

# 第1章

## 初识游戏图形

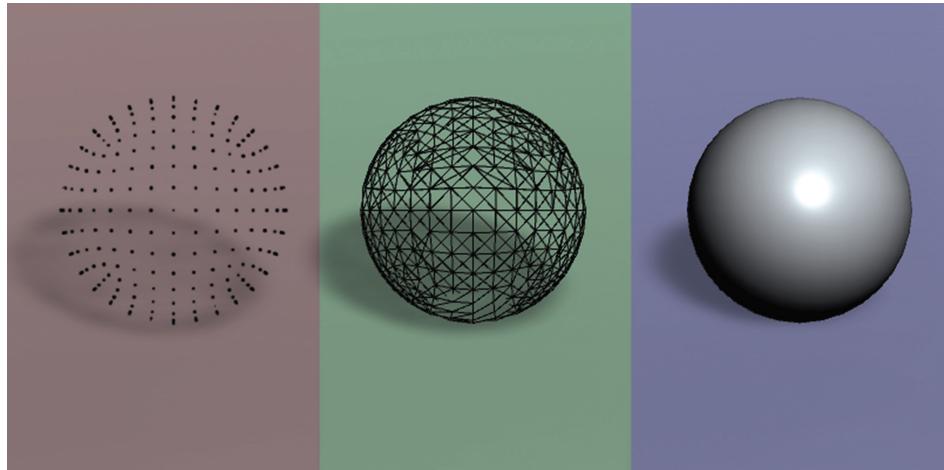


图 1-1 网格的不同成分的示例：从左到右是顶点、边和面

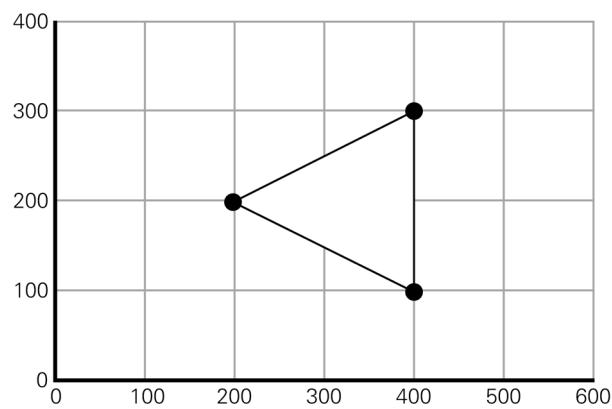


图 1-2 一个简单的三角形网格

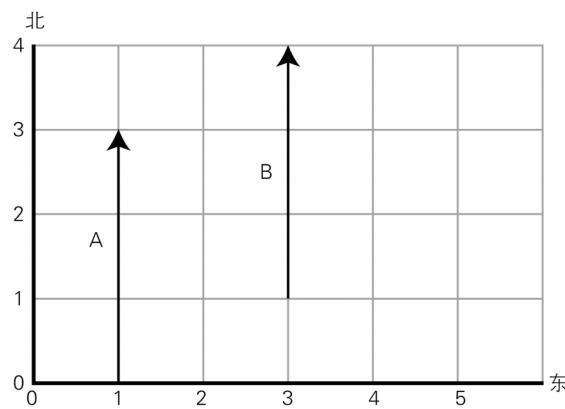


图 1-3 两个位置上相同向量的示例

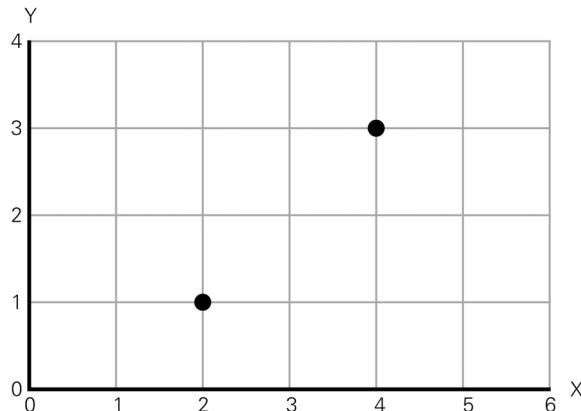


图 1-4 空间中的两个位置，通常把这样的位置表示为向量

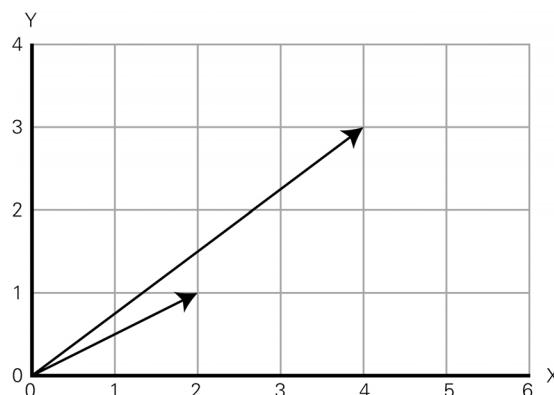


图 1-5 将向量线添加到图 1-4 中的位置上

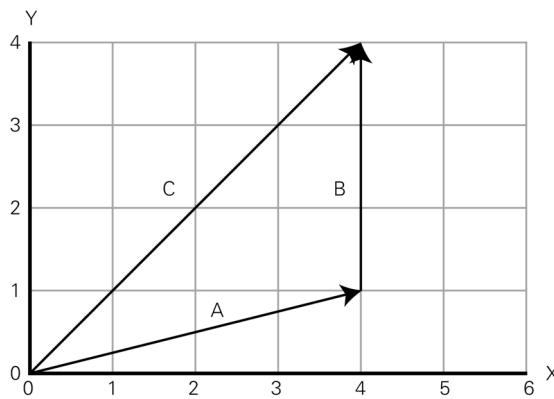


图 1-6 向量 A 和 B 相加，结果是向量 C

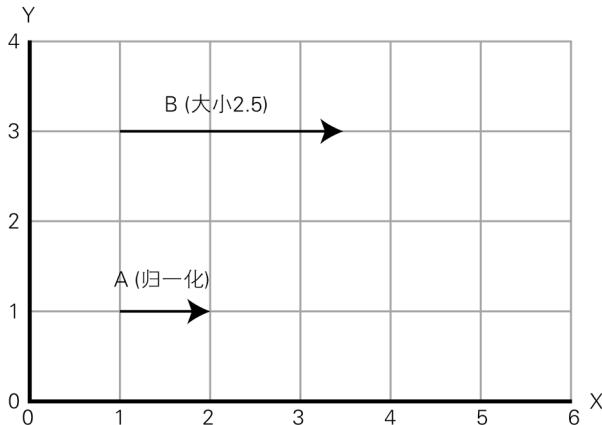


图 1-7 向量 A 是一个归一化向量。如果我们把向量乘以 2.5，我们就得到向量 B

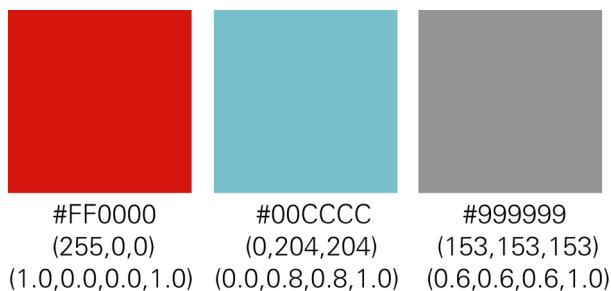


图 1-8 程序员表示颜色的不同方法示例

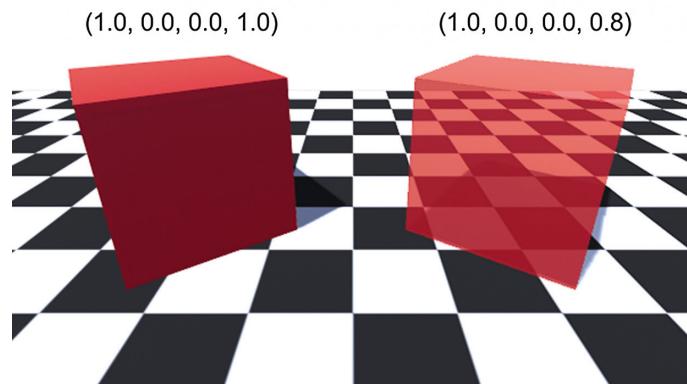


图 1-9 左边立方体的 alpha 值是 1.0，因此是实心的；右边立方体的 alpha 值是 0.8

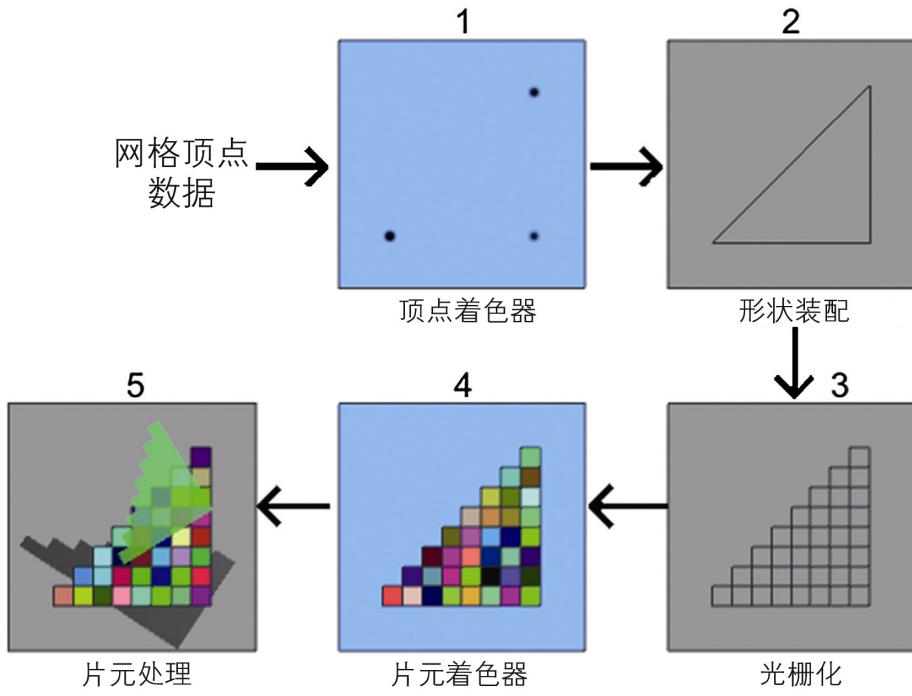


图 1-10 图形管线的简化视图，步骤的顺序由箭头和位于每个框上的数字所示

## 第 2 章

# 第一个着色器

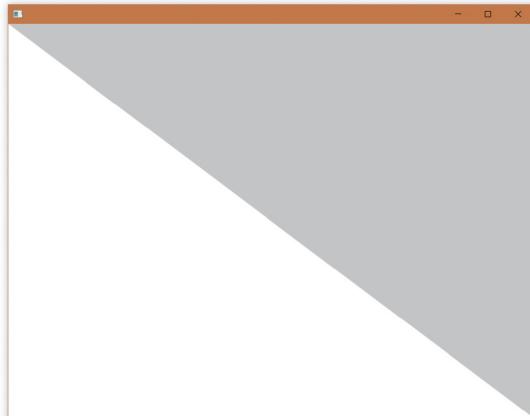


图 2-1 一个白色三角形遮住了一半的窗口

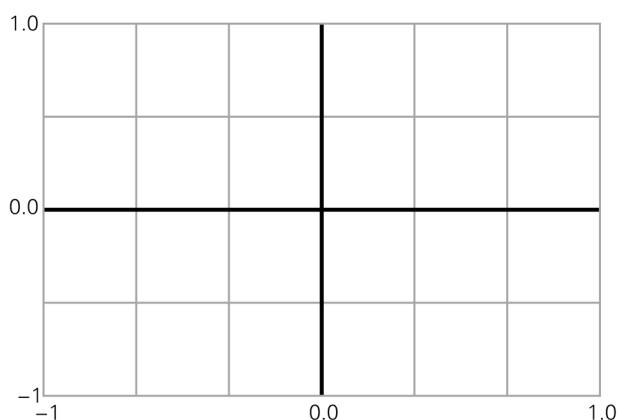


图 2-2 将归一化设备坐标覆盖在窗口上的情形

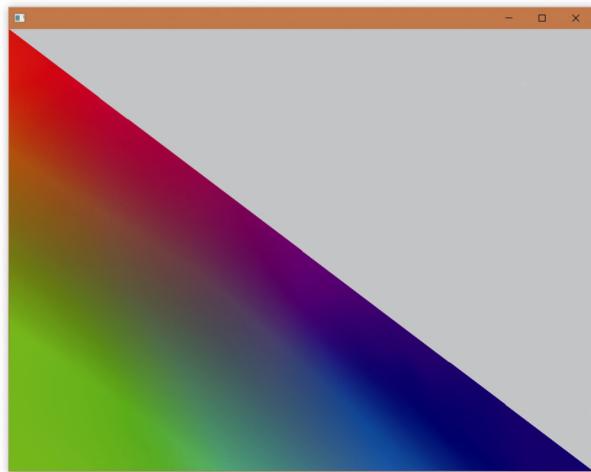


图 2-3 炫酷，对吧？

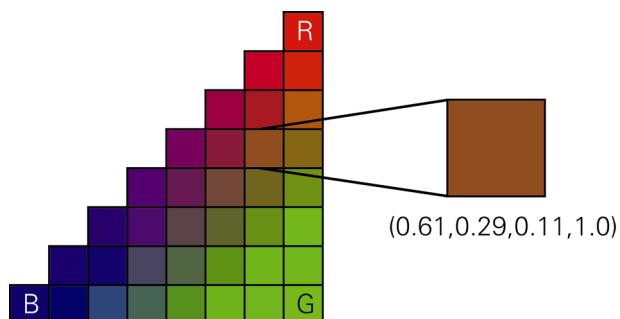


图 2-4 片元插值的一个例子

## 第3章

# 使 用 纹 理



图 3-1 左纹理用于三维网格；右纹理显示在二维平面上

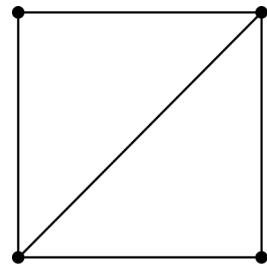


图 3-2 由两个三角形拼成的四边形

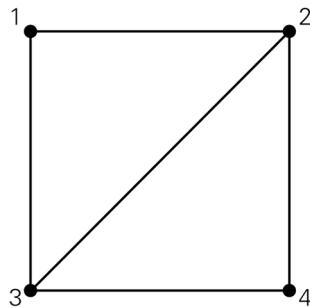


图 3-3 标记了每个顶点索引的四边形网格

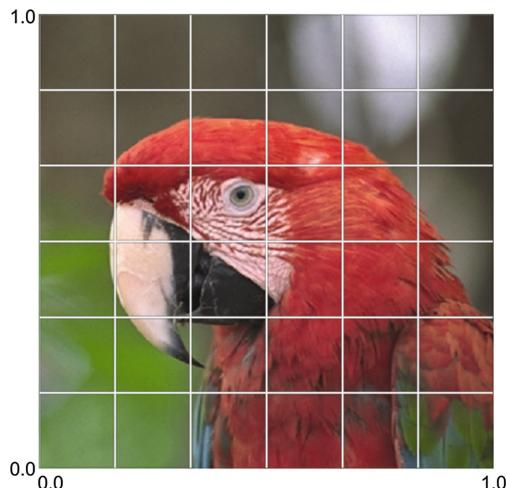


图 3-4 标注了 UV 坐标的鹦鹉纹理

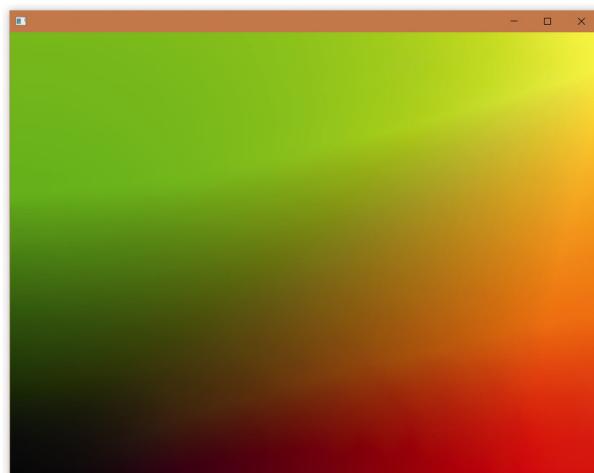


图 3-5 在全屏四边形上可视化 UV 坐标



图 3-6 显示在全屏四边形上的鹦鹉纹理



图 3-7 加上偏移 UV 坐标的鹦鹉纹理



图 3-8 全屏四边形上使用“REPEAT” wrap mode 的鹦鹉纹理

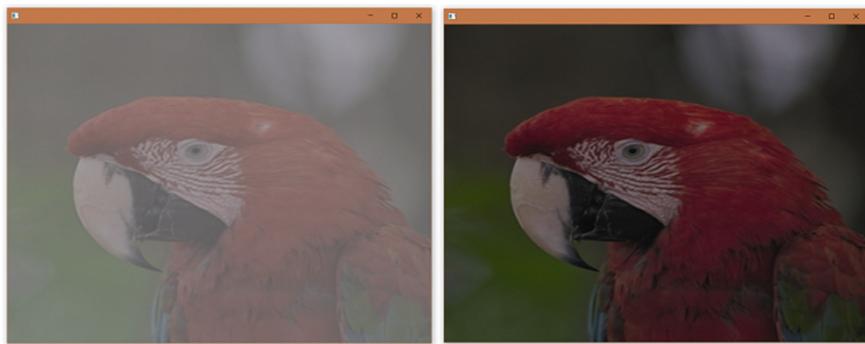


图 3-9 纹理颜色乘以 0.5, 左边图像是启用了 alpha 混合后的效果, 右边图像是禁用了 alpha 混合后的效果

$\begin{array}{ c } \hline + \\ \hline - \\ \hline \end{array}$	1.0,0,0,0.0	0.0,0,5,0.8	0.8,0,8,0.8
0.5,0,5,0.5	1.0,0,5,0.5 0.5,0,0,0.0	0.5,1,0,1.0 0.0,0,0,0.3	1.0,1,0,1.0 0.3,0,3,0.3
0.0,1,0,0.0	1.0,1,0,0.0 1.0,0,0,0.0	0.0,1,0,0.8 0.5,1,0,1.0	0.8,1,0,0.8 0.8,0,0,0.8

图 3-10 颜色加减表

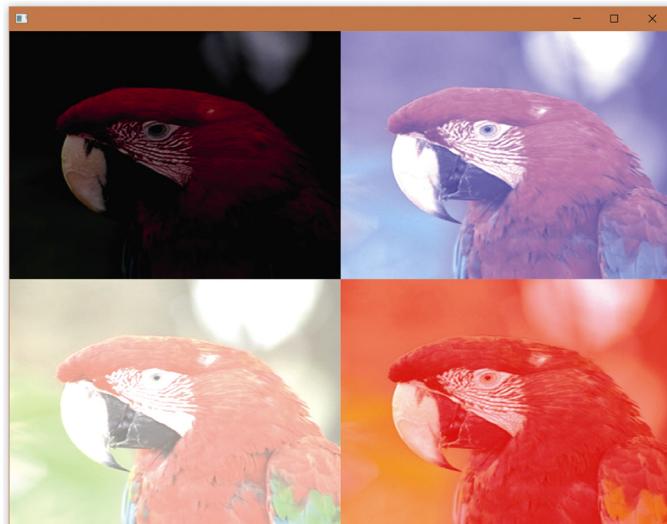


图 3-11 从鹦鹉纹理均匀地增加和减去颜色的示例

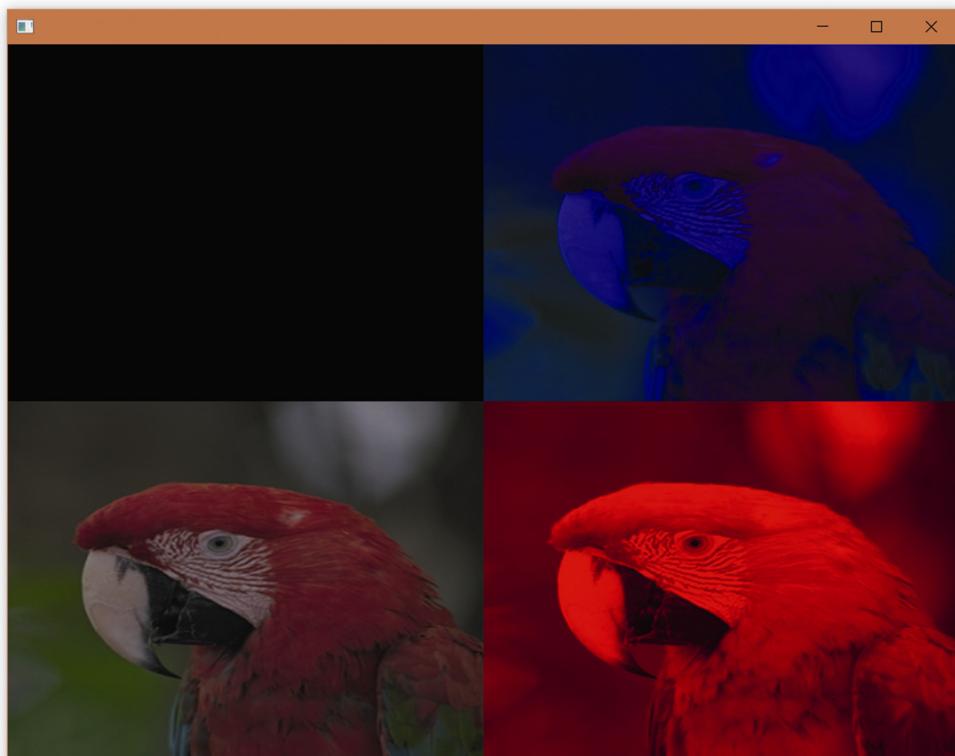


图 3-12 将鹦鹉纹理乘以图 3-11 中使用的相同颜色

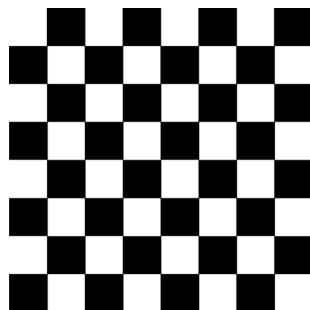


图 3-13 将在下面的示例中使用的棋盘格纹理

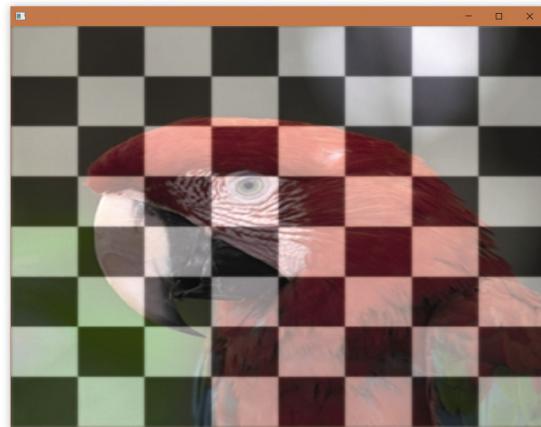


图 3-14 棋盘格纹理和鹦鹉纹理 50/50 混合

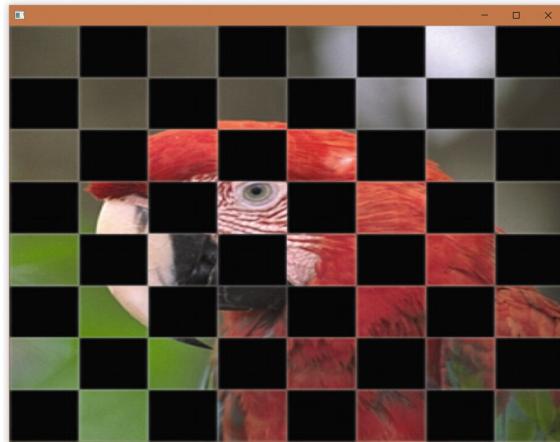


图 3-15 基于棋盘格纹理红色通道的纹理间混合

## 第 4 章

# 半透明与深度

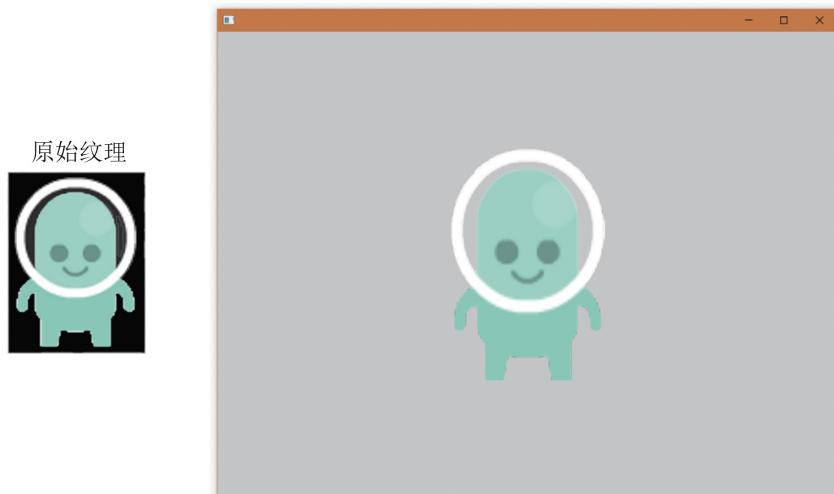


图 4-1 在四边形上渲染的二维角色。原始纹理显示在左侧。黑色区域是 alpha 为 0 的像素

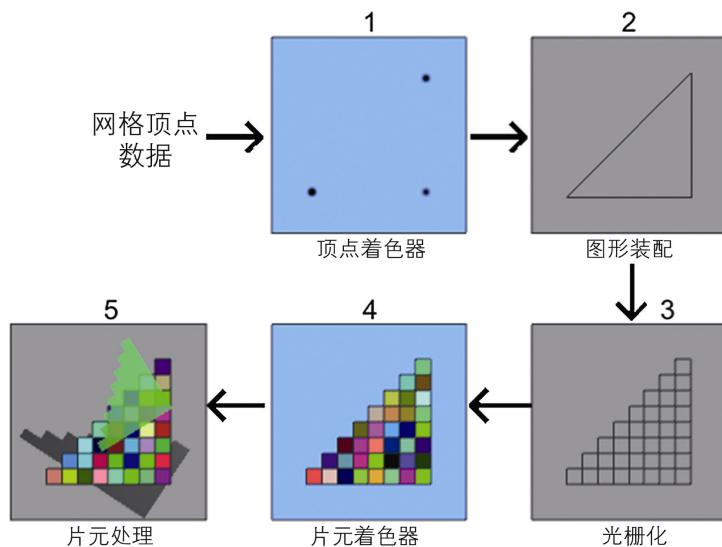


图 4-2 图形管线的简化视图

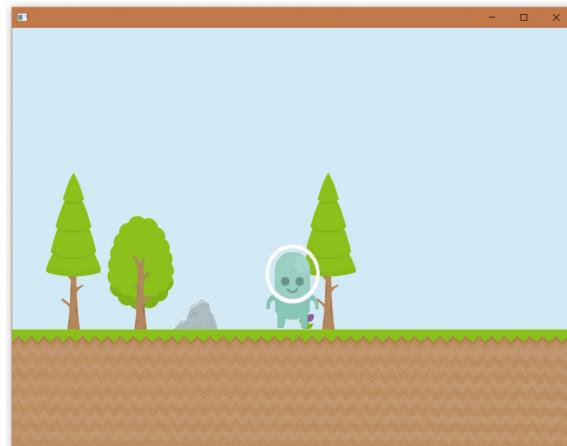


图 4-3 到目前为止，本章的示例程序的运行结果

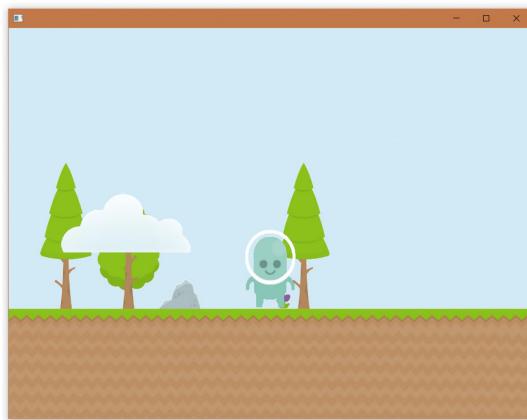


图 4-4 使用 alpha 测试的云朵

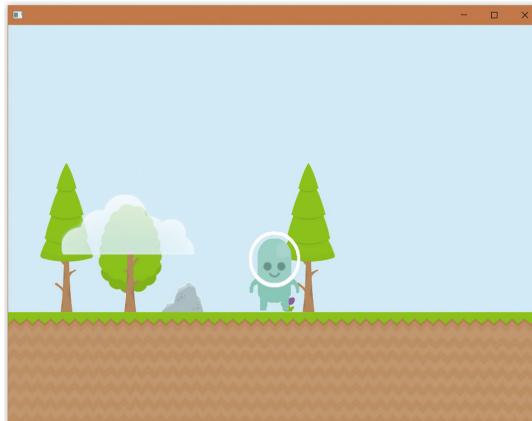


图 4-5 使用了 alpha 混合的云朵



图 4-6 添加了太阳光的场景



图 4-7 将一些附加的太阳光添加到场景后示例的画面效果

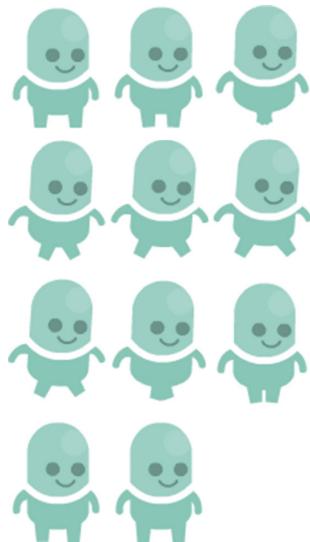


图 4-8 外星人精灵表

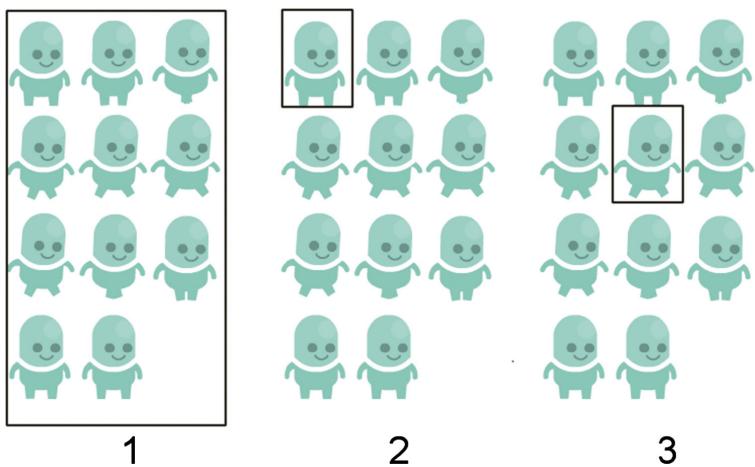


图 4-9 如何调整 UV 坐标以显示精灵表的单个帧

## 第 5 章

# 使物体动起来

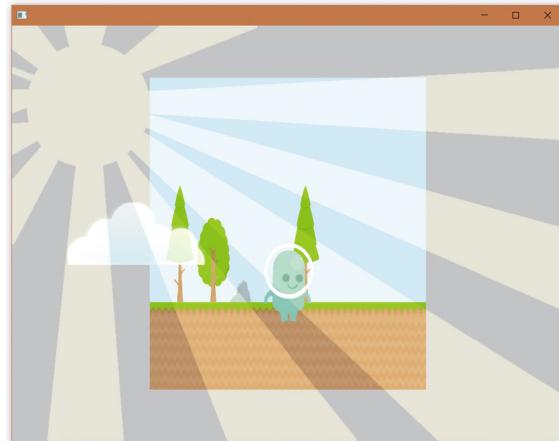


图 5-1 在 X 和 Y 维度缩放四边形网格

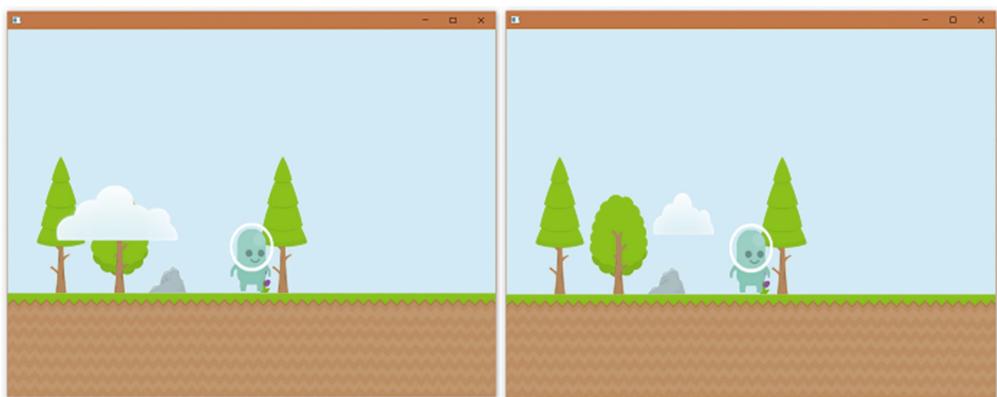


图 5-2 左图是没有阳光的场景。右图是云朵网格缩放到一半大小的场景——(0.5, 0.5, 1.0)。它似乎向屏幕的中心移动，这是它的对象空间的原点

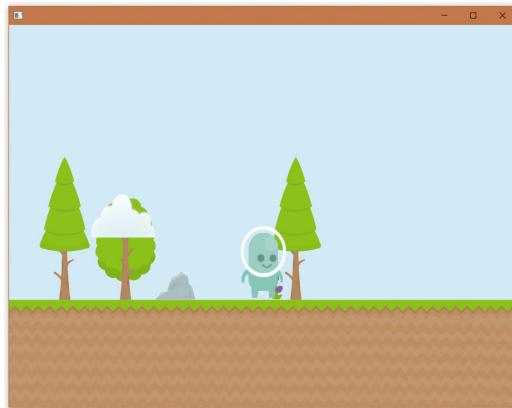


图 5-3 正确缩放的云朵

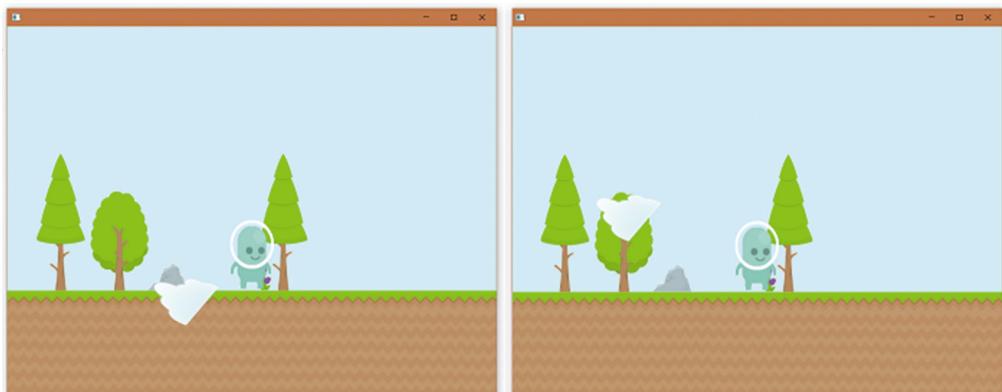


图 5-4 左边的屏幕截图展示了在平移之后执行旋转的结果。

右边的屏幕截图展示了先执行旋转后平移的结果

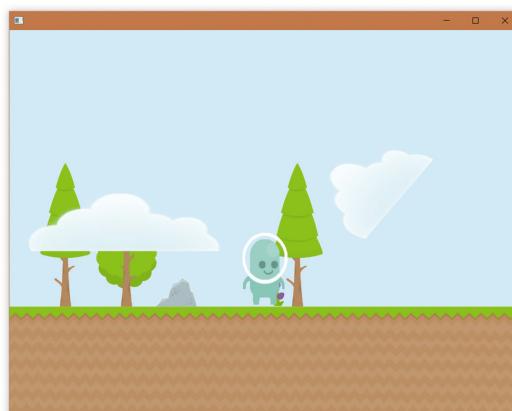


图 5-5 使用不同的变换统一变量提交多个绘图调用

## Shader 开发实战

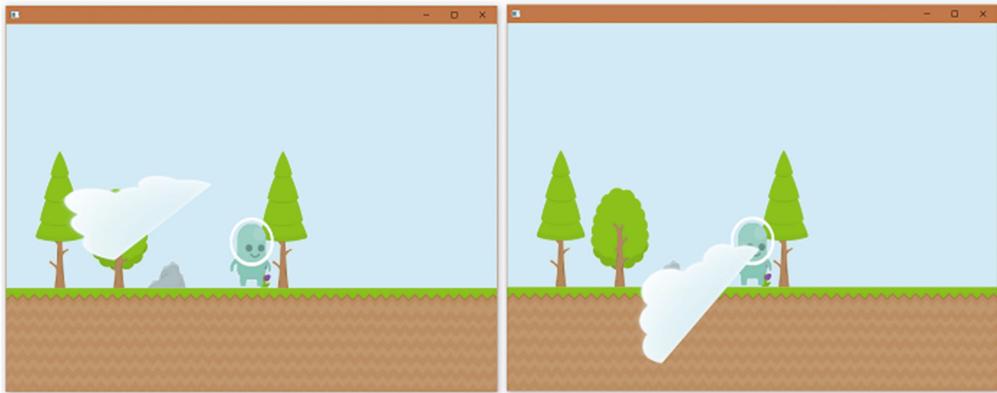


图 5-8 左边的屏幕截图显示了在顶点着色器中使用代码清单 5-18 中的矩阵 resultA 的结果。

右屏幕截图使用了矩阵 resultB

## 第 6 章

# 摄像机和坐标

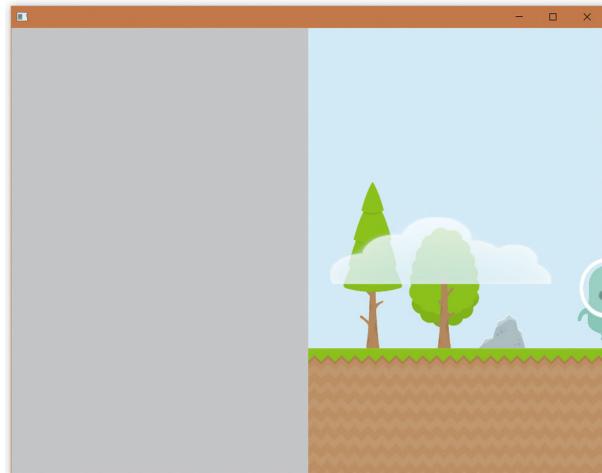


图 6-1 使用视图矩阵渲染场景

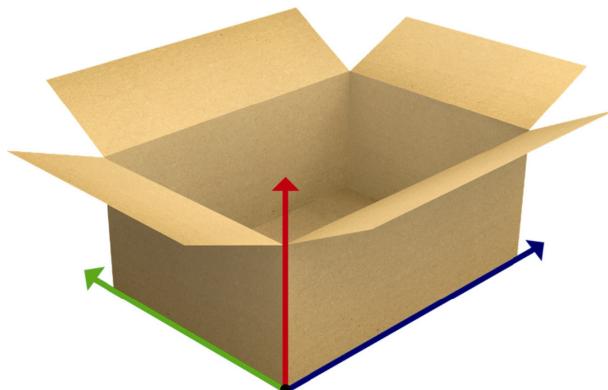


图 6-2 定义纸板箱的“盒子空间”

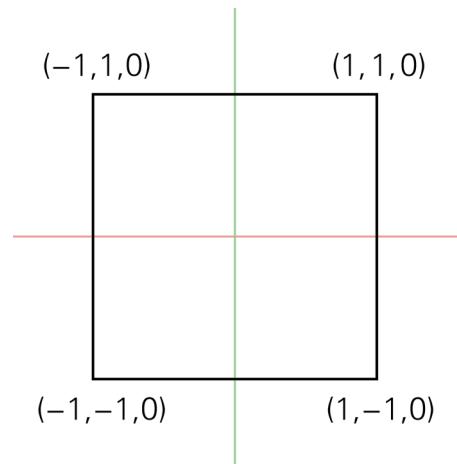


图 6-3 标有对象空间顶点位置的四边形网格

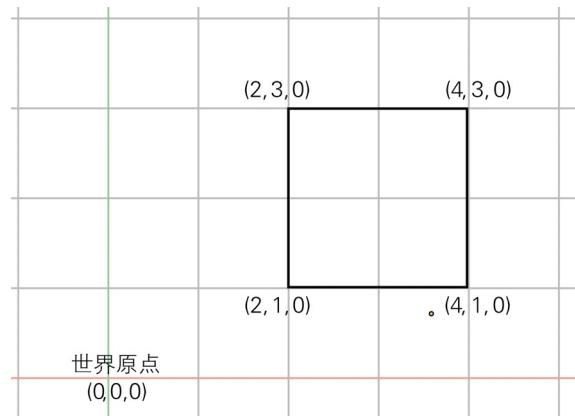


图 6-4 在世界空间中放置的四边形

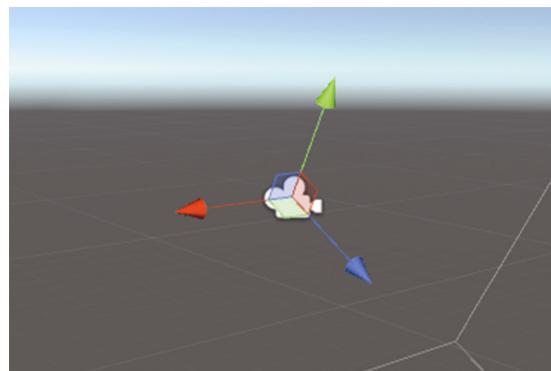


图 6-5 蓝色箭头(右下方的箭头)是该摄像机视图空间中的 Z 轴; 它指向摄像机正在查看的方向,  
而不是世界空间 Z 轴的方向

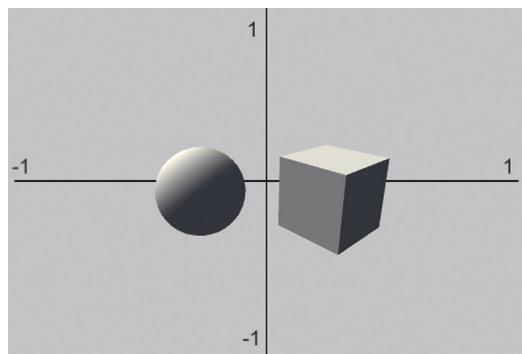


图 6-6 标记了裁剪空间坐标轴的帧

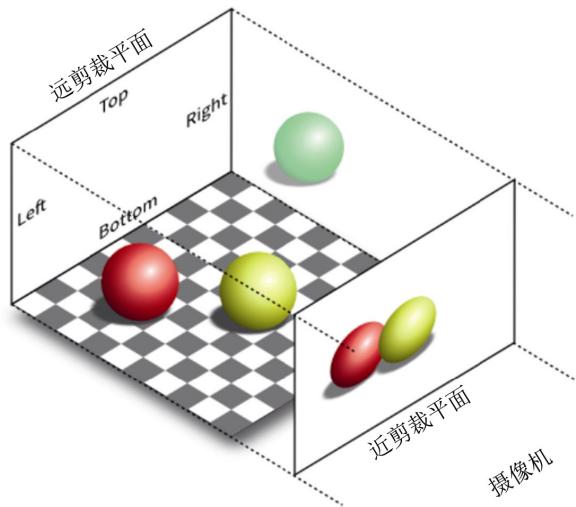


图 6-7 正交投影，图片来自 Nicolas P. Rougier, ERSF Code camp.

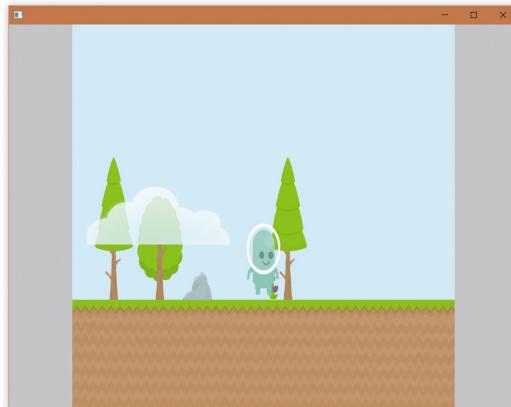


图 6-8 用投影矩阵渲染的场景。画面看起来很不舒服，不是吗？

## 第 7 章

# 第一个 3D 项目

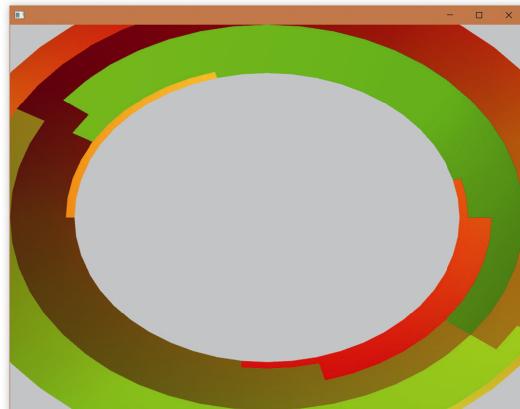


图 7-1 3D 项目现在的效果

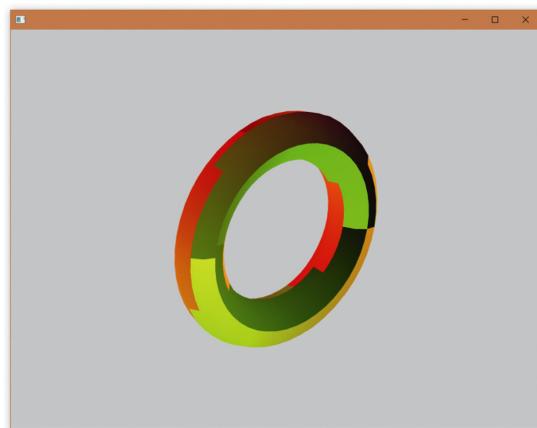


图 7-2 使用正交投影渲染的圆环

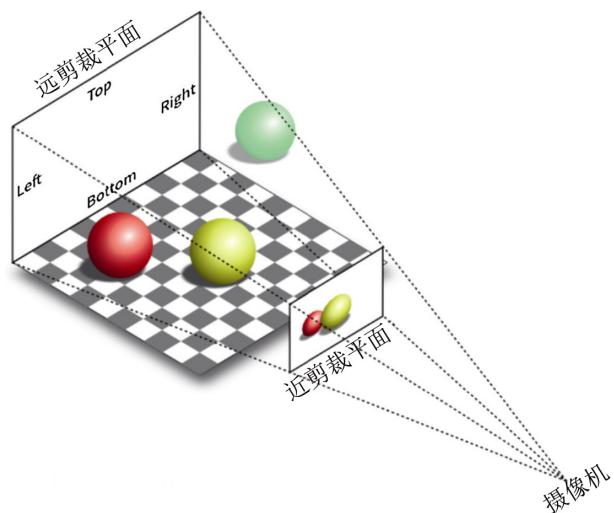


图 7-3 透视投影视锥体，图片来自 Nicolas P. Rougier, ERSF Code camp

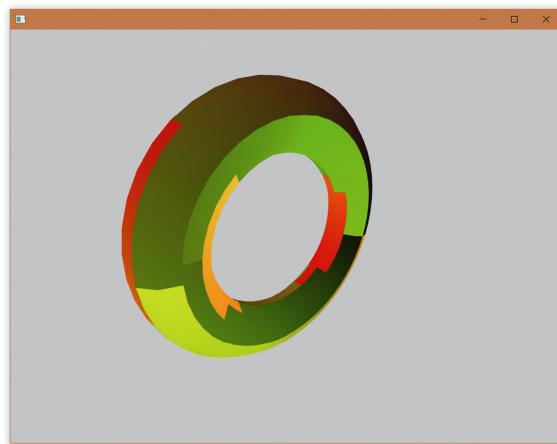


图 7-4 具有透视投影的三维程序

## 第 8 章

# 漫反射光照

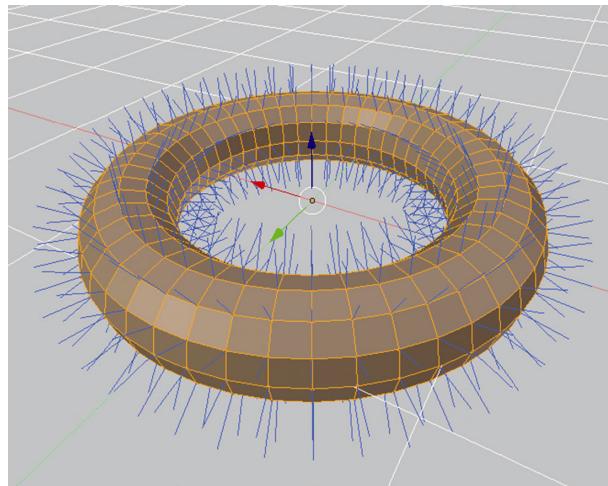


图 8-1 圆环网格的法线——蓝色线段显示了每个顶点的法线

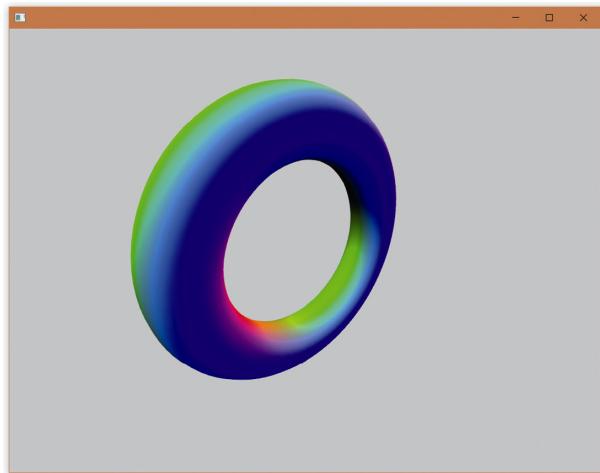


图 8-2 使用 mesh.vert 和 normal\_vis.frag 可视化圆环网格上的法线

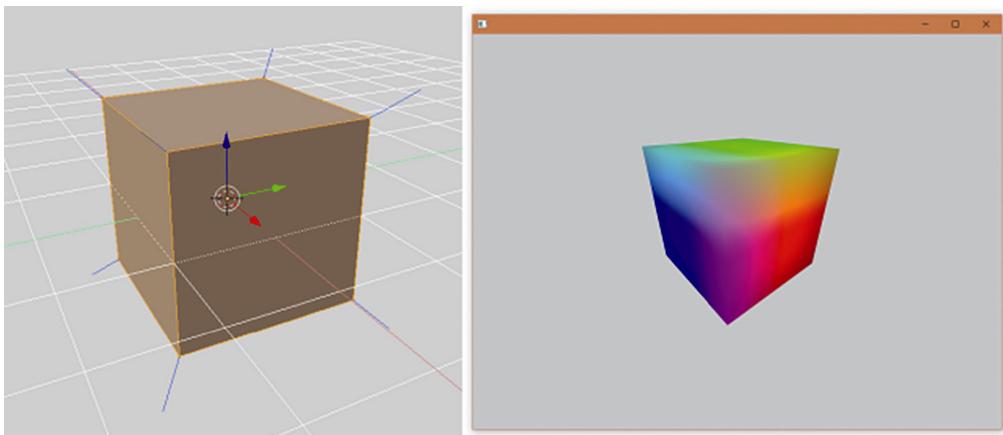


图 8-3 立方体网格上的法线。左图显示建模程序中的立方体网格，并显示了法线。

右图是用 normal\_vis 着色器渲染的同一个立方体

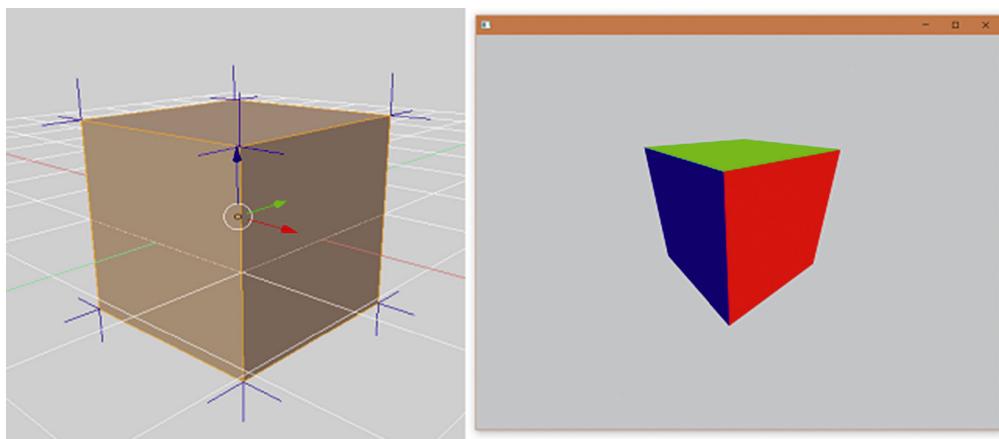


图 8-4 立方体网格上的平面法线。左图显示建模程序中具有可见法线的立方体网格。  
右图是用 `normal_vis` 着色器渲染的同一个立方体

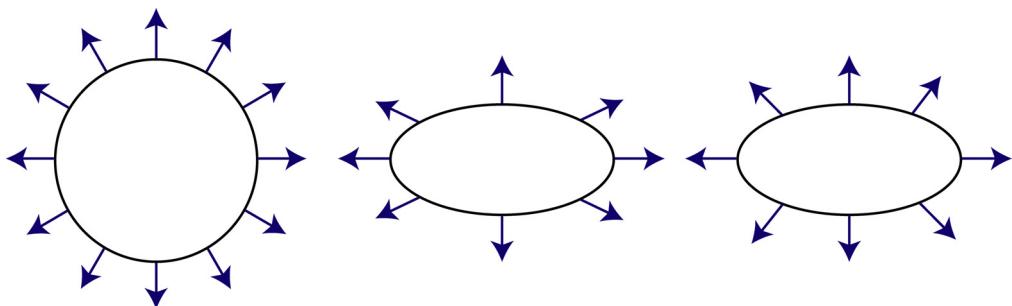


图 8-5 非等比缩放。中间球体使用模型矩阵变换其法线，而右侧球体使用法线矩阵

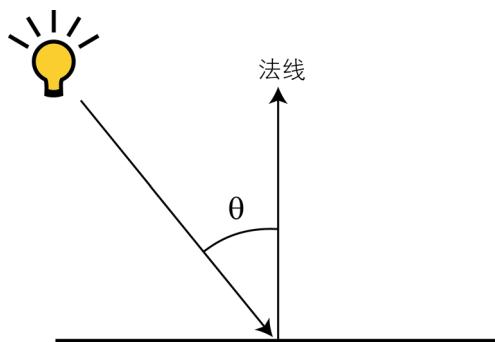


图 8-6 光线照射片元

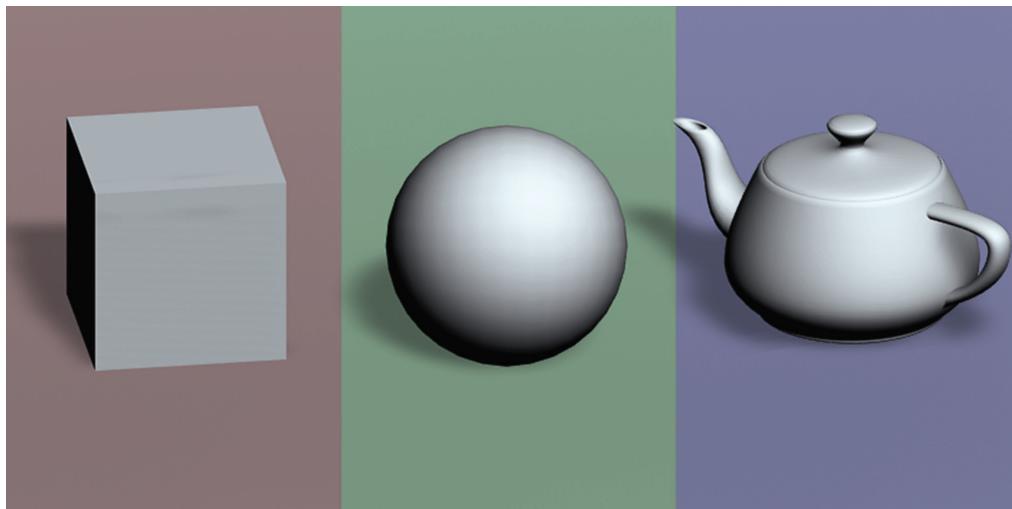


图 8-7 使用漫反射光照渲染的三个网格

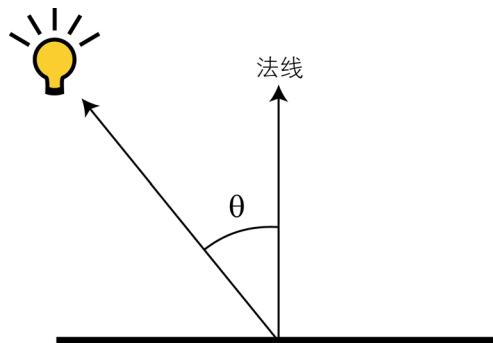


图 8-8 如何将灯光数据发送到着色器代码。请注意，灯光方向已反转，现在指向光源

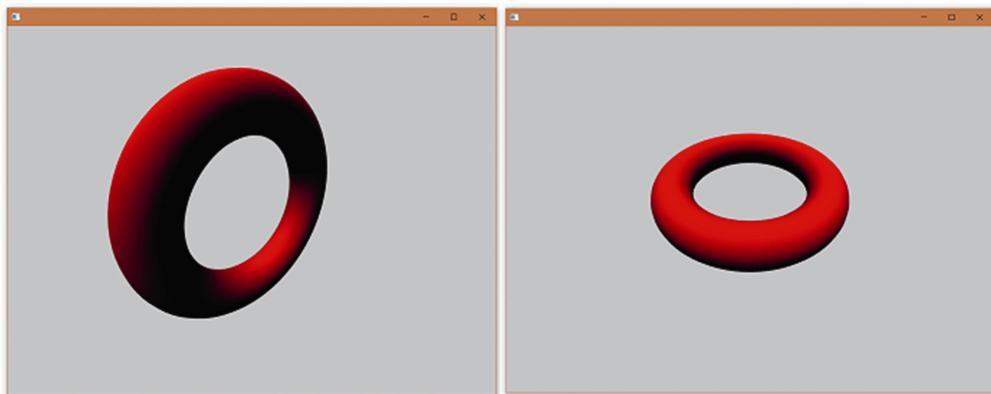


图 8-9 圆环网格上的漫反射光照

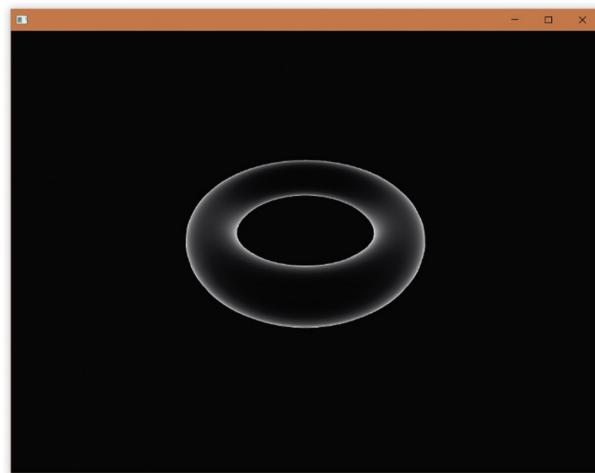


图 8-10 使用轮廓光渲染圆环

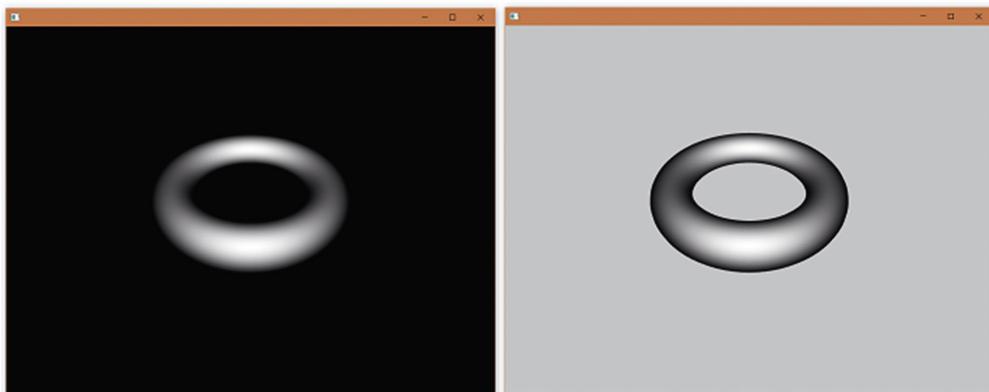


图 8-11 如果忘记反转点积值，轮廓光照的效果。如果你的背景是黑色的，如左图所示，这使网格看起来像幽灵一样



图 8-12 如果将 `rimAmt` 变量提高到不同的幂次，那么 `rim` 着色器会是什么效果。从左到右，这些截图展示了不使用 `pow()`、提升到 2 次方，以及提升到 4 次方的渲染效果

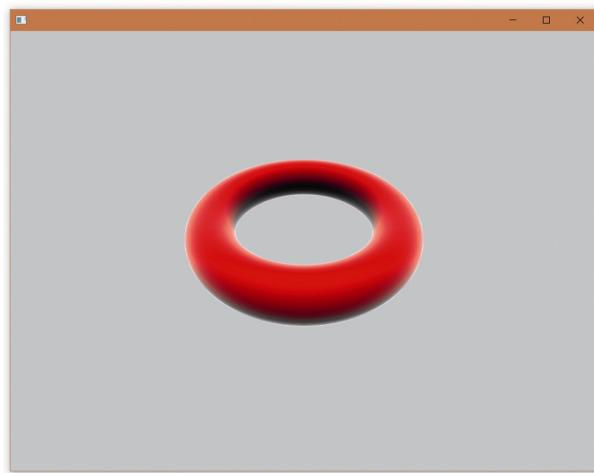


图 8-13 使用白色轮廓光渲染的圆环网格

## 第 9 章

# 第一个光照模型

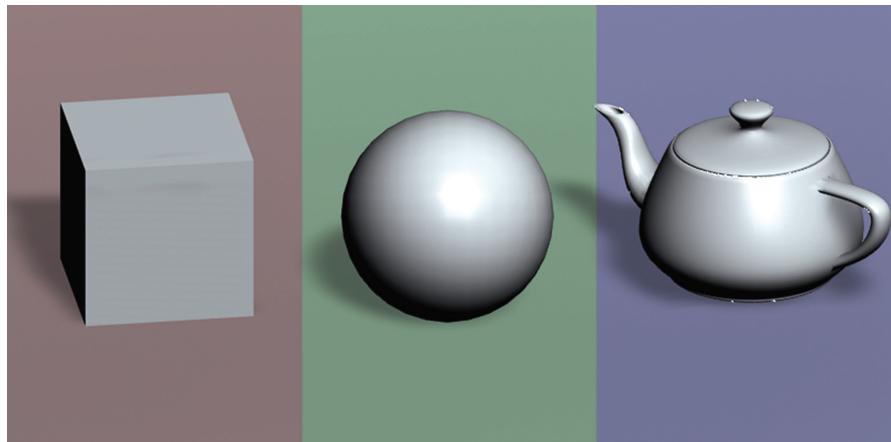


图 9-1 三个不同示例网格上的镜面反射光照

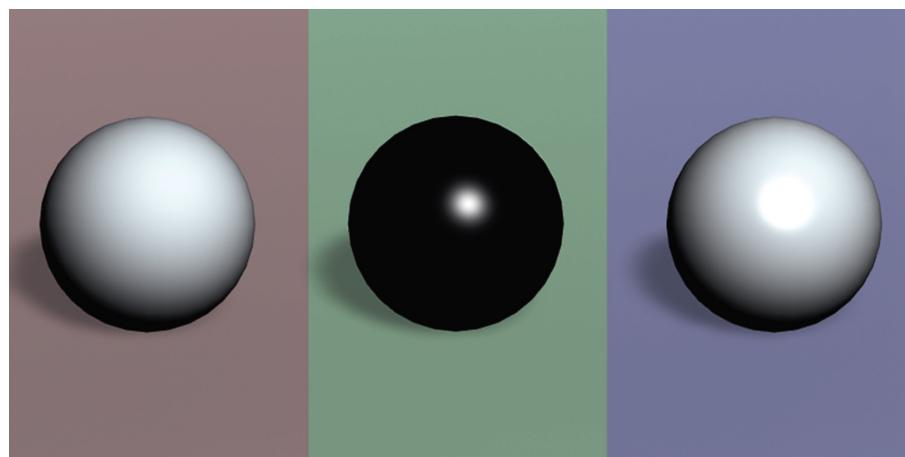


图 9-2 左侧球体仅使用漫反射光照，中心球体仅使用高光光照，右侧球体则结合了两种类型的光照

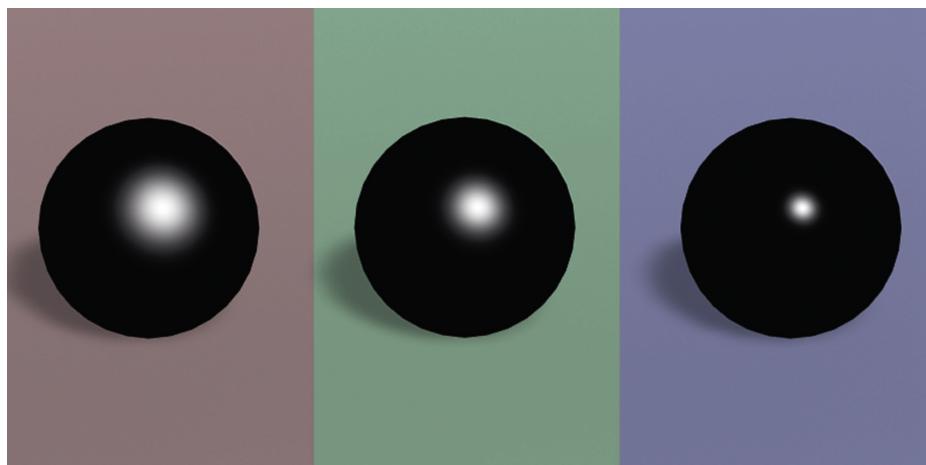


图 9-3 通过将 toCam 和反射向量的点积提高到 4 次方、16 次方和 32 次方(从左到右)来得到具有不同尺寸光斑的球体

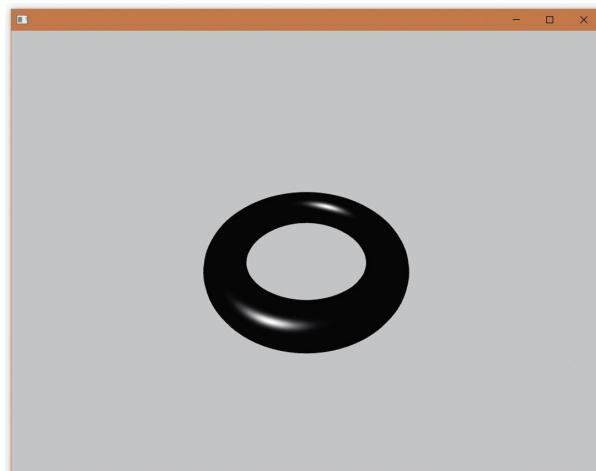


图 9-4 使用高光着色器的圆环网格

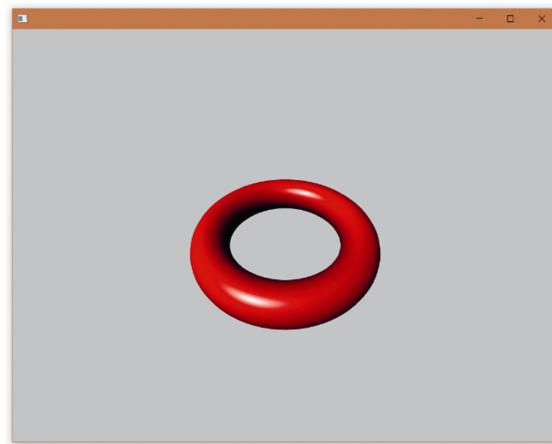


图 9-5 使用漫反射和镜面光照渲染的圆环

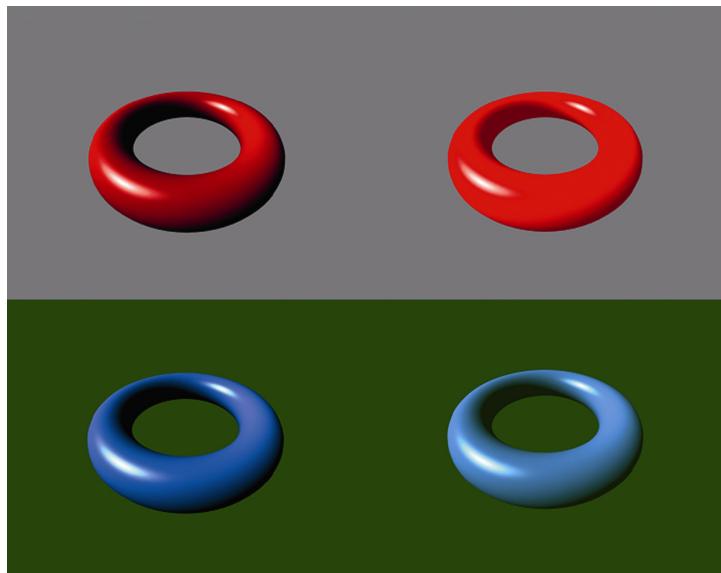


图 9-6 使用环境光照。左列的两张图是没有环境光照的网格效果。

右列中两张图的网格使用了与画面背景颜色相同的环境光

$$I_p = k_a i_a + \sum_{m \in \text{lights}} (k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{R}_m \cdot \hat{V})^\alpha i_{m,s}).$$

图 9-7 以数学符号表示的 Phong 光照模型

$$I_p = k_a i_a + \sum_{m \in \text{lights}} (k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{R}_m \cdot \hat{V})^\alpha i_{m,s}).$$

A      B      C      D

图 9-8 划分成几部分的 Phong 光照方程



图 9-9 Phong 光照在处理低光泽指数时的镜面反射问题的示例

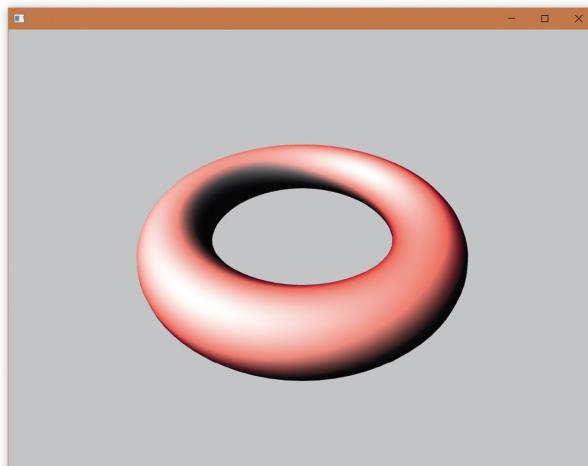


图 9-10 具有低光泽度以及使用 Blinn-Phong 镜面反射光照的圆环

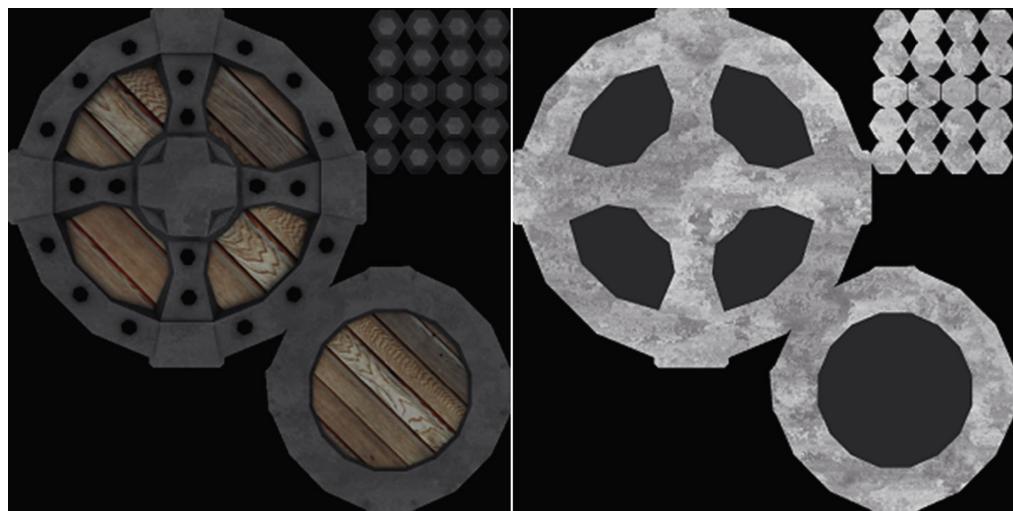


图 9-11 将用于渲染盾牌网格的两种纹理



图 9-12 使用漫反射和镜面映射渲染的盾牌网格

## 第 10 章

# 法线贴图

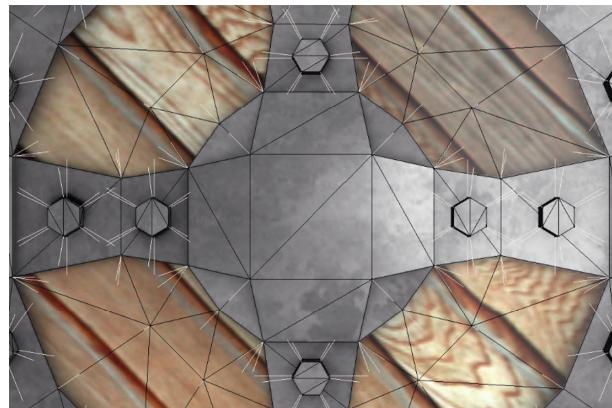


图 10-1 盾牌网格的中心，显示了法线向量。请注意，中心金属件的最大截面仅使用四个顶点建模

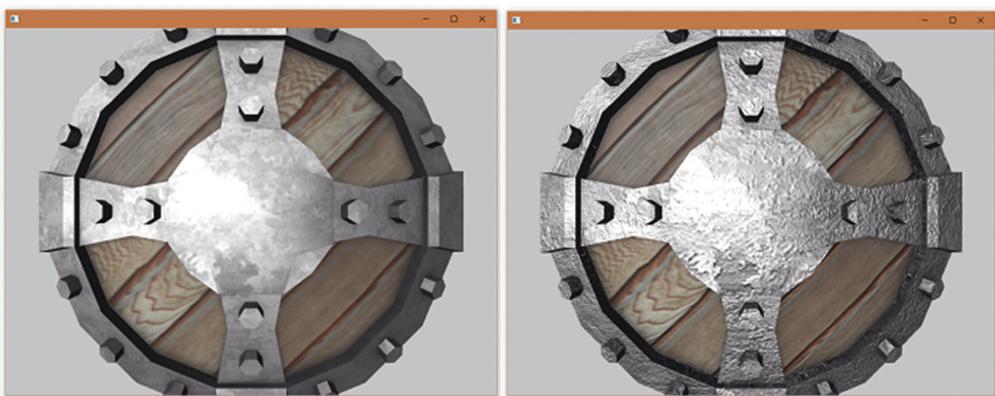


图 10-2 没有法线贴图(左)，和有法线贴图(右)的盾牌

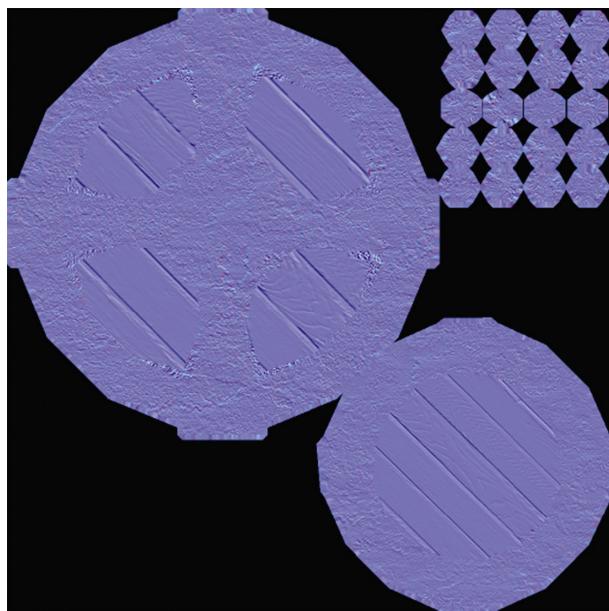


图 10-3 用于盾牌的法线纹理

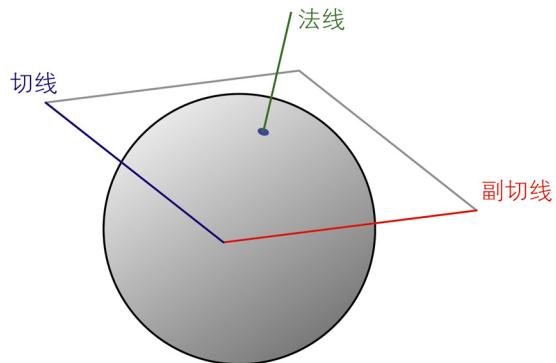


图 10-4 球面网格上一点的切线空间轴向量示意图

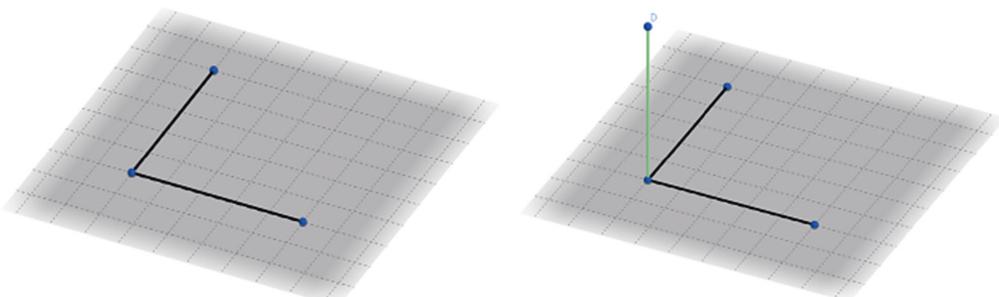


图 10-5 如果我们计算左图中两个向量的叉积, 结果就是右图中的绿色向量(竖直向上的向量)



图 10-6 水面效果

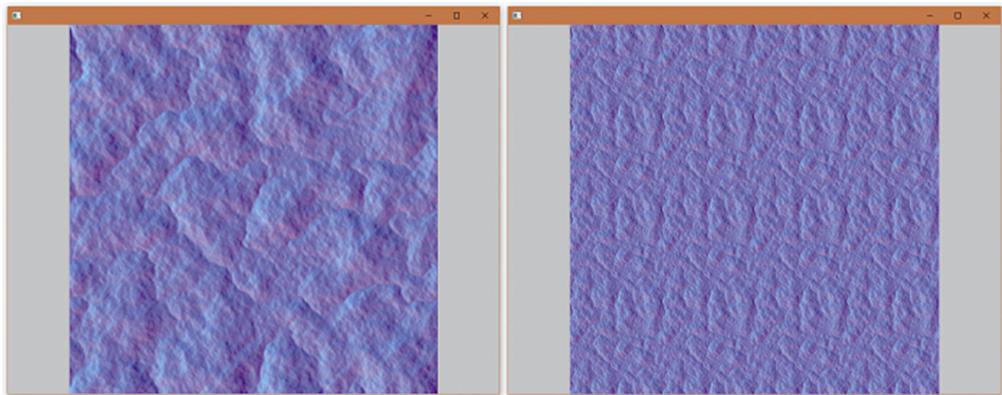


图 10-7 左图显示了使用法线贴图纹理的平面。右侧图像显示了相同的平面和法线贴图，但纹理的 UV 坐标乘以 4

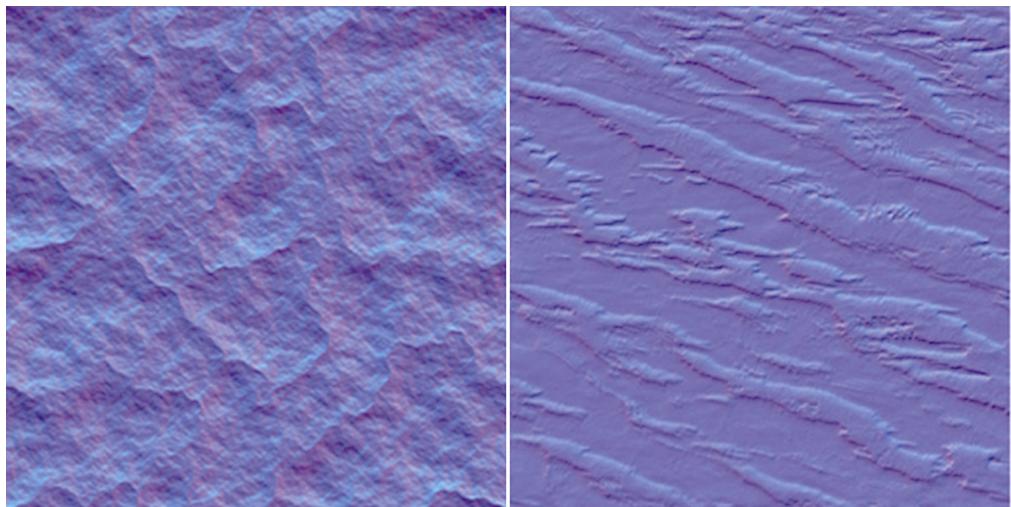


图 10-8 左边是正在用于水面的法线贴图。右边是具有很强的水平特征的法线贴图，当平铺在水面上时会非常明显

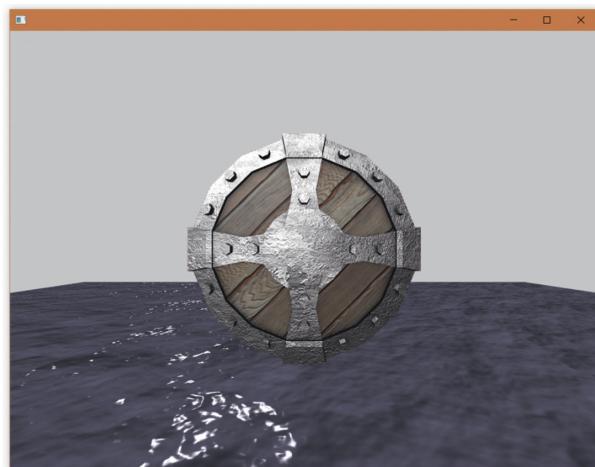


图 10-9 水面示例最终的渲染效果

## 第 11 章

# 立方体贴图和天空盒

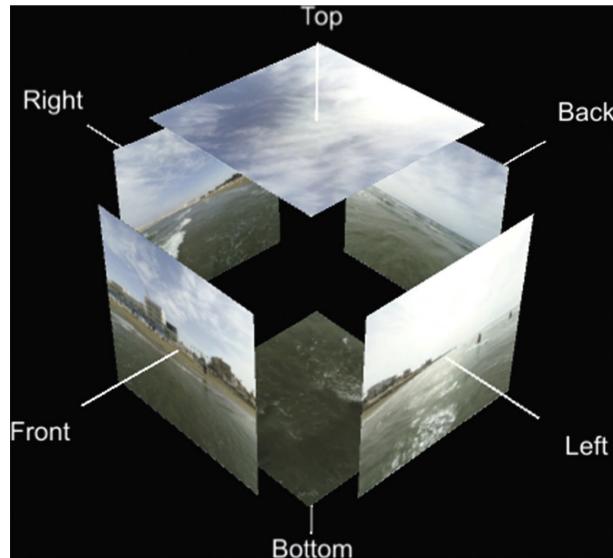


图 11-1 立方体贴图的面以立方体的形状摆放



图 11-2 在立方体上渲染的立方体贴图



图 11-3 使用天空盒渲染的场景

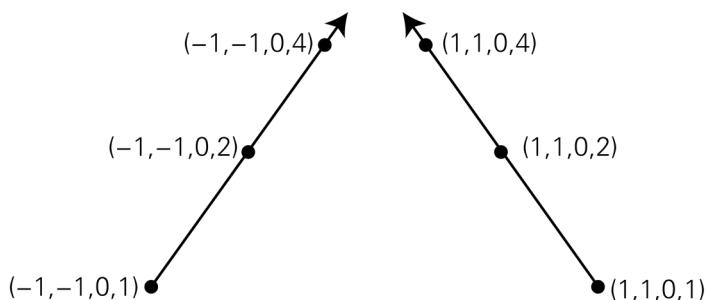


图 11-4 带有不同 W 分量的透视除法效果的示意图

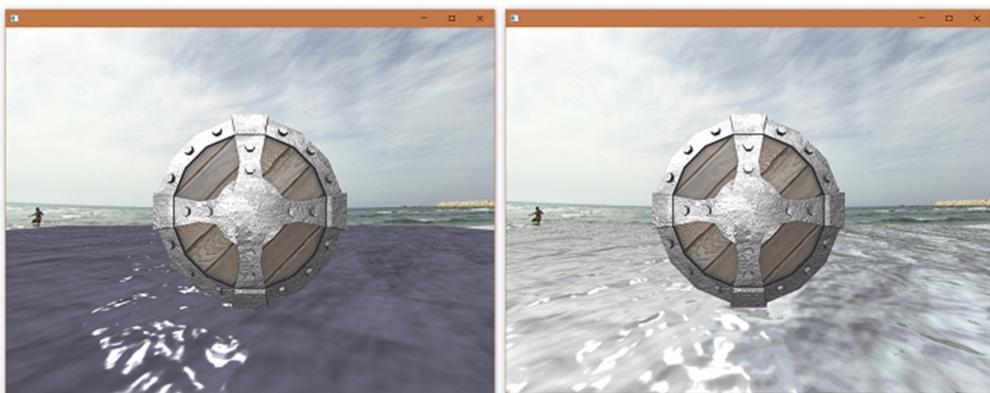


图 11-5 左边没有使用立方体贴图反射，右边使用了立方体贴图反射

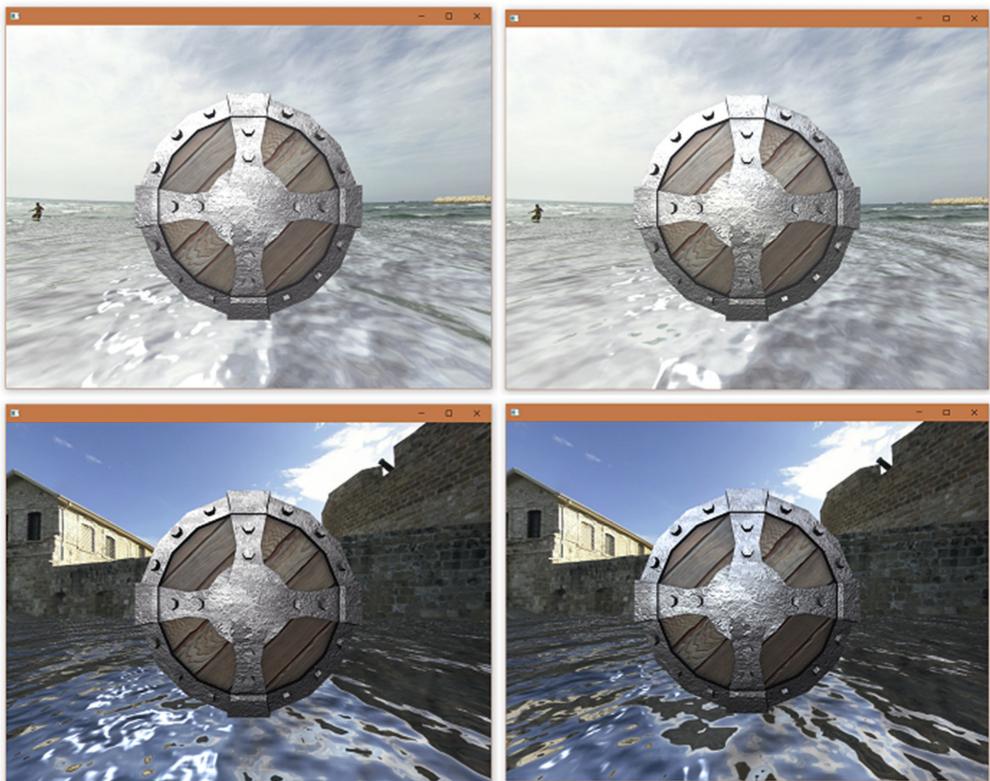


图 11-6 在盾牌上使用和不使用立方体贴图光照时渲染的场景(右面板使用了立方体贴图光照)

## 第 12 章

# 深入光照

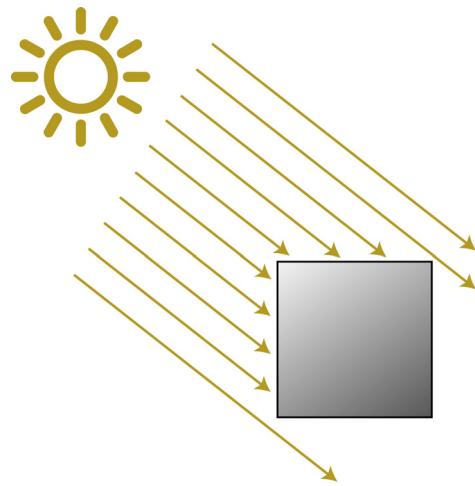


图 12-1 定向光源的平行光线

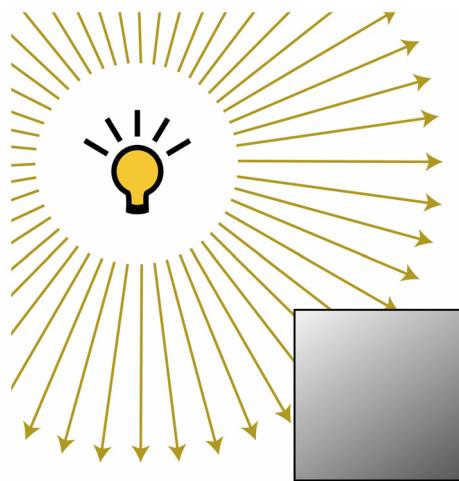


图 12-2 点光源的光线

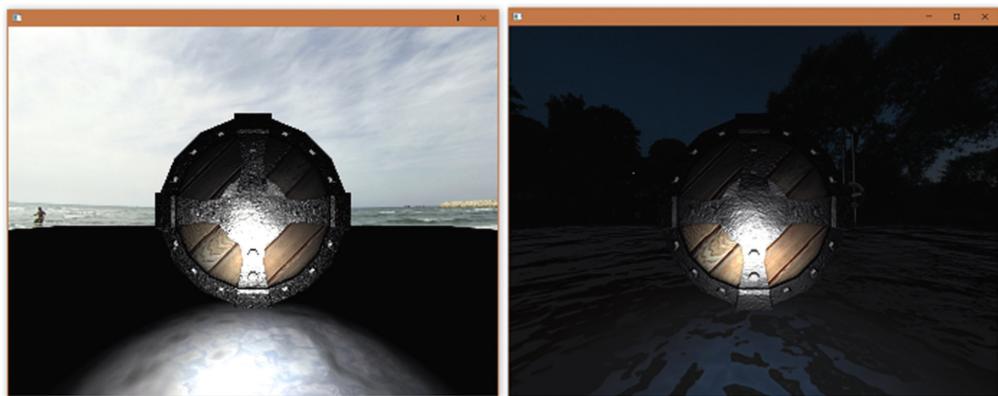


图 12-3 使用两个不同的天空盒渲染点光源示例。请注意，由于水面颜色是基于天空盒计算的，因此较暗的天空盒也意味着水的颜色更暗，即使是在光照区域

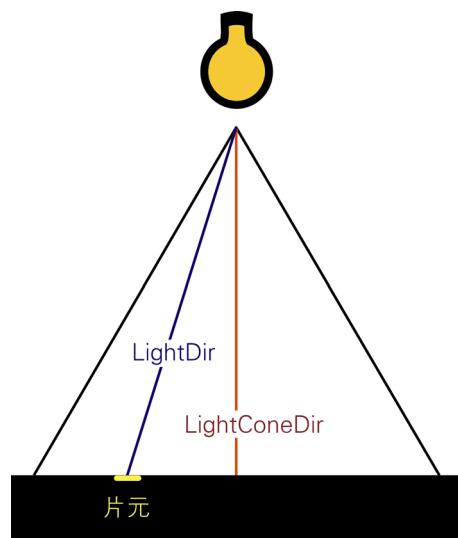


图 12-5 图解聚光灯数学问题

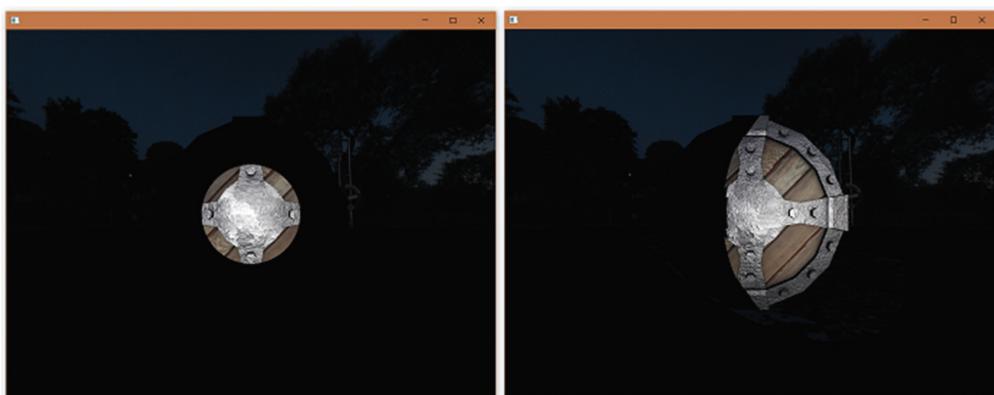


图 12-6 使用两个不同设置的聚光灯渲染的场景

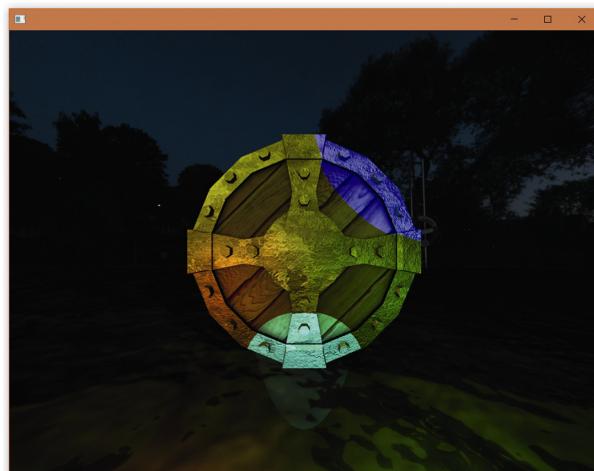
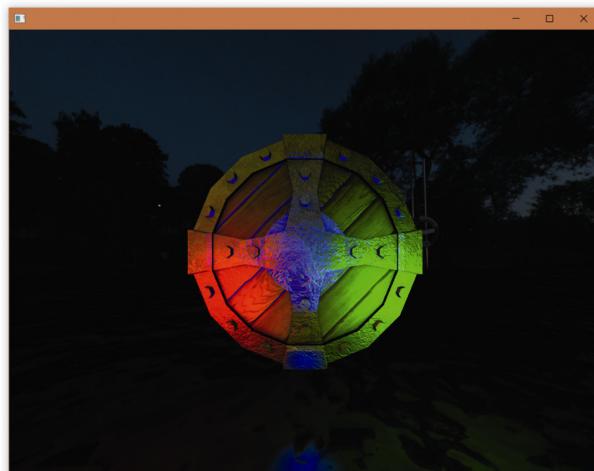


图 12-7 用五盏灯渲染的场景。请注意红色和绿色的点光源、蓝色和青色的聚光灯以及黄色的平行光







## 第 13 章

# 剖析着色器性能



图 13-1 模拟屏幕撕裂(感谢维基百科的 VanessaEzekowitz)。撕裂点用红线(图中的两根小平线)标出

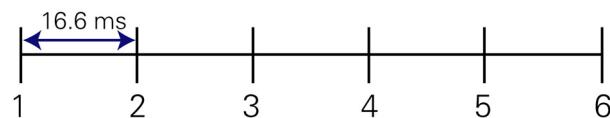


图 13-2 在时间轴上可视化显示屏幕刷新率

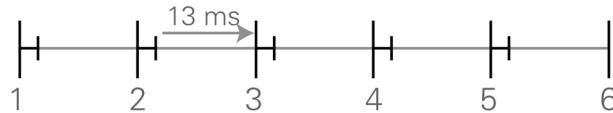


图 13-3 vsync 在渲染每帧 3.6 毫秒的应用程序中引入的延迟的示意图

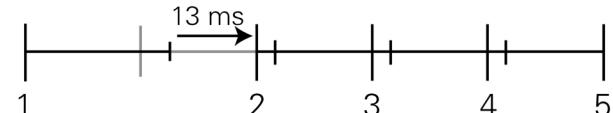


图 13-4 如果应用程序错过刷新间隔, 为什么 vsync 会导致帧率非常低

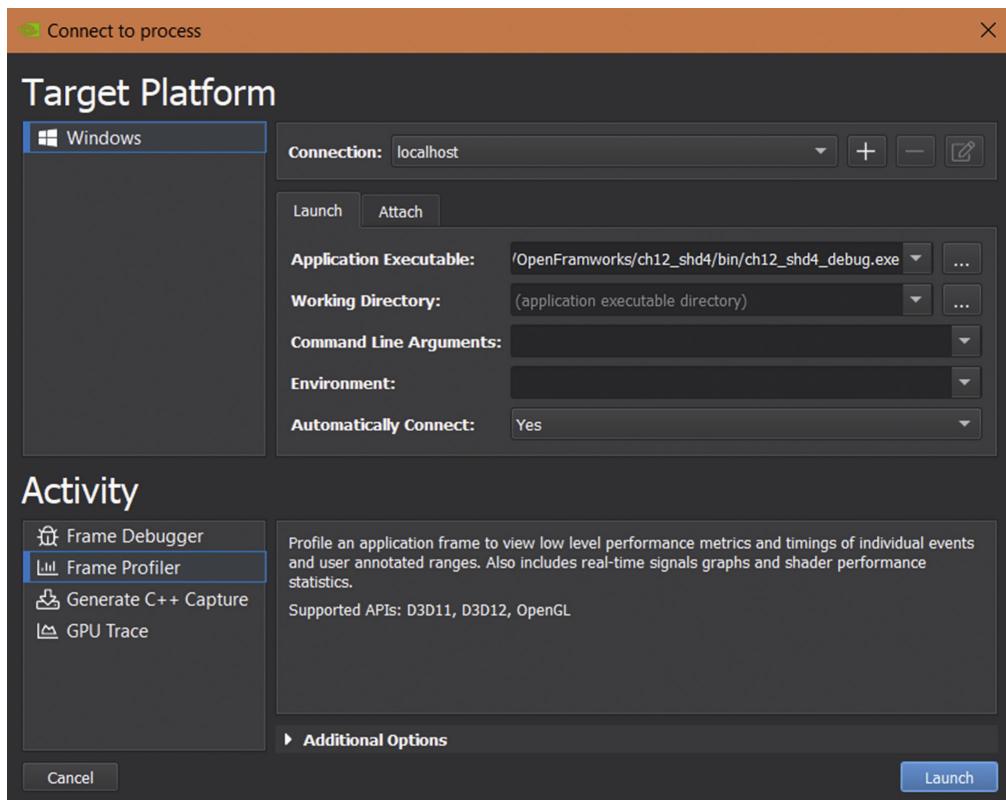


图 13-5 Nsight Graphics 的 Connect to Process 对话框

## Shader 开发实战

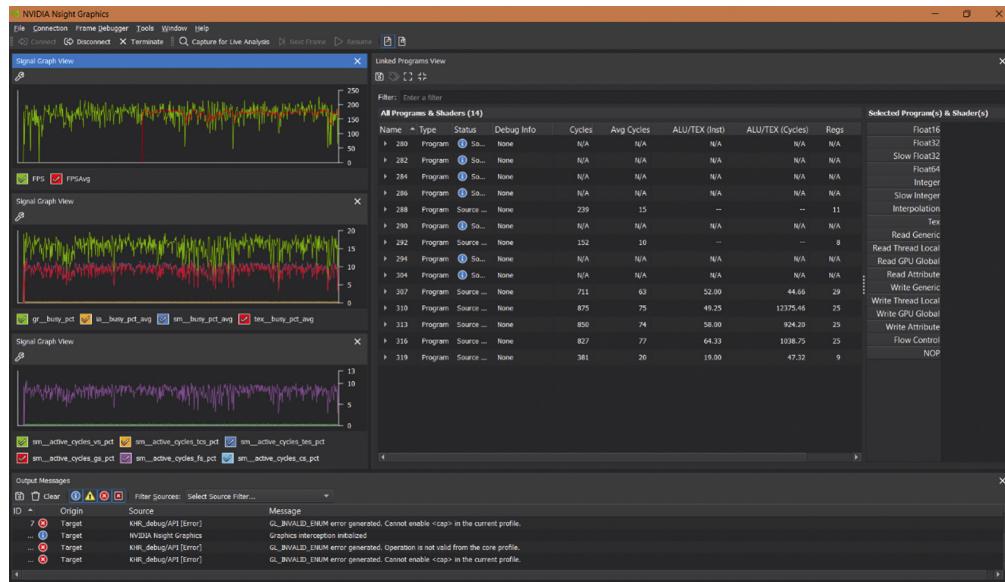


图 13-6 Nsight Graphics 的 frame profiler 视图

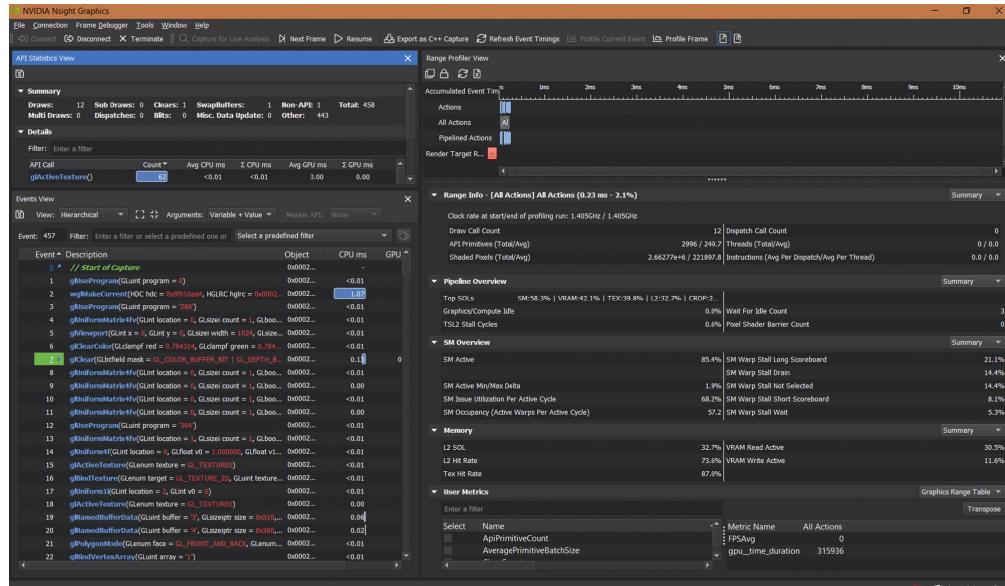


图 13-7 Nsight 帧调试视图

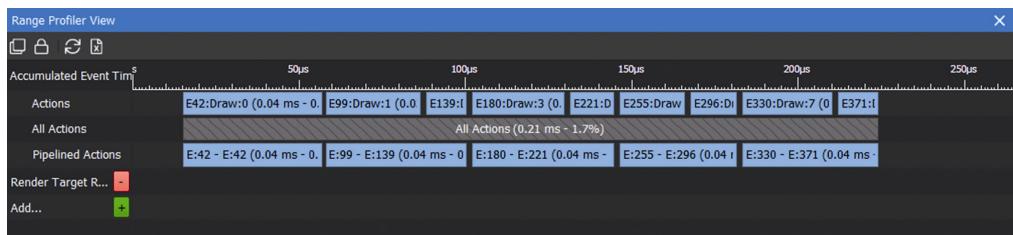


图 13-8 放大 Range Profiler View 中的事件



图 13-9 Nsight 图形回放视图

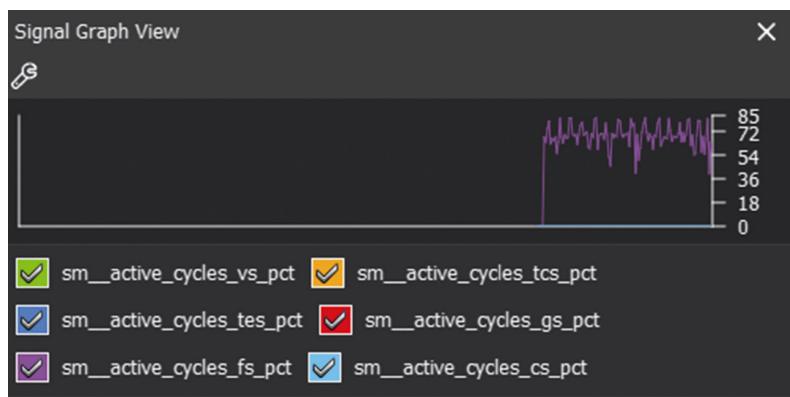


图 13-10 片元着色器百分比太高，以至于甚至看不到顶点着色器的 GPU 时间百分比

sm\_active\_cycles\_vs\_pct

## Shader 开发实战

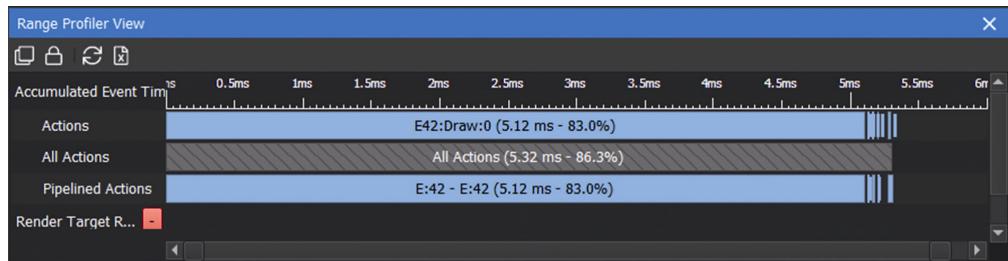


图 13-11 Range Profiler View 的屏幕截图

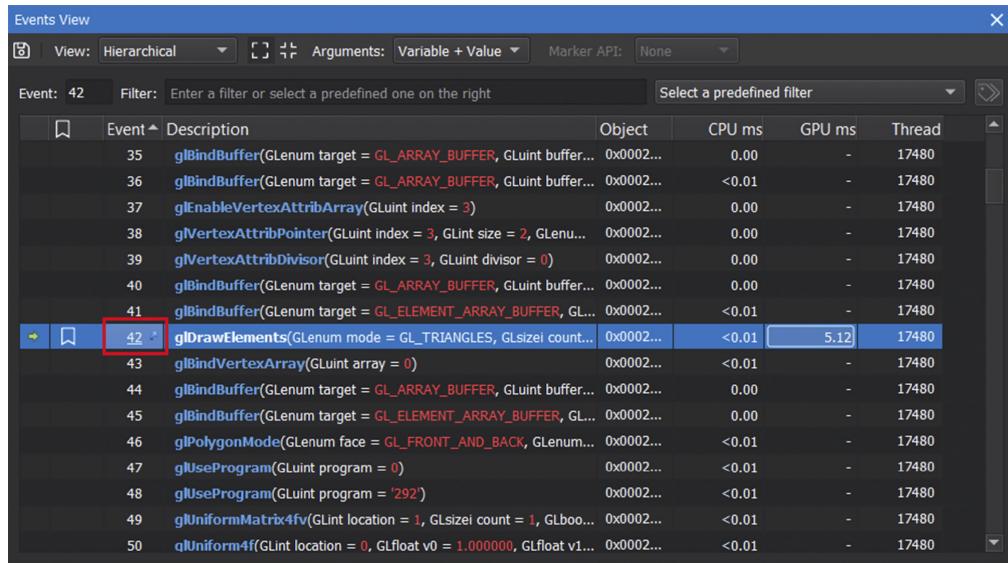


图 13-12 Nsight 的帧调试器视图的事件视图

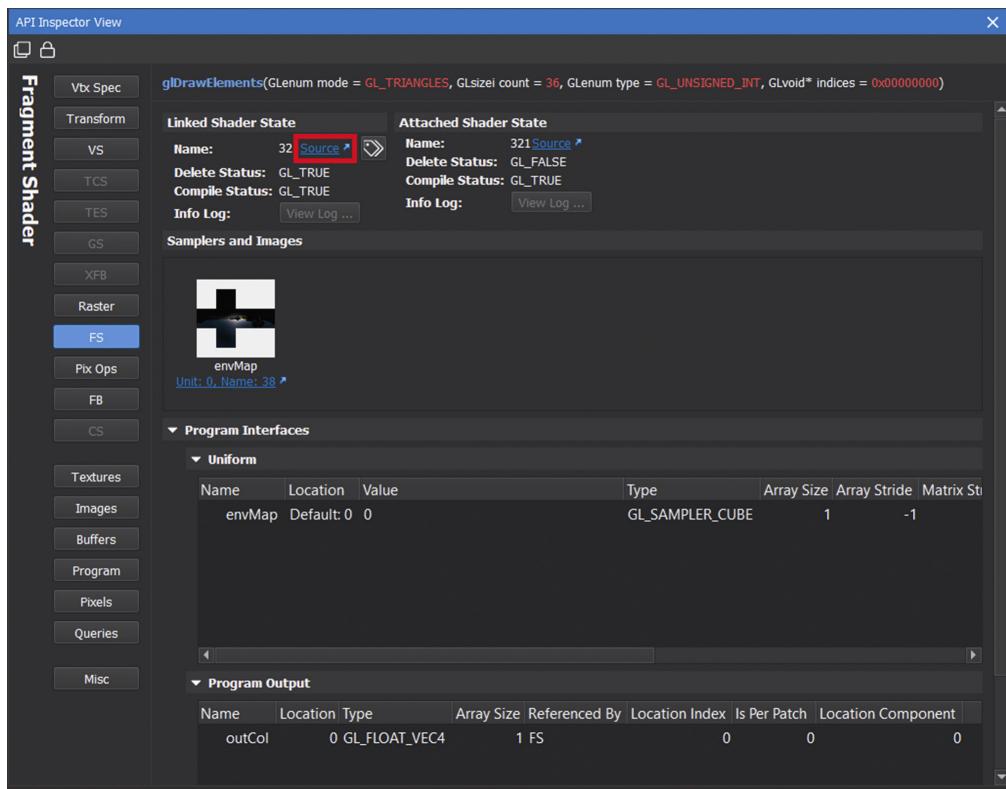


图 13-13 API Inspector View 的片元着色器部分

## 第 14 章

# 优化着色器

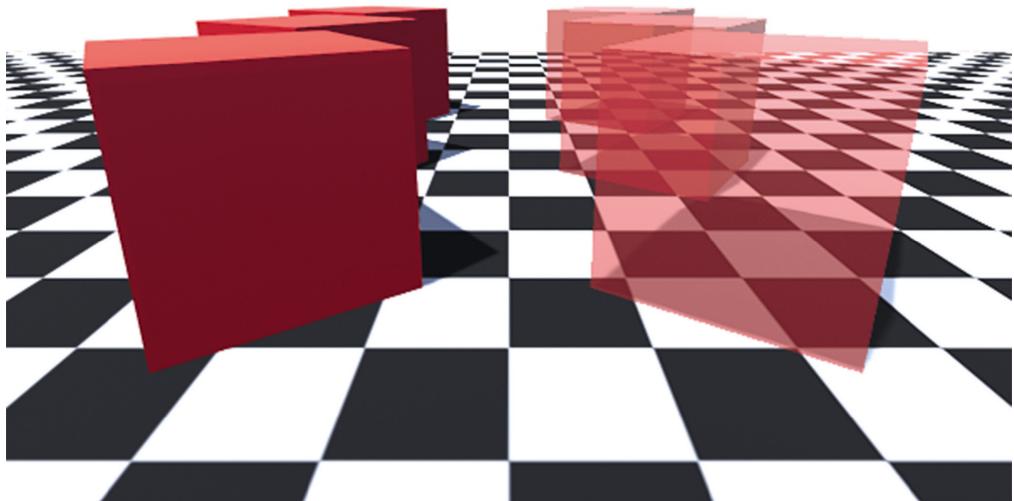


图 14-1 不透明与半透明物体

## 第 15 章

# 精 度

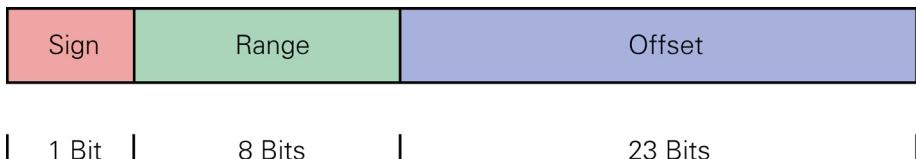


图 15-1 32 位浮点数的内存布局

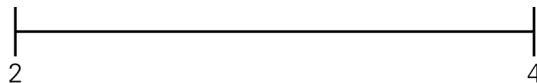


图 15-2 浮点数范围显示在数轴上

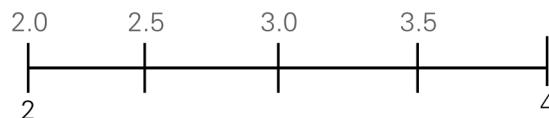


图 15-3 2 个“偏移”位可以表示的值，显示在图 15-2 的数轴上方

值	计算的过程
2	$2^1 + (\frac{0}{4} * 2)$
2.5	$2^1 + (\frac{1}{4} * 2)$
3.0	$2^1 + (\frac{2}{4} * 2)$
3.5	$2^1 + (\frac{3}{4} * 2)$

图 15-4 计算出四个可能值的过程

1024	$2^{10} + (\frac{0}{4} * 1024)$
1280	$2^{10} + (\frac{1}{4} * 1024)$
1536	$2^{10} + (\frac{2}{4} * 1024)$
1792	$2^{10} + (\frac{3}{4} * 1024)$

图 15-5 如果指数设置为 10，可以表示的值

2	$2^1 + (\frac{0}{8388608} * 2)$
2.00000023842	$2^1 + (\frac{1}{8388608} * 2)$
2.00000047684	$2^1 + (\frac{2}{8388608} * 2)$
2.00000071526	$2^1 + (\frac{3}{8388608} * 2)$

图 15-6 可以用 23 个“偏移”位来表示的前四个可能的值

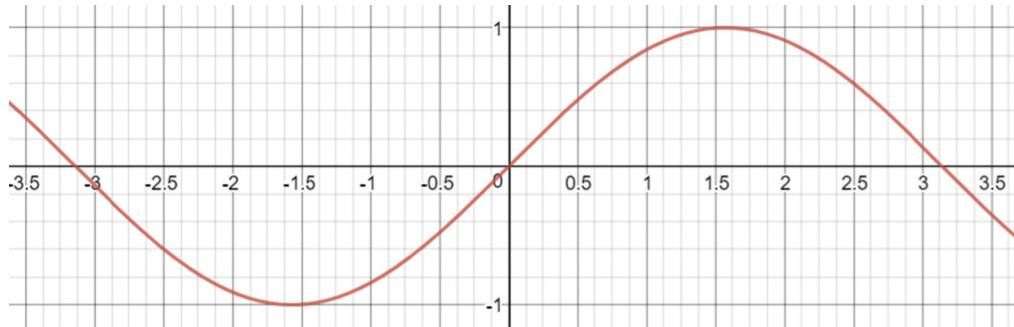


图 15-7 正弦曲线的图形， $\sin()$ 函数的输出。此曲线沿 X 轴无限重复

数据类型	范围位数	偏移位数	范围
<b>“highp” or “full precision”</b>	8	23	$-2^{126}, 2^{127}$
<b>“mediump” or “half precision”</b>	5	10	$-2^{14}, 2^{14}$
<b>“lowp” or “fixed precision”</b>	1	8	$-2^1, 2^1$

图 15-8 三种不同类型的浮点数变量

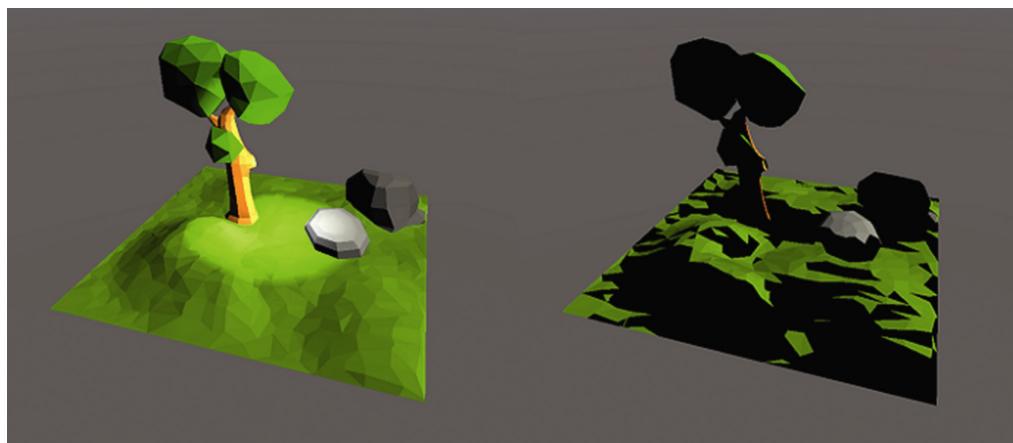


图 15-9 场景在点光源附近时的渲染效果，以及场景离点光源非常远时的渲染效果

## 第 16 章

# 在 Unity 中编写着色器

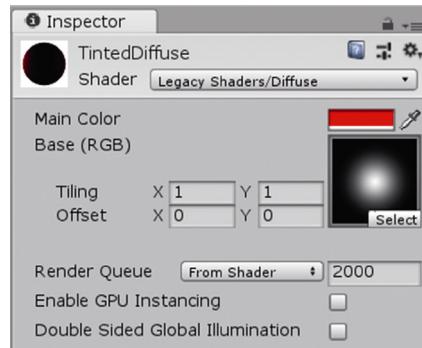


图 16-1 具有许多可修改属性的材质

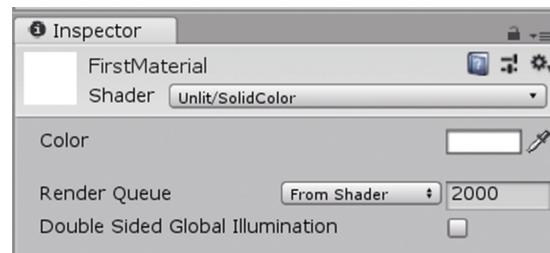


图 16-2 材质的 Unity 编辑器图形用户界面。注意，统一变量有一个颜色选择器，  
因为我们指定了它将存储颜色数据

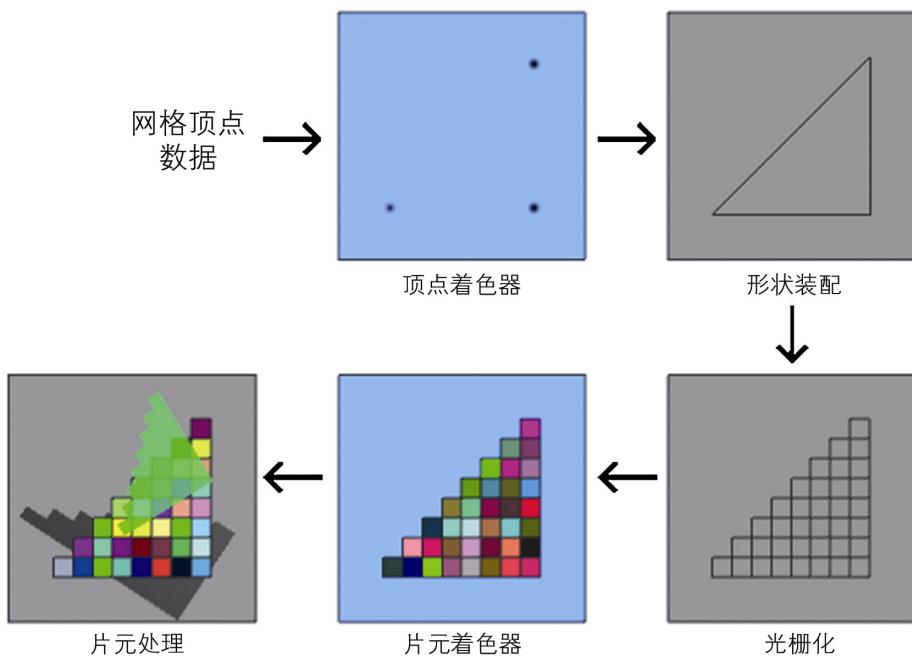


图 16-3 简化了的图形管线图

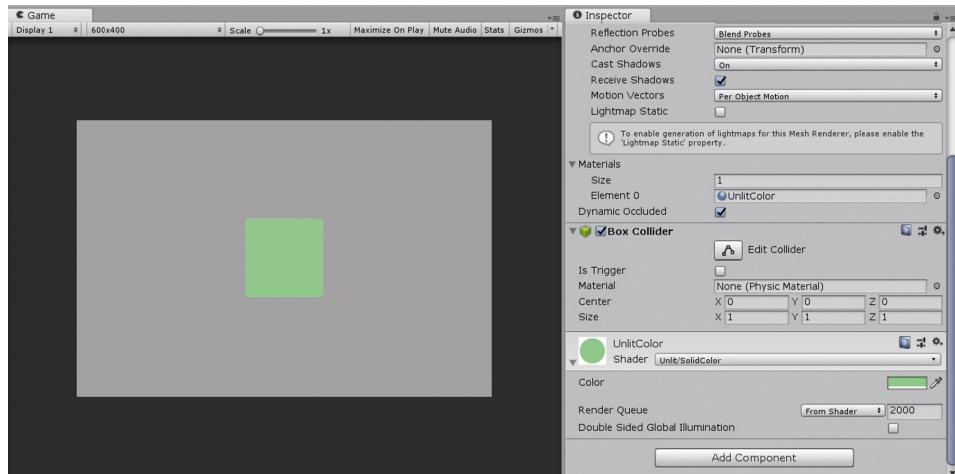


图 16-4 编辑器的屏幕截图，其中的网格添加了 Unlit/Color 材质

$$\text{transpose}\left(\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}\right) = \begin{bmatrix} 0 & 3 & 6 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \end{bmatrix}$$

图 16-5 transpose()函数的作用

## Shader 开发实战

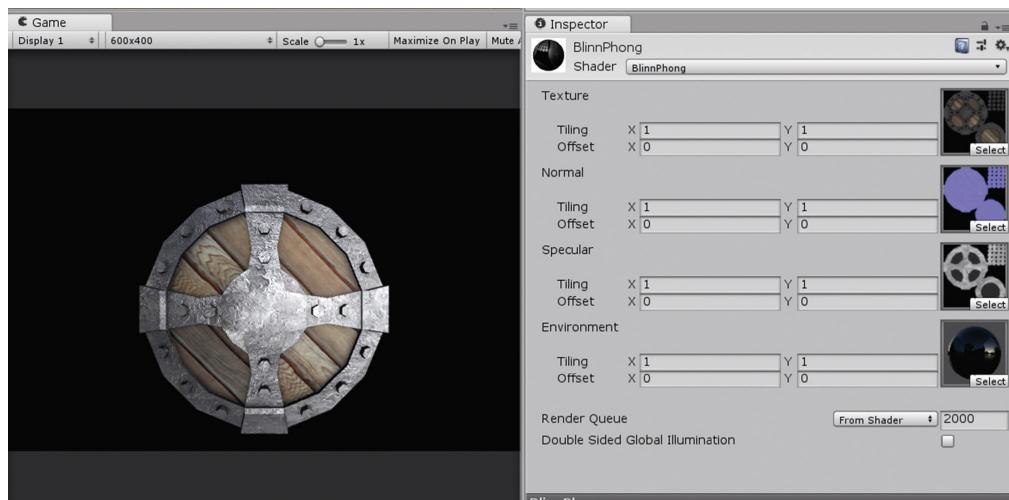


图 16-6 使用 Blinn-Phong 着色器在 Unity 中渲染盾牌网格

## 第 17 章

# 在 UE4 中编写着色器

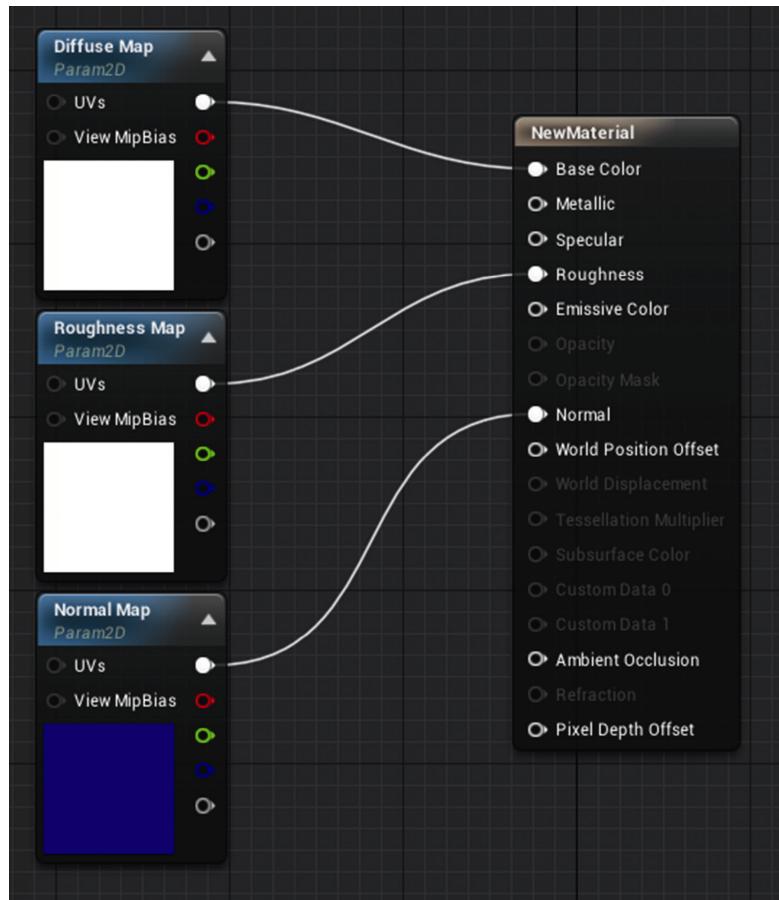


图 17-1 UE4 的材质编辑器中的节点图

## Shader 开发实战

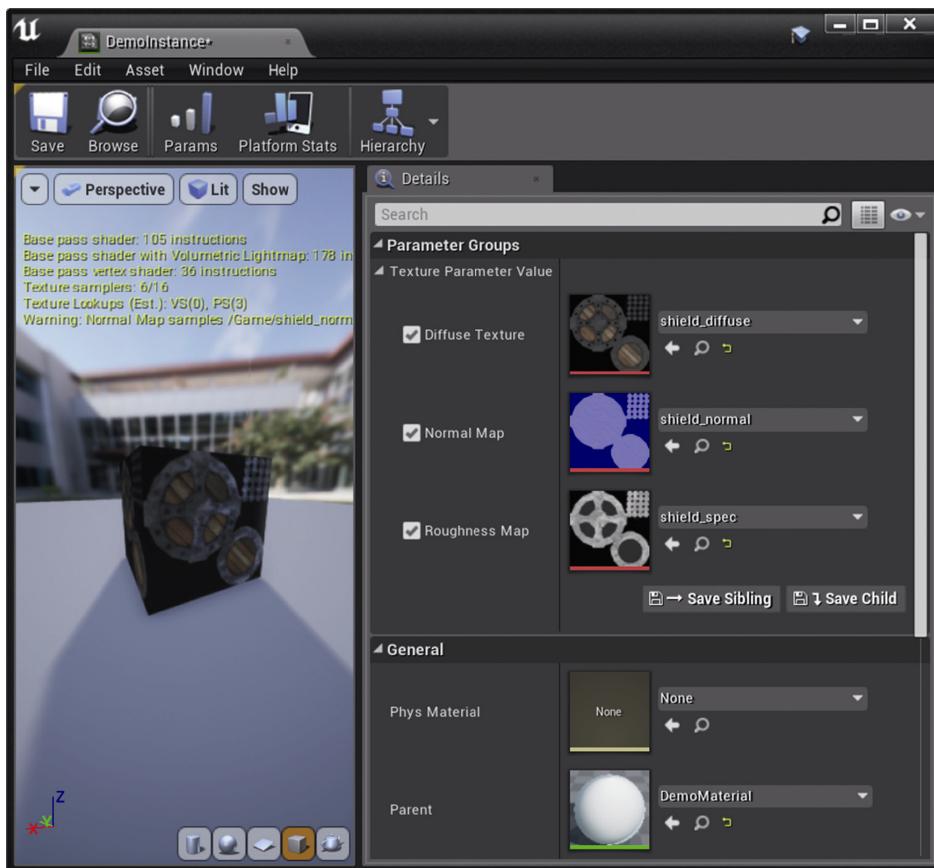


图 17-2 使用盾牌纹理的材质实例

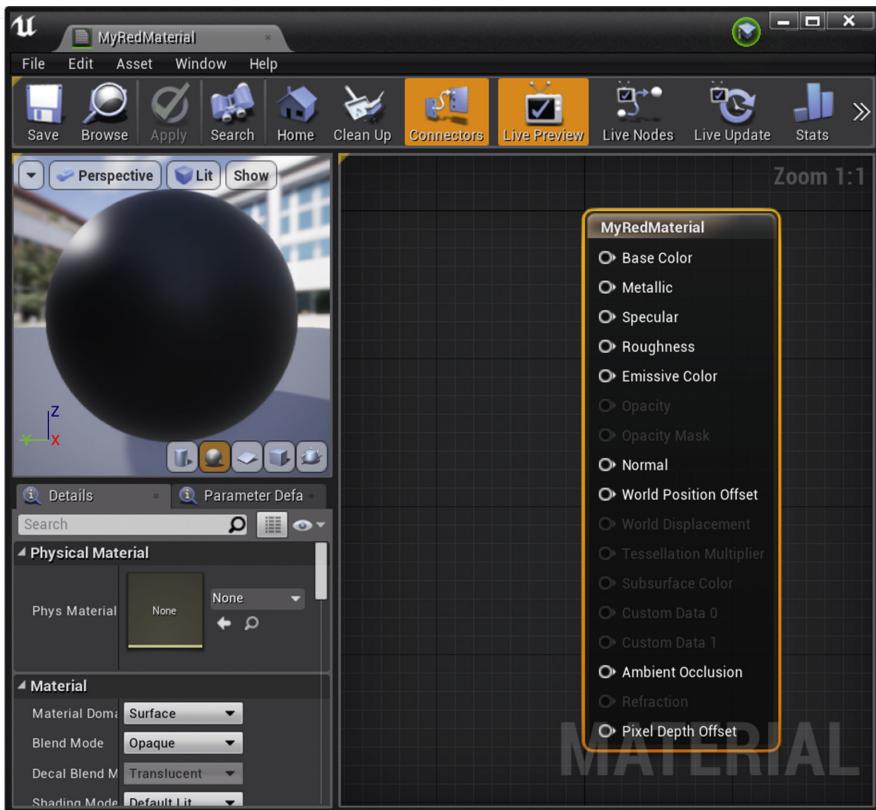


图 17-3 UE4 材质编辑器窗口

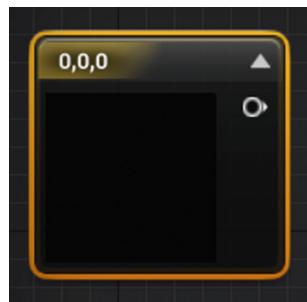


图 17-4 Constant3Vector 材质节点

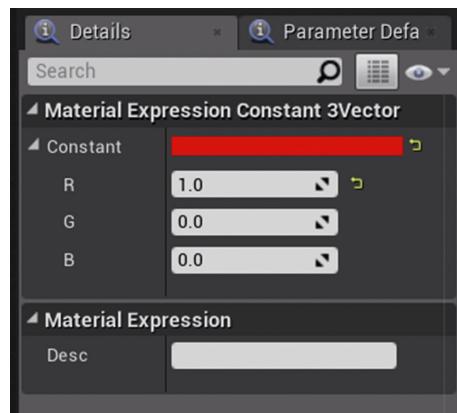


图 17-5 节点的 Details 面板



图 17-6 点积和乘法材质节点

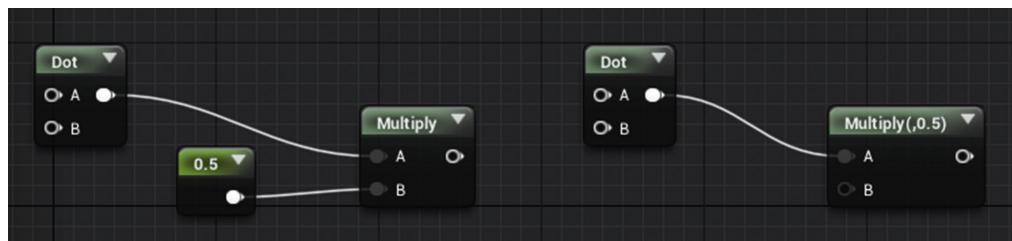


图 17-7 使用乘法节点的两个例子。左侧，第二个值由常量节点提供；右侧，该值已在节点本身上设置



图 17-8 Texture Sample 节点

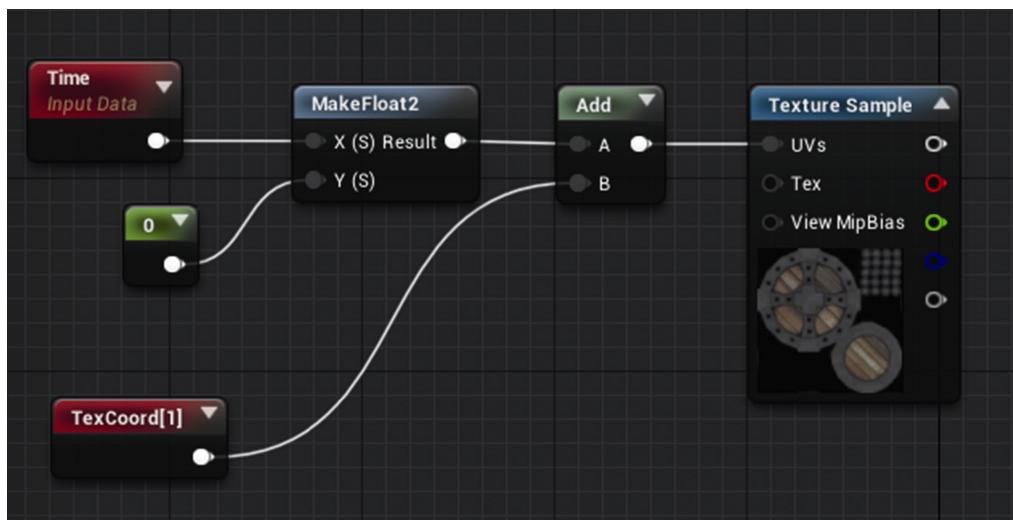


图 17-9 向右滚动 UV 坐标的节点图

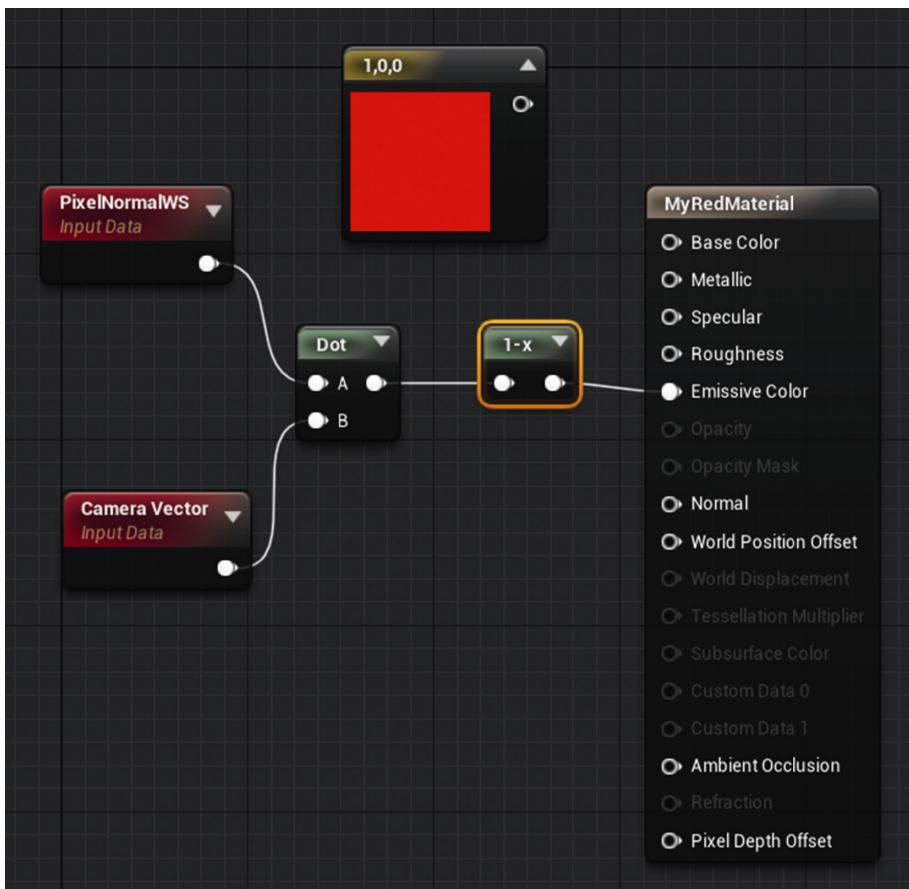


图 17-10 到目前为止我们的边缘光材质

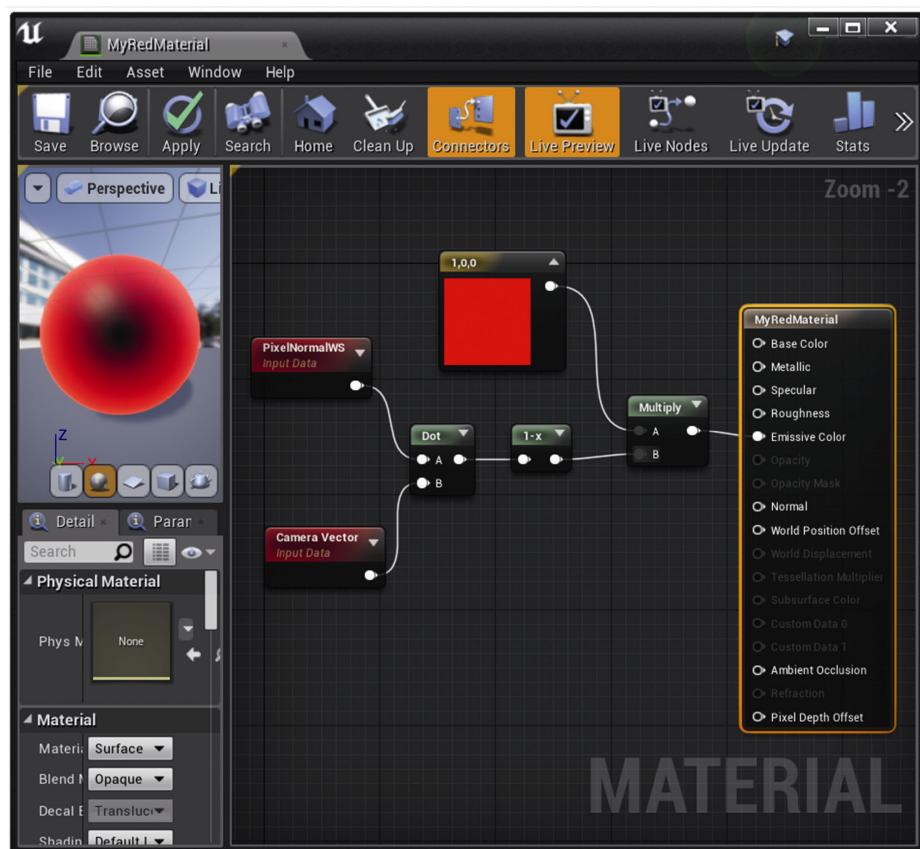


图 17-11 完成的边缘光材质

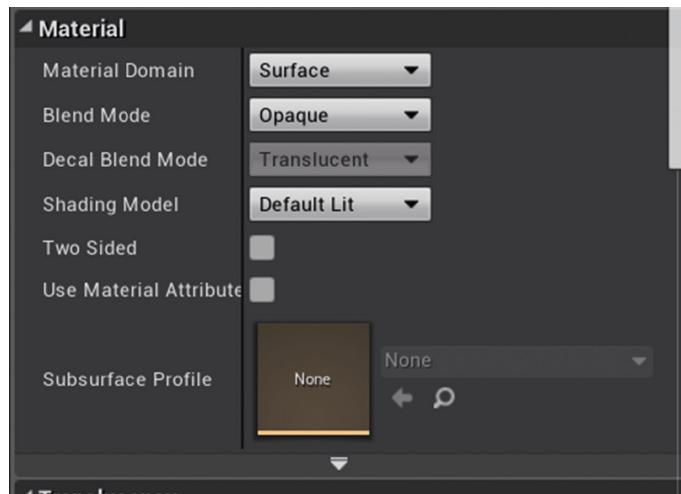


图 17-12 “详细信息”面板的 Material 部分

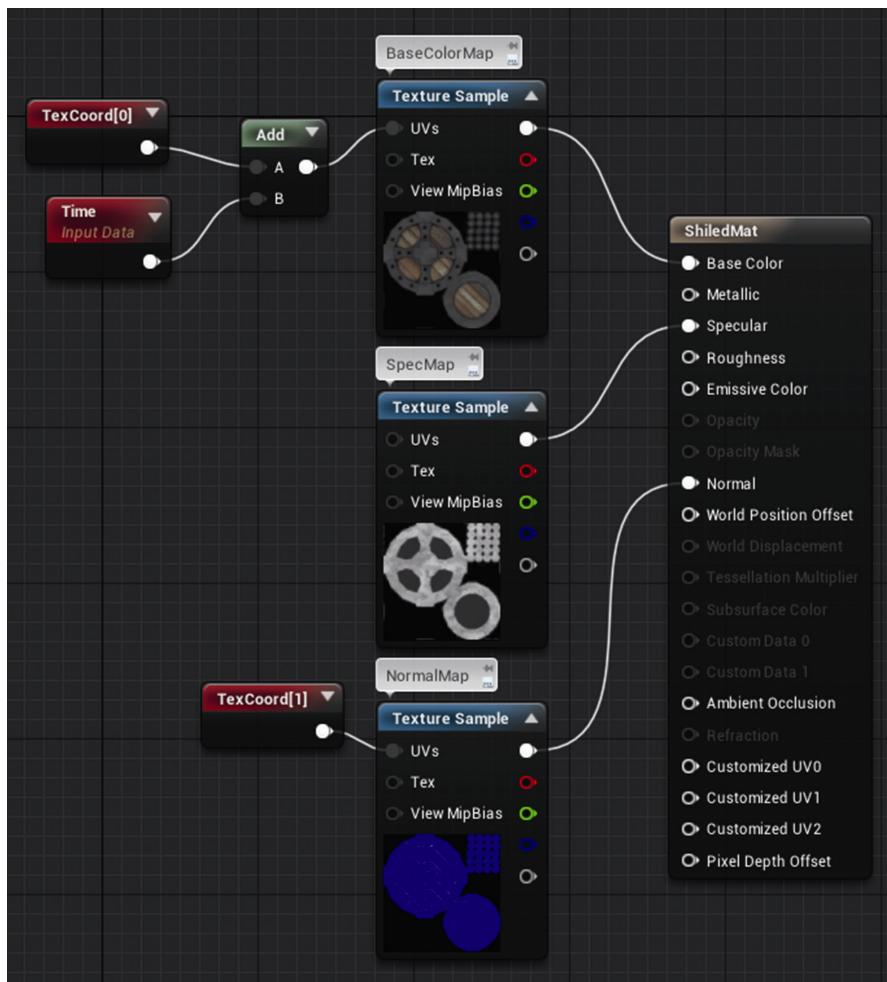


图 17-13 使用多个纹理坐标采样纹理的材质示例

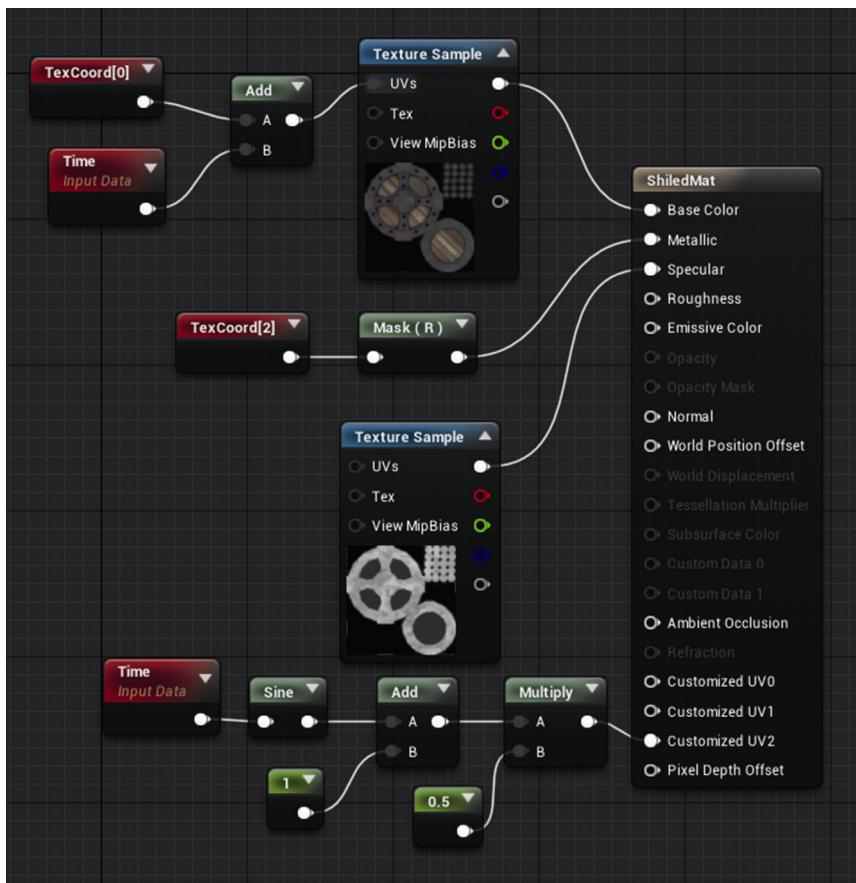


图 17-14 使用自定义 UV 输入将逻辑移到顶点着色器

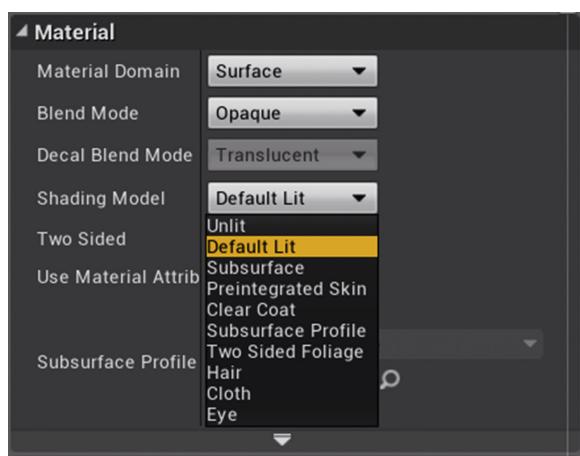


图 17-15 UE4 中提供的不同着色模型

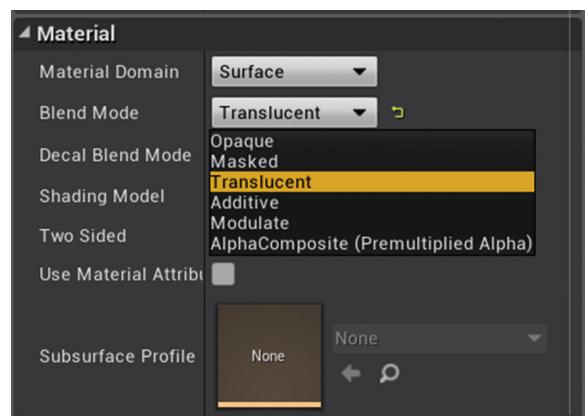


图 17-16 材质“详细信息”面板中的 Blend Mode 下拉菜单

## 第 18 章

# 在 Godot 中编写着色器

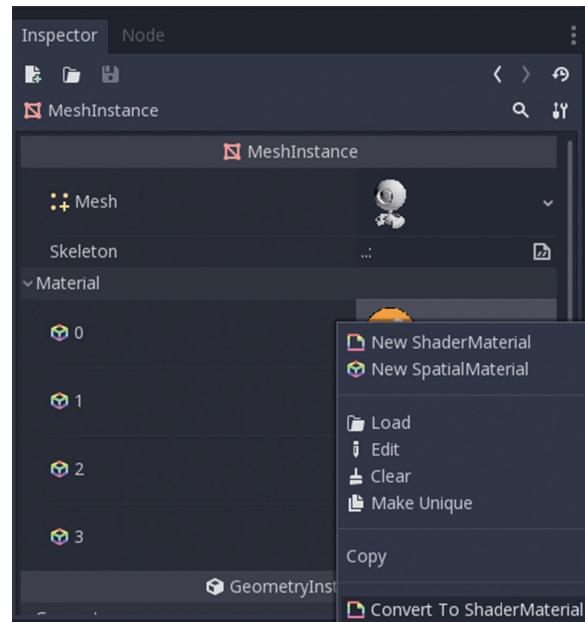


图 18-1 在 Godot 中创建新的着色器材质

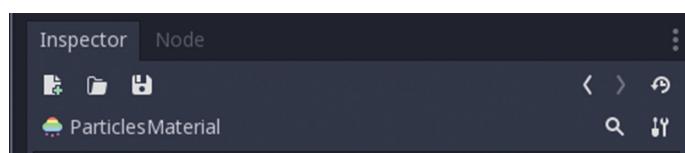


图 18-2 检查器面板的顶部。Create 按钮是 Inspector 文字下面最左边的按钮

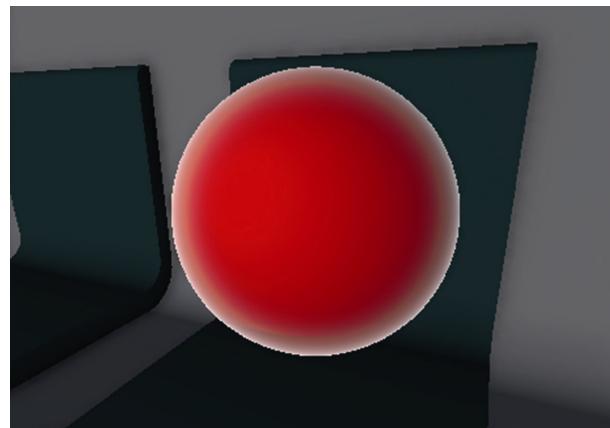


图 18-3 默认的边缘光效果

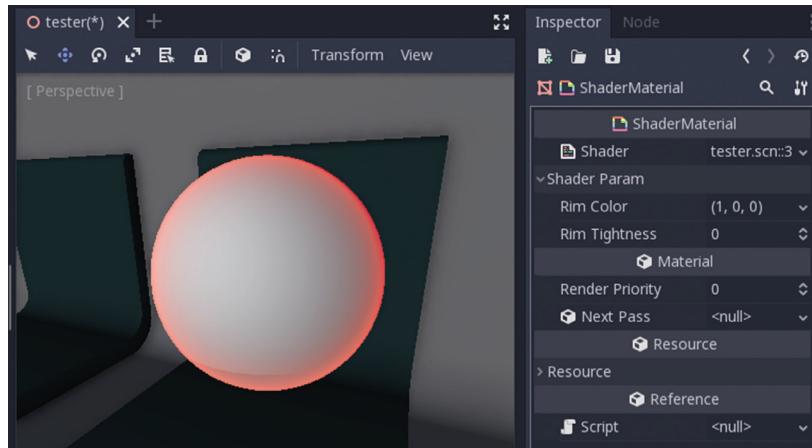


图 18-4 自定义边缘光着色器在白色球体上创建红色边缘

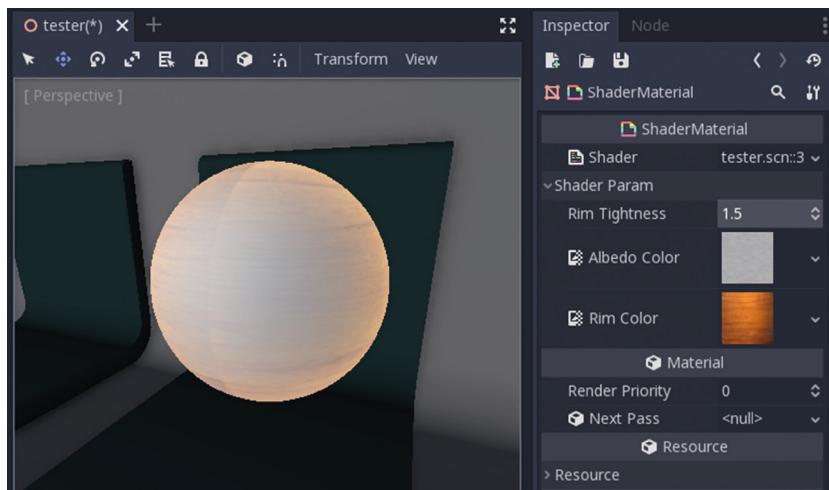


图 18-5 边缘光着色器的最终输出

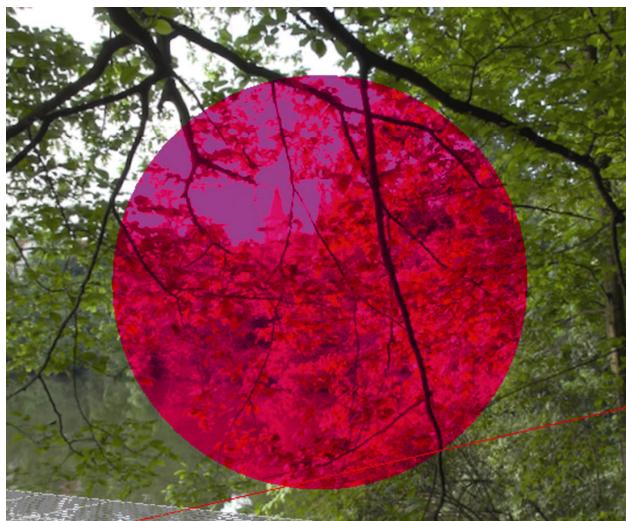


图 18-6 使用乘法混合着色器