个高性能 Arm Cortex -A73 CPU 内核。
- 4 个高效 Arm Cortex -A53 CPU 内核。
- 1 个 Arm Mali -G71 MP8 GPU。

12.3.2 Unity 中的分析

下文详细介绍了 Unity 中的分析器。但不包括电池使用等关键数据。因此，你可以使用 Unity 中的分析器将信息传递到 Streamline 做进一步分析。

Unity 内的分析器显示作业的调度时间。然而，分析器不显示可用 CPU 或 GPU 资源的任何细节，也不显示资源的使用情况。

目标应用程序可能达到 60 *帧每秒* (FPS)，但监控其他区域也很重要。包括 CPU 核心利用率和 GPU 性能信息。Streamline 可用于监控关键区域。

下图显示了 Unity 内置分析器中提供的信息的截屏：

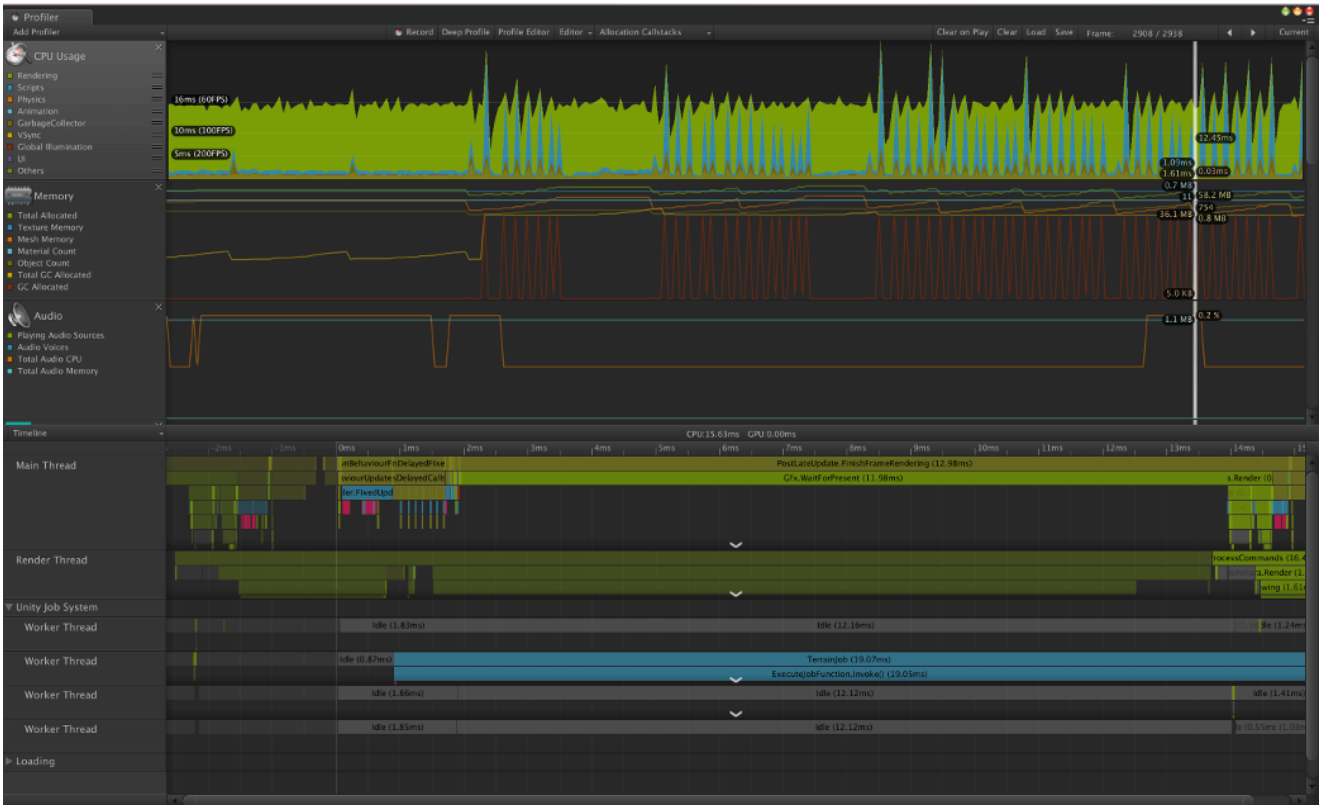

图 12-11 Unity 分析器

12.3.3 使用 Streamline 分析 Unity 的能力

Streamline 提供多个关键分析功能，帮助你在线程级别上优化应用程序性能。

此处示例场景已经修改为在 Streamline 中使用了三种不同类型的注释。这些注释有：

1. **标记** - 标记是一个带有显示在**时间轴**视图顶部的标签的单个时间点。
2. **通道** - 通道在每个线程旁提供单独的一行信息。
3. ***自定义活动贴图*** (CAM) - CAM 可以显示存在复杂依赖关系的跨线程活动。每个 CAM 都以独一无二的视图显示在*用户界面* (UI) 的下半部分。

以下屏幕快照显示了不规则帧率在**时间轴**视图上的效果：

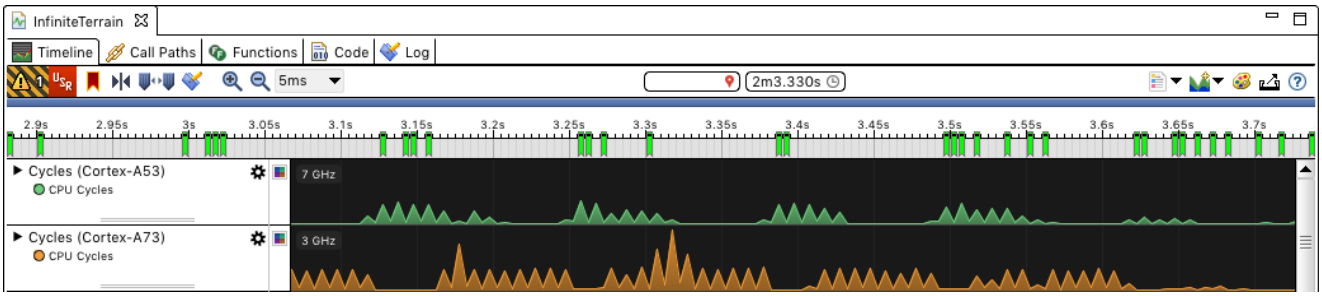

图 12-12 帧率

上图显示，帧率并没有预期的那样平滑，并且每个 CPU 内核存在大量的突发活动。帧率下降，然后上升，中间偶尔会有停顿。然而，这种结果与地形生成的预期是一致的。这是一个值得小心留意的问题。

Streamline 中的时间轴视图分为两个部分。顶部视图显示指标图表，下半部分显示各种不同的简况。热图等简况显示了任务在整个系统中的分布情况。你可对顶部时间轴进行过滤，从而仅显示已经归属于特定进程或线程的任务。

首先选择 **Unity 主线程**，然后选择所有**工作线程**，你就可以查看热图。你可以看到 Unity 主线程和作业调度程序中的线程是如何划分 CPU 活动的。

下图为 **Unity 主线程**的 CPU 简况的屏幕快照。 Cortex -A73 CPU 上存在大量突发活动：

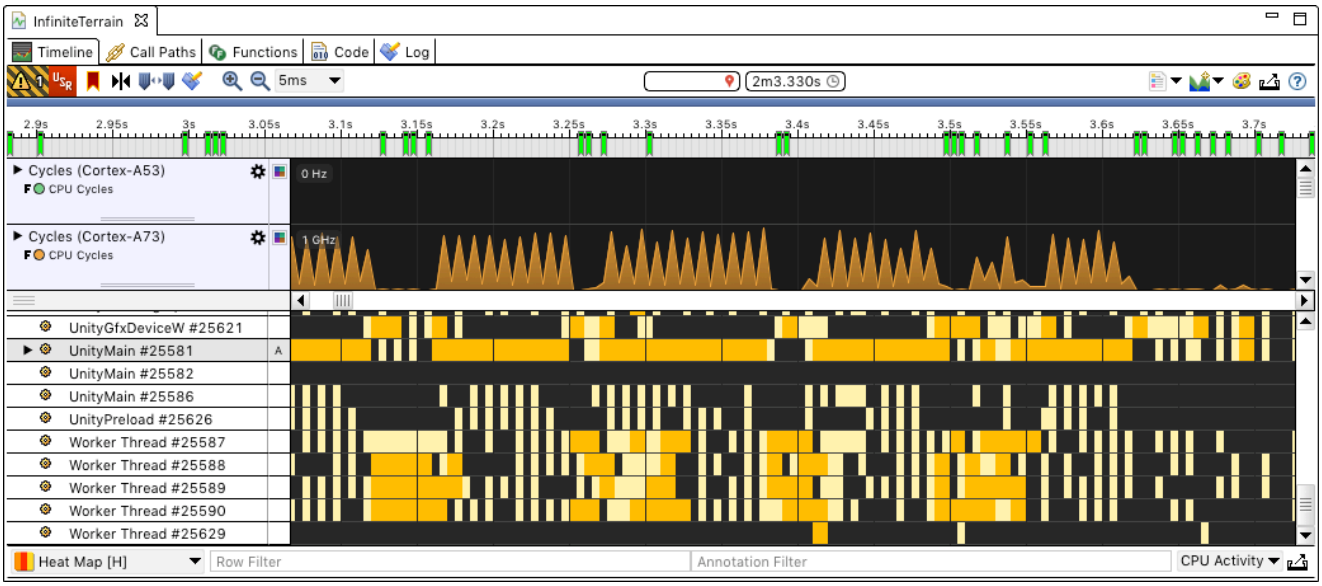

图 12-13 Unity 主线程 的 CPU 简况

下图为**工作线程**所有线程上的 CPU 简况。线程表明 Cortex -A73 和 Cortex -A53 CPU 上存在较少的突发活动：

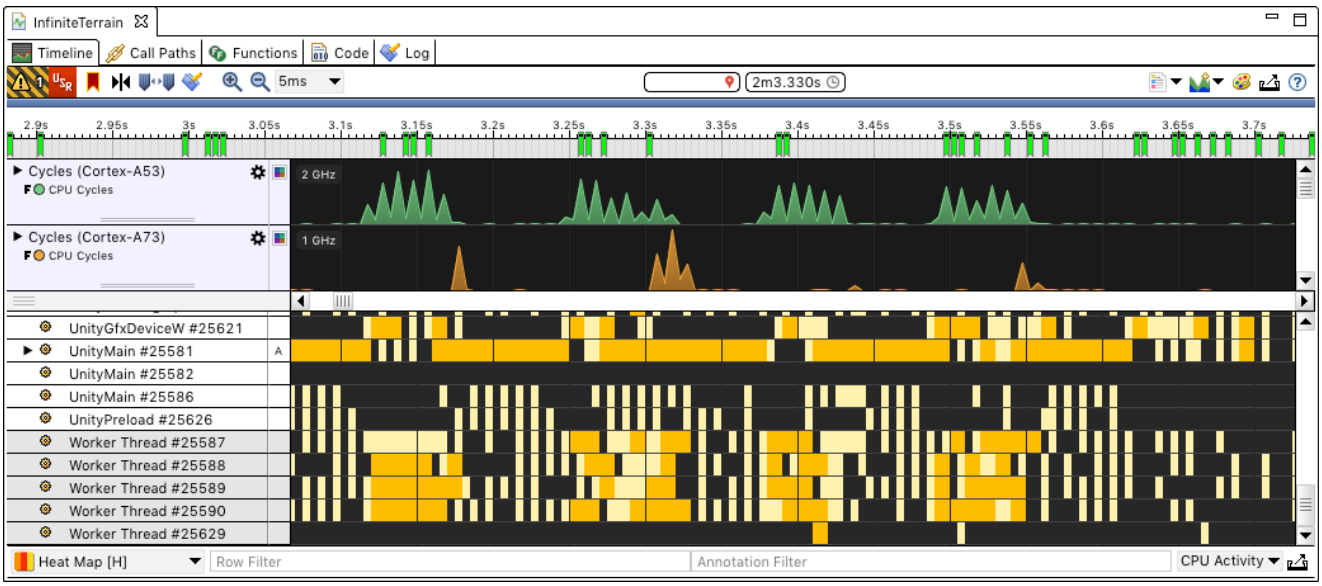

图 12-14 工作线程的 CPU 简况

以下屏幕快照显示了放大后的 **Unity 主线程**的视图。屏幕快照左侧，在 **Unity 主线程**编号的右侧，有个 A 图标。表示存在 *Streamline 注解通道* (SAC)：

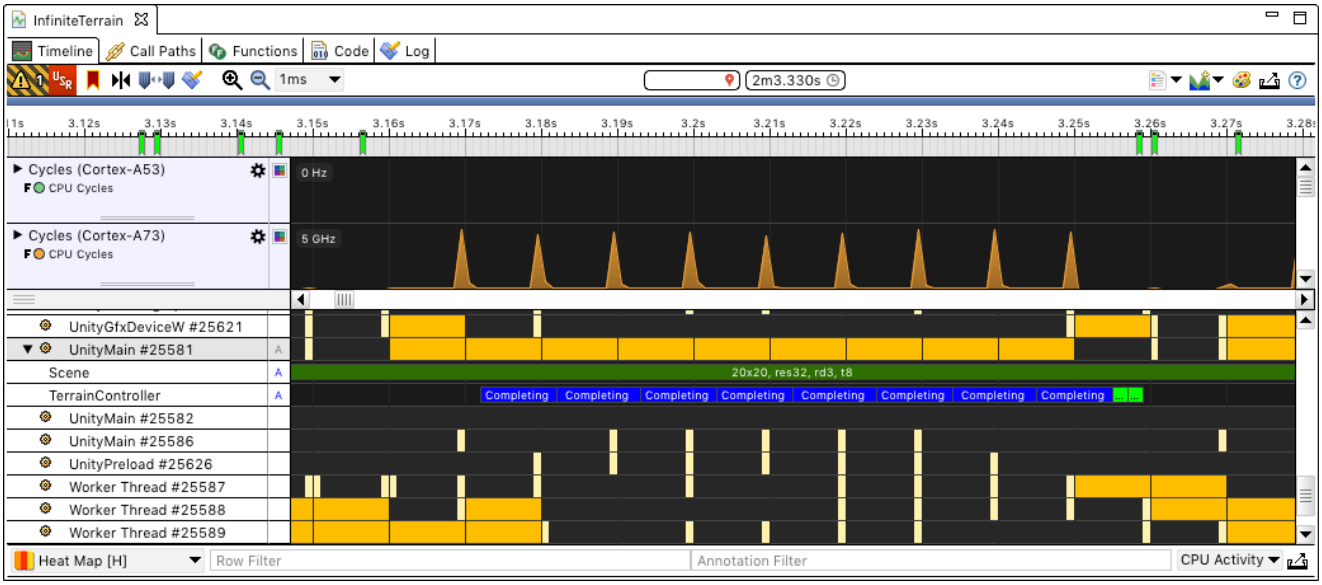

图 12-15 放大 Unity 主线程

游戏中的注释控制**场景**和**地形控制器**两个 Streamline 通道。**场景**通道展示哪个场景正在执行。上图显示正在执行的版本是 20x20、32x32 场景，渲染距离为 3，且 8 个线程并行运行。

地形控制器通道用于指示重要代码片段何时在 Unity 主线程上运行。蓝色块标记地形作业完成后运行的代码。绿色块标记当前计划生成新地形的位置。

所有主线程活动都是基于作业完成时必须完成的任务，需要生成最终网格，然后插入到场景中。

你可针对特定线程，将分析限制在特定时间段内。您可使用 Streamline 中的卡尺标记特定时间区域进行分析。

下图为卡尺的屏幕快照。蓝色的条和箭头代表卡尺，位于时间轴上方。在本场景中选择了一个与地形完成相关的活动高发期：

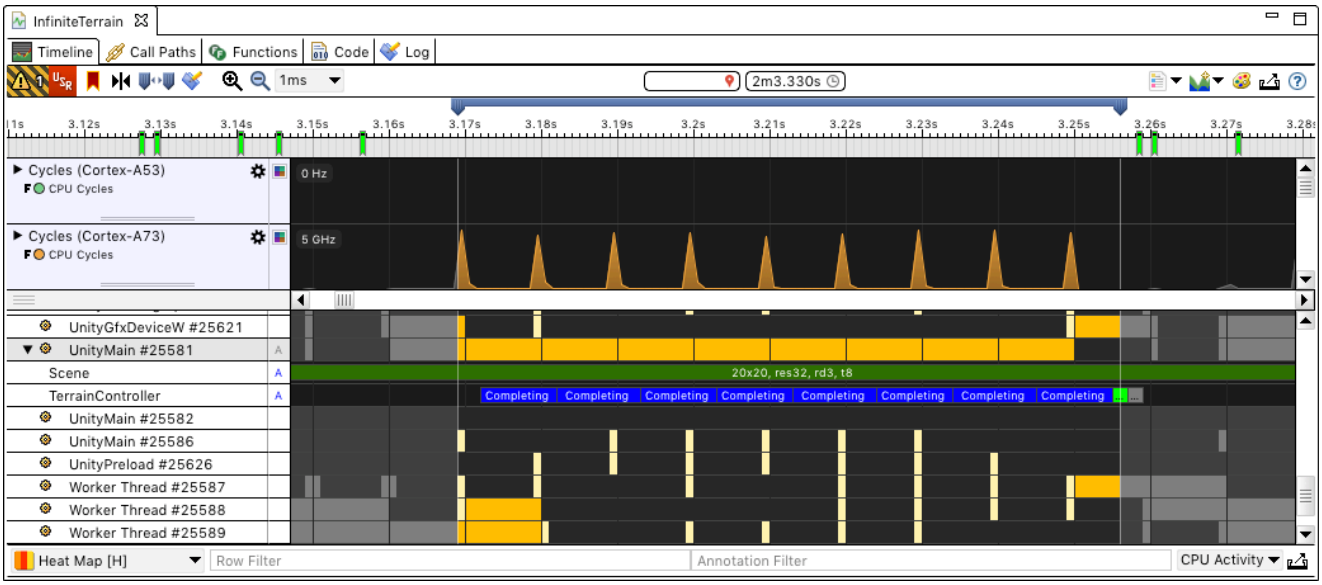

图 12-16 Streamline 卡尺

在**调用路径**视图下，你可以了解在卡尺选择的时间范围内，时间的分配情况。因为使用了 *Unity IL2CPP 脚本后端*，与在运行时使用 Mono 相比，可以获得更多信息。

以下屏幕快照显示了 Unity 内 IL2CPP 脚本后端的详细视图：

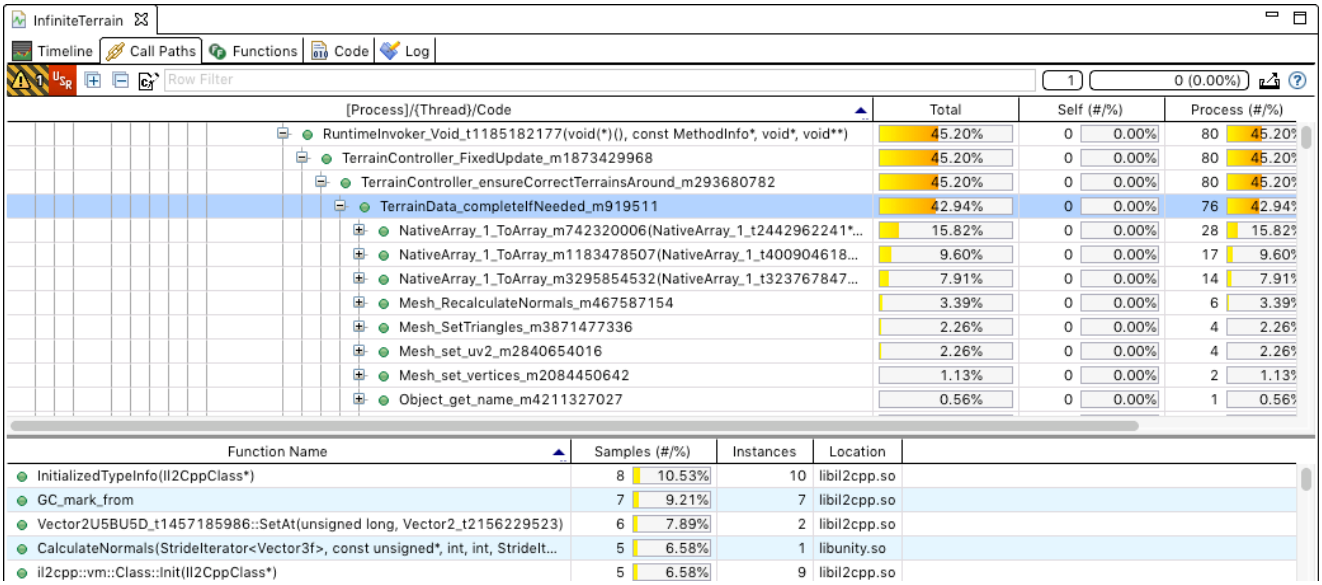

图 12-17 Unity IL2CPP 脚本后端

12.3.4 工作线程

如果使用游戏本身的注释来提供更多的高级上下文，则可以解锁 Streamline 中的更多功能。

过滤后，根据最多八个并行作业的规范，可以看到八个**工作线程**。在此屏幕快照中，Cortex-A53 处理器群集展开后可查看各个内核的利用率。

图 12-18 展开的 Cortex-A53 处理器群集

地形控制器通道中的绿色块表示正在计划新的地形。接下来是所有核心的密集活动。然后在**地形控制器**中存在一些蓝色活动，用于在主线程中处理这些生成的地形。由于未选择 **Unity 主线程**，因此在这些图中看不到主线程活动。

将此场景与第二个场景进行比较，可以看到地形图块具有相同的复杂度，但一次只能安排一个图块：

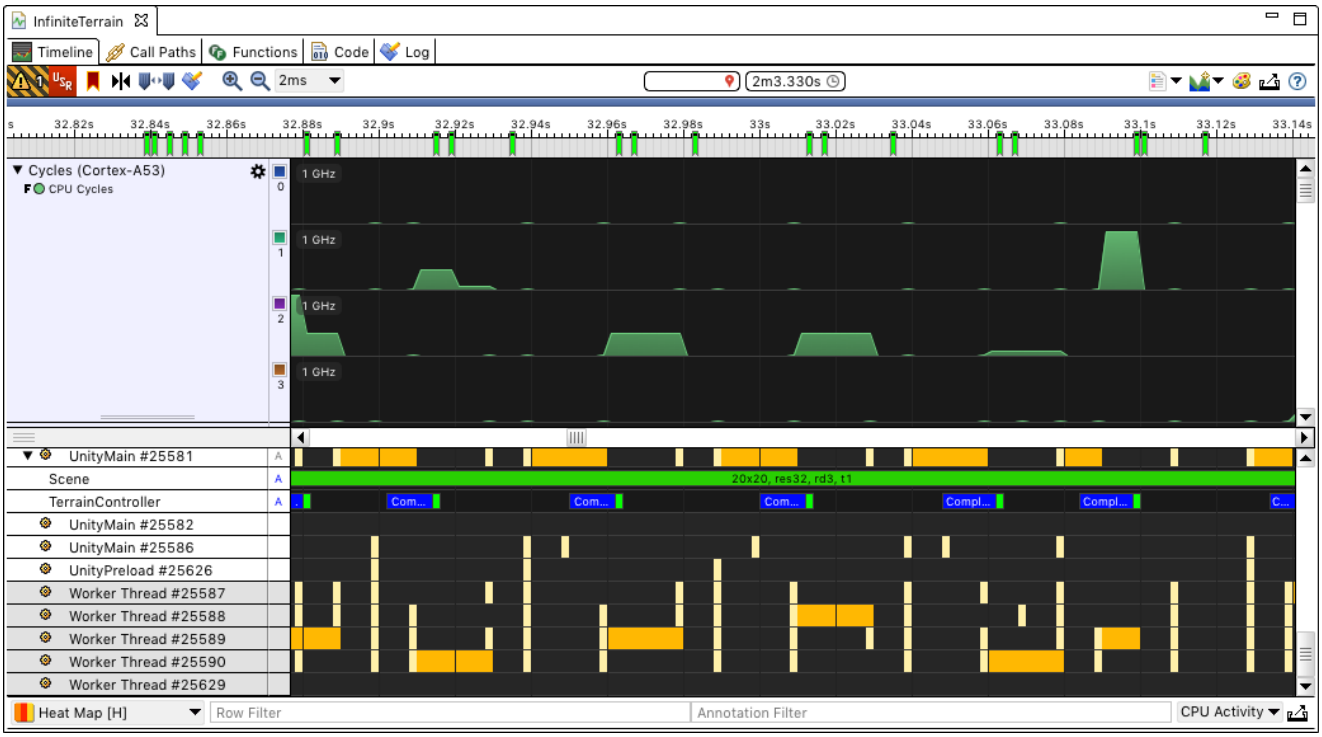

图 12-19 与第二个场景进行比较

此处的注意事项：

1. 由于大部分时间内大多数内核处于空闲或接近空闲状态，因此 CPU 活动较少。
2. 帧速率虽然不完美，但更平滑了。由于任何时候仅完成一帧，因此减少了主线程上的突发活动。

同样，将分析结果与使用较小图块的第三个场景进行比较：

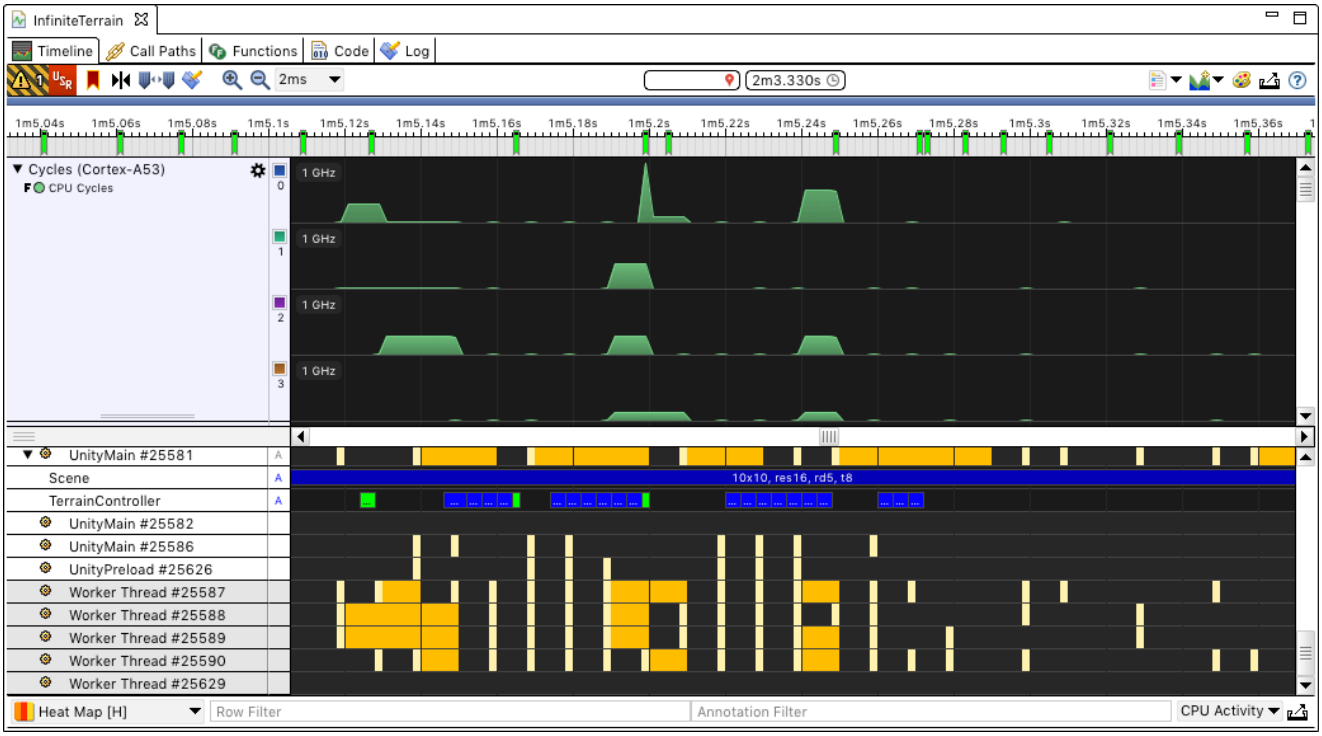

图 12-20 第三个场景

观察到的 CPU 活动强度更低，主线程中的蓝色完成工作块也更短。因此，相对于第一个场景，帧速率更平滑，但总体作业更多，因此请确保地形生成速率与镜头在地形上方飞行的速度保持一致。

第四个场景具有小图块，并且一次只能运行一个地形生成活动。因此，它显示了总体上最平滑的帧速率，但要确保地形生成速率足够块，能够跟上镜头的速度。当镜头在第四场景中快速移动时，并不一定能确保这一点：

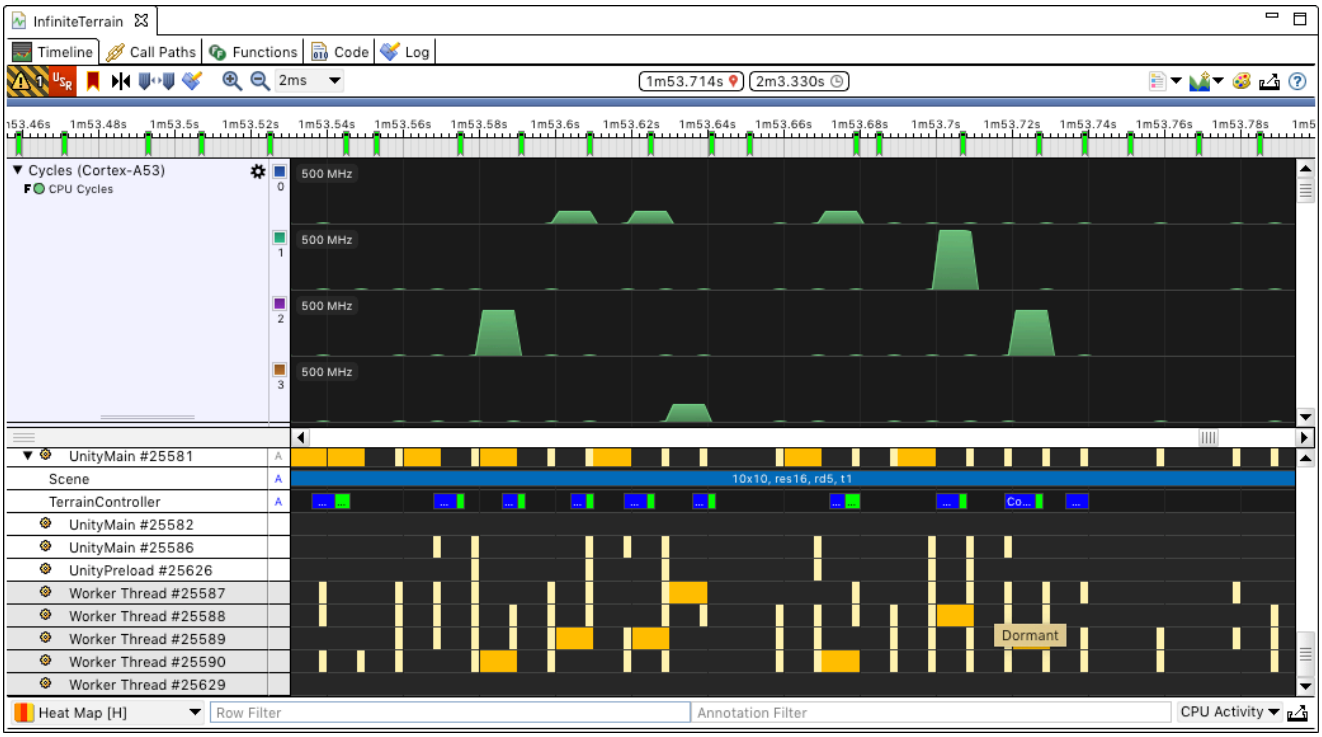

图 12-21 第四个场景

要更深入了解工作线程如何执行地形生成，请使用“自定义活动贴图”。每个自定义活动贴图在左下方菜单中显示为一个选项：

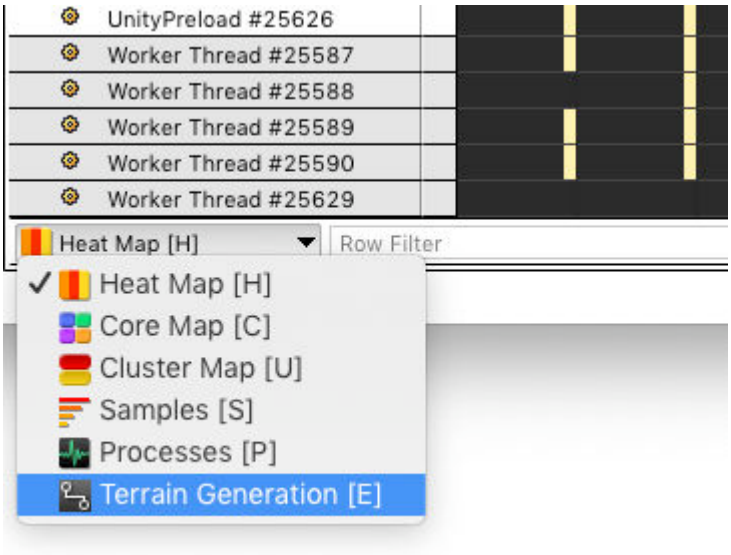

图 12-22 自定义活动贴图

选择**地形生成**视图后，每个地形生成活动的开始和停止时间都会显示一个彩色框。鼠标悬停会显示该地形图块的世界坐标、其启动时间以及完成所需的时间。屏幕快照将显示在 Mali GPU 上进行的计算工作以及随着地形的显示而稳定增长的 GPU 活动。该屏幕快照是在聚焦第一个场景的开始时截取的，场景中最多同时生成了八个大图块。在主线程准备新的几何体时暂停会导致 GPU 长时间空闲：

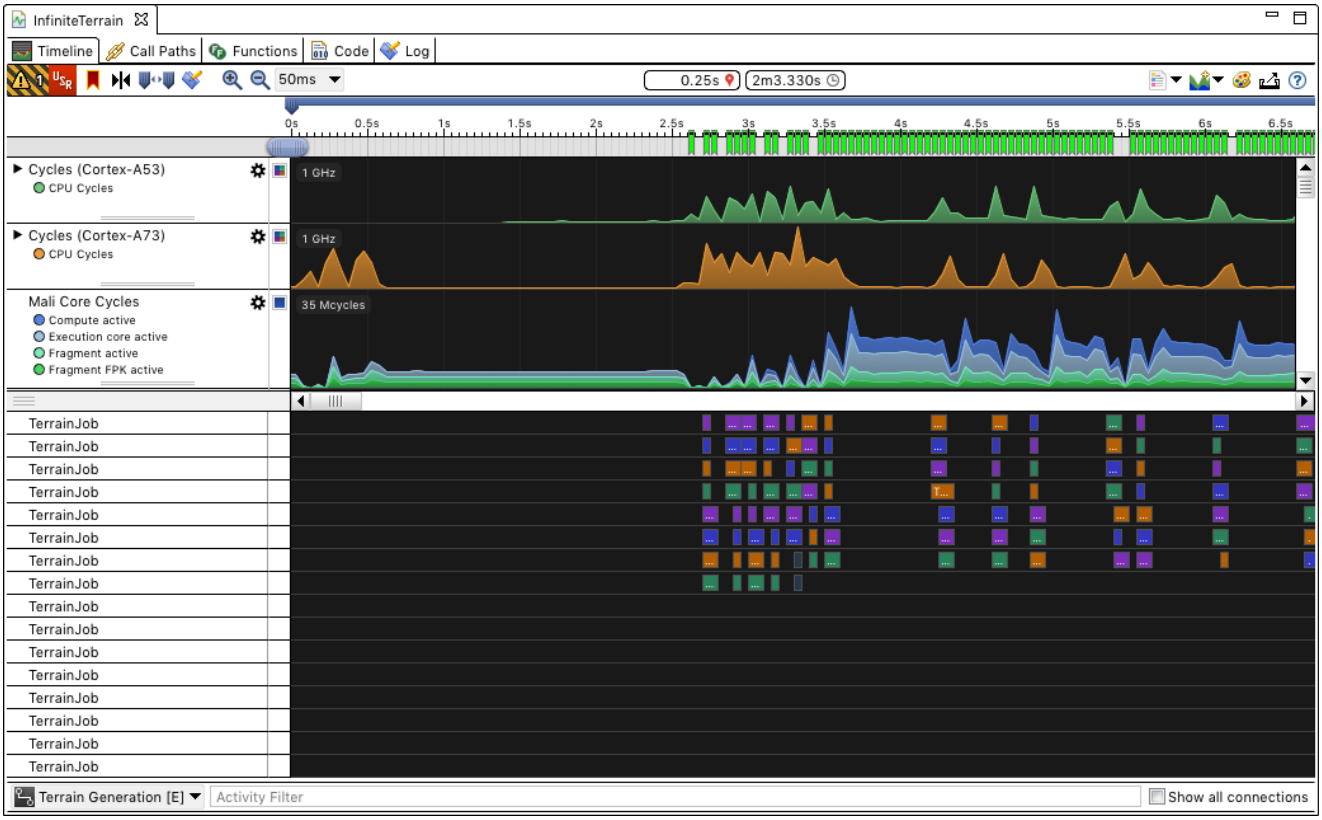

图 12-23 将计算工作图形化

在第四个场景中，由于较小的图块按顺序挨个生成，因此 GPU 活动上升更加平滑。一次仅运行一个地形作业，由于图块大小更小，每个作业都更短。

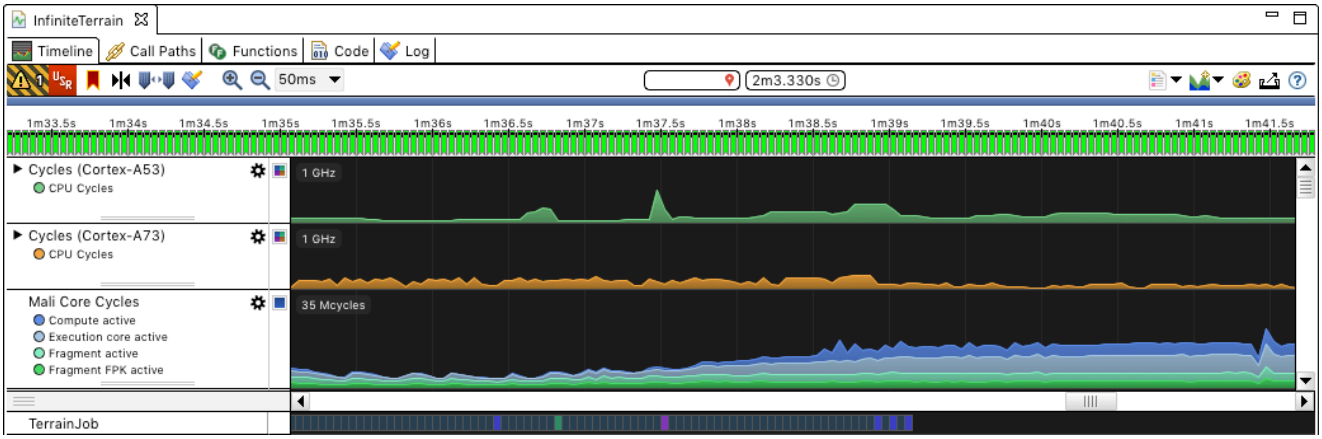

图 12-24 第四个场景的 CPU 周期

12.3.5 Streamline 注解

了解 Streamline 注解的定义、工作原理以及如何在 Unity 中使用，这点很重要。

关于 Streamline 注解

分析安卓应用程序时，在设备上运行一个与该应用程序具有相同用户的独立进程，称为 gator。gator 从各种硬件来源收集分析信息，例如 Mali GPU 和 Arm Cortex -A CPU，并将指标汇总传回计算机。应用程序本身使用一种称为 Streamline 注解的机制将自己的标记和指标插入到汇总指标中。

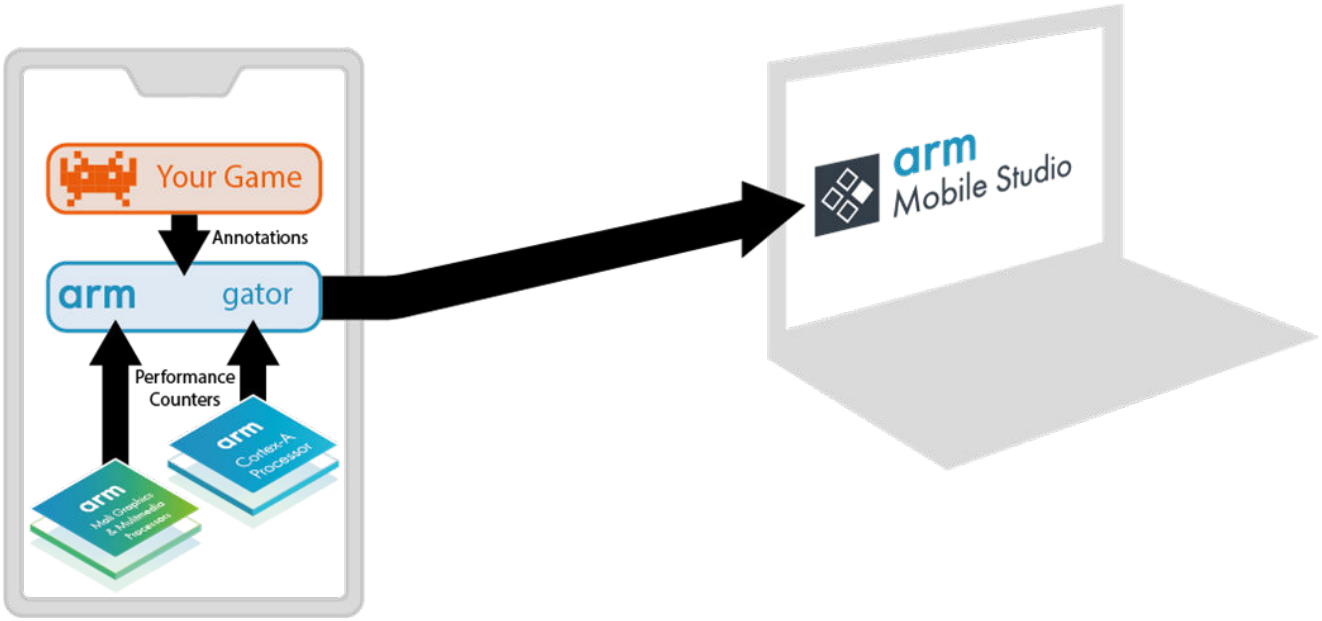

图 12-25 Streamline 的工作原理

如何在 Unity 中使用 Streamline 注解

Streamline 注解使用特定协议，且 Arm Mobile Studio 内置一个开源的 C 实现。为了方便地从 Unity 内容中生成 Streamline 注解，需要一些围绕 C 实现的 C# 封装器。本书中使用的封装器以及所需的 C 实现都放在 Unity 资源包中，你可从 *https://github.com/ARM-software/Tool-Solutions/blob/master/mobile-application-profiling/mobile-studio-with-unity/ArmMobileStudio.unitypackage?raw=true* 上下载。

下载资源包，并根据 Unity 提供的说明将其作为自定义资源包导入到你的项目中。有关说明，请参阅 *https://docs.unity3d.com/Manual/AssetPackages.html*。

该资源包为 `Arm` 命名空间增加了新方法，你可以借此在自己的项目中轻松使用 Streamline 注解。你可在资源包内的 `README.md` 文件内找到 API 文档，*https://github.com/ARM-software/Tool-Solutions/blob/master/mobile-application-profiling/mobile-studio-with-unity/InfiniteTerrain/Assets/Arm%20Mobile%20Studio/README.md*。

12.3.6 有效设置 Unity 项目

你可以使用 Streamline 配置项目、添加标记、添加通道、使用自定义活动贴图和收集配置文件。

配置特定的 Android 播放器设置，以便快速简单地分析 Android 版本：

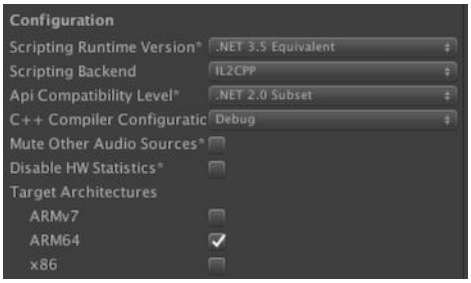

图 12-26 配置设置

确保 IL2CPP 用作脚本后端，并将 C++ 编译器配置设置为调试。这种设置不仅将脚本编译为原生代码以获得更好的性能，还意味着 Streamline 可以查看调试信息，将性能数据映射回调用路径视图中的函数。

将目标体系结构设置为 ARM64（默认为 ARMv7）。如今大多数移动设备都是 64 位的，因此生成的代码质量更高。

添加标记

标记是最容易使用的注解。此处提供使用字符串和颜色的方法。例如，为了显示绿色的每帧标记，在一个游戏对象中使用了以下代码（每帧自动调用一次 `Update()` 方法）。

```
void Update ()
{
  Arm.Annotations.marker("Frame " + Time.frameCount, Color.green);
}
```

添加通道

通道也易于使用。首先，创建通道，并指定通道名称。然后对通道对象使用方法将注解记录到通道中，例如：

```
channel = new Arm.Annotations.Channel("Scene");

channel.annotate(sceneDescription, color);
```

请记住，通道中的注解跨越一个时间段。要在开始下一个注解之前结束注解，请使用 `end()` 方法。例如，在主线程中执行地形完成操作的地形控制器部分打包如下：

```
// Begin annotation
channel.annotate("Completing", Color.blue);

Mesh mesh = obj.GetComponent<MeshFilter>().mesh;

mesh.vertices = job.vertices.ToArray();
mesh.uv = job.uv.ToArray();
mesh.uv2 = job.uv2.ToArray();

mesh.SetTriangles(job.grassTriangles.ToArray(), 0);
mesh.SetTriangles(job.sandTriangles.ToArray(), 1);
mesh.SetTriangles(job.waterTriangles.ToArray(), 2);
mesh.RecalculateNormals();

// End annotation
channel.end();
```

使用自定义活动贴图

自定义活动贴图 (CAM) 只是通道上的另一层。在 CAM 中创建轨道之前，要先给 CAM 命名。给轨道添加注解的方法类似于给通道添加注解。

在示例中，地形生成 CAM 的创建过程如下：

```
terrainCAM = new Arm.Annotations.CustomActivityMap("地形生成");

terrainTracks = new Arm.Annotations.CustomActivityMap.Track[16];

for (int i = 0; i < 16; i++)
{
  terrainTracks[i] = terrainCAM.createTrack("TerrainJob " + i);
}
```

然而，这种情况有一个复杂之处。在 Unity 作业系统中运行的作业不能与游戏的其他对象模型进行太多的交互，这有助于保持线程安全。因此，在作业中存储开始和停止时间，然后当主线程清理作业时，在 CAM 中注册作业的活动。

C# 封装器提供了一个可从作业内部安全调用的函数，该函数以 Streamline 注解需要的格式返回当前时间。

```
UInt64 startTime = Arm.Annotations.getTime();
```

回到主线程中时，选择要使用的轨道（在池中管理轨道，确保没有重叠，这对视觉效果很有帮助），并将作业注册到该轨道上。其中，`job.timings` 是一个两项数组，由作业填写，包含作业的开始时间和停止时间。

```
track.registerJob(obj.name, Color.grey, job.timings[0], job.timings[1]);
```

在 Streamline 中收集基本配置文件

在 Streamline 中收集配置文件的步骤如下：

首先，从以下链接中下载 Arm Mobile Studio Starter 的 Windows、Mac 或 Linux 免费版本：*https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/downloads?_ga=2.211441174.2077719384.1568032571-433518899.1564491901*。

注意：

- 确保 gator 在运行时能够访问移动设备上的应用程序。
- 确保可以将数据从设备中取出并放入工具中。

要使用能在一系列设备中保持稳健的最简单方法，请确保了解以下内容：

- 应用程序的大小（32 位或 64 位）。如果你按照前面的说明用 `ARM64` 选项构建 Unity 游戏，则应用程序的大小为 64 位。
- 应用程序的安装包名（在 Unity Android 播放器设置中指定）。我们应用程序的安装包名为 `com.Arm.InfiniteTerrain`。
- 如何获取 gator 二进制文件。在你 Mobile Studio 安装包的 `streamline/bin/arm` （32 位）或 `streamline/bin/arm64` （64 位）文件夹下查找 Arm。

了解上述信息后，按如下步骤进行分析：

- 确保应用程序安装在设备上。
- 使用 Android `adb` 工具将网络端口从移动设备转发至运行 Arm Mobile Studio 的系统的本地端口上。
- 使用 `adb` 将 32 位或 64 位的 gator（具体取决于你的应用程序）推送到设备上，并采用与应用程序相同的安装包名运行 gator。
- 启动 Streamline，并将其连接到接收 `adb` 转发流量的本地端口上。选择要收集的性能计数器。
- 应用程序在移动设备上启动时，Streamline 会在分析数据到来时开始收集数据。

gator 运行时，可以安装新版本的应用程序，启动和停止 Streamline 并执行更多分析，而无需重新启动 gator。

为了简化过程，请从 *https://github.com/ARM-software/Tool-Solutions/tree/master/mobile-application-profiling/mobile-studio-scripts* 上下载 `gatorme` 脚本。使用该脚本配置并运行 `gator` 和 `adb`。这需要你要运行的 gator 二进制文件的路径、应用程序的安装包名以及设备使用的 Mali GPU。如果 gator 对设备进行探测后不能确定其使用的是哪种 GPU，那么提供此信息会有所帮助。这一脚本还执行其他多个步骤，以确保这种方法在大多数移动设备上运行良好，并确保 gator 在分析完成时正确关闭。

`gatorme` 文档解释了细节，但是这里的例子描述了 APK 安装在设备上时，如何分析 InfiniteTerrain 内容。

首先，在命令行中运行 `gatorme`：

```
$ ./gatorme.sh com.Arm.InfiniteTerrain G71 ./mobilestudio-macosx/streamline/bin/arm64/gatord
```

现在启动 Streamline 并做好数据收集准备。按照如下设置启动 Streamline：

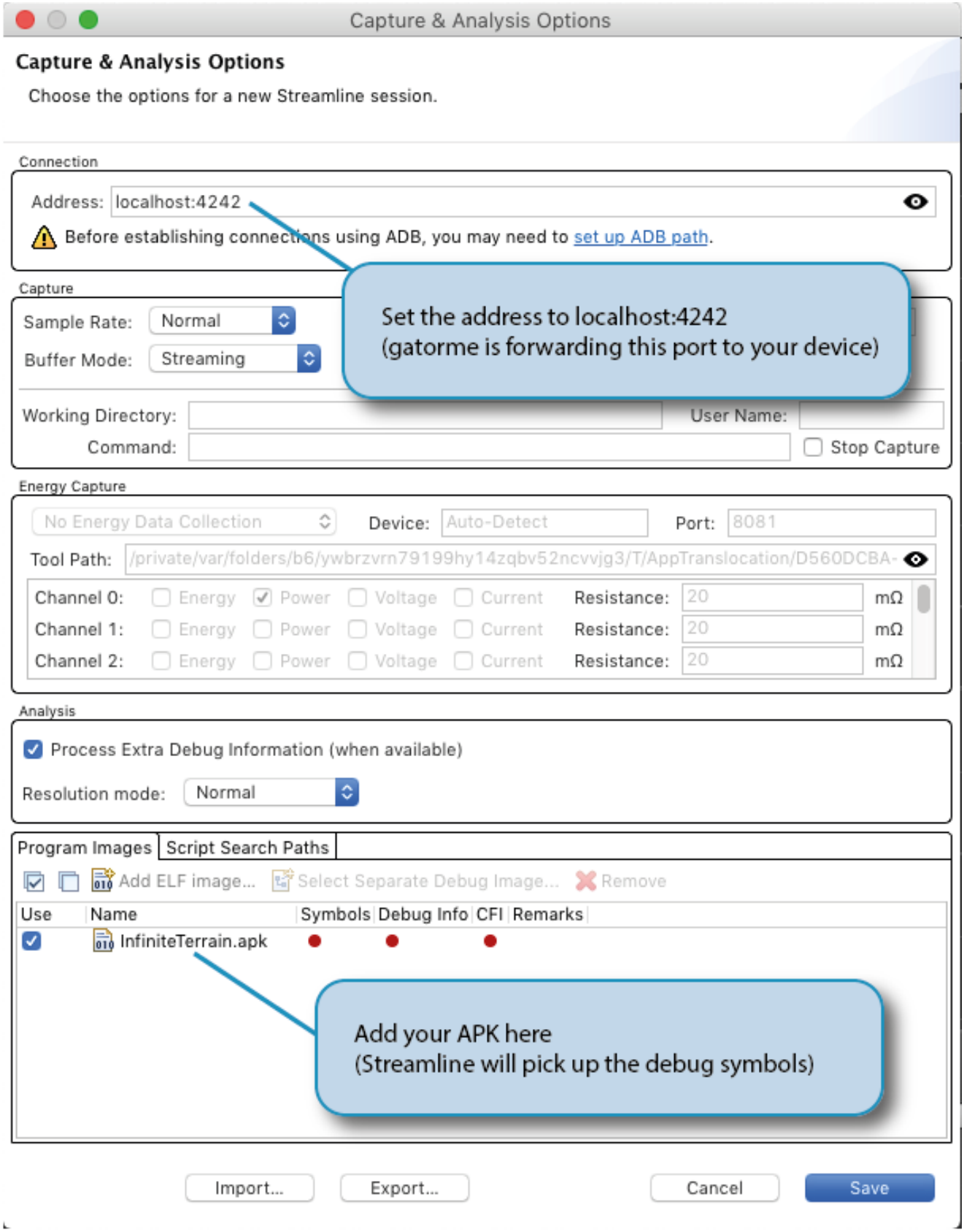

图 12-27 用于启动 Streamline 的设置

设置时，重复的部署/分析/修复步骤很容易使 `gatorme` 在应用程序关闭时仍保持运行，从 Unity 直接构建并运行，并获取更多的 Streamline 信息。

如果持有 Apache 2.0 许可证，你可以从 GitHub 下载 InfiniteTerrain 示例的源代码。要下载源代码，请访问 *https://github.com/ARM-software/Tool-Solutions/tree/master/mobile-application-profiling/mobile-studio-with-unity*。

对于源代码和图形资源，还有一个名为 `ArmMobileStudio.unitypackage` 的 Unity 自定义资源包，你可从 *https://github.com/ARM-software/Tool-Solutions/blob/master/mobile-application-profiling/mobile-studio-with-unity/ArmMobileStudio.unitypackage?raw=true* 下载。

Unity 提供了有关说明，指导你如何将该包导入到你自己的项目中，以及如何向你的包中添加 Streamline 注解。有关说明，请参阅 *https://docs.unity3d.com/Manual/AssetPackages.html*。

要下载预构建版本的 `InfiniteTerrain.apk`，请访问 *https://github.com/ARM-software/Tool-Solutions/blob/master/mobile-application-profiling/mobile-studio-with-unity/InfiniteTerrain.apk?raw=true*。该版本为 64 位的 Android 开发版本，已准备就绪，可以部署到设备上并开始快速分析内容。

附录 A
修订

本附录将介绍本书发行版本之间的变化。

它包含以下部分:

- *A.1 修订* on page 附录-A-276.

A.1 修订

本附录将介绍本书发行版本的增添内容。

表 A-1 3.0 版的增添内容

增添内容	位置	受影响的版本
在自定义着色器中使用 Enlighten	-	3.0 版
组合反射	*8.3 组合反射* on page 8-137	3.0 版
使用 Early-z	*8.7 使用 Early-z* on page 8-160	3.0 版
脏镜头效果	*8.8 脏镜头效果* on page 8-161	3.0 版
光柱	*8.9 光柱* on page 8-164	3.0 版
雾化效果	*8.10 雾化效果* on page 8-167	3.0 版
高光溢出	*8.11 高光溢出* on page 8-174	3.0 版
冰墙效果	*8.12 冰墙效果* on page 8-181	3.0 版
过程天空盒	*8.13 过程天空盒* on page 8-186	3.0 版
萤火虫	*8.14 萤火虫* on page 8-194	3.0 版
切线空间至世界空间转换工具。	*8.15 切线空间至世界空间法线转换工具* on page 8-198	3.0 版

表 A-2 3.0_01 版的变更

变更	位置	受影响的版本
从“使用 Early-z”中删除了一个列表条目	*8.7 使用 Early-z* on page 8-160	3.0 版

表 A-3 3.1 版的变更

变更	位置	受影响的版本
添加了有关虚拟现实的章节	*章 9 虚拟现实* on page 9-205	3.1 版

表 A-4 3.2 版的变更

变更	位置	受影响的版本
更新了 Enlighten 章节的信息和结构	-	3.2 版首次发布

表 A-5 3.3 版的变更

变更	位置	受影响的版本
添加了有关 Vulkan 的章节	*章 11 Vulkan* on page 11-240	3.3 版首次发布
添加了有关 Mali 图形调试器的章节	*章 12 Arm Mobile Studio* on page 12-249	3.3 版首次发布

表 A-6 3.3_01 版的变更

变更	位置	受影响的版本
更新了第 8.3 节中的屏幕快照	*11.3 在 Unity 中启用 Vulkan* on page 11-244	3.3 版第二次发布

表 A-7 4.0 版的变更

变更	位置	受影响的版本
添加了 Arm Mobile Studio 章节。	*章 12 Arm Mobile Studio* on page 12-249	4.0 版首次发布
将 Mali 图形调试器的所有实例重命名为“图形分析器”。	在整篇文档中	4.0 版首次发布
将图形分析器的内容移到了 Arm Mobile Studio 章节中。	*章 12 Arm Mobile Studio* on page 12-249	4.0 版首次发布
更新了 Arm 品牌图像。	在整篇文档中	4.0 版首次发布
在正文中删除了所有对 Enlighten 的引用。	在整篇文档中	4.0 版首次发布

表 A-8 4.1 版的变更

变更	位置	受影响的版本
添加了几何最佳实践章节。	*章 5 实时 3D 美术最佳实践：几何图形* on page 5-62	4.1 版首次发布
添加了纹理最佳实践。	*章 6 实时 3D 美术最佳实践：纹理* on page 6-76	4.1 版首次发布
添加了材质和着色器最佳实践。	*章 7 实时 3D 美术最佳实践：材质和着色器* on page 7-96	4.1 版首次发布
添加了高级 VR 图形方法章节。	*章 10 高级 VR 图形技巧* on page 10-217	4.1 版首次发布