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：

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。

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]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：

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：

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：

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：

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：

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

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：

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：

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：

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：

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：

片段着色器中的相应属性声明：

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：

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：

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）。