第5章 纹理贴图

纹理贴图是在光栅化的模型表面上覆盖图像的技术。它是为渲染 场景添加真实感的最基本和最重要的方法之一。

纹理贴图非常重要，因此硬件也为它提供了支持，使得它具备了 实现实时的照片级真实感的超高性能。纹理单元是专为纹理设计的硬 件组件，现代显卡通常带有数个纹理单元。

5.1 加载纹理图像文件

为了在OpenGL/GLSL中有效地完成纹理贴图，需要协调好以下几个 不同的数据集和机制：

用于保存纹理图像的纹理对象（在本章中我们仅考虑2D图像）； 一个特殊的统一采样器变量，以便顶点着色器可以访问纹理； 用于保存纹理坐标的缓冲区； 用于将纹理坐标传递给管线的顶点属性；

显卡上的纹理单元。

纹理图像可以是任何图像。它可以是人造的或者自然产生的事物 的图片，例如布、草或行星表面；它也可以是几何图样，例如图5.1中 的棋盘图样。在电子游戏和动画电影中，纹理图像通常用于给角色绘 制面部和衣服，或者在像图5.1中的海豚等生物身上绘制皮肤。



图5.1 使用两张不同的图像给同一个海豚模型添加纹理[TU16]

图像通常存储在图像文件中，例如.jpg、.png、.gif或.tiff格 式。为了使纹理图像可以被用于OpenGL管线中的着色器，我们需要从 图像中提取颜色并将它们放入OpenGL纹理对象（用于保存纹理图像的 内置OpenGL结构）中。

许多C++库可用于读取和处理图像文件，常见的选择包括Cimg、 BoostGIL和Magick++。我们选择使用专为OpenGL设计的名为SOIL2

[SO17]的库，它基于曾经非常流行但现在已经过时的库SOIL。在附录A

和附录B中介绍了SOIL2的安装步骤。

通常我们将纹理加载到OpenGL应用程序的步骤是：（a）使用 SOIL2实例化OpenGL纹理对象并从图像文件中读入数据；（b）调用 glBindTexture()以使新创建的纹理对象处于激活状态；（c）使用

glTexParameter()函数调整纹理设置。最终获得的结果就是现在可用

的OpenGL纹理对象的整型ID。

创建一个纹理对象，首先需要声明一个GLuint类型的变量。正如 我们所看到的，这是一个用于保存OpenGL对象的整型ID引用的OpenGL 类型。接下来，我们调用SOIL\_load\_OGL\_texture()来实际生成纹理对 象。SOIL\_load\_OGL\_texture()函数接受图像文件名作为其参数之一

（稍后将描述一些其他参数）。这些步骤在以下函数中实现：

GLuint loadTexture(const char \*texImagePath) { GLuint textureID;

textureID = SOIL\_load\_OGL\_texture(texImagePath,

SOIL\_LOAD\_AUTO, SOIL\_CREATE\_NEW\_ID, SOIL\_FLAG\_INVERT\_Y);

if (textureID == 0) cout << "could not find texture file" << texImagePath << endl;

return textureID;

}

我们会经常使用这个函数，所以我们将它添加到Utils.cpp实用工 具类中。这样，我们的C++应用程序就只需调用上述的loadTexture() 函数来创建OpenGL纹理对象，如下所示。

GLuint myTexture = Utils::loadTexture("image.jpg");

其中image.jpg是纹理图像文件，myTexture是生成的OpenGL纹理 对象的整型ID。这里支持多种图像文件类型，包括前面列出的所有图 像文件类型。

5.2 纹理坐标

现在我们已经有了将纹理图像加载到OpenGL中的方法，我们需要

指定我们希望如何将纹理应用于对象的渲染表面。我们通过为模型中 的每个顶点指定纹理坐标来完成此操作。

纹理坐标是对纹理图像（通常是2D）中的像素的引用。纹理图像 中的像素被称为纹素（Texel），以便将它们与在屏幕上呈现的像素区 分开。纹理坐标用于将3D模型上的点映射到纹理中的位置。除了将它

定位在3D空间中的( ,

x y

, )坐标之外，模型表面上的每个点还具有纹理

z

坐标(

s

, )，用来指定纹理图像中的哪个纹素为它提供颜色。这样，物

t

体的表面被按照纹理图像“涂画”。纹理在对象表面上的朝向由分配 给对象顶点的纹理坐标来确定。

要使用纹理贴图，必须为要添加纹理的对象中的每个顶点提供纹 理坐标。OpenGL将使用这些纹理坐标，查找存储在纹理图像中的引用 的纹素的颜色，来确定模型中每个光栅化像素的颜色。为了确保渲染 模型中的每个像素都使用纹理图像中的适当纹素进行绘制，纹理坐标 也需要被放入顶点属性中，以便它们也由光栅着色器进行插值。以这 种方式，纹理图像与模型顶点一起被插值或者填充。

对于通过顶点着色器的每组顶点坐标( , ,

x y z

)，会有一组相应的纹

理坐标(

s

, )。因此，我们将设置两个缓冲区，一个用于顶点（每个条

t

目中有3个分量 、

x y

和 ），另一个用于相应的纹理坐标（每个条目中

z

有两个分量

s

和 ）。这样，每次顶点着色器的调用接收到一个顶点的

t

数据，现在包括了其空间坐标和相应的纹理坐标。

2D纹理坐标最为常见（OpenGL确实支持其他一些维度，但我们不

会在本章中介绍它们）。2D纹理图像被设定为矩形，左下角的位置坐 标为(0,0)，右上角的位置坐标为(1,1)。[[1]](#_bookmark115)理想情况下，纹理坐标应 该在[0…1]范围内取值。

考虑图5.2中的示例。回想一下，立方体模型由三角形构成。我们 的示意图中突出显示了立方体一侧的4个角，但请记住，立方体的每个 正方形侧面需要两个三角形。指定这一个立方体侧面的6个顶点中的每 一个的纹理坐标沿着4个角列出，左上角和右下角各自由一对顶点组

成。示例里也显示了纹理图像。纹理坐标（由

s

和 描述）将图像的部

t

分（纹素）映射到模型正面的光栅化像素上。请注意，顶点之间的所 有中间像素都已使用图像中间插值的纹素进行绘制。这正是因为纹理 坐标在顶点属性中被发送到片段着色器，因此也像顶点本身一样被插 值。

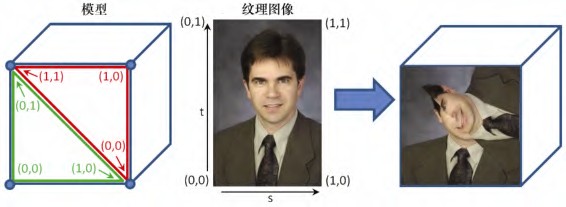


图5.2 纹理坐标

在这个示例中，出于说明的目的，我们故意指定了会导致奇怪的 表面绘制的纹理坐标。仔细观察，您还可以看到图像看起来略微拉伸

——这是因为纹理图像的长宽比与立方体面相关的给定纹理坐标的长

宽比不匹配。

对于立方体或金字塔这样的简单模型，选择纹理坐标相对容易。 但对于具有大量三角形的更复杂的弯曲模型，手动确定它们是不切实 际的。在弯曲的几何形状（例如球形或环面）的情况下，可以通过算 法或数学方式计算纹理坐标。对于使用Maya [MA16]或Blender [BL16]等 建模工具构建的模型，这些工具提供有“UV映射”功能（在本书范围 之外），使得这项任务更容易。

让我们回去渲染我们的金字塔，只是这次用砖的图像添加纹理。 我们需要指定：（a）引用纹理图像的整型ID；（b）模型顶点的纹理 坐标；（c）用于保存纹理坐标的缓冲区；（d）顶点属性，以便顶点 着色器可以接收并通过管线转发纹理坐标；（e）显卡上用于保存纹理 对象的纹理单元；（f）我们将很快看到的用于访问GLSL中纹理单元的 统一采样器变量。这些将在下一节中描述。

5.3 创建纹理对象

假设此处显示的纹理图像（如图5.3所示）存储在名为 “brick1.jpg”[LU16]的文件中。

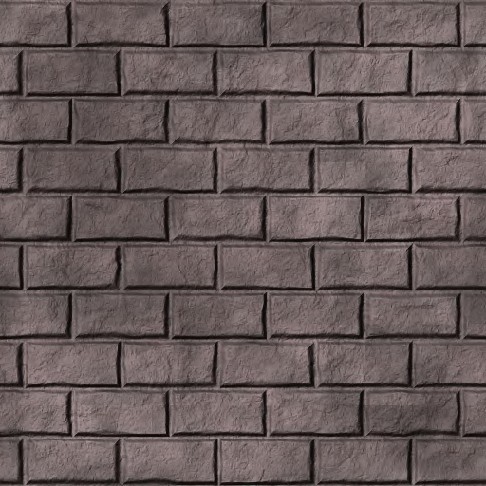


图5.3 纹理图像

如前所示，我们可以通过调用loadTexture()函数来加载此图像， 如下所示：

GLuint brickTexture = Utils::loadTexture("brick1.jpg");

回想一下，纹理对象由整型ID标识，因此brickTexture的类型为 GLuint。

5.4 构建纹理坐标

我们的金字塔有4个三角形侧面和底部的正方形底面。虽然在几何 上这只需要5个点，但我们得用三角形来渲染它。这需要4个三角形用 于侧面，以及2个三角形用于正方形底面，总共6个三角形。每个三角 形有3个顶点，必须在模型中指定总共6×3 = 18个顶点。

我们已经在程序4.3的浮点数组pyramidPositions[ ]中列出了金

字塔的几何顶点。我们可以通过多种方式定位纹理坐标，以便将砖纹 理绘制到金字塔上。一种简单（尽管不完美）的方法是使图像的顶部 中心对应于金字塔的尖顶，如图5.4所示。

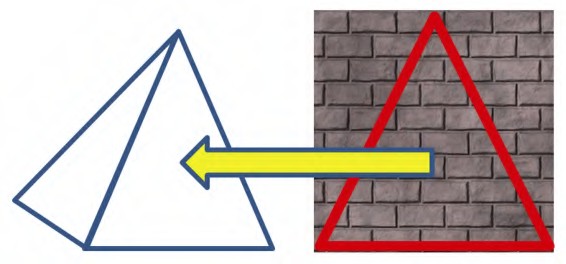


图5.4 纹理图像的顶部中心对应金字塔的尖顶

我们可以为所有4个三角形侧面这样做。我们还需要绘制金字塔的 正方形底面，它由2个三角形组成。一个简单而合理的方法是用图片中 的整个区域为其添加纹理（图5.5所示的金字塔已被向后放倒，一个侧 面朝下）。

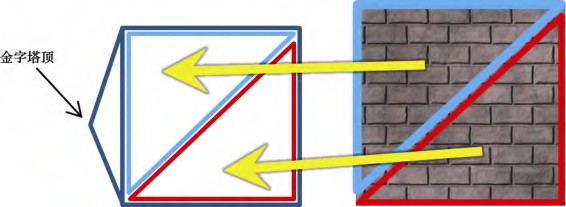


图5.5 为金字塔底面添加纹理

对程序4.3中前9个金字塔顶点使用这个非常简单的策略，相应的

顶点和纹理坐标数据组如图5.6所示。

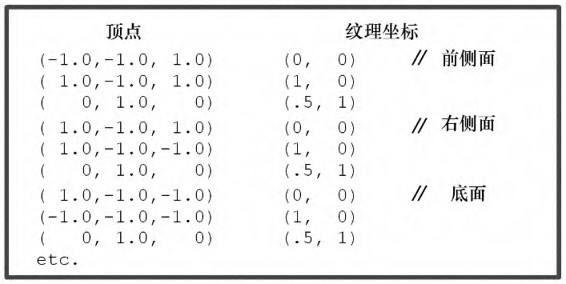


图5.6 金字塔的纹理坐标（部分清单）

5.5 将纹理坐标载入缓冲区

我们可以用与前面加载顶点相似的方式将纹理坐标加载到VBO中。 在setupVertices()中，我们添加以下纹理坐标值声明：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| float pyrTexCoords[36] = | | | | | | |
| { 0.0f, 0.0f, | 1.0f, 0.0f, 0.5f, | 1.0f, | 0.0f, | 0.0f, | 1.0f, | 0.0f, |
| 0.5f, 1.0f, | // 前侧面、右侧面 |  |  |  |  |  |
| 0.0f, 0.0f, | 1.0f, 0.0f, 0.5f, | 1.0f, | 0.0f, | 0.0f, | 1.0f, | 0.0f, |
| 0.5f, 1.0f, | // 后侧面、左侧面 |  |  |  |  |  |
| 0.0f, 0.0f, | 1.0f, 1.0f, 0.0f, | 1.0f, | 1.0f, | 1.0f, | 0.0f, | 0.0f, |
| 1.0f, 0.0f }; | // 底面的两个三角形 |  |  |  |  |  |

然后，在创建至少两个VBO（一个用于顶点，一个用于纹理坐标） 之后，我们添加以下代码行以将纹理坐标加载到VBO #1中：

glBindBuffer(GL\_ARRAY\_BUFFER, vbo[1]); glBufferData(GL\_ARRAY\_BUFFER, sizeof(pyrTexCoords), pyrTexCoords, GL\_STATIC\_DRAW);

5.6 在着色器中使用纹理：采样器变量和纹理单元

为了最大限度地提高性能，我们希望在硬件中执行纹理处理。这 意味着我们的片段着色器需要一种访问我们在C++/OpenGL应用程序中 创建的纹理对象的方法。它的实现机制是通过一个叫作统一采样器变 量的特殊GLSL工具。这是一个变量，用于指示显卡上的纹理单元，从 加载的纹理对象中提取或“采样”哪个纹素。

在着色器中声明一个采样器变量很简单——只需将其添加到您的 统一变量中：

layout (binding=0) uniform sampler2D samp;

我们声明的变量名字叫作“samp”。声明的“layout (binding=0)”部分指定此采样器与纹理单元0相关联。

纹理单元（和相关的采样器）可用于对您希望的任何纹理对象进 行采样，并且可以在运行时进行更改。您的display()函数需要指定纹 理单元要为当前帧采样的纹理对象。因此，每次绘制对象时，都需要 激活纹理单元并将其绑定到特定的纹理对象，例如：

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, brickTexture);

可用纹理单元的数量取决于图形卡上提供的数量。根据OpenGL API文档，OpenGL 4.5版要求每个着色器阶段至少有16个，所有阶段总 共至少80个单元[OP16]。在这个例子中，我们通过在

glActiveTexture()调用中指定GL\_TEXTURE0，使得第0个纹理单元处于

激活状态。

要实际执行纹理处理，我们需要修改片段着色器输出颜色的方 式。以前，我们的片段着色器要么输出一个固定的颜色常量，要么从 顶点属性获取颜色。相反，这次我们需要使用从顶点着色器（通过光 栅着色器）接收的插值纹理坐标来对纹理对象进行采样，像这样调用 texture()函数：

in vec2 tc;

. . .

// 纹理坐标

color = texture(samp, tc);

5.7 纹理贴图：示例程序

程序5.1将前面介绍的步骤合并为一个程序。输出结果显示了用砖 图像纹理贴图的金字塔，如图5.7所示。两个旋转（代码清单中未显 示）被添加到金字塔的模型矩阵中以暴露金字塔的底面。

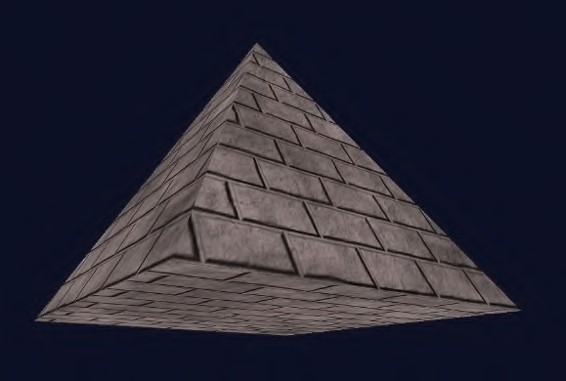


图5.7 使用砖图像纹理贴图后的金字塔

现在，根据需要，通过更改loadTexture()调用中的文件名，将砖

纹理图像替换为其他纹理图像是一件简单的事情。例如，如果我们用 图像文件“ice.jpg”[LU16]替换“brick1.jpg”，我们得到的结果如 图5.8所示。

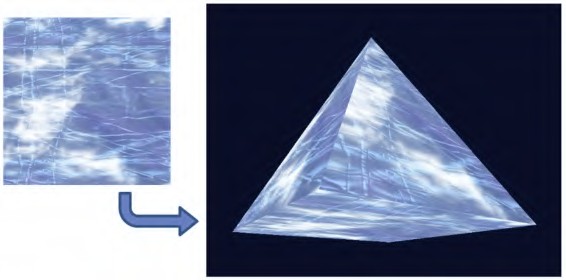


图5.8 使用“冰”图像纹理贴图后的金字塔

程序5.1 砖纹理的金字塔

C++/OpenGL应用程序

#include <SOIL2/soil2.h>

// 其他#include和以前一样

. . .

#define numVAOs 1

#define numVBOs 2

// 摄像机和对象位置、渲染程序、VAO和VBO的变量和以前一样

. . .

// 显示函数的变量分配和以前一样

. . .

GLuint brickTexture;

void setupVertices(void) {

float pyramidPositions[54] = { /\* 如程序4.2中列出的数据 \*/

float pyrTexCoords[36] = {

0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,

0.0f, 0.0f, 1.0f,

0.0f, 0.5f, 1.0f,

0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f, 0.0f, 0.0f, 1.0f,

0.0f, 0.5f, 1.0f,

0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f, 1.0f, 1.0f, 0.0f,

0.0f, 1.0f, 0.0f

};

// . . . 像以前一样生成VAO和至少两个VBO，并加载两个缓冲区： glBindBuffer(GL\_ARRAY\_BUFFER, vbo[0]); glBufferData(GL\_ARRAY\_BUFFER, sizeof(pyramidPositions),

pyramidPositions, GL\_STATIC\_DRAW);

glBindBuffer(GL\_ARRAY\_BUFFER, vbo[1]); glBufferData(GL\_ARRAY\_BUFFER, sizeof(pyrTexCoords),

pyrTexCoords, GL\_STATIC\_DRAW);

}

void init(GLFWwindow\* window) {

// 渲染程序配置、摄像机和对象位置没有改变

. . .

brickTexture = Utils::loadTexture("brick1.jpg");

}

void display(GLFWwindow\* window, double currentTime) {

. . .

// 背景颜色配置、深度缓冲区、渲染程序，以及M、V、MV、PROJ矩阵没有变化

. . .

glBindBuffer(GL\_ARRAY\_BUFFER, vbo[0]); glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0); glEnableVertexAttribArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, vbo[1]); glVertexAttribPointer(1, 2, GL\_FLOAT, GL\_FALSE, 0, 0); glEnableVertexAttribArray(1);

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, brickTexture);

glEnable(GL\_DEPTH\_TEST); glDepthFunc(GL\_LEQUAL);

glDrawArrays(GL\_TRIANGLES, 0, 18);

}

// main() 和以前一样 顶点着色器

#version 430

layout (location=0) in vec3 pos;

layout (location=1) in vec2 texCoord;

out vec2 tc; // 纹理坐标输出到光栅着色器用于插值

uniform mat4 mv\_matrix; uniform mat4 proj\_matrix;

layout (binding=0) uniform sampler2D samp; // 顶点，着色器中未使 用

void main(void)

{ gl\_Position = proj\_matrix \* mv\_matrix \* vec4(pos,1.0); tc = texCoord;

}

片段着色器

#version 430

in vec2 tc; // 输入插值过的材质坐标

out vec4 color;

uniform mat4 mv\_matrix; uniform mat4 proj\_matrix;

layout (binding=0) uniform sampler2D samp;

void main(void)

{ color = texture(samp, tc);

}

5.8 多级渐远纹理贴图

纹理贴图经常会在渲染图像中产生各种不期望的伪影。这是因为 纹理图像的分辨率或长宽比很少与被纹理贴图的场景中区域的分辨率 或长宽比相匹配。

当图像分辨率小于所绘制区域的分辨率时，会出现一种很常见的 伪影。在这种情况下，需要拉伸图像以覆盖整个区域，就会变得模糊

（并且可能变形）。根据纹理的性质，有时可以通过改变纹理坐标分 配方式来对抗这种情况，使得纹理需要较少的拉伸。另一种解决方案 是使用更高分辨率的纹理图像。

相反的情况是当图像纹理的分辨率大于被绘制区域的分辨率时。

可能并不是很容易理解为什么这会造成问题，但确实如此！在这种情 况下，可能会出现明显的叠影伪影，从而产生奇怪的错误图案，或移 动物体中的“闪烁”效果。

叠影是由采样错误引起的。它通常与信号处理有关，不充分采样 的信号被重建时，看起来会具有和实际不同的特性（例如波长）。例 子如图5.9所示（见彩插）。原始波形显示为红色，沿波形的黄点代表 采样点。如果采样点被用于重建波形，并且采样频率不足，则可能会 定义出不同的波形（以蓝色显示）。

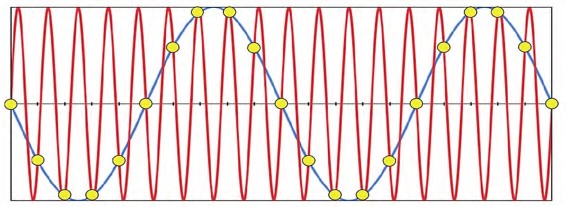


图5.9 不充分采样造成的叠影

类似地，在纹理贴图中，当稀疏地采样高分辨率（和高细节）图 像时（例如使用统一采样器变量时），提取到的颜色将不足以反映图 像中的实际细节，而是可能看起来很随机。如果纹理图像具有重复图 案，则叠影可能导致生成与原始图像不同的图案。如果被纹理贴图的 对象正在移动，则纹素查找中的舍入误差可能导致给定纹理坐标处的

采样像素的不断变化，从而在被绘制对象的表面上产生不希望的闪烁

效果。

图5.10显示了一个立方体顶部的倾斜渲染特写，该立方体使用大 尺寸高分辨率棋盘图像进行纹理贴图。

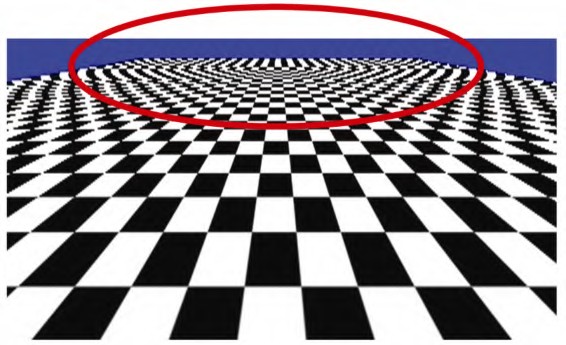


图5.10 纹理贴图中的叠影

在图像顶部附近明显发生了混叠，棋盘的欠采样产生了“条纹” 效果。虽然我们无法在静止图像中展示，但如果这是一个动画场景， 则看起来的图案可能会在各种不正确的图案（包括图示的这一个在 内）之间波动。

另一个例子如图5.11所示，其中的立方体已经使用月球表面的图 像[HT16]进行纹理贴图。乍一看，这张图片显得清晰而细节丰富。然 而，图像右上部分的某些细节是错误的，并且当立方体对象（或相 机）移动时会导致“闪烁”。（不幸的是，我们无法在静止图像中清 楚地显示闪烁效果。）



图5.11 纹理贴图中的“闪烁”

使用多级渐远纹理贴图（Mipmapping）技术可以在很大程度上校 正这一类的采样误差伪影，它需要用各种分辨率创建纹理图像的不同 版本。然后，OpenGL使用最适合正在处理的这一点处的分辨率的纹理 图像进行纹理贴图。更好的是，可以为被贴图的区域使用最适合的分 辨率的纹理图像的平均颜色。多级渐远纹理贴图应用于图5.10和图 5.11中的图像的结果如图5.12所示。

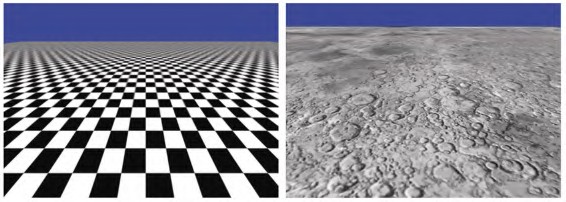


图5.12 多级渐远纹理贴图结果

多级渐远纹理贴图通过一种巧妙的机制来工作，它在纹理图像中

存储相同图像的连续的一系列较低分辨率的副本，所用的纹理图像比

原始图像大1/3。这是通过将图像的

R

、 和 分量分别存储在纹理图像

G B

空间的3个1/4中来实现的，然后在剩余的1/4图像空间中对于同一图像 重复相当于原始分辨率1/4的处理。重复该细分直到剩余象限太小而不 包含任何有用的图像数据。示例图像和生成的多级渐远纹理的可视化 如图5.13所示（见彩插）。



图5.13 为图片生成多级渐远纹理

这种将几个图像填充到一个小空间中的方法（只比存储原始图像 所需的空间大一点）是Mipmapping得名的原因。MIP代表拉丁语Multum In Parvo [WI83]，意思是“在很小的空间里有很多东西”。

实际给对象添加纹理时，可以通过多种方式对多级渐远纹理进行 采样。在OpenGL中，可以通过将GL\_TEXTURE\_MIN\_FILTER参数设置为所

需的缩小方法来选择多级渐远纹理的采样方式，可以选取以下方法之

一。

GL\_NEAREST\_MIPMAP\_NEAREST

选择具有与纹素区域最相似的分辨率的多级渐远纹理。然后，它 获得所需纹理坐标的最近纹素。

GL\_LINEAR\_MIPMAP\_NEAREST

选择具有与纹素区域最相似的分辨率的多级渐远纹理。然后它取 最接近纹理坐标的4个纹素的插值。这被称为“线性过滤”。 GL\_NEAREST\_MIPMAP\_LINEAR

选择具有与纹素区域最相似的分辨率的2个多级渐远纹理。然后， 它从每个多级渐远纹理获取纹理坐标的最近纹素并对其进行插 值。这被称为“双线性过滤”。

GL\_LINEAR\_MIPMAP\_LINEAR

选择具有与纹素区域最相似的分辨率的2个多级渐远纹理。然后， 它取各自最接近纹理坐标的4个纹素，并计算插值。这被称为“三 线性过滤”，如图5.11所示。

三线性过滤通常是比较好的选择，因为较低的混合级别通常会产 生伪影，例如多级渐远纹理级别之间的可见分离。图5.14显示了只启 用了线性过滤的使用多级渐远纹理的棋盘的特写。请注意在多级渐远 纹理的边界处垂直线突然从粗变为细（图中圈出的位置的伪影）。相 比之下，图5.15中的示例使用了三线性过滤。

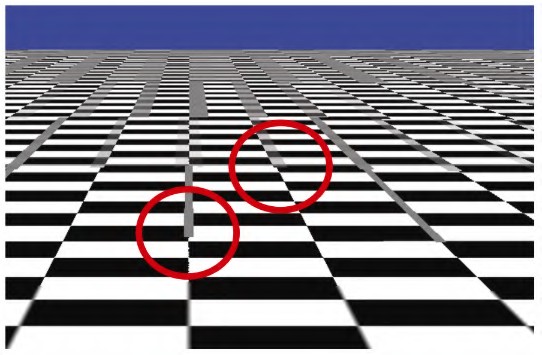


图5.14 线性过滤伪影

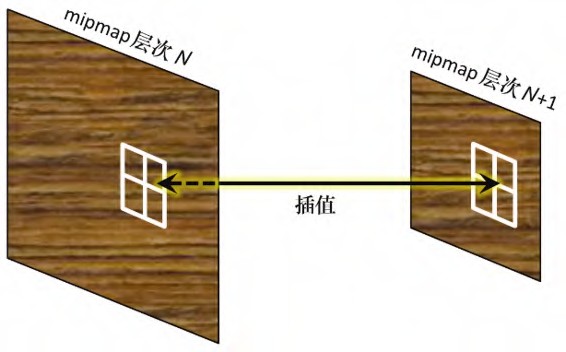


图5.15 三线性过滤

OpenGL提供了丰富的多级渐远纹理支持。有一些机制可用于构建 你自己的多级渐远纹理级别，或者让OpenGL为你构建它们。在大多数 情况下，OpenGL自动构建的多级渐远纹理已足够。这是通过将以下代 码行添加进getTextureObject()函数之后立即执行的Utils:: loadTexture()函数（前面的5.1节中介绍过）中实现的：

glBindTexture(GL\_TEXTURE\_2D, textureID); glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glGenerateMipmap(GL\_TEXTURE\_2D);

这通知OpenGL生成多级渐远纹理。使用glBindTexture()调用激活

砖纹理，然后glTexParameteri()函数调用启用前面列出的缩小方法之 一，例如上面调用中显示的GL\_LINEAR\_MIPMAP\_LINEAR，它启用三线性 过滤。

构建多级渐远纹理后，可以通过再次调用glTexParameteri()来更 改过滤选项（尽管这很少需要），例如在display函数中。甚至可以通 过选择GL\_NEAREST或GL\_LINEAR来禁用多级渐远纹理。

对于关键应用程序，可以使用您喜欢的任何图像编辑软件自行构 建多级渐远纹理。然后可以通过为每个多级渐远纹理级别重复调用 OpenGL的glTexImage2D()函数来创建纹理对象，并将它们添加为多级 渐远纹理级别。对这种方法的进一步讨论超出了本书的范围。

5.9 各向异性过滤

多级渐远纹理贴图有时看起来比非多级渐远纹理贴图更模糊，尤 其是当被贴图对象以严重倾斜的视角渲染时。我们在图5.12中看到了 一个这样的例子，使用多级渐远纹理减少伪影的同时也减少了图像细 节（与图5.11相比）。

这种细节的丢失是因为当物体倾斜时，其基元看起来沿一个轴

（即宽度或高度）比沿另一个轴更小。当OpenGL为图元贴图时，它选 择适合两个轴中较小的轴的多级渐远纹理（以避免“闪烁”伪影）。

在图5.12中，表面远离观察者倾斜，因此每个渲染图元将使用适合其

更小的高度的多级渐远纹理，这可能对其宽度来说分辨率太小了。

一种恢复一些丢失细节的方法是使用各向异性过滤（AF）。标准 的多级渐远纹理贴图以各种正方形分辨率（例如256像素×256像素、 128像素×128像素等）对纹理图像进行采样，而AF却以多种矩形分辨 率对纹理进行采样，例如256像素×128像素、64像素×128像素等。这 使得从各种角度观看并同时在纹理中保留尽可能多的细节成为可能。

各向异性过滤比标准多级渐远纹理贴图在计算上代价更高，并且 不是OpenGL的必需部分。但是，大多数显卡都支持AF（这被称为 OpenGL扩展），而OpenGL确实提供了一种查询显卡是否支持AF的方 法，以及一种访问AF的方法。生成多级渐远纹理贴图后立即添加代 码：

. . .

// 如果启用多级渐远纹理贴图

glBindTexture(GL\_TEXTURE\_2D, textureID); glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glGenerateMipmap(GL\_TEXTURE\_2D);

// 如果还启用各向异性过滤

if (glewIsSupported("GL\_EXT\_texture\_filter\_anisotropic")) { GLfloat anisoSetting = 0.0f; glGetFloatv(GL\_MAX\_TEXTURE\_MAX\_ANISOTROPY\_EXT, &anisoSetting); glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT,

anisoSetting);

}

对glewIsSupported()的调用测试显卡是否支持AF。如果支持，我 们将其设置为支持的最大采样程度，这个最大值使用glGetFloatv()获

取。然后使用glTexParameterf()将其应用于激活纹理对象。结果如图

5.16所示。请注意，图5.11中的大部分丢失细节已经恢复，同时仍然 消除了闪烁的伪影。

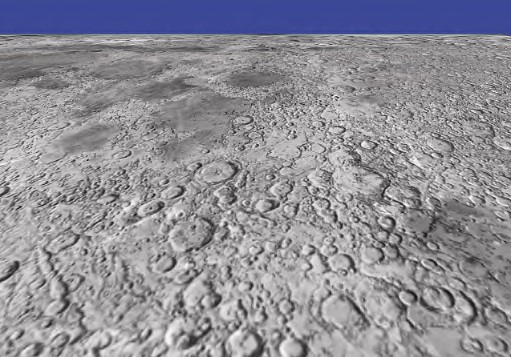


图5.16 各向异性过滤

5.10 环绕和平铺

到目前为止，我们假设纹理坐标都落在[0…1]范围内。但是， OpenGL实际上支持任何取值的纹理坐标。有几个选项可以用来指定当 纹理坐标超出范围[0…1]时会发生什么。使用glTexParameteri()设置 所需的行为，选项如下。

GL\_REPEAT：忽略纹理坐标的整数部分，生成重复或“平铺”图 案。这是默认行为。 GL\_MIRRORED\_REPEAT：忽略整数部分，但是当整数部分为奇数时 坐标反转，因此重复的图案在正常和镜像之间交替。 GL\_CLAMP\_TO\_EDGE：小于0或大于1的坐标分别设置为0和1。 GL\_CLAMP\_TO\_BORDER：将[0…1]以外的纹素设置成指定的边框颜 色。

例如，考虑一个金字塔，其纹理坐标已在[0…5]范围，而不是通

常的[0…1]范围内定义。默认行为（GL\_REPEAT），使用前面图5.2中 显示的纹理图像，将导致纹理在表面上重复五次（有时称为“平 铺”），如图5.17所示。

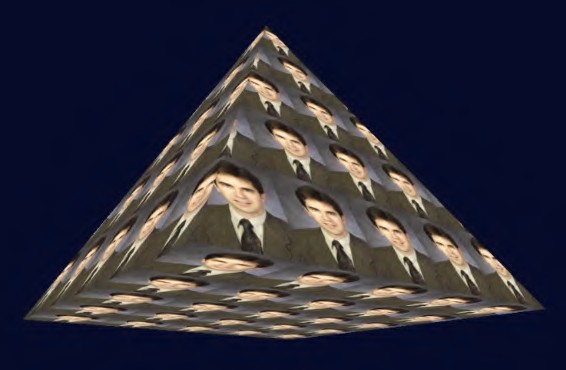


图5.17 使用GL\_REPEAT环绕的纹理坐标

为了使平铺块的外观在正常和镜像之间交替，我们可以指定以下 内容：

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_MIRRORED\_REPEAT);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_MIRRORED\_REPEAT);

通过将GL\_MIRRORED\_REPEAT替换为GL\_CLAMP\_TO\_EDGE，可以指定 将小于0或大于1的值分别设置为0和1。

可以按如下方式来指定小于0或大于1的值输出“边框”颜色：

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_BORDER);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T,

GL\_CLAMP\_TO\_BORDER);

float redColor[4] = { 1.0f, 0.0f, 0.0f, 1.0f }; glTexParameterfv(GL\_TEXTURE\_2D, GL\_TEXTURE\_BORDER\_COLOR, redColor);

图5.18（见彩插）中分别（从左到右）显示了每一个选项（镜像

重复、夹紧到边缘和夹紧到边框）的效果，纹理坐标范围为−2～+3。



图5.18 使用不同环绕选项的金字塔材质贴图

在中间的示例（夹紧到边缘）中，沿纹理图像边缘的像素向外复 制。注意，作为副作用，金字塔面的左下和右下区域分别从纹理图像 的左下和右下像素获得它们的颜色。

5.11 透视变形

我们已经看到，当纹理坐标从顶点着色器传递到片段着色器时， 它们通过光栅着色器并被插值。我们还看到，这是自动线性插值的结 果，总是在顶点属性上执行。

然而，在纹理坐标的情况下，线性插值可能导致具有透视投影的 3D场景中的可以察觉的失真。

考虑一个由两个三角形组成的矩形，纹理贴图是棋盘图像，面向

相机。当矩形围绕

X

轴旋转时，矩形的顶部会倾斜并远离相机，而矩形

的下半部分则更靠近相机。因此，我们希望顶部的方块变小，底部的

方块变大。但是，纹理坐标的线性插值将导致所有正方形的高度相 等。沿着构成矩形的两个三角形之间的对角线的失真加剧。产生的失 真如图5.19所示。

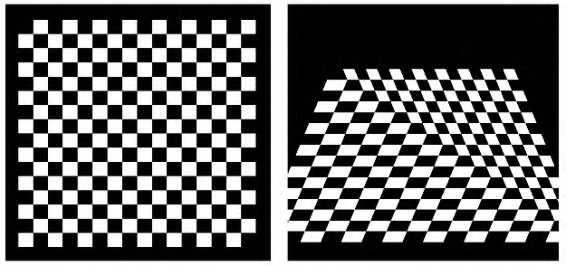


图5.19 纹理透视失真

幸运的是，存在用于校正透视失真的算法，并且默认情况下， OpenGL在光栅化期间会应用透视校正算法[OP14]。图5.20显示了由 OpenGL正确呈现的相同的旋转棋盘。

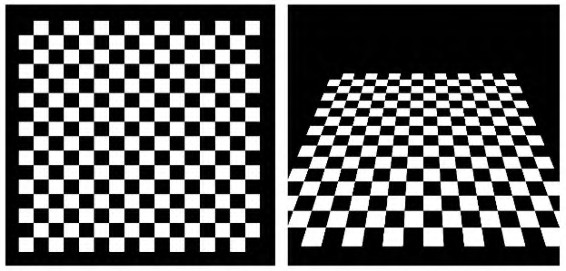


图5.20 OpenGL透视校正

虽然不常见，但可以通过在包含纹理坐标的顶点属性的声明中添 加关键字“noperspective”来禁用OpenGL的透视校正。必须在顶点着

色器和片段着色器中都这样添加。例如，顶点着色器中的顶点属性将

声明如下：

noperspective out vec2 texCoord;

片段着色器中的相应属性声明：

noperspective in vec2 texCoord;

实际上，我使用了这种语法来生成图5.19中的扭曲棋盘格。

5.12 材质——更多OpenGL细节

我们在本书中使用的SOIL2纹理图像加载库具有使用起来相对简单 和直观的优点。但是，在学习OpenGL时，使用SOIL2会产生一项我们不 想要的后果，即用户会接触不到一些有用的重要OpenGL细节。在本节 中，我们将描述程序员在没有纹理加载库（如SOIL2）的情况下加载和 使用纹理时需要了解的一些细节。

可以使用C++和OpenGL函数直接将纹理图像文件数据加载到OpenGL 中。虽然它有点复杂，但并不少见。一般步骤如下。

（1）使用C++工具读取图像文件。

（2）生成OpenGL纹理对象。

（3）将图像文件数据复制到纹理对象中。

我们不会详细描述第一步——有太多方法了。在opengl-

tutorials.org（具体的教程页面为[OT18]）中很好地描述了一种方 法，并使用C++函数fopen()和fread()将数据从.bmp图像文件读入 unsigned char类型的数组中。

步骤2和步骤3更通用，主要涉及OpenGL调用。在第2步中，我们使 用OpenGL的glGenTextures()命令创建一个或多个纹理对象。例如，生 成单个OpenGL纹理对象（使用整型引用ID）可以按如下方式完成：

GLuint textureID;

纹理对象

// 或者GLuint类型的数组，如果需要创建多于一个

glGenTextures(1, &textureID);

在步骤3中，我们将步骤1中的图像数据关联到步骤2中创建的纹理 对象。这是使用OpenGL的glTexImage2D()命令完成的。下面的示例将 图像数据从步骤1中描述的unsigned char数组（此处表示为 “data”）加载到步骤2中创建的纹理对象中：

glBindTexture(GL\_TEXTURE\_2D, textureID) glTexImage2D(GL\_TEXTURE\_2D, 0,GL\_RGB, width, height, 0, GL\_BGR,

GL\_UNSIGNED\_BYTE, data);

此时，本章前面介绍的用于设置多级渐远纹理贴图等的各种 glTexParameteri()调用也可以应用于纹理对象。我们现在也以与本章 所述相同的方式使用整型引用（textureID）。

补充说明

研究人员开发了纹理单元的许多用途，不仅仅是场景中的纹理模

型。在后面的章节中，我们将看到如何使用纹理单元来改变物体反射 光线的方式，使其看起来凹凸不平。我们还可以使用纹理单元来存储 “高度图”以生成地形，以及存储“阴影贴图”以有效地为场景添加 阴影。这些用途将在后续章节中描述。

着色器还可以向纹理写入数据，允许着色器修改纹理图像，甚至 将一个纹理的一部分复制到另一个纹理的某个部分。

多级渐远纹理贴图和各向异性过滤不是减少纹理中的叠影伪影的 唯一工具。例如，全屏抗锯齿（Full-scene anti-aliasing，FSAA） 和其他超采样方法也可以改善3D场景中纹理的外观。虽然不是OpenGL 核心的一部分，但它们通过OpenGL的扩展机制[OE16]在许多显卡上得到 支持。

还有一种用于配置和管理纹理和采样器的替代机制。OpenGL 3.3 版引入了采样器对象（有时称为“采样器状态”——不要与采样器变 量混淆），可用于保存一组独立于实际纹理对象的纹理设置。采样器 对象附加到纹理单元，可以方便有效地更改纹理设置。本教材中显示 的示例非常简单，我们决定暂不介绍采样器对象。对于感兴趣的读 者，采样器对象的使用很容易学习，并且有许多优秀的在线教程（例 如[GE11]）。

习题

5.1 如5.11节所述，通过在纹理坐标顶点属性中添加

“noperspective”声明来修改程序5.1。然后重新运行程序并将输出 与原始输出进行比较。是否有任何明显的透视变形？

5.2 使用简单的“画图”程序（如Windows“画图”或 GIMP[GI16]），绘制自己设计的手绘画面。然后使用您的图像在程序 5.1中为金字塔添加纹理贴图。

5.3 （项目）修改程序4.4，使“太阳”“行星”和“月亮”具 有纹理。您可以继续使用已存在的形状，也可以使用任何您喜欢的纹 理。通过搜索一些发布的代码示例可以获得立方体的纹理坐标，或者 您可以手动构建它们（尽管这有点单调乏味）。

# 参考资料

[BL16] Blender, The Blender Foundation, accessed October 2018.

[GE11] Geeks3D, “OpenGL Sampler Objects: Control Your Texture Units,” September 8, 2011, accessed October 2018.

[GI16] GNU Image Manipulation Program, accessed October 2018.

[HT16] J. Hastings-Trew, JHT’s Planetary Pixel Emporium, accessed October 2018.

[LU16] F. Luna,

Introduction to 3D Game Programming with

, 2nd ed. (Mercury Learning, 2016).

DirectX 12

[MA16] Maya, AutoDesk, Inc., accessed October 2018.

[OE16] OpenGL Registry, The Khronos Group, accessed July 2016.

[OP14] “OpenGL Graphics System: A Specification (version 4.4),” M. Segal and K. Akeley, March 19, 2014, accessed July 2016.

[OP16] OpenGL 4.5 Reference Pages, accessed July 2016. [OT18] OpenGL Tutorial, “Loading BMP Images Yourself,”

opengl-tutorial.org, accessed October 2018.

[SO17] Simple OpenGL Image Library 2 (SOIL2), ,

SpartanJ

accessed October 2018.

[TU16] J. Turberville, Studio 522 Productions, Scottsdale, AZ.

[WI83] L. Williams, “Pyramidal Parametrics,” Computer Graphics 17, no. 3 (July 1983).

[[1]](#_bookmark101) 这是OpenGL纹理对象所采用的方向。然而，这与存储在许多标准

图像文件格式中的图像的方向不同，在那些图像中原点位于左上角。 我们通过指定SOIL\_FLAG\_INVERT\_Y参数，垂直翻转图像来重新定向， 使其与OpenGL的预期格式相对应，就像我们在loadTexture()函数中对 SOIL\_load\_OGL\_texture()进行的调用一样。

第6章 3D模型

到目前为止，我们只处理了非常简单的3D对象，例如立方体和金 字塔。这些对象非常简单，我们能够在源代码中明确列出所有顶点信 息，并将其直接放入缓冲区。

然而，大多数有趣的3D场景包括的对象过于复杂，使得我们无法 像之前那样继续手工构建它们。在本章中，我们将探索更复杂的对象 模型，以及如何构建并将它们加载到场景中。

3D建模本身就是一个广阔的领域，我们在这里讲到的必然非常有 限。我们将重点关注以下两个主题：

通过程序来构建模型； 加载外部创建的模型。

虽然这只涉及丰富的3D建模领域中非常浅层的部分，但它将使我 们能够在场景中包含各种复杂和逼真的细节对象。

6.1 程序构建模型——构建一个球体

某些类型的对象（例如球体、圆锥体等）具有数学定义，这些定

义有助于算法生成。例如，对于半径为

R

的圆，围绕其圆周的点的坐标

可以被很好地定义（见图6.1）。

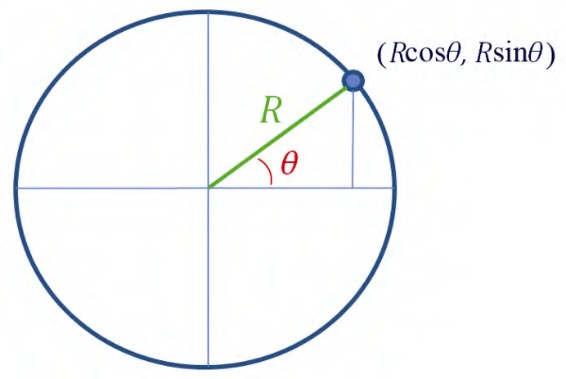


图6.1 构成圆周的点

我们可以系统地使用圆的几何知识来通过算法建立球体模型。我 们的策略如下。

（1）在整个球体上，选择表示一系列圆形“水平切片”的精度。 见图6.2的左侧。

（2）将每个圆形切片的圆周细分为若干个点。见图6.2的右侧。 更多的点和水平切片可以生成更精确、更平滑的球体模型。在我们的 模型中，每个切片将具有相同数量的点。

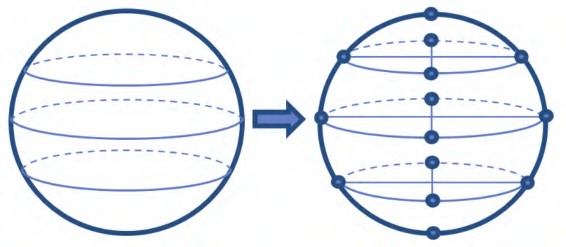


图6.2 构建圆形顶点

（3）将顶点分组为三角形。一种方法是逐步遍历顶点，在每一步

构建两个三角形。例如，当我们沿着图6.3中球体上5个彩色顶点这一 行移动时，对于这5个顶点中的每一个，我们构建了以相应颜色显示的 两个三角形（见彩插，下面将更详细地描述这些步骤）。

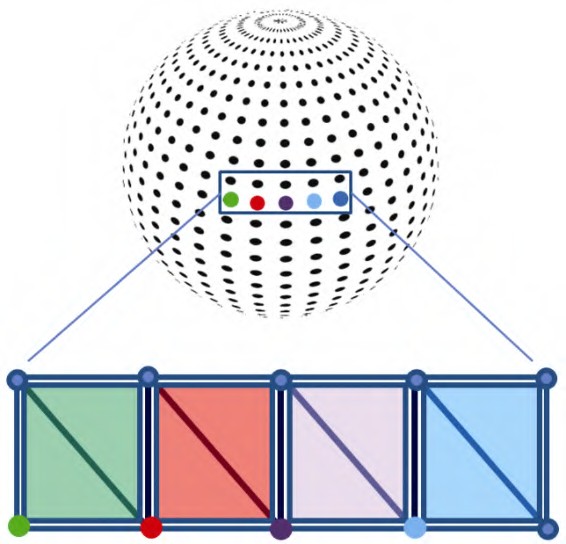


图6.3 将顶点组合成三角形

（4）根据纹理图像的性质选择纹理坐标。在球体的情况下，存在 许多地形纹理图像，假设我们选择这种纹理图像，想象一下，让这个 图像围绕球体“包裹”，我们可以根据图像中纹素的最终对应位置为 每个顶点指定纹理坐标。

（5）对于每个顶点，通常还希望生成法向量（Normal Vector）

——垂直于模型表面的向量。我们将很快在第7章中将它们用于光照。

确定法向量可能很棘手，但是在球体的情况下，从球体中心指向 顶点的向量恰好等于该顶点的法向量！图6.4说明了这个特点（球体的

中心用“星形”表示）。

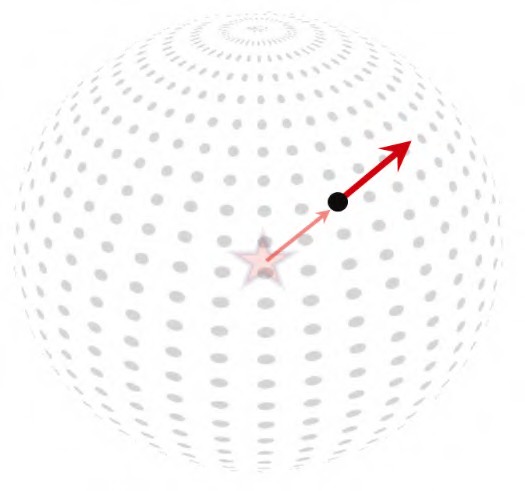


图6.4 球体顶点法向量

一些模型使用索引定义三角形。请注意，在图6.3中，每个顶点出 现在多个三角形中，这将导致每个顶点被多次指定。我们不希望这样 做，而是会存储每个顶点一次，然后为三角形的每个角指定索引，引 用所需的顶点。我们需要存储每个顶点的位置、纹理坐标和法向量， 因此这么做可以为大型模型节省内存。

顶点存储在一维数组中，从最下面的水平切片中的顶点开始。使 用索引时，关联的索引数组包括每个三角形角的条目。其内容是顶点

数组中的整型引用（具体地说，是下标）。假设每个切片包含 个顶

n

点，顶点数组以及相应索引数组的示例部分，如图6.5所示。

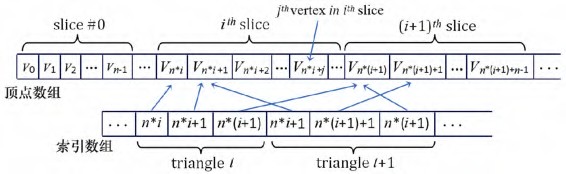


图6.5 顶点数组和相应的索引数组

然后，我们可以从球体底部开始，围绕每个水平切片以圆形方式 遍历顶点。当我们访问每个顶点时，我们构建两个三角形，在其右上 方形成一个方形区域，如图6.3所示。我们将整个处理过程组织成嵌套 循环，如下所示。

对于球体中的每个水平切片i（i的取值从0到球体中的所有切片）

{ 对于切片 i 中的每个顶点 j（j 的取值从 0 到切片中的所有顶点）

{ 计算顶点 j 的指向右边相邻顶点、上方顶点，以及右上方顶点的两个三角形的索

引

} }

例如，考虑图6.3中的“红色”顶点（图6.6中重复出现）。这个 顶点位于图6.6所示的黄色三角形的左下方，按照我们刚刚描述的循

环，它的索引序号是 \*

i n

+ ，其中

j i

是当前正在处理的切片（外循

环），

j

是当前正在该切片中处理的顶点（内循环），

n

是每个切片的

顶点数。图6.6显示了这个顶点（红色）以及它的3个相关的相邻顶点

（见彩插），每个顶点都有公式显示它们的索引序号。

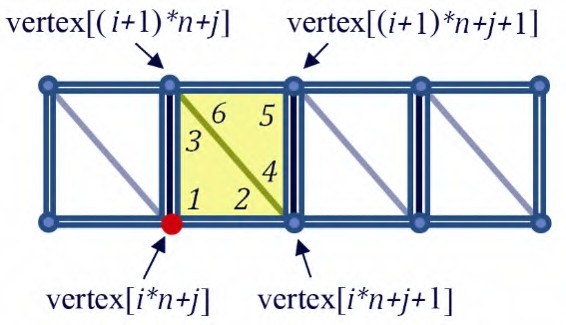


图6.6 第

i

个切片中的第

j

个顶点的索引序号（

n

= 每个切片的顶点数）

然后使用这4个顶点构建为此（红色）顶点生成的两个三角形（以 黄色显示）。这两个三角形的索引表中的6个条目在图中以数字1～6的 顺序表示。注意，条目3和6都指向相同的顶点，对于条目2和4也是如

此。当我们到达以红色突出显示的顶点（即vertex[

i

\* + ]）时由此定

n j

义的两个三角形是由这6个顶点构成的——其中一个三角形的条目标记

为1、2、3，引用的顶点包括vertex[ \* +

i n j

]、vertex[ \* +

i n j

+1]和

vertex[(

i

+1)\*

n

+ ]；另一个三角形的条目标记为4、5、6，引用的顶

j

点包括vertex[

i

\* + +1]、vertex[(

n j i

+1)\* +

n j

+1]和

vertex[(

i

+1)\*

n

+ ]。

j

程序6.1显示了我们的球体模型的实现，类名为Sphere。生成的球 体的中心位于原点。这里还显示了使用Sphere的代码。请注意，每个 顶点都存储在包含GLM类vec2和vec3实例的C++向量中（这与之前的示 例不同，之前顶点存储在浮点数组中）。vec2和vec3包括了获得所需

的 、 和

x y z

分量浮点值的方法，然后就可以如前所述将它们放入浮点缓

冲区。我们将这些值存储在可变长度C++向量中，因为长度取决于运行 时指定的切片数。

请注意Sphere类中三角形索引的计算，如前面的图6.6所述。变量

“prec(precision)”指的是“精度”，在这里它被用来确定球形切片 的数量和每个切片中的顶点数量。因为纹理贴图完全包裹在球体周 围，所以在纹理贴图的左右边缘相交的每个点处需要一个额外的重合 顶点。因此，顶点的总数是(prec+1)\*(prec+1)。由于每个顶点生成6 个三角形索引，因此索引的总数是prec\*prec\*6。

程序6.1 程序生成的球体

球体类（Sphere.cpp）

#include <cmath>

#include <vector>

#include <iostream>

#include <glm\glm.hpp>

#include "Sphere.h" using namespace std;

Sphere::Sphere() { init(48);

}

Sphere::Sphere(int prec) { // prec是精度，也就是切片的数量 init(prec);

}

float Sphere::toRadians(float degrees) { return (degrees \* 2.0f \* 3.14159f) / 360.0f; }

void Sphere::init(int prec) {

numVertices = (prec + 1) \* (prec + 1); numIndices = prec \* prec \* 6;

// std::vector::push\_back() 在向量的末尾增加一个新元素，并为向量长度加

1

for (int i = 0; i < numVertices; i++) {

vertices.push\_back(glm::vec3()); }

for (int i = 0; i < numVertices; i++) { texCoords.push\_back(glm::vec2()); }

for (int i = 0; i < numVertices; i++) { normals.push\_back(glm::vec3()); }

for (int i = 0; i < numIndices; i++) { indices.push\_back(0); }

// 计算三角形顶点

for (int i = 0; i <= prec; i++) {

for (int j = 0; j <= prec; j++) {

float y = (float)cos(toRadians(180.0f - i \* 180.0f /

prec));

float x = -(float)cos(toRadians(j\*360.0f / prec)) \*

(float)abs(cos(asin(y)));

float z = (float)sin(toRadians(j\*360.0f / prec)) \* (float)abs(cos(asin(y)));

vertices[i\*(prec + 1) + j] = glm::vec3(x, y, z); texCoords[i\*(prec + 1) + j] = glm::vec2(((float)j /

prec), ((float)i / prec));

normals[i\*(prec + 1) + j] = glm::vec3(x,y,z);

}

}

// 计算三角形索引

for (int i = 0; i<prec; i++) {

for (int j = 0; j<prec; j++) {

indices[6 \* (i\*prec + j) + 0] = i\*(prec + 1) + j; indices[6 \* (i\*prec + j) + 1] = i\*(prec + 1) + j + 1; indices[6 \* (i\*prec + j) + 2] = (i + 1)\*(prec + 1) + j; indices[6 \* (i\*prec + j) + 3] = i\*(prec + 1) + j + 1; indices[6 \* (i\*prec + j) + 4] = (i + 1)\*(prec + 1) + j +

1;

indices[6 \* (i\*prec + j) + 5] = (i + 1)\*(prec + 1) + j;

}

}

}

// 读取函数

int Sphere::getNumVertices() { return numVertices; } int Sphere::getNumIndices() { return numIndices; }

std::vector<int> Sphere::getIndices() { return indices; } std::vector<glm::vec3> Sphere::getVertices() { return vertices; } std::vector<glm::vec2> Sphere::getTexCoords() { return texCoords;

}

std::vector<glm::vec3> Sphere::getNormals() { return normals; }

球体头文件（Sphere.h）

#include <cmath>

#include <vector>

#include <glm\glm.hpp>

class Sphere

{

private:

int numVertices; int numIndices;

std::vector<int> indices; std::vector<glm::vec3> vertices; std::vector<glm::vec2> texCoords; std::vector<glm::vec3> normals; void init(int);

float toRadians(float degrees);

public:

Sphere(int prec);

int getNumVertices(); int getNumIndices();

std::vector<int> getIndices(); std::vector<glm::vec3> getVertices(); std::vector<glm::vec2> getTexCoords(); std::vector<glm::vec3> getNormals();

};

使用球体类

. . .

#include "Sphere.h"

. . .

Sphere mySphere(48);

. . .

void setupVertices(void) {

std::vector<int> ind = mySphere.getIndices(); std::vector<glm::vec3> vert = mySphere.getVertices(); std::vector<glm::vec2> tex = mySphere.getTexCoords(); std::vector<glm::vec3> norm = mySphere.getNormals();

std::vector<float> pvalues; // 顶点位置 std::vector<float> tvalues; // 纹理坐标 std::vector<float> nvalues; // 法向量

int numIndices = mySphere.getNumIndices(); for (int i = 0; i < numIndices; i++) {

pvalues.push\_back((vert[ind[i]]).x);

pvalues.push\_back((vert[ind[i]]).y);

pvalues.push\_back((vert[ind[i]]).z);

tvalues.push\_back((tex[ind[i]]).s);

tvalues.push\_back((tex[ind[i]]).t);

nvalues.push\_back((norm[ind[i]]).x);

nvalues.push\_back((norm[ind[i]]).y);

nvalues.push\_back((norm[ind[i]]).z);

}

glGenVertexArrays(1, vao); glBindVertexArray(vao[0]);

glGenBuffers(3, vbo);

// 把顶点放入缓冲区 #0

glBindBuffer(GL\_ARRAY\_BUFFER, glBufferData(GL\_ARRAY\_BUFFER,

GL\_STATIC\_DRAW);

vbo[0]);

pvalues.size()\*4,

&pvalues[0],

// 把纹理坐标放入缓冲区 #1

glBindBuffer(GL\_ARRAY\_BUFFER, glBufferData(GL\_ARRAY\_BUFFER,

GL\_STATIC\_DRAW);

vbo[1]);

tvalues.size()\*4,

&tvalues[0],

// 把法向量放入缓冲区 #2

glBindBuffer(GL\_ARRAY\_BUFFER, glBufferData(GL\_ARRAY\_BUFFER,

GL\_STATIC\_DRAW);

}

vbo[2]);

nvalues.size()\*4,

&nvalues[0],

在display()中

. . .

glDrawArrays(GL\_TRIANGLES, 0, mySphere.getNumIndices());

. . .

使用Sphere类时，每个顶点位置和法向量需要3个值，但每个纹理 坐标只需要两个值。这反映在Sphere.h文件中显示的向量

（vertices、texCoords和normals）的声明中，稍后数据从这些向量 中加载到缓冲区中。

值得注意的是，虽然在构建球体的过程中使用了索引，但存储在 VBO中的最终球体顶点数据不使用索引。相反，当setupVertices()循 环遍历球体索引时，它会在VBO中为每个索引条目生成单独的（通常是 冗余的）顶点条目。OpenGL确实有一种索引顶点数据的机制；为简单 起见，我们在此示例中没有使用它，但我们将在下一个示例中使用 OpenGL的索引。

从几何形状到现实世界的物体，使用程序的方式可以创建许多其

他的模型。其中最著名的一个是“犹他茶壶”[CH16]，在1975年由马丁

· 纽厄尔（Martin Newell）开发，使用各种贝塞尔曲线和曲面。 OpenGL Utility Toolkit（或“GLUT”）[GL16]甚至包括了绘制茶壶的 程序（见图6.7）。我们在本书中没有涉及GLUT，但贝塞尔曲面将在第 11章中介绍。



图6.7 OpenGL GLUT茶壶

6.2 OpenGL索引——构建一个环面

6.2.1 环面

用于产生环面的算法可以在各种网站上找到。Paul Baker逐步描 述了定义圆形切片，然后围绕圆圈旋转切片以形成环面的方法[PP07]。 图6.8显示了侧面和上面的两种视图。

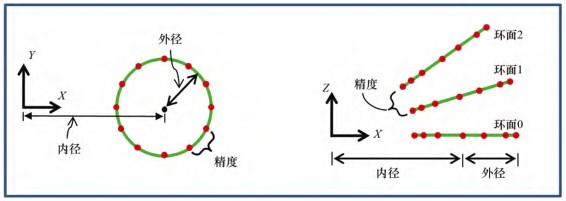


图6.8 构建一个环面

生成环面顶点位置的方式与构建球体的方式有很大不同。对于环

面，算法将一个顶点定位到原点的右侧，然后在

XY

平面上的圆中让这

个顶点围绕

Z

轴旋转，以形成“环”。然后，将这个环“向外”移动

“内半径”那么长的距离。在构建这些顶点时，为每个顶点计算纹理 坐标和法向量。还会额外为每个顶点生成与环面表面相切的向量（称 为切向量）。

围绕

Y

通过围绕

Y

轴旋转最初的这个环，形成用来构成环面的其他环的顶点。 轴旋转最初的环的切向量和法向量来计算每个结果顶点的切

向量和法向量。在顶点创建之后，逐个环地遍历所有顶点，并且对于 每个顶点生成两个三角形。两个三角形的六个索引表条目的生成方式 和之前的球体类似。

我们为剩余的环选择纹理坐标的策略，是将它们排列成使得纹理

图像的

S

轴环绕环面的水平周边的一半，然后再对另一半重复。当我们

绕 轴旋转生成环时，我们指定一个从1开始并增加到指定精度的变量

Y

环（再次称为“prec”）。然后我们将

S

纹理坐标值设置为

ring\*2.0/prec，使

S

的取值范围介于0.0和2.0之间，然后每当纹理坐

标大于1.0时减去1.0。这种方法的动机是避免纹理图像在水平方向上 过度“拉伸”。反之，如果我们确实希望纹理完全围绕环面拉伸，我 们只须从纹理坐标计算中删除“\*2.0”乘数即可。

在C++/OpenGL中构建Torus类可以用与Sphere类几乎完全相同的方 式完成。但是，我们有机会利用OpenGL对顶点索引的支持来利用我们 在构建环面时创建的索引（我们也可以为球体做到这一点，但我们没 有这样做）。对于具有数千个顶点的超大型模型，使用OpenGL索引可 以提高性能，因此我们将描述如何执行此操作。

6.2.2 OpenGL中的索引

在我们的球体和环面模型中，我们生成一个引用顶点数组的整型 索引数组。在球体的情况下，我们使用索引列表来构建一组完整的单 个顶点，并将它们加载到VBO中，就像我们在前面章节的示例中所做的 那样。实例化环面并将其顶点、法向量等加载到缓冲区中可以采用与 程序6.1中类似的方式完成，但我们将使用OpenGL的索引。

使用OpenGL索引时，我们还需要将索引本身加载到VBO中。我们生 成一个额外的VBO用于保存索引。由于每个索引值只是一个整型引用， 我们首先将索引数组复制到整型的C++向量中，然后使用 glBufferData()将向量加载到新增的VBO中，指定VBO的类型为 GL\_ELEMENT\_ARRAY\_BUFFER（这会告诉OpenGL这个VBO包含索引）。执 行此操作的代码可以添加到setupVertices()：

std::vector<int> ind = myTorus.getIndices();

数返回整型向量类型的索引

. . .

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, vbo[3]); VBO

// 环面索引的读取函

// vbo #3是新增的

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, ind.size()\*4, &ind[0],

GL\_STATIC\_DRAW);

在display()方法中，我们将glDrawArrays()调用替换为 glDrawElements()调用，它告诉OpenGL利用索引VBO来查找要绘制的顶 点。我们还使用glBindBuffer()启用包含索引的VBO，指定哪个VBO包 含索引并且是GL\_ELEMENT\_ARRAY\_BUFFER类型。代码如下：

numTorusIndices = myTorus.getNumIndices(); glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, vbo[3]); glDrawElements(GL\_TRIANGLES, numTorusIndices, GL\_UNSIGNED\_INT, 0);

有趣的是，即使我们在C++/OpenGL应用程序中进行了更改，实现 了索引，用于绘制球体的着色器对于环面来说仍然可以继续工作，不 需要修改。OpenGL能够识别GL\_ELEMENT\_ARRAY\_BUFFER的存在并利用它 来访问顶点属性。

程序6.2显示了一个基于Baker实现的名为Torus的类。“内”和 “外”变量指的是图6.9中相应的内半径和外半径。prec变量具有与球 体类似的作用，对顶点数量和索引数量进行类似的计算。相比之下， 确定法向量比使用球体复杂得多。我们使用了Baker描述中给出的策 略，其中计算了两个切向量（Baker称为sTangent和tTangent，尽管通 常称为“切向量（tangent）”和“副切向量（bitangent）”），它 们的叉乘积形成法向量。

在本书的其余部分中，我们将在许多示例中使用此环面类（以及

前面描述的球体类）。

程序6.2 程序生成的环面

Torus 类（Torus.cpp）

#include <cmath>

#include <vector>

#include <iostream>

#include "Torus.h" using namespace std;

Torus::Torus() { prec = 48; inner = 0.5f; outer = 0.2f; init();

}

Torus::Torus(float innerRadius, float outerRadius, int precIn) { prec = precIn;

inner = innerRadius; outer = outerRadius; init();

}

float Torus::toRadians(float degrees) { return (degrees \* 2.0f \* 3.14159f) / 360.0f; }

void Torus::init() {

numVertices = (prec + 1) \* (prec + 1); numIndices = prec \* prec \* 6;

for (int i = 0; i < numVertices; i++) { vertices.push\_back(glm::vec3()); }

for (int i = 0; i < numVertices; i++) { texCoords.push\_back(glm::vec2()); }

for (int i = 0; i < numVertices; i++) { normals.push\_back(glm::vec3()); }

for (int i = 0; i < numVertices; i++) { sTangents.push\_back(glm::vec3()); }

for (int i = 0; i < numVertices; i++) { tTangents.push\_back(glm::vec3()); }

for (int i = 0; i < numIndices; i++) { indices.push\_back(0); }

// 计算第一个环

for (int i = 0; i < prec + 1; i++) {

float amt = toRadians(i\*360.0f / prec);

// 绕原点旋转点，形成环，然后将它们向外移动

glm::mat4 rMat = glm::rotate(glm::mat4(1.0f), amt, glm::vec3(0.0f, 0.0f, 1.0f));

glm::vec3 initPos(rMat \* glm::vec4(outer, 0.0f, 0.0f,

1.0f));

0.0f));

vertices[i] = glm::vec3(initPos + glm::vec3(inner, 0.0f,

// 为环上的每个顶点计算纹理坐标

texCoords[i] = glm::vec2(0.0f, ((float)i / (float)prec));

// 计算切向量和法向量，第一个切向量是绕Z轴旋转的Y轴

rMat = glm::rotate(glm::mat4(1.0f), amt, glm::vec3(0.0f, 0.0f, 1.0f));

tTangents[i] = glm::vec3(rMat \* glm::vec4(0.0f, -1.0f, 0.0f, 1.0f));

sTangents[i] = glm::vec3(glm::vec3(0.0f, 0.0f, -1.0f));

// 第二个切向量是 -Z 轴

normals[i] = glm::cross(tTangents[i], sTangents[i]);

// 它们的叉乘积就是法向量

}

// 绕Y轴旋转最初的那个环，形成其他的环

for (int ring = 1; ring < prec + 1; ring++) {

for (int vert = 0; vert < prec + 1; vert++) {

// 绕Y轴旋转最初那个环的顶点坐标

float amt = (float)( toRadians(ring \* 360.0f / prec)); glm::mat4 rMat = glm::rotate(glm::mat4(1.0f), amt,

glm::vec3(0.0f, 1.0f, 0.0f));

vertices[ring\*(prec + 1) + i] = glm::vec3(rMat \* glm::vec4(vertices[i], 1.0f));

// 计算新环顶点的纹理坐标

texCoords[ring\*(prec + 1) + vert] = glm::vec2((float)ring\*2.0f / (float)prec, texCoords

[vert].t);

if (texCoords[ring\*(prec + 1) + i].s > 1.0) texCoords[ring\*(prec+1)+i].s -= 1.0f;

// 绕Y轴旋转切向量和副切向量

rMat = glm::rotate(glm::mat4(1.0f), amt, glm::vec3(0.0f, 1.0f, 0.0f));

sTangents[ring\*(prec + 1) + i] = glm::vec3(rMat \* glm::vec4(sTangents[i], 1.0f));

rMat = glm::rotate(glm::mat4(1.0f), amt, glm::vec3(0.0f, 1.0f, 0.0f));

tTangents[ring\*(prec + 1) + i] = glm::vec3(rMat \*

glm::vec4(tTangents[i], 1.0f));

// 绕Y轴旋转法向量

rMat = glm::rotate(glm::mat4(1.0f), amt, glm::vec3(0.0f, 1.0f, 0.0f));

normals[ring\*(prec + 1) + i] = glm::vec3(rMat \* glm::vec4(normals[i], 1.0f));

} }

// 按照逐个顶点的两个三角形，计算三角形索引

for (int ring = 0; ring < prec; ring++) {

for (int vert = 0; vert < prec; vert++) {

indices[((ring\*prec + vert) \* 2) \* 3 + 0] = ring\*(prec +

1) + vert;

indices[((ring\*prec + vert) \* 2) \* 3 + 1] = (ring + 1)\*

(prec + 1) + vert;

indices[((ring\*prec + vert) \* 2) \* 3 + 2] = ring\*(prec +

1) + vert + 1;

indices[((ring\*prec + vert) \* 2 + 1) \* 3 + 0] = ring\* (prec + 1) + vert + 1;

indices[((ring\*prec + vert) \* 2 + 1) \* 3 + 1] = (ring + 1)\*(prec + 1) + vert;

indices[((ring\*prec + vert) \* 2 + 1) \* 3 + 2] = (ring + 1)\*(prec + 1) + vert + 1;

} } }

// 环面索引和顶点的访问函数

int Torus::getNumVertices() { return numVertices; } int Torus::getNumIndices() { return numIndices; }

std::vector<int> Torus::getIndices() { return indices; } std::vector<glm::vec3> Torus::getVertices() { return vertices; } std::vector<glm::vec2> Torus::getTexCoords() { return texCoords; } std::vector<glm::vec3> Torus::getNormals() { return normals; } std::vector<glm::vec3> Torus::getStangents() { return sTangents; } std::vector<glm::vec3> Torus::getTtangents() { return tTangents; }

环面头文件（Torus.h）

#include <cmath>

#include <vector>

#include <glm\glm.hpp> class Torus

{

private:

int numVertices; int numIndices; int prec;

float inner; float outer;

std::vector<int> indices;

std::vector<glm::vec3> vertices; std::vector<glm::vec2> texCoords; std::vector<glm::vec3> normals; std::vector<glm::vec3> sTangents; std::vector<glm::vec3> tTangents; void init();

float toRadians(float degrees);

public:

Torus();

Torus(float innerRadius, float outerRadius, int prec); int getNumVertices();

int getNumIndices(); std::vector<int> getIndices();

std::vector<glm::vec3> getVertices(); std::vector<glm::vec2> getTexCoords(); std::vector<glm::vec3> getNormals(); std::vector<glm::vec3> getStangents(); std::vector<glm::vec3> getTtangents();

};

使用 Torus 类（用 OpenGL 索引）

. . .

#include "Torus.h"

. . .

Torus myTorus(0.5f, 0.2f, 48);

. . .

void setupVertices(void) {

std::vector<int> ind = myTorus.getIndices(); std::vector<glm::vec3> vert = myTorus.getVertices(); std::vector<glm::vec2> tex = myTorus.getTexCoords(); std::vector<glm::vec3> norm = myTorus.getNormals();

std::vector<float> pvalues; std::vector<float> tvalues; std::vector<float> nvalues;

int numVertices = myTorus.getNumVertices(); for (int i = 0; i < numVertices; i++) {

pvalues.push\_back(vert[i].x); pvalues.push\_back(vert[i].y); pvalues.push\_back(vert[i].z);

tvalues.push\_back(tex[i].s); tvalues.push\_back(tex[i].t);

nvalues.push\_back(norm[i].x); nvalues.push\_back(norm[i].y); nvalues.push\_back(norm[i].z);

}

glGenVertexArrays(1, vao); glBindVertexArray(vao[0]);

glGenBuffers(4, vbo); // 像以前一样生成VBO，并新增一个用于 索引

glBindBuffer(GL\_ARRAY\_BUFFER, vbo[0]); // 顶点位置

glBufferData(GL\_ARRAY\_BUFFER, pvalues.size() \* 4, &pvalues[0], GL\_STATIC\_DRAW);

glBindBuffer(GL\_ARRAY\_BUFFER, vbo[1]); // 纹理坐标

glBufferData(GL\_ARRAY\_BUFFER, tvalues.size() \* 4, &tvalues[0], GL\_STATIC\_DRAW);

glBindBuffer(GL\_ARRAY\_BUFFER, vbo[2]); // 法向量

glBufferData(GL\_ARRAY\_BUFFER, nvalues.size() \* 4, &nvalues[0], GL\_STATIC\_DRAW);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, vbo[3]); // 索引

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, ind.size() \* 4, &ind[0], GL\_STATIC\_DRAW);

}

在display()中

. . .

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, vbo[3]); glDrawElements(GL\_TRIANGLES, myTorus.getNumIndices(), GL\_UNSIGNED\_INT, 0);

请注意，在使用Torus类的代码中，setupVertices()中的循环现 在只存储一次与每个顶点关联的数据，而不是每个索引条目存储一次

（如球体示例中的情况）。这种差异也反映在要输入VBO的数据的数组 声明大小中。另请注意，在环面示例中，不是在检索顶点数据时使用 索引值，而是直接将它们简单地加载到VBO #3中。由于此VBO被指定为 GL\_ELEMENT\_ARRAY\_BUFFER，OpenGL知道该VBO包含顶点索引。

图6.9显示了实例化环面并使用砖纹理对其进行纹理化的结果。



图6.9 程序生成的环面

6.3 加载外部构建的模型

复杂的3D模型，例如在视频游戏或计算机生成的电影中的人物角 色，通常使用建模工具生成。这种“DCC”（数字内容创建）工具使人 们（例如艺术家）能够在3D空间中构建任意形状并自动生成顶点、纹 理坐标、顶点法向量等。有太多这样的工具，此处无法一一列出，有 几个例子是MAYA、Blender、Lightwave、Cinema4D等。Blender是免费 和开源的。图6.10显示了编辑3D模型时的Blender屏幕示例。

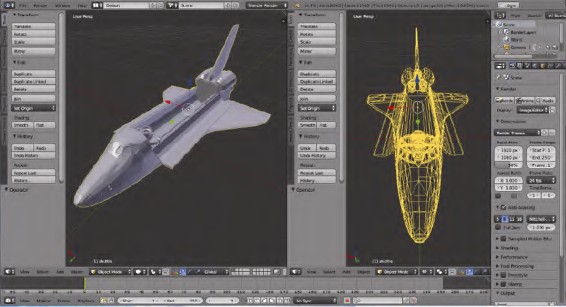


图6.10 Blender模型创建示例[BL16]

为了让我们在OpenGL场景中使用DCC工具创建的模型，需要以我们 可以读取（导入）到我们程序中的格式保存（导出）该模型。有好几 种标准的3D模型文件格式；再次说明，有太多无法一一列出，有一些 例子是Wavefront（.obj）、3D Studio Max（.3ds）、斯坦福扫描存 储库（.ply）、Ogre3D（.mesh），供参考。其中最简单的是 Wavefront（通常被称为OBJ），所以我们将仔细讲解它。

OBJ文件很简单，我们可以相对容易地开发一个基本的导入器。在 OBJ文件中，通过文本行的形式指定顶点几何数据、纹理坐标、法向量 和其他信息。它有一些限制——例如，OBJ文件无法指定模型动画。

OBJ文件中的行，以字符标记开头，表示该行上的数据类型。一些 常见的标签包括：

v-几何（顶点位置）数据； vt-纹理坐标；

vn-顶点法向量；

f-面（通常是三角形中的顶点）。

还有其他标签可以用来存储对象名称、使用的材质、曲线、阴影 和许多其他细节。我们这里只讨论上面列出的4个标签，这些标签足以 导入各种复杂模型。

假设我们使用Blender构建一个简单的金字塔，例如我们为程序 4.3开发的金字塔。图6.11是在Blender中创建的类似的金字塔的屏幕

截图。

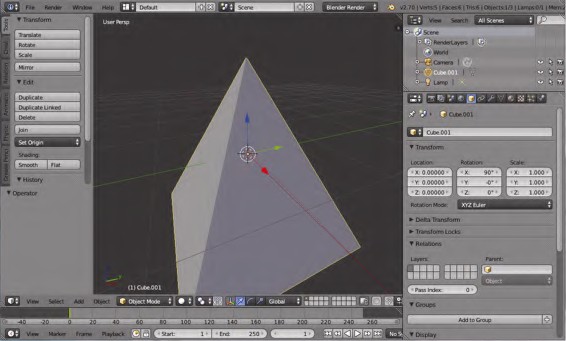


图6.11 在Blender中构建的金字塔

在Blender中，如果我们现在导出我们的金字塔模型，指定.obj格 式，并设置Blender输出纹理坐标和顶点法向量，则会创建一个包含所 有这些信息的OBJ文件。生成的OBJ文件如图6.12所示。（纹理坐标的 实际值可能因模型的构建方式而异。）

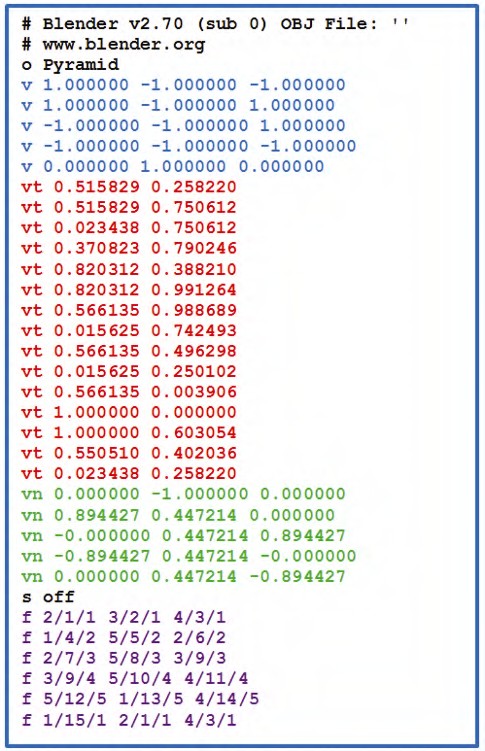


图6.12 金字塔导出的OBJ文件

我们对OBJ文件的重要部分进行了颜色标记以供参考。顶部以 “#”开头的行是由Blender放置的注释，我们的导入器忽略了这些注 释。接下来是以“o”开头的行，给出对象的名称。我们的导入器也可 以忽略这一行。之后，有一行以“s”开头，指定表面不应该被平滑。 我们的代码也会忽略以“s”开头的行。

OBJ文件中的第一部分有实际内容的行是以“v”开头的那些（第4

行～第8行）。它们指定金字塔模型的5个顶点相对于原点的

X

、 和 局

Y Z

部空间坐标。在这里，原点位于金字塔的中心。

红色的值（以“vt”开头）是各种纹理坐标。纹理坐标列表比顶 点列表长的原因是一些顶点参与多个三角形，并且在这些情况下可能 使用不同的纹理坐标。

绿色的值（以“vn”开头）是各种法向量。该列表通常也比顶点 列表长（尽管在该示例中不是这样），同样是因为一些顶点参与多个 三角形，并且在那些情况下可能使用不同的法向量。

在文件底部附近标记为紫色的值（以“f”开头）指定三角形（即 “面”）。在此示例中，每个面（三角形）具有3个元素，每个元素具 有由“/”分隔的3个值（OBJ也允许其他格式）。每个元素的值分别是 顶点列表、纹理坐标和法向量的索引。例如，第三个面是：

f 2 / 7 / 3 5 / 8 / 3 3 / 9 / 3 这表明顶点列表中的第2个、第5个和第3个顶点（蓝色）组成了一

个三角形（请注意OBJ索引从1开始）。相应的纹理坐标是红色部分中 纹理坐标列表中的第7项、第8项和第9项。所有3个顶点都具有相同的 法向量，也就是以绿色显示的法向量列表中的第3项。

OBJ格式的模型并不要求具有法向量，甚至纹理坐标。如果模型没 有纹理坐标或法向量，则面的数值将仅指定顶点索引：

f 2 5 3

如果模型具有纹理坐标，但不具有法向量，则格式如下：

f 2 / 7 5 / 8 3 / 9 并且，如果模型具有法向量但没有纹理坐标，则格式为：

f 2 / / 3 5 / / 3 3 / / 3

模型具有数万个顶点并不罕见。对于所有可以想象的应用场景， 几乎都可以在互联网上下载到数百种这样的模型，包括动物、建筑 物、汽车、飞机、神话生物、人等。

在互联网上可以获得可以导入OBJ模型的复杂程序各不相同的导入 程序。编写一个非常简单的OBJ加载器函数也并不困难，它可以处理我 们看到的基本标记（v、vt、vn和f）。程序6.3显示了一个这样的加载 器，尽管功能非常有限。它包含一个类来保存任意的导入模型，该模 型又调用导入器。

在我们讲述简单OBJ导入器的代码之前，我们必须警告读者其局限 性。

它仅支持包含所有3个面属性字段的模型。也就是说，顶点位置、 纹理坐标和法向量都必须以f #/#/# #/#/# #/#/#这种形式存在。 材质标签将被忽略——必须使用第5章中描述的方法完成纹理化。 仅支持由单个三角形网格组成的OBJ模型（OBJ格式支持复合模 型，但我们的简单导入器不支持）。

它假设每行上的元素只用一个空格分隔。

如果您的OBJ模型不满足上述所有条件，并且您希望使用程序6.3 中的简单加载程序导入它，则可能仍然可行。通常可以将这样的模型 加载到Blender中，然后将其导出到另一个满足加载器限制条件的OBJ 文件中。例如，如果模型不包含法向量，则可以让Blender在导出修改 后的OBJ文件时生成法向量。

我们的OBJ加载器的另一个限制与索引有关。在前面的描述中提到 了“f”标签允许混合和匹配顶点位置、纹理坐标和法向量的可能性。 例如，两个不同的“面”行可以包括指向相同v条目但是不同vt条目的 索引。遗憾的是，OpenGL的索引机制不支持这种灵活性——OpenGL中 的索引条目只能指向特定的顶点及其属性。这使得在某种程度上编写 OBJ模型加载器变得复杂，因为我们不能简单地将三角形面条目中的引 用复制到索引数组中。相反，使用OpenGL索引需要确保面条目的v、vt 和vn值的整个组合在索引数组中都有自己的引用。一种更简单但效率 更低的替代方案是为每个三角形面条目创建一个新顶点。尽管使用 OpenGL索引具有节省空间的优势（特别是在加载较大模型时），但为 了清晰，我们选择这种更简单的方法。

ModelImporter类包含一个parseOBJ()函数，它逐行读取OBJ文 件，分别处理v、vt、vn和f这4种情况。在每种情况下，提取行上的后 续数字，首先使用erase()跳过初始的v、vt、vn或f字符，然后使用 C++ stringstream类的“>>”运算符提取每个后续参数值，然后将它

们存储在C++浮点向量中。当处理面（f）条目时，使用C++浮点向量中

的对应条目构建顶点，包括顶点位置、纹理坐标和法向量。

ModelImporter类和ImportedModel类包含在同一个文件中， ImportedModel类通过将导入的顶点放入vec2和vec3对象的向量中，简 化了加载和访问OBJ文件顶点的过程。回想一下这些GLM类；我们在这 里使用它们来存储顶点位置、纹理坐标和法向量。然后， ImportedModel类中的读取函数使它们可用于C++/OpenGL应用程序，其 方式与Sphere和Torus类中的方式相同。

在ModelImporter和ImportedModel类之后是一系列调用示例，加 载OBJ文件，然后将顶点信息传输到一组VBO中以供后续渲染。

图6.13显示了从NASA网站[NA16]下载的OBJ格式的航天飞机渲染模 型，使用程序6.3中的代码导入，并使用程序5.1中的代码和相应的带 有各向异性过滤的NASA纹理图像文件进行纹理化。该纹理图像是使用 UV映射的示例，其中模型中的纹理坐标被仔细地映射到纹理图像的特 定区域。（如第5章所述，UV映射的细节超出了本书的范围。）

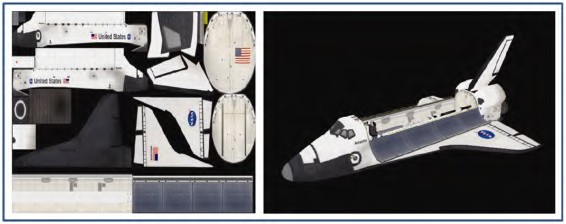


图6.13 带有纹理的NASA航天飞机模型

程序6.3 简化的（有限制的）OBJ加载器

ImportedModel和ModelImporter类(ImportedModel.cpp)

#include <fstream>

#include <sstream>

#include <glm\glm.hpp>

#include "ImportedModel.h" using namespace std;

// ------------ ImportedModel类 ImportedModel::ImportedModel(const char \*filePath) {

ModelImporter modelImporter = ModelImporter(); modelImporter.parseOBJ(filePath); // 使用modelImporter

获取顶点信息

numVertices = modelImporter.getNumVertices(); std::vector<float> verts = modelImporter.getVertices();

std::vector<float> tcs = modelImporter.getTextureCoordinates(); std::vector<float> normals = modelImporter.getNormals();

for (int i = 0; i < numVertices; i++) { vertices.push\_back(glm::vec3(verts[i\*3], verts[i\*3+1],

verts[i\*3+2]));

texCoords.push\_back(glm::vec2(tcs[i\*2], tcs[i\*2+1])); normalVecs.push\_back(glm::vec3(normals[i\*3], normals[i\*3+1],

normals[i\*3+2]));

} }

int ImportedModel::getNumVertices() { return numVertices; }

// accessors

std::vector<glm::vec3> ImportedModel::getVertices() { return vertices; }

std::vector<glm::vec2> ImportedModel::getTextureCoords() { return texCoords; }

std::vector<glm::vec3> ImportedModel::getNormals() { return normalVecs; }

// -------------- ModelImporter类 ModelImporter::ModelImporter() {}

void ModelImporter::parseOBJ(const char \*filePath) { float x, y, z;

string content;

ifstream fileStream(filePath, ios::in);

string line = "";

while (!fileStream.eof()) { getline(fileStream, line);

if (line.compare(0, 2, "v ") == 0) { // 顶点位置

（"v"的情况）

stringstream ss(line.erase(0, 1));

ss >> x; ss >> y; ss >> z; // 提取顶点位置

数值

vertVals.push\_back(x); vertVals.push\_back(y); vertVals.push\_back(z);

}

if (line.compare(0, 2, "vt") == 0) { // 纹理坐标

（"vt"的情况）

stringstream ss(line.erase(0, 2));

ss >> x; ss >> y; // 提取纹理坐标

数值

stVals.push\_back(x); stVals.push\_back(y);

}

if (line.compare(0, 2, "vn") == 0) { // 顶点法向量

（"vn"的情况）

stringstream ss(line.erase(0, 2));

ss >> x; ss >> y; ss >> z; // 提取法向量数

值

normVals.push\_back(x); normVals.push\_back(y); normVals.push\_back(z);

}

if (line.compare(0, 2, "f") == 0) { // 三角形面

（"f"的情况）

string oneCorner, v, t, n; stringstream ss(line.erase(0, 2)); for (int i = 0; i < 3; i++) {

getline(ss, oneCorner, ' '); // 提取三角形面

引用

符串转化为整型

stringstream oneCornerSS(oneCorner); getline(oneCornerSS, v, '/'); getline(oneCornerSS, t, '/'); getline(oneCornerSS, n, '/');

int vertRef = (stoi(v) - 1) \* 3; // "stoi"将字 int tcRef = (stoi(t) - 1) \* 2;

int normRef = (stoi(n) - 1) \* 3;

triangleVerts.push\_back(vertVals[vertRef]); // 构建

顶点向量

纹理坐标向量

量的向量

} } } }

triangleVerts.push\_back(vertVals[vertRef + 1]);

triangleVerts.push\_back(vertVals[vertRef + 2]); textureCoords.push\_back(stVals[tcRef]); // 构建 textureCoords.push\_back(stVals[tcRef + 1]); normals.push\_back(normVals[normRef]); // 法向

normals.push\_back(normVals[normRef + 1]);

normals.push\_back(normVals[normRef + 2]);

int ModelImporter::getNumVertices() { return (triangleVerts.size()/3); } // 读取函数

std::vector<float> ModelImporter::getVertices() { return triangleVerts; }

std::vector<float> ModelImporter::getTextureCoordinates() { return textureCoords; }

std::vector<float> ModelImporter::getNormals() { return normals; } ImportedModel 和 ModelImporter 头文件（ImportedModel.h）

#include <vector>

class ImportedModel

{

private:

int numVertices; std::vector<glm::vec3> vertices; std::vector<glm::vec2> texCoords; std::vector<glm::vec3> normalVecs;

public:

ImportedModel(const char \*filePath); int getNumVertices(); std::vector<glm::vec3> getVertices();

std::vector<glm::vec2> getTextureCoords(); std::vector<glm::vec3> getNormals();

};

class ModelImporter

{

private:

// 从OBJ文件读取的数值

std::vector<float> vertVals; std::vector<float> stVals; std::vector<float> normVals;

// 保存为顶点属性以供后续使用的数值 std::vector<float> triangleVerts; std::vector<float> textureCoords; std::vector<float> normals;

public:

ModelImporter();

void parseOBJ(const char \*filePath); int getNumVertices(); std::vector<float> getVertices();

std::vector<float> getTextureCoordinates(); std::vector<float> getNormals();

};

使用模型导入器

. . .

ImportedModel myModel("shuttle.obj"); // 在顶层声明中

. . .

void setupVertices(void) {

std::vector<glm::vec3> vert = myModel.getVertices(); std::vector<glm::vec2> tex = myModel.getTextureCoords(); std::vector<glm::vec3> norm = myModel.getNormals();

int numObjVertices = myModel.getNumVertices();

std::vector<float> pvalues; // 顶点位置 std::vector<float> tvalues; // 纹理坐标 std::vector<float> nvalues; // 法向量

for (int i = 0; i < numObjVertices(); i++) { pvalues.push\_back((vert[i]).x); pvalues.push\_back((vert[i]).y); pvalues.push\_back((vert[i]).z); tvalues.push\_back((tex[i]).s); tvalues.push\_back((tex[i]).t); nvalues.push\_back((norm[i]).x); nvalues.push\_back((norm[i]).y); nvalues.push\_back((norm[i]).z);

}

glGenVertexArrays(1, vao); glBindVertexArray(vao[0]); glGenBuffers(numVBOs, vbo);

// 顶点位置的VBO glBindBuffer(GL\_ARRAY\_BUFFER, vbo[0]);

glBufferData(GL\_ARRAY\_BUFFER, pvalues.size() \* 4, &pvalues[0], GL\_STATIC\_DRAW);

// 纹理坐标的VBO

glBindBuffer(GL\_ARRAY\_BUFFER, vbo[1]); glBufferData(GL\_ARRAY\_BUFFER, tvalues.size() \* 4, &tvalues[0],

GL\_STATIC\_DRAW);

// 法向量的VBO glBindBuffer(GL\_ARRAY\_BUFFER, vbo[2]);

glBufferData(GL\_ARRAY\_BUFFER, nvalues.size() \* 4, &nvalues[0], GL\_STATIC\_DRAW);

}

在display()中

. . .

glDrawArrays(GL\_TRIANGLES, 0, myModel.getNumVertices());

补充说明

虽然我们讨论了使用DCC工具创建3D模型，但我们没有讨论如何使 用这些工具。相关教程超出了本书的范围，但是所有流行的工具都有 大量的教程视频材料文档可供使用，例如Blender和MAYA。

3D建模的主题本身就是一个丰富的研究领域。我们在本章中所说 只是一个基本的介绍，重点是它与OpenGL的关系。许多大学提供3D建 模的全部课程，并且我们也鼓励有兴趣学习更多的读者参考一些提供 更多细节的流行资源（例如：[BL16, CH11, VA12]）。

我们重申，本章中介绍的OBJ导入器是很有限的，并且只能处理 OBJ格式支持的一部分功能。虽然足以满足我们的需求，但它会在某些 OBJ文件上失败。在这些情况下，有必要首先将模型加载到 Blender（或MAYA等）工具中，然后将其重新导出为符合导入器限制的 OBJ文件，如本章前面所述。

习题

6.1 修改程序4.4，使“太阳”“行星”和“月亮”成为纹理球 体，如图4.11所示。

6.2 （项目）修改您的习题6.1的程序，以使得图6.16中导入的 NASA航天飞机对象也绕“太阳”运行。您需要试验应用于航天飞机的 尺度和旋转方式，使其看起来更逼真。

6.3 （研究和项目）了解如何使用Blender创建自己的3D对象的 基础知识。想要在您的OpenGL应用程序中充分利用Blender，您将需要 学习如何使用Blender的UV展开工具来生成纹理坐标和相关的纹理图 像。然后，您可以将对象导出为OBJ文件，并使用程序6.3中的代码加 载它。

# 参考资料

[BL16] Blender, The Blender Foundation, accessed October 2018.

[CH11] A. Chopine,

3D Art Essentials: The Fundamentals of

(Focal Press, 2011).

3D Modeling, Texturing, and Animation

[CH16] Computer History Museum, accessed October 2018.

[GL16] GLUT and OpenGL Utility Libraries, accessed

October 2018.

[NA16] NASA 3D Resources, accessed October 2018.

[PP07] P. Baker,

Paul’s Projects

, 2007, accessed October

2018.

[VA12] V. Vaughan,

Digital Modeling

(New Riders, 2012).

[VE16] Visible Earth, NASA Goddard Space Flight Center Image, accessed October 2018.

第7章 光照

光照以不同的方式影响着我们世界的外观，有时甚至是很戏剧化 的方式。当手电筒照射在物体上时，我们会期望它面向光线的一侧看 起来更亮。我们所居住的地球，在中午朝向太阳时候被照得很亮，但 随着它的自转，同一个地点的亮度会逐渐由白天转变为傍晚，直到午 夜变得完全黑暗。物体对光的反射也各不相同。物体除了颜色的差 别，也可以具有不同的反射特性。考虑两个物体，在都是绿色的情况 下，其中一个是布制的，而另一个是抛光钢材质的——那么后者看起 来会更“闪亮”。

7.1 光照模型

我们所观察到的光是高能量源发出的光子，经过反射直到一些光 子到达我们的眼睛的产物。不幸的是，在计算上模拟这个自然过程是 不可行的，因为这需要模拟并跟踪大量光子的运动，即向我们的场景 添加海量的对象（和矩阵）。因此，我们需要的是光照模型。

光照模型（Lighting model）有时也被称为着色模型（Shading model），在着色器编程存在的情况下，这可能有点令人困惑。有时又 使用术语反射模型（Reflection model），进一步使术语复杂化。我 们将尽力坚持使用简单而实用的术语。

现在最常见的光照模型称为“ADS”模型，因为它们基于标记为

A、D和S的3种类型的反射。

环境光反射（Ambient reflection）模拟低级光照，影响场景中 的所有物体。

漫反射（Diffuse reflection）根据光线的入射角度调整物体亮 度。

镜面反射（Specular reflection）用以展示物体的光泽，通过在 物体表面上，光线最直接地反射到我们的眼睛的位置，策略性地 放置适当大小的高光来实现。

ADS模型可用于模拟不同的光照效果和各种材质。 图7.1（见彩插）展示了位置光对于闪亮黄金环面的环境光反射、

漫反射和镜面反射分量。

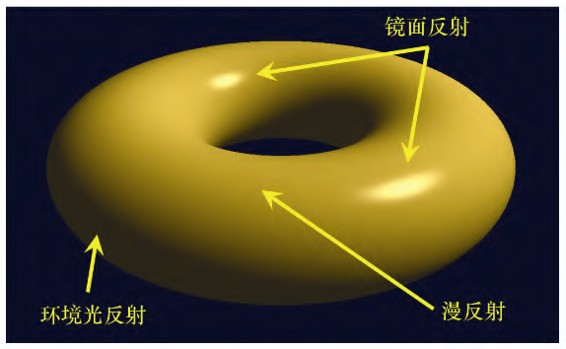


图7.1 ADS光照分量

回想一下，场景的绘制最终是由片段着色器为屏幕上的每个像素 输出颜色而实现的。使用ADS光照模型需要指定由于像素的RGBA输出值 上的光照而产生的分量。因素包括：

光源类型及其环境、漫反射和镜面反射特性；

对象材质的环境、漫反射和镜面反射特征； 对象的材质指定为“光泽”； 光线照射物体的角度； 从中查看场景的角度。

7.2 光源

光源有许多类型，每种光源具有不同的特性，需要不同的步骤来 模拟其效果。常见光源类型有：

全局光（通常称为“全局环境光”，因为它仅包含环境光组 件）；

定向光（或“远距离光”）； 位置光（或“点光源”）； 聚光灯。

全局环境光是最简单的光源模型。它没有光源位置——无论场景 中的对象在何处，其上的每个像素都有着相同的光照。全球环境光照 模拟了现实世界中的一种光线现象，即光线经过很多次反射，其光源 和方向都已经无法确定。全局环境光仅具有环境光反射分量，用RGBA 值设定；它没有漫反射或镜面反射分量。例如，全局环境光可以定义 如下：

float globalAmbient[4] = { 0.6f, 0.6f, 0.6f, 1.0f };

RGBA的取值范围为0～1，全局环境光通常被建模为偏暗的白光， 其中RGB各值设为0～1的相同的小数，alpha设置为1。

定向光或远距离光也没有源位置，但它具有方向。它可以用来模

拟光源距离非常远，以至于光线接近平行的情况，例如阳光。通常在 这种情况下，我们可能只对建模光照感兴趣，而对发光的物体不感兴 趣。定向光对物体的影响取决于光照角度，物体在朝向定向光的一侧 比在切向或相对侧更亮。建模定向光需要指定其方向（以向量形式）

及其环境、漫反射和镜面特征（以RGBA值）。指向

Z

轴负方向的红色定

向光可以指定如下：

float dirLightAmbient[4] = { 0.1f, 0.0f, 0.0f, 1.0f }; float dirLightDiffuse[4] = { 1.0f, 0.0f, 0.0f, 1.0f }; float dirLightSpecular[4] = { 1.0f, 0.0f, 0.0f, 1.0f }; float dirLightDirection[3] = { 0.0f, 0.0f, -1.0f };

在已经有全局环境光的情况下，定向光的环境光分量看起来似乎 是多余的。然而，当光源“开启”或“关闭”时，全局环境光和定向 光的环境光分量的区别就很明显了。当“开启”时，总环境光分量将 如预期的那样增加。上面的例子中，我们只使用了很小的环境光分 量。在实际场景中，应当根据场景的需要平衡两个环境光分量。

位置光在3D场景中具有特定位置。靠近场景的光源，例如台灯， 蜡烛等。像定向光一样，位置光的效果取决于撞击角度；但是，它没 有方向，因为它对场景中的每个顶点的光照方向都不同。位置光还可 以包含衰减因子，以模拟它们的强度随距离减小的程度。与我们看到 的其他类型的光源一样，位置光具有指定为RGBA值的环境光反射、漫

反射和镜面反射特性。位置（5,2,−3）处的红色位置光可以指定如下

例：

float posLightAmbient[4] = { 0.1f, 0.0f, 0.0f, 1.0f }; float posLightDiffuse[4] = { 1.0f, 0.0f, 0.0f, 1.0f }; float posLightSpecular[4] = { 1.0f,0.0f, 0.0f, 1.0f }; float posLightLocation[3] = { 5.0f, 2.0f, -3.0f };

衰减因子有多种建模方式。其中一种方式是使用恒定、线性和二

次方（分别称为 , 和

kc kl kq

）衰减，并引入非负可调参数。这些参数

与离光源的距离（

d

）结合进行计算：



将这个因子与光的强度相乘可以使距光更远时，光的强度衰减更

多。注意，

kc

应当永远设置为大于等于1的值，从而使得衰减因子落入

[0…1]区间，并当

d

增大时接近于0。

聚光灯（spotlight）同时具有位置和方向。其“锥形”效果可以

使用0°～90°的截光角

θ

来模拟，指定光束的半宽度，并使用衰减指

数来模拟随光束角度的强度变化。如图7.2所示，我们确定聚光灯方向

与从聚光灯到像素的向量之间的角度 。当

φ φ

小于 时，我们通过将

θ

的余弦提高到衰减指数来计算强度因子（当

φ φ

大于 时，强度因子

θ

设置为0）。结果是强度因子的范围为0～1。衰减指数会影响当角度

φ

增加时，强度因子趋于0的速率。然后将强度因子乘以光的强度以模拟 锥形效果。

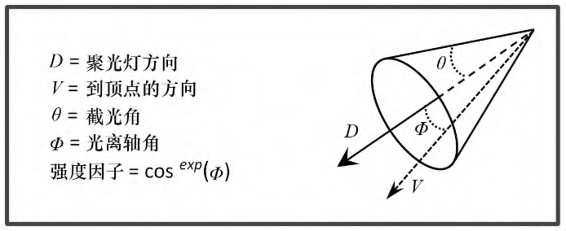


图7.2 聚光灯参数

位于（5,2,−3）向下照射

Z

轴负方向的红色聚光灯可以表示为：

float spotLightAmbient[4] = { 0.1f, 0.0f, 0.0f, 1.0f }; float spotLightDiffuse[4] = { 1.0f, 0.0f, 0.0f, 1.0f }; float spotLightSpecular[4] = { 1.0f,0.0f, 0.0f, 1.0f }; float spotLightLocation[3] = { 5.0f, 2.0f, -3.0f }; float spotLightDirection[3] = { 0.0f, 0.0f, -1.0f }; float spotLightCutoff = 20.0f;

float spotLightExponent = 10.0f;

聚光灯也可以引入衰减因子。我们没有在上面的代码中展示它 们，不过，聚光灯衰减因子可以用与前述定向光源相同的方式实现。

历史上，自1986年皮克斯的著名动画《小台灯》（

Luxo Jr.

）出现

起，聚光灯就成为了计算机图形学的标志。

当设计拥有许多光源的系统时，程序员应该考虑创建相应的类结 构，如定义Light类以及其子类GlobalAmbient、Directional、 Positional以及Spotlight。由于聚光灯同时具有定向光和位置光的特 性，这里就值得使用C++的多继承能力，让Spotlight类同时继承于实 现位置光和定向光的类。在示例中，由于内容足够简单，因此我们在 当前版本中没有加入这种层次结构。

7.3 材质

我们场景中物体的“外观”目前仅使用颜色和纹理进行表现。增

加的光照使得我们可以加入表面的反射特性。即对象如何与我们的ADS 光照模型相互作用。这可以通过将每个对象视为“由某种材质制成” 来建模。

通过指定4个值（我们已经熟悉其中3个值——环境光、漫反射和 镜面RGB颜色），可以在ADS光照模型中模拟材质。第四种叫作光泽， 正如我们将要看到的那样，它被用来为所选材质建立一个合适的镜面 高光。目前许多不同类型的常见材质已经有ADS和光泽度值了。例如， “锡镴”可以指定如下：

float pewterMatAmbient[4] = { .11f, .06f, .11f, 1.0f }; float pewterMatDiffuse[4] = { .43f, .47f, .54f, 1.0f }; float pewterMatSpecular[4] = { .33f, .33f, .52f, 1.0f }; float pewterMatShininess = 9.85f;

一些其他材质的ADS RGBA值见图7.3（引自[BA16]）。

有时候一些其他特性也属于材质特性。透明度由RGBA标准中的第 四个（alpha）通道的不透明度来实现。取值为1.0是表示完全不透 明，取值为0时表示完全透明。对于大多数材质而言，只需要把不透明 度设置为1.0就行了，但是对于某些特定的材质，加入一些透明度是很 重要的。例如，图7.3中材质“玉”和“珍珠”都含有少量透明度（取 值略微小于1.0）以显得更加真实。



图7.3 其他材质的ADS系数

放射性有时也包含在ADS材质规范中。在模拟自身发光的材质（例 如磷光材质）时非常有用。

没有纹理的物体在渲染时，通常需要指定材质特性。因此，预定 义一些可供选择的材质，在使用时会很方便。因此我们需要在 Utils.cpp文件中添加如下代码：

// 黄金材质 — 环境光、漫反射、镜面反射和光泽

float \* Utils::goldAmbient() { static float a[4] = { 0.2473f, 0.1995f, 0.0745f, 1 }; return

(float \* ) a; }

float \* Utils::goldDiffuse() { static float a[4] = { 0.7516f, 0.6065f, 0.2265f, 1 }; return

(float \* ) a; }

float \* Utils::goldSpecular() { static float a[4] = { 0.6283f, 0.5559f, 0.3661f, 1 }; return

(float \* ) a; }

float Utils::goldShininess() { return 51.2f; }

// 白银材质 — 环境光、漫反射、镜面反射和光泽

float \* Utils::silverAmbient() { static float a[4] = { 0.1923f,

0.1923f, 0.1923f, 1 }; return

(float \* ) a; }

float \* Utils::silverDiffuse() { static float a[4] = { 0.5075f, 0.5075f, 0.5075f, 1 }; return

(float \* ) a; }

float \* Utils::silverSpecular() { static float a[4] = { 0.5083f, 0.5083f, 0.5083f, 1 }; return

(float \* ) a; }

float Utils::silverShininess() { return 51.2f; }

// 青铜材质 — 环境光、漫反射、镜面反射和光泽

float \* Utils::bronzeAmbient() { static 0.1275f, 0.0540f, 1 }; return

(float \* ) a; }

float \* Utils::bronzeDiffuse() { static 0.4284f, 0.1814f, 1 }; return

(float \* ) a; }

float a[4] = { 0.2125f,

float a[4] = { 0.7140f,

float \* Utils::bronzeSpecular() { static float a[4] = { 0.3936f,

0.2719f, 0.1667f, 1 }; return (float \* ) a; }

float Utils::bronzeShininess() { return 25.6f; }

这样在init()函数中或全局中为物体指定“黄金”材质就非常容 易了，如下所示。

float\* matAmbient = Utils::goldAmbient(); float\* matDiffuse = Util::goldDiffuse(); float\* matSpecular = util.goldSpecular(); float matShininess = util.goldShininess();

注意，目前为止的各小节中，我们所用来实现的光照和材质特性 的代码并没有引入光照。这些代码仅仅提供了用于描述并存储场景中 元素所需光照和材质特性的一种方式。我们仍然需要自己计算光照。 编写计算光照的代码需要在我们的着色器代码中引入一些严肃的数学 过程。因此，让我们先来看看在C++/OpenGL和GLSL图形程序中实现ADS 光照的基础。

7.4 ADS光照计算

当我们绘制场景时，每个顶点坐标都会进行变换以将3D世界模拟

到2D屏幕上。每个像素的颜色都是光栅化、纹理贴图以及插值的结 果。现在我们需要加入一个新的步骤来调整这些光栅化之后的像素颜 色，以便反应场景中的光照和材质。我们需要做的基础ADS计算是确定 每个像素的反射强度（Reflection Intensity，I）。计算过程如下：

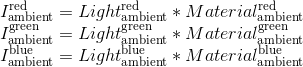


我们需要计算每个光源对于每个像素的环境光反射、漫反射和镜 面反射分量，并求和。当然，这些计算都基于场景内的光源类型以及 渲染中模型的材质类型。

环境光分量是最简单的。它的值是场景环境光与材质环境光分量 的乘积：



请记住光与材质亮度都是RGB值，计算可以更准确地描述为：



漫反射分量会更复杂一些，因为它基于光对于平面的入射角。朗 伯余弦定律（1760年出版）确定了表面反射的光量与光入射角的余弦 成正比。可以建模为如下公式：



与上面的计算相同，实际计算中所用到的是红、绿、蓝分量。

确定入射角

θ

需要（a）求解从所绘制向量到光源的向量（或者与

光照方向相反的向量），（b）求解所渲染物体表面的法（垂直）向

量。让我们将其分别称为

L

和 ，如图7.4所示。

N

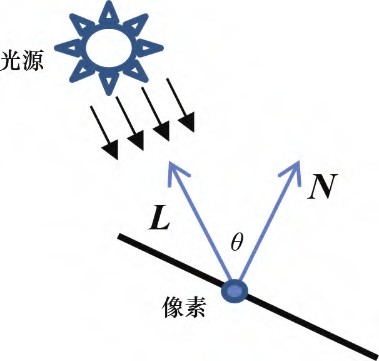


图7.4 入射角

基于场景中光的物理特性，向量

L

可以通过对光照方向向量取反，

或通过计算像素位置到光源位置的向量得到。计算向量

N

会麻烦一些

——法向量有可能已经在模型中给出了，但是如果模型没有给出法向

量 ，那么就需要基于周围顶点位置，在几何上对向量

N N

进行估计。在

本章剩下的内容中，我们假设所渲染的模型每个顶点都包含法向量

（使用建模工具如MAYA或Blender创建的模型，通常都包含法向量）。

事实上，在计算法向量时，没必要计算出

θ

角本身的角度。我们

真正需要的是cos(

θ

)。在第3章中讲过，这可以通过点乘计算得出。

因此，漫反射分量可以通过如下公式得出：



漫反射分量仅当表面暴露在光照中时起作用，即当−90 < <

θ

90，cos(

θ

) > 0时。因此，我们需要将之前等式的最右项替换为：



镜面反射分量决定所渲染的像素是否需要作为“镜面高光”的一 部分变亮。它不止与光源的入射角相关，也与光在表面上的反射角以 及观察点与反光表面之间的夹角相关。

在图7.5中，

R

代表光反射的方向，

V

（叫作观察向量view

vector）是从像素到眼睛的向量。注意，

V

是对从眼睛到像素的向量取

反（在相机空间中，眼睛位于原点）。在

R

与 之间的小夹角

V φ

越小，

眼睛越靠近光轴，或者说看向反射光，因此像素的镜面高光分量也就 越大（像素看来应该更亮）。

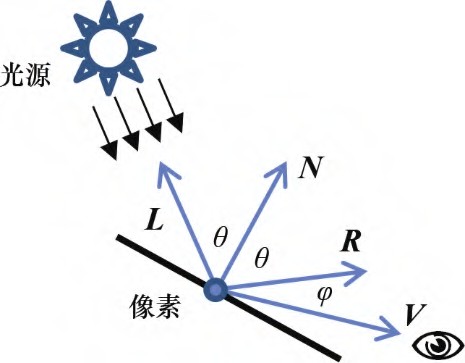


图7.5 观察点入射角

用于计算镜面反射分量的方式取决于所渲染物体的“光泽

φ

度”。极端闪亮的物体，如镜子，其镜面高光非常小——它们将入射 的光直接反射给了眼睛。不那么闪亮的物体，其镜面高光会扩散开 来，因此高光会包含更多的像素。

反光度通常用衰减函数来建模，这个衰减函数用来表达随着角度

的增大，镜面反射分量降低到0的速度。我们可以用cos(

φ φ

)来对衰

减进行建模，通过余弦函数的乘方来增减反光度，如cos( ),

φ

cos2(

φ

), cos3(

φ

), cos10(

φ

), cos50(

φ

)等，如图7.6所示。

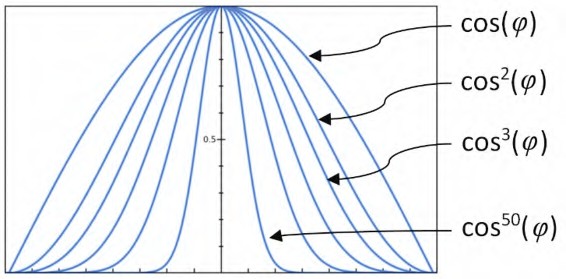


图7.6 以余弦指数建模的反光度

注意，指数中的阶数越高，衰减越快，因此在视角光轴外的反光

像素镜面反射分量越小。我们将衰减函数cosn(

φ

)中的指数

n

叫作材质

的反光度因子。注意在之前的图7.3中，每个材质的反光度因子在最右 列给出。

现在我们可以给出完整的镜面反射计算：



注意，与之前计算漫反射一样，我们使用了max()函数。在本例

中，我们需要确保镜面反射分量不使用cos(

φ

)所产生的负值，如果使

用了负值，则会有奇怪的伪影，如“暗”镜面高光。 同时，如之前一样，真正的计算中包含了红、绿、蓝3个分量。

7.5 实现ADS光照

在7.4节中所讲述的计算目前为止都是理论上的，其中包含的假设

是，我们可以对每个像素都实行这些操作。但是真实情况会更复杂，

通常模型中只有用来定义模型的顶点才有法向量（

N

），而非每个像素

都有。因此我们要么需要计算每个像素的法向量，这会非常耗时，要 么需要使用其他方法对所需的值进行估计，以实现足够好的效果。

其中一种途径称为“面片着色”或“平坦着色”。这里我们假定 所渲染图元（如多边形或三角形）中每个像素的光照值都一样。因此 我们只需要对模型每个多边形的一个顶点进行光照计算，然后以每个 多边形或每个三角形为基础，将计算结果的光照值复制到相邻的像素 中。

现在面片着色几乎已经不再使用，因为其渲染结果看来不够真 实，同时现代硬件已经可以进行更加精确的计算了。图7.7中展示了一 个面片着色环面的例子，其中每个三角形都作为平坦的反射表面。

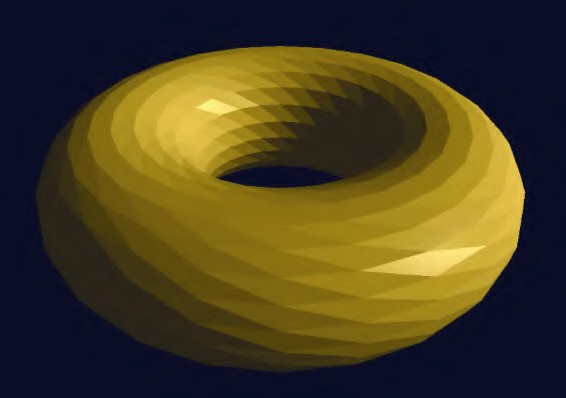


图7.7 面片着色的环面

虽然某些情况下，面片着色可能已经够用了（或者故意使用其效

果），但是通常“平滑着色”是一种更好的途径。在平滑着色的过程 中，会对每个像素计算光照强度。现代显卡的并行处理功能，以及 OpenGL图形管线中的插值渲染让平滑着色变得可行。

我们将会观察两个流行的平滑着色方法：Gouraud着色和Phong着 色。

7.5.1 Gouraud着色（双线性光强插值法）

法国计算机科学家Henri Gouraud在1971年发表的平滑着色算法后 来被称为Gouraud着色[GO71]。由于使用了3D图形管线（如OpenGL）中 的自动插值渲染，它特别适用于现代显卡。Gouraud着色过程如下。

（1）确定每个顶点的颜色，以及光照相关计算。

（2）允许正常的光栅化过程在插入像素时对颜色也进行插值（同 时也对光照进行插值）。

在OpenGL中，这表示大多数光照计算都是在顶点着色器中完成 的，片段着色器仅做传递并展示自动插值的光照后的颜色。

图7.8展示了在场景中包含环面和单一位置光的情况下，我们将会 用来在OpenGL中实现Gouraud着色器的策略。程序7.1中实现了这个策 略。



图7.8 实现Gouraud着色

程序7.1 位置光和Gouraud着色器下的环面

C++/OpenGL应用程序

. . .

#include "Torus.h"

#include "Utils.h"

. . .

// 用于创建着色器和渲染程序的声明，如前

// VAO、两个VBO以及环面的声明，如前

// 环面与相机位置的声明和赋值，如前

// Utils.cpp中现在已经添加有金、银、青铜材质

. . .

// 为display() 函数分配变量

GLuint mvLoc, projLoc, nLoc;

// 着色器统一变量中的位置 GLuint globalAmbLoc, ambLoc, diffLoc, specLoc, posLoc, mAmbLoc, mDiffLoc, mSpecLoc, mShiLoc;

glm::mat4 pMat, vMat, mMat, mvMat, invTrMat; glm::vec3 currentLightPos, lightPosV; *//* 在模型和视觉空间中的光照位 置，Vector3f类型

float lightPos[3]; *//* 光照位置的浮点数组

// 初始化光照位置

glm::vec3 initialLightLoc = glm::vec3(5.0f, 2.0f, 2.0f);

// 白光特性

float globalAmbient[4] = { 0.7f, 0.7f, 0.7f, 1.0f }; float lightAmbient[4] = { 0.0f, 0.0f, 0.0f, 1.0f }; float lightDiffuse[4] = { 1.0f, 1.0f, 1.0f, 1.0f }; float lightSpecular[4] = { 1.0f, 1.0f, 1.0f, 1.0f };

// 黄金材质特性

float\* matAmb = Utils::goldAmbient();

float\* matDif = Utils::goldDiffuse(); float\* matSpe = Utils::goldSpecular(); float matShi = Utils::goldShininess();

void setupVertices(void) {

// 该函数与之前章节中的相同，没有改动

// 下面的部分在这里出现是为了更清晰，现在我们将真的使用法向量

. . .

glBindBuffer(GL\_ARRAY\_BUFFER, vbo[2]); glBufferData(GL\_ARRAY\_BUFFER, nvalues.size() \* 4, &nvalues[0],

GL\_STATIC\_DRAW);

}

void display(GLFWwindow\* window, double currentTime) {

// 清除深度缓冲区，如之前例子中一样载入渲染程序

. . .

// 用于模型-视图变换、投影以及逆转置(法向量)矩阵的统一变量

mvLoc = glGetUniformLocation(renderingProgram, "mv\_matrix"); projLoc = glGetUniformLocation(renderingProgram,

"proj\_matrix"); nLoc = glGetUniformLocation(renderingProgram, "norm\_matrix");

// 初始化投影及视图矩阵，如前例

. . .

// 基于环面位置，构建模型矩阵

mMat = glm::translate(glm::mat4(1.0f), glm::vec3(torLocX, torLocY, torLocZ));

// 旋转环面以便更容易看到

mMat \*= glm::rotate(mMat, toRadians(35.0f), glm::vec3(1.0f, 0.0f, 0.0f));

// 基于当前光源位置，初始化光照

currentLightPos = glm::vec3(initialLightLoc.x, initialLightLoc.y, initialLightLoc.z);

installLights(vMat);

// 通过合并矩阵v和m，创建模型-视图(MV)矩阵，如前

mvMat = vMat \* mMat;

// 构建MV矩阵的逆转置矩阵，以变换法向量

invTrMat = glm::transpose(glm::inverse(mvMat));

// 将MV、PROJ以及逆转置(法向量)矩阵传入相应的统一变量 glUniformMatrix4fv(mvLoc, 1, GL\_FALSE, glm::value\_ptr(mvMat)); glUniformMatrix4fv(projLoc, 1, GL\_FALSE, glm::value\_ptr(pMat)); glUniformMatrix4fv(nLoc, 1, GL\_FALSE,

glm::value\_ptr(invTrMat));

// 在顶点着色器中，将顶点缓冲区(VBO #0)绑定到顶点属性#0 glBindBuffer(GL\_ARRAY\_BUFFER, vbo[0]); glVertexAttribPointer(0, 3, GL\_FLOAT, false, 0, 0); glEnableVertexAttribArray(0);

// 在顶点着色器中，将法向缓冲区(VBO #2)绑定到顶点属性#1 glBindBuffer(GL\_ARRAY\_BUFFER, vbo[2]); glVertexAttribPointer(1, 3, GL\_FLOAT, false, 0, 0); glEnableVertexAttribArray(1);

glEnable(GL\_CULL\_FACE); glFrontFace(GL\_CCW); glEnable(GL\_DEPTH\_TEST); glDepthFunc(GL\_LEQUAL);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, vbo[3]); glDrawElements(GL\_TRIANGLES, myTorus.getNumIndices(),

GL\_UNSIGNED\_INT, 0);

}

void installLights(glm::mat4 vMatrix) {

// 将光源位置转换为视图空间坐标，并存入浮点数组

lightPosV = glm::vec3(vMatrix \* glm::vec4(currentLightPos, 1.0));

lightPos[0] = lightPosV.x; lightPos[1] = lightPosV.y; lightPos[2] = lightPosV.z;

// 在着色器中获取光源位置和材质属性

globalAmbLoc = glGetUniformLocation(renderingProgram, "globalAmbient");

ambLoc = glGetUniformLocation(renderingProgram, "light.ambient");

diffLoc = glGetUniformLocation(renderingProgram, "light.diffuse");

specLoc = glGetUniformLocation(renderingProgram, "light.specular");

posLoc = glGetUniformLocation(renderingProgram, "light.position");

mAmbLoc = glGetUniformLocation(renderingProgram, "material.ambient");

mDiffLoc = glGetUniformLocation(renderingProgram, "material.diffuse");

mSpecLoc = glGetUniformLocation(renderingProgram, "material.specular");

mShiLoc = glGetUniformLocation(renderingProgram, "material.shininess");

// 在着色器中为光源与材质统一变量赋值

glProgramUniform4fv(renderingProgram,

globalAmbient); glProgramUniform4fv(renderingProgram, glProgramUniform4fv(renderingProgram,

lightDiffuse); glProgramUniform4fv(renderingProgram,

lightSpecular); glProgramUniform3fv(renderingProgram, glProgramUniform4fv(renderingProgram, glProgramUniform4fv(renderingProgram, glProgramUniform4fv(renderingProgram,

globalAmbLoc, 1,

ambLoc, 1, lightAmbient);

diffLoc, 1,

specLoc, 1,

posLoc, 1, lightPos);

mAmbLoc, 1, matAmb);

mDiffLoc, 1, matDif);

mSpecLoc, 1, matSpe);

glProgramUniform1f(renderingProgram, mShiLoc, matShi);

}

// init() 以及 main() 函数如前

程序7.1中的很多元素我们都已经熟悉了。首先，定义了环面、光 照和材质特性。接着将环面顶点以及相关法向量读入缓冲区。 display()函数与之前程序中的类似，在这里不同的是它同时也将光照 和材质信息传入顶点着色器。为了传入这些信息，它调用 installLights()，将光源在视觉空间中的位置，以及材质的ADS特 性，读入相应的统一变量以供着色器使用。注意，我们提前定义了这 些统一位置变量，以求更好的性能。

其中一个重要的细节是变换矩阵MV，用来将顶点位置移动到视觉 空间，但它并不总能正确地将法向量也调整进视觉空间。直接对法向 量应用MV矩阵不能保证法向量依然与物体表面垂直。正确的变换是MV 的逆转置矩阵，在第3章“补充说明”中有描述。在程序7.1中，这个 新增的矩阵叫作“invTrMat”，通过统一变量传入着色器。

变量lightPosV包含光源在相机空间中的位置。我们每帧只需要计 算一次，因此我们在installLights()中[在display()中调用]而非着 色器中计算。着色器在下方的续程序7.1中。其中顶点着色器使用了一

些我们目前没有见过的符号。注意，在顶点着色器最后进行了向量加

法——在第3章中有讲，并且在GLSL中可用。我们将会在展示着色器之 后讨论其他符号。

续程序7.1

顶点着色器

#version 430

layout (location=0) in vec3 vertPos; layout (location=1) in vec3 vertNormal; out vec4 varyingColor;

struct PositionalLight

{ vec4 ambient; vec4 diffuse; vec4 specular; vec3 position;

};

struct Material

{ vec4 ambient; vec4 diffuse; vec4 specular; float shininess;

};

uniform vec4 globalAmbient; uniform PositionalLight light; uniform Material material; uniform mat4 mv\_matrix;

uniform mat4 proj\_matrix;

uniform mat4 norm\_matrix; *//* 用来变换法向量

void main(void)

{ vec4 color;

*//* 将顶点位置转换到视觉空间

*//* 将法向量转换到视觉空间

*//* 计算视觉空间光照向量(从顶点到光源)

vec4 P = mv\_matrix \* vec4(vertPos,1.0);

vec3 N = normalize((norm\_matrix \* vec4(vertNormal,1.0)).xyz); vec3 L = normalize(light.position - P.xyz);

// 视觉向量等于视觉空间中的负顶点位置

vec3 V = normalize(-P.xyz);

// R是-L的相对于表面向量N的镜像 vec3 R = reflect(-L,N);

// 环境光、漫反射和镜面反射分量

vec3 ambient = ((globalAmbient \* material.ambient) + (light.ambient \* material.ambient)).xyz;

vec3 diffuse = light.diffuse.xyz \* material.diffuse.xyz \* max(dot(N,L), 0.0);

vec3 specular =

material.specular.xyz \* light.specular.xyz \* pow(max(dot(R,V), 0.0f), material.shininess);

*//* 将颜色输出发送到片段着色器

varyingColor = vec4((ambient + diffuse + specular), 1.0);

*//* 将位置发送到片段着色器，如前

gl\_Position = proj\_matrix \* mv\_matrix \* vec4(vertPos,1.0);

}

片段着色器

#version 430

in vec4 varyingColor; out vec4 fragColor;

*//* 与顶点着色器相同的统一变量

*//* 但并不直接在当前片段着色器使用

struct PositionalLight

{ vec4 ambient; vec4 diffuse; vec4 specular; vec3 position;

};

struct Material

{ vec4 ambient; vec4 diffuse; vec4 specular; float shininess;

};

uniform vec4 globalAmbient; uniform PositionalLight light; uniform Material material; uniform mat4 mv\_matrix;

uniform mat4 proj\_matrix; uniform mat4 norm\_matrix;

void main(void)

{ fragColor = varyingColor;

}

程序7.1的输出如图7.9所示。



图7.9 Gouraud着色的环面

顶点着色器代码中有我们第一次使用了结构体语法的示例。 GLSL“结构体”就像一个数据类型，它有名称和一组字段。当使用结 构体名称声明变量时，这个变量将包含结构体中声明的字段，并可以 通过“.”语法访问字段。例如，变量“light”声明为 “PositionalLight”类型，因此我们可以在其后引用其字段 light.ambient，light.diffuse等。

还要注意字段选择器符号“.xyz”，我们在顶点着色器中的多个 地方都使用了这种语法。这是将vec4转换为仅包含其前3个元素的等效 vec3的快捷方式。

绝大多数光照计算发生在顶点着色器中。对于每个顶点，将适当

的矩阵变换应用于顶点位置和相关的法向量，并计算用于光方向（ ）

L

和反射（

R

）的向量。然后执行7.4节中描述的ADS计算，得到每个顶点

的颜色（代码中名为varyingColor）。颜色作为正常光栅化过程的一

部分进行插值。之后片段着色器仅作为简单传递。冗长的统一变量声 明列表也在片段着色器中（由于前面第4章中描述的原因），但实际上 并没有在那里使用它们。

注意GLSL函数normalize()，它用来将向量转换为单位长度。正确 地进行点积运算必须要先使用该函数。reflect()函数则计算一个向量 基于另一个向量的反射。

图7.9输出的环面中有很明显的伪影。其镜面高光有着块状、面片 感。这种伪影在物体移动时会更加明显（但我们在书中没法展示移动 的物体）。

Gouraud着色也容易受到其他伪影影响。如果镜面高光整个范围都 在模型中的一个三角形内——即高光范围内一个模型顶点也没有—— 那么它可能不会被渲染出来。由于镜面反射分量是依顶点计算的，因 此，当模型所有顶点都没有镜面反射分量时，其光栅化后的像素也不 会有镜面反射光。

7.5.2 Phong着色

Bui Tuong Phong在犹他大学的研究生期间开发了一种平滑的着色 算法，在1973年的论文[PH73]中对其进行了描述，并在[PH75]中发表。 该算法的结构类似于Gouraud着色的算法，其不同之处在于光照计算是

按像素而非顶点完成。由于光照计算需要法向量

N

和光向量

L

，但在模

型中仅顶点包含这些信息，因此Phong着色通常使用巧妙的“技巧”来

实现，其中

N

和 在顶点着色器中进行计算，并在光栅化期间插值。图

L

7.10概述了此策略。

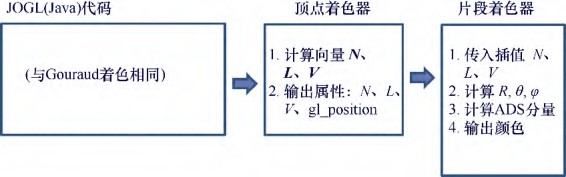


图7.10 实现Phong着色

C++/OpenGL代码完全如前。之前部分在顶点着色器中完成的过程 现在回放入片段着色器中进行。法向量插值的效果如图7.11所示。

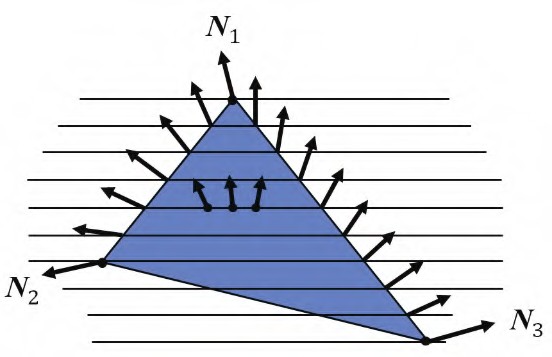


图7.11 法向量插值

现在我们已经准备好使用Phong着色实现位置光照射下的环面了。 大多数代码与实现Gouraud着色的代码相同。由于C++/OpenGL代码完全 没有改变，在此我们只展示修改过的顶点着色器和片段着色器，见程 序7.2。程序7.2的输出如图7.12所示，Phong着色修正了Gouraud着色 中出现的伪影。



图7.12 Phong着色的环面

程序7.2 Phong着色的环面

顶点着色器

#version 430

layout (location=0) in vec3 vertPos; layout (location=1) in vec3 vertNormal;

out vec3 varyingNormal; // 视觉空间顶点法向量

out vec3 varyingLightDir; // 指向光源的向量

out vec3 varyingVertPos; // 视觉空间中的顶点位置

// 结构体和统一变量与Gouraud着色相同

. . .

void main(void)

{ // 输出顶点位置、光照方向和法向量到光栅器以进行插值

varyingVertPos=(mv\_matrix \* vec4(vertPos,1.0)).xyz; varyingLightDir = light.position - varyingVertPos; varyingNormal=(norm\_matrix \* vec4(vertNormal,1.0)).xyz;

gl\_Position=proj\_matrix \* mv\_matrix \* vec4(vertPos,1.0);

}

片段着色器

#version 430

in vec3 varyingNormal;

in vec3 varyingLightDir; in vec3 varyingVertPos; out vec4 fragColor;

// 结构体和统一变量与Gouraud着色相同

. . .

void main(void)

{ // 正规化光照向量、法向量、视觉向量

vec3 L = normalize(varyingLightDir); vec3 N = normalize(varyingNormal); vec3 V = normalize(-varyingVertPos);

// 计算光照向量基于N的反射向量

vec3 R = normalize(reflect(-L, N));

// 计算光照与平面法向量间的角度

float cosTheta = dot(L,N);

// 计算视觉向量与反射光向量的角度

float cosPhi = dot(V,R);

// 计算ADS分量(按像素)，并合并以构建输出颜色

vec3 ambient = ((globalAmbient \* material.ambient) + (light.ambient \* material.ambient)).xyz;

vec3 diffuse = light.diffuse.xyz \* material.diffuse.xyz \* max(cosTheta,0.0);

vec3 specular =

light.specular.xyz \* material.specular.xyz \* pow(max(cosPhi,0.0), material.shininess);

fragColor = vec4((ambient + diffuse + specular), 1.0);

}

虽然Phong着色有着比Gouraud着色更真实的效果，但这是建立在 增大性能消耗的基础上的。James Blinn在1977年提出了一种对于 Phong着色的优化方法[BL77]，被称为Blinn-Phong反射模型。这种优化

是基于观察到Phong着色中消耗最大的计算之一是解出反射向量 。

R

Blinn发现向量

R

在计算过程中并不是必需的——

R

只是用来计算

角 的手段。角

φ φ

的计算可以不用向量

R

，而通过 与

L V

的角平分线向

量 得到。如图7.13所示， 和

H H N

之间的角

α

刚好等于1⁄2(

φ

)。虽然

α

与 不同，但Blinn展示了使用 代替

φ α φ

就已经可以获得足够好的结

果。

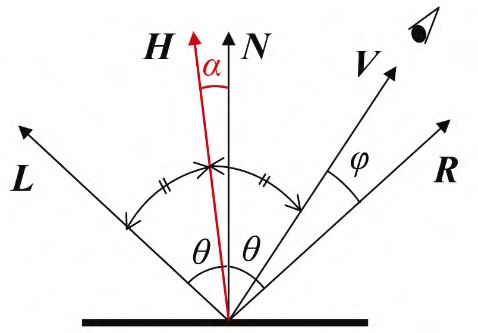


图7.13 Blinn-Phong反射

角平分线向量可以简单地使用

L

+ 得到（见图7.14），之后

V

cos(

α

)可以通过 的点积计算。

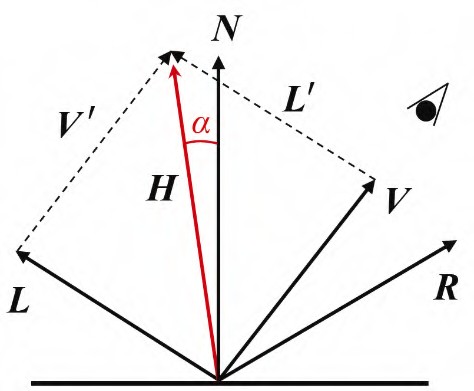


图7.14 Blinn-Phong计算

这些计算可以在片段着色器中进行，甚至为了性能考虑（经过一 些调整）也可以在顶点着色器中进行。图7.15展示了使用Blinn-Phong 着色的环面。它在图形质量上几乎与Phong渲染相同，同时节省了大量 性能损耗。



图7.15 Blinn-Phong着色的环面

程序7.3中展示了修改后顶点着色器和片段着色器，它们用来将程 序7.2中的Phong着色示例转换为Blinn-Phong着色。C++ / OpenGL代码 与之前一样没有变化。

程序7.3 Blinn-Phong着色的环面

顶点着色器

. . .

// 角平分线向量 H 作为新增的输出

out vec3 varyingHalfVector;

. . .

void main(void)

{ // 与之前的计算相同，增加了L+V的计算

varyingHalfVector = (varyingLightDir + (-varyingVertPos)).xyz;

// (其余顶点着色器代码没有改动)

}

片段着色器

. . .

in vec3 varyingHalfVector;

. . .

void main(void)

{ // 注意，现在已经不需要在片段着色器中计算R

vec3 L = normalize(varyingLightDir); vec3 N = normalize(varyingNormal);

vec3 V = normalize(-varyingVertPos); vec3 H = normalize(varyingHalfVector);

. . .

*//* 计算法向量N与角平分线向量H之间的角度

float cosPhi = dot(H,N);

// 角平分线向量H已经在顶点着色器中计算过，并在光栅器中进行过插值

vec3 ambient = ((globalAmbient \* material.ambient) + (light.ambient \* material.ambient)).xyz;

vec3 diffuse = light.diffuse.xyz \* material.diffuse.xyz \* max(cosTheta,0.0);

vec3 specular =

light.specular.xyz \* material.specular.xyz \* pow(max(cosPhi,0.0), material.shininess\*3.0);

// 最后乘以3.0作为改善镜面高光的微调

fragColor = vec4((ambient + diffuse + specular), 1.0);

}

图7.16（见彩插）所示的两个例子展示了Phong着色应用在比较复 杂的外部软件生成模型上所产生的效果。图7.16上图展示了Jay Turberville在Studio 522 Productions [TU16]创建的OBJ格式海豚模 型的渲染图。图7.16下图是著名的“斯坦福龙”的渲染，斯坦福龙是 1996年对一个小模型进行3D扫描所得到的模型[ST96]。两个模型都使用 我们放在“Utils.cpp”文件中的“黄金”材质进行渲染。斯坦福龙因 其大小而被广泛用于测试图形算法和硬件——它包含超过800 000个三 角形。





图7.16 Phong着色的外部模型

7.6 结合光照与纹理

目前为止，在光照模型中，都是假设我们使用按ADS定义的光源， 照亮按ADS定义材质的物体。但是，正如我们在第5章中所讲的，某些 对象的表面可能会指定纹理图像。因此，我们需要一种方法来结合采 样纹理所得的颜色和光照模型产生的颜色。

我们结合光照和纹理的方式取决于物体的特性以及其纹理的目 的。这里有多种情况，其中常见的有：

纹理图像很写实地反映了物体真实的表面外观；

物体同时具有材质和纹理；

材质包括了阴影和反射信息（在第8章、第9章中）； 有多种光和/或多个纹理。

我们先来观察第一种情景，物体拥有一个简单的纹理，同时我们 对它进行光照。实现这种光照的一种简单方法是在片段着色器中完全 将材质特性去除掉，之后使用纹理取样所得纹理颜色代替材质的ADS 值。下面的伪代码展示了这种策略：

fragColor = textureColor \* ( ambientLight + diffuseLight ) + specularLight

这种策略下，纹理颜色影响了环境光和漫反射分量，而镜面反射 颜色仅由光源决定。镜面反射分量仅由光源决定是一种很常见的做 法，尤其是对于金属或“闪亮”的表面。但是，对于不那么闪亮的表 面，如织物或未上漆的木材（甚至一小部分金属，如黄金），其镜面 高光部分都应当包含物体表明颜色。在这些情况下，之前的策略应该 做适当微调：

fragColor = textureColor \* ( ambientLight + diffuseLight + specularLight )

同时也有一些情况下，物体本身具有ADS材质，并伴有纹理图像。 如银质物体使用纹理为表面添加一些氧化痕迹。在这些情况下，如之 前章节中所讲过的，既用到光照又用到材质的标准ADS模型就可以与纹 理颜色相结合，并加权求和。如：

textureColor = texture(sampler, texCoord)

lightColor = (ambLight \* ambMaterial) + (diffLight \* diffMaterial)

+ specLight

fragColor = 0.5 \* textureColor + 0.5 \* lightColor

这种策略结合了光照、材质、纹理，并能够扩展到多个光源以及 多种材质的情况。如：

texture1Color = texture(sampler1, texCoord) texture2Color = texture(sampler2, texCoord)

light1Color = (ambLight1 \* ambMaterial) + (diffLight1 \* diffMaterial) + specLight1

light2Color = (ambLight2 \* ambMaterial) + (diffLight2 \* diffMaterial) + specLight2

|  |  |  |
| --- | --- | --- |
| fragColor = | 0.25 | \* texture1Color |
| + | 0.25 | \* texture2Color |
| + | 0.25 | \* light1Color |
| + | 0.25 | \* light2Color |

图7.17（见彩插）展示了拥有UV映射纹理图像（来自Jay Turberville[TU16]）的Studio 522海豚，以及我们之前在第6章见过的 NASA航天飞机模型。这两个有纹理的模型都使用了增强后的Blinn- Phong光照，没有使用材质，并在镜面高光中仅使用光照进行计算。在 这两幅图中，片段着色器中颜色相关的计算为：

vec4 texColor = texture(sampler, texCoord);

fragColor = texColor \* (globalAmbient + lightAmb + lightDiff \* max(dot(L,N),0.0))

+ lightSpec \* pow(max(dot(H,N),0.0), matShininess\*3.0);

注意，计算过程中fragColor可能产生大于1.0的值。在这种情况 下，OpenGL会将它限制回1.0。



图7.17 结合光照与纹理

补充说明

图7.7所展示的面片着色的环面是通过在顶点着色器和片段着色器 中，将“flat”插值限定符添加到相应的法向量属性声明中得到的。 这样会使得光栅器不对所限定的变量进行插值，而是直接将相同的值 赋给每个片段（在默认情况下，它会选择三角形第一个顶点上的 值）。在Phong着色示例代码中，可以通过如下修改实现面片着色：

在顶点着色器中

flat out vec3 varyingNormal;

在片段着色器中

flat in vec3 varyingNormal;

我们还没有讨论的一类很重要的光是分布式光（distributed light）（或区域光（area light）），这种光的光源是一片区域而非

一个单点。它在现实世界相对应的例子是通常在办公室或教室中的日

光灯管。有兴趣的读者可以在[MH02]找到更多有关区域光的详细信息。

历史记录

在本章中我们过度简化了Gouraud和Phong的一些术语。Gouraud着 色归功于Gouraud——通过计算顶点上光的强度并使用光栅器对光强进 行插值以生成平滑的曲面外观（有时也被称为“平滑着色”）。Phong 着色则归功于Phong，这是另一种平滑着色，对法向量插值并计算每个 像素的光照。Phong同时也被认为是成功将镜面高光纳入平滑着色的先 驱者。因此，ADS光照模型在计算机图形学中也通常被称为Phong反射 模型。因此，我们例子中的Gouraud着色准确地来说是使用了Phong反 射模型的Gouraud着色。由于Phong的反射模型在3D图形编程中非常普 及，通常Gouraud着色模型都是在Phong反射模型中进行展示。不过这 可能会引起误会，因为原本Gouraud在1971年的工作中并没有任何镜面 反射分量。

习题

7.1 （项目）修改程序7.1以使光能随鼠标而移动。在实现这个 功能之后，四处移动鼠标，并记录下镜面高光的移动以及Gouraud着色 伪影的出现。你可能会需要在光源处渲染一个点（或者小物体）以便 完成该项目。

7.2 在程序7.2中重复练习7.1的内容。这里应该只需要将Phong

着色的着色器放入练习7.1的解决方案中。从Gouraud着色到Phong着色 的进步在光四处移动时应当更明显。

7.3 （项目）修改程序7.2以使其包括两个位于不同位置的位置 光。片段着色器需要混合每个光的漫反射和镜面反射分量。尝试使用 与7.6节所示相似的加权求和方法。你可以尝试简单地将它们加起来并 限制结果不超出光照值的上限。

7.4 （研究和项目）将程序7.2中的位置光替换为7.2节中所描述 的探照灯。尝试设置不同的遮光角、衰减指数并观察其效果。

# 参考资料

[BA16] N. Barradeu, accessed October 2018.

[BL77] J. Blinn, “Models of Light Reflection for Computer Synthesized Pictures,”

Proceedings of the 4th

Annual Conference on Computer Graphics and Interactive

, 1977.

Techniques

[DI16]

Luxo Jr.

(Pixar – copyright held by Disney),

accessed October 2018.

[GO71] H. Gouraud, “Continuous Shading of Curved

Surfaces,”

IEEE Transactions on Computers

C-20, no. 6 (June

1971).

[MH02] T. Akenine-Möller and E. Haines,

Real-Time

Rendering

, 2nd ed. (A. K. Peters, 2002).

[PH73] B. Phong, “Illumination of Computer-Generated Images” (PhD thesis, University of Utah, 1973).

[PH75] B. Phong, “Illumination for Computer Generated

Pictures,”

Communications of the ACM

18, no. 6 (June 1975):

311-317.

[ST96] Stanford Computer Graphics Laboratory, 1996, accessed October 2018.

[TU16] J. Turberville, Studio 522 Productions, Scottsdale, AZ.

第8章 阴影

8.1 阴影的重要性

在第7章中，我们学会了如何为3D场景添加光照。但是，我们并没 有真的添加光线，而是模拟光照在物体上的效果——使用ADS模型—— 并相应地调整这些物体的绘制方式。

当我们用这种方法照亮同一个场景中的多个物体时，它的局限性 就体现出来了。考虑图8.1所示的场景，其中包含了砖块纹理环面以及 地平面（地平面是一个巨大立方体的顶部，使用了来自[LU16]的草地纹 理）。

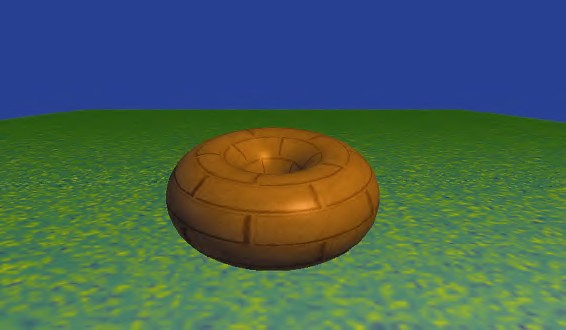


图8.1 没有阴影的场景

一眼望去我们的场景好像没问题。但是，仔细观察会发现有什么 重要的东西没有出现。具体来说，就是我们没有办法分辨出环面距离

它下方纹理立方体的距离。环面究竟是浮在立方体上面呢，还是放置

在立方体顶部呢？

我们无法回答这个问题的原因正是因为场景中缺乏阴影。我们期 望看到阴影，因为大脑需要通过阴影，才能针对我们所看到的物体以 及他们的位置关系构建完整的心理模型。

考虑图8.2所示的同样的场景，不过添加了阴影。现在就很明显 了，左图中环面放在地平面上；而右图中，环面则浮于其上。



图8.2 带阴影的光照

8.2 投影阴影

为了给3D场景添加阴影，人们设计了许多有趣的方法。其中一种 很适合在地平面上（如图8.1所示）绘制阴影，又相对不需要太大计算 代价的方法，叫作投影阴影（projective shadows）。给定一个位于

（ , ,

XL YL ZL

）的点光源、一个需要渲染的物体以及一个投射阴影的平

面，可以通过生成一个变换矩阵，将物体上的点（

XW

, ,

YW ZW

）变换为

相应阴影在平面上的点（

XS

,0,

ZS

）。之后将其生成的“阴影多边形”

绘制出来，通常使用暗色物体与地平面纹理混合作为其纹理，如图8.3

所示。

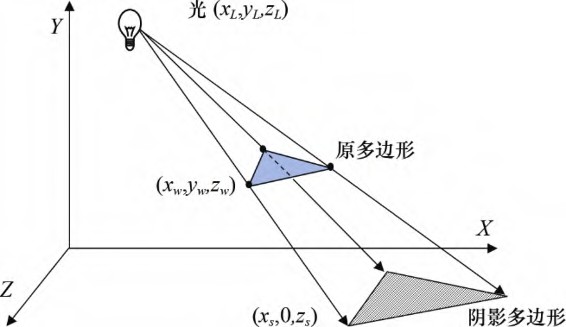


图8.3 投影阴影

使用投影阴影进行投射的优点是它的高效和易于实现。但是，它 仅适用于平坦表面——这种方法无法投射阴影于曲面或其他物体。即 使如此，它仍然适用于有室外场景并对性能要求较高的应用，很多游 戏中的场景都属于这类。

投影阴影变换矩阵的发展在[BL88], [AS14]以及[KS16]中有讨论。

8.3 阴影体

Franklin C. Crow在1977年提出了另一个重要的方法，这个方法 先找到被物体阴影覆盖的阴影体，之后减少视体与阴影体相交部分中 的多边形的颜色强度。图8.4展示了阴影体中的立方体，因此，立方体 绘制时会更暗。

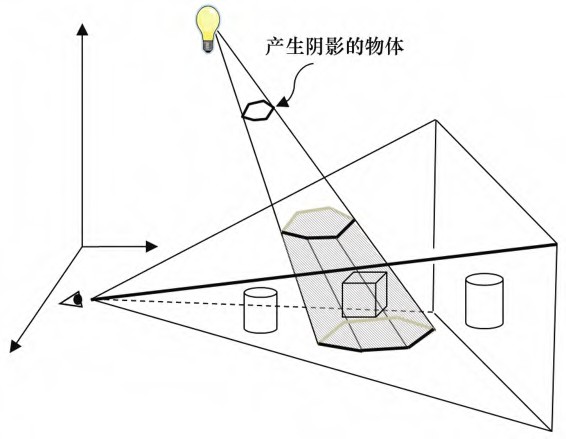


图8.4 阴影体

阴影体的优点在于其高度准确，比起其他方法来更不容易产生伪 影。但是，计算出阴影体以及每个多边形是否在其中这件事，即使对 于现代GPU来说，计算代价也很大。几何着色器可以用于计算阴影体， 模板缓冲区[[1]](#_bookmark157)可以用于判断像素是否在阴影体内。有些显卡对于特定 的阴影体操作优化提供了硬件支持。

8.4 阴影贴图

阴影贴图是用于投射阴影最实用也最流行的方法之一。虽然它并 不总是像阴影体一样准确（且通常伴随着讨厌的伪影），但阴影贴图 实现起来更简单，可以在各种情况下使用，并享有强大的硬件支持。

如果我们不在这里澄清前一段中的“更简单”这个词，那将是我 们的疏忽。虽然阴影贴图比阴影体（在概念和实践中）更简单，但它 绝不“简单”！对学生来说，通常在3D图形课程中最难实现的技术之 一就是阴影贴图。着色器程序本质上很难调试，阴影贴图需要几个组

件和着色器模块的完美协调。请注意，通过使用前面2.2节中描述的调

试工具，可以极大地促进阴影贴图的成功实现。

阴影贴图基于一个非常简明的想法：光线无法看到的任何东西都 在阴影中。也就是说，如果对象＃1阻挡光到达对象＃2，等同于光不 能“看到”对象＃2。

这个想法的强大之处在于我们已经有了方法来确定物体是否可以 被“看到”——使用Z缓冲区的隐藏面消除算法（HSR），如2.1.7节所 述。因此，计算阴影的策略是，暂时将摄像机移动到光的位置，应用Z 缓冲区HSR算法，然后使用生成的深度信息来计算阴影。

因此，渲染场景需要两轮：第1轮从灯光的角度渲染场景（但实际 上没有将其绘制到屏幕上），第2轮从摄像机的角度渲染场景。第1轮 的目的是从光的角度生成Z缓冲区。完成第1轮之后，我们需要保留Z缓 冲区并使用它来帮助我们在第2轮生成阴影。第2轮实际绘制场景。

我们的策略可以更加精炼。

（第1轮）从灯光的位置渲染场景。然后，对于每个像素，深度缓 冲区包含光与最近的对象之间的距离。 将深度缓冲区复制到单独的“阴影缓冲区”。

（第2轮）正常渲染场景。对于每个像素，在阴影缓冲区中查找相 应的位置。如果相机到渲染点的距离大于从阴影缓冲区检索到的 值，则在该像素处绘制的对象离光线的距离，比离光线最近的对 象更远，因此该像素处于阴影中。

当发现像素处于阴影中时，我们需要使其更暗。一种简单而有效

的方法是仅渲染其环境光，忽略其漫反射和镜面反射分量。

上述方法通常被称为“阴影缓冲区”。而当我们在第二步中，将 深度缓冲区复制到纹理中，则称为“阴影贴图”。当纹理对象用于储 存阴影深度信息时，我们称其为阴影纹理，OpenGL通过 sampler2DShadow类型支持阴影纹理（稍后讨论）。这样，我们就可以 利用片段着色器中纹理单元和采样器变量（即“纹理贴图”）的硬件 支持功能，在第2轮快速执行深度查找。我们现在修改的策略是：

（第1轮）与之前相同； 将深度缓冲区的内容复制进纹理对象；

（第2轮）与之前相同，不过阴影缓冲区变为阴影纹理。 现在我们来实现这些步骤。

8.4.1 阴影贴图（第1轮）——从光源位置“绘 制”物体

在第一步中，我们首先将相机移动到灯光的位置然后渲染场景。 我们的目标不是在显示器上实际绘制场景，而是完成足够的渲染过程 以正确填充深度缓冲区。因此，没有必要为像素生成颜色，我们的第 一遍将仅使用顶点着色器，但片段着色器不执行任何操作。

当然，移动相机需要构建适当的观察矩阵。根据场景的内容，我 们需要在光源处依合适的方向来看场景。通常，我们希望此方向朝向

最终在第2轮中呈现的区域。

这个方向通常依场景而定——在我们的场景中，我们通常会将相 机从光源指向原点。

第1轮中有几个需要处理的重要细节。

配置缓冲区和阴影纹理。 禁用颜色输出。

从光源到视野中的物体构建一个LookAt矩阵。 启用GLSL第1轮着色器程序，该程序仅包含图8.5中的简单顶点着 色器，准备接收MVP矩阵。在这种情况下，MVP矩阵将包括对象的

模型矩阵

M

、前一步中计算的LookAt矩阵（作为观察矩阵

V

），以

及透视矩阵

P

。我们将该MVP矩阵称为“shadowMVP”，因为它是基

于光而不是相机的观察点。由于实际上没有显示来自光源的视 图，因此第1轮着色器程序的片段着色器不会执行任何操作。

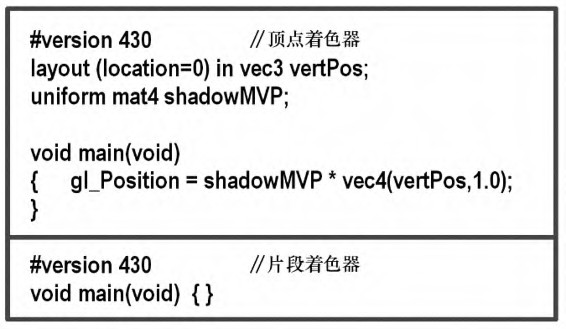


图8.5 阴影贴图第1轮的顶点着色器和片段着色器

为每个对象创建shadowMVP矩阵，并调用glDrawArrays()。第1轮 中不需要包含纹理或光照，因为对象不会渲染到屏幕上。

8.4.2 阴影贴图（中间步骤）——将Z缓冲区复制

到纹理

OpenGL提供了两种将Z缓冲区深度数据放入纹理单元的方法。第一 种方法是生成空阴影纹理，然后使用命令glCopyTexImage2D()将活动 深度缓冲区复制到阴影纹理中。

第二种方法是在第1轮中构建一个“自定义帧缓冲区”（而不是使 用默认的Z缓冲区），并使用命令glFrameBufferTexture()将阴影纹理 附加到它上面。OpenGL在3.0版中引入该命令，以进一步支持阴影纹 理。使用这种方法时，无须将Z缓冲区“复制”到纹理中，因为缓冲区 已经附加了纹理，深度信息由OpenGL自动放入纹理中。我们将在实现 中使用这种方法。

8.4.3 阴影贴图（第2轮）——渲染带阴影的场景

第2轮中的大部分内容与我们在第7章中看到的类似，即我们在这 里渲染完整的场景及其中的所有物体，以及光照、材质和装饰场景中 物体的纹理。同时，我们还需要添加必要的代码，以确定每个像素是 否在阴影中。

第2轮的一个重要特征是它使用了两个MVP矩阵。一个是将对象坐 标转换为屏幕坐标的标准MVP矩阵（如我们之前的大多数示例所示）。 另一个是在第1轮中生成的shadowMVP矩阵，用于从光源的角度进行渲 染——现在将在第2轮中用于从阴影纹理中查找深度信息。

在第2轮中，从纹理贴图尝试查找像素时，情况比较复杂。OpenGL

相机使用[−1…+ 1]坐标空间，而纹理贴图使用[0…1]空间。常见的

解决方案是构建一个额外的矩阵变换，通常称为

B

，它将用于从摄像机

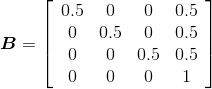
空间到纹理空间的转换（或“偏离”，biases，因此名称）。得到 的

B

过程很简单——先缩放为1/2，再平移1/2。

矩阵 如下：

B



之后将

B

合并入shadowMVP矩阵以备在第2轮中使用，如下：



假设我们使用阴影纹理附加到我们的自定义帧缓冲区的方法， OpenGL提供了一些相对简单的工具，用于确定绘制对象时，像素是否 处于阴影中。以下是第二阶段处理的详细信息摘要。

构建变换矩阵B，用于从光照空间转换到纹理空间[更合适在 init()中进行]。

启用阴影纹理以进行查找。 启用颜色输出。

启用GLSL第2轮渲染程序，包含顶点着色器和片段着色器。 根据摄像机位置（正常）为正在绘制的对象构建MVP矩阵。

构建shadowMVP2矩阵（包含

B

矩阵，如前所述）——着色器将需要

用它查找阴影纹理中的像素坐标。 将生成的矩阵变换发送到着色器统一变量。

像往常一样启用包含顶点、法向量和纹理坐标（如果使用）的缓

冲区。 调用glDrawArrays()。

除了渲染任务外，顶点和片段着色器还需要额外承担一些任务。

顶点着色器将顶点位置从相机空间转换为光照空间，并将结果坐 标发送到顶点属性中的片段着色器，以便对它们进行插值。这样 片段着色器可以从阴影纹理中检索正确的值。 片段着色器调用textureProj()函数，该函数返回0或1，指示像素 是否处于阴影中（所涉及的机制将在后面解释）。如果它在阴影 中，则着色器通过剔除其漫反射和镜面反射分量来输出更暗的像 素。

阴影贴图是一种常见任务，因此GLSL为其提供了一种特殊类型的 采样器变量，称为sampler2DShadow（如前所述），可以附加到C++ / OpenGL应用程序中的阴影纹理。textureProj()函数用于从阴影纹理中 查找值，它类似于我们之前在第5章中看到的texture()，其区别是除 了textureProj()函数使用vec3来索引纹理而不是通常的vec2。由于像 素坐标是vec4，因此需要将其投影到2D纹理空间上，以便在阴影纹理 贴图中查找深度值。正如我们将在下面看到的，textureProj()为完成 了这些功能。

顶点着色器和片段着色器代码的其余部分实现了Blinn-Phong着 色。这些着色器如图8.6和图8.7所示，并增加了阴影贴图的代码。

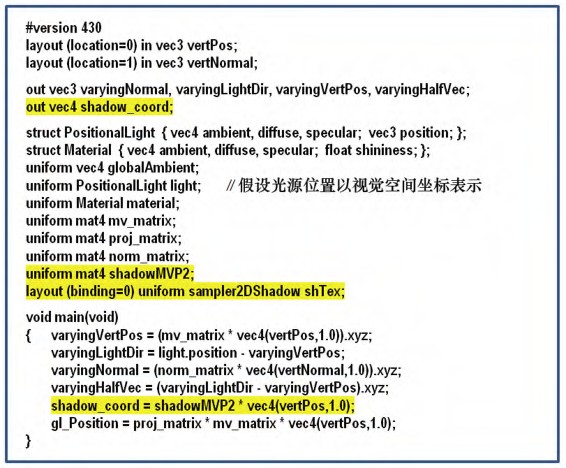


图8.6 阴影贴图第2轮顶点着色器

让我们更仔细地研究一下如何使用OpenGL来执行正在渲染的像素 和阴影纹理中的值之间的深度比较。首先，从顶点着色器开始，在模 型空间中使用顶点坐标，我们将其与shadowMVP2相乘以生成阴影纹理 坐标，这些坐标对应于投影到光照空间中的顶点坐标，是之前从光源

的视角生成的。经过插值后的（3D）光照空间坐标（ ,

x y

, ）在片段

z

着色器中使用如下。

z

分量表示从光到像素的距离。（ ,

x y

）分量用于

检索存储在（2D）阴影纹理中的深度信息。将该检索的值（到最靠近

光的物体的距离）与

z

进行比较。该比较产生“二元”结果，告诉我们

我们正在渲染的像素是否比最接近光的物体离光更远（即像素是否处 于阴影中）。

假设光源位置以视觉空间坐标表示。

与顶点着色器相同的结构体和统一变量。

如果我们在OpenGL中使用前面介绍过的glFrameBufferTexture() 并启用深度测试，然后使用片段着色器（见图8.7）的 sampler2DShadow和textureProj()，所渲染的结果将完全满足我们的 需求。即textureProj()将输出0.0或1.0，具体取决于深度比较。基于 此值，当像素离光源比离光源最近的物体更远时，我们可以在片段着 色器中忽略漫反射和镜面反射分量，从而有效地创建阴影。概述如图 8.8所示。

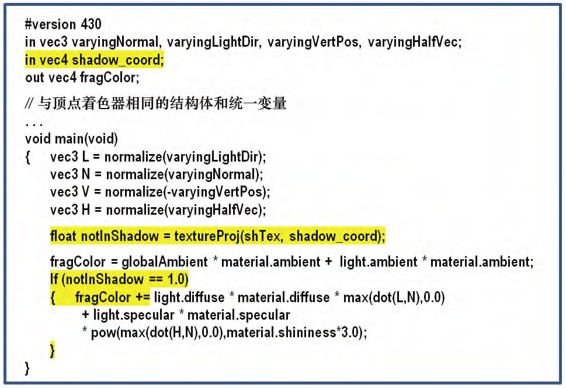


图8.7 阴影贴图第2轮片段着色器

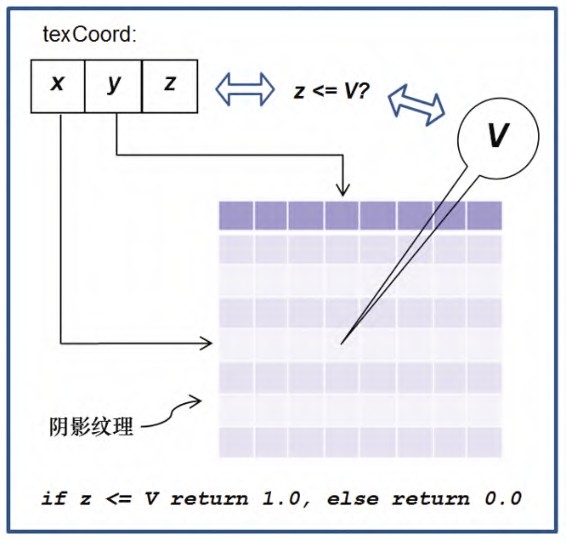


图8.8 自动深度比较

我们现在准备构建C++ / OpenGL应用程序以使用上述着色器。

8.5 阴影贴图示例

考虑图8.9中包含环面和金字塔的场景。位置光源放置在左侧（注 意镜面高光）。

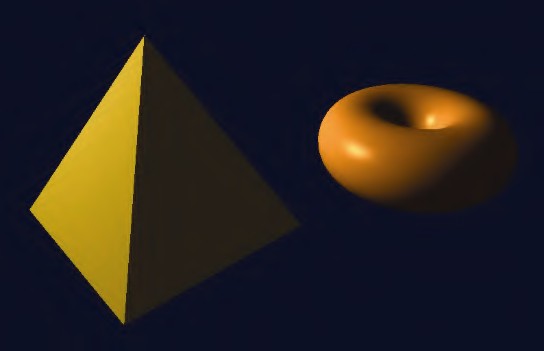


图8.9 有光照无阴影的场景

金字塔应该在环面上投下阴影。

为了阐明示例的开发，我们的第一步是将第1轮渲染到屏幕以确保

它正常工作。为此，我们将临时添加一个简单的片段着色器（它不会 包含在最终版本中）并在第1轮中仅输出一种固定颜色（如红色）；例 如：

#version 430

out vec4 fragColor; void main(void)

{ fragColor = vec4(1.0, 0.0, 0.0, 0.0);

}

让我们假设场景的原点位于图的中心在金字塔和环面之间。在第1 轮中，我们将相机放在光源的位置（图8.10中的左图）并指向

（0,0,0）。然后我们用红色绘制对象，它会产生如图8.10（见彩插） 右图所示的输出。注意金字塔顶部附近的环面——这个制高点附近的 环面部分位于金字塔后面。

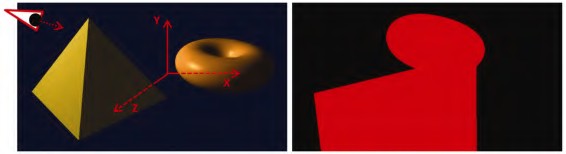


图8.10 第1轮：场景（左）和从光源视角渲染的场景（右）

包含光照与阴影贴图的完整第2轮C++/OpenGL代码见程序8.1。

程序8.1 阴影贴图

// 大部分与之前相同。高亮部分代码是新加入的，用以实现阴影

// 实现光照所需的大部分引用需要在代码开始引入，与之前相同

// 因此不在这里重复

glm::mat4 lightVmatrix; glm::mat4 lightPmatrix;

glm::mat4 shadowMVP1; glm::mat4 shadowMVP2;

// 在这里定义渲染程序所用的变量、缓冲区、着色器源代码等

. . .

ImportedModel pyramid("pyr.obj"); // 定义金字塔

Torus myTorus(0.6f, 0.4f, 48); // 定义环面

int numPyramidVertices, numTorusVertices, numTorusIndices;

. . .

// 环面、金字塔、相机和光源的位置

glm::vec3 torusLoc(1.6f, 0.0f, -0.3f);

glm::vec3 pyrLoc(-1.0f, 0.1f, 0.3f); glm::vec3 cameraLoc(0.0f, 0.2f, 6.0f); glm::vec3 lightLoc(-3.8f, 2.2f, 1.1f);

// 场景中所使用白光的属性(全局光和位置光)

float globalAmbient[4] = { 0.7f, 0.7f, 0.7f, 1.0f }; float lightAmbient[4] = { 0.0f, 0.0f, 0.0f, 1.0f }; float lightDiffuse[4] = { 1.0f, 1.0f, 1.0f, 1.0f }; float lightSpecular[4] = { 1.0f, 1.0f, 1.0f, 1.0f };

// 金字塔的黄金材质

float\* goldMatAmb = Utils::goldAmbient(); float\* goldMatDif = Utils::goldDiffuse(); float\* goldMatSpe = Utils::goldSpecular(); float goldMatShi = Utils::goldShininess();

// 环面的青铜材质

float\* bronzeMatAmb = Utils::bronzeAmbient(); float\* bronzeMatDif = Utils::bronzeDiffuse(); float\* bronzeMatSpe = Utils::bronzeSpecular(); float bronzeMatShi = Utils::bronzeShininess();

// 在display()中将光照传入着色器的变量

float curAmb[4], curDif[4], curSpe[4], matAmb[4], matDif[4], matSpe[4];

float curShi, matShi;

// 阴影相关变量 int screenSizeX, screenSizeY; GLuint shadowTex, shadowBuffer;

glm::mat4 b;

// 这里定义类型为mat4的光源观察矩阵与相机观察矩阵的矩阵变换(mMat, vMat等)

// 其他在display中所使用的变量也在此定义

. . .

int main(void) {

// 与前例相同，无改动

}

// init()函数依然执行调用以编译着色器并初始化物体

// 同时它也调用setupShadowBuffers()函数以初始化阴影贴图相关缓冲区

// 最后，它构造B矩阵以进行从光照空间到纹理空间的转换

void init(GLFWwindow\* window) { renderingProgram1 =

Utils::createShaderProgram("./vert1Shader.glsl", "./frag1Shader.glsl");

renderingProgram2 = Utils::createShaderProgram("./vert2Shader.glsl", "./frag2Shader.glsl");

setupVertices(); setupShadowBuffers();

b = glm::mat4(

0.5f, 0.0f, 0.0f, 0.0f,

0.0f, 0.5f, 0.0f, 0.0f,

0.0f, 0.0f, 0.5f, 0.0f,

0.5f, 0.5f, 0.5f, 1.0f);

}

void setupShadowBuffers(GLFWwindow\* window) { glfwGetFramebufferSize(window, &width, &height); screenSizeX = width;

screenSizeY = height;

// 创建自定义帧缓冲区

glGenFramebuffers(1, &shadowBuffer);

// 创建阴影纹理并让它存储深度信息

// 这些步骤与程序5.2中相似

glGenTextures(1, &shadowTex); glBindTexture(GL\_TEXTURE\_2D, shadowTex); glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_DEPTH\_COMPONENT32,

screenSizeX, screenSizeY, 0, GL\_DEPTH\_COMPONENT, GL\_FLOAT,

0);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_COMPARE\_MODE, GL\_COMPARE\_REF\_TO\_TEXTURE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_COMPARE\_FUNC, GL\_LEQUAL);

}

void setupVertices(void) {

// 与之前的例子相同。这个函数用来创建VAO和VBO

// 之后将环面及金字塔的顶点与法向量读入缓冲区

}

// display()函数分别管理第1轮需要使用的自定义帧缓冲区

// 以及第2轮需要使用的阴影纹理初始化过程。阴影相关新功能已高亮

void display(GLFWwindow\* window, double currentTime) { glClear(GL\_COLOR\_BUFFER\_BIT); glClear(GL\_DEPTH\_BUFFER\_BIT);

// 从光源视角初始化视觉矩阵以及透视矩阵，以便在第1轮中使用

lightVmatrix = glm::lookAt(currentLightPos, origin, up); // 从光 源到原点的矩阵

lightPmatrix = glm::perspective(toRadians(60.0f), aspect, 0.1f, 1000.0f);

// 使用自定义帧缓冲区，将阴影纹理附着到其上 glBindFramebuffer(GL\_FRAMEBUFFER, shadowBuffer); glFramebufferTexture(GL\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT,

shadowTex, 0);

// 关闭绘制颜色，同时开启深度计算 glDrawBuffer(GL\_NONE); glEnable(GL\_DEPTH\_TEST);

passOne();

// 使用显示缓冲区，并重新开启绘制 glBindFramebuffer(GL\_FRAMEBUFFER, 0); glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, shadowTex);

glDrawBuffer(GL\_FRONT); // 重新开启绘制颜色

passTwo();

}

// 接下来是第1轮和第2轮的代码

// 这些代码和之前的大体相同

// 与阴影相关的新增代码已高亮

void passOne(void) {

// renderingProgram1包含了第1轮中的顶点着色器和片段着色器 glUseProgram(renderingProgram1);

. . .

// 接下来的代码段通过从光源角度渲染环面获得深度缓冲区

mMat = glm::translate(glm::mat4(1.0f), torusLoc);

// 轻微旋转以便查看

mMat = glm::rotate(mMat, toRadians(25.0f), glm::vec3(1.0f, 0.0f, 0.0f));

// 我们从光源角度绘制，因此使用光源的P、V矩阵 shadowMVP1 = lightPmatrix \* lightVmatrix \* mMat; sLoc = glGetUniformLocation(renderingProgram1, "shadowMVP"); glUniformMatrix4fv(sLoc, 1, GL\_FALSE,

glm::value\_ptr(shadowMVP1));

// 在第1轮中我们只需要环面的顶点缓冲区，而不需要它的纹理或法向量 glBindBuffer(GL\_ARRAY\_BUFFER, vbo[0]); glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0); glEnableVertexAttribArray(0);

glClear(GL\_DEPTH\_BUFFER\_BIT); glEnable(GL\_CULL\_FACE); glFrontFace(GL\_CCW); glEnable(GL\_DEPTH\_TEST); glDepthFunc(GL\_LEQUAL);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, vbo[4]); // vbo[4]

包含环面索引

glDrawElements(GL\_TRIANGLES, numTorusIndices, GL\_UNSIGNED\_INT,

0);

// 对金字塔做同样的处理(但不清除GL\_DEPTH\_BUFFER\_BIT)

// 金字塔没有索引，因此我们使用glDrawArrays()而非glDrawElements()

. . .

glDrawArrays(GL\_TRIANGLES, 0, numPyramidVertices);

}

void passTwo(void) {

glUseProgram(renderingProgram2); // 第2轮顶点着色器和片段着 色器

// 绘制环面，这次我们需要加入光照、材质、法向量等

// 同时我们需要为相机空间以及光照空间都提供MVP变换

mvLoc = glGetUniformLocation(renderingProgram2, "mv\_matrix"); projLoc = glGetUniformLocation(renderingProgram2,

"proj\_matrix");

nLoc = glGetUniformLocation(renderingProgram2, "norm\_matrix"); sLoc = glGetUniformLocation(renderingProgram2, "shadowMVP");

// 环面是黄铜材质

curAmb[0] = bronzeMatAmb[0]; curAmb[1] = bronzeMatAmb[1];

curAmb[2] = bronzeMatAmb[2];

curDif[0] = bronzeMatDif[0]; curDif[1] = bronzeMatDif[1]; curDif[2] = bronzeMatDif[2];

curSpe[0] = bronzeMatSpe[0]; curSpe[1] = bronzeMatSpe[1]; curSpe[2] = bronzeMatSpe[2];

curShi = bronzeMatShi;

vMat = glm::translate(glm::mat4(1.0f), glm::vec3(-cameraLoc.x,

-cameraLoc.y, -cameraLoc.z));

currentLightPos = glm::vec3(lightLoc); installLights(renderingProgram2, vMat);

mMat = glm::translate(glm::mat4(1.0f), torusLoc);

// 轻微旋转以便查看

mMat = glm::rotate(mMat, toRadians(25.0f), glm::vec3(1.0f, 0.0f, 0.0f));

// 构建相机视角环面的MV矩阵

mvMat = vMat \* mMat;

invTrMat = glm::transpose(glm::inverse(mvMat));

// 构建光源视角环面的MV矩阵

shadowMVP2 = b \* lightPmatrix \* lightVmatrix \* mMat;

// 将MV以及PROJ矩阵传入相应的统一变量

glUniformMatrix4fv(mvLoc, 1, GL\_FALSE, glm::value\_ptr(mvMat)); glUniformMatrix4fv(projLoc, 1, GL\_FALSE, glm::value\_ptr(pMat)); glUniformMatrix4fv(nLoc, 1, GL\_FALSE,

glm::value\_ptr(invTrMat)); glUniformMatrix4fv(sLoc, 1, GL\_FALSE,

glm::value\_ptr(shadowMVP2));

// 初始化环面顶点和法向量缓冲区() glBindBuffer(GL\_ARRAY\_BUFFER, vbo[0]); // 环面顶点 glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0); glEnableVertexAttribArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, vbo[2]); // 环面法向量 glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 0, 0); glEnableVertexAttribArray(1);

glClear(GL\_DEPTH\_BUFFER\_BIT); glEnable(GL\_CULL\_FACE); glFrontFace(GL\_CCW); glEnable(GL\_DEPTH\_TEST); glDepthFunc(GL\_LEQUAL);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, vbo[4]); // vbo[4]

包含环面索引

glDrawElements(GL\_TRIANGLES, numTorusIndices, GL\_UNSIGNED\_INT,

0);

. . .

// 对黄金金字塔重复同样步骤

}

程序8.1展示了与之前详述过的第1轮、第2轮着色器交互部分的 C++ / OpenGL应用程序。之前已经展示过的模块，如读取着色器、编 译着色器、构建模型及相关缓冲区、在着色器中初始化位置光源ADS特 性以及进行透视矩阵和LookAt矩阵计算等，这些模块同之前一样。

8.6 阴影贴图的伪影

虽然我们已经实现了为场景添加阴影的所有基本要求，但运行程 序8.1会产生错杂的结果，如图8.11所示。

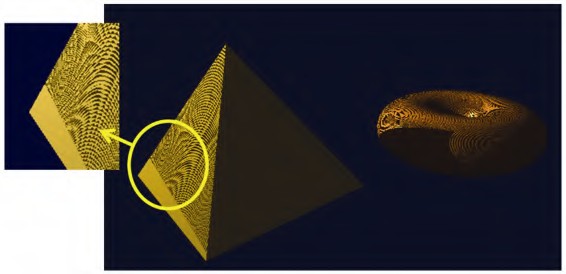


图8.11 阴影的“痤疮”

好消息是我们的金字塔现在在环面上投下阴影！坏消息则是，这 种成功伴随着严重的伪影。有许多波浪线覆盖在场景中的表面。这是

阴影贴图的常见副作用，称为阴影痤疮（也称为阴影斑块，shadow

acne）或错误的自阴影。

阴影痤疮是由深度测试期间的舍入误差引起的。在阴影纹理中查 找深度信息时计算的纹理坐标通常与实际坐标不完全匹配。因此，从 阴影纹理中查找到的深度值可能并非当前渲染中像素的深度，而是相 邻像素的深度。如果相邻像素在更远位置，则当前像素会被错误地显 示为阴影。

阴影痤疮也会由纹理贴图和深度计算之间的精度差引起。这也可 能导致舍入误差，并造成对像素是否处于阴影中的误判。

幸运的是，阴影痤疮很容易修复。由于阴影痤疮通常发生在没有 阴影的表面上，这里有个简单的技巧，在第1轮中将每个像素稍微移向 光源，之后在第2轮将它们移回原位。通常，这么做足以补偿各类舍入 误差。在我们的实现中简单地在display()函数中调用glPolygon- Offset()即可，如图8.12所示（突出显示部分）。

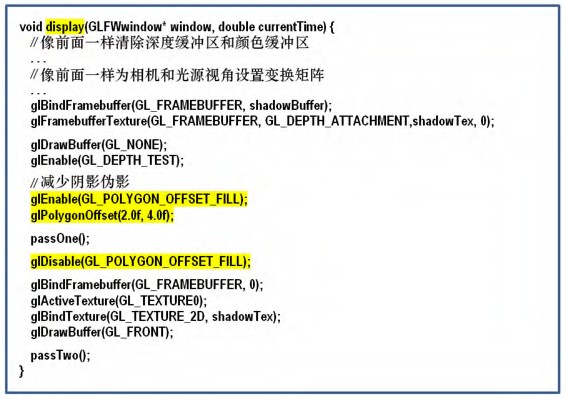


图8.12 与阴影痤疮的战斗

将这几行代码添加到display()函数，可以显著改善程序的输出， 如图8.13所示。还要注意，随着伪影的消失，现在可以看到环面的内 圆在其自身上显示了一个正确的小阴影。

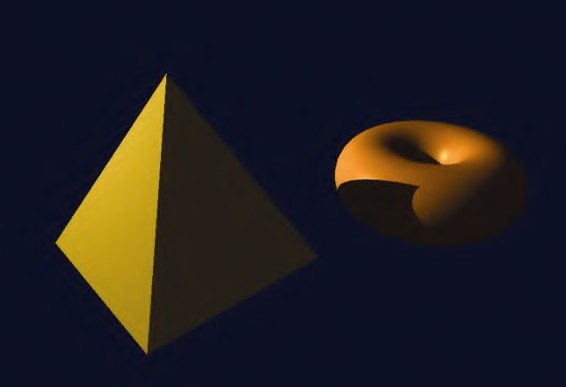


图8.13 渲染带阴影的场景

虽然修复阴影痤疮很容易，但有时修复会引起新的伪影。在第1轮

之前移动对象的“技巧”有时会导致在对象阴影中出现间隙。图8.14 显示了一个这样的例子。这种伪影通常被称为“Peter Panning”，因 为有时它会导致静止物体的阴影与物体底部分离的问题（从而使物体 的阴影部分与阴影的其余部分分离，让人想起詹姆斯·马修·巴利笔 下的角色Peter Pan[PP16]）。修复此伪影需要调整glPolygonOffset() 的参数。如果它们太小，就会出现阴影痤疮；如果太大，则会出现 Peter Panning。

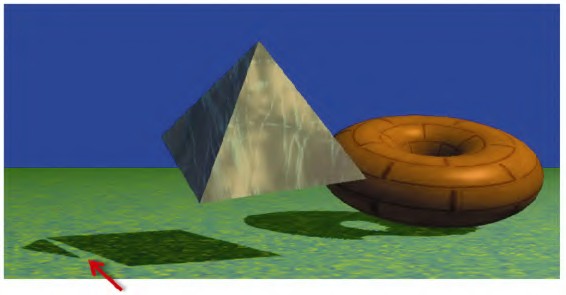


图8.14 修复引起新的伪影

在实现阴影贴图时可能会发生许多其他伪影。如重复阴影，因为 在第1轮（存入阴影缓冲区时）渲染的场景区域与第2轮中渲染的场景 区域不同（来自不同的观察位置）。这种差异可能导致在第2轮中渲染 的场景中，某些区域尝试使用范围[0…1]之外的特征坐标来访问阴影 纹理。回想一下，在这种情况下默认行为是GL\_REPEAT，因此，这可能 导致错误的重复阴影。

一种可能的解决方案是将以下代码行添加到

setupShadowBuffers()，将纹理换行模式设置为“夹紧到边缘”。

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

这样纹理边缘以外的值会被限制为边缘处的值（而非重复）。注 意，这种方法自身也有可能造成伪影，即当阴影纹理的边缘处存在阴 影时，截取边缘可能产生延伸到场景边缘的“阴影条”。

另一种常见错误是锯齿状阴影边缘。当投射的阴影明显大于阴影 缓冲区可以准确表示的阴影时，就有可能出问题。通常，这取决于场 景中物体和灯光的位置。尤其当光源在距离物体较远时，更容易发 生。一个例子如图8.15所示。

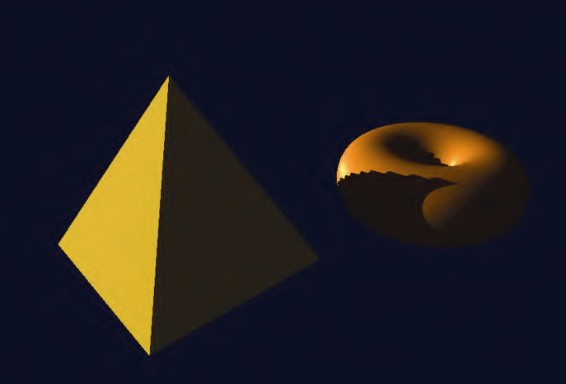


图8.15 锯齿状阴影边缘

消除锯齿状阴影边缘就没有处理之前的伪影那么简单了。一种技

术是在第1轮期间将光位置移动到更接近场景的位置，然后在第2轮放 回原始位置。另一种常用的有效方法则是我们将在下面讨论的“柔和 阴影”方法之一。

8.7 柔和阴影

目前我们所展示的阴影生成方法都仅限于生成硬阴影，即带锐边 的阴影。但是，现实世界中出现的大多数阴影都是柔和阴影。它们的 边缘都会发生不同程度的模糊。在本节中，我们将探讨现实世界中柔 和阴影的外观，然后描述在OpenGL中模拟它们的常用算法。消除锯齿 状阴影边缘并不像处理之前的伪影那么简单。

8.7.1 现实世界中的柔和阴影

柔和阴影的成因有很多，同时也有许多类型的柔和阴影。通常在 自然界中产生柔和阴影原因是，真实世界的光源很少是点光源——它 们常常是区域光源。另一个原因是材料和表面的缺陷积累，以及物体 本身通过其自身的反射特性产生环境光的作用。

图8.16展示了物体向桌面投射柔和阴影的照片示例。注意，这不 是计算机渲染的3D场景，而是真实的照片，是本书作者之一在家中拍 摄的。



图8.16 现实世界中的柔和阴影示例

对于图8.16中的阴影，有两点需要注意。

离物体越远的阴影越“柔和”，离物体越近的阴影越“硬”。在 对比物体腿附近的阴影与右边更宽的阴影时，这一点就很明显 了。

距离物体越近的阴影显得越暗。

光源本身的维度会导致柔和阴影。如图8.17所示，光源上各处会 投射出略微不同的阴影。各种阴影不同的区域称为半影

（penumbra），包括阴影边缘的柔和区域。

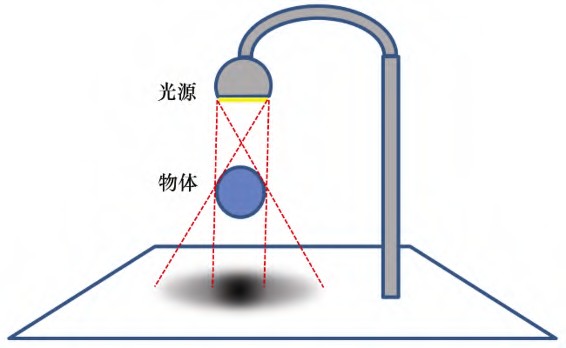


图8.17 柔和阴影的半影效果

8.7.2 生成柔和阴影——百分比邻近滤波（PCF）

有多种方法可以用来模拟半影效果以在软件中生成柔和阴影。最 简单也最常见的一种方法叫作百分比邻近滤波（Percentage Closer Filtering，PCF）。在PCF中，我们对单个点周围的几个位置的阴影纹 理进行采样，以估计附近位置在阴影中的百分比。根据附近位置在阴 影中的数量，对正在渲染的像素的光照分量进行修改。整个计算可以 在片段着色器中完成，所以我们只需要对其中的代码进行修改。PCF还 可用于减少锯齿线伪影。

在研究实际的PCF算法之前，我们先看一个类似的简单示例来展示 PCF的目标。考虑图8.18中所示的输出片段（像素）集，其颜色由片段 着色器计算。



图8.18 硬阴影渲染

假设深色像素处于阴影中，这是阴影贴图计算的结果。假设我们 可以访问相邻的像素信息，而不是简单地如图所示渲染像素（即包括

或不包括漫反射和镜面反射分量），这样我们就可以看到有多少相邻

像素处于阴影中。例如，考虑图8.19（见彩插）中以黄色突出显示的 特定像素，根据图8.18，该像素不在阴影中。

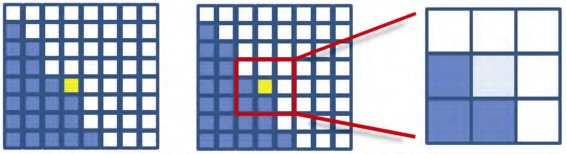


图8.19 单像素PCF采样

在高亮像素的9个像素邻域中，3个像素处于阴影中而6个像素处于 阴影外。因此，渲染像素的颜色可以被计算为该像素处的环境光分量 加上漫反射和镜面反射分量的9/6，这样会使像素一定程度（但不是完 全）变亮。在整个网格中重复此过程将会产生图8.20所示的像素颜 色。注意，对于那些邻域完全位于阴影中（或阴影外）的像素，生成 的颜色与标准阴影贴图相同。

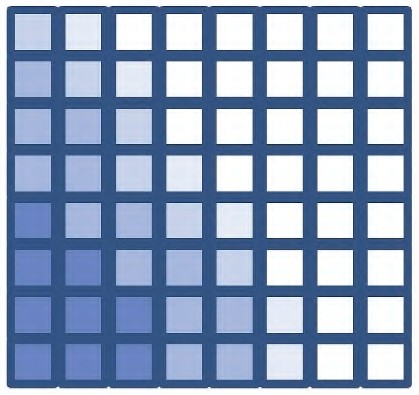


图8.20 柔和阴影渲染

与上例不同的是，在PCF的实现中，不是对渲染像素临近区域内的

每个像素进行采样。这有两个原因：（a）我们想在片段着色器中执行 此计算，但片段着色器无法访问其他像素；（b）获得足够宽的半影效 果（例如，10～20像素宽）将需要为每个被渲染的像素采样数百个附 近的像素。

PCF解决了以下两个问题。首先，我们不试图访问附近的像素，而 是在阴影贴图中对附近的纹素进行采样。片段着色器可以执行此操 作，因为虽然它无法访问附近像素的值，但它可以访问整个阴影贴 图。其次，为了获得足够宽的半影效果，需要对附近一定数量的阴影 贴图纹素进行采样，每个采样的纹素都距离所渲染像素的纹素一定距 离。

半影的宽度和采样点数可以根据场景和性能要求调整。例如，图 8.21所示PCF生成的图像是，每个像素的亮度是通过对64个不同的纹素 进行采样确定的，它们与像素的纹素距离各不相同。

柔和阴影的准确度或平滑度取决于所采样附近纹素的数量。因 此，在性能和质量之间需要权衡——采样点越多，效果越好，但计算 开销也越多。场景的复杂性和给定应用所需的帧率对于阴影可实现的 质量有着相应的限制。每像素采样64个点（如图8.21所示）通常是不 切实际的。



图8.21 柔和阴影渲染——每像素64次采样

一种用于实现PCF的常见算法是对每个像素附近的4个纹素进行采 样，其中样本通过指定从像素对应纹素的偏移量选择。对于每个像 素，我们都需要改变偏移量，并用新的偏移量确定采样的4个纹素。用 交错方式改变偏移量的方法被称为抖动，它旨在使得柔和阴影边界不 会由于采样点不足看起来“结块”。

一种常见的方法是假设有4种不同偏移模式，每次取其中一种—— 我们可以通过计算像素的glFragCoord mod 2来选择当前像素的偏移模

式。之前有提到，glFragCoord是vec2类型，包含像素位置的 、 坐

x y

标。因此，mod计算的结果有4种可能的值：(0,0)、(0,1)、(1,0)或 (1,1)。我们使用glFragCoord mod 2的结果来从纹素空间（即阴影贴 图）4种不同偏移模式中选择一种。

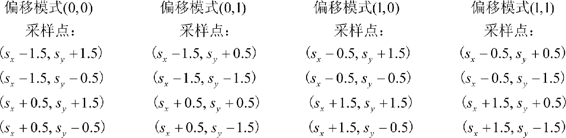
偏移模式通常在

x

和 方向上指定，具有−1.5，−0.5，+0.5和+1.5

y

的不同组合（也可以根据需要进行缩放）。更具体来说，由 glFragCoord mod 2计算得到的每种情况的4种常用偏移模式是：



和

Sx Sy

指的是与正在渲染的像素相对应的阴影贴图中的位置

（ ，

Sx Sy

），在本章的代码示例中标识为shadow\_coord。这4种偏移模

式如图8.22所示（见彩插），每种情况都以不同的颜色显示。在每种 情况下，对应于正被渲染的像素的纹素位于该情况的图的原点。请注 意，当在图8.23（见彩插）中一起显示时，偏移的交错/抖动很明显。

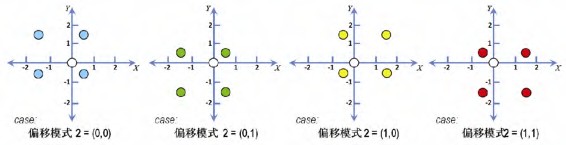


图8.22 抖动的4像素PCF采样示例

让我们来针对特定像素看看整个计算过程。假设正在渲染的像素 位于glFragCoord =（48,13）。首先我们确定像素在阴影贴图的4个采 样点。为此，我们将计算vec2(48,13) mod 2，等于（0,1）。因此我 们选择（0,1）所对应的偏移，在图8.22中以绿色显示，并且在阴影贴 图对相应的点进行采样（假设没有指定偏移的缩放量），得到：

(shadow\_coord.x–1.5, shadow\_coord.y+0.5) (shadow\_coord.x–1.5, shadow\_coord.y–1.5)

(shadow\_coord.x+0.5, shadow\_coord.y+0.5)

(shadow\_coord.x+0.5, shadow\_coord.y–1.5)

（回想一下，shadow\_coord是阴影贴图中与正在渲染的像素相对 应的纹素的位置——在图8.22和图8.23中显示为白色圆圈）。

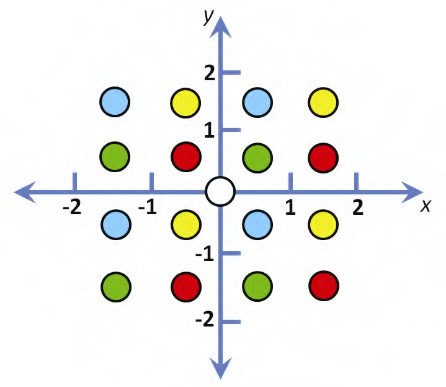


图8.23 抖动的4像素PCF采样（4种偏移模式）

接下来，对我们选取的这4个点分别调用textureProj()，在每种 情况下都返回0.0或1.0，具体取决于该采样点是否在阴影中。将4个结 果相加并除以4.0，就可以确定阴影中采样点的百分比。然后将此百分 比用作乘数，确定渲染当前像素时要应用的漫反射和镜面反射分量。

尽管采样尺寸很小——每个像素只有4个样本——这种抖动方法通 常可以产生好得惊人的柔和阴影。图8.24是使用4像素抖动PCF生成 的。虽然它不如之前图8.21所示的64点采样版本好，但渲染速度要快 得多。



图8.24 柔和阴影渲染——每像素4次采样，使用抖动

在下一节中，我们对GLSL片段着色器进行编码，实现4采样抖动的 PCF柔和阴影以及之前展示的64采样PCF柔和阴影。

8.7.3 柔和阴影/PCF程序

如前所述，柔和阴影计算可以完全在片段着色器中完成。程序8.2 展示了片段着色器代码，取代图8.7中的片段着色器。添加的PCF相关 代码已突出显示。

程序8.2 百分比邻近滤波（PCF）

片段着色器

#version 430

// 所有变量定义未改动

. . .

// 从shadow\_coord返回距离（x，y）处的纹素的阴影深度值

// shadow\_coord是阴影贴图中与正在渲染的当前像素相对应的位置

float lookup(float ox, float oy)

{ float t = textureProj(shadowTex, shadow\_coord + vec4(ox \* 0.001 \* shadow\_coord.w, oy \* 0.001 \*

shadow\_coord.w,

-0.01, 0.0)); //第三个参数（-0.01）是用于消除阴影痤疮的偏移量

return t;

}

void main(void)

{ float shadowFactor = 0.0;

vec3 L = normalize(vLightDir); vec3 N = normalize(vNormal); vec3 V = normalize(-vVertPos); vec3 H = normalize(vHalfVec);

// -----此部分生成一个4采样抖动的柔和阴影 float swidth = 2.5; //可调整的阴影扩散量

// 根据glFragCoord mod 2生成4采样模式中的一个

vec2 offset = mod(floor(gl\_FragCoord.xy), 2.0) \* swidth; shadowFactor += lookup(-1.5\*swidth + offset.x, 1.5\*swidth -

offset.y);

shadowFactor += lookup(-1.5\*swidth + offset.x, -0.5\*swidth - offset.y);

shadowFactor += lookup( 0.5\*swidth + offset.x, 1.5\*swidth - offset.y);

shadowFactor += lookup( 0.5\*swidth + offset.x, -0.5\*swidth - offset.y);

shadowFactor = shadowFactor / 4.0; // shadowFactor是4个采样点的 平均值

// ----- 取消本节注释以生成64采样的高分辨率柔和阴影

// *float swidth = 2.5;* // 可调整的阴影扩散量

// *float endp = swidth\*3.0 +swidth/2.0;*

// *for (float m=-endp ; m<=endp ; m=m+swidth)*

// *{ for (float n=-endp ; n<=endp ; n=n+swidth)*

// *{ shadowFactor += lookup(m,n);*

// *} }*

// *shadowFactor = shadowFactor / 64.0;*

vec4 shadowColor = globalAmbient \* material.ambient + light.ambient \* material.ambient;

vec4 lightedColor = light.diffuse \* material.diffuse \* max(dot(L,N),0.0)

+ light.specular \* material.specular

\* pow(max(dot(H,N),0.0),material.shininess\*3.0);

fragColor = vec4((shadowColor.xyz + shadowFactor\* (lightedColor.xyz)),1.0);

}

程序8.2中展示的片段着色器包含4采样和64采样的PCF柔和阴影的

代码。为了更方便进行采样，我们需要定义lookup()函数。在 lookup()函数中调用GLSL函数textureProj()，从而在阴影纹理中以指

定偏移量(

ox

, )进行查找。偏移量需要乘以1 /

oy windowsize

，这里

我们简单地假设窗口大小为1 000像素×1 000像素，将乘数硬编码为 0.001。[[2]](#_bookmark158)

4样本抖动的计算代码在main()函数中高亮显示，其实现遵循上一 节中描述的算法。同时添加了一个比例因子swidth，用于调整阴影边 缘的“柔和”区域的大小。

64采样代码以注释形式出现在后面。可以通过取消64采样代码注 释并注释4采样代码以使用64采样。在64采样代码中，swidth比例因子 用作嵌套循环中的步长，其采样距离正被渲染的像素的不同距离处的 点。例如，当使用代码中的swidth值(2.5)时，程序将沿着每个轴在两 个方向上以1.25、3.75、6.25和8.25的距离选择采样点——然后根据 窗口大小进行缩放（如前所述）并用作纹理坐标采样阴影纹理。在这 么多采样的情况下，通常不需要使用抖动来获得更好的结果。

图8.25展示了我们运行的环面/金字塔阴影贴图示例，它将PCF柔 和阴影与程序8.2中的片段着色器相结合，分别使用了4采样和64采样 的方法。swidth的选值取决于场景；对于环面/金字塔示例，它的值为 2.5，而对于之前的图8.21中显示的海豚示例，swidth的值为8.0。

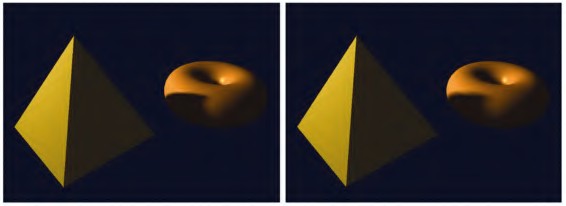


图8.25 PCF柔和阴影渲染——每像素4次采样，抖动（左）；每像素64次采样， 不抖动（右）

# 补充说明

在本章中，我们仅给出了3D图形中阴影世界的最基本介绍。在更 复杂的场景中，即便使用本章提供的基础阴影贴图方法，也可能需要 进行进一步的研究。

例如，当场景中的某些对象拥有纹理的情况下，添加阴影时必须 确保片段着色器正确区分阴影纹理和其他纹理。一种简单的方法是将 它们绑定到不同的纹理单元，例如：

layout (binding = 0) uniform sampler2DShadow shTex; layout (binding = 1) uniform sampler2D otherTexture;

然后，C++ / OpenGL应用程序可以通过它们的绑定值来引用两个 采样器。

当场景使用多个灯光时，则需要多个阴影纹理——每个光源需要

一个阴影纹理。此外，每个光源都需要单独执行第1轮渲染，并在第2 轮渲染中合并结果。

尽管我们在阴影贴图的每个阶段都使用了透视投影，但值得注意 的是，当光源是远距离光源和定向光源而非我们使用的位置光时，正 射投影通常才是首选。

生成真实的阴影在计算机图形学中仍然是一个活跃而又复杂的领 域，其中提出的许多技术超出了本书的范畴。我们鼓励对更多细节感 兴趣的读者研究更专业的资源，如[ES12], [GP10]和[MI16]。

8.7.3小节包含一个GLSL函数的例子（除了“main”）。与在C语 言中一样，必须在调用它们之前（或“上方”）定义函数，否则必须 提供前向声明。在该示例中则不需要前向声明，因为函数定义在调用 代码上方。

习题

8.1 在程序8.1中，尝试在不同设置下使用glPolygonOffset()， 并观察对像的伪影效果，如Peter Panning。

8.2 （项目）修改程序8.1，以便通过移动鼠标移动灯光，类似 于练习7.1。你可能会注意到某些照明位置会出现阴影伪影，而其他位 置则没有。

8.3 （项目）给程序8.1添加动画，使得对象或光源（或两者一

起）自行移动——例如一个绕另一个旋转。如果向场景添加地平面， 阴影效果将更加明显，如图8.14所示。

8.4 （项目）修改程序8.2，将lookup()函数中的硬编码值0.001 替换为更准确的1.0 / shadowbufferwidth和1.0 / shadowbufferheight。观察在窗口大小变化的情况下，这种变化产生 了何种程度的影响（或没有影响）。

8.5 （研究）更复杂的百分比邻近滤波（PCF）的实现会加入光 和阴影与光和遮挡物之间的相对距离。通过光线靠近或远离遮挡物时

（或当遮挡物靠近或远离阴影时），调整半影的大小，可以使柔和阴 影更逼真。研究此功能现有的实现方法，并将其添加到程序8.2中。

# 参考资料

[AS14] E. Angel and D. Shreiner,

Interactive Computer

, 7th ed. (Pearson,

Graphics: A Top-Down Approach with WebGL

2014).

[BL88] J. Blinn, “Me and My (Fake) Shadow,”

IEEE

8, no. 2 (1988).

Computer Graphics and Applications

[CR77] F. Crow, “Shadow Algorithms for Computer Graphics,” Proceedings of SIGGRAPH ’77 11, no. 2 (1977).

[ES12] E. Eisemann, M. Schwarz, U. Assarsson, and M.

Wimmer,

Real-Time Shadows

(CRC Press, 2012).

[GP10]

GPU Pro

(series), ed. Wolfgang Engel (A. K.

Peters, 2010–2016).

[KS16] J. Kessenich, G. Sellers, and D. Shreiner,

OpenGL

Programming Guide: The Official Guide to Learning OpenGL,

, 9th ed. (Addison-Wesley, 2016).

Version 4.5 with SPIR-V

[LO12] “Understanding OpenGL’s Matrices,” Learn OpenGL ES (2012), accessed October 2018.

[LU16] F. Luna,

Introduction to 3D Game Programming with

, 2nd ed. (Mercury Learning, 2016).

DirectX 12

[MI16] “Common Techniques to Improve Shadow Depth Maps” (Microsoft Corp., 2016), accessed October 2018.

[PP16]

Peter Pan

, Wikipedia, accessed October 2018.

[[1]](#_bookmark142) 模板缓冲区是通过OpenGL访问的第三个缓冲区——在颜色缓冲区 和Z缓冲区之后。本书中不对模板缓冲区进行讲解。 ——译者注

[[2]](#_bookmark153) 我们还需要将偏移乘以阴影坐标的

w

分量，因为OpenGL在纹理查

找期间会自动将输入坐标除以

w

。迄今为止我们一直忽略了这种称为透

视分割的操作，因此必须在这里说明。有关透视分割的更多信息，请

参阅参考资料[LO12]。

第9章 天空和背景

对于室外3D场景，通常可以通过在地平线上创造一些逼真的效 果，来增强其真实感。当我们极目远眺，目光越过附近的建筑和森 林，我们习惯于看到远处的大型物体，例如：云、群山或太阳（或夜 空中的星星和月亮）。但是，将这些对象作为单个模型添加到场景中 可能会产生高到无法承受的性能成本。天空盒或天空穹顶提供了有效 且相对简单的方法，用来生成令人信服的地平线景观。

9.1 天空盒

天空盒的概念非常巧妙而又简单：

（1）实例化一个立方体对象；

（2）将立方体的纹理设置为所需的环境；

（3）将立方体围绕相机放置。 我们已经知道如何完成以上这些步骤。但还有少量其他细节需要

注意。

如何为地平线制作纹理？

立方体有6个面，我们需要为这些面都添加纹理。一种方法是使用

6个图像文件和6个纹理单元。另一种常见（且高效）的方式则是使用 一个包含6个面的纹理的图像，如图9.1所示。

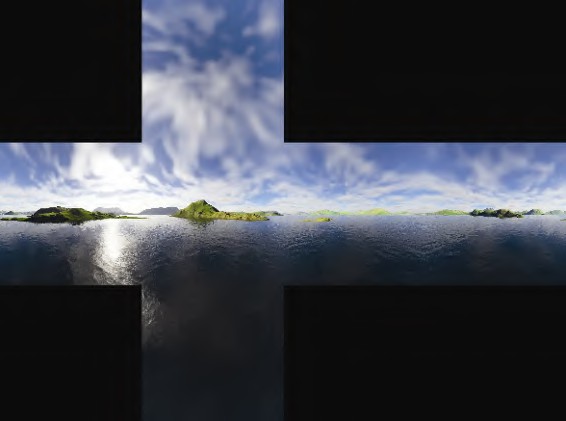


图9.1 6面天空盒纹理立方体贴图

上例中的纹理立方体贴图，仅用一个纹理单元，就可以为6个面添 加纹理的图像。立方体贴图的6个部分对应于立方体的顶部、底部、正 面、背面和两侧。当贴图“包裹”在立方体周围时，对于立方体内的 相机而言，它扮演了地平线的角色，如图9.2所示。

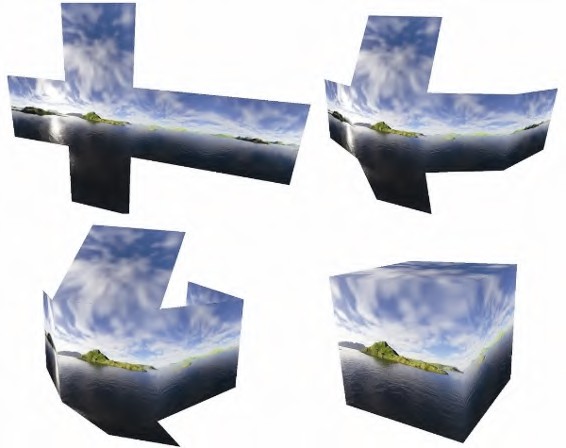


图9.2 立方体贴图包裹相机

使用纹理立方体贴图为立方体添加纹理需要指定适当的纹理坐 标。图9.3展示了纹理坐标的分布，这些坐标接着会分配给立方体的每 个顶点。

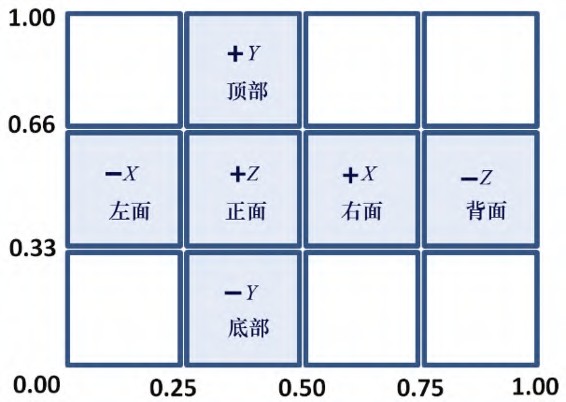


图9.3 立方体贴图纹理坐标

如何让天空盒看起来“距离很远”？

构建天空盒的另一个重要因素是确保纹理的表现看起来像是远处 的地平线。首先，人们可能会认为这需要构建巨大的天空盒。然而， 事实证明这并不可取，因为巨大的天空盒会拉伸和扭曲纹理。相反， 通过使用以下两个技巧，可以使天空盒显得巨大（从而感觉距离很 远）：

（a）禁用深度测试并先渲染天空盒（在渲染场景中的其他对象时 重新启用深度测试）；

（b）天空盒随相机移动（如果相机需要移动）。

通过在禁用深度测试的情况下先绘制天空盒，深度缓冲器的值仍

将全设为1.0（即最远距离）。因此，场景中的所有其他对象将被完全 渲染，即天空盒不会阻挡任何其他对象。这样，无论天空盒的实际大 小如何，会使天空盒的各面的位置看起来比其他物体都更远。而实际 的天空盒立方体本身可以非常小，只要它在相机移动时随相机一起移 动即可。图9.4展示了从天空盒内部查看简单的场景（实际上只有一个 砖纹理环面）。



图9.4 从天空盒内部查看场景

这里我们得益于对图9.4与之前图9.2和图9.3的关系的仔细研究。 注意，场景中可见的天空盒部分是立方体贴图的最右侧部分。这是因

为摄像机处于默认方向，面向−

Z

方向，因此正在观察天空盒立方体的

背面（如图9.3所示）。另请注意，立方体贴图的背面在场景中渲染时 会呈水平反转状态；这是因为立方体贴图的“背面”部分已经折叠在 相机周围，因此看起来是经过侧向翻转的，如图9.2所示。

如何构建纹理立方体贴图？

从图稿或照片构建纹理立方体贴图图像时，需要注意避免在立方

体面交汇点处的“接缝”，并创建正确的透视图，才能让天空盒看起 来逼真且无畸变。有许多工具可以辅助达成这一目标：Terragen、 Autodesk 3Ds Max、Blender和Adobe Photoshop都有用于构建或处理 立方体贴图的工具。同时，还有许多网站提供各种现成的立方体地 图，既有付费的，也有免费的。

9.2 天空穹顶

建立地平线效果的另一种方法是使用天空穹顶。除了使用带纹理 的球体（或半球体）代替带纹理的立方体，其基本思路与天空盒相 同。与天空盒相同，我们首先渲染天空穹顶（禁用深度测试），并将 摄像机保持在天空穹顶的中心位置（图9.5中的天空穹顶纹理是使用 Terragen [TE16]制作的）。

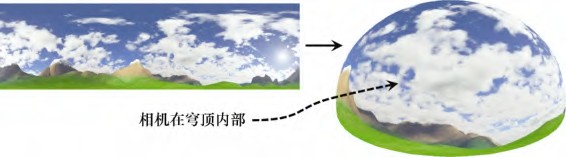


图9.5 天空穹顶与其中的相机

天空穹顶相比天空盒有自己的优势。例如，它们不易受到畸变和 接缝的影响（尽管在纹理图像中必须考虑极点处的球形畸变）。而天 空穹顶的缺点之一则是球体或穹顶模型比立方体模型更复杂，天空穹 顶有更多的顶点，其数量取决于期望的精度。

当使用天空穹顶呈现室外场景时，通常与地平面或某种地形相结

合。当使用天空穹顶呈现宇宙中的场景（例如星空）时，使用图9.6所 示的球体通常更为实际（为了清晰地使球体可视化，球体表面添加了 一道虚线）。

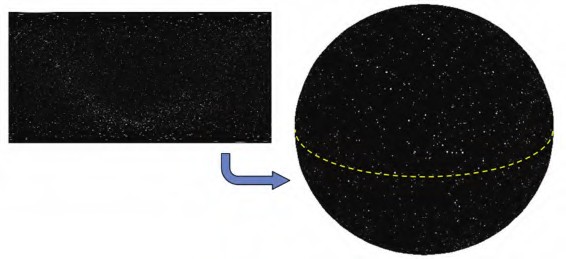


图9.6 使用球体的星空天空穹顶（星图来自[BO01]）

9.3 实现天空盒

尽管天空穹顶有许多优点，天空盒仍然更为常见。OpenGL对天空 盒的支持也更好，在进行环境贴图时更方便（本章后面会介绍）。出 于这些原因，我们将专注于天空盒的实现。

天空盒有两种实现方法：从头开始构建一个简单的天空盒；或使 用OpenGL中的立方体贴图工具。它们有各自的优点，因此我们下面都 会进行介绍。

9.3.1 从头开始构建天空盒

我们已经涵盖了构建简单天空盒所需的几乎所有内容。第4章介绍

了立方体模型；分配纹理坐标已经在本章前面图9.3中进行了展示；使 用SOIL2库读取纹理以及在3D空间中放置对象也都已经在之前的章节进 行过讲解。这里，我们将看到如何简单地启用和禁用深度测试（只需 要一行代码）。

程序9.1展示了简单天空盒的代码结构，场景中仅包含一个带纹理 的环面。纹理坐标分配和启用/禁用深度测试的调用已突出显示。

程序9.1 简单的天空盒

C++/OpenGL应用程序

//所有变量声明，构造函数和init()与之前相同

. . .

void display(GLFWwindow\* window, double currentTime) {

// 清除颜色缓冲区和深度缓冲区，并像之前一样创建投影视图矩阵和摄像机视图矩阵

. . .

glUseProgram(renderingProgram);

// 准备首先绘制天空盒。M矩阵将天空盒放置在摄像机位置

mMat = glm::translate(glm::mat4(1.0f), glm::vec3(cameraX, cameraY, cameraZ));

// 构建MODEL-VIEW矩阵

mvMat = vMat \* mMat;

// 如前，将MV和PROJ矩阵放入统一变量

. . .

// 设置包含顶点的缓冲区

glBindBuffer(GL\_ARRAY\_BUFFER, vbo[0]); glVertexAttribPointer(0,3, GL\_FLOAT, GL\_FALSE, 0, 0); glEnableVertexAttribArray(0);

// 设置包含纹理坐标的缓冲区

glBindBuffer(GL\_ARRAY\_BUFFER, vbo[1]); glVertexAttribPointer(1,2, GL\_FLOAT, GL\_FALSE, 0, 0); glEnableVertexAttribArray(1);

//激活天空盒纹理 glActiveTexture(GL\_TEXTURE0);

glBindTexture(GL\_TEXTURE\_2D, skyboxTexture);

glEnable(GL\_CULL\_FACE);

glFrontFace(GL\_CCW); // 立方体缠绕顺序是顺时针的，但我们从内部查 看，因此使用逆时针缠绕顺序GL\_CCW

glDisable(GL\_DEPTH\_TEST);

glDrawArrays(GL\_TRIANGLES, 0, 36); // 在没有深度测试的情况下 绘制天空盒

glEnable(GL\_DEPTH\_TEST);

//现在像之前一样绘制场景中的对象

. . .

glDrawElements( . . . ); //和之前的场景中的对象一样

}

void setupVertices(void) {

// cube\_vertices定义与之前相同

// 天空盒的立方体纹理坐标，如图9.3所示

float cubeTextureCoord[72] = { 1.00f, 0.66f, 1.00f, 0.33f, 0.75f, 0.33f, // 背面右下角

0.75f, 0.33f, 0.75f, 0.66f, 1.00f, 0.66f, // 背面左上角

0.75f, 0.33f, 0.50f, 0.33f, 0.75f, 0.66f, // 右面右下角

0.50f, 0.33f, 0.50f, 0.66f, 0.75f, 0.66f, // 右面左上角

0.50f, 0.33f, 0.25f, 0.33f, 0.50f, 0.66f, // 正面右下角

0.25f, 0.33f, 0.25f, 0.66f, 0.50f, 0.66f, // 正面左上角

0.25f, 0.33f, 0.00f, 0.33f, 0.25f, 0.66f, // 左面右下角

0.00f, 0.33f, 0.00f, 0.66f, 0.25f, 0.66f, // 左面左上角

0.25f, 0.33f, 0.50f, 0.33f, 0.50f, 0.00f, // 下面右下角

0.50f, 0.00f, 0.25f, 0.00f, 0.25f, 0.33f, // 下面左上角

0.25f, 1.00f, 0.50f, 1.00f, 0.50f, 0.66f, // 上面右下角

0.50f, 0.66f, 0.25f, 0.66f, 0.25f, 1.00f // 上面左上角

};

//像往常一样为立方体和场景对象设置缓冲区

}

//用于加载着色器、纹理、main()等的模块，如前

标准纹理着色器现在用于场景中的所有对象，包括立方体贴图：

顶点着色器

#version 430

layout (location = 0) in vec3 position;

layout (location = 1) in vec2 tex\_coord; out vec2 tc;

uniform mat4 mv\_matrix; uniform mat4 proj\_matrix;

layout (binding = 0) uniform sampler2D s;

void main(void)

{ tc = tex\_coord;

gl\_Position = proj\_matrix \* mv\_matrix \* vec4(position,1.0);

}

片段着色器

#version 430 in vec2 tc;

out vec4 fragColor; uniform mat4 mv\_matrix; uniform mat4 proj\_matrix;

layout (binding = 0) uniform sampler2D s;

void main(void)

{ fragColor = texture(s,tc);

}

程序9.1的输出如图9.7所示，包括两个不同立方体贴图纹理以及 各自的渲染结果。

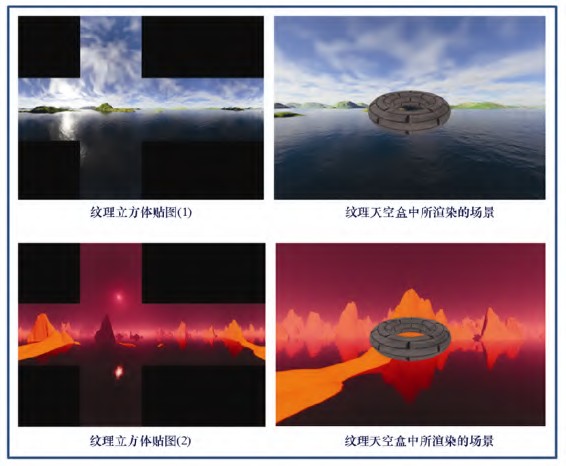


图9.7 简单天空盒渲染结果

如前所述，天空盒容易受到图像畸变和接缝的影响。接缝指两个 纹理图像接触的地方（比如沿着立方体的边缘）有时出现的可见线 条。图9.8展示了一个图像上半部分出现接缝的示例，它是运行程序 9.1时出现的伪影。为了避免接缝，需要仔细构建立方体贴图图像，并 分配精确的纹理坐标。有一些工具可以用来沿图像边缘减少接缝（例 如[GI16]），不过这个主题超出了本书的范围。

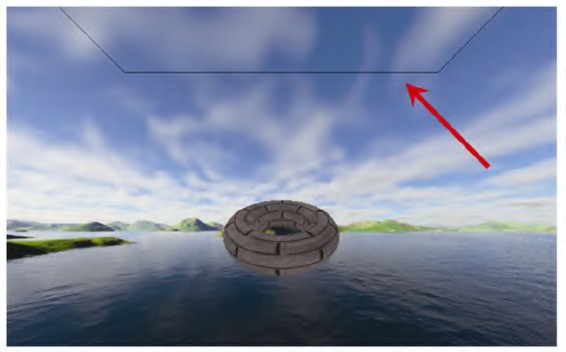


图9.8 天空盒“接缝”伪影

9.3.2 使用OpenGL立方体贴图

构建天空盒的另一种方法是使用OpenGL纹理立方体贴图。OpenGL 立方体贴图比我们在上一节中看到的简单方法稍微复杂一点。但是， 使用OpenGL立方体贴图有自己的优点，例如减少接缝以及支持环境贴 图。

OpenGL纹理立方体贴图类似于稍后将要研究的3D纹理，它们都使

用3个纹理坐标访问——通常标记为（

s

, ,

t r

）——而不是我们目前

为止用到的两个。OpenGL纹理立方体贴图的另一个特性是，其中的图 像以纹理图像的左上角（而不是通常的左下角）作为纹理坐标（0, 0, 0），这通常是混乱产生的源头。

程序9.1中展示的方法通过读入单个图像来为立方体贴图添加纹 理，而程序9.2中展示的loadCubeMap()函数则读入6个单独的立方体面

图像文件。正如我们在第5章中所学的，有许多方法可以读取纹理图

像，我们选择使用SOIL2库。在这里，SOIL2用于实例化和加载OpenGL 立方体贴图也非常方便。我们先找到需要读入的文件，然后调用 SOIL\_load\_OGL\_cubemap()，其参数包括6个图像文件和一些其他参 数，类似于我们在第5章中看到的SOIL\_load\_OGL\_texture()。在使用 OpenGL立方体贴图时，无须垂直翻转纹理，OpenGL会自动进行处理， 注意，loadCubeMap()函数放在“Utils.cpp”文件中。

init()函数现在包含一个函数调用以启用 GL\_TEXTURE\_CUBE\_MAP\_SEAMLESS，它告诉OpenGL尝试混合立方体相邻 的边以减少或消除接缝。在display()中，立方体的顶点像以前一样沿 管线向下发送，但这次不需要发送立方体的纹理坐标。我们将会看 到，OpenGL纹理立方体贴图通常使用立方体的顶点位置作为其纹理坐 标。之后禁用深度测试并绘制立方体。然后为场景的其余部分重新启 用深度测试。

完成后的OpenGL纹理立方体贴图使用了int类型的标识符进行引 用。与阴影贴图时一样，通过将纹理包裹模式设置为“夹紧到边 缘”，可以减少沿边框的伪影。在这种情况下，它还可以帮助进一步

缩小接缝。请注意，这里需要为3个纹理坐标

s

、 和 都设置纹理包裹

t r

模式。

在片段着色器中使用名为samplerCube的特殊类型的采样器访问纹

理。在纹理立方体贴图中，从采样器返回的值是沿着方向向量(

s

, ,

t

)从原点“看到”的纹素。因此，我们通常可以简单地使用传入的插

r

值顶点位置作为纹理坐标。在顶点着色器中，我们将立方体顶点位置

分配到输出纹理坐标属性中，以便在它们到达片段着色器时进行插 值。另外需要注意，在顶点着色器中，我们将传入的视图矩阵转换为 3×3，然后再转换回4×4。这个“技巧”有效地移除了平移分量，同 时保留了旋转（回想一下，平移值在转换矩阵的第四列中）。这样， 就将立方体贴图固定在了摄像机位置，同时仍允许合成相机“环顾四 周”。

程序9.2 OpenGL立方体贴图天空盒

C++/OpenGL application

. . .

int brickTexture, skyboxTexture;

int renderingProgram, renderingProgramCubeMap;

. . .

void init(GLFWwindow\* window) { renderingProgram =

Utils::createShaderProgram("vertShader.glsl", "fragShader.glsl"); renderingProgramCubeMap =

Utils::createShaderProgram("vertCShader.glsl", "fragCShader.glsl");

setupVertices();

brickTexture = Utils::loadTexture("brick1.jpg"); // 场景中 的环面

skyboxTexture = Utils::loadCubeMap("cubeMap"); // 包含天 空盒纹理的文件夹

glEnable(GL\_TEXTURE\_CUBE\_MAP\_SEAMLESS);

}

void display(GLFWwindow\* window, double currentTime) {

// 清除颜色缓冲区和深度缓冲区，并像之前一样创建投影视图矩阵和摄像机视图矩阵

. . .

// 准备首先绘制天空盒—注意，现在它的渲染程序不同了

glUseProgram(renderingProgramCubeMap);

// 将P、V矩阵传入相应的统一变量

. . .

// 初始化立方体的顶点缓冲区（这里不再需要纹理坐标缓冲区）

glBindBuffer(GL\_ARRAY\_BUFFER, vbo[0]); glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0); glEnableVertexAttribArray(0);

// 激活立方体贴图纹理

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_CUBE\_MAP, skyboxTexture);

// 禁用深度测试，之后绘制立方体贴图 glEnable(GL\_CULL\_FACE); glFrontFace(GL\_CCW); glDisable(GL\_DEPTH\_TEST); glDrawArrays(GL\_TRIANGLES, 0, 36); glEnable(GL\_DEPTH\_TEST);

// 绘制场景其余内容

. . .

}

GLuint Utils::loadCubeMap(const char \*mapDir) { GLuint textureRef;

// 假设6个纹理图像文件xp、xn、yp、yn、zp、zn都是JPG格式图像

string xp = mapDir; xp = xp + "/xp.jpg"; string xn = mapDir; xn = xn + "/xn.jpg"; string yp = mapDir; yp = yp + "/yp.jpg"; string yn = mapDir; yn = yn + "/yn.jpg"; string zp = mapDir; zp = zp + "/zp.jpg"; string zn = mapDir; zn = zn + "/zn.jpg";

textureRef = SOIL\_load\_OGL\_cubemap(xp.c\_str(), xn.c\_str(), yp.c\_str(), yn.c\_str(),

zp.c\_str(), zn.c\_str(), SOIL\_LOAD\_AUTO, SOIL\_CREATE\_NEW\_ID, SOIL\_FLAG\_MIPMAPS);

if (textureRef == 0) cout << "didnt find cube map image file"

<< endl;

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, textureRef);

// 减少接缝

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_R, GL\_CLAMP\_TO\_EDGE);

return textureRef;

}

顶点着色器

#version 430

layout (location = 0) in vec3 position; out vec3 tc;

uniform mat4 v\_matrix; uniform mat4 proj\_matrix;

layout (binding = 0) uniform samplerCube samp;

void main(void)

{

tc = position; // 纹理坐标就是顶点坐

标

mat4 vrot\_matrix = mat4(mat3(v\_matrix)); // 从视图矩阵中删除平

移

gl\_Position = proj\_matrix \* vrot\_matrix \* vec4(position, 1.0);

}

片段着色器

#version 430 in vec3 tc;

out vec4 fragColor;

uniform mat4 v\_matrix; uniform mat4 proj\_matrix;

layout (binding = 0) uniform samplerCube samp;

void main(void)

{ fragColor = texture(samp,tc);

}

9.4 环境贴图

在照明和材质章节中，我们考虑了物体的“光泽”。然而，我们 从未对非常闪亮的物体进行建模，例如镜子或铬制品。这些物体在有 小范围镜面高光的同时，还能够反射出周围物体的镜像。当我们看向 这些物品时，我们会看到房间里的其他东西，有时甚至会看到我们自 己的倒影。ADS照明模型并没有提供模拟这种效果的方法。

不过，纹理立方体贴图提供了一种相对简单的方法来模拟（至少

部分模拟）反射表面。其诀窍是使用立方体贴图来构造反射对象本 身。[[1]](#_bookmark170)如果想要做得看起来真实，则需要找我们从物体上看到的周围 环境所对应的纹理坐标。

图9.9展示了使用视图向量和法向量组合计算反射向量的策略，之 后，该反射向量会用来从立方体贴图中查找纹素。因此，反射向量可 用来直接访问纹理立方体贴图。当立方体贴图用于上述功能时，称其 为环境贴图。

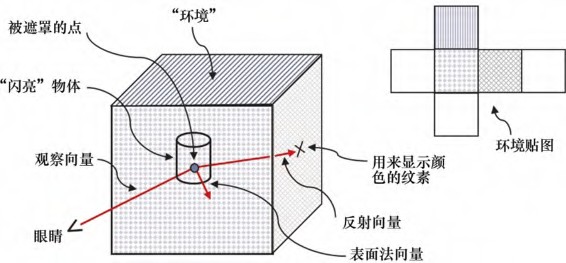


图9.9 环境贴图总览

我们在之前研究Blinn-Phong照明时计算过反射向量。除了我们现 在使用反射向量从纹理贴图中查找值，这里的反射向量概念和之前类 似。这种技术称为环境贴图或反射贴图。如果使用我们描述的第二种 方法（在9.3.2小节中，使用OpenGL GL\_TEXTURE\_CUBE\_MAP）实现立方 体贴图，那么OpenGL可以使用与之前为立方体添加纹理相同的方法来 进行环境贴图查找。我们使用视图向量和曲面法向量计算视图向量对 应的离开对象表面的反射向量。然后可以使用反射向量直接对纹理立

方体贴图图像进行采样。查找过程由OpenGL samplerCube辅助实现；

回忆上一节中，samplerCube使用视图方向向量索引。因此，反射向量 非常适用于查找所需的纹素。

实现环境贴图需要添加相对少量的代码。程序9.3展示了 display()函数和init()函数以及相关着色器中的更改，以使用环境贴 图渲染“反射”环面。所有更改都已经高亮显示。值得注意的是，如 果使用了Blinn-Phong照明，那么很多需要添加的代码可能已经存在 了。真正新的代码部分在片段着色器中[在main()函数中]。

乍一看程序9.3中突出显示的代码好像并不是新代码。实际上，在 我们研究照明的时候，已经看到过几乎相同的代码。然而，在当前情 况下，法向量和反射向量用于完全不同的目的。在之前的代码中，它 们用于实现ADS照明模型。而在这里，它们用于计算环境贴图的纹理坐 标。因此，我们将部分代码高亮，以便读者可以更轻松地追踪法向量 和反射向量计算的使用，以实现这一新目的。

渲染的结果会显示使用了环境贴图的“铬制”环面，如图9.10所 示（见彩插）。



图9.10 用于创建反射环面的环境贴图示例

程序9.3 环境贴图

void display(GLFWwindow\* window, double currentTime) {

// 用来绘制立方体贴图的代码未改变

. . .

// 所有修改都在绘制环面的部分

glUseProgram(renderingProgram);

// 矩阵变换的统一变量位置，包括法向量的变换

mvLloc = glGetUniformLocation(renderingProgram, "mv\_matrix"); projLoc = glGetUniformLocation(renderingProgram,

"proj\_matrix"); nLoc = glGetUniformLocation(renderingProgram, "norm\_matrix");

// 构建MODEL矩阵，如前

mMat = glm::translate(glm::mat4(1.0f), glm::vec3(torLocX, torLocY, torLocZ));

// 构建MODEL-VIEW矩阵，如前

mvMat = vMat \* mMat; invTrMat = glm::transpose(glm::inverse(mvMat));

// 法向量变换现在在统一变量中

glUniformMatrix4fv(mvLoc, 1, GL\_FALSE, glm::value\_ptr(mvMat)); glUniformMatrix4fv(projLoc, 1, GL\_FALSE, glm::value\_ptr(pMat)); glUniformMatrix4fv(nLoc, 1, GL\_FALSE ,

glm::value\_ptr(invTrMat));

// 激活环面顶点缓冲区，如前

glBindBuffer(GL\_ARRAY\_BUFFER, vbo[1]); glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0); glEnableVertexAttribArray(0);

// 我们需要激活环面法向量缓冲区 glBindBuffer(GL\_ARRAY\_BUFFER, vbo[2]); glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 0, 0); glEnableVertexAttribArray(1);

// 环面纹理现在是立方体贴图 glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_CUBE\_MAP, skyboxTexture);

// 绘制环面的过程未做更改

glClear(GL\_DEPTH\_BUFFER\_BIT); glEnable(GL\_CULL\_FACE); glFrontFace(GL\_CCW); glDepthFunc(GL\_LEQUAL);

0);

}

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, vbo[3]); glDrawElements(GL\_TRIANGLES, numTorusIndices, GL\_UNSIGNED\_INT,

顶点着色器

#version 430

layout (location = 0) in vec3 position; layout (location = 1) in vec3 normal; out vec3 varyingNormal; out vec3 varyingVertPos;

uniform mat4 mv\_matrix; uniform mat4 proj\_matrix; uniform mat4 norm\_matrix;

layout (binding = 0) uniform samplerCube tex\_map;

void main(void)

{ varyingVertPos = (mv\_matrix \* vec4(position,1.0)).xyz; varyingNormal = (norm\_matrix \* vec4(normal,1.0)).xyz; gl\_Position = proj\_matrix \* mv\_matrix \* vec4(position,1.0);

}

片段着色器

#version 430

in vec3 varyingNormal; in vec3 varyingVertPos; out vec4 fragColor; uniform mat4 mv\_matrix;

uniform mat4 proj\_matrix; uniform mat4 norm\_matrix; layout (binding = 0) uniform samplerCube tex\_map;

void main(void)

{ vec3 r = -reflect(normalize(-varyingVertPos), normalize(varyingNormal));

fragColor = texture(tex\_map, r);

}

虽然该场景需要两组着色器—— 一组用于立方体贴图，另一组用 于环面——但是程序9.3中仅展示了用于绘制环面的着色器。这是因为

用于渲染立方体贴图的着色器与程序9.2中的相同。通过对程序9.2的

修改得到程序9.3的过程，总结如下。 在init()函数中：

创建环面的法向量缓冲区[实际上在setupVertices()中完成，由 init()调用]；

不再需要环面的纹理坐标缓冲区。 在display()函数中：

创建用于变换法向量的矩阵（在第7章中称为“norm\_matrix”） 并将其连接到关联的统一变量；

激活环面法向量缓冲区； 激活纹理立方体贴图为环面的纹理（而非之前的“砖”纹理）。

在顶点着色器中： 将法向量和norm\_matrix相加；

输出变换的顶点和法向量以备计算反射向量，与在照明和阴影章

节中所做的相似。 在片段着色器中：

以与照明章节中相似的方式计算反射向量； 从纹理（现在是立方体贴图）检索输出颜色，使用反射向量而非 纹理坐标进行查找。

图9.10中显示的渲染结果是一个很好的例子，展示了通过简单的 技巧能够实现强大的幻觉。通过在对象上简单地绘制背景，我们使对

象看起来有“金属质感”，而根本没有进行ADS材质建模。即使没有任

何ADS照明被整合到场景中，这种技巧也能让人感觉光从物体反射出 来。在这个例子中，我们甚至会感到在环面的左下方似乎有一个镜面 高光，因为立方体贴图中包括太阳在水中反射的倒影。

# 补充说明

正如我们在第5章中第一次研究纹理时的情况一样，使用SOIL2使 得构建立方体贴图和为立方体贴图添加纹理变得容易。同时它也可能 会有一些副作用，即阻挡用户学习一些有用的OpenGL细节内容。当 然，用户也可以在没有SOIL2的情况下实例化并加载OpenGL立方体贴 图。虽然该主题超出了本书的范围，但基本步骤如下：

（1）使用C++工具读取6个图像文件（它们必须是正方形）；

（2）使用glGenTextures()为立方体贴图创建纹理及其整型引 用；

（3）调用glBindTexture()，指定纹理的ID和 GL\_TEXTURE\_CUBE\_MAP；

（4）使用glTexStorage2D()指定立方体贴图的存储需求；

（5）调用glTexImage2D()或glTexSubImage2D()将图像分配给立 方体的各个面。

更多有关在没有SOIL2的情况下创建OpenGL立方体贴图的详细信

息，请浏览互联网上的一些相关教程[dV14], [GE16]。

如本章所述，环境贴图的主要限制之一是它只能构建反射立方体 贴图内容的对象。在场景中渲染的其他对象并不会出现在使用贴图模 拟反射的对象中。这种限制是否可以接受取决于场景的性质。如果场 景中存在必须出现在镜面或铬制对象中的对象，则必须使用其他方 法。一种常见的方法是使用模板缓冲区（在第8章中有提到），许多网 络教程（例如[OV12]、[NE14]和[GR16]）中都有描述，不过它超出了本书 的范围。

我们没有介绍天空穹顶的实现，虽然它们在某些方面可以说比天 空盒更简单，并且不易受到失真的影响，甚至用它实现环境贴图也更 简单——至少在数学上——但OpenGL对立方体贴图的支持常常使得天 空盒更加实用。

在书后面部分涵盖的主题中，天空盒和天幕在概念上可以说是最 简单的。然而，让它们看起来令人信服可能会耗费大量时间。我们只 简要介绍了可能出现的一些问题（例如接缝），但根据使用的纹理图 像文件，可能会出现其他问题，需要额外修复。尤其是在动画场景中 或相机可以通过交互进行移动时。

我们还大致介绍了如何生成可用且令人信服的纹理立方体贴图图 像。这方面有许多优秀的工具，其中最受欢迎的是Terragen [TE16]。

本章中的所有立方体贴图均由作者使用Terragen制作（图9.6中的星域

图除外）。

习题

9.1 （项目）在程序9.2中，添加使用鼠标或键盘移动相机的功 能。为此，你需要使用先前在习题4.2中开发的代码来构建视图矩阵。 还要为前后移动以及绕各轴旋转相机的功能分配鼠标或键盘操作（你 需要编写这些函数）。完成这些操作后，你应该能够在场景中“飞来 飞去”，并能够注意到天空盒始终看起来保持在遥远的地平线上。

9.2 在6个立方体贴图图像文件上绘制标签，以确认实现中使用 了正确的方向。例如，你可以在图像上绘制轴标签，如图9.11所示。



图9.11 6个立方体贴图

还可以使用“带标记的”立方体贴图来验证环境贴图环面中的反 射是否正确呈现。

9.3 （项目）向程序9.3添加动画，以便场景中的一个（或多 个）使用了环境贴图的对象旋转或翻转。当天空盒纹理在物体表面上 移动时，物体的模拟反射性质会更明显。

9.4 （项目）修改程序9.3，以使场景中的对象将环境贴图与纹

理混合。在片段着色器中加权求和，如第7章中所述。

9.5 （研究和项目）了解使用Terragen [TE16]创建简单立方体贴 图的基础知识。通常需要（在Terragen中）制作具有所需地形和大气 模式的“世界”，然后将Terragen的合成相机放置在前、后、右、 左、顶部和底部以保存作为各视图的6个图像。在程序9.2和程序9.3中 使用新生成的图像，观察它们作为立方体贴图和环境贴图呈现的外 观。使用Terragen的免费版本足以进行此练习。

# 参考资料

[BO01] P. Bourke, “Representing Star Fields,” October 2018, accessed July 2016.

[dV14] J. de Vries, “Learn OpenGL – Cubemaps,” 2014, accessed October 2018.

[GE16] A. Gerdelan, “Cube Maps: Sky Boxes and Environment Mapping,” 2016, accessed October 2018.

[GI16] GNU Image Manipulation Program, accessed October 2018.

[GR16] OpenGL Resources, “Planar Reflections and Refractions Using the Stencil Buffer,” accessed October

2018.

[NE14] NeHeProductions, “Clipping and Reflections Using the Stencil Buffer,” 2014, accessed October 2018.

[OV12] A. Overvoorde, “Depth and Stencils,” 2012, accessed October 2018.

[TE16] Terragen, Planetside Software, LLC, accessed October 2018.

[[1]](#_bookmark166) 同样的技巧也适用于通过对反光物体添加天空穹顶纹理图像来用 天空穹顶替代天空盒的情况。

第10章 增强表面细节

假设我们想要对不规则表面的物体进行建模，例如橘子凹凸的表 皮、葡萄干褶皱的表面或月球的陨石坑表面。我们该怎么做？到目前 为止，我们已经学会了两种可能的方法：（a）我们可以对整个不规则 表面进行建模，但这么做通常不切实际（一个有许多坑的表面需要大 量的顶点）；（b）我们可以将不规则表面的纹理图图像应用于平滑的 对象。第二种选择通常比较高效。但是，如果场景中有光源，当光源

（或摄像机角度）移动时，我们很快就会发现物体使用了静态纹理渲 染（以及物体表面是平滑的），因为纹理上的亮区和暗区不会像真正 凹凸不平的表面那样，随着光源或摄像机移动而改变。

在本章中，我们将探讨几种与实现凹凸表面相关的方法，通过使 用光照效果，即使在实际对象模型表面平滑的情况下，也能使对象看 起来具有逼真的表面纹理。我们将首先观察凹凸贴图和法线贴图，当 直接为对象添加微小表面细节会使得计算代价过高时，它们可以为场 景中的对象增加相当程度的真实感。我们还将研究通过高度贴图实际 扰乱光滑表面中顶点的方法，这对于生成地形（和其他一些用途）非 常有用。

10.1 凹凸贴图

在第7章中，我们了解了表面法向量在创建令人信服的光照效果中 是至关重要的。像素处的光强度主要由反射角确定，即需要考虑到光 源位置、相机位置和像素处的法向量。因此，如果我们能找到生成相 应法向量的方法，就可以避免生成与凹凸不平或褶皱表面相对应的顶 点。

图10.1展示了对于单个“凸起”修改法向量的概念。

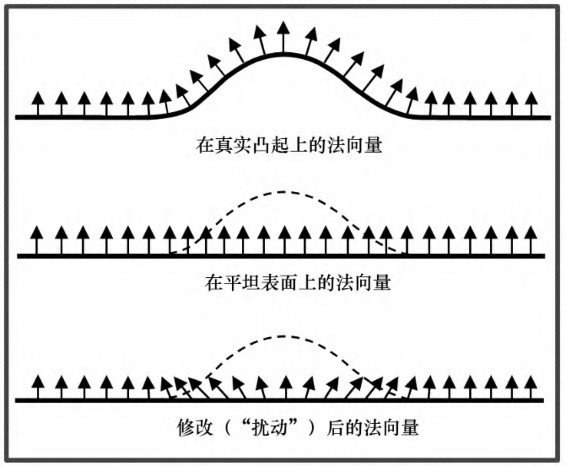


图10.1 用于凹凸贴图的扰动法向量

因此，如果我们想让一个物体看起来好像有凹凸（或皱纹，陨石 坑等），一种方法是计算当表面确实凹凸不平时其上的法向量。当场 景点亮时，光照会让人产生我们所期望的幻觉。这是Blinn在1978年首 次提出的[BL78]，随着在片段着色器拥有了可以对每个像素进行光照计 算的能力，这种方法就变得切实可行了。

程序10.1中展示了顶点着色器和片段着色器的一个示例，这段程 序会生成一个带有“高尔夫球”表面的环面，如图10.2所示。其代码 几乎与我们之前在程序7.2中看到的相同。片段着色器中唯一显著的变 化是——输入的已插值法向量（在原程序中名为“varyingNormal”） 在这里变得凹凸不平了，其方法是对环面模型的原始（未变形）顶点

的 、 和

X Y Z

轴应用正弦函数。请注意，这里需要顶点着色器将未经变换

的顶点沿管线传递给片段着色器。

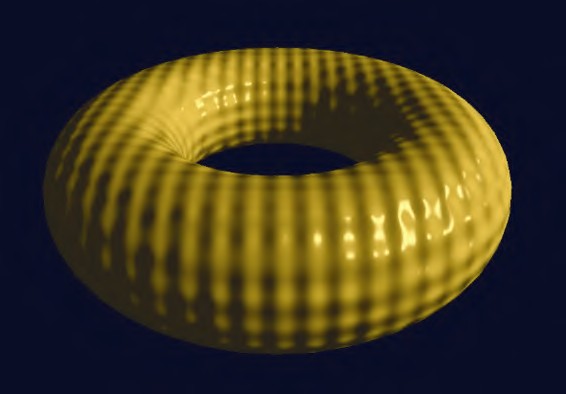


图10.2 过程式凹凸贴图示例

以这种方式对法向量进行改变，即在运行时使用数学函数进行计 算，称为过程式凹凸贴图。

程序10.1 过程式凹凸贴图

顶点着色器

#version 430

// 与Phong着色相同，但添加此输出顶点属性

out vec3 originalVertex;

. . .

void main(void)

{ // 添加原始顶点，传递以进行插值

originalVertex = vertPos;

. . .

}

片段着色器

#version 430

// 与Phong着色相同，但添加此输入顶点属性

in vec3 originalVertex;

. . .

void main(void)

{ . . .

// 添加如下代码以扰乱传入的法向量 float a = 0.25; // a 控制凸起的高度 float b = 100.0; // b 控制凸起的宽度 float x = originalVertex.x;

float y = originalVertex.y; float z = originalVertex.z;

N.x = varyingNormal.x + a\*sin(b\*x); // 使用正弦函数扰乱传入法向量

N.y = varyingNormal.y + a\*sin(b\*y);

N.z = varyingNormal.z + a\*sin(b\*z); N = normalize(N);

// 光照计算以及输出的fragColor（未更改）现在使用扰动过的法向量N

. . .

}

10.2 法线贴图

凹凸贴图的一种替代方法是使用查找表来替换法向量。这样我们 就可以在不依赖数学函数的情况下，对凸起进行构造，例如月球上的 陨石坑所对应的凸起。一种使用查找表的常见方法叫作法线贴图。

为了理解法线贴图的工作原理，我们首先注意，向量通过3字节存

储， 、

X Y

和 分量各占1字节，就可以达到合理的精度。这样，我们就

Z

可以将法向量存储在彩色图像文件中，其中 、 和

R G B

分量分别对应于

、 和

1. Y Z

。图像中的

RGB

值以字节存储，通常被解释为[0…1]范围内的

值，但是向量可以有正负值分量。如果我们将法向量分量限制在[−1…

+1]范围内，那么在图像文件中将法向量

N

存储为像素的简单转换是：



法线贴图使用一个图像文件（称为法线贴图），该图像文件包含 在光照下所期望表面外观的法向量。在法线贴图中，向量相对于任意

平面 表示，其

XY X

和 分量表示与“垂直”的偏差，其

1. Z

分量设置为1，

严格垂直于

XY

平面的向量（即没有偏差）将表示为（0, 0, 1），而不

垂直的向量将具有非零的

X

和/或

Y

分量。我们需要使用上面的公式将值

转换至

RGB

空间；例如，（0, 0, 1）将存储为（0.5, 0.5, 1），因为

实际偏移的范围为[−1…+1]，而

RGB

值的范围为[0…1]。

我们可以通过纹理单元的另一种妙用来生成这样一幅法线贴图： 我们在纹理单元中存储所需的法向量而非颜色。然后，在给定片段 中，我们就可以使用采样器从法线贴图中查找值，接下来，我们将所 得的值作为法向量，而非输出像素颜色（在纹理贴图中我们是这么做 的）。

图10.3展示了一个法线贴图图像文件的例子，通过将GIMP法线贴 图插件[GI16]应用于Luna [LU16]纹理而生成。法线贴图图像文件并不适 合作为图像查看，我们展示这幅图就是为了指明这一点，法线贴图最

终看起来基本都是蓝色的。这是因为图像文件中每个像素的

B

值（蓝色

值）都是1（最大蓝色值），这会让它在作为图像时看起来是“蓝色 的”。



图10.3 法线贴图图像文件示例[ME11]

图10.4展示了两个不同的法线贴图图像文件（它们都由Luna

[LU16]的纹理构建）以及在Blinn-Phong光照模型下将它们应用于球体

的结果。



图10.4 法线贴图示例

从法线贴图查找到的法向量不能直接使用，因为它们是相对于上

述的任意

XY

平面定义的，并没有考虑它们在物体上的位置以及在相机

空间中的方向。这个问题的解决策略是建立一个转换矩阵，用于将法 向量转换为相机空间，如下所示。

在对象的每个顶点处，我们考虑与对象相切的平面。顶点处的物 体的法向量垂直于该切面。我们在该切面中定义两个相互垂直的向 量，同时也垂直于法向量，称为切向量和副切向量（有时称为副法向 量）。构造我们期望的变换矩阵要求我们的模型包括每个顶点的切向 量（可以通过计算切向量和法向量的叉积来构建副切向量）。如果模 型中没有定义切向量，则需要通过计算得到它们。在球体的情况下， 可以通过计算得到精确的切向量。以下是对程序6.1的修改：

. . .

for (int i=0; i<=prec; i++) {

for (int j=0; j<=prec; j++) {

float y = (float)cos(toRadians(180.0f - i\*180.0f / prec)); float x = -(float)cos(toRadians(j\*360.0f / prec)) \*

(float)abs(cos(asin(y)));

float z = (float)sin(toRadians(j\*360.0f / prec)) \* (float)abs(cos(asin(y)));

vertices[i\*(prec+1)+j] = glm::vec3(x, y, z);

// 计算切向量

if (((x==0) && (y==1) && (z==0)) || ((x==0) && (y==-1) && (z==0))) // 如果是北极或南极，

{ tangent[i\*(prec+1)+j] = glm::vec3(0.0f, 0.0f, -1.0f);

// 设置切向量为 -Z 轴

}

else // 否则，计算切向量

{ tangent[i\*(prec+1)+j] = glm::cross(glm::vec3(0.0f, 1.0f, 0.0f), glm::vec3(x,y,z));

}

. . . // 其余计算代码不变

}

}

对于那些表面不可导以至于无法精确求解切向量的模型，其切向 量可以通过近似得到，例如在构造（或加载）模型时，将每个顶点指 向下一个顶点的向量作为切向量。请注意，这种近似可能会导致切向 量与顶点法向量不严格垂直。因此，如果要实现适用于各种模型的法 线贴图，需要考虑这种可能性（我们的解决方案中对此进行了处 理）。

切向量与顶点、纹理坐标以及法向量一样，是从缓冲区（VBO）传 递到顶点着色器中的顶点属性。然后，顶点着色器通过应用MV矩阵的 逆转置并将结果沿着流水线转发以由光栅器进行插值并最终进入片段 着色器，从而对正常向量进行处理。逆转置的应用将法向量和切向量 转换为相机空间，之后我们使用叉积构造副切向量。

一旦我们在相机空间中得到法向量、切向量和副切向量，就可以 使用它们来构造矩阵（依其分量命名为“TBN”矩阵），该矩阵用于将 从法线贴图中检索到的法向量转换为在相机空间中相对于物体表面的 法向量。

在片段着色器中，新法向量的计算在calcNewNormal()函数中完 成。函数的第三行［包含dot（tangent，normal）］的计算确保切向 量垂直于法向量。新的切向量和法向量的叉积就是副切向量。

然后，我们创建一个类型为mat3的3×3矩阵，作为TBN。mat3构造 函数接收3个向量作为参数，生成一个矩阵，其中顶行是第一个向量， 中间行是第二个向量，底行是第三个向量（类似于从摄像机位置构建 视图矩阵，见图3.13）。

着色器使用片段的纹理坐标来提取与当前片段对应的法线贴图单 元。着色器在提取时使用采样器变量“normMap”，并被绑定到纹理单 元0（注意：因此在C++ / OpenGL应用程序中必须将法线贴图图像附加 到纹理单元0）。因为需要将颜色分量从纹理中存储范围[0…1]转换为

其原始范围[−1 … + 1]，我们将其乘以2.0再减去1.0。

然后将TBN矩阵应用于所得法向量以产生当前像素的最终法向量。

着色器的其余部分与用于Phong光照的片段着色器相同。片段着色器代 码基于Etay Meiri [ME11]的版本，如程序10.2所示。

制作法线贴图图像可以使用各种各样的工具。有的图像编辑工具 就有制作法线贴图的功能，例如GIMP [GI16]和Photoshop [PH16]。它们 通过分析图像中的边缘，推断凸起和凹陷，并产生相应的法线贴图。

图10.5显示了由Hastings-Trew [HT16]基于NASA卫星数据创建的月 面纹理图。其相应的法线贴图由GIMP法线贴图插件[GP16]，通过处理由 Hastings-Trew创建的黑白版本月面纹理图生成。

程序10.2 法线贴图片段着色器

#version 430

in vec3 varyingLightDir; in vec3 varyingVertPos; in vec3 varyingNormal; in vec3 varyingTangent; in vec3 originalVertex; in vec2 tc;

in vec3 varyingHalfVector; out vec4 fragColor;

layout (binding=0) uniform sampler2D normMap;

// 其余统一变量同前

. . .

vec3 calcNewNormal()

{ vec3 normal = normalize(varyingNormal); vec3 tangent = normalize(varyingTangent); tangent = normalize(tangent - dot(tangent, normal) \* normal); //

切向量垂直于法向量

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| vec3 | bitangent = cross(tangent, normal); | | | | |
| mat3 | tbn | = | mat3(tangent, | bitangent, | normal); |

机空间的TBN矩阵

vec3 retrievedNormal = texture(normMap,tc).xyz;

// 用来变换到相

retrievedNormal = retrievedNormal \* 2.0 - 1.0; // 从RGB空间转换

vec3 newNormal = tbn \* retrievedNormal; newNormal = normalize(newNormal); return newNormal;

}

void main(void)

{ // 正规化光照向量，法向量和视图向量

vec3 L = normalize(varyingLightDir); vec3 V = normalize(-varyingVertPos); vec3 N = calcNewNormal();

// 获得光照向量和曲面法向量之间的角度

float cosTheta = dot(L,N);

// 为Blinn优化计算半向量

vec3 H = normalize(varyingHalfVector);

// 视图向量和反射光向量之间的角度

float cosPhi = dot(H,N);

// 计算ADS贡献（每个像素）

fragColor = globalAmbient \* material.ambient

+ light.ambient \* material.ambient

+ light.diffuse \* material.diffuse \* max(cosTheta,0.0)

+ light.specular \* material.specular \* pow(max(cosPhi,0.0), material.shininess\*3.0);

}

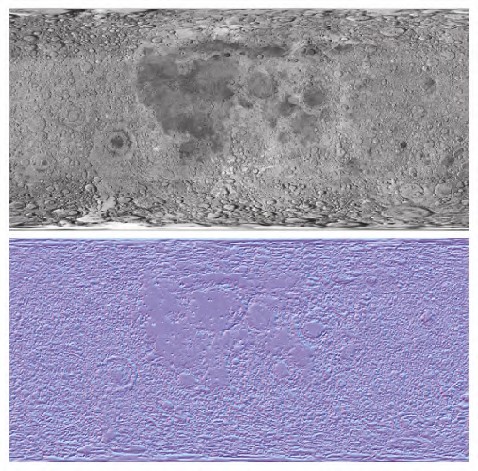


图10.5 月球纹理（上）和法线贴图（下）

图10.6展示了使用两种不同方式渲染的，用以表现月球表面的球 体。图10.6左图中，球体使用了原始的纹理贴图；图10.6右图中，球 体使用法线贴图的图像作为纹理（供参考）。它们都没有应用法线贴 图。虽然左侧使用了纹理的“月球”非常逼真，但仔细观察可以发 现，纹理图案很明显拍摄于阳光从左侧照亮月球的时候，因为其山脊 的阴影投射到了右侧（在底部中心附近的火山口中最明显）。如果我 们使用Phong着色为此场景添加光照，然后移动月球、相机或灯光来给 场景添加动画，就会发现月球上的阴影不会如我们期望地改变。

此外，随着光源的移动（或相机移动），期望中会在山脊上出现 许多镜面高光。但是图10.6左图使用了标准纹理的球体将只产生一个 镜面高光，对应于光滑球体上所出现的高光，这看起来非常不现实。 配合法线贴图可以显著提高这类对象在光照下的真实感。

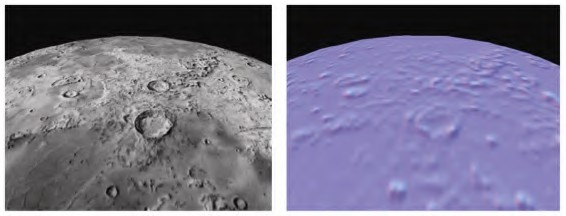


图10.6 使用月面纹理的球体（左）和使用法线贴图的球体（右）

当我们在球体上使用法线贴图（而不是纹理）时，我们会得到图 10.7所示的结果。尽管它不像标准纹理那么真实（现在），但是现在 它确实响应了光照变化。图10.7的第一张图像中从左侧进行光照，第 二张图像中则从右侧进行光照。请注意蓝色和黄色箭头所示部分展示 了山脊周围漫反射光的变化以及镜面反射高光的移动。

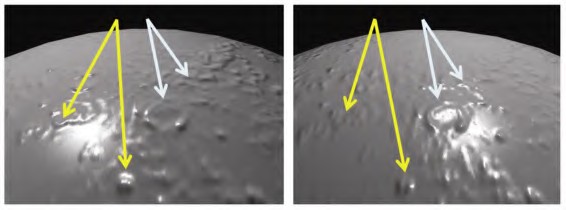


图10.7 法线贴图对月球的影响

图10.8展示了在使用Phong光照模型的情况下，将法线贴图与标准 纹理相结合的效果。月球的图像通过漫射区域进行了增强，镜面高光 区域也会响应光源的移动（或相机或物体移动）。两个图像中的光照 分别来自左侧和右侧。

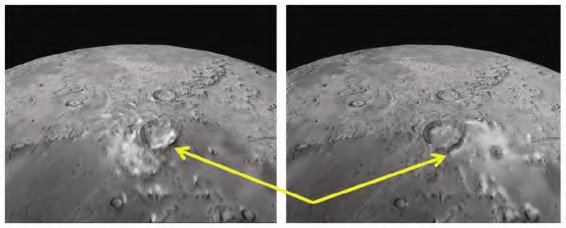


图10.8 纹理加法线贴图，分别从左侧和右侧进行光照

我们的程序现在需要两个纹理—— 一个用于月球表面图像，一个 用于法线贴图——因此需要有两个采样器。片段着色器使用之前在7.6

节中所描述的技术，将纹理颜色与经光照计算所得的颜色进行混合， 如程序10.3所示。

程序10.3 纹理加法线贴图

// 片段着色器中的变量和结构与之前相同，加上

layout (binding=0) uniform sampler2D s0; layout (binding=1) uniform sampler2D s1; void main(void)

{ // 计算与之前相同，直到

// 法线贴图

// 纹理

vec3 N = calcNewNormal();

vec4 texel = texture(s1,tc);

. . .

// 反射计算与之前相同，然后混合结果

fragColor = globalAmbient +

// 标准纹理

texel \* (light.ambient + light.diffuse \* max(cosTheta,0.0)

+ light.specular \* pow(max(cosPhi,0.0), material.shininess));

}

有趣的是，法线贴图可以从多级渐远纹理贴图（Mipmapping）中 受益，因为在第5章中看到的纹理化产生的“锯齿”伪影，在使用纹理 图像进行法线贴图时也会发生。图10.9分别展示了未使用多级渐远纹 理贴图和使用了多级渐远纹理贴图进行法线贴图的月球。尽管在静止 的图像中不容易观察到，但是左边的球体（未使用多级渐远纹理贴 图）周边有闪烁的伪影。

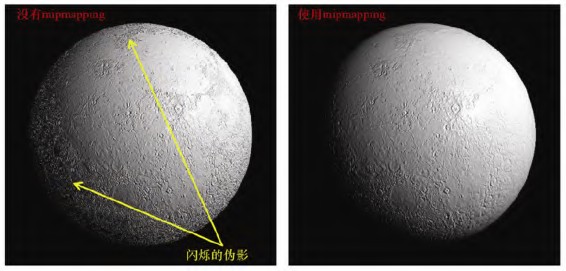


图10.9 发现贴图伪影，以及使用多级渐远纹理贴图校正后的图像

对于法线贴图而言，各向异性过滤（AF）更有效，它不但减少了 闪烁的伪影，同时还保留了细节，如图10.10所示（比较右下角边缘的 细节）。图10.11中展示了使用相等的纹理权重和光照权重，光照应用 了法线贴图及AF的情况下得到的结果。

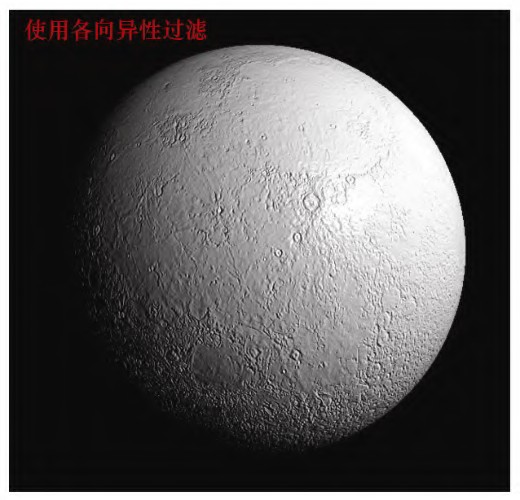


图10.10 使用各向异性过滤进行法线贴图

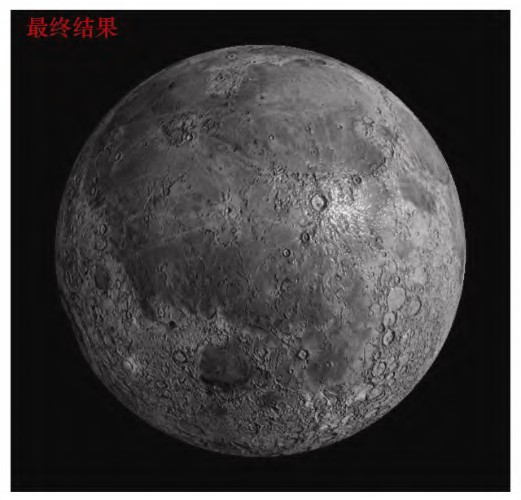


图10.11 纹理加各向异性过滤法线贴图

最终的渲染结果并不完美。无论光照如何，原始纹理图像中出现 的阴影仍将显示在渲染结果上。此外，虽然法线贴图可以影响漫反射 和镜面反射效果，但它无法投射阴影。因此，当表面特征较小时，最 适用法线贴图。

10.3 高度贴图

现在我们扩展法线贴图的概念——从纹理图像用于扰动法向量到 扰乱顶点位置本身。实际上，以这种方式修改对象的几何体具有一定 的优势，例如使表面特征沿着对象的边缘可见，并使特征能够响应阴 影贴图。我们将会看到，它还可以帮助构建地形。

一种实用的方法是使用纹理图像来存储高度值，然后使用该高度 值来提升（或降低）顶点位置。含有高度信息的图像称为高度图，使 用高度图更改对象的顶点的方法称为高度贴图[[1]](#_bookmark179)。高度图通常将高度 信息编码为灰度颜色：（0,0,0）（黑色）=低高度，（1,1,1）（白

色）=高高度。这样一来通过算法或使用“画图”程序就可以轻松创建 高度图。图像的对比度越高，其表示的高度变化越大。这些概念将在 图10.12（显示随机生成的地图）和图10.13（显示有组织的模式的地 图）中说明。

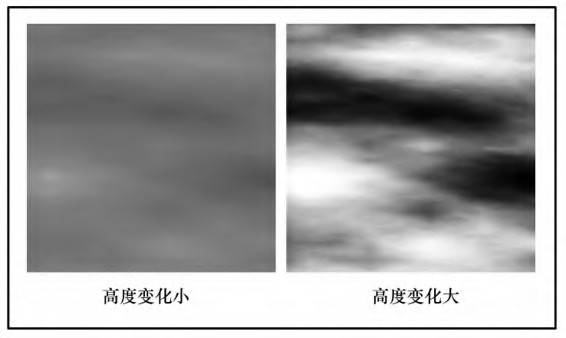


图10.12 高度图示例

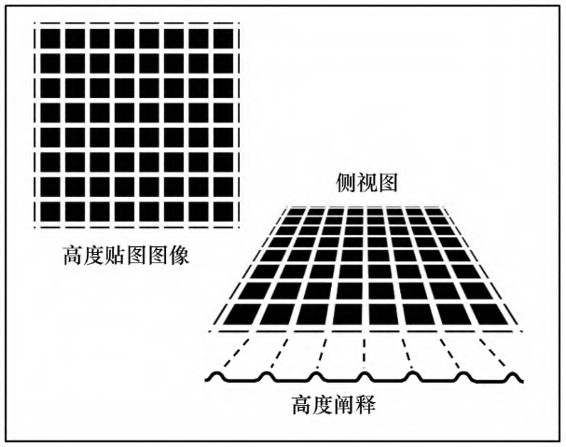


图10.13 高度贴图阐释

改变顶点位置是否有用取决于改变的模型。顶点操作可以在顶点 着色器中轻松完成，当模型顶点细节级别够高（例如在足够高精度的

球体中）时，改变顶点高度的方法效果很好。但是，当模型的顶点数 量很少（例如立方体的角）时，渲染对象的表面需要依赖于光栅器中 的顶点插值来填充细节。当顶点着色器中可用于改变高度的顶点很少 时，许多像素的高度将无法从高度图中检索，而需要由插值生成，从 而导致表面细节较差。当然，在片段着色器中是不可能进行顶点操作 的，因为这时顶点已被光栅化为像素位置。

程序10.4展示了一个将顶点“向外”（即在表面法向量的方向 上）移动的顶点着色器代码。它通过将顶点法向量乘以从高度图检索 所得的值，然后将该乘积与顶点位置相加，以“向外”移动顶点。

程序10.4 顶点着色器中的高度贴图

#version 430

layout (location=0) in vec3 vertPos; layout (location=1) in vec2 texCoord; layout (location=2) in vec3 vertNormal;

out vec2 tc;

uniform mat4 mv\_matrix; uniform mat4 proj\_matrix;

layout (binding=0) uniform sampler2D t; layout (binding=1) uniform sampler2D h;

// 用于纹理

// 用于高度图

void main(void)

{ // "p"是高度图所改变的顶点位置

// 由于高度图是灰度图，因此使用其任何颜色分量

// 都可以（我们使用"r"）。除以5.0用来调整高度

vec4 p = vec4(vertPos,1.0) + vec4( (vertNormal \* ((texture(h, texCoord).r) / 5.0f)),1.0f );

tc = tex\_coord;

gl\_Position = proj\_matrix \* mv\_matrix \* p;

}

图10.14（见彩插）展示了通过在画图程序中涂鸦创建的简单高度 图（左上角）。高度图图像中还绘制了一个白色矩形。绿色版本的高 度图（左下角）用作纹理。使用程序10.4中展示的着色器将高度图应 用于100×100的矩形网格模型时，会产生类似“地形”的感觉（如图 10.14右图所示）。注意白色矩形是如何生成右边的悬崖的。

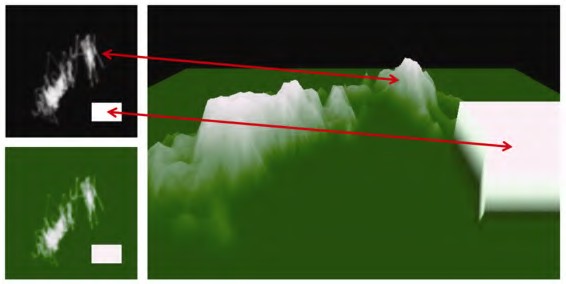


图10.14 地形，在顶点着色器中进行高度贴图

图10.14展示的渲染结果还算可以，因为模型（网格和球体）有足 够数量的顶点来对高度贴图值进行采样。也就是说，模型具有大量的 顶点，而高度图相对粗糙并且以低分辨率充分地采样。然而，仔细观 察仍然会发现存在分辨率伪影，例如沿图10.14中地形右侧凸起的矩形 盒子的左下边缘。凸起的矩形盒子两侧看起来不是完美矩形，而且颜 色有渐变效果，其原因是底层网格100像素×100像素的分辨率无法与 高度图中的白色矩形完全对齐，从而导致纹理的光栅化坐标沿侧面产 生伪影。

当尝试将其应用于要求更严苛的高度贴图时，在顶点着色器中进 行高度贴图的限制会进一步暴露。考虑图10.5中展示的月球图像。法 线贴图在捕获图像细节方面表现非常出色（如图10.9和图10.11所 示），而且由于它是灰度图，因此尝试将其作为高度图应用似乎很自 然。但是，基于顶点着色器的高度贴图会无法胜任这个任务，因为顶 点着色器中采样的顶点数（即使对于精度=500的球体）比起图像中的 细节级别，仍然太少。相较之下，法线贴图能够很好地捕获细节，因 为在片段着色器中对法线贴图的采样是像素级的。

我们将会在之后的第12章继续学习高度图，在那里我们会了解使 用曲面细分着色器生成大量顶点的方法。

补充说明

凹凸贴图或法线贴图的一个基本限制是，虽然它们能够在渲染对 象的内部提供表面细节的外观，但是物体轮廓（外边界）无法显示这 些细节（它保持平滑）。高度贴图在用于实际修改顶点位置时修复了 这个缺陷，但它也有其自身的局限性。正如我们将在本书后面看到 的，有时可以使用几何着色器或曲面细分着色器来增加顶点的数量， 使高度贴图更加实用、有效。

我们冒昧地简化了一些凹凸贴图和法线贴图计算。在重要应用中 可以使用更准确和/或更有效的解决方案[BN12]。

习题

10.1 使用程序10.1进行试验，修改片段着色器中的设置和/或计 算并观察结果。

10.2 使用绘图程序生成一份高度图并在程序10.4中使用它。尝 试识别由于顶点着色器无法充分采样高度图而缺少细节的位置。你可 能会发现使用高度图图像文件在对地形进行纹理化时也很有用，如图 10.14所示（或者使用可以暴露表面结构的图案，例如网格），这样你 就可以看到所生成地形中的山和谷。

10.3 （项目）向程序10.4中添加光照，以便进一步暴露高度贴 图地形的表面结构。

10.4 （项目）为练习10.3中的代码添加阴影贴图，从而使高度 贴图地形投射阴影。

参考资料

[BL78] J. Blinn, “Simulation of Wrinkled Surfaces,”

12, no. 3 (1978): 286–292.

Computer Graphics

[BN12] E. Bruneton and F. Neyret, “A Survey of Non- Linear Pre-Filtering Methods for Efficient and Accurate Surface Shading,”

IEEE Transactions on Visualization and

18, no. 2 (2012).

Computer Graphics

[GI16] GNU Image Manipulation Program, accessed October 2018.

[GP16] GIMP Plugin Registry, normalmap plugin, accessed October 2018.

[HT16] J. Hastings-Trew, JHT’s Planetary Pixel Emporium, accessed October 2018.

[LU16] F. Luna,

Introduction to 3D Game Programming with

, 2nd ed. (Mercury Learning, 2016).

DirectX 12

[ME11] E. Meiri, OGLdev tutorial 26, 2011, accessed October 2018.

[PH16] Adobe Photoshop, accessed October 2018. [SS16] SS Bump Generator, accessed October 2018.

[[1]](#_bookmark175) 这里使用了高度贴图说法，通过纹理图像更改顶点的方法一般称 为位移贴图/置换贴图。高度图除了用于位移贴图/置换贴图，有时也 用于视差贴图，请读者阅读时注意区别。——译者注

第11章 参数曲面

在20世纪50年代和60年代在雷诺公司工作期间，皮埃尔·贝塞尔

（Pierre Bézier）开发了用于设计汽车车身的软件系统。他的程序利 用了Paul de Casteljau之前开发的数学方程组，后者曾为竞争对手雪 铁龙汽车制造商[BE72，DC63]工作。de Casteljau方程仅使用几个标量 参数描述曲线，同时使用一种高明的的递归算法，称为“de Casteljau算法”，就可以生成任意精度的曲线。现在它们分别被称为 “贝塞尔曲线”和“贝塞尔曲面”，这些方法通常用于高效地对各种 曲面3D物体进行建模。

11.1 二次贝塞尔曲线

二次贝塞尔曲线由一组参数方程定义，方程组中使用3个控制点指 定特定的曲线的形状，每个控制点都是2D空间中的一个点。[[1]](#_bookmark189)考虑图

，

，

11.1中所示的一组3个点[

p

0

p1 p2]。

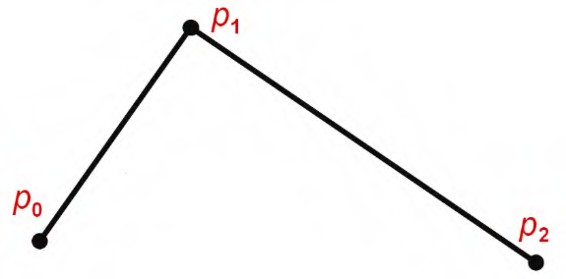


图11.1 贝塞尔曲线的控制点

通过引入参数

t

，我们可以构建一个用来定义曲线的参数方程组。

表示从一个控制点到另一控制点间线段距离的分数。对于在线段上的

t

点， 的值在[0…1]的范围内。图11.2显示了一个这样的值： =

t t

0.75，分别应用于连接

-

和

-

p p

0

1

p1 p2

的线段。通过

t

在两条原始线段上

定义了两个新点

)和

p

01

( p12

( )。我们对连接两个新点

t p

01

( p12( )

的线段重复该过程，产生点

)和

)和

t

t

t

P

( )，其中沿线段

t p

01

( p12

( )在 =

t t

0.75得到点

t

P

( )。

t P

( )是最终得到的曲线上的点，因此用大写字母 表

1. P

示。

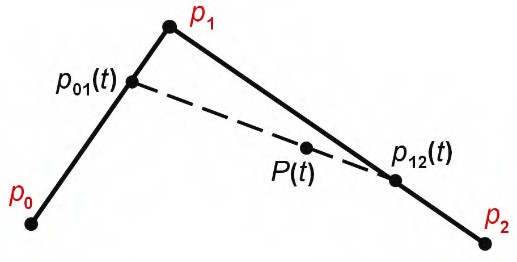


图11.2 参数位置处的点

t

= 0.75

针对各种

t

值收集大量的点

P

( )，则会产生一条曲线，如图11.3所

t

示。采样的

t

的参数值越多，生成的点 (

P t

)越多，得到的曲线则越平

滑。

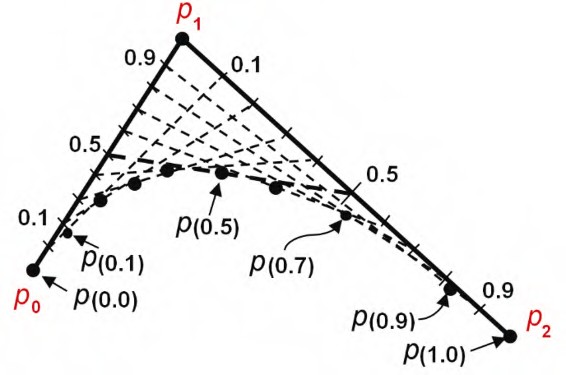


图11.3 建立二次贝塞尔曲线

现在可以导出二次贝塞尔曲线的分析定义。首先，我们注意到连

接两个点 和

pa pb

的线段

pa

− 上的任意点

pb p

可以用参数

t

表示如下：



使用该等式，我们解出点

p

01

p12

（分别在

p

0

p1 p1 p 上的

点）如下：

和

−

和

−

2



同理，在这两点所连接的线段上的点可以表示为：



p12

替换

和

p01

的定义得：



分解并重新合并各项可得：



或



其中



因此，我们通过控制点的加权和解出曲线上的任意点。加权函数

B

通常被称为“混合函数”（尽管名称“

B

”实际上源自Sergei

Bernstein [BE16]，他首先描述了这个多项式族）。请注意，混合函数 的形式都是二次的，这就是为什么得到的曲线称为二次贝塞尔曲线。

11.2 三次贝塞尔曲线

我们现在将曲线模型扩展到4个控制点，就会得到一个三次贝塞尔 曲线，如图11.4所示。与二次曲线相比，三次贝塞尔曲线能够定义的 形状更加丰富，而二次曲线仅限于定义凹形。

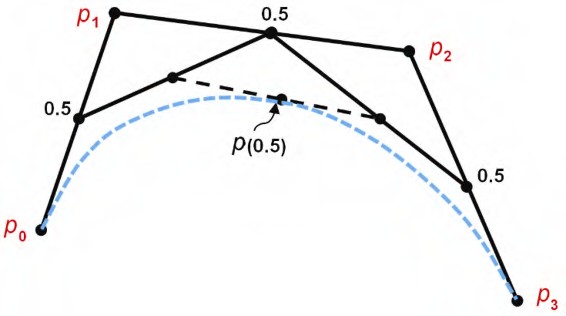
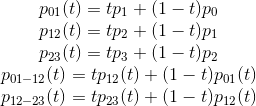


图11.4 建立一个三次贝塞尔曲线

同二次曲线时的情形，我们可以推导出三次贝塞尔曲线的解析定 义：



曲线上的点则是：



p12−23



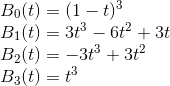
使用

和

其中：

p01−12

的定义替换等式中的项，再合并得：



渲染贝塞尔曲线时，可以使用许多不同的技术。其中一种方法

是，使用固定的增量，在0.0～1.0范围内，迭代增加得出

t

的后继值。

例如，当增量为0.1时，我们可以使用

t

值为0.0、0.1、0.2、0.3等的

循环。对于

t

的每个值，计算贝塞尔曲线上的对应点，并绘制连接连续

点的一系列线段，如图11.5中的算法所述。

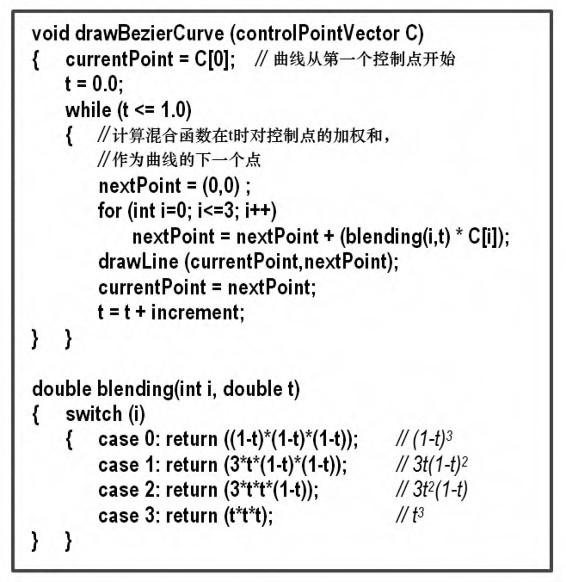


图11.5 渲染贝塞尔曲线的迭代算法

另一种方法是使用de Casteljau算法递归地将曲线对半细分，其

中，在每个递归步骤

t

= 1/2。图11.6展示了左侧曲线细分后的新三次

控制点（

，

，

q q q

0

1

2

），以绿色显示（见彩插）。该算法由de

q

，

3

Casteljau提出（完整推导见[AS14]）。

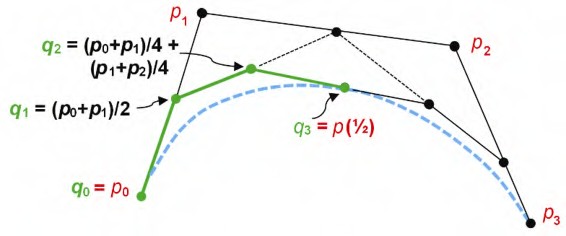


图11.6 细分三次贝塞尔曲线

算法见图11.7。该算法重复将曲线段细分为两半的过程，直到每

个曲线段足够直，进一步的细分不会产生实际的好处。在极限情况下

（随着生成的控制点越来越靠近），曲线段本身实际上与第一个控制

点和最后一个控制点（

q

0

）之间的线段相同。因此，可以通过比

q

和

3

较从第一控制点到最后一个控制点的距离与连接4个控制点的3条线段 的长度之和来确定曲线段是否“足够直”：



D1 D2

当

−

小于一个足够小的阈值时，进一步的细分就没有意义

了。

de Casteljau算法有一个有趣的特性，它可以在不使用之前描述

的混合函数的情况下，生成曲线上所有的点。同时请注意，

p

(1/2)处

的中心点是“共享”的，即它既是左细分中最右的控制点，也是右细

分中最左的控制点。它可以使用

t

= 1/2处的混合函数或使用由de

Casteljau导出的公式(

q

2

+

r

1

)/2来计算。

另请注意，图11.7中所示的subdivide()函数假定传入的参数 、

p q

和 是“引用”参数，因此，图11.7上方列出的drawBezierCurve函数

r

对于subdivide()的调用，导致subdivide()函数中的计算修改了调用 中所传的实际参数。

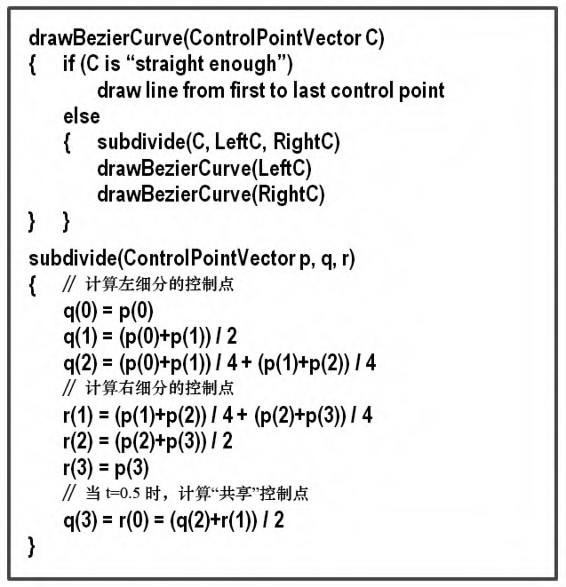


图11.7 贝塞尔曲线的递归细分算法

11.3 二次贝塞尔曲面

贝塞尔曲线定义了曲线（在2D或3D空间中），而贝塞尔曲面定义 了3D空间中的曲面。将我们在曲线中看到的概念扩展到曲面，需要将 参数方程组中的参数个数从一个扩展到两个。对于贝塞尔曲线，我们

将参数称为

t

。对于贝塞尔曲面，我们将参数称为

u

和 。曲线由点 ( )

v P t

组成，而曲面将由点 ( ,

P u v

)组成，如图11.8所示。

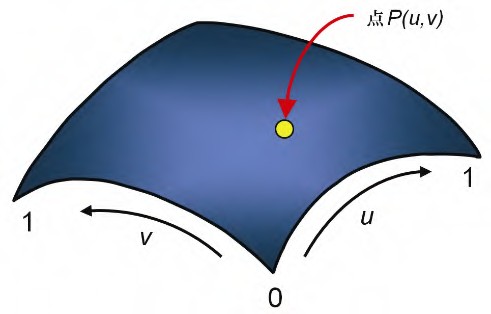


图11.8 参数曲面

对于二次贝塞尔曲面，每个轴 和

u v

上有3个控制点，总共9个控制

点。图11.9（见彩插）使用蓝色展示了一组共9个控制点（通常称为控 制点“网格”）的示例，以及相应的曲面（红色）。

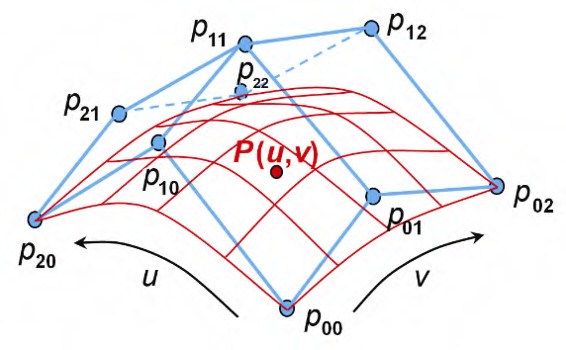


图11.9 二次贝塞尔控制网格和相应的表面

网格中的9个控制点标记为

pij

，其中

i

和 分别代表

j u

和 方向上的

v

索引。每组3个相邻控制点（例如（

，

，

p

00

p01

p02

））会定义一条贝塞

尔曲线。然后将表面上的点

P

( ， )定义为两个混合函数的和，一个在

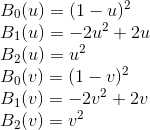
1. v

方向，一个在

u v

方向。则用于构建贝塞尔曲面的两个混合函数的形式

遵循先前为贝塞尔曲线给出的方法：



接下来生成构成贝塞尔曲面的点

P

( ,

u v

)。对于每个控制点 ，

pij

将其与第

i

个混合函数在

u

处的值相乘，再与第

j

个混合函数在

v

处的值

相乘。最后将所有控制点的结果求和，生成贝塞尔曲面上的点 ( ,

P u

)：

v



组成贝塞尔曲面的生成点集有时会称为补丁。术语“补丁”有时 会让人感到困惑，我们稍后在研究曲面细分着色器时会看到（对于实 际实现贝塞尔曲面非常有用）。因为通常控制点组成的网格才称为 “补丁”。

11.4 三次贝塞尔曲面

从二次曲面到三次曲面需要使用更大的网格——4×4而非3×3。 图11.10（见彩插）显示了16控制点网格（蓝色）和相应曲面（红色） 的示例。

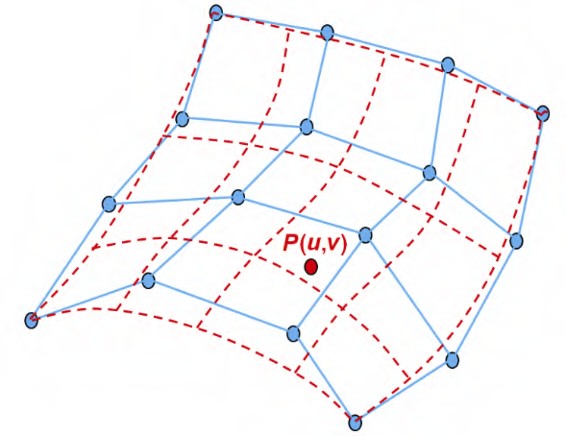


图11.10 三次贝塞尔控制网格和相应的曲面

同上，我们可以通过组合三次贝塞尔曲线的相关混合函数来推导

表面上的点

P

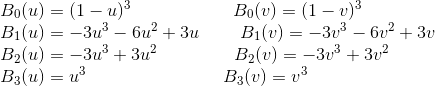
其中：

( ,

u v



)的公式：



渲染贝塞尔曲面也可以通过递归细分[AS14]完成，方法是交替地将 曲面沿每个维度分成两半，如图11.11所示。每个细分产生4个新的控 制点网格，每个网格包含16个点，这些点定义了曲面的一个象限。

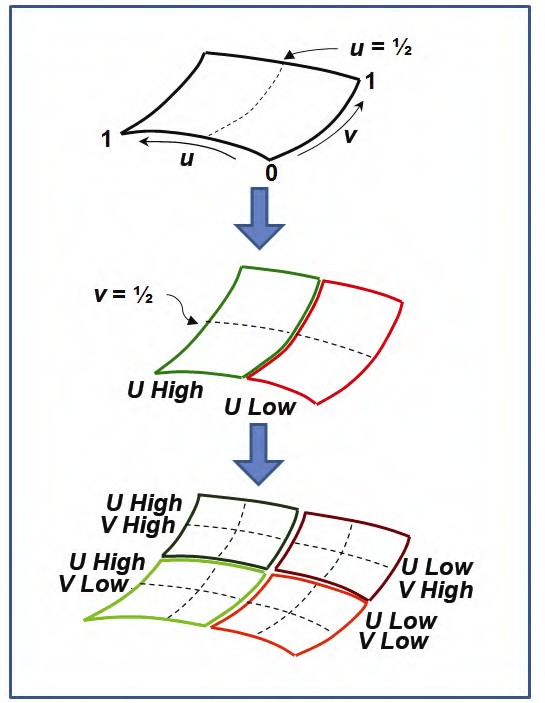


图11.11 贝塞尔曲面的递归细分

当渲染贝塞尔曲线时，我们在曲线“足够直”时停止细分。而对 于贝塞尔曲面，我们在曲面“足够平坦”时停止递归。一种实现方法 是，确保子象限控制网格上所有递归生成的点，距由该网格的4个角点

中的3个定义的平面的距离，都小于一个允许的范围。点( ,

x y

, )与平

z

面( ,

A B

, , )之间的距离 为：

C D d



如果 小于某个足够小的阈值，则我们停止细分过程，并简单地使

d

用子象限网格的4个角的控制点来绘制两个三角形。

对于贝塞尔曲线，OpenGL管线的细分阶段为基于图11.5中的迭代

算法渲染贝塞尔曲面提供了一种有吸引力的替代方法。其策略是让曲 面细分生成一个大的顶点网格，然后使用混合函数将这些顶点重新定 位到贝塞尔曲面上，由三次贝塞尔控制点指定。我们在第12章中实现 了这一点。

补充说明

本章重点介绍参数贝塞尔曲线和曲面的数学基础。我们推迟了在 OpenGL中呈现其中任何一个的实现，因为实现它们需要适当的曲面细 分着色器知识作为载体，我们将在下一章中进行介绍。我们还跳过了 一些推导过程，例如递归细分算法。

在3D图形中，使用贝塞尔曲线建模对象有许多优点。首先，理论 上，这些物体可以任意缩放，并且仍然保持光滑的表面而不“像素 化”。其次，许多由复杂曲线组成的物体可以使用贝塞尔控制点集合 进行更有效的存储，而不是存储数千个顶点。

除计算机图形和汽车外，贝塞尔曲线还有许多实际应用。在桥梁 设计中也可以找到它们的身影，例如耶路撒冷的Chords Bridge

[CB16]。类似的技术也用于构建TrueType字体，因此可以将其缩放到任

意大小，或者将视角任意拉近观看，而字体边缘始终保持平滑。

习题

11.1 二次贝塞尔曲线仅限于定义完全“凹”或“凸”的曲线。

描述（或绘制）一个曲线作为例子，该曲线既不以完全凹的形式，也 不以完全凸的方式弯曲，因此无法通过二次贝塞尔曲线进行近似描 述。

11.2 使用钢笔或铅笔在一张纸上绘制一组任意4个点，按任意顺 序编号为1～4，然后尝试大致绘制一条由这4个有序控制点定义的三次 贝塞尔曲线。接着重新排列控制点的编号（改变它们的顺序，但不改 变它们的位置）并重新绘制新产生的三次贝塞尔曲线。互联网上有许 多在线工具可以用于绘制贝塞尔曲线，你可以使用它们来检验你绘制 的曲线。

# 参考资料

[AS14] E. Angel and D. Shreiner,

Interactive Computer

, 7th ed. (Pearson,

Graphics: A Top-Down Approach with WebGL

2014).

[BE16] S. Bernstein, Wikipedia, accessed October 2018. [BE72] P.Bézier,

Numerical Control:Mathematics and

(JohnWiley& Sons, 1972).

Applications

[CB16] Chords Bridge, Wikipedia, accessed October 2018.

[DC63] P. de Casteljau, “Courbes et surfaces à ,”

technical report (A. Citro n, 1963).



[[1]](#_bookmark182) 当然，曲线可以存在于3D空间中。然而，二次曲线完全位于2D平 面内。

第12章 曲面细分

术语Tessellation（镶嵌）是指一大类设计活动，通常是指在平 坦的表面上，用各种几何形状的瓷砖相邻排列以形成图案。它的目的 可以是艺术性的或实用性的，很多例子可以追溯到几千年前[TS16]。

在3D图形学中，Tessellation指的是有点不同的东西（曲面细 分），但显然是由它的经典对应物（镶嵌）启发而成的。在这里，曲 面细分指的是生成并且操控大量三角形以渲染复杂的形状和表面，尤 其是使用硬件进行渲染。曲面细分是OpenGL核心近期才增加的新功 能，在2010年的4.0版本中出现。[[1]](#_bookmark202)

12.1 OpenGL中的曲面细分

OpenGL对硬件曲面细分的支持，通过3个管线阶段提供：

（1）曲面细分控制着色器；

（2）曲面细分器；

（3）曲面细分评估着色器。 第（1）和第（3）阶段是可编程的；而中间的第（2）阶段不是。

为了使用曲面细分，程序员通常会提供控制着色器和评估着色器。

曲面细分器（其全名是曲面细分图元生成器，或TPG）是硬件支持

的引擎，可以生成固定的三角形网格。[[2]](#_bookmark203)控制着色器允许我们配置曲 面细分器要构建什么样的三角形网格。然后，评估着色器允许我们以 各种方式操控网格。然后，被操控过的三角形网格，会作为通过管线 前进的顶点的源数据。回想一下图2.2，在管线上，曲面细分着色器位 于顶点着色器和几何着色器阶段之间。

让我们从一个简单的应用程序开始，该应用程序只使用曲面细分 器创建顶点的三角形网格，然后在不进行任何操作的情况下显示它。 为此，我们需要以下模块。

（1）C++/OpenGL应用程序：

创建一个摄像机和相关的MVP矩阵，视图（

v

）和投影（

p

）矩阵确

定摄像机朝向，模型（

m

）矩阵可用于修改网格的位置和方向。

（2）顶点着色器： 在这个例子中基本上什么都不做，顶点将在曲面细分器中生成。

（3）曲面细分控制着色器： 指定曲面细分器要构建的网格。

（4）曲面细分评估着色器： 将MVP矩阵应用于网格中的顶点。

（5）片段着色器：

只需为每个像素输出固定颜色。 程序12.1显示了整个应用程序的代码。即使像这样的简单示例也

相当复杂，因此许多代码元素都需要解释。请注意，这是我们第一次 使用除顶点和片段着色器之外的组件构建GLSL渲染程序。因此，我们 实现了createShaderProgram()的4参数重载版本。

程序12.1 基本曲面细分器网格

C++ / OpenGL应用程序

GLuint createShaderProgram(const char \*vp, const char \*tCS, const char \*tES, const char \*fp) {

string vertShaderStr = readShaderSource(vp); string tcShaderStr = readShaderSource(tCS); string teShaderStr = readShaderSource(tES); string fragShaderStr = readShaderSource(fp);

const char \*vertShaderSrc = vertShaderStr.c\_str(); const char \*tcShaderSrc = tcShaderStr.c\_str(); const char \*teShaderSrc = teShaderStr.c\_str(); const char \*fragShaderSrc = fragShaderStr.c\_str();

GLuint vShader = glCreateShader(GL\_VERTEX\_SHADER);

GLuint tcShader = glCreateShader(GL\_TESS\_CONTROL\_SHADER); GLuint teShader = glCreateShader(GL\_TESS\_EVALUATION\_SHADER); GLuint fShader = glCreateShader(GL\_FRAGMENT\_SHADER);

glShaderSource(vShader, 1, &vertShaderSrc, NULL); glShaderSource(tcShader, 1, &tcShaderSrc, NULL); glShaderSource(teShader, 1, &teShaderSrc, NULL); glShaderSource(fShader, 1, &fragShaderSrc, NULL);

glCompileShader(vShader); glCompileShader(tcShader); glCompileShader(teShader); glCompileShader(fShader);

GLuint vtfprogram = glCreateProgram(); glAttachShader(vtfprogram, vShader);

glAttachShader(vtfprogram, tcShader); glAttachShader(vtfprogram, teShader); glAttachShader(vtfprogram, fShader); glLinkProgram(vtfprogram);

return vtfprogram;

}

void init(GLFWwindow\* window) {

. . .

renderingProgram = createShaderProgram("vertShader.glsl", "tessCShader.glsl", "tessEShader.glsl", "fragShader.glsl");

}

void display(GLFWwindow\* window, double currentTime) {

. . .

glUseProgram(renderingProgram);

. . .

glPatchParameteri(GL\_PATCH\_VERTICES, 1); glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE); glDrawArrays(GL\_PATCHES, 0, 1);

}

顶点着色器

#version 430

uniform mat4 mvp\_matrix; void main(void) { }

曲面细分控制着色器

#version 430

uniform mat4 mvp\_matrix; layout (vertices = 1) out;

void main(void)

{ gl\_TessLevelOuter[0] = 6; gl\_TessLevelOuter[1] = 6;

gl\_TessLevelOuter[2] = 6;

gl\_TessLevelOuter[3] = 6;

gl\_TessLevelInner[0] = 12;

gl\_TessLevelInner[1] = 12;

}

曲面细分评估着色器

#version 430

uniform mat4 mvp\_matrix; layout (quads, equal\_spacing, ccw) in;

void main (void)

{ float u = gl\_TessCoord.x; float v = gl\_TessCoord.y;

gl\_Position = mvp\_matrix \* vec4(u,0,v,1);

}

片段着色器

#version 430

out vec4 color;

uniform mat4 mvp\_matrix;

void main(void)

{ color = vec4(1.0, 1.0, 0.0, 1.0); // 黄色

}

得到的输出网格如图12.1所示（见彩插）。



图12.1 Tessellator三角形网格输出

曲面细分器生成由两个参数定义的顶点网格：内层级别和外层级 别。在这种情况下，内层级别为12，外层级别为6——网格的外边缘被 分为6段，而跨越内部的线被分为12段。

程序12.1中的特别相关的新结构被高亮显示。让我们首先讨论第 一部分——C++/ OpenGL代码。

编译这两个新着色器，跟顶点和片段着色器完全相同。然后将它 们附加到同一个渲染程序，并且链接调用保持不变。唯一的新项目是 用于指定要实例化的着色器类型的常量——新常量如下：

GL\_TESS\_CONTROL\_SHADER GL\_TESS\_EVALUATION\_SHADER

请注意display()函数中的新项目。glDrawArrays()调用现在指定 GL\_PATCHES。当使用曲面细分时，从C++/OpenGL应用程序发送到管线

（即在VBO中）的顶点不会被渲染，但通常会被当作控制点，就像我们 在贝塞尔曲线中看到的那些一样。一组控制点被称作“补丁”，并且 在使用曲面细分的代码段中，GL\_PATCHES是唯一允许的图元类型。 “补丁”中顶点的数量在glPatchParameteri()的调用中指定。在这个 特定示例中，没有任何控制点被发送，但我们仍然需要指定至少一 个。类似地，在glDrawArrays()调用中，我们指示起始值为0，顶点数 量为1，即使我们实际上没有从C++程序发送任何顶点。

对glPolygonMode()的调用指定了如何光栅化网格。默认值为 GL\_FILL。而我们的代码中显示的是GL\_LINE，如我们在图12.1中看到 的那样，它只会导致连接线被光栅化（因此我们可以看到由曲面细分 器生成的网格本身）。如果我们将该行代码更改为GL\_FILL（或将其注 释掉，从而使用默认行为GL\_FILL），我们将得到如图12.2所示的版 本。



图12.2 使用GL\_FILL渲染的细分网格

现在让我们来过一遍4个着色器。如前所述，顶点着色器几乎没什 么可做的，因为C++/OpenGL应用程序没有提供任何顶点。它包含的是 一个统一变量声明，以和其他着色器相匹配，以及一个空的main()。 在任何情况下，所有着色器程序都必须包含顶点着色器。

曲面细分控制着色器指定曲面细分器要生成的三角形网格的拓扑 结构。通过将值分配给名为gl\_TessLevelxxx的保留字，设置6个“级 别”参数——两个“内部”和4个“外部”级别。我们这里细分了一个 由三角形组成的大矩形网格，称为四边形。[[3]](#_bookmark204)级别参数告诉曲面细分 器在形成三角形时如何细分网格，它们的排列如图12.3所示。

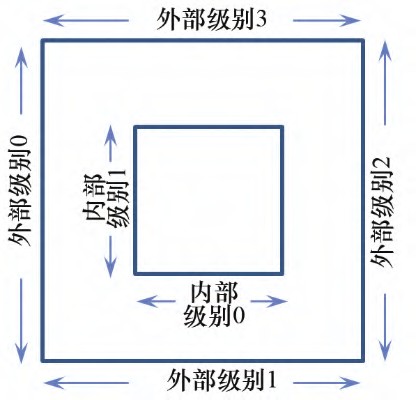


图12.3 细分级别

请注意控制着色器中的代码行：

layout (vertices=1) out;

这与之前的GL\_PATCHES讨论有关，用来指定从顶点着色器传递给 控制着色器（以及“输出”给评估着色器）的每个“补丁”的顶点

数。在我们现在这个程序中没有任何顶点，但我们仍然必须指定至少

一个，因为它也会影响控制着色器被执行的次数。稍后这个值将反映 控制点的数量，并且必须与C++/OpenGL应用程序中 glPatchParameteri()调用中的值匹配。

接下来让我们看一下曲面细分评估着色器。它以一行代码开头， 形如：

layout (quads, equal\_spacing, ccw) in;

乍一看这好像与控件着色器中的“out”布局语句有关，但实际上 它们是无关的。相反，这行代码是我们指示曲面细分器去生成排列在 一个大矩形（“四边形”）中顶点的位置。它还指定了细分线段（包 括内部和外部）具有相等的长度（稍后我们将看到长度不等的细分的 应用场景）。“ccw”参数指定生成曲面细分网格顶点的缠绕顺序（在 当前情况下，是逆时针）。

然后，由曲面细分器生成的顶点被发送到评估着色器。因此，评 估着色器既可以从控制着色器（通常作为控制点），又可以从曲面细 分器（曲面细分网格）接收顶点。在程序12.1中，仅从曲面细分器接 收顶点。

评估着色器对曲面细分器生成的每个顶点执行一次。可以使用内 置变量gl\_TessCoord访问顶点位置。曲面细分网格的朝向使得它位于

- 平面中，因此gl\_TessCoord的

X Z X

和 分量被应用于网格的 和

Y X Z

坐标。

网格坐标，以及gl\_TessCoord的值，范围为0.0～1.0（这在计算纹理

坐标时会很方便）。然后，评估着色器使用MVP矩阵定向每个顶点（这

在前面章节的示例中，是由顶点着色器完成的）。

最后，片段着色器只为每个像素输出一个恒定的黄色。当然，我 们也可以使用它来为我们的场景应用纹理或光照，就像我们在前面的 章节中看到的那样。

12.2 贝塞尔曲面细分

现在让我们扩展我们的程序，使它将我们简单的矩形网格转换为 贝塞尔曲面。细分网格应该为我们提供了足够的顶点来对曲面进行采 样（如果我们想要更多的话，我们可以增加内部/外部细分级别）。我 们现在需要的是通过管线发送控制点，然后使用这些控制点执行计算 以将细分网格转换为我们所需的贝塞尔曲面。

假设我们希望建立一个立方体贝塞尔曲面，我们将需要16个控制 点。我们可以通过VBO从C++端发送它们，或者我们可以在顶点着色器 中硬编码写死它们。图12.4概述了来自C++端的控制点的过程。

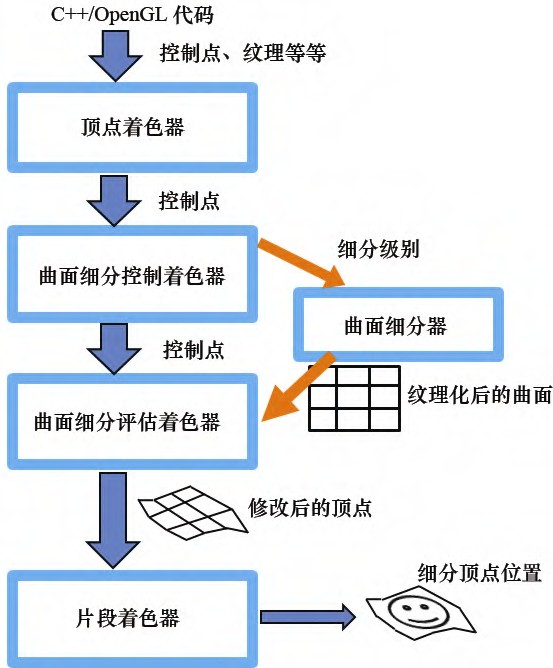


图12.4 贝塞尔曲面的曲面细分概述

现在是更准确地解释曲面细分控制着色器（TCS）如何工作的好时 机。与顶点着色器类似，TCS对每个传入顶点执行一次。另外，回想一 下第2章，OpenGL提供了一个名为gl\_VertexID的内置变量，它保存一 个计数器，指示顶点着色器当前正在执行哪次调用。曲面细分控制着 色器中存在一个类似的内置变量gl\_InvocationID。

曲面细分的一个强大功能是TCS（以及TES）着色器可以同时访问 数组中的所有控制点顶点。首先，当每个调用都可以访问所有顶点 时，TCS对每个顶点执行一次可能会让人感到困惑。在每个TCS调用 中，冗余地在赋值语句中指定曲面细分级别也是违反直觉的。尽管所 有这些看起来都很奇怪，但这样做是因为曲面细分的架构设计使得TCS 调用可以并行运行。

OpenGL提供了几个用于TCS和TES着色器的内置变量。我们已经提

到过的是gl\_InvocationID，当然还有gl\_TessLevelInner和 gl\_TessLevelOuter。以下是一些最有用的内置变量的更多细节和描 述。

曲面细分控制着色器（TCS）内置变量。

gl\_in[ ]——包含每个传入的控制点顶点的数组——每个传入顶 点是一个数组元素。可以使用“.”表示法将特定顶点属性作为字 段进行访问。一个内置属性是gl\_Position——因此，输入顶点 “i”的位置可以通过gl\_in[i].gl\_Position访问。

gl\_out[ ]——用于将输出控制点的顶点发送到TES的一个数组

——每个输出顶点是一个数组元素。可以使用“.”表示法将特定 顶点属性作为字段进行访问。一个内置属性是gl\_Position——因 此，输出顶点“i”的位置可以通过gl\_out[i].gl\_Position访 问。 gl\_InvocationID——整型ID计数器，指示TCS当前正在执行哪个 调用。一个常见的用途是用于传递顶点属性；例如，将当前调用 的顶点位置从TCS传递到TES可以用如下方式完成： gl\_out[gl\_InvocationID].gl\_Position = gl\_in[gl\_InvocationID].gl\_Position。

曲面细分评估着色器（TES）内置变量。

gl\_in[ ]——包含每个传入的控制点顶点的数组——每个传入顶 点是一个数组元素。可以使用“.”表示法将特定顶点属性作为字 段进行访问。一个内置属性是gl\_Position——因此，输入顶点 “i”的位置可以通过gl\_in[i].gl\_Position访问。

gl\_Position——曲面细分网格顶点的输出位置，可能在TES中被

修改。重要的是要注意gl\_Position和gl\_in[xxx].gl\_Position是 不同的——gl\_Position是起源于曲面细分器的输出顶点的位置， 而gl\_in[xxx].gl\_Position是一个从TCS进入TES的控制点顶点位 置。

值得注意的是，TCS中的输入和输出控制点顶点属性是数组。不同 的是，TES中的输入控制点顶点和顶点属性是数组，但输出顶点是标 量。此外，很容易混淆哪些顶点来自于控制点，哪些顶点是细分建立 的，然后移动以形成结果曲面。总而言之，TCS的所有顶点输入和输出 都是控制点，而在TES中，gl\_in[ ]保存输入控制点，gl\_TessCoord保 存输入的细分网格点，gl\_Position保存用于渲染的输出表面顶点。

我们的曲面细分控制着色器现在有两个任务：指定曲面细分级别 并将控制点从顶点着色器传递到评估着色器。然后，评估着色器可以 根据贝塞尔控制点修改网格点（gl\_TessCoords）的位置。

程序12.2显示了所有4个着色器——顶点、TCS、TES和片段——用 于指定控制点补丁，生成平坦的曲面细分顶点网格，在控制点指定的 曲面上重新定位这些顶点，并使用纹理图像绘制生成的曲面。它还显 示了C++/OpenGL应用程序的相关部分，特别是在display()函数中。在 此示例中，控制点源自顶点着色器（它们在那里硬编码写死），而不 是从C++/OpenGL应用程序进入OpenGL管线。代码后面会讲述其他详细 信息。

程序12.2 贝塞尔曲面的曲面细分

顶点着色器

#version 430

out vec2 texCoord; uniform mat4 mvp\_matrix;

layout (binding = 0) uniform sampler2D tex\_color;

void main(void)

{ // 这次由顶点着色器指定和发送控制点

const vec4 vertices[ ] =

vec4[ ] (vec4(-1.0, 0.5, -1.0, 1.0), vec4(-0.5, 0.5, -1.0, 1.0),

vec4( 0.5, 0.5, -1.0, 1.0), vec4( 1.0, 0.5, -1.0, 1.0),

vec4(-1.0, 0.0, -0.5, 1.0), vec4(-0.5, 0.0, -0.5, 1.0),

vec4( 0.5, 0.0, -0.5, 1.0), vec4( 1.0, 0.0, -0.5, 1.0),

vec4(-1.0, 0.0, 0.5, 1.0), vec4(-0.5, 0.0, 0.5, 1.0),

vec4( 0.5, 0.0, 0.5, 1.0), vec4( 1.0, 0.0, 0.5, 1.0),

vec4(-1.0, -0.5, 1.0, 1.0), vec4(-0.5, 0.3, 1.0, 1.0),

vec4( 0.5, 0.3, 1.0, 1.0), vec4( 1.0, 0.3, 1.0, 1.0) );

// 为当前顶点计算合适的纹理坐标，从[-1...+1]转换到[0...1] texCoord = vec2((vertices[gl\_VertexID].x + 1.0) / 2.0,

(vertices[gl\_VertexID].z + 1.0) / 2.0); gl\_Position = vertices[gl\_VertexID];

}

曲面细分控制着色器

#version 430 in vec2 texCoord[ ];

out vec2 texCoord\_TCSout[ ]; // 以标量形式从顶点着色器传来的纹理坐标输 出，以数组形式被接收，然后被发送给评估着色器

uniform mat4 mvp\_matrix;

layout (binding = 0) uniform sampler2D tex\_color; layout (vertices = 16) out; // 每个补丁有16个控制点

void main(void)

{ int TL = 32; // 曲面细分级别都被设置为这个值

if (gl\_InvocationID == 0)

{ gl\_TessLevelOuter[0] = TL; gl\_TessLevelOuter[2] = TL; gl\_TessLevelOuter[1] = TL; gl\_TessLevelOuter[3] = TL; gl\_TessLevelInner[0] = TL; gl\_TessLevelInner[1] = TL;

}

// 将纹理和控制点传递给TES

texCoord\_TCSout[gl\_InvocationID] = texCoord[gl\_InvocationID]; gl\_out[gl\_InvocationID].gl\_Position =

gl\_in[gl\_InvocationID].gl\_Position;

}

曲面细分评估着色器

#version 430

layout (quads, equal\_spacing,ccw) in; uniform mat4 mvp\_matrix;

layout (binding = 0) uniform sampler2D tex\_color; in vec2 texCoord\_TCSout[ ];

out vec2 texCoord\_TESout; // 以标量形式传来的纹理坐标数组被一 个个传出

void main (void)

{ vec3 p00 = (gl\_in[0].gl\_Position).xyz; vec3 p10 = (gl\_in[1].gl\_Position).xyz; vec3 p20 = (gl\_in[2].gl\_Position).xyz; vec3 p30 = (gl\_in[3].gl\_Position).xyz; vec3 p01 = (gl\_in[4].gl\_Position).xyz; vec3 p11 = (gl\_in[5].gl\_Position).xyz; vec3 p21 = (gl\_in[6].gl\_Position).xyz; vec3 p31 = (gl\_in[7].gl\_Position).xyz; vec3 p02 = (gl\_in[8].gl\_Position).xyz; vec3 p12 = (gl\_in[9].gl\_Position).xyz; vec3 p22 = (gl\_in[10].gl\_Position).xyz; vec3 p32 = (gl\_in[11].gl\_Position).xyz; vec3 p03 = (gl\_in[12].gl\_Position).xyz; vec3 p13 = (gl\_in[13].gl\_Position).xyz; vec3 p23 = (gl\_in[14].gl\_Position).xyz; vec3 p33 = (gl\_in[15].gl\_Position).xyz;

float u = gl\_TessCoord.x; float v = gl\_TessCoord.y;

// 立方贝塞尔基础函数

float bu0 = (1.0-u) \* (1.0-u) \* (1.0-u); // (1-u)^3

float bu1 = 3.0 \* u \* (1.0-u) \* (1.0-u); // 3u(1-u)^2 float bu2 = 3.0 \* u \* u \* (1.0-u); // 3u^2(1-u) float bu3 = u \* u \* u; // u^3

float bv0 = (1.0-v) \* (1.0-v) \* (1.0-v); // (1-v)^3

float bv1 = 3.0 \* v \* (1.0-v) \* (1.0-v); // 3v(1-v)^2 float bv2 = 3.0 \* v \* v \* (1.0-v); // 3v^2(1-v) float bv3 = v \* v \* v; // v^3

// 输出曲面细分补丁中的顶点位置

vec3 outputPosition = bu0 \* ( bv0\*p00 + bv1\*p01 + bv2\*p02 + bv3\*p03 )

+ bu1 \* ( bv0\*p10 + bv1\*p11 + bv2\*p12 + bv3\*p13 )

+ bu2 \* ( bv0\*p20 + bv1\*p21 + bv2\*p22 + bv3\*p23 )

+ bu3 \* ( bv0\*p30 + bv1\*p31 + bv2\*p32 + bv3\*p33 ); gl\_Position = mvp\_matrix \* vec4(outputPosition,1.0f);

// 输出插值过的纹理坐标

vec2 tc1 = mix(texCoord\_TCSout[0], texCoord\_TCSout[3], gl\_TessCoord.x);

vec2 tc2 = mix(texCoord\_TCSout[12], texCoord\_TCSout[15], gl\_TessCoord.x);

vec2 tc = mix(tc2, tc1, gl\_TessCoord.y); texCoord\_TESout = tc;

}

片段着色器

#version 430 in vec2 texCoord\_TESout; out vec4 color;

uniform mat4 mvp\_matrix;

layout (binding = 0) uniform sampler2D tex\_color;

void main(void)

{ color = texture(tex\_color, texCoord\_TESout);

}

C++/OpenGL 应用程序

// 这次我们也传入一个纹理以用来绘制表面

// 像往常一样在init()里加载纹理，并在display()里启用

void display(GLFWwindow\* window, double currentTime) {

. . .

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, textureID);

glFrontFace(GL\_CCW);

glPatchParameteri(GL\_PATCH\_VERTICES, 16); // 每个补丁的顶 点数量 = 16

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_FILL); glDrawArrays(GL\_PATCHES, 0, 16); // 补丁顶点的总

数量：16 x 1个补丁= 16

}

顶点着色器现在指定代表特定贝塞尔曲面的16个控制点（“补

丁”顶点）。在这个例子中，它们都被归一化到范围[−1…+1]。顶点 着色器还使用控制点来确定适合细分网格的纹理坐标，其值在[0…1]

范围内。很重要的是，要重申顶点着色器输出的顶点不是将要用来光

栅化的顶点，而是贝塞尔控制点。使用曲面细分时，补丁顶点永远不

会被光栅化——只有曲面细分顶点会被光栅化。

控制着色器仍然会指定内部和外部曲面细分级别。它现在还负责 将控制点和纹理坐标发送到评估着色器。请注意，曲面细分级别只需 要指定一次，因此该步骤仅在第0次调用期间完成（回想一下TCS每个 顶点运行一次，因此在此示例中有16次调用）。为方便起见，我们为 每个细分级别指定了32个细分。

接下来，评估着色器执行所有贝塞尔曲面计算。main()开头的大 块赋值语句从每个传入gl\_in的gl\_Position中提取控制点（请注意， 这些控制点对应于控制着色器的gl\_out变量）。然后使用来自曲面细 分器的网格点计算混合函数的权重，从而生成一个新的 outputPosition，然后应用模型-视图-投影矩阵，为每个网格点生成 输出gl\_Position并形成贝塞尔曲面。

另外，还需要创建纹理坐标。顶点着色器仅为每个控制点位置提 供一个纹理坐标。但我们并不是要渲染控制点，我们最终需要更多的 曲面细分网格点的纹理坐标。有很多方法可以做到这一点，在这里我 们利用GLSL方便的混合功能对它们进行线性插值。mix()函数需要3个 参数：（a）起始点；（b）结束点；（c）内插值，范围为0～1。它返 回与内插值对应的起点和终点之间的值。由于细分网格坐标的范围也 是0～1，所以它们可以直接用于此目的。

这次在片段着色器中，不再是输出单一颜色，而是应用标准纹

理。属性texCoord\_TESout中的纹理坐标是在评估着色器中生成的纹理 坐标。对C++程序的更改同样很简单——请注意，现在指定的补丁大小 为16。结果输出如图12.5所示（应用了[LU16]的平铺纹理）。



图12.5 曲面细分过的贝塞尔曲面

12.3 地形、高度图的细分

回想一下，在顶点着色器中执行高度贴图可能会遇到顶点数量不 足以用来渲染所需的细节的情况。现在我们有了生成大量顶点的方 法，让我们回到Hastings-Trew的月球表面纹理贴图[HT16]并将其用作 高度贴图，提升曲面细分顶点来生成月球表面细节。正如我们将看到 的，这具有一些优点，可以让顶点的几何形状更好地匹配月亮图像， 并且提升轮廓（边缘）细节。

我们的策略是修改程序12.1，在

X

- 平面中放置细分网格，并使用

Z

高度贴图来设置每个细分网格点的

Y

坐标。要做到这一点，我们不需要

补丁，因为可以硬编码细分网格的位置，因此我们将在

glDrawArrays()和glPatchParameteri()中为每个补丁指定所需的最少

的1个顶点，如程序12.1中所做的那样。Hastings-Trew的月亮纹理图 像既用于颜色，也用作高度图。

我们通过将曲面细分网格的gl\_TessCoord值映射到顶点和纹理的 适当范围，在评估着色器中生成顶点和纹理坐标。[[4]](#_bookmark205)评估着色器也通

过添加月亮纹理的一小部分颜色分量到输出顶点的

Y

分量，来实现高度

贴图。着色器的更改显示在程序12.3中。

程序12.3 简单的地形曲面细分

顶点着色器

#version 430 uniform mat4 mvp\_matrix;

layout (binding = 0) uniform sampler2D tex\_color; void main(void) { }

曲面细分控制着色器

. . .

layout (vertices = 1) out; // 这个应用程序中不需要控制点

void main(void)

{ int TL=32;

if (gl\_InvocationID == 0)

{ gl\_TessLevelOuter[0] = TL; gl\_TessLevelOuter[2] = TL; gl\_TessLevelOuter[1] = TL; gl\_TessLevelOuter[3] = TL; gl\_TessLevelInner[0] = TL; gl\_TessLevelInner[1] = TL;

}

}

曲面细分评估着色器

. . .

out vec2 tes\_out; uniform mat4 mvp\_matrix;

layout (binding = 0) uniform sampler2D tex\_color;

void main (void)

{ // 将曲面细分网格顶点从[0...1]映射到想要的顶点[-0.5...+0.5]

vec4 tessellatedPoint = vec4(gl\_TessCoord.x - 0.5, 0.0,

gl\_TessCoord.y - 0.5, 1.0);

// 垂直“翻转” Y值，以将曲面细分网格顶点映射到纹理坐标

// 左上顶点坐标是(0,0)，左下纹理坐标是(0,0)

vec2 tc = vec2(gl\_TessCoord.x, 1.0 - gl\_TessCoord.y);

// 图像是灰度图，所以任何一个颜色分量（R、G或B）都可以作为高度偏移量

tessellatedPoint.y += (texture(tex\_color, tc).r) / 40.0; //

将颜色值等比例缩小应用于Y值

// 将高度贴图提升的点转换到视觉空间

gl\_Position = mvp\_matrix \* tessellatedPoint; tes\_out = tc;

}

片段着色器

. . .

in vec2 tes\_out; out vec4 color;

layout (binding = 0) uniform sampler2D tex\_color;

void main(void)

{ color = texture(tex\_color, tes\_out);

}

这里的片段着色器类似于程序12.2的，只是根据纹理图像输出颜 色。C++/OpenGL应用程序基本上没有变化——它加载纹理（用作纹理 和高度图）并为其启用采样器。图12.6显示了纹理图像（左侧）和第 一次尝试的最终输出，遗憾的是，它还没有实现正确的高度贴图。

第一次结果存在严重缺陷。虽然我们现在可以看到远处地平线上 的轮廓细节，但是那里的凸起与纹理贴图中的实际细节不对应。回想 一下，在高度图中，白色应该表示“高”，而黑色应该表示“低”。 特别是图像右上方的区域显示的大山丘与其中的浅色和深色无关。

导致此问题的原因是细分网格的分辨率。曲面细分器可以生成的 最大顶点数取决于硬件。要符合OpenGL标准，唯一的要求是每个曲面

细分级别的最大值至少为64。我们的程序指定了一个内部和外部曲面

细分级别均为32的单一细分网格，因此我们生成了大约32×32或者说 刚刚超过1 000个顶点，这不足以准确反映图像中的细节。这在图12.6 右上方（图中放大）尤其明显——边缘细节仅在沿地平线的32个点处 采样，这会产生巨大而看起来很随机的山丘。即使我们将曲面细分值 增加到64，总共64×64或刚刚超过4 000个顶点仍然不足以满足使用月 球图像进行高度贴图的需要。

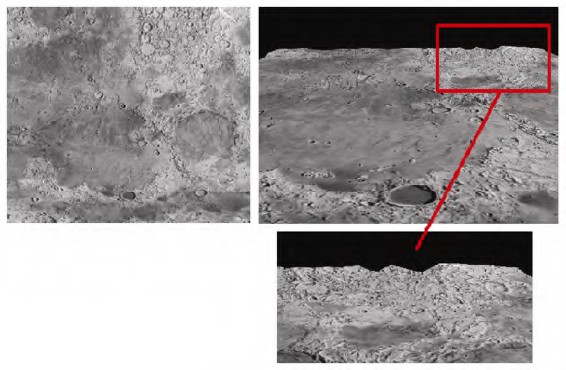


图12.6 细分地形——首次尝试失败，顶点数量不足

增加顶点数量的一个好方法是使用我们在第4章中看到的实例化。 我们的策略是让曲面细分器生成网格，并使用实例化重复数次。在顶 点着色器中，我们构建了一个由4个顶点定义的补丁，每个顶点用于细 分网格的每个角。在我们的C++/OpenGL应用程序中，我们将

glDrawArrays()调用更改为glDrawArraysInstanced()。如此，我们指

定一个64×64个补丁的网格，每个补丁包含一个细分级别为32的网 格。这将带给我们总共64×64×32×32个，或者说超过400万个顶点。

顶点着色器首先指定4个纹理坐标(0,0)、(0,1)、(1,0)和(1,1)。 使用实例化时，请回想一下，顶点着色器可以访问整数变量 gl\_InstanceID，它包含一个对应于当前正在处理的 glDrawArraysInstanced()调用的计数器。我们使用此ID值来分配大网 格中各个补丁的位置。补丁位于行和列中，第一个补丁位于(0,0)，第 二个位于(1,0)，下一个位于(2,0)，依此类推，第一列中的最后一个 补丁在(63,0)。下一列的补丁位于(0,1)、(1,1)，依此类推，直至 (63,1)。最后一列的补丁位于(0,63)、(1,63)，依此类推，最后是

(63,63)。给定补丁的

X

坐标是实例ID整除64，

Y

坐标是实例ID除以

64（整数除法）。然后着色器将坐标向下缩放到范围[0...1]。 控制着色器没有更改，除了它将顶点和纹理坐标传递下去。

接下来，评估着色器获取传入的细分网格顶点（由gl\_TessCoord 指定）并将它们移动到传入补丁指定的坐标范围内。它对纹理坐标也 进行一样的处理，并且也会以与程序12.3中相同的方式应用高度贴 图。片段着色器没有修改。

每个组件的更改显示在程序12.4中。结果如图12.7所示。请注 意，高点和低点现在更接近于图像的亮部和暗部。

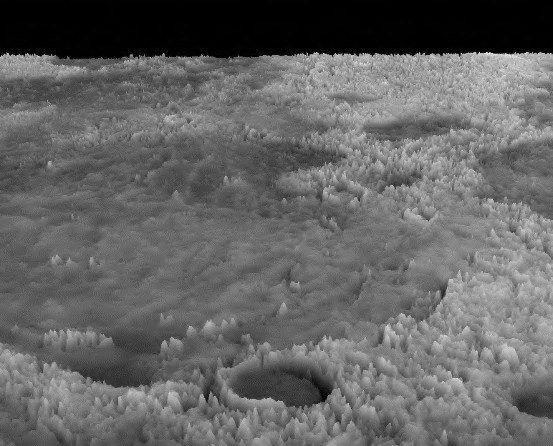


图12.7 细分地形——第二次尝试，使用实例化

程序12.4 实例化细分地形

C++/OpenGL应用程序

// 和贝塞尔曲面例子相同，并做如下修改

glPatchParameteri(GL\_PATCH\_VERTICES, 4);

glDrawArraysInstanced(GL\_PATCHES, 0, 4, 64\*64);

顶点着色器

. . .

out vec2 tc;

void main(void)

{ vec2 patchTexCoords[ ] = vec2[ ] (vec2(0,0), vec2(1,0), vec2(0,1), vec2(1,1));

// 基于当前是哪个实例计算出坐标偏移量

int x = gl\_InstanceID % 64; int y = gl\_InstanceID / 64;

// 纹理坐标被分配进64个补丁中，并归一化到[0..1]。翻转Y轴坐标

tc = vec2( (x+patchTexCoords[gl\_VertexID].x) / 64.0, (63 - y+patchTexCoords[gl\_VertexID].y) / 64.0);

// 顶点位置和纹理坐标相同，只是它的取值范围从-0.5到+0.5 gl\_Position = vec4(tc.x - 0.5, 0.0, (1.0 - tc.y) - 0.5, 1.0);

// 并且将Y轴坐标翻转回来

}

曲面细分控制着色器

. . .

layout (vertices = 4) out; in vec2 tc[ ]; out vec2 tcs\_out[ ];

void main(void)

{ // 曲面细分级别的指定和之前例子中相同

. . .

tcs\_out[gl\_InvocationID] = tc[gl\_InvocationID]; gl\_out[gl\_InvocationID].gl\_Position =

gl\_in[gl\_InvocationID].gl\_Position;

}

曲面细分评估着色器

. . .

in vec2 tcs\_out[ ]; out vec2 tes\_out; void main (void)

{ // 将纹理坐标映射到传入的控制点指定的子网格上

vec2 tc = vec2(tcs\_out[0].x + (gl\_TessCoord.x) / 64.0, tcs\_out[0].y + (1.0 - gl\_TessCoord.y) / 64.0);

// 将细分网格映射到传入的控制点指定的子网格上

vec4 tessellatedPoint = vec4(gl\_in[0].gl\_Position.x + gl\_TessCoord.x / 64.0, 0.0,

gl\_in[0].gl\_Position.z +

gl\_TessCoord.y / 64.0, 1.0);

// 将高度图的高度增加给顶点

tessellatedPoint.y += (texture(tex\_height, tc).r) / 40.0; gl\_Position = mvp\_matrix \* tessellatedPoint;

tes\_out = tc;

}

现在我们已经实现了高度贴图，我们可以着手改进它并整合光 照。一个挑战是我们的顶点还没有与它们相关的法向量。另一个挑战 是简单地使用纹理图像作为高度图产生了过度“锯齿状”的结果—— 在这种情况下是因为并非纹理图像中的所有灰度变化都是由高度引起 的。对于这个特定的纹理贴图，Hastings-Trew已经生成了一个改进的 高度贴图，我们可以使用[HT16]。如图12.8左图所示。

我们可以通过生成相邻顶点（或高度图中的相邻纹素）的高度，

构建连接它们的向量以及使用叉积来计算法向量，以动态计算和创建 法向量。这需要一些细微的调整，具体取决于场景的精度（和/或高度 图图像）。在这里，我们使用GIMP“normalmap”插件[GP16]来根据 Hastings-Trew的高度图生成法线贴图，如图12.8右图所示。

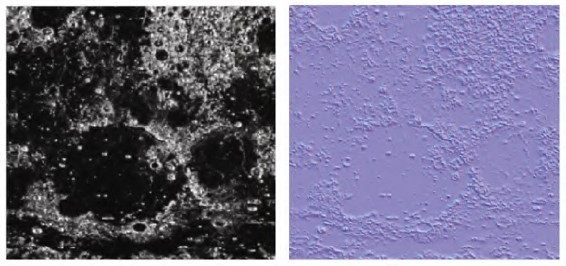


图12.8 月球表面：高度图[HT16]（左）和法线贴图（右）

我们对代码进行的大部分更改现在只是为了实现Phong着色的标准 方法。

C++/OpenGL应用程序。 我们加载并激活一个额外的纹理来保存法线贴图，还添加了代码

来指定光照和材质，就像我们在以前的应用程序中所做的那样。 顶点着色器。

唯一的增补是光照统一变量的声明和法线贴图的采样器。通常在 顶点着色器中完成的光照代码被移动到曲面细分评估着色器，因为直

到曲面细分阶段才生成顶点。

曲面细分控制着色器。 唯一的增补是光照统一变量的声明和法线贴图的采样器。 曲面细分评估着色器。 Phong光照的准备代码现在放在评估着色器中：

varyingVertPos = (mv\_matrix \* position).xyz; varyingLightDir = light.position - varyingVertPos;

片段着色器。

这里完成了用于计算Phong（或Blinn-Phong）照明的典型代码 段，以及从法线贴图中提取法向量的代码。然后将光照结果与纹理图 像用加权求和的方式结合起来。

带有高度和法线贴图以及Phong照明的最终结果如图12.9所示。地 形现在会响应光照。在此示例中，位置光已放置在左侧图像中心的左 侧，右侧图像中心的右侧。



图12.9 具有法线贴图和光照的曲面细分地形（光源分别位于左侧和右侧）

尽管从静止图像很难判断出对光的移动的响应，但是读者应该能 够辨别出漫射光的变化，并且山峰的镜面高光在两个图像中是非常不 同的。当摄像机或光源移动时，这当然会更明显。结果仍然不完美， 因为无论什么样的光照，输出中包含的原始纹理都包括了将出现在渲 染结果上的阴影。

12.4 控制细节级别（LOD）

在程序12.4中，使用实例化来实时生成数百万个顶点，即使是装 备精良的现代计算机也可能会感受到负担。幸运的是，将地形划分为 单独的补丁的策略，正如我们为增加生成的网格顶点的数量所做的那 样，也为我们提供了一种减少负担的好机制。

在生成的数百万个顶点中，许多顶点不是必需的。靠近摄像机的 补丁中的顶点非常重要，因为我们希望能够识别附近物体的细节。但

是，补丁越远离摄像机，甚至光栅化过程中有足够的像素来体现我们

生成的顶点数量的可能性就越小！

根据距摄像机的距离更改补丁中的顶点数量是一种称为细节级别 或LOD的技术。Sellers等人描述了一种通过修改控制着色器来控制实 例化曲面细分中的LOD的方法[SW15]。程序12.5显示了Sellers等人的方 法的简化版本。策略是使用补丁的感知大小来确定其曲面细分级别的 值。由于补丁的细分网格最终将放置在由进入控制着色器的4个控制点 定义的方格内，我们可以使用控制点相对于摄像机的位置来确定应该 为补丁生成多少个顶点。其步骤如下。

（1）通过将MVP矩阵应用于4个控制点，计算它们的屏幕位置。

（2）计算由控制点（在屏幕上的空间中）定义的正方形边长（即 宽度和高度）。请注意，即使4个控制点形成正方形，这些边长也可能 不同，因为应用了透视矩阵。

（3）根据曲面细分级别所需的精度（基于高度图中的细节数 量），将长度的值按可调整常数进行缩放。

（4）将缩放长度值加1，以避免将曲面细分级别指定为0（这将导 致不生成顶点）。

（5）将曲面细分级别设置为相应的计算宽度和高度值。 回想一下，在我们的实例中，我们不是只创建一个网格，而是创

建64×64个网格。因此，对每个补丁执行以上列表中的5个步骤，细节

级别因补丁而异。

所有更改都在控制着色器中，并显示在程序12.5中，生成的输出 如图12.10所示。请注意，变量gl\_InvocationID指的是正在处理补丁 中的哪个顶点（而不是正在处理哪个补丁）。因此，告诉曲面细分器 在每个补丁中生成多少个顶点的LOD计算发生在每个补丁的第0个顶点 期间。

程序12.5 曲面细分细节级别（LOD）

曲面细分控制着色器

. . .

void main(void)

{ float subdivisions = 16.0; // 基于高度图中细节密度的可调整的常 量

if (gl\_InvocationID == 0)

{ vec4 p0 = mvp \* gl\_in[0].gl\_Position; // 屏幕空间中控制点的 位置

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| vec4 vec4 | | p1 p2 | =  = | mvp \* gl\_in[1].gl\_Position; mvp \* gl\_in[2].gl\_Position; | |
| p0 | = | p0 | / | p0.w; |  |
| p1 | = | p1 | / | p1.w; |
| p2 | = | p2 | / | p2.w; |

float width = length(p2.xy - p0.xy) \* subdivisions + 1.0;

// 曲面细分网格的感知"宽度"

float height = length(p1.xy - p0.xy) \* subdivisions + 1.0;

// 曲面细分网格的感知"高度"

gl\_TessLevelOuter[0] = height; // 基于感知的边长设置曲面 细分级别

gl\_TessLevelOuter[1] = width; gl\_TessLevelOuter[2] = height; gl\_TessLevelOuter[3] = width; gl\_TessLevelInner[0] = width; gl\_TessLevelInner[1] = height;

}

// 像以前一样将纹理坐标和控制点发送给TES

tcs\_out[gl\_InvocationID] = tc[gl\_InvocationID]; gl\_out[gl\_InvocationID].gl\_Position =

gl\_in[gl\_InvocationID].gl\_Position;

}

将这些控制着色器的更改应用于图12.7中我们场景的实例化（但 不带光照）版本，并将高度图替换为Hastings-Trew的更精细调整的版 本（如图12.8所示），将会生成改善的场景，带有更逼真的地平线细 节（如图12.10所示）。

在此示例中，更改评估着色器中的布局说明符也很有用：

layout (quads, equal\_spacing) in;

更改为：

layout (quads, fractional\_even\_spacing) in;

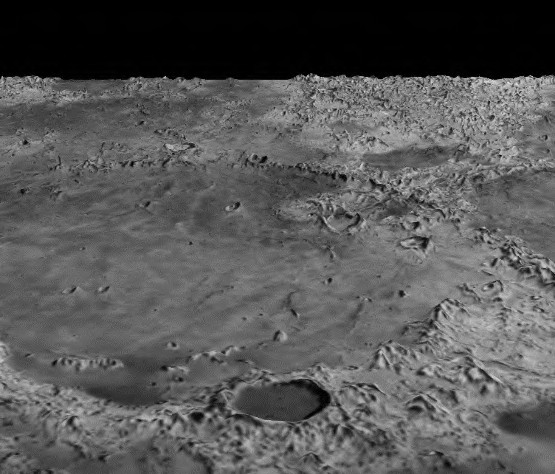


图12.10 具有控制细节级别（LOD）的曲面细分月亮

在静止图像中难以说明这种修改的原因。在动画场景中，当曲面 细分对象在3D空间中移动时，如果使用LOD，有时可以在对象表面上看 到曲面细分级别的变化，看起来像一种叫作“弹出”的摆动伪影。从

等间距变为分数间距，通过使相邻补丁实例的网格几何体更相似，达

成了即使它们的细节级别不同，也可以减少此影响的目的。（参见习 题12.2和12.3。）

使用LOD可以显著降低系统负载。例如，在动画时，如果不控制 LOD，场景可能会出现不稳定或滞后的情况。

将这种简单的LOD技术应用于包含Phong着色的版本（程序12.4） 有点棘手。这是因为相邻补丁实例之间的LOD变化反过来会导致相关法 向量的突然变化，从而导致光照中的弹出伪影！与以往一样，在构建 复杂的3D场景时需要权衡和妥协。

补充说明

将曲面细分与LOD组合在实时虚拟现实应用中特别有用，例如在计 算机游戏中，其需要复杂的现实主义细节和频繁的物体移动和/或摄像 机位置的变化。在本章中，我们已经说明了曲面细分和LOD用于实时地 形生成的应用场景，尽管它也可以应用于其他领域，例如3D模型的位 移贴图（曲面细分顶点被添加到模型的表面，然后被移动以便添加细 节）在计算机辅助设计应用程序中也很有用。

Sellers等人通过消除摄像机后方的补丁中的顶点（他们通过将内 部和外部级别设置为零来实现这一点）[SW15]，进一步扩展了LOD技术

（在程序12.5中显示）。这是一个剔除技术的示例，是一项非常有用 的技术，因为实例化细分的负载仍然可以在系统上正常运行。

程序12.1中描述的createShaderProgram()的4参数版本被添加到

Utils.cpp文件中。稍后，我们将添加其他版本以适应几何着色器阶 段。

习题

12.1 修改程序12.1以试验内部和外部曲面细分级别的各种值， 并观察生成的渲染网格。

12.2 修改程序12.1，将评估着色器中的布局说明符从 equal\_spacing更改为fractional\_even\_spacing，如第12.4节所述。 观察对生成的网格的影响。

12.3 测试程序12.5，将评估着色器中的布局说明符设置为 equal\_spacing，然后设置为fractional\_even\_spacing，如第12.4节 所述。在摄像机移动时观察渲染表面上的效果。您应该能够在第一种 情况下观察弹出伪影，这在第二种情况下大多得到缓解。

12.4 （项目）修改程序12.3以使用自己设计的高度图（可以使 用之前在习题10.2中构建的高度图）。然后添加光照和阴影贴图，以 便细分地形投射阴影。这是一个复杂的练习，因为第一个和第二个阴 影贴图过程中的某些代码需要被移动到评估着色器中。

# 参考资料

[GP16] GIMP Plugin Registry, normalmap plugin, accessed

October 2018.

[HT16] J. Hastings-Trew, JHT’s Planetary Pixel Emporium, accessed October 2018.

[LU16] F. Luna,

Introduction to 3D Game Programming with

, 2nd ed. (Mercury Learning, 2016).

DirectX 12

[SW15] G. Sellers, R. Wright Jr., and N. Haemel,

OpenGL

, 7th ed.

SuperBible: Comprehensive Tutorial and Reference

(Addison-Wesley, 2015).

[TS16] Tessellation, Wikipedia, accessed October 2018.

[[1]](#_bookmark191) GLU工具集之前已经包含了一个名为gluTess的曲面细分实用程 序。2001年，Radeon发布了第一款带有曲面细分支持的商用图形卡， 但很少有工具可以利用它。

[[2]](#_bookmark193) 或线段，但我们将专注于三角形。

[[3]](#_bookmark194) 曲面细分器还能够构建由三角形组成的三角形网格，但本书中未 对此进行介绍。

[[4]](#_bookmark197) 在某些应用程序中，纹理坐标是在外部生成的，例如，在使用曲 面细分为导入的模型提供额外顶点时。在这种情况下，需要对提供的

纹理坐标进行插值。

第13章 几何着色器

在OpenGL管线中，紧跟着曲面细分阶段的是几何阶段。在这一阶 段中，程序员可以选择包含几何着色器。这个阶段实际上在曲面细分 阶段出现之前就已经存在，它在3.2版本（2009年）成为OpenGL核心的 一部分。

与曲面细分一样，几何着色器使程序员能够以顶点着色器中无法 实现的方式操纵顶点组。在某些情况下，可以使用曲面细分着色器或 者几何着色器完成同样的任务，因为它们的功能在某些方面重叠。

13.1 OpenGL中的逐个图元处理

几何着色器阶段位于曲面细分和光栅化之间，位于用于图元处理 的管线段内（见图2.2）。顶点着色器允许一次操作一个顶点，而片段 着色器一次可以操作一个片段（实际上是一个像素），但几何着色器 却可以一次操作一个图元。

回想一下，图元是OpenGL中绘制对象的基本元件。只有少数几种 类型的图元；我们将主要关注操纵三角形图元的几何着色器。因此， 当我们说几何着色器可以一次操作一个图元时，我们通常意味着着色 器一次可以访问三角形的3个顶点。几何着色器允许一次性访问图元中 的所有顶点，然后：

输出相同的图元保持不变；

输出修改了顶点位置的相同类型图元； 输出不同类型的图元； 输出更多的其他图元； 删除图元（根本不输出）。

与曲面细分评估着色器类似，可以在几何着色器中将传入的顶点 属性作为数组进行访问。但是，在几何着色器中，传入属性数组仅索 引到图元尺寸那么大。例如，如果图元是三角形，则可用索引为0、 1、2。使用预先定义的数组gl\_in访问顶点数据本身，如下所示。

gl\_in[2].gl\_Position // 第三个顶点的位置

与曲面细分评估着色器类似，几何着色器输出的顶点属性都是标 量。也就是说，输出是形成图元的各个顶点（它们的位置和其他属性 变量，如果有的话）的流。

有一个布局修饰符用于设置图元输入/输出类型和输出大小。特殊 的GLSL命令EmitVertex()指定了将要输出一个顶点。特殊的GLSL命令 EndPrimitive()表示一个特定的图元构建完成。

有一个内置变量gl\_PrimitiveIDIn，它保存当前图元的ID。ID从0 开始，并计数到图元总数减1。

我们将探讨四种常见的操作类型：

修改图元； 删除图元；

添加图元；

更改图元类型。

13.2 修改图元

当通过对图元（通常为三角形）的单独更改就可以影响对象形状 的改变时，使用几何着色器就很方便。

例如，考虑我们之前在图7.12中呈现的环面。假设环面代表内部 的空间（例如当表示轮胎时），而我们想要给它“充气”。简单地在 C++/OpenGL代码中应用比例缩放因子将无法实现这一点，因为它的基 本形状不会改变。想要让其显示出“充气”的外观，还需要在环面伸 入空的中心空间时使内孔变小。

解决这个问题的一种方法是将表面法向量添加到每个顶点。虽然 这可以在顶点着色器中完成，但是我们在几何着色器中进行练习。程 序13.1显示了GLSL几何着色器的代码。其他模块与程序7.3相同，只有 一些小改动：片段着色器输入名称现在需要反映几何着色器的输出

（例如，varyingNormal变为varyingNormalG），C++/OpenGL应用程序 需要编译几何着色器并在链接之前将其附加到着色器程序。新着色器 被指定为几何着色器，如下所示。

GLuint gShader = glCreateShader(GL\_GEOMETRY\_SHADER);

程序13.1 几何着色器：修改顶点

#version 430

layout (triangles) in;

in vec3 varyingNormal[ ]; // 来自顶点着色器的输入

in vec3 varyingLightDir[ ];

in vec3 varyingHalfVector[ ];

out vec3 varyingNormalG; // 输出给光栅着色器然后到片段着色器

out vec3 varyingLightDirG; out vec3 varyingHalfVectorG;

layout (triangle\_strip, max\_vertices=3) out;

// 矩阵和光照统一变量和以前一样

. . .

void main (void)

{ // 沿着法向量移动顶点，并将其他顶点属性原样传递

for (int i=0; i<3; i++)

{ gl\_Position = proj\_matrix \* gl\_in[i].gl\_Position + normalize(vec4(varyingNormal[i],1.0))

\* 0.4;

varyingNormalG = varyingNormal[i]; varyingLightDirG = varyingLightDir[i]; varyingHalfVectorG = varyingHalfVector[i]; EmitVertex();

}

EndPrimitive();

}

在程序13.1中需要注意，与顶点着色器的输出变量对应的输入变 量被声明为数组。这为程序员提供了一种机制，可以使用索引0、1和2 访问三角形图元中的每个顶点及其属性。我们希望沿着它们的表面法 向量向外移动这些顶点。在顶点着色器中，顶点和法向量都已经被转 换到视图空间。我们为每个传入的顶点位置

（gl\_in[i].gl\_Position）添加法向量的一小部分，然后将投影矩阵 应用于结果，生成每个输出gl\_Position。

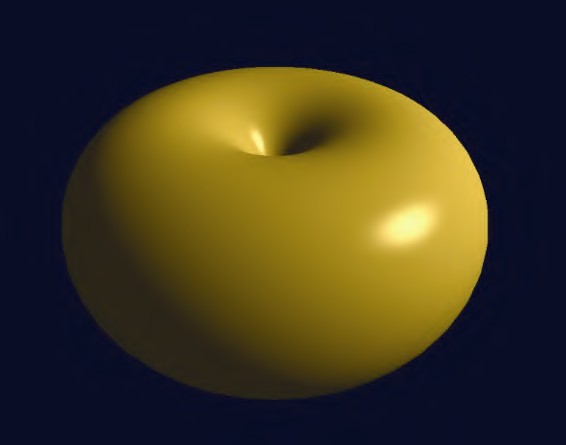


图13.1 “充气”的环面，顶点由几何着色器修改

值得注意的是，使用GLSL调用EmitVertex()来指定我们何时完成 了计算输出gl\_Position及其相关的顶点属性并准备输出顶点。 EndPrimitive()调用指定我们已经完成了组成图元（在本例中为三角 形）的一组顶点的定义。结果如图13.1所示。

几何着色器包括两个布局限定符。第一个指定输入图元类型，并 且必须与C++端glDrawArrays()或glDrawElements()调用中的图元类型 兼容。选项如表13.1所示。

表13 .1 图元输入类型的选项

|  |  |  |
| --- | --- | --- |
| 几何着色器输入图元 类型 | 与gl D raw A rrays() 调用兼容的图元类 型 | 每次调用顶 点的数量 |
| points | GL\_POINTS | 1 |
| lines | GL\_LINES, GL\_LINE\_STRIP | 2 |

|  |  |  |
| --- | --- | --- |
| 几何着色器输入图元 类型 | 与gl D raw A rrays() 调用兼容的图元类 型 | 每次调用顶 点的数量 |
| lines\_adjacency | GL\_LINES\_ADJACENCY, GL\_LINE\_STRIP\_ADJACENCY | 4 |
| triangles | GL\_TRIANGLES, GL\_TRIANGLE\_STRIP, GL\_TRIANGLE\_FAN | 3 |
| triangles\_adjacency | CL\_TRIANGLES\_ADJACENCY, GL\_TRIANGLE\_STRIP\_ADJACENCY | 6 |

各种OpenGL图元类型（包括“strip”和“fan”类型）在第4章中 讲过。“相邻”类型在OpenGL中用来与几何着色器一起使用，并且它 们可以访问与图元相邻的顶点。我们在本书中不使用它们，但为了完 整性，依然列出它们。

输出图元类型必须是points、line\_strip或triangle\_strip。请 注意，输出布局限定符也会指定着色器在每次调用中输出的最大顶点 数。

在顶点着色器中可以更容易地对环面进行这种特定的改变。然 而，假设不是沿着自己的表面法向量向外移动每个顶点，而是希望将 每个三角形沿其表面法向量向外移动，实际上是将环面的组成三角形 向外“爆炸”。顶点着色器做不到这一点，因为计算三角形的法向量

需要对3个三角形顶点的顶点法向量进行平均，并且顶点着色器一次只

能访问三角形中一个顶点的顶点属性。但是，我们可以在几何着色器 中执行此操作，因为几何着色器可以访问每个三角形中的所有3个顶 点。我们平均它们的法向量来计算三角形的曲面法向量，然后将该平 均法向量加给三角形图元中的每个顶点。图13.2、图13.3和图13.4分 别显示了曲面法向量的平均值、修改后的几何着色器main()代码和输 出的结果。

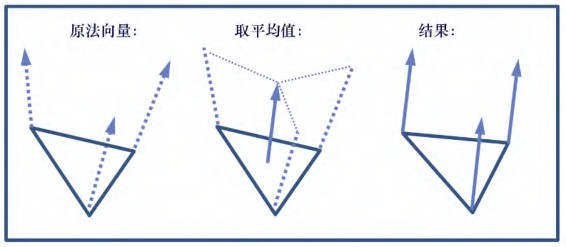


图13.2 将平均三角形曲面法向量应用于三角形顶点

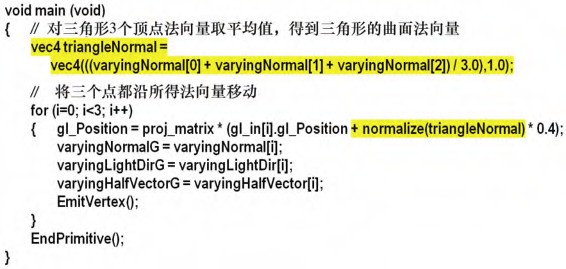


图13.3 修改了几何着色器，用于“爆炸”环面



图13.4 “爆炸”的环面

通过确保环面的内部也是可见的（通常这些三角形会被OpenGL剔 除，因为它们是“背面”），可以改善“爆炸”环面的外观。一种解 决方式是使环面被渲染两次，一次以正常方式进行，一次使缠绕顺序 反转（使缠绕顺序反转实际上相当于切换哪些面朝向前方，哪些面朝 向后方）。我们还向着色器（通过统一变量）发送一个标志，以禁用 背向三角形上的漫反射和镜面光，以使它们不那么突出。代码的更改 如下。

对display()函数的修改：

. . .

// 绘制前向三角形——启用光照

glUniform1i(lLoc, 1); // 用来启用、禁用漫反射、镜面光组件的统一变量的 位置

glFrontFace(GL\_CCW);

glDrawElements(GL\_TRIANGLES, numTorusIndices, GL\_UNSIGNED\_INT, 0);

// 绘制后向三角形——禁用光照 glUniform1i(lLoc, 0); glFrontFace(GL\_CW);

glDrawElements(GL\_TRIANGLES, numTorusIndices, GL\_UNSIGNED\_INT, 0);

对片段着色器的修改：

. . .

if (enableLighting == 1)

{ fragColor = …

}

else

// 当渲染前向表面时，使用正常的光照计算

// 当渲染后向表面时，只启用环境光照组件

{ fragColor = globalAmbient \* material.ambient + light.ambient \*

material.ambient;

}

由此产生的“爆炸”环面，包括背面，如图13.5所示。

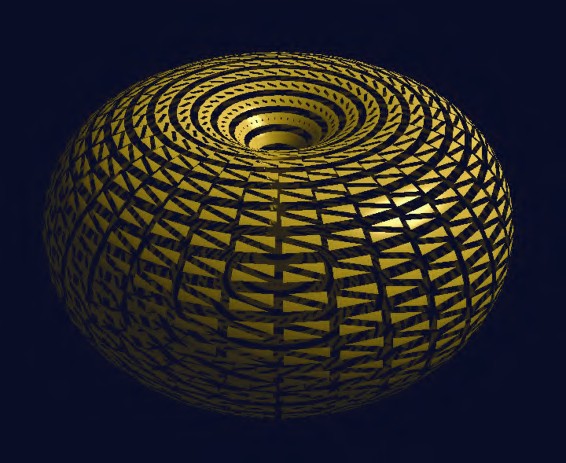


图13.5 “爆炸”的环面，包括背面

13.3 删除图元

几何着色器的一个常见用途是通过合理地删除一些图元来从简单 的对象构建丰富的装饰对象。例如，从我们的环面中移除一些三角形 可以将其变成一种复杂的格子结构，而从零开始建模这个结构是更加 困难的。执行此操作的几何着色器显示在程序13.2中，输出如图13.6 所示。

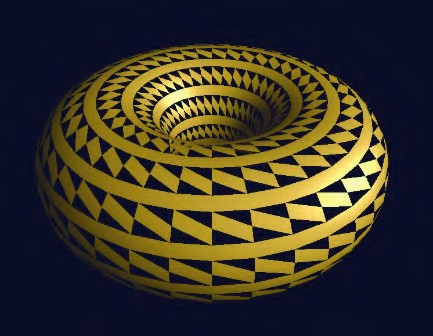


图13.6 几何着色器：删除图元

程序13.2 几何着色器：删除图元

// 输入、输出和统一变量和以前一样

. . .

void main (void)

{ if ( mod(gl\_PrimitiveIDIn,3) != 0 )

{ for (int i=0; i<3; i++)

{ gl\_Position = proj\_matrix \* gl\_in[i].gl\_Position; varyingNormalG = varyingNormal[i]; varyingLightDirG = varyingLightDir[i]; varyingHalfVectorG = varyingHalfVector[i]; EmitVertex();

}

}

EndPrimitive();

}

不需要对代码进行其他更改。请注意这里使用了mod函数——所有 顶点，除了每3个图元中的第一个图元的顶点被忽略之外，都被传递。 在这里，渲染背向三角形也可以提高真实感，如图13.7所示。

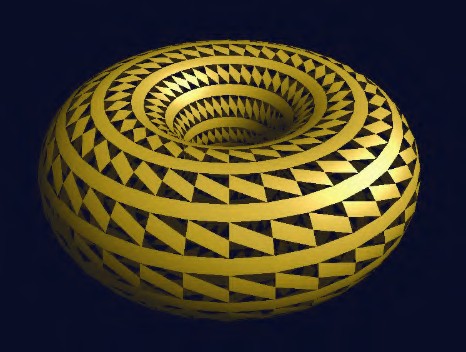


图13.7 显示背面的图元删除

13.4 添加图元

也许几何着色器最有趣和最有用的用途是为正在渲染的模型添加 额外的顶点和/或图元。这使得可以进行诸如增加对象中的细节以改善 高度贴图，或者完全改变对象的形状之类的事情。

考虑以下示例，我们将环面中的每个三角形更改为一个微小的三 角形金字塔。

我们的策略类似于我们之前的“爆炸”环面示例，如图13.8所 示。传入三角形图元的顶点用于定义金字塔的基座。金字塔的壁由那 些顶点和通过平均原始顶点的法向量计算的新点（称为“尖峰点”） 构成。然后通过从尖峰点到基座的两个向量的叉积计算金字塔的3个 “边”中的每一个的新法向量。

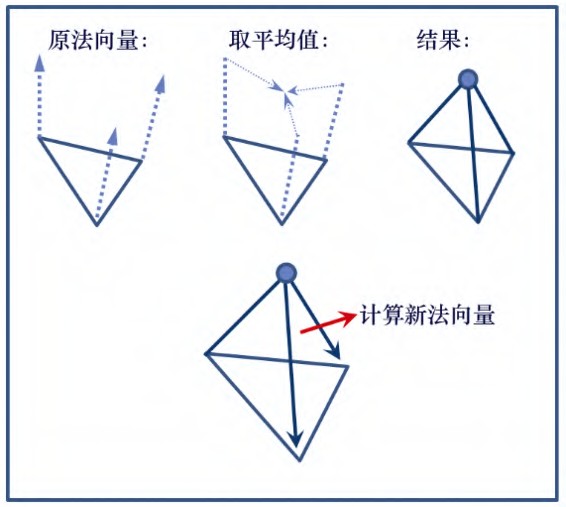


图13.8 将三角形转换为金字塔

程序13.3中的几何着色器为环面中的每个三角形图元执行此操 作。对于每个输入三角形，它输出3个三角形图元，总共9个顶点。每 个新三角形都在函数makeNewTriangle()中构建，该函数被调用3次。 它计算指定三角形的法向量，然后调用函数setOutputValues()为发出 的每个顶点分配适当的输出顶点属性。在发出所有3个顶点之后，它调 用EndPrimitive()。为了确保准确地执行光照，为每个新创建的顶点 计算光照方向向量的新值。

程序13.3 几何着色器：添加图元

. . .

vec3 newPoints[ ], lightDir[ ];

float sLen = 0.01; // sLen 是"尖峰长度"，小金字塔的高度

void setOutputValues(int p, vec3 norm)

{ varyingNormal = norm; varyingLightDir = lightDir[p]; varyingVertPos = newPoints[p];

gl\_Position = proj\_matrix \* vec4(newPoints[p], 1.0);

}

void makeNewTriangle(int p1, int p2)

{ // 为这个三角形生成表面法向量

vec3 c1 = normalize(newPoints[p1] - newPoints[3]); vec3 c2 = normalize(newPoints[p2] - newPoints[3]); vec3 norm = cross(c1,c2);

// 生成并发出3个顶点

setOutputValues(p1, norm); EmitVertex(); setOutputValues(p2, norm); EmitVertex(); setOutputValues(3, norm); EmitVertex(); EndPrimitive();

}

void main(void)

{ // 给三个三角形顶点加上原始表面法向量

vec3 sp0 = gl\_in[0].gl\_Position.xyz + varyingOriginalNormal[0]\*sLen;

vec3 sp1 = gl\_in[1].gl\_Position.xyz + varyingOriginalNormal[1]\*sLen;

vec3 sp2 = gl\_in[2].gl\_Position.xyz + varyingOriginalNormal[2]\*sLen;

// 计算组成小金字塔的新点

newPoints[0] = gl\_in[0].gl\_Position.xyz; newPoints[1] = gl\_in[1].gl\_Position.xyz; newPoints[2] = gl\_in[2].gl\_Position.xyz;

newPoints[3] = (sp0 + sp1 + sp2)/3.0; // 尖峰点

// 计算从顶点到光照的方向

lightDir[0] = light.position - newPoints[0]; lightDir[1] = light.position - newPoints[1]; lightDir[2] = light.position - newPoints[2]; lightDir[3] = light.position - newPoints[3];

// 构建3个三角形，以组成小金字塔的表面 makeNewTriangle(0,1); // 第三个点永远是尖峰点 makeNewTriangle(1,2);

makeNewTriangle(2,0);

}

结果输出如图13.9所示。如果尖峰长度（sLen）变量增加，则添 加的表面“金字塔”将更高。然而，在没有阴影的情况下，它们可能 看起来并不真实。将阴影贴图添加到程序13.3留作练习。

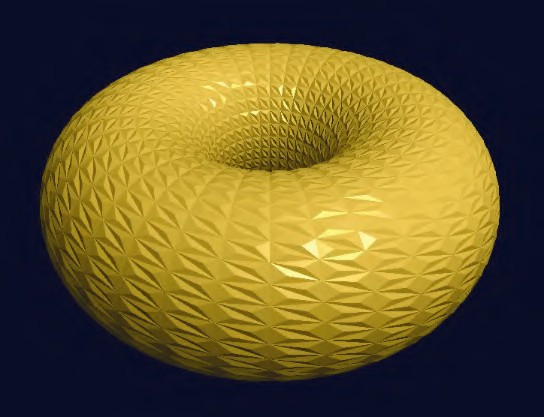


图13.9 几何着色器：添加图元

仔细应用这种技术可以模拟尖峰、荆棘和其他精细表面突起，或 者反向的压痕、凹坑（参考资料[DV14, TR13, KS16]）等。

13.5 更改图元类型

OpenGL允许在几何着色器中更改图元类型。此功能的一个常见用 途是将输入三角形转换为一个或多个输出线段，来模拟毛发或头发。 虽然生成令人信服的头发仍然是更难的现实世界项目之一，但几何着 色器可以在许多情况下帮助实现实时渲染。

程序13.4显示了一个几何着色器，它将每个输入的3个顶点的三角 形转换为一个向外的两个顶点的线段。它首先通过平均三角形顶点位 置生成三角形的质心，来计算头发束的起点。然后它使用和程序13.3 中相同的“尖峰点”作为头发的终点。输出图元被指定为具有两个顶 点的线段，第一个顶点是起点，第二个顶点是终点。结果显示在图 13.10中，用于实例化维数为72个切片的环面。

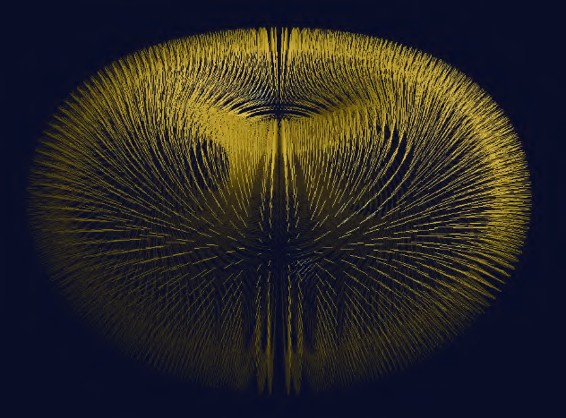


图13.10 将三角形图元更改为线图元

当然，这仅仅是产生完全逼真头发的起点。使头发弯曲或移动将 需要若干修改，例如为线条生成更多顶点并沿曲线计算它们的位置和/ 或结合随机性。由于线段没有明显的表面法向量，光照会很复杂；在 这个例子中，我们简单地指定法向量与原始三角形的表面法向量相 同。

程序13.4 几何着色器：改变图元类型

layout (line\_strip, max\_vertices=2) out;

. . .

void main(void)

{ vec3 op0 = gl\_in[0].gl\_Position.xyz;

// 原始三角形顶点

vec3 op1 = gl\_in[1].gl\_Position.xyz; vec3 op2 = gl\_in[2].gl\_Position.xyz;

vec3 ep0 = gl\_in[0].gl\_Position.xyz + varyingNormal[0]\*sLen;

// 偏移三角形顶点

vec3 ep1 = gl\_in[1].gl\_Position.xyz + varyingNormal[1]\*sLen; vec3 ep2 = gl\_in[2].gl\_Position.xyz + varyingNormal[2]\*sLen;

// 计算组成小线段的新点

vec3 newPoint1 = (op0 + op1 + op2)/3.0;

始）点

vec3 newPoint2 = (ep0 + ep1 + ep2)/3.0;

// 原始（起

// 结束点

gl\_Position = proj\_matrix \* vec4(newPoint1, 1.0);

varyingVertPosG = newPoint1;

varyingLightDirG = light.position - newPoint1; varyingNormalG = varyingNormal[0]; EmitVertex();

gl\_Position = proj\_matrix \* vec4(newPoint2, 1.0); varyingVertPosG = newPoint2;

varyingLightDirG = light.position - newPoint2; varyingNormalG = varyingNormal[1]; EmitVertex();

EndPrimitive();

}

补充说明

几何着色器吸引人的一点在于它们相对容易使用。虽然几何着色 器的许多应用可以使用曲面细分来实现，但几何着色器的机制通常使 它们更容易实现和调试。当然，几何与曲面细分的相对适用范围取决 于特定的应用。

生成令人信服的真实头发或毛发具有挑战性，并且根据应用场景 需要采用多种技术。在某些情况下，简单的纹理就足够了，或者可以 使用曲面细分或几何着色器，例如本章所示的基本技术。当需要更真 实的效果时，移动（动画）和光照变得棘手。头发和毛发生成的两个 专用工具是HairWorks和TressFX。HairWorks是NVIDIA GameWorks套件

[GW18]的一部分，而TressFX是由AMD开发的[TR18]。前者适用于OpenGL

和DirectX，而后者仅适用于DirectX。使用TressFX的例子可以在

[GP14]中找到。

习题

13.1 修改程序13.1，使其将每个顶点略微移向其原始三角形的

中心。结果应该类似于图13.5中的爆炸环面，但没有环面尺寸的整体 变化。

13.2 修改程序13.2，使其删除每第2个图元或每第4个图元（而 不是每第3个图元），并观察对生成的渲染环面的影响。此外，尝试将 实例化环面的维度更改为不是3的倍数的值（例如40），同时仍然删除 每第3个图元，这会有许多可能的影响。

13.3 （项目）修改程序13.4以额外渲染原始环面。也就是说， 渲染一个有光照的环面（如前面第7章所述）和向外线段（使用几何着 色器），使“头发”看起来像是从环面中出来的。

13.4 （研究和项目）修改程序13.4，使其生成具有两个以上顶 点的向外线段，这些顶点排列使得线段看起来略微弯曲。

# 参考资料

[DV14] J. deVries, LearnOpenGL, 2014, accessed October 2018.

[GP14]

GPU Pro 5: Advanced Rendering Techniques

Engel (CRC Press, 2014).

, ed. W.

[GW18] NVIDIA GameWorks Suite, 2018, accessed May 2018.

[KS16] J. Kessenich, G. Sellers, and D. Shreiner,

OpenGL

Programming Guide: The Official Guide to Learning OpenGL,

, 9th ed. (Addison-Wesley, 2016).

Version 4.5 with SPIR-V

[TR13] P. Trettner, “Prototype Grass” (blog), 2013, accessed October 2018.

[TR18] TressFX Hair, AMD, 2018, accessed May 2018.

第14章 其他技术

在本章中，我们将使用在本书中学到的工具来探索各种技术。有 些我们会完全讲解，而其他一些我们将只会粗略描述。图形编程是一 个巨大的领域，本章绝不是全面的，而是介绍了多年来发展的一些创 造性效果。

14.1 雾

通常当人们想到雾时，他们会想到有雾的早晨，能见度很低。事 实上，大气雾霾（如雾）比我们大多数人认为的更常见。大多数时 候，空气中都会有一定程度的雾霾，我们已经习惯于看到它，通常不 会意识到它的存在，所以我们可以通过引入雾来增强我们室外场景的 真实感——即使只是少量。

雾也可以增强深度感。近处物体比远处物体具有更高的清晰度， 对于我们的大脑是可以用来破译3D场景的地形结构的另一个视觉提 示。

模拟雾的方法有很多种，从非常简单的模型到包含光散射效应的 复杂模型。即使非常简单的方法也是有效的。有一种方法是基于物体 距眼睛的距离将实际像素颜色与另一种颜色（“雾”的颜色通常是灰 色或蓝灰色——也用于背景颜色）混合。

图14.1（见彩插）说明了这个概念。眼睛（相机）显示在左侧，

两个红色物体放置在视锥体中。圆柱体更靠近眼睛，所以它主要是原 始颜色（红色）；立方体远离眼睛，所以它主要是雾色。对于这个简 单的实现，几乎所有的计算都可以在片段着色器中执行。

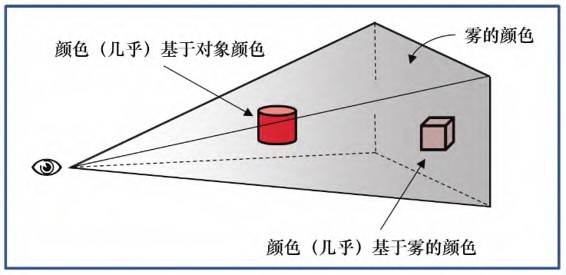


图14.1 雾：基于距离的混合

程序14.1显示了一个非常简单的雾算法的相关代码，该算法按照 从相机到像素的距离，使用从对象颜色到雾颜色的线性混合。具体来 说，此示例将雾添加到程序10.4中的高度贴图示例。

程序14.1 简单的雾生成

顶点着色器

. . .

out vec3 vertEyeSpacePos;

. . .

// 在视觉空间中不考虑透视计算顶点位置，并将它发送给片段着色器

// 变量"p"是高度贴图后的顶点，正如程序10.4中所述

vertEyeSpacePos = (mv\_matrix \* p).xyz;

片段着色器

. . .

in vec3 vertEyeSpacePos; out vec4 fragColor;

. . .

void main(void)

{ vec4 fogColor = vec4(0.7, 0.8, 0.9, 1.0);

float fogStart = 0.2; float fogEnd = 0.8;

// 在视觉空间中从摄像机到顶点的距离就是到这个顶点的向量的长度，因为摄像机在 视觉空间中的(0,0,0)位置

float dist = length(vertEyeSpace.xyz);

float fogFactor = clamp(((fogEnd - dist) / (fogEnd - fogStart)), 0.0, 1.0);

fragColor = mix(fogColor, (texture(t,tc), fogFactor);

}

变量fogColor指定雾的颜色。变量fogStart和fogEnd指定输出颜 色从对象颜色过渡到雾色的范围（在视觉空间中），并且可以调整以 满足场景的需要。在对象颜色中混合的雾的百分比在变量fogFactor中 计算，该变量是顶点与fogEnd的接近程度与过渡区域的总长度之比。 GLSL的clamp()函数用于将此比率限制在值0.0和1.0之间。然后，GLSL 的mix()函数根据fogFactor的值返回雾颜色和对象颜色的加权平均 值。图14.2（见彩插）展示了向具有高度贴图地形的场景添加雾

（[LU16]的岩石纹理也已应用）。

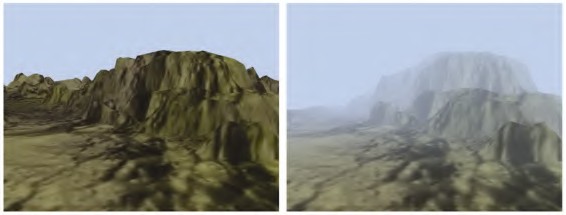


图14.2 雾的例子

14.2 复合、混合、透明度

我们已经看到了一些混合的例子，比如第7章的补充说明以及我们

刚才实现的雾。但是，我们还没有看到如何在像素操作期间利用片段 着色器之后的混合（或合成）功能（回想一下图2.2所示的管线序 列）。透明度在那个步骤被处理，我们现在来了解一下。

在本书中，我们经常使用vec4数据类型来表示齐次坐标系中的3D 点和向量。您可能已经注意到我们还经常使用vec4来存储颜色信息， 其中前3个值由红色、绿色和蓝色组成，那么第四个元素是什么？

颜色中的第四个元素称为Alpha通道，用来指定颜色的不透明度。 不透明度是衡量像素颜色不透明程度的指标。Alpha值为0表示“无不 透明度”或完全透明。Alpha值为1表示“不透明度满值”，也就是完

全不透明。在某种意义上，颜色的“透明度”是1−α，其中α是

Alpha通道的值。

回忆一下第2章，像素操作利用Z缓冲区，当发现另一个对象在该 像素的位置更近时，通过替换现有的像素颜色来实现隐藏面消除。我 们实际上可以更好地控制这个过程——可以选择混合两个像素。

当渲染一个像素时，它被称为“源”像素。已经在帧缓冲器中的 像素（可能是从先前的对象渲染得来）被称为“目标”像素。OpenGL 提供了许多选项，用于决定最终将两个像素中的哪一个或者它们的组 合，放置在帧缓冲区中。请注意，像素操作步骤不是可编程阶段—— 因此用于配置所需合成的OpenGL工具可在C++应用程序中（而不是在着 色器中）找到。

用于控制合成的两个OpenGL函数是glBlendEquation(mode)和

glBlendFunc(srcFactor, destFactor)。图14.3显示了合成过程的概 述。

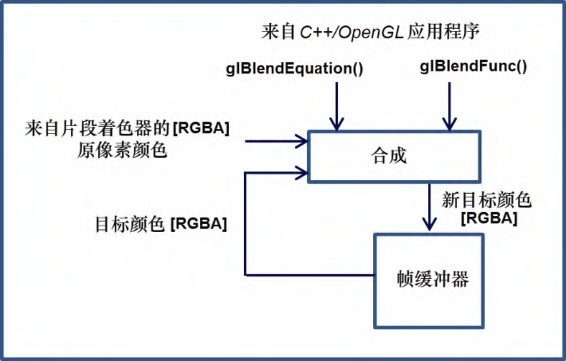


图14.3 OpenGL合成概述

合成过程的工作过程如下。

（1）源像素和目标像素分别乘以源因子和目标因子。源和目标因 子在blendFunc()函数调用中指定。

（2）然后使用指定的blendEquation来组合修改后的源像素和目 标像素以生成新的目标颜色。混合方程在glBlendEquation()调用中指 定。

glBlendFunc()参数的常见选项（即srcFactor和destFactor）如 表14.1所示。

表14 .1 glB lend Func()参数的常见选项

|  |  |
| --- | --- |
| gl B lend Func() 参数 | src Factor或dest Factor的结果 |
| GL\_ZERO | (0,0,0,0) |
| GL\_ONE | (1,1,1,1) |
| GL\_SRC\_COLOR | (Rsrc, Gsrc, Bsrc, Asrc) |
| GL\_ONE\_MINUS\_SRC\_COLOR | (1,1,1,1)− (Rsrc, Gsrc, Bsrc, Asrc) |
| GL\_DST\_COLOR | (Gdest, Gdest, Bdest, Adest) |
| GL\_ONE\_MINUS\_DST\_COLOR | (1,1,1,1)− (RblendColor, GblendColor, BblendColor, AblendColor) |
| GL\_SRC\_ALPHA | (Asrc, Asrc, Asrc, Asrc) |
| GL\_ONE\_MINUS\_SRC\_ALPHA | (1, 1, 1, 1)−(Asrc, Asrc, Asrc, Asrc) |
| GL\_DST\_ALPHA | (Adest, Adest, Adest, Adest) |
| GL\_ONE\_MINUS\_DST\_ALPHA | (1, 1, 1, 1)−(Adest, Adest, Adest, Adest) |
| GL\_CONSTANT\_COLOR | (RblendColor, GblendColor, BblendColor, AblendColor) |

|  |  |
| --- | --- |
| gl B lend Func() 参数 | src Factor或dest Factor的结果 |
| GL\_ONE\_MINUS\_CONSTANT\_COLOR | (1,1,1,)− (RblendColor, GblendColor, BblendColor, AblendColor) |
| GL\_CONSTANT\_ALPHA | (AblendColor, AblendColor, AblendColor, AblendColor) |
| GL\_ONE\_MINUS\_CONSTANT\_ALPHA | (1,1,1,)− (AblendColor, AblendColor, AblendColor, AblendColor) |
| GL\_ALPHA\_SATURATE | (f, f, f, 1)，其中 = min(Asrc, 1)  f |

那些用到“blendColor”（GL\_CONSTANT\_COLOR等）的选项需要额 外调用glBlendColor()来指定将用于计算混合函数结果的常量颜色。 还有一些其他混合函数未在表14.1中显示。

glBlendEquation()参数（混合模式）的可能选项如表14.2所示。

表14 .2 glB lend E quation()参数的可能选项

|  |  |
| --- | --- |
| 模式 | 混合颜色 |
| GL\_FUNC\_ADD | result=sourceRGBA+destinationRGBA |

|  |  |
| --- | --- |
| 模式 | 混合颜色 |
| GL\_FUNC\_SUBTRACT | result=sourceRGBA−destinationRGBA |
| GL\_FUNC\_REVERST\_SUBTRACT | result=destinationRGBA−sourceRGBA |
| GL\_MIN | result=min(sourceRGBA, destinationRGBA) |
| GL\_MAX | result=max(sourceRGBA, destinationRGBA) |

glBlendFunc()默认设置srcFactor为GL\_ONE（1.0），destFactor 为GL\_ZERO（0.0）。glBlendEquation()的默认值为GL\_FUNC\_ADD。因 此，在默认情况下，源像素不变（乘以1），目标像素被按比例缩小到 0，并且两者相加意味着源像素变为帧缓冲区的颜色。

还有命令glEnable(GL\_BLEND)和glDisable(GL\_BLEND)，它们可用 于告诉OpenGL应用指定的混合，或忽略它。

我们不会在这里说明所有选项的效果，但我们将介绍一些说明性 示例。假设我们在C++/OpenGL应用程序中指定以下设置：

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA) glBlendEquation(GL\_FUNC\_ADD)

合成将如下进行。

（1）源像素按其Alpha值缩放。

（2）目标像素按1−srcAlpha（源透明度）缩放。

（3）像素值加在一起。

例如，如果源像素为红色，具有75%不透明度，即[1,0,0,0.75]， 并且目标像素包含完全不透明的绿色，即[0,1,0,1]，则结果放在帧缓 冲区将是：

srcPixel \* srcAlpha = [0.75, 0, 0, 0.5625]

destPixel \* (1-srcAlpha) = [0, 0.25, 0, 0.25]

resulting pixel = [0.75, 0.25, 0, 0.8125]

也就是说，主要是红色，有些是绿色的，而且基本上是实色。这 个设置的总体效果是让目标像素以与源像素的透明度相对应的量显 示。在此示例中，帧缓冲区中的像素为绿色，输入像素为红色，透明 度为25%（不透明度为75%）。因此允许一些绿色通过红色显示。

事实证明，混合函数和混合方程的这些设置在许多情况下都能很 好地工作。我们将它们应用到包含两个3D模型的场景中的实际示例中 去：一个环面和环面前的金字塔。图14.4显示了这样一个场景，左边 是一个不透明的金字塔，右边是金字塔的Alpha值设置为0.8。光照已 经添加。

对于许多应用——例如创建平面“窗口”作为房屋模型的一部 分，这种简单的透明度实现可能就足够了。但是，在图14.4所示的示 例中，存在相当明显的不足之处。尽管金字塔模型现在实际上是透明

的，但实际透明的金字塔不仅应该显示其背后的对象，还应该显示其

自身的背面。

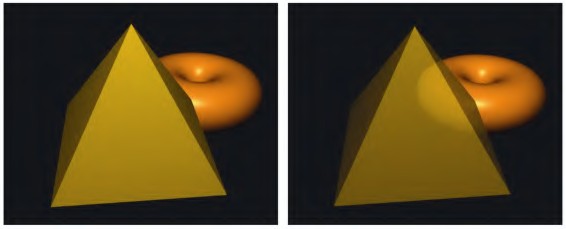


图14.4 金字塔的Alpha = 1.0（左），Alpha = 0.8（右）

实际上，金字塔的背面没有出现的原因是因为我们启用了背面剔 除。一个合理的想法可能是在绘制金字塔时禁用背面剔除。但是，这 通常会产生其他伪影，如图14.5左图所示。简单地禁用背面剔除的问 题在于混合的效果取决于渲染表面的顺序（因为这决定了源像素和目 标像素），并且我们不总是能够控制渲染顺序。通常有利的是首先渲 染不透明对象，以及在后面的对象（例如环面），最后再渲染透明对 象。这也适用于金字塔的表面，并且在这种情况下，包括金字塔底部 的两个三角形看起来不同的原因是它们中的一个在金字塔的前面之前 被渲染而一个在之后被渲染。诸如此类的伪影有时被称为“顺序”伪 影，并且它们可以在透明模型中显示，因为我们不总是能预测其三角 形将被渲染的顺序。

我们可以通过从背面开始分别渲染正面和背面来解决金字塔示例 中的问题。程序14.2显示了执行此操作的代码。我们通过统一变量来

指定金字塔的Alpha值并传递给着色器程序，然后通过将指定的Alpha

替换为计算的输出颜色将其应用于片段着色器中。

另请注意，要使光照正常工作，我们必须在渲染背面时翻转法向 量。我们通过向顶点着色器发送一个标志来完成此操作，然后我们在 其中翻转法向量。

程序14.2 透明度的两遍混合

C++ / OpenGL应用程序 —— 在渲染金字塔的display()函数中：

. . .

glEnable(GL\_CULL\_FACE);

. . .

glEnable(GL\_BLEND); // 配置混合设置

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA); glBlendEquation(GL\_FUNC\_ADD);

glCullFace(GL\_FRONT); // 先渲染金字塔的背面 glProgramUniform1f(renderingProgram, aLoc, 0.3f); // 背面非 常透明

glProgramUniform1f(renderingProgram, fLoc, -1.0f); // 翻转背 面的法向量

glDrawArrays(GL\_TRIANGLES, 0, numPyramidVertices); glCullFace(GL\_BACK); // 然后渲

染金字塔的正面

glProgramUniform1f(renderingProgram, aLoc, 0.7f); // 正面略 微透明

glProgramUniform1f(renderingProgram, fLoc, 1.0f); // 正面不 需要翻转法向量

glDrawArrays(GL\_TRIANGLES, 0, numPyramidVertices);

glDisable(GL\_BLEND);

顶点着色器：

. . .

if (flipNormal < 0) varyingNormal = -varyingNormal;

. . .

片段着色器：

. . .

fragColor = globalAmbient \* material.ambient + ... etc. Blinn-Phong光照一样

fragColor = vec4(fragColor.xyz, alpha);

统一变量中发送的Alpha值替换

// 和

// 使用

这种“两遍”解决方案的结果如图14.5右图所示。

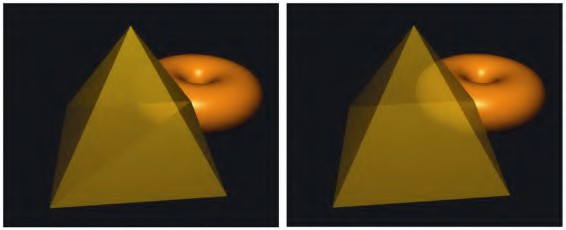


图14.5 透明度和背面：排序伪影（左）和两遍校正（右）

虽然它在这里运行良好，但程序14.2中显示的两遍解决方案并不 总是足够的。例如，一些更复杂的模型可能具有面向前方的隐藏表 面，并且如果这样的对象变得透明，我们的算法将无法渲染模型的那 些隐藏的前向部分。Alec Jacobson描述了一个适用于大量案例的五遍 序列[JA12]。

14.3 用户定义剪裁平面

OpenGL 不仅可以应用于视锥体，还包括了指定剪裁平面的功能。 用户定义的剪裁平面的一个用途是对模型切片。这样就可以通过从简 单的模型开始并从中切片来创建复杂的形状。

剪裁平面使用平面的标准数学定义来定义：

+

ax by

+

cz

+

d

= 0

其中 、

a b

、 和 是用来定义有

c d X

、 和 轴的3D空间中特定平面的参数。

Y Z

参数表示垂直于平面的向量( , ,

a b c

)，以及从原点到平面的距离 。可

d

以使用vec4在顶点着色器中指定这样的平面，如下所示：

vec4 clip\_plane = vec4（0.0,0.0，−1.0,0.2）;

这对应于平面：

(0.0)

x

+ (0.0)

y

+ (−1.0)

z

+ 0.2 = 0

然后，通过使用内置的GLSL变量gl\_ClipDistance[ ]，可以在顶 点着色器中实现裁剪，如下例所示：

gl\_ClipDistance [0] = dot(clip\_plane.xyz, vertPos) + clip\_plane.w;

在此示例中，vertPos指的是在顶点属性（例如来自VBO）中进入 顶点着色器的顶点位置，clip\_plane定义如上。然后我们计算从裁剪 平面到传入顶点的带符号距离（如第3章所示），如果顶点在平面上， 则为0，或者取决于顶点在平面的哪一侧而为负或正。 gl\_ClipDistance数组的下标允许定义多个裁剪距离（即多个平面）。 可以定义的最大用户裁剪平面数量取决于图形卡的OpenGL实现。

然后必须在C++/OpenGL应用程序中启用用户定义的裁剪。内置

OpenGL标识符GL\_CLIP\_DISTANCE0、GL\_CLIP\_DISTANCE1等，对应于每 个gl\_ClipDistance[ ]数组元素。例如，启用第0个用户定义剪裁平 面，如下所示。

glEnable(GL\_CLIP\_DISTANCE0);

将前面的步骤应用到我们的发光环面会产生如图14.6所示的输 出，其中环面的前半部分已经被剪裁了（还应用了旋转以提供更清晰 的视图）。

可能看起来好像环面的底部也被修剪了，但这是因为环面的内表 面没有被渲染。当裁剪会显示形状的内部表面时，也就需要渲染它 们，否则模型将显示得不完整（如图14.6所示）。



图14.6 剪裁一个环面

渲染内表面需要再次调用gl\_DrawArrays()，并颠倒缠绕顺序。此 外，在渲染背向三角形时，必须反转曲面法向量（如上一节所述）。 C++应用程序和顶点着色器的相关修改如程序14.3所示，输出如图14.7 所示。



图14.7 带背面的剪裁

程序14.3 带背面的剪裁

C++ / OpenGL应用程序：

void display(GLFWwindow\* window, double currentTime) {

. . .

flipLoc = glGetUniformLocation(renderingProgram, "flipNormal");

. . .

glEnable(GL\_CLIP\_DISTANCE0);

// 正常绘制外表面 glUniform1i(flipLoc, 0); glFrontFace(GL\_CCW);

glDrawElements(GL\_TRIANGLES, numTorusIndices, GL\_UNSIGNED\_INT, 0);

// 渲染背面，法向量反转 glUniform1i(flipLoc, 1); glFrontFace(GL\_CW);

glDrawElements(GL\_TRIANGLES, numTorusIndices, GL\_UNSIGNED\_INT, 0);

}

顶点着色器：

. . .

vec4 clip\_plane = vec4(0.0, 0.0, -1.0, 0.5); uniform int flipNormal; // 反转法向量的标志

. . .

void main(void)

{ . . .

if (flipNormal==1) varyingNormal = -varyingNormal;

. . .

gl\_ClipDistance[0] = dot(clip\_plane.xyz, vertPos) - clip\_plane.w;

. . .

}

14.4 3D纹理

2D纹理包含由两个变量索引的图像数据，而3D纹理包含相同类型 的图像数据，但是处在由3个变量索引的3D结构中。前两个维度仍然代 表纹理贴图中的宽度和高度，第三个维度代表深度。

因为3D纹理中的数据以与2D纹理类似的方式存储，所以很容易将 3D纹理视为一种3D“图像”。但是，我们通常不将3D纹理源数据称为

3D图像，因为对于这种结构没有常用的图像文件格式（即没有类似的

3D版JPEG，至少没有真正三维的图像）。相反，我们建议将3D纹理视 为一种物质，我们将其浸没（或“浸入”）被纹理化的对象，从而使 对象的表面点从纹理中的相应位置获得颜色。或者可以想象这个物体 被从3D纹理“立方体”中“雕刻”出来，就像雕塑家用一块坚固的大 理石雕刻出一个人物一样。

OpenGL支持3D纹理对象。为了使用它们，我们需要学习如何构建 3D纹理以及如何使用它来纹理化对象。

与可以从标准图像文件构建的2D纹理不同，3D纹理通常是在程序 上生成的。正如之前对2D纹理所做的那样，我们决定分辨率，即每个 维度中的纹素数量。根据纹理中的颜色，我们可以构建包含这些颜色 的三维数组。如果纹理包含可以与各种颜色一起使用的“图案”，我 们可能会建立一个保存图案的数组，例如0和1。

例如，我们可以通过填充对应于所需条纹图案的0和1的数组来构

建表示水平条纹的3D纹理。假设纹理的所需分辨率是200×200×200纹 素，并且纹理由交替的条纹组成，每个条纹高10纹素。通过在嵌套循 环中使用适当的0和1填充数组来构建此类结构的简单函数（假设在这 种情况下，宽度、高度和深度变量均设置为200）将如下所示。

void generate3Dpattern() {

for (int x=0; x<texWidth; x++) {

for (int y=0; y<texHeight; y++) { for (int z=0; z<texDepth; z++) {

if ((y/10) % 2 == 0)

tex3Dpattern[x][y][z] = 0.0; else

tex3Dpattern[x][y][z] = 1.0;

}

}

}

}

存储在tex3Dpattern数组中的图案如图14.8所示（见彩插），0呈 蓝色，1呈黄色。

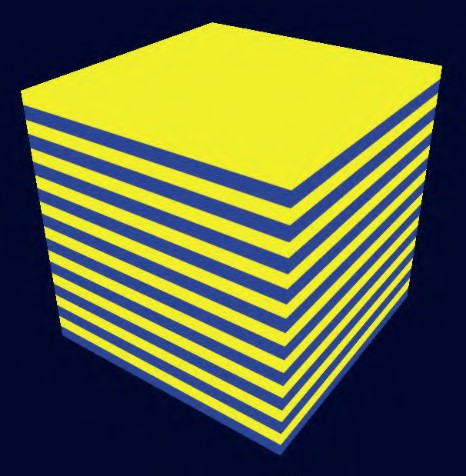


图14.8 条纹3D纹理图案

使用条纹图案对对象进行纹理处理，如图14.8所示，需要执行以

下步骤。

（1）生成如上所示的图案。

（2）使用图案填充所需颜色的字节数组。

（3）将字节数组加载到纹理对象中。

（4）确定对象顶点的适当3D纹理坐标。

（5）在片段着色器中使用适当的采样器来纹理化对象。 3D纹理的纹理坐标范围为［0…1］，与2D纹理的方式相同。 有趣的是，步骤（4）（确定3D纹理坐标）通常比最初怀疑的要简

单得多。事实上，它通常比2D纹理更简单！这是因为（在2D纹理的情 况下）3D对象被2D图像纹理化，我们需要决定如何“展平”3D对象的 顶点（例如通过UV映射）来创建纹理坐标。但是当3D纹理化时，对象 和纹理都具有相同的维度。在大多数情况下，我们希望对象反映纹理 图案，就像它被“雕刻”出来一样（或浸入其中）。所以顶点位置本 身就是纹理坐标！通常所需的只是应用一些简单的缩放以确保对象的 顶点的位置坐标映射到3D纹理坐标的范围[0, 1]。

由于我们通过程序来生成3D纹理，所以我们需要一种从生成的数 据中构造OpenGL纹理贴图的方法。将数据加载到纹理中的过程与我们

之前在第5.12节中看到的类似。在这种情况下，我们用颜色值填充3D

数组，然后将它们复制到纹理对象中。

程序14.4展示出了用于实现所有先前步骤的各种组件，以便使用 程序构建的3D纹理来纹理化具有蓝色和黄色水平条纹的对象。所需的 图案在generate3Dpattern()函数中构建，该函数将图案存储在名为 “tex3Dpattern”的数组中。然后在函数fillDataArray()中构建“图

像”数据，按照图案，该函数使用与RGB颜色

R

、 、 和

G B A

相对应的字节

数据填充3D数组，每个数据在[0, 255]范围内。然后将这些值复制到 load3DTexture()函数中的纹理对象中。

程序14.4 3D纹理：条纹图案

C++ / OpenGL应用程序：

. . .

const int texHeight= 200; const int texWidth = 200; const int texDepth = 200;

double tex3Dpattern[texWidth][texHeight][texDepth];

. . .

// 按照由generate3Dpattern()构建的图案，用蓝色、黄色的RGB值来填充字节数组

void fillDataArray(GLubyte data[ ]) { for (int i=0; i<texWidth; i++) {

for (int j=0; j<texHeight; j++) { for (int k=0; k<texDepth; k++) {

if (tex3Dpattern[i][j][k] == 1.0) {

// 黄色

data[i\*(texWidth\*texHeight\*4) + j\*(texHeight\*4)+ k\*4+0]

= (GLubyte) 255; // red

data[i\*(texWidth\*texHeight\*4) + j\*(texHeight\*4)+ k\*4+1]

= (GLubyte) 255; // green

data[i\*(texWidth\*texHeight\*4) + j\*(texHeight\*4)+ k\*4+2]

= (GLubyte) 0; // blue

data[i\*(texWidth\*texHeight\*4) + j\*(texHeight\*4)+ k\*4+3]

= (GLubyte) 255; // alpha

}

else {

// 蓝色

data[i\*(texWidth\*texHeight\*4) + j\*(texHeight\*4)+ k\*4+0]

= (GLubyte) 0; // red

data[i\*(texWidth\*texHeight\*4) + j\*(texHeight\*4)+ k\*4+1]

= (GLubyte) 0; // green

data[i\*(texWidth\*texHeight\*4) + j\*(texHeight\*4)+ k\*4+2]

= (GLubyte) 255; // blue

data[i\*(texWidth\*texHeight\*4) + j\*(texHeight\*4)+ k\*4+3]

= (GLubyte) 255; // alpha

} } } } }

// 构建条纹的3D图案

void generate3Dpattern() {

for (int x=0; x<texWidth; x++) {

for (int y=0; y<texHeight; y++) { for (int z=0; z<texDepth; z++) {

if ((y/10)%2 == 0)

tex3Dpattern[x][y][z] = 0.0; else

tex3Dpattern[x][y][z] = 1.0;

} } } }

// 将顺序字节数据数组加载进纹理对象

int load3DTexture() { GLuint textureID;

GLubyte\* data = new GLubyte[texWidth\*texHeight\*texDepth\*4]; fillDataArray(data);

glGenTextures(1, &textureID); glBindTexture(GL\_TEXTURE\_3D, textureID); glTexParameteri(GL\_TEXTURE\_3D, GL\_TEXTURE\_MIN\_FILTER,

GL\_LINEAR);

glTexStorage3D(GL\_TEXTURE\_3D, 1, GL\_RGBA8, texWidth, texHeight, texDepth);

glTexSubImage3D(GL\_TEXTURE\_3D, 0, 0, 0, 0, texWidth, texHeight, texDepth,

GL\_RGBA, GL\_UNSIGNED\_INT\_8\_8\_8\_8\_REV, data);

return textureID;

}

void init(GLFWwindow\* window) {

. . .

generate3Dpattern(); // 3D图案和纹理只加载一次，所 以在init()里作

stripeTexture = load3DTexture(); // 为3D纹理保存整型图案ID

}

void display(GLFWwindow\* window, double currentTime) {

. . .

glActiveTexture(GL\_TEXTURE0);

glBindTexture(GL\_TEXTURE\_3D, stripeTexture); glDrawArrays(GL\_TRIANGLES, 0, numObjVertices);

}

顶点着色器

. . .

out vec3 originalPosition; // 原始模型顶点将被用于纹理坐 标

. . .

layout (binding=0) uniform sampler3D s;

void main(void)

{ originalPosition = position; // 将原始模型坐标传递，用作3D 纹理坐标

gl\_Position = proj\_matrix \* mv\_matrix \* vec4(position,1.0);

}

片段着色器

. . .

in vec3 originalPosition; // 接受原始模型坐标，用作3D纹理坐标

out vec4 fragColor;

. . .

layout (binding=0) uniform sampler3D s;

void main(void)

{

fragColor = texture(s, originalPosition/2.0 + 0.5); // 顶点范围为

[−1,+1]，纹理坐标范围为[0,1]

}

在C++/OpenGL应用程序中，load3Dtexture()函数将生成的数据加 载到3D纹理中。它不使用SOIL2来加载纹理，而是直接进行相关的 OpenGL调用，其方式类似于前面5.12节中所述的方式。图像数据应该 被格式化为对应于RGBA颜色分量的字节序列。函数fillDataArray()执 行此操作，应用黄色和蓝色的RGB值，依据由generate3Dpattern()函 数构建并保存在tex3Dpattern数组中的条带图案。另请注意display() 函数中指定了纹理类型GL\_TEXTURE\_3D。

由于我们希望将对象的顶点位置用作纹理坐标，我们将它们从顶

点着色器传递到片段着色器。片段着色器缩放它们，以便它们按照纹 理坐标的标准，被映射到范围[0, 1]。最后，通过sampler3D统一变量 访问3D纹理，该统一变量采用3个参数而不是两个参数。我们使用顶点

的原始

X

、 和 坐标，缩放到正确的范围，以访问纹理。结果如图14.9

Y Z

所示（见彩插）。



图14.9 3D条纹纹理的龙对象

通过修改generate3Dpattern()可以生成更复杂的图案。图14.10 显示了将条带图案转换为3D棋盘的简单更改，产生的效果如图14.11所 示。值得注意的是，如果龙的表面采用2D棋盘纹理图案进行纹理处 理，效果与情况则大不相同。（见习题14.3。）

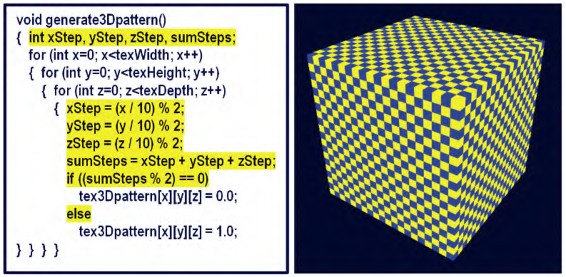


图14.10 生成棋盘3D纹理图案



图14.11 3D棋盘纹理的龙

14.5 噪声

可以使用随机性或噪声来模拟许多自然现象。一种常见的技术是 Perlin噪声[PE85]，它以Ken Perlin命名。Ken Perlin在1997年因开发 生成和使用2D和3D噪声的实用方法而获得奥斯卡奖。[[1]](#_bookmark230)这里描述的程 序基于Perlin的方法。

图形场景中存在许多噪声应用。一些常见的例子是云、地形、木 纹、矿产（如大理石中的矿脉）、烟雾、燃烧、火焰、行星表面和随 机运动。在本节中，我们将重点关注生成包含噪声的3D纹理，然后使 用噪声数据生成复杂材质（如大理石和木材），并模拟动画云纹理以 用于立方体贴图或天幕。包含噪声的空间数据（例如2D或3D）的集合 有时被称为噪声图。

我们首先从随机数据中构建3D纹理贴图。这可以使用上一节中显 示的函数完成，只需进行一些修改。首先，我们使用以下更简单的

generateNoise()函数替换程序14.4中的generate3Dpattern()函数：

#include <random>;

. . .

double noise[noiseWidth][noiseHeight][noiseDepth];

. . .

void generateNoise() {

for (int x=0; x<noiseWidth; x++) {

for (int y=0; y<noiseHeight; y++) {

for (int z=0; z<noiseDepth; z++) { noise[x][y][z] = (double) rand() / (RAND\_MAX +

1.0); // 计算出[0...1]范围内的

// 一个double类型数值

} } } }

接下来，修改程序14.4中的fillDataArray()函数，以便将噪声数 据复制到字节数组中，以便加载到纹理对象中，如下所示。

void fillDataArray(GLubyte data[ ]) { for (int i=0; i<noiseWidth; i++) {

for (int j=0; j<noiseHeight; j++) {

for (int k=0; k<noiseDepth; k++) { data[i\*(noiseWidth\*noiseHeight\*4)+j\*

(noiseHeight\*4)+k\*4+0] =

(GLubyte) (noise[i][j][k] \* 255); data[i\*(noiseWidth\*noiseHeight\*4)+j\*

(noiseHeight\*4)+k\*4+1] =

(GLubyte) (noise[i][j][k] \* 255); data[i\*(noiseWidth\*noiseHeight\*4)+j\*

(noiseHeight\*4)+k\*4+2] =

(GLubyte) (noise[i][j][k] \* 255); data[i\*(noiseWidth\*noiseHeight\*4)+j\*

(noiseHeight\*4)+k\*4+3] =

(GLubyte) 255;

} } } }

程序14.4的其余部分，用于将数据加载到纹理对象并将其应用于 模型，依然保持不变。我们可以通过将它应用于我们的简单立方体模

型来查看这个3D噪声图，如图14.12所示。在此示例中，noiseHeight

= noiseWidth = noiseDepth = 256。

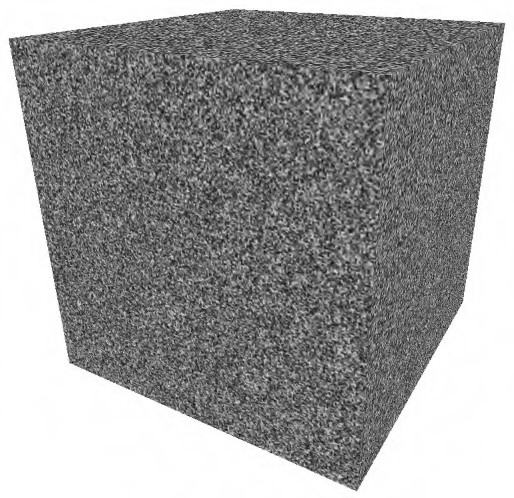


图14.12 3D噪声数据纹理的立方体

这是一个3D噪声图，虽然它不是非常有用（因为它太嘈杂了，很 难有很多实际应用）。为了制作更实用、更可调的噪声模式，我们将 使用不同的噪声生成过程替换fillDataArray()函数。

假设我们使用整数除法作为索引，通过“放大”，填充数据数组 到图14.12所示的噪声图的一小部分。对fillDataArray()函数的修改 如下所示。根据用于除法索引的“缩放”因子，可以使得到的3D纹理 更多或少地呈现“块状”。在图14.13中，纹理显示了放大的结果，将 索引分别除以缩放因子8、16和32（从左到右）。

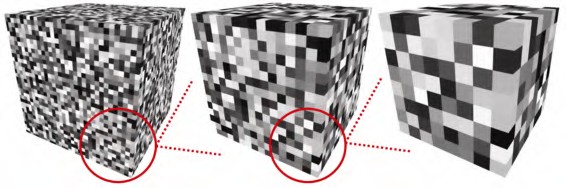


图14.13 不同“缩放”因子的“块状”3D噪声图

void fillDataArray(GLubyte data[ ]) { int zoom = 8; // 缩放因子

for (int i=0; i<noiseWidth; i++) {

for (int j=0; j<noiseHeight; j++) {

for (int k=0; k<noiseDepth; k++) { data[i\*(noiseWidth\*noiseHeight\*4)+j\*

(noiseHeight\*4)+k\*4+0] =

(GLubyte) (noise [i/zoom] [j/zoom]

[k/zoom] \* 255);

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+1] =

(GLubyte) (noise [i/zoom] [j/zoom]

[k/zoom] \* 255);

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+2] =

(GLubyte) (noise [i/zoom] [j/zoom]

[k/zoom] \* 255);

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+3] = (GLubyte) 255;

} } } }

通过从每个离散灰度颜色值插值到下一个灰度颜色值，我们可以 平滑特定的噪声图内的“块效应”。也就是说，对于给定3D纹理内的 每个小“块”，我们通过从其颜色到其相邻块的颜色进行插值来设置 块内的每个纹素的颜色。插值代码在下面所示的函数smoothNoise() 中，还有修改后的fillDataArray()函数。图14.14所示的是得到的 “平滑”纹理（分别是缩放因子2、4、8、16、32和64——从左到右，

从上到下）。请注意，缩放因子现在是一个double类型量，因为我们

需要小数分量来确定每个纹素的插值灰度值。

void fillDataArray(GLubyte data[ ]) { double zoom = 32.0;

for (int i=0; i<noiseWidth; i++) {

for (int j=0; j<noiseHeight; j++) {

for (int k=0; k<noiseDepth; k++) { data[i\*(noiseWidth\*noiseHeight\*4) + j\*

(noiseHeight\*4) + k\*4 +0] =

(GLubyte) (smoothNoise(i/zoom, j/zoom,

k/zoom) \* 255);

data[i\*(noiseWidth\*noiseHeight\*4) + j\*

(noiseHeight\*4) + k\*4 +1] =

(GLubyte) (smoothNoise(i/zoom, j/zoom,

k/zoom) \* 255);

data[i\*(noiseWidth\*noiseHeight\*4) + j\*

(noiseHeight\*4) + k\*4 +2] =

(GLubyte) (smoothNoise(i/zoom, j/zoom,

k/zoom) \* 255);

data[i\*(noiseWidth\*noiseHeight\*4) + j\*

(noiseHeight\*4) + k\*4 +3] = (GLubyte) 255;

} } } }

double smoothNoise(double x1, double y1, double z1) {

// x1、y1和z1的小数部分（对于当前纹素，从当前块到下一个块的百分比）

double fractX = x1 - (int) x1; double fractY = y1 - (int) y1; double fractZ = z1 - (int) z1;

// 在X、Y和Z方向上的相邻像素的索引

int x2 = ((int)x1 + noiseWidth + 1) % noiseWidth; int y2 = ((int)y1 + noiseHeight + 1) % noiseHeight; int z2 = ((int)z1 + noiseDepth + 1) % noiseDepth;

// 通过按照3个轴方向插值灰度，平滑噪声

double value = 0.0;

value += (1-fractX) \* (1-fractY) \* (1-fractZ) \* noise[(int)x1][(int)y1][(int)z1];

value += (1-fractX) \* fractY \* (1-fractZ) \* noise[(int)x1][(int)y2][(int)z1];

value += fractX \* (1-fractY) \* (1-fractZ) \* noise[(int)x2][(int)y1][(int)z1];

value += fractX \* fractY \* (1-fractZ) \* noise[(int)x2][(int)y2][(int)z1];

value += (1-fractX) \* (1-fractY) \* fractZ \*

noise[(int)x1][(int)y1][(int)z2];

value += (1-fractX) \* fractY

\* fractZ

\*

noise[(int)x1][(int)y2][(int)z2];

value += fractX \* (1-fractY) \* fractZ \* noise[(int)x2][(int)y1][(int)z2];

value += fractX \* fractY \* fractZ \* noise[(int)x2][(int)y2][(int)z2];

return value;

}

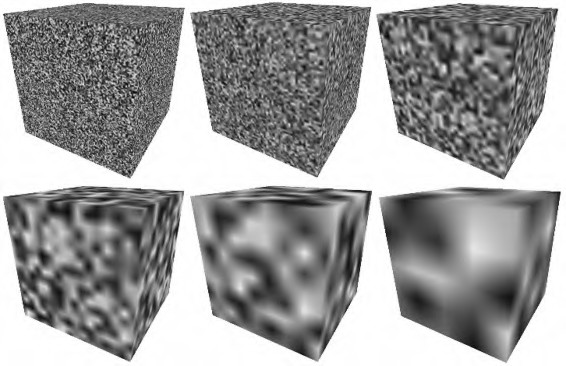


图14.14 在各种缩放级别平滑3D纹理

smoothNoise()函数通过计算相应原始“块状”噪声图中纹素周围 的8个灰度值的加权平均值来计算给定噪声图的平滑版本中的每个纹素 的灰度值。也就是说，它平均纹素所在的小“块”的8个顶点处的颜色 值。这些“邻居”颜色中的每一个的权重基于纹素与其每个邻居的距 离，并归一化到范围[0…1]。

接下来，组合各种缩放因子的平滑噪声图。创建一个新的噪声 图，其中每个纹素由另一个加权平均值形成，这次基于每个“平滑” 噪声图中相同位置的纹素的总和，其中缩放因子用作权重。这种效应 被Perlin [PE85]称为“湍流”，尽管它与通过求和各种波形产生的谐 波实际上更为密切相关。新的turbulence()函数和fillDataArray()的

修改版本指定了一个噪声图，该图对缩放级别1～32（2的各次幂）进

行求和，如下所示。其中还显示了以此产生的噪声图在立方体上贴图 的结果。

double turbulence(double x, double y, double z, double maxZoom) { double sum = 0.0, zoom = maxZoom;

while (zoom >= 1.0) { // 最后一遍是当zoom = 1时

// 计算平滑后的噪声图的加权和

sum = sum + smoothNoise(x / zoom, y / zoom, z / zoom) \*

zoom;

}

zoom = zoom / 2.0; // 对每个2的幂的缩放因子

sum = 128.0 \* sum / maxZoom; // 对不大于64的maxZoom值， 保证RGB < 256

return sum;

}

void fillDataArray(GLubyte data[ ] ) { double maxZoom = 32.0;

for (int i=0; i<noiseWidth; i++) {

for (int j=0; j<noiseHeight; j++) {

for (int k=0; k<noiseDepth; k++) { data[i\*(noiseWidth\*noiseHeight\*4)+j\*

(noiseHeight\*4)+k\*4+0] =

(GLubyte) turbulence(i, j, k, maxZoom); data[i\*(noiseWidth\*noiseHeight\*4)+j\*

(noiseHeight\*4)+k\*4+1] =

(GLubyte) turbulence(i, j, k, maxZoom); data[i\*(noiseWidth\*noiseHeight\*4)+j\*

(noiseHeight\*4)+k\*4+2] =

(GLubyte) turbulence(i, j, k, maxZoom); data[i\*(noiseWidth\*noiseHeight\*4)+j\*

(noiseHeight\*4)+k\*4+3] =

(GLubyte) 255;

} } } }

3D噪声图（如图14.15所示）可用于各种富有想象力的应用。在接 下来的部分中，我们将使用它们来生成大理石、木材和云。可以通过 放大级别的不同组合来调整噪声的分布。

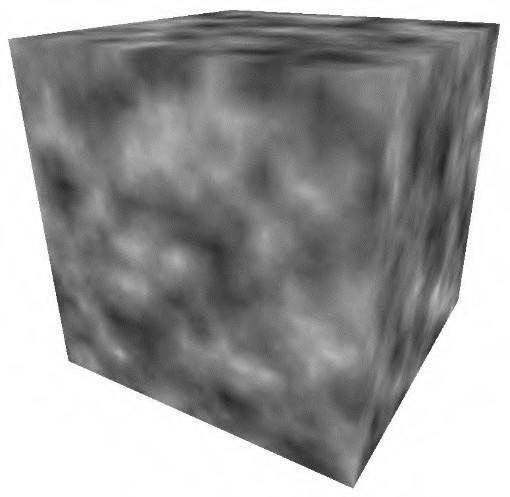


图14.15 “湍流”噪声的3D纹理贴图

14.6 噪声应用——大理石

通过修改噪声图并使用适当的ADS材料添加Phong照明，我们可以 使龙模型看起来像一块大理石般的石头，如图7.3所示。

我们首先生成一个条纹图案，有点类似于本章前面的“条纹”示 例——新条纹与之前的条纹不同，首先是因为它们是对角线，还因为 它们是由正弦波产生的，因此边缘是模糊的。然后，我们使用噪声图 来扰动这些线，将它们存储为灰度值。fillDataArray()函数的更改如 下：

void fillDataArray(GLubyte data[ ])

double veinFrequency = 2.0; double turbPower = 1.5; double maxZoom = 64.0;

for (int i=0; i<noiseWidth; i++) for (int j=0; j<noiseHeight;

{

{

j++) {

for (int k=0; k<noiseDepth; k++) {

double xyzValue = (float)i / noiseWidth + (float)j

/ noiseHeight + (float)k /

noiseDepth + turbPower \*

turbulence(i,j,k,maxZoom) / 256.0;

double sineValue = abs(sin(xyzValue \* 3.14159 \* veinFrequency));

data[i\*(noiseWidth\*noiseHeight\*4)+j\*

(noiseHeight\*4)+k\*4+0] = (GLubyte) redPortion;

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+1] = (GLubyte) greenPortion;

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+2] = (GLubyte) bluePortion;

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+3] = (GLubyte) 255;

} } } }

float redPortion = 255.0f \* (float)sineValue; float greenPortion = 255.0f \* (float)sineValue; float bluePortion = 255.0f \* (float)sineValue;

变量veinFrequency用于调整条纹数量，turbSize调整生成湍流时 使用的缩放系数，turbPower调整条纹中的扰动量（将其设置为0，使 条纹不受干扰）。由于相同的正弦波值用于所有3个（RGB）颜色分 量，所以存储在图像数据阵列中的最终颜色是灰度级的。图14.16显示 了各种turbPower值（0.0、5.5、1.0和1.5，从左到右）的结果纹理贴 图。

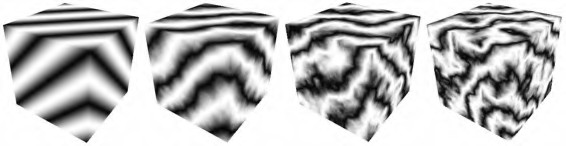


图14.16 构建3D“大理石”噪声图

由于我们希望大理石具有闪亮的外观，我们采用Phong着色使得 “大理石”纹理物体看起来令人信服。程序14.5总结了生成大理石龙

的代码。除了我们还传递了原始顶点坐标以用作3D纹理坐标（如前所

述），顶点和片段着色器与用于Phong着色的相同。片段着色器使用前 面7.6节中描述的技术将噪声结果与光照结果结合。

程序14.5 构建大理石龙

C++ / OpenGL应用程序：

. . .

// 用于Phong着色的白光ADS设置

float globalAmbient[4] = {0.5f, 0.5f, 0.5f, 1.0f}; float lightAmbient[4] = {0.0f, 0.0f, 0.0f, 1.0f}; float lightDiffuse[4] = {1.0f, 1.0f, 1.0f, 1.0f}; float lightSpecular[4] = {1.0f, 1.0f, 1.0f, 1.0f}; float matShi = 75.0f;

void init(GLFWwindow\* window) {

. . .

generateNoise();

noiseTexture = load3DTexture(); // 和程序14.4一样，负责调 用fillDataArray()

}

void fillDataArray(GLubyte data[ ]) { double veinFrequency = 1.75; double turbPower = 3.0;

double turbSize = 32.0;

// 剩下部分构建大理石噪声图的和之前的一样

. . .

}

顶点着色器

// 和程序14.4一样

片段着色器

. . .

void main(void)

{ . . .

// 模型顶点取值[-1.5, +1.5]，纹理坐标取值[0, 1]

vec4 texColor = texture(s, originalPosition / 3.0 + 0.5);

fragColor =

0.7 \* texColor \* (globalAmbient + light.ambient + light.diffuse \* max(cosTheta,0.0))

+ 0.5 \* light.specular \* pow(max(cosPhi, 0.0),

material.shininess);

}

有多种方法可以模拟不同颜色的大理石（或其他石材）。改变大 理石中“矿脉”颜色的一种方法是修改fillDataArray()函数中Color 变量的定义，例如，通过增加绿色成分：

float redPortion = 255.0f \* (float)sineValue;

float greenPortion = 255.0f \* (float)min(sineValue\*1.5 - 0.25, 1.0);

float bluePortion = 255.0f \* (float)sineValue;

我们还可以引入ADS材料值［即在init()中指定］来模拟完全不同 类型的石头，例如“玉石”。

图14.17（见彩插）显示了4个示例，前3个使用程序14.5所示的设 置，第四个示例包含前面图7.3所示的“jade”ADS材料值。



图14.17 3D噪声图纹理的龙——3个大理石和1个玉质

创建“木材”纹理可以采用与之前“大理石”示例中类似的方 式。树木按照年轮生长，正是这些年轮成了我们在用木头制成的物体 中看到的“木纹”。随着树木的生长，环境压力会在年轮中产生变 化，我们也会在木纹中看到这种变化。

我们首先构建一个程序性的“年轮”3D纹理贴图，类似于本章前 面的“棋盘格”。然后，我们使用噪声图来扰动这些年轮，将深色和 浅棕色插入年轮纹理贴图中。通过调整年轮的数量以及扰动年轮的程 度，我们可以用各种类型的木纹模拟木材。棕色的色调可以通过组合 相似数量的红色和绿色、少量蓝色来制作。然后，我们应用具有低 “光泽”的Phong着色。

我们可以通过修改fillDataArray()函数来生成环绕我们3D纹理贴

图中 轴的年轮，使用三角函数指定与

Z Z

轴等距的 和

X Y

值。我们使用正

弦波循环重复此过程，根据此正弦波均匀地升高和降低红色和绿色成 分，以产生不同的棕色调。变量sineValue保持精确的色调，可以通过 稍微偏移一个分量或另一个分量来调整（在这种情况下，将红色增加 80，将绿色增加30）。我们可以通过调整xyPeriod的值来创建更多

（或更少）的年轮。得到的纹理如图14.18所示（见彩插）。

void fillDataArray(GLubyte data[ ])

double xyPeriod = 40.0;

for (int i=0; i<noiseWidth; i++) for (int j=0; j<noiseHeight;

{

{

j++) {

for (int k=0; k<noiseDepth; k++) {

double xValue = (i - (double)noiseWidth/2.0) / (double)noiseWidth;

double yValue = (j - (double)noiseHeight/2.0) / (double)noiseHeight;

double distanceFromZ = sqrt(xValue \* xValue + yValue \* yValue)

double sineValue = 128.0 \* abs(sin(2.0 \* xyPeriod \* distanceFromZ \* 3.14159));

float redPortion = (float)(80 + (int)sineValue); float greenPortion = (float)(30 + (int)sineValue); float bluePortion = 0.0f;

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+0] = (GLubyte) redPortion;

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+1] = (GLubyte) greenPortion;

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+2] = (GLubyte) bluePortion;

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+3] = (GLubyte) 255;

} } } }

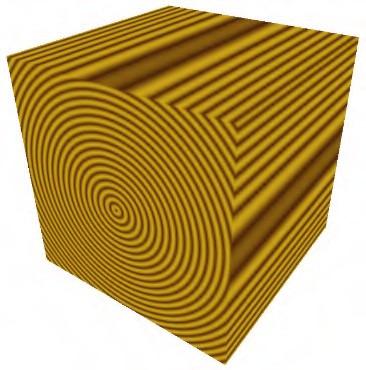


图14.18 为3D木材纹理创建年轮

图14.18中的木质年轮环是一个很好的开始，但它们看起来不太逼 真——它们太完美了。为了改善这一点，我们使用噪声图（更具体地 说，是湍流）来扰动distanceFromZ变量，使得环具有轻微的变化。计 算修改如下：

double distanceFromZ = sqrt(xValue \* xValue + yValue \* yValue)

+ turbPower \* turbulence(i, j,

k, maxZoom) / 256.0;

同样，变量turbPower调整应用了多少湍流（将其设置为0.0，产

生图14.18所示的未受干扰的版本），并且maxZoom指定缩放值（在此 示例中为32）。图14.19显示了turbPower值0.05、1.0和2.0（从左到 右）得到的木材纹理。

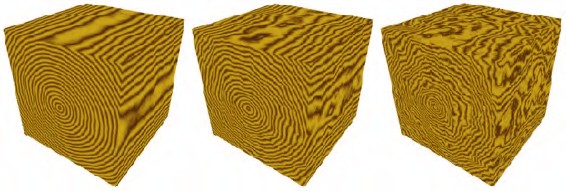


图14.19 “木材”3D纹理贴图与噪声地图扰动的年轮

我们现在可以将3D木材纹理贴图应用于模型。通过对用于纹理坐 标的originalPosition顶点位置应用旋转，可以进一步增强纹理的真 实感，这是因为用木头雕刻的大多数物品与年轮的方向不完全对齐。 为此，我们向着色器发送一个额外的旋转矩阵，以旋转纹理坐标。我 们还添加了Phong着色，具有适当的木色ADS值和适度的光泽度。创建 “木质海豚”的完整代码补充和更改见程序14.6。

程序14.6 构建木质海豚

C++ / OpenGL应用程序： glm::mat4 texRot;

// 木质材质（棕色）

float float float

matAmbient[4] = {0.5f, 0.35f, 0.15f, 1.0f};

matDiffuse[4] = {0.5f, 0.35f, 0.15f, 1.0f};

matSpecular[4] = {0.5f, 0.35f, 0.15f, 1.0f};

float matShi = 15.0f;

void init(GLFWwindow\* window) {

. . .

// 旋转应用于纹理坐标——增加额外的木纹变化

texRot = glm::rotate(glm::mat4(1.0f), toRadians(20.0f), glm::vec3(0.0f, 1.0f, 0.0f));

}

void fillDataArray(GLubyte data[ ]) { double xyPeriod = 40.0;

double turbPower = 0.1; double maxZoom = 32.0;

for (int i=0; i<noiseWidth; i++) {

for (int j=0; j<noiseHeight; j++) {

for (int k=0; k<noiseDepth; k++) {

double xValue = (i - (double)noiseWidth/2.0) / (double)noiseWidth;

double yValue = (j - (double)noiseHeight/2.0) / (double)noiseHeight;

double distanceFromZ = sqrt(xValue \* xValue + yValue \* yValue)

+ turbPower \* turbulence(i,

j, k, maxZoom) / 256.0;

double sineValue = 128.0 \* abs(sin(2.0 \* xyPeriod \* distanceFromZ \* Math.PI));

float redPortion = (float)(80 + (int)sineValue); float greenPortion = (float)(30 + (int)sineValue); float bluePortion = 0.0f;

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+0] = (GLubyte) redPortion;

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+1] = (GLubyte) greenPortion;

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+2] = (GLubyte) bluePortion;

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+3] = (GLubyte) 255;

} } } }

void display(GLFWwindow\* window, double currentTime) {

. . .

tLoc = glGetUniformLocation(renderingProgram, "texRot"); glUniformMatrix4fv(tLoc, 1, false, glm::value\_ptr(texRot));

. . .

}

顶点着色器

. . .

uniform mat4 texRot;

void main(void)

{ . . .

originalPosition = vec3(texRot \* vec4(position,1.0)).xyz;

. . . .

}

片段着色器

. . .

void main(void)

{ . . .

uniform mat4 texRot;

. . .

// 将光照和 3D 纹理结合

fragColor =

0.5 \* ( . . . )

+

0.5 \* texture(s,originalPosition / 2.0 + 0.5);

}

3D材质的木质海豚如图14.20所示。

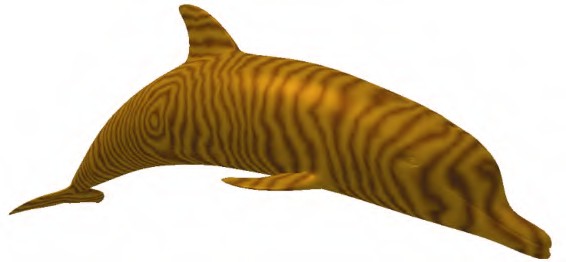


图14.20 “木材”3D噪声图纹理的海豚

片段着色器中还有一个值得注意的细节。由于我们在3D纹理内旋 转模型，所以有时可能会导致顶点位置因旋转而移动超出所需的[0… 1]纹理坐标范围。如果发生这种情况，我们可以通过将原始顶点位置 除以更大的数字（例如4.0而不是2.0）来调整这种可能性，然后添加 稍大一些的数字（例如0.6）以使其在纹理空间中居中。

14.8 噪声应用——云

前面图14.15中构建的“湍流”噪声图看起来有点像云。当然，它 不是正确的颜色，所以我们首先将它从灰度变为适当的浅蓝色和白色 混合。一种直接的方法是为蓝色分量指定一个最大值为1.0的颜色，为 红色和绿色分量指定0.0～1.0的变化（但相等的）值，具体取决于噪 声图中的值。新的fillDataArray()函数如下：

void fillDataArray(GLubyte data[ ]) { for (int i=0; i<noiseWidth; i++) {

for (int j=0; j<noiseHeight; j++) {

for (int k=0; k<noiseDepth; k++) { float brightness = 1.0f - (float)

turbulence(i,j,k,32) / 256.0f;

float redPortion = brightness\*255.0f; float greenPortion = brightness\*255.0f; float bluePortion = 1.0f\*255.0f; data[i\*(noiseWidth\*noiseHeight\*4)+j\*

(noiseHeight\*4)+k\*4+0] = (GLubyte) redPortion;

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+1] = (GLubyte) greenPortion;

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+2] = (GLubyte) bluePortion;

data[i\*(noiseWidth\*noiseHeight\*4)+j\* (noiseHeight\*4)+k\*4+3] = (GLubyte) 255;

} } } }

生成的蓝色版本的噪声图现在可用于纹理化天幕。回想一下，天 幕是一个球体或半球体，在禁用深度测试的情况下被纹理化和渲染， 并放置使其围绕相机（类似于天空盒）。

构建天幕的一种方法是使用顶点坐标作为纹理坐标，以与我们对 其他3D纹理相同的方式对其进行纹理化。然而，在这种情况下，事实 证明使用天幕的2D纹理坐标会产生看起来更像云的图案，因为球面扭

曲会略微拉伸纹理贴图。我们可以通过将GLSL的texture()调用中的第

三维设置为常量值来从噪声图中获取2D切片。假设天幕的纹理坐标已 经以标准方式发送到顶点属性中的OpenGL管线，下面的片段着色器使 用噪声图的2D切片对其进行纹理化：

#version 430

in vec2 tc;

out vec4 fragColor; uniform mat4 mv\_matrix; uniform mat4 proj\_matrix;

layout (binding=0) uniform sampler3D s;

void main(void)

{ fragColor = texture(s,vec3(tc.x, tc.y, 0.5));

了tc.z

}

// 常量替代

得到的纹理化天幕如图14.21所示（见彩插）。虽然相机通常被放 置在天幕内，但我们在外面使用相机进行渲染，因此可以看到圆顶本 身的效果。当前的噪声图导致云“看起来模糊不清”。

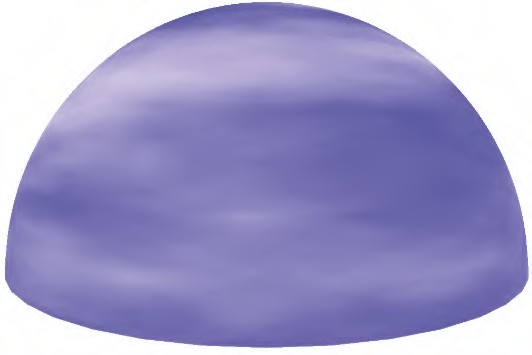


图14.21 云雾缭绕纹理的天幕

虽然我们的朦胧云看起来不错，但我们希望能够塑造它们——也 就是说，让它们更多或更少朦胧。一种方法是修改turbulence()函

数，使其使用指数（如logistic函数），[[2]](#_bookmark231)让云看起来更“明显”。 修改后的turbulence()函数以及相关的logistic()函数如程序14.7所 示。完整的程序14.7还包含前面描述的smooth()、fillDataArray()和 generateNoise()函数。

程序14.7 云纹理生成

C++ / OpenGL应用程序：

double turbulence(double x, double y, double z, double size) { double value = 0.0, initialSize = size, cloudQuant; while(size >= 0.9) {

value = value + smoothNoise(x/size, y/size, z/size) \* size; size = size / 2.0;

}

cloudQuant = 110.0; // 可微调的云质量

value = value / initialSize; value = 256.0 \* logistic(value \* 128.0 - cloudQuant); return value;

}

double logistic(double x) {

double k = 0.2; // 可微调的云朦胧程度，产生更多或更少的分明的云边界

return (1.0 / (1.0 + pow(2.718, -k\*x)));

}

Logistic函数使颜色更倾向于白色或蓝色，而不是介于两者之间 的值，从而产生具有更多不同云边界的视觉效果。变量cloudQuant调 整噪声图中白色（相对于蓝色）的相对量，这反过来导致当应用 logistic函数时产生更多（或更少）的白色区域（即不同的云）。由 此产生的天幕现在具有更明显的云层，如图14.22所示（见彩插）。

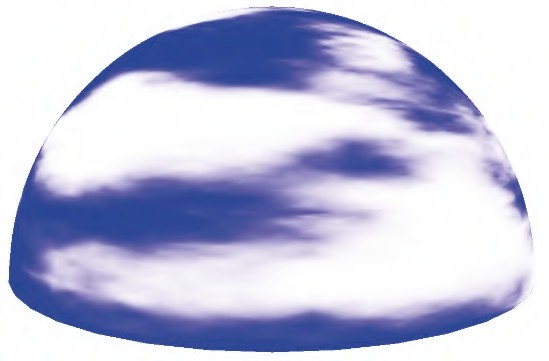


图14.22 指数云纹理的天幕

最后，真正的云不是静态的。为了增强云的真实感，我们应该通 过以下方式使它们变得生动：（a）使它们随着时间的推移而移动或 “漂移”；（b）随着它们漂移逐渐改变它们的形状。

使云“漂移”的一种简单方法是缓慢旋转天幕。这不是一个完美 的解决方案，因为真实的云往往会沿着直线方向漂移，而不是围绕观 察者旋转。但是，如果旋转缓慢且云只是用于装饰场景，则效果可能 是足够的。

随着云的漂移，云逐渐变化，起初可能看起来很棘手。然而，考 虑到我们用于纹理云的3D噪声图，实际上有一种非常简单而聪明的方 法来实现这种效果。回想一下，虽然我们为云构建了一个3D纹理噪声 图，但到目前为止我们只使用了它的一个“切片”，跟天幕的2D纹理

坐标相交（我们将纹理查找的

Z

坐标设置为一个常量值）。到目前为

止，3D纹理的其余部分尚未使用。

我们的技巧是将纹理查找的常量

Z

坐标替换为随时间逐渐变化的变

量。也就是说，当我们旋转天幕时，我们逐渐增加深度变量，导致纹

理查找使用不同的切片。回想一下，当我们构建3D纹理贴图时，我们

将平滑应用于沿3个轴的颜色变化。因此，纹理贴图中的相邻切片非常

相似，但略有不同。因此，通过逐渐改变texture()调用中的

Z

值，云

的外观将逐渐改变。 代码更改导致云缓慢移动并随时间变化，如程序14.8所示。

程序14.8 动画云纹理

C++ / OpenGL应用程序：

. . .

|  |  |  |  |
| --- | --- | --- | --- |
| double | rotAmt | | = 0.0; |
| float | depth | = | 0.01f; |

// 用来让云看起来漂移的Y轴旋转量

// 3D噪声图的深度查找，用来使云逐渐变化

void display(GLFWwindow\* window, double currentTime) {

. . .

// 逐渐旋转天幕

mMat = glm::translate(glm::mat4(1.0f), glm::vec3(domeLocX, domeLocY, domeLocZ);

rotAmt += 0.02; mMat = glm::rotate(mMat, rotAmt, glm::vec3(0.0f, 1.0f, 0.0f));

// 逐渐修改第三个纹理坐标，以使云变化 dLoc = glGetUniformLocation(program, "d"); depth += 0.00005f;

if (depth >= 0.99f) depth = 0.01f; // 当我们到达纹理贴图 的终点时返回开头

glUniform1f(dLoc, depth);

. . .

}

片段着色器

#version 430

in vec2 tc;

out vec4 fragColor;

uniform mat4 mv\_matrix; uniform mat4 proj\_matrix; uniform float d;

layout (binding=0) uniform sampler3D s;

void main(void)

{ fragColor = texture(s, vec3(tc.x, tc.y, d));

改变的"d"替换前面的常量

}

// 逐渐

虽然我们无法在单个静止图像中显示逐渐改变漂移和动画的云的 效果，但图14.23显示了3D生成云的一系列快照中的这些变化，因为它 们从右到左漂移在天幕上，并在漂移时缓慢改变形状。

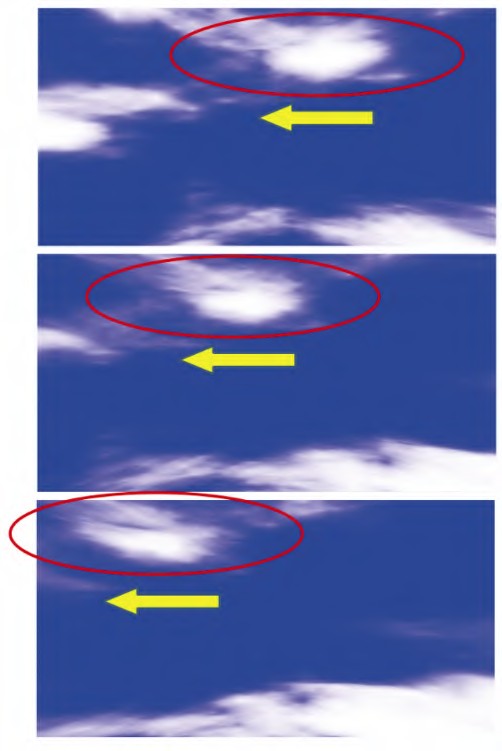


图14.23 3D云在漂移时改变

14.9 噪声应用——特殊效果

噪声纹理可用于各种特殊效果。事实上，有许多可能的用途，其 适用性仅受到想象力的限制。

我们将在此展示的一个非常简单的特殊效应是溶解效应。我们使

物体看起来逐渐溶解成小颗粒，直到它最终消失。给定3D噪声纹理， 可以使用非常少的附加代码实现此效果。

为了促进溶解效果，我们引入了GLSL的discard命令。此命令仅在 片段着色器中是合法的，并且在执行时，它会导致片段着色器丢弃当 前片段（意味着不渲染它）。

我们的策略很简单。在C++/OpenGL应用程序中，我们创建了一个 与图14.12所示相同的细粒度噪声纹理贴图，以及随时间逐渐增加的浮 点变量计数器。然后，此变量在着色器管线中以统一变量发送，并且 噪声图也放置在具有关联采样器的纹理贴图中。然后片段着色器使用 采样器访问噪声纹理——在这种情况下，我们使用返回的噪声值来确 定是否丢弃该片段。我们通过将灰度噪声值与计数器进行比较来实现 这一点，计数器用作一种“阈值”值。因为阈值随着时间的推移逐渐 变化，我们可以将其设置为逐渐丢弃越来越多的片段。结果是物体似 乎逐渐溶解。程序14.9显示了相关的代码部分，它们被添加到程序6.1 中的地球渲染球体中。生成的输出如图14.24所示。

程序14.9 使用discard命令的溶解效果

C++ / OpenGL应用程序：

float threshold = 0.0f;

. . .

// 用于保留、丢弃片段的逐渐增长的阈值

在 display() 中

. . .

tLoc = glGetUniformLocation(renderingProgram, "t"); threshold += .002f;

glUniform1f(tLoc, threshold);

. . .

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_3D, noiseTexture);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, earthTexture);

. . .

glDrawArrays(GL\_TRIANGLES, 0, numSphereVertices);

片段着色器

#version 430

in vec2 tc; // 当前片段的纹理坐标

in vec3 origPos; // 模型中的原始顶点位置，用于访问3D纹理

. . .

layout (binding=0) uniform sampler3D n; // 用于噪声纹理的采样 器

layout (binding=1) uniform sampler2D e; // 用于地球纹理的采样 器

. . .

uniform float t; // 用于保留或丢弃片段的阈值

void main(void)

{ float noise = texture(n, origPos).x; // 从片段中取得噪声值

if (noise > t) // 如果噪声值大于当前

阈值

{ fragColor = texture(e, tc); // 则使用地球纹理渲染 片段

}

else

{ discard; // 否则，丢弃片段（不

要渲染）

}

}

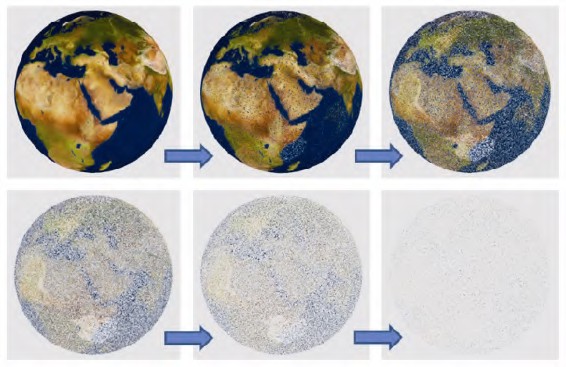


图14.24 使用discard着色器的溶解效果

如果可能，丢弃命令应该谨慎使用，因为它可能会导致性能损 失。这是因为它的存在使OpenGL更难以优化Z缓冲深度测试。

补充说明

在本章中，我们使用Perlin噪声生成云、模拟木材和大理石般的 石头，并且用它们渲染龙。人们发现了Perlin噪音的许多其他用途。 例如，它可用于创建火焰和烟雾[CC16，AF14]，构建逼真的凹凸贴图

[GR05]，并已在电子游戏Minecraft [PE11]中用于生成地形。

本章生成的噪声图基于Lode Vandevenne [VA04]描述的程序。我们 的3D云生成仍存在一些不足之处。纹理不是无缝的，所以在360°点有 一条明显的垂直线（这也是我们在程序14.8中以0.01而不是0.0开始深

度变量的原因，以避免在噪声图的

Z

维中遇到接缝）。如果需要，也有

用于去除接缝[AS04]的简单方法。另一个问题是在天幕的北峰处，天幕 中的球形畸变会产生枕形效应。

我们在本章中实现的云也无法模拟真实云的一些重要方面，例如 它们散射太阳光的方式。真正的云也往往在顶部更白，在底部更灰 暗。我们的云也没有达到许多实际云所具有的3D“蓬松”外观。

类似地，存在用于产生雾的更全面的模型，例如Kilgard和 Fernando [KF03]描述的模型。

在阅读OpenGL文档时，读者可能会注意到GLSL包含一些名为 noise1()、noise2()、noise3()和noise4()的噪声函数，它们被描述 为采用输入种子并产生类似高斯的随机输出。我们在本章中没有使用 这些函数，因为在撰写本文时，大多数供应商都没有实现它们。例 如，无论输入种子如何，许多NVIDIA显卡目前只会为这些函数返回0 值。

# 习题

14.1 修改程序14.2以逐渐增加对象的Alpha值，使其逐渐淡出并 最终消失。

14.2 修改程序14.3以沿水平方向剪裁环面，形成圆形“槽”。

14.3 修改程序14.4（包含图14.10中修改的版本，产生3D立方纹 理），将它改为纹理化Studio 522海豚，然后观察结果。许多人在第

一次观察结果时——例如龙上显示的结果，甚至更简单的物体——都

认为程序中存在一些错误。即使在简单的情况下，也可以通过从3D纹 理“雕刻”对象来产生意外的表面图案。

14.4 用于定义木质“年轮环”的简单正弦波（如图14.18所示） 产生环，其中亮区和暗区的宽度相等。尝试修改相关的 fillDataArray()函数，目的是使暗环的宽度比光环窄。然后观察对所 得木质纹理物体的影响。

14.5 （项目）将logistic函数（来自程序14.7）整合进程序

14.5中的大理石龙，并试验设置以创建更多不同的矿脉。

14.6 修改程序14.9以包含前面章节中描述的缩放、平滑、湍流 和逻辑步骤。观察所产生的溶解效果的变化。

# 参考资料

[AF14] S. Abraham and D. Fussell, “Smoke Brush,” Proceedings of the Workshop on Non-Photorealistic Animation and Rendering (NPAR’14), 2014, accessed October 2018.

[AS04] D. Astle, “Simple Clouds Part 1,” gamedev.net, 2004, accessed October 2018.

[CC16] A “Fire Shader in GLSL for your WebGL Games (2016),” Clockwork Chilli (blog), accessed October 2018.

[GR05] S. Green, “Implementing Improved Perlin Noise,”

, NVIDIA, 2005, accessed October 2018.

GPU Gems 2

[JA12] A. Jacobson, “Cheap Tricks for OpenGL Transparency,” 2012, accessed October 2018.

[KF03] M. Kilgard and R. Fernando, “Advanced Topics,”

(Addison-Wesley, 2003), accessed October

The Cg Tutorial

2018.

[LU16] F. Luna,

Introduction to 3D Game Programming with

, 2nd ed. (Mercury Learning, 2016).

DirectX 12

[PE11] M. Persson, “Terrain Generation, Part 1,” The Word of Notch (blog), Mar 9, 2011, accessed October 2018.

[PE85] K. Perlin, “An Image Synthesizer,” SIGGRAPH ‘85 Proceedings of the 12th annual conference on computer

graphics and interactive techniques (1985).

[VA04] L. Vandevenne, “Texture Generation Using Random Noise,” Lode’s Computer Graphics Tutorial, 2004, accessed October 2018.

[[1]](#_bookmark221) 由电影艺术与科学学院颁发的奥斯卡技术成就奖。

[[2]](#_bookmark225) “logistic”（或“sigmoid”）函数具有S形曲线，两端都有渐

近线。常见的例子是双曲正切函数和

f

( ) = 1/(1+

x e

− )。它们有时也x

被称为“挤压”函数